

UNIT-I

Introduction Algorithm Specification, Performance analysis, Performance Measurement. Arrays: Arrays, Dynamically Allocated Arrays. Structures and Unions. Sorting: Motivation, Quick sort, how fast can we sort, Merge sort, Heap sort

What is data structure?

A **data structure** is a **data** organization, management and storage format that enable efficient access and modification. a **data structure** is a collection of **data** values, the relationships among them, and the functions or operations that can be applied to the **data**.

Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

Data: Data can be defined as an elementary value or the collection of value.

For example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item.

For example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items.

For example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity,

For example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

Need of Data Structures

As applications are getting complex and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process. In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Advantages of Data Structures

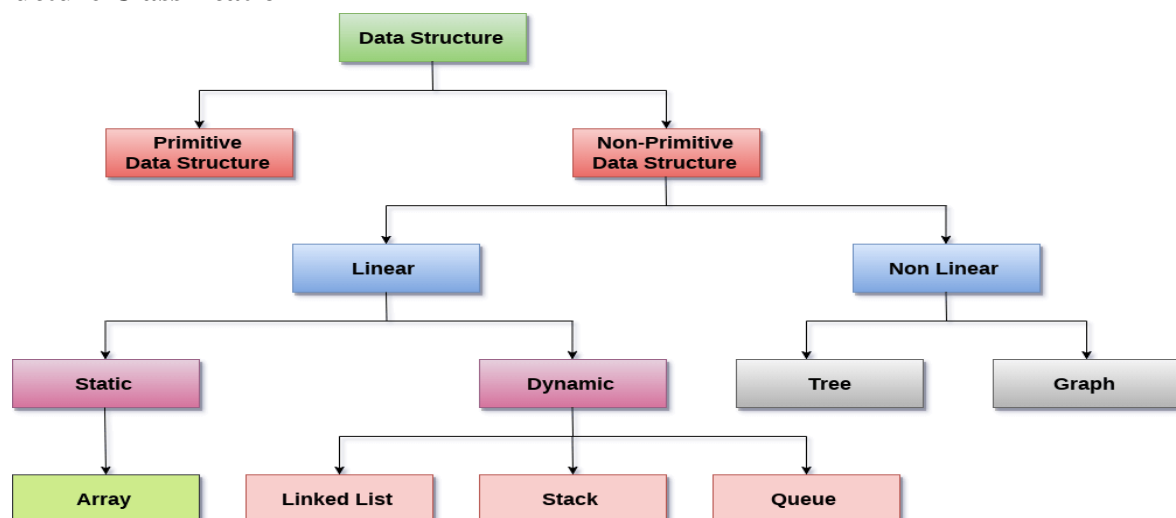
Efficiency: Efficiency of a program depends upon the choice of data structures.

For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. Hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called top.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack.

For example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures: This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called leaf node while the topmost node is called root node. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Operations on data structure

1) Traversing: Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2) Insertion: Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is n then we can only insert $n-1$ data elements into it.

3) Deletion: The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then underflow occurs.

4) Searching: The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search.

5) Sorting: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) Merging: When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size $(M+N)$, then this process is called merging

Characteristics of a Data Structure

- **Correctness** – Data structure implementation should implement its interface correctly.
- **Time Complexity** – Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

Algorithm

An algorithm is a procedure having well defined steps for solving a particular problem. Algorithm is finite set of logic or instructions, written in order for accomplish the certain predefined task. It is not the complete program or code, it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

The major categories of algorithms are given below:

- **Sort:** Algorithm developed for sorting the items in certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.
- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.

The performance of algorithm is measured on the basis of following properties:

Time complexity: It is a way of representing the amount of time needed by a program to run to the completion.

Space complexity: It is the amount of memory space required by an algorithm, during a course of its execution. Space complexity is required in situations when limited memory is available and for the multi user system.

Each algorithm must have:

Specification: Description of the computational procedure.

Pre-conditions: The condition(s) on input.

Body of the Algorithm: A sequence of clear and unambiguous instructions.

Post-conditions: The condition(s) on output.

Example: Design an algorithm to multiply the two numbers x and y and display the result in z.

Step 1: START

Step 2: declare three integers x, y & z

Step 3: define values of x & y

Step 4: multiply values of x & y

Step 5: stores the output of step 4 in z

Step 6: print z

Step 7: STOP

Alternatively the algorithm can be written as?

Step 1 START MULTIPLY

Step 2 get values of x & y

Step 3 $z \leftarrow x * y$

Step 4 display z

Step 5 STOP

Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

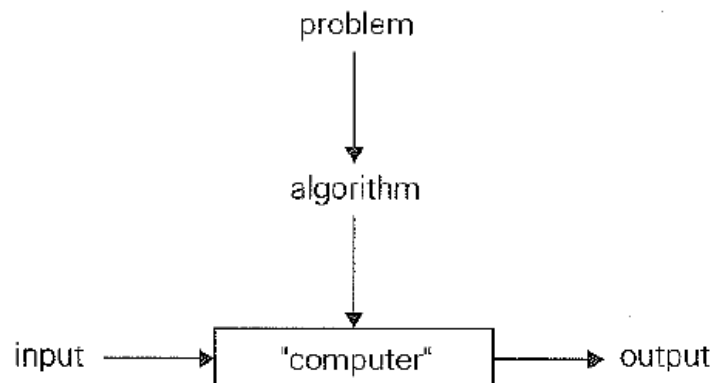
- ✓ **Input:** An algorithm must have 0 or well defined inputs.
- ✓ **Output:** An algorithm must have 1 or well defined outputs, and should match with the desired output.
- ✓ **Feasibility:** An algorithm must be terminated after the finite number of steps.
- ✓ **Independent:** An algorithm must have step-by-step directions which is independent of any programming code.
- ✓ **Unambiguous:** An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

Algorithm:

Algorithm is a finite set of instructions that is followed, accomplishes a particular task.

(OR)

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate (lawful, legal, reasonable genuine) input in a finite amount of time.



There is something or someone capable of understanding and following the instructions given. We call this a "computer".

All algorithms must satisfy the following criteria:

1. **Input:** It may be zero or more quantities are externally supplied. But input not necessary for all Algorithms.
2. **Output:** At least one quantity is produced. It is must for all the algorithms
3. **Definiteness:** Each instruction is clear and unambiguous.
4. **Finiteness:** The instruction (Steps) of an algorithm must be finite. Means Algorithm terminate after finite number of steps (Instructions).
5. **Effectiveness:** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite, it also must be feasible (possible or realistic).

The study of algorithms includes many important and active areas of research. There are four distinct areas of study one can identify:

1. How to devise algorithms -
2. How to validate algorithms
3. How to Design algorithms
4. How to testing algorithms

Example for algorithm:

Write an algorithm to add two numbers entered by user.

Step 1: Start.
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
 $\text{sum} \leftarrow \text{num1} + \text{num2}$.
Step 5: Display sum.
Step 6: Stop.

Algorithm specification.

Pseudocode Conventions

- ✓ **Algorithm** can be described in many ways like English for specifying algorithm in step by step and **Flowchart** for graphical representation.
- ✓ But these two ways work well only if algorithm is small or simple. For large or big algorithm we are going to use pseudo code.
- ✓ Pseudo code is a kind of structured English for describing algorithms.
- ✓ It allows the designer to focus on the logic of the algorithm without being distracted (diverted) by details of language syntax.
- ✓ At the same time, the pseudo code needs to be complete. It describes the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line into source code.

NOTE: Alternatively, we can express the same algorithm in a **pseudocode**:

Example for pseudocode:

```
ALGORITHM Sum(num1,num2)
Input: read num1,num2
Output: write sum of num1&num2
{
Read num1,num2;
Sum = num1+num2;
Write sum;
}
```

Pseudo code conversion:

1. “// ” Is used to provide comment line.
2. Using of matching parenthesis “{}” to form blocks. A compound statement (i.e., Collection of simple statements) can be represented as a block.
3. An identifier begins with a letter. Data types of variables are not explicitly declared. But types will be clear from the context whether a variable is global or local to a procedure.

For example:

```
node=record
{
    datatype_1 data_1;
    .
    .
    datatype_n data_n;
    node *link;
}
```

4. Assignment of values to variables is done using assignment statement.
<variable> := <expression>;
5. There are
 - Two Boolean values true and false.
 - Logical operator “and, or and not” instead of “&, ||, !” respectively.
 - Relation operator “<, ≤, =, ≠, ≥ and >”.

6. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i, j)th element of the array is denoted as A[i, j]. Array indices (index) start at zero.
7. Looping statement are employed: **while** , **repeat until** and **for** as like below

General representation	Pseudo code representation
While (condition){ Stmt1; Stmt n; }	While <condition> do { <stmt 1>; <stmt n>; }
do{ Stmt1; Stmt n; } While(condition);	repeat { <stmt 1>; <stmt n>; Until <condition> }
for(initialization;condition;expression) { Stmt1; Stmt n; }	for variable :=value1 to n step step do { <stmt 1>; <stmt n>; }

- 8 A conditional statement: **if** and **switch** has the following forms:

General form	Pseudo code form
if (condition) stmt1; else stmt 2	If <condition> then <Stmt 1>; else <stmt 2>;
Switch (condition){ Case label: stmt1; break; . default: stmt; }	case { :<condition 1> : <stmt 1> . :<condition n>: <stmt n> :else: <stmt n+1> }

- 9 Input and output are done using the instructions read and write. No format is used to specify the size of input or output quantities.
- 10 There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and a body. The heading takes the form

Algorithm Name (<parameter list>)
--

Algorithm for finds & returns the maximum of n given numbers	
In general way (program)	In algorithm (pseudo code)
<pre> A[n]={9,3,...}; Result=A[0]; for (i=1;i<=n-1;i++){ if A[i] > Result; Result =A[i]; } printf (Result); </pre>	<pre> Algorithm max(A, n) // A is an array of size n { Result :=A[1]; for i:=2 to n do{ if A[i] > Result; then Result :=A[i]; return Result; } </pre>
Algorithm for selection sorting	
Program	Algorithm
<pre> main(){ int s, i, j, temp, a[20]; for (i=0; i<s; i++){ for(j=i+1; j<s; j++){ temp=a[i]; a[i]=a[j]; a[j]=temp; } } } </pre>	<pre> Algorithm SelectionSort(a,n) // sor the array a of n-elements. { for i:=1 to n do { j:=i; for k:=i+1 to n do if(a[k]<a[j] then j:=k; t:=a[i]; a[i]:=a[j]; a[j]:=t; } } </pre>

Recursive Algorithms:

A function is called itself then it called recursive function.

```

Void main()
{
Fun1();// main() calls Fun1
}
Void Fun1()
{
Fun1();
}

```

For example

```

// a part of c- program
Void main(){
int i=5;
printf(fact(i));
}
int fact(int i)
{
if(i<=1)
{
return 1;
}
return i*fact(i-1); }

```

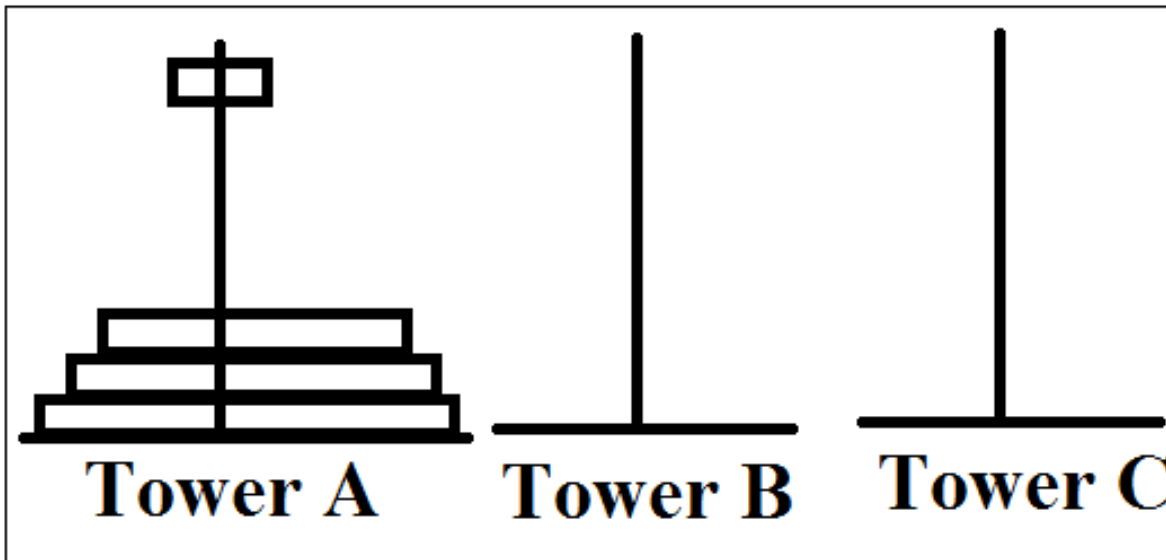
Similarly an algorithm is said to be recursive if the same algorithm is invoked in the body. There are two types of Recursive algorithms.

- Direct recursive algorithm → an algorithm that calls itself is **directive recursive**.
- Indirect recursive algorithm → an algorithm, if it calls another algorithm is indirect recursive. Means algorithm A is said to be **indirect recursive** if it calls another algorithm which in turn calls A.

Example for recursive algorithm:

Towers of Hanoi Problem simple rules:

- ✓ Only one disk can be moved at a time.
- ✓ Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- ✓ No disk may be placed on top of a smaller disk.



A very elegant (Style or design or dress) solution for this tower of Hanoi by using Recursion.

For n disks, total $2^n - 1$ moves are required

Algorithm for solving “Towers of Hanoi” by using recursion algorithm:

```
Algorithm TowersOfHanoi(n, x, y, z)
//Move the top n disks form Tower x to Tower y
{
  if(n≥1) then
    TowerOfHanoi(n-1, x, z,y);
    write(“Move top disk form Tower x to top of tower y);
    TowerOfHanoi(n-1,z,y,x);
  }
}
```

Explanation:

Assume that the number of disks is n . To get the largest disk to the bottom of Tower y , we move the remaining $n-1$ disks to tower z . And then move the largest disk from tower x to tower y .

Now tower y has largest disk and tower x is empty and tower z has $n-1$ in decreasing order.

Here we left the remaining task that is moving $n-1$ disks from tower Z to tower Y . For this we use same procedure by using tower x as intermediate storage.

For example:

Input: if number of disks $n=2$

Output: Total number of moves $=2^n-1$

$2^2-1=3$ moves as shown below

S-1: Disk 1 moved from A to B

S-2: Disk 2 moved from A to C

S-3: Disk 1 moved from B to C

Similarly for number of disks $=3$

Input : 3

Output :

S-1: Disk 1 moved from A to C

S-2: Disk 2 moved from A to B

S-3: Disk 1 moved from C to B

S-4: Disk 3 moved from A to C

S-5: Disk 1 moved from B to A

S-6: Disk 2 moved from B to C

S-7: Disk 1 moved from A to C

Performance Analysis**Introduction:****Analyze Algorithms or performance analysis:**

If an algorithm is executed, it used the

- Computer's CPU → to perform the operations.
- Memory (both RAM and ROM) → to hold the program & data.

Analysis of algorithms is the task of determining how much computing time and storage memory required for an algorithm.

There are many criteria upon which we can judge an algorithm to say it performs well.

- 1) Does it do what we want it to do?
- 2) Does it work correctly according to the original specifications of the task?
- 3) Is there documentation that describes how to use it and how it works?
- 4) Are procedures created in such a way that they perform logical sub functions?
- 5) Is the code readable?

Performance evaluation has two major phases

- Priori estimates → Performance analysis
- Posterior testing → Performance measurements

Space Complexity:

Definition:

The space complexity of an algorithm is the amount of memory it needs to run to completion.

The space needed by algorithm is the sum of following components

- A fixed part → that is independent of the characteristics of input and output. Example Number & size. In this includes instructions space [space for code] + Space for simple variables + Fixed-size component variables + Space for constant & soon.
- A variable part → that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved + The space needed by referenced variables + Recursion stack space.

The space requirement $S(P)$ of any algorithm P can be written as

$$S(P) = C + S_P$$

$C \rightarrow$ Constant

$S_P \rightarrow$ Instance characteristics.

Find space complexity of iterative Algorithm of sum of 'n' numbers.	Find space complexity for Recursive algorithm:
Algorithm: Algorithm sum(a,n) //Here a is array of Size n { S:=0; if(n≤0)) then return 0; for i:=1 to n do s:=s+a[i]; return s; }	Algorithm: Algorithm RSUM(a,n) //a is an array of size n { If(n≤0) then return 0; Else return a[n]+RSUM(a,n-1); }
Space complexity $S(P)$: s → 1 word i → 1 word n → 1 word a → n word <div style="border: 1px solid black; border-radius: 10px; padding: 5px; display: inline-block; margin: 10px 0;">$S(P) \geq n+3$</div> Word is a String of bits stored in computer memory, its size is a 4 to 64 bits.	Space Complexity: The recursion stack space includes space for the formal parameters, the local variables and the return address. Return address requires → 1 word memory Each call to RSUM requires → at least 3 words (It includes space for the values of n, the return address and a pointer to a[]). The depth recursion is n+1 The recursion stack space needed is $\geq 3(n+1)$

Time complexity T(P):

Definition:

Time complexity of an algorithm is the amount of computer time it needs to run to completion.

Here RUN means Compile + Execution.

Time Complexity

$$T(P) = t_c + t_p$$

But we are neglecting t_c Because the compile time does not depends on the instance characteristics. The compiled program will be run several times without recompilation.

So $T(P) = t_p$

Here $t_p \rightarrow$ instance characteristics.

For example:

The program **p** do some operations like ADD, SUB, MUL etc.

If we knew the characteristics of the compiler to be used, we could process to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores and so on.

We obtain $t_p(n)$ express as follow:

$$t_p(n) = C_a \text{ADD}(n) + C_s \text{SUB}(n) + C_m \text{MUL}(n) + C_d \text{DIV}(n) + \dots$$

$n \rightarrow$ Instance characteristics

$C_a, C_s, C_m, C_d, \dots \rightarrow$ time needed for an addition, Subtraction, multiplication, division, and etc.

ADD, SUBB, MUL, DIV \rightarrow Are functions, whose values are the numbers of additions, subtractions, multiplications,.. etc, that are performed when the code for p is used on an instance with characteristics.

Performance Measurement

Program step: Program step is the syntactically or semantically meaningful segment of a program. And it has an execution time that is independent of the instance characteristics.

Example:

- For comment \rightarrow // -- zero steps
- For assignment statements (Which does not involve any calls to other algorithms)
 $:= \rightarrow$ one step
- For iterative statements such as “for, while and until-repeat” statements, we consider the step counts only for control part of the statement.
For while loop “while (<expr>) do “ : the step count for control part of a while stmt is \rightarrow
Number of step counts for assignable to <expr>

For for loop ie for $i := \langle \text{expr} \rangle$ to $\langle \text{expr1} \rangle$ do: The step count of control part of “for” statement is \rightarrow
Sum of the count of $\langle \text{expr} \rangle$ & $\langle \text{expr1} \rangle$ N and remaining execution of the “for” statement, i.e., one.

We determine the number of steps needed by a program to solve a particular problem. For this there are two methods.

- 1) **First method** is, introduce a new variable “**count**” in to the program for finding number of steps in program.
- 2) **Second method** is, building a “**table**” in which we list the total number of steps contributed by each statement.

Example for 1st method:

Find time complexity of Iterative algorithm of sum of ‘n’ numbers.	Find time complexity for Recursive algorithm:
<p>Algorithm: Algorithm sum(a,n) { // count is global it is initially zero S:=0; Count:=count+1; // count for assignment for i:=1 to n do { Count :=count+1; //for “for” loop s:=s+a[i]; count :=count+1; //for assingment } Count :=count+1; //for last time of for loop Count :=count+1; // for return stmt return s; }</p> <p>.....</p> <p>Finally count values is =2n+3; So total number of steps= 2n+3</p>	<p>Algorithm: Algorithm RSUM(a,n) { Count :=count+1; // for if condition If(n≤0) then { Count:=count+1; // for return stmt return 0; } Else { Count :=count+1; //for the addition, function invocation & return return a[n]+RSUM(a,n-1); } }</p> <p>.....</p> <p>If n=0 then $t_{Rsum}(0)=2$ If n>0 then increases by 2, ie., $2+ t_{Rsum}(n-1)$ Means $t_{Rsum}(n)=2+ t_{Rsum}(n-1)$ $=2+2+ t_{Rsum}(n-2)$ $=2+2+2+ t_{Rsum}(n-3)$. . $=2(n)+ t_{Rsum}(n-n)$ $=2n+ t_{Rsum}(0)=\mathbf{2n+2}$</p>

In the above example the recursive algorithm has **less** time complexity then iterative algorithm.

Example for 2nd method:

Find time complexity of Algorithm of sum of 'n' numbers.			
Statements	s/e	Frequency	Total steps
1. Algorithm sum(a,n)	0	—	0
2. {	0	—	0
3. S:=0;	1	1	1
4. for i:=1 to n do	1	n+1	n+1
5. s:=s+a[i];	1	n	n
6. return s;	1	1	1
7. }	0	—	0
			2n+3

Find time complexity for Recursive algorithm				
Statements	s/e	Frequency n=0 n>0		Total steps n=0 n>0
1. Algorithm RSUM(a,n)	0	—	—	— —
2. {	0	—	—	— —
3. If(n≤0) then	1	1	1	1 1
4. return 0;	1	1	0	1 0
5. Else	0			
6. return a[n]+RSUM(a,n-1);	1+x	0	1	0 1+x
7. }	0	—	—	0 0
				2 2+x

Here $x = t_{\text{Rsum}}(n-1)$

Asymptotic Notation:

A problem may have numerous (many) algorithmic solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run.

Asymptotic notation of an algorithm is a mathematical representation of its complexity

Asymptotic notation is used to judge the best algorithm among numerous algorithms for a particular problem.

Asymptotic complexity is a way of expressing the main component of algorithms like

- Cost
- Time complexity
- Space complexity

Some Asymptotic notations are

1. Big oh $\rightarrow O$
2. Omega $\rightarrow \Omega$
3. Theta $\rightarrow \theta$
4. Little oh $\rightarrow o$
5. Little Omega $\rightarrow \omega$

1. Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

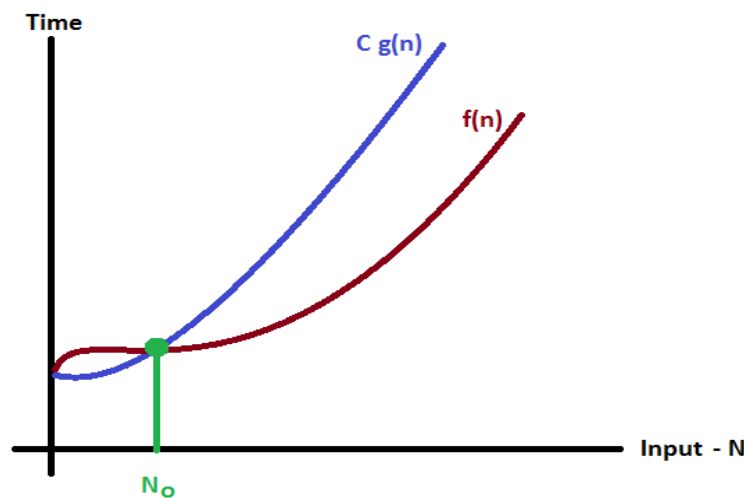
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be **defined** as follows...

The function $f(n) = O(g(n))$ (read as “f of n is big oh of g of n) iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n, n \geq n_0$

$f(n) = O(g(n))$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C g(n)$ is greater than $f(n)$ which indicates the algorithm's upper bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $O(g(n))$ then it must satisfy $f(n) \leq C \times g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \leq C g(n)$$

$$\Rightarrow 3n + 2 \leq C n$$

Above condition is always TRUE for all values of $C = 4$ and $n \geq 2$.

By using Big - Oh notation we can represent the time complexity as follows...

$$3n + 2 = O(n)$$

2. Big - Omega Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

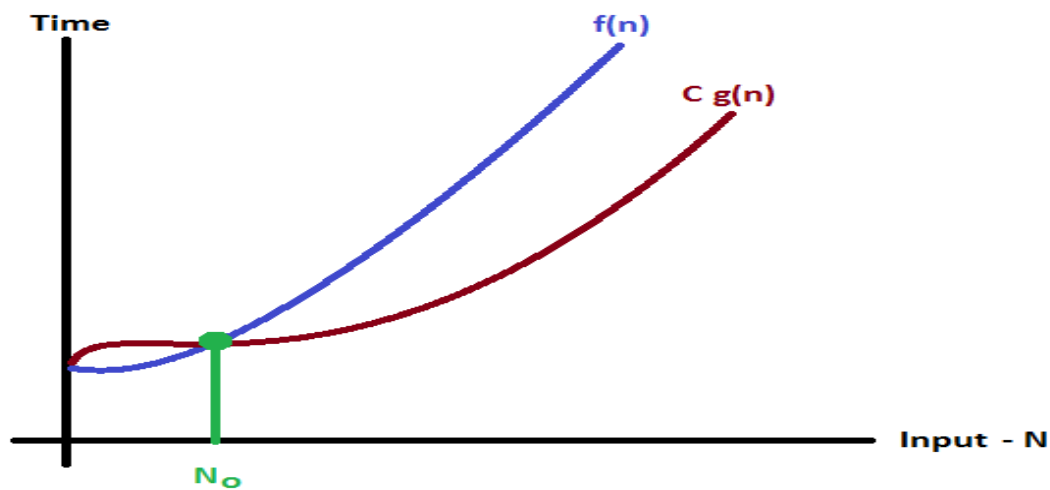
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

The function $f(n) = \Omega(g(n))$ (read as “f of n is omega of g of n) iff (if and only if) there exist positive constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n, n \geq n_0$

$f(n) = \Omega(g(n))$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C \times g(n)$ is less than $f(n)$ which indicates the algorithm's lower bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) \geq C g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \geq C g(n)$$

$$\Rightarrow 3n + 2 \leq C n$$

Above condition is always TRUE for all values of $C = 1$ and $n \geq 1$.

By using Big - Omega notation we can represent the time complexity as follows...

$$3n + 2 = \Omega(n)$$

3. Big - Theta Notation (Θ)

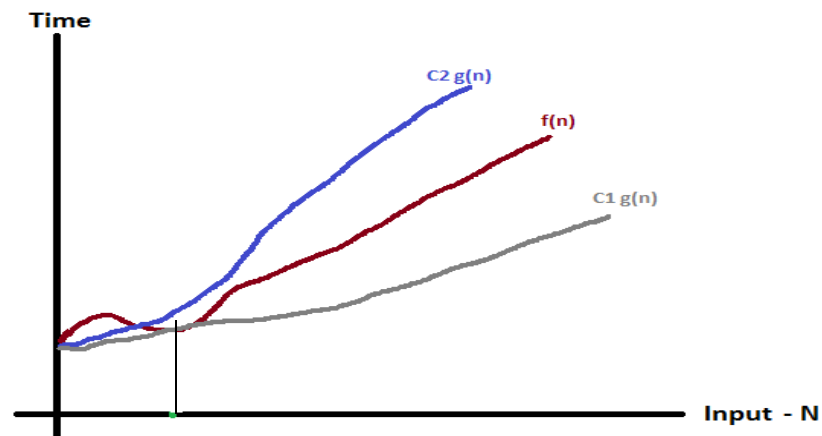
Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

The function $f(n) = \Theta(g(n))$ (read as “f of n is theta of g of n) iff (if and only if) there exist positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n, n \geq n_0$
 $f(n) = \Theta(g(n))$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C_1 g(n)$ is less than $f(n)$ and $C_2 g(n)$ is greater than $f(n)$ which indicates the algorithm's average bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Theta(g(n))$ then it must satisfy $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all values of C_1 , $C_2 > 0$ and $n_0 \geq 1$

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

$$C_1 n \leq 3n + 2 \leq C_2 n$$

Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n \geq 1$.

By using Big - Theta notation we can represent the time complexity as follows...

$$3n + 2 = \Theta(n)$$

1. Little oh: o

The function $f(n)=o(g(n))$ (read as “f of n is little oh of g of n”) iff

$$\lim_{n \rightarrow \infty} f(n)/g(n)=0 \quad \text{for all } n, n \geq 0$$

$n \rightarrow \infty$

Example:

$$3n+2 = o(n^2) \text{ as}$$

$$\lim_{n \rightarrow \infty} ((3n+2)/n^2)=0$$

$n \rightarrow \infty$

2. Little Omega: ω

The function $f(n)=\omega(g(n))$ (read as “f of n is little ohomega of g of n”) iff

$$\lim_{n \rightarrow \infty} g(n)/f(n)=0 \quad \text{for all } n, n \geq 0$$

$n \rightarrow \infty$

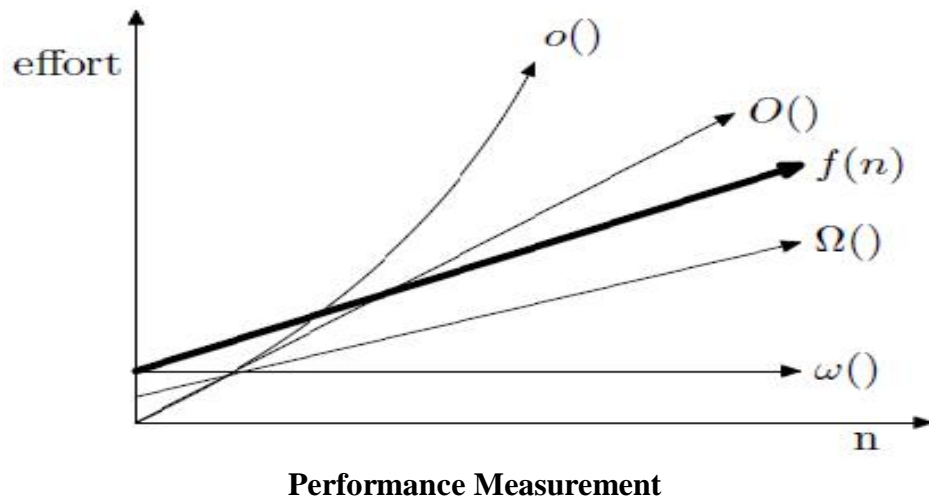
Example:

$$3n+2 = \omega(n^2) \text{ as}$$

$$\lim_{n \rightarrow \infty} (n^2/(3n+2)) = \infty$$

$n \rightarrow \infty$

Graph for visualize the relationships between these notations



Order of Growth

- ✓ Any algorithm expected to do work fast for any input size
- ✓ For example: sorting of elements ($n=5$) when the input size is small the algorithm works fine
- ✓ When we increase the size of input we can analyze how our algorithm works for example :when $n=5000$ elements how our algorithm works
- ✓ Order of growth means how your algorithm performs according to the input size. when the input changes then algorithm performance also changes. The calculation of this performance is an analysis of the algorithm.

For example: **Linear Search**

The simplest and most obvious way to search a table is to use Linear search i.e. examine the 1st, 2nd, 3rd,...entries until the entry with the required key is found or the end of the table is reached without the entry being found. The body of the search function could be written as follows if Linear search is used:

```
int found;
int i;
found = FALSE;
i = 0;
while (!found && i<n)
{
    if (!strcmp(word, table[i].key)) // strcmp returns zero
                                   // if strings are same
    {
        found = TRUE;
        index = i;
    }
    else i++;
}
return found;
```

- ✓ The efficiency of Linear search is now considered. In assessing the efficiency of an algorithm it is usual to count the number of occurrences of some typical operation as a function of the size of the problem.
- ✓ In searching the major operation is the number of comparisons of the searched-for key against the table entries and the problem size is taken to be the number of entries in the table. Hence in Linear search of a table with n entries we have:

Best Case:

Is the minimum number of steps that can be achieved for given parameters.

Find at first place - one comparison

Worst Case

Is the maximum number of steps that can be achieved for given parameters

Find at nth place or not at all - n comparisons

Average Case

Average case is the situation in which it is not either best case or worst case

Element is found somewhere in the middle of the list

Probability of successful search is p

Probability of unsuccessful search is 1-p

Consider the given element is found at position 'i' then the probability of finding that element 'i' is p/n

$$C \text{ Avg}(n) = \underbrace{[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \dots + n \cdot \frac{p}{n}]}_{\text{Successful search}} + \underbrace{n[1-p]}_{\text{unsuccessful search}}$$

$$= \frac{p}{n}[1+2+3+4+\dots+n] + n[1-p]$$

$$= \frac{p}{n}[\frac{n(n+1)}{2}] + n[1-p]$$

$$= \frac{p(n+1)}{2} + n[1-p]$$

$$C \text{ Avg}(n) = \frac{p(n+1)}{2} + n[1-p]$$

For **successful search** p=1 the above equation can be written as

$$C \text{ Avg}(n) = \frac{1(n+1)}{2} + n[1-1]$$

$$= \boxed{\frac{n+1}{2}}$$

For **unsuccessful search** p=0 the above equation can be written as

$$C \text{ Avg}(n) = \frac{0(n+1)}{2} + n[1-0]$$

$$= \boxed{n}$$

Hence Linear Search is an order(n) process - i.e. it takes a time proportional to the number of entries. For example if n was doubled then, on average, the search would take twice as long.

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n ²)	O(n ²)	O(1)
Selection Sort	O(n ²)	O(n ²)	O(n ²)	O(1)
Insertion Sort	O(n)	O(n ²)	O(n ²)	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	O(n ²)	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n ²)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(n log n)	O(n log n)	O(n)
Shell Sort	O(n)	O((n log(n)) ²)	O((n log(n)) ²)	O(1)

- ✓ Performance measurement is the process of executing a correct program on different data sets to measure the time and space that it takes to compute the results. Complexity of a program is generally some function of the instance characteristics
- ✓ The ultimate test is performed to ensure that the program developed on the basis of the designed algorithm, runs satisfactorily. Testing a program involves two phases viz., **debugging** and **profiling**.

- ✓ Debugging is the process of executing a program with sample datasets to determine if the results obtained are satisfactory.
- ✓ However, it is pointed out that “debugging can only indicate the presence of errors but not their absence” i.e., a program that yields unsatisfactory results on a sample data set is definitely faulty, but on the other hand a program producing the desirable results on one/more data sets need not be correct. In order to actually prove that a program is perfect, a process of “proving” is necessary wherein the program is analytically proved to be correct and in such cases, it is bound to yield perfect results for all possible sets of data.
- ✓ On the other hand, **profiling** or performance measurement is the process of executing a correct program on different data sets to measure the time and space that it takes to compute the results. That several different programs may do a given job satisfactorily.
- ✓ This is the final stage of algorithm evaluation. A question to be answered when the program is ready for execution, (after the algorithm has been devised, made a priori analysis of that, coded into a program debugged and compiled) is how do we actually evaluate the time taken by the program? Obviously, the time required to read the input data or give the output should not be taken into account. If somebody is keying in the input data through the keyboard or if data is being read from an input device, the speed of operation is dependent on the speed of the device, but not on the speed of the algorithm. So, we have to exclude that time while evaluating the programs
- ✓ The entire procedure explained above is called “profiling”. However, unfortunately, the times provided by the system clock are not always dependable. Most often, they are only indicative in nature and should not be taken as an accurate measurement. Especially when the time durations involved are of the order of 1-2 milliseconds, the figures tend to vary often between one run and the other, even with the same program and all same input values .

Common Asymptotic Notations

constant	—	$O(1)$
logarithmic	—	$O(\log n)$
linear	—	$O(n)$
$n \log n$	—	$O(n \log n)$
quadratic	—	$O(n^2)$
cubic	—	$O(n^3)$
polynomial	—	$n^{O(1)}$
exponential	—	$2^{O(n)}$

ARRAYS

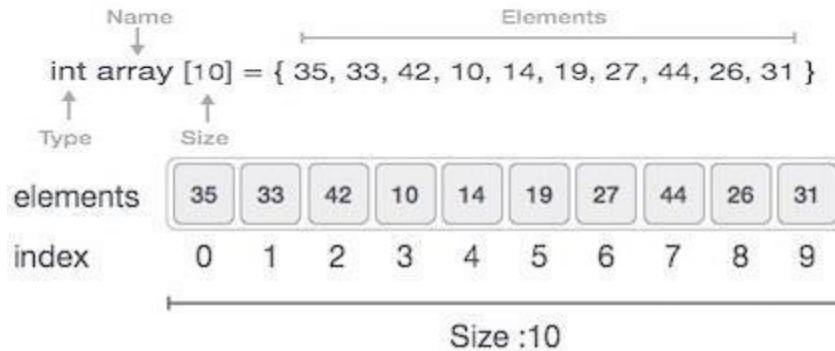
One Dimensional array

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element** – each item stored in an array is called an element.
- **Index** – each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Merging** -- two arrays
- **Sorting an array** in ascending order or descending order

In C, when an array is initialized with size, then it assigns default values to its elements in following order.

Data Type	Default Value
bool	false
char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Traverse Operation

Traversing Linear Structures. Traversing a linear structure means moving through it sequentially, node by node. Processing the data element of a node may be complex, but the general pattern is as follows: ... Repeat until there are no more nodes. Process the current node.

Algorithm for array Traversal

Step 1. Start

Step 2. [INITIALIZATION] Set I lower bound

Step 3. Repeat steps 3to 4 while $I \leq$ upper bound

Step 4. Apply process to A [I]

Step 5. Set $I = I + 1$

[End loop]

Step 6: Exit

Example Program to read and display n numbers using an array

```
#include<stdio.h>
#include<conio.h>
int main( )
{
    inti,n,arr[20];
    clrscr( );
    printf("\n enter the number of elements in the array:");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
        Printf("\n arr[%d]= ", i);
        Scanf("%d",&arr[i]);
    }
    Printf("\n the array elements are");
    For(i=0; i<n; i++)
        Printf("\t %d",arr[i]);
    return 0;
}
```

Output

Enter the number of elements in the array:5

Arr[0] =1

Arr[1]=2

Arr[2]=3

Arr[3]=4

Arr[4]=5

The elements in the array are 1 2 3 4 5

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Algorithm for inserting a element to an existing array

Step 1. Start

Step 2. Set upper_bound = upper_bound+1

Step 3. Set A[upper_bound] = val

Step 4. Stop

Example

Data []={12, 23, 34, 45, 56, 67, 78, 89, 90,100};

- Calculate the length of the array
- Find the lower and upper-bound
- Show the memory representation of the array
- If the new data element with the value 75 has to be inserted find its position
- Insert the 75 and show the memory representation

Solution:

Length of the array =number of the elements

Lower bound= 0 upper bound = 9

Index	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]
Data value	12	23	34	45	56	67	78	89	90	100

After inserting the 75 value

Index	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]	Data[10]
Data value	12	23	34	45	56	67	75	78	89	90	100

Algorithm for insert an element in middle of the array

Step 1. Start

Step 2. [INITIALIZATION] set I = N

Step 3. Repeat Steps 4 and 5 While I >= pos

Step 4. Set A[I+1] = A[I]

Step 5. Set I = I – 1

[End of the loop]

Step 6. Set N = N + 1

Step 7. Set A[pos] = value

Step 8. Stop

Example

Initial data array

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	34	12	56	20

Calling Insert (Data, 3, 100) will lead to the following in the array

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
45	23	34	12	56	20	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
45	23	34	12	56	56	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
45	23	34	12	12	56	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
45	23	34	100	12	56	20

Example program

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int i, n, num, pos, arr[10] ;
    clrscr();
    printf("\n Enter the number of elements in the array:");
    scanf("%d", &n);
    for(i=0;i<n;i++)
    {
        printf("\n Arr[%d] = :",i);
        scanf("%d", &arr [i]);
    }
    printf("\n Enter the number to be inserted:");
    scanf("%d",&num);
    printf("\n Enter the position at which the number has to be added:");
    scanf("%d", &pos);
    for(i=n;i>=pos;i--)
    {
        arr[i+1] = arr[i];
    }
    arr[pos] = num;
    n++;
    printf("\n The array after insertion is: ",&num);
    for(i=0;i<n;i++)
    {
        printf (" \n Arr [%d]=%d",i,arr[i]);
    }
    getch();
    return 0;
}
```

Output

Enter the number of elements in the array :4

Arr[0] =1

Arr[1] =2

Arr[2] =3

Arr[3] =4

Enter the number to be inserted: 9

Enter the position at which the number has to be added: 2

The array after insertion is

Arr[0] =1

Arr[1] =2

Arr[2] =3

Arr[3] =4

Arr[4] =9

Algorithm for Deleting a last Element from an array

1. Start
2. Set upper_bound = upper_bound -1
3. Stop

Algorithm for Deleting a Middle Element from an array

1. [INITIALIZATION] set I = pos
2. Repeat steps 3 and 4 while $I \leq N - 1$
3. Set $A[I] = A[I + 1]$
4. Set $I = I + 1$
[End of the loop]
5. Set $N = N - 1$
6. Exit

Example

Initial data array

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	34	12	56	20

Calling Insert (Data, 2) will lead to the following in the array

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	12	12	56	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	12	56	56	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	12	56	20	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]
45	23	12	56	20

Example program to delete an element from array at specified position

```
#include <stdio.h>
#define MAX_SIZE 100
int main()
{
    int arr[MAX_SIZE];
    int i, size, pos;
    printf("Enter size of the array : ");
    scanf("%d", &size);
    printf("Enter elements in array : ");
    for(i=0; i<size; i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Enter the element position to delete : ");
    scanf("%d", &pos);

    if(pos<0||pos> size)
    {
        printf("Invalid position! Please enter position between 1 to %d", size);
    }
    else
    {
        for(i=pos-1; i<size-1; i++)
        {
            arr[i]=arr[i + 1];
        }
        size--;
    }
    printf("\nElements of array after delete are : ");
    for(i=0; i<size; i++)
    {
        printf("%d\t", arr[i]);
    }
    return 0;
}
```

Output

```
Enter size of the array: 5
Enter elements in array: 10 20 30 40 50
Enter the element position to delete: 2
Elements of array after delete are: 10    30    40    50
```


Merging of two arrays

Merging of two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence the merged array contains the contents of the first array followed by the contents of the second array.

If the arrays are unsorted, then merging the arrays is very simple, as one just needs to copy the contents of one array into another array.

But merging the arrays is not a trivial task when the two arrays are sorted and the merged array also needs to be sorted.

Array 1	90	56	89	77	69
---------	----	----	----	----	----

Array 2	45	88	76	99	12	58	81
---------	----	----	----	----	----	----	----

Array 3	90	56	89	77	69	45	88	76	99	12	58	81
---------	----	----	----	----	----	----	----	----	----	----	----	----

Merging of two unsorted arrays

If we have two sorted arrays and the resultant merged array needs to be a sorted one, then the task of merging the arrays becomes a little difficult.

Array 1	20	30	40	50	60
---------	----	----	----	----	----

Array 2	15	22	31	45	56	62	78
---------	----	----	----	----	----	----	----

Array 3	15	20	22	30	31	40	45	50	56	60	62	78
---------	----	----	----	----	----	----	----	----	----	----	----	----

Merging of two sorted arrays

The merged array is formed using two sorted arrays. Here we first compare the 1st element of the array 1 with the 1st element of the array 2, and then put the smaller element in the merged array. Since $20 > 15$, we put 15 as the first element in the merged array. Then compare the 2nd element of the second array with the 1st element of the first array, since $20 < 22$, now 20 is stored as second element of the merged array and the process will continue until the merged array will complete.

Passing Arrays to Functions

Passing data values

<pre>main() { intarr[5]={1, 2, 3, 4, 5}; func(arr[3]); }</pre>	<pre>void func(intnum) { printf("%d", num); }</pre>
--	---

Passing addresses

<pre>main() { intarr[5]={1, 2, 3, 4, 5}; func(&arr[3]); }</pre>	<pre>void func(int *num) { printf("%d", *num); }</pre>
---	--

Passing the entire array

<pre>main() { intarr[5] = {1, 2, 3, 4, 5}; func(arr); }</pre>	<pre>void func(intarr[5]) { int i; for(i=0;i<5;i++) printf("%d", arr[i]); }</pre>
---	--

Time Complexity

Algorithm	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Space Complexity

In array, space complexity for worst case is O(n).

Advantages of Array

- Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

Memory Allocation of the array

As we have mentioned, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing.

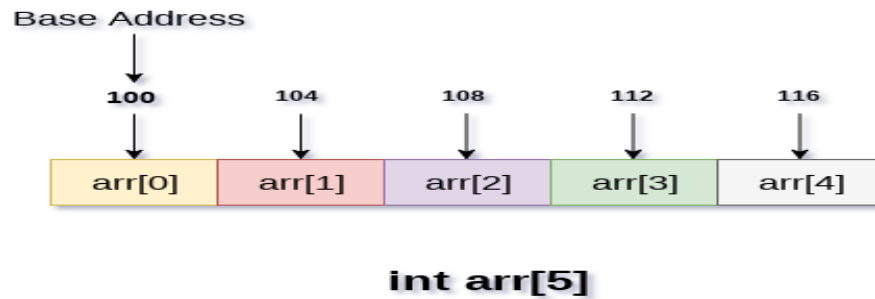
The indexing of the array can be defined in three ways.

0 (zero - based indexing) : The first element of the array will be arr[0].

1 (one - based indexing) : The first element of the array will be arr[1].

n (n - based indexing) : The first element of the array can reside at any random index number.

we have shown the memory allocation of an array arr of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of arr[0]. Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.



In 0 based indexing, if the size of an array is n then the maximum index number, an element can have is $n-1$. However, it will be n if we use 1 based indexing.

Accessing Elements of an array

To access any random element of an array we need the following information:

- Base Address of the array.
- Size of an element in bytes.
- Which type of indexing, array follows?

Address of any element of a 1D array can be calculated by using the following formula:

Byte address of element $A[i] = \text{base address} + \text{size} * (i - \text{first index})$

Example:

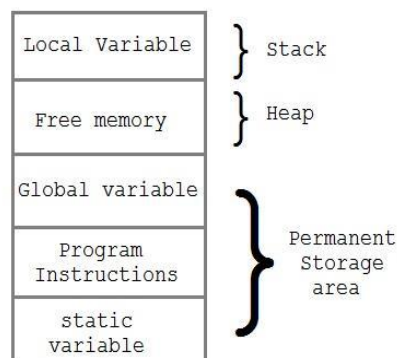
In an array, $A[-10 \dots +2]$, Base address (BA) = 999, size of an element = 2 bytes, find the location of $A[-1]$.

$$\begin{aligned}
 L(A[-1]) &= 999 + [(-1) - (-10)] \times 2 \\
 &= 999 + 18 \\
 &= 1017
 \end{aligned}$$

Memory Allocation Process

Global variables, static variables and program instructions get their memory in permanent storage area whereas local variables are stored in a memory area called Stack.

The memory space between these two regions is known as Heap memory. This region is used for dynamic memory allocation during execution of the program. The size of heap keeps changing.



Dynamic Memory Allocation

The process of allocating memory at runtime is known as dynamic memory allocation. Library routines known as memory management functions are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h** header file.

Function	Description
malloc ()	allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
calloc ()	allocates space for an array of elements, initialize them to zero and then returns a void pointer to the memory
free ()	releases previously allocated memory
realloc ()	modify the size of previously allocated space

Let's understand the difference between static memory allocation and dynamic memory allocation.

Static memory allocation	Dynamic memory allocation
Memory is allocated at compile time.	Memory is allocated at run time.
Memory can't be increased while executing program.	Memory can be increased while executing program.
used in array.	used in linked list.

Malloc () function

- ✓ The malloc() function allocates single block of requested memory.
- ✓ It doesn't initialize memory at execution time, so it has garbage value initially.
- ✓ It returns NULL if memory is not sufficient.

The syntax of malloc() function is:-

ptr=(cast-type*)malloc(byte-size)

(or)

void* malloc(byte-size)

malloc() function is used for allocating block of memory at runtime. This function reserves a block of memory of the given size and returns a pointer of type void. This means that we can assign it to any type of pointer using typecasting. If it fails to allocate enough space as specified, it returns a NULL pointer.

Example Of Malloc() Function.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    clrscr();
```

```

printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
if(ptr==NULL)
{
    printf("Sorry! unable to allocate memory");
    exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
getch();
return 0;
}

```

Output:

```

Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30

```

calloc() function

- ✓ The calloc() function allocates multiple block of requested memory.
- ✓ It initially initialize all bytes to zero.
- ✓ It returns NULL if memory is not sufficient.

The syntax of calloc() function is:

ptr=(cast-type*)calloc(number, byte-size)

(or)

void *calloc(number of items, element-size)

calloc() is another memory allocation function that is used for allocating memory at runtime. calloc function is normally used for allocating memory to derived data types such as arrays and structures. If it fails to allocate enough space as specified, it returns a NULL pointer.

Let's see the example of calloc() function.

```

#include<stdio.h>
#include<stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    clrscr();
    printf("Enter number of elements: ");

```

```

scanf("%d",&n);
ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc
if(ptr==NULL)
{
    printf("Sorry! unable to allocate memory");
    exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
getch();
return 0;
}

```

Output:

```

Enter elements of array: 3
Enter elements of array: 10
20
30
Sum=60

```

realloc() function

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short we can say that, it changes the memory size.

Let's see the syntax of realloc() function.

```

ptr=realloc(ptr, new-size)

(or)

void* realloc(pointer, new-size)

```

Example for : realloc()

```

int *x;
x = (int*)malloc(50 * sizeof(int));
x = (int*)realloc(x,100); //allocated a new memory to variable x

```

free() function

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Syntax of free() function.

```

free(ptr)

```

Diffrence between malloc() and calloc()

Calloc()	Malloc()
1 .calloc() initializes the allocated memory with 0 value.	1. Malloc () initializes the allocated memory with garbage values.
2. Number of arguments is 2	2.Number of argument is 1
Syntax : (cast_type *)calloc(blocks, size_of_block);	Syntax : (cast_type *)malloc(Size_in_bytes);

Passing array to the function:

As we have mentioned earlier that, the name of the array represents the starting address or the address of the first element of the array. All the elements of the array can be traversed by using the base address.

Example:

```
#include <stdio.h>
int summation(int[]);
void main ()
{
    int arr[5] = {0,1,2,3,4};
    int sum = summation(arr);
    printf("%d",sum);
}
int summation (int arr[])
{
    int sum=0,i;
    for (i = 0; i<5; i++)
    {
        sum = sum + arr[i];
    }
    return sum;
}
```

The above program defines a function named as summation which accepts an array as an argument. The function calculates the sum of all the elements of the array and returns it.

Two Dimensional Arrays

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

How to declare 2D Array

The syntax of declaring two dimensional array is very much similar to that of a one dimensional array.

int arr[max_rows][max_columns];

	0	1	2	n-1
0	a[0][0]	a[0][1]	a[0][2]	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	a[2][n-1]
3	a[3][0]	a[3][1]	a[3][2]	a[3][n-1]
4	a[4][0]	a[4][1]	a[4][2]	a[4][n-1]
.
.
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]	a[n-1][n-1]

a[n][n]

The two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by a[0][0] where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

How do we access data in a 2D array

The elements of 2D arrays can be random accessed. Similar to one dimensional arrays, we can access the individual cells in a 2D array by using the indices of the cells. There are two indices attached to a particular cell, one is its row number while the other is its column number.

However, we can store the value stored in any particular cell of a 2D array to some variable x by using the following syntax.

int x = a[i][j];

where i and j is the row and column number of the cell respectively.

Example:

```
for ( int i=0; i<n ;i++)
{
    for (int j=0; j<n; j++)
    {
        a[i][j] = 0;
    }
}
```

We know that, when we declare and initialize one dimensional array in C programming simultaneously, we don't need to specify the size of the array. However this will not work with 2D arrays. We will have to define at least the second dimension of the array.

The syntax to declare and initialize the 2D array is

int arr[2][2] = {0,1,2,3};

The number of elements that can be present in a 2D array will always be equal to **(number of rows * number of columns)**.

Example : Storing User's data into a 2D array and printing it.

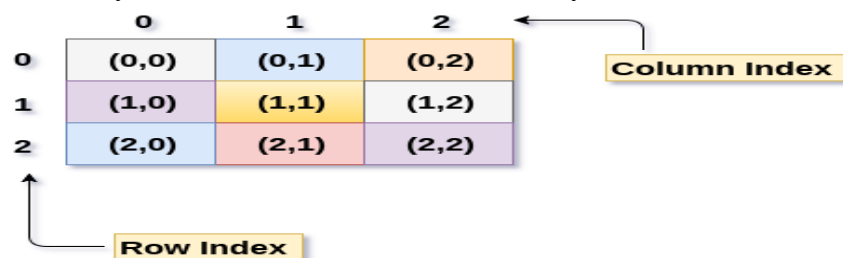
```
#include <stdio.h>
void main ()
{
    int arr[3][3],i,j;
    clrscr();
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
            printf("Enter a[%d][%d]: ",i,j);
            scanf("%d",&arr[i][j]);
        }
    }
    printf("\n printing the elements ....\n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for (j=0;j<3;j++)
        {
            printf("%d\t",arr[i][j]);
            getch();
        }
    }
}
```

Mapping 2D array to 1D array

When it comes to map a 2 dimensional array, most of us might think that why this mapping is required. However, 2 D arrays exists from the user point of view. 2D arrays are created to implement a relational database table lookalike data structure, in computer memory, the storage technique for 2D array is similar to that of an one dimensional array.

The size of a two dimensional array is equal to the multiplication of number of rows and the number of columns present in the array. We do need to map two dimensional array to the one dimensional array in order to store them in the memory.

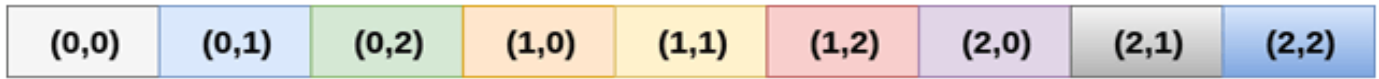
A 3 X 3 two dimensional array is shown in the following image. However, this array needs to be mapped to a one dimensional array in order to store it into the memory.



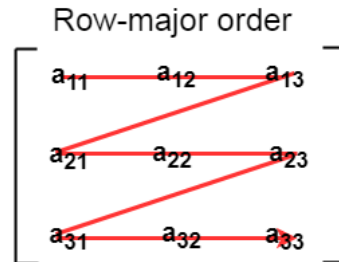
There are two main techniques of storing 2D array elements into memory

1. Row major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is.

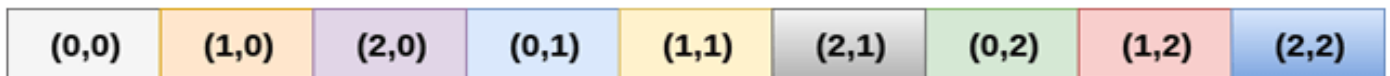


First, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.

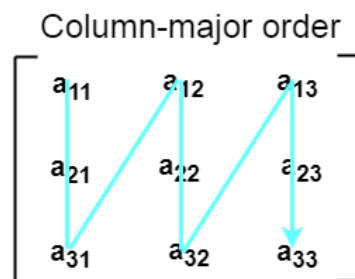


2. Column major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array is.



First, the 1st column of the array is stored into the memory completely, and then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.



APPLICATION OF ARRAYS

- ✓ Arrays are widely used to implement mathematical vectors, matrices, and other kinds of rectangular tables.
- ✓ Many databases include one-dimensional arrays whose elements are records.
- ✓ Arrays are also used to implement other data structures such as strings, stacks, queues, heaps, and hash tables. We will read about these data structures in the subsequent chapters.
- ✓ Arrays can be used for sorting elements in ascending or descending order.

Structure

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure.

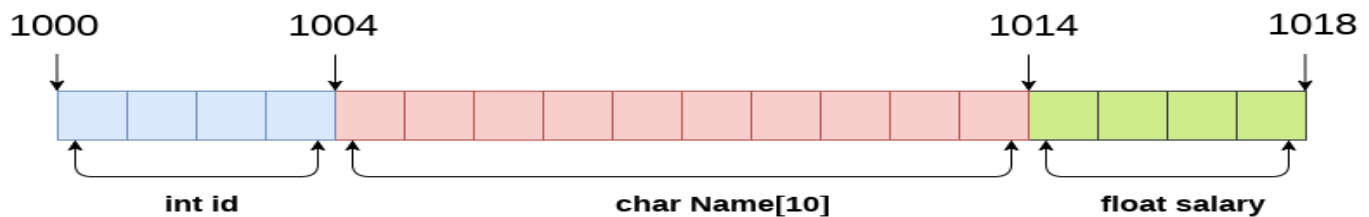
Syntax

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    .
    .

    data_type memeber;
};
```

Let's see the example to define a structure for an entity employee.


```
struct employee
{
    int id;
    char name[10];
    float salary;
};
```



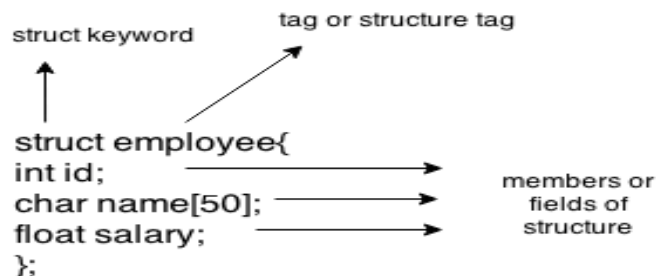
```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

sizeof (emp) = 4 + 10 + 4 = 18 bytes

where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte

 → 1 byte

Here, struct is the keyword; employee is the name of the structure; id, name, and salary are the members or fields of the structure.



Advantages

- ✓ It can hold variables of different data types.
- ✓ We can create objects containing different types of attributes.
- ✓ It allows us to re-use the data layout across programs.
- ✓ It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

Example program

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    struct employee
    {
        int id ;
        float salary ;
        int mobile ;
    } ;
    struct employee e1,e2,e3 ;
    clrscr();
    printf ("\nEnter ids, salary & mobile no. of 3 employee\n");
    scanf ("%d %f %d", &e1.id, &e1.salary, &e1.mobile);
    scanf ("%d%f %d", &e2.id, &e2.salary, &e2.mobile);
    scanf ("%d %f %d", &e3.id, &e3.salary, &e3.mobile);
    printf ("\n Entered Result ");
    printf ("\n%d %f %d", e1.id, e1.salary, e1.mobile);
    printf ("\n%d%f %d", e2.id, e2.salary, e2.mobile);
    printf ("\n%d %f %d", e3.id, e3.salary, e3.mobile);
    getch();
}
```

Union

Like structure, Union in c language is a user-defined data type that is used to store the different type of elements.

At once, only one member of the union can occupy the memory. In other words, we can say that the size of the union in any instance is equal to the size of its largest element.

<u>Structure</u>	<u>Union</u>
<pre>struct Employee{ char x; // size 1 byte int y; //size 2 byte float z; //size 4 byte }e1; //size of e1 = 7 byte</pre>	<pre>union Employee{ char x; // size 1 byte int y; //size 2 byte float z; //size 4 byte }e1; //size of e1 = 4 byte</pre>
size of e1= 1 + 2 + 4 = 7	size of e1= 4 (maximum size of 1 element)

Advantage of union over structure

It occupies less memory because it occupies the size of the largest member only.

Disadvantage of union over structure

Only the last entered data can be stored in the union. It overwrites the data previously stored in the union.

Union example

```
#include <stdio.h>
#include <string.h>
```

```

union employee
{
    int id;
    char name[50];
}e1; //declaring e1 variable for union
int main( )
{
    //store first employee information
    e1.id=101;
    strcpy(e1.name, "Super star");//copying string into char array
    //printing first employee information
    printf( "employee 1 id : %d\n", e1.id);
    printf( "employee 1 name : %s\n", e1.name);
    return 0;
}

```

Output:

```

employee 1 id : 1869508435
employee 1 name : Super star

```

Sorting

Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. A collection of records called a list where every record has one or more fields. The fields which contain a unique value for each record is termed as the key field.

For example, a phone number directory can be thought of as a list where each record has three fields - 'name' of the person, 'address' of that person, and their 'phone numbers'. Being unique phone number can work as a key to locate any record in the list.

Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion. Sorting is performed according to some key value of each record.

The records are either sorted either numerically or alphanumerically. The records are then arranged in ascending or descending order depending on the numerical value of the key. Here is an example, where the sorting of a lists of marks obtained by a student in any particular subject of a class.

Categories of Sorting

The techniques of sorting can be divided into two categories.

- ✓ Internal Sorting
- ✓ External Sorting

Internal Sorting: If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

External Sorting: When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.

The complexity of the sorting algorithms

The complexity of sorting algorithm calculates the running time of a function in which 'n' number of items are to be sorted. The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems. The most noteworthy of these considerations are:

- ✓ The length of time spent by the programmer in programming a specific sorting program
- ✓ Amount of machine time necessary for running the program
- ✓ The amount of memory necessary for running the program

The Efficiency of Sorting Techniques

To get the amount of time required to sort an array of 'n' elements by a particular method, the normal approach is to analyze the method to find the number of comparisons (or exchanges) required by it. Most of the sorting techniques are data sensitive, and so the metrics for them depends on the order in which they appear in an input array.

Various sorting techniques are analyzed in various cases and named these cases are:

- ✓ Best case
- ✓ Worst case
- ✓ Average case

Hence, the result of these cases is often a formula giving the average time required for a particular sort of size 'n.' Most of the sort methods have time requirements that range from $O(n \log n)$ to $O(n^2)$.

Types of Sorting Techniques

- ✓ Bubble Sort
- ✓ Selection Sort
- ✓ Insertion Sort
- ✓ Merge Sort
- ✓ Quick Sort
- ✓ Heap Sort
- ✓ Radix Sort

Quick Sort Algorithm

Quick sort is a fast sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by C. A. R. Hoare.

The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use divide and conquer strategy. In quick sort, the partition of the list is performed based on the element called pivot. Here pivot element is one of the elements in the list.

The list is divided into two partitions such that "all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot".

Step by Step Process/ Algorithm

In Quick sort algorithm, partitioning of the list is performed using following steps...

- Step 1 - Consider the first element of the list as pivot (i.e., Element at first position in the list).
- Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.
- Step 3 - Increment i until $\text{list}[i] > \text{pivot}$ then stop.
- Step 4 - Decrement j until $\text{list}[j] < \text{pivot}$ then stop.
- Step 5 - If $i < j$ then exchange $\text{list}[i]$ and $\text{list}[j]$.
- Step 6 - Repeat steps 3,4 & 5 until $i > j$.
- Step 7 - Exchange the pivot element with $\text{list}[j]$ element.

Consider the following unsorted list of elements...

List

5	3	8	1	4	6	2	7
---	---	---	---	---	---	---	---

Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.

List

5	3	8	1	4	6	2	7
---	---	---	---	---	---	---	---

leftright
pivot

Compare List[left] with List[pivot]. If List[left] is greater than List[pivot] then stop left otherwise move left to the next.

Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.

Repeat the same until left >= right.

If both left & right are stopped but left < right then swap List[left] with List[right] and continue the process.

If left >= right then swap List[pivot] with List[right].

List

5	3	8	1	4	6	2	7
---	---	---	---	---	---	---	---

leftright
pivot

Compare List[left] < List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.

Compare List[right] > List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.

List

5	3	8	1	4	6	2	7
---	---	---	---	---	---	---	---

leftright
pivot

Here left & right both are stopped and left is not greater than right so we need to swap List[left] and List[right]

List

5	3	2	1	4	6	8	7
---	---	---	---	---	---	---	---

leftright
pivot

Compare List[left] < List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.

Compare List[right] > List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.

List

5	3	2	1	4	6	8	7
---	---	---	---	---	---	---	---

rightleft
pivot

Here left & right both are stopped and left is greater than right so we need to swap List[pivot] and List[right]

List

4	3	2	1	5	6	8	7
---	---	---	---	---	---	---	---

Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.

Repeat the same process on the left sublist and right sublist to the number 5.

List

4	3	2	1	5	6	8	7
---	---	---	---	---	---	---	---

leftrightleftright
pivotpivot

In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.

List

1	3	2	4	5	6	8	7
---	---	---	---	---	---	---	---

leftright
pivot

In the right sublist left is greater than the pivot, left will stop at same position.

As the List[right] is greater than List[pivot], right moves towards left and stops at pivot number position.

Now left > right so we swap pivot with right. (6 is swap by itself).

List

1	3	2	4	5	6	8	7
---	---	---	---	---	---	---	---

leftright
pivot

Repeat the same recursively on both left and right sublists until all the numbers are sorted.

The final sorted list will be as follows...

List

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

Complexity of the Quick Sort Algorithm

To sort an unsorted list with 'n' number of elements, we need to make $((n-1) + (n-2) + (n-3) + \dots + 1) = (n(n-1))/2$ number of comparisons in the worst case. If the list is already sorted, then it requires 'n' number of comparisons.

Worst Case : $O(n^2)$

Best Case : $O(n \log n)$

Average Case : $O(n \log n)$

Example Program for Quick Sort

```
#include<stdio.h>
#include<conio.h>
void quickSort(int [10],int,int);
void main()
{
    int list[20],size,i;
    printf("Enter size of the list: ");
    scanf("%d",&size);
    printf("Enter %d integer values: ",size);
    for(i = 0; i < size; i++)
        scanf("%d",&list[i]);
    quickSort(list,0,size-1);
    printf("List after sorting is: ");
    for(i = 0; i < size; i++)
        printf(" %d",list[i]);
    getch();
}
void quickSort(int list[10],int first,int last){
    int pivot,i,j,temp;
    if(first < last)
    {
        pivot = first;
        i = first;
        j = last;
        while(i < j)
        {
            while(list[i] <= list[pivot] && i < last)
                i++;
            while(list[j] > list[pivot])
                j--;
            if(i < j)
            {
                temp = list[i];
                list[i] = list[j];
```



```

        list[j] = temp;
    }
}
temp = list[pivot];
list[pivot] = list[j];
list[j] = temp;
quickSort(list,first,j-1);
quickSort(list,j+1,last);
}
}

```

Out Put

Enter the size of the list: 8

Enter the 8 Integer values: 5 3 8 1 4 6 2 7

List after sorting is: 1 2 3 4 5 6 7 8

Merge Sort Algorithm

Merge Sort follows the rule of Divide and Conquer to sort a given set of numbers/elements, recursively, hence consuming less time.

In the last two tutorials, we learned about Selection Sort and Insertion Sort, both of which have a worst-case running time of $O(n^2)$. As the size of input grows, insertion and selection sort can take a long time to run.

Merge sort, on the other hand, runs in $O(n \cdot \log n)$ time in all the cases.

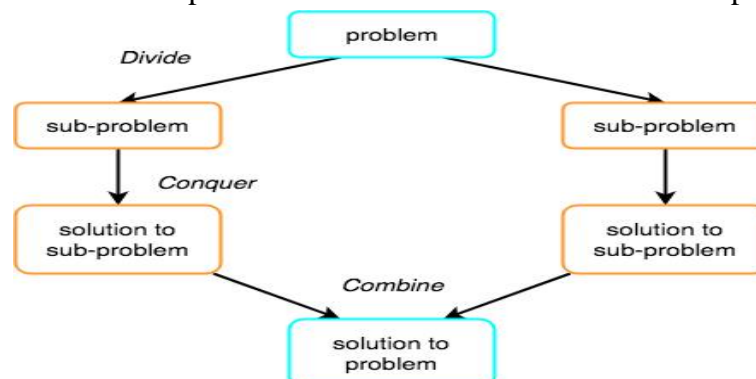
Divide and Conquer

If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

In Merge Sort, the given unsorted array with n elements is divided into n sub arrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these sub arrays, to produce new sorted sub arrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

- ✓ Divide the problem into multiple small problems.
- ✓ Conquer the sub problems by solving them. The idea is to break down the problem into atomic sub problems, where they are actually solved.
- ✓ Combine the solutions of the sub problems to find the solution of the actual problem.



How Merge Sort Works?

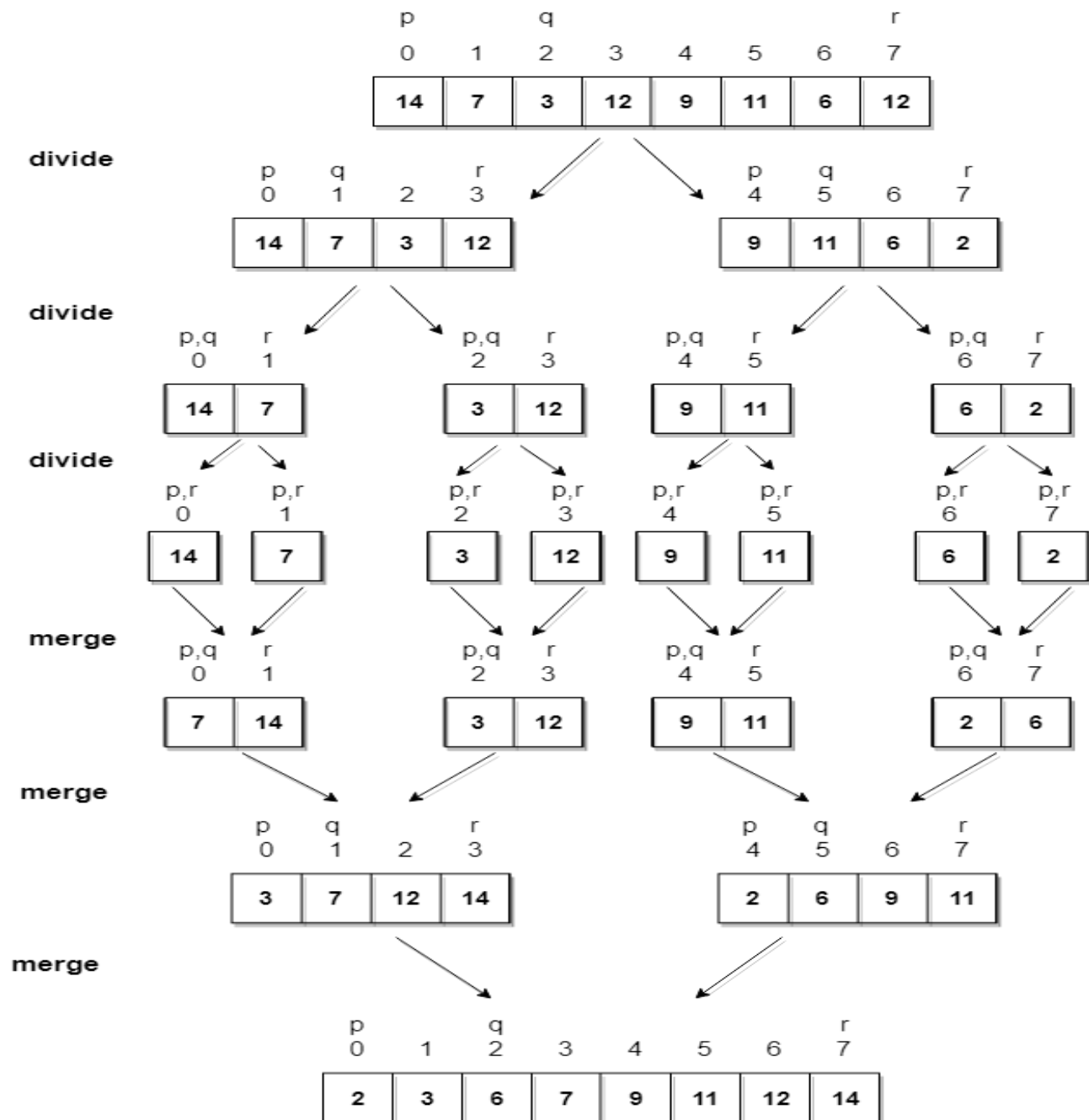
As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, sorting a given array.

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two sub arrays with 3 elements each.

But breaking the original array into 2 smaller sub arrays is not helping us in sorting the array.

So we will break these sub arrays into even smaller sub arrays, until we have multiple sub arrays with single element in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into sub arrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted sub arrays, step by step to form one single sorted array. Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, and 12}



In merge sort we follow the following steps:

- We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
- Then we find the middle of the array using the formula $(p + r)/2$ and mark the middle index as q, and break the array into two sub arrays, from p to q and from q + 1 to r index.
- Then we divide these 2 sub arrays again, just like we divided our main array and this continues.
- Once we have divided the main array into sub arrays with single elements, then we start merging the sub arrays.

Example program

```
#include<stdio.h>
void mergeSort(int[],int,int);
void merge(int[],int,int,int);
void main ()
{
    int a[10]= { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    int i;
    clrscr();
    mergeSort(a,0,9);
    printf("printing the sorted elements");
    for(i=0;i<10;i++)
    {
        printf("\n%d\n",a[i]);
    }
    getch();
}
void mergeSort(int a[], int beg, int end)
{
    int mid;
    if(beg<end)
    {
        mid = (beg+end)/2;
        mergeSort(a,beg,mid);
        mergeSort(a,mid+1,end);
        merge(a,beg,mid,end);
    }
}
void merge(int a[], int beg, int mid, int end)
{
    int i=beg,j=mid+1,k,index = beg;
    int temp[10];
    while(i<=mid && j<=end)
    {
        if(a[i]<a[j])
        {
            temp[index] = a[i];
            index++;
            i++;
        }
        else
        {
            temp[index] = a[j];
            index++;
            j++;
        }
    }
    while(i<=mid)
    {
        temp[index] = a[i];
        index++;
        i++;
    }
    while(j<=end)
    {
        temp[index] = a[j];
        index++;
        j++;
    }
    for(i=beg;i<=end;i++)
    {
        a[i] = temp[i];
    }
}
```

```

        i = i+1;
    }
    else
    {
        temp[index] = a[j];
        j = j+1;
    }
    index++;
}
if(i>mid)
{
    while(j<=end)
    {
        temp[index] = a[j];
        index++;
        j++;
    }
}
else
{
    while(i<=mid)
    {
        temp[index] = a[i];
        index++;
        i++;
    }
}
k = beg;
while(k<index)
{
    a[k]=temp[k];
    k++;
}
}

```

Output:

Printing the sorted elements

```

7
9
10
12
23
23
34
44
78
101

```

Heap Sort Algorithm

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called Heap Tree. In this sorting algorithm, we use Max Heap to arrange list of elements in Descending order and Min Heap to arrange list elements in Ascending order.

Step by Step Process/ Algorithm

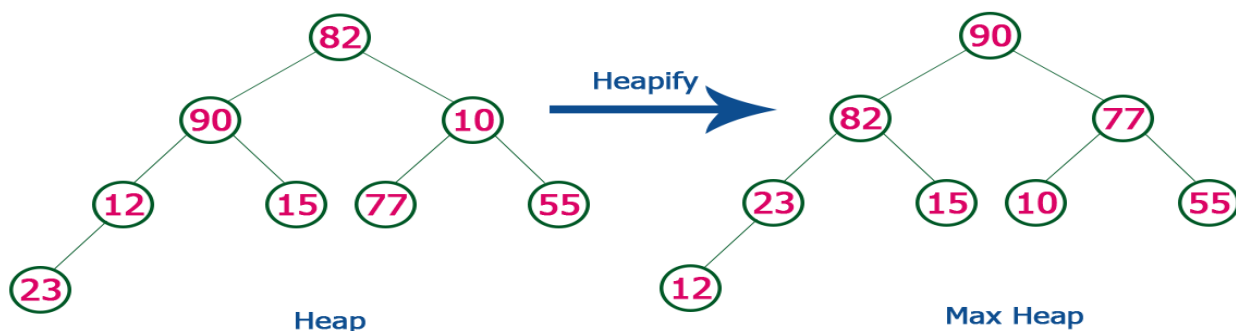
- Step 1 - Construct a Binary Tree with given list of Elements.
- Step 2 - Transform the Binary Tree into Min Heap.
- Step 3 - Delete the root element from Min Heap using Heapify method.
- Step 4 - Put the deleted element into the Sorted list.
- Step 5 - Repeat the same until Min Heap becomes empty.
- Step 6 - Display the sorted list.

Example Problem

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

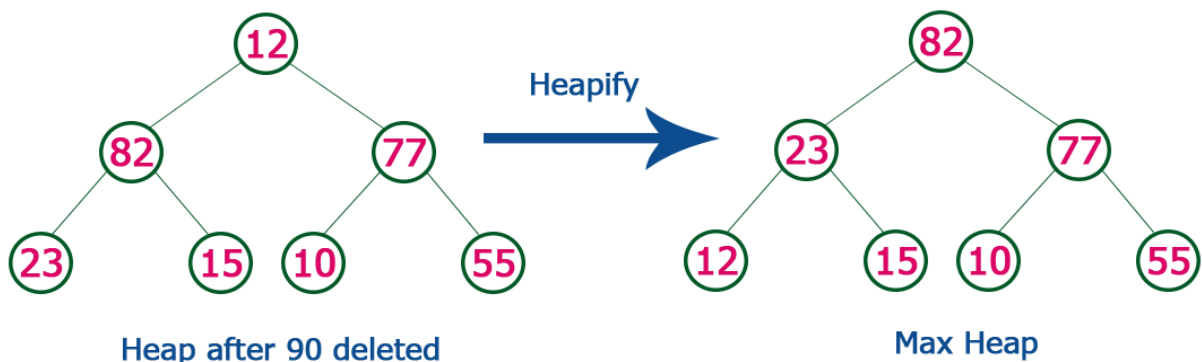
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

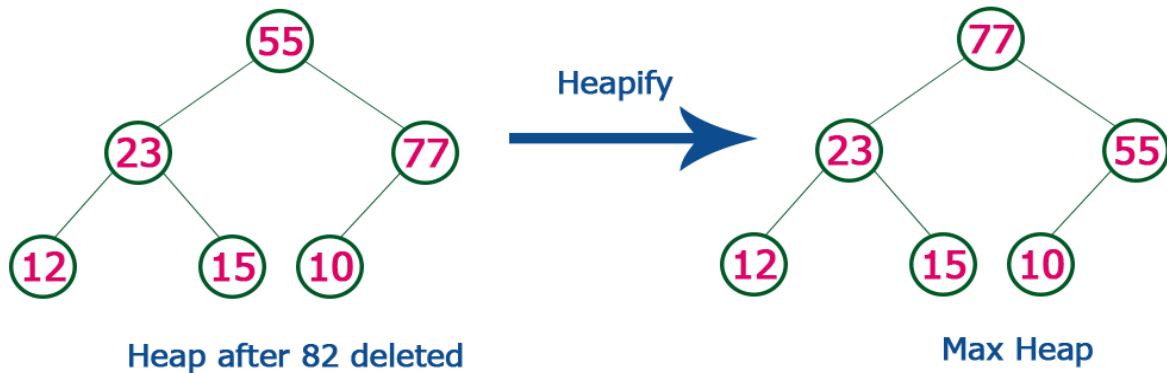
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, 90

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, 82, 90

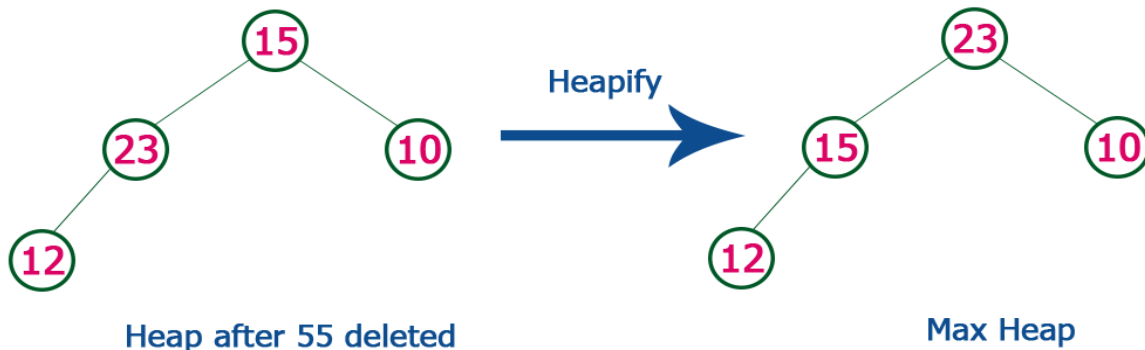
Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, 77, 82, 90

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

12, 15, 10, 23, 55, 77, 82, 90

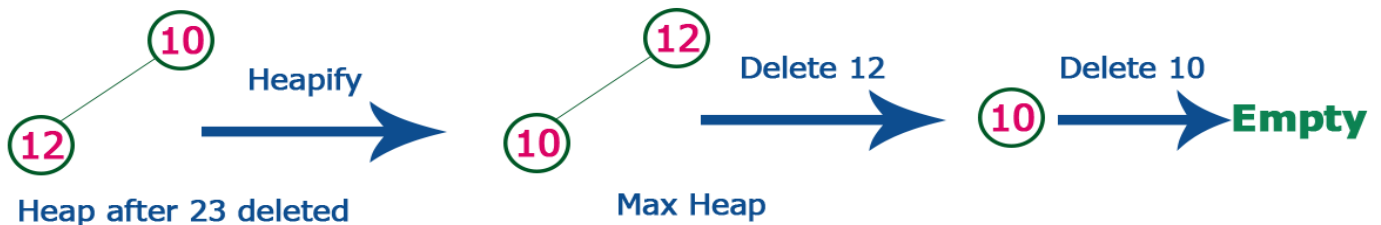
Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

12, 15, 10, 23, 55, 77, 82, 90

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Complexity of the Heap Sort Algorithm

To sort an unsorted list with 'n' number of elements, following are the complexities...

Worst Case : $O(n \log n)$

Best Case : $O(n \log n)$

Average Case : $O(n \log n)$

Example program

```

#include<stdio.h>
int temp;
void heapify(int arr[], int size, int i)
{
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;
    if (left < size && arr[left] > arr[largest])
        largest = left;

```

```

if (right < size && arr[right] > arr[largest])
largest = right;
if (largest != i)
{
temp = arr[i];
arr[i]= arr[largest];
arr[largest] = temp;
heapify(arr, size, largest);
}
}
void heapSort(int arr[], int size)
{
int i;
for (i = size / 2 - 1; i >= 0; i--)
heapify(arr, size, i);
for (i=size-1; i>=0; i--)
{
temp = arr[0];
arr[0]= arr[i];
arr[i] = temp;
heapify(arr, i, 0);
}
}
void main()
{
int arr[] = {1, 10, 2, 3, 4, 1, 2, 100,23, 2};
int i;
int size = sizeof(arr)/sizeof(arr[0]);
clrscr();
heapSort(arr, size);
printf("printing sorted elements\n");
for (i=0; i<size; ++i)
printf("%d\n",arr[i]);
getch();
}

```

Output:

Printing sorted elements

```

1
1
2
2
2
3
4
10
23
100

```


Stack, Queue and Linked lists

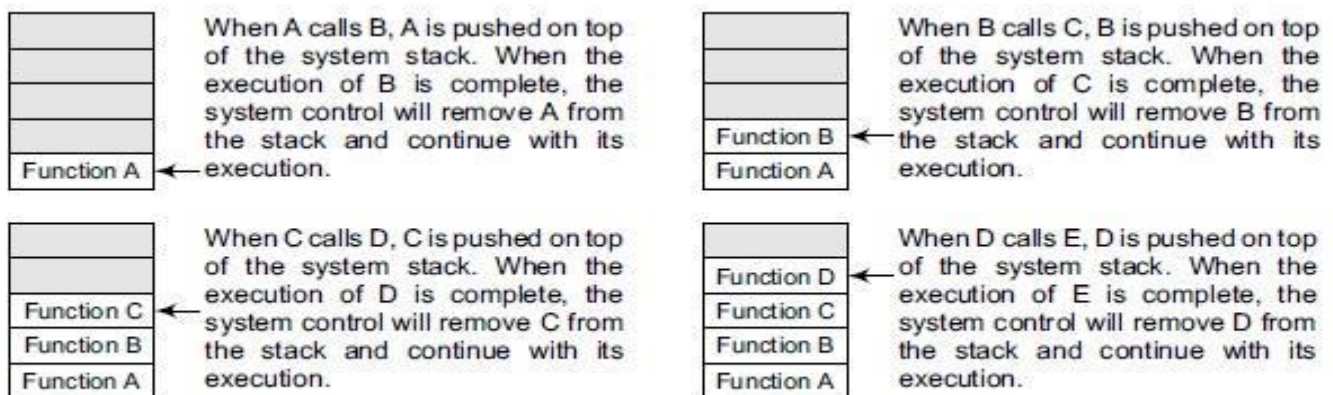
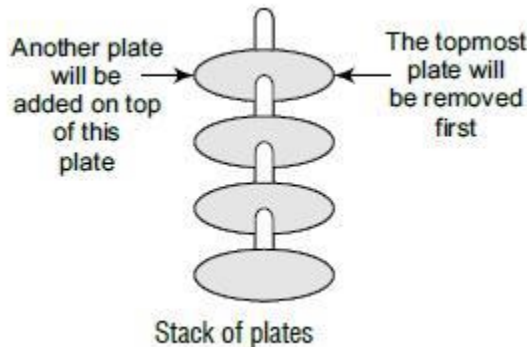
Stacks, Stacks using Dynamic Arrays, Queues, Circular Queues Using Dynamic Arrays, Evaluation of Expressions, Multiple Stacks and Queues. Linked lists: Singly Linked Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Additional List Operations, Doubly Linked Lists.

INTRODUCTION

Stack is an important data structure which stores its elements in an ordered manner. We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Fig. Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.

A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

Now the question is where do we need stacks in computer science? The answer is in function calls. Consider an example, where we are executing function A. In the course of its execution, function A calls another function B. Function B intern calls another function C, which calls function D.



System stack in the case of function calls

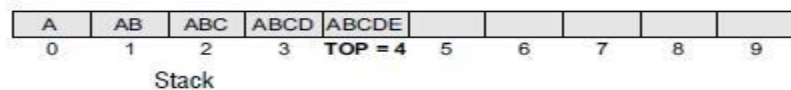
In order to keep track of the returning point of each active function, a special stack called system stack or call stack is used. Whenever a function calls another function, the calling function is pushed onto the Top of the stack. This is because after the called function gets executed, the control is passed back to the calling function.

Now when function E is executed, function D will be removed from the top of the stack and executed. Once function D gets completely executed, function C will be removed from the stack for execution. The whole procedure will be repeated until all the functions get executed. Let us look at the stack after each function is executed.

The system stack ensures a proper execution order of functions. Therefore, stacks are frequently used in situations where the order of processing is very important, especially when the processing needs to be postponed until other conditions are fulfilled.

ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX-1, then the stack is full. (You must be wondering why we have written MAX-1. It is because array indices start from 0.)



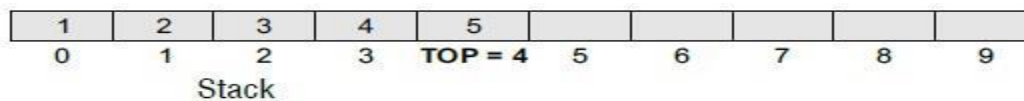
The stack in Fig. shows that TOP = 4, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

OPERATIONS ON A STACK

A stack supports three basic operations: **push**, **pop**, and **peek**. The push operation adds an element to the top of the stack and the pop operation removes the element from the top of the stack. The peek operation returns the value of the topmost element of the stack.

Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if TOP=MAX-1, because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.



To insert an element with value 6, we first check if TOP=MAX-1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack [TOP].

Algorithm to PUSH an element in a stack

```

Step 1: If Top = Max-1, Then
    Print "Overflow"
    Goto Step 4
[End Of If]
Step 2: Set Top = Top + 1
Step 3: Set Stack [Top] = Value
Step 4: End
    
```



To insert an element in a stack, we first check for the OVERFLOW condition, then TOP is incremented so that it points to the next location in the array, the value is stored in the stack at the location pointed by TOP.

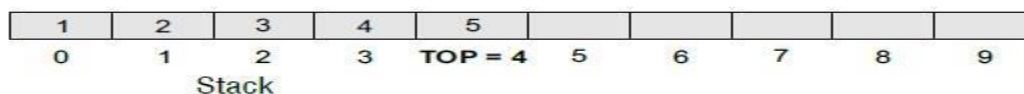
Pop Operation

The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

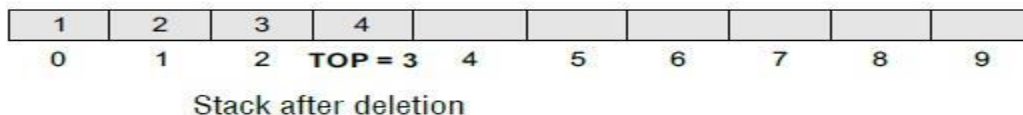
Algorithm to POP an element from a stack

```

Step 1: If Top = Null, Then
    Print "Underflow"
    Goto Step 4
[End Of If]
Step 2: Set Val = Stack[Top]
Step 3: Set Top = Top - 1
Step 4: End
    
```



To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP.



To delete an element from a stack, we first check for the UNDERFLOW condition, the value of the location in the stack pointed by TOP is stored in VAL, TOP is decremented.

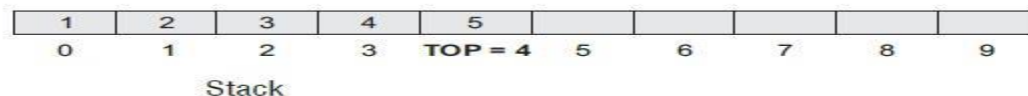
Peek/Peep Operation

Peek is an operation that returns the value of the top most element of the stack without deleting it from the stack. However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.

Algorithm for Peek/Peep Operation

```

Step 1: If Top =Null, Then
    Print "Stack Is Empty"
    Go To Step 3
[End of If]
Step 2: Return Stack [Top]
Step 3: End
    
```



Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

Stacks using Dynamic Arrays

- Shortcoming of static stack implementation: is the need to know at compile-time, a good bound(MAX_STACK_SIZE) on how large the stack will become.
- This shortcoming can be overcome by
 - using a dynamically allocated array for the elements &
 - then increasing the size of the array as needed
- Initially, capacity=1 where capacity=maximum no. of stack-elements that may be stored in array.
- The CreateS() function can be implemented as follows

Stack CreateS()::=

```
    struct element {  
        int key; };
```

```
    element *stack;
```

```
    MALLOC(stack,sizeof(*stack));
```

```
    int capacity=-1;
```

```
    int top=-1;
```

```
    Boolean IsEmpty(Stack)::= top<0;
```

```
    Boolean IsFull(Stack) ::= top>=capacity-1;
```

```
    void stackFull()
```

```
    {
```

```
        REALLOC(stack,2*capacity*sizeof(*stack));
```

```
        capacity=2*capacity;
```

```
    }
```

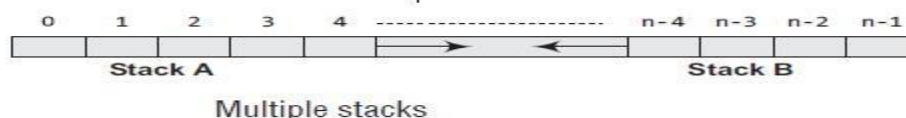
- Once the stack is full, realloc() function is used to increase the size of array.
- In array-doubling, we double array-capacity whenever it becomes necessary to increase the capacity of an array.

ANALYSIS

- In worst case, the realloc function needs to
 - allocate $2*capacity*sizeof(*stack)$ bytes of memory and
 - copy $capacity*sizeof(*stack)$ bytes of memory from the old array into the new one.
- The total time spent over all array doublings = $O(2^k)$ where $capacity=2^k$
- Since the total number of pushes is more than 2^{k-1} , the total time spend in array doubling is $O(n)$ where n =total number of pushes.

MULTIPLE STACKS

While implementing a stack using an array, we had seen that the size of the array must be known in advance. If the stack is allocated less space, then frequent OVERFLOW conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array. In case we allocate a large amount of space for the stack, it may result in sheer wastage of memory. Thus, there lies a trade-off between the frequency of overflows and the space allocated. So, a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size.



An array $STACK[n]$ is used to represent two stacks, Stack A and Stack B. The value of n is such that the combined size of both the stacks will never exceed n . While operating on these stacks, it is important to note one thing Stack A will grow from left to right, whereas Stack B will grow from right to left at the same time. Extending this concept to multiple stacks, a stack can also be used to represent number of stacks in the same array. That is, if we have a $STACK[n]$, then each stack i will be allocated an equal amount of space bounded by indices $b[i]$ and $e[i]$.



- Assume that we have 'n' stacks, we can divide the available memory into 'n' segments (Fig: 3.18).
- Let 'i' = stack number of one of n stacks.
- Let $boundary[i]$ ($0 \leq i < MAX_STACKS$) points to position immediately to left of bottom element of stack i , while $top[i]$ ($0 \leq i < MAX_STACKS$) points to top element.
- Stack i is empty iff $boundary[i] = top[i]$
- The relevant declaration are:

```
#define MEMORY_SIZE 100 /* size of memory */
#define MAX_STACKS 10 /* max number of stacks plus 1 */
element memory[MEMORY_SIZE];
int top[MAX_STACKS];
int n; /* number of stacks entered by the user */
```

- To divide the array into equal segment, we use the following code:

```
top[0] = boundary[0] = -1;
for(j=1; j<n; j++)
    top[j] = boundary[j] = (MEMORY_SIZE/n)*j;
boundary[n] = MEMORY_SIZE-1;
```

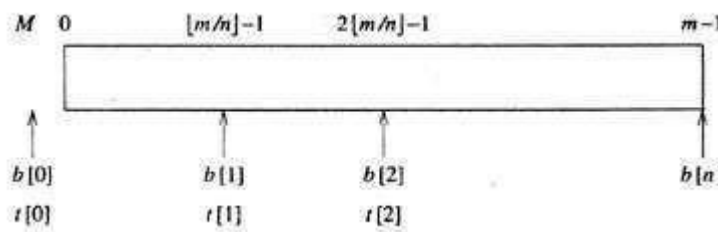


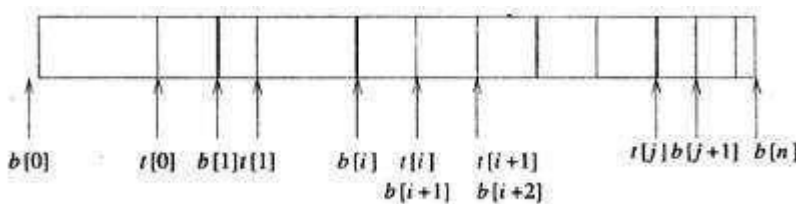
Figure3.18:Initial configuration for n stacks in memory[m]

```
void add(int i, element item)
{
    /* add an item to the ith stack */
    if (top[i] == boundary[i+1])
        stack_full(i);
    memory[++top[i]] = item;
}
```

Program3.16:Addanitemto theith stack

```
element delete(int i)
{
    /* remove top element from the ith stack */
    if (top[i] == boundary[i])
        return stack_empty(i);
    return memory[top[i]--];
}
```

Program3.17:Deletean itemfromtheithstack



- In push function, $\text{top}[i] == \text{boundary}[i+1]$ condition implies only that a particular stack ran out of memory, not that the entire memory is full. (In fact, there may be a lot of unused space between other stacks in array memory).
- Therefore, we create an error recovery function, `stackFull`, which determines if there is any free space in memory.
- If there is space available, it should shift the stacks so that space is allocated to the full stack.
- We can guarantee that `stackFull` adds elements as long as there is free space in array memory if we:
 - 1) Determine the least j , $i < j < n$ such that there is free space between stacks j and $j+1$ i.e. $\text{top}[j] < \text{boundary}[j+1]$. If there is such a j , then move stacks $i+1, i+2, \dots, j$ one position to the right. This creates a space between stacks i and $i+1$.
 - 2) If there is no j as in (i), then look to the left of stack i . Find the largest j such that $0 \leq j < i$ and there is space between stacks j and $j+1$ i.e. $\text{top}[j] < \text{boundary}[j+1]$. If there is such a j , then move stacks $j+1, j+2, \dots, i$ one space to the left. This also creates a space between stacks i and $i+1$.
 - 3) If there is no j satisfying either (i) or (ii), then all `MEMORY_SIZE` spaces of memory are utilized and there is no free space. In this case, `stackFull` terminates with an error message

APPLICATIONS OF STACKS

In this section we will discuss typical problems where stacks can be easily applied for a simple and efficient solution. The topics that will be discussed in this section include the following:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi
- Parsing
- Browsers
- Editors
- Tree Traversals

Evaluation of Arithmetic Expressions

Polish Notations

Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions. But before learning about prefix and postfix notations, let us first see what an infix notation is. We all are familiar with the infix notation of writing algebraic expressions.

While writing an arithmetic expression using infix notation, the operator is placed in between the operands. For example, $A+B$; here, plus operator are placed between the two operands A and B . Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer needs a lot of information to evaluate the expression. Information is needed about operator precedence and

associability rules, and brackets which override these rules. So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix notation was developed by **Jan Łukasiewicz** who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN.

In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as $A+B$ in infix notation, the same expression can be written as $AB+$ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation.

The expression $(A + B) * C$ can be written as:

$[AB+]*C$

$AB+C*$ in the postfix notation

A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands. For example, given a postfix notation $AB+C*$. While evaluation, addition will be performed prior to multiplication.

Thus we see that in a postfix notation, operators are applied to the operands that are immediately left to them. In the example, $AB+C*$, $+$ is applied on A and B , then $*$ is applied on the result of addition and C .

Although a prefix notation is also evaluated from left to right, the only difference between a postfix notation and a prefix notation is that in a prefix notation, the operator is placed before the operands. For example, if $A+B$ is an expression in infix notation, then the corresponding expression in prefix notation is given by $+AB$.

While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator. Like postfix, prefix expressions also do not follow the rules of operator precedence and associativity, and even brackets cannot alter the order of evaluation.

Convert the following infix expressions into postfix expressions.

Solution

(i) $(A-B) * (C+D)$

$[AB-] * [CD+]$

$AB-CD+*$

(ii) $(A + B) / (C + D) - (D * E)$

$[AB+] / [CD+] - [DE*]$

$[AB+CD+]/ - [DE*]$

$AB+CD+/DE*-$

Conversion of an Infix Expression into a Postfix Expression

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only $+$, $-$, $*$, $/$, $\%$ operators. The precedence of these operators can be given as follows:

Higher priority $*$, $/$, $\%$

Lower priority +, –

No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression $A + B * C$, then first $B * C$ will be done and the result will be added to A. But the same expression if written as, $(A + B) * C$, will evaluate $A + B$ first and then the result will be multiplied with C.

The algorithm given below transforms an infix expression into postfix expression, as shown in Fig. The algorithm accepts an infix expression that may contain operators, operands, and parentheses. For simplicity, we assume that the infix operation contains only modulus (%), multiplication (*), division (/), addition (+), and subtraction (—) operators and that operators with same precedence are performed from left-to-right.

The algorithm uses a stack to temporarily hold operators. The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack. The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression. The algorithm is repeated until the stack is empty.

Algorithm to convert an Infix notation into postfix notation

Step 1: Add ‘)’ to the end of the infix expression

Step 2: Push ‘(’ on to the stack

Step 3: Repeat until each character in the infix notation is scanned

IF a ‘(’ is encountered, push it on the stack

IF an operand (whether a digit or an alphabet) is encountered,

Add it to the postfix expression.

IF a ‘)’ is encountered, then;

a . Repeatedly pop from stack and add it to the postfix expression until a ‘(’ is encountered.

b. Discard the ‘(’. That is, remove the ‘(’ from stack and do not add it to the postfix expression

IF an operator X is encountered, then;

a Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than X

b. Push the operator X to the stack

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 5: EXIT

Convert the following infix expression into postfix expression using the algorithm

$A - (B / C + (D \% E * F) / G) * H$

$(A - (B / C + (D \% E * F) / G) * H)$

Solution

Infix Character Scanned	Stack	Postfix Expression
	(
A	(A
-	(-	A
((- (A
B	(- (A B
/	(- (/	A B
C	(- (/	A B C
+	(- (+	A B C /
((- (+ (A B C /
D	(- (+ (A B C / D
%	(- (+ (%	A B C / D
E	(- (+ (%	A B C / D E
*	(- (+ (% *	A B C / D E
F	(- (+ (% *	A B C / D E F
)	(- (+	A B C / D E F * %
/	(- (+ /	A B C / D E F * %
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

Evaluation of a Postfix Expression

The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation. That is, given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.

Both these tasks converting the infix notation into postfix notation and evaluating the postfix expression make extensive use of stacks as the primary tool.

Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

Evaluation of a postfix expression

- Step 1: Add a ")" at the end of the postfix expression
- Step 2: Scan every character of the postfix expression and repeat Steps 3 and 4 until ")" is encountered
- Step 3: IF an operand is encountered, push it on the stack
IF an operator O is encountered, then
- Pop the top two elements from the stack as A and B as A and B
 - Evaluate B O A, where A is the topmost element and B is the element below A.
 - Push the result of evaluation on the stack
- [END OF IF]
- Step 4: SET RESULT equal to the topmost element of the stack
- Step 5: EXIT

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Let us now take an example that makes use of this algorithm. Consider the infix expression given as $9 - ((3 * 4) + 8) / 4$. Evaluate the expression.

The infix expression $9 - ((3 * 4) + 8) / 4$ can be written as $9\ 3\ 4\ *\ 8\ +\ 4\ /\ -$ using postfix notation. Look at Table 7.1, which shows the procedure.

Queues

A queue is an important data structure which is extensively used in computer applications. In this we will study the operations that can be performed on a queue. Will also discuss the implementation of a queue by using both arrays as well as linked lists, Will illustrate different types of queues like multiple queues, double ended queues, circular queues, and priority queues. And also lists some real-world applications of queues.

INTRODUCTION

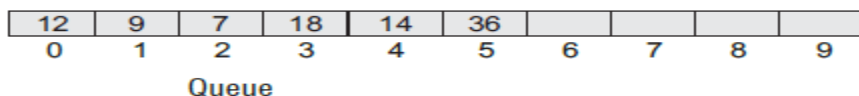
Let us explain the concept of queues using the analogies given below.

- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- Luggage kept on conveyor belts. The bag which was placed first will be the first to come out at the other end.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

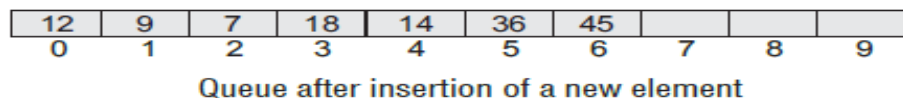
A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT. Queues can be implemented by using either arrays or linked lists.

ARRAY REPRESENTATION OF QUEUES

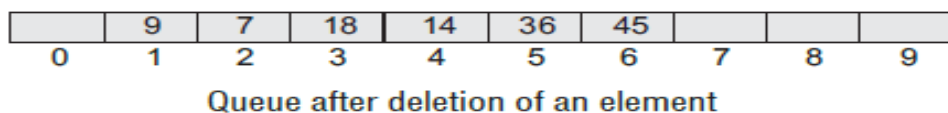
Queues can be easily represented using linear arrays. As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.



In above Fig $FRONT = 0$ and $REAR = 5$. Suppose we want to add another element with value 45, then $REAR$ would be incremented by 1 and the value would be stored at the position pointed by $REAR$.



Here, $FRONT = 0$ and $REAR = 6$. Every time a new element has to be added, we repeat the same procedure. If we want to delete an element from the queue, then the value of $FRONT$ will be incremented. Deletions are done from only this end of the queue. Here, $FRONT = 1$ and $REAR = 6$.



However, before inserting an element in a queue, we must check for overflow conditions. An overflow will occur when we try to insert an element into a queue that is already full. When $REAR = MAX - 1$, where MAX is the size of the queue, we have an overflow condition. Note that we have written $MAX - 1$ because the index starts from 0.

ALGORITHM TO INSERT AN ELEMENT IN A QUEUE

```
Step 1: If Rear = Max
        Write Overflow
        Goto Step 4
    [End of If]
Step 2: If Front = 0 and Rear = 0
        Set Front = Rear = 1
    Else
        Set Rear = Rear + 1
    [End of If]
Step 3: Set Queue [Rear] = Num
Step 4: Exit
```

The algorithm to insert an element in a queue, In Step 1, we first check for the overflow condition. In Step 2, we check if the queue is empty. In case the queue is empty, then both $FRONT$ and $REAR$ are set to zero, so that the new value can be stored at the 0th location. Otherwise, if the queue already has some values, then $REAR$ is incremented so that it points to the next location in the array. In Step 3, the value is stored in the queue at the location pointed by $REAR$.

Similarly, before deleting an element from a queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If $FRONT = -1$ and $REAR = -1$, it means there is no element in the queue.

ALGORITHM TO DELETE AN ELEMENT FROM A QUEUE

```
Steps:
1. If (FRONT = 0) then
    1. Print "Queue is empty"
    2. Exit
2. Else
    1. ITEM = Q[FRONT]           //Get the element
    2. If (FRONT = REAR)         //When queue contains single element
        1. REAR = 0             //The queue becomes empty
        2. FRONT = 0
    3. Else
        1. FRONT = FRONT + 1
    4. EndIf
3. EndIf
4. Stop
```

The algorithm to delete an element from a queue, In Step 1, we check for underflow condition. An underflow occurs if $FRONT = -1$ or $FRONT > REAR$. However, if queue has some values, then $FRONT$ is incremented so that it now points to the next value in the queue.

Note: The process of inserting an element in the queue is called en-queue, and the process of deleting an element from the queue is called de-queue.

TYPES OF QUEUES

A queue data structure can be classified into the following types:

1. Circular Queue
2. Deque
3. Priority Queue
4. Multiple Queues

Circular Queues

In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.

54	9	7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

Linear queue

Here, FRONT = 0 and REAR = 9.

Now, if you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted. Consider a scenario in which two successive deletions are made.

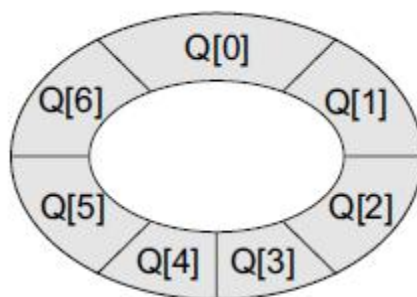
		7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

Queue after two successive deletions

Here, front = 2 and REAR = 9.

Even though there is space available, the overflow condition still exists because the condition $\text{rear} = \text{MAX} - 1$ still holds true. This is a major drawback of a linear queue.

To resolve this problem, we have two solutions. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large. The second option is to use a circular queue. In the circular queue, the first index comes right after the last index.



Circular queue

The circular queue will be full only when $\text{front} = 0$ and $\text{rear} = \text{Max} - 1$. A circular queue is implemented in the same manner as a linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations. For insertion, we now have to check for the following three conditions:

- If $\text{front} = 0$ and $\text{rear} = \text{MAX} - 1$, then the circular queue is full.

90	49	7	18	14	36	45	21	99	72
FRONT = 0	1	2	3	4	5	6	7	8	REAR = 9

Full queue

- If $\text{rear} \neq \text{MAX} - 1$, then rear will be incremented and the value will be inserted.

90	49	7	18	14	36	45	21	99	
FRONT = 0	1	2	3	4	5	6	7	REAR = 8	9

Increment rear so that it points to location 9 and insert the value here

- If $\text{front} \neq 0$ and $\text{rear} = \text{MAX} - 1$, then it means that the queue is not full. So, set $\text{rear} = 0$ and insert the new element there

Queue with vacant locations

		7	18	14	36	45	21	80	81
0	1	FRONT = 2	3	4	5	6	7	8	REAR = 9

Set REAR = 0 and insert the value here

ALGORITHM TO INSERT AN ELEMENT IN A CIRCULAR QUEUE

Algorithm Enqueue_CQ

Input: An element ITEM to be inserted into the circular queue.

Output: Circular queue with the ITEM at FRONT, if the queue is not full.

Data structures: CQ be the array to represent the circular queue. Two pointers FRONT and REAR are known.

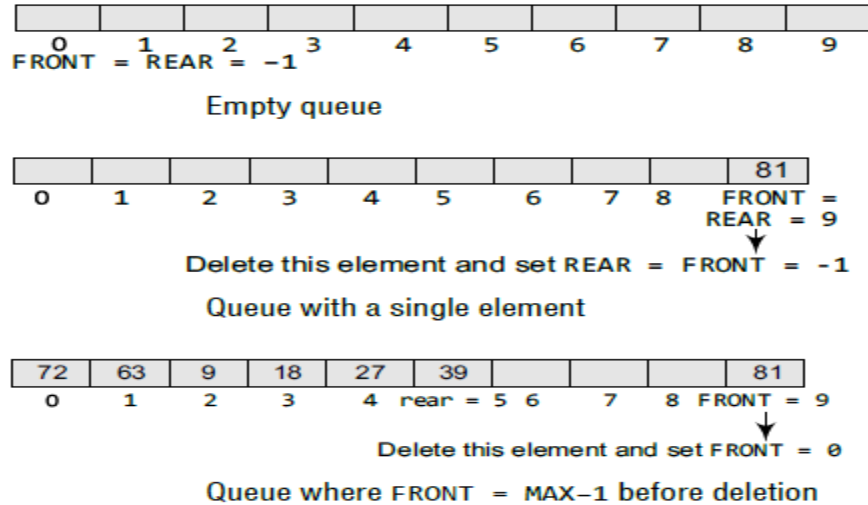
Steps:

1. **If** (FRONT = 0) **then** // When the queue is empty
2. FRONT = 1
3. REAR = 1
4. CQ[FRONT] = ITEM
5. **Else** // Queue is not empty
6. next = (REAR MOD LENGTH) + 1
7. **If** (next \neq FRONT) **then** // If the queue is not full
8. REAR = next
9. CQ[REAR] = ITEM
10. **Else**
11. **Print** "Queue is full"
12. **EndIf**
13. **EndIf**
14. **Stop**

The algorithm to insert an element in a circular queue, In Step 1, we check for the overflow condition. In Step 2, we make two checks. First to see if the queue is empty, and second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end. In Step 3, the value is stored in the queue at the location pointed by REAR.

After seeing how a new element is added in a circular queue, let us now discuss how deletions are performed in this case. To delete an element, again we check for three conditions.

- If $\text{front} = -1$, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and $\text{front} = \text{rear}$, then after deleting the element at the front the queue becomes empty and so front and rear are set to -1 .
- If the queue is not empty and $\text{front} = \text{MAX}-1$, then after deleting the element at the front, front is set to 0 .



ALGORITHM TO DELETE AN ELEMENT FROM A CIRCULAR QUEUE

Algorithm Dequeue_CQ

Input: A queue CQ with elements. Two pointers FRONT and REAR are known.

Output: The deleted element is ITEM if the queue is not empty.

Data structures: CQ is the array representation of circular queue.

Steps:

- ```

1. If (FRONT = 0) then
2. Print "Queue is empty"
3. Exit
4. Else
5. ITEM = CQ[FRONT]
6. If (FRONT = REAR) then // If the queue contains a single element
7. FRONT = 0
8. REAR = 0
9. Else
10. FRONT = (FRONT MOD LENGTH) + 1
11. EndIf
12. EndIf
13. Stop

```

Which shows the algorithm to delete an element from a circular queue, In Step 1, we check for the underflow condition. In Step 2, the value of the queue at the location pointed by FRONT is stored in VAL. In Step 3, we make two checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue. The value of FRONT is then updated based on the outcome of these checks.

## CIRCULAR QUEUES USING DYNAMICALLY ALLOCATED ARRAYS

- Shortcoming of static queue implementation: is the need to know at compile-time, a good bound(MAX\_QUEUE\_SIZE) on how large the queue will become.
- This short coming can be overcome by
  - using a dynamically allocated array for the elements &
  - then increasing the size of the array as needed
- In array-doubling, we double array-capacity whenever it becomes necessary to increase the capacity of an array (Figure3.8).

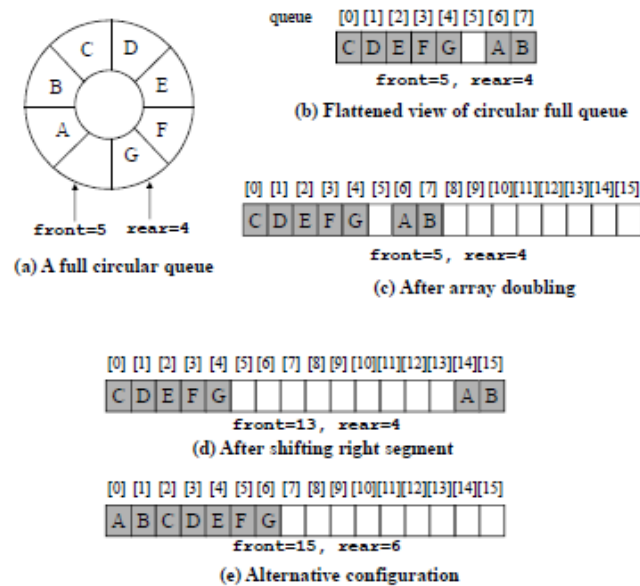


Figure3.8: Doubling queue capacity

- To get a proper circular queue configuration, we must slide the elements in the right segment to the right end of the array(Figure3.8d).
- The array doubling and the slide to the right together copy at most  $2 * \text{capacity} - 2$  elements.
- The number of elements copied can be limited to capacity-1 by customizing the array doubling codes so as to obtain the configuration of Figure3.8e. This configuration may be obtained as follows:

- 1) Create a new array newQueue of twice the capacity.
- 2) Copy the second segment to positions in newQueue beginning at 0.
- 3) Copy the first segment to positions in newQueue beginning at capacity-front-1.

```
void addq(element item)
{
 /* add an item to the queue */
 rear = (rear + 1) % capacity;
 if (front == rear)
 queueFull();
 queue[rear] = item;
}
```

Program3.9: Add to a circular queue



```

Void queueFull()
{
 /* allocate an array with twice the capacity */
 element * newQueue;
 MALLOC(newQueue, 2*capacity*sizeof(*queue));

 /* copy from queue to newQueue*/
 int start = (front + 1) % capacity;
 if(start < 2)
 copy(queue+start, queue+start+capacity-1, newQueue);
 else
 {
 copy(queue+start, queue+capacity, newQueue);
 copy(queue, queue+rear+1, newQueue+capacity-start);
 }
 /* Switch to newQueue */

 front = 2*capacity-1;
 rear = capacity-2;
 capacity = capacity * 2;
 free(queue);
 queue = newQueue;
}

```

Program 3.10: Doubling queue capacity

## Multiple Queues

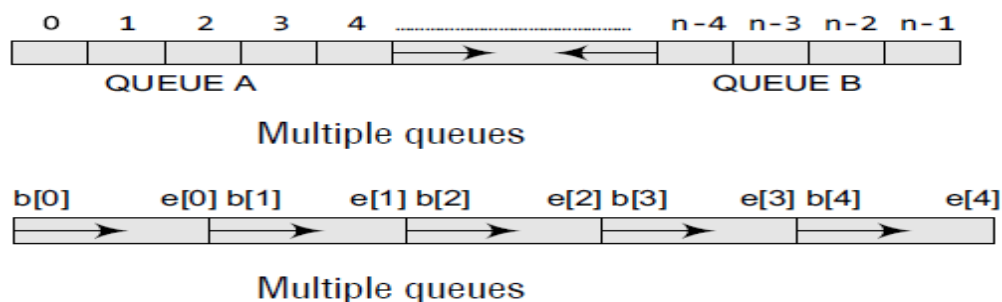
When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.

In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.

So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size. Figure 8.31 illustrates this concept.

In the figure, an array Queue[n] is used to represent two queues, Queue A and Queue B. The value of n is such that the combined size of both the queues will never exceed n. While operating on these queues, it is important to note one thing queue A will grow from left to right, whereas queue B will grow from right to left at the same time.

Extending the concept to multiple queues, a queue can also be used to represent n number of queues in the same array. That is, if we have a QUEUE[n], then each queue I will be allocated an equal amount of space bounded by indices b[i] and e[i].



## APPLICATIONS OF QUEUES

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.

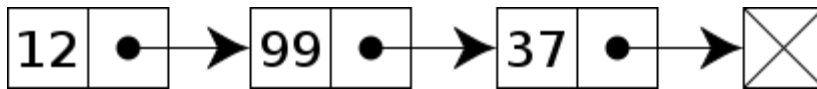


- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.

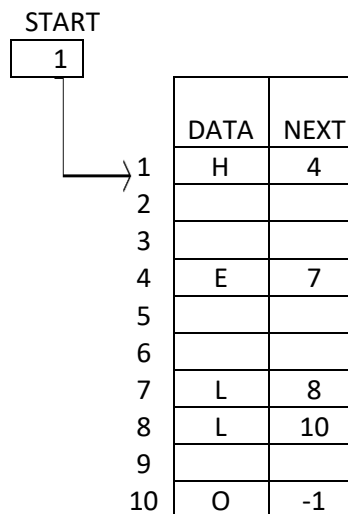
## LINKED LISTS

A linked list is ordered collection of finite. Homogeneous data elements called nodes where the linear order is maintained by means of links or pointers.

A **linked list** is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a link) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.



### Memory Representation of Linked Lists



Let us see how a linked list is maintained in the memory, in order to form a linked list, we need a structure called node which has two fields, DATA and NEXT. Data will store the information part and next will store the address of the node in sequence. We see that the variable START is used to store the address of the first node.

## Representing Chains in C

### Types of Linked Lists

- Single linked list
- Double linked list
- Circular linked list

### Representation of a Linked List

**There** are two ways to represent a linked list in memory:

1. Static representation using array
2. Dynamic representation using free pool of storage

Linked lists are a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. Specifically, the programmer writes a struct definition that contains variables holding information about something and that has a pointer to a struct of its same type (it has to be a pointer--otherwise, every time an element was created, it would create a new element, infinitely). Each of these individual structs or classes in the list is commonly known as a node or element of the list. I.linkedLists

In memory a linked list is often described as looking like this:

```

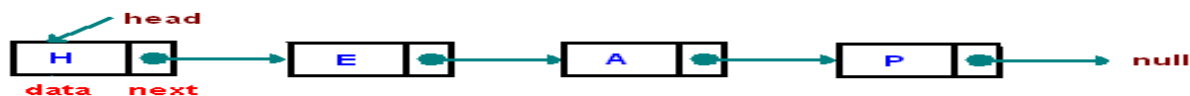
- Data - - Data -

- Pointer- - -> - Pointer-

```

### Head node in linked list

The entry point into a **linked list** is called the **head** of the **list**. It should be noted that **head** is not a separate **node**, but the reference to the first **node**. If the **list** is empty then the **head** is a null reference. **Linked list** is a dynamic data structure.



### Node is represented as

Struct node

```
{
int data;
struct node *next;
}
```

### Linked List Operations

- **Display/ Traverse**– Displays the complete list.
- **Search** – Searches an element using the given key.
- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.

### Traversing/Display

Start with the head and access each node until you reach null. Do not change the head reference

### Algorithm for Traversing a Linked List

#### Algorithm TRAVERSE\_SL (HEADER)

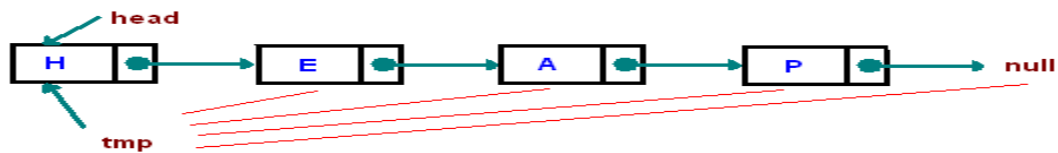
Input: HEADER is the pointer to the header node.

Output: According to the PROCESS( )

Data structures: A single linked list whose address of the starting node is known from HEADER.

#### Steps:

- |                          |                                                 |
|--------------------------|-------------------------------------------------|
| 1. ptr = HEADER.LINK     | //ptr is to store the pointer to a current node |
| 2. While (ptr ≠ NULL) do | //Continue till the last node                   |
| 1. PROCESS(ptr)          | //Perform PROCESS( ) on the current node        |
| 2. ptr = ptr.LINK        | //Move to the next node                         |
| 3. EndWhile              |                                                 |
| 4. Stop                  |                                                 |



### Algorithm to Search a Linked List

Step 1: [Initialize] Set Ptr = Start

Step 2: Repeat Step 3 While Ptr! = Null

Step 3: If Val = Ptr->Data

Set Pos = Ptr

Go To Step 5

Else

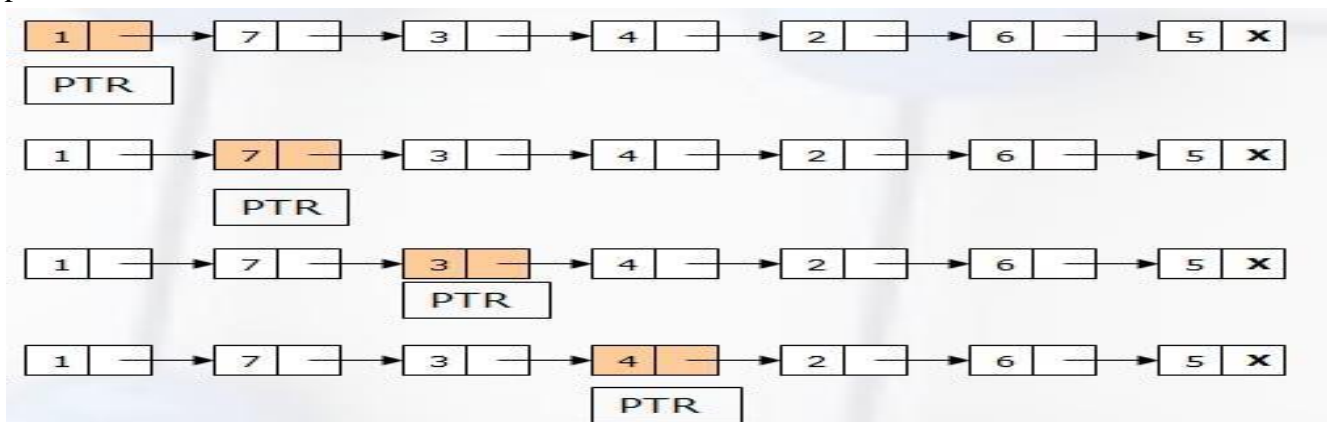
Set Ptr = Ptr->Next

[End of If]

[End of Loop]

Step 4: Set Pos = Null

Step 5: Exit



### Inserting a Node at the Beginning

Suppose we want to add a new node with the data 9 and add it as the first node of the list. Then the changes will do in the linked list. Allocate the memory for the new node and initialize its DATA part to 9. Add the new node as the first node of the list by making the NEXT part of the new node contain the address of the START. Now make START to point to the first node of the list.

**Algorithm for GETNODE(NODE) to get a pointer of a memory block which suits the type NODE:**

---

**Procedure GETNODE(NODE)**

1. If (AVAIL = NULL) //AVAIL is the pointer to the pool of free storage
  1. Return (NULL)
2. Print "Insufficient memory: Unable to allocate memory"
2. Else //Sufficient memory is available
  1. ptr = AVAIL
  2. While (SIZEOF(ptr)  $\neq$  SIZEOF(NODE)) and (ptr  $\neq$  NULL)
    1. ptr1 = ptr //Till the desired block is not found or search reaches
    2. ptr = ptr.LINK //at the end of the pool
  3. EndWhile //To keep the track of the previous block
  4. If (SIZEOF(ptr) = SIZEOF(NODE)) //Memory block of right size is found
    1. ptr1.LINK = ptr.LINK //Update the AVAIL List
    2. Return(ptr)
  5. Else
    1. Print "The memory block is too large to fit"
    2. Return(NULL)
  6. EndIf
3. EndIf
4. Stop

### Algorithm to Insert a New Node in the Beginning

**Insertion of a node into a single linked list at the front.** The algorithm INSERT\_SL\_FRONT is to insert a node into a single linked list at the front of the list.

**Algorithm INSERT\_SL\_FRONT(HEADER, X)**

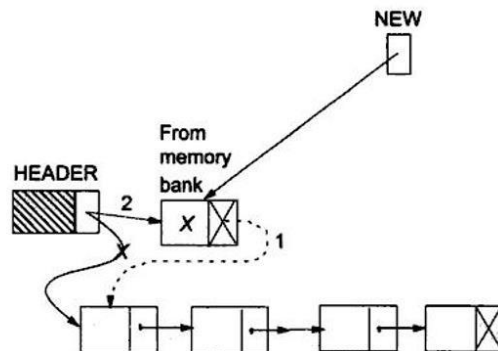
Input: HEADER is the pointer to the header node and X is the data of the node to be inserted.

Output: A single linked list with newly inserted node in the front of the list.

Data structures: A single linked list whose address of the starting node is known from HEADER.

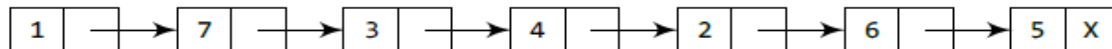
**Steps:**

1. new = GETNODE(NODE) //Get a memory block of type NODE and store //its pointer in new
2. If (new = NULL) then //Memory manager returns NULL on searching //the memory bank
  1. Print "Memory underflow: No insertion"
  2. Exit //Quit the program
3. Else //Memory is available and get a node from //memory bank
  1. new.LINK = HEADER.LINK //Change of pointer 1 as shown in Figure 3.5(a)
  2. new.DATA = X //Copy the data X to newly availed node
  3. HEADER.LINK = new //Change of pointer 2 as shown in Figure 3.5(a)
4. EndIf
5. Stop



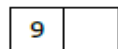
**Fig. 3.5(a)** Insertion of a node at the front of a single linked list.

Example:

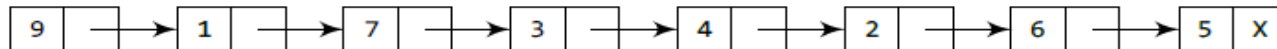


START

Allocate memory for the new node and initialize its DATA part to 9.

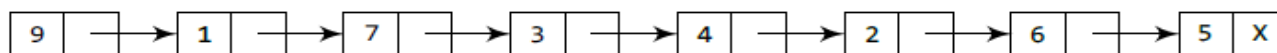


Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.



START

Now make START to point to the first node of the list.



START

## Inserting a Node at the End

If we want to add a node with data 9 as the last node of the list, Then insert a new node at the end of a linked list. Take pointer variable initialize with START. That is the pointer now points to the first node of the linked list. With the help of loop will traverse through the linked list to reach the last node. Once reached the last node the next of the last node to store the address of the new node, then last node next will contains NULL.

### Algorithm to Insert a New Node at the End of the Linked List

#### Algorithm INSERT\_SL\_END (HEADER, X)

Input: HEADER is the pointer to the header node and X is the data of the node to be inserted.

Output: A single linked list with newly inserted node having data X at end.

Data structures: A single linked list whose address of the starting node is known from HEADER.

#### Steps:

1. new = GETNODE(NODE) //Get a memory block of type NODE and store //its pointer in new
2. If (new = NULL) then //Unable to allocate memory for a node
  1. Print "Memory is insufficient: Insertion is not possible"
  2. Exit //Quit the program
3. Else //Move to the end of the given list and then insert
  1. ptr = HEADER //Start from the HEADER node
  2. While (ptr.LINK ≠ NULL) do //Move to the end
    1. ptr = ptr.LINK //Change pointer to the next node
  3. EndWhile
  4. ptr.LINK = new //Change the link field of last node: Pointer 1 as //in Figure 3.5(b)
5. new.DATA = X //Copy the content X into new node
4. EndIf
5. Stop

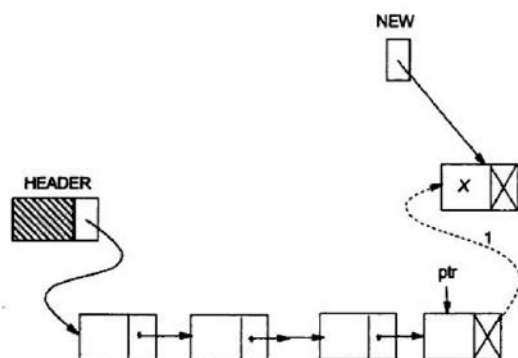
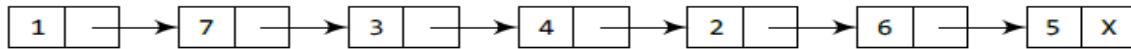


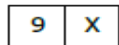
Fig. 3.5(b) Insertion at the end of a single linked list.

Example:

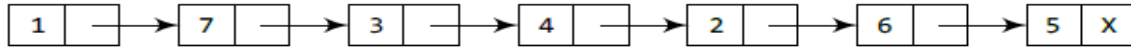


START

Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.

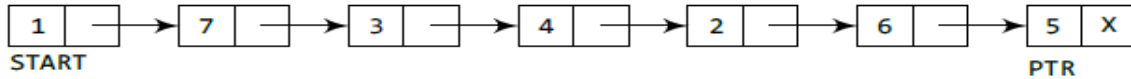


Take a pointer variable PTR which points to START.

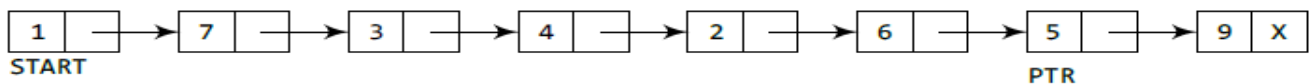


START, PTR

Move PTR so that it points to the last node of the list.



Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.



## INSERTING A NODE AT ANY POSITION:

### Algorithm INSERT\_SL\_ANY(HEADER, X, KEY)

Input: HEADER is the pointer to the header node, X is the data of the node to be inserted, and KEY being the data of the key node after which the node has to be inserted.

Output: A single linked list enriched with newly inserted node having data X after the node with data KEY.

Data structures: A single linked list whose address of the starting node is known from HEADER.

#### Steps:

1. new = GETNODE(NODE) //Get a memory block of type NODE and store  
//its pointer in new
2. If (new = NULL) then //Unable to allocate memory for a node
  1. Print "Memory is insufficient: Insertion is not possible"
  2. Exit //Quit the program
3. Else
  1. ptr = HEADER //Start from the HEADER node
  2. While (ptr.DATA ≠ KEY) and (ptr.LINK ≠ NULL) do //Move to the node having data as KEY or at  
//the end if KEY is not in the list
    1. ptr = ptr.LINK
  3. EndWhile
  4. If (ptr.LINK = NULL) then //Search fails to find the KEY
    1. Print "KEY is not available in the list"
    2. Exit
  5. Else
    1. new.LINK = ptr.LINK //Change the pointer 1 as shown in Figure 3.5(c)
    2. new.DATA = X //Copy the content into the new node
    3. ptr.LINK = new //Change the pointer 2 as shown in Figure 3.5(c)
  6. EndIf
4. EndIf
5. Stop



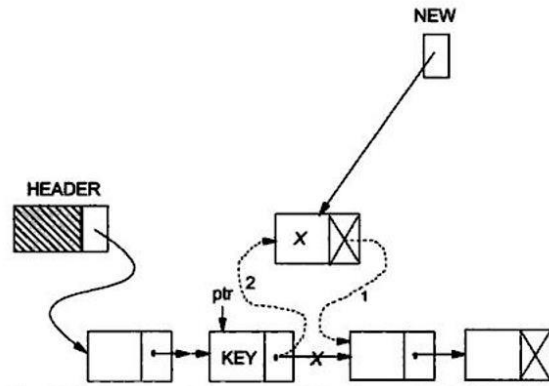


Fig. 3.5(c) Insert at any position in a single linked list.

## Deletions from List:

Like Insertions, there are also various cases of deletions

- (i) Deletion at front of the list
- (ii) Deletion at end of the list
- (iii) Deletion at any position in the list

### Return Node (PTR):

RETURNNODE (PTR) returns a node having pointer PTR to the free pool of storage.

### Algorithm for Return Node (PTR):

|                                  |                                               |
|----------------------------------|-----------------------------------------------|
| <b>Procedure RETURNNODE(PTR)</b> | //PTR is the pointer of a node to be returned |
| 1. ptr1 = AVAIL                  | //Start from the beginning of the free pool   |
| 2. While (ptr1.LINK ≠ NULL) do   |                                               |
| 1. ptr1 = ptr1.LINK              |                                               |
| 3. EndWhile                      |                                               |
| 4. ptr1.LINK = PTR               | //Insert the node at the end                  |
| 5. PTR.LINK = NULL               | //Node inserted is the last node              |
| 6. Stop                          |                                               |

## Deleting the First Node

### Algorithm to Delete the First Node from the Linked List

#### Algorithm DELETE\_SL\_FRONT(HEADER)

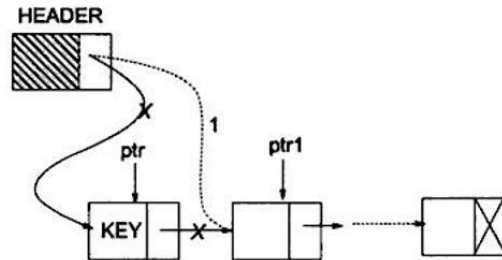
Input: HEADER is the pointer to the header node.

Output: A single linked list eliminating the node at the front.

Data structures: A single linked list whose address of the starting node is known from HEADER.

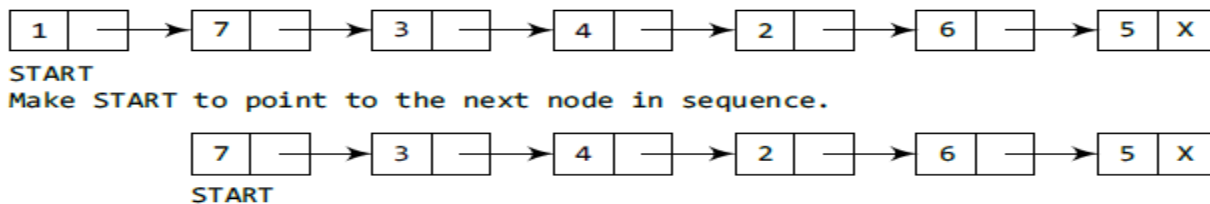
**Steps:**

1. ptr = HEADER.LINK //Pointer to the first node
2. If (ptr = NULL) //If the list is empty
  1. Print "The list is empty: No deletion"
  2. Exit //Quit the program
3. Else //The list is not empty
  1. ptr1 = ptr.LINK //ptr1 is the pointer to the second node, if any
  2. HEADER.LINK = ptr1 //Next node becomes the first node as in Figure 3.6(a)
  3. RETURNNODE(ptr) //Deleted node is freed to the memory bank for future use
4. EndIf
5. Stop



**Fig. 3.6(a)** Deletion of a node from a single linked list at the front.

Ex:



**ALGORITHM TO DELETE THE LAST NODE:**

**Algorithm DELETE\_SL\_END(HEADER)**

Input: HEADER is the pointer to the header node.

Output: A single linked list eliminating the node at the end.

Data structures: A single linked list whose address of the starting node is known from HEADER.

**Steps:**

1. ptr = HEADER //Move from the header node
2. If (ptr.LINK = NULL) then
  1. Print "The list is empty: No deletion possible"
  2. Exit //Quit the program
3. Else
  1. While (ptr.LINK ≠ NULL) do //Go to the last node
    1. ptr1 = ptr //To store the previous pointer
    2. ptr = ptr.LINK //Move to the next
  2. EndWhile
  3. ptr1.LINK = NULL //Last but one node become the last node as in Figure 3.6(b)
  4. RETURNNODE(ptr) //Deleted node is returned to the memory bank for future use
4. EndIf
5. Stop



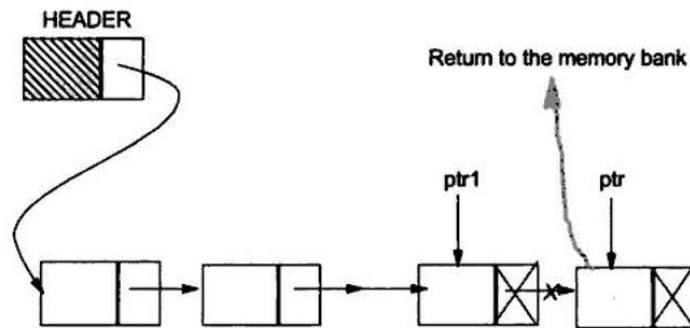
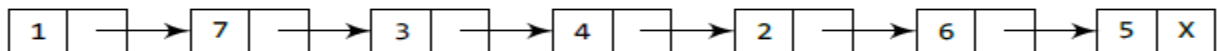


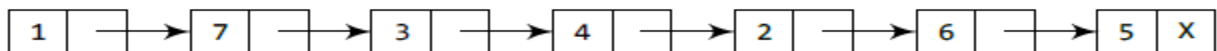
Fig. 3.6(b) Deletion of a node from a single linked list at the end.

Ex:



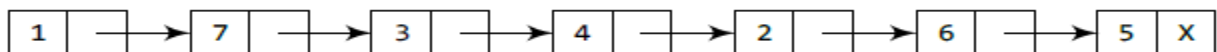
START

Take pointer variables PTR and PREPTR which initially point to START.



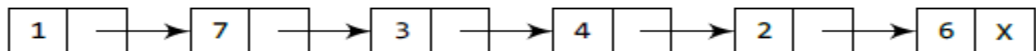
START  
PREPTR  
PTR

Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.



START

Set the NEXT part of PREPTR node to NULL.



START

## DELETION AT ANY POSITION IN THE LIST ALGORITHM:

### Algorithm DELETE\_SL\_ANY

Input: HEADER is the pointer to the header node, KEY is the data content of the node to be deleted.

Output: A single linked list except the node with data content as KEY.

Data structures: A single linked list whose address of the starting node is known from HEADER.

#### Steps:

1. ptr1 = HEADER //Start from the header node
2. ptr = ptr1.LINK //This points to the first node, if any
3. While (ptr ≠ NULL) do
  1. If (ptr.DATA ≠ KEY) then //If not found the key
    1. ptr1 = ptr //Keep a track of the pointer of the previous node
    2. ptr = ptr.LINK //Shift to the next
  2. Else //The node is found
    1. ptr1.LINK = ptr.LINK //Link field of the predecessor is to point //the successor of node under deletion, see Figure 3.6(c)
    2. RETURNNODE(ptr) //Return the deleted node to the memory bank
    3. Exit //Exit the program
3. EndIf
4. EndWhile
5. If (ptr = NULL) then //When the desired node is not available in the list
  1. Print "Node with KEY does not exist: No deletion"
6. EndIf
7. Stop

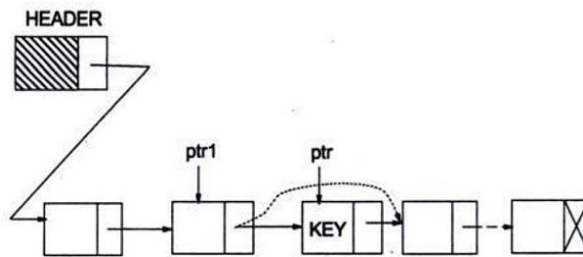
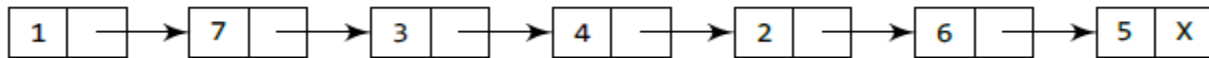


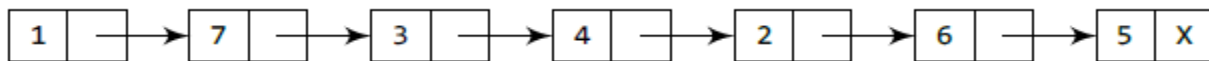
Fig. 3.6(c) Deletion of a node at any position in a single linked list.

Ex:



START

Take pointer variables PTR and PREPTR which initially point to START.

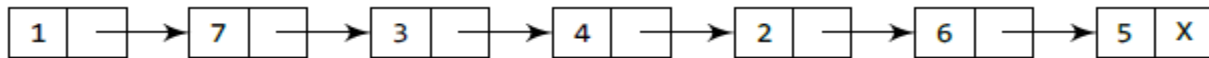


START

PREPTR

PTR

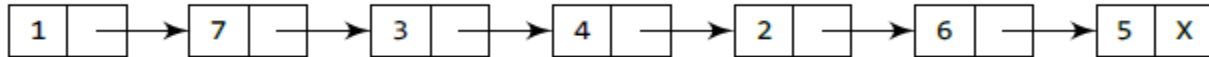
Move PREPTR and PTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node.



START

PREPTR

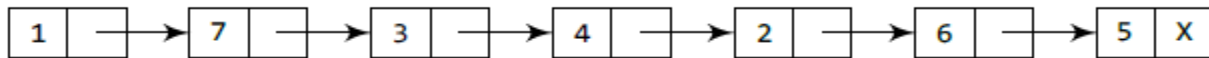
PTR



START

PREPTR

PTR

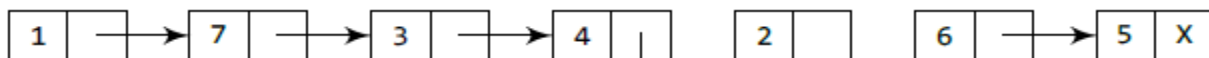


START

PREPTR

PTR

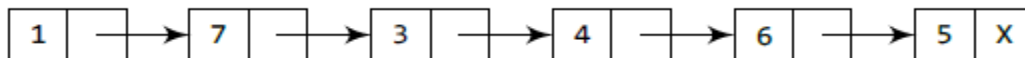
Set the NEXT part of PREPTR to the NEXT part of PTR.



START

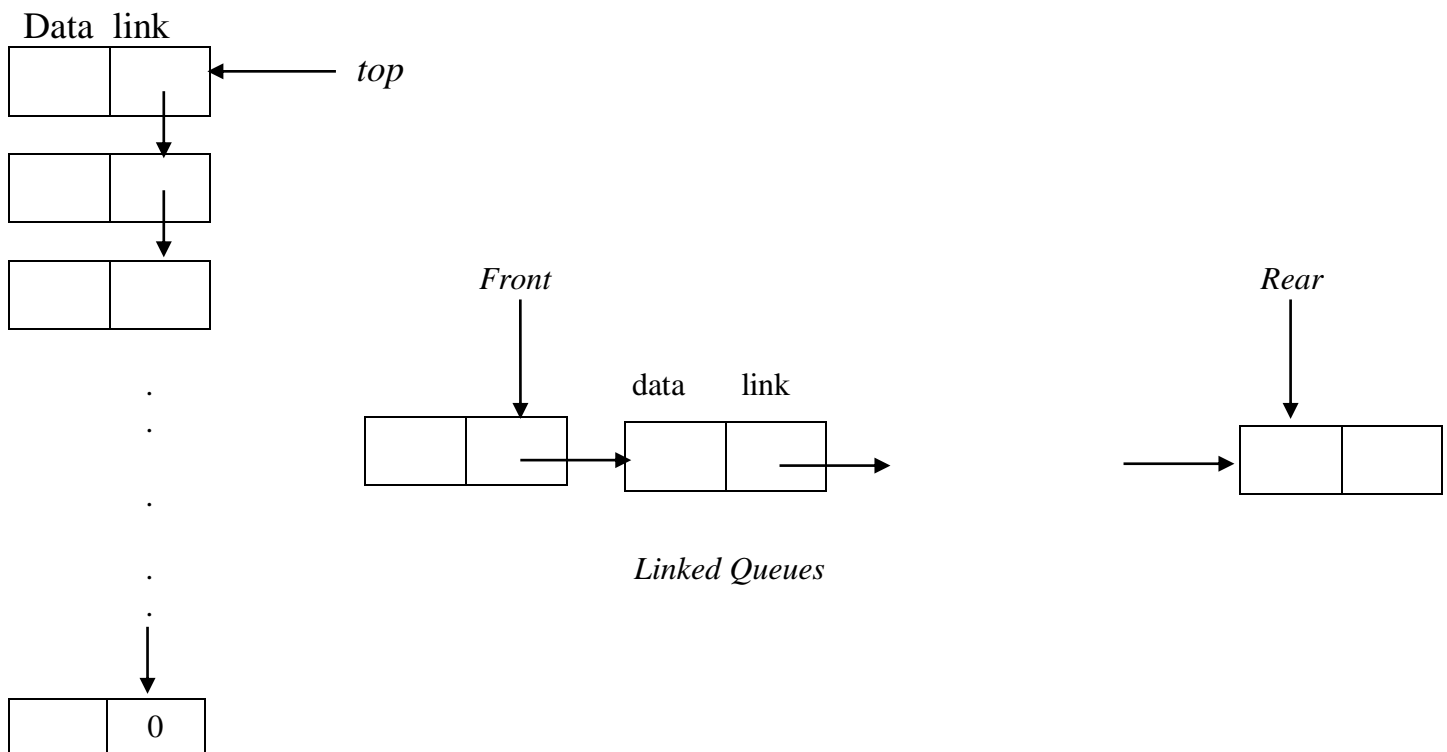
PREPTR

PTR



START

## Linked Stacks and Queues:



### Linked Stacks

Directions of links for both the stack and queue are such as to facilitate easy insertion and deletion of nodes. In the case of figure (a), one can easily add a node at the top or delete one from the top. In figure (b), one can easily add a node at the rear and both addition and deletion can be performed at the front, though for a queue we normally would not wish to add nodes at the front.

We assume initial condition for the stack is:

$\text{Top}[i] = \text{NULL}, 0 \leq i < \text{MAX\_STACKS}$

And boundary condition is:

$\text{Top}[i] = \text{NULL}$  iff the  $i^{\text{th}}$  stack is empty

Functions *push* and *pop*, add and delete items to/from a stack. The code for each is straight forward. Function *push* creates a new node, *temp*, and places item in the data field and *top* in the link field. The variable *top* is then changed to point to *temp*. A typical function call to add an element to the  $i^{\text{th}}$  stack would be *push* (*i,item*). Function of returns the top element and changes *top* to point to the address contained in its link field. The removed node is then returned to system memory. A typical function call to delete an element from the  $i^{\text{th}}$  stack would be *item=pop*(*i*);

```
void push(int i, element item)
{
 stackPointer temp;
 MALLOC(temp, sizeof(*temp));
 temp->data = item;
 temp->link = top[i];
 top[i] = temp;
}
```

---

Add to a linked list

```
Element pop(int i)
```

```
{
 stackPointer temp = top[i];
 element item;
 if(!item)
 return stackEmpty();
 item = temp → data;
 top[i] = temp → link;
 free(temp);
 return item;
}
```

Delete from a linked stack

To represent  $m \leq \text{MAX\_QUEUES}$  queues simultaneously, we begin with the declarations:

We assume that the initial condition for the queue is:

Front[i]=NULL,  $0 \leq i < \text{MAX\_QUEUES}$

And the boundary is:

Front[i]=NULL, iff the ith queue is empty

Functions *addq* and *deleteq* implement the add and delete operations for multiple queues. Function *addq* is more complex than *push* because we must check for an empty queue. If the queue is empty, we change *front* to point to the new node; otherwise we change rear's link field to point to the new node. In either case, we then change rear to point to the new node. Function *deleteq* is similar to *pop* since we are removing a node that is currently at the start of the list. Typical function call would be *addq(i,item)* and *item=delete(i)*;

```
void addq(i,item)
```

```
{
 queuePointer temp;
 MALLOC(temp,sizeof(*temp));
 temp → data = item;
 temp → link = NULL;
 if(front[i])
 rear[i] → link = temp;
 else
 front[i] = temp;
 rear[i] = temp;
}
```

## Additional List Operations

### Operations for Chains

Inverting a chain is another useful operation.

```
typedef struct list Node *listPointer;
```

```
typedef struct
```

```
{
 Char data;
 listPointer link;
}listNode;
```

For a list of  $\text{length} \geq 1$  nodes, the while loop is executed length times and so the computing time is linear or  $O(\text{length})$ .

```
listPointer invert(listPointer lead)
```

```
{
```

```

listPointer middle, trail;
middle=NULL;
while(lead)
{
 trail=middle;
 middle=lead;
 lead=lead → link;
 middle → link=trail;
}
return middle;
}

```

Inverting a list

### Merging two single linked lists into one list:

Two single linked lists, namely L1 and L2 are available and we want to merge the list L2 after L1. Also assume that, HEADER1 and HEADER2 are the header nodes of the lists L1 and L2, respectively. Merging can be done by setting the pointer of the link field of the last node in the list L1 with the pointer of the first node in L2. The header node in the list L2 should be returned to the pool of free storage. Merging two single linked lists into one list is illustrated in Figure.

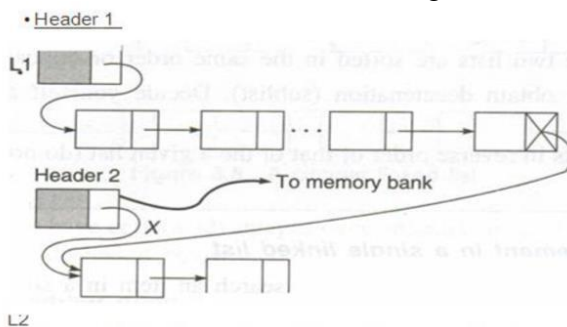


Figure 3.7 Merging two single linked lists into one single linked list.

#### Algorithm MERGE\_SL(HEADER1, HEADER2; HEADER)

Input: HEADER1 and HEADER2 are pointers to header nodes of lists (L1 and L2, respectively) to be merged.

Output: HEADER is the pointer to the resultant list.

Data structures: Single linked list structure.

#### Steps:

1. ptr = HEADER1
2. While (ptr.LINK ≠ NULL) do //Move to the last node in the list L1
  1. ptr = ptr.LINK
3. EndWhile
4. ptr.LINK = HEADER2.LINK //Last node in L1 points to the first node in L2
5. RETURNNODE(HEADER2) //Return the header node to the memory bank
6. HEADER = HEADER1 //HEADER becomes the header node of the merged list
7. Stop

Another function is one that concatenates two chains, ptr1 and ptr2. The complexity of this function is  $O(\text{length of list ptr1})$ .

```

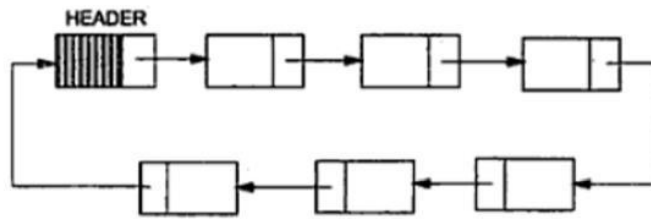
ListPointer concatenate(listPointer ptr1, listPointer ptr2)
{
 ListPointer temp;
 If(!ptr1) return ptr2;
 If(!ptr2) return ptr1;
 For(temp=ptr1; temp → link; temp=temp → link)
 temp → link=ptr2;
}

```

Concatenating singly linked lists

### Operations for Circular Lined Lists:

In a single linked list, the link field of the last node is null (hereafter a single linked list may be read as ordinary linked list), but a number of advantages can be gained if we utilize this link field to store the pointer of the header node. A linked list where the last node points the header node is called the circular linked list. Figure shows a pictorial representation of a circular linked list.



**Fig. 3.8** A circular linked list.

By keeping a pointer last to the last node in the list rather than to the first, we are able to insert an element at both the front and end with ease. Had we kept a pointer to the first node instead of the last node, inserting at the front would require us to must move down the entire length of the list until we find the last node so that we can change the pointer in the last node to point to the new first node.

```
Void insertFront(listPointer *last,listPointer node)
{
if(!(*last))
{
*last=node;
node → link=node;
}
else
{
Node → link=(*last) → link;
(*last) → link=node;
}
}
```

Inserting at the front of a list

```
Int length(listPointer last)
{
listPointer temp;
int count=0;
if(last)
{
Temp=last;
Do
{
Count++;
Temp=temp → link;
}while(temp!=last);
}
Return count;
}
```

finding the length of a circular list

### **Doubly(e) Linked List:**

In a single linked list one can move beginning from the header node to any node in one direction only (from left to right). This is why a single linked list is also termed a one-way list. On the other hand, a double linked list is a two-way list because one can move in either direction, either from left to right or from right to left. This is accomplished by maintaining two link fields instead of one as in a single linked list. A structure of a node for a double linked list is represented as in Figure.

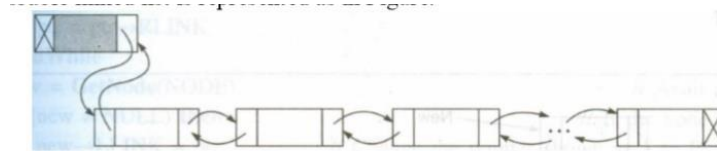


Figure Structure of a node and a double linked list.

From the figure, it can be noticed that two links, viz. RLINK and LLINK, point to the nodes on the right side and left side of the node, respectively. Thus, every node, except the header node and the last node, points to its immediate predecessor and immediate successor.

### **Operations on a Double Linked List**

In this section, only the insertion and deletion operations are discussed.

#### **Inserting a node into a double linked list**

Figure shows a schematic representation of various cases of inserting a node into a double linked list. Let us consider the algorithms of various cases of insertion.

#### **Inserting a node in the front**

The algorithm Insertliront DL: is used to define the insertion operation in a double linked list.

#### **Algorithm INSERT\_DL\_FRONT(X)**

Input: X the data content of the node to be inserted.

Output: A double linked list enriched with a node containing data as X at the front.

Data structure: Double linked list structure whose pointer to the header node is HEADER.

#### **Steps:**

1. ptr = HEADER.RLINK //Points to the first node
2. new = GETNODE(NODE) //Avail a new node from the memory bank
3. If (new ≠ NULL) then //If new node is available
  1. new.LLINK = HEADER //Newly inserted node points the header as 1 in Figure //3.11(a)
  2. HEADER.RLINK = new //Header now points to the new node as 2 in Figure //3.11(a)
  3. new.RLINK = ptr //See the change in pointer shown as 3 in Figure 3.11(a)
  4. ptr.LLINK = new //See the change in pointer shown as 4 in Figure 3.11(a)
  5. new.DATA = X //Copy the data into newly inserted node
4. Else
  1. Print "Unable to allocate memory: Insertion is not possible"
5. EndIf
6. Stop



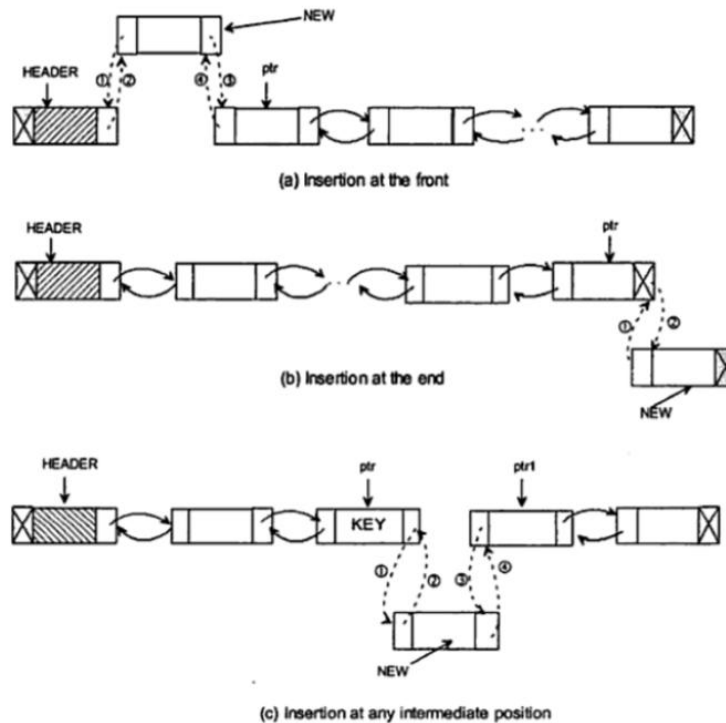


Fig. 3.11 Insertion of a node at various positions in a double linked list.

### Inserting a node at the end

The algorithm InsertEnd\_DL is to insert a node at the end into a double linked list.

#### Algorithm INSERT\_DL\_END(X)

Input:  $X$  the data content of the node to be inserted.

Output: A double linked list enriched with a node containing data as  $X$  at the end of the list.

Data structure: Double linked list structure whose pointer to the header node is HEADER.

#### Steps:

1.  $ptr = \text{HEADER}$
2. While ( $ptr.\text{RLINK} \neq \text{NULL}$ ) do      //Move to the last node
  1.  $ptr = ptr.\text{RLINK}$
3. EndWhile
4.  $new = \text{GETNODE}(\text{NODE})$       //Avail a new node
5. If ( $new \neq \text{NULL}$ ) then      //If the node is available
  1.  $new.\text{LLINK} = ptr$       //Change the pointer shown as 1 in Figure 3.11(b)
  2.  $ptr.\text{RLINK} = new$       //Change the pointer shown as 2 in Figure 3.11(b)
  3.  $new.\text{RLINK} = \text{NULL}$       //Make the new node as the last node
  4.  $new.\text{DATA} = X$       //Copy the data into the new node
6. Else
1. print "Unable to allocate memory: Insertion is not possible"
7. EndIf
8. Stop

### Inserting a node at any position in the list :

The algorithm InsertAny\_DL is used to insert a node at any position into a double linked list.



**Algorithm INSERT\_DL\_ANY(X, KEY)**

Input:  $X$  be the data content of the node to be inserted, and  $KEY$  the data content of the node after which the new node to be inserted.

Output: A double linked list enriched with a node containing data as  $X$  after the node with data  $KEY$ , if any.

Data structure: Double linked list structure whose pointer to the header node is  $HEADER$ .

**Steps:**

1.  $ptr = HEADER$
2. While ( $ptr.DATA \neq KEY$ ) and ( $ptr.RLINK \neq NULL$ ) do // Move to the key node if the current node is not the  $KEY$  node or list reaches at the end
  1.  $ptr = ptr.RLINK$
3. EndWhile
4.  $new = GETNODE(NODE)$  //Get a new node from the pool of free storage
5. If ( $new = NULL$ ) then //When the memory is not available
  1. Print "Memory is not available"
  2. Exit //Quit the program
6. EndIf
7. If ( $ptr.RLINK = NULL$ ) //If the  $KEY$  node is at the end or not found in the list
  1.  $new.LLINK = ptr$
  2.  $ptr.RLINK = new$  //Insert at the end
  3.  $new.RLINK = NULL$
  4.  $new.DATA = X$  //Copy the information to the newly inserted node
8. Else //The  $KEY$  is available
  1.  $ptr1 = ptr.RLINK$  //Next node after the key node
  2.  $new.LLINK = ptr$  //Change the pointer shown as 2 in Figure 3.11(c)
  3.  $new.RLINK = ptr1$  //Change the pointer shown as 4 in Figure 3.11(c)
  4.  $ptr.RLINK = new$  //Change the pointer shown as 1 in Figure 3.11(c)
  5.  $ptr1.LLINK = new$  //Change the pointer shown as 3 in Figure 3.11(c)
  6.  $ptr = new$  //This becomes the current node
  7.  $new.DATA = X$  //Copy the content to the newly inserted node
9. EndIf
10. Stop

**Deleting a node from a double linked list**

Deleting a node from a double linked list may take place from any position in the list, as shown in Figure. Let us consider each of those cases separately. Deleting a node from the front of a double linked list

**Algorithm DELETE\_DL\_FRONT( )**

Input: A double linked list with data.

Output: A reduced double linked list.

Data structure: Double linked list structure whose pointer to the header node is  $HEADER$ .

**Steps:**

1.  $ptr = HEADER.RLINK$  //Pointer to the first node
2. If ( $ptr = NULL$ ) then //If the list is empty
  1. Print "List is empty: No deletion is made"
  2. Exit
3. Else
  1.  $ptr1 = ptr.RLINK$  //Pointer to the second node
  2.  $HEADER.RLINK = ptr1$  //Change the pointer shown as 1 in Figure 3.12(a)
  3. If ( $ptr1 \neq NULL$ ) //If the list contains a node after the first node of deletion
    1.  $ptr1.LLINK = HEADER$  //Change the pointer shown as 2 in Figure 3.12(a)
4. EndIf
5.  $RETURNNODE(ptr)$  //Return the deleted node to the memory bank
6. EndIf
7. Stop

### Deleting a node at the end of a double linked list

#### **Algorithm DELETE\_DL\_END( )**

Input: A double linked list with data.

Output: A reduced double linked list.

Data structure: Double linked list structure whose pointer to the header node is HEADER.

#### **Steps:**

1. ptr = HEADER
2. While (ptr.RLINK  $\neq$  NULL) do //Move to the last node
  1. ptr = ptr.RLINK
3. EndWhile
4. If (ptr = HEADER) then //If the list is empty
  1. Print "List is empty: No deletion is made"
  2. Exit //Quit the program
5. Else
  1. ptr1 = ptr.LLINK //Pointer to the last but one node
  2. ptr1.RLINK = NULL //Change the pointer shown as 1 in Figure 3.12(b)
  3. RETURNNODE(ptr) //Return the node to the memory bank
6. EndIf
7. Stop

### Deleting a node from any position in a double linked list

#### **Algorithm DELETE\_DL\_ANY(KEY)**

Input: A double linked list with data, and KEY, the data content of the key node to be deleted.

Output: A double linked list without a node having data content KEY, if any.

Data structure: Double linked list structure whose pointer to the header node is HEADER.

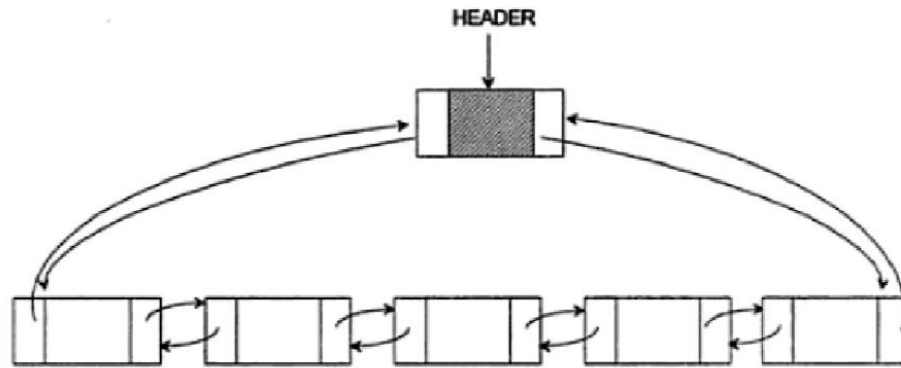
#### **Steps:**

1. ptr = HEADER.RLINK //Move to the first node
2. If (ptr = NULL) then
  1. Print "List is empty: No deletion is made"
  2. Exit
3. EndIf //Quit the program
4. While (ptr.DATA  $\neq$  KEY) and (ptr.RLINK  $\neq$  NULL) do //Move to the desired node
  1. ptr = ptr.RLINK
5. EndWhile
6. If (ptr.DATA = KEY) then //If the node is found
  1. ptr1 = ptr.LLINK //Track to the predecessor node
  2. ptr2 = ptr.RLINK //Track to the successor node
  3. ptr1.RLINK = ptr2 //Change the pointer as shown 1 in Figure 3.12(c)
  4. If (ptr2  $\neq$  NULL) then //If the deleted node is the last node
    1. ptr2.LLINK = ptr1 //Change the pointer shown as 2 in Figure 3.12(c)
  5. EndIf
  6. RETURNNODE(ptr) //Return the free node to the memory bank
7. Else
  1. Print "The node does not exist in the given list"
8. EndIf
9. Stop

### **Circular double Linked List:**

The advantages of both double linked list and circular linked list are incorporated into another type of list structure called circular double linked list and it is known to be the best of its kind. Figure shows a schematic representation of a circular double linked list.

Here, note that the RLINK (right link) of the rightmost node and LLINK (left link) of the leftmost node contain the address of the header node; again the RLINK and LLINK of the header node contain the address of the rightmost node and the leftmost node, respectively. An empty circular double linked list is represented as shown in Figure. In case of an empty list, both LLINK and RLINK of the header node point to itself.



**Fig. 3.13** A circular double linked list.



**Fig. 3.14** An empty circular double linked list.

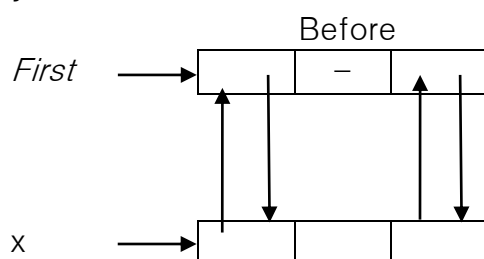
#### Insertion into a doubly circular linked list:

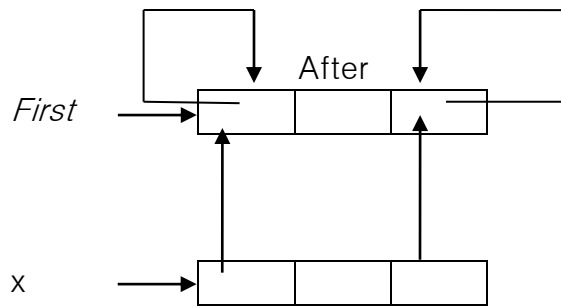
```
void dinsert(nodePointer node, nodePointer newnode)
{
 newnode->llink=node;
 newnode->rlink=node->rlink;
 node->rlink->llink=newnode;
 node->rlink=newnode;
}
```

Insertion into a doubly linked list

#### Deletion from a doubly circular linked list:

```
void ddelete(nodePointer node, nodePointer deleted)
{
 if(node==deleted)
 printf("Deletion of header node not permitted.\n");
 else
 {
 deleted->llink->rlink=deleted->rlink;
 deleted->rlink->llink=deleted->llink;
 free(deleted);
 }
}
```





## UNIT – 3

**Trees :** Introduction, Binary Trees, Binary Tree Traversals, Additional Binary Tree Operations, Binary Search Trees, Counting Binary Trees, Optimal Binary search Trees, AVL Trees. B-Trees: B- Trees, B + Trees.

## TREES

### TREE:

- This is a finite set of one or more nodes such that
  - 1) There is a specially designated node called root.
  - 2) Remaining nodes are partitioned into disjoint sets  $T_1, T_2, \dots, T_n$  where each of these are called sub trees of root (Figure 5.2).

Consider the tree shown below

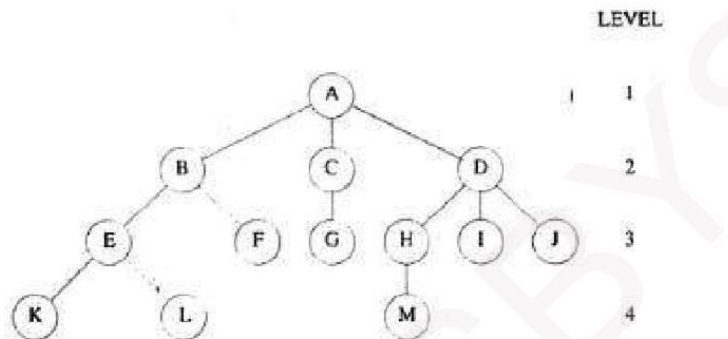
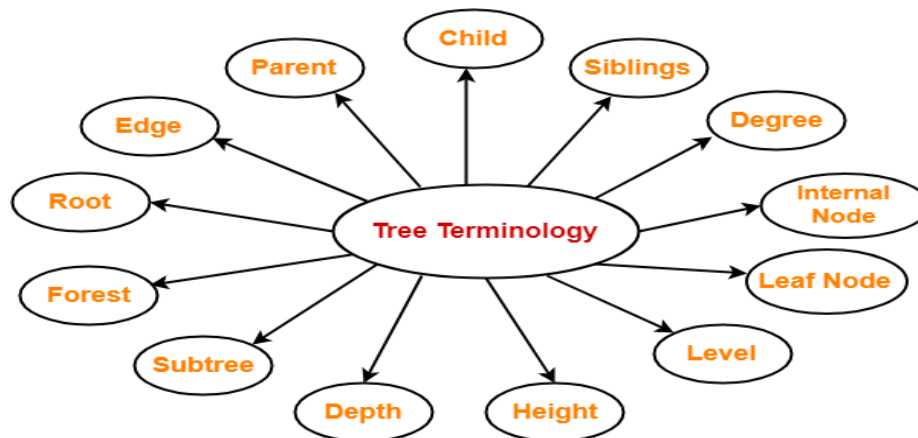


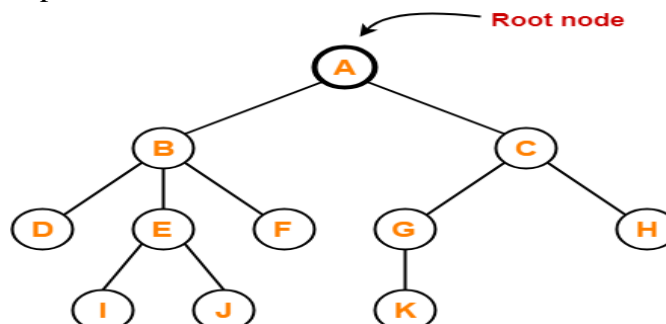
Figure 5.2: A sample tree

### Tree Terminology



### Root

- ✓ The first node from where the tree originates is called as a root node.
- ✓ In any tree, there must be only one root node.
- ✓ We can never have multiple root nodes in a tree data structure.

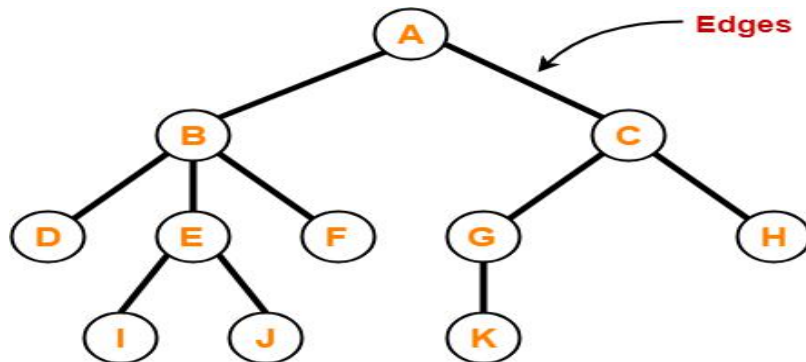


Here, node A is the only root node

## Edge

- ✓ The connecting link between any two nodes is called as an edge.
- ✓ In a tree with  $n$  number of nodes, there is exactly  $(n-1)$  number of edges.

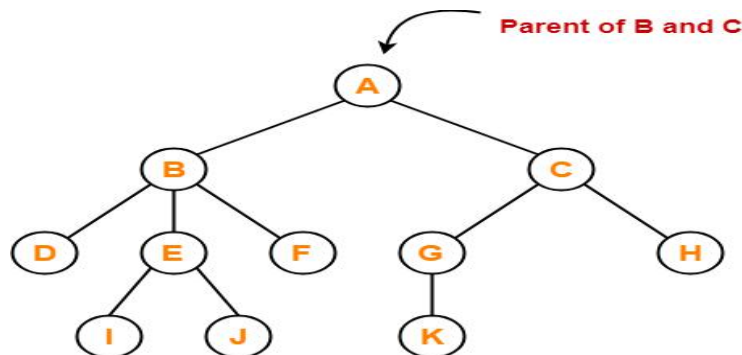
## Example



## Parent

- ✓ The node which has a branch from it to any other node is called as a parent node.
- ✓ In other words, the node which has one or more children is called as a parent node.
- ✓ In a tree, a parent node can have any number of child nodes.

## Example



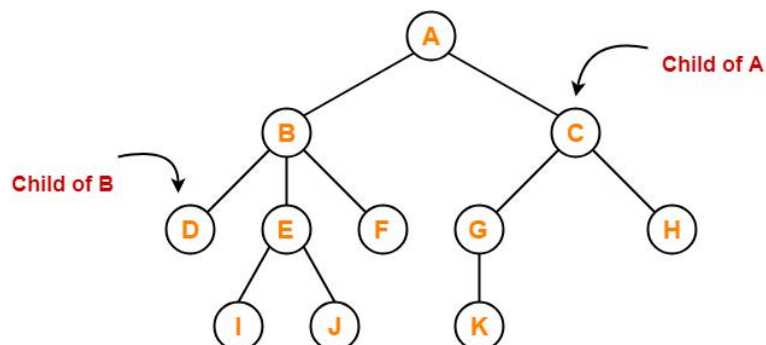
## Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

## Child

- ✓ The node which is a descendant of some node is called as a child node.
- ✓ All the nodes except root node are child nodes.

## Example



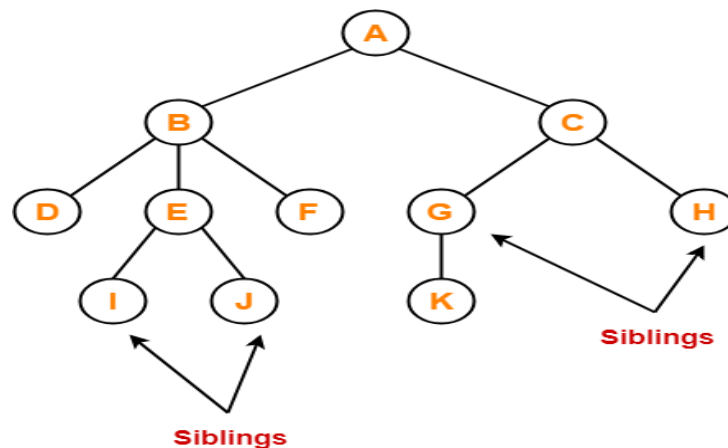
Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

### Siblings

- ✓ Nodes which belong to the same parent are called as siblings.
- ✓ In other words, nodes with the same parent are sibling nodes.

### Example



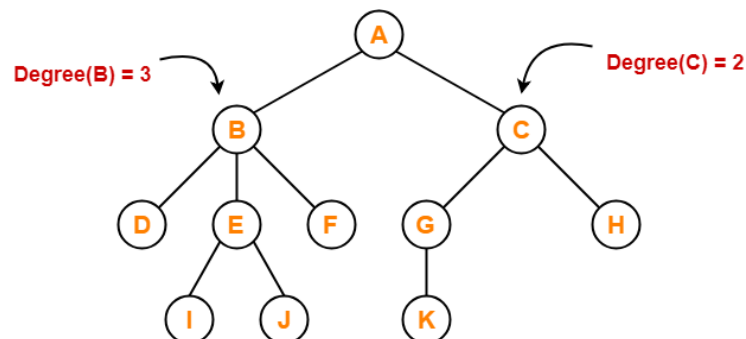
Here,

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

### Degree

- ✓ Degree of a node is the total number of children of that node.
- ✓ Degree of a tree is the highest degree of a node among all the nodes in the tree.

### Example



Here,

- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0

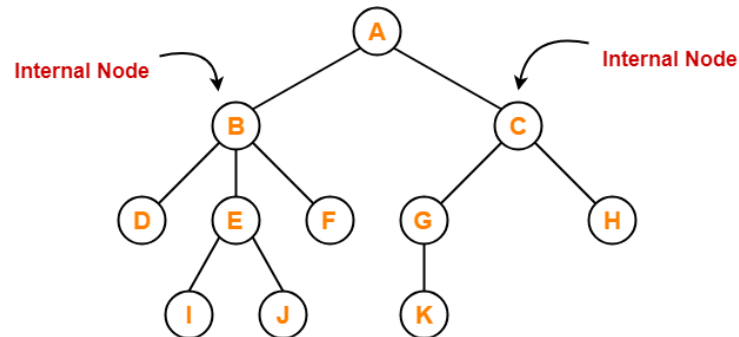


- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

### Internal Node

- ✓ The node which has at least one child is called as an internal node.
- ✓ Internal nodes are also called as non-terminal nodes.
- ✓ Every non-leaf node is an internal node.

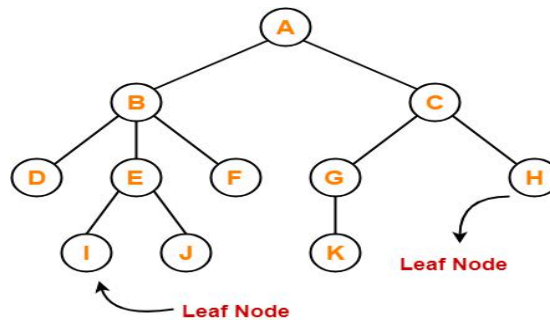
### Example



Here, nodes A, B, C, E and G are internal nodes.

### Leaf Node

- ✓ The node which does not have any child is called as a leaf node.
- ✓ Leaf nodes are also called as external nodes or terminal nodes.

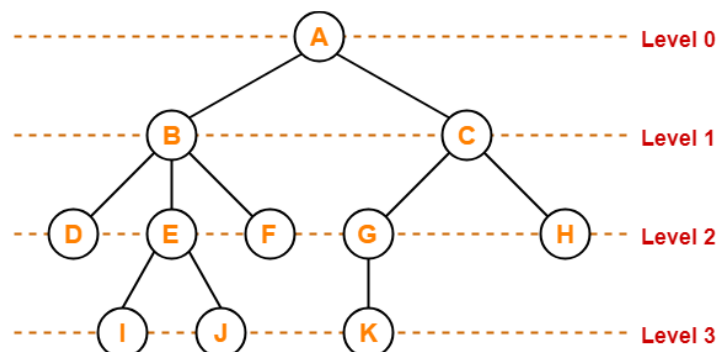


Here, nodes D, I, J, F, K and H are leaf nodes.

### Level

- ✓ In a tree, each step from top to bottom is called as level of a tree.
- ✓ The level count starts with 0 and increments by 1 at each level or step.

### Example

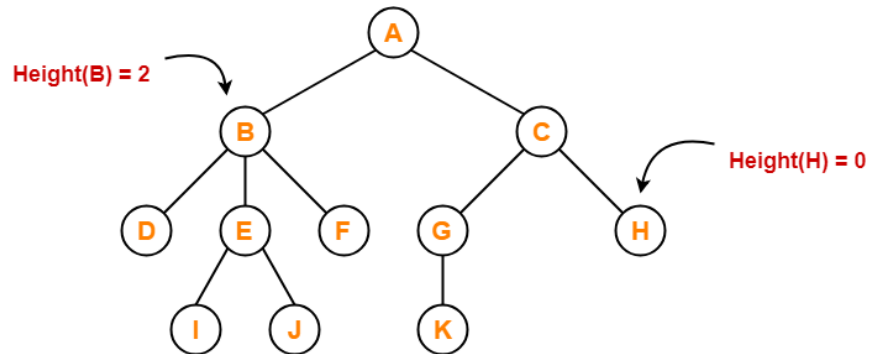




## Height

- ✓ Total number of edges that lies on the longest path from any leaf node to a particular node is called as height of that node.
- ✓ Height of a tree is the height of root node.
- ✓ Height of all leaf nodes = 0

## Example



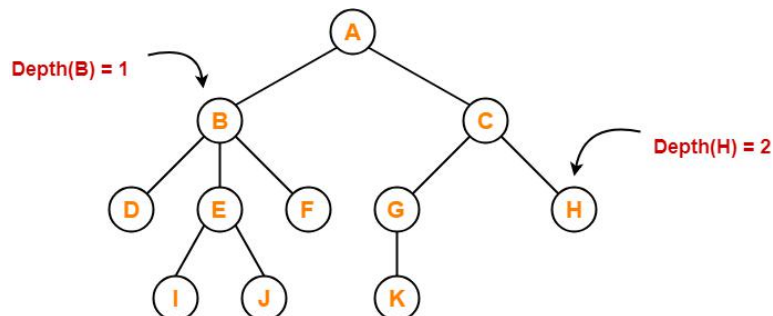
## Here

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

## Depth

- ✓ Total number of edges from root node to a particular node is called as depth of that node.
- ✓ Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
- ✓ Depth of the root node = 0
- ✓ The terms “level” and “depth” are used interchangeably.

## Example



## Here

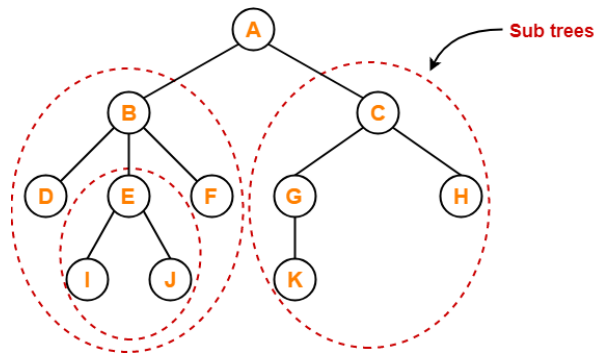
- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1

- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

### Sub-tree

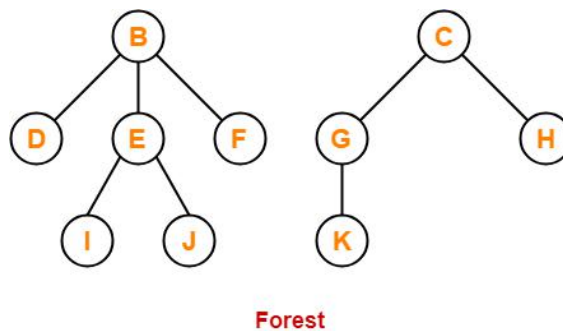
- ✓ In a tree, each child from a node forms a sub-tree recursively.
- ✓ Every child node forms a sub-tree on its parent node.

### Example



### Forest

- ✓ A forest is a set of disjoint trees.



### Advantages of Tree

- ✓ Tree reflects structural relationships in the data.
- ✓ It is used to represent hierarchies.
- ✓ It provides an efficient insertion and searching operations.

**Trees are flexible. It allows to move subtrees around with minimum effort.**

### TERMINOLOGIES USED IN A TREE

- Node contains
  - ✓ item of information &
  - ✓ links to other nodes
- Number of subtrees of a node is called its degree.  
For e.g., degree of A=3; degree of C=1
- Nodes with degree=0 are called terminal (leaf) nodes (For e.g., K, L, F, G, M, I, J) whereas other nodes

are referred to as non-terminals (For e.g., B, E, F, C, H, I, J).

- The subtrees of a node A are the children of A. A is the parent of its children. For e.g., children of D are H, I and J. Parent of D is A.
- Children of same parent are called siblings. For e.g., H, I and J are siblings.
- The maximum number of children that is possible for a node is known as **Degree of a node**. Ex: A-B-E-K in above fig.
- **Degree of a tree** is maximum of the degree of the nodes in the tree. Degree of given tree=3.
- **Ancestors** of a node are all nodes along the path from root to that node.  
For e.g., ancestors of M are A, D and H
- If a node is at level 'l', then its children are at level 'l+1'.
- **Height (or depth)** of a tree is defined as maximum level of any node in the tree. For e.g., Height of given tree = 4.

## REPRESENTATION OF TREES

A tree can be represented in three forms, namely:

- 1) List representation
- 2) Left-child right-sibling representation
- 3) Degree-two tree representation (Binary Tree)

### LIST REPRESENTATION

Consider the tree shown below

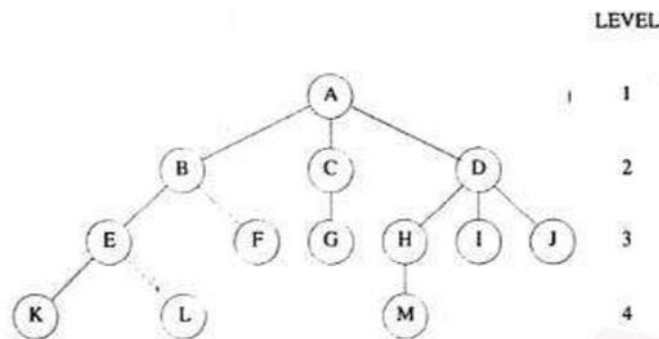
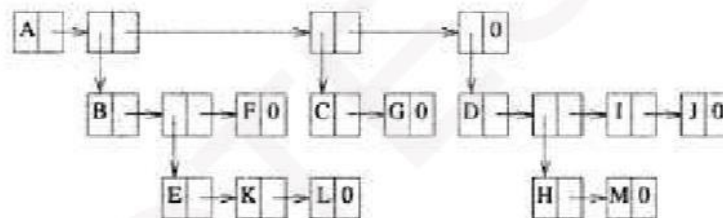


Figure 5.2: A sample tree

- The tree can be drawn as a list: (A(B(E(K,L),F),C(G),D(H(M),I,J)))
- The information in the root node comes first, followed by a list of subtrees of that node.
- Each tree-node can be represented by a memory-node that has
  - Fields for data &
  - Pointers to children of tree-node (Figure 5.3)



tag fields not shown

Figure 5.3: List representation of the tree

- For a tree of degree 'k', we can use the node-structure as shown below (Figure 5.4).

|      |         |         |     |         |
|------|---------|---------|-----|---------|
| DATA | CHILD 1 | CHILD 2 | ... | CHILD k |
|------|---------|---------|-----|---------|

Figure 5.4: Possible node structure for a tree of degree k

### LEFT CHILD-RIGHT SIBLING REPRESENTATION

- Figure 5.5 shows the node-structure used in left child-right sibling representation.

|            |               |
|------------|---------------|
| Data       |               |
| Left child | Right sibling |

Figure 5.5: Left child-Right sibling node structure

- Left-child field of each node points to its leftmost child (if any), and right-sibling field points to the closest right sibling (if any). (Figure 5.6)

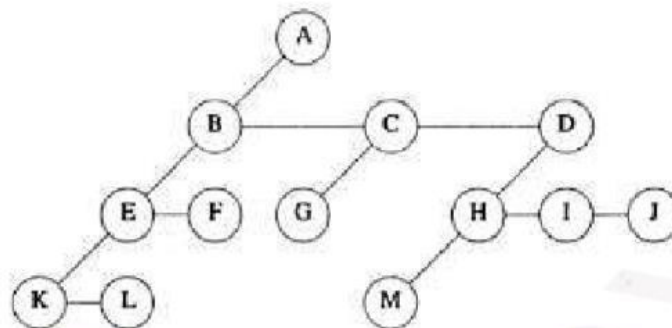


Figure 5.6: Left child-right sibling representation of tree

### DEGREE-TWO TREE REPRESENTATION

- To obtain the two-degree representation of a tree, we simply rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees.
- In this representation, we refer to 2 children of a node as left & right children (Fig 5.7).
- Left child-right child trees are also known as binary trees.

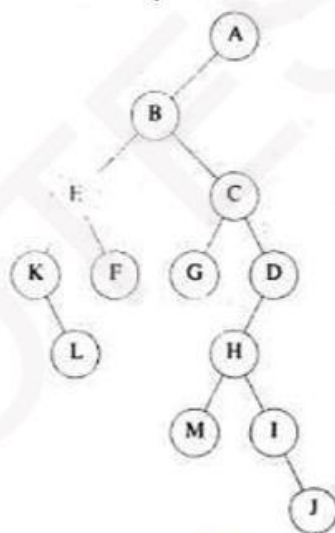


Figure 5.7: Left child-right child tree representation of tree

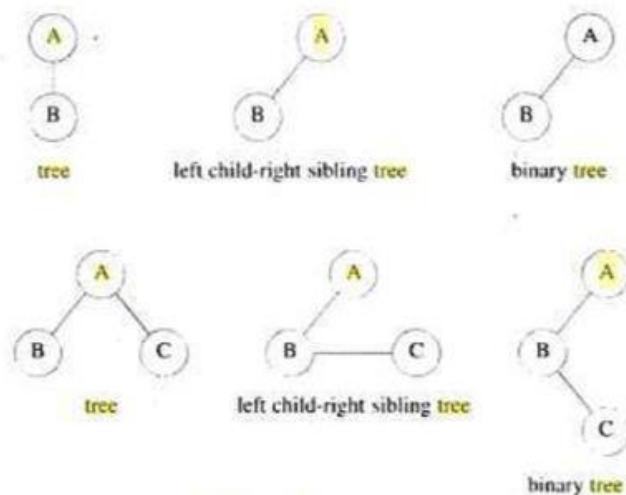


Figure 5.8: Tree representations

## BINARY TREE

- This is a finite set of nodes that is either empty or consists of
  - ✓ A root &
  - ✓ Two disjoint binary trees called left subtrees and right subtrees

**ADT Binary\_Tree**(abbreviated **BinTree**) is

**objects:** a finite set of nodes either empty or consisting of a root node, left **Binary\_Tree**, and right **Binary\_Tree**.

**functions:**

for all  $bt, bt1, bt2 \in \text{BinTree}$ ,  $item \in \text{element}$

**BinTree** Create()

::= creates an empty binary tree

**Boolean** IsEmpty( $bt$ )

::= **if** ( $bt == \text{empty binary tree}$ )

**return** TRUE **else** **return** FALSE

**BinTree** MakeBT( $bt1, item, bt2$ )

::= **return** a binary tree whose left subtree is  $bt1$ , whose right subtree is  $bt2$ , and whose root node contains the data item

**BinTree** Lchild( $bt$ )

::= **if** (IsEmpty( $bt$ ))

element Data(bt)

Bintree Rchild(bt)

**return error else return** the left subtree of bt.

::= if (IsEmpty(bt))

**return error else return** the data in the root node of bt

::= if (IsEmpty(bt)) **return error else return** the right subtree of bt

- Difference between a binary tree and a tree:

1) There is no tree having zero nodes, but there is an empty binary tree.

2) In a binary tree, we distinguish between the order of the children while in a tree we do not. i.e., in the case of binary tree a node may have at most two children(that is, a tree having a degree=2), whereas in the case of tree, a node may have any number of children.

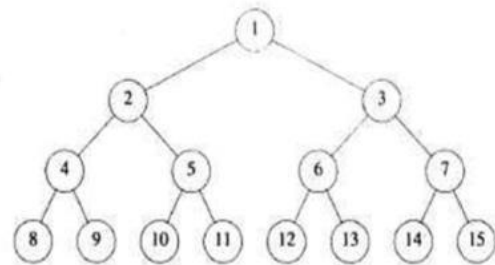
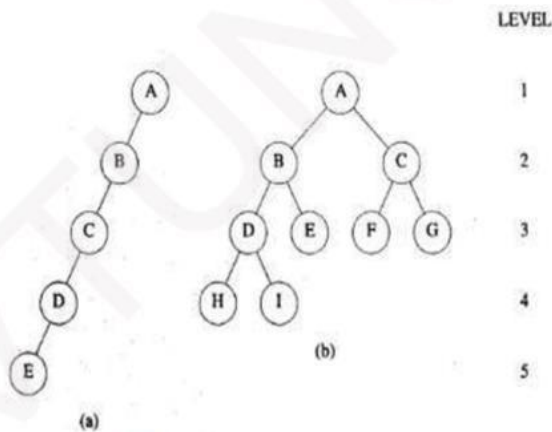
## TYPES OF BINARY TREE

1) Skewed tree is a tree consisting of only left subtree or only right subtree (Figure 5.10a).

2) A binary tree is a *full binary tree* if it contains the maximum possible number of nodes at all levels.

Full binary tree is a binary tree of depth  $k$  having  $2^k - 1$  nodes,  $k \geq 0$  (Figure 5.11).

3) Complete tree is a binary tree in which every level except possibly last level is completely filled. A binary tree with  $n$  nodes & depth  $k$  is complete iff its nodes correspond to nodes numbered from 1 to  $n$  in full binary tree of depth  $k$  (Figure 5.10B).



## PROPERTIES OF BINARY TREES

- The maximum number of nodes on level ' $i$ ' of a binary tree is  $2^{i-1}$ ,  $i \geq 1$ . (For e.g. maximum number of nodes on level  $4 = 2^{4-1} = 2^3 = 8$ ).
- The maximum number of nodes in a binary tree of depth ' $k$ ' is  $2^k - 1$ ,  $k \geq 1$ . (For e.g. maximum number of nodes with depth  $4 = 2^4 - 1 = 16 - 1 = 15$ ).
- Relation between number of leaf nodes and degree-2 nodes: For any non-empty binary tree ' $T$ ', if  $n_0$  is the number of leaf nodes and  $n_2$  the number of nodes of degree 2, then  $n_0 = n_2 + 1$ .

## BINARY TREE REPRESENTATIONS

A binary tree can be represented in two forms, namely: 1) Array Representation 2) Linked Representation

### ARRAY REPRESENTATION

- We can use a one-dimensional array to store nodes of binary tree (Figure 5.12).

**Theorem:** If a complete binary tree with ' $n$ ' nodes is represented sequentially, then for any node with index  $i$  ( $1 \leq i \leq n$ ), we have

1) parent( $i$ ) is at  $\lceil i/2 \rceil$  if  $i \neq 1$ .

If  $i = 1$ ,  $i$  is the root and has no parent.

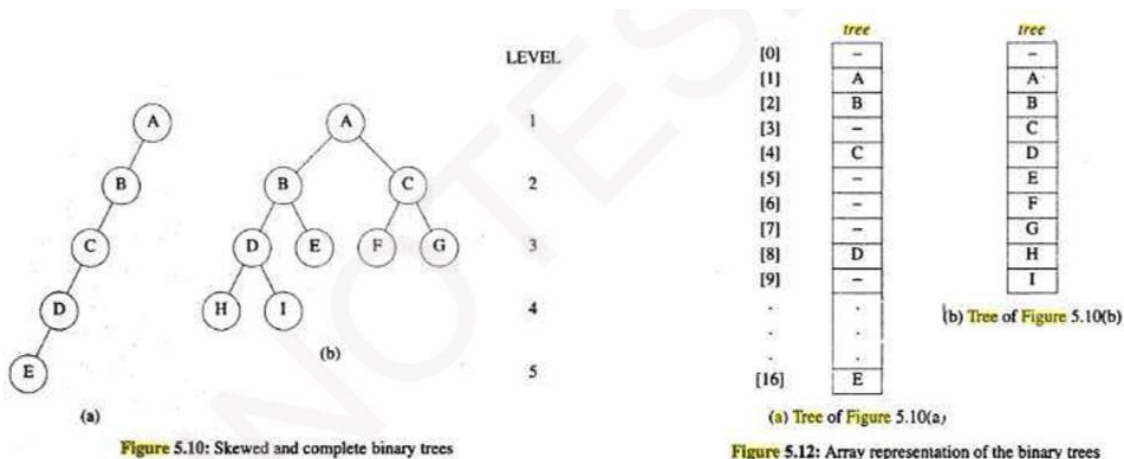
2) leftChild( $i$ ) is at  $2i$  if  $2i \leq n$ .

If  $2i > n$ , then  $i$  has no left child.

3) rightChild(i) is at  $2i+1 \leq n$ .

If  $2i+1 > n$ , then i has no right child.

- Consider the tree shown below



**Advantage:** For complete binary tree, array representation is ideal, as no space is wasted.

**Disadvantage:** For skewed tree, less than half the array is utilized. In the worst case, a skewed tree of depth k will require  $2^k - 1$  spaces. Of these, only k will be used.

### LINKED REPRESENTATION

- Shortcoming of array representation: Insertion and deletion of nodes from middle of a tree requires movement of potentially many nodes to reflect the change in level number of these nodes. These problems can be overcome easily through the use of a linked representation (Figure 5.14).
- Each node has three fields:
  - 1) leftChild,
  - 2) data and
  - 3) rightChild (Figure 5.13).

```
typedef struct node *treePointer;
typedef struct
{
 int data;
 treePointer leftChild, rightChild;
}
node;
```

- Root of tree is stored in the data member 'root' of Tree. This data member serves as access-pointer to the tree.

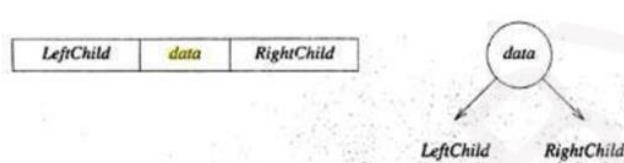


Figure 5.13: Node representations

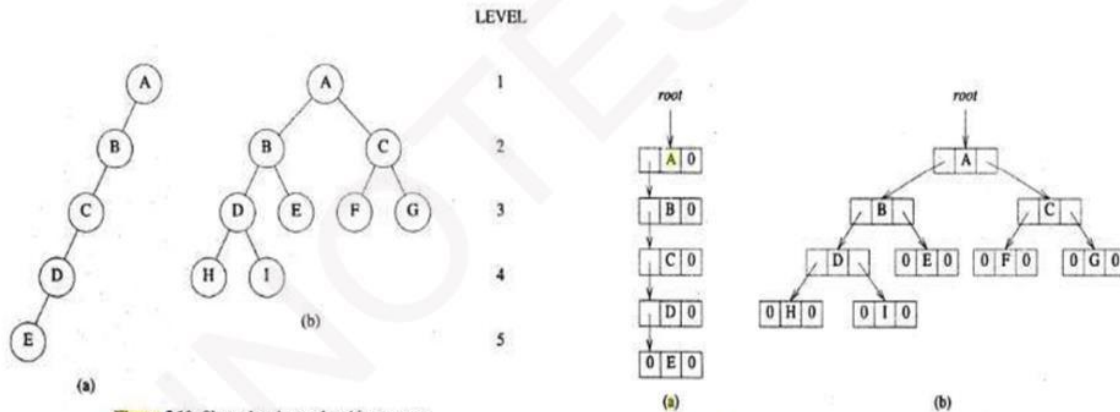


Figure 5.10: Skewed and complete binary trees

Figure 5.14: Linked representation for the binary trees

## BINARY TREE TRAVERSALS

- Tree traversal refers to process of visiting all nodes of a tree exactly once (Figure 5.16).
- There are 3 techniques, namely:
  - 1) Inorder traversal(LVR);
  - 2) Preorder traversal(VLR);
  - 3) Postorder traversal(LRV). (Let L= moving left, V= visiting node and R=moving right).
- In postorder, we visit a node after we have traversed its left and right subtrees.

In preorder, the visiting node is done before traversal of its left and right subtrees.

In inorder, firstly node's left subtrees is traversed, then node is visited and finally node's right subtrees is traversed.

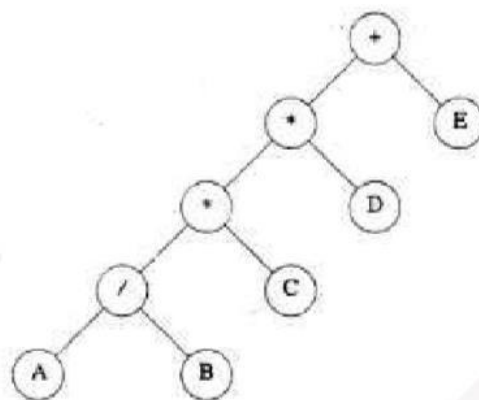


Figure 5.16: Binary tree with arithmetic expression

## INORDER TRAVERSAL (LVR):

- Inorder traversal calls for moving down tree toward left until you can go no farther (Program 5.1).
- Then, you "visit" the node, move one node to the right and continue.



- If you cannot move to the right, go back one more node.

```
void inorder(tree_pointer ptr)
{
 /* inorder tree traversal */
 if (ptr)
 {
 inorder(ptr->left_child);
 printf("%d", ptr->data);
 indorder(ptr->right_child);
 }
}
```

Program 5.1: Inorder traversal of binary tress

| Call of<br>inorder | Value in<br>root | Action | inorder | In root | Value<br>Action |
|--------------------|------------------|--------|---------|---------|-----------------|
| 1                  | +                |        | 11      | C       |                 |
| 2                  | *                |        | 12      | NULL    |                 |
| 3                  | *                |        | 11      | C       | printf          |
| 4                  | /                |        | 13      | NULL    |                 |
| 5                  | A                |        | 2       | *       | printf          |
| 6                  | NULL             |        | 14      | D       |                 |
| 5                  | A                | printf | 15      | NULL    |                 |
| 7                  | NULL             |        | 14      | D       | printf          |
| 4                  | /                | printf | 16      | NULL    |                 |
| 8                  | B                |        | 1       | +       | printf          |
| 9                  | NULL             |        | 17      | E       |                 |
| 8                  | B                | printf | 18      | NULL    |                 |
| 10                 | NULL             |        | 17      | E       | printf          |
| 3                  | *                | printf | 19      | NULL    |                 |

Figure: Trace of

- Each shows inorder,

root, and whether or not the printf function is invoked (Figure 5.17).

- Since there are 19 nodes in the tree, inorder() is invoked 19 times for the complete traversal. The nodes of figure 5.16 would be output in an inorder as

A/B\*C\*D+E

#### PREORDER TRAVERSAL (VLR):

- Visit a node, traverse left, and continue (Program 5.2).
- When you cannot continue, move right and begin again or move back until you can move right and resume.
- The nodes of figure 5.16 would be output in preorder as

Program 5.1

step of the trace  
the call of  
the value in the

+\*\*/ABCDE

```
void preorder(tree_pointer ptr)
{
 /* preorder tree traversal */
 if (ptr)
 {
 printf("%d", ptr->data);
 preorder(ptr->left_child);
 preorder(ptr->right_child);
 }
}
```

Program 5.2: Preorder traversal of binary tree

### POSTORDER TRAVERSAL (LRV):

- Visit a node, traverse right, and continue (Program 5.3).
- When you cannot continue, move left and begin again or move back until you can move left and resume.
- The nodes of figure 5.16 would be output in postorder as

AB/C\*D\*E+

```
void preorder(tree_pointer ptr)
{
 /* preorder tree traversal */
 if (ptr)
 {
 printf("%d", ptr->data);
 preorder(ptr->left_child);
 preorder(ptr->right_child);
 }
}
```

Program 5.3: Postorder traversal of binary tree

### ITERATIVE INORDER TRAVERSAL

- wrt figure 5.17, a node that has no action indicates that the node is added to the stack, while a node that has a printf action indicates that the node is removed from the stack (Program 5.4).
- The left nodes are stacked until a null node is reached, the node is then removed from the stack, and the node's right child is stacked.
- The traversal continues with the left child.
- The traversal is complete when the stack is empty.

```
void iter_inorder(tree_pointer node)
{
 int top = -1; /* initialize stack */
 tree_pointer stack[MAX_STACK_SIZE];
 for (;;)
 {
 for (; node; node = node->left_child)
 add(&top, node); /* add to stack */
 node = delete(&top); /* delete from stack */
 if (!node) break; /* empty stack */
 printf("%d", node->data);
 node = node->right_child;
 }
}
```

Program 5.4: Iterative Inorder traversal of binary tree

### LEVEL-ORDER TRAVERSAL

- This traversal uses a queue (Program 5.7).

- We visit the root first, then the root's left child followed by the root's right child.
- We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.

```

void level_order(tree_pointer ptr)
{
 /* level order tree traversal */
 int front = rear = 0;
 tree_pointer queue[MAX_QUEUE_SIZE];
 if (!ptr) return; /* empty queue */
 addq(front, &rear, ptr);
 for (;;)
 {
 ptr = deleteq(&front, rear);
 if (ptr)
 {
 printf("%d", ptr->data);
 if (ptr->left_child)
 addq(front, &rear, ptr->left_child);
 if (ptr->right_child)
 addq(front, &rear, ptr->right_child);
 }
 else
 break;
 }
}

```

Program 5.7: Level order traversal of binary tree

Example for Traversals:

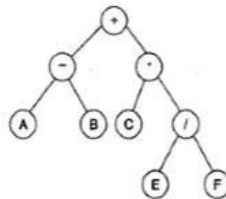


Figure 7.21 A binary tree representing an arithmetic expression.

Visit 1:

$+ T_{R_L} T_{R_R}$   
 $+ - T_{L_L} T_{R_R}$   
 $+ - A T_{L_R} T_{R_R}$   
 $+ - A B T_{R_R}$   
 $+ - A B * T_{R_R}$   
 $+ - A B * C T_{R_R}$   
 $+ - A B * C / T_{R_R}$   
 $+ - A B * C / E T_{R_R}$   
 $+ - A B * C / E F T_{R_R}$   
 $+ - A B * C / E F$

Likewise, one can obtain the other visits as shown below (only the result):

Visit 2:

$A - B + C * E / F$

Visit 3:

$A B - C E F / * +$

Visit 4:

$F E / C * B A - +$

Visit 5:

$F / E * C + B - A$

Visit 6:

$+ * / F E C - B A$

Here Visit 1 is: VLR(Vector, Left, Right)

Visit 2 is: LVR (Left, Vector, Right)

Visit 3 is: LRV (Left, Right, Vector)

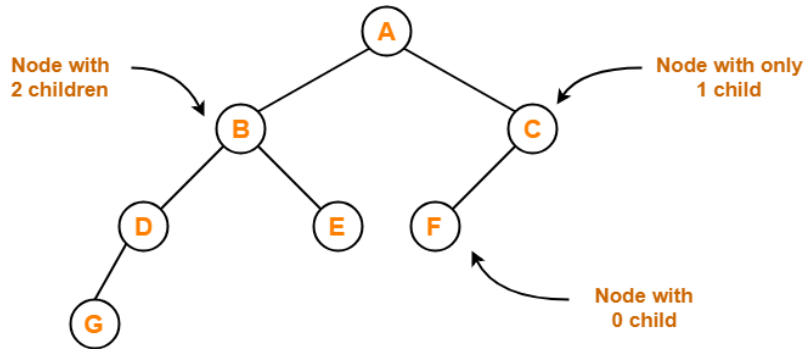
Visit 4 is: RLV (Right, Left, Vector)

Visit 5 is: RVL (Right, Vector, Left)

Visit 6 is: VRL (Vector, Right, Left)

## Binary Tree

- ✓ Binary tree is a special tree data structure in which each node can have at most 2 children.
- ✓ Thus, in a binary tree, each node has either 0 child or 1 child or 2 children.



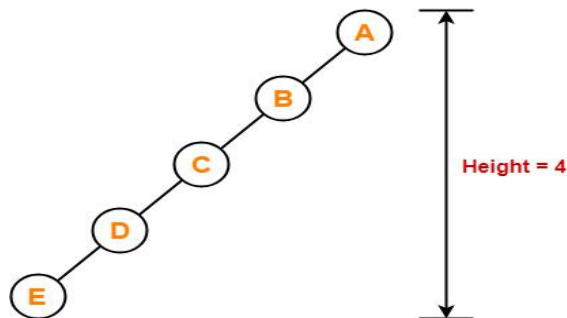
Binary Tree Example

## Binary Tree Properties

1 Minimum number of nodes in a binary tree of height  $H = H + 1$

### Example

To construct a binary tree of height = 4, we need at least  $4 + 1 = 5$  nodes.



2. Maximum number of nodes in a binary tree of height  $H = 2^{H+1} - 1$

### Example

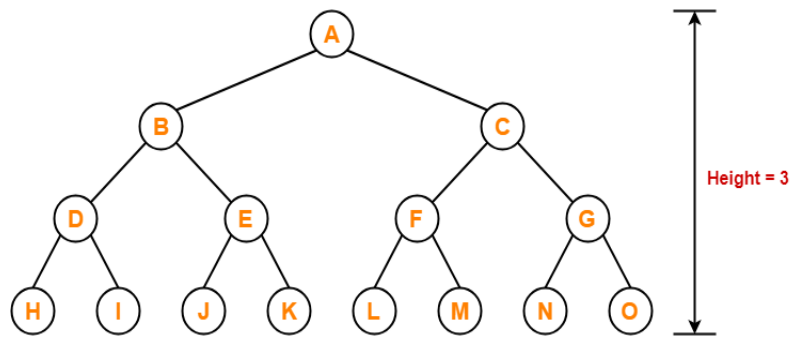
Maximum number of nodes in a binary tree of height 3

$$= 2^{3+1} - 1$$

$$= 16 - 1$$

$$= 15 \text{ node}$$

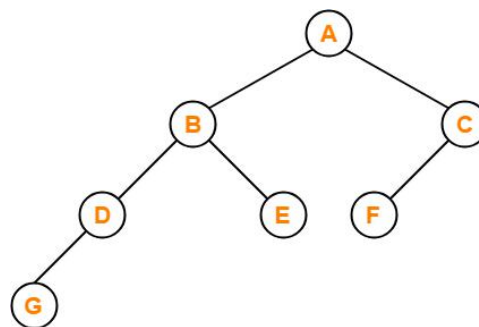
Thus, in a binary tree of height = 3, maximum number of nodes that can be inserted = 15.



**We cannot insert more number of nodes in this binary tree**

**3. Total Number of leaf nodes in a Binary Tree = Total Number of nodes with 2 children + 1**

**Example**



Here

Number of leaf nodes = 3

Number of nodes with 2 children = 2

Clearly, number of leaf nodes is one greater than number of nodes with 2 children.  
This verifies the above relation.

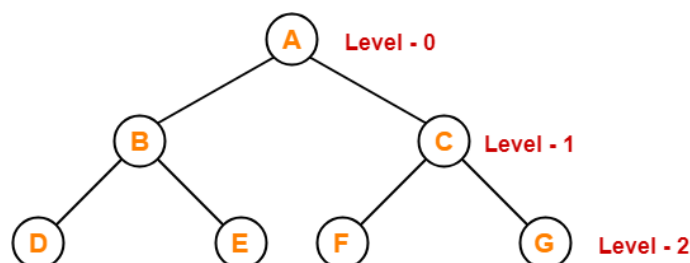
**4. Maximum number of nodes at any level 'L' in a binary tree =  $2^L$**

**Example**

Maximum number of nodes at level-2 in a binary tree

= 22

= 4



Thus, in a binary tree, maximum number of nodes that can be present at level-2 = 4.

## Types of Binary Trees

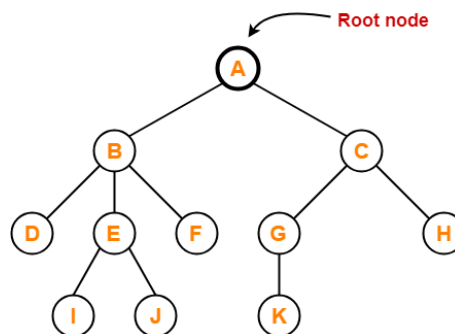
- ✓ Rooted Binary Tree
- ✓ Full / Strictly Binary Tree
- ✓ Complete / Perfect Binary Tree
- ✓ Almost Complete Binary Tree
- ✓ Skewed Binary Tree

### 1. Rooted Binary Tree

A rooted binary tree is a binary tree that satisfies the following 2 properties

- ✓ It has a root node.
- ✓ Each node has at most 2 children.

**Example:**

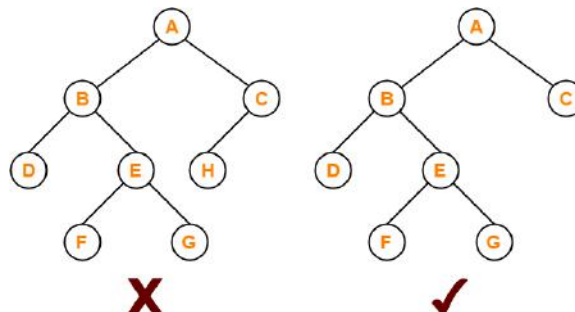


### 2. Full / Strictly Binary Tree-

A binary tree in which every node has either 0 or 2 children is called as a full binary tree.

Full binary tree is also called as strictly binary tree.

**Example:**



**Here**

- First binary tree is not a full binary tree.
- This is because node C has only 1 child.

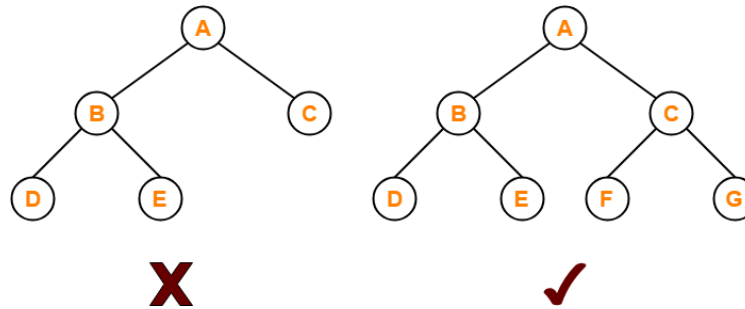
### 3. Complete / Perfect Binary Tree

A complete binary tree is a binary tree that satisfies the following 2 properties

- ✓ Every internal node has exactly 2 children.
- ✓ All the leaf nodes are at the same level.

Complete binary tree is also called as Perfect binary tree.

**Example:**



**Here**

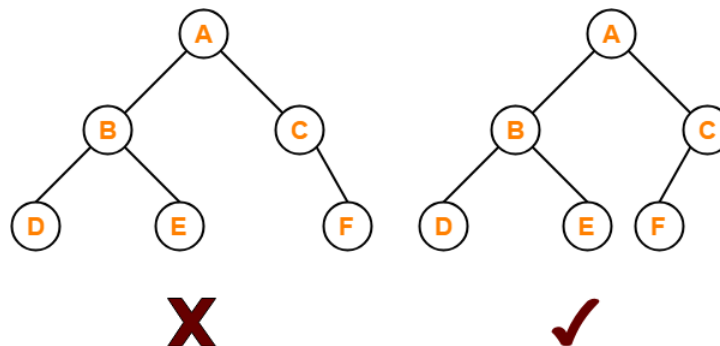
- First binary tree is not a complete binary tree.
- This is because all the leaf nodes are not at the same level.

#### 4. Almost Complete Binary Tree

An almost complete binary tree is a binary tree that satisfies the following 2 properties

- ✓ All the levels are completely filled except possibly the last level.
- ✓ The last level must be strictly filled from left to right.

**Example:**



#### 5. Skewed Binary Tree

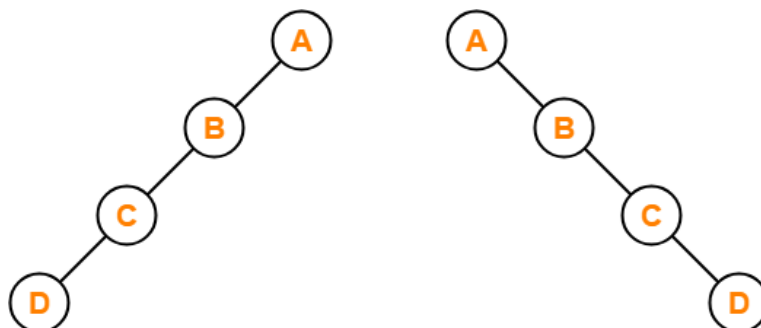
A skewed binary tree is a binary tree that satisfies the following 2 properties

- ✓ All the nodes except one node have one and only one child.
- ✓ The remaining node has no child.

OR

- ✓ A skewed binary tree is a binary tree of  $n$  nodes such that its depth is  $(n-1)$ .

**Example:**



**Left Skewed Binary Tree**

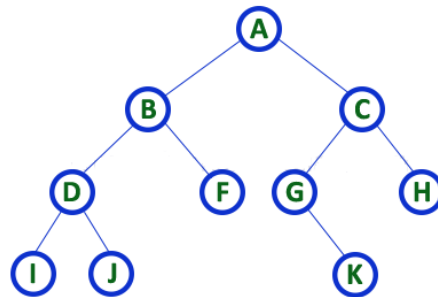
**Right Skewed Binary Tree**

## Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

- ✓ Array Representation
- ✓ Linked List Representation

Consider the following binary tree



### 1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.



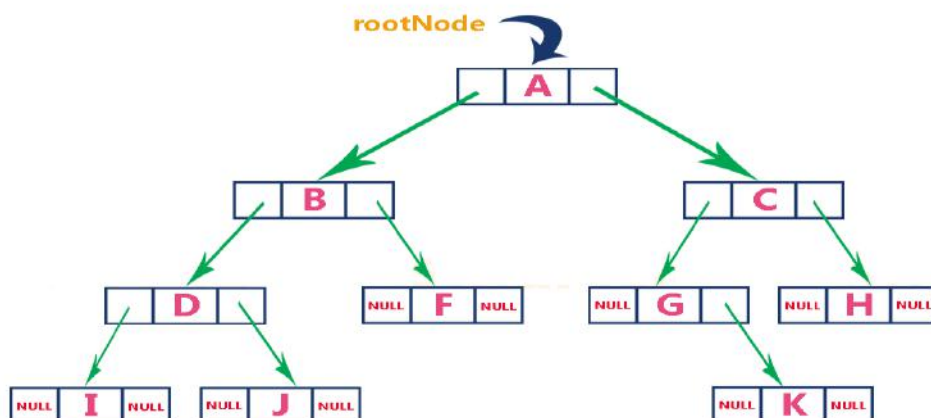
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of  $2n + 1$ .

### 2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.



The above example of the binary tree represented using Linked list representation is



## Linked Representation

struct node



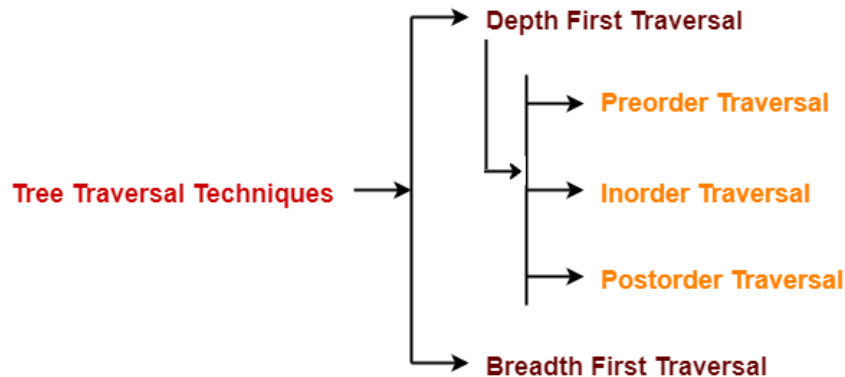
```

{
 int data;
 struct node *left;
 struct node *right;
};

```

## Binary Tree Traversals

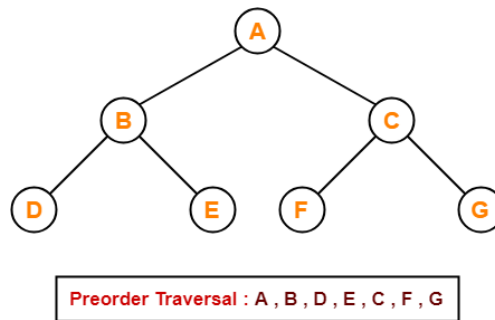
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree.



- ✓ Pre-order Traversal
- ✓ In-order Traversal
- ✓ Post-order Traversal

## Pre-order Traversal

In this traversal method, the root node is visited first, then the left sub tree and finally the right sub tree.



We start from A, and following pre-order traversal, we first visit **A** itself and then move to its left sub tree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be

**A → B → D → E → C → F → G**

## Steps

- ✓ Visit the root node
- ✓ traverse the left sub-tree in pre-order
- ✓ traverse the right sub-tree in pre-order

**Root → Left → Right**

## Algorithm

Step 1: Repeat Steps 2 to 4 while TREE! = NULL

Step 2: Write TREE -> DATA

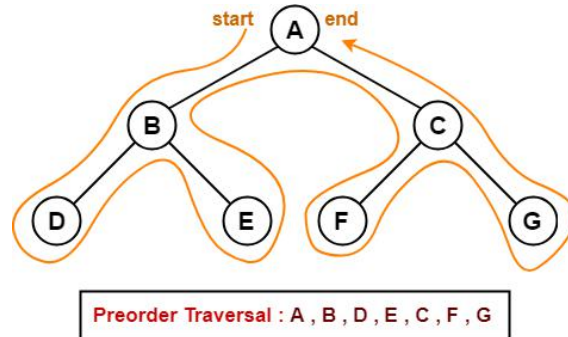
Step 3: PREORDER (TREE -> LEFT)

Step 4: PREORDER (TREE -> RIGHT)

[END OF LOOP]

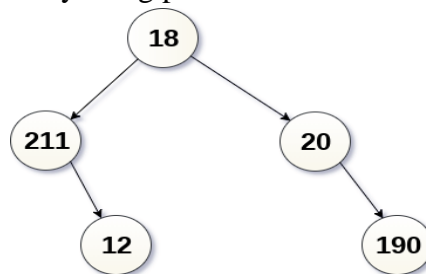
Step 5: END

Traverse the entire tree starting from the root node keeping yourself to the left.



### Example

Traverse the following binary tree by using pre-order traversal



- ✓ Since, the traversal scheme, we are using is pre-order traversal, therefore, the first element to be printed is 18.
- ✓ Traverse the left sub-tree recursively. The root node of the left sub-tree is 211, print it and move to left.
- ✓ Left is empty therefore print the right children and move to the right sub-tree of the root.
- ✓ 20 are the root of sub-tree therefore, print it and move to its left. Since left sub-tree is empty therefore move to the right and print the only element present there i.e. 190.
- ✓ Therefore, the printing sequence will be 18, 211, 90, 20, and 190.

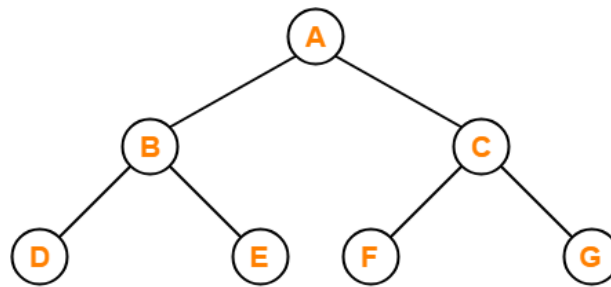
### Applications

- ✓ Preorder traversal is used to get prefix expression of an expression tree.
- ✓ Preorder traversal is used to create a copy of the tree.

### In order Traversal

In this traversal method, the left sub tree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a sub tree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



Inorder Traversal : D , B , E , A , F , C , G

We start from **A**, and following in-order traversal, we move to its left sub tree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in order traversal of this tree will be –

**D → B → E → A → F → C → G**

### Steps

- ✓ Traverse the left sub-tree in in-order
- ✓ Visit the root
- ✓ Traverse the right sub-tree in in-order

**Left → Root → Right**

### Algorithm

Step 1: Repeat Steps 2 to 4 while TREE! = NULL

Step 2: INORDER (TREE → LEFT)

Step 3: Write TREE → DATA

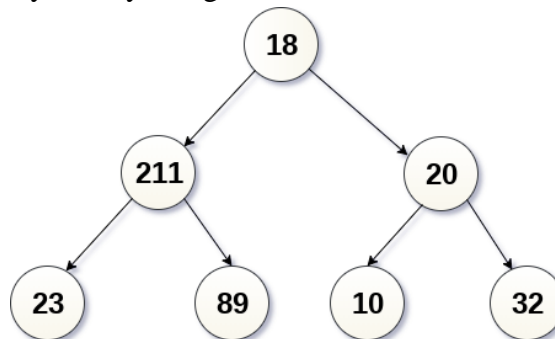
Step 4: INORDER (TREE → RIGHT)

[END OF LOOP]

Step 5: END

### Example

Traverse the following binary tree by using in-order traversal.



- ✓ Print the left most node of the left sub-tree i.e. 23.
- ✓ Print the root of the left sub-tree i.e. 211.
- ✓ Print the right child i.e. 89.
- ✓ Print the root node of the tree i.e. 18.

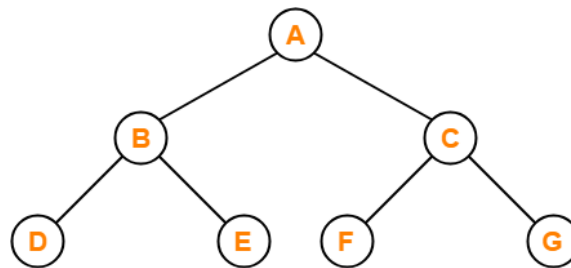
- ✓ Then, move to the right sub-tree of the binary tree and print the left most node i.e. 10.
- ✓ Print the root of the right sub-tree i.e. 20.
- ✓ Print the right child i.e. 32.
- ✓ Hence, the printing sequence will be 23, 211, 89, 18, 10, 20, and 32.

### Application

In order traversal is used to get infix expression of an expression tree.

### Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left sub tree, then the right sub tree and finally the root node.



**Postorder Traversal : D , E , B , F , G , C , A**

We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be

**D → E → B → F → G → C → A**

### Steps

- ✓ Traverse the left sub-tree in post-order
- ✓ Traverse the right sub-tree in post-order
- ✓ visit the root

**Left → Right → Root**

### Algorithm

Step 1: Repeat Steps 2 to 4 while TREE! = NULL

Step 2: POSTORDER (TREE -> LEFT)

Step 3: POSTORDER (TREE -> RIGHT)

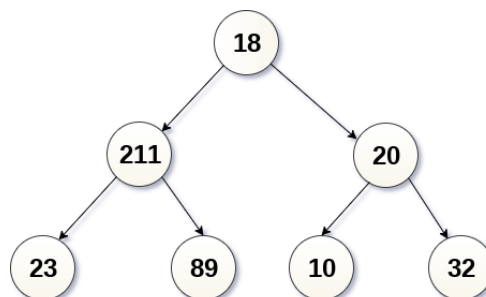
Step 4: Write TREE -> DATA

[END OF LOOP]

Step 5: END

### Example

Traverse the following tree by using post-order traversal



- ✓ Print the left child of the left sub-tree of binary tree i.e. 23.
- ✓ Print the right child of the left sub-tree of binary tree i.e. 89.
- ✓ Print the root node of the left sub-tree i.e. 211.
- ✓ Now, before printing the root node, move to right sub-tree and print the left child i.e. 10.
- ✓ Print 32 i.e. right child.
- ✓ Print the root node 20.
- ✓ Now, at the last, print the root of the tree i.e. 18.
- ✓ The printing sequence will be 23, 89, 211, 10, 32, and 18.

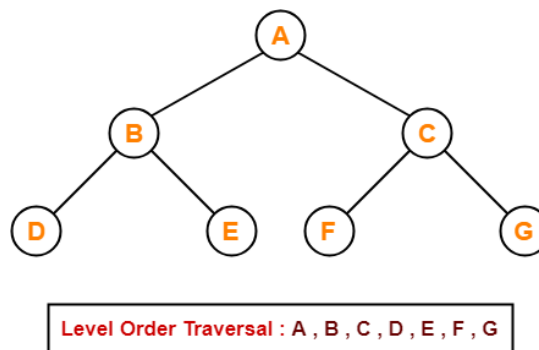
### Applications

- ✓ Post order traversal is used to get postfix expression of an expression tree.
- ✓ Post order traversal is used to delete the tree.
- ✓ This is because it deletes the children first and then it deletes the parent.

### Breadth First Traversal

- ✓ Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- ✓ Breadth First Traversal is also called as Level Order Traversal.

#### Example



#### Application

Level order traversal is used to print the data in the same order as stored in the array representation of a complete binary tree.

### Additional Binary Tree Operations

#### Copying Binary Trees

Using the definition of a binary tree and the recursive version of the traversals, we can easily write other routines for working with binary trees. For instance, if we want to implement a copy constructor to initialize a binary tree with an exact copy of another binary tree, we can modify the postorder traversal algorithm only slightly to get Program

#### Testing Equality

Another problem that is especially easy to solve using recursion is determining the equivalence of two binary trees. Binary trees are equivalent if they have the same topology and the information in corresponding nodes is identical. By the same topology we mean that every branch in one tree corresponds to a branch in the second in the same order and vice versa.

The function operator=O calls the helper function equal (Program 5.10) which traverses the binary trees in preorder, though any order could be used.

## The Satisfiability Problem

Consider the set of formulas we can construct by taking variables  $x_1, x_2, \dots, x_n$  and the operators  $\wedge$  (and),  $\vee$  (or), and  $\neg$  (not). These variables can hold only one of two possible values, true or false. The Set of expressions that can be formed using these variables and operation is defined by the following rules:

- (1) It variable is an expression.
- (2) if  $x$  and  $y$  are expressions, then  $\neg x$ ,  $x \wedge y$ ,  $x \vee y$  are expressions.
- (3) Parentheses can be used to alter the normal order of evaluation, which is  $\neg$  before  $\wedge$  before  $\vee$ .

This set defines the formulas of the propositional calculations (other operations such as implication can be expressed using  $\vee, \wedge$  and  $\neg$ ).

$$x_1 \vee (x_2 \wedge \neg x_3)$$

is a formula (read as “ $x_1$  or  $x_2$  and not  $x_3$ ”). If  $x_1$  and  $x_3$  are false and  $x_2$  is true, then the value of expression is:

$$\text{false} \vee (\text{true} \wedge \neg \text{false})$$

$$= \text{false} \vee \text{true}$$

$$= \text{true}$$

The satisfiability problem for formulas of propositional calculus asks if there is an assignment of values to the variables that causes the value of the expression to be true.

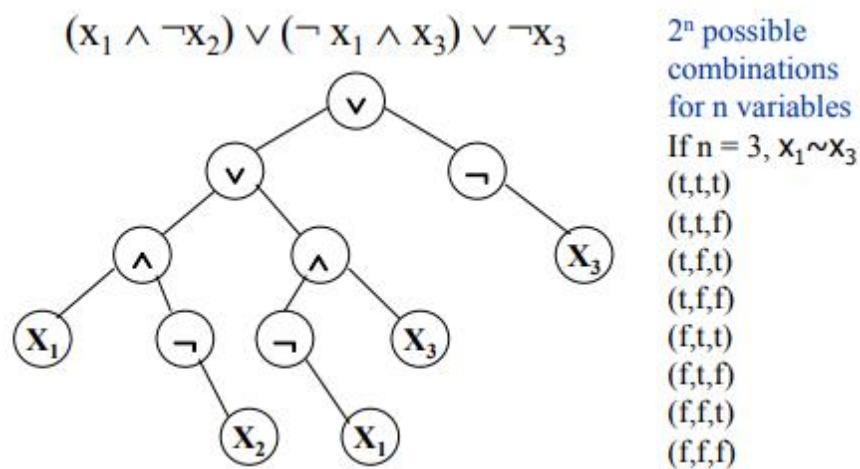
Again, let us assume that our formula is already in a binary tree, say

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

In the tree of below fig., the inorder traversal of this tree is:

$$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_3 \vee \neg x_3$$

which is the infix form of the expression. The most obvious algorithm to determine satisfiability is to let  $(x_1, x_2, x_3)$  take on all possible combinations of true and false values and to check the formula for each combination. For  $n$  variables there are  $2^n$  possible combinations of true = t and false = f. For example, for  $n = 3$ , the eight combinations are: (t,t,t), (t,t,f), (t,f,t), (t,f,f), (f,t,t), (f,t,f), (f,f,t), (f,f,f). The algorithm will take  $O(g 2^n)$ , or exponential time, where  $g$  is the time to substitute values for  $x_1, x_2, \dots, x_n$  and evaluate the expression.



## Propositional formula in a binary tree

To evaluate an expression we can traverse its tree in post order, evaluating sub trees until the entire expression is reduced to a single value. This corresponds to the postfix evaluation of an arithmetic expression that we saw earlier. Viewing this from the perspective of the tree representation, for every node we reach, the

values of its arguments (or children) have already been computed. So when we reach the  $v$  node on level 2, the values of  $x_1 \wedge \neg x_2$  and  $\neg x_1 \wedge x_3$  will already be available to us, and we can apply the rule for or. Notice that a node. Containing  $\neg$  has only a right branch, since  $\neg$  is a unary operator. For the purposes of our evaluation algorithm, we assume each node has four fields:

|            |      |       |             |
|------------|------|-------|-------------|
| Left child | data | value | Right child |
|------------|------|-------|-------------|

This is the node structure. The left child and right child fields are similar to those used previously. The fields data holds either the value of a variable or a propositional calculus operator, while value holds either a value of TRUE and FALSE.

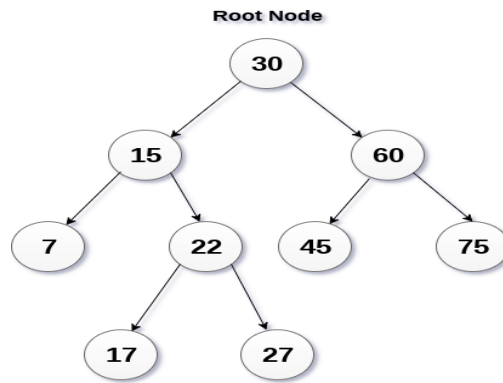
We define node structure in C as:

```
typedef enum
{
 not,
 and,
 or,
 true,
 false
} logical;
typedef struct node *treePointer;
typedef struct
{
 treePointer leftChild;
 logical data;
 short int value;
 treePointer rightChild;
} node;
```

Also we assume that for leaf nodes, node  $\rightarrow$  data contains the current value (i.e., Logical True or Logical False) of the variable represented at this node. The first version of our algorithm for the satisfiability problem is Program 5.11. In this,  $n$  is the number of variables in the formula and formula is the binary tree that represents the formula.

## Binary Search Trees

- ✓ Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
- ✓ In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
- ✓ Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
- ✓ This rule will be recursively applied to all the left and right sub-trees of the root.



**Binary Search Tree**

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

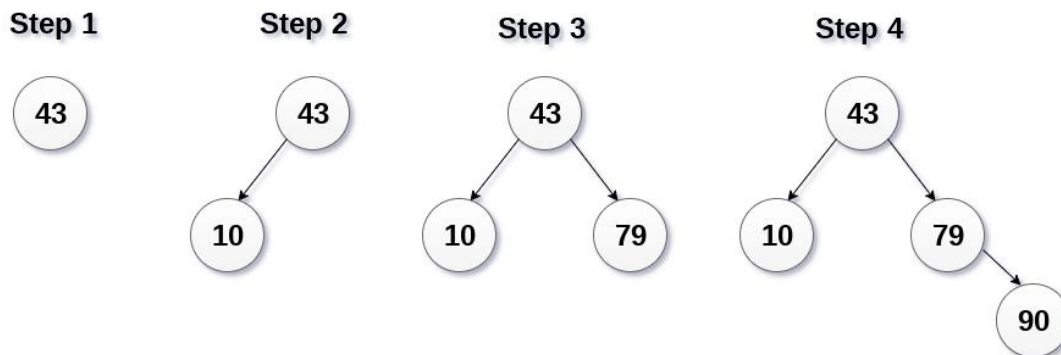
#### **Advantages of using binary search tree**

- ✓ Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
- ✓ The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes  $O(\log_2 n)$  time. In worst case, the time it takes to search an element is  $O(n)$ .
- ✓ It also speed up the insertion and deletion operations as compare to that in array and linked list.

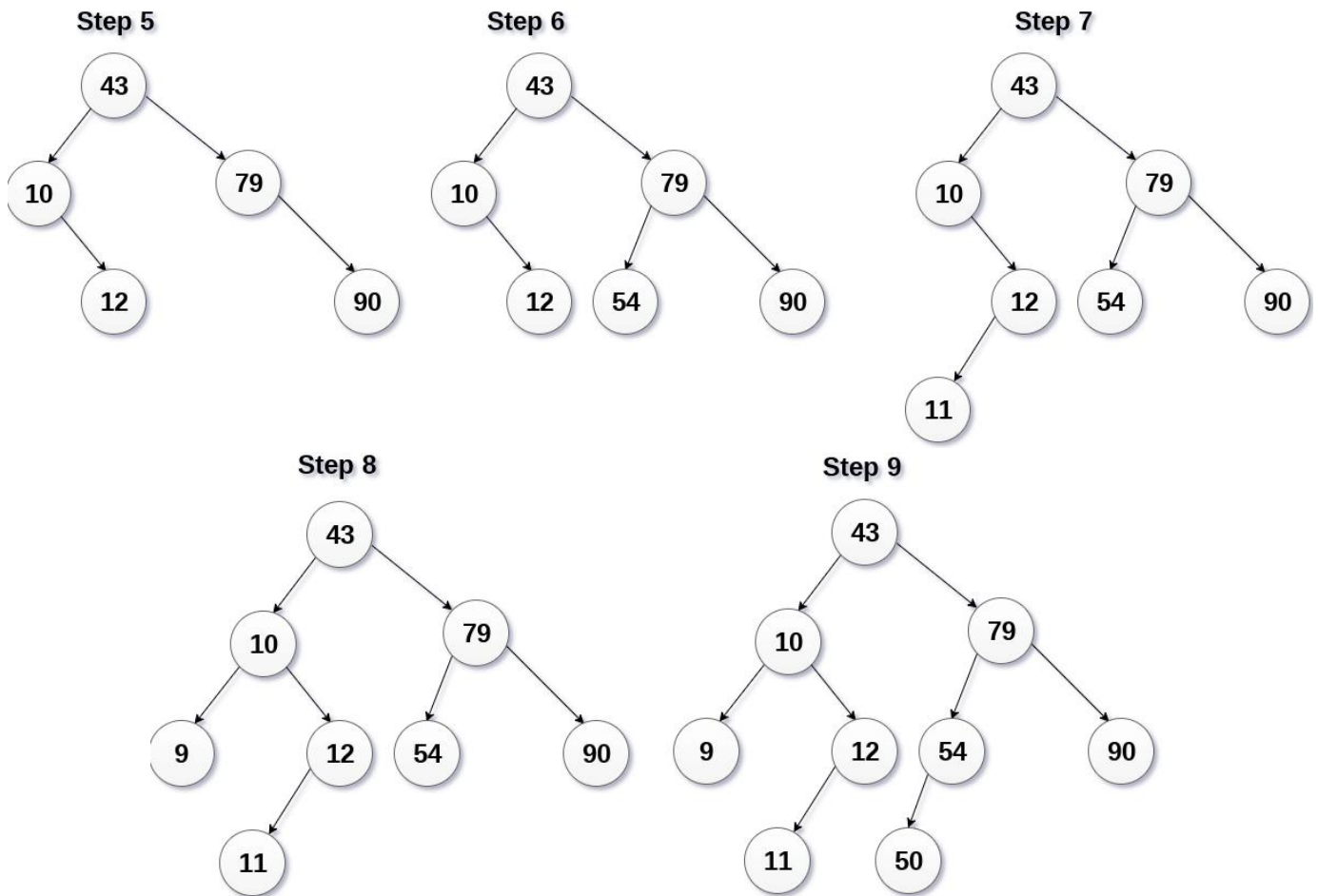
**Create the binary search tree using the following data elements.**

43, 10, 79, 90, 12, 54, 11, 9, 50

- ✓ Insert 43 into the tree as the root of the tree.
- ✓ Read the next element, if it is lesser than the root node element insert it as the root of the left sub-tree.
- ✓ Otherwise, insert it as the root of the right of the right sub-tree.







## Operations on Binary Search Tree

Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST is stored in a particular order.

- ✓ Compare the element with the root of the tree.
- ✓ If the item is matched then return the location of the node.
- ✓ Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
- ✓ If not, then move to the right sub-tree.
- ✓ Repeat this procedure recursively until match found.
- ✓ If element is not found then return NULL.

### Algorithm:

Search (ROOT, ITEM)

Step 1: IF ROOT -> DATA = ITEM OR ROOT = NULL

Return ROOT

ELSE

IF ROOT < ITEM -> DATA

Return search (ROOT -> LEFT, ITEM)

ELSE

Return search (ROOT -> RIGHT, ITEM)

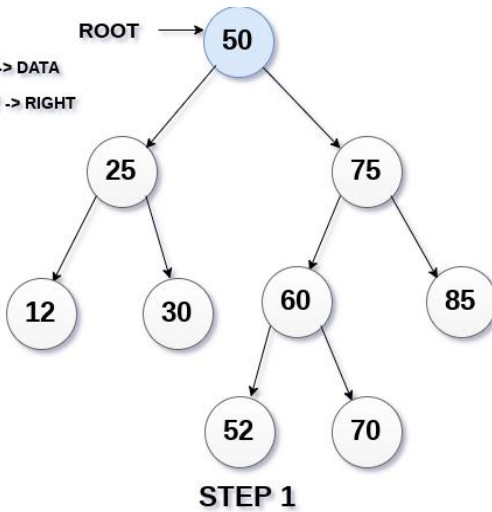
[END OF IF]

[END OF IF]

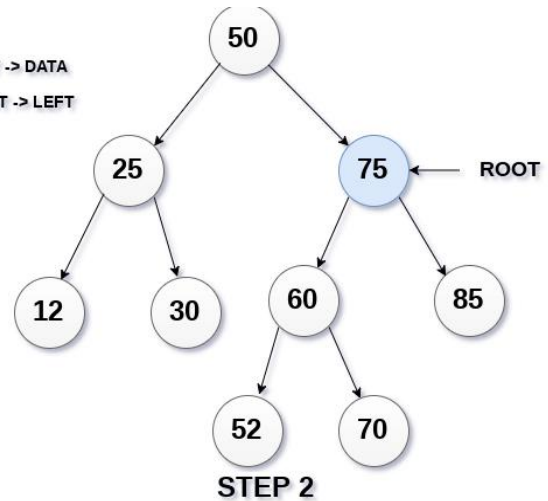
Step 2: END

### Example

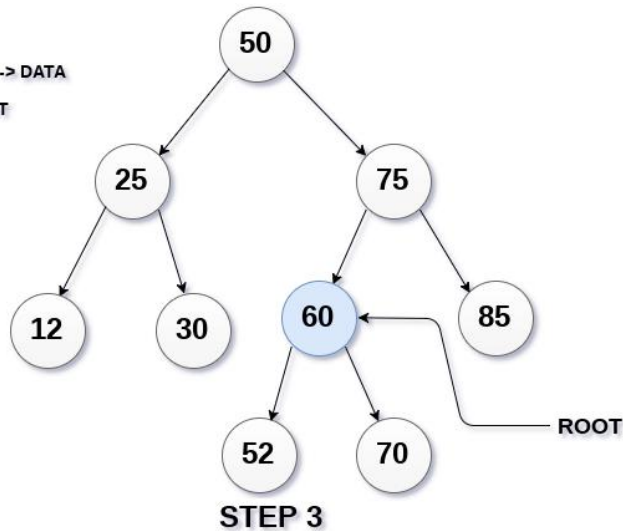
Item = 60  
ITEM > ROOT -> DATA  
ROOT = ROOT -> RIGHT



Item = 60  
ITEM < ROOT -> DATA  
ROOT = ROOT -> LEFT



Item = 60  
ITEM = ROOT -> DATA  
RETURN ROOT



### Insertion

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must not violate the property of binary search tree at each value.

- ✓ Allocate the memory for tree.
- ✓ Set the data part to the value and set the left and right pointer of tree, point to NULL.
- ✓ If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
- ✓ Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
- ✓ If this is false, then perform this operation recursively with the right sub-tree of the root.

### Algorithm for Insert (TREE, ITEM)

Step 1: IF TREE = NULL

```

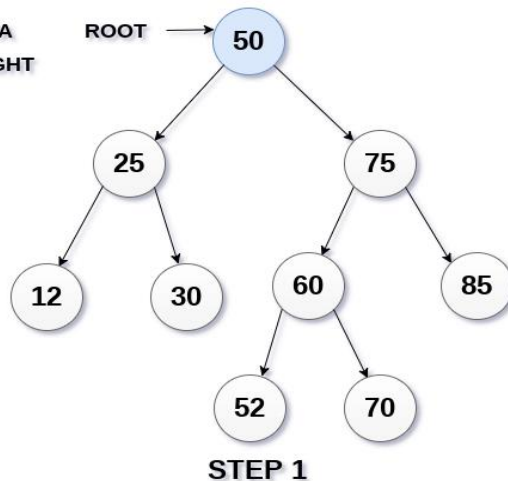
Allocate memory for TREE
SET TREE -> DATA = ITEM
SET TREE -> LEFT = TREE -> RIGHT = NULL
ELSE
 IF ITEM < TREE -> DATA
 Insert (TREE -> LEFT, ITEM)
 ELSE
 Insert (TREE -> RIGHT, ITEM)
 [END OF IF]
[END OF IF]

```

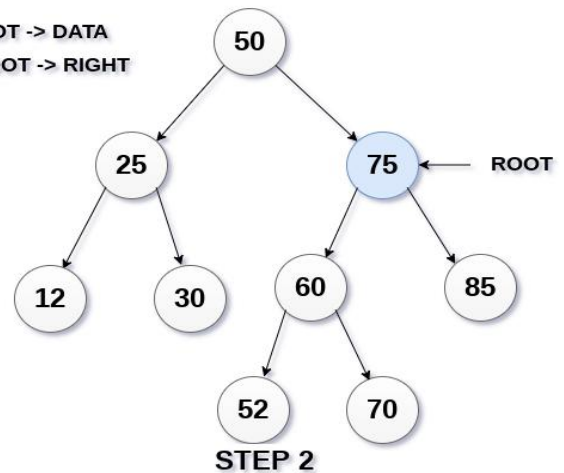
Step 2: END

ITEM=95

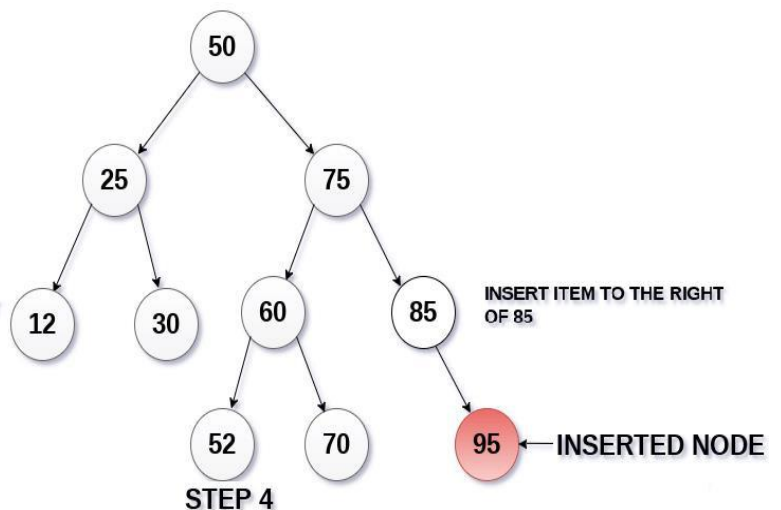
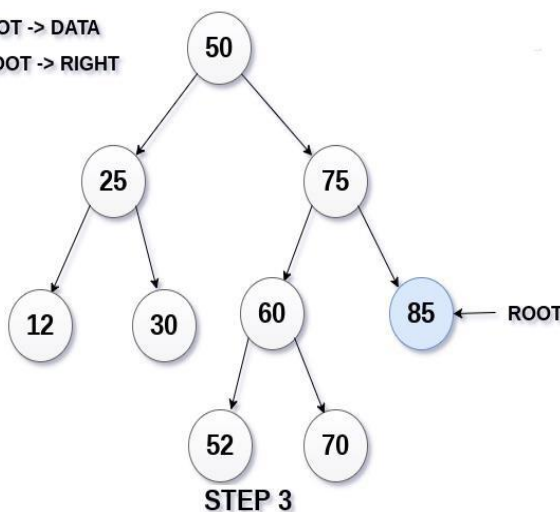
ITEM > ROOT -> DATA  
ROOT = ROOT -> RIGHT



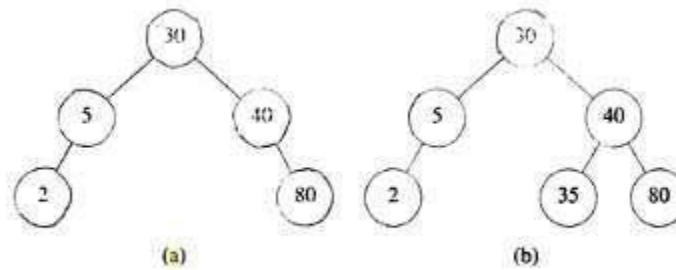
ITEM > ROOT -> DATA  
ROOT = ROOT -> RIGHT



ITEM > ROOT -> DATA  
ROOT = ROOT -> RIGHT



- 1) We must first verify if the tree already contains the node with the same data (Figure 5.29 & Program 5.15).
- 2) If the search is successful, then the new node cannot be inserted into the binary search tree.
- 3) If the search is unsuccessful, then we can insert the new node at that point where the search terminated.



**Figure 5.29: Inserting into a binary search tree**

```

void insert_node(tree_pointer *node, int num)
{
 tree_pointer ptr,
 temp = modified_search(*node, num);
 if (temp || !(*node))
 {
 ptr = (tree_pointer) malloc(sizeof(node));
 if (IS_FULL(ptr))
 {
 fprintf(stderr, "The memory is full\n");
 exit(1);
 }
 ptr->data = num;
 ptr->left_child = ptr->right_child = NULL;
 if (*node)
 if (num < temp->data)
 temp->left_child = ptr;
 else
 temp->right_child = ptr;
 else
 *node = ptr;
 }
}

```

*Program: Insertion into a binary search tree*

## Deletion

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

- 1) Deletion of a leaf: To delete 35 from the tree of figure 5.29b, the left-child field of its parent is set to NULL.

- 2) Deletion of a non-leaf that has only one child: The node containing the dictionary pair to be deleted is freed, and its single-child takes the place of the freed node. So, to delete the 5 from the tree in figure 5.29a, we simply change the pointer from the parent node to the single-child node.
- 3) The pair to be deleted is in a non-leaf node that has two children: The pair to be deleted is replaced by either the largest pair in its left subtree or the smallest one in its right subtree. For instance, if we wish to delete the pair with key 30 from the tree in figure 5.29b, then we replace it by key 5 as shown in figure.

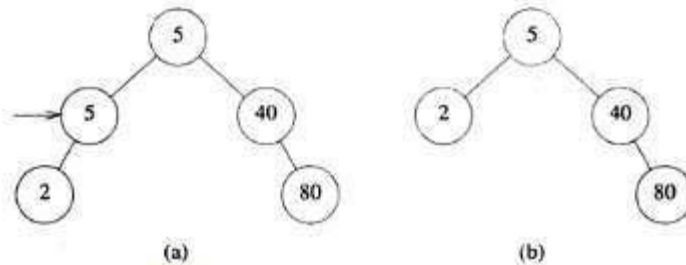
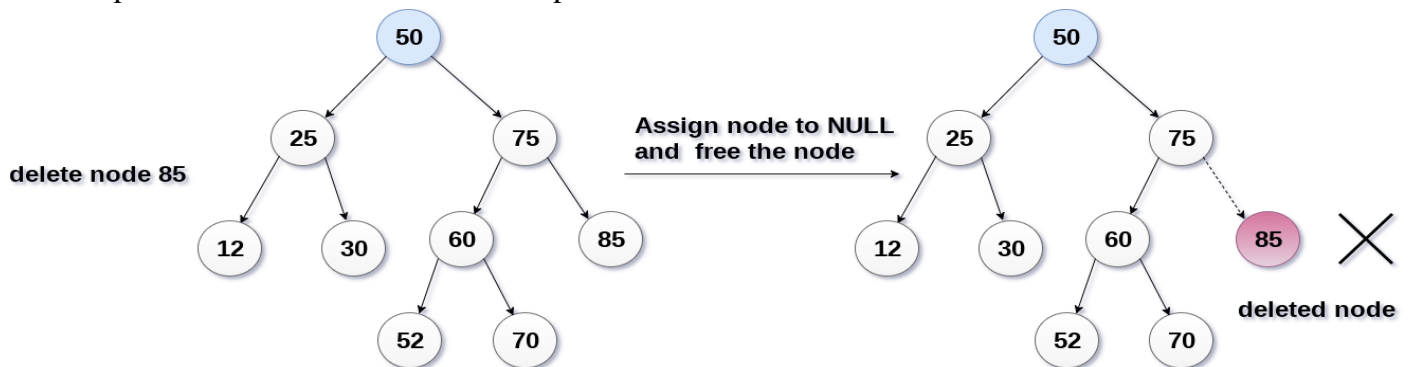


Figure 5.30: Deletion from a binary search tree

### The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simply free the allocated space.

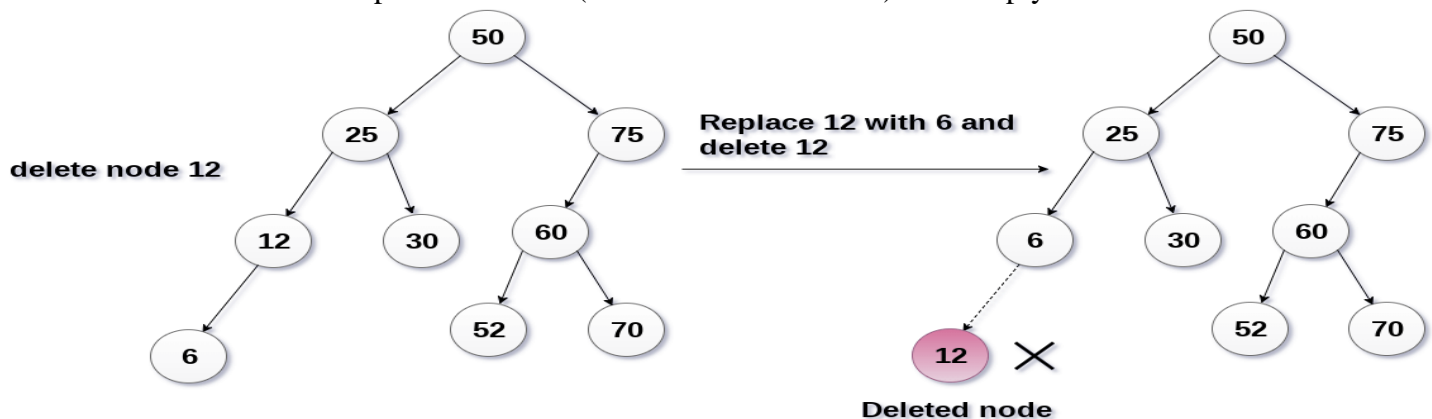
In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



### The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following Example, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



### The node to be deleted has two children.

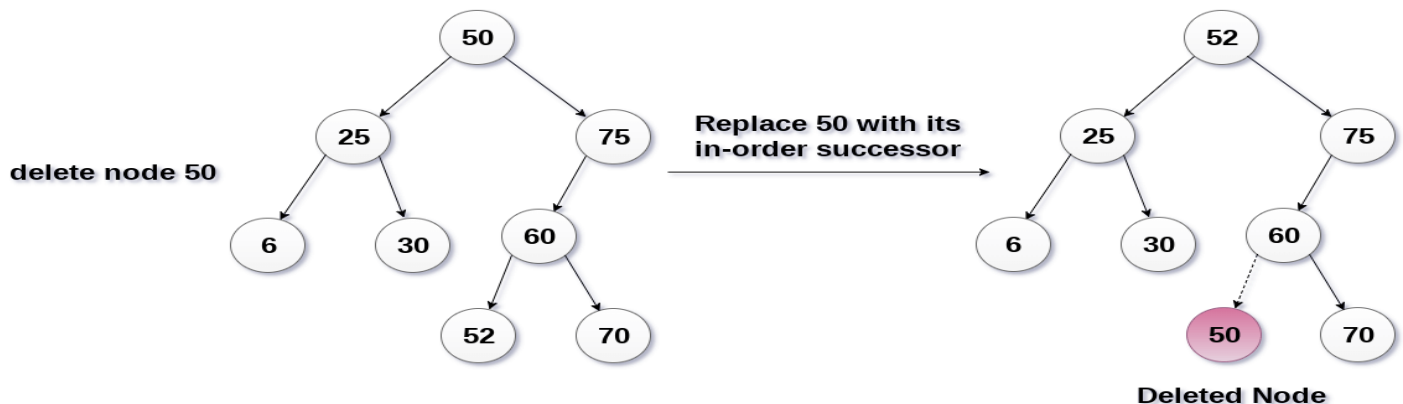
It is a bit complex case compare to other two cases. However, the node which is to be deleted is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following Example, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below

**6, 25, 30, 50, 52, 60, 70, 75**

Replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.

### LEFT>VECTOR (ROOT)>RIGHT



### Algorithm for Delete (TREE, ITEM)

Step 1: IF TREE = NULL

Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA

Delete (TREE->LEFT, ITEM)

ELSE IF ITEM > TREE -> DATA

Delete (TREE -> RIGHT, ITEM)

ELSE IF TREE -> LEFT AND TREE -> RIGHT

SET TEMP = find Largest Node (TREE -> LEFT)

SET TREE -> DATA = TEMP -> DATA

Delete (TREE -> LEFT, TEMP -> DATA)

ELSE

SET TEMP = TREE

IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL

SET TREE = NULL

ELSE IF TREE -> LEFT! = NULL

SET TREE = TREE -> LEFT

ELSE

SET TREE = TREE -> RIGHT

[END OF IF]

FREE TEMP

[END OF IF]

Step 2: END

## JOINING BINARY TREE

- There are two types of join operation on a binary search tree:

1) **threeWayJoin(small,mid,big)**: We simply create a new node and set its data field to mid, its left-child pointer to small and its right-child pointer to big.

2) **twoWayJoin(small,big)**:

i) If small is empty, then the result is big.

ii) If big is empty, then the result is small.

iii) If both are non-empty, then we have to first delete from 'small' the pair mid with the largest key.

After this, a 3-way join of small, mid and big must be performed.

## SPLITTING BINARY TREE

- Splitting a binary search tree will produce three trees: small, mid and big.

1) If key is equal to root->data, then root->llink is the small, root->data is mid & root->rlink is big.

2) If key is lesser than the root->data, then the root's along with its right subtree would be in the big.

3) if key is greater than root->data, then the root's along with its left subtree would be in the small

## AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honor of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1, 0, 1, otherwise, the tree will be unbalanced and need to be balanced

### Definition:

An empty tree is height balanced. If T is a nonempty binary tree with  $T_L$  and  $T_R$  as its left and right subtrees respectively, then T is height-balanced iff (1)  $T_L$  and  $T_R$  are height balanced and (2)  $|h_L - h_R| \leq 1$  where  $h_L$  and  $h_R$  are the heights of  $T_L$  and  $T_R$ , respectively.

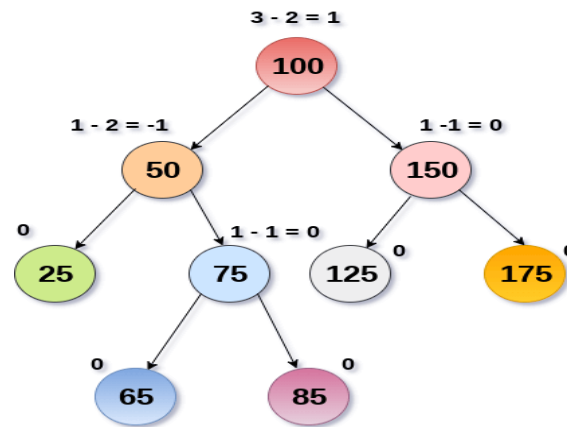
**Balance Factor (k) = height (left (k)) - height (right (k))**

### Definition:

The Balance factor,  $BF(T)$ , of a node T in a binary tree is defined to be  $h_L - h_R$ , where  $h_L$  and  $h_R$ , respectively, are the heights of the left and right subtrees of T. For any node T in an AVL tree,  $BF = -1, 0$ , or  $1$ .

- ✓ If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- ✓ If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- ✓ If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.
- ✓ An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1, 0, and +1.





**AVL Tree**

### AVL Tree Properties

- ✓ Maximum possible number of nodes in AVL tree of height  $H = 2^{H+1} - 1$
- ✓ Minimum number of nodes in AVL Tree of height  $H$  is given by a recursive relation-  $N(H) = N(H-1) + N(H-2) + 1$
- ✓ Minimum possible height of AVL Tree using  $N$  nodes =  $\lfloor \log_2 N \rfloor$
- ✓ Maximum height of AVL Tree using  $N$  nodes is calculated using recursive relation-  $N(H) = N(H-1) + N(H-2) + 1$

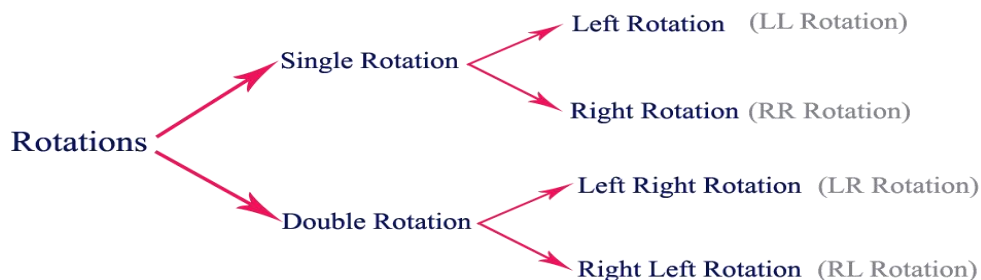
### NOTE

- If there are  $n$  nodes in AVL Tree, its maximum height can not exceed  $1.44 \log_2 n$ .
- In other words, Worst case height of AVL Tree with  $n$  nodes =  $1.44 \log_2 n$ .

### Complexity for AVL Tree

| Algorithm | Average case | Worst case  |
|-----------|--------------|-------------|
| Space     | $O(n)$       | $O(n)$      |
| Search    | $O(\log n)$  | $O(\log n)$ |
| Insert    | $O(\log n)$  | $O(\log n)$ |
| Delete    | $O(\log n)$  | $O(\log n)$ |

### Rotations



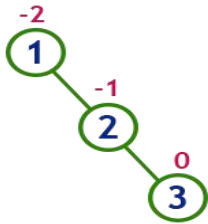
| SN | Rotation    | Description                                                                                |
|----|-------------|--------------------------------------------------------------------------------------------|
| 1  | LL Rotation | The new node is inserted to the left sub-tree of left sub-tree of critical node.           |
| 2  | RR Rotation | The new node is inserted to the right sub-tree of the right sub-tree of the critical node. |
| 3  | LR Rotation | The new node is inserted to the right sub-tree of the left sub-tree of the critical node.  |
| 4  | RL Rotation | The new node is inserted to the left sub-tree of the right sub-tree of the critical node.  |



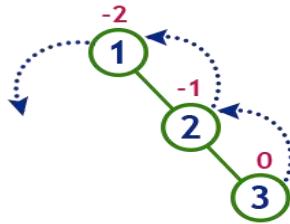
### Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

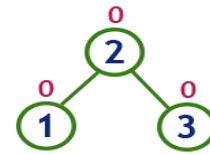
insert 1, 2 and 3



Tree is imbalanced



To make balanced we use LL Rotation which moves nodes one position to left

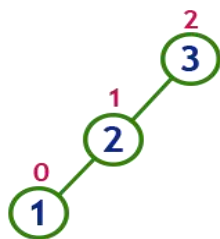


After LL Rotation Tree is Balanced

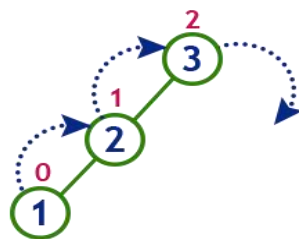
### Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

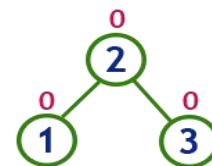
insert 3, 2 and 1



Tree is imbalanced  
because node 3 has balance factor 2



To make balanced we use RR Rotation which moves nodes one position to right

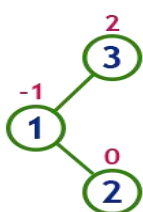


After RR Rotation Tree is Balanced

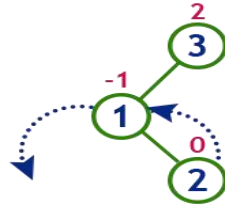
### Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...

insert 3, 1 and 2

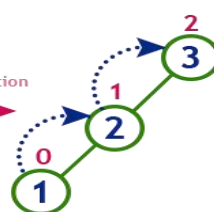


Tree is imbalanced  
because node 3 has balance factor 2



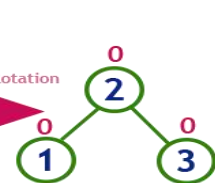
LL Rotation

After LL Rotation



RR Rotation

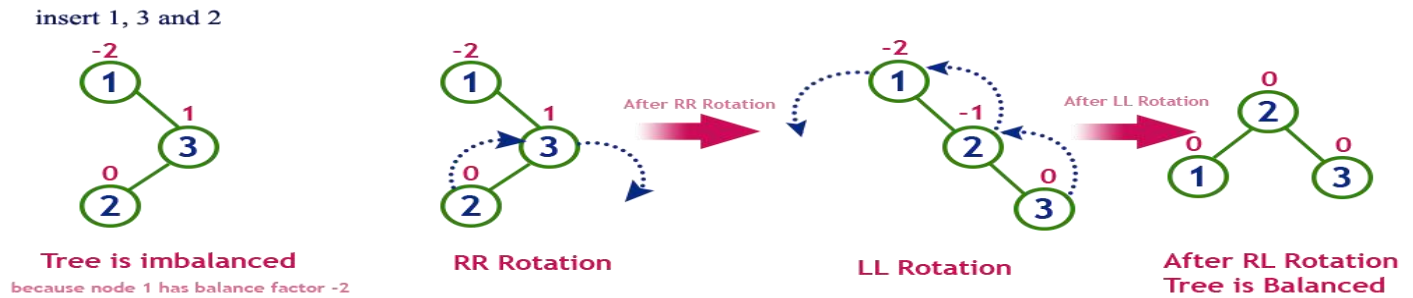
After RR Rotation



After LR Rotation Tree is Balanced

### Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



## Operations on an AVL Tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree.

However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

- ✓ Searching
- ✓ Insertion
- ✓ Deletion

## Search Operation in AVL Tree

In an AVL tree, the search operation is performed with  $O(\log n)$  time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the value of root node in the tree.
- Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
- Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
- Step 5 - If search element is smaller, then continue the search process in left subtree.
- Step 6 - If search element is larger, then continue the search process in right subtree.
- Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

## Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with  $O(\log n)$  time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2 - After insertion, check the Balance Factor of every node.

Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

## Construct an AVL Tree by inserting numbers from 1 to 8

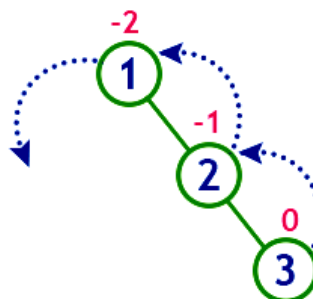
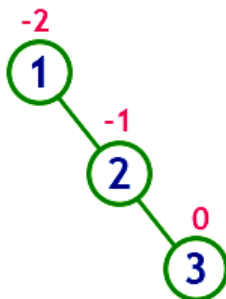
insert 1



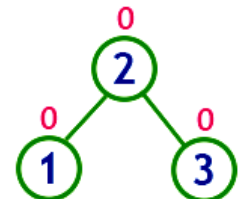
insert 2



insert 3



After LL Rotation



Tree is imbalanced

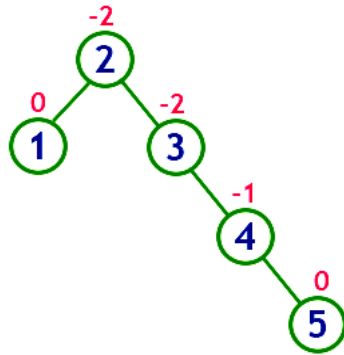
LL Rotation

Tree is balanced

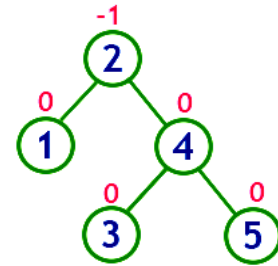
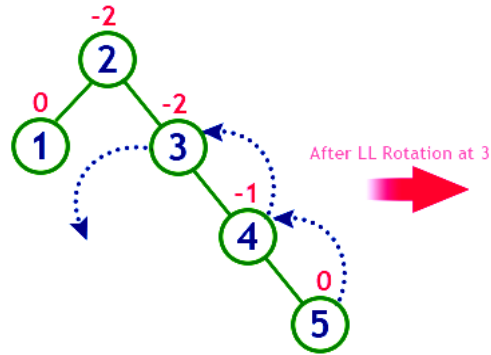
insert 4



insert 5

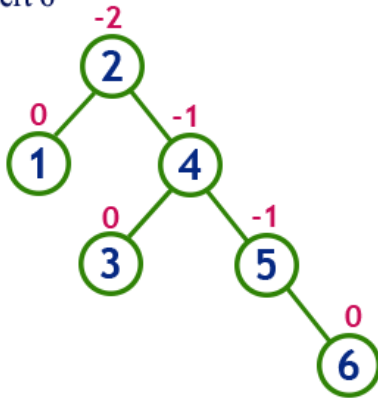


Tree is imbalanced

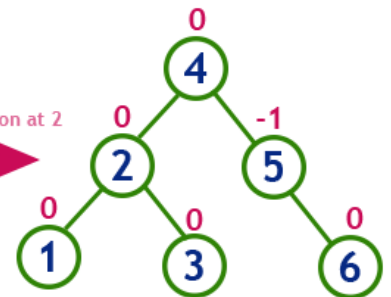
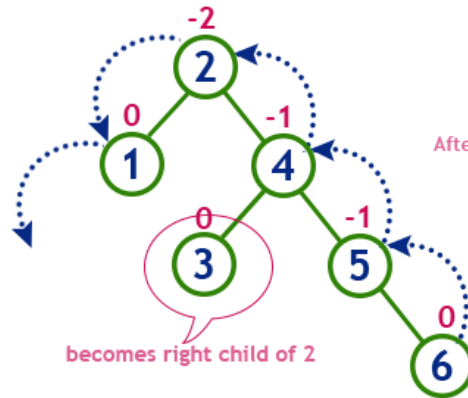


Tree is balanced

insert 6

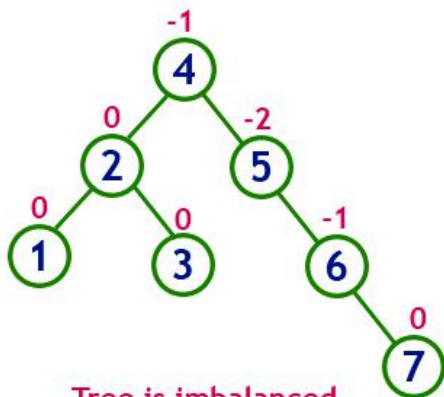


Tree is imbalanced

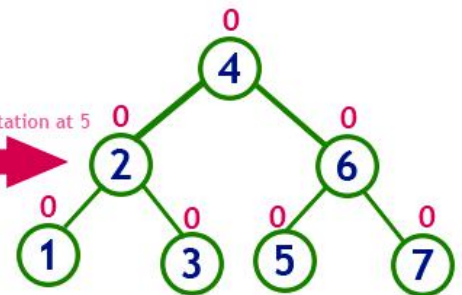
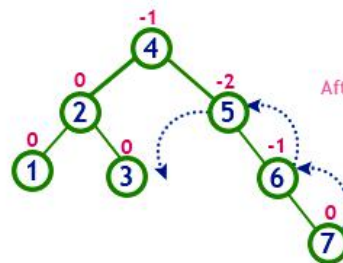


Tree is balanced

insert 7

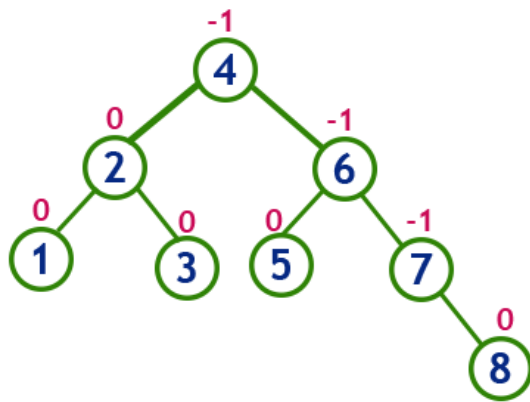


Tree is imbalanced



Tree is balanced

insert 8



Tree is balanced

### Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

### B - Tree Data structure

In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children. B-Tree was developed in the year 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.

### B-Tree defined as

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order

### B-Tree of Order 'm' properties:

#### Formal Definition:

A B-Tree of order m is an m-way search tree that either is empty or satisfies the following properties:

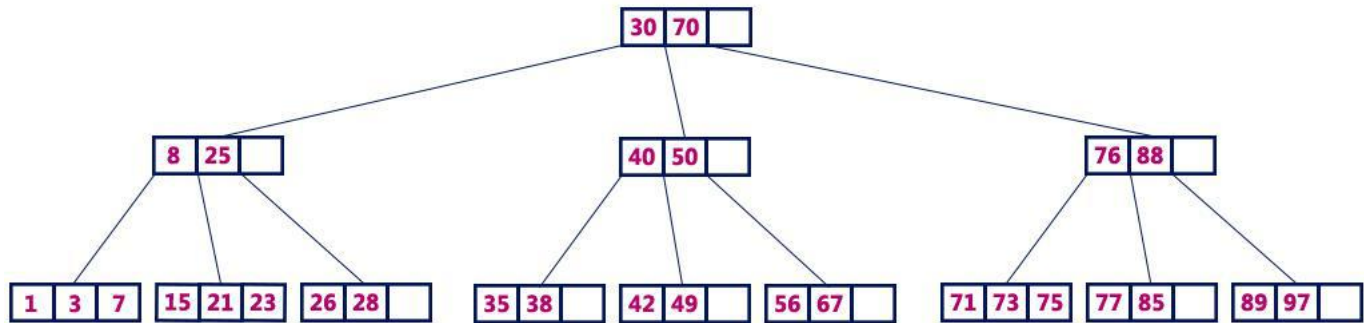
- 1) The root node has at least two children.
- 2) All nodes other than the root node and external nodes have at least  $\lceil m/2 \rceil$  children.
- 3) All external nodes are at the same level.

- ✓ Property #1 - All leaf nodes must be at same level.
- ✓ Property #2 - All nodes except root must have at least  $\lceil m/2 \rceil - 1$  keys and maximum of m-1 keys.
- ✓ Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least  $m/2$  children.
- ✓ Property #4 - If the root node is a non leaf node, then it must have at least 2 children.
- ✓ Property #5 - A non leaf node with n-1 keys must have n number of children.
- ✓ Property #6 - All the key values in a node must be in Ascending Order.

Observe that when  $m=3$ , all internal nodes of a B-Tree have a degree that is either 2 or 3 and when  $m=4$ , the permissible degree for these nodes are 2, 3 and 4. For this reason, a B-Tree of order 3 is known as a 2-3 tree and a B-Tree of order 4 is known as a 2-3-4 tree.

**For example, B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.**

B-Tree of Order 4



### Operations on a B-Tree

- ✓ Search
- ✓ Insertion
- ✓ Deletion

### Search Operation in B-Tree

The search operation in B-Tree is similar to the search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left sub tree or right sub tree). In B-Tree also search process starts from the root node but here we make an  $n$ -way decision every time. Where ' $n$ ' is the total number of children the node has. In a B-Tree, the search operation is performed with  $O(\log n)$  time complexity.

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with first key value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeated steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

### Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only.

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

Step 5 - If that leaf node is already full, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

### Example

Construct a B-Tree of Order 3 by inserting numbers from 1 to 10.

#### insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.



#### insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



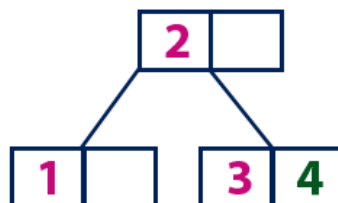
#### insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't have an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't have a parent. So, this middle value becomes a new root node for the tree.



#### insert(4)

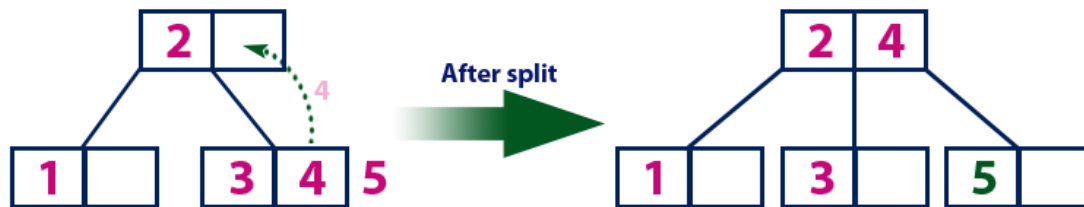
Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.





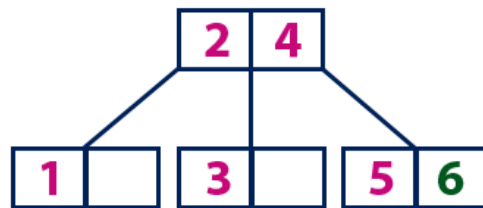
#### insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



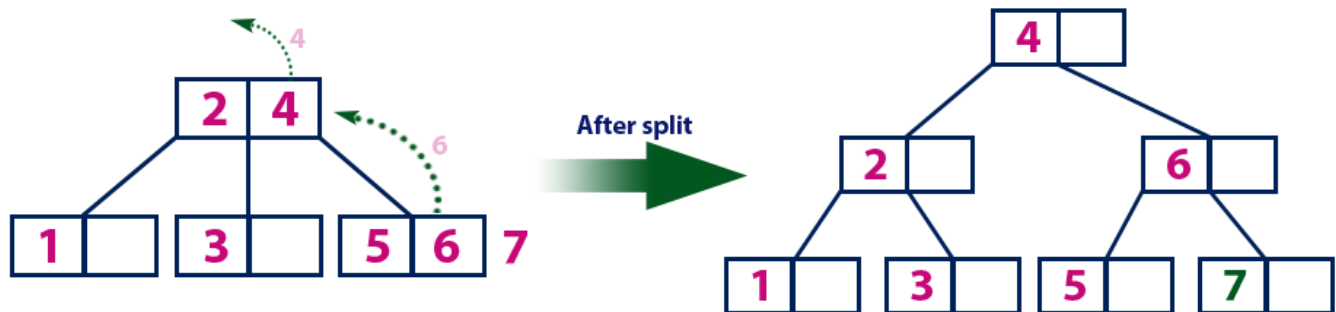
#### insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



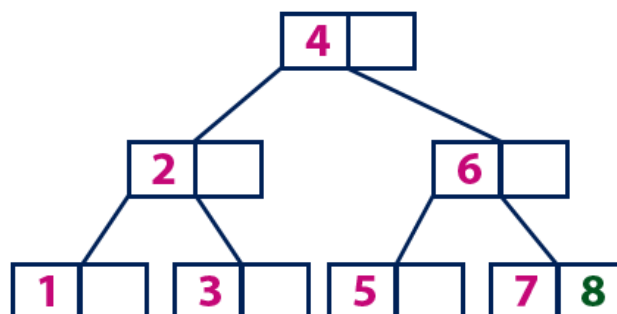
#### insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



#### insert(8)

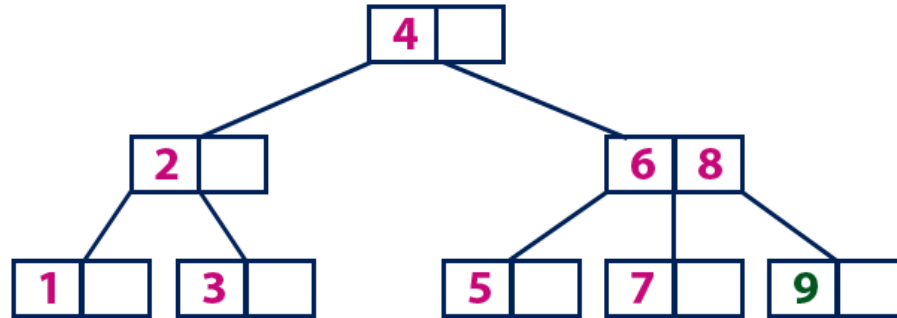
Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.





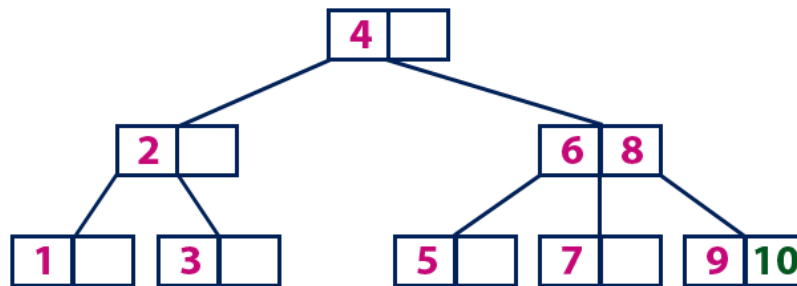
### insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



### insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8'. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.

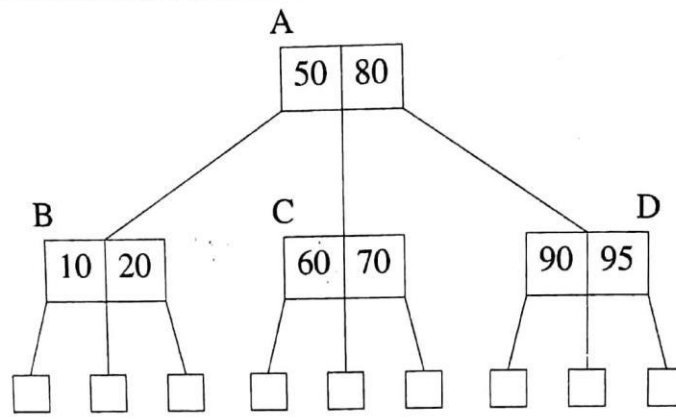


### Deletion:

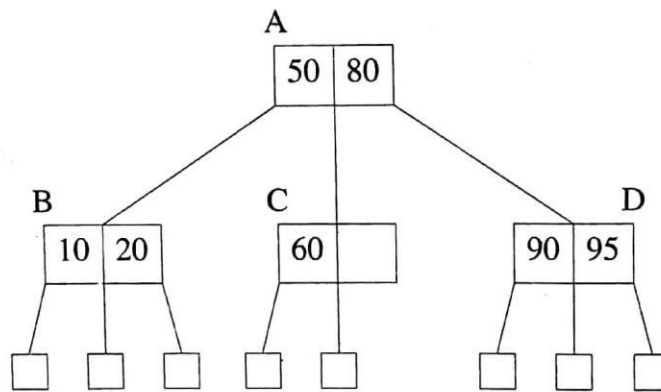
### Example:

**Example 11.2:** Let us begin with the 2-3 tree of Figure 11.9(a). Suppose that the two element fields in a node of a 2-3 tree are called *dataL* and *dataR*. To delete the element with key 70, we must merely delete this element from node C. The result is shown in Figure 11.9(b). To delete the element with key 10 from the 2-3 tree of Figure 11.9(b), we need to shift *dataR* to *dataL* in node B. This results in the 2-3 tree of Figure 11.9(c).

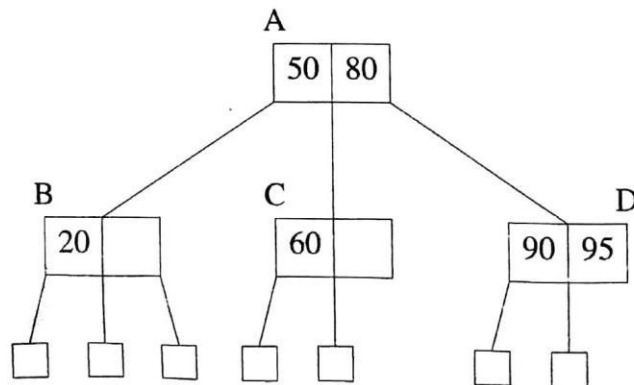
Next consider the deletion of the element with key 60. This leaves node C deficient. Since the right sibling, D, of C has 3 elements, we are in case 3 and a rotation



(a) Initial 2-3 tree

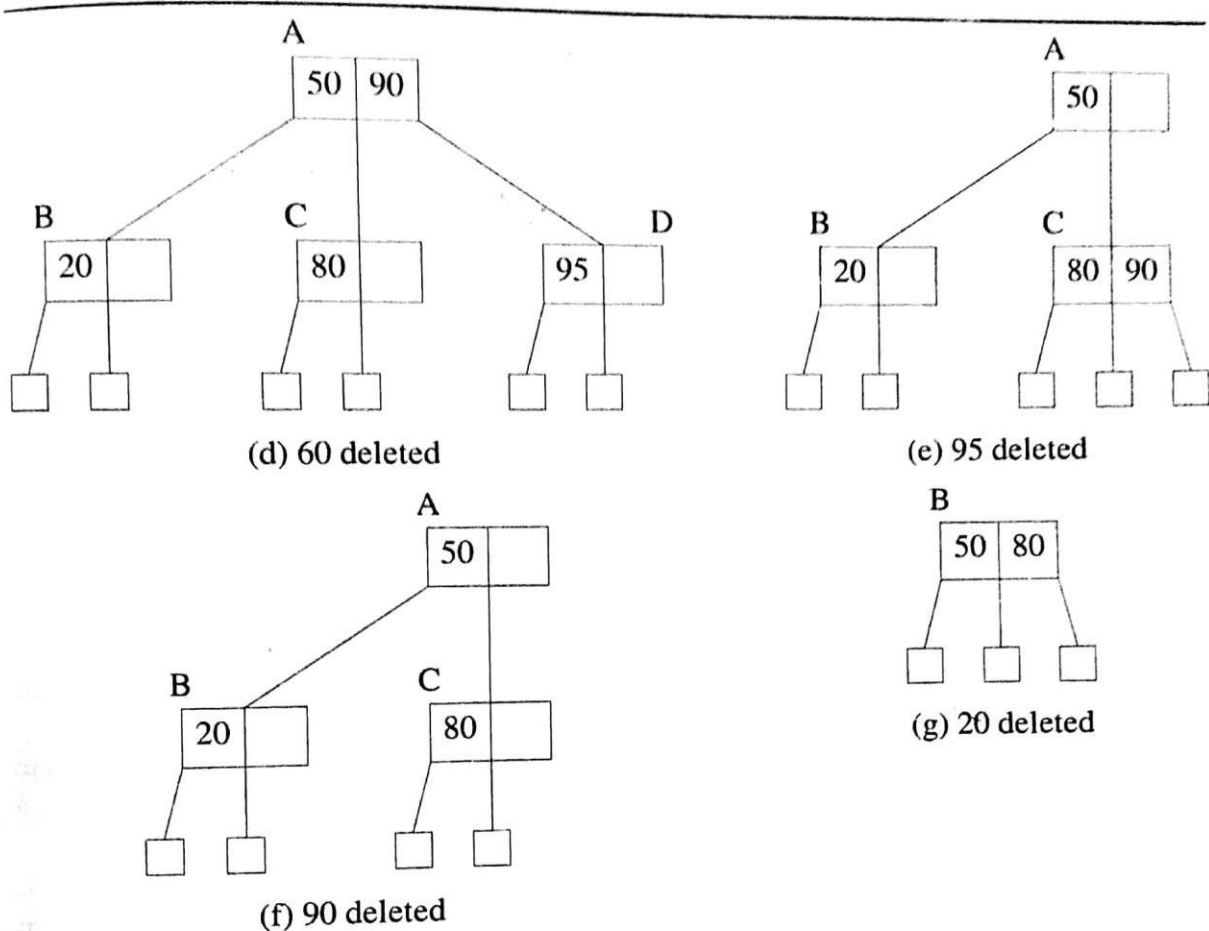


(b) 70 deleted



(c) 10 deleted

**Figure 11.9:** Deletion from a 2-3 tree (continued on next page)



**Figure 11.9:** Deletion from a 2-3 tree

is performed. In this rotation, we move the the in-between element (i.e., the element whose key is 80) of the parent A of C and D to the *dataL* position of C and move the smallest element of D (i.e., the element whose key is 20) into the in-between position of the parent A of C and D (i.e., the *dataR* position of A). The resulting 2-3 tree takes the form shown in Figure 11.9(d). When the element with key 95 is deleted, node D becomes deficient. The rotation performed when the 60 was deleted is not possible now, as the left sibling, C, has the minimum number of elements required by a node in a B-tree of order 3. We now are in case 4 and must combine nodes C and D and the in-between element (90) in the parent A of C and D. For this, we move the 90 into the left sibling, C, and delete node D. Notice that in a combine, one node is deleted, whereas in a rotation, no node is deleted. The deletion of 95 results in the 2-3 tree of Figure 11.9(e). Deleting the element with key 90 from this tree results in the 2-3 tree of Figure 11.9(f). Now consider deleting the element with key 20 from this tree. Node B becomes

deficient. At this time, we examine B's right sibling, C. If C has excess elements, we can perform a rotation similar to that done during the deletion of 60. Otherwise, a combine is performed. Since C doesn't have excess elements, we proceed in a manner similar to the deletion of 95 and do a combine. This time the elements with keys 50 and 80 are moved into B, and node C is deleted. This, however, causes the parent node A to become deficient. If the parent had not been a root, we would examine its left or right sibling, as we did when nodes C (deletion of 60) and D (deletion of 95) became empty. Since A is the root, it is simply deleted, and B becomes the new root (Figure 11.9(g)). Recall that a root is deficient iff it has no element.  $\square$

**Analysis of B-tree Deletion:** Once again, we assume a disk-resident B-tree and that disk nodes accessed during the downward search pass may be saved in a stack in main memory, so they do not need to be reaccessed from the disk during the upward restructuring pass. For a B-tree of height  $h$ ,  $h$  disk accesses are made to find the node from which the key is to be deleted and to transform the deletion to a deletion from a leaf. In the worst case, a combine takes place at each of the last  $h - 2$  nodes on the root-to-leaf path, and a rotation takes place at the second node on this path. The  $h - 2$  combines require this many disk accesses to retrieve a nearest sibling for each node and another  $h - 2$  accesses to write out the combined nodes. The rotation requires one access to read a nearest sibling and three to write out the three nodes that are changed. The total number of disk accesses is  $3h$ .

The deletion time can be reduced at the expense of disk space and a slight increase in node size by including a delete bit,  $F_i$ , for each element,  $E_i$ , in a node. Then we can set  $F_i = 1$  if  $E_i$  has not been deleted and  $F_i = 0$  if it has. No physical deletion takes place. In this case a delete requires a maximum of  $h + 1$  accesses ( $h$  to locate the node containing the element to be deleted and 1 to write out this node with the appropriate delete bit set to 0). With this strategy, the number of nodes in the tree never decreases. However, the space used by deleted entries can be reused during further insertions (see Exercises). As a result, this strategy has little effect on search and insert times (the number of levels increases very slowly when  $m$  is large). The time taken to insert an item may even decrease slightly because of the ability to reuse deleted element space. Such reuse would not require us to split any nodes.  $\square$



Scanned with CamScanner



### B<sup>+</sup> - Trees:

A B<sup>+</sup> - tree is distinguished from B - trees as:

1) In a B<sup>+</sup> - tree we have two types of nodes – index and data. The index nodes of B<sup>+</sup> - tree correspond to the internal nodes of a B-tree while the data nodes correspond to external nodes. The index nodes store keys and pointers and the data nodes store elements (together with their keys but no pointers).

2) The data nodes are linked together, in left to right order, to form a doubly linked list.

Following fig gives an example of B<sup>+</sup> - tree with order of 3. The data nodes are shaded while the index nodes not. In figure data node can hold 3 elements while each index node can hold 2 keys.

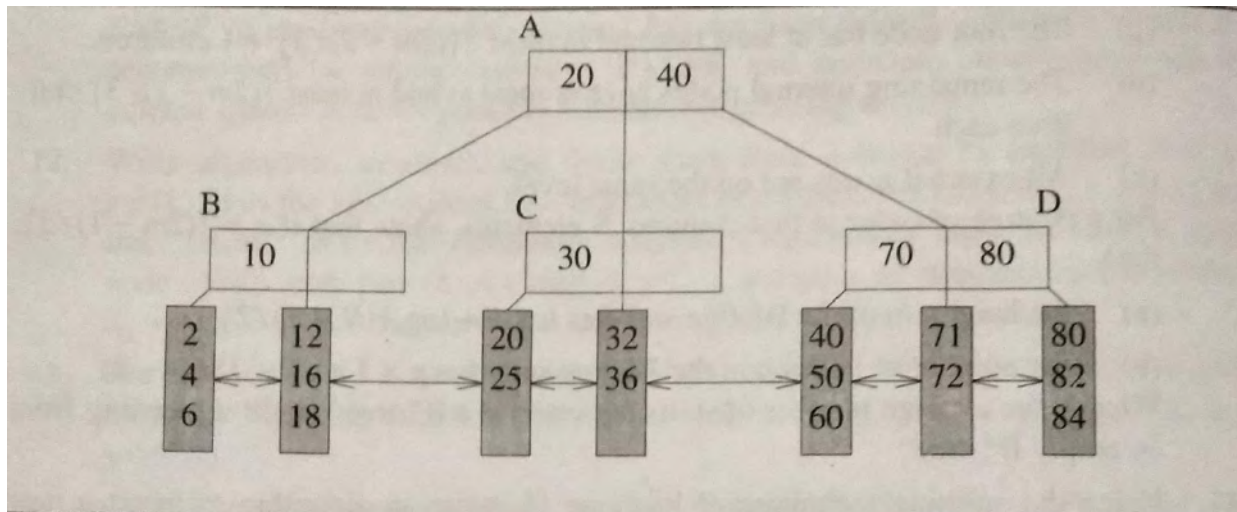


Fig11.11: A B<sup>+</sup> - tree of order 3

#### Definition:

A B<sup>+</sup> - tree of order m is a tree that either is empty or satisfies the following properties:

- 1) All the data nodes are at the same level and are leaves. Data nodes contain elements only.
- 2) The index nodes define a B-tree of order m; each index node has keys but not elements.
- 3) Let

$$n, A_0, (K_1, A_1), (K_2, A_2), \dots, (K_n, A_n)$$

Where  $A_i$ ,  $0 \leq i \leq n$ , are pointers to subtrees, and the  $K_i$ ,  $1 \leq i \leq n$ , are keys to be the format of some index node. Let  $K_0 = -\infty$  and  $K_{n+1} = \infty$ . All the elements in the subtree  $A_i$  have key less than  $K_{i+1}$  and greater than or equal to  $K_i$ ,  $0 \leq i \leq n$ .

#### Searching a B<sup>+</sup> - tree:

B<sup>+</sup> - tree support two types of searches –

- Exact match and
- Range

For example in above figure we can search for an element whose key is 32, we begin at the root A, which is an index node. From the definition of B<sup>+</sup> - tree we know that all elements in the left subtree of A have a key smaller than 20; those in the subtree with root C have keys  $\geq 20$  and  $\leq 40$ ; and those in the subtree with root D have keys  $\geq 40$ . So, the search moves to the index node C. this is a procedure of exact search.

To search for all elements with keys in the range [16, 70], we proceed as in an exact match search for the start, 16, of the range. In our example 4 additional data nodes are examined.

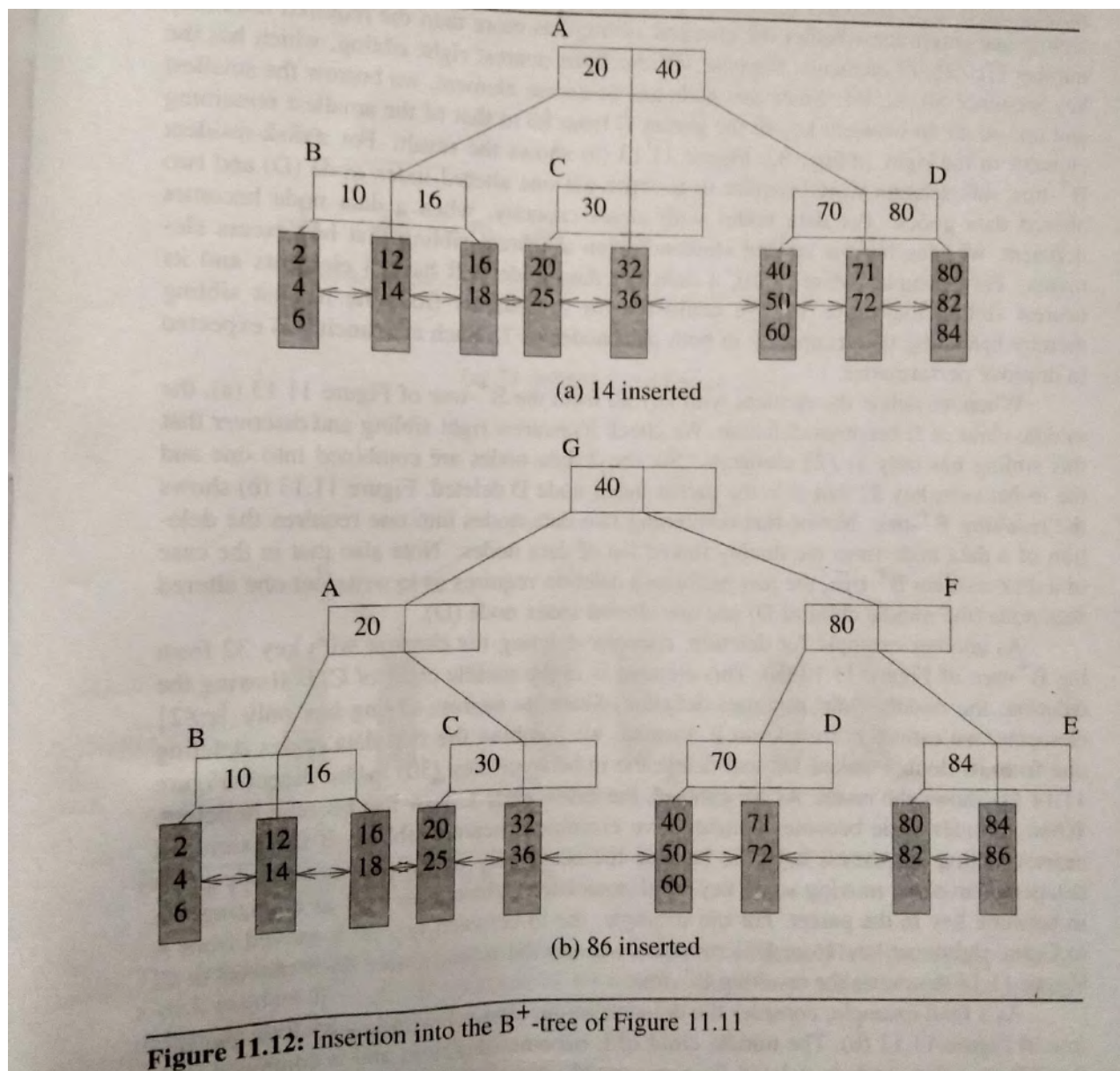
### Insertion into a B<sup>+</sup> - tree:

In insertion when a data node becomes overfull, half the elements are moved into a new node; the key of the smallest element so moved together with a pointer to the newly created data node are inserted into the parent index node using the insertion procedure for a B-tree. The splitting of an index node is identical to splitting of an internal node of a B-tree.

Consider inserting an element with key 27 into the B<sup>+</sup> - tree of above fig. we first search for this key. The search gets us to the data node that is the left child of C. since this data node contains no element with key 27 and since this data node is not full, we insert the new element as the third element in this data node.

Next, consider the insertion of an element with key 14. The search for 14 gets us to the second data node, which is full. Symbolically inserting the new element into this full node results in an overfull node with the key sequence 12, 14, 16, 18. The overfull node split into two by moving the largest half of elements into a new data node, which is then inserted into the doubly linked list of data nodes. The smallest key, 16, in this new data node together with a pointer to the new data node are inserted in the parent index node B to get the configuration of below fig.

Following fig shows the insertion of element with keys 14 and 86.



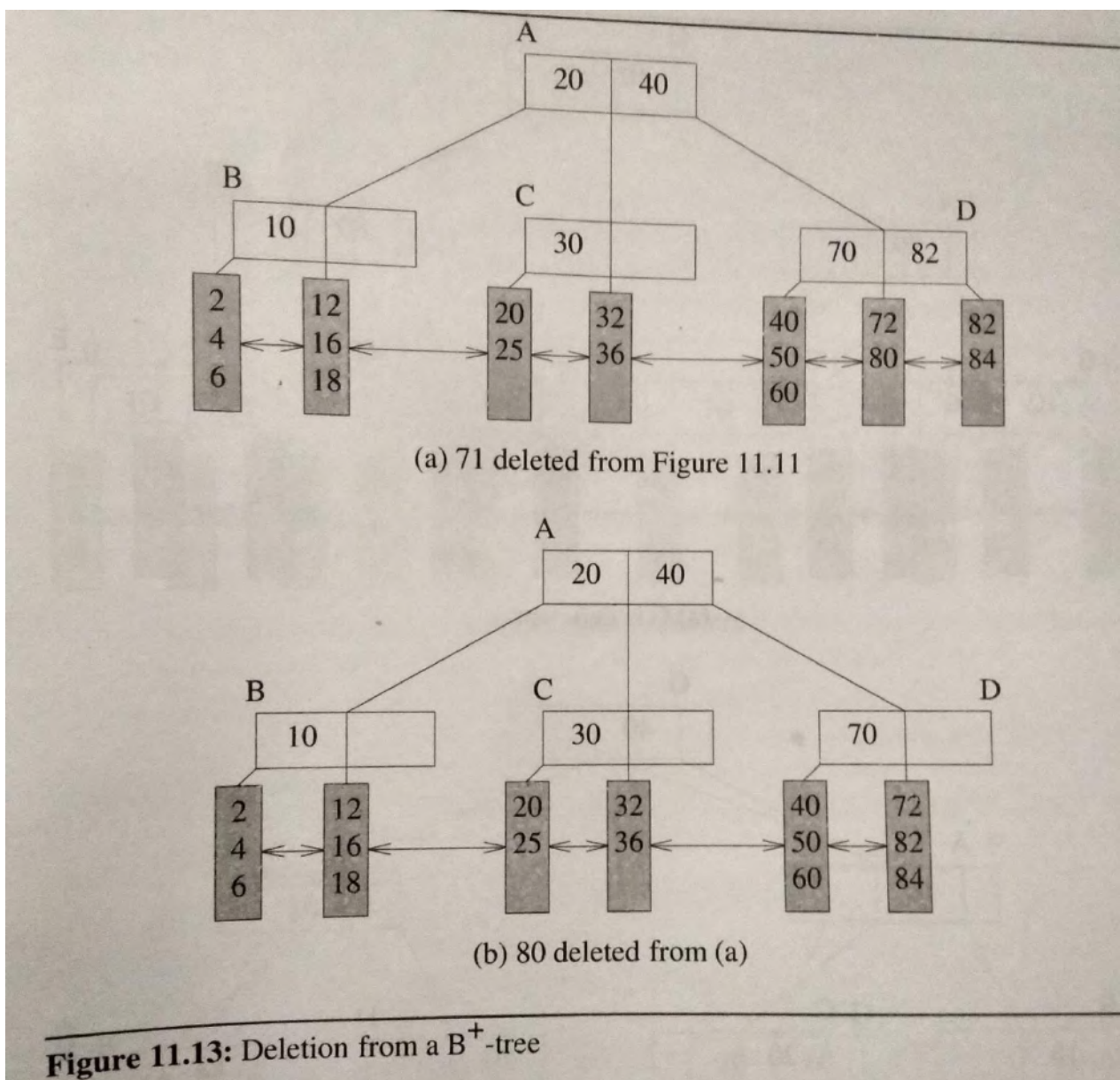
### Deletion from a B<sup>+</sup> - tree:

The definition of a B<sup>+</sup> - tree does not specify a minimum occupancy for a data node. Following the split of an overfull data node as well as the new each have at least  $\lceil c/2 \rceil$  elements, where  $c$  is the capacity of a data node. So, except when a data node is the root of the B<sup>+</sup> - tree, its occupancy is at least  $\lceil c/2 \rceil$ . We shall say that a non-root node is deficient iff it has fewer than  $\lceil c/2 \rceil$  elements; a root data node is deficient iff it is empty.

To delete an element with key 40, there will be no problem since it would not get deficient.

Next consider an element with key 71, node will get deficient at that time *we check either it's nearest right or nearest left sibling and determine whether the checked sibling has more than the required minimum number  $\lceil c/2 \rceil$* . Suppose we check the nearest right sibling, which has the key sequence 80, 82, 84. Since the node has an excess element, we borrow the smallest and update the in-between key in the parent D from 80 to that of the smallest remaining element in the right sibling, 82 as in fig 11.13(a).

Another fig shows deletion of an element with key 80.



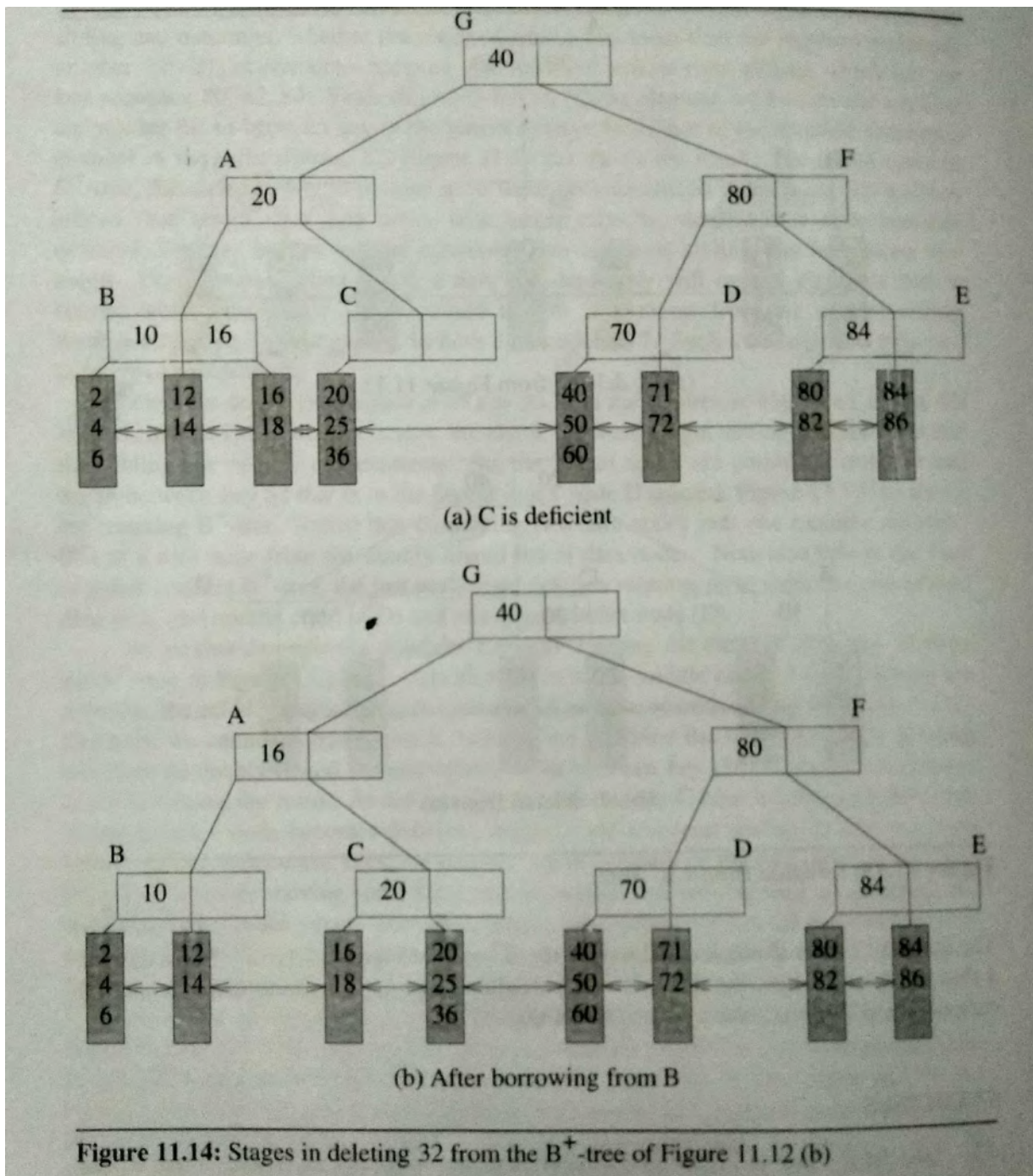
**Figure 11.13:** Deletion from a B<sup>+</sup>-tree



As an example for deletion, consider deleting the element with key 32 from the  $B^+$  - tree of fig 11.12(b). This element is the middle child of C. Following the deletion middle child becomes deficient. Since its nearest sibling has only  $\lceil c/2 \rceil$  elements we cannot borrow from it. Instead, we combine the two data nodes deleting one from its doubly linked list and delete the in-between key (30) in the parent. As we can see, the index node C now has become deficient.

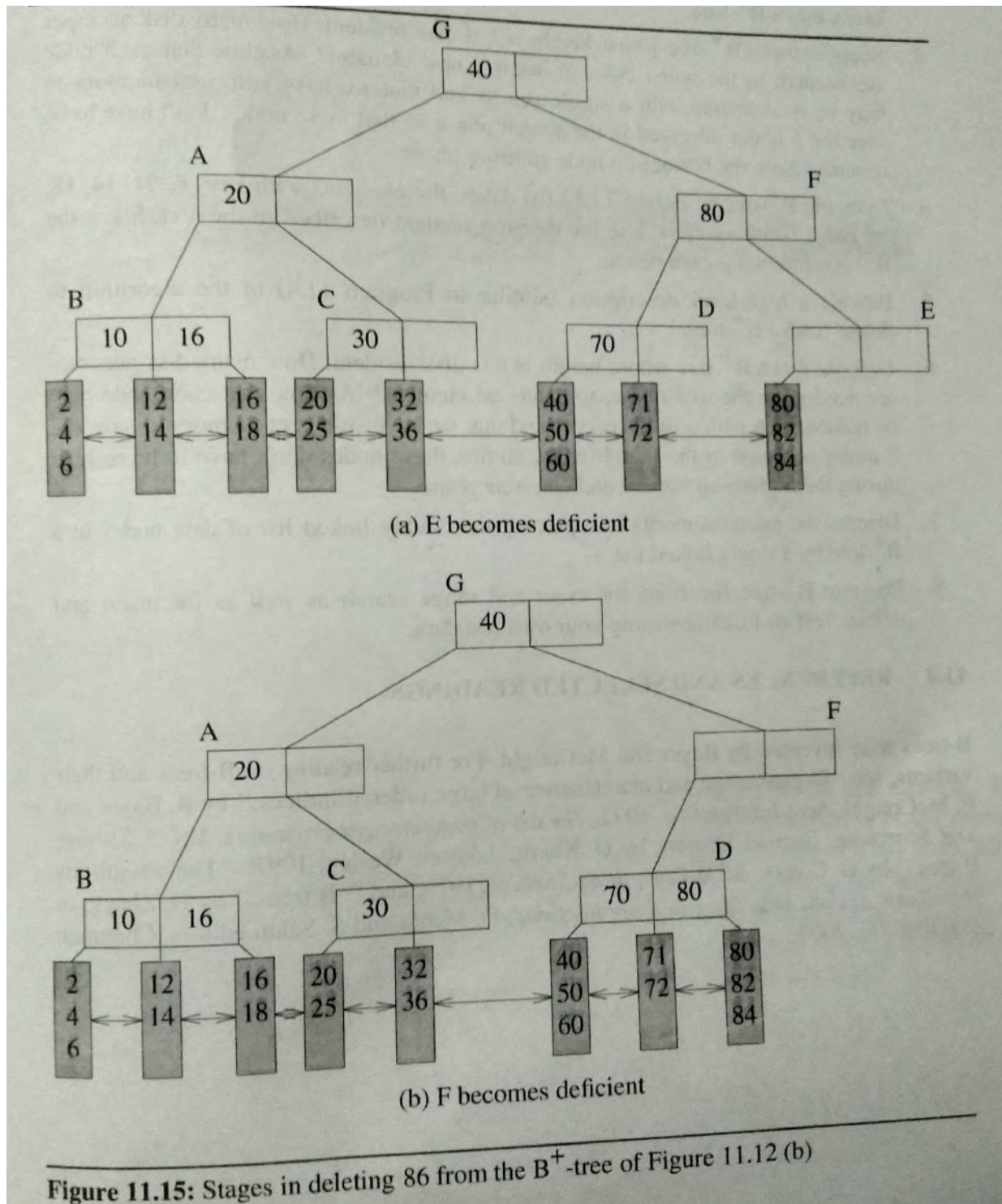
*When an index node deficient we examine a nearest sibling. If the examined nearest sibling has excess keys, we balance the occupancy of the two index nodes; this balancing involves moving some keys and associated subtrees as well as changing the in-between key in the parent.*

For our example, the in-between key 20 is moved from A to C, the rightmost key 16 of B is moved to A, and the right subtree of B moved to C in fig 11.14(b).





When we are deleting element with key 86, the middle child E becomes deficient and is combined with its sibling; a data node is deleted from the doubly linked list of data nodes and the in-between key 84 in the parent also is deleted as in 11.15(a). The deficient index node E combines with its sibling index node D and the in-between key 80 as in 11.15(b). Finally, the deficient index node F combine with its sibling A and in-between key 40 in its parent G. This causes the parent G, which is the root, to become deficient. The deficient root is discarded and we get the B<sup>+</sup> - tree of Fig 11.12(a).



## Graphs and Hashing

The Graph Abstract Data Type, Elementary Graph Operations, Minimum Cost Spanning Trees, Shortest Paths and Transitive Closure. Hashing: Introduction to Hash Table, Static Hashing, Dynamic Hashing

## Graphs

### GRAPH

A graph  $G$  consists of 2 sets,  $V$  and  $E$ .

$V$  is a finite, non empty set of vertices.

$E$  is a set of pairs of vertices, these pairs are called edges.

$V(G)$  and  $E(G)$  represents the set of vertices and edges respectively of graph  $G$  (Fig 6.2).

- In an undirected graph, the pair of vertices representing any edge is unordered. Thus, the pairs  $(u,v)$  and  $(v,u)$  represent the same edge.
- In a directed graph, each edge is represented by a directed pair  $\langle u,v \rangle$ ;  $u$  is the tail and  $v$  is the head of the edge. Therefore,  $\langle u,v \rangle$  and  $\langle v,u \rangle$  represent two different edges.

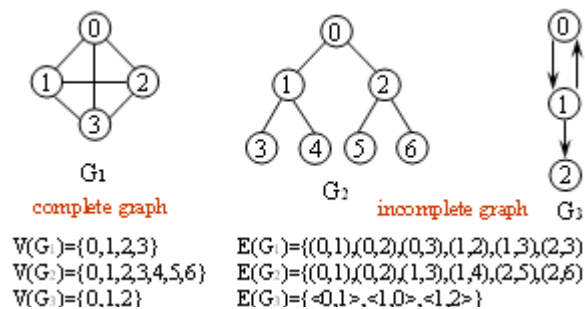


Figure 6.2: Three sample graphs

### Following are the restrictions on graphs

- 1) A graph may not have an edge from a vertex  $v$  back to itself. Such edges are known as self loops (Figure 6.3).
- 2) A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data object referred to as multigraph.

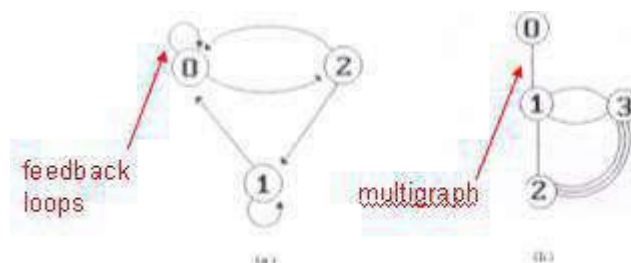


Figure 6.3: Examples of graph like structures

- Maximum number of edges in any  $n$ -vertex, undirected graph is  $n(n-1)/2$ .
- Maximum number of edges in any  $n$ -vertex, directed graph is  $n(n-1)$ .

### TERMINOLOGIES USED IN A GRAPH

- Subgraph of  $G$  is a graph  $G'$  such that  $V(G') \subseteq V(G)$  and  $E(G') \subseteq E(G)$  (Fig 6.4).
- A path from vertex  $u$  to vertex  $v$  in graph  $G$  is a sequence of vertices  $u, i_1, i_2, \dots, i_k, v$  such that  $(u, i_1), (i_1, i_2), \dots, (i_k, v)$  are edges in  $E(G)$ .
- A **simple path** is a path in which all vertices except possibly the first and last are distinct.
- A **cycle** is a simple path in which the first and last vertices are the same.
- A undirected graph is said to be **connected** iff  
for every pair of distinct vertices  $u$  &  $v$  in  $V(G)$  there is a path from  $u$  to  $v$  in  $G$

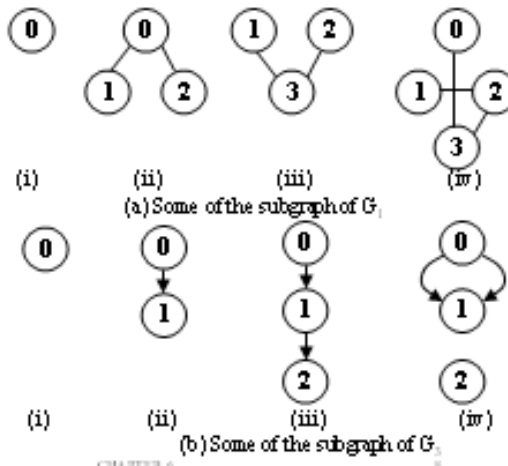


Figure 6.4: Some subgraph

- A connected component H of an undirected graph is a maximal connected subgraph (Figure 6.5)

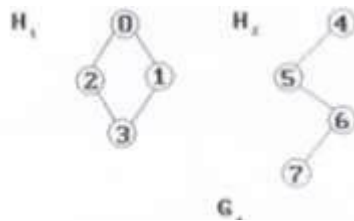


Figure 6.5: A graph with two connected components

- A tree is a connected acyclic (i.e. has no cycles) graph.
- A directed graph G is said to be strongly connected iff for every pair of distinct vertices u and v in V(G), there is a directed path from u to v and also from v to u (Figure 6.6).

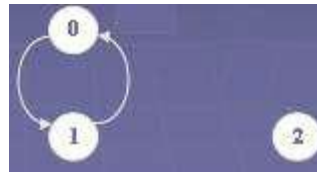


Figure 6.6: Strongly connected components of G3

- The degree of a vertex is the number of edges incident to that vertex. (Degree of vertex 0 is 3)
- In a directed graph G, in-degree of a vertex v defined as the number of edges for which v is the head. The out-degree is defined as the number of edges for which v is the tail. (Vertex 1 of G3 has in-degree 1, out-degree 2 and degree 3).

## GRAPH ABSTRACT DATA TYPE

```

structure Graph is
 objects: a nonempty set of vertices and a set of undirected edges, where each edge is a
 pair of vertices
 functions: for all graph \in Graph, v, v1 and v2 \in Vertices
 Graph Create()::=return an empty graph
 Graph InsertVertex(graph, v)::= return a graph with v inserted. v has no incident edge.
 Graph InsertEdge(graph, v1,v2)::= return a graph with new edge between v1 and v2
 Graph DeleteVertex(graph, v)::= return a graph in which v and all edges incident to it
 are removed
 Graph DeleteEdge(graph, v1, v2)::= return a graph in which the edge (v1, v2) is
 removed
 Boolean IsEmpty(graph)::= if (graph==empty graph)
 return TRUE
 else
 return FALSE
 List Adjacent(graph,v)::= return a list of all vertices that are adjacent to v

```

ADT 6.1: Abstract data type Graph

## GRAPH REPRESENTATIONS

- Three commonly used representations are:
  - 1) Adjacency matrices,
  - 2) Adjacency lists and
  - 3) Adjacency multilists

### Adjacency Matrix

- Let  $G=(V,E)$  be a graph with  $n$  vertices,  $n \geq 1$ .
- The adjacency matrix of  $G$  is a two-dimensional  $n \times n$  array (say  $a$ ) with the property that  $a[i][j]=1$  iff the edge  $(i,j)$  is in  $E(G)$ .  $a[i][j]=0$  if there is no such edge in  $G$  (Figure 6.7).
- The space needed to represent a graph using its adjacency matrix is  $n^2$  bits.
- About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix.

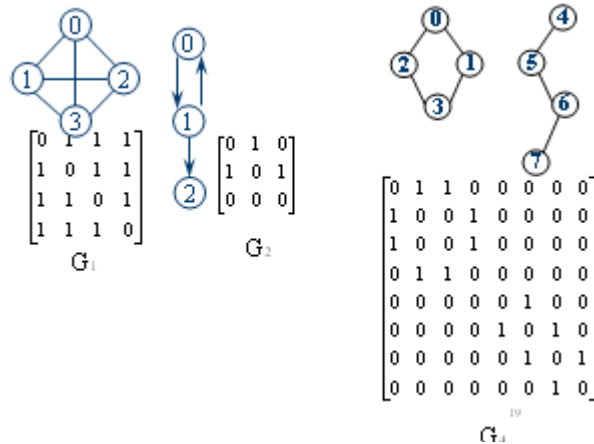


Figure 6.7 Adjacency matrices

### Adjacency Lists

- The  $n$  rows of the adjacency matrix are represented as  $n$  chains.
- There is one chain for each vertex in  $G$ .
- The data field of a chain node stores the index of an adjacent vertex (Figure 6.8).
- For an undirected graph with  $n$  vertices and  $e$  edges, this representation requires an array of size  $n$  and  $2e$  chain nodes.

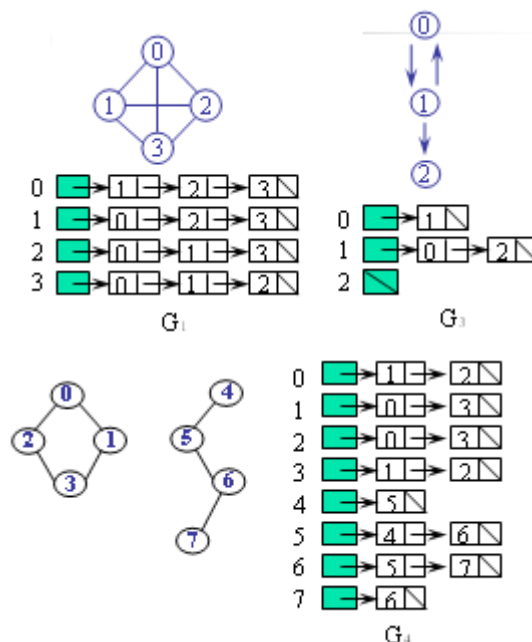


Figure 6.8 adjacency lists

## Adjacency Multilists

- Each edge (u,v) is represented by two entries, one on the list for u and the other on the list for v.
- The new node structure is

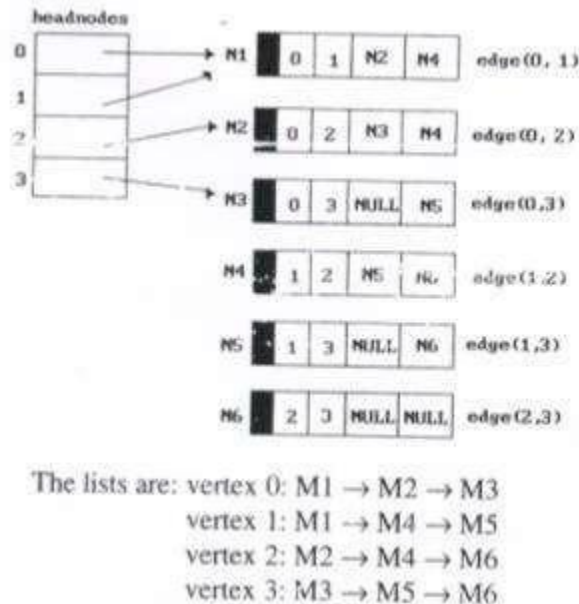


Figure 6.12: Adjacency multilists for G1

## Elementary Graph Operations:

Graph traversal is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way. Tree traversal is a special case of graph traversal.

### Depth First Search:

A depth-first search (DFS) is an algorithm for traversing a finite graph. DFS visits the child nodes before visiting the sibling nodes; that is, it traverses the depth of any particular path before exploring its breadth. A stack (often the program's call stack via recursion) is generally used when implementing the algorithm.

The algorithm begins with a chosen "root" node; it then iteratively transitions from the current node to an adjacent, unvisited node, until it can no longer find an unexplored node to transition to from its current location. The algorithm then backtracks along previously visited nodes, until it finds a node connected to yet more uncharted territory. It will then proceed down the new path as it had before, backtracking as it encounters dead-ends, and ending only when the algorithm has backtracked past the original "root" node from the very first step.

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
/* depth first search of a graph beginning at v */
{
 nodePointer w;
 visited[v] = TRUE;
 printf("%5d", v);
 for (w = graph[v]; w; w = w->link)
 if (!visited[w->vertex])
 dfs(w->vertex);
}
```

Program 6.1: Depth first search



We begin the search by visiting the start vertex,  $v$ . In this simple application; visiting consists of printing the node's vertex field. Next, we select an unvisited vertex,  $w$ , from  $v$ 's adjacency list and carry out a depth first search on  $w$ . We preserve our current position in  $v$ 's adjacency list by placing it on a stack. Eventually our search reaches a vertex,  $u$ , that has no unvisited vertices on its adjacency list. At this point, we remove a vertex from the stack and continue processing its adjacency list. Previously visited vertices are discarded; unvisited vertices are visited and placed on the stack. The search terminates when the stack is empty.

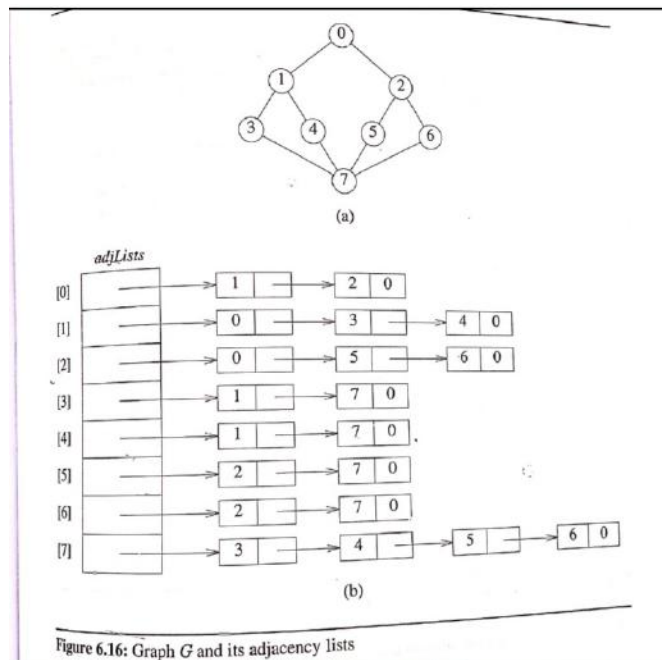


Figure 6.16: Graph  $G$  and its adjacency lists

Example: We wish to carry out a DFS of Graph  $G$  of Fig 6.16. If we initiate this search from vertex  $v_0$ , then the vertices of  $G$  are visited in the following order:  $v_0, v_1, v_3, v_7, v_4, v_5, v_2, v_6$ .

### Breadth First Search:

A breadth-first search (BFS) is another technique for traversing a finite graph. BFS visits the neighbor nodes before visiting the child nodes, and a queue is used in the search process. This algorithm is often used to find the shortest path from one node to another.

BFS starts at vertex  $v$  and marks it as visited. It then visits each of the vertices on  $v$ 's adjacency list. When we have visited all the vertices on  $v$ 's adjacency list, we visit all the unvisited vertices that are adjacent to the first vertex on  $v$ 's adjacency list. To implement this scheme, as we visit each vertex we place the vertex in queue. When we have exhausted an adjacency list, we remove a vertex from the queue and proceed by examining each of the vertices on its adjacency list. Unvisited vertices are visited and then placed on the queue; visited vertices are ignored. We have finished the search when the queue is empty.

To implement BFS, we use a dynamically linked queue. Each queue node contains vertex and fields.

```

void bfs(int v)
/* breadth first traversal of a graph, starting at v
the global array visited is initialized to 0, the queue
operations are similar to those described in
Chapter 4, front and rear are global */
nodePointer w;
front = rear = NULL; /* initialize queue */
printf("%5d", v);
visited[v] = TRUE;
addq(v);
while (front) {
 v = deleteq();
 for (w = graph[v]; w; w = w->link)
 if (!visited[w->vertex]) {
 printf("%5d", w->vertex);
 addq(w->vertex);
 visited[w->vertex] = TRUE;
 }
 }
}

```

**Program 6.2:** Breadth first search of a graph

The queue definition and the function prototypes used by *bfs* are:

```

typedef struct queue *queuePointer;
typedef struct {
 int vertex;
 queuePointer link;
} queue;
queuePointer front, rear;
void addq(int);
int deleteq();

```

## Connected Components:

We can use the two elementary graph searches to create additional, more interesting, graph operations. For illustrative purposes, let us look at the problem of determining whether or not an undirected graph is connected. We can implement this operation by simply calling either *dfs*(0) or *bfs*(0) and then determining if there are any unvisited vertices. For example, the call *dfs*(0) applied to graph  $G_4$  of Figure 6.5 terminates without visiting vertices 4, 5, 6, and 7. Therefore, we can conclude that graph  $G_4$  is not connected. The computing time for this operation is  $O(n + e)$  if adjacency lists are used.

A closely related problem is that of listing the connected components of a graph. This is easily accomplished by making repeated calls to either *dfs*(*v*) or *bfs*(*v*) where *v* is an unvisited vertex. The function *connected* (Program 6.3) carries out this operation. Although we have used *dfs*, *bfs* may be used with no change in the time complexity.

```

void connected(void)
/* determine the connected components of a graph */
int i;
for (i = 0; i < n; i++)
 if (!visited[i]) {
 dfs(i);
 printf("\n");
 }
}

```

**Program 6.3:** Connected components

Total time is needed to generate all the connected components is  $O(n+e)$ .

## Spanning Trees:

Spanning tree is any tree that consists solely of edges in  $G$  and that includes all the vertices in  $G$ .

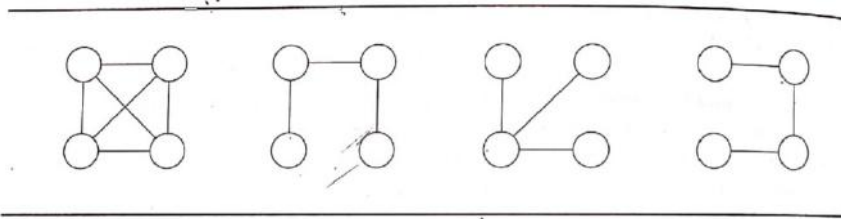


Figure 6.17: A complete graph and three of its spanning trees

When DFS is used, the resulting spanning tree is known as a depth first spanning tree. When BFS is used, the resulting spanning tree is known as a breadth first spanning tree. Following fig shows the results:

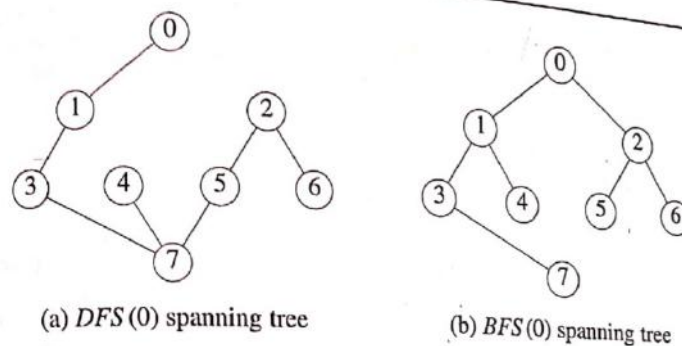


Figure 6.18: Depth-first and breadth-first spanning trees for graph of Figure 6.16

- 1) Now suppose we add a nontree edge,  $(v,w)$ , into any spanning tree,  $T$ . The result is a cycle that consists of the edge  $(v,w)$  and all the edges on the path from  $w$  to  $v$  in  $T$ .
- 2) A spanning tree is a minimal subgraph,  $G'$ , of  $G$  such that  $V(G')=V(G)$  and  $G'$  is connected. We define a minimal subgraph as one with the fewest number of edges. Spanning tree has  $n-1$  edges.

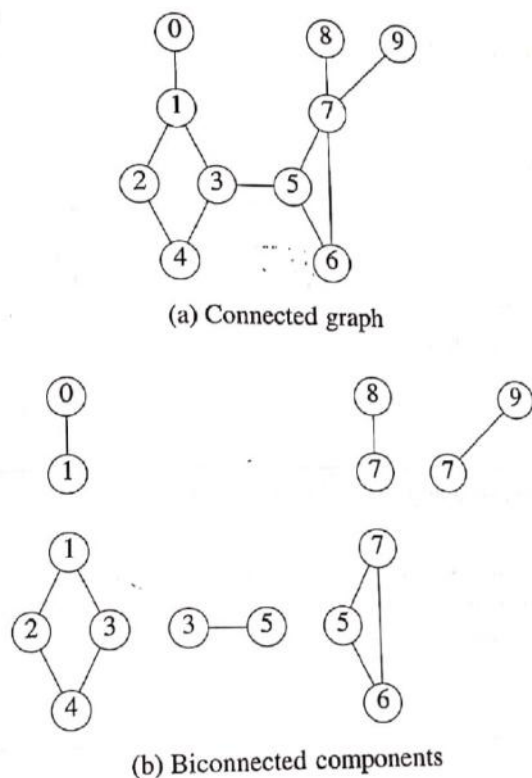
## Biconnected Components:

**Articulation point** is vertex  $v$  of  $G$ , such that the deletion of  $v$ , together with all edges incident on  $v$ , produces on graph,  $G'$ , that has at least two connected components. For example, the connected graph of Fig 6.19 has four articulation points, 1, 3, 5 and 7.

A biconnected graph is a connected graph that has no articulation points. For example, the graph of Fig 6.16 is biconnected, while the graph of Fig 6.19 obviously is not. In Fig 6.19(a), vertices represent communication stations and edges represent communication links. Now suppose one of the stations that is an articulation point fails. The result is a loss of communication not just to and from that single station, but also between certain other pairs of stations.



We can find the biconnected components of a connected undirected graph,  $G$ , by using any depth first spanning tree of  $G$ . For example, the function call  $dfs(3)$  applied to the graph of Figure 6.19(a) produces the spanning tree of Figure 6.20(a). We have redrawn the tree in Figure 6.20(b) to better reveal its tree structure. The numbers outside the vertices in either figure give the sequence in which the vertices are visited during the depth first search. We call this number the *depth first number*, or *dfn*, of the vertex. For example,  $dfn(3) = 0$ ,  $dfn(0) = 4$ , and  $dfn(9) = 8$ . Notice that vertex 3, which is an ancestor of both vertices 0 and 9, has a lower *dfn* than either of these vertices. Generally, if  $u$  and  $v$  are two vertices, and  $u$  is an ancestor of  $v$  in the depth first spanning tree, then  $dfn(u) < dfn(v)$ .



The broken lines in Fig 6.20(b) represent nontree edges. A nontree edge  $(u,v)$  is a back edge iff either  $u$  is an ancestor of  $v$  or  $v$  is an ancestor of  $u$ .

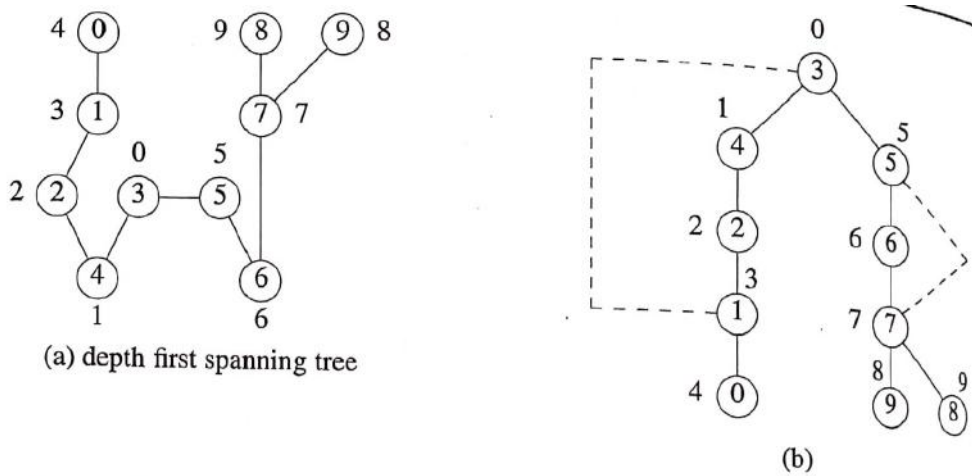


Figure 6.20: Depth first spanning tree of Figure 6.19(a)

### Minimum Cost Spanning Trees:

The cost of a spanning tree of weighted undirected graph is the sum of the costs (weights) of the edges in the spanning tree. Minimum cost spanning tree is a spanning tree of least cost. There are 3 different algorithms used to obtain a minimum cost spanning tree of a connected undirected graph. They are: *Kruskal's*, *Prim's*, and *Sollin's algorithms* respectively. All three use an algorithm design strategy called the greedy method.

In greedy method, we construct an optimal solution in stages. At each stage we make a decision that is the best decision at this time. Since we cannot change later, we make sure that the decision will result in a feasible solution.

For spanning trees, we use a least cost criterion. Our solution must satisfy the following constraints:

- 1) We must use only edges within the graph.
- 2) We must use exactly  $n-1$  edges.
- 3) We may not use edges that would produce a cycle.

### Kruskal's Algorithm:

Kruskal's algorithm builds a minimum cost spanning tree  $T$  by adding edges to  $T$  one at a time. The algorithm selects the edges for inclusion in  $T$  in nondecreasing order of their cost. An edge is added to  $T$  if it does not form a cycle with the edges that are already in  $T$ . since  $G$  is connected and has  $n > 0$  vertices, exactly  $n-1$  edges will be selected for inclusion in  $T$ .

**Example 6.3:** We will construct a minimum cost spanning tree of the graph of Figure 6.22(a). Figure 6.23 shows the order in which the edges are considered for inclusion, as well as the result and the changes (if any) in the spanning tree. For example, edge (0, 5) is the first considered for inclusion. Since it obviously cannot create a cycle, it is added to the tree. The result is the tree of Figure 6.22(c). Similarly, edge (2, 3) is considered next. It is also added to the tree, and the result is shown in Figure 6.22(d). This process continues until the spanning tree has  $n-1$  edges (Figure 6.22(h)). The cost of the spanning tree is 99.  $\square$

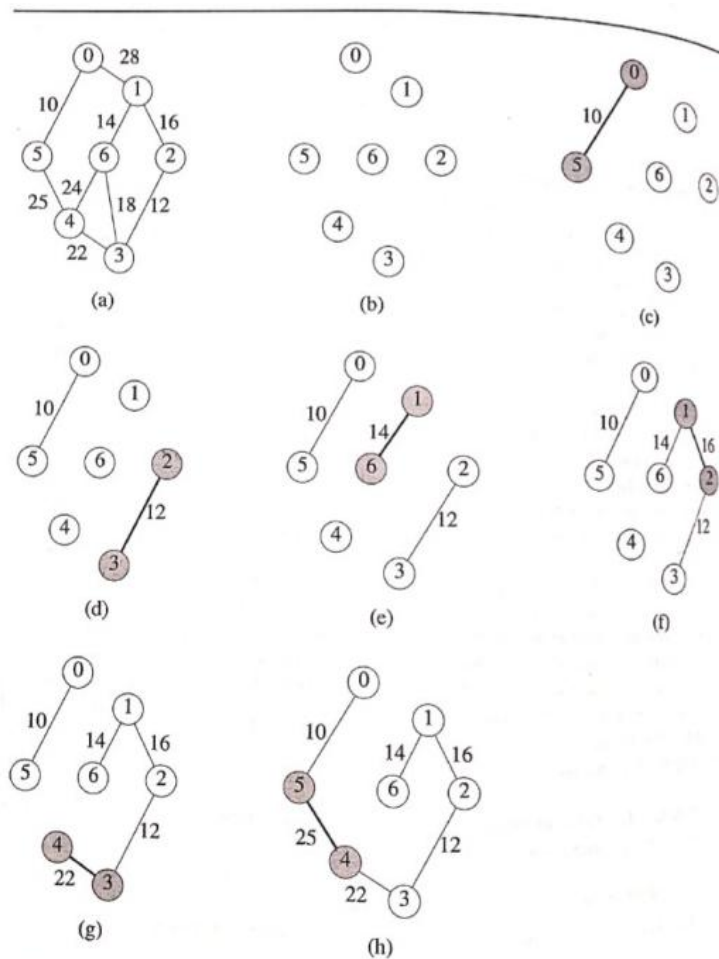


Figure 6.22: Stages in Kruskal's algorithm

**Theorem 6.1:** Let  $G$  be an undirected connected graph. Kruskal's algorithm generates a minimum cost spanning tree.

**Proof:** We shall show that:

- (a) Kruskal's method produces a spanning tree whenever a spanning tree exists.
- (b) The spanning tree generated is of minimum cost.

For (a), we note that Kruskal's algorithm only discards edges that produce cycles. We know that the deletion of a single edge from a cycle in a connected graph produces a



graph that is also connected. Therefore, if  $G$  is initially connected, the set of edges in  $T$  and  $E$  always form a connected graph. Consequently, if  $G$  is initially connected, the algorithm cannot terminate with  $E = \{\}$  and  $|T| < n - 1$ .

Now let us show that the constructed spanning tree,  $T$ , is of minimum cost. Since  $G$  has a finite number of spanning trees, it must have at least one that is of minimum cost. Let  $U$  be such a tree. Both  $T$  and  $U$  have exactly  $n - 1$  edges. If  $T = U$ , then  $T$  is of minimum cost and we have nothing to prove. So, assume that  $T \neq U$ . Let  $k, k > 0$ , be the number of edges in  $T$  that are not in  $U$  ( $k$  is also the number of edges in  $U$  that are not in

$T$ ).

We shall show that  $T$  and  $U$  have the same cost by transforming  $U$  into  $T$ . This transformation is done in  $k$  steps. At each step, the number of edges in  $T$  that are not in  $U$  is reduced by exactly 1. Furthermore, the cost of  $U$  is not changed as a result of the transformation. As a result,  $U$  after  $k$  transformation steps has the same cost as the initial  $U$  and contains exactly those edges that are in  $T$ . This implies that  $T$  is of minimum cost.

For each transformation step, we add one edge,  $e$ , from  $T$  to  $U$  and remove one edge,  $f$ , from  $U$ . We select the edges  $e$  and  $f$  in the following way:

- (1) Let  $e$  be the least cost edge in  $T$  that is not in  $U$ . Such an edge must exist because  $k > 0$ .
- (2) When we add  $e$  to  $U$ , we create a unique cycle. Let  $f$  be any edge on this cycle that is not in  $T$ . We know that at least one of the edges on this cycle is not in  $T$  because  $T$  contains no cycles.

Given the way  $e$  and  $f$  are selected, it follows that  $V = U + \{e\} - \{f\}$  is a spanning tree and that  $T$  has exactly  $k - 1$  edges that are not in  $V$ . We need to show that the cost of  $V$  is the same as the cost of  $U$ . Clearly, the cost of  $V$  is the cost of  $U$  plus the cost of the edge  $e$  minus the cost of the edge  $f$ . The cost of  $e$  cannot be less than the cost of  $f$  since this would mean that the spanning tree  $V$  has a lower cost than the tree  $U$ . This is impossible. If  $e$  has a higher cost than  $f$ , then  $f$  is considered before  $e$  by Kruskal's algorithm. Since it is not in  $T$ , Kruskal's algorithm must have discarded this edge at this time. Therefore,  $f$  together with the edges in  $T$  having a cost less than or equal to the cost of  $f$  must form a cycle. By the choice of  $e$ , all these edges are also in  $U$ . Thus,  $U$  must contain a cycle. However, since  $U$  is a spanning tree it cannot contain a cycle. So the assumption that  $e$  is of higher cost than  $f$  leads to a contradiction. This means that  $e$  and  $f$  must have the same cost. Hence,  $V$  has the same cost as  $U$ .  $\square$

### Prim's Algorithm:

Prim's Algorithm begins with a tree,  $T$ , that contains a single vertex. This may be any of the vertices in the original graph. Next, we add least cost edge  $(u,v)$  to  $T$  such that  $T \cup \{(u,v)\}$  is also a tree. We repeat this edge addition step until  $T$  contains  $n-1$  edges. To make sure that the added edge does not form a cycle, at each step we choose the edge  $(u,v)$  such that exactly one of  $u$  or  $v$  is in  $T$ . Fig 6.24 shows the progress of Prim's algorithm on the graph of Fig 6.22(a).

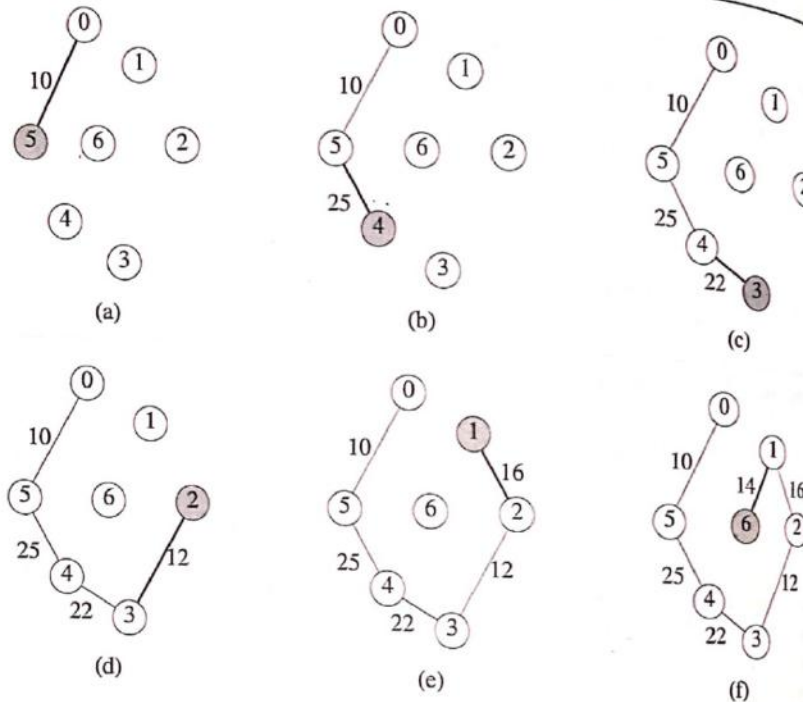
```

T = {};
TV = {0}; /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
 let (u, v) be a least cost edge such that u ∈ TV and
 v ∉ TV;
 if (there is no such edge)
 break;
 add v to TV;
 add (u, v) to T;
}
if (T contains fewer than n-1 edges)
 printf("No spanning tree\n");

```

**Program 6.8:** Prim's algorithm

To implement Prim's algorithm, we assume that each vertex  $v$  that is not in  $TV$  has a companion vertex,  $near(v)$ , such that  $near(v) \in TV$  and  $cost(near(v), v)$  is minimum over all such choices for  $near(v)$ . (We assume that  $cost(v, w) = \infty$  if  $(v, w) \notin E$ ). At each stage we select  $v$  so that  $cost(near(v), v)$  is minimum and  $v \notin TV$ . Using this strategy we can implement Prim's algorithm in  $O(n^2)$ , where  $n$  is the number of vertices in  $G$ . Asymptotically faster implementations are also possible. One of these results from the use of Fibonacci heaps which we examine in Chapter 9.



**Figure 6.24:** Stages in Prim's algorithm

### Sollin's Algorithm:

This algorithm selects several edges for inclusion in  $T$  at each stage. At the start of the stage, the selected edges, together with all  $n$  vertices, form a spanning forest. During a stage we select one edge for each tree in the forest. This edge is a minimum cost edge that has exactly one vertex in the tree. Since two trees in the forest could select same edge, we need to eliminate multiple copies of edges. At the start of the first stage the set selected set of edges is empty. The algorithm terminates when there is only one tree at the end of a stage or no edges remain for selection.

Fig 6.25 shows Sollin's algorithm applied to the graph of 6.22(a). The initial configuration of zero selected edged is the same as shown in Fig 6.22(b). Each tree in this forest is single vertex. At the next stage, we select edges for each of the vertices. The edges selected are  $(0,5), (1,6), (2,3), (3,2), (4,3), (5,0), (6,1)$ . After eliminating the duplicate edges, we are left with edges  $(0,5), (1,6), (2,3)$ , and  $(4,3)$ . We add these edges to the set of selected

edges, thereby producing the forest of Fig 6.25(a). In the next stage, the tree with vertex set  $\{0,5\}$  selects edge(5,4), and the two remaining trees selects edge (1,2).

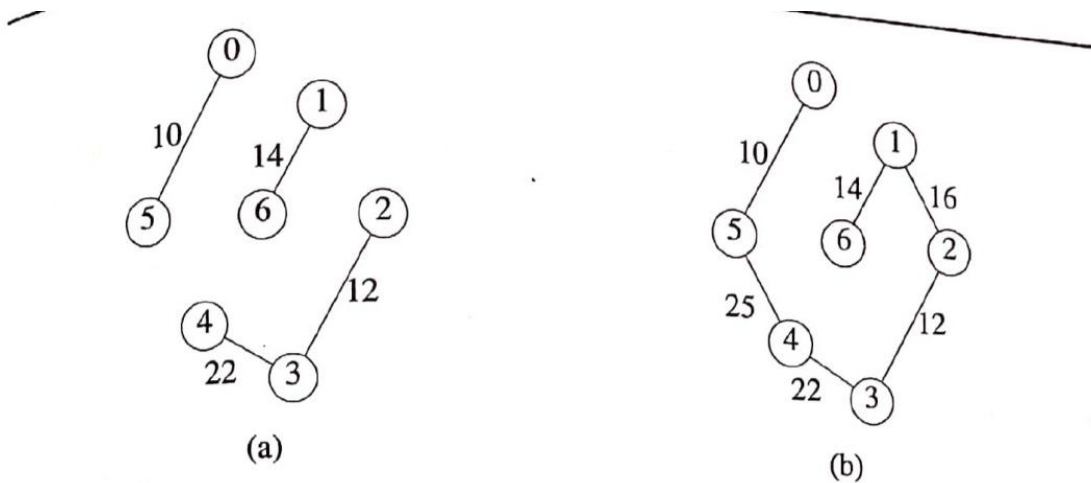


Figure 6.25: Stages in Sollin's algorithm

### Shortest Paths and Transitive Closure:

Each edge has a weight representing the distance between two vertices. An edge weight is also referred to as an edge length or edge cost. The length of a path is now defined to be the sum of the lengths of the edges on that path, rather than the number of edges. The starting vertex of the path will be referred to as the source and the last vertex the destination.

#### Single Source/All Destinations: Nonnegative Edge Costs:

In this problem we are given a directed graph,  $G=(V,E)$ , a weighting function,  $w(e)$ ,  $w(e)>0$ , for the edges of  $G$ , and a source vertex,  $v_0$ . We wish to determine a shortest path from  $v_0$  to each of the remaining vertices of  $G$ . As an example, consider the graph of 6.26(a). If  $v_0$  is the source vertex, then the shortest path from  $v_0$  to  $v_1$  is  $v_0, v_3, v_4, v_1$ . The length of this path is  $10+15+20=45$ . Although there are three edges on this path, it is shorter than the path  $v_0 v_1$ , which has a length of 50. Fig 6.26(b) may use greedy algorithm to generate shortest paths and it lists the shortest paths from  $v_0$  to  $v_1, v_0, v_3$  and  $v_4$  in nondecreasing order of path length. There is no path from  $v_0$  to  $v_5$ .

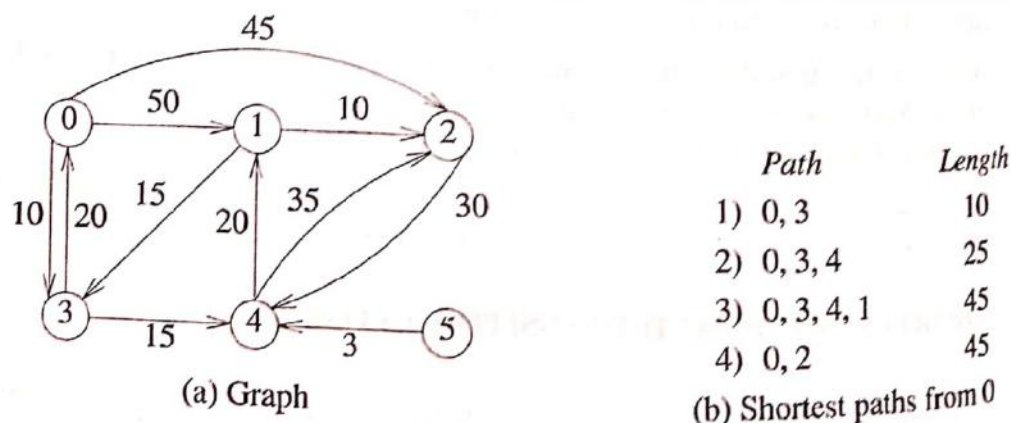


Fig 6.26: Graph and Shortest paths from vertex 0 to all destinations

Let  $S$  denote set of vertices, including  $v_0$ , whose shortest paths have been found. For  $w$  not in  $S$ , let  $\text{distance}[w]$  be the length of the shortest path starting from  $v_0$ , going through vertices only in  $S$ , and ending in  $w$ . we can observe following points while generating paths:

(1) If the next shortest path is to vertex  $u$ , then the path from  $v_0$  to  $u$  goes through only those vertices that are in  $S$ . There cannot be any intermediate vertex that is not in  $S$ .



- (2) Vertex  $u$  is chosen so that it has the minimum distance,  $\text{distance}[u]$ , among all the vertices not in  $S$ . If there are several vertices not in  $S$  with the same distance, then we may select any one of them.
- (3) Once we have selected  $u$  and generated the shortest path from  $v_0$  to  $u$ ,  $u$  becomes a member of  $S$ . Adding  $u$  to  $S$  can change the distance of shortest paths starting at  $v_0$ , going through vertices only in  $S$ , and ending at a vertex,  $w$ , that is not currently in  $S$ . If the distance changes, we have found a shorter such path from  $v_0$  to  $w$ . This path goes through  $u$ . The intermediate vertices on this path are in  $S$  and its subpath from  $u$  to  $w$  can be chosen so as to have no intermediate vertices. The length of the shorter path is  $\text{distance}[u] + \text{length}(\langle u, w \rangle)$ .

To implement Dijkstra's algorithm, we assume that the  $n$  vertices are numbered from 0 to  $n-1$ .

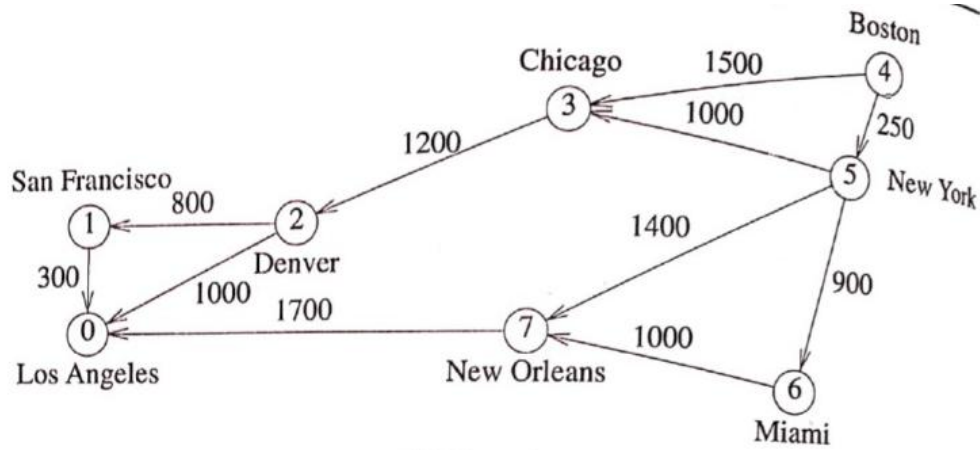
```
void shortestPath(int v, int cost[][MAX-VERTICES],
 int distance[], int n, short int found[])
/* distance[i] represents the shortest path from vertex v
 to i, found[i] is 0 if the shortest path from i
 has not been found and a 1 if it has, cost is the
 adjacency matrix */
int i, u, w;
for (i = 0; i < n; i++) {
 found[i] = FALSE;
 distance[i] = cost[v][i];
}
found[v] = TRUE;
distance[v] = 0;
for (i = 0; i < n-2; i++) {
 u = choose(distance, n, found);
 found[u] = TRUE;
 for (w = 0; w < n; w++)
 if (!found[w])
 if (distance[u] + cost[u][w] < distance[w])
 distance[w] = distance[u] + cost[u][w];
}
}
```

*Prog 6.9: Single source shortest paths*

```
int choose(int distance[], int n, short int found[])
/* find the smallest distance not yet checked */
int i, min, minpos;
min = INT_MAX;
minpos = -1;
for (i = 0; i < n; i++)
 if (distance[i] < min && !found[i]) {
 min = distance[i];
 minpos = i;
 }
return minpos;
}
```

*Prog 6.10: Choosing the least cost edge*

**Example:** Consider the eight-vertex digraph of Figure 6.27(a) with eight adjacency matrix as in 6.27(b). Suppose that the source vertex is Boston. The values of  $\text{dist}$  and the vertex  $u$  selected in each iteration of the outer for loop of program 6.9 are shown in figure 6.28.



(a) Digraph

|   | 0    | 1   | 2    | 3    | 4 | 5   | 6   | 7    |
|---|------|-----|------|------|---|-----|-----|------|
| 0 | 0    |     |      |      |   |     |     |      |
| 1 | 300  | 0   |      |      |   |     |     |      |
| 2 | 1000 | 800 | 0    |      |   |     |     |      |
| 3 |      |     | 1200 | 0    |   |     |     |      |
| 4 |      |     |      | 1500 | 0 | 250 |     |      |
| 5 |      |     |      | 1000 |   | 0   | 900 | 1400 |
| 6 |      |     |      |      |   |     | 0   | 1000 |
| 7 | 1700 |     |      |      |   |     |     | 0    |

(b) Length-adjacency matrix

Fig 6.27 Digraph for Example 6.4

| Iteration | Vertex selected | Distance |          |          |      |      |     |          |          |
|-----------|-----------------|----------|----------|----------|------|------|-----|----------|----------|
|           |                 | LA       | SF       | DEN      | CHI  | BOST | NY  | MIA      | NO       |
|           |                 | [0]      | [1]      | [2]      | [3]  | [4]  | [5] | [6]      | [7]      |
| Initial   | ----            | $\infty$ | $\infty$ | $\infty$ | 1500 | 0    | 250 | $\infty$ | $\infty$ |
| 1         | 5               | $\infty$ | $\infty$ | $\infty$ | 1250 | 0    | 250 | 1150     | 1650     |
| 2         | 6               | $\infty$ | $\infty$ | $\infty$ | 1250 | 0    | 250 | 1150     | 1650     |
| 3         | 3               | $\infty$ | $\infty$ | 2450     | 1250 | 0    | 250 | 1150     | 1650     |
| 4         | 7               | 3350     | $\infty$ | 2450     | 1250 | 0    | 250 | 1150     | 1650     |
| 5         | 2               | 3350     | 3250     | 2450     | 1250 | 0    | 250 | 1150     | 1650     |
| 6         | 1               | 3350     | 3250     | 2450     | 1250 | 0    | 250 | 1150     | 1650     |

Figure 6.28: Action of *shortestPath* on digraph of Figure 6.27

### Single Source/All Destinations: General Weights

Consider the general case when some or all of the edges of the directed graph  $G$  may have negative length. When negative edge lengths are permitted, we require that the graph have no cycles of negative length. For example, consider the graph of Fig 6.30. the length of the shortest path from vertex 0 to vertex 2 is  $-\infty$ , as the length of the path



0,1,0,1,0,1, ..., 0,1,2

can be arbitrarily small. This is so because of the presence of the cycle 0,1,0, which has a length of -1.

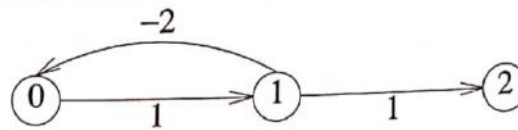


Fig: Directed Graph with a cycle of negative length

When there are no cycles of negative length, there is a shortest path between any two vertices of an  $n$ -vertex graph that has at most  $n-1$  edges on it. Elimination of the cycles from the path results in another path with the same source and destination. This path is cycle-free and has a length that is no more than that of the original path, as the length of the eliminated cycles was at least zero. This observation can be used to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph.

When there are no cycles of negative length, we can limit our search for shortest paths to paths with at most  $n-1$  edges. Hence,  $\text{dist}^{n-1}[u]$  rather than  $\text{dist}[u]$  is the length of an unrestricted shortest path from  $v$  to  $u$ .

While computing  $\text{dist}^{n-1}[u]$ , we make the following observations:

(1) If the shortest path from  $v$  to  $u$  with at most  $k$ ,  $k > 1$ , edges has no more than  $k-1$  edges, then

$$\text{dist}^k[u] = \text{dist}^{k-1}[u].$$

(2) If the shortest path from  $v$  to  $u$  with at most  $k$ ,  $k > 1$ , edges has exactly  $k$  edges, then it is comprised of a shortest path from  $v$  to some vertex  $j$  followed by the edge  $\langle j, u \rangle$ . The path from  $v$  to  $j$  has  $k-1$  edges, and its length is  $\text{dist}^{k-1}[j]$ . All vertices  $i$  such that the edge  $\langle i, u \rangle$  is in graph are candidates for  $j$ . since we are interested in a shortest path, the  $i$  that minimizes  $\text{dist}^{k-1}[i] + \text{length}[i][u]$  is the correct value for  $j$ .

$$\text{dist}^k[u] = \min\{\text{dist}^{k-1}[u], \min_i\{\text{dist}^{k-1}[i] + \text{length}[i][u]\}\}$$

This recurrence may be used to compute  $\text{dist}^k$  from  $\text{dist}^{k-1}$ , for  $k = 2, 3, \dots, n-1$ .

```

1 void BellmanFord(int n, int v)
2 /* single source all destination shortest paths
3 with negative edge lengths. */
4 for (int i = 0; i < n; i++)
5 dist[i] = length[v][i]; /* initialize dist */
6
7 for (int k = 2; k <= n-1; k++)
8 for (each u such that u != v and u
9 has at least one incoming edge)
10 for (each <i, u> in the graph)
11 if (dist[u] > dist[i] + length[i][u])
12 dist[u] = dist[i] + length[i][u];

```

Program 6.11: Bellman and Ford algorithm to compute shortest paths

The overall complexity is  $O(n^3)$  when adjacency matrices are used and  $O(ne)$  when adjacency lists are used.

### All Pairs Shortest Paths:

Here, we must find the shortest paths between all pairs of vertices,  $v_i, v_j, i \neq j$ .

Basic idea in all pairs algorithm is to begin with the matrix  $A^{-1}$  and successively generate the matrices  $A^0, A^1, A^2, \dots, A^{n-1}$ . If we have already generated  $A^{k-1}$ , then we may generate  $A^k$  by realizing that any pair of vertices  $i, j$  one of the two rules below applies.

(1) The shortest path from  $i$  to  $j$  going through no vertex with index greater than  $k$  does not go through the vertex with index  $k$  and so its cost is  $A^{k-1}[i][j]$ .

(2) The shortest such path does go through vertex  $k$ . such a path consists of a path from  $i$  to  $k$  followed by one

from  $k$  to  $j$ . Neither of these goes through a vertex with index greater than  $k-1$ . Hence, their costs are  $A^{k-1}[i][k]$  and  $A^{k-1}[i][j]$ .

These rules yield the following formulas for  $A^k[i][j]$ :

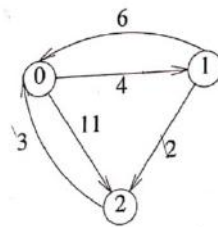
$$A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$$

and

$$A^{-1}[i][j] = \text{cost}[i][j]$$

Example:

For the digraph 6.33(a), the initial a matrix,  $A^{-1}$ , plus its value after each of three iterations,  $A^0$ ,  $A^1$ ,  $A^2$  is also given in Fig 6.33.



(a) Example digraph

| $A^{-1}$ | 0 | 1        | 2  |
|----------|---|----------|----|
| 0        | 0 | 4        | 11 |
| 1        | 6 | 0        | 2  |
| 2        | 3 | $\infty$ | 0  |

(b)  $A^{-1}$

| $A^0$ | 0 | 1 | 2  |
|-------|---|---|----|
| 0     | 0 | 4 | 11 |
| 1     | 6 | 0 | 2  |
| 2     | 3 | 7 | 0  |

(c)  $A^0$

| $A^1$ | 0 | 1 | 2 |
|-------|---|---|---|
| 0     | 0 | 4 | 6 |
| 1     | 6 | 0 | 2 |
| 2     | 3 | 7 | 0 |

(d)  $A^1$

| $A^2$ | 0 | 1 | 2 |
|-------|---|---|---|
| 0     | 0 | 4 | 6 |
| 1     | 5 | 0 | 2 |
| 2     | 3 | 7 | 0 |

(e)  $A^2$

Fig 6.33: Example for all-pairs shortest paths problem

### Transitive Closure:

Assume that we have a directed graph  $G$  with unweighted edges. We want to determine if there is a path from  $i$  to  $j$  for all values of  $i$  and  $j$ . Two cases are there:

- (1) requires positive lengths,
- (2) requires only nonnegative path lengths.

These cases are known as the transitive closure and reflexive transitive closure of a graph, respectively.

### Definition:

The transitive closure matrix, denoted  $A^+$ , of a directed graph,  $G$ , is a matrix such that  $A^+[i][j]=1$  if there is a path of length  $>0$  from  $i$  to  $j$ ; otherwise,  $A^+[i][j]=0$ .

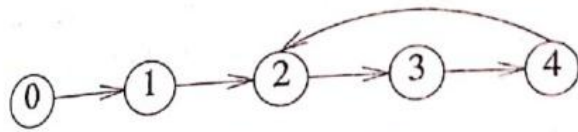
### Definition:

The reflexive transitive closure matrix, denoted  $A^*$ , of a directed graph,  $G$ , is a matrix such that  $A^*[i][j]=1$  if there is a path of length  $\geq 0$  from  $i$  to  $j$ ; otherwise  $A^*[i][j]=0$ .

Figure 6.34 shows  $A^+$  and  $A^*$  for a digraph, clearly, the only difference between  $A^*$  and  $A^+$  is in the terms on the diagonal.  $A^+[i][j]=1$  iff there is a cycle of length  $>1$  containing vertex  $i$ , whereas  $A^*[i][j]$  is always one, as there is a path 0 from  $i$  to  $j$ .

**Transitive closure of a graph:** Given a directed graph find out if a vertex  $j$  is reachable from another vertex  $i$  for all vertex pairs  $(i,j)$  in the graph.

**Reflective Transitive closure of a graph:** Let  $R \subseteq A^2$  be a directed graph defined on a set  $A$ . The reflexive transitive closure of  $R$  is the relation,  $R^* = \{(a,b): a,b \in A \text{ and there is a path from } a \text{ to } b \text{ in } R\}$ , and its reflective transitive closure  $R^* = \{(a_1,a_1), (a_1,a_2), (a_1,a_3), (a_1,a_4), (a_2,a_2), (a_2,a_3), (a_2,a_4), (a_3,a_3), (a_3,a_4), (a_4,a_4)\}$ .



(a) Digraph  $G$

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 1 | 0 | 0 |

(b) Adjacency matrix  $A$

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 |
| 4 | 0 | 0 | 1 | 1 | 1 |

(c)  $A^+$

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 |
| 4 | 0 | 0 | 1 | 1 | 1 |

(d)  $A^*$

Fig 6.34 Graph  $G$  and its adjacency matrix  $A, A^+, A^*$

The total time is  $O(n^3)$ .

The transitive closure of undirected graph  $G$  can be found very easily from its connected components. From the definition of a connected component, it follows that there is a path between every pair of vertices in the component and there is no path in  $G$  between two vertices that are in different components. Hence, if  $A$  is the adjacency matrix of an undirected graph then its transitive closure  $A^+$  may be determined in  $O(n^2)$  time by first determining the connected components of the graph.  $A^+[i][j]=1$  iff there is a path from vertex  $i$  to  $j$ . for every pair of vertices in the same component,  $A^+[i][j]=1$ . On the diagonal,  $A^+[i][i]=1$  iff the component containing  $i$  has at least two vertices.

# Hashing

## STATIC HASHING

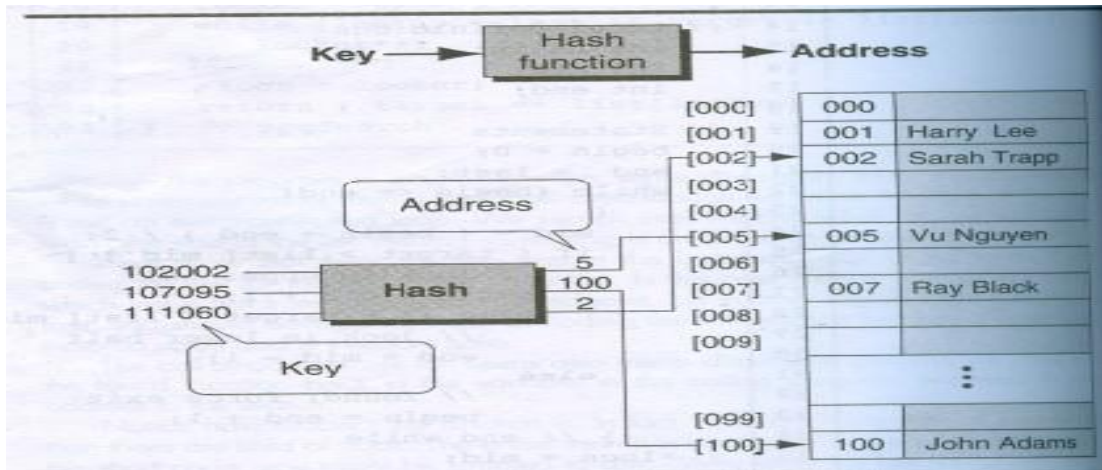
### Hash Tables:

#### 5.6.1 Basic concept:

In a hashed search, the key, through an algorithmic function, determines the *location* of the data.

We use a hashing algorithm to transform the key into the index that contains the data we need to locate. Another way to describe hashing is as a key-to-address transformation in which the keys map to addresses in a *list*.

Hashing is a key-to address mapping process.

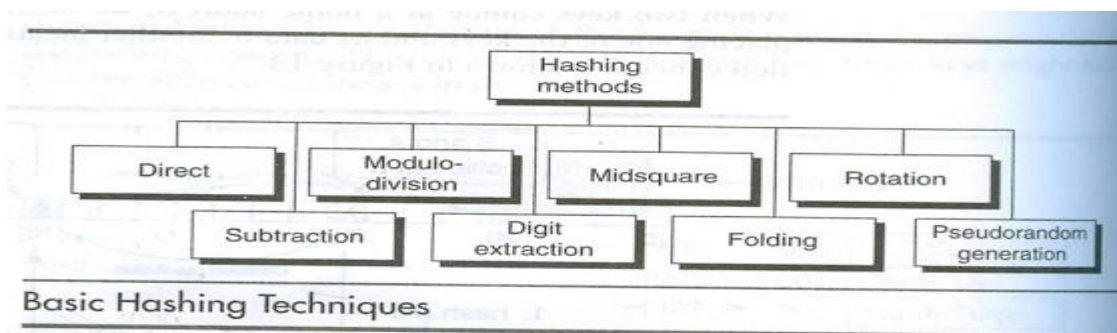


The memory that contains all of the home addresses is known as the **prime area**.

The address produced by the hashing algorithm is known as the **home address**.

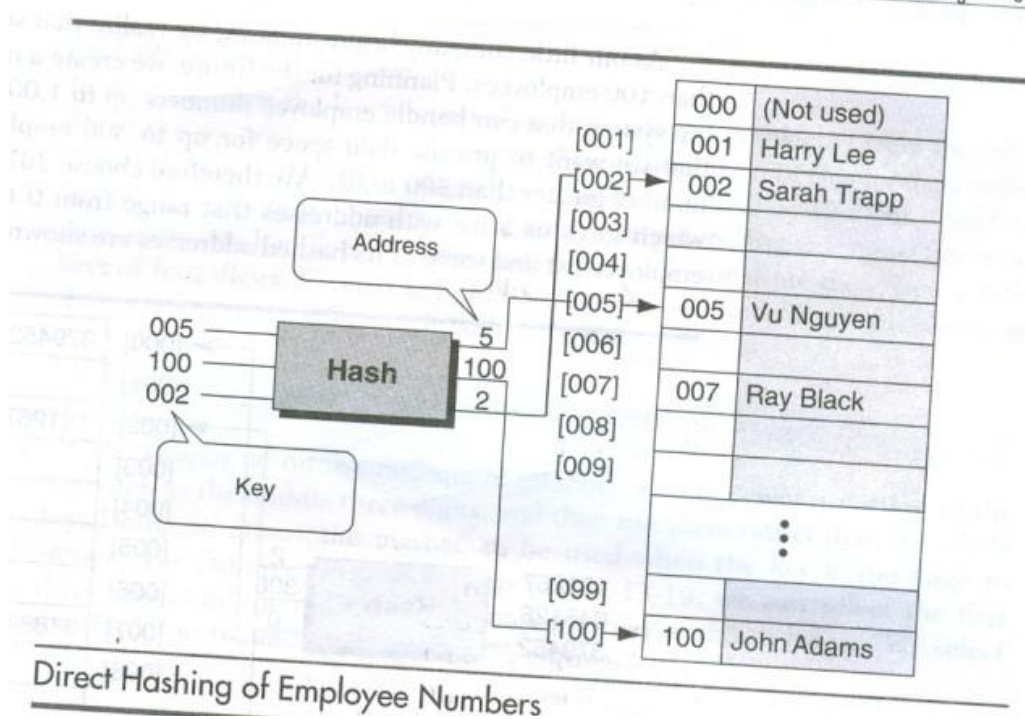
A **collision** occurs when a hashing algorithm produces an address for an insertion key and that address is already occupied.

## 5.7 Hashing Methods



### 5.7.1 Direct Method

In direct hashing the key is the address without any algorithmic manipulation. The data structure must therefore contain an element for every possible key. The situations in which you can use direct hashing are limited.



### 5.7.2 Subtraction Method

Sometimes keys are consecutive but do not start from 1. For example, a company may have only 100 employees, but the employee numbers start from 1001 and go to 1100. In this case we use subtraction hashing, a very simple hashing function that subtracts 1000 from the key to determine the address.

The direct and subtraction hash functions both guarantee a search effort of one with no collisions. They are 'one-to-one hashing methods: only one key hashes to each address.

### 5.7.3 Modulo-division Method

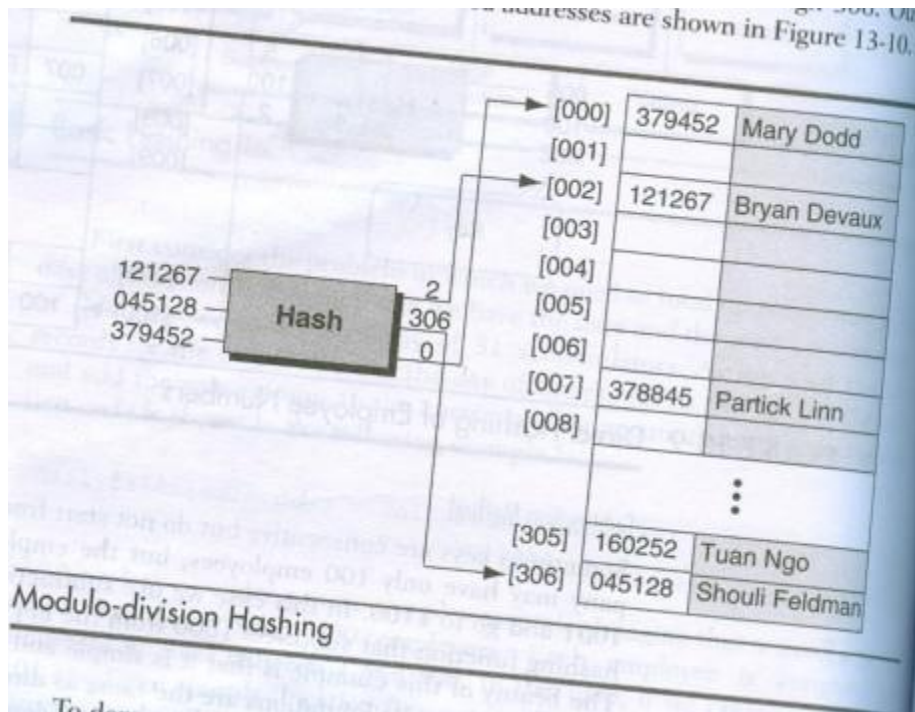
Also known as division remainder, the modulo-division method divides the key by the array size and uses the remainder for the address. This method gives us the simple hashing algorithm shown below in which listSize is the number of elements in the array:

$$\text{address} = \text{key} \text{ MODULO } \text{listSize}$$

$$121267 / 307 = 395 \text{ with remainder of } 2$$

Therefore:  $\text{hash}(121267) = 2$





#### 5.7.4 Digit-extraction Method

Using **digit extraction** selected digits are extracted from the key and used as the address. For example, using our six-digit employee number to hash to a three digit address (000-999), we could select the first, third, and fourth digits (from the left) and use them as the address.

|        |   |     |
|--------|---|-----|
| 379452 | ⇒ | 394 |
| 121267 | ⇒ | 112 |
| 378845 | ⇒ | 388 |
| 160252 | ⇒ | 102 |
| 045128 | ⇒ | 051 |

#### 5.7.5 Mid square Method

In mid square hashing the key is squared and the address is selected from the middle of the squared number.

$$9452^2 = 89340304: \text{ address is } 3403$$

We can select the first three digits and then use the mid square method as shown below.

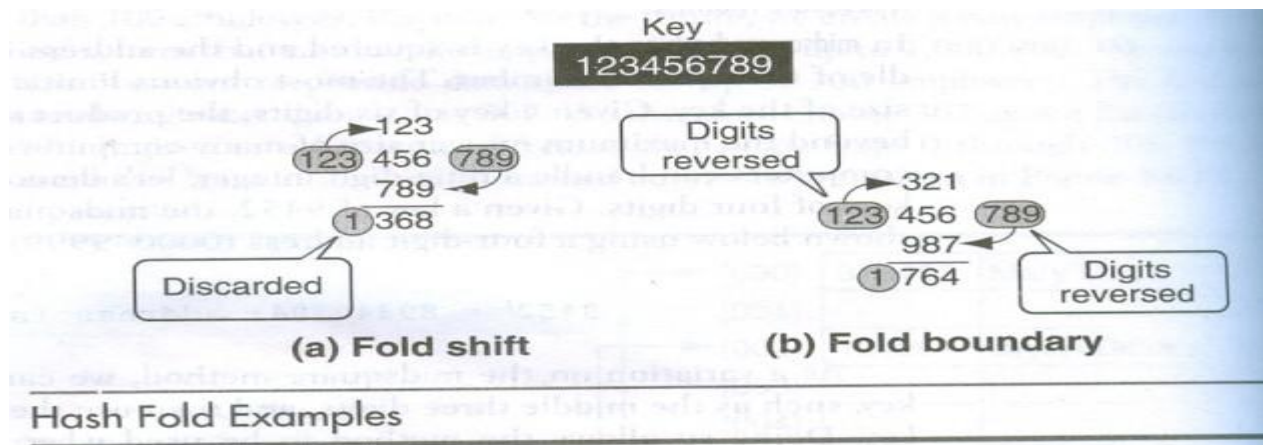
|         |                  |   |    |
|---------|------------------|---|----|
| 379452: | $379^2 = 143641$ | ⇒ | 36 |
| 121267: | $121^2 = 014641$ | ⇒ | 46 |
| 378845: | $378^2 = 142884$ | ⇒ | 28 |
| 160252: | $160^2 = 025600$ | ⇒ | 56 |
| 045128: | $045^2 = 002025$ | ⇒ | 20 |

### 5.7.6 Folding Methods

Two folding methods are used: fold shift and fold boundary.

In **fold shift** the key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with the middle part.

In **fold boundary** the left and right numbers are folded on a fixed boundary between them and the center number.



### 5.7.7 Rotation Method

It is most useful when keys are assigned serially. Rotating the last character to the front of the key minimizes this effect.

|                 |          |                |
|-----------------|----------|----------------|
| 600101          | 600101   | 160010         |
| 600102          | 600102   | 260010         |
| 600103          | 600103   | 360010         |
| 600104          | 600104   | 460010         |
| 600105          | 600105   | 560010         |
| Original<br>key | Rotation | Rotated<br>key |

Rotation Hashing

### 5.7.8 Pseudorandom Hashing

In pseudorandom hashing the key is used as the seed in a pseudorandom-number generator and the resulting random number is then scaled into the possible address range using modulo-division. A common random-number generator is shown below.

$$y = ax + c$$

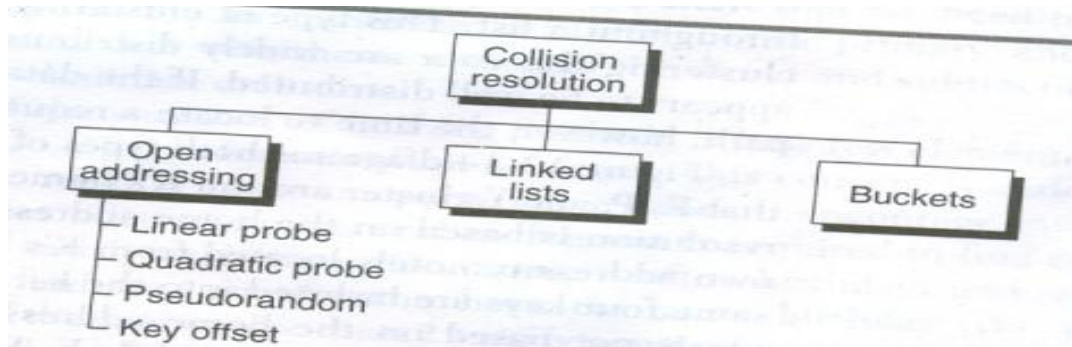
To use the pseudorandom-number generator as a hashing method, we set  $x$  to the key, multiply it by the coefficient  $a$ , and then add the constant  $c$ . The result is then divided by the list size, with the remainder being the hashed address. For maximum efficiency, the factors  $a$  and  $c$  should be prime numbers.

To keep the calculation reasonable, we use 17 and 7 for factors  $a$  and  $c$ , respectively. Also, the list size in the example is the prime number 307.

```
y = ((17 * 121267) + 7) modulo 307
y = (2061539 + 7) modulo 307
y = 2061546 modulo 307
y = 41
```

## 5.8 Collision Resolutions

A **collision** occurs when a hashing algorithm produces an address for an insertion key and that address is already occupied.



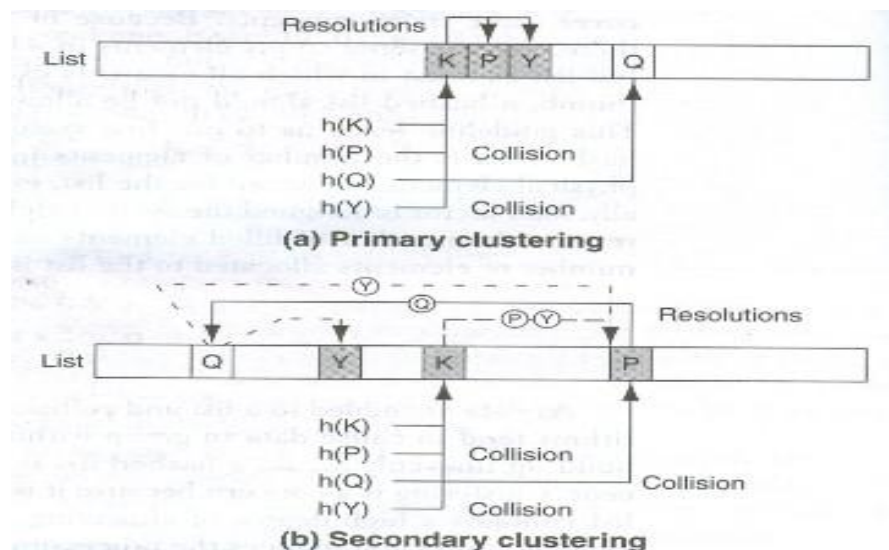
### Concepts

a) The **load factor** of a hashed list is the number of elements in the list divided by the number of physical elements allocated for the list, expressed as a percentage. Traditionally, load factor is assigned the symbol alpha ( $\alpha$ ). The formula in which  $k$  represents the number of filled elements in the list and  $n$  represents the total number of elements allocated to the list is

$$\alpha = \frac{k}{n} \times 100$$

b) Computer scientists have identified two distinct types of clusters.

- Primary clustering occurs when data cluster around a home address. Primary clustering is easy to identify.
- Secondary clustering occurs when data become grouped along a collision throughout a list. This type of clustering is not easy to identify.



### 5.8.1 Open Addressing

The first collision resolution method, open addressing, resolves collisions in the prime area—that is, the area that contains all of the home addresses.

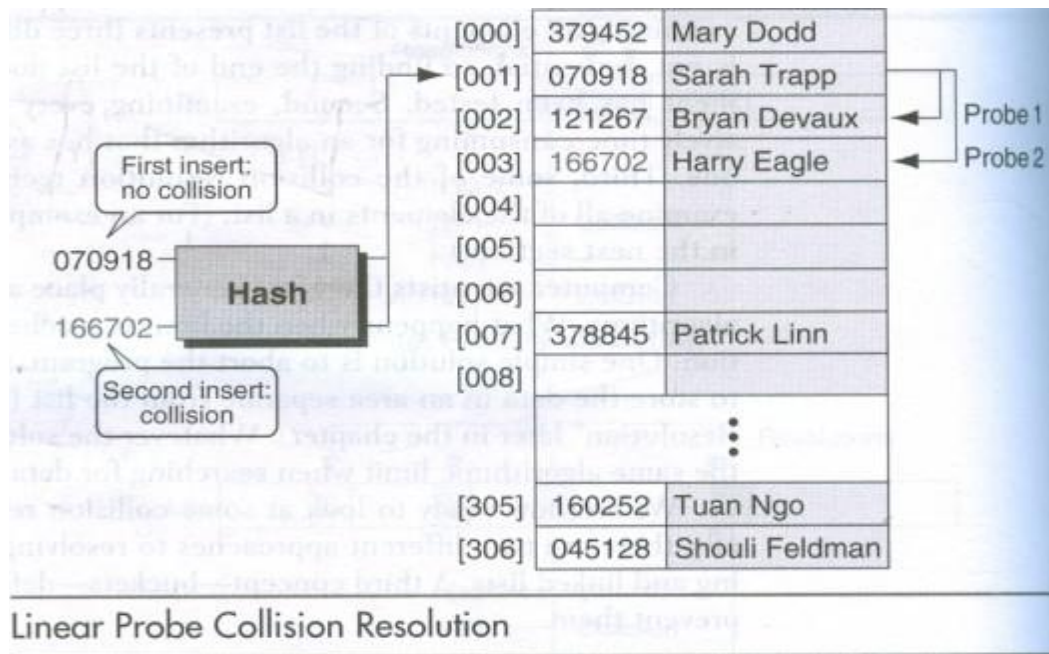
When a collision occurs, the prime area addresses are searched for an 0 or unoccupied element where the new data can be placed.

#### 5.8.1.1 Linear Probe

In a linear probe, which is the simplest, when data cannot be stored in the home address we resolve the collision by adding 1 to the current address.



Linear probes have two advantages. First, they are quite simple to implement. Second, data tend to remain near their home address.



### 5.8.1.2 Quadratic Probe

In the quadratic probe, the increment is the collision probe number squared.

Thus for the first probe we add  $1^2$ , for the second collision probe we add  $2^2$ , for the third collision probe we add  $3^2$ , and so forth until we either find an empty element or we exhaust the possible elements.

| Probe number | Collision location | Probe <sup>2</sup> and increment | New address               |
|--------------|--------------------|----------------------------------|---------------------------|
| 1            | 1                  | $1^2 = 1$                        | $1 + 1 \Rightarrow 02$    |
| 2            | 2                  | $2^2 = 4$                        | $2 + 4 \Rightarrow 06$    |
| 3            | 6                  | $3^2 = 9$                        | $6 + 9 \Rightarrow 15$    |
| 4            | 15                 | $4^2 = 16$                       | $15 + 16 \Rightarrow 31$  |
| 5            | 31                 | $5^2 = 25$                       | $31 + 25 \Rightarrow 56$  |
| 6            | 56                 | $6^2 = 36$                       | $56 + 36 \Rightarrow 92$  |
| 7            | 92                 | $7^2 = 49$                       | $92 + 49 \Rightarrow 41$  |
| 8            | 41                 | $8^2 = 64$                       | $41 + 64 \Rightarrow 05$  |
| 9            | 5                  | $9^2 = 81$                       | $5 + 81 \Rightarrow 86$   |
| 10           | 86                 | $10^2 = 100$                     | $86 + 100 \Rightarrow 86$ |

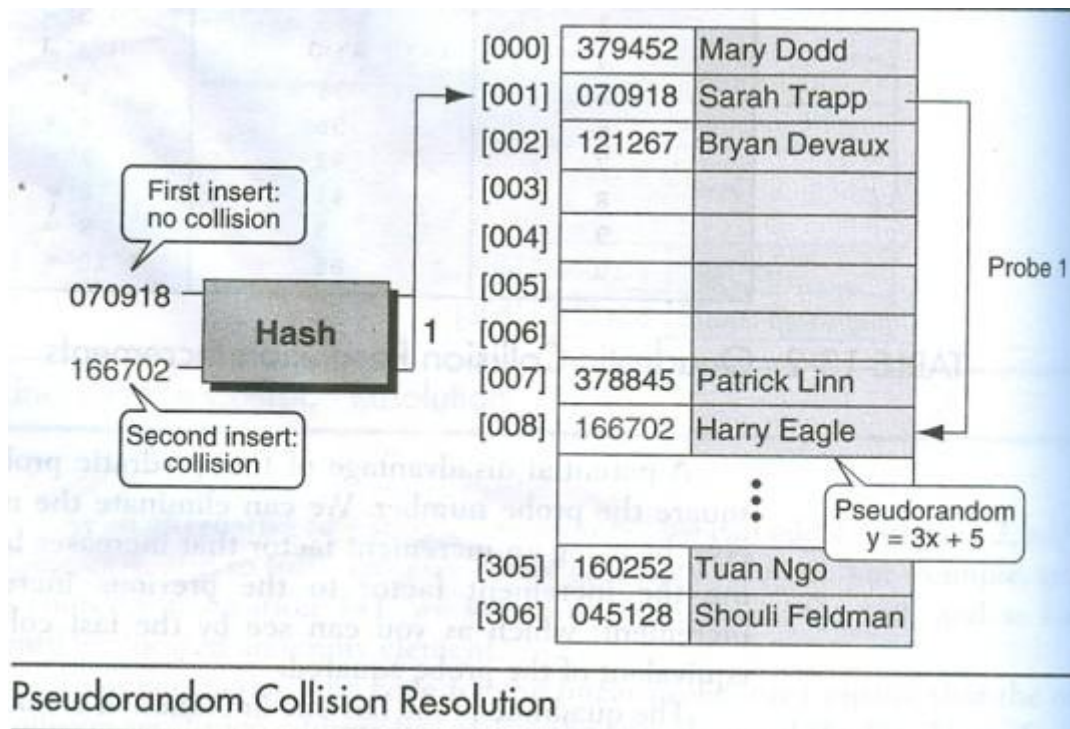
### Quadratic Collision Resolution Increments

### 5.8.1.3 Pseudorandom Collision Resolution

Pseudorandom collision resolution uses a pseudorandom number to resolve the collision. We now use it a collision resolution method. In this case, rather than use the key as a factor in the random-number calculation, we use the collision address.

We now resolve the collision using the following pseudorandom-number generator, where  $a$  is 3 and  $c$  is 5:

$$\begin{aligned}
 y &= (ax + c) \text{ modulo } \text{listSize} \\
 &= (3 \times 1 + 5) \text{ Modulo } 307 \\
 &= 8
 \end{aligned}$$



#### 5.8.1.4 Key Offset

Key offset is a double hashing method that produces different collision paths for different keys. Whereas the pseudorandom-number generator produces a new address as a function of the previous address, key offset calculates the new address as a function of the old address and the key.

One of the simplest versions simply adds the quotient of the key divided by the list size to the address to determine the next collision resolution address, as shown in the formula below.

```
offset = [key/listSize]
address = ((offset + old address) modulo listSize)
```

Example:

```
offset = [166702/307] = 543
address = ((543 + 001) modulo 307) = 237
```

If 237 were also a collision, we would repeat the process to locate the next address, as shown below.

```
offset = [166702/307] = 543
address = ((543 + 237) modulo 307) = 166
```

| Key    | Home address | Key offset | Probe 1 | Probe 2 |
|--------|--------------|------------|---------|---------|
| 166702 | 1            | 543        | 237     | 166     |
| 572556 | 1            | 1865       | 024     | 047     |
| 067234 | 1            | 219        | 220     | 132     |

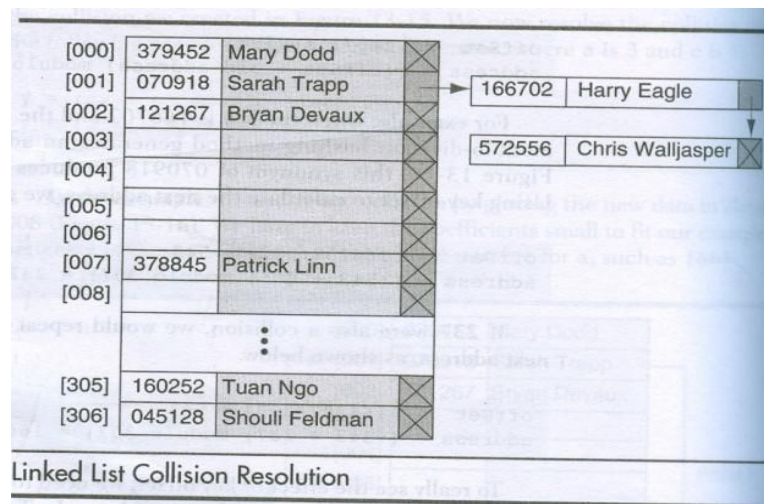
#### 5.8.2 Linked list Collision Resolution

A linked list is ordered collection of data in which each element contains the location of next element.

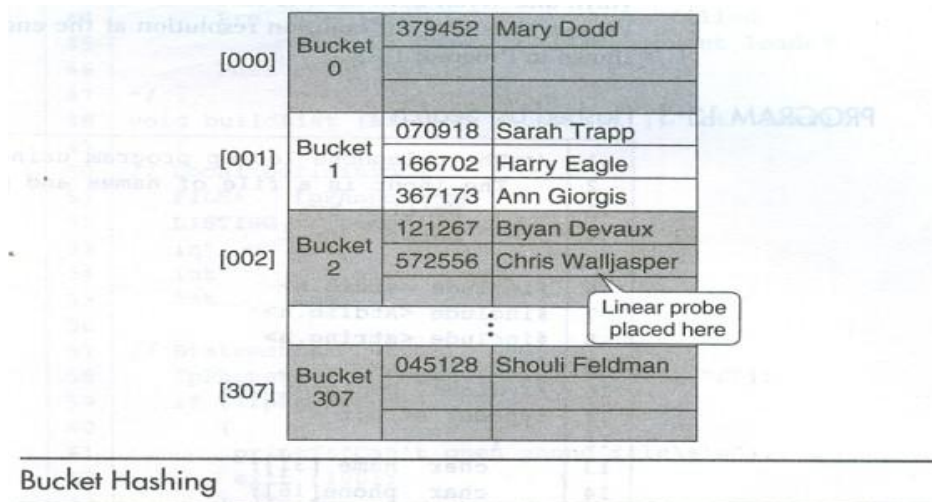
It uses two storage areas: the prime area and the overflow area.

Each element in the prime area contains an additional field-a link head pointer to a linked list of overflow data in the overflow area. When a collision occurs, one element is stored in the prime area and chained to its corresponding linked list in the overflow area.

Although the overflow area can be any data structure, it is typically implemented as a linked list in dynamic memory.



### **5.8.3 Bucket Hashing**



## DYNAMIC HASHING:

Dynamic hashing is also known as extendible hashing, aims to reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only 1 bucket. The objective of the dynamic hashing is to provide acceptable hash table performance on a per operation basis.

For example here, we use a hash function  $h(k)$  that transforms keys into 6-bit non-negative integers. Our example keys will be characters each and  $h$  transforms letters such as A,B and C into bit sequences 100,101, and 110 respectively. Digits 0 through 7 are transformed into their 3-bit representation. For our example hash function,  $h(A0,1)=0$ ,  $h(A1,3)=1$ ,  $h(B1,4)=1001=9$ , and  $h(C1,6)=110001=49$ .



| k  | h(k)    |
|----|---------|
| A0 | 100 000 |
| A1 | 100 001 |
| B0 | 101 000 |
| B1 | 101 001 |
| C1 | 110 001 |
| C2 | 110 010 |
| C3 | 110 011 |
| C5 | 110 101 |

Fig 8.7: An example of Hash Function

### Dynamic Hashing using Directories:

We employ a directory,  $d$ , of pointers to buckets. The size of the directory depends on the number of bits of  $h(k)$  used to index into the directory. When indexing is done using, say,  $h(k,2)$ , the directory size is  $2^2=4$ ; when indexing is done using, say,  $h(k,5)$ , the directory size is  $2^5=32$ . The number of bits of  $h(k)$  used to index the directory is called the *directory depth*. The size of the directory is  $2^t$ , where  $t$  is the directory depth and the number of buckets is at most equal to the directory size. Figure 8.8(a) shows dynamic hash table that contains the keys A0, B0, A1, B1, C2, and C3. This hash table uses directory whose depth is 2 and uses buckets that have 2 slots each other. In below figure directory is shaded while the buckets are not. In practice, the bucket size is often chosen to match some physical characteristic of the storage media.

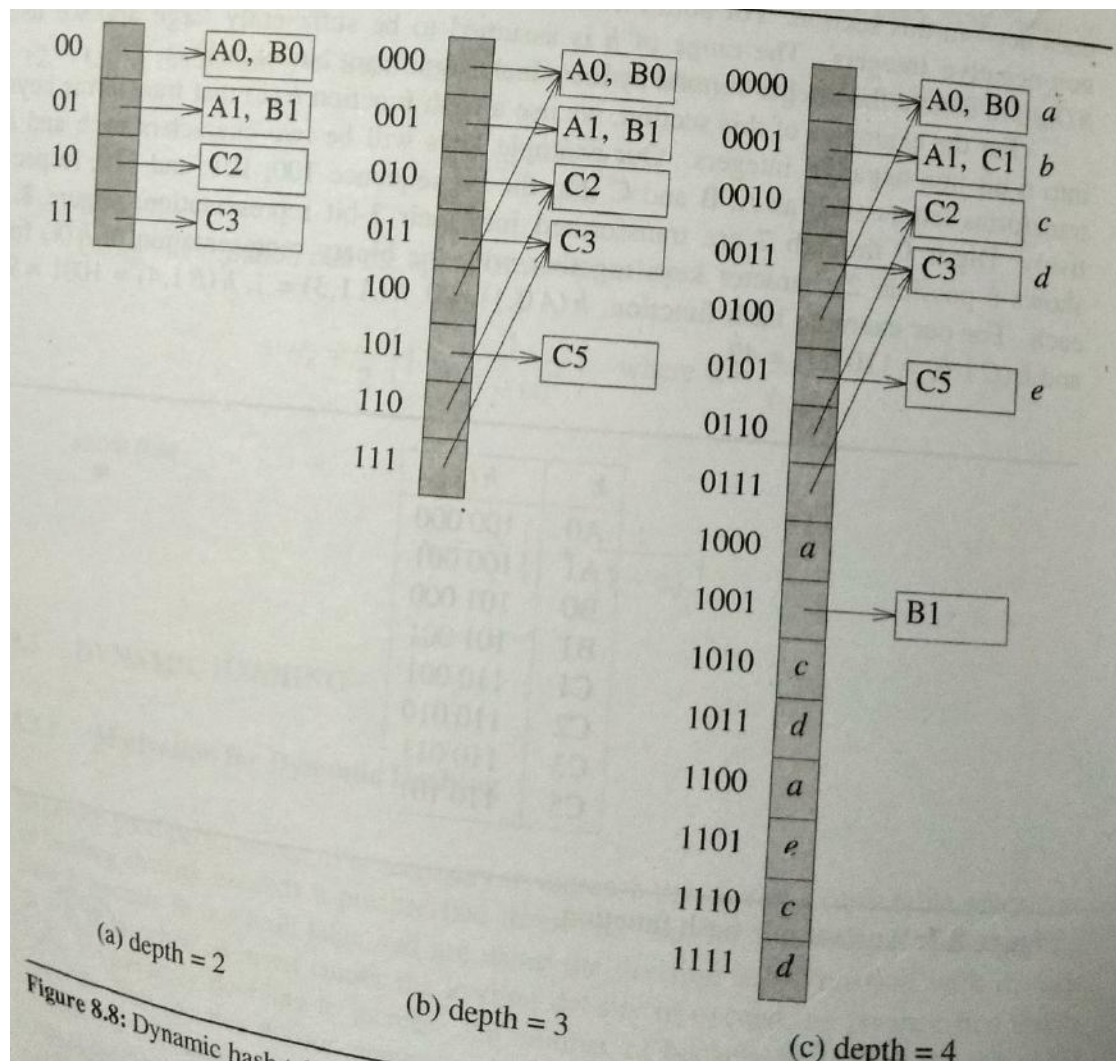


Fig 8.8: Dynamic Hash table with Directories

To search for a key  $k$ , we merely examine the bucket pointed to by  $d[h(k,t)]$ , where  $t$  is the directory depth. Suppose we insert C5 into the hash table of Figure 8.8(a). Since,  $h(C5,2)=01$ , we follow the pointer,  $d[01]$ , in position 01 of the directory. This gets us to bucket with A1, B1. This bucket is full and we get a bucket overflow. To resolve the overflow, we determine the least  $u$  such that  $h(k,u)$  is not the same for all keys in the overflow bucket. In case the least  $u$  is greater than the directory depth, we increase the directory depth to this least  $u$  value. This requires us to increase the directory size but not number of buckets. A quadrupling of the directory size may be handled as two doublings and so on. For our example, the least  $u$  for which  $h(k,u)$  is not the same for A1, B1 and C5 is 3. So, the directory is expanded to have depth 3 and size 8.

Following the resizing of the directory, we split the overflowed bucket using  $h(k,u)$ . In our case, the overflowed bucket is split using  $h(k,3)$ . For A1 and B1,  $h(k,3)=001$  and for C5,  $h(k,3)=101$ . Figure 8.8(b) shows the result. As a final insert example, consider inserting C1 into Fig 8.8(b).  $h(C1,3)=001$ . This time, bucket  $d[001]$  overflows. The minimum  $u$  is 4 and so it is necessary to double the directory size and increase the directory depth to 4. When the directory is doubled, we replicate the pointers in the first half into second half. Next we split the overflowed bucket using  $h(k,4)$ . Since  $h(k,4)=0001$  for A1 and C1 and 1001 for B1, we create a new bucket with B1 and put C1 into the slot previously occupied by B1. A pointer to the new bucket is placed in  $d[1001]$ . Fig 8.8(c) shows the result.

Deletion from a dynamic hash table with a directory is similar to insertion. Although dynamic hashing employs array doubling, the time for this is considerably less than that for the array doubling used in the static hashing.

### Directoryless Dynamic Hashing:

Which also known as linear dynamic hashing, we dispense with the directory,  $d$ . in this method we use an array,  $ht$ , of buckets is used. We assume that this array is as large as possible and so there is no possibility of increasing its size dynamically. To avoid initializing such a large array, we use two variables  $q$  and  $r$ ,  $0 \leq q \leq 2^r$ , to keep track of the active buckets. At any time, only buckets 0 through  $2^r+q-1$  are active. Each active bucket is the start of the chain of buckets. The remaining buckets on chain are called overflow buckets.

*buckets. Informally,  $r$  is the number of bits of  $h(k)$  used to index into the hash table and  $q$  is the bucket that will split next. More accurately, buckets 0 through  $q-1$  as well as buckets  $2^r$  through  $2^r+q-1$  are indexed using  $h(k, r+1)$  while the remaining active buckets are indexed using  $h(k, r)$ . Each dictionary pair is either in an active or an overflow bucket.*

Figure 8.9 (a) shows a directoryless hash table  $ht$  with  $r = 2$  and  $q = 0$ . The hash function is that of Figure 8.7,  $h(B4) = 101\ 100$ , and  $h(B5) = 101\ 101$ . The number of active buckets is 4 (indexed 00, 01, 10, and 11). The index of an active bucket identifies its chain. Each active bucket has 2 slots and bucket 00 contains B4 and A0. There are 4 bucket chains, each chain begins at one of the 4 active buckets and comprises only that active bucket (i.e., there are no overflow buckets). In Figure 8.9 (a), all keys have been mapped into chains using  $h(k, 2)$ . In Figure 8.9 (b),  $r = 2$  and  $q = 1$ ;  $h(k, 3)$  has been used for chains 000 and 100 while  $h(k, 2)$  has been used for chains 001, 010, and 011. Chain 001 has an overflow bucket; the capacity of an overflow bucket may or may not be the same as that of an active bucket.

To search for  $k$ , we first compute  $h(k, r)$ . If  $h(k, r) < q$ , then  $k$ , if present, is in a chain indexed using  $h(k, r+1)$ . Otherwise, the chain to examine is given by  $h(k, r)$ . Program 8.5 gives the algorithm to search a directoryless dynamic hash table.

To insert C5 into the table of Figure 8.9 (a), we use the search algorithm of Program 8.5 to determine whether or not C5 is in the table already. Chain 01 is examined and we verify that C5 is not present. Since the active bucket for the searched chain is full, we get an overflow. An overflow is handled by activating bucket  $2^r + q$ ; reallocating the entries in the chain  $q$  between  $q$  and the newly activated bucket (or chain)  $2^r + q$ .



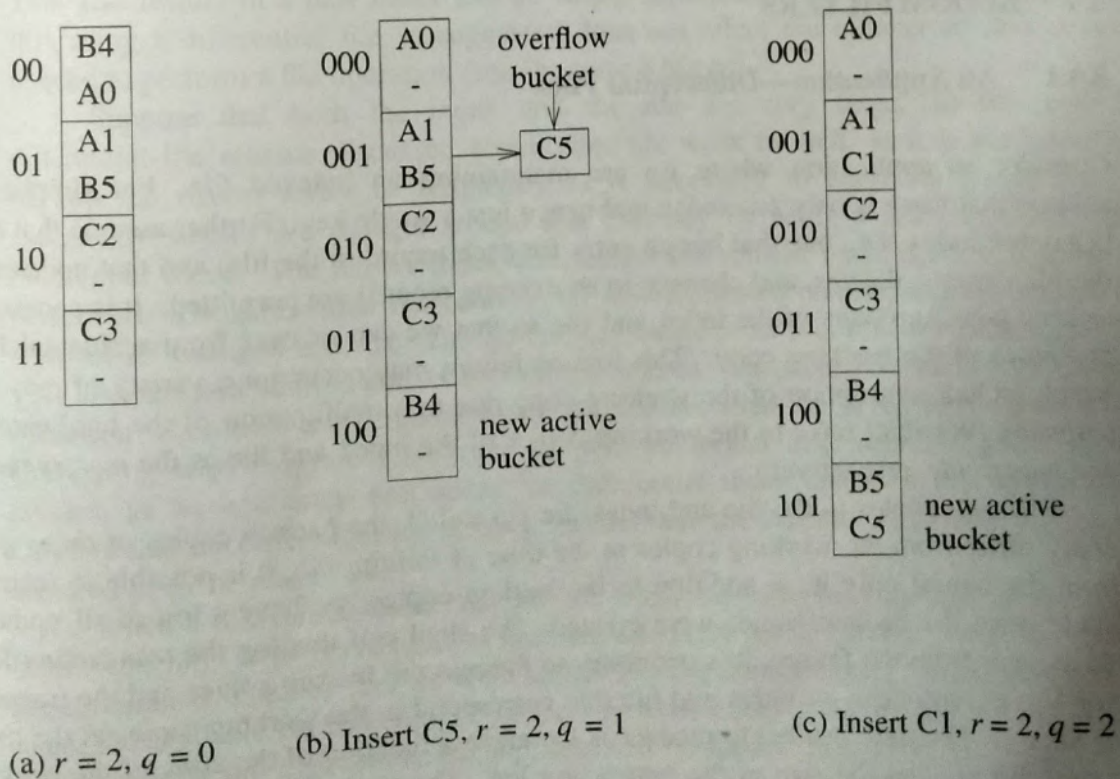
Example Figure is shown below:

if ( $h(k, r) < q$ ) search the chain that begins at bucket  $h(k, r+1)$ ;  
 else search the chain that begins at bucket  $h(k, r)$ ;

**Program 8.5:** Searching a directoryless hash table

and incrementing  $q$  by 1. In case  $q$  now becomes  $2^r$ , we increment  $r$  by 1 and reset  $q$  to 0. The reallocation is done using  $h(k, r+1)$ . Finally, the new pair is inserted into the chain where it would be searched for by Program 8.5 using the new  $r$  and  $q$  values.

For our example, bucket 4 = 100 is activated and the entries in chain 00 ( $q = 0$ ) are rehashed using  $r+1 = 3$  bits. B4 hashes to the new bucket 100 and A0 to bucket 000. Following this,  $q = 1$  and  $r = 2$ . A search for C5 would examine chain 1 and so C5 is added to this chain using an overflow bucket (see Figure 8.9 (b)). Notice that at this time, the keys in buckets 001, 010 and 011 are hashed using  $h(k, 2)$  while those in buckets 000 and 100 are hashed using  $h(k, 3)$ .



**Figure 8.9:** Inserting into a directoryless dynamic hash table

Let us now insert C1 into the table of Figure 8.9 (b). Since,  $h(C1, 2) = 01 = q$ , chain 01 = 1 is examined by our search algorithm (Program 8.5). The search verifies that C1 is not in the dictionary. Since the active bucket 01 is full, we get an overflow. We activate bucket  $2^r + q = 5 = 101$  and rehash the keys A1, B5, and C5 that are in chain  $q$ . The rehashing is done using 3 bits. A1 is hashed into bucket 001 while B5 and C5 hash into bucket 101.  $q$  is incremented by 1 and the new key C1 is inserted into bucket 001. Figure 8.9 (c) shows the result.

## Sequential File Organization

We'll study data structures appropriate for organizing

1. Large quantities of records.
2. Information that may have special space or performance requirements.
3. Information that need to be processed in different ways for different applications.
4. Information that enables new tasks to be accomplished.

Three primary file organizations

1. Sequential
2. Direct
3. Indexed Sequential

There is a distinction between how we store or *organize* information and how we process or *access* it.

| Organization       | Access                |
|--------------------|-----------------------|
| Sequential         | Sequential            |
| Indexed sequential | Sequential and direct |
| Direct             | Direct                |

Sequential access refers to accessing *multiple* records, often an entire file, whereas direct access refers to locating a *single* record.

We have records which contain fields.



An example employee record:

employee record

|               |        |       |       |         |        |              |
|---------------|--------|-------|-------|---------|--------|--------------|
| employee name | number | title | dept. | manager | salary | other fields |
|---------------|--------|-------|-------|---------|--------|--------------|

- The *primary key* is a field, or a composite of several fields, which uniquely distinguishes a record from all others;
- All the remaining fields are the *secondary keys* or attributes.
- A *file* consists of records of the same format.

The sizes of the fields within a record may vary

- For fixed-length records, all the records within a file have the same length.
- With variable-length records, records within a file do not need to be the same length.



| Employee name    | Number | Title      | Department | Manager          | Salary  |   |
|------------------|--------|------------|------------|------------------|---------|---|
| Scott Matthews   | 0123   | Programmer | Accounting | Melissa Jones    | 0026000 | . |
| Tammy Boger      | 0240   | Analyst    | Data proc. | Morris Lancaster | 0032000 | . |
| Morris Lancaster | 0067   | Manager    | Data proc. | Charles Hall     | 0045000 | . |
| Larry Cookman    | 0189   | Programmer | Security   | John Bittle      | 0025000 | . |
| David Caudle     | 0423   | Analyst    | Accounting | Melissa Jones    | 0047000 | . |
| Richard Lehner   | 0847   | Instructor | Training   | Tom Nelson       | 0031000 | . |
| Robert Blom      | 1002   | Programmer | Accounting | George Steel     | 0027000 | . |
| Nancy White      | 0417   | Programmer | Operations | Curt Alexander   | 0028000 | . |
| Randy Wheeler    | 0293   | Analyst    | Data proc. | William Crocker  | 0031500 | . |
| John Bittle      | 0367   | Manager    | Security   | Robert Wilson    | 0043000 | . |

File of records.

## Organization

|   |   |   |   |   |   |   |       |       |   |   |   |       |   |
|---|---|---|---|---|---|---|-------|-------|---|---|---|-------|---|
| 1 | 2 | 3 | . | . | : | i | i + 1 | i + 2 | . | . | . | n - 1 | n |
|---|---|---|---|---|---|---|-------|-------|---|---|---|-------|---|

- Sequential organization is well suited for sequential access in which many or all the records will be processed.
- Can be processed easily with a loop. A **sequential search** processes the records of a file in their order of occurrence until it either locates the desired record or processes all the records.

### Binary Search

- In a file of  $n$  records,  $n/2$  probes are needed to locate the record of interest, on the average.

#### Algorithm BINARY SEARCH

```
proc binary__search
 /* The n records of the file are ordered in in-
 creasing order of the keys. */
1 LOWER := 1
2 UPPER := n
3 while LOWER ≤ UPPER do
4 MIDDLE := ⌊ (LOWER + UPPER) / 2 ⌋
5 if key[sought] = key[MIDDLE]
6 then terminate successfully.
7 else if key[sought] > key[MIDDLE]
8 then LOWER := MIDDLE + 1
9 else UPPER := MIDDLE - 1
10 end
11 terminate unsuccessfully.
end binary__search
```

- For an unsuccessful search we need to probe the entire file of  $n$  records.
- Appropriate when  $n$  is small.
- To improve performance, we can *sort* the records in the file. Then only  $n/2$  records need to be examined, for when we go beyond the position, we end the search.
- Example: Assume, we are searching for the record with key 39.

|      |     |     |     |                  |      |                  |      |     |
|------|-----|-----|-----|------------------|------|------------------|------|-----|
| [13, | 16, | 18, | 27, | 28,              | 29,  | 38,              | 39,  | 53] |
| 13,  | 16, | 18, | 27, | <sup>^</sup> 28, | [29, | 38,              | 39,  | 53] |
| 13,  | 16, | 18, | 27, | 28,              | 29,  | <sup>^</sup> 38, | [39, | 53] |

- The complexity of both successful and unsuccessful search is  $O(\log n)$ . Also referred to as “logarithmic search” or “bisection”.
- As  $n$  becomes large,  $\log n$  becomes significant. We want to be able to retrieve a record in a single access.
- The pre processing cost of ordering a file and the continuing cost of maintaining the order must also be considered in binary search.
- Another example: retrieve a record with the key 17:

|      |                  |                   |      |                  |     |     |     |     |
|------|------------------|-------------------|------|------------------|-----|-----|-----|-----|
| [13, | 16,              | 18,               | 27,  | 28,              | 29, | 38, | 39, | 53] |
| [13, | 16,              | 18,               | 27], | <sup>^</sup> 28, | 29, | 38, | 39, | 53  |
| 13,  | <sup>^</sup> 16, | [18,              | 27], | 28,              | 29, | 38, | 39, | 53  |
| 13,  | 16],             | <sup>^</sup> [18, | 27,  | 28,              | 29, | 38, | 39, | 53  |

## Interpolation Search

- Consider finding a phone number for Phyllis Bishop. Do you begin your search at the middle of the book? Most likely not.
- You try to approximate its relative position.
- You would probably begin searching Bishop near the front of the book.
- An *interpolation search* chooses the next position for a comparison based on the estimated position of the sought key relative to the remainder of the file to be searched.
- Instead of computing a MIDDLE position, it calculates the next position as follows:

$$\text{NEXT} = \left\lceil \text{LOWER} + \frac{\text{key}[\text{sought}] - \text{key}[\text{LOWER}]}{\text{key}[\text{UPPER}] - \text{key}[\text{LOWER}]} (\text{UPPER} - \text{LOWER}) \right\rceil$$

$\lceil \quad \rceil$  denotes “ceiling”.

- Although the worst case computational complexity is  $O(n)$ , the average complexity is  $O(\log \log n)$ .
- It improves as the distribution of the keys becomes more uniform
- A binary search is preferable when the data is stored in primary memory because the additional calculations needed for the interpolation search cancel any savings.
- When the data is stored in auxiliary memory and the key distribution is uniform, an interpolation search is preferable.

### Self Organizing Sequential Search

- In the previous searches, we assumed that all records in the file would be accessed equally often. If this assumption is not true, we would desire the most frequently retrieved records to appear at the beginning of the file.
- Three popular algorithms:

(1) Move\_to\_front, (2) Transpose, (3) Count.

#### **1) Move\_to\_front**

When the sought record is located, it is moved to the front position of the file and all the intervening records are moved back one position. This is an extensive amount of re positioning. So, a linked implementation is preferable even though it takes more storage. It is a mistake if a record is accessed, moved to the front and rarely if ever accessed again.

**Locality** means that a record that has recently been accessed is more likely to be accessed again in the near future. All the self-organizing methods assume some degree of locality of accesses. Move\_to\_front is appropriate when space is not limited and locality of access is important.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| t |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| i | l | i | f | a | b | c | d | e | g | h | j | k | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| m |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| e | a | g | r | o | e | l | i | f | b | c | d | h | j | k | m | n | p | q | s | t | u | v | w | x | y | z |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| i | n | o | i | t | a | z | g | r | e | l | f | b | c | d | h | j | k | m | p | q | s | u | v | w | x | y |

An example: The file consists of 26 records containing the letters of the alphabet. The records are accessed in the order of “file organization”.

## 2)Transpose

This algorithm interchanges the sought record with its immediate predecessor unless the sought record is in the first position. Converges to a near steady state more slowly than the Move\_to\_front algorithm. More stable. It does not make big mistakes. Does not need the additional space required for a linked implementation. It should be used when space is at a premium.

## 3)Count

Keeps a count of the number of accesses of each record. The record is moved in the file to a position in front of all the records with fewer accesses. The file is then always ordered in a decreasing order of frequency of access. Disadvantages: Requires extra storage to keep the count and does not handle the locality of access phenomenon well. Because of its storage requirements, use it only when the counts are needed for another purpose.

### Locating Information

Ways to organize a file for direct access:

The key is a unique address.

The key converts to a unique address.

The key converts to a probable address (hashing).

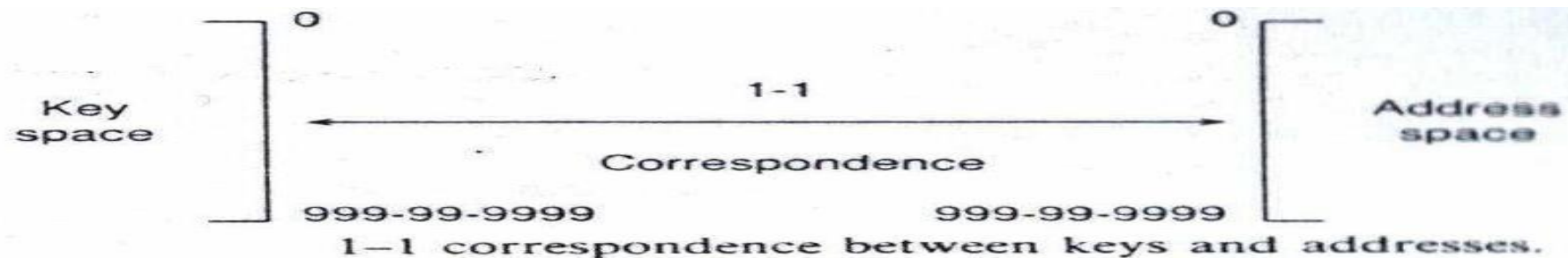
### The Key is a unique address

We want to go *directly* to the address where the record is stored. This is possible if the key were also an address.

If the employee's nine-digit social security number were the record's primary key, we may have one probe per retrieval using a table of size  $10^9$ .

The disadvantage is that a great deal of space must be reserved. One location for each *possible* social security number.

But after employees began leaving the company, gaps would begin to appear.



### The Key converts to a unique address

An algorithm to convert the primary key field (possibly a composite) into a unique storage address.

An example of conversion:

Location  $A[i] = \alpha + m * s$  where  $A$  = an array

$i$  = typical element

$\alpha$  = location of first element in array  $m$  = number of elements preceding  $i$ th element.

$s$  = size of an element

- Another example of conversion:

Flights are numbered from 1 to 999.

Days are numbered from 1 to 366.

Flight number and day of the year could be *concatenated* to determine the location.

Location = flight number || day\_of\_the\_year

The addresses would range from 001001 to 999366.

We need only about one-third as much storage if:

Location = day\_of\_the\_year || flight number

It is preferable to place the wider range last so that we do not have large gaps in the key values.

Since the airline probably would not have 999 flights, many potential numbers would still go unused.

### The Key converts to a Probable Address(HASHING)

- If we remove most of the empty spaces in the address space, we have a key space larger than the address space.
- We need a function for mapping. Such functions are referred to as **hashing functions**.

Hash(key) @ probable address

- This initial probable address is known as the **home address** for the record.





A good function:

Evenly distributes the keys among the addresses

To reduce the number of collision

Executes efficiently

To keep the retrieval time to a minimum.

A collision occurs when two distinct keys map to the same address. Hashing is composed of two aspects:

The function

The collision resolution method

**Key mod  $N$**  (where  $N$  is the table size).

**Key mod  $P$**  (where  $P$  is the smallest prime number  $\geq N$ )

**Truncation** (or sub stringing): If we have social security numbers as the primary keys such as 123-45-6789, the digits may come from *any* part of the key.

**Truncation:** Would it be reasonable to truncate the social security number key after the first three digits?

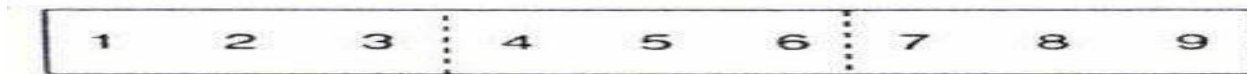
Depends on the application. If we are applying this function to employees who work at a single plant, it would not be a good choice. Numbers are assigned by geographic regions, most of the workers would have the same leftmost two or three digits.

A better choice would be to use the three rightmost digits.

**Folding:** Two kinds of folding exist:

(1) Folding by boundary, (2) Folding by shifting

If we have the nine-digit key



Folding

Fold the number on the dashed boundaries into three parts. The digits would then be superimposed as

$$\begin{array}{r} \phantom{+} \phantom{0} \phantom{0} \phantom{0} \\ \phantom{+} \phantom{0} \phantom{0} \phantom{0} \\ \phantom{+} \phantom{0} \phantom{0} \phantom{0} \\ + \phantom{0} \phantom{0} \phantom{0} \phantom{0} \\ \hline \end{array} \begin{array}{r} 3 \phantom{0} \phantom{0} \phantom{0} \\ 4 \phantom{0} \phantom{0} \phantom{0} \\ 9 \phantom{0} \phantom{0} \phantom{0} \end{array} \begin{array}{r} 2 \phantom{0} \phantom{0} \phantom{0} \\ 5 \phantom{0} \phantom{0} \phantom{0} \\ 8 \phantom{0} \phantom{0} \phantom{0} \end{array} \begin{array}{r} 1 \phantom{0} \phantom{0} \phantom{0} \\ 6 \phantom{0} \phantom{0} \phantom{0} \\ 7 \phantom{0} \phantom{0} \phantom{0} \end{array}$$

Codes were added together (without carry) to obtain the probable address of 654.

Folding

Folding by shifting slides the parts one over another until all are superimposed.

$$\begin{array}{r} \phantom{+} \phantom{0} \phantom{0} \phantom{0} \\ \phantom{+} \phantom{0} \phantom{0} \phantom{0} \\ \phantom{+} \phantom{0} \phantom{0} \phantom{0} \\ + \phantom{0} \phantom{0} \phantom{0} \phantom{0} \\ \hline \end{array} \begin{array}{r} 1 \phantom{0} \phantom{0} \phantom{0} \\ 4 \phantom{0} \phantom{0} \phantom{0} \\ 7 \phantom{0} \phantom{0} \phantom{0} \end{array} \begin{array}{r} 2 \phantom{0} \phantom{0} \phantom{0} \\ 5 \phantom{0} \phantom{0} \phantom{0} \\ 8 \phantom{0} \phantom{0} \phantom{0} \end{array} \begin{array}{r} 3 \phantom{0} \phantom{0} \phantom{0} \\ 6 \phantom{0} \phantom{0} \phantom{0} \\ 9 \phantom{0} \phantom{0} \phantom{0} \end{array}$$

Which would yield the different address of 258.

- We could also have added with carry. The results would be (1)764 and (1)368 respectively.

- **Squaring** (a key and then substringing or truncating a portion of the result is another way).
- **Radix Conversion** (the key is considered to be in a base other than 10 and is then converted into a number in base 10. For example the key 1234 in base 11 would become 1610 in base 10. Substringing or truncation could then be used).

### Polynomial Hashing

The function

$f(\text{information area}) \oplus \text{cyclic check bytes}$

is in fact a hashing function. The key is divided by a polynomial.

### Alphabetic Keys

Alphabetic or alphanumeric key values can be input to a hashing function if the values are interpreted as integers. All information is represented internally in a computer as bits.

Variant records or “unions” may be used.

- One hashing function may distribute the keys more evenly than another. One strategy for finding such a hashing function would be to combine simpler functions.
- A hashing function that has a large number of collisions is said to exhibit **primary clustering**.
- Secondary memory is slower. It is better to have a slightly more expensive hashing function if it reduces the number of collisions.

One method for reducing collisions is to change hashing functions.

Another method is to reduce the **packing factor**.

Packing factor = number of records / total number of storage locations

Other names for it: packing density and load factor. As the packing factor increases, the likelihood of a collision increases. The disadvantage of decreasing the packing factor to reduce the number of collisions is that a lower packing factor takes more space.

Changing the hashing function or decreasing the packing factor may reduce the number of collisions but not eliminate them.

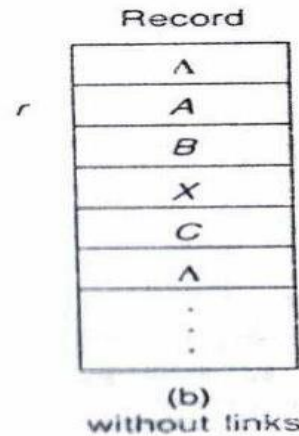
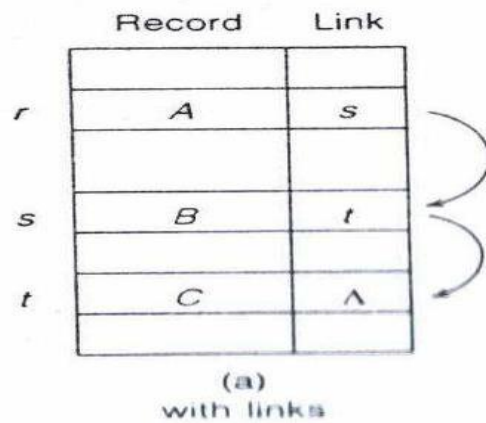
We need a procedure to position a synonym when a collision occurs. The method requires a minimum number of additional probes from its home address.

One way is to *point* to the location of the synonym record.

We form a chain of synonym records. The NULL pointer indicates the end of the synonym chain.

A disadvantage is that storage is needed for the link fields.

- We need a mechanism that does not involve the use of actual links.
- We can use *implied* links by applying a convention, or set of rules. We compute the next search address by applying the set of rules.
- A simple convention is to look at the next location. If it is occupied, we then examine the following location.
- The disadvantage is that we may need to make more probes.



out links.

Synonym chains: with and with-

There are several mechanisms for resolving collisions grouped according to the mechanism used to locate the next address to search and the attribute of whether a record once stored can be relocated;

Collision resolution with links

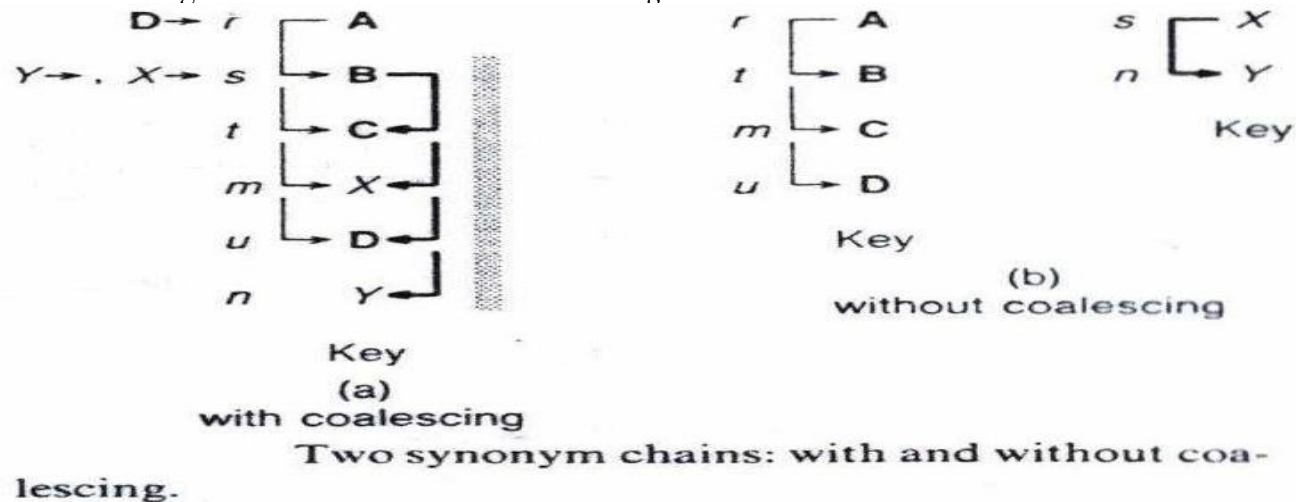
Collision resolution without links

Static positioning of records.

Dynamic positioning of records

Collision resolution with pseudo links

- Uses pointers to connect the elements of a synonym chain.
- The insertion of a synonym is the same as the insertion into a linked list.
- On retrieval, the elements of the synonym chain are searched until the desired record is located or until the end indicated by NULL is reached.
- “Coalescing” occurs when we attempt to insert a record with a home address that is already occupied by a record from a chain with a *different* home address.
- The two chains with records having different home addresses **coalesce** or grow together.
- Coalesced hashing is also referred to as *direct chaining*.



A, B, C, D form one set of synonyms and X, Y form another

In (a), when we attempt to insert the record  $X$ , it collides with the record  $B$  which is *not* a synonym of  $X$  since  $B$  has a home address of  $r$  and  $X$  has a home address of  $s$ .

The two chains coalesce where the bar on the right indicates the portion of the chain in which coalescing has occurred,  $|$  represents insertions on the synonym chain with  $r$  as its home address, and  $|$  represents those with  $s$  as its home address.

This will result in more probes than would be required for non coalesced chains.

When  $X$  is inserted, it is inserted at the *end* of the chain. Instead of one, three probes are needed for  $X$ .

- In (a), 19 probes in total is needed to retrieve each record once, in (b) where two synonym chains were separated, 13 probes.
- Insertion of a new record is made at the bottommost (highest address) empty location. An available space pointer is continually decremented until either an empty location is found or table is full.
- The available space pointer is initially set to a location one record length beyond the last storage location in the table.

#### Algorithm 3.1 COALESCED HASHING

- 
- I. Hash the key of the record to be inserted to obtain the home address, or the probable address for storing the record.
  - II. If the home address is empty, insert the record at that location, else if the record is a duplicate, terminate with a "duplicate record" message, else
    - A. Locate the last record in the probe chain by following the link fields until encountering a  $\Lambda$  link that signifies the end of the chain.
    - B. Find the bottommost empty location in the table. If none is found, terminate with a "full table" message.
    - C. Insert the record into the identified empty location and set the link field of the record at the end of the chain to point to the location of the newly inserted record.
-

For the collision resolution examples of this chapter we use

$$\text{Hash}(\text{key}) = \text{key} \bmod 11$$

Where 11 is the prime number table size.

For comparison purposes, we use the same initial subset of data with the keys of

27, 18, 29, 28, 39, 13, and 16

When we insert 29, there is a collision.

The available space pointer, R is decremented from the bottom to check if location 10 is available.

We set the link field in location 7 to point to location 10.

|              | Record | Link |
|--------------|--------|------|
| 0            |        | Λ    |
| 1            |        | Λ    |
| 2            |        | Λ    |
| 3            |        | Λ    |
| 4            |        | Λ    |
| 5            | 27     | Λ    |
| 6            | 28     | 9    |
| 7            | 18     | 10   |
| 8            |        | Λ    |
| <i>R</i> → 9 | 39     | Λ    |
| 10           | 29     | Λ    |

|              | Record | Link |
|--------------|--------|------|
| 0            |        | Λ    |
| 1            |        | Λ    |
| 2            | 13     | Λ    |
| 3            |        | Λ    |
| 4            |        | Λ    |
| 5            | 27     | 8    |
| 6            | 28     | 9    |
| 7            | 18     | 10   |
| <i>R</i> → 8 | 16     | Λ    |
| 9            | 39     | Λ    |
| 10           | 29     | Λ    |



|                   | Record | Link      |
|-------------------|--------|-----------|
| 0                 |        | $\Lambda$ |
| 1                 |        | $\Lambda$ |
| 2                 | 13     | $\Lambda$ |
| $R \rightarrow$ 3 | 17/6   | $\Lambda$ |
| 4                 | 42/7   | 3         |
| 5                 | 27     | 8         |
| 6                 | 28/6   | 9         |
| 7                 | 18     | 10        |
| 8                 | 16     | $\Lambda$ |
| 9                 | 39/6   | 4         |
| 10                | 29     | $\Lambda$ |

ing.

- If we add two additional records with keys of 42 and 17 and home addresses of 9 and 6 respectively, we will have coalescing of the chains with these two home addresses.
- If we add records to both of the probe chains the performance of each chain would be degraded.

Insertion using coalesced hash-

- One method of reducing the coalescing is to reduce the packing factor. The lower it is, the less chance that there will be a collision from one chain with that of another.
- Also there are variants of coalesced hashing to reduce coalescing.
- An unsuccessful search takes more time since we need to examine more records.
- The number of retrieval probes is a function of the *order of insertion*. A record inserted early in the process will be placed near the beginning.
- So if the frequencies are known, it is good to place the most frequently accessed records early in the insertion process.
- We cannot merely remove the record as if we were deleting an element from a singly linked list.
- We might lose the remainder of a probe chain that had coalesced as a result of being its home address.
- A simple deletion procedure is to move a record later in the chain into the position of the deleted record. The relocated record should be the last element on the chain having the home address of the location of the deleted record.
- This relocation process is repeated at the vacated location until no other records need to be moved.
- Then, prior-to-moving predecessor of the most recently moved record is re linked to the moved record's prior-to-moving successor.
- A more complex scheme could be used to reposition records in the coalesced chains to reduce the amount of coalescing.
- Determining if coalescing has occurred in a probe chain may be time-consuming.
- If we know that a coalescing has not occurred, we just remove the deleted record from a singly linked list.
- Deleted record is reinitialized to null, the available space pointer,  $R$ , is reset to the maximum of (1) its current address and (2) the vacated address plus one.

|                   | Record | Link      |
|-------------------|--------|-----------|
| 0                 |        | $\Lambda$ |
| 1                 |        | $\Lambda$ |
| 2                 | 13     | $\Lambda$ |
| $R \rightarrow$ 3 | 17/6   | $\Lambda$ |
| 4                 | 42/7   | 3         |
| 5                 | 27     | 8         |
| 6                 | 28/6   | 9         |
| 7                 | 18     | 10        |
| 8                 | 16     | $\Lambda$ |
| 9                 | 39/6   | 4         |
| 10                | 29     | $\Lambda$ |

ing.

To delete the record with key 39 requires that

- Because coalescing has occurred, we move the last element in the chain with a home address of 9 (record 42) into location 9 and adjust the links and table entries.
- R is set to location 5 (the vacated address plus one).

Insertion using coalesced hash-

|       | Record | Link |
|-------|--------|------|
| 0     |        | Λ    |
| 1     |        | Λ    |
| 2     | 13     | Λ    |
| 3     | 17     | Λ    |
| 4     |        | Λ    |
| R → 5 | 27     | 8    |
| 6     | 28     | 9    |
| 7     | 18     | 10   |
| 8     | 16     | Λ    |
| 9     | 42     | 3    |
| 10    | 29     | Λ    |

To reduce coalescing, the variants may be classified in three ways:

The table organization (whether or not a separate overflow area is used)

The manner of linking a colliding item into a chain.

The manner of choosing on occupied locations

- In coalesced hashing, storage is needed for link fields. When this storage is not available, we need a convention for where to search next.
- **Progressive overflow (linear probing)** is one convention. If a location is occupied, we look at the next location to see if it is empty. Table is circular. We continue until we find an empty slot or we encounter the home address of the record a second time (table is full).
- For retrieval, we follow the same process.
- Performance is poor for an unsuccessful search.

| Key |    |
|-----|----|
| 0   |    |
| 1   |    |
| 2   |    |
| 3   |    |
| 4   |    |
| 5   | 27 |
| 6   |    |
| 7   | 18 |
| 8   | 29 |
| 9   |    |
| 10  |    |

(29 overflows from location 7)

Insert 27, 18, 29, 28, 39, 13, 16

Hash(key) = key mod 11

| Key |    |
|-----|----|
| 0   |    |
| 1   |    |
| 2   |    |
| 3   |    |
| 4   |    |
| 5   | 27 |
| 6   | 28 |
| 7   | 18 |
| 8   | 29 |
| 9   | 39 |
| 10  |    |

| Key |    |
|-----|----|
| 0   |    |
| 1   |    |
| 2   | 13 |
| 3   |    |
| 4   |    |
| 5   | 27 |
| 6   | 28 |
| 7   | 18 |
| 8   | 29 |
| 9   | 39 |
| 10  | 16 |

- **Secondary clustering** is the consequence of two or more records following the same sequence or probe addresses. It results in a bunching of records within the table.
- Contrasts with primary clustering which occurs when a large number of records have the same home address.
- In the example, the secondary clustering was caused by records with *different* home addresses.

#### Progressive Overflow - Deletion

|    | Key |
|----|-----|
| 0  |     |
| 1  |     |
| 2  | 13  |
| 3  |     |
| 4  |     |
| 5  | 27  |
| 6  | 28  |
| 7  | ◆   |
| 8  | 29  |
| 9  | 39  |
| 10 | 16  |

◆ represents a tombstone.

- To *delete*, an indicator called a **tombstone**, in the location of the deleted record is put.
- It tells us that additional records may follow and to keep on searching on a retrieval, or that the location may be filled on a subsequent insertion.
- On the left, the record (18) is deleted.

## Use of Buckets

**Bucket** (block or page) is a storage unit of multiple records at one file address.

Also the unit in which information is accessed and transferred between storage devices.

The number of records that may be stored in a bucket is called the **blocking factor**. As the blocking factor increases, the number of auxiliary storage accesses decreases.

Within a bucket, we need a means of separating the individual records. We can achieve this by knowing the record length for fixed-length records, or by placing a special delimiter.

Buckets are used to improve any collision res. methods.

### Use of Buckets - Example

Blocking factor 2.  $\text{Hash}(\text{key}) = \text{key} \bmod 11$  The keys are 27, 18, 29, 28, 39, 13, 16

|    | Key <sub>1</sub> | Key <sub>2</sub> |
|----|------------------|------------------|
| 0  |                  |                  |
| 1  |                  |                  |
| 2  |                  |                  |
| 3  |                  |                  |
| 4  |                  |                  |
| 5  | 27               |                  |
| 6  |                  |                  |
| 7  | 18               | 29               |
| 8  |                  |                  |
| 9  |                  |                  |
| 10 |                  |                  |

|    | Key <sub>1</sub> | Key <sub>2</sub> |
|----|------------------|------------------|
| 0  |                  |                  |
| 1  |                  |                  |
| 2  | 13               |                  |
| 3  |                  |                  |
| 4  |                  |                  |
| 5  | 27               | 16               |
| 6  | 28               | 39               |
| 7  | 18               | 29               |
| 8  |                  |                  |
| 9  |                  |                  |
| 10 |                  |                  |



**TABLE 3.2 MEAN NUMBER OF PROBES FOR SUCCESSFUL LOOKUP, PROGRESSIVE OVERFLOW, VARIOUS BUCKET SIZES**

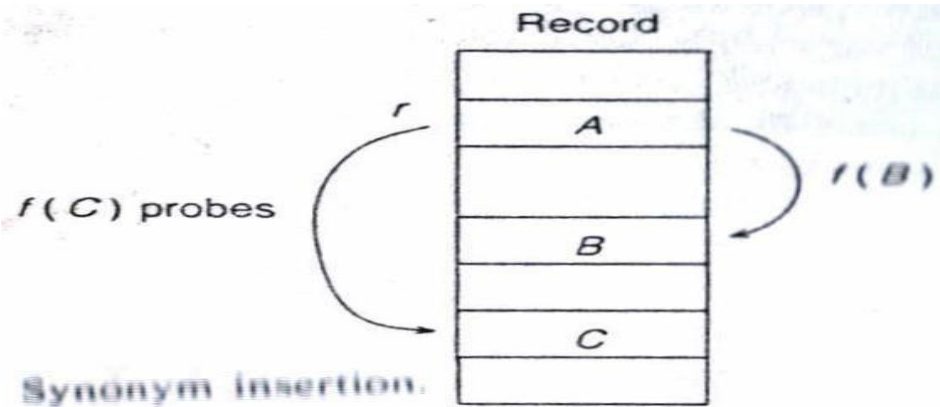
| $\alpha$<br>(in percent) | Blocking factor |       |       |       |
|--------------------------|-----------------|-------|-------|-------|
|                          | 1               | 2     | 5     | 10    |
| 20                       | 1.125           | 1.024 | 1.007 | 1.000 |
| 40                       | 1.333           | 1.103 | 1.012 | 1.001 |
| 60                       | 1.750           | 1.293 | 1.066 | 1.015 |
| 70                       | 2.167           | 1.494 | 1.136 | 1.042 |
| 80                       | 3.000           | 1.903 | 1.289 | 1.110 |
| 90                       | 5.500           | 3.147 | 1.777 | 1.345 |
| 95                       | 10.500          | 5.600 | 2.700 | 1.800 |

Here we use a *variable* increment instead of a constant increment of one to reduce secondary clustering.

Here, the increment is a function of the key being inserted which may be viewed as another hashing function.

So, referred to as **double hashing**,  $H_1$  to get the home address,  $H_2$  to get the increment.

- Possibilities for  $H_2$ :
  - $H_2 = \text{Quotient}(\text{Key} / P) \bmod P$
  - $H_2' = (\text{Key} \bmod (P - 2)) + 1$
- $H_2$  requires two divide operations.  $H_2'$  requires only a single divide operation.  $H_2'$  is more difficult for *people* to compute. So we use  $H_2$  in the example.
- The home address for a record will then be determined by the *remainder* of the key divided by the table size and the increment for collision resolution by the *quotient* of the same operation.



If  $A$ ,  $B$ , and  $C$  are synonyms at  $r$  as illustrated,  $B$  and  $C$  will usually have different increments yielding different probe

### Algorithm 3.2 LINEAR QUOTIENT INSERTION

- I. Hash the key of the record to be inserted to obtain the home address for storing the record.
- II. If the home address is empty, insert the record at that location, else
  - A. Determine the increment by obtaining the quotient of the key divided by the table size. If the result is zero, set the increment to one.
  - B. Initialize the count of locations searched to one.
  - C. While the number of locations searched is less than the table size
    1. Compute the next search address by adding the increment to the current address and then moding the result by the table size.
    2. If that address is unoccupied, then insert the record and terminate with a successful insertion.
    3. If the record occupying the location has the same key as the record being inserted, terminate with a "duplicate record" message.
    4. Add one to the count of the locations searched.
  - D. Terminate with a "full table" message.

|   |  |
|---|--|
| 0 |  |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |

Method requires a *prime number* table size, for otherwise searching could cycle through a subset of the table several times.

| Key |    |
|-----|----|
| 0   |    |
| 1   |    |
| 2   |    |
| 3   |    |
| 4   |    |
| 5   | 27 |
| 6   |    |
| 7   | 18 |
| 8   |    |
| 9   | 29 |
| 10  |    |

In the example, locations 0, 2 and 4 were occupied. If we attempted to insert a record with home address of 0 and an increment of 2, we would cycle through the three occupied locations twice.

$$H1(\text{key}) = \text{key} \bmod 11$$

Keys = 27, 18, 29, 28, 39, 13, 16

Alternatively: New address = (current address + increment) mod table\_size

| Key |    |
|-----|----|
| 0   |    |
| 1   | 39 |
| 2   |    |
| 3   |    |
| 4   |    |
| 5   | 27 |
| 6   | 28 |
| 7   | 18 |
| 8   |    |
| 9   | 29 |
| 10  |    |

| Key |    |
|-----|----|
| 0   |    |
| 1   | 39 |
| 2   | 13 |
| 3   |    |
| 4   |    |
| 5   | 27 |
| 6   | 28 |
| 7   | 18 |
| 8   | 16 |
| 9   | 29 |
| 10  |    |

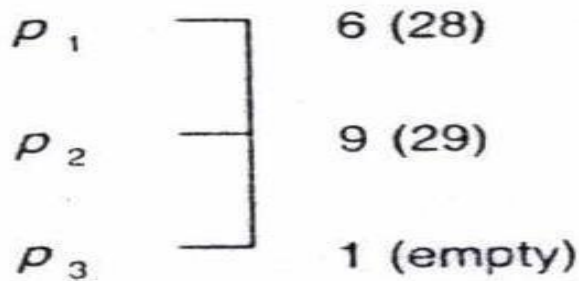
- The mechanism for determining an unsuccessful search is the same as that for progressive overflow. When we encounter an *empty* probe location, we terminate the search unsuccessfully.
- But an unsuccessful search requires fewer probes with linear quotient since we are more likely to encounter an empty location to terminate the search as a result of eliminating secondary clusters.
- *Deleting* a record from a table requires the use of a tombstone.
- We can improve on linear quotient by observing that the number of retrieval probes is dependent on the *placement* of the records.
- E.g. if we insert a record with a key of 67, it has a home address of 1 which is already filled with 39. We then try locations 7, 2 8 and finally 3. Then it requires *five* retrieval probes to find 67. What if 39 had not been stored at location 1?
- The next method *moves* an item already inserted in the table if the move reduces the average number of probes required to retrieve *all* the records.

## Brents Method

We examined static methods. We now examine several *dynamic* methods where an item once stored may be moved.

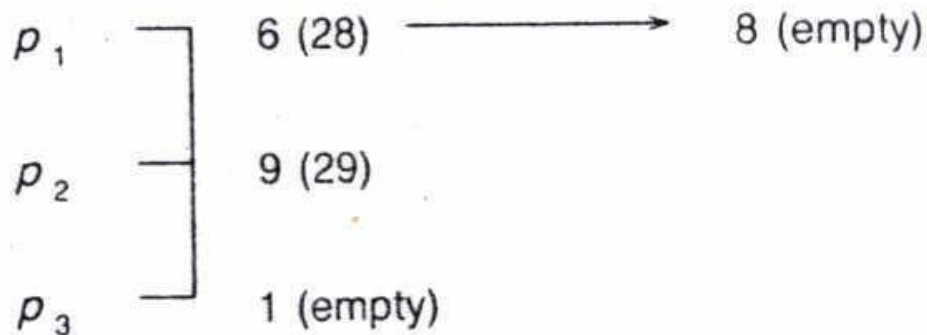
Requires additional processing when inserting a record but reduce the number of probes needed for retrieval (we insert once but retrieve many times).

- The **primary probe chain** of a record is the sequence of locations visited during the insertion or retrieval of the record



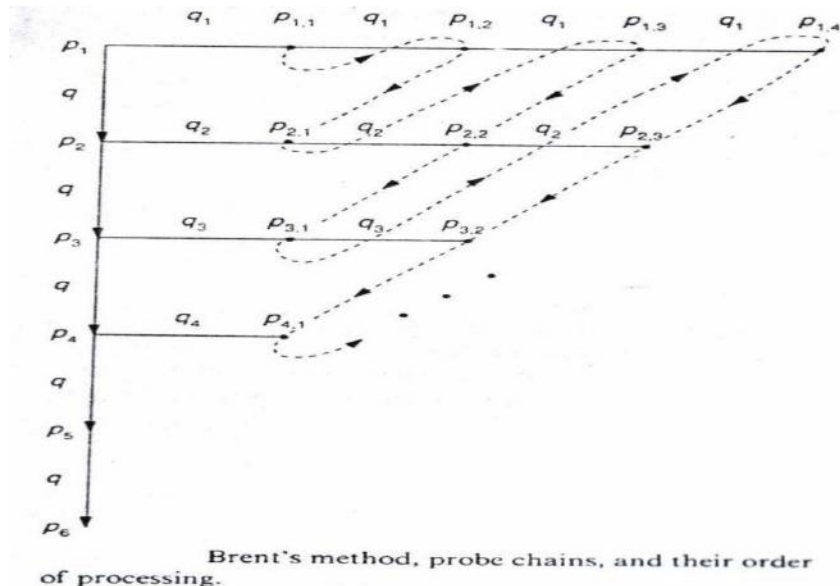
The primary probe chain for 39 is shown on the left.  $p_1$  is the home address. Three positions had to be visited.

- What if its home address were empty? 39 could have been inserted at its home address which requires only 1 probe for retrieval. We could make home address available by moving what is stored there



- 28 will require one more probe for its retrieval but 39 will require two fewer probes for a net reduction of one probe achieved by *moving* 28.

- Move 28 to a location such that it could still be retrieved (still want to use the linear quotient method).
- The sequence of positions visited when attempting to move a record from the primary probe chain is called the **secondary probe chain**.



- The solid vertical line represents the primary probe chain (the addresses that would be considered in storing an item using linear quotient).
- The horizontal lines represent the secondary probe chains (the addresses that would be searched in attempting to move an item from a position along the primary probe chain).

The  $q$  value along the primary probe chain is the increment for the item being inserted.

The  $q_i$ 's along the secondary probe chains represent the increments associated with the item being moved.

The subscript  $i$  gives the number of probes needed to retrieve the item being inserted along its primary probe chain.

The subscript  $j$  gives the number of additional probes needed to retrieve the item being moved along its secondary probe chain. To minimize the number of retrieval probes, we want to minimize  $(i + j)$ .

In the case where  $i = j$ , we will arbitrarily choose to minimize on  $i$ .

Let  $s$  be the number of probes required to retrieve an item if nothing is moved.

We try all combinations of  $(i + j) < s$  such that we minimize  $(i + j)$ .

On equality, since there would be no reduction in the number of probes, no movement would occur.

#### **Algorithm 3.3 BRENT'S METHOD INSERTION**

- 
- I. Hash the key of the record to be inserted to obtain the home address for storing the record.
  - II. If the home address is empty, insert the record at that location, else
    - A. Compute the next potential address for storing the incoming record. Initialize  $s = 2$ .
    - B. While the potential storage address is not empty,
      1. Check if it is the home address. If it is, the table is full, terminate with a "full table" message.
      2. If the record stored at the potential storage address is the same as the incoming record, terminate with a "duplicate record" message.
      3. Compute the next potential address for storing the incoming record. Set  $s = s + 1$ .
- /\* Attempt to move a record previously inserted. \*/
- C. Initialize  $i = 1$  and  $j = 1$ .
  - D. While  $(i + j < s)$ 
    1. Determine if the record stored at the  $i$ /th position on the primary probe chain can be moved  $j$  offsets along its secondary probe chain.
    2. If it can be moved, then
      - a. Move it and insert the incoming record into the vacated position  $i$  along its primary probe chain; terminate with a successful insertion, else
      - b. Vary  $i$  and/or  $j$  to minimize the sum of  $(i + j)$ ; if  $i = j$ , minimize on  $i$ .
- /\* Moving has failed. \*/
- E. Insert the incoming record at position  $s$  on its primary probe chain; terminate with a successful insertion.
-



| Key |    |
|-----|----|
| 0   |    |
| 1   |    |
| 2   |    |
| 3   |    |
| 4   |    |
| 5   | 27 |
| 6   |    |
| 7   | 18 |
| 8   |    |
| 9   | 29 |
| 10  |    |

$$\text{Hash}(\text{key}) = \text{key} \bmod 11$$

And the increment function is  $i(\text{key}) = \text{Quotient}(\text{Key} / 11) \bmod 11$

| Key |           |
|-----|-----------|
| 0   |           |
| 1   |           |
| 2   |           |
| 3   |           |
| 4   |           |
| 5   | 27        |
| 6   | 39        |
| 7   | 18        |
| 8   | <b>28</b> |
| 9   | 29        |
| 10  |           |

28 is moved from its original location to location 8.

Note that we use the increment associated with the item being *moved* and not with the record being inserted.

| Key |           |
|-----|-----------|
| 0   | <b>27</b> |
| 1   |           |
| 2   | 13        |
| 3   |           |
| 4   |           |
| 5   | 16        |
| 6   | 39        |
| 7   | 18        |
| 8   | <b>28</b> |
| 9   | 29        |
| 10  |           |

27 is moved to location 0.

Some probes overall are saved by making the moves.

## Binary Tree

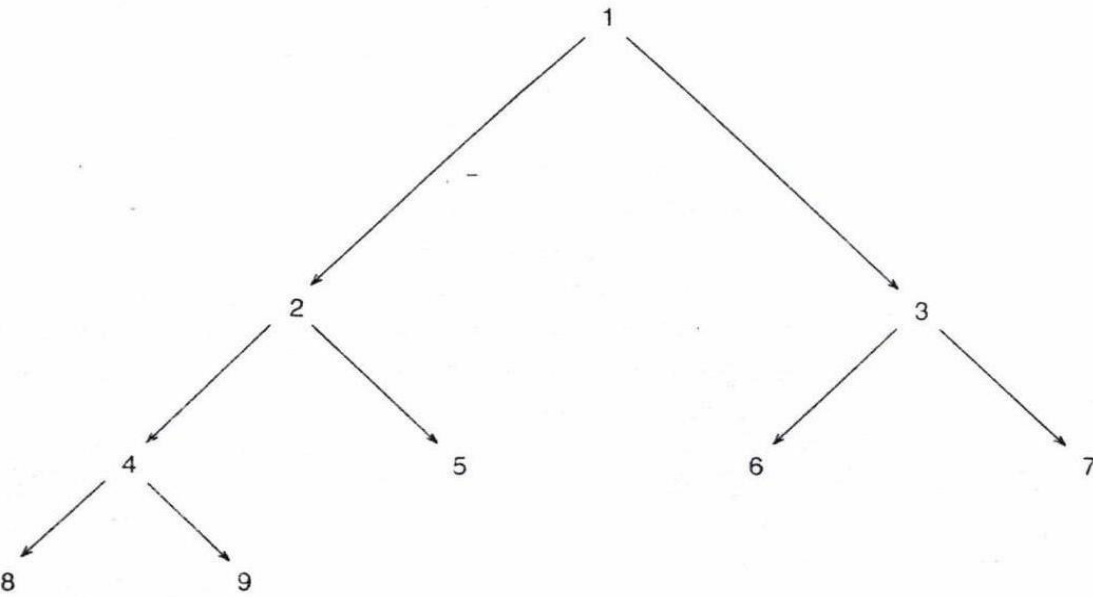
Carries the concept in Brent's method one step further and move items from secondary and subsequent probe chains.

Needs fewer retrieval probes than Brent's method.

Two choices at each probable storage address—*continue* to the next address along the probe chain of the item being inserted or *move* the item stored at that address to the next position on *its* probe chain.



A left branch signifies the *continue* option, a right branch the *move* option.



- The decision tree is generated in a breadth first fashion from the top down left to right.
- It is used *only* as a control mechanism in deciding where to store an item and is not used for storing records.

A different binary tree is constructed for each insertion. Encountering either an empty leaf node in the binary tree or a full table terminates the process.

By moving items from secondary and subsequent probe chains, we are achieving a placement of records that will further reduce the average number of retrieval probes (we insert once but retrieve often).

Like Brent's method, processing is only for insertion, *retrieval* is done with linear quotient.

#### **Algorithm 3.4** **BINARY TREE INSERTION**

---

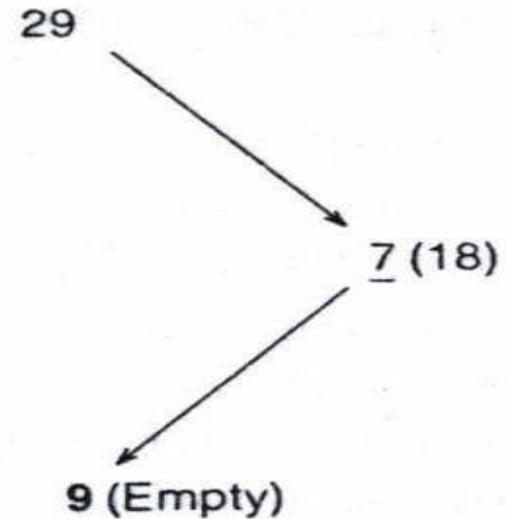
- I. Hash the key of the record to be inserted to obtain the home address for storing the record.
  - II. If the home address is empty, insert the record at that location, else
    - A. Until an empty location or a "full table" is encountered,
      1. Generate a binary tree control structure in a breadth first left to right fashion. The address of the lchild of a node is determined by adding (i) the increment associated with the key of the record *coming in* to the node to (ii) the current address. The address of the rchild of a node is determined by adding (i) the increment associated with the key of the record *stored* in the node to (ii) the current address.
      2. At the leftmost node on each level, check against the record associated with it for a duplicate record. If found, terminate with a "duplicate record" message.
    - B. If a "full table," terminate with a "full table" message.
    - C. If an empty node is found, the path from the empty node back to the root determines which records, if any, need to be moved. Each right link signifies that a relocation is necessary. First, set the current node pointer to the last node generated in the binary tree and set the empty location pointer to the table address associated with the last node generated.
    - D. Until the current pointer equals the root node of the binary tree (bottom up), note the type of branch from the parent of the current node to the current node.
      1. On a right branch, move the record stored at the location contained in the parent node into the location indicated by the empty location pointer. Set the empty location pointer to the newly vacated position and make the parent node the current node.
      2. On a left branch, make the parent node the current node.
    - E. Insert the record coming into the root position into the empty location. Terminate with a successful insertion.
-

Keys to insert: 27, 18, 29, 28, 39, 13, 16, 41, 17, 19

$\text{Hash}(\text{key}) = \text{key} \bmod 11$

$i(\text{key}) = \text{Quotient}(\text{key} / 11) \bmod 11$

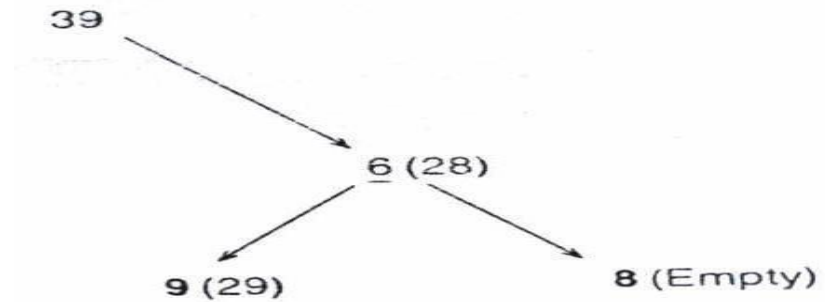
- 27 and 18 are inserted without difficulty.
- 29 causes a collision
- So we generate a binary tree to determine which, if any, records to move.
- Location 9 is empty.
- The tree appears on the right.



- The bold numbers represent locations in the table and the values in parentheses are the keys stored at those locations. The underlined node is the root node.
- The tree tells us to attempt to move 29 into location 7.
- That location is occupied so we continue along the primary probe chain until we reach location 9. The table is shown on the right.

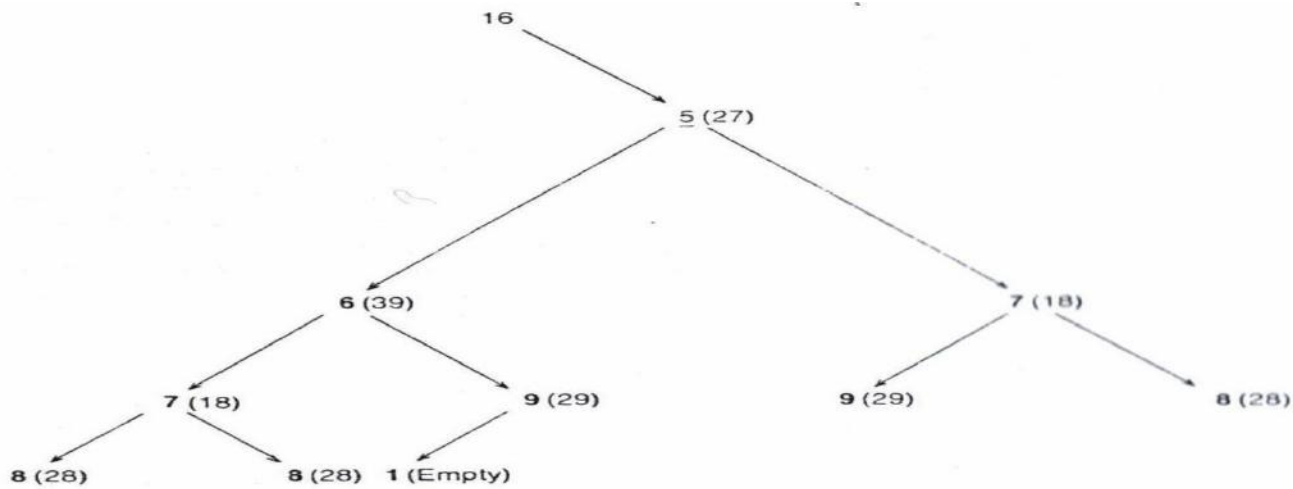
|    | Key |
|----|-----|
| 0  |     |
| 1  |     |
| 2  |     |
| 3  |     |
| 4  |     |
| 5  | 27  |
| 6  |     |
| 7  | 18  |
| 8  |     |
| 9  | 29  |
| 10 |     |

The path formed by the leftmost branch at each level of the tree is equivalent to the primary

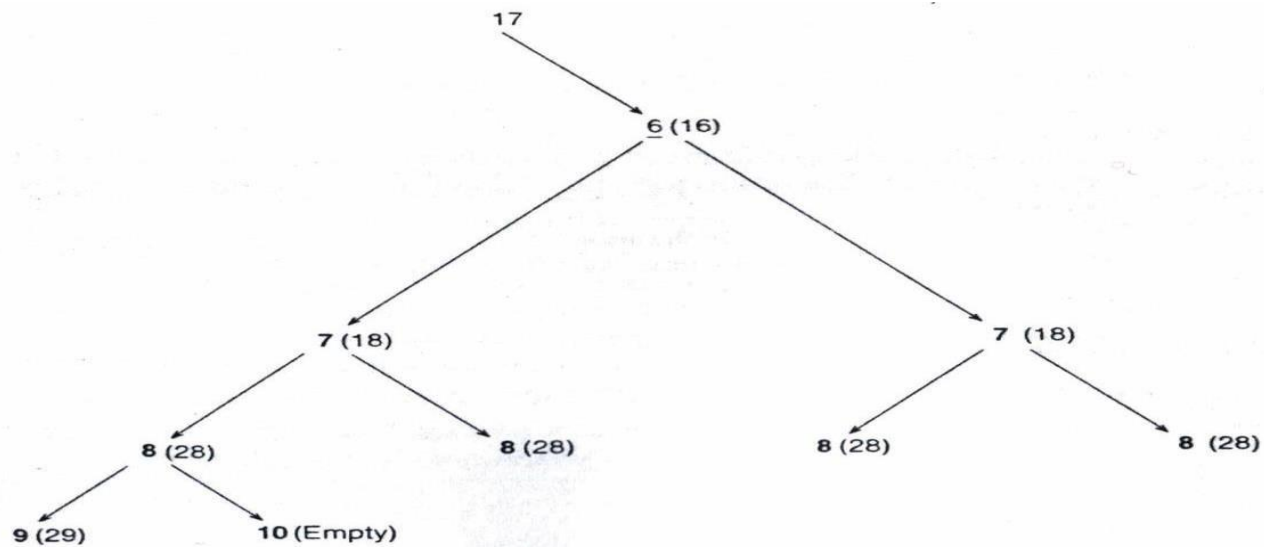


|    | Key |
|----|-----|
| 0  |     |
| 1  |     |
| 2  |     |
| 3  |     |
| 4  |     |
| 5  | 27  |
| 6  | 39  |
| 7  | 18  |
| 8  | 28  |
| 9  | 29  |
| 10 |     |

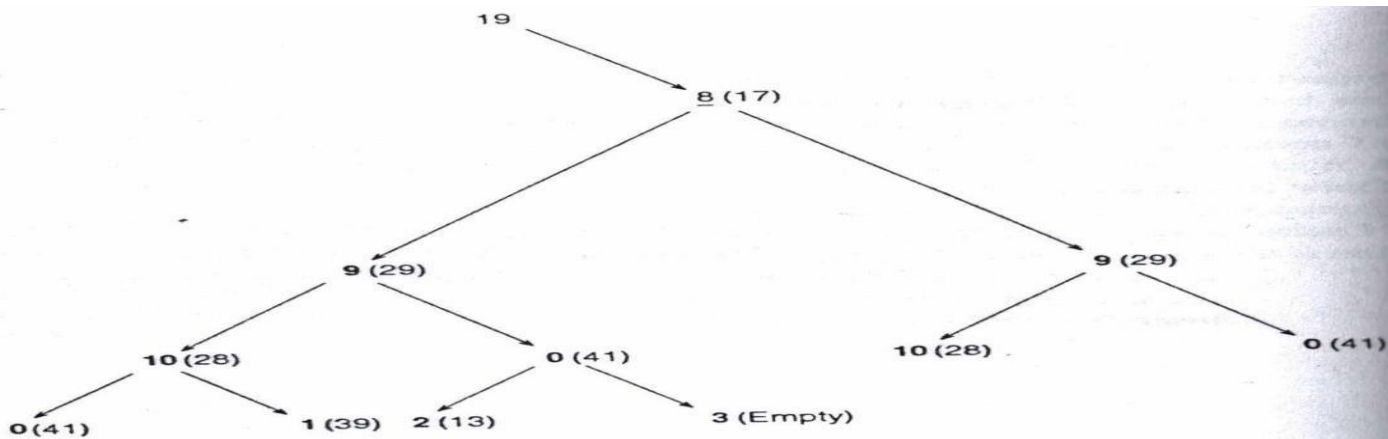
probe chain of the linear quotient method and of Brent's method



|    | Key |
|----|-----|
| 0  |     |
| 1  | 39  |
| 2  | 13  |
| 3  |     |
| 4  |     |
| 5  | 27  |
| 6  | 16  |
| 7  | 18  |
| 8  | 28  |
| 9  | 29  |
| 10 |     |



|    | Key |
|----|-----|
| 0  | 41  |
| 1  | 39  |
| 2  | 13  |
| 3  |     |
| 4  |     |
| 5  | 27  |
| 6  | 16  |
| 7  | 18  |
| 8  | 17  |
| 9  | 29  |
| 10 | 28  |



|    | Key |
|----|-----|
| 0  | 29  |
| 1  | 39  |
| 2  | 13  |
| 3  | 41  |
| 4  |     |
| 5  | 27  |
| 6  | 16  |
| 7  | 18  |
| 8  | 17  |
| 9  | 19  |
| 10 | 28  |

Binary tree used in inserting items is a complete binary tree.

The  $n$  nodes of a **complete binary tree** corresponds to the first  $n$  nodes of a full binary tree numbered top down, left to right.

An advantage of it: can be implemented as a sequential structure:

$$\text{lchild}(i) = 2 * i$$

$$\text{rchild}(i) = 2 * i + 1$$

$$\text{parent}(i) = i/2$$

The depth of the binary tree is  $O(\log n)$ .

After the tree gets to be a certain size, secondary storage can be used (for example for the bottom two levels of the tree)

How to check for a full table?

Keep a counter of the depth of the tree. When it exceed  $\log n$  table is full. But this requires the generation of a possibly enormous tree.

Better solution: Keep a counter of how many records have already been inserted into table, check that number before generating the tree.

### Computed Chaining

The methods that use a link field provide better performance but require more storage.

- Those that don't use a link field require less space but provide poorer performance.
- Consider a third class in between. Instead of storing an actual address in the link, they store a pseudolink (which requires additional processing before it yields an actual address)
- If storage is limited, information needed can be computed rather than stored.
- The performance of the pseudolink methods are better than the nonlink methods. The number of offsets or locations which would need to be searched to locate the successor item are stored in the pseudolink so that we can compute the successor's actual address.

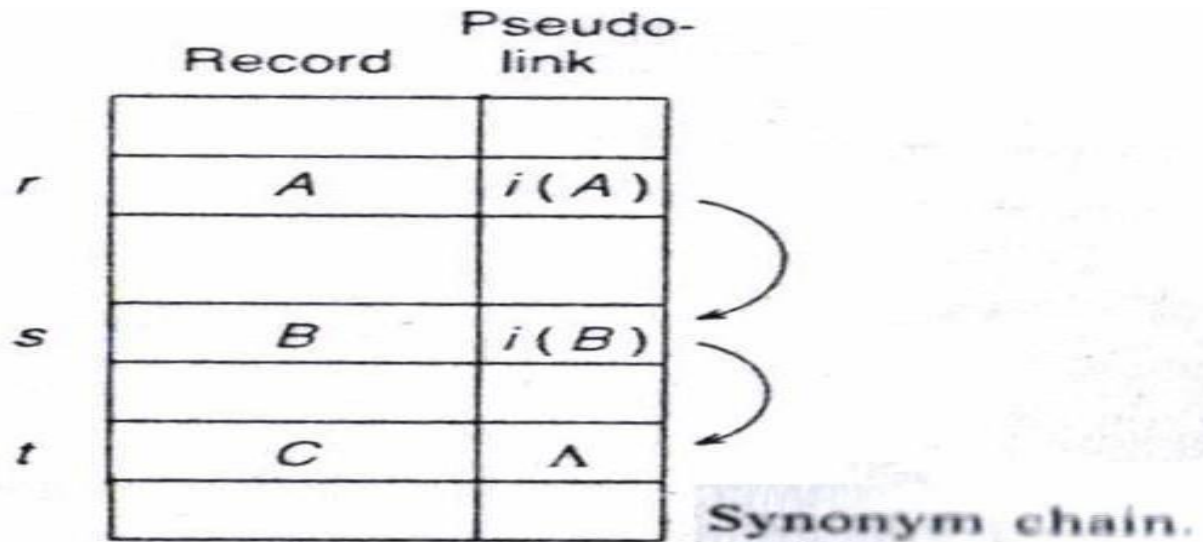


The intermediate locations do not have to be searched so performance is improved.

It takes fewer bits (less storage) to store the number of offsets rather than a full address.

On the negative side, do not perform as well as the coalesced hashing methods and they require more storage than the nonlink methods.

- Follows the process of coalesced hashing in that the first item on a chain points to its immediate successor which points to its immediate successor.
- To eliminate the coalescing of chains, computed chaining also *moves* a record stored at another record's home address.
- There is no coalescing (improves performance).
- Unlike the nonlink methods, computed chaining uses the key of the record *stored* at a probe address to locate the next probe address and not the key of the record being *inserted* or *retrieved*.



- Using a function of the item stored at a location ensures that only one actual probe will be needed to locate the successor record.

### Algorithm 3.5

#### COMPUTED CHAINING INSERTION

---

- I. Hash the key of the record to be inserted to obtain the home address for storing the record.
  - II. If the home address is empty, insert the record at that location.
  - III. If the record is a duplicate, terminate with a "duplicate record" message.
  - IV. If the item stored at the hashed location is not at its home address, move it to the next empty location found by stepping through the table using the increment associated with its predecessor element, and then insert the incoming record into the hashed location, else
    - A. Locate the end of the probe chain and in the process, check for a duplicate record.
    - B. Use the increment associated with the last item in the probe chain to find an empty location for the incoming record. In the process, check for a full table.
    - C. Set the pseudolink at the position of the predecessor record to connect to the empty location.
    - D. Insert the record into the empty location.
- 

### Algorithm 3.6

#### PSEUDOCODE COMPUTED CHAINING INSERTION

---

**proc** computed\_chaining\_insert

*/\* Inserts a record with a key  $x$  according to the computed chaining hashing method. Table[ $r$ ] refers to the contents at location  $r$  in the file and pseudolink[ $r$ ] contains the offset value associated with that location. \*/*

```
1 $h \leftarrow \text{Hash}(x)$ /* Locate the home address. */
2 if Table[h] = Λ then [Table[h] $\leftarrow x$; return] /* The home address is empty, so insert the item. */
3 if Table[h] = x then return /* The item is a duplicate. */
4 if Hash(Table[h]) $\neq h$ then move the item /* The item stored at h is not stored at its home address, so move it and store x at h . */
5 while pseudolink[h] $\neq \Lambda$
6 do
7 $h \leftarrow \text{probe}(\text{Table}[h], \text{pseudolink}[h], h)$ /* probe is a function to locate the address of the next item in the probe chain */
8 if Table[h] = x then return /* The item is a duplicate */
9 end
```

The **probe** function *computes* the address of the successor element given the key of the record stored at the current location and the pseudolink of the current location. The incrementing scheme of linear probing

```
10 i ← 1 /* Initialize a loop variable to locate the first empty cell for storing x. */
11 j ← probe(Table[h], i, h) /* Locate the next probe address. */
12 while Table[j] ≠ Λ
13 do
14 i ← i + 1
15 if i > table__size then [print "table full"; stop]
16 j ← probe(Table[h], i, h) /* Locate the next probe address. */
17 end
18 pseudolink[h] ← i /* Insert the probe number */
19 Table[j] ← x /* Store the item */

end computed__chaining__insert
```

---

### Computed Chaining Example

Keys: 27, 18, 29, 28, 39, 13, 16, 38, 53

$\text{Hash}(\text{key}) = \text{key} \bmod 11$

$i(\text{key}) = \text{Quotient}(\text{Key} / 11) \bmod 11$

|    | Key | nof_ |
|----|-----|------|
| 0  |     |      |
| 1  |     |      |
| 2  |     |      |
| 3  |     |      |
| 4  |     |      |
| 5  | 27  |      |
| 6  |     |      |
| 7  | 18  | 1    |
| 8  | 29  |      |
| 9  |     |      |
| 10 |     |      |

“nof” represents the number of offsets or the pseudolink value.

|    | Key | nof |
|----|-----|-----|
| 0  |     |     |
| 1  |     |     |
| 2  |     |     |
| 3  |     |     |
| 4  |     |     |
| 5  | 27  |     |
| 6  | 28  | 2   |
| 7  | 18  | 1   |
| 8  | 29  |     |
| 9  |     |     |
| 10 | 39  |     |

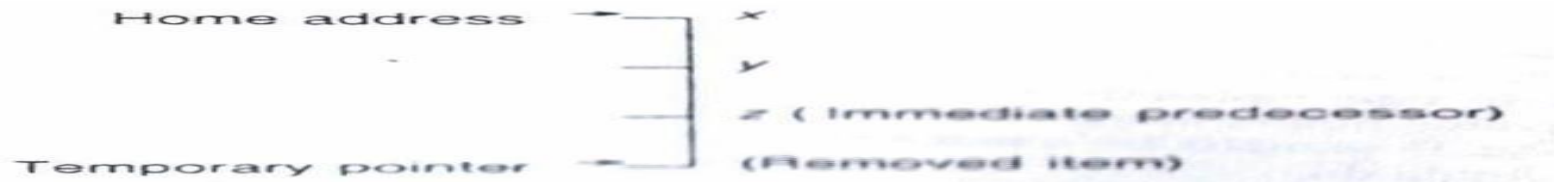
- The increment is changed only when we reach the successor element on a probe chain.
- The current increment is always the one associated with the most recently visited record *on the current probe chain*.

|    | Key | nof |
|----|-----|-----|
| 0  |     |     |
| 1  |     |     |
| 2  | 13  |     |
| 3  |     |     |
| 4  |     |     |
| 5  | 27  | 2   |
| 6  | 28  | 2   |
| 7  | 18  | 1   |
| 8  | 29  |     |
| 9  | 16  |     |
| 10 | 39  |     |

- Which pseudolink field do we need to set?
- The one associated with 16.
- Since we had to look at two additional locations, the pseudolink value is two.

- We finally add the record with key 53.
- We observe a *move* process. We move 16 *plus* all its successors. Otherwise we would no longer be able to retrieve them.
- The record with key 53 may be inserted directly at location 9.
- We now reinsert the two records that were displaced by the insertion.
- We need to keep a pointer or remember location 9 so that we can find the end of the chain that those two records were on previously.
- We need to identify the immediate predecessor to the item that was removed from location 9 so that we can relink it to the new location for the removed record.
- We know where to begin the search by hashing 16 which yields location 5. We search from that location until we find a pseudolink to location 9. We find that immediately at location 5.

In general, the search for the immediate predecessor of a removed item may be represented as



- We begin the search at the home address which contains a record and continue the search until we find a record that points to the location of the removed item which is indicated with the temporary pointer.
- We now know the location of the pseudolink field that needs to be modified in the reinsertion process.
- In the example, we then need to use that increment associated with 27 until we can find another empty space to insert the record with key 16.

|    | Key | nof |
|----|-----|-----|
| 0  | 16  | 1   |
| 1  | 38  |     |
| 2  | 13  |     |
| 3  |     |     |
| 4  |     |     |
| 5  | 27  | 3   |
| 6  | 28  | 2   |
| 7  | 18  | 1   |
| 8  | 29  |     |
| 9  | 53  |     |
| 10 | 39  |     |

- Did we degrade the performance for retrieving 16 by moving it?
- No, we essentially have a linked list such that the actual physical location of a record is not important

The concept of having multiple probe chains instead of a single probe chain for organizing records. Each chain will be smaller and the searching will be faster.





- In addition to the hashing function,  $\text{Hash}(\text{key})$ , to obtain the home address for a record and the incrementing function,  $i(\text{key})$ , to obtain an increment, we introduce a *third* hashing function,  $g(\text{key})$ , to tell us which probe chain to insert into or to search. This is an example of *triple hashing*.

$$g(\text{key}) \rightarrow 0, 1, \dots, R - 1$$

where  $R$  is the number of subgroupings. A simple function for  $g(\text{key})$  is

$$g(\text{key}) = \text{key} \bmod R$$

- Keys: 27, 18, 29, 28, 39, 13, 16, 38, 53
- $\text{Hash}(\text{key}) = \text{key} \bmod 11$
- $i(\text{key}) = \text{Quotient}(\text{Key} / 11) \bmod 11$
- $g(\text{key}) = \text{key} \bmod 2$
- If a key is even, it will go on the zeroth chain, if it is odd, it will go on the first chain. The records are inserted as before with the only difference being the placement of the pseudolink values: in chain zero for even key values and in chain one for odd key values.



|    | Key | nof |   |
|----|-----|-----|---|
|    |     | 0   | 1 |
| 0  | 16  | 1   |   |
| 1  | 38  |     |   |
| 2  | 13  |     |   |
| 3  |     |     |   |
| 4  |     |     |   |
| 5  | 27  | 3   |   |
| 6  | 28  |     | 2 |
| 7  | 18  |     | 1 |
| 8  | 29  |     |   |
| 9  | 53  |     |   |
| 10 | 39  |     |   |

|    | Key | nof |   |
|----|-----|-----|---|
|    |     | 0   | 1 |
| 0  | 16  | 1   |   |
| 1  | 38  |     |   |
| 2  | 13  |     |   |
| 3  |     |     |   |
| 4  | 49  |     |   |
| 5  | 27  | 3   | 5 |
| 6  | 28  |     | 2 |
| 7  | 18  |     | 1 |
| 8  | 29  |     |   |
| 9  | 53  |     |   |
| 10 | 39  |     |   |

**TABLE 3.5** ADVANTAGES, DISADVANTAGES, AND WHEN TO USE VARIOUS COLLISION RESOLUTION METHODS

| Method                   | Advantages                                              | Disadvantages                                                            | When to use                                                                                          |
|--------------------------|---------------------------------------------------------|--------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| Coalesced hashing [DCWC] | Excellent performance<br>Superior performance           | Requires space for link field<br>Requires space for link field           | When ample space is available<br>When ample space is available and insertion time is not critical    |
| Progressive overflow     | No additional space; simple algorithm                   | Very poor performance                                                    | Almost NEVER                                                                                         |
| Linear quotient          | No additional space; reasonable performance             | Performance below that of other methods                                  | When space is at a premium; retrieval performance is not critical                                    |
| Brent's method           | No additional space; good performance                   | Performance below that of linked methods                                 | When space is at a premium, even space for a binary tree                                             |
| Binary tree              | No additional space; very good performance              | Performance below that of linked methods                                 | When file space is limited but temporary space is available for tree; performance is important       |
| Computed chaining        | Small amount of additional space; excellent performance | Requires some space for pseudolink field; performance below that of DCWC | When some extra file space is available and performance is important; insertion time is not critical |

## **Perfect Hashing**

hash(key) → **unique** address

Both primary and secondary clustering are eliminated.

With perfect hashing, we need only a single probe to retrieve a record.

Perfect hashing is applicable to files with only a relatively small number of records because the computing time is big.

A **perfect hashing function** maps a key into a unique address. If the range of potential addresses is the same as the number of keys, the function is a minimal (in space) perfect hashing function.

A separate hashing needs to be devised for *each* set of keys. If one or more of the keys change, a new hashing function must be constructed.

- In *minicycle algorithm*, for a table of size  $N$ , a perfect hashing function may be characterized as:

$$p.\text{hash}(\text{key}) = (h_0(\text{key}) + g[h_1(\text{key})] + g[h_2(\text{key})]) \bmod N$$

- What needs to be decided are the functions  $h_0, h_1, h_2$  and  $g$ .
- These functions should be efficient.
- The worst case of minicycle algorithm is  $O(r^6)$  where  $r$  is the number of records in the set. An upper bound for  $r$  seems to be about 512.
- A special case of minicycle alg. is the Cichelli's algorithm which is not as efficient as minicycle alg., appropriate for  $r$  upto 60, plus more disadvantages, but it is straightforward to understand.
- For larger amounts, the minicycle algorithm is appropriate.

### Cichellis Algorithm

$$h_0 = \text{length}(\text{key})$$

$$h_1 = \text{first\_character}(\text{key})$$

$$h_2 = \text{last\_character}(\text{key})$$

$$g = T(x)$$

- where  $T$  is a table of values associated with individual characters  $x$  which may appear in a key. The time consuming part is determining  $T$ .
- In the table, a value may be assigned to more than one character.

**TABLE 3.6** VALUES ASSOCIATED WITH THE CHARACTERS OF THE PASCAL RESERVED WORDS

|        |        |        |        |       |        |
|--------|--------|--------|--------|-------|--------|
| A = 11 | B = 15 | C = 1  | D = 0  | E = 0 | F = 15 |
| G = 3  | H = 15 | I = 13 | J = 0  | K = 0 | L = 15 |
| M = 15 | N = 13 | O = 0  | P = 15 | Q = 0 | R = 14 |
| S = 6  | T = 6  | U = 14 | V = 10 | W = 6 | X = 0  |
| Y = 13 | Z = 0  |        |        |       |        |

Let's apply the alg. to the keyword **begin**.  $p.hash(\mathbf{begin}) = 5 + T(h_1(key)) + T(h_2(key))$

$$= 5 + T(b) + T(n)$$

$$= 5 + 15 + 13$$

$$= 33$$

### **Algorithm 3.7**

### **CICHELLI'S ALGORITHM FOR DETERMINING A PERFECT HASH FUNCTION**

---

- I. Order the set of keys based upon the sum of the frequencies of occurrence of the first and last characters of the keys in the entire set. Place the keys with the most frequently occurring first and last characters at the top of the ordering.
  - II. Modify the ordering. Place a key whose first and last characters both appear in prior keys as high in the ordering such that this condition holds.
  - III. Assign the value 0 to the first and last characters of the first key in the set.
  - IV. For each remaining key in the set, in order,
    - A. If the first and last characters have already been assigned values, hash the key to determine if a conflict arises. If yes, discontinue processing on this key and recursively apply this step with the previous key (backtrack).
    - B. If either the first *or* last character has already been assigned a value, assign a value to the other character by trying all possibilities from 0 to the maximum allowable value. If a conflict still exists after trying all possible values for the other character, then discontinue processing on this key and recursively apply this step with the previous key (backtrack).
    - C. If both the first and last characters are as yet unassigned, vary the first and then the last character, trying each combination. If all combinations cause conflicts, then discontinue processing on this key and recursively apply this step with the previous key (backtrack).
  - V. If all keys have been processed, terminate successfully, else terminate unsuccessfully.
- 

The algorithm involves ordering the keys based on the frequencies of occurrence of the first and last characters in the keys.

Assignment of values are made to the characters in the first and last positions of the keys from the top of the ordering to the bottom.

The algorithm uses an exhaustive search with backtracking.

Keys: cat, ant, dog, gnat, chimp, rat, toad

- We assume that the maximum value that may be assigned to a character is 4. If we cannot find a solution using 4, we would try a larger value. If this maximum value is too small, we will either have no solution or a great amount of backtracking. If the value is too large we won't obtain a minimal solution.
- The frequencies of occurrence of the first and last characters are

$a=1$        $c=2$        $d=2$        $g=2$        $p=1$        $r=1$        $t=5$

We can compute the sums of the frequencies of occurrence of the first and last characters of each key. We obtain

|                      |   |
|----------------------|---|
| <u>c</u> at          | 7 |
| <u>a</u> nt          | 6 |
| <u>d</u> og          | 4 |
| gn <u>a</u> t        | 7 |
| <u>c</u> himp        | 3 |
| <u>r</u> at          | 6 |
| <u>t</u> oa <u>d</u> | 7 |

- We next order the keys in descending order based on the sum of the frequencies to obtain what is on the right.
- We arbitrarily chose the descending order. A random ordering would have been equally acceptable.

|                        |   |
|------------------------|---|
| <u>t</u> oad <u>d</u>  | 7 |
| g <u>n</u> at <u>t</u> | 7 |
| <u>c</u> at <u>t</u>   | 7 |
| <u>r</u> at <u>t</u>   | 6 |
| <u>a</u> nt <u>t</u>   | 6 |
| <u>d</u> og            | 4 |
| <u>c</u> himp          | 3 |

- Then we check the ordering to see if any keys exist that have both their first and last characters appearing in previous keys.
- Notice that the *d* and *g* of “dog” have appeared previously so that “dog” would then be moved to the position following “gnat”.
- The ordering is shown on the right.

|                        |   |
|------------------------|---|
| <u>t</u> oad <u>d</u>  | 7 |
| g <u>n</u> at <u>t</u> | 7 |
| <u>d</u> og            | 4 |
| <u>c</u> at <u>t</u>   | 7 |
| <u>r</u> at <u>t</u>   | 6 |
| <u>a</u> nt <u>t</u>   | 6 |
| <u>c</u> himp          | 3 |



We begin assigning values to the characters.

$$t = 0 \qquad d = 0$$

$$p.hash(toad) = 4$$

$$t = 0 \qquad d = 0 \qquad g = 1$$

$$p.hash(toad) = 4$$

$$p.hash(gnat) = 5$$

- In “dog” we have a conflict. So, backtrack

$$t = 0 \qquad d = 0 \qquad g = 2$$

$$p.hash(toad) = 4$$

$$p.hash(gnat) = 6$$

$$t = 0 \qquad d = 0 \qquad g = 2$$

$$p.hash(toad) = 4$$

$$p.hash(gnat) = 6$$

$$p.hash(dog) \qquad \qquad \qquad = 5$$

$t = 0$                        $d = 0$                        $g = 2$                        $c = 0$

p.hash(toad) = 4

p.hash(gnat) = 6

p.hash(dog)                      = 5

p.hash(cat)                      = 3

$t = 0$                        $d = 0$                        $g = 2$                        $c = 0$                        $r = 4$

p.hash(toad) = 4

p.hash(gnat) = 6

p.hash(dog)                      = 5

p.hash(cat)                      = 3

p.hash(rat)                      = 7

$t = 0$                        $d = 0$                        $g = 3$                        $c = 0$                        $r = 2$   
 $p.hash(toad) = 4$

$p.hash(gnat) = 7$   
 $p.hash(dog) = 6$   
 $p.hash(cat) = 3$   
 $p.hash(rat) = 5$

We are not able to assign  $a$  value to  $a$  without a collision. So, we backtracked.

$t = 0$                        $d = 0$                        $g = 4$                        $c = 0$                        $r = 2$                        $a = 3$

$p.hash(toad) = 4$   
 $p.hash(gnat) = 8$   
 $p.hash(dog) = 7$   
 $p.hash(cat) = 3$   
 $p.hash(rat) = 5$   
 $p.hash(ant) = 6$

- Problem still existed and we backtracked.

$$t = 0$$

$$d = 0$$

$$g = 4$$

$$c = 0$$

$$r = 2$$

$$a = 3$$

$$p = 4$$

$$p.\text{hash}(\text{toad}) = 4$$

$$p.\text{hash}(\text{gnat}) = 8$$

$$p.\text{hash}(\text{dog}) = 7$$

$$p.\text{hash}(\text{cat}) = 3$$

$$p.\text{hash}(\text{rat}) = 5$$

$$p.\text{hash}(\text{ant}) = 6$$

$$p.\text{hash}(\text{chimp}) = 9$$

Since this solution maps the seven keys into seven consecutive locations, it is a minimal perfect hashing function.

## Indexed Sequential File Organization

Indexed sequential file organization is a hybrid of sequential and direct file organization.

Suitable to access data both directly and sequentially. sequential search is slow.



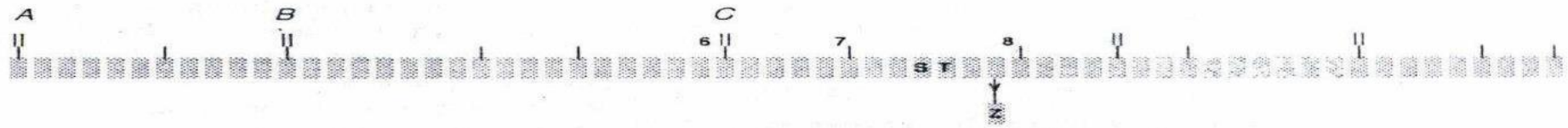
- To accelerate the search, we order the information and put tabs or an index to groups.



- We search the tabs until we find the related group, then search exhaustively only those records in that group
- We apply this grouping process a second time and organize the current groups into larger ones, e.g. file drawers.
- We search a much smaller number of higher-level tabs, then search the tabs of the information in that group, then search to locate the record.
- The tab or index structure is, a *tree* structure



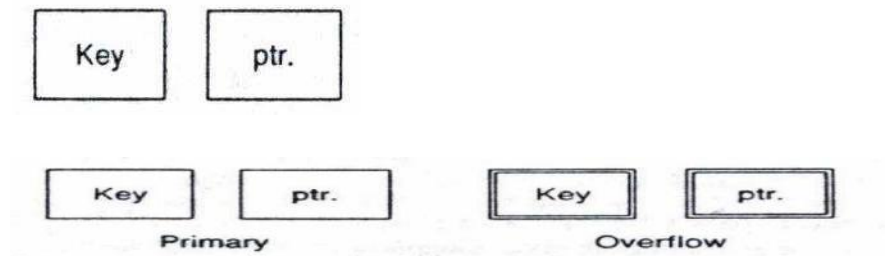
- For insertions we associate an *overflow* area with each original or *primary* storage area for a group of records; so at most we only have to move the number of records in such a group to open place for the record to insert.



- In computers with block addressable disks, we use *tracks* as the lowest level of grouping information.
- The next higher level is a grouping by *cylinders*. Then we have additional levels of *master* indexes.
- An indexed sequentially structured file is referred to as an ISAM (for indexed sequential access method) file.

The cylinder index contains pointers to several cylinders. A pair of entries is used for each cylinder

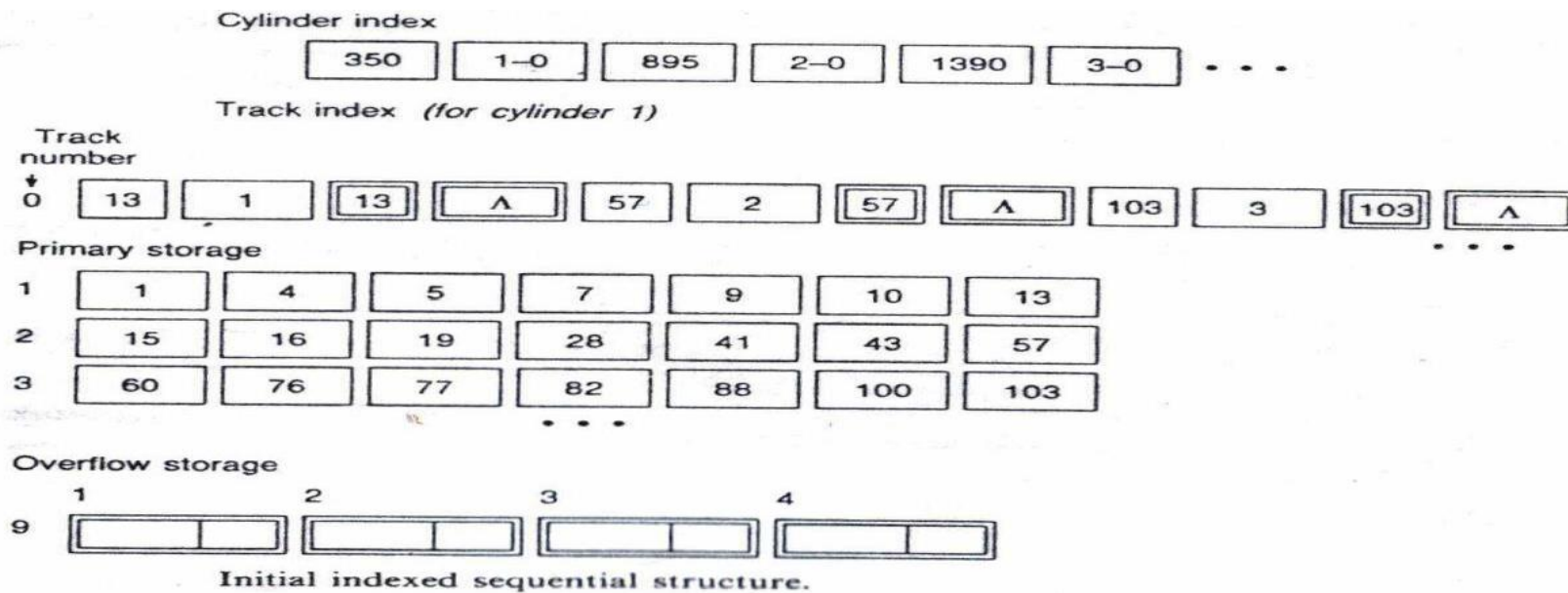
*key* is the highest key on that cylinder and *ptr* is a pointer to the track index for that cylinder.



Each track has two pairs of entries in the track index. One pair contain information on the primary storage area and the other pair has information on the overflow records associated with the track.

The key in the first pair provides the highest key on that track in the *primary* area, and the key in the second pair provides the highest key in the *overflow* area.

The primary pointer indicates the track containing the primary records, and the overflow pointer indicates the first overflow record.



The highest key on cylinder 1 is 350.

Pointer entry notation is  $x-y$ , where  $x$  gives the cylinder number and  $y$  gives the track number where the track index for that cylinder is stored.

The track index would not normally require an entire track of storage; so the remaining area could be used for additional primary or overflow storage.

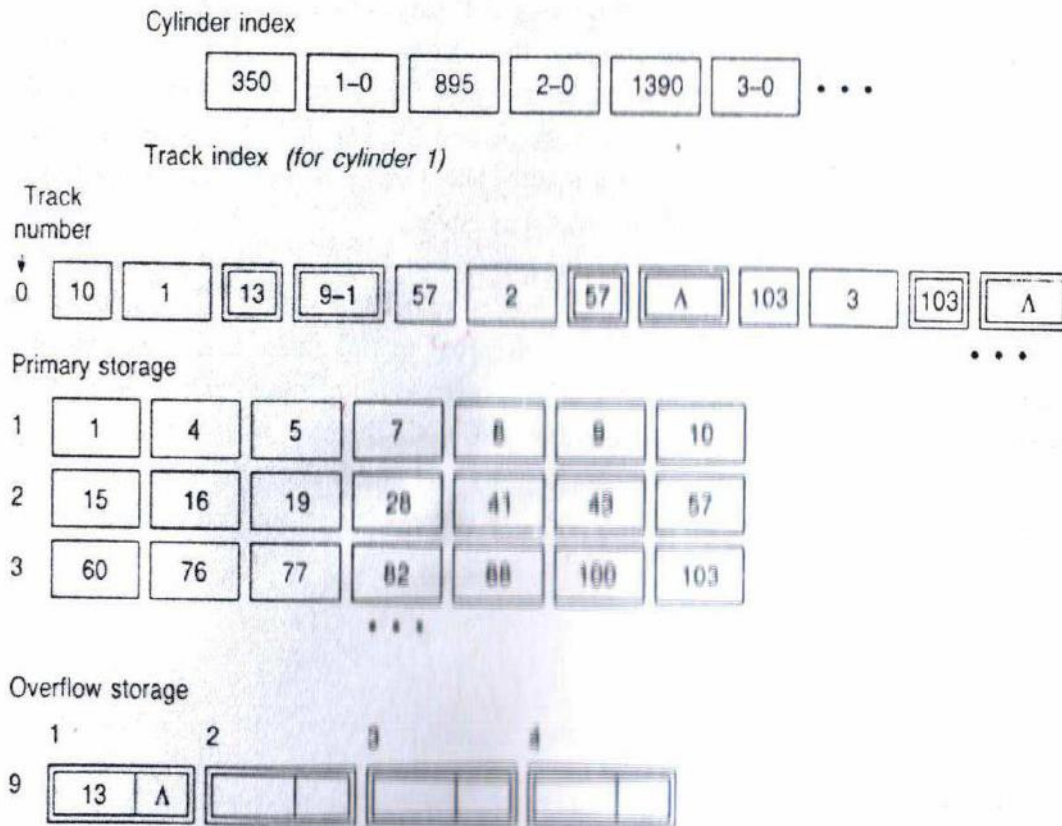
For simplicity, we assume that the records in primary storage have a blocking factor (bucket size) of one and that all primary storage is filled.

- The NULL in the overflow pointer indicates that no overflow records currently exist.
- We can have overflow records associated with each track. We never have to *move* more than the number of records on a single track.



- An available space list would link the overflow space together as records are added and deleted from the file.
- The overflow pointers follow the format  $z-w$ , where  $z$  gives the track number and  $w$  gives the record number.

Here we insert 8



When shifting 13, we *update* the entries for the highest key in the primary storage area, which becomes 10, and the overflow pointer to 9-1.

Here we insert 99

Cylinder index

|     |     |     |     |      |     |     |
|-----|-----|-----|-----|------|-----|-----|
| 350 | 1-0 | 895 | 2-0 | 1390 | 3-0 | ... |
|-----|-----|-----|-----|------|-----|-----|

Track index (for cylinder 1)

Track  
number

|   |    |   |    |     |    |   |    |   |     |   |     |     |     |
|---|----|---|----|-----|----|---|----|---|-----|---|-----|-----|-----|
| ↓ | 10 | 1 | 13 | 9-1 | 57 | 2 | 57 | Λ | 100 | 3 | 103 | 9-2 | ... |
| 0 |    |   |    |     |    |   |    |   |     |   |     |     |     |

Primary storage

|   |    |    |     |    |    |    |     |
|---|----|----|-----|----|----|----|-----|
| 1 | 1  | 4  | 5   | 7  | 8  | 9  | 10  |
| 2 | 15 | 16 | 19  | 28 | 41 | 43 | 57  |
| 3 | 60 | 76 | 77  | 82 | 88 | 99 | 100 |
|   |    |    | ... |    |    |    |     |

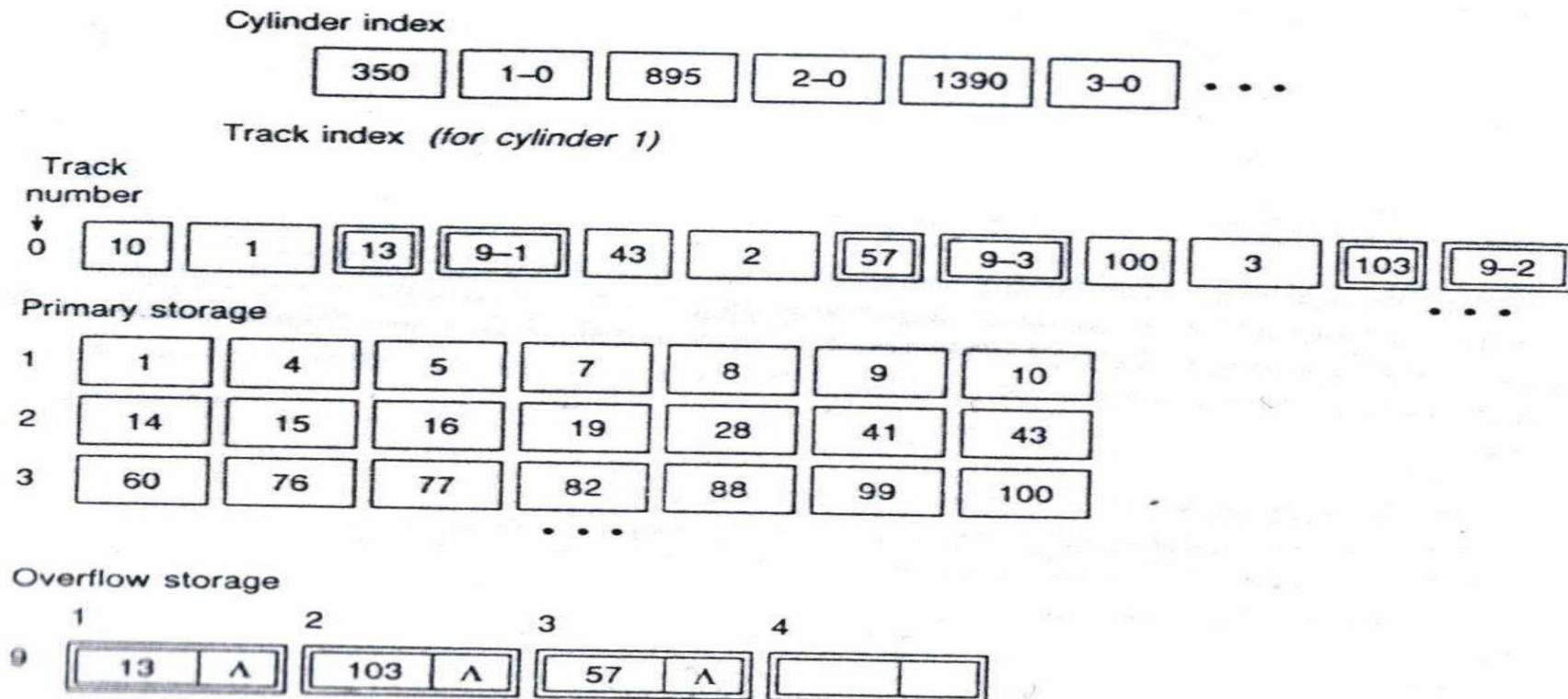
Overflow storage

|   |    |   |     |   |
|---|----|---|-----|---|
|   | 1  | 2 | 3   | 4 |
| 9 | 13 | Λ | 103 | Λ |
|   |    |   |     |   |

The next record to insert has a key of 14. The track to insert is 2. We move *all* of the records stored in the primary area of the track.

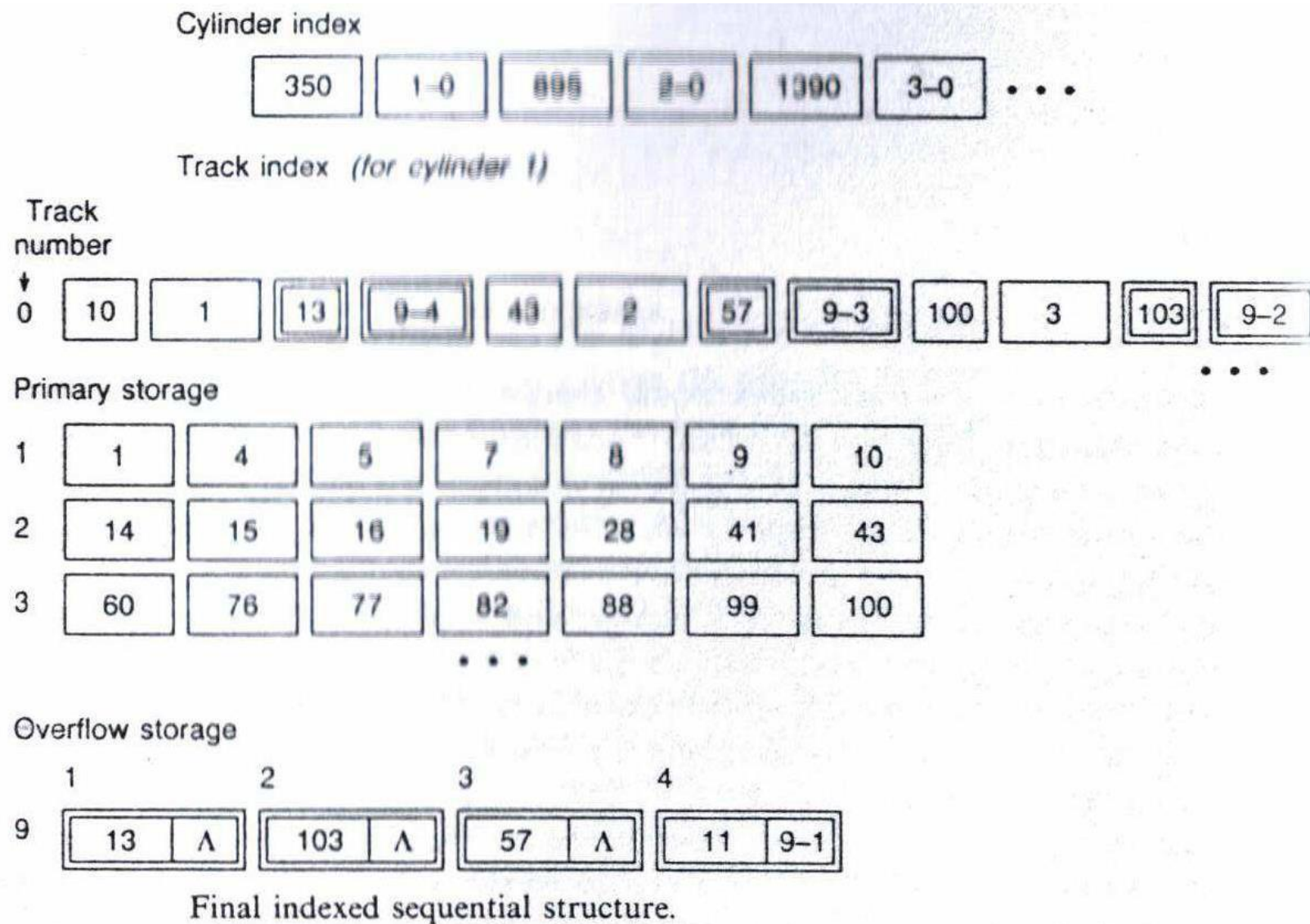
Wouldn't it be better to insert it onto the end of track 1? In that way it would be placed in the overflow area and *no* records would need to be moved in the primary area of track 2.

But this adds complexity to the algorithm. It requires we search two tracks on each insertion and choose between them. No gain on the average for retrieval.



The key of the final insertion record is 11.

- $11 < 13$ . To maintain order, it should be placed into the overflow area.
- All we need to do is to place 11 into its proper position in the chain of overflow records.
- What we are doing here, then is inserting into a linked list.

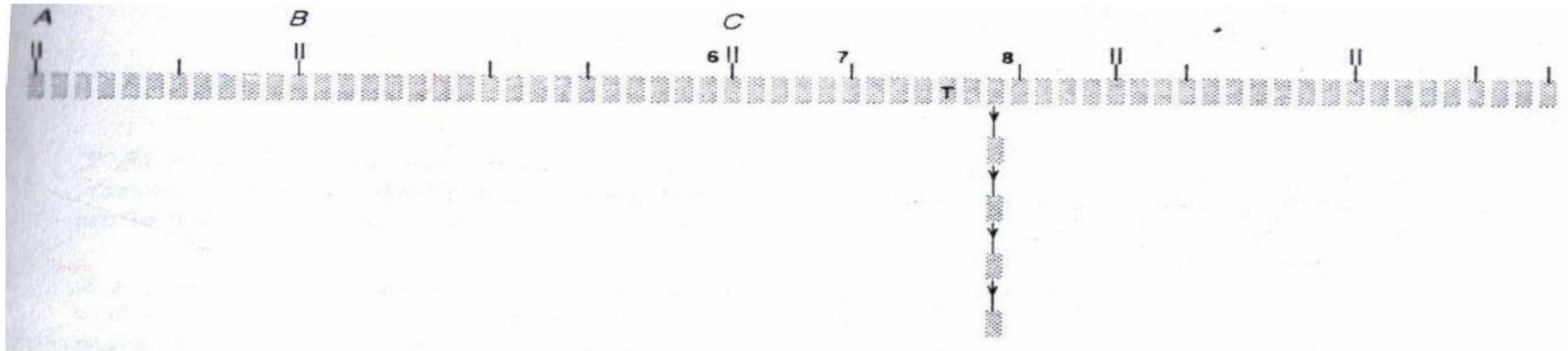


## Data Access

- To *sequentially* process the data in an indexed sequential file entails stepping through the indexes to locate track after track of primary and overflow records.
- There may be intervening tracks containing overflow records or other info from a different file.
- We also need to use the index when we move from one cylinder to another since the cylinders may not be contiguous.
- Sequential processing of an indexed sequential file is faster than for a directly organized file but slower than for a sequentially organized file.
- For *direct* access, we search the cylinder index, then search the primary and overflow entries of the track index. We follow the link field of the overflow entry.
  - We need to search *either* the primary or overflow area but *not both*, since each track has *two* pairs of entries associated with it.
  - Which takes more time: a successful search or an unsuccessful one?
  - In the case of a sequentially organized file and a directly organized file, an unsuccessful search takes more time.
  - But with an indexed sequential file, both require the *same* amount of time. Since the records in an indexed sequential file are *ordered*, we can end a search when we move beyond the related position.

## Performance

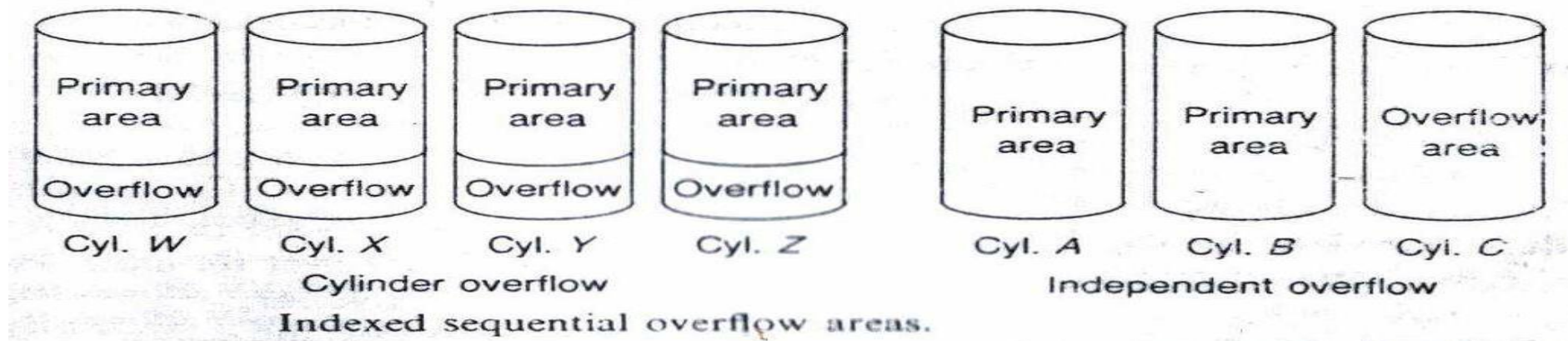
- Having overflow factor of one simplifies the insertion algorithm, but it can reduce the efficiency of search. Consider



- 4 overflow records mean 4 separate access of disk. We are adding to the depth of the subtree. Such a long chain would degrade performance for both insertions and retrievals.
- To correct the imbalance, it is advisable to reorganize the file periodically.
- Reorganization takes time, and usually done at nonpeak times.
- Another technique to reduce the effect of long overflow chains is to include unused space or dummy records in the primary area which records inserted later can occupy.
- If the record is in primary area, it is merely removed from the file.
- Removing a record may be accomplished by replacing it with a dummy record or marking its storage location as being available (tombstone). It is not worthwhile to move records since an insertion would soon occur.
- On an overflow chain, it is relinked and space occupied by the record is returned to the available space list.
- May be implemented in either one of two ways:  
(1) cylinder overflow, (2) independent overflow.
- With cylinder overflow, the overflow area is on the *same* cylinder as the primary storage area.



- With an independent overflow area, several cylinders of primary area share a separate cylinder for overflow.
- The advantage of the cylinder overflow is that disk head does not need to be repositioned to another cylinder to access an overflow record.



In time-sharing environments, the advantages of a cylinder overflow may be lost. To lessen such an effect, some operating systems schedule input/output to minimize seek time.

The advantage of the independent overflow area is that primary cylinders may *share* a single overflow region so that the amount of overflow space that needs to be reserved per cylinder can be lessened.

- We don't need to reserve more overflow tracks per cylinder, because we can share storage with other cylinders.
- Understanding its basic structures allows us to use it more effectively.



- It helps if we sorted the records into descending order before insertion. Consider the following keys of records (B is sorted)

| A  | B  | A  | B  |
|----|----|----|----|
| 39 | 72 | 22 | 36 |
| 47 | 68 | 68 | 22 |
| 72 | 47 | 36 | 17 |
| 17 | 39 |    |    |

- When the primary area is full, the indexed sequential insertion procedure performs an insertion sort on the items in an overflow chain of a particular group.
- That means stepping through the records already in the chain to locate the proper position for the insertion.
- Each element encountered on the chain requires an access of auxiliary storage, plus we need to do this processing for each insertion.
- What is the advantage then of descending order?
- The new record that we are inserting is always at the top of the chain for the group of records being inserted. That eliminates accessing a record multiple times during the insertion process for a group of records.
- To increase the number of transactions, one technique is to place the cylinder index (or top level master index) in primary memory since all accesses of an indexed sequential file need to use it.