UNIT-4
TREES

Trees: Non-Linear data structure

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.1.1 shows a tree and a non-tree.

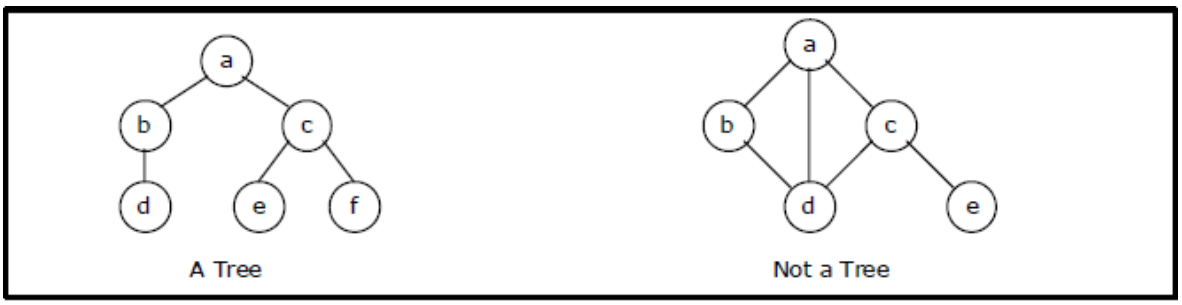


Figure 5.1.1 A Tree and a not a tree

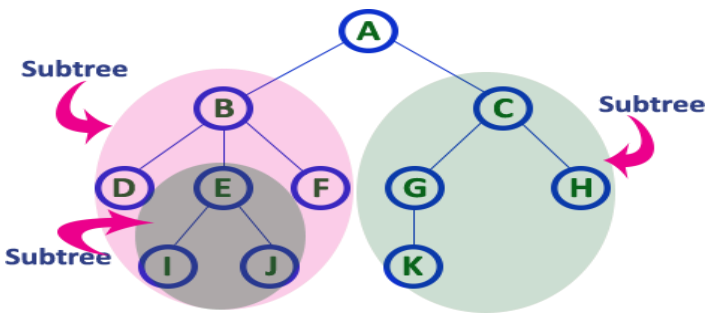
Tree is a popular data structure used in wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

A tree data structure can also be defined as follows...

A tree is a finite set of one or more nodes such that:

There is a specially designated node called the root. The remaining nodes are partitioned into n>=0 disjoint sets T1, ..., Tn, where each of these sets is a tree. We call T1, ..., Tn are the subtrees of the root.



A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

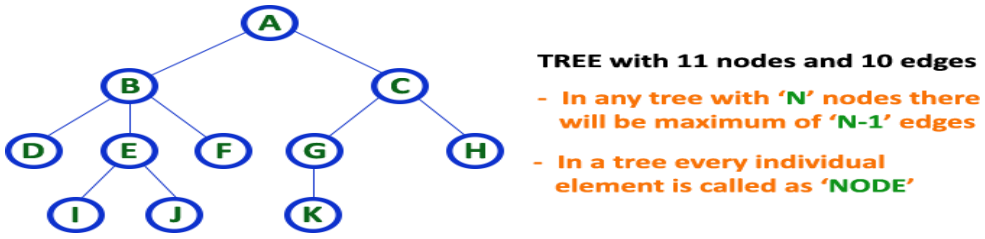
Advantages of trees

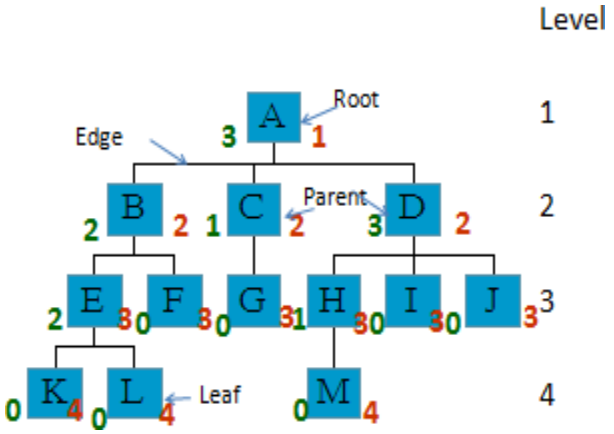
Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move sub trees around with minimum effort

Introduction Terminology

In a Tree, Every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure. Example





1. Node

A **node** is a structure which may contain a value or condition, or represent a separate data structure (which could be a **tree** of its own). Each **node** in a **tree** has zero or more child **nodes**,

2. Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree. In above tree, **A** is a **Root** node

3. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

4. Parent

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children". e.g., Parent (A,B,C,D).

5. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes. e.g., Children of D are (H, I,J).

6. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes. Ex: Siblings (B,C, D)

7. Leaf Node(External Node)

In a tree data structure, the node which does not have a child (or) node with degree zero is called as LEAF Node. In simple words, a leaf is a node with no child.

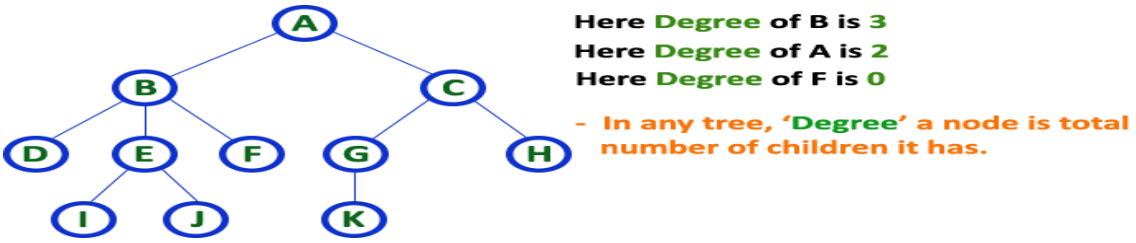
In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node. Ex: (K,L,F,G,M,I,J)

8. Internal Nodes(Non Leaf Node)

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.
In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes. Ex:B,C,D,E,H

9. Degree

In a tree data structure, the total number of children of a node (or)number of subtrees of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



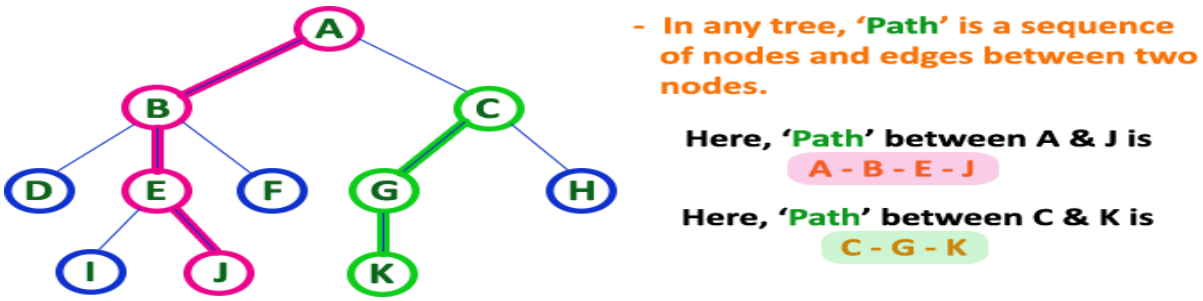
10. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step). Some authors start root level with 1.

11. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that

path. In below example the path A - B - E - J has length 4.



12. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

13. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the
longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.

14. Ancestor

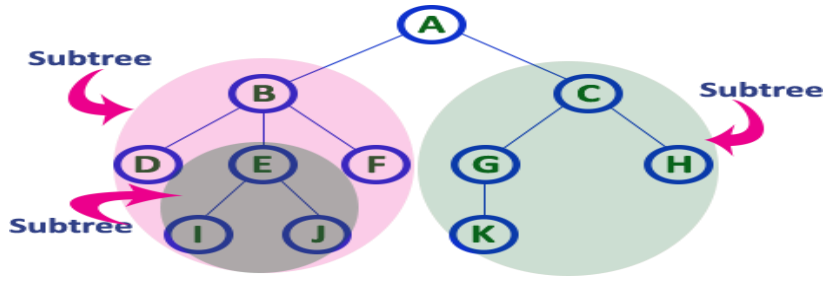
Any predecessor node on the path from root to that node
Ex: J= E,B,A
\ K= G,C,A

15. Descendant

Any successor node on the path from that node to leaf node
Ex: C= G,K,H
B= D,E,I,J,F

16. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Tree Representations

A tree data structure can be represented in 3 methods. Those methods are as follows

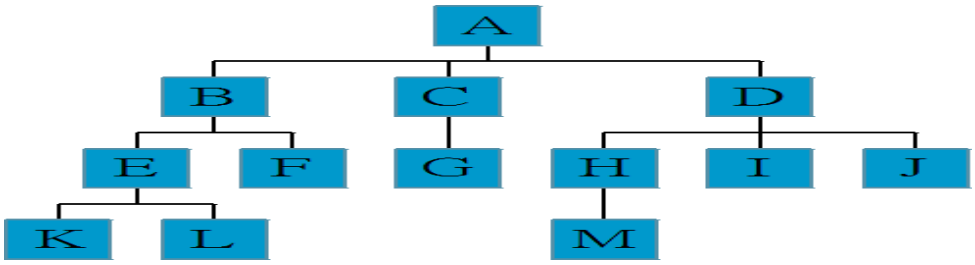
... 1. List Representation

2. Left Child - Right Sibling

Representation

3. Representation as a Degree –Two Tree

Consider the following tree...



1. List Representation

In this representation, we use two types of nodes one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other node directly. This process repeats for all the nodes in the tree.

The above tree example can be represented using List representation as follows...

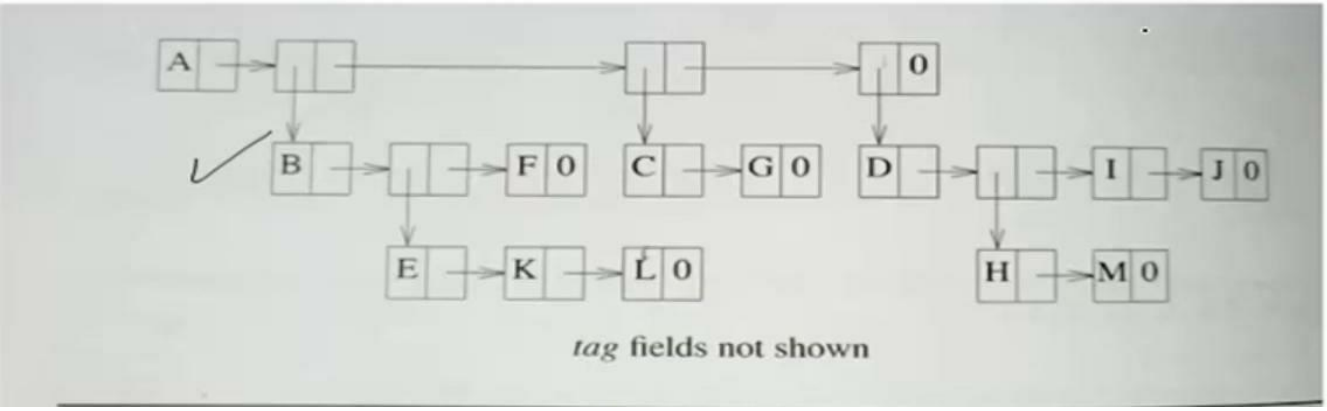


Fig: List representation of above Tree

List Representation

– (A (B (E (K, L), F), C (G), D (H (M), I, J)))

– The root comes first, followed by a list of sub-trees

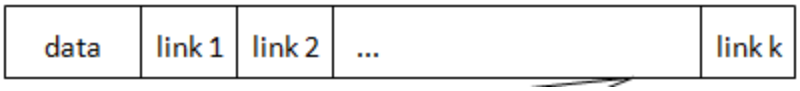
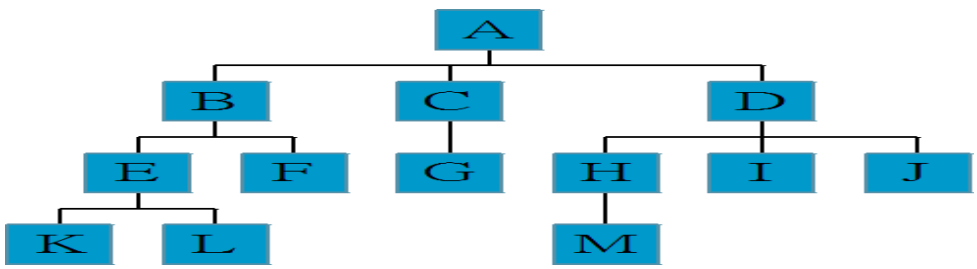


Fig: Possible node structure for a tree of degree k

2. Left Child - Right Sibling Representation

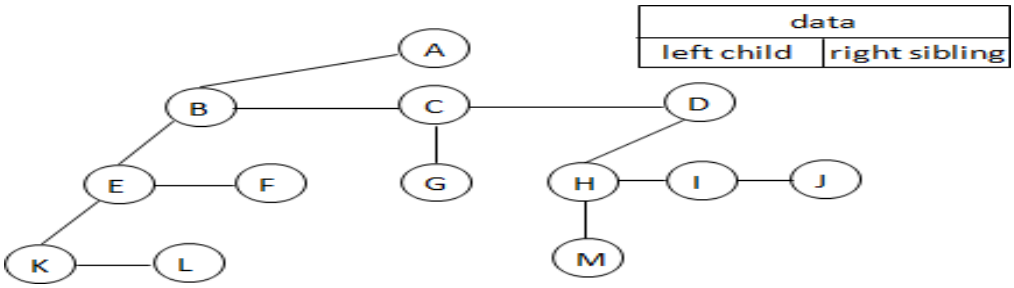
In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...



In this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right

sibling node otherwise that field stores NULL.

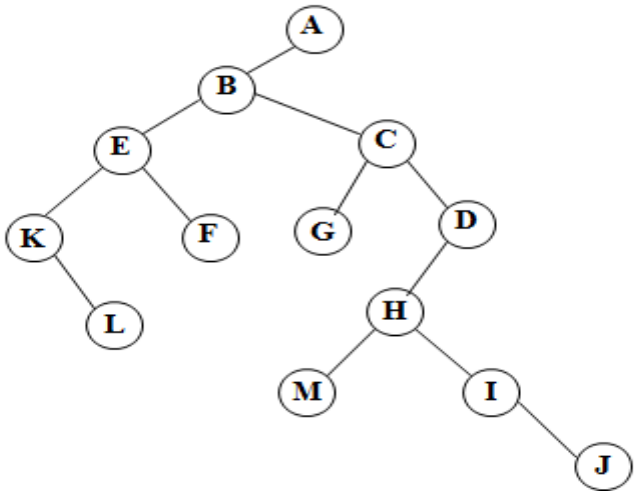
The above tree example can be represented using Left Child - Right Sibling representation as follows...



3) Representation as a Degree –Two Tree

To obtain degree-two tree representation of a tree, rotate the right- sibling pointers in the left child- right sibling tree clockwise by 45 degrees. In a degree-two representation, the two children of anode are referred as left and right children.

***Figure 5.6:** Left child-right child tree representation of a tree (p.191)*



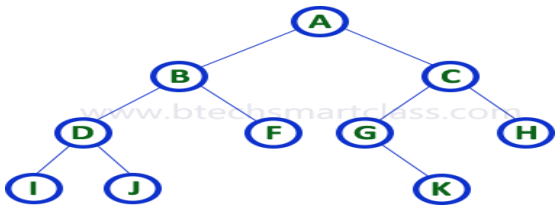
Binary Trees

Introduction

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children. Example



There are different types of binary trees and they are...

1. Strictly Binary Tree/full binary tree/proper Binary tree

   Each node contains exactly two children's

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree

2. Complete Binary Tree/Perfect Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2 level number of nodes. For example at level 2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

3. Incomplete Binary Tree/Almost Complete Binary Treee

Every node must have two children in all levels except in last level but filled from left to right

4. Left skewed binary tree:

   Every node should have only left children

5. Right skewed binary tree:

   Every node should have only Right children

Properties of Binary Trees

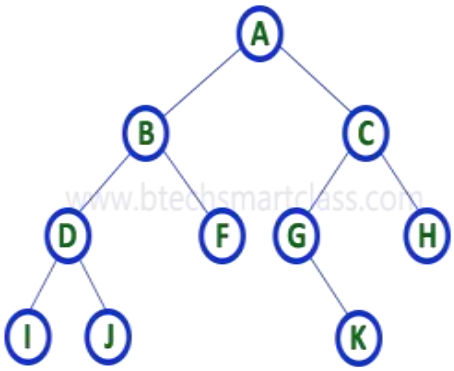
- The maximum number of nodes on level i of a binary tree is : 2^i
- The maximum number of nodes in a binary tree of height h is : $2^{h+1}-1$
- The minimum number of nodes in a binary tree of height h is : h+1
- Total number of leaf nodes in binary tree= total number of nodes with 2 children+1
- In a binary tree T has ' n' nodes then it has = n-1 edges

Binary Tree Representation

There are two ways to represent binary trees. These are:

- Using arrays
- Using Linked lists

In the computer's memory, a binary tree can be maintained either by using a linked representation or by using a sequential representation.



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.

Consider the above example of a binary tree and it is represented as follows...

A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

Here, we will discuss about array representation of binary tree. For this we need to number the nodes of the BT. This numbering can start from 0 to (n-1) or from 1 to n.

Lets derive the positions of nodes and their parent and child nodes in the array.

CASE 1: When we use **0 index based sequencing,**

Suppose parent node is an index p.

Then, the left_child node is at index (2*p)+ 1.

The right_child node is at index (2*p) + 2.

The parent node is at index (c-1) /2

Root node is at index 0.

left_child is at index 1.

Right_child is at index 2.

CASE 2: When we use **1 index based sequencing,**

Suppose parent node is at **index p,**

left_node is at **index (2*p).**

Right_node is at **index (2*p)+1**.

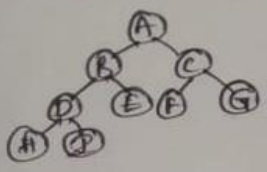
The parent node is at index (c) /2

Root node is at index 1.

left_child is at index 2.

Right_child is at index 3.

1. Array representation: -



A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

case1

A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9

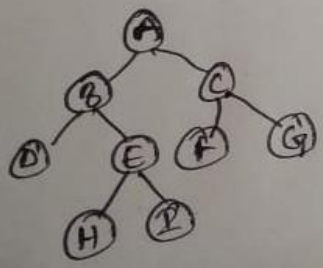
case2

These all formulas are applicable for complete binary only.

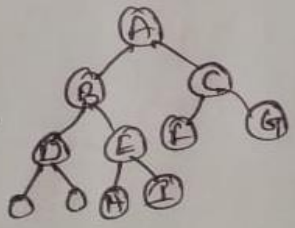
If given tree is not in complete-binary tree, first we should convert and then apply these formulas as follows.

case1:
If a node is at i th index
→ the left child at $(2*i)+1$
→ right child at $(2*i)+2$
→ Parent at $\left\lfloor \frac{(i-1)}{2} \right\rfloor$

case2:-
If node is at i th index
→ left child at $(2*i)$
→ right child at $[(2*i)+1]$
→ Parent at $\left\lfloor \frac{i}{2} \right\rfloor$



⇒



case1

A	B	C	D	E	F	G	.	.	H	I
0	1	2	3	4	5	6	7	8	9	10

case2

A	B	C	D	E	F	G	.	.	H	I
1	2	3	4	5	6	7	8	9	10	11

2. *Linked list Representation of Binary Tree*

In the linked representation of a binary tree, every node
will have three parts: the data element, a pointer to the left node, and a pointer to the right node. So in C, the binary tree is built with a node type given below.

```
struct node
    {
            struct node *left;
            int data;
            struct node *right;
    };
```

Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree. If ROOT = NULL, then the tree is empty. Consider the binary tree given in Fig.. The schematic diagram of the linked representation of the binary tree is shown in Fig.

The left position is used to point to the left child of the node or to store the address of the left child of the node. The middle position is used to store the data. Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node. Empty sub-trees are represented using X (meaning NULL).

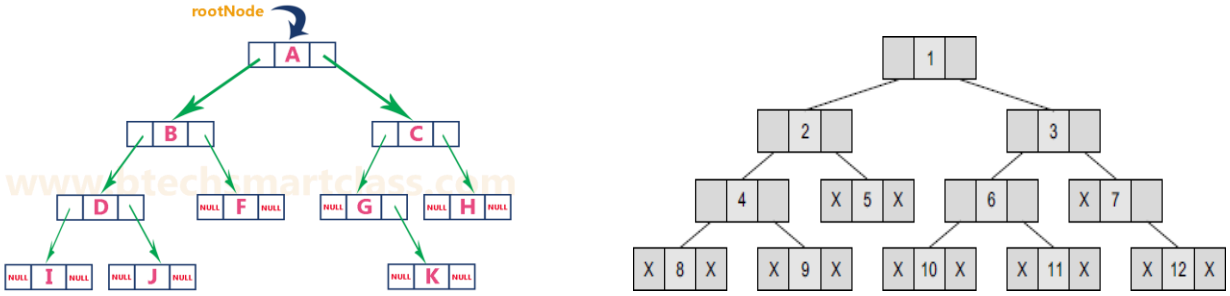


Fig: Linked representation of a binary tree The below fig ,tree is represented in the main memory using a linked list.

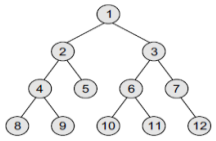


Figure: Binary tree T

		LEFT	DATA	RIGHT
ROOT 3	1	–1	8	–1
	2	–1	10	–1
→	3	5	1	8
	4			
	5	9	2	14
	6			
	7			
	8	20	3	11
	9	1	4	12
	10			
	11	–1	7	18
	12	–1	9	–1
	13			
	14	–1	5	–1
→	15			
15 AVAIL	16	–1	11	–1
	17			
	18	–1	12	–1
	19			
	20	2	6	16

Figure: Linked representation of binary tree T

Traversing in the Binary Tree

Tree traversal is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner. If search result in a visit to all the vertices, it is called a traversal. There are basically three traversal techniques for a binary tree that are,

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

1) Preorder traversal

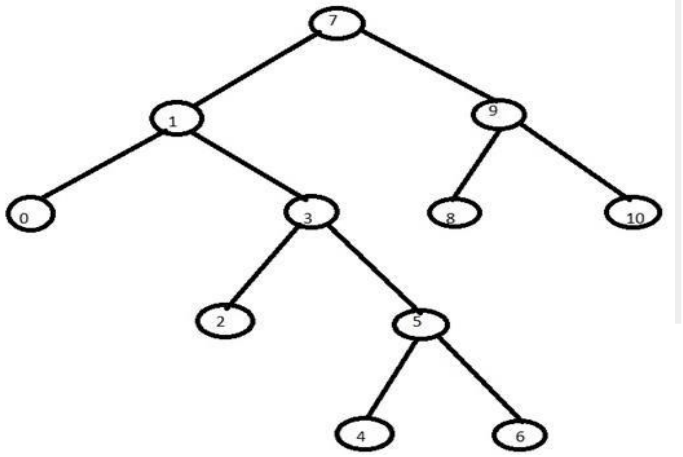
To **traverse a binary tree in preorder**, following operations are carried out:

1. Visit the root.
2. Traverse the left sub tree of root.
3. Traverse the right sub tree of root.

Note: Preorder traversal is also known as NLR traversal.

Algorithm:

```
preorder(t)
{
        If t! =0 then
        {
                Visit(t);
                Preorder(t->lchild);
                Preorder(t->rchild);
        }
}
```



Therefore, the preorder traversal of tree will be: **7,1,0,3,2,5,4,6,9,8,10**

2) Inorder traversal

To traverse a binary tree in inorder traversal, following operations are carried out:

1. Traverse the left most sub tree.
2. Visit the root.
3. Traverse the right most sub tree.

Note: Inorder traversal is also known as LNR traversal.

Algorithm:

```
inorder(t)
{
        If t! =0 then
        {
                Inorder(t->lchild);
                Visit(t);
                Inorder(t->rchild);
        }
}
```

Therefore the inorder traversal of tree will be: **0,1,2,3,4,5,6,7,8,9,10**

3) Postorder traversal

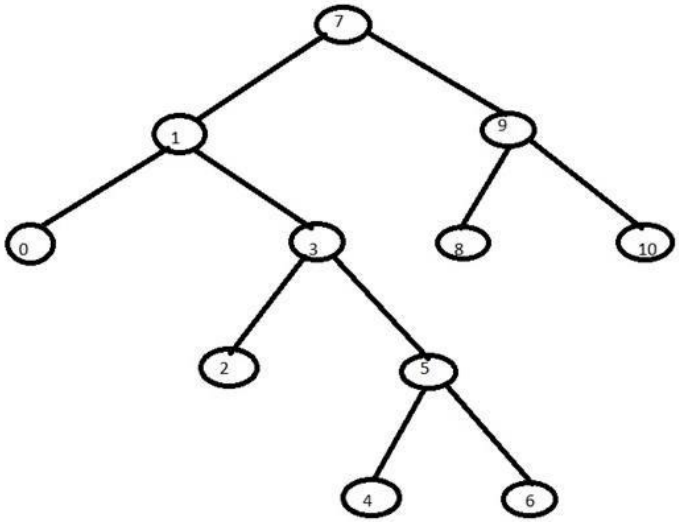
To traverse a binary tree in postorder traversal, following operations are carried out:

1. Traverse the left sub tree of root.
2. Traverse the right sub tree of root.
3. Visit the root.

Note: Postorder traversal is also known as LRN traversal

Algorithm:

```
postorder(t)
{
        If t! =0 then
        {
                Postorder(t->lchild);
                Postorder(t->rchild);
                Visit(t);
        }
}
```



Example: Let us consider a given binary tree.

Therefore the postorder traversal of tree will be: **0,2,4,6,5,3,1,8,10,9,7**

Threads:

A. J. Perlis and C. Thornton have proposed new binary tree called "Threaded Binary Tree", which makes use of NULL pointers to improve its traversal process. In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called as threads.

Threaded Binary Trees.

A binary tree can be represented using array representation or linked list representation. When a binary tree is represented using linked list representation, the reference part of the node which doesn't have a child is filled with a NULL pointer. In any binary tree linked list representation, there is a number of NULL pointers than actual pointers.

In the linked representation, a number of nodes contain a NULL pointer, either in their left or right fields or in both. This space that is wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information. For example, the NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node. These special

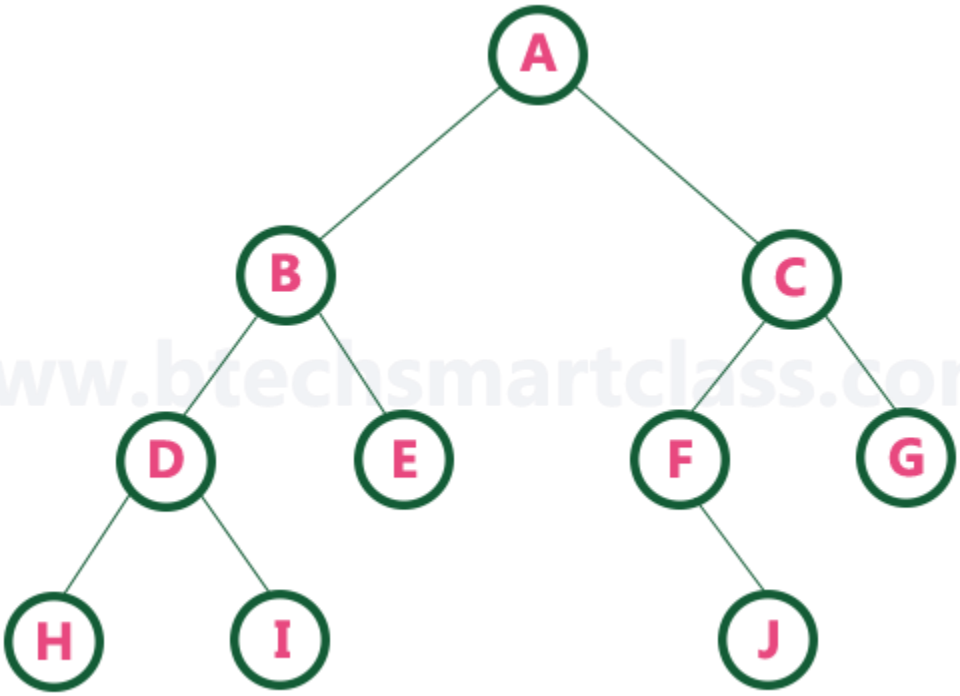
pointers are called **threads** and binary trees containing threads are called **threaded trees.** In the linked representation of a threaded binary tree, threads will be denoted using arrows. There are many ways of threading a binary tree and each type may vary according to the way the tree is traversed. In this book, we will discuss in-order traversal of the tree. Apart from this, a threaded binary tree may correspond to one-way threading or a two way threading.

1. one-way threading, a thread will appear either in the right field or the left field of the node.

A one-way threaded tree is also called a **single-threaded tree**. If the thread appears in the left field, then the left field will be made to point to the in-order predecessor of the node. Such a one- way threaded tree is called a **left-threaded binary tree.** On the contrary, if the thread appears in the right field, then it will point to the in-order successor of the node. Such a one-way threaded tree is called a **right threaded binary tree.**

1. two-way threaded tree, also called a double-threaded tree, threads will appear in both the left and the right field of the node. While the left field will point to the in-order predecessor of the node, the right field will point to its successor. A two-way threaded binary tree is also called a fully threaded binary tree. One-way threading and two-way threading of binary trees are explained below. Figure shows a binary tree without threading and its corresponding linked representation.

Consider the following binary tree...



To convert the above example binary tree into a threaded binary tree, first find the in-order traversal of that tree...

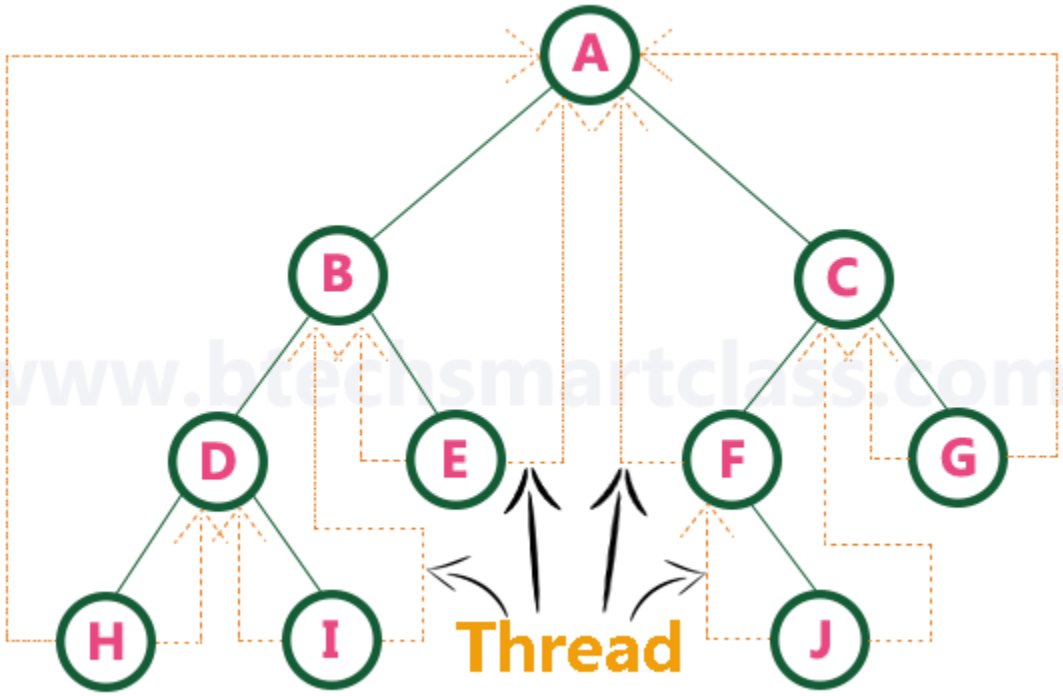
In-order traversal of above binary tree...

H - D - I - B - E - A - F - J - C - G

When we represent the above binary tree using linked list representation, nodes **H, I, E, F, J** and **G** left child pointers are NULL. This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it

points to the root node A. And nodes **H, I, E, J** and **G** right child pointers are NULL. These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

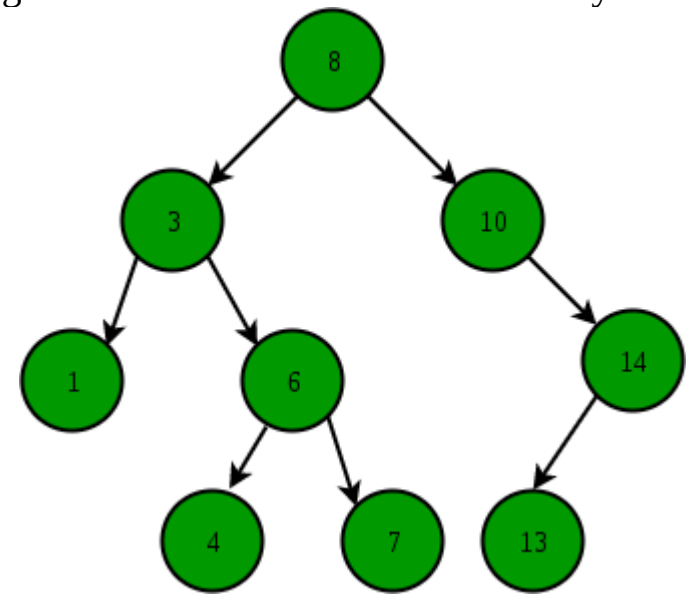
Above example binary tree is converted into threaded binary tree as follows.



In the above figure, threads are indicated with dotted links.

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the Root.
- The right subtree of a node contains only nodes with keys greater than the Root.
- The left and right subtree each must also be a binary search tree.



.

Example 10.2 Create a binary search tree using the following data elements:
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

Solution

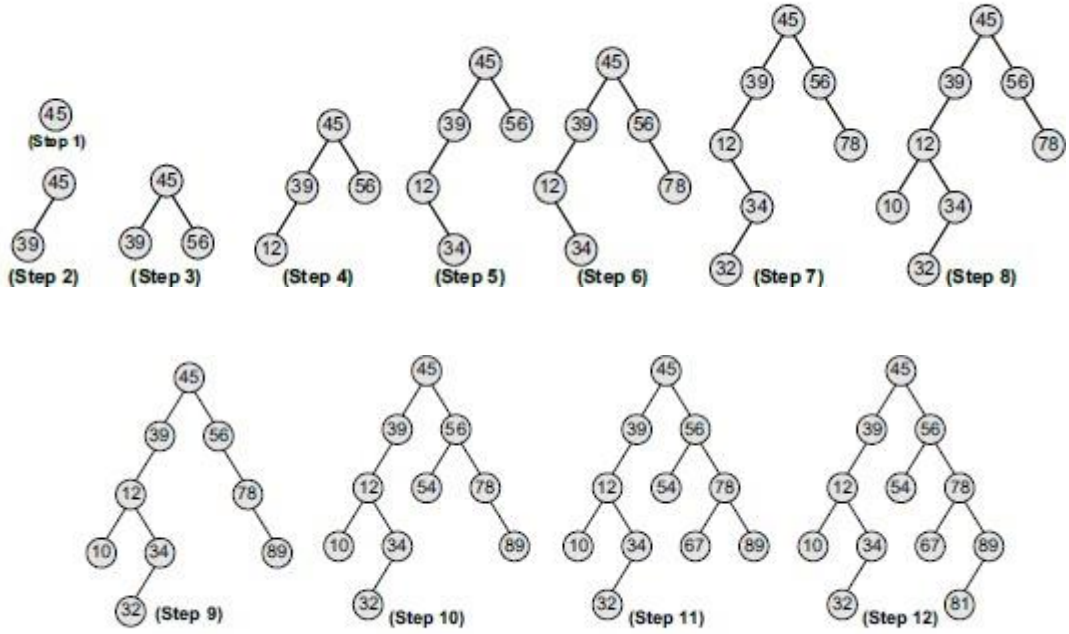


Figure 10.4 Binary search tree

OPERATIONS ON BINARY SEARCH TREES

1.Searching for a Node in a Binary Search Tree

The search function is used to find whether a given value is present in the tree or not. The searching process begins at the root node.

The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. So, the search algorithm terminates by displaying an appropriate message.

However, if there are nodes in the tree, then the search function checks to see if the key value of the current node is equal to the value to be searched. If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node. In case the value is greater than the value of the current node, it should be recursively called on the right child node.

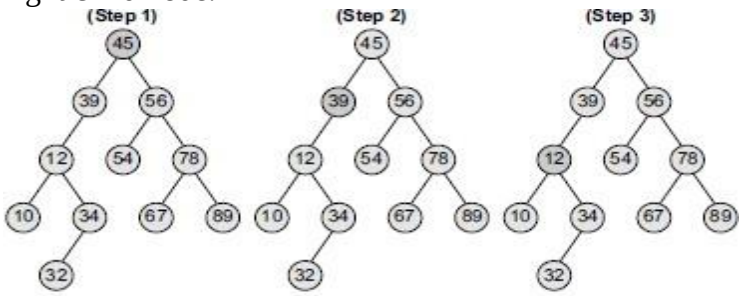


Figure 10.5 Searching a node with value 12 in the given binary search tree

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
           Return searchElement(TREE -> LEFT, VAL)
         ELSE
           Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

Figure 10.8 Algorithm to search for a given value in a binary search tree

Inserting a New Node in a Binary Search Tree

The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. The algorithm to insert a given value in a binary search tree. The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position. The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

In Step 1 of the algorithm, the insert function checks if the current node of TREE is NULL. If it is NULL, the algorithm simply adds the node, else it looks at the current node's value and then recurs down the left or right sub-tree. If the current node's value is less than that of the new node, then the right sub-tree is traversed, else the left sub-tree is traversed. The insert function continues moving down the levels of a binary tree until it reaches a leaf node. The new node is

added by following the rules of the binary search trees. That is, if the new node's value is greater than that of the parent node, the new node is inserted in the right sub-tree, else it is inserted in the left sub-tree. The insert function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and O(n) time in the worst case.

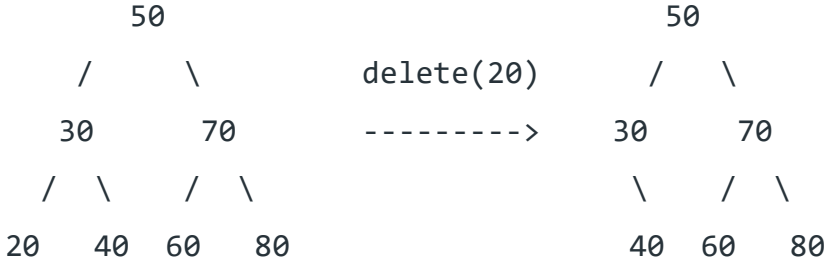
```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

Figure 10.9 Algorithm to insert a given value in a binary search tree

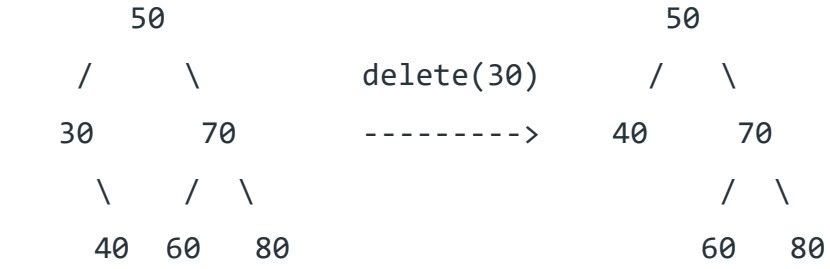
Deleting a Node from a Binary Search Tree

The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process. We will take up three cases in this section and discuss how a node is deleted from a binary search tree.

1) ***Deleting a Node that has No Children:*** Simply remove from the tree.



2) ***Deleting a Node that has One Children:*** Copy the child to the node and delete the child



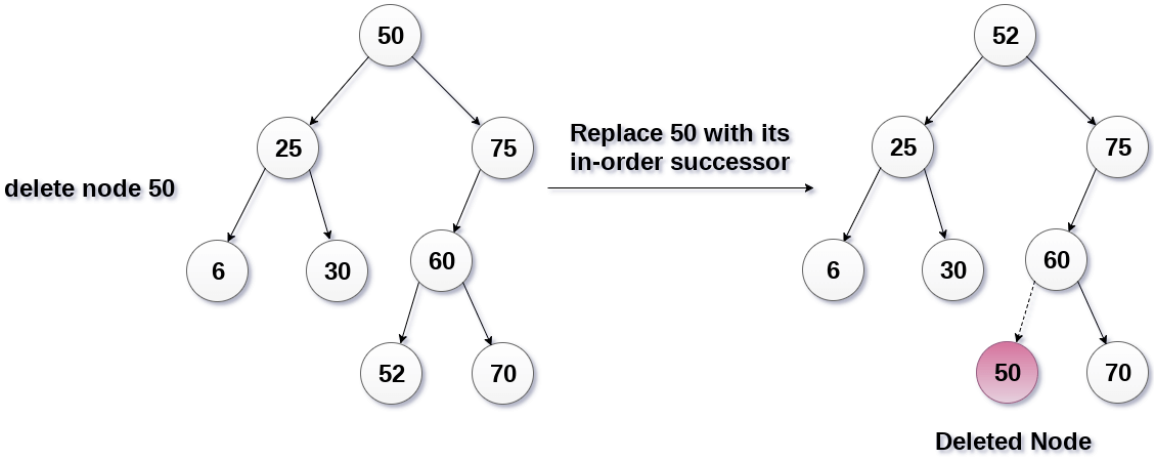
3) ***Deleting a Node that has two Children***

It is a bit complexed 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 image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

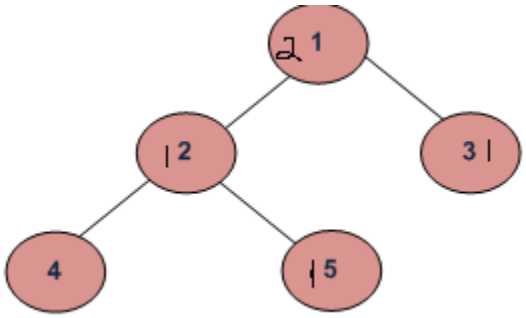
6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Maximum Depth or Height of a Tree

Given a binary tree, find height of it. Height of empty tree is 0 and height of below tree is 2.



Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two . See below pseudo code and program for details.

Algorithm:

```
 maxDepth()
1. If tree is empty then return 0
2. Else
     (a) Get the max depth of left subtree recursively  i.e.,
          call maxDepth( tree->left-subtree)
     (a) Get the max depth of right subtree recursively  i.e.,
          call maxDepth( tree->right-subtree)
     (c) Get the max of max depths of left and right
          subtrees.
         max_depth = max(max dept of left subtree,
                             max depth of right subtree)

     (d) Return max_depth
```

M-Way Search Trees:

In a binary search tree contains one value and two pointers, *left* and *right*, which point to the node's left and right sub-trees, respectively.

The same concept is used in an *M-way* search tree .The **m-way** search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple elements. **In an m-Way tree of order** m, **each node contains a maximum of** m – 1 **Key elements and m children(Pointers).**

Pointer to left sub-tree	Value or Key of the node	Pointer to right sub-tree

Figure 11.1 Structure of a binary search tree node

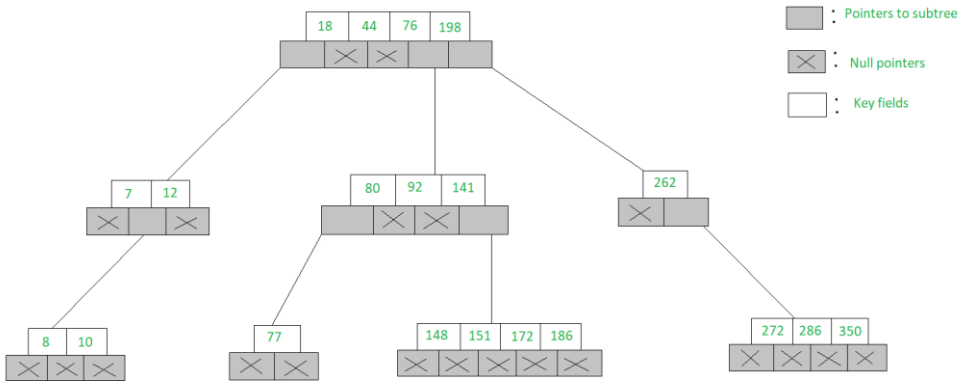
The structure of an M-way search tree node is shown in Fig. 11.2.

P_0	K_0	P_1	K_1	P_2	K_2	.....	P_{n-1}	K_{n-1}	P_n

Figure 11.2 Structure of an M-way search tree node

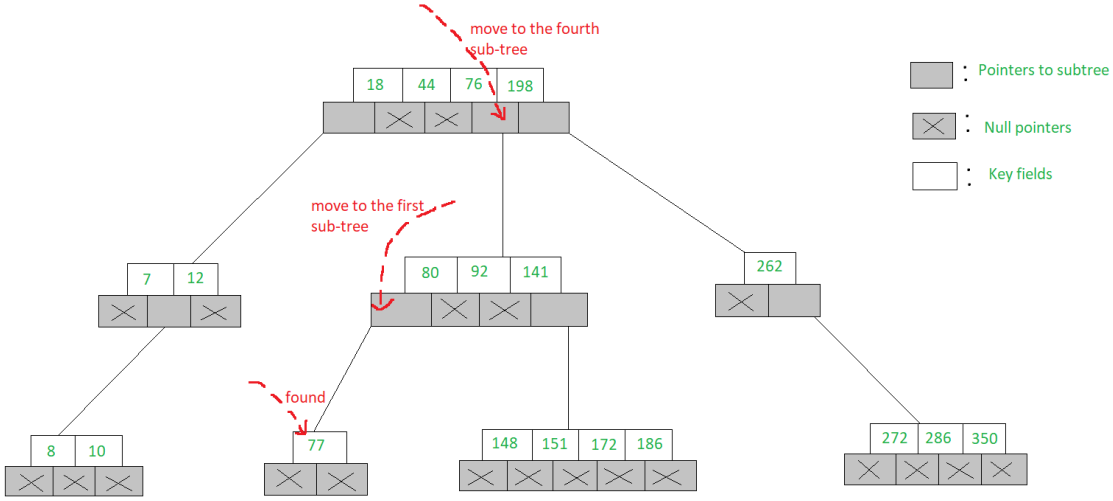
In the structure shown, P0, P1, P2, ..., Pn are pointers to the node's sub-trees and K0, K1, K2, ..., Kn–1 are the key values of the node. All the key values are stored in ascending order. That is,
Ki < Ki+1 for 0<= i <= n–2.

An example of a 5-Way search tree is shown in the figure below. Observe how each node has at most 5 child nodes & therefore has at most 4 keys contained in it.



Searching in an m-Way search tree:

- Searching for a key in an m-Way search tree is similar to that of binary search tree
- To search for 77 in the 5-Way search tree, shown in the figure, we begin at the root & as 77> 76> 44> 18, move to the fourth sub-tree
- In the root node of the fourth sub-tree, 77< 80 & therefore we move to the first sub-tree of the node. Since 77 is available in the only node of this sub-tree, we claim 77 was successfully searched

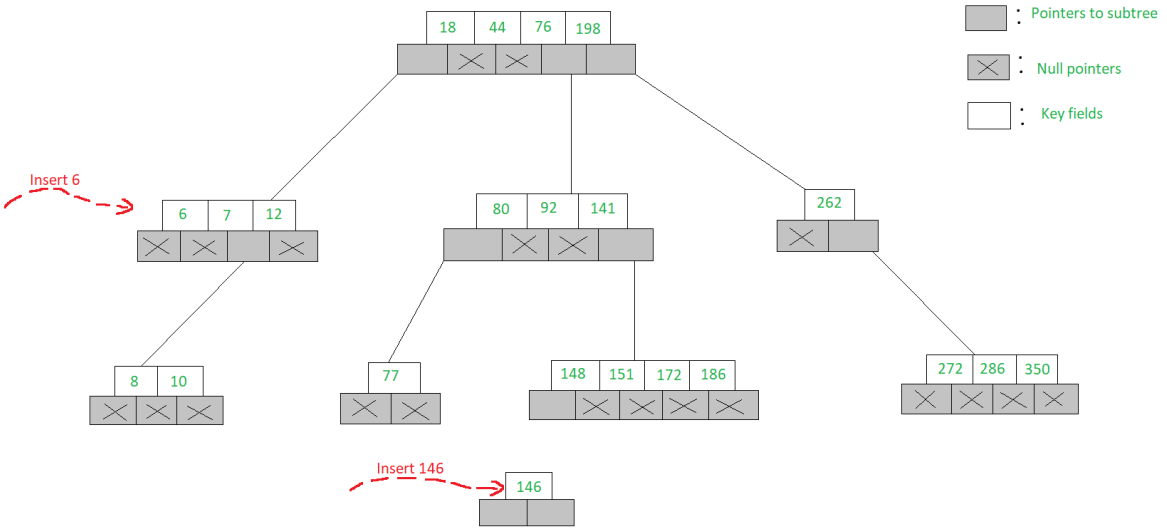


Insertion in an m-Way search tree:

The insertion in an m-Way search tree is similar to binary trees but there should be no more than **m-1** elements in a node. If the node is full then a child node will be created to insert the further elements.

Example:

- To insert a new element into an m-Way search tree we proceed in the same way as one would in order to search for the element
- To insert 6 into the 5-Way search tree shown in the figure, we proceed to search for 6 and find that we fall off the tree at the node [7, 12] with the first child node showing a null pointer
- Since the node has only two keys and a 5-Way search tree can accommodate up to 4 keys in a node, 6 is inserted into the node like [6, 7, 12]
- But to insert 146, the node [148, 151, 172, 186] is already full, hence we open a new child node and insert 146 into it. Both these insertions have been illustrated below



B- TREES

A B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access. A B tree of order m can have a maximum of m–1 keys and m pointers to its sub-trees. A B tree may contain a large number of key values and pointers to sub- trees. Storing a large number of keys in a single node keeps the height of the tree relatively small.

A B tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic amortized time. A B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree. In addition it has the following properties:

1. Every node in the B tree has at most (maximum) m children.
2. Every node in the B tree has at most (maximum) m-1 key elements.
3. Every internal node (except the root node and leaf nodes) in the B tree has at least (minimum) m/2 children.
4. The minimum number of keys in root is 1and other nodes has the minimum number of keys m/2-1.
5. The root node has at least two children.
6. All leaf nodes are at the same level.

An internal node in the B tree can have n number of children, where 0 <= n <= m. It is not necessary that every node has the same number of children, but the only restriction is that the node should hav

at least m/2 children. As B tree of order 4 is given in Fig. 11.4.

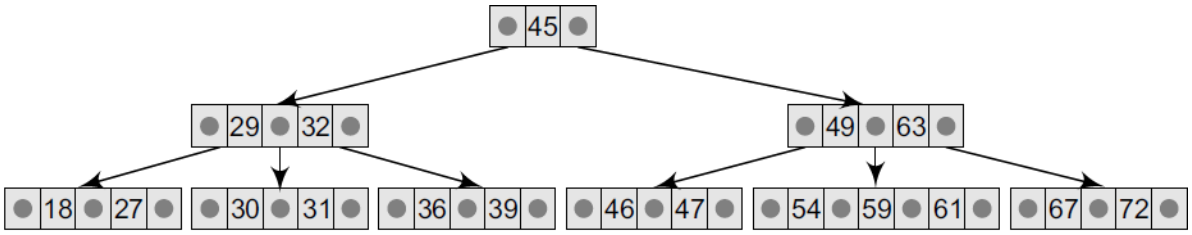


Figure 11.4 B tree of order 4

While performing insertion and deletion operations in a B tree, the number of child nodes may change. So, in order to maintain a minimum number of children, the internal nodes may be joined or split. We will discuss search, insertion, and deletion operations in this section.

Searching for an Element in a B Tree

Searching for an element in a B tree is similar to that in binary search trees. Consider the B tree given in Fig. 11.4. To search for 59, we begin at the root node. The root node has a value 45 which is less than 59. So, we traverse in the right sub-tree. The right sub-tree of the root node has two key values, 49 and 63. Since 49 < 59 < 63, we traverse the right sub-tree of 49, that is, the left sub-tree of 63. This sub-tree has three values, 54, 59, and 61. On finding the value 59, the search is successful. Take another example. If you want to search for 9, then we traverse the left sub-tree of the root node. The left sub-tree has two key values, 29 and 32. Again, we traverse the left sub-tree of 29. We find that it has two key values, 18 and 27. There is no left sub-tree of 18, hence the value 9 is not stored in the tree. Since the running time of the search operation depends upon the height of the tree, the algorithm to search for an element in a B tree takes O(logt n) time to execute.

Inserting a New Element in a B Tree

In a B tree, all insertions are done at the leaf node level. A new value is inserted in the B tree using the algorithm given below.

1.check whether tree is empty or not

2. if B tree is empty then create a new node and inserting the key value into that node. Now it is the root node.

3. if B tree is not empty ,then find leaf node where the new key value should be inserted.

4.If the leaf node is not full, that is, it contains less than m–1 key values, then insert the new element in the node keeping the node's elements ordered.

5. If the leaf node is full, that is, the leaf node already contains m–1 key values, then

(a) split the node at its median into two nodes ,by sending the middle key value to its parent node.

(b) If the parent's node is already full, then split the parent node by following the same steps.

Example 11.1 Look at the B tree of order 5 given below and insert 8, 9, 39, and 4 into it.

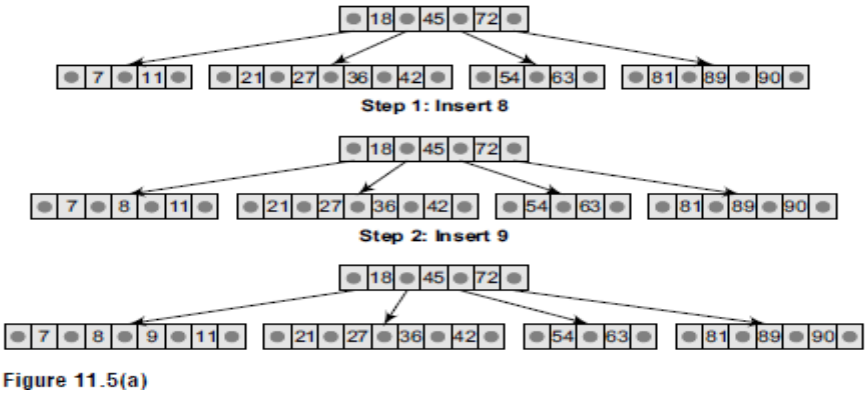


Figure 11.5(a)

Till now, we have easily inserted 8 and 9 in the tree because the leaf nodes were not full. But now, the node in which 39 should be inserted is already full as it contains four values. Here we split the nodes to form two separate nodes. But before splitting, arrange the key values in order (including the new value). The ordered set of values is given as 21, 27, 36, 39, and 42. The median value is 36, so push 36 into its parent's node and split the leaf nodes.

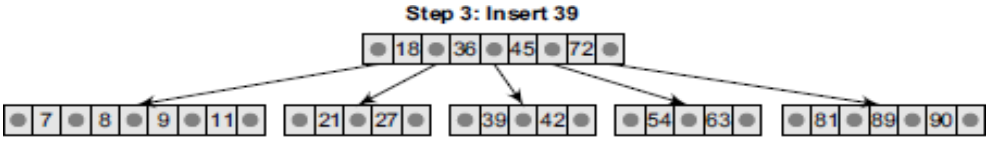


Figure 11.5(b)

Now the node in which 4 should be inserted is already full as it contains four key values. Here we split the nodes to form two separate nodes. But before splitting, we arrange the key values in order (including the new value). The ordered set of values is given as 4, 7, 8, 9, and 11. The median value is 8, so we push 8 into its parent's node and split the leaf nodes. But again, we see that the parent's node is already full, so we split the parent node using the same procedure.

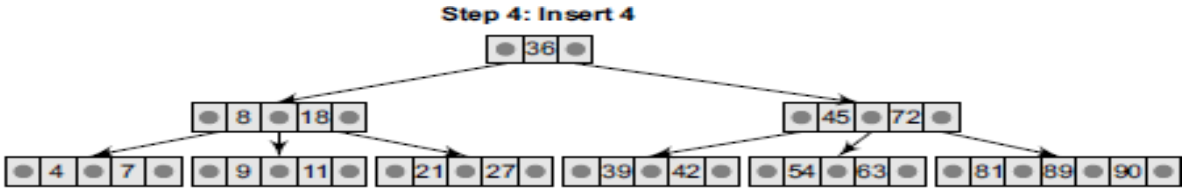


Figure 11.5(c) B tree

Deleting an Element from a B Tree

Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. In the first case, a leaf node has to be deleted. In the second case, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

CASE 1: **a leaf node has to be deleted**:

1. The leaf node consists more than the minimum number of key elements simply we have delete the target key element.
2. The leaf node contains only minimum number of keys ,then following steps we have followed
   a) Borrow from immediate left node or siblings through the parent
      (Or)
   b) Borrow from immediate Right node or siblings through the parent
      (Or)
   c) If immediate left or right nodes or siblings have only minimum number of keys, then merging operation is done with left sibling through the parent.

CASE 2: **an internal node has to be deleted:** if the target key is in internal node, then follow the 3 steps

1. Target key is replaced with in order predecessor of that key
2. Target key is replaced with in successor predecessor of that key
3. Merge operation is done with left sibling or right sibling through the parent

Example 11.2 Consider the following B tree of order 5 and delete values 93, 201, 180, and 72 from it (Fig. 11.6(a)).

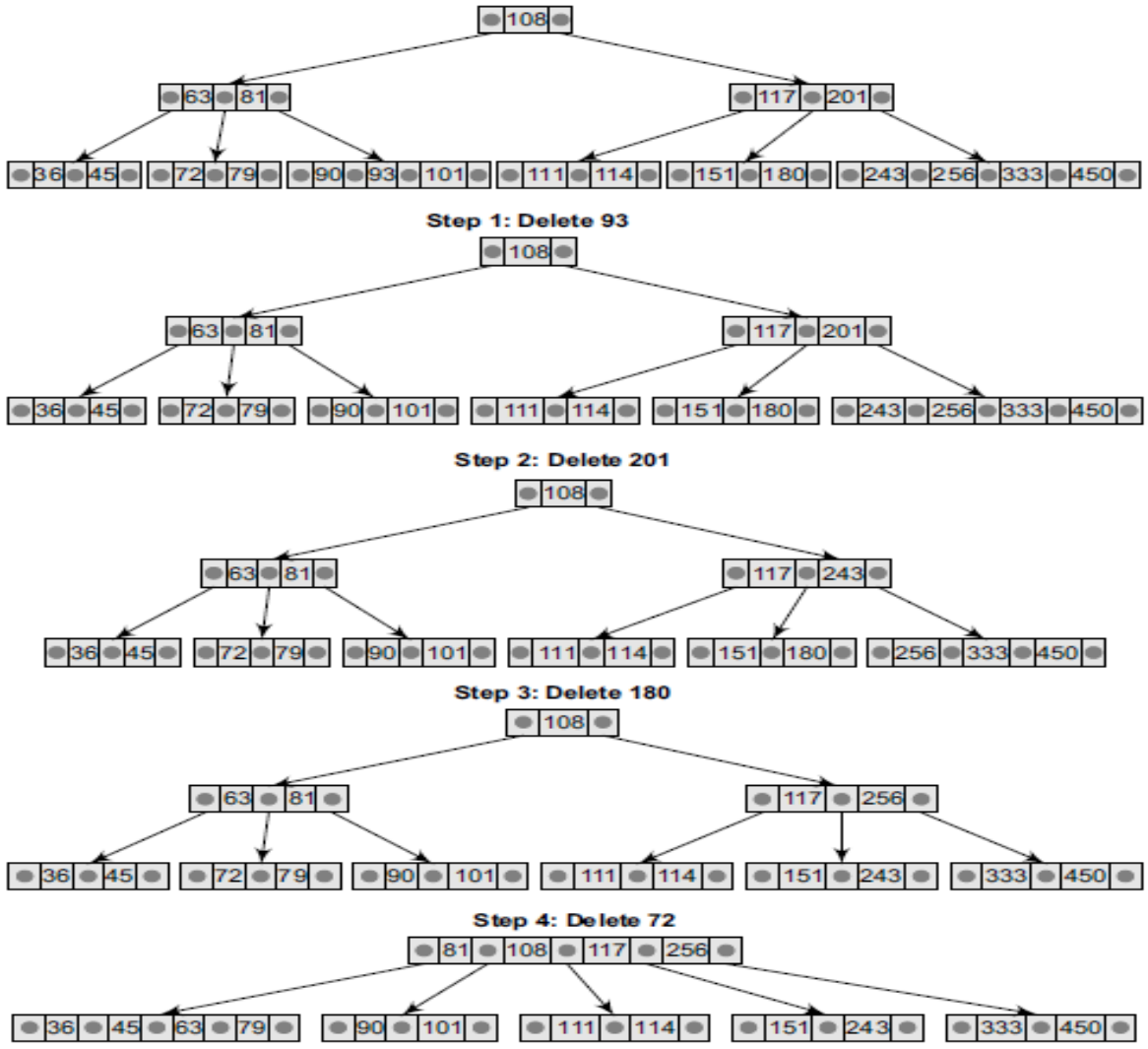


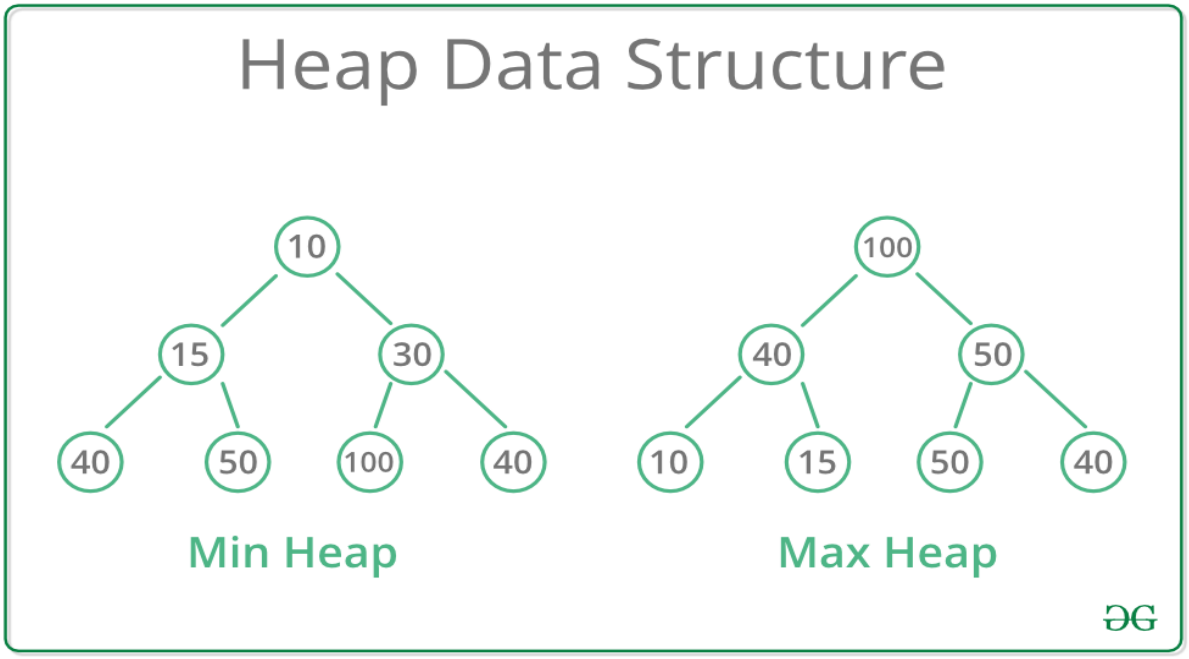
Figure 11.6 B tree

Example 11.3 Consider the B tree of order 3 given below and perform the following operations: (a) insert 121, 87 and then (b) delete 36, 109.

HEAP TREE:

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greater than are equal to of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap**: In a Min-Heap the key present at the root node must be less than are equal to of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

```
Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.
```

Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Insertion in the Heap tree

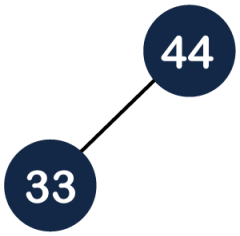
44, 33, 77, 11, 55, 88, 66

Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

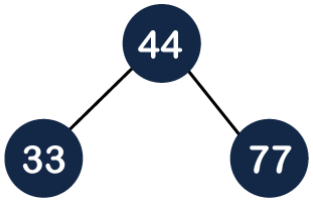
- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

Step 1: First we add the 44 element in the tree as shown below:

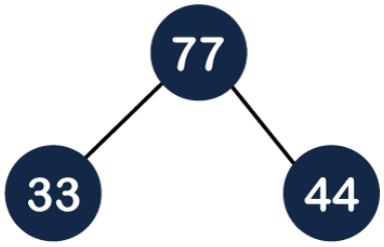
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:



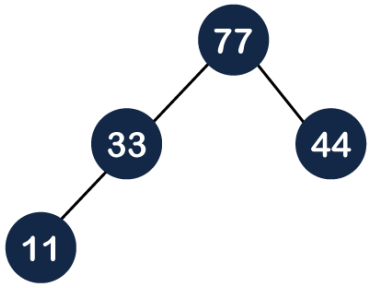
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



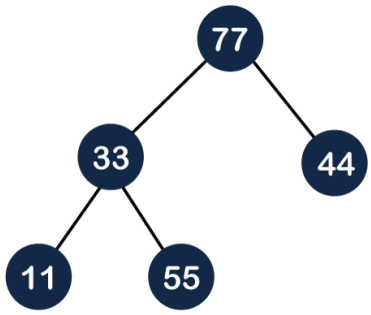
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



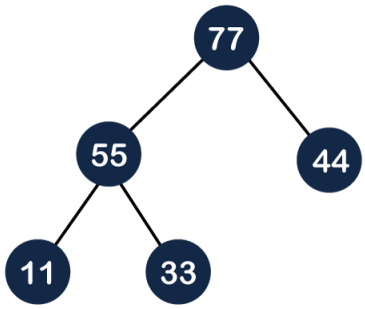
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



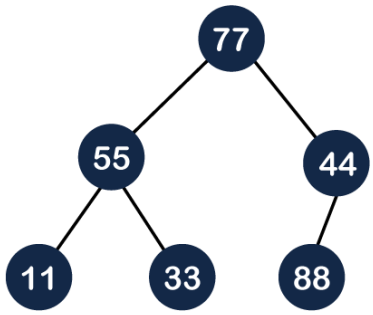
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:

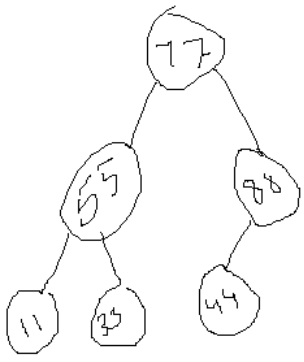


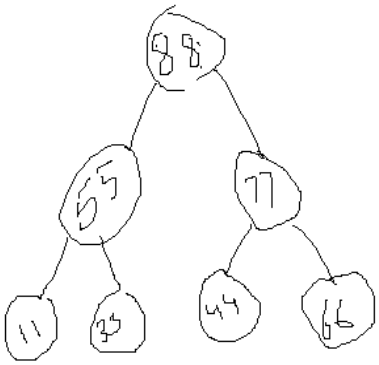
As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown below:



Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:







As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:

Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:

Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:

In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.

Deletion in Heap Tree

In Deletion in the heap tree, the root node is always deleted and it is replaced with the last element.

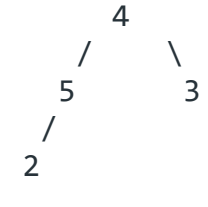
Let's understand the deletion through an example.

Suppose the Heap is a Max-Heap as:

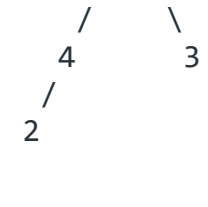
The element to be deleted is root, i.e. 10.

Process:
The last element is 4.

Step 1: Replace the last element with root, and delete it.



Step 2: Heapify root.
Final Heap:



BALANCED BINARY TREE:

A **balanced** binary tree is a binary tree structure in which the left and right subtrees of every node differ in height by no more than 1.

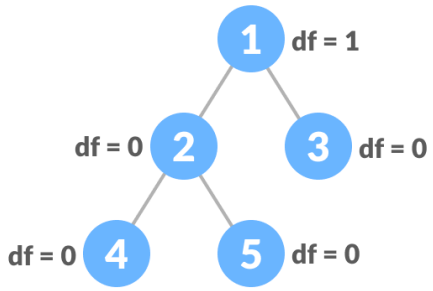
A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

To learn more about the height of a tree/node, visit Tree Data Structure. Following are the conditions for a height-balanced binary tree:

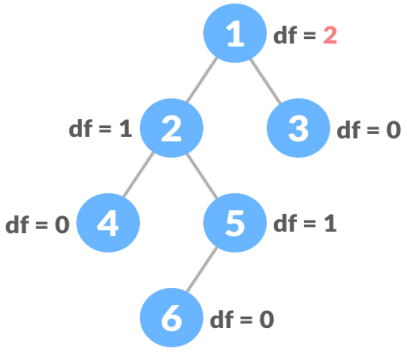
1. difference between the left and the right subtree for any node is not more than one

difference factor = Height (left sub-tree) – Height (right sub-tree)

2. the left subtree is balanced
3. the right subtree is balanced



Balanced Binary Tree with height at each level



df = |height of left child - height of right child|

Unbalanced Binary Tree with height at each level

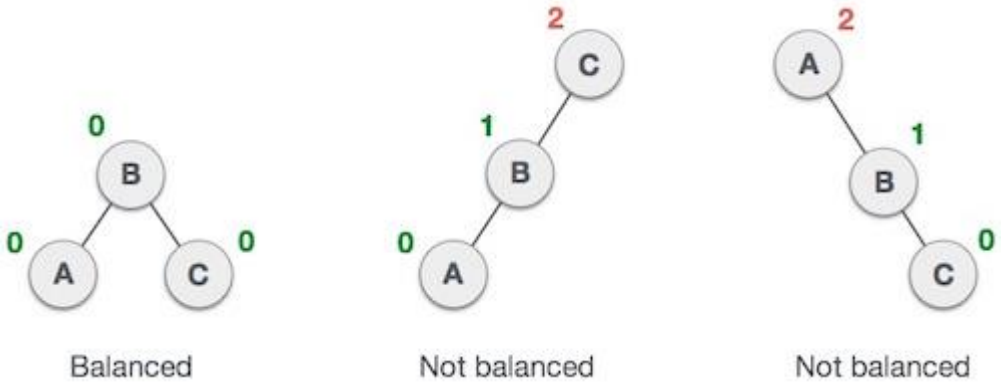
AVL TREES:

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree..

The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the BalanceFactor. Thus, every node has a balance factor associated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree. A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

Balance factor = Height (left sub-tree) – Height (right sub-tree)

Here we see that the first tree is balanced and the next two trees are not balanced −



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

```
BalanceFactor = height(left-sutree) – height(right-sutree)
```

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

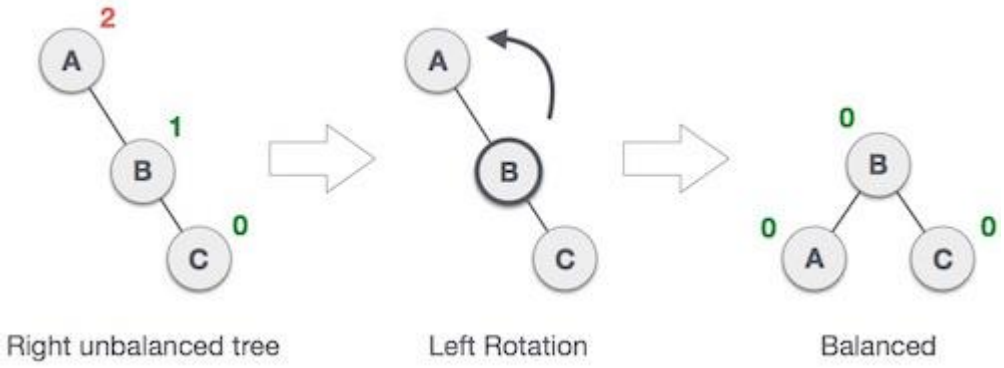
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

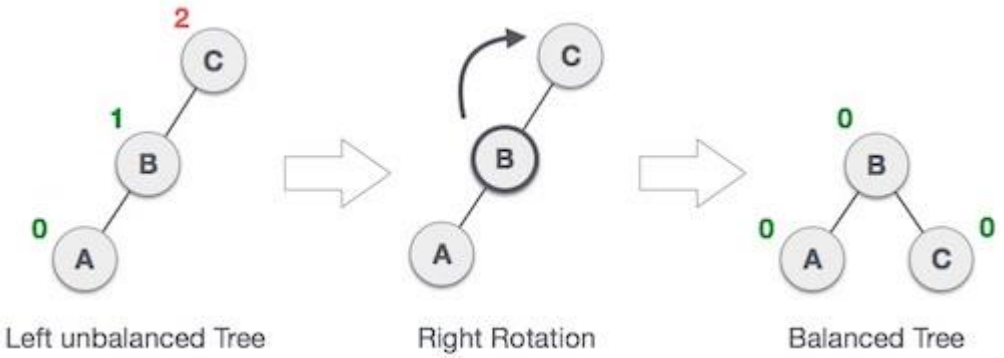
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

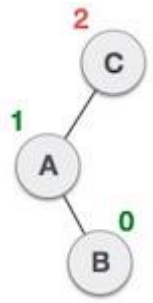
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

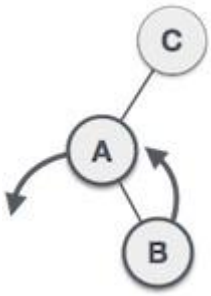
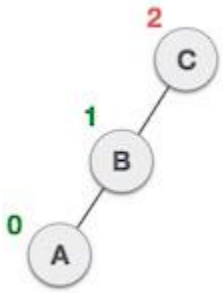
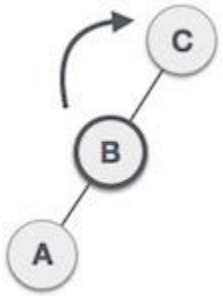
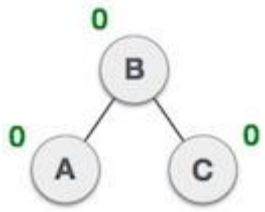


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

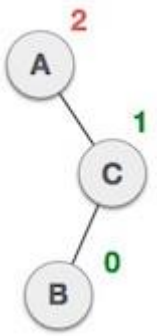
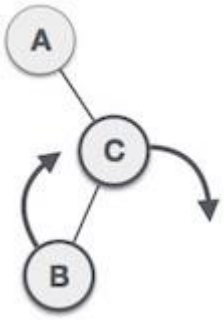
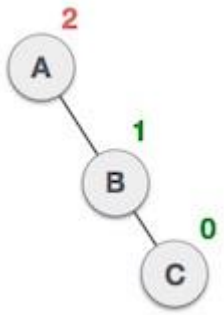
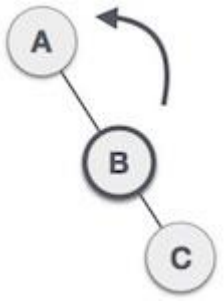
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

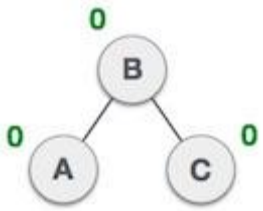
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action

	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requ rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.

	The tree is now balanced.