TREES

Introduction

Trees are non linear data structures and are used to impose a hierarchical structure on a collection of data items. For example, we need to impose a hierarchical structure on a collection of data items while preparing organizational charts and to represent the syntactic structure of a source program in compilers.

Definition of a Tree

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

i. there is a specially designated node called a root
ii. The remaining nodes are partitioned into *n disjointed* set of nodes T_1, $T_2,\ldots,T_n$, each of which is a tree.

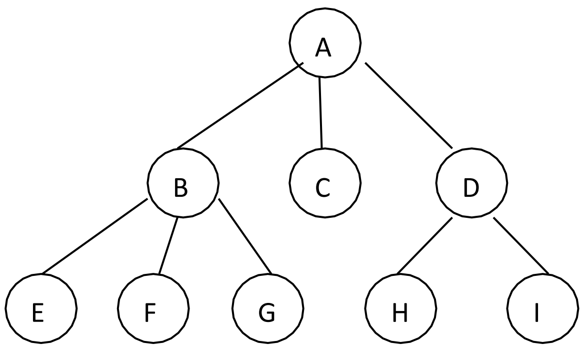


Figure1. Tree Structure

This is a tree because it is a set of nodes {A,B,C,D,E,F,G,H,I}, with node A as a root node and the remaining nodes partitioned into three disjointed sets {B,E,F,G}, { C} and {D,H,I}, respectively. Each of these sets is a tree because each satisfies the aforementioned definition properly.

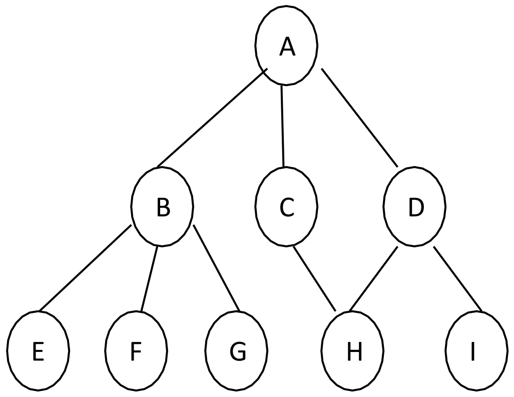


Figure 2: A non-tree structure.

Even though this is a set of nodes {A,B,C,D,E,F,G,H,I}, with node A as a root node, this is not a tree because H is a child node of both C and D.(A,C,H,D forms a cycle).

Applications of Trees

1. On a computer - The file system uses tree architecture; a menu system also has the same idea of selecting progressive layers of options.
2. Indexes in a book have a shallow tree structure - letter/subjects beginning with that letter.
3. Trees are used commonly in internet protocols. Routers use BGP (border gateway protocols) to identify how to reach computers in their networks.
4. Trees can be searched using Djikstra's algorithm to find nodes that are closest to the router for the shortest paths.
5. Trees are used in Compilers (Compiler uses tree structures do convert data types and do calculations on Booleans and variables) and database Management systems (Indexing).
6. Make information easy to search (see tree traversal).

BINARY TREE AND ITS REPRESENTATION

Introduction

A *binary tree* is a special case of tree as defined in the preceding section, in which no node of a tree can have a degree of more than 2. Therefore, a binary tree , T is a set of zero or more nodes such that:

i. there is a specially designated node called the root of the tree
ii. the remaining nodes are partitioned into two disjointed sets, T_1 and T_2, each of which is a binary tree. T_1 is called the left subtree and T_2 is called right subtree, or vice-versa.

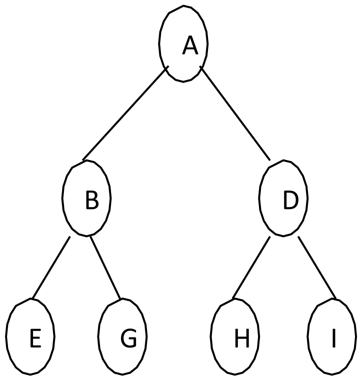


Figure 3: Binary tree structure.

The level of a node in a binary tee is defined as follows: The root of the has level 0 and the level of any other node in the tree is one more than the level of its parent node. The depth of the binary tree is the maximum level of any leaf in the tree. The depth of the above tree in fig 3 is 2.Height of the tree is 3.(i.e.,depth+1).

Total no of Binary Trees with 'n' number of nodes are = $\frac{(2n)!}{(n+1)!\,n!}$

Uses of binary tree:

- To create sorting routine.
- Persisting data items for the purpose of fast lookup later.
- For inserting a new item faster

Strictly Binary Tree: If every non leaf node in a binary tree has non-empty left and right sub trees, the tree is termed as strictly binary tree.

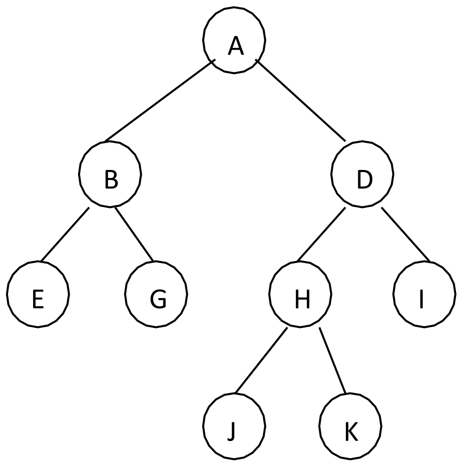


Figure4: Strictly Binary Tree

Complete Binary Tree of depth d is the strictly binary tree all of whose leaves are at level d. The below figure shows complete binary tree of level 2. In a complete binary tree, if the parent is at nth position then the children will be at 2n and 2n+1.

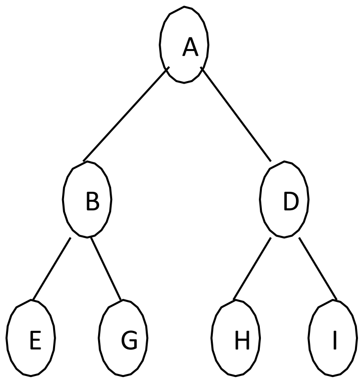


Figure 5: Complete binary tree

Almost Complete binary Tree /Full Binary Tree

A binary tree of depth d is **an almost complete binary tree** if:

1. Every node at level less than *d* has two sons.

2. For any node in the tree with a right descendant at level *d*, must have a left son and nodes in level *d* must be inserted from left to right.

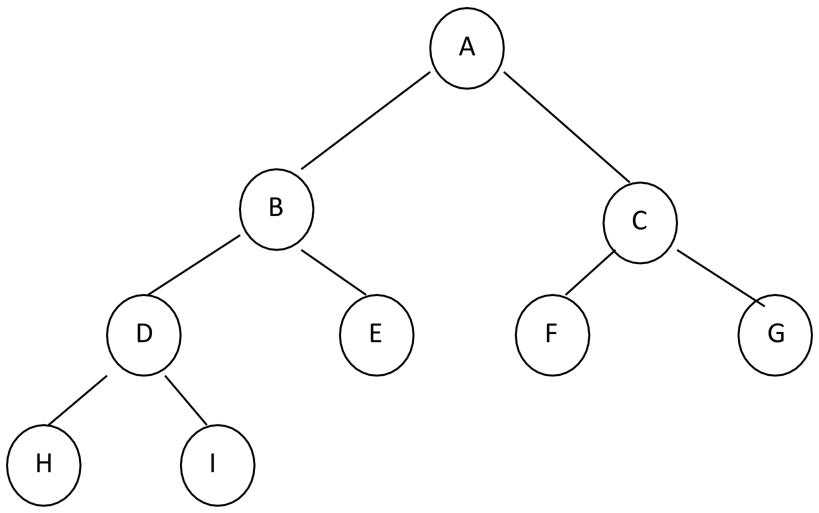


Figure 6(a): Almost Complete Binary Tree

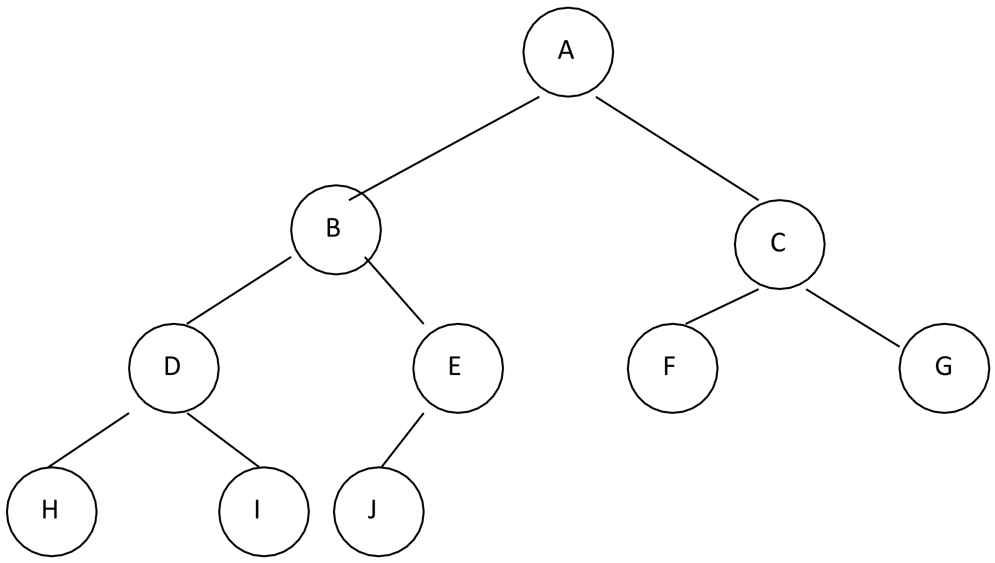


Figure 6(b): Almost Complete Binary Tree

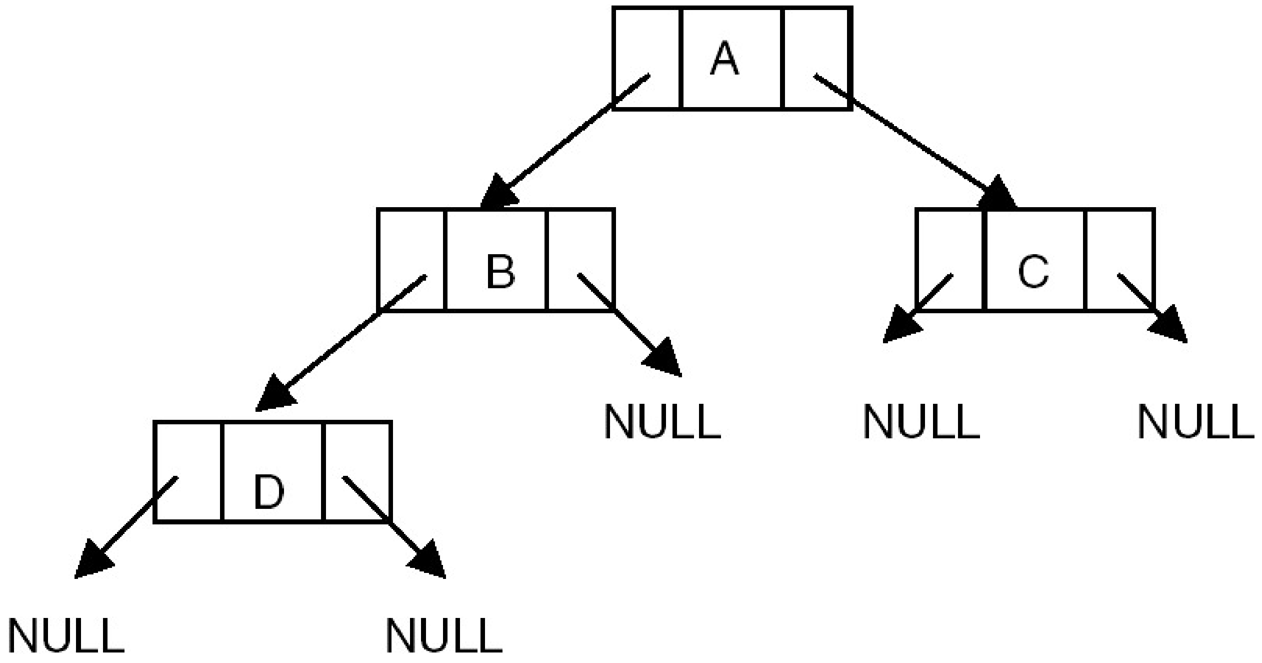


Figure 7: Linked representation of a binary tree

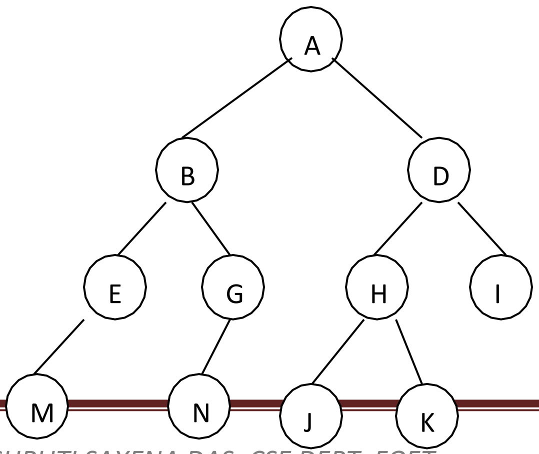
BINARY TREE TRAVERSALS

To traverse a nonempty binary tree in preorder or depth first order

1. Visit the root

2. Traverse the left subtree in preorder

3. Traverse the right subtree in preorder



Preorder: A B E M G N D H J K I

To traverse a nonempty binary tree in inorder or Symmetric order

1. Traverse the left subtree in inorder

2. Visit the root

3. Traverse the right subtree in inorder For the Above tree:

   Inorder: M E B N G A J H K D I

To traverse a nonempty binary tree in post order or breadth first order

1. Traverse the left subtree in postorder

2. Traverse the right subtree in postorder

3. Visit the root

For the Above tree:

Postorder: M E N G B J K H I D A

Constructing a Binary Tree Using the Preorder and Inorder Traversals

To obtain the binary tree,

1. First node of preorder traversal will be the root node.
2. Find the position of root node in inorder traversal.

   All the nodes left to the root node in inorder will be in the left subtree and nodes right to the root node in inorder will be in the right subtree.

3. Consider next node in preorder traversal as the root node and repeat step 2

till end of preorder expression.

Construct a Binary Tree to the given:

Inorder : D B E A C
Preorder: A B D E C And give its postorder

Solution:

1) Inorder:D B E A C

   Preorder: A B D E C

   DBE- to the left of A

   C- to the right of A

2) Inorder:D B E A C

   Preorder: A B D E C

   B is the root node, left to A.

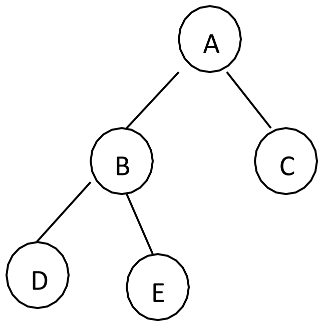
   D and E are left and right nodes of B.

3) Inorder:D B E A C

   Preorder: A B D E C

   As D and E are left and right nodes of B, place D and E as left and right nodes of B.

4) Consider C and as C is the only node to the right of A in given inorder,
   place it as a right child.



Post order for the above tree is D E B C A

Constructing a Binary Tree Using the Postorder and Inorder Traversals

To obtain the binary tree,

1. Last node of postorder traversal will be the root node.
2. Find the position of root node in inorder traversal.
   All the nodes left to the root node in inorder will be in the left subtree and nodes right to the root node in inorder will be in the right subtree.
3. Consider next node in postorder traversal as the root node and repeat step 2 till end of postorder expression.

Construct a Binary Tree to the given:

Inorder: D B E A C

Postorder: D E B C A

And give its preorder

Solution:

1)Inorder:D B E A C

Postorder: D E B C A // A is the root node

DBE- to the left of A

C- to the right of A

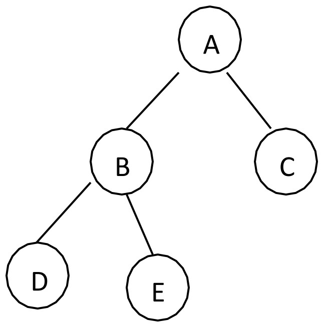
2)Inorder:D B E A C

Postorder: D E B C A

C is the right node to A.

3) Inorder:D B E A C
4) Postorder: D E B C A
B is left to A and is the parent node to D and E.

D is placed as left to B and E is placed as right to B.

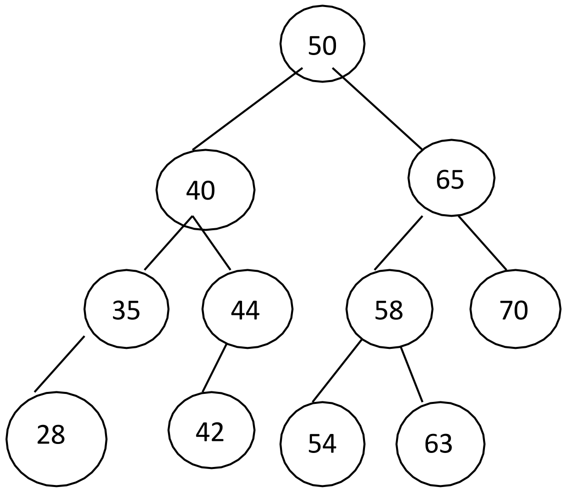


Preorder for the above tree is A B D E C

Binary Search Tree or Binary Sort Tree

A *binary search tree* is a binary tree that may be empty, or non empty. In non-empty binary search tree,every node must contain data. The data of any node in the left subtree is less than the data of the root. The data of any node in the right subtree is greater than the data of the root. Both the left subtree and right subtree \-+5 are binary search trees. A binary search tree is also called as ordered binary tree.

-



The binary search tree is basically a binary tree, and therefore it can be traversed in inorder, preorder, and postorder. If we traverse a binary search tree in inorder and print the identifiers contained in the nodes of the tree, we get a sorted list of identifiers in ascending order

Preorder Traversal:

The procedure for traversing a tree in preorder non recursively is as:- Initially p contains the address of the root

```
1. Push the address of root node on the stack.
2. Pop an address from the stack
3. If the popped address is not NULL
        Traverse the node
        Push right child of node on stack
         Push left child of node on stack
4. Repeat steps 2, 3 until the stack is not empty
```

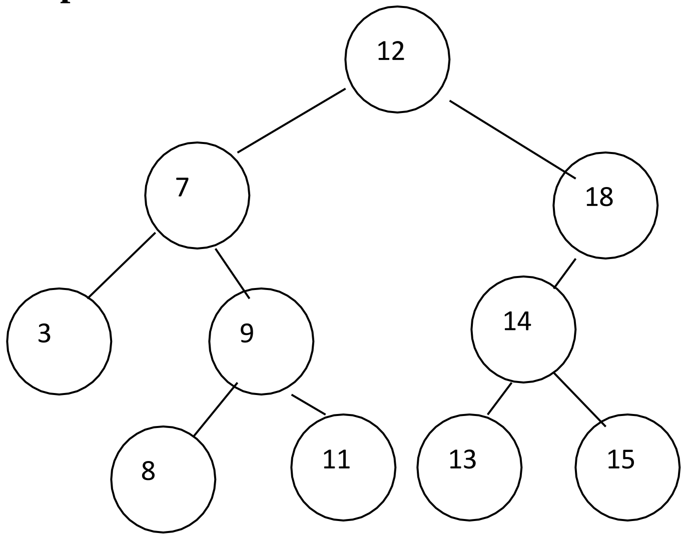
Inorder Traversal:

The procedure for traversing a tree in inorder non recursively is as:-
Initially p contains the address of the root

```
1. Repeat steps 2, 3 while stack is not empty or p is not equal to NULL.
2. If p is not NULL
        push p on to stack
        p=p->rchild
3. IF p is NULL
        pop an address from stack traverse the node at that address
        p=p->rchild
```

Postorder Traversal:

1. In a postorder traversal, a node's left subtree is first output, followed by the right subtree, and finally the node is output. Thus, we need a stack in which we push the right child, followed by the left child. But we also need the node itself in the output. So, we will have to push it again in the stack before pushing its children. But then how will we keep track of whether that node is already processed? If we don't, we will again push it and its children in the stack, forming an infinite loop!
2. One way is to tag it as 'processed.' This can be done by adding a field to each node that will specify its status. Another way is to keep a list of processed nodes. We use this latter approach, maintaining a stack of processed nodes. The reason for using a stack is that when all the nodes in the tree are processed, the nodes in the stack give postorder traversal in reverse. Thus, by popping the nodes and outputting one by one, we get the required postorder traversal.
3. **Example**:



The postorder traversal of this tree is 3, 8, 11, 9, 7, 13, 15, 14, 18, 12. The stepwise run of the algorithm is shown next:

step	node	stack 1	stack 2
0	Nil	empty	Empty
1	12	7 18	12
2	18	7 14	12 18
3	14	7 13	15 12 18 14
4	15	7 13	12 18 14 15
5	13	7 12	18 14 15 13
6	7	3 9	12 18 14 15 13 7
7	9	3 8 11	12 18 14 15 13 7 9
8	11	3 8	12 18 14 15 13 7 9 11
9	8	3	12 18 14 15 13 7 9 11 8
10	3	empty	12 18 14 15 13 7 9 11 8 3

Stack 2 can now be output to get the required postorder traversal of the tree.

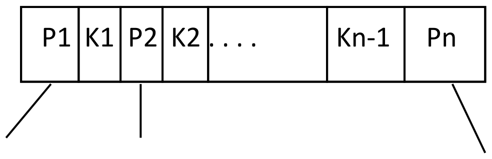
B –TREES

B-tree of order **n** or n-ary tree can be defined as:

1. Except root node, all nodes must have at least **n/2 keys** and maximum **n-1 keys and n pointers.**
2. All leaf nodes must be at same level.

The node in the B-Tree has following structure.

If the B-Tree is of order **n,** then it has n pointers and n-1 key values.

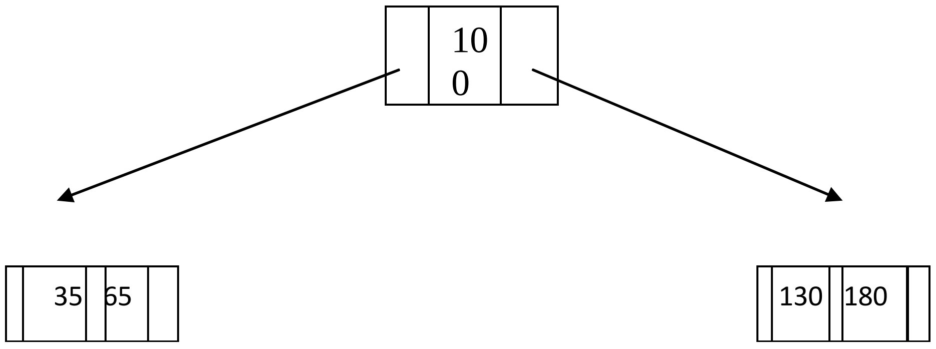


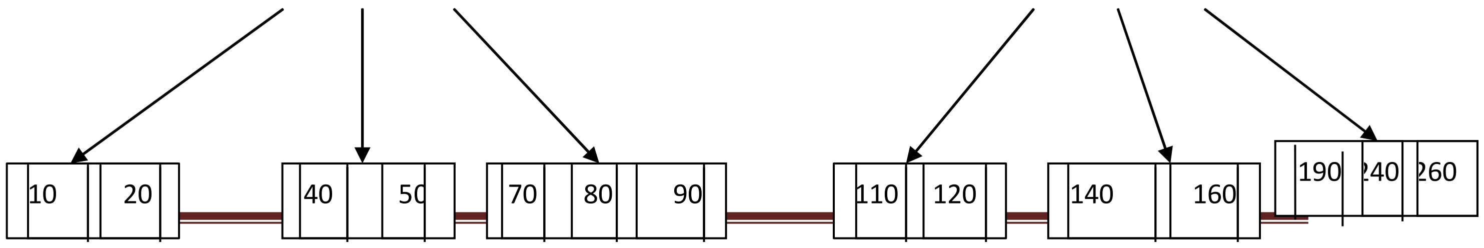
Keys in node are arranged in the following order.

$K_i < K_{i+1}$

Pointer P_1 points to the key values less than K_1. Pointer Pi points to key values between K_i and K_{i+1} Pointer P_n points to key values greater than K_{n-1}.

Example Tree of Order 4:





NOTE: 2–3–4 trees or 2-3 trees are B-Trees of order 4.

Applications of B -Tree:

1. B-trees are useful when the time required to access a node is much greater than access time within a node.

2. B-trees are used in databases and file systems

Insertion in B -Tree:

Insertion of key requires first traversal in B-tree. Traversal is done from leaf node level to root node level. Through traversal it will find that key to be inserted is already existing or not. Now we have two cases in inserting the key-

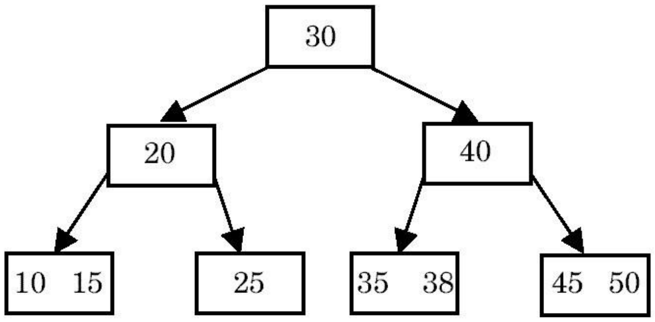
1. Node is not full
2. Node is already full.

In the first case we can simply add the key in node. But in the second case we will need to split the node into two nodes and median key will go to the parent of that node.

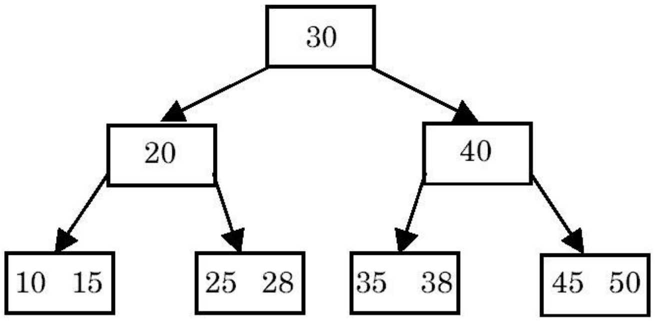
Median Key = 1+((N-1)/2), where N is the order of the tree.

If parent is also full, then the same thing will be repeated until it will get non full parent node. Suppose root is full then it will split into two nodes and median key will be root.

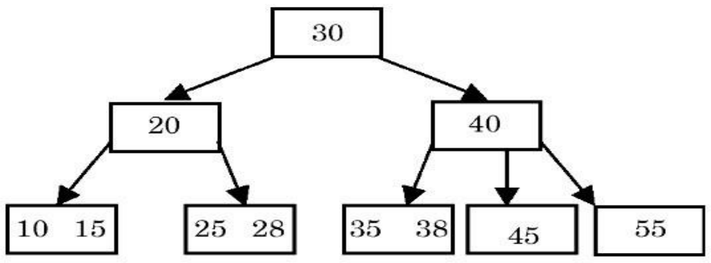
Examples: Original tree of order K = 3



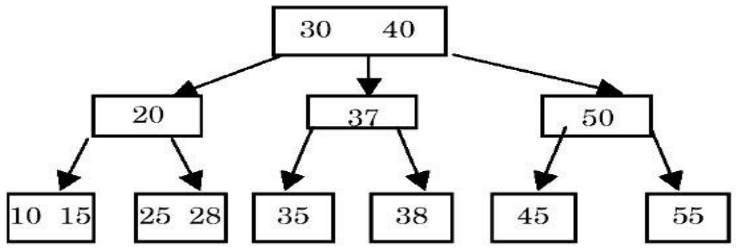
Insertion of value 28



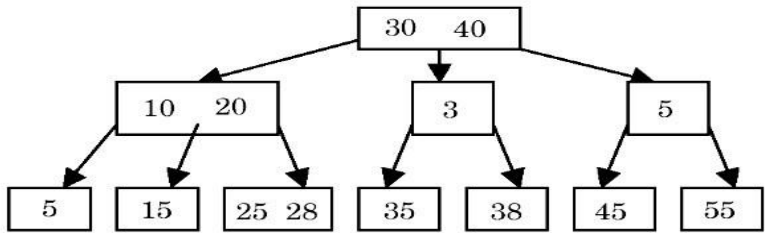
Insertion of value 55



Insertion of value 37



Insertion of value 5



Question.1)Create a B-tree of order 5.

10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160

Deletion in B-Tree:

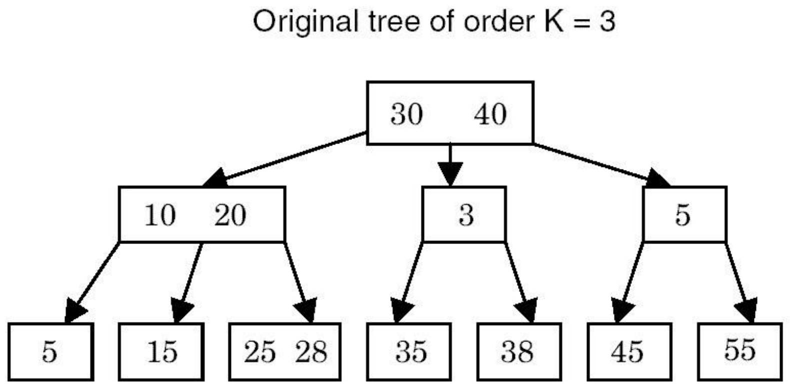
Deletion of key also requires first traversal in B-tree. After reaching on particular node, two Cases may occur.

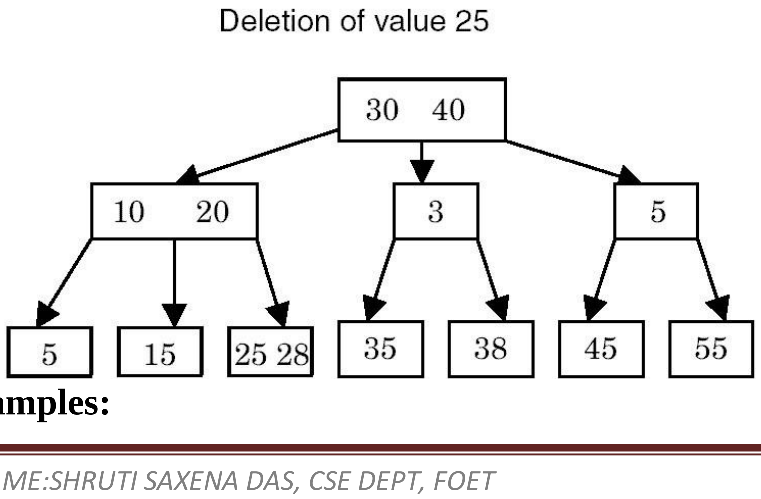
1. Node is leaf node.

2. Node is non leaf node.

For the first case suppose node has more than minimum number of keys then it can be easily deleted. But suppose it has only minimum number of keys then first we will see the number of keys in adjacent leaf node if it has more than minimum number of keys then first key of the adjacent node will go to the parent node and key in parent node which is partitioning will be combined together in one node. Suppose now parent has also less than the minimum number of keys then the same thing will be repeated until it will get the node which has more than the minimum number of keys.

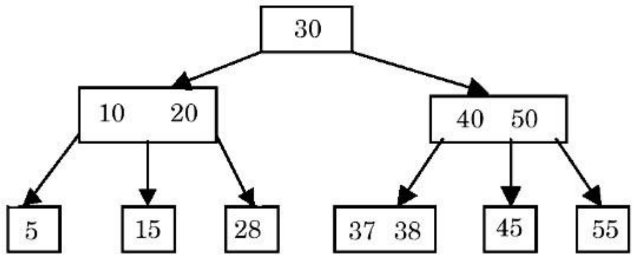
For the second case key will be deleted and its predecessor or successor key will come on its place. Suppose both nodes of predecessor and successor key have minimum number of keys, then the nodes of predecessor and successor keys will be combined.



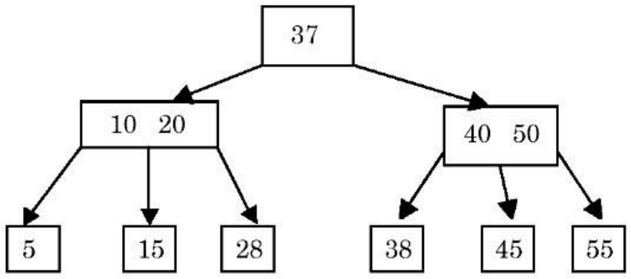


Examples:

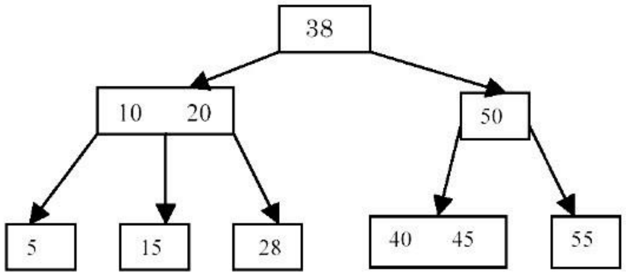
Deletion of value 35

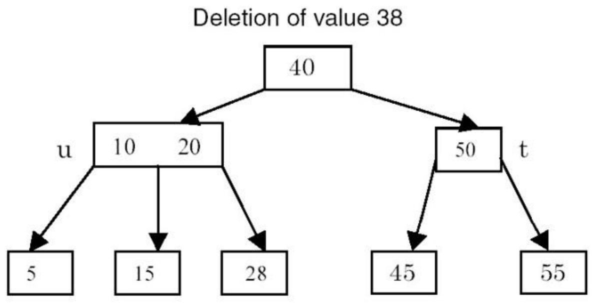


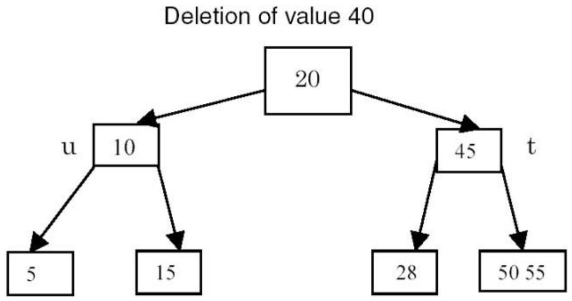
Deletion of value 30



Deletion of value 37







Note that in the deletion of value 40, since it is a non-leaf node, its next value in the search order, 45, is promoted to its place so that 45 appears in the root. Then 45 is deleted from the leaf node r. After deletion, r has less than K/2 values. So its sibling s is checked for any extra values. s, with a value of 55, has only K/2 values. So the values of r, s, and 50 (the value of the parent node t between the two children) are combined to get a node with the values (50, 55). Now t contains less than K/2 values. Since it is a non-leaf node, it checks its adjacent node to see if it has more than K/2 values. The node u, with the values (10, 20), actually has more than K/2 values. So the last value of u, 20, is promoted to the root and the value 45 is demoted to t. The rightmost sibling of u (the node containing a value of 28) becomes the leftmost sibling of t. 20, the rightmost sibling of u, becomes the leftmost sibling of t. A symmetric transformation is done when the adjacent node appears in the right. If none of the adjacent nodes of t contain any extra values, then t, u, and the value in their parent node between these two pointers are combined to form one node. Then their parent node is checked to see if it has less than K/2 values, and the procedure continues.

Points to Remember

1. In a B-tree of order K, there can be at most K pointers in a node and K – 1 values.
2. All the leaf nodes of a B-tree are at the same level.

3. By writing the functions of insertion and deletion in a similar manner, the code becomes easier to write and understand.
4. Searching for a value inside a node can be done using binary search as the values are sorted. However, if K is small enough, even the linear search will give a similar performance.

AVL Tree(Balanced Tree)

Adel'son-Vel'skii and Landis first defined AVL trees.

An AVL tree is a binary search tree where height of left and right sub tree of any node will be with maximum difference 1.

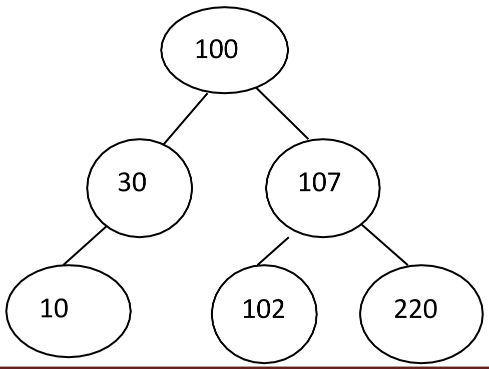
Each node of AVL tree has a balance factor. Balance factor of a node is defined as the difference between the height of left sub tree and right sub tree of a node.

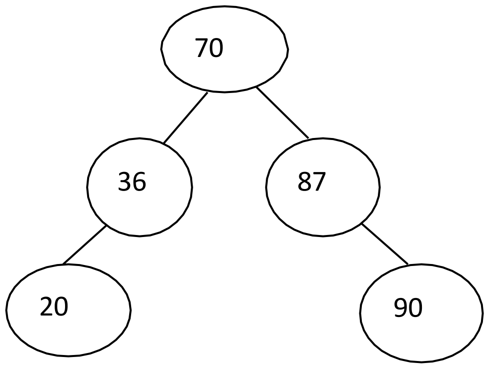
Balance factor=Height of left sub tree-Height of right sub tree.

A node is called right heavy or right high if height if its right sub tree is one more than height of its left sub tree. A node is called left heavy or left high, if height of its left sub tree is one more than height of its right sub tree. A node is called balanced if the heights of right and left sub tree are equal.

The balance factor will be 1 for left high,-1 for right high and 0 for balanced node. So in AVL tree each node can have only 3 values of balance factor which are -1,0,1.

Example:





Insertion in AVL tree

Insertion in AVL tree is same as in binary search tree. Here also we will search for the position where the new node is to be inserted and then insert the node. But AVL tree has a property that the height of left and right sub tree will be with maximum difference 1. Suppose after inserting new node, this difference becomes more than 1 i.e., the value of balance factor has some value other than -1, 0, 1. So now, the property of AVL tree is to be restored.

To restore the property of AVL tree we should convert the tree in such a way that

1. The new converted tree is a balanced tree i.e., the balance factor of each node should be -1,0,1
2. The new converted tree should be a binary search tree.

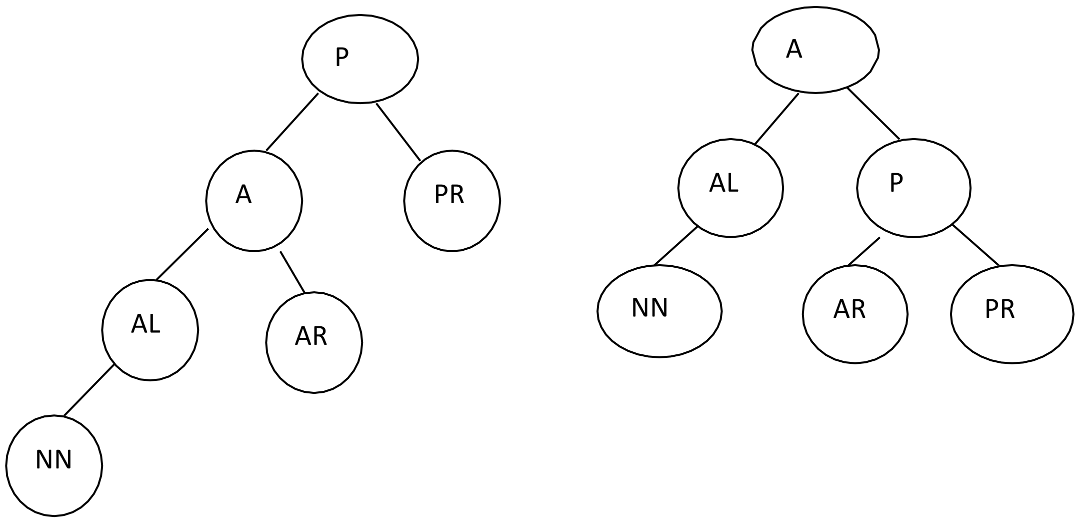
The outline of the procedure of insertion of a node is as-

1. Insert the node at its proper place following the same procedure as in binary search tree.
2. Calculate the balance factors of all the nodes on the path starting from the inserted node to the root node. If the value of balance factor of each node on this path is -1,0, or 1 then tree is balanced and has not lost its AVL tree property after the insertion of new node. There is no need to proceed further. If the absolute value of balance factor of any node in this path becomes more than 1 then the tree becomes unbalanced. The node which is nearest to the inserted node and has absolute value of balance factor greater than 1 is marked as the

pivot node.

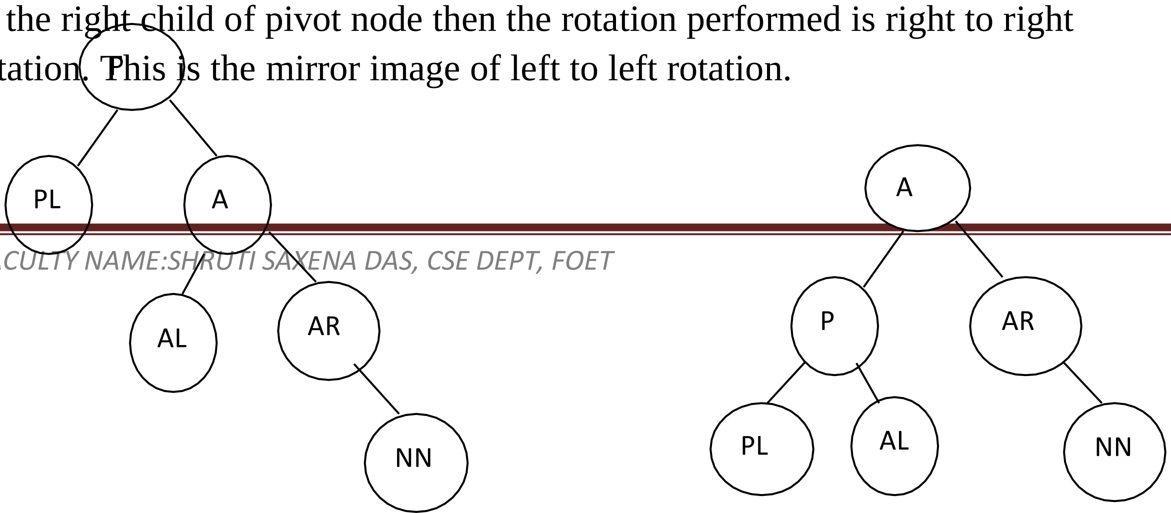
3. If the tree has become unbalanced after the insertion then there is a need to convert the above tree by performing rotations about the pivot node such that the converted tree has all the properties of the AVL tree.
4. AVL Rotations:
   There can be four types of rotations depending upon where the new node is inserted.
   1. Left to left rotation-Insertion in left to left heavy sub tree
   2. Right to right rotation-Insertion in right to right heavy sub tree.
   3. Left to right rotation—Insertion in right to the left heavy sub tree.
   4. Right to left rotation—Insertion in left to the right heavy sub tree.

Left to left Rotation— When the pivot node is left heavy and the new node is inserted in left sub tree of the left child of pivot then the rotation performed is left to left rotation.



Right to right rotation—

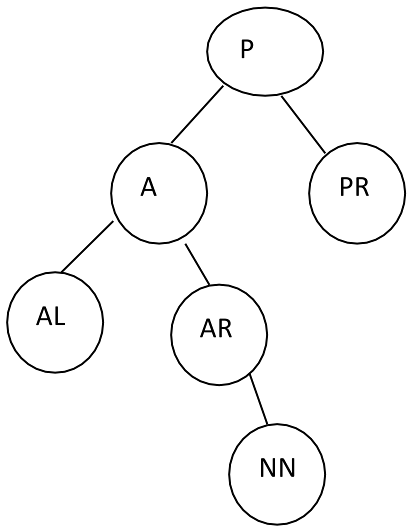
When the pivot node is right heavy and the new node is inserted in right sub tree of the right child of pivot node then the rotation performed is right to right rotation. This is the mirror image of left to left rotation.



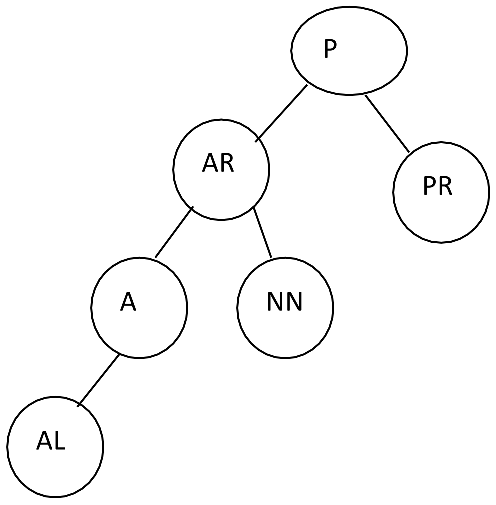
Left to right rotation—

When the pivot node is left heavy and the new node is inserted in right sub tree of the left child of pivot then the rotation performed is left to right rotation.

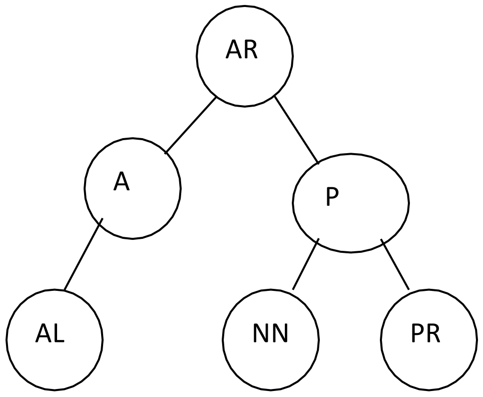
In this case a single rotation around the pivot node will not balance the tree so we have to perform double rotation here. First we will perform a right to right rotation around node A, and then we will perform a left to left rotation around the pivot node.



1. Right to right to rotation on node A



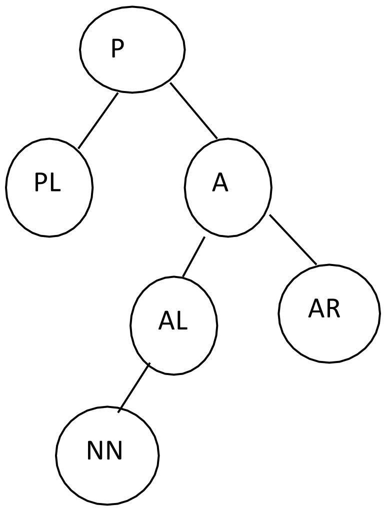
2. Left to Left to rotation on node P



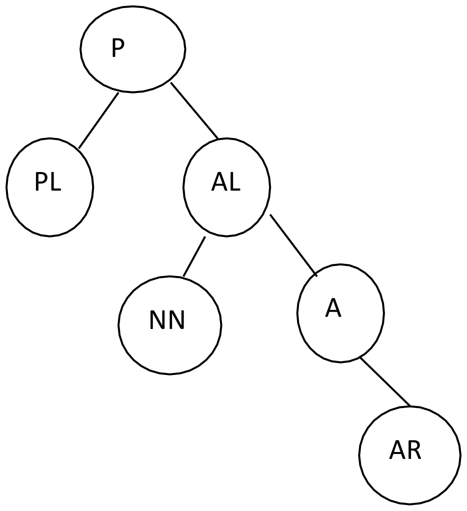
Right to left rotation

When the pivot node is right heavy and the new node is inserted in left sub tree of the right child of pivot node then the rotation performed is right to left rotation.

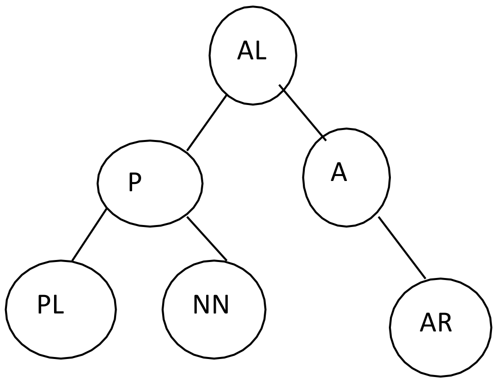
In this case a single rotation around the pivot node will not balance the tree so we have to perform double rotation here. First we will perform a left to left rotation around node A, and then we will perform a right to right rotation around the pivot node. This is the mirror image of the left to right rotation.



1. Apply Left to Left Rotation on node A



2. Apply Right to Right Rotation on node A



Example Problem:

Construct an AVL tree with values: **50, 40, 35, 58, 48, 60, 30, 33, 25**

Deletions from an AVL Tree

1. Delete the node as in a binary search tree
2. Traverse up the tree from the deleted node checking the balance of each node. Make necessary adjustments to maintaining property of Binary search Tree and balance factor.