TREE

r more nodes such that

ed the root node R.

ivided into n ≥ 0 disjoint sets T1, T2,.,.,. TN, where

T1, T2…., T n is called the sub tree of the root

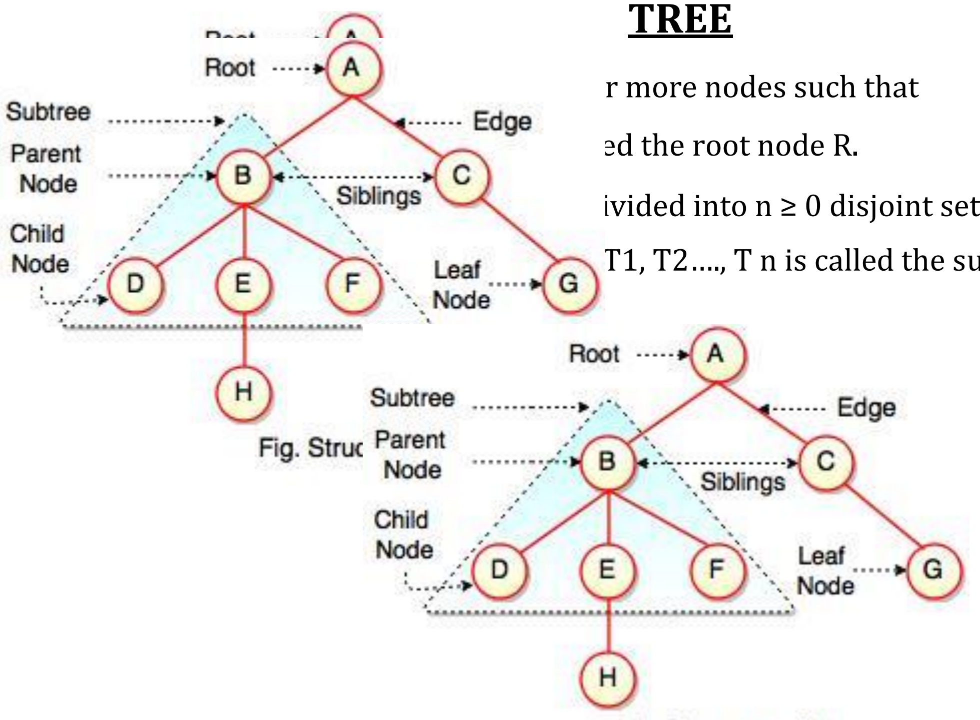


Fig. Structure of Tree

Forest: A forest is a set of n ≥ 0 disjoint trees.

Adjacent node: Any two nodes in the graph which are connected by an edge in the graph are called adjacent nodes.

Root Node: In a directed tree a node which has in degree 0 is called the root node.

Leaf Node: In a directed tree a node which has out degree 0 is called the leaf node or terminal node.

Directed Tree: A directed tree is an acyclic digraph which has one node called the root node with in degree 0 while all other nodes have in degree 1.

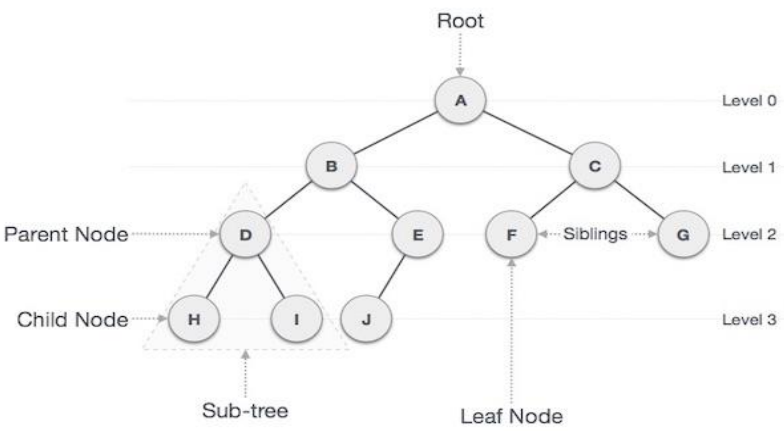
m-ary Tree: If in a directed tree the out degree of every node is less than or equal to m then the tree is called the m – ary tree.

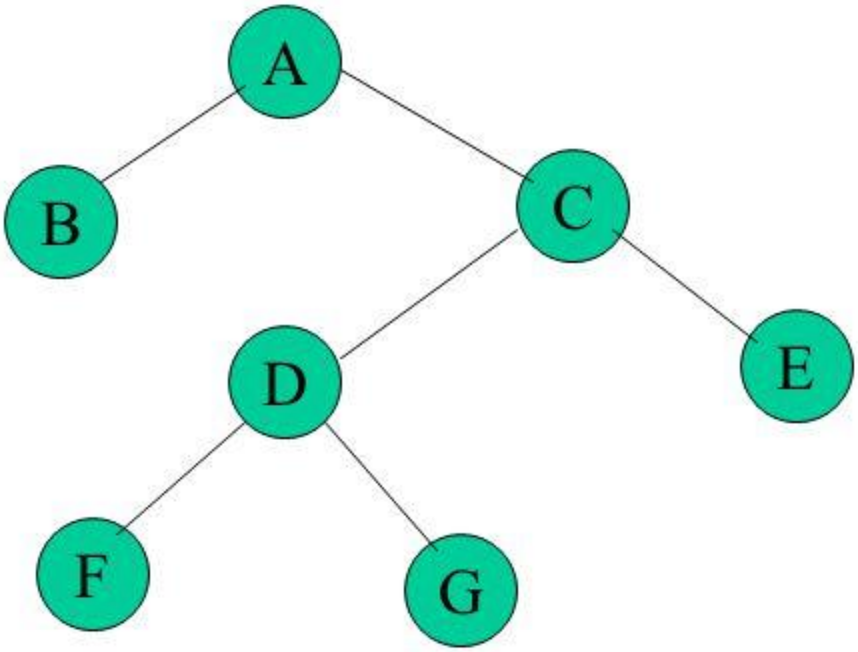
Complete m - ary Tree: if the out degree of every node is exactly equal to m or 0 and the number of nodes at level i is m_{i-1} then the tree is called full or complete m – ary tree.

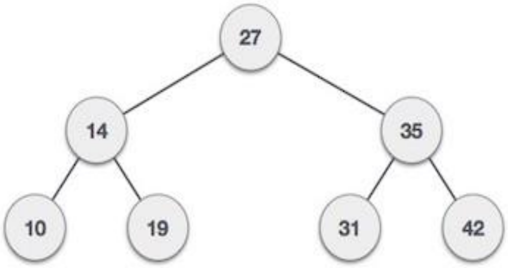
Binary Tree: If in a directed tree the out degree of every node is less than or equal to 2 then the tree is called the binary tree.

Complete Binary Tree: if the out degree of every node is exactly equal to 2 or 0 and the number of nodes at level i is 2_{i-1} then the tree is called full or complete binary tree.

Strictl
If ever has nonempty
right sı tly binary tree.
A stric vays contains 2

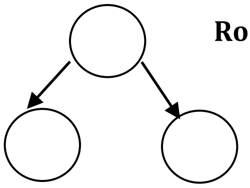






Difference between root node and leaf node.

- Root node has the in degree 0 while the leaf node has the out degree 0



Comparison of Tree and Linked list

Linked List	Tree
(1) Linked list is an example of linier data structure.	**(1)** Tree is an example of non linier data structure.
(2) Searching time is more required in linked list. Because we must have to traverse each node sequentially in linked list even if the list is sorted.	**(2)** Searching time is less required in tree because we can search an element using binary search method.

Operations on Binary Search Tree:

- Following operations can be performed on Binary Search tree:
  1. Traversal
  2. Insertion
  3. Deletion
  4. Search
  5. Copy

Traversal operation of Binary Search Tree

Traversal:

- Traversal is the method of processing every nodes in the tree exactly once in a systematic manner.

Types of Traversal: There are three different types of tree traversal.

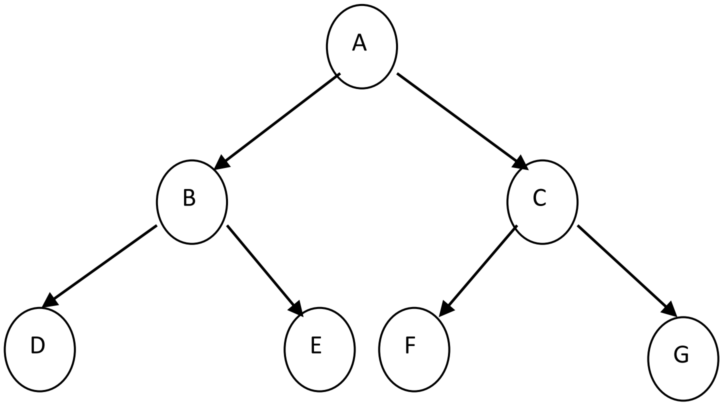
1. Preorder Traversal
2. In order Traversal
3. Post order Traversal

(1) Preorder Traversal:

The preorder traversal follows three steps:

1. Process the root node first.(***V***ertices or Node)
2. Traverse the left sub tree in preorder.(***L***eft node)
3. Traverse the right sub tree in preorder.(***R***ight node)

Example:



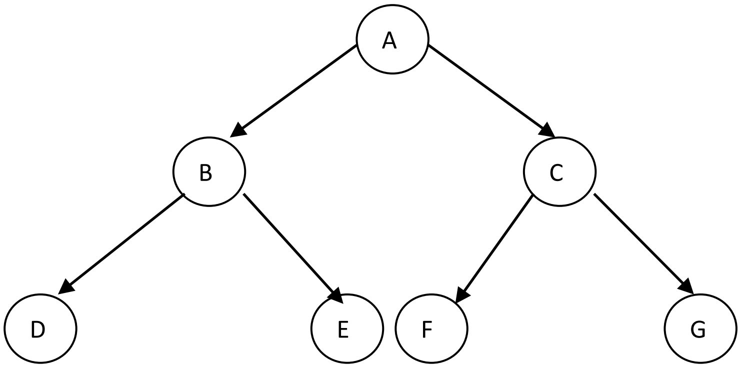
Preorder Traversal: A B D E C F G

(2) In order Traversal:

The In order traversal follows the three steps:

1. Traverse the left sub tree in inorder(***L***eft node).
2. Process the root node. (***V***ertices or Node)
3. Traverse the right sub tree in inorder.(***R***ight node)

Example:

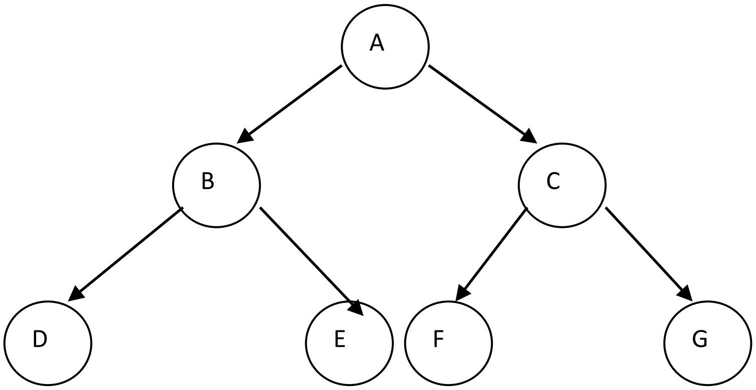


InOrder Traversal: D B E A F C G

(3) Post order Traversal:

- The Post order traversal follows the three steps:
  1. Traverse the left sub tree in Post order. (***L***eft node)
  2. Traverse the right sub tree in Post order. (***R***ight node)
  3. Process the root node. (***V***ertices or Node)

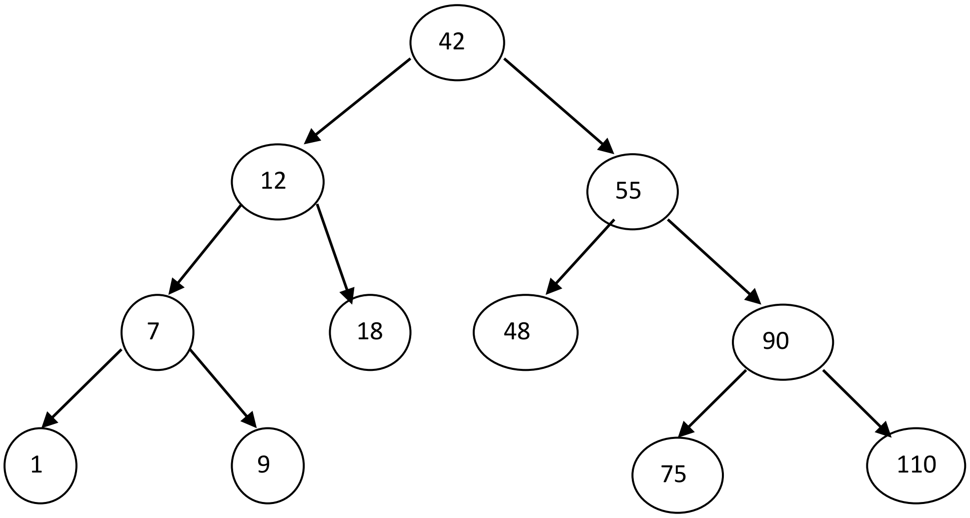
Example:



Post Order Traversal: D E B F G C A

<u>Examples of tree traversal</u>

Ex-1: Travers the following Tree in Pre Order, In Order, Post Order.



Pre Order: 42 12 7 1 9 18 55 48 90 75 110

In Order: 1 7 9 12 18 42 48 55 75 90 110

Post Order: 1 9 7 18 12 48 75 110 90 55 42

Algorithms for Binary Tree Traversal

(1) Pre Order Traversal:

PREORDER(T)

- This function traverses the tree in pre order.
- T is a pointer which points to the root node of tree.

```
1. [Process the root node]
   If T ≠ NULL then
        Write (DATA(T))
   Else
        Write "Empty Tree"
2. [Process the left sub tree]
   If LPTR (T) ≠ NULL then
        Call PREORDER (LPTR (T))
3. [Process the right sub tree]
   If RPTR (T) ≠ NULL then
        Call PREORDER (RPTR (T))
4. [Finished]
   Return
```

(2) In Order Traversal:

INORDER(T)

- This function traverses the tree in inorder.
- T is a pointer which points to the root node of tree.

```
1. [Check for empty tree]
   If T = NULL then
        Write "Empty Tree"
2. [Process the left sub tree]
   If LPTR (T) ≠ NULL then
        Call INORDER (LPTR (T))
3. [Process the root node]
   Write (DATA (T))
4. [Process the right sub tree]
   If RPTR (T) ≠ NULL then
        Call INORDER (RPTR (T))
5. [Finished]
   Return
```

(3) Post Order Traversal:

POSTORDER(T)

- This function traverses the tree in post order.
- T is a pointer which points to the root node of tree.

```
1. [Check for empty tree]
    If T = NULL then
        Write “Empty Tree”
2. [Process the left sub tree]
   If LPTR (T) ≠ NULL then
        Call POSTORDER (LPTR (T))
3. [Process the right sub tree]
   If RPTR (T) ≠ NULL then
        Call POSTORDER (RPTR (T))
4. [Process the root node]
    Write (DATA (T))
5. [Finished]
Return
```

Insertion Operation on Binary Search tree

Insertion Operation

- Binary search tree has following characteristics:

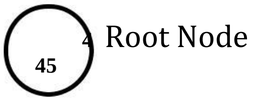
  (1) All the nodes to the left of root node have value less than the value of root node.

  (2) All the nodes to the right of root node have value greater than the value of root node.

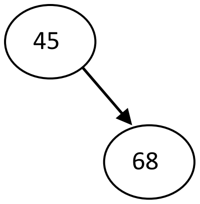
- Suppose we want to construct binary search tree for following set of data:

  45 68 35 42 15 64 78

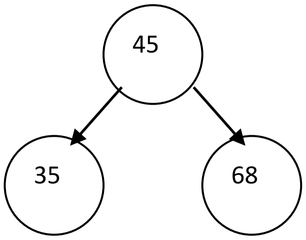
- **Step 1:** First element is 45 so it is inserted as a root node of the tree.



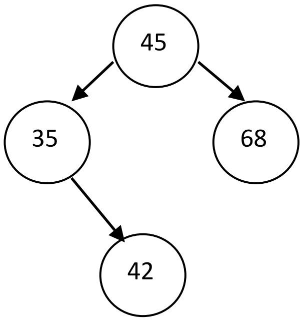
- **Step 2:** Now we have to insert 68. First we compare 68 with the root node which is 45. Since the value of 68 is greater then 45 so it is inserted to the right of the root node.
-



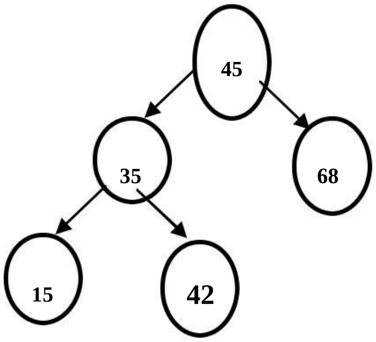
- **Step 3:** Now we have to insert 35. First we compare 35 with the root node which is 45. Since the value of 35 is less then 45 so it is inserted to the left of the root node.



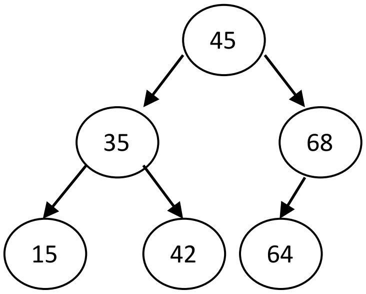
- **Step 4:** Now we have to insert 42. First we compare 42 with the root node which is 45. Since the value of 42 is less than 45 so it is inserted to the left of the root node. But the root node has already one left node 35. So now we compare 42 with 35. Since the value of 42 is greater than 35 we insert 42 to the right of node 35.



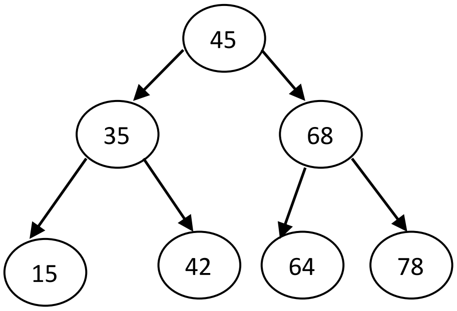
- **Step 5:** Now we have to insert 15. First we compare 15 with the root node which is 45. Since the value of 15 is less than 45 so it is inserted to the left of the root node. But the root node has already one left node 35. So now we compare 15 with 35. Since the value of 15 is less than 35 we insert 15 to the left of node 35.



- **Step 6:** Now we have to insert 64. First we compare 64 with the root node which is 45. Since the value of 64 is greater than 45 so it is inserted to the right of the root node. But the root node has already one right node 68. So now we compare 64 with 68. Since the value of 64 is less than 68 we insert 64 to the left of node 68.



- **Step 7:** Now we have to insert 78. First we compare 78 with the root node which is 45. Since the value of 78 is greater than 45 so it is inserted to the right of the root node. But the root node has already one right node 68. So now we compare 78 with 68. Since the value of 78 is greater than 68 we insert 78 to the right of node 68.



Algorithm for Insertion Operation on Binary Search tree

INSERT(Root, Info)

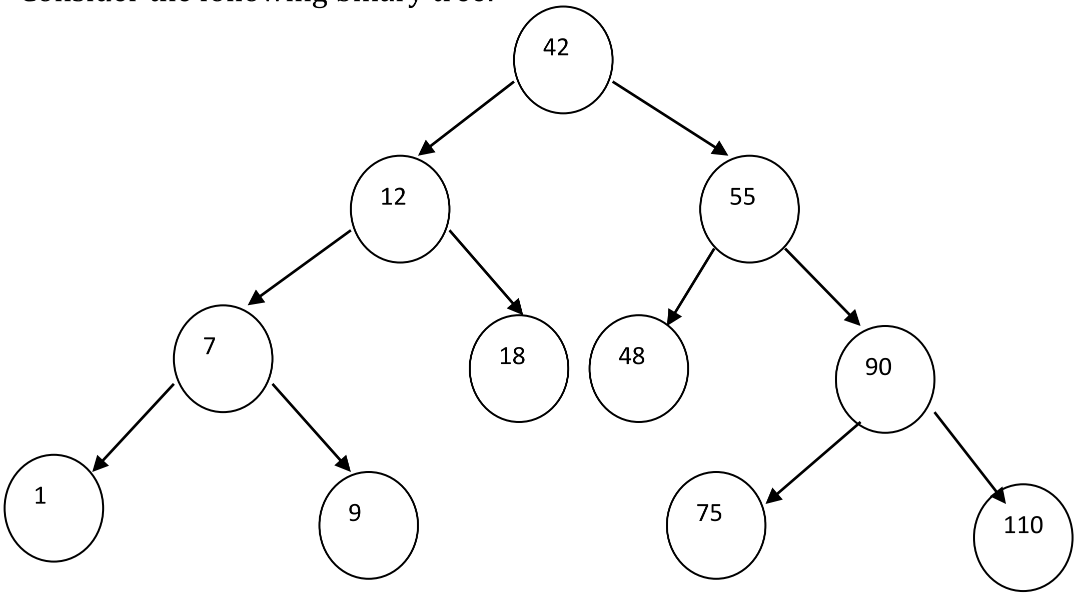
- Root is a pointer which points to the root node of tree.
- Info is an element which we want to insert

```
1. [Check for empty tree]
   If Root = NULL then
       DATA(Root) ← Info
       LPTR(Root) ← NULL
       RPTR(Root) ← NULL
2. [Initialize]
       CURRENT ← Root
3. [left sub tree]
   If Info < DATA(CURRENT) then
       If LPTR(CURRENT) ≠ NULL then
           CURRENT ← LPTR(CURRENT)
       Else
           DATA(Root ) ← Info
           LPTR(Root) = NULL
           RPTR(Root) = NULL
4. [right sub tree]
   If Info > DATA(CURRENT) then
       If RPTR(CURRENT) ≠ NULL then
           CURRENT ← RPTR(CURRENT)
       Else
           DATA(Root) ← Info
           LPTR(Root) ← NULL
           RPTR(Root) ← NULL
5. [Check for duplicate node]
   If DATA(CURRENT) = DATA(Root) then
       Write "Item already in tree"
6. [finished]
   Return
```

Deletion Operation on Binary Search tree

Delete Operation

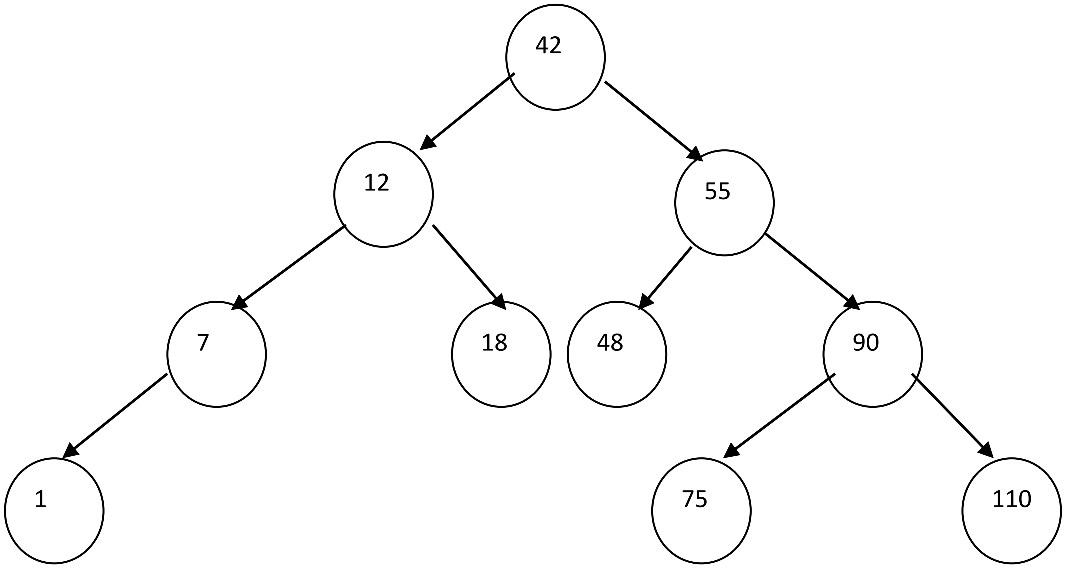
- Consider the following binary tree:



- There are three possibilities when we want to delete an element from binary search tree.

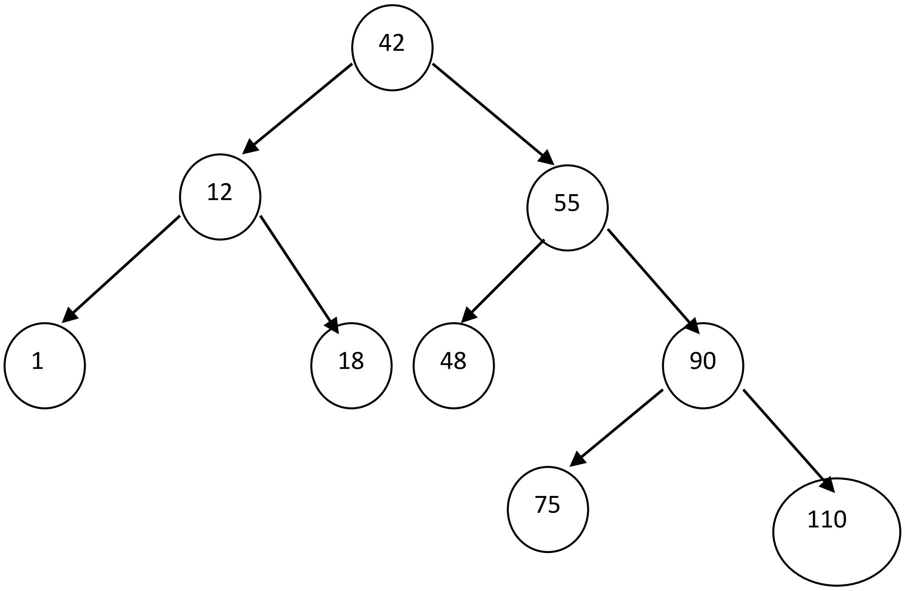
(1) If a node to be deleted has empty left and right sub tree then a node is deleted directly.

Suppose we want to delete node 9. Here node 9 has no left or right sub tree so we can delete it directly. Thus tree after deleting node 9 is as follow:



(2) If a node to be deleted has only one left sub tree or right sub tree then the sub tree of the deleted node is linked directly with the parent of the deleted node.

Suppose we want to delete node 7. Here node 7 has one left sub tree so we link this left sub tree with parent of node 7 which is 12. Now the node which we want to link with node 12 is 1. Thus tree after deleting node 48 is as follow:



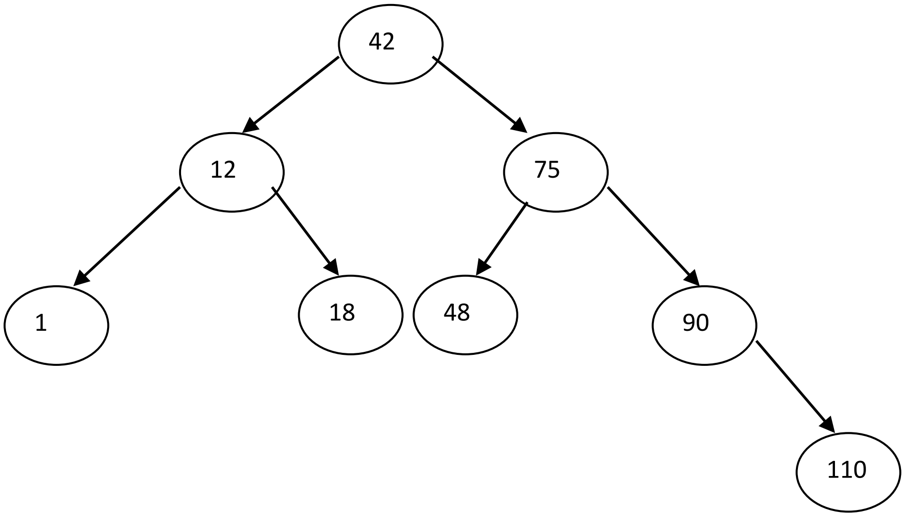
(3) If a node to be deleted has left and right sub tree then we have to do following steps:

(a) Find next maximum (in order successor) of the deleted node.

(b) Append the right sub tree of the in order successor to its grandparent.

(c) Replace the node to be deleted with its next maximum (in order successor).

Suppose we want to delete node 55. Here node 55 has both left and right sub tree. So first we have to find next maximum (in order successor) of node 55 which is 75.now replace the node to be deleted with its next maximum (in order successor). So we replace node 55 with node 75. Thus tree after deleting node 55 is as follow:



Search Operation on Binary Search tree

Search Operation

- First the key to be searched is compared with root node. If it is not in the root node then we have to possibilities :

  (1) If the key to be searched having value less than root node then we search the key in left sub tree.

  (2) If the key to be searched having value greater than root node then we search the

  key in right sub tree.

Representation of binary tree

- We can represent binary tree in memory by two methods:

  (1) Array implementation

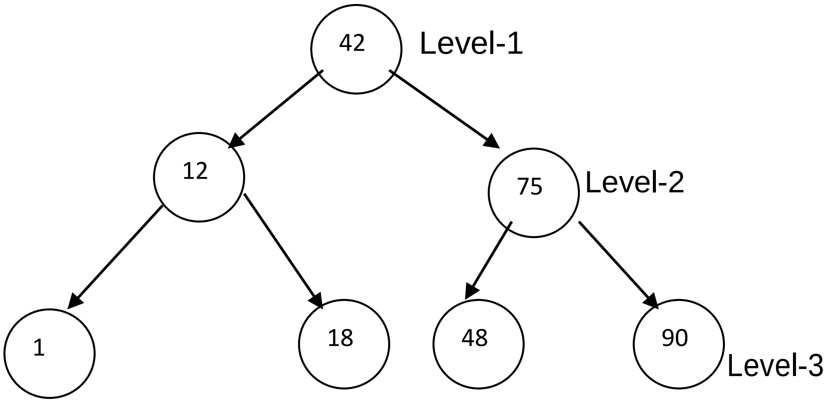
  (2) Linked implementation

Array implementation of Binary tree

- While implementing binary tree we have to consider two cases:

(1) Array implementation of complete binary tree:

- A complete binary tree is a tree in which there is one node at the root level, two nodes at level 2, four nodes at level 3 etc…
- Example of such a tree and its array implementation is shown below:

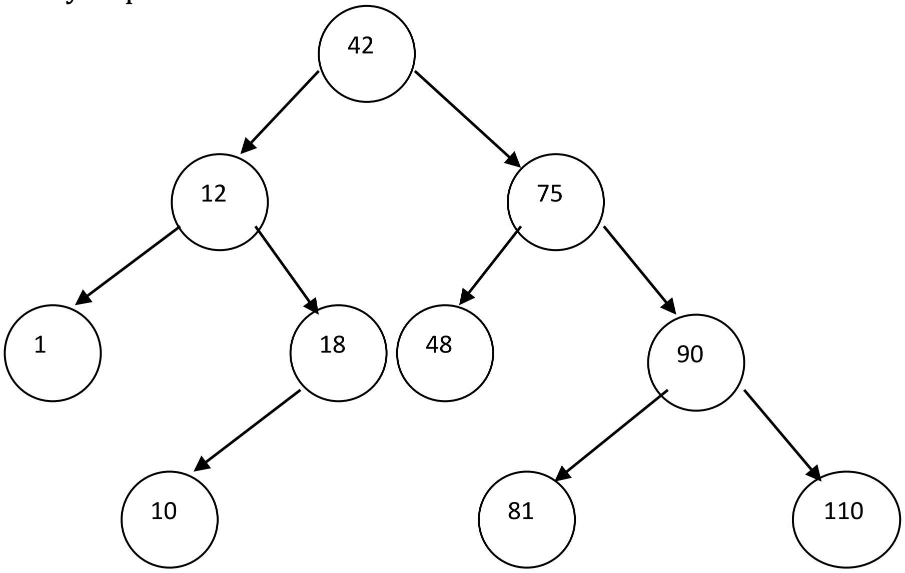


Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Info	**42**	**12**	**1**	**18**	**75**	**48**	**90**

- In this implementation to find the location of left child of node I is given by **2*I** and the location of right child of node I is given by **2*I + 1**.
- For example: consider the node 2 in above example which is 12. The location of left child is given by 2*I = 2*2 = 4 which is 1. location of right child is given by 2*I + 1 = 2*2 + 1 = 5 which is 18.
- Similarly the location of the parent of left node is given by **I/2** & right node is given by **(I-1)/2**. For example location of parent of node 4 and 5 is 2.
- Thus to represent complete binary tree with 2_{n-1} node we require 2_{n-1} array elements.

(2) Array implementation of incomplete binary tree:

- Now consider the tree which has number of nodes less than or more than 2_{n-1}
- Array implementation of such a tree is shown below:



Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Info	42	12	75	1	18	48	90	-	-	10	-	-	-	81	100

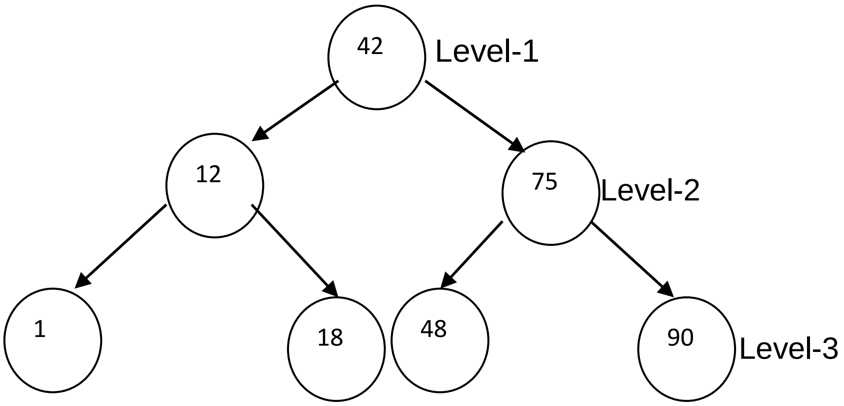
- In this type of implementation the location of left children of node I is given by **2*I** and the location of right children of node I is given by **2*I + 1**.
- Thus location of left children and right children of node 3 is given by 2*I = 2*3 = 6 and 2*I + 1 = 2*3 + 1 = 7. But node 3 has no left or right node so this memory location becomes wastage in array implementation.
- Thus this method is not efficient for incomplete binary tree.

Linked implementation of Binary tree

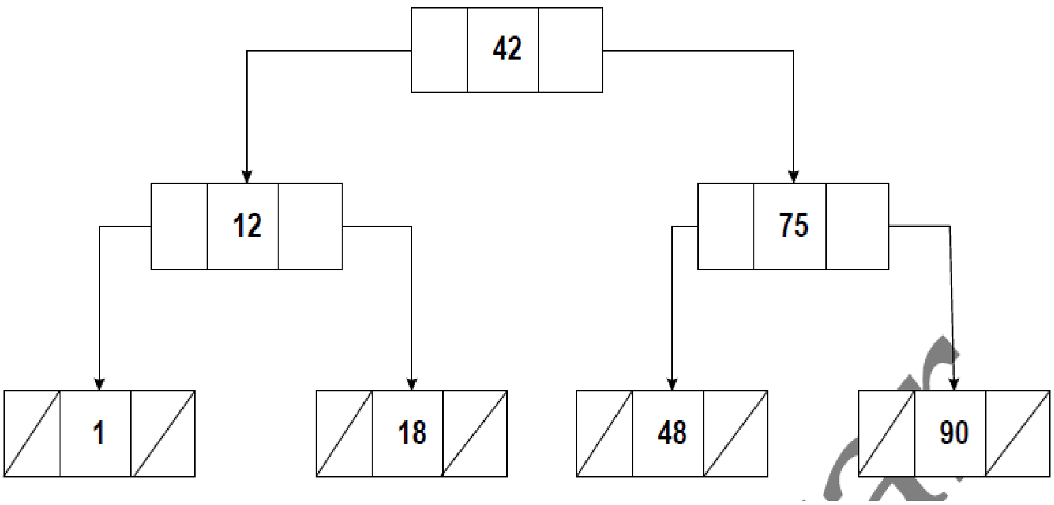
- In binary tree there is a node called root node. This root node again has two parts left sub tree and right sub tree.
- Thus a node in binary tree is represented as below:

LPTR	**INFO**	**RPTR**

- Each node have three parts:
  - (1) Info: information of node
  - (2) LPTR: pointer which points to left child of the node.
  - (3) RPTR: pointer which points to right child of the node.
- If a node which has no left or right child then its LPTR and RPTR fields are represented with NULL.
- **Example:**



- The above tree can be represented in linked implementation as below:



Applications of Tree

- Following are the application of Tree:

(1) Manipulation of arithmetic expressions

Binary tree is used to represent formulas in prefix or pofix notation.

For Example: Infix- A+B

So in above example operator(+) becomes root node and operand(A and B) becomes left and right child node.

(2) Construction and maintenance of symbol table

As an application of binary tree ,we will formulate the algorithms that will maintain a stack implemented tree structured symbol table.

So here binary tree structure is chosen for two reasons. The first reason is because if the symbol entries as encountered are uniformly distributed according to lexicographic order, then table searching becomes approximately equivalent to binary search, as long as the tree is maintained in lexicographic order. second a binary tree is easily maintained in lexicographic order in the sense that only a few pointers need be changed.

(3) Syntax Analysis: It deals with the use of grammars in syntax analysis or parsing.

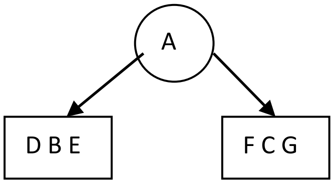
Creation of Binary tree from Basic Traversal

- **Example:**

The in order and pre order traversal of binary tree are
D B E A F C G
A B D E C F G Construct Binary tree

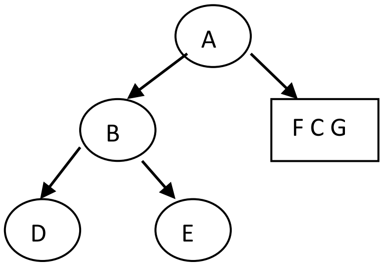
Step-1

In order -D B E **A** F C G
Pre order - **A** B D E C F G
↓
Root



Step-2

In order- D **B** E
Pre order- **B** D E
↓
Root

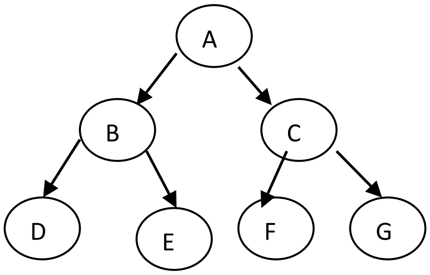


Step-3

In order- F **<u>C</u>** G

Pre order- **<u>C</u>** F G

Root



<u>Threaded Binary Trees</u>

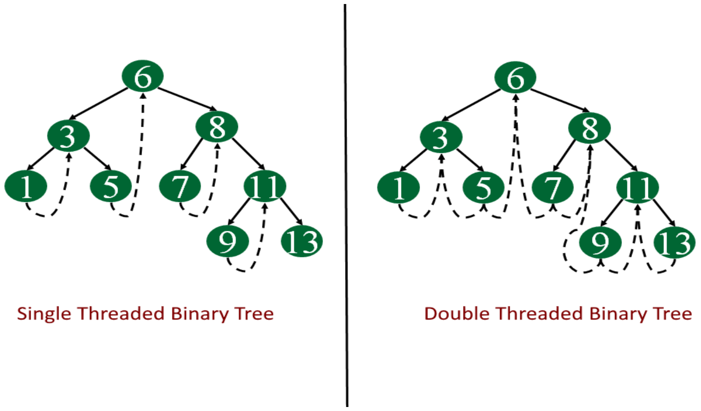
Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the in order successor of the node (if it exists).

There are two types of threaded binary trees.

- Single Threaded: Where a NULL right pointers is made to point to the in order successor (if successor exists)
- Double Threaded: Where both left and right NULL pointers are made to point to in order predecessor and in order successor respectively. The predecessor threads are useful for reverse in order traversal and post order traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



Single Threaded Binary Tree

Double Threaded Binary Tree

<u>Concept of Balanced Trees</u>

A binary tree is balanced if for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most 1. A binary tree is balanced if for any two leaves the difference of the depth is at most 1.

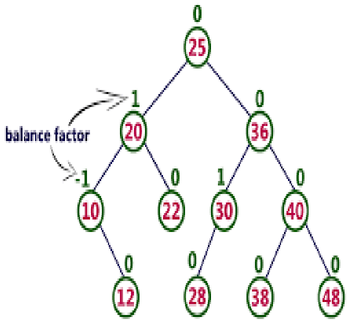
Various Types of balanced trees are-

1. Height balanced tree or AVL tree

2. B-Tree

AVL Trees

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the heights of left and right sub trees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as balance factor.



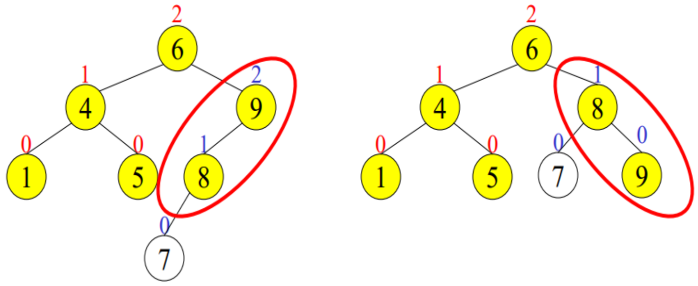
- **Insertion**

Insert and Rotation in AVL Trees
Insert operation may cause balance factor to become 2 or –2 for some node

- only nodes on the path from insertion point to root node have possibly changed in height
- So after the Insert, go back up to the root node by node, updating heights
- If a new balance factor (the difference hleft - hright) is 2 or –2, adjust tree by rotation around the node

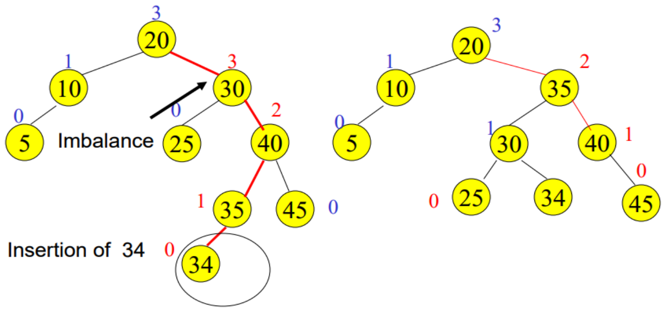
Single Rotation in an AVL Tree

1.Left-Left(LL) rotation
2.Right-Right(RR) rotation



Double Rotation in an AVL Tree

1.Left-Right(LR) rotation
2.Right-Left(RL) rotation

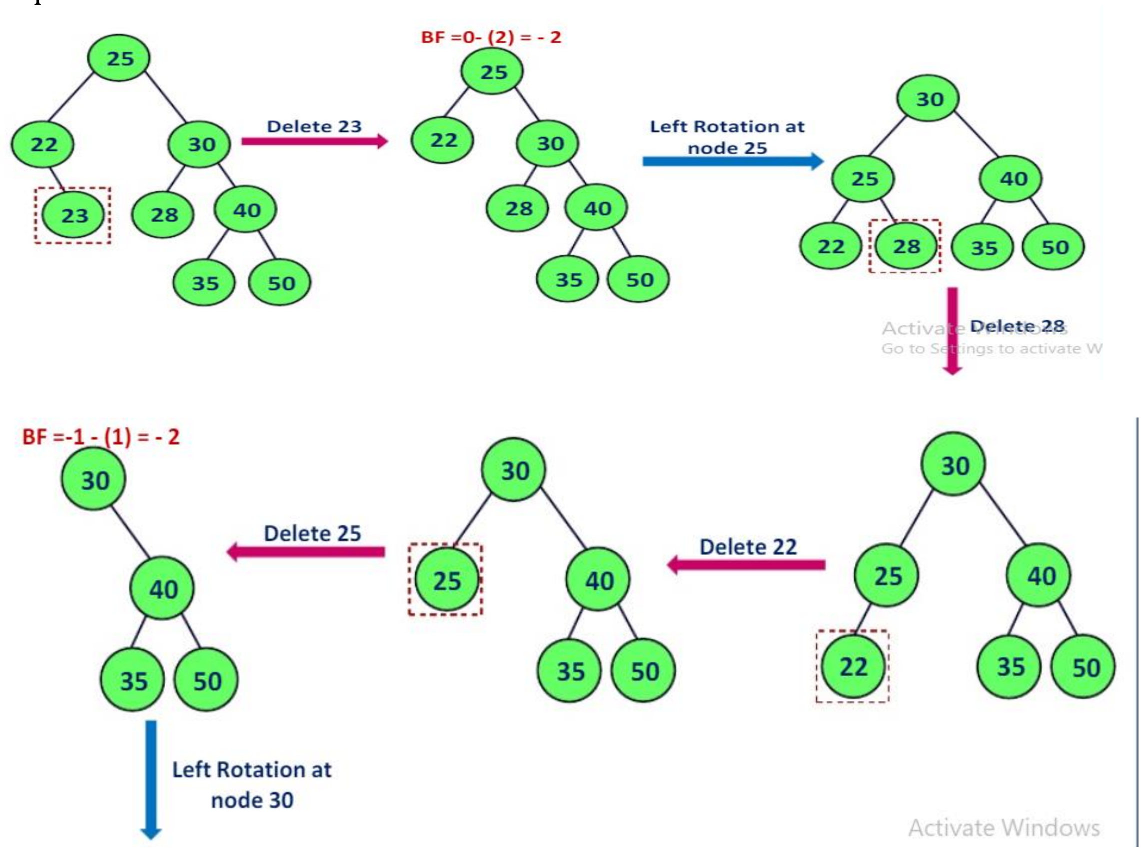


➢ Deletion

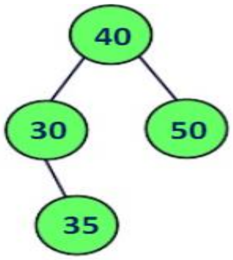
Deletion of nodes from an AVL tree is similar to that of Insertion But the main difference between AVL Insertion and AVL Deletion is that

Imbalance caused in insertion can be corrected only with one rotation (single or double). In Insertion if imbalance occur then we need to find the first unbalanced node only from newly inserted node, by traversing toward Root node, and performing required rotation at that node will fix the whole AVL tree.

But deletion of a node can imbalance multiple nodes in the tree (In ancestry line up-to Root node), so multiple rotations may be required and we need to traverse all the way back up-to the root node looking for imbalance nodes and performing required rotations.



Activate Windows
Go to Settings to activate Windows.
Activate Windows



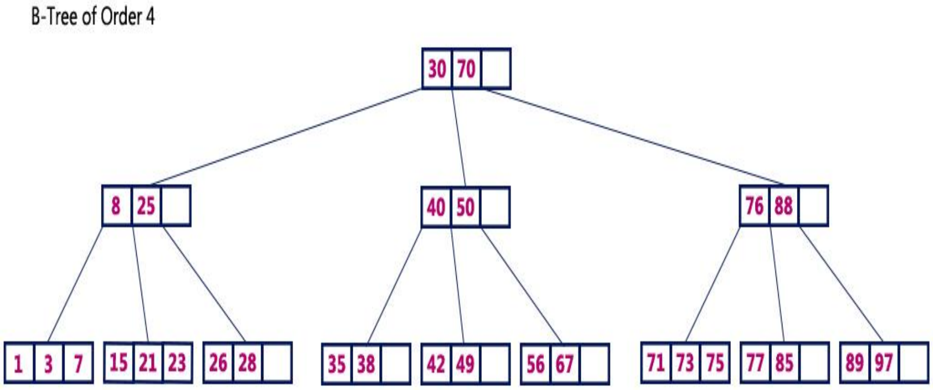
Example

Build an AVL tree with the following values: 15, 20, 24, 10, 13, 7, 30, 36, 25

B-trees

A B-tree of order m is an m-ary search tree with the following properties:

- The root is either a leaf or has at least two children
- Each node, except for the root and the leaves, has between [m/2] and m children
- Each path from the root to a leaf has the same length.
- The root, each internal node, and each leaf is typically a disk block.
- Each internal node has up to (m - 1) key values and up to m pointers (to children)
- The records are typically stored in leaves (in some organizations, they are also stored in internal nodes)
- For example, B-Tree of Order 4 contains maximum 3 key values in a node and maximum 4 children for a node.



Operations on B-tree

The following operations are performed on a B-Tree...

- ➢ Search
- ➢ Insertion
- ➢ Deletion

Search operation on B-tree

- ➢ Step 1: Read the search element from the user
- ➢ Step 2: Compare, the search element with first key value of root node in the tree.
- ➢ Step 3: If both are matching, then display "Given node found!!!" and terminate the function

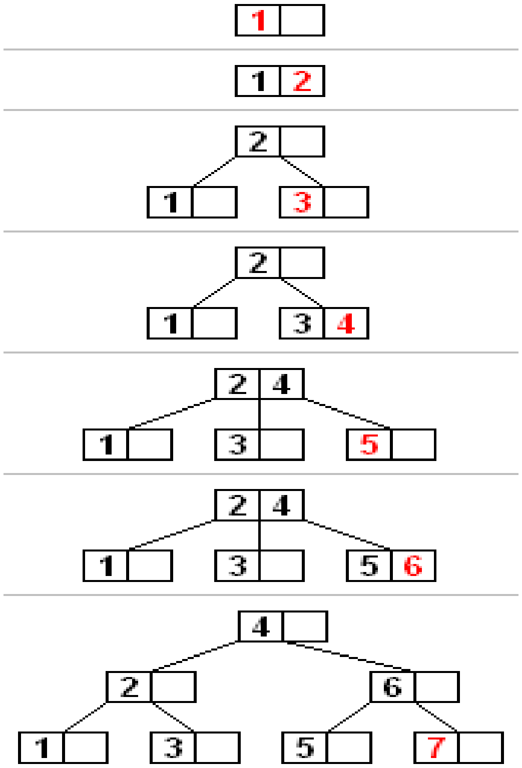
- Step 4: If both are not matching, then check whether search element is smaller or larger than that key value.
- Step 5: If search element is smaller, then continue the search process in left subtree.
- Step 6: If search element is larger, then compare with next key value in the same node and repeate step 3, 4, 5 and 6 until we found exact match or comparision completed with last key value in a leaf node.
- Step 7: If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.

Insertion operation on B-tree

- Step 1: Check whether tree is Empty.
- Step 2: If tree is Empty, then create a new node with new key value and insert into the tree as a root node.
- Step 3: If tree is Not Empty, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
- Step 4: If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
- Step 5: If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat tha same until sending value is fixed into a node.
- Step 6: If the spilting is occuring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

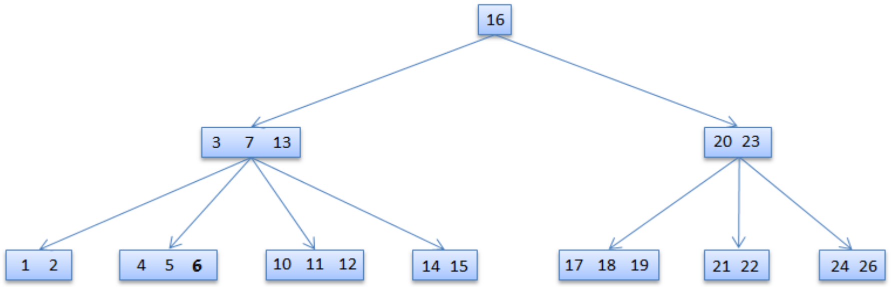
A B Tree insertion example with each iteration. The nodes of this B tree have at most 3 children (order 3).insert value from 1 to 7



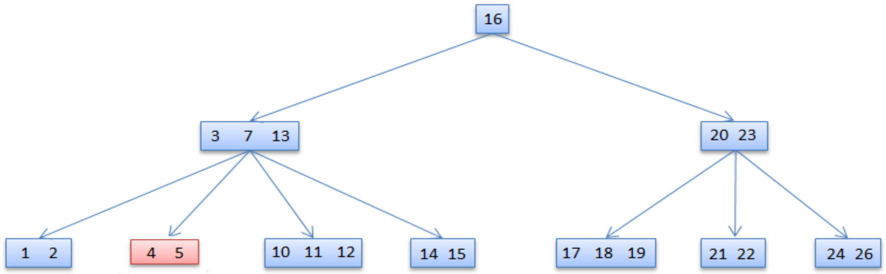
Deletion operation on B-tree

Case-1

If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted. key k is in node x and x is a leaf, simply delete k from x.



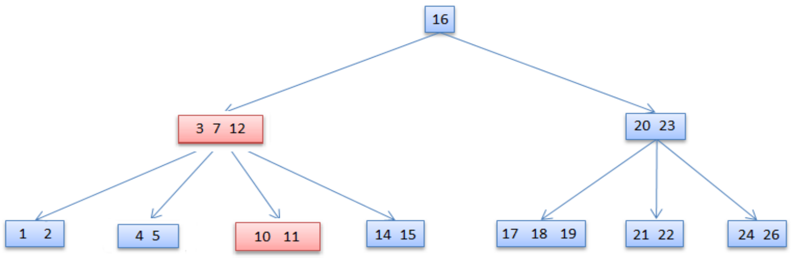
6 is Deleted



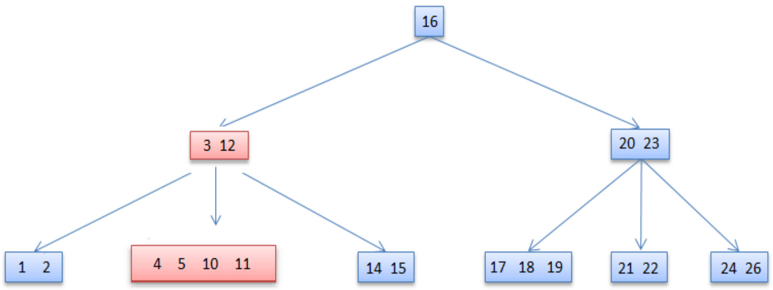
Case-2

If key k is in node x and x is an internal node.

13 is Deleted



7 is Deleted



2-3 Trees

A 2-3 Tree is a specific form of a B tree. A 2-3 tree is a search tree. However, it is very different from a binary search tree.

- Here are the properties of a 2-3 tree:

1. each node has either one value or two value
2. a node with one value is either a leaf node or has exactly two children (non-null). Values in left subtree < value in node < values in right subtree
3. a node with two values is either a leaf node or has exactly three children (non-null). Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
4. all leaf nodes are at the same level of the tree

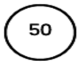
Example

Insertion

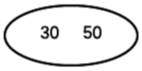
The insertion algorithm into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:

1. If the tree is empty, create a node and put value into the node
2. Otherwise find the leaf node where the value belongs.
3. If the leaf node has only one value, put the new value into the node
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent then has three values, continue to split and promote, forming a new root node if necessary

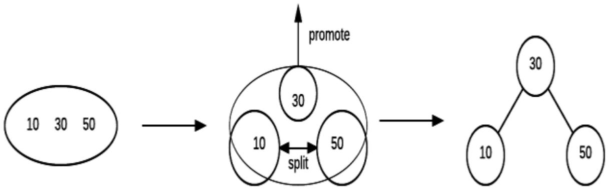
Insert 50



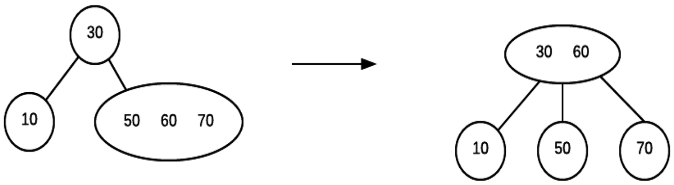
Insert 30



Insert 10



Insert 60,70



Deletion

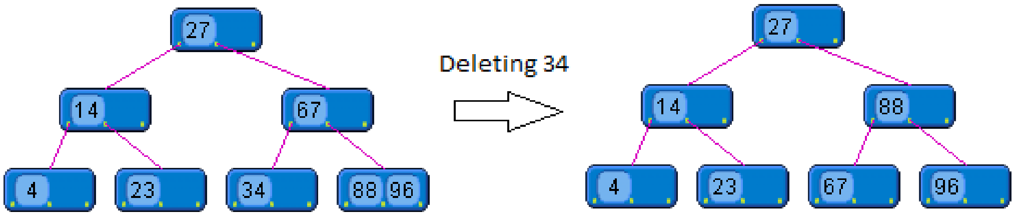


Figure 4: Deleting element from a 2-node, local rotation performed to fix the tree.

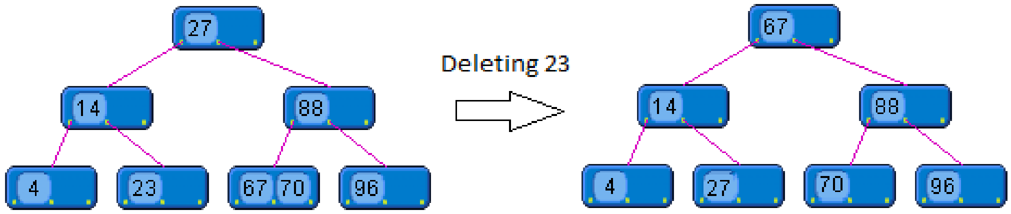


Figure 5: Deleting element from a 2-node, global rotation performed to fix the tree.