UNIT – III TREES

3.1 Basic Terminologies

Terminologies used in Trees

- **Root** – The top node in a tree.
- **Child** – A node directly connected to another node when moving away from the Root.
- **Parent** – The converse notion of a *child*.
- **Siblings** – Nodes with the same parent.
- **Descendant** – A node reachable by repeated proceeding from parent to child.
- **Ancestor** – A node reachable by repeated proceeding from child to parent.
- **Leaf** – A node with no children.
- **Internal node** – A node with at least one child.
- **External node** – A node with no children.
- **Degree** – Number of sub trees of a node.
- **Edge** – Connection between one node to another.
- **Path** – A sequence of nodes and edges connecting a node with a descendant.
- **Level** – The level of a node is defined by 1 + (the number of connections between the node and the root).
- **Height of node** – The height of a node is the number of edges on the longest downward path between that node and a leaf.
- **Height of tree** – The height of a tree is the height of its root node.
- **Depth** – The depth of a node is the number of edges from the node to the tree's root node.
- **Forest** – A forest is a set of $n \geq 0$ disjoint trees.

3.2 Definition and Representation

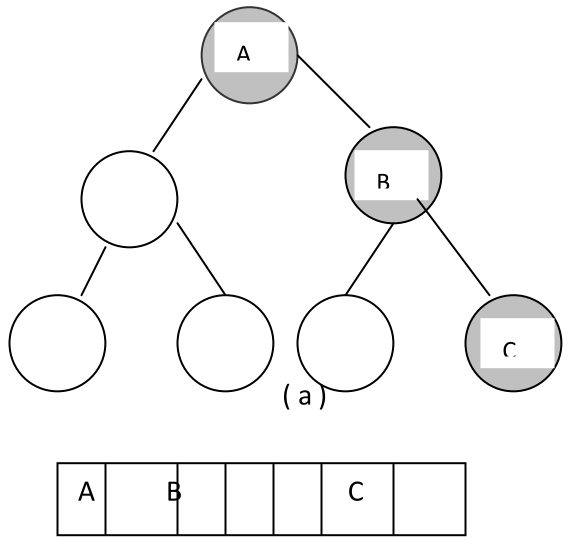
A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

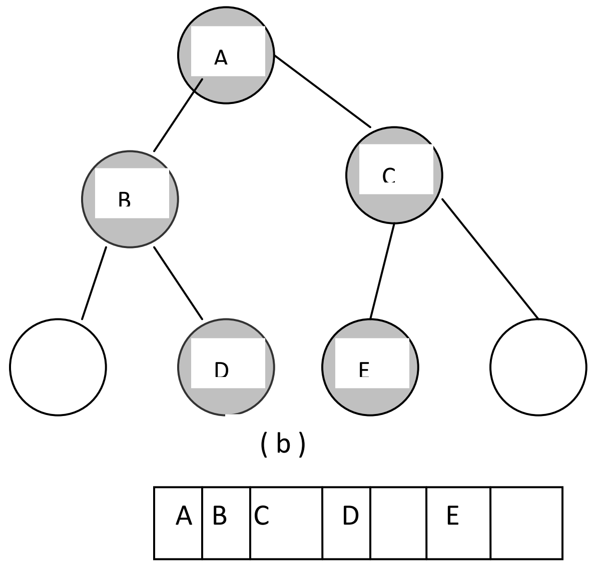
3.3 Representation of Binary Tree

There are two representations used to implement binary trees.

(i) Array Representation (ii) Linked list Representation

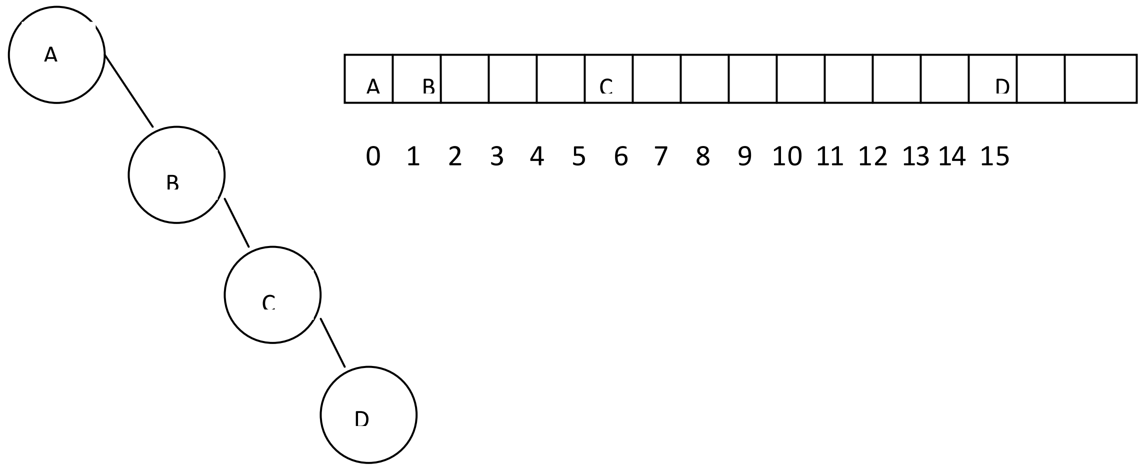
Array Representation: In this, the given binary trees even though are not complete binary trees, they are shown as complete binary trees in which missing elements are un shaded circles.The array representations for the following trees are shown in below.





In array, the elements of the binary are placed in the array according to their number assigned. The array starts indexing from 1.The main drawback of array representation is wasteful of memory when there are many missing elements.

The binary tree with n elements requires array size up to 2^n. Suppose array positions indexing from 0, then array size reduces to 2^n-1. The right skewed binary trees have maximum waste of space. The following right skewed binary tree's array representation is shown as follows.



Linked list Representation: The most popular way to represent a binary tree is by using links or pointers. The node structure used in this representation consists of two pointers and an element for each node. The node structure is given as:

leftchild	element	rightchild

The first field leftchild pointer maintains left child of it. The middle field is the element of the node and last field is the right child pointer maintains right child of it.

Binary tree traversals: There are four ways to traverse a binary tree. They are

(a) Pre order (b) In order (c) Post order (d) Level order

The first three traversals are performed using recursive approach and are done using linked list scheme. In these, the left sub tree is visited before visiting right sub tree. The difference among these is position of visiting the node.

(a) **Pre order:**

i) Visit Root node. ii) Visit Left sub tree. iii) Visit Right Sub tree.

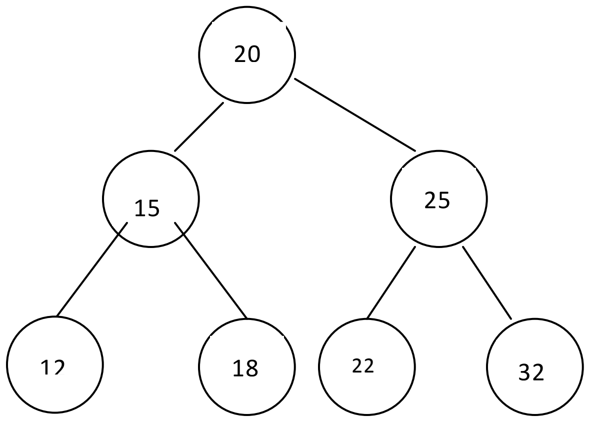
(b) In order:

i) Visit Left sub tree. ii) Visit Root node. iii) Visit Right Sub tree.

(c) Post order:

i) Visit Left sub tree. ii) Visit Root node. iii) Visit Right Sub tree

The following is an example binary tree with pre order, in order, post order and level order traversals:



(a)

pre order is : 20 15 12 18 25 22 32

in order is : 12 15 18 20 22 25 32

post order is : 12 18 15 22 32 25 20

level order is : 20 15 25 12 18 22 32

3.4 Types of Binary trees

3.4.1 Binary Search Trees:

Any empty binary tree is an Binary Search Tree. A nonempty binary search tree has the following properties.

(i) Every element has the key or value and no two elements have the same key. Therefore, all keys in the tree must be distinct.

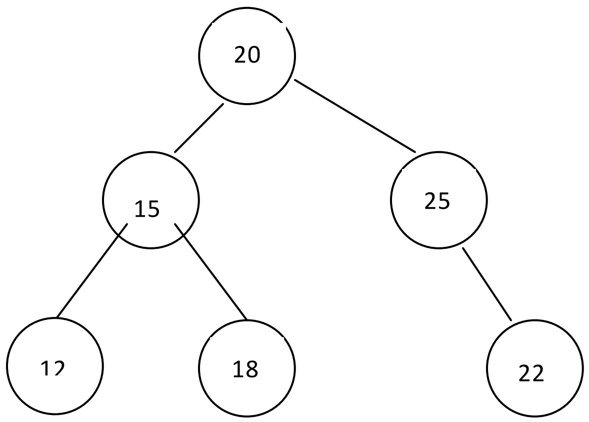
(ii) Any element key in left sub tree is less than the key of the root.

(iii) Any element key in right sub tree is greater than the key of the root.

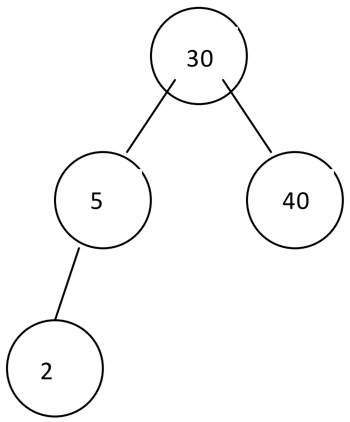
(iv) Both left and right sub trees are also binary search trees.

There is some redundancy in this definition. The properties 2, 3, 4 together imply the keys must be distinct.

Some binary trees in which the elements with distinct keys are shown in the following figures.



(a)



(b)

The number inside a node is the element key. The tree (a) is not a binary search tree because the right sub tree of element 25 violating property 4.It means 22 is smaller than its parent 25. The trees of (b) is not a binary search tree.

When the property all keys are distinct is removed, then property 2 is replaced by smaller or equal and property 3 is replaced by larger or equal. The resulting tree is called a binary search tree with duplicates.

3.4.2 Binary Search tree Applications: It is mainly used in

(i) Histogramming (ii) Best fit bin packaging iii) Crossing Distribution

<u>3.5.2 Heap Trees</u>

In computer science, a **heap** is a specialized tree-based data structure that satisfies the *heap property:* If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap. A heap can be classified further as either a "**max heap**" or a "**min heap**". In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node. In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node. Heaps are crucial in several efficient graph algorithms such as Dijkstra's algorithm, and in the sorting algorithm heap sort. A common implementation of a heap is the binary heap, in which the tree is a complete binary tree (see figure).

In a heap, the highest (or lowest) priority element is always stored at the root, hence the name **heap**. A heap is not a sorted structure and can be regarded as partially ordered. As visible from the heap-diagram, there is no particular relationship among nodes on any given level, even among the siblings. When a heap is a complete binary tree, it has a smallest possible height—a heap with N nodes always has log N height. A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority.

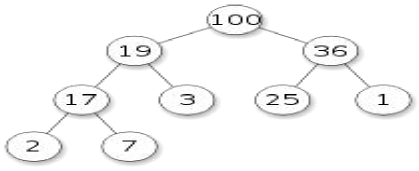


Fig (i) Heap Tree

Note that, as shown in the graphic, there is no implied ordering between siblings or cousins and no implied sequence for an in-order traversal (as there would be in, e.g., a binary search tree). The heap relation mentioned above applies only between nodes and their parents, grandparents, etc. The maximum number of children each node can have depends on the type of heap, but in many types it is at most two, which is known as a binary heap.

The heap is one maximally efficient implementation of an abstract data type called a priority queue, and in fact priority queues are often referred to as "heaps", regardless of how they may be implemented. Note that despite the similarity of the name "heap" to "stack" and "queue", the latter two are abstract data types, while a heap is a specific data structure, and "priority queue" is the proper term for the abstract data type.

A *heap* data structure should not be confused with *the heap* which is a common name for the pool of memory from which dynamically allocated memory is allocated. The term was originally used only for the data structure.

3.6 Height Balanced Trees

Height balanced trees (or AVL trees) is named after its two inventors, G.M. Adelson-Velskii and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information." As the name sugests AVL trees are used for organizing information.

AVL trees are used for performing search operations on high dimension external data storage.

For example, a phone call list may generate a huge database which may be recorded only on external hard drives, hard-disks or other storage devices.

AVL is a data structure that allows storing data such that it may be used efficiently regarding the operations that may be performed and the resources that are needed. AVL trees are binary search trees, wich have the balance propriety. The balance property is true for any node and it states: "the height of the left subtree of any node differs from the height of the right subtree by 1".

The structure of the nodes of a balanced tree can be represented like:

```
struct NodeAVL{
int key;
 int ech;
node *left, *right;
};
```

Where: - key represents the tag of the node(integer number),

ech represents the balancing factor - left and right represent pointers to the left and right children.

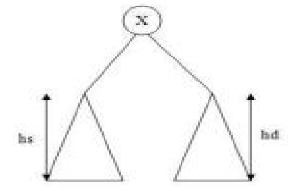


Figure 1. Balancing factor equals $h_d - h_s$

Here are some important notions:

[1] The length of the longest road from the root node to one of the terminal nodes is what we call the height of a tree.

[2] The difference between the height of the right sub tree and the height of the left sub tree is what we call the balancing factor.

[3] The binary tree is balanced when all the balancing factors of all the nodes are -1,0,+1. We present below the function drum, which calculates the longest road from the current node , meaning the height of a sub tree:

```
void path(NodeAVL* p,int &max, int length)
{
if (p!=NULL)
{
path(p->right,max,length+1);
if((p->left==NULL)&&(p->right==NULL)&&(maxleft,max,length+1);
}
}
```

Using this function we can determine the balancing indicator of each node of the tree with the function balance Factor:

```
 void balanceFactor(NodeAVL *p)
{
 int max1,max2;
max1=1;
max2=1;
if(p->left!=NULL)
path(p->left,max1,1);
else max1=0;
if(p->right!=NULL)
 path(p->right,max2,1);
 else max2=0;
 p->ech=max2-max1;
}
```

3.7 B Trees

AVL and red black trees are used when the dictionary is small enough to reside in internal memory. The search trees of higher degree are needed to get better performance for external dictionaries. ISAM is used to get good sequential and random access for external dictionaries.

3.8 m-way search trees**:** An m-way search tree may be empty. If it is not empty, it must satisfy the following properties.

(i) In the tree, each internal node has up to m children and has elements between 1 & m-1.

(ii) Every internal node with p elements has p+1 children.

(iii) Consider any node with p elements. Let k1, k2,….kp be the keys of these elements. The elements are ordered such that k1<k2<……<kp. Let c0,c1,……cp be p+1 children of the node. The elements in the sub tree with root c0 have keys smaller than k1, those in the sub tree with root cp have keys larger than kp and those in sub tree with root c_i have the keys larger than ki but smaller than ki+1 where i≤ i < p. Although external nodes are included when defining m-way search tree, external nodes are not represented physically in actual representation.

Consider the following seven way search tree, which have external nodes as solid squares and all other nodes are internal nodes. The root has 2 elements and 3 children. The middle child of the root has 6 elements and 7 children in which 6 are external nodes.

3.9.1.1 searching: To search for an element with key 31, begin checking with root first. The searching drops to middle child of the root because 31 lies between 10 & 80. Since search finds k2 (=30) < 31 < k3 (=40), search drops to third sub tree of this node. There 31 < k1 (=32), search moves to first sun tree of this node but external node is reached. When search terminates at external node, the key is not found. Otherwise means search exited at any internal node, the key is found.

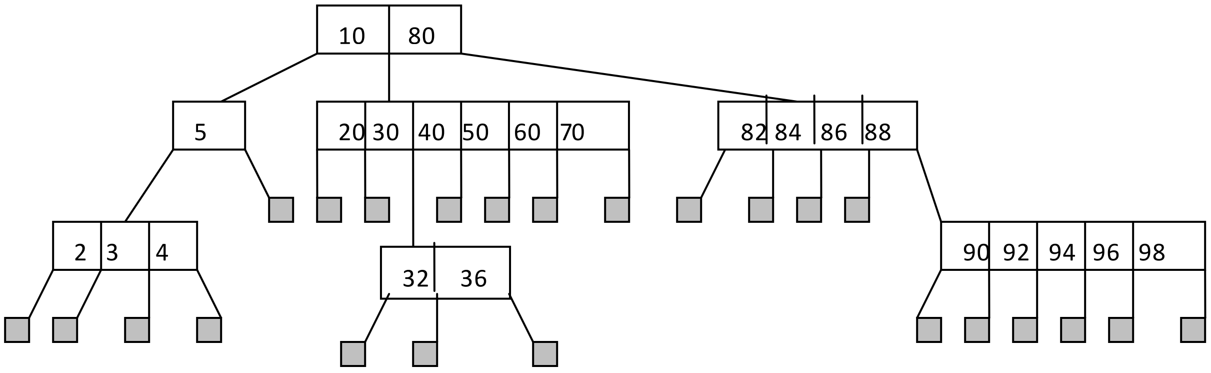


Fig:1 **m-way search tree example**

3.9.1.2 insertion: To insert an element with key, first search the tree. If it doesn't contain the key, then insert it. To insert the key 31, the search begins at root, then goes to middle child of root, then third child of this node, then first child of this node which is the external node. Since the node [32,36] can hold up to 6 elements, the new element is inserted as first element of this node.

Another example is insert an element with key 65, the search terminates at sixth child of middle child of the root. The new element is created and inserted into there.

3.9.1.3 deletion: To delete an element with key, first search for it. If it is there, then delete it.(i) If the deleted element is 20 in fig above, search for it. The searching ends at first element of the middle child of the root. Since it's childs c0 & c1 are 0, it can be deleted easily and results a node [30,40,50,60,70].
(ii) To delete 84, search ends at second element in third child of the root. Since it's childs c1 & c2 are 0, it can be deleted easily by resulting [82,86,88].
(iii) To delete an element with key 5, more work to be done. It has a nonnull c0 and c1 is external node. The largest key in the c0 is brought to the deleted node place.
(iv) To delete an element with key 10, the root take either largest in its c0 or smallest in c1. Suppose 5 of c0 was brought to the root, 4 in the c0 of deleted node 5 is brought to 5's old place.

3.9.1.4 Height: An m-way search tree of height h may have as few as h elements and as many as m^h-1. This upper bound is obtained from the levels 1 through h -1 has exactly m children and nodes at level h have no children. Since each of these nodes has m-1 elements, the number of elements is m^h-1.
An 200 way search tree of height 5 can hold $32 * 10^{10}$ -1 elements but might fold as few as 5 only.

<u>3.9.2 B – trees</u>: A B – tree of order m is an m – way search tree. If the B – tree is not empty, the corresponding extended tree must satisfy the following properties:

(i) The root has at least two children.

(ii) All internal nodes other than the root have at least [m/2] children.

(iii) All external nodes are at the same level.

The seven way search tree in fig 1 is not a B-tree of order 7 because

(i) all the external nodes are not on the same level.

(ii) Some of the internal nodes have two (= node [5]) and three (= node 32,36]) children which is not satisfying 2nd property i.e [7/2]=4 children.

The following is an B-tree of order 7:

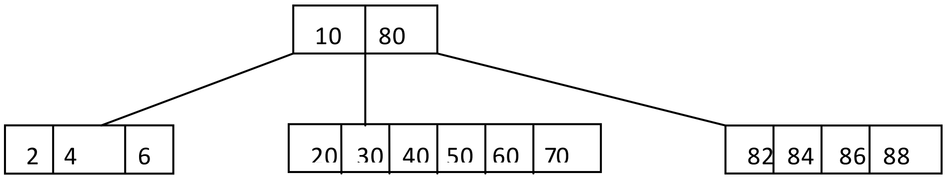


Fig 2 : **B – tree of order 7**

In a B-tree of order 2, all the internal nodes have exactly two children. This requirement coupled with all external nodes on the same level results full binary trees.

In a B-tree of order 3, internal nodes have either two or three children. It is also called 2 -3 tree.

In a B- tree of order 4, internal nodes have two, three or four children. These are also referred as 2-3-4 trees and are called 2-4 trees. The following is an 2-3 tree. It becomes 2-3-4 tree when adding 14 and 16 to left child of 20.

<u>3.9.2.1 Height of a B – tree</u>: Let T be a B – tree of order m and height be h. Let m=[d/2] and n be number of elements in T.

(a) $2d^{h-1} - 1 \leq n \leq m^{h} - 1$

(b) $\log_{m}(n+1) \leq h \leq \log_{d}(n+1/2)+1$

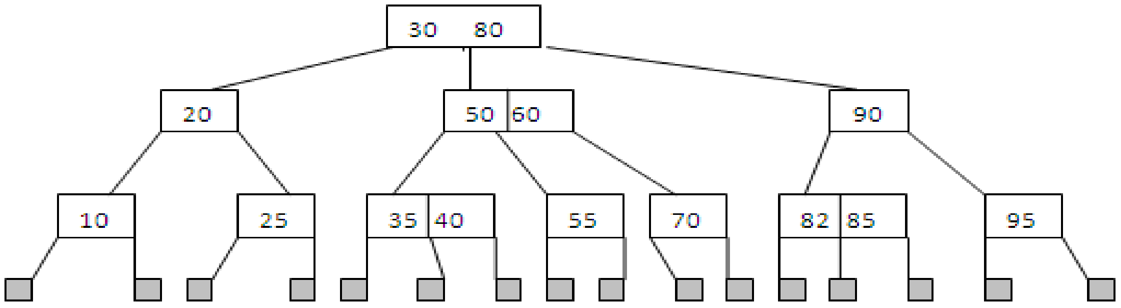


Fig 3: **B – tree of order 3**

3.9.2.1 Searching: The searching an element in a B-tree is same as algorithm used for m-way search tree. Searching an element in an internal node of a B-tree of height takes at most h because all internal nodes need to be checked during the search.

3.9.2.2 Insertion: To insert an element, first search for the presence of the element with same key. If such an element is found, insertion fails because duplicates are allowed. When searching is unsuccessful, then insert new element into the last internal node encountered on the search path. To insert an element with key 3 into **Fig 2**, search terminates at second child of left child of the root. It can be inserted into [2,4,6] node results [2,3,4,6] node since this node hold up to 6 elements. The number of disk accesses to do this is 3 in which two accesses for reading root and then its left child and another for writing out modified node after insertion. It can be shown as follows.

To insert an element with key 25 in B-tree of order 7 (Fig 2), the element goes into middle child of the root i.e [20,30,40,50,60,70] but this node is full. When element goes to full node, the overfull node need to be split as follows.

Let the overfull node be P=[20,25,30,40,50,60,70]. Let it has m elements and m+1 children. It can be denoted as m, c_0, (e_1, c_1), (e_2, c_2),………, (e_m, c_m).

where ei 's indicate elements and ci's represent children pointers. The node is spli around ed where d=[m/2].

The elements to the left remain in P and to the right move into a new node Q but P & Q must contain at least [m/2] children.

The element ed moved to the parent of P. The format of P and Q are

P: d-1, c0, (e1,c1), …….., (ed-1,cd-1)

Q: m-d, cd, (ed+1,cd+1),……., (em,cm).

In this case, the overfull node is 7,0,(20,0),(25,0),(30,0),(40,0),(50,0),(60,0),(70,0). It can be split around d=4 which yields P=3,0,(20,0),25,0),30,0) and Q=3,0,(50,0),(60,0),(70,0). The e4=40 moved to P's parent. Here, it is the root. It can be shown as follows. The number of disk accesses required is 5 in which two for searching the proper position in the tree, two for writing out the split nodes and one for writing modified root.

To insert element with key 44 into B – tree of order 3 like Fig 3 (c), the element goes to [35,40] node. Since it is full, the overfull node is [35,40,44] can be represented as 3,0,(35,0),(40,0),(44,0). It can be split around d=[3/2]=2 yields

P=1,0,(35,0) and Q=1,0,(44,0). The element with key 40 move to P's parent

A =[50,60]. The resulted overfull node be 3,P,(40,Q),(50,C),(60,D) where C & D are pointers to the nodes [55] & [60]. The overfull node **A** be split to create a new node B. The new A & B are

A: 1,P,(40,Q) and B: 1, C, (60,D) .

Before insertion, root format is R: 2, S, (30,A),(80,T) where S & T are first and third sub trees of the root. After insertion, the overfull node is R: 3, S, (30,A), (50,B),(80,T). This node is split around d=[3/2]=2 yields R: 1,S,(30,A) and U= 1,B, (80,T). The element 50 moved to R's parent. Since R has no parent, it can be created as new root and that has format 1,R, (50,U). The resulting tree is shown as below.

The total number of disk accesses is 10 in which 3 accesses for reading [30,80],[50,60] and [30,40], six disk accesses for writing out 3 split nodes and one for writing out new root.
When insertion cause s nodes to split, the number of disk accesses is h (reading the nodes on the search path) + 2s (to write out two split parts of each node that is split) + 1 (to write out new root). The total number of disk accesses is h+2s+1 which is at most 3h+1.

<u>3.9.2.3 Deletion</u>: Deletion first divided into 2 cases. (1) The element to be deleted in a node whose children are external nodes. (2) the element to be deleted from a non leaf. case (2) is transformed into case (1) by either largest element in its left neighboring sub tree or smallest element in its right neighboring sub tree.

(i) To delete an element with key 80 in Fig 3 (a), the suitable replacement used is either the largest element in its left sub tree 70 or smallest element 82 in its right sub tree.

(ii) To delete an element with key 80 in Fig 3 (c), the replacing element used is either 70 or 82. If 82 is selected, the problem of deleting 82 from the leaf remains.

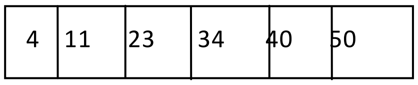
The (ii) falls into 2 cases. One is delete an element from a leaf contains more than the minimum number of elements (1 if the leaf is also root and [m/2]-1 if it is not) requires to simply write out modified node.

The deletion from a B-tree of height h is when merging tales at levels h,h-1,…..and 3 and getting an element from a nearest sibling at level 2 is 3h.

Note: when the element size is large relative to the size of a key, the following node structure is used. $s,c_0,(k_1,c_1,p_1),(k_2,c_2,p_2),\ldots\ldots.,(k_s, c_s, p_s)$ where s is the number of elements in the node, k i 's are element keys, pi's are the disk locations of the corresponding elements and ci's are children pointers.

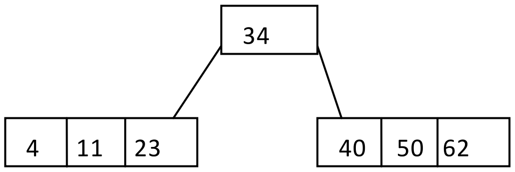
<u>Exercise 1:</u> Draw the B-tree of order 7 resulting from inserting the following keys into an empty tree T: 4,40,23,50,11,34,62,78,66,22,90,59,25,72,64,77,39 & 12.

Step 1: Since it is a B-tree of order 7, the maximum number of elements a node contain is 6.

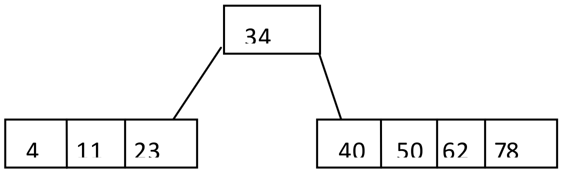


Step 2: Next element to be inserted is 62, but this is full because the maximum number of children that internal node have is 7 and minimum number of children is 4[=7/2].

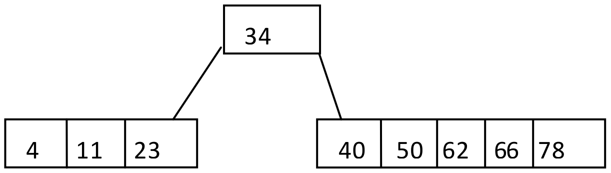
The overfull node is P= [4,11, 23,34, 40,50,62]. It can be split around e4=34. The elements to left are remain in P and to the right in Q. The element e4 goes to the node parent.



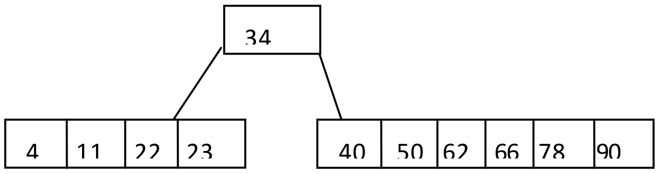
Step 3: The next element 78 goes to root right child.



Step 4: The next element inserted is 66, it goes into root right child.

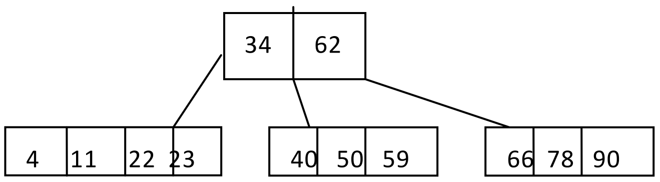


Step 5: The next elements 22 & 90 goes into root left child and root right child respectively. Now, root right child is full. If any element insert into it needs split.

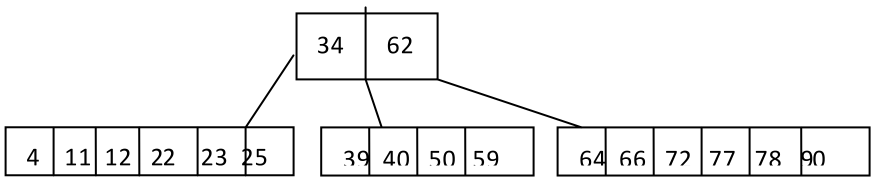


Step 6: The next element 59 goes into root right child and it becomes overfull node. This needs to be split.

Let C= [40,50,59,62,66,78,90]. It can be split around e4=62 leaves C=[40,50,59] and D=[66,78,90]. The element 62 moves to the parent. Now, the root is 34,62]. Its Childs are P,C & D.



Step 7: The elements 25, 72,64,77,39 & 12 are inserted in which 25 & 12 are in root first sub tree, 39 to root middle sub tree and 64,72 & 77 to root third sub tree.



3.8 Red Black trees

It is an extended binary search tree in which null pointers are replaced by external nodes and also in which every node is either colored red or black. It has same properties of binary search tree and has the following additional properties.

i) The root and all external nodes are in black color.

ii) No two red nodes occur consecutively on any path from root to external nodes.

iii) The number of black nodes are same on any path from root to external node.

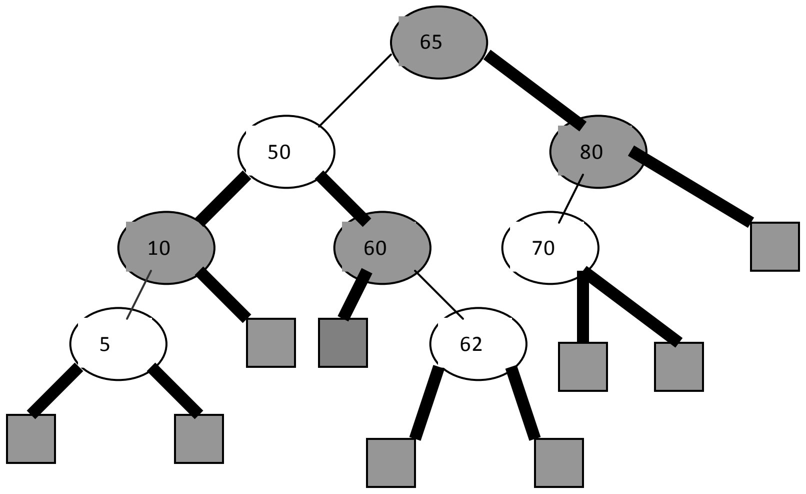
Another definition arises from assigning colors to the pointers between a node and its children. The pointer from a parent to a black node is black and to a red node is red. The additional properties for this are:

The pointers from an internal node to external nodes are black.

i) No two red pointers occur consecutively on any path from root to external node.

ii) The number black pointers are same on any path.

Note: If node colors are known, then deduce pointer colors and vice versa.



: **Red black tree example**

The above tree is an example for red black tree because it obeys red black tree properties and also it is an binary search tree.

i) The root with key 65 and external nodes are in black color.

ii) No two red nodes occur consecutively on any path from root to external node.

iii) The number of black nodes is same on any path from root to external nodes i.e 2.

In the above figure, the external nodes are shaded squares, root and black nodes are shaded circles, red nodes are unshaded circles, black pointers are thick lines and red pointers are thin lines.

The rank of a node in red black tree is the number of black pointers or number of black nodes minus 1. The rank of a root (key 65) is 2, the rank of its left tree is 2, the rank of its right tree is 1.

Representation of a Red Black tree: The node structure of a red black tree is same as the node structure employed for binary search tree but an extra field is included to represent the node color. There are two schemes used to represent a node.

i) One stores the color of the node in addition to its left, right pointers and its data.

ii) Another uses the color of its two children in addition to data, left and right pointers.

3.10.1 Searching for a node in red black tree: The search for a key in red black tree is same as search operation of binary search tree. The time complexity of search is **O(log n).**

Insertion: Inserting an element into a red black tree is same as the method employed for binary search trees. The only concern is determines which color the node must set to. Suppose the new node set to black, the path from root to that node has one extra black node that violates black condition. The alternative is set the node to red also violates red condition. This imbalance set right to balance using rotations.

Suppose u be the newly red node inserted and p-u be its consecutive red node which is parent of u and also has grandparent named gp-u which is a black node. Depending on position of u relative to p-u, gp-u and also gp-u other child color, the imbalances are classified **as LLb, LLr, LRb, LRr, RRb, RRr, RLb and RLr**.

3.10.2 deletion: The deletion in red black tree is same as the method used for binary search tree such as deleting a leaf, a node that has single sub tree or a node that have two sub trees. If the deletion results imbalance that calls the rotations.

If the deleted node is red, there is no problem with this. Suppose deleted node is black, then the specific path in which black node is deleted have shortage of one black node. It violates black condition which results imbalance red black tree. This imbalance is classified as L or R based on whether deleted node v occurs to the left or right of its parent node p-v.

(i) If the sibling node s-v is black, then imbalance is classified as Lb or Rb. Based on whether s-v has 0 or 1 0r 2 red children, Lb and Rb imbalances are further classified as Lb0,Lb1,Lb2 and Rb0,Rb1 and Rb2 respectively.

(ii) If the s-v is a red node, then the imbalance is classified as Lr or Rr. Based on whether s-v has 0 or 1 or 2 red children, Lr and Rr imbalances are further classified as Lr0,Lr1 & Lr2 and Rr0,Rr1 & Rr2.

During rebalancing, the deleted node v is replaced by its descend.

GRAPHS

3.11 Introduction

Graphs are a fundamental data structure in the world of programming, and this is no less so on top coder. Usually appearing as the hard problem in Division 2, or the medium or hard problem in Division 1, there are many different forms solving a graph problem can take. They can range in difficulty from finding a path on a 2D grid from a start location to an end location, to something as hard as finding the maximum amount of water that you can route through a set of pipes, each of which has a maximum capacity (also known as the maximum-flow minimum-cut problem – which we will discuss later). Knowing the correct data structures to use with graph problems is critical. A problem that appears intractable may prove to be a few lines with the proper data structure, and luckily for us the standard libraries of the languages used by top coder help us a great deal here!

- Define a graph *G = (V, E)* by defining a pair of sets:
  1. V = a set of **vertices**
  2. E = a set of **edges**

3.12 Graph Terminology

Edges:

- ✓ Each edge is defined by a pair of vertices
- ✓ An edge **connects** the vertices that define it
- ✓ In some cases, the vertices can be the same.

Vertices:

- ✓ Vertices also called **nodes** .
- ✓ Denote vertices with labels

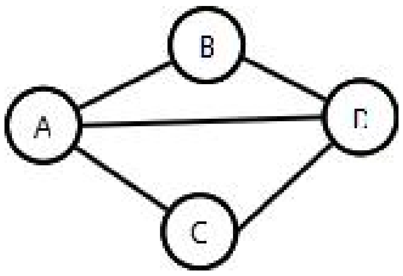
Representation:

- ✓ Represent vertices with circles, perhaps containing a label.
- ✓ Represent edges with lines between circles.

Example:

V = {A, B, C,D}

E = {(A,B),(A,C),(A,D),(B,D),(C,D)}



3.13 Representation of Graphs

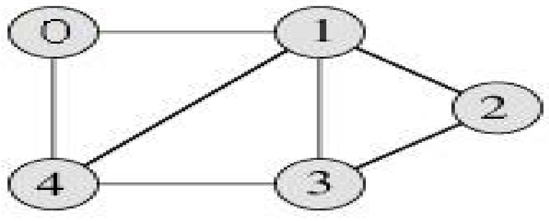
Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.

2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. This can be easily viewed by http://graph.facebook.com/barnwal.aashish where barnwal.aashish is the profile name.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix

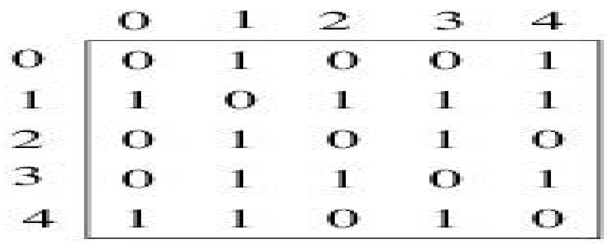
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

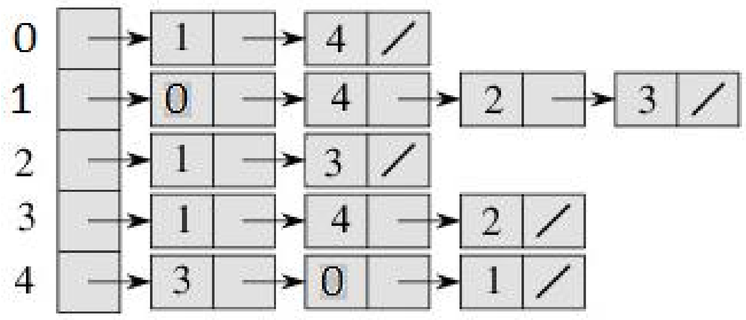
Adjacency Matrix Representation of the above graph

Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Adjacency List Representation of the above Graph

3.14 Operations on Graphs

Graph Traversals:

In computer science, **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.

i) 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.

```
procedure DFS(G,v):
    label v as explored
    for all edges e in G.incidentEdges(v) do
        if edge e is unexplored then
            w ← G.adjacentVertex(v,e)
            if vertex w is unexplored then
                label e as a discovered edge
                recursively call DFS(G,w)
            else
                label e as a back edge
```

ii) 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.

```
procedure BFS(G,v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            o ← G.adjacentVertex(t,e)
            if o is not marked:
                mark o
                enqueue o onto Q
    return null
```

3.15 Applications of Graphs

1) Shortest Path:

In graph theory, the **shortest path problem** is the **problem** of finding a **path** between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

This is probably the most often used algorithm. It may be applied in situations where the shortest path between 2 points is needed.

Examples of such applications would be:

- Computer games - finding the best/shortest route from one point to another.
- Maps - finding the shortest/cheapest path for a car from one city to another, by using given roads.
- May be used to find the fastest way for a car to get from one point to another inside a certain city. E.g. satellite navigation system that shows to drivers which way they should better go.

2) Minimal Spanning Tree:

Consider some communications stations (for telephony, cable television, Internet etc.) and a list of possible connections between them, having different costs. Find the cheapest way to connect these stations in a network, so that a station is connected to any other (directly, or through intermediate stations). This may be used for example to connect villages to cable television, or to Internet.