UNIT-1

Chapter 1(Introduction and overview)

1. Asymptotic Notations
2. One Dimensional array
3. Multi Dimensional array
4. Pointer arrays.

Chapter 2 (Linked lists)

1. Definition
2. Single linked list
3. Circular linked list
4. Double linked list
5. Circular Double linked list
6. Application of linked lists.

Chapter 1(Introduction and overview)

Overview

Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.

Interface − Each data structure has an interface. Interface represents the set of Operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
Implementation − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

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.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.
Data Search − Consider an inventory of 1 million(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) items every time slowing down the search. As data grows, search will become slower.

Processor Speed − Processor speed although being very high, falls limited if the data grows to billion records.

Multiple Requests − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.
Worst Case − This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is $f(n)$ then this operation will not take more than $f(n)$ time, where $f(n)$ represents function of n.
Average Case − This is the scenario depicting the average execution time of an

operation of a data structure. If an operation takes $f(n)$ time in execution, then m operations will take $mf(n)$ time.
Best Case − This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes $f(n)$ time in execution, then the actual operation may take time as the random number which would be maximum as $f(n)$.
Basic Terminology
Data − Data are values or set of values.
Data Item − Data item refers to single unit of values.
Group Items − Data items that are divided into sub items are called as Group Items.
Elementary Items − Data items that cannot be divided are called as Elementary Items.
Attribute and Entity − An entity is that which contains certain attributes or properties, which may be assigned values.
Entity Set − Entities of similar attributes form an entity set.
Field − Field is a single elementary unit of information representing an attribute of an entity.
Record − Record is a collection of field values of a given entity.
File − File is a collection of records of the entities in a given entity set.

Algorithms –basics

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.
From the data structure point of view, following are some important categories of algorithms −
Search − Algorithm to search an item in a data structure.
Sort − Algorithm to sort items in a certain order.
Insert − Algorithm to insert item in a data structure.
Update − Algorithm to update an existing item in a data structure**.**
Delete − Algorithm to delete an existing item from a data structure.
Characteristics of an Algorithm
Not all procedures can be called an algorithm. An algorithm should have the following characteristics −
Unambiguous − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
Input − An algorithm should have 0 or more well-defined inputs.
Output − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
Finiteness − Algorithms must terminate after a finite number of steps.
Feasibility − Should be feasible with the available resources.
Independent − An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

- There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent.

- Algorithms are never written to support a particular programmingcode.
- As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

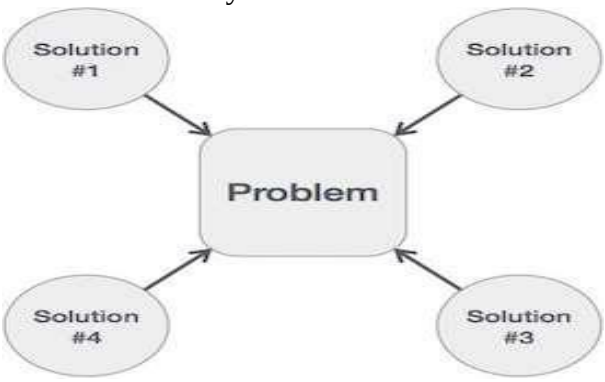
Problem − Design an algorithm to add two numbers and display the result.

step 1 − START
step 2 − declare three integers a, b & c
step 3 − define values of a & b
step 4 − add values of a & b
step 5 − store output of step 4 to c
step 6 − print c
step 7 − STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

step 1 − START ADD
step 2 − get values of a & b
step 3 − c ← a + b
step 4 − display c
step 5 − STOP

- In design and analysis of algorithms, usually the second method is used to describe an algorithm.
- It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions.
- He can observe what operations are being used and how the process is flowing. Writing step numbers, is optional.
- We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



- Hence, many solution algorithms can be derived for a given problem.
- The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

A Priori Analysis − This is a theoretical analysis of an algorithm. Efficiency of an

algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
A Posterior Analysis − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.
We shall learn about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity
Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.
Time Factor – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
Space Factor − Space is measured by counting the maximum memory space required by the algorithm.
The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

Space Complexity
Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −
A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.
Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.
Following is a simple example that tries to explain the concept −
Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop

- Here we have three variables A, B, and C and one constant.
- Hence S(P) = 1+3. Now,space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

- Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion.
- Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

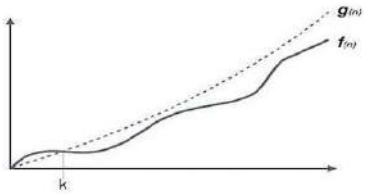
For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c*n,
where c is the time taken for the addition of two bits.
Here, we observe that T(n) grows linearly as the input size increases.

Asymptotic Notations

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance.
- Using asymptotic analysis, we can verywell conclude the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time.
- Other than the "input" all other factors are considered constant.
- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.
- For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2).
- This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

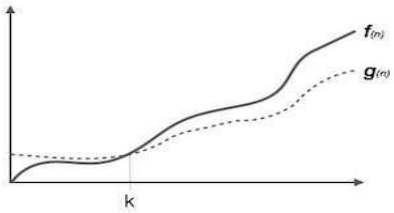
Usually, the time required by an algorithm falls under three types −
Best Case − Minimum time required for program execution.
Average Case − Average time required for program execution.
Worst Case − Maximum time required for program execution.
Asymptotic Notations
Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.
O Notation
Ω Notation
θ Notation
Big Oh Notation, O

- The notation O(n) is the formal way to express the upper bound of an algorithm's running time.
- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



For example,
for a function f(n),O(f(n)) = { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }
Omega Notation, Ω

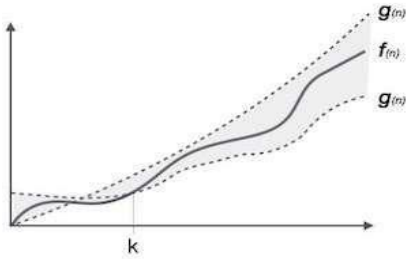
- The notation Ω(n) is the formal way to express the lower bound of an algorithm's runningtime.
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



For example, for a function f(n)
Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }

Theta Notation, θ

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.
- It is represented as follows –



θ(f(n)) = { g(n) if and only if g(n) = O(f(n)) and g(n) = Ω(f(n)) for all n >n0. }

Common Asymptotic Notations
Following is a list of some common asymptotic notations:
constant − O(1)
logarithmic − O(log n)
linear − O(n)
n log n − O(n log n)
quadratic − O(n2)
cubic − O(n3)
polynomial − nO(1)
exponential − 2O(n)

Data structure Basic concepts

This chapter explains the basic terms related to data structure.

Data Definition
Data Definition defines a particular data with the following characteristics.
Atomic − Definition should define a single concept.
Traceable − Definition should be able to be mapped to some data element.
Accurate − Definition should be unambiguous.
Clear and Concise − Definition should be understandable.

Data Object
Data Object represents an object having a data.

Data Type
Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

Built-in Data Type
Derived Data Type

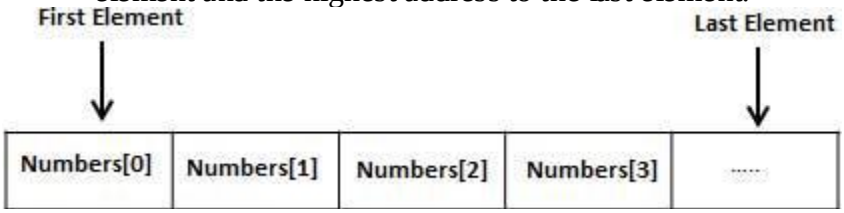
Built-in Data Type
Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.
Integers
Boolean (true, false)
Floating (Decimal numbers)
Character and Strings

Derived Data Type
Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −
List
Array
Stack
Queue

Basic Operations
The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.
Traversing
Searching
Insertion
Deletion
Sorting
Merging

One Dimensional array

- Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type.
- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.
- A specific element in an array is accessed by an index.
- All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

- To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [arraySize];

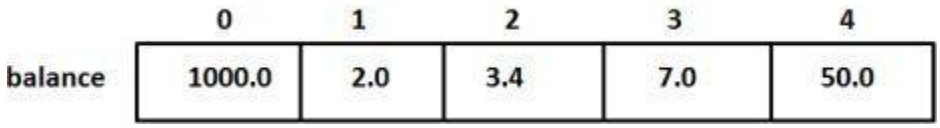
- This is called a single-dimensional array.
- The arraySize must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called balance of type double, use this statement −

double balance[10];

Here balance is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

- You can initialize an array in C either one by one or using a single statement as follows −
  double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
- The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].
- If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −
  double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
  You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −
  balance[4] = 50.0;
  The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −



Accessing Array Elements

- An element is accessed by indexing the array name.
- This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

- The above statement will take the 10th element from the array and assign the value to salary variable.

  The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays −

```
#include <stdio.h>
 int main () {
 int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;
 /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
   }
    /* output each array element's value */
   for (j = 0; j < 10; j++ ) {
      printf("Element[%d] = %d\n", j, n[j] );
   }
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

Element[0] = 100
Element[1] = 101

Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer −

S.N.	Concept & Description
1	Multi-dimensional arrays C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.
3	Return array from a function C allows a function to return an array.
4	Pointer to an array You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

Multi Dimensional array

- C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration −
  type name[size1][size2]...[sizeN];
- For example, the following declaration creates a three dimensional integer array −
  int threedim[5][10][4];

Two-dimensional Arrays

- The simplest form of multidimensional array is the two-dimensional array.
- A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −
  type arrayName [x][y];
  Where type can be any valid C data type and arrayName will be a valid C identifier.
- A two-dimensional array can be considered as a table which will have x number of rows and y number of columns.
- A two-dimensional array a, which contains three rows and four columns can be shown as follows −

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Thus, every element in the array a is identified by an element name of the form a[i][j], where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays

- Multidimensional arrays may be initialized by specifying bracketed values for each row.
- Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
  {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
  {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
  {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

- An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

- The above statement will take the 4th element from the 3rd row of the array.

Let us check the following program where we have used a nested loop to handle a two-dimensional array −

```
#include <stdio.h>
 int main () {
   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;
   /* output each array element's value */
   for ( i = 0; i < 5; i++ ) {
   for ( j = 0; j < 2; j++ ) {
        printf("a[%d][%d] = %d\n", i,j, a[i][j] );
     }
   }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

- As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Pointer arrays

- It is most likely that you would not understand this section until you are through with the chapter 'Pointers'.
- Assuming you have some understanding of pointers in C, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration −
  double balance[50];
  balance is a pointer to &balance[0], which is the address of the first element of the array balance.

Thus, the following program fragment assigns p as the address of the first element of balance −

```
double *p;
double balance[10];
p = balance;
```

- It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].
- Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2) and so on. Given below is the example to show all the concepts discussed above −

```
#include <stdio.h>
int main () {
   /* an array with 5 elements */
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *p;
   int i;
  p = balance;
   /* output each array element's value */
   printf( "Array values using pointer\n");
  for ( i = 0; i < 5; i++ ) {
      printf("*(p + %d) : %f\n",  i, *(p + i) );
   }
   printf( "Array values using balance as address\n");
   for ( i = 0; i < 5; i++ ) {
      printf("*(balance + %d) : %f\n",  i, *(balance + i) );
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Array values using pointer
*(p + 0) : 1000.000000
*(p + 1) : 2.000000
*(p + 2) : 3.400000
*(p + 3) : 17.000000
*(p + 4) : 50.000000
Array values using balance as address
*(balance + 0) : 1000.000000
*(balance + 1) : 2.000000
*(balance + 2) : 3.400000
*(balance + 3) : 17.000000
*(balance + 4) : 50.000000
```

In the above example, p is a pointer to double, which means it can store the address of a variable of double type. Once we have the address in p, *p will give us the value available at the address stored in p, as we have shown in the above example.

Chapter 2 (Linked lists)

Definition
Single linked list
Circular linked list
Double linked list
Circular Double linked list
Application of linked lists.

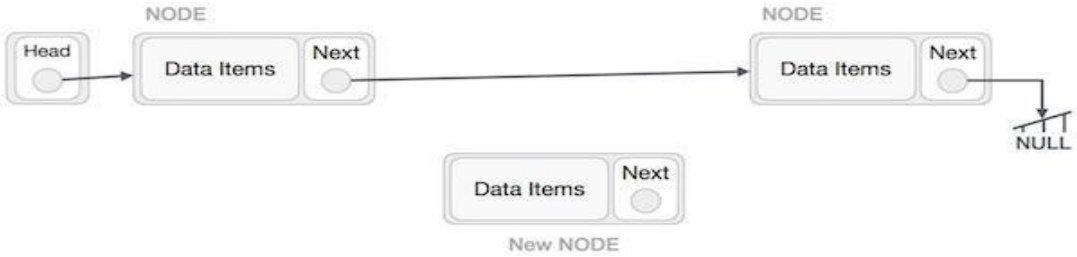
LINKED LISTS:
DEFINITION: A linked list is a sequence of data structures, which are connected together via links.
Linked List is a sequence of links which contains items. Each link contains a connection to another link.
Linked list is the second most-used data structure after array.
Following are the important terms to understand the concept of Linked List.
Link − Each link of a linked list can store a data called an element.
Next − Each link of a linked list contains a link to the next link called Next.
LinkedList − A Linked List contains the connection link to the first link called First.
Linked List Representation
Linked list can be visualized as a chain of nodes, where every node points to the next node.

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

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List
Following are the various types of linked list.
Single Linked List − Item navigation is forward only.
Doubly Linked List − Items can be navigated forward and backward.
Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.
SINGLE LINKED LISTS:
Basic Operations
Following are the basic operations supported by a list.
Insertion − Adds an element at the beginning of the list.
Deletion − Deletes an element at the beginning of the list.
Display − Displays the complete list.
Search − Searches an element using the given key.
Delete − Deletes an element using the given key.
Insertion Operation

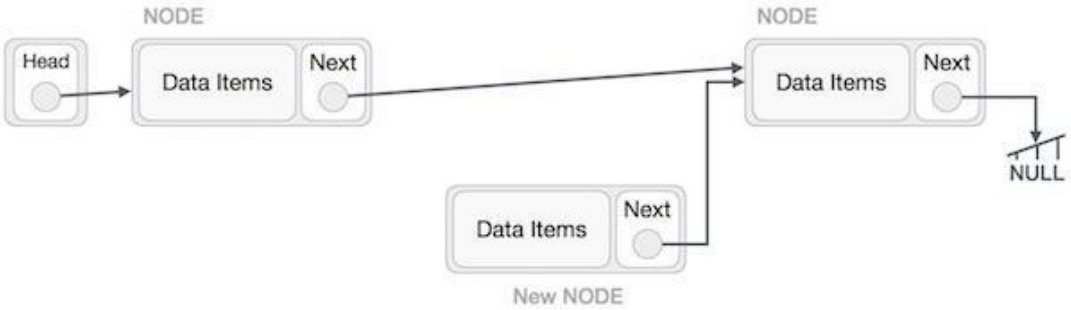
- Adding a new node in linked list is a more than one step activity.
- We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C −

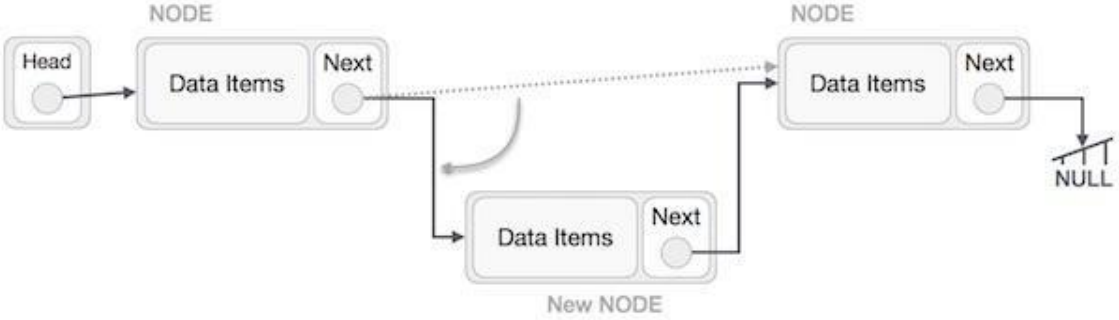
NewNode.next −> RightNode;

It should look like this −

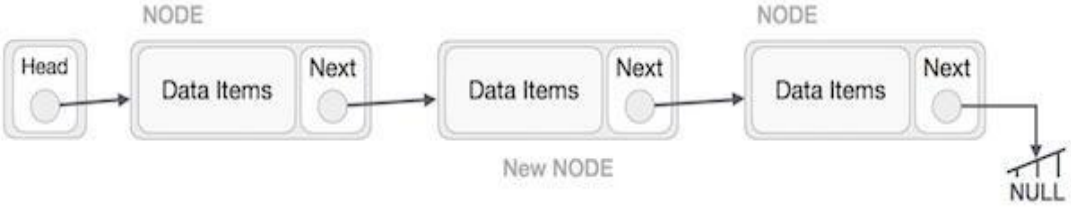


Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;



This will put the new node in the middle of the two. The new list should look like this −



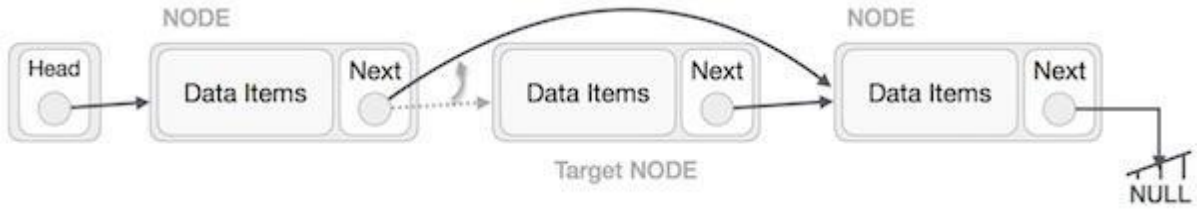
Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

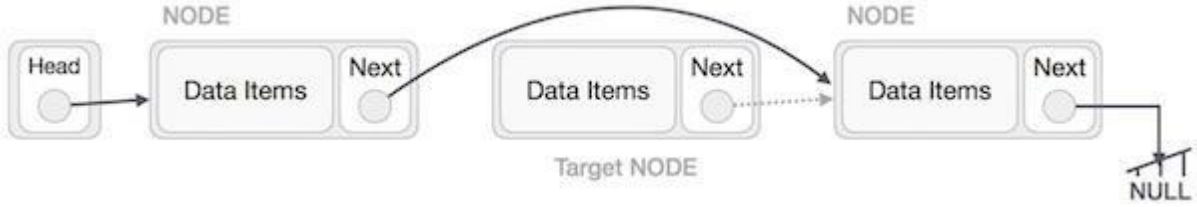
The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −> TargetNode.next;

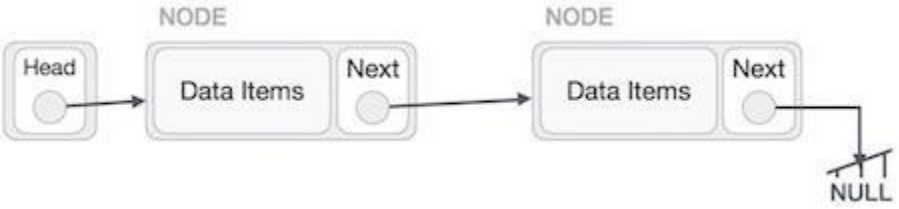


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what what the target node is pointing at.

TargetNode.next −> NULL;

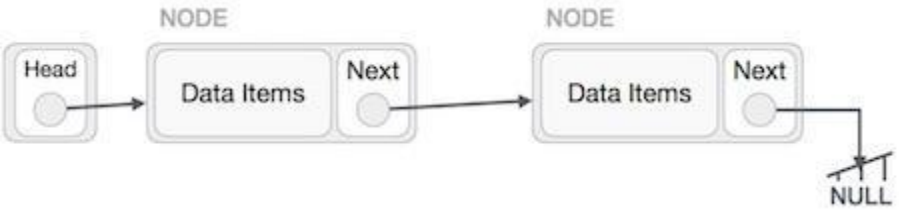


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

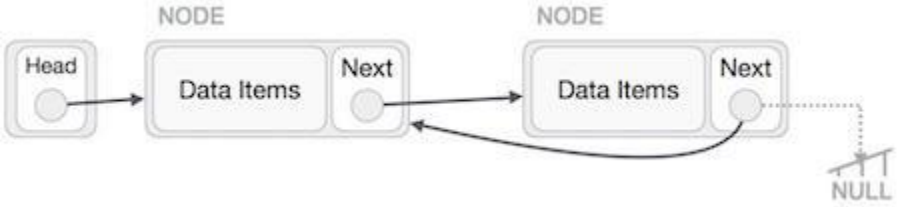


Reverse Operation

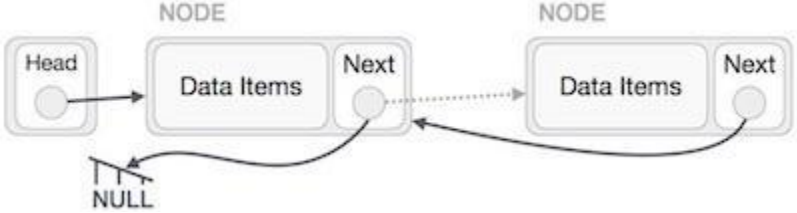
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



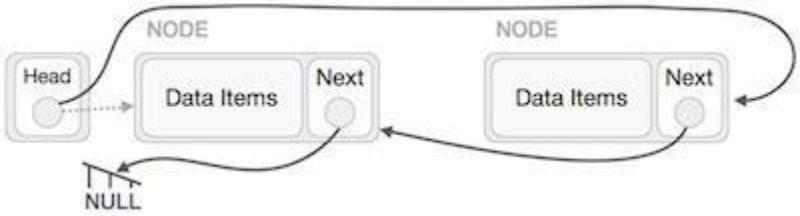
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



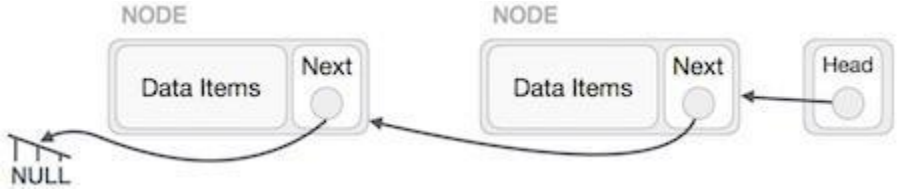
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



The linked list is now reversed.

DOUBLY LINKED LISTS

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

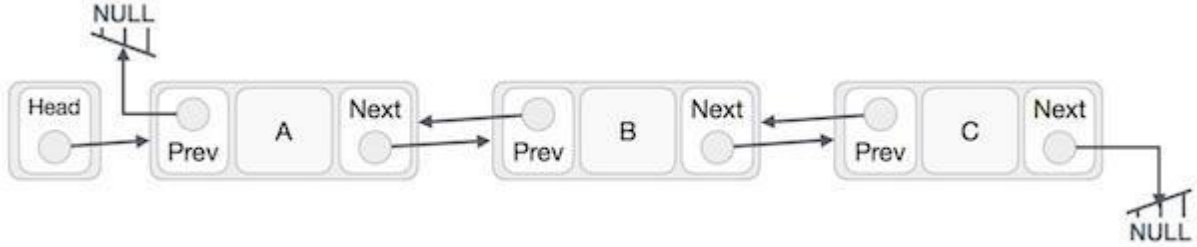
Link − Each link of a linked list can store a data called an element.

Next − Each link of a linked list contains a link to the next link called Next.

Prev − Each link of a linked list contains a link to the previous link called Prev.

LinkedList − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



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

Doubly Linked List contains a link element called first and last.

- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by a list.

Insertion − Adds an element at the beginning of the list.

Deletion − Deletes an element at the beginning of the list.

Insert Last − Adds an element at the end of the list.

Delete Last − Deletes an element from the end of the list.

Insert After − Adds an element after an item of the list.

Delete − Deletes an element from the list using the key.

Display forward − Displays the complete list in a forward manner.

Display backward − Displays the complete list in a backward manner.

Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data) {
  //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;
            if(isEmpty()) {
      //make it the last link
      last = link;
   } else {
      //update first prev link
      head->prev = link;
   }
  //point it to old first link
   link->next = head;
            //point first to new first link
   head = link;
}
```

Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

Example

```
//delete first item
struct node* deleteFirst() {
  //save reference to first link
   struct node *tempLink = head;
            //if only one link
   if(head->next == NULL) {
      last = NULL;
   } else {
      head->next->prev = NULL;
   }
          head = head->next;
            //return the deleted link
   return tempLink;
}
```

Insertion at the End of an Operation

Following code demonstrates the insertion operation at the last position of a doubly linked list.

Example

```
//insert link at the last location
void insertLast(int key, int data) {

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;
            if(isEmpty()) {
      //make it the last link
      last = link;
   } else {
      //make link a new last link
```

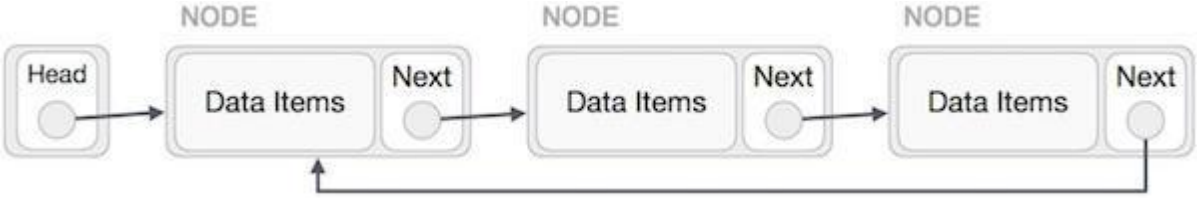
```
    last->next = link;
      //mark old last node as prev of new link
    link->prev = last;
  }
 //point last to new last node
  last = link;
}
```

Circular Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

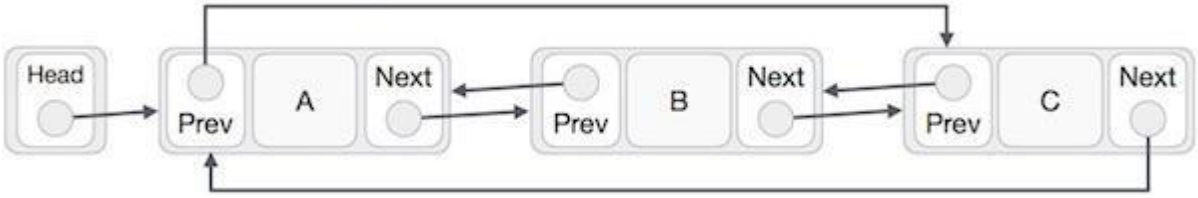
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



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

The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.

The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

insert − Inserts an element at the start of the list.

delete − Deletes an element from the start of the list.

display − Displays the list.

Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data) {
  //create a link
  struct node *link = (struct node*) malloc(sizeof(struct node));
  link->key = key;
  link->data= data;
          if (isEmpty()) {
    head = link;
    head->next = head;
  } else {
    //point it to old first node
    link->next = head;
```

```
    //point first to new first node
    head = link;
  }
}
```

Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
//delete first item
struct node * deleteFirst() {
  //save reference to first link
  struct node *tempLink = head;
          if(head->next == head) {
    head = NULL;
    return tempLink;
  }
 //mark next to first link as first
  head = head->next;
         //return the deleted link
  return tempLink;
}
```

Display List Operation

Following code demonstrates the display list operation in a circular linked list.

```
//display the list
void printList() {
  struct node *ptr = head;
  printf("\n[ ");
          //start from the beginning
  if(head != NULL) {
    while(ptr->next != ptr) {
      printf("(%d,%d) ",ptr->key,ptr->data);
      ptr = ptr->next;
    }
  }

  printf(" ]") }
```

END OF UNIT -1