UNIT- SEVEN
TREES

Topics to be Covered

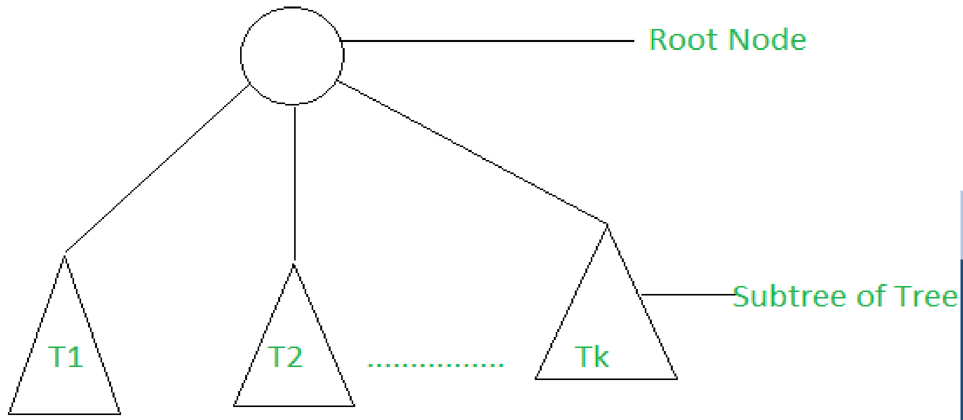
- Introduction
- Basic operation in binary tree
- Tree search and insertion/deletion
- Binary tree traversals(pre-order, post-order and in-order)
- Tree height
- Level and Depth
- Balanced Trees: AVL Balanced Tree
- Balancing Algorithm
- The Huffman Algorithm
- Game tree
- B- Tree

Introduction: Trees

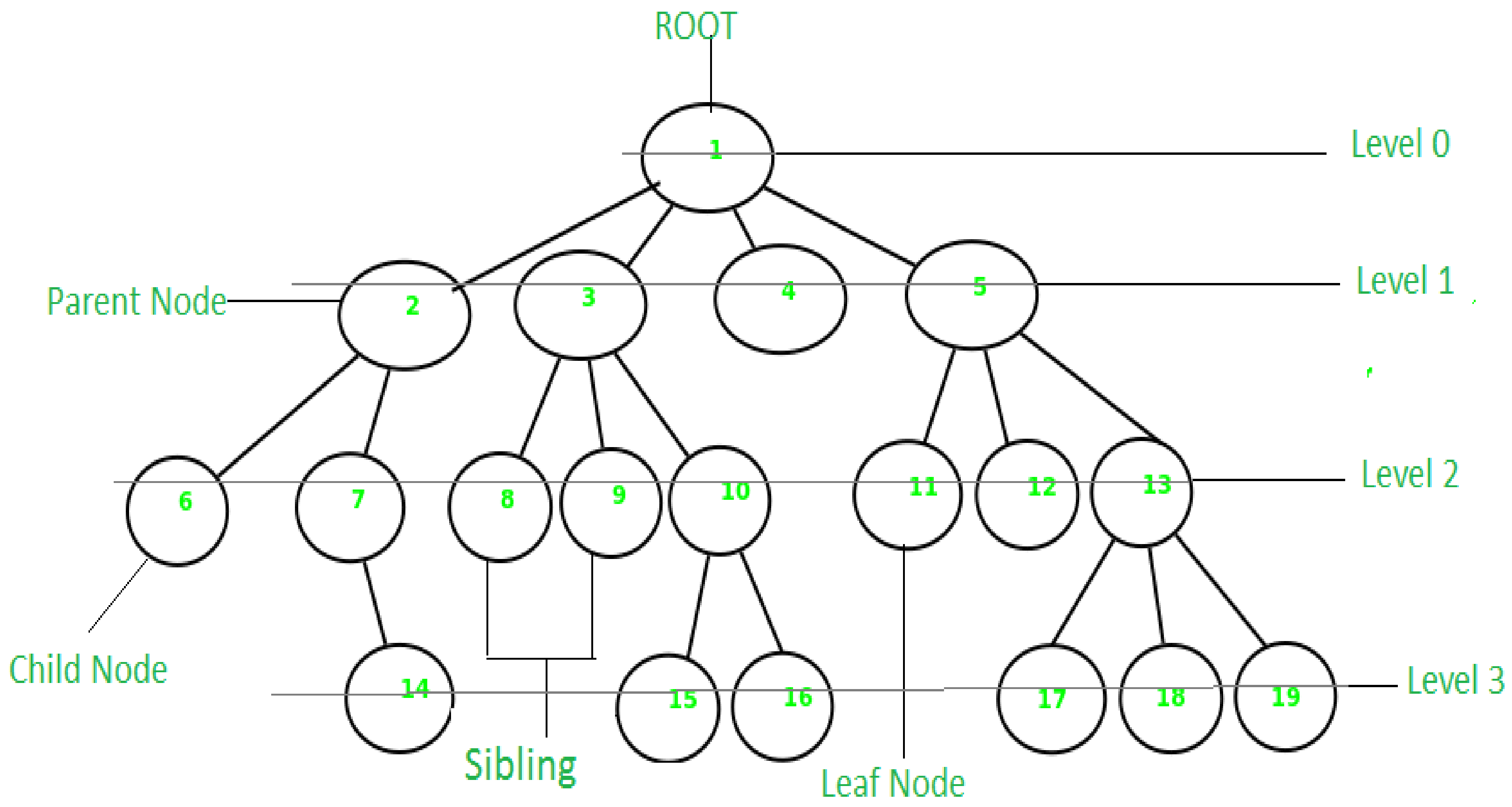
- A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").
- A tree is a specialized method to organize and store data in the computer to be used more effectively.
- It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.

Recursive definition: A tree consists of a root, and zero or more subtrees T1, T2, … , Tk such that there is an edge from the root of the tree to the root of each subtree.

Figure: graphical notation of tree



Example: A tree



Why Tree is considered a non-linear data structure?

The data in a tree are not stored in a sequential manner i.e, they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a non-linear data structure.

Basic Terminologies:

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {**2**} is the parent node of {**6, 7**}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {**6, 7**} are the child nodes of {**2**}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {**1**} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {**6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19**} are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {**1, 2**} are the ancestor nodes of the node **{7}**

- **Descendant:** Any successor node on the path from the leaf node to that node. {**7, 14**} are the descendants of the node. {**2**}.
- **Sibling:** Children of the same parent node are called siblings. {**8, 9, 10**} are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

Properties of a tree:

- **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

Some more properties are:

- Traversing in a tree is done by depth first search and breadth first search algorithm.
- It has no loop and no circuit
- It has no self-loop
- Its hierarchical model.

C implementation of a tree

Syntax:

```
struct Node
{
  int data;
  struct Node *left_child;
  struct Node *right_child;
};
```

Basic Operation Of Tree:

Create – create a tree in data structure.

Insert − Inserts data in a tree.

Search − Searches specific data in a tree to check it is present or not.

Preorder Traversal – perform Traveling a tree in a pre-order manner in data structure .

In order Traversal – perform Traveling a tree in an in-order manner.

Post order Traversal –perform Traveling a tree in a post-order manner.

Types of Tree data structures

The different types of tree data structures are as follows:

1. **General tree**

A general tree data structure has no restriction on the number of nodes. It means that a parent node can have any number of child nodes.

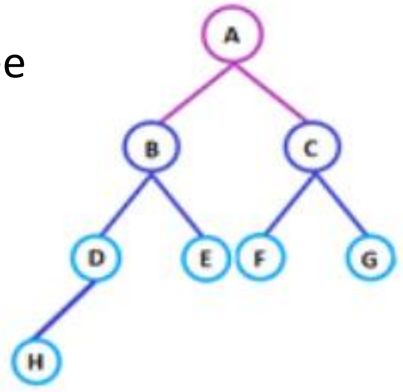
2. **Binary tree**

A node of a binary tree can have a maximum of two child nodes. In the given tree diagram, node B, D, and F are left children, while E, C, and G are the right children.

3. **Balanced tree**

If the height of the left sub-tree and the right sub-tree is equal or differs at most by 1, the tree is known as a balanced tree.

Figure: Balanced Tree



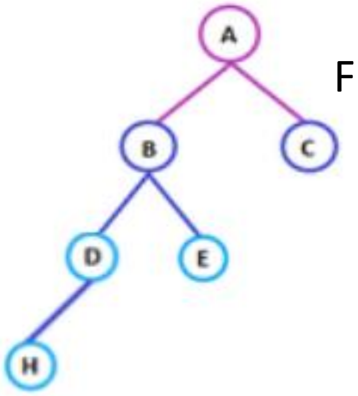
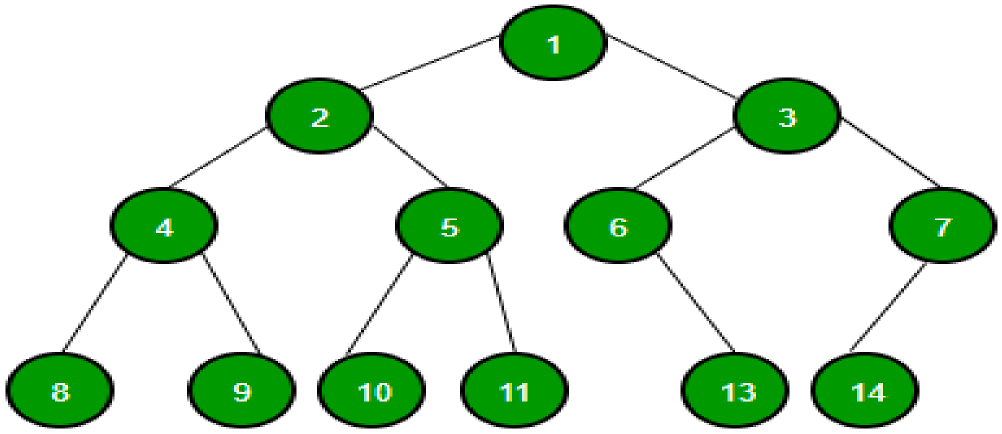


Figure: Unbalanced Tre

Binary Tree

Binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



Binary Tree Representation

- A Binary tree is represented by a pointer to the topmost node of the tree. If the tree is empty, then the value of the root is NULL.

Binary Tree node contains the following parts:

- Data
- Pointer to left child
- Pointer to right child

Binary Tree Operations

Basic Operation On Binary Tree:

- Inserting an element.
- Removing an element.
- Searching for an element.
- Traversing an element.

Auxiliary Operation On Binary Tree:

- Finding the height of the tree
- Find the level of the tree
- Finding the size of the entire tree.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm:

```
If root is NULL
  then create root node
return
If root exists then
  compare the data with node.data
  while until insertion position is located
    If data is greater than node.data
      goto right subtree
    else
      goto left subtree
  endwhile
  insert data
end If
```

C implementation : insertion in Binary Tree

```
void insert(int data) {
  struct node *tempNode = (struct
node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;
  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;
  //if tree is empty, create root node
  if(root == NULL) {
    root = tempNode;
  } else
  {
    current = root;
    parent  = NULL;

while(1) {
  parent = current;

  //go to left of the tree
  if(data < parent->data) {
    current = current->leftChild;

    //insert to the left
    if(current == NULL) {
      parent->leftChild =
tempNode;
      return;
    }
  }

//go to right of the tree
     else {
       current = current->rightChild;

       //insert to the right
       if(current == NULL) {
         parent->rightChild = tempNode;
         return;
       }
     }
   }
  }
}
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree.

Algorithm:

```
If root.data is equal to search.data
  return root
else
  while data not found
    If data is greater than node.data
      goto right subtree
    else
      goto left subtree
      If data found
      return node
  endwhile
  return data not found
end if
```

C implementation

```
struct node* search(int data)
{
  struct node *current = root;
  printf("Visiting elements: ");
while(current->data != data)
 {
    if(current != NULL)
    printf("%d ",current->data);
        //go to left tree
    if(current->data > data)
    {
      current = current->leftChild;
    }
//else go to right tree
    else {
      current = current->rightChild;
    }
    //not found
    if(current == NULL) {
      return NULL;
    }
  }
  return current;
```

Deletion of a node : Binary Tree

Given a binary tree, delete a node from it by making sure that the tree shrinks from the bottom (i.e. the deleted node is replaced by the bottom-most and rightmost node).

- ***Input :*** *Delete 10 in below tree*

```
   10
  /  \
20    30
```

Output:

```
     30
    /
  20
```

Input : *Delete 20 in below tree*

```
     10
    /  \
  20    30
          \
           40
```

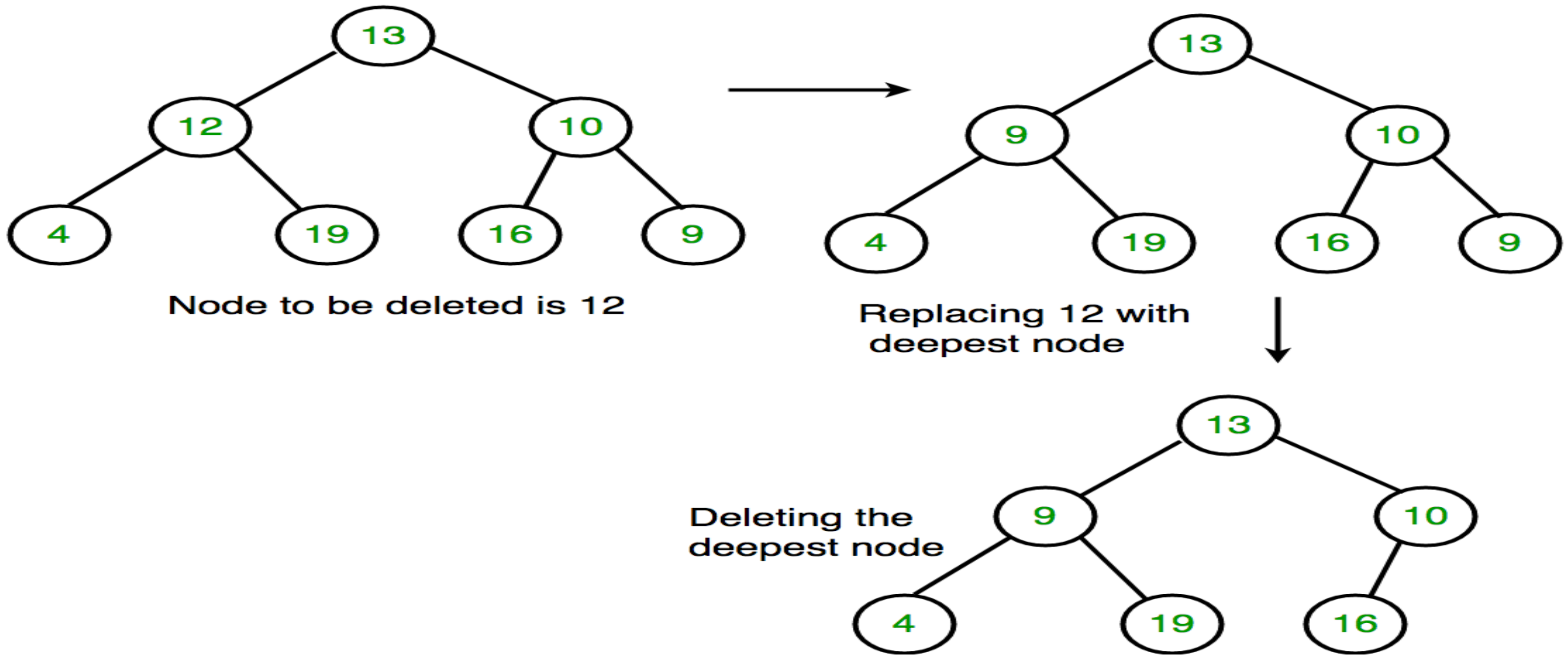
Output:

```
     10
   /    \
 40      30
```

Deletion : Algorithm

Algorithm:

- *Starting at the root, find the deepest and rightmost node in the binary tree and the node which we want to delete.*
- *Replace the deepest rightmost node's data with the node to be deleted.*
- *Then delete the deepest rightmost node.*



Traversing in Binary Tree

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

Traversing in Binary Tree: Inorder Traversal

- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order

Algorithm:

Step 1:Traverse the left most sub tree.

Step 2: Visit the root.

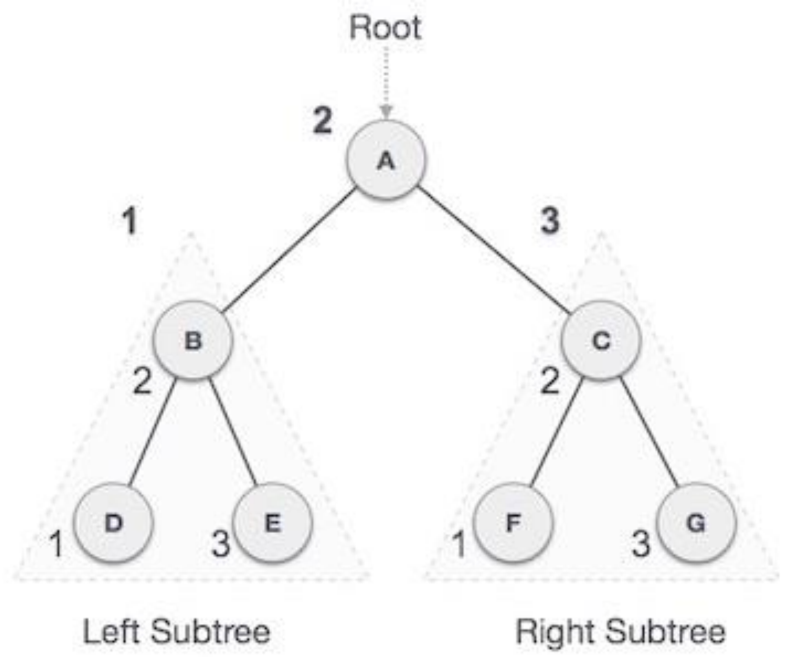
Step 3: Traverse the right most sub tree.

Note: Inorder traversal is also known as LNR traversal.

Pseudo code:

```
 inorder(t)
/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Inorder(t->lchild);
                Visit(t);
                Inorder(t->rchild);
        }
}
```

Example



We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

D → B → E → A → F → C → G

Traversing in Binary Tree: Preorder Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Algorithm:

Step 1: Visit the root.

Step 2: Traverse the left sub tree of root.

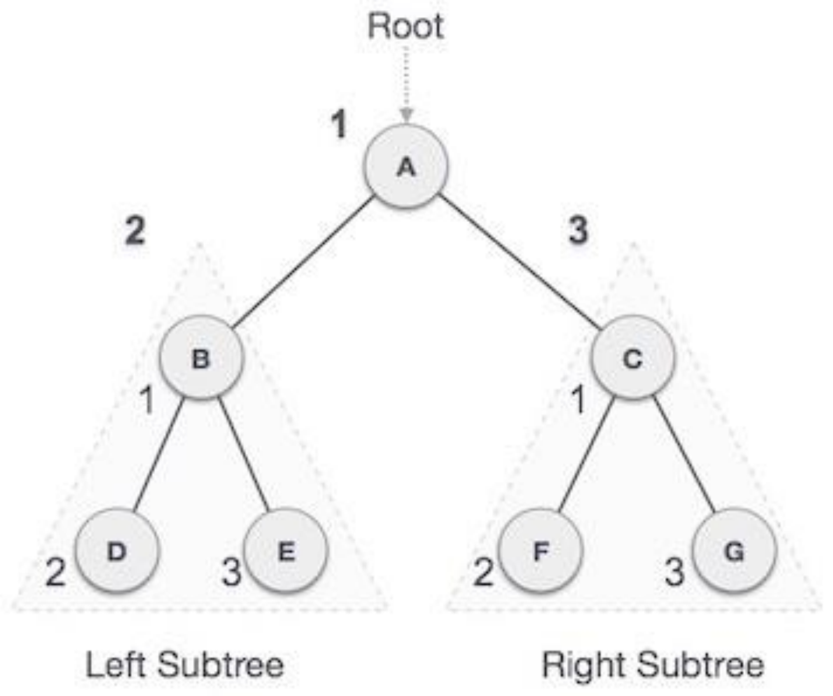
Step 3: Traverse the right sub tree of root.

- Note: Preorder traversal is also known as NLR traversal.

Pseudocode:

```
preorder(t)
/*t is a binary tree. Each node of t has
three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Visit(t);
                Preorder(t->lchild);
                Preorder(t->rchild);
        }
}
```

Example



We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Traversing in Binary Tree: Postorder Traversal

- In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

Algorithm:

Step 1: Traverse the left sub tree of root.

Step 2: Traverse the right sub tree of root.

Step 3: Visit the root.

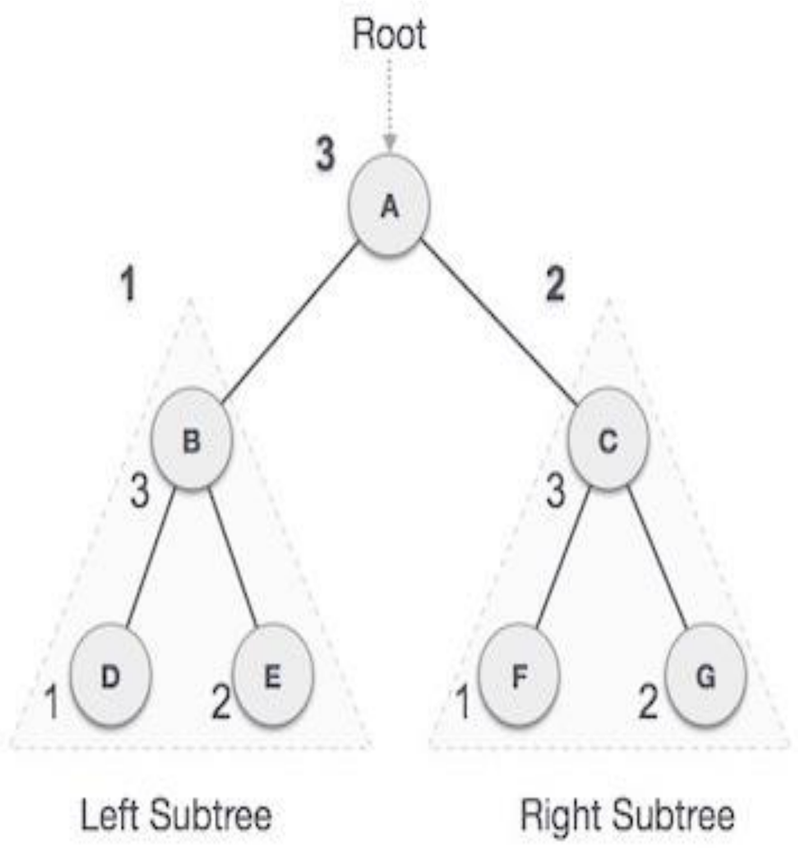
Note: Postorder traversal is also known as LRN traversal.

Pseudo code:

```
postorder(t)

/*t is a binary tree .Each node of t has three
fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Postorder(t->lchild);
                Postorder(t->rchild);
                Visit(t);
        }
}
```

Example



We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Creation of Binary tree using traversals

Here tree traversals output are given and we need to construct a binary tree. There are two different ways of creating binary tree.

1. Given Preorder and inorder traversals
2. Given Postorder and inorder traversals

Creation of binary tree from pre-order and inorder traversal:

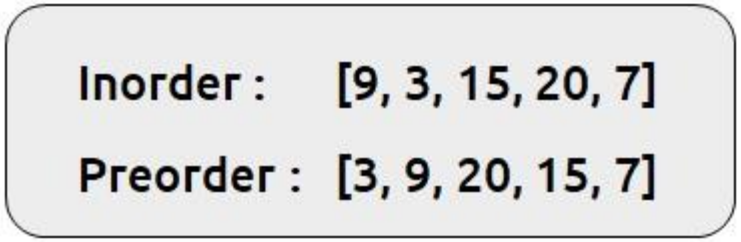
Step1: Scan the pre order traversal from left to right.

Step 2: For each node scanned, locate its position in inorder traversal. Let the scanned node be X.

Step 3: The node preceding X in inorder form its left sub tree and nodes succeeding it form right sub-tree.

Step 4: Repeat Step 1 for each symbol in the preorder

Example:

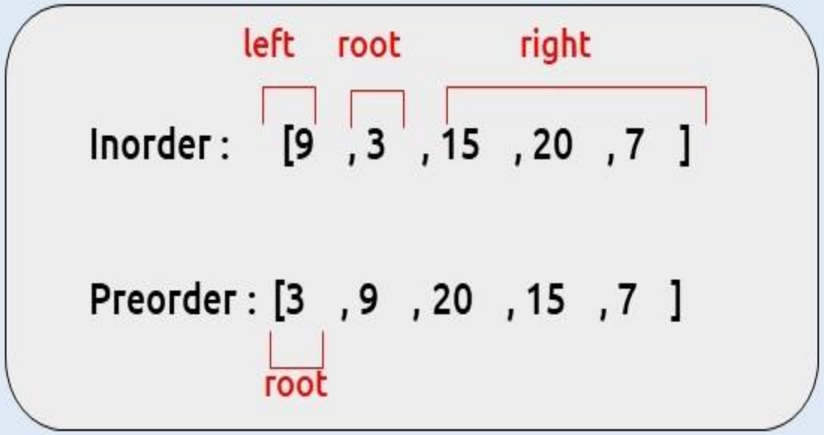


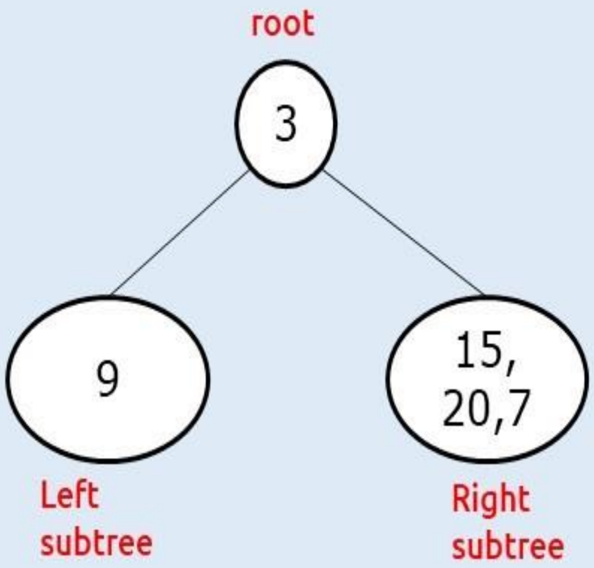
Inorder : [9, 3, 15, 20, 7]

Preorder : [3, 9, 20, 15, 7]

Step 1:

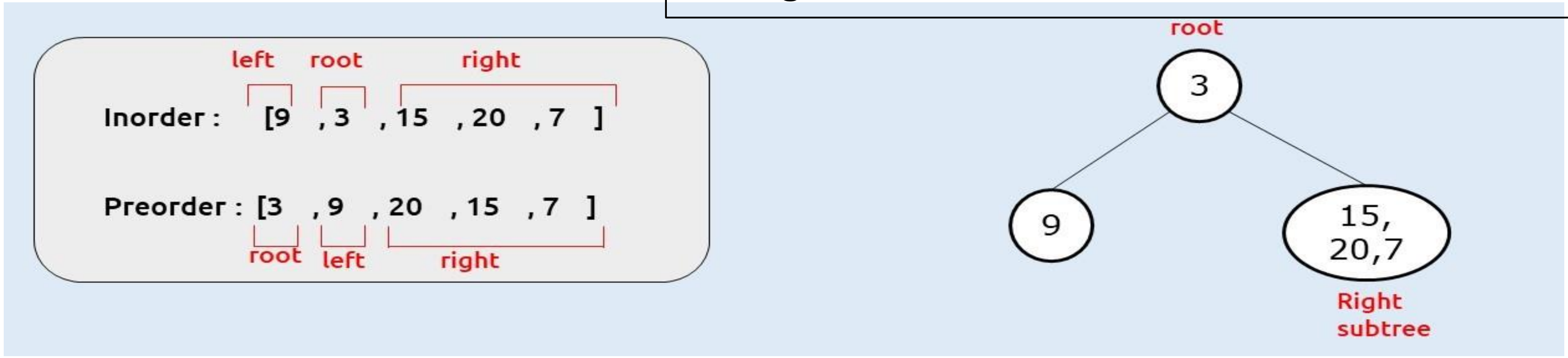
We know that the first element of a preorder traversal (root, left, right) is the root of the tree therefore, we can say that 3 is the root of the tree. We can then find 3 in the inorder traversal and find the left subtree and right subtree of node 3 from it as inorder traversal is (left, root, right).


left
root
right
Inorder : [9 , 3 , 15 , 20 , 7]
Preorder : [3 , 9 , 20 , 15 , 7]
root



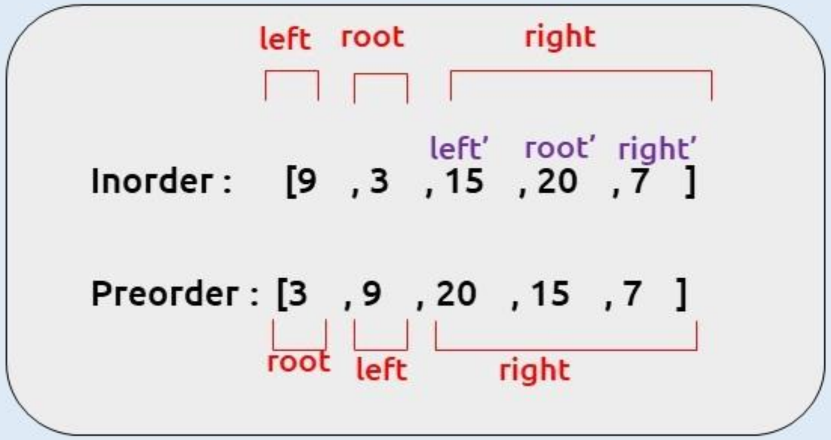
Step 2:

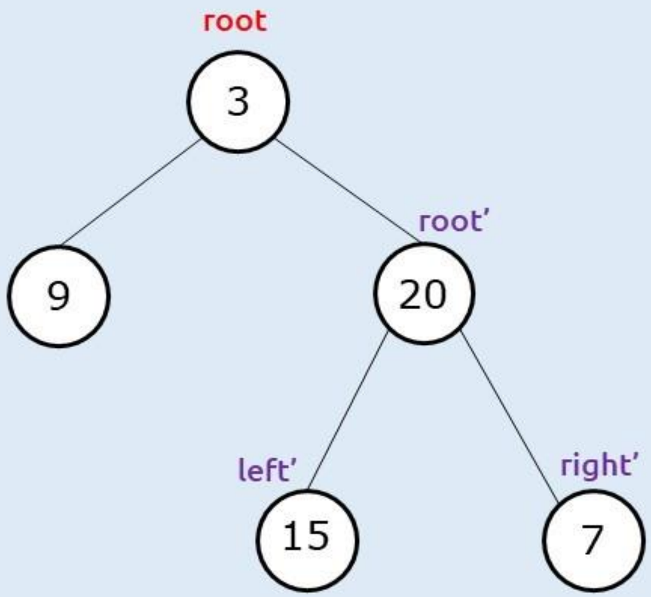
As 9 is the only element to the left of 3 in inorder traversal, we can conclude that the left subtree of node 3 contains a single node 9. Preorder traversal also contains node 9 as the second element. Therefore the remaining nodes in the preorder traversal will be the right subtree of node 3.


left
root
right
Inorder : [9 , 3 , 15 , 20 , 7]
Preorder : [3 , 9 , 20 , 15 , 7]
root
left
right
root
3
9
15, 20,7
Right subtree

Step 3:

Now, we will work on the right subtree of node 3. We know that is given by the last three elements of preorder traversal. As discussed in step 1, the first node will give us the root of the right subtree, then we can find that element (20) in the inorder traversal. Elements left to element 20 will give us the left subtree and elements right to it will give the right subtree of node 20.


left
root
right
left'
root'
right'
Inorder : [9 , 3 , 15 , 20 , 7]
Preorder : [3 , 9 , 20 , 15 , 7]
root
left
right



Example:

Inorder: DHBEAFCG

Preorder: ABDHECFG

Creation of binary tree from Post-order and inorder traversal:

Step1: Scan the post order traversal from right to left.

Step 2: For each node scanned, locate its position in inorder traversal. Let the scanned node be X.

Step 3: The node preceding X in inorder form its left sub tree and nodes succeeding it form right sub-tree.

Step 4: Repeat Step 1 for each symbol in the postorder

Example:

Post order:{8, 4, 5, 2, 6, 7, 3, 1}

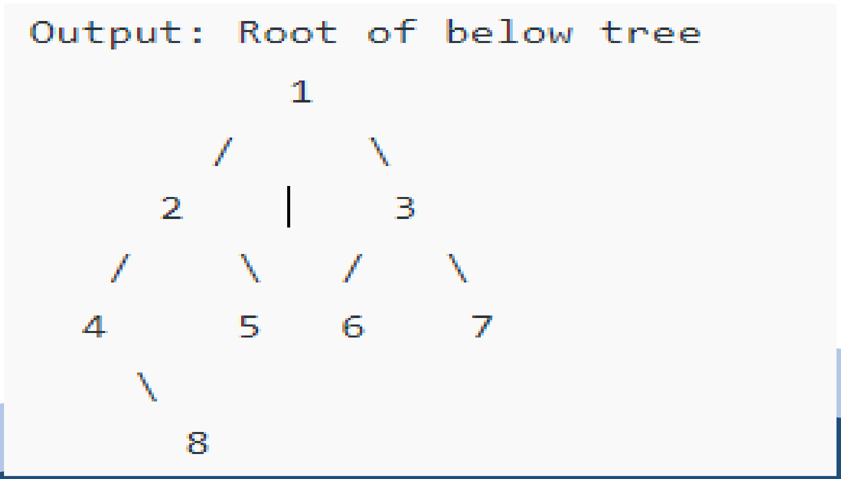
Inorder: {4, 8, 2, 5, 1, 6, 3, 7}

Let us see the process of constructing tree from inorder:{4, 8, 2, 5, 1, 6, 3, 7} and postorder {8, 4, 5, 2, 6, 7, 3, 1}

1) We first find the last node in postorder. The last node is "1", we know this value is root as the root always appears at the end of postorder traversal.

2) We search "1" in inorder to find the left and right subtrees of the root. Everything on the left of "1" in inorder is in the left subtree and everything on right is in the right subtree.

```
      1
    /   \
[4, 8, 2, 5]   [6, 3, 7]
```



```
Output: Root of below tree
                 1
              /     \
            2    |    3
          /    \   /    \
         4      5  6     7
          \
            8
```

Example 2:

Inorder: HDBIEAFJCKGL

Postorder: HDIEBJFKLGCA

Create a binary tree

Example 3: If the inorder traversal of a binary tree is BIDACGEHF and its preorder traversal is IDBGCHFEA. Determine its binary tree.

Converting Algebraic Expression into Binary Tree

- The arithmetic expressions represented as binary trees are known as **expression trees.** In expression trees, the root node is operator and the left and right children are operands.
- In case of unary operators, the left child is not present and the right child is the operand.
- The leaf nodes always denote the operands.
- The operations are always performed on these operands.
- The operator present in the depth of the tree is always at the highest priority.
- The operator, which is not much at the depth in the tree, is always at the lowest priority compared to the operators lying at the depth.
- The operand will always present at a depth of the tree; hence it is considered the **highest priority** among all the operators.

Example: Construct a binary tree (Expression tree) to represent the following expression.

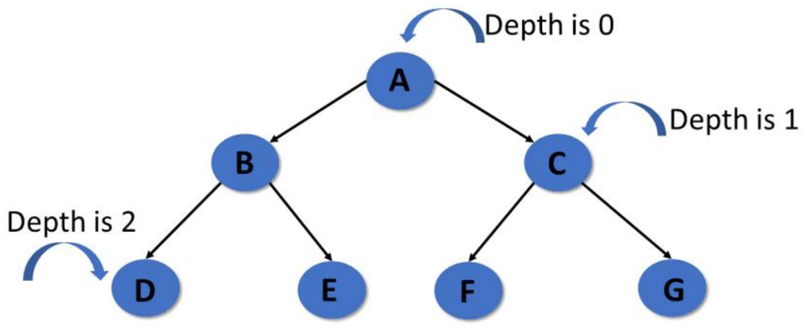
i) (5+4*(6-7))/(5+8)

ii) ((a*x+b)*x+e)*x+f

Tree level , height and depth

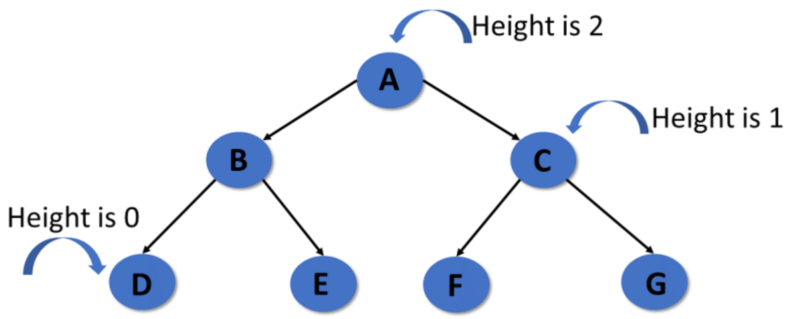
Depth: In a tree, many edges from the root node to the particular node are called the depth of the tree.

- In the tree, the total number of edges from the root node to the leaf node in the longest path is known as "Depth of Tree".
- In the tree data structures, the depth of the root node is 0.



Height: the number of edges from the leaf node to the particular node in the **longest path is known** as the height of that node.

- In the tree, the height of the root node is called "Height of Tree".
- The tree height of all leaf nodes is 0.



Given a Binary Tree consisting of **N nodes** and a integer **K**, the task is to find the depth and height of the node with value K in the Binary Tree.

Input: *K = 25,*

```
        5
      /    \
   10      15
 /   \    /   \
20   25  30   35
       \
        45
```

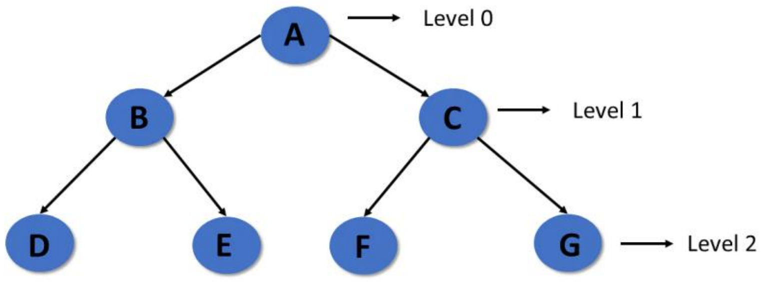
Output:
Depth of node 25 = 2
Height of node 25 = 1
Explanation:
The number of edges in the path from root node to the node 25 is 2. Therefore, depth of the node 25 is 2.
The number of edges in the longest path connecting the node 25 to any leaf node is 1. Therefore, height of the node 25 is 1.

Input: *K = 10,*

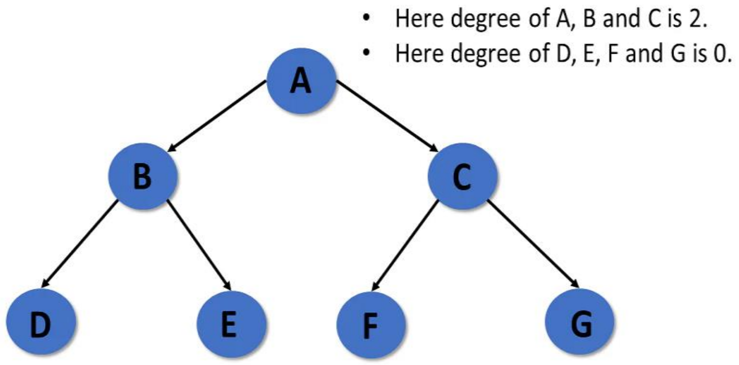
```
        5
      /    \
   10      15
 /   \    /   \
20   25  30   35
       \
        45
```

Output:
Depth of node 10 = 1
Height of node 10 = 2

Level: In tree data structures, the root node is said to be at level 0, and the root node's children are at level 1, and the children of that node at level 1 will be level 2, and so on.



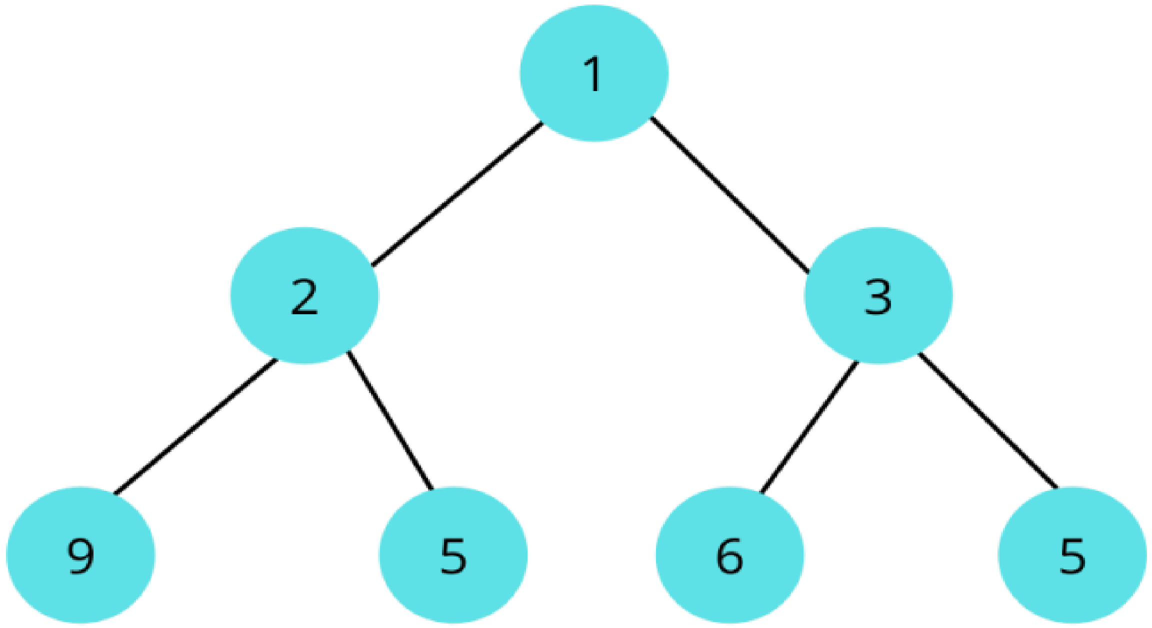
- **Degree:** In the tree data structure, the total number of children of a node is called the degree of the node.
- The highest degree of the node among all the nodes in a tree is called the Degree of Tree.


Here degree of A, B and C is 2.
Here degree of D, E, F and G is 0.
A
B
C
D
E
F
G

Balanced Trees

- Balanced Binary trees are computationally efficient to perform operations on.
- A balanced binary tree will follow the following conditions:
  - The absolute difference of heights of left and right subtrees at any node is less than 1.
  - For each node, its left subtree is a balanced binary tree.
  - For each node, its right subtree is a balanced binary tree.
- **Height-balanced Binary Trees:**
- Balanced binary trees are also known as height-balanced binary trees. Height balanced binary trees can be denoted by HB(k), where k is the difference between heights of left and right subtrees. ‘k’ is known as the balance factor.
- If for a tree, the balance factor (k) is equal to zero, then that tree is known as a fully balanced binary tree. It can be denoted as **HB(0)**.

Fully Balanced Binary Tree: Example



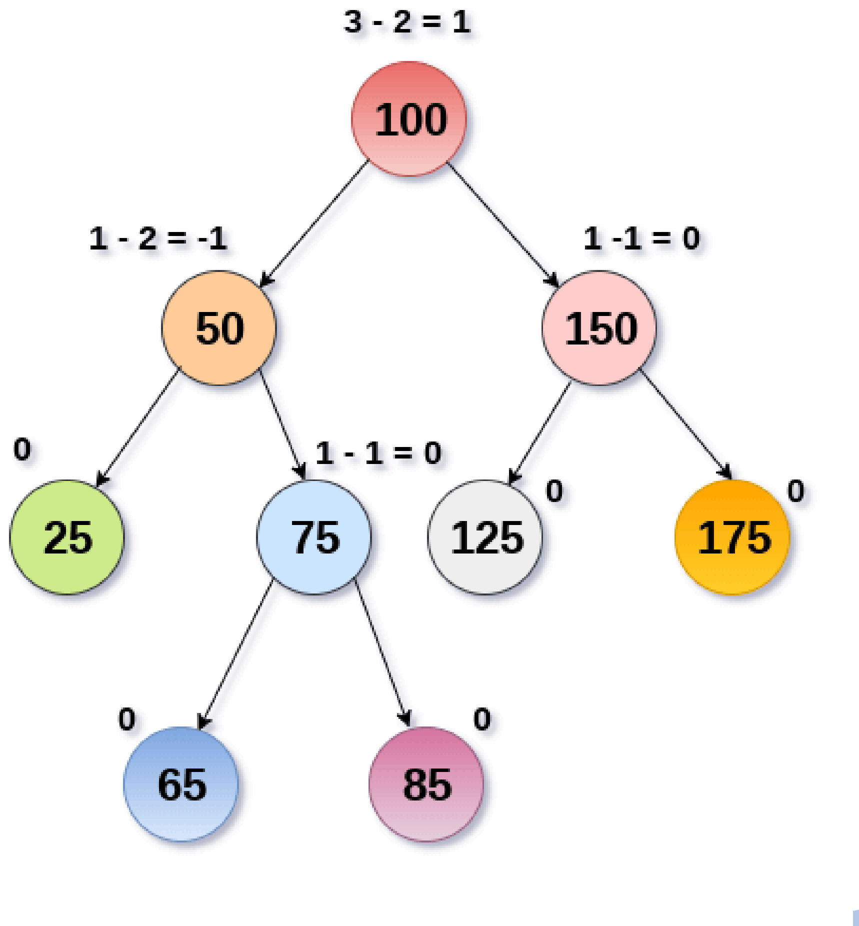
AVL Balanced Tree

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

Balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Complexity:

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

C implementation of AVL tree:

```
struct node
{
 int info;
int balance;
struct node *lchild;
struct node *rchild;
};
```

Operations on AVL tree

Insertion:

Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.

Deletion:

Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

AVL Rotations:

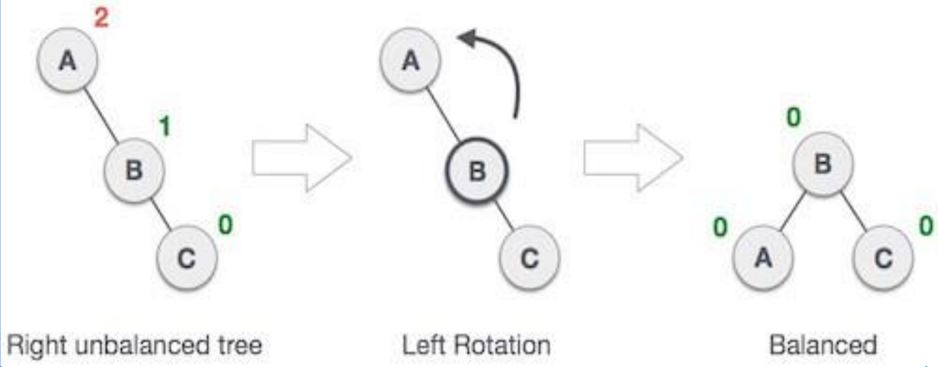
We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

- L L rotation: Inserted node is in the left subtree of left subtree of A
- R R rotation : Inserted node is in the right subtree of right subtree of A
- L R rotation : Inserted node is in the right subtree of left subtree of A
- R L rotation : Inserted node is in the left subtree of right subtree of A

- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

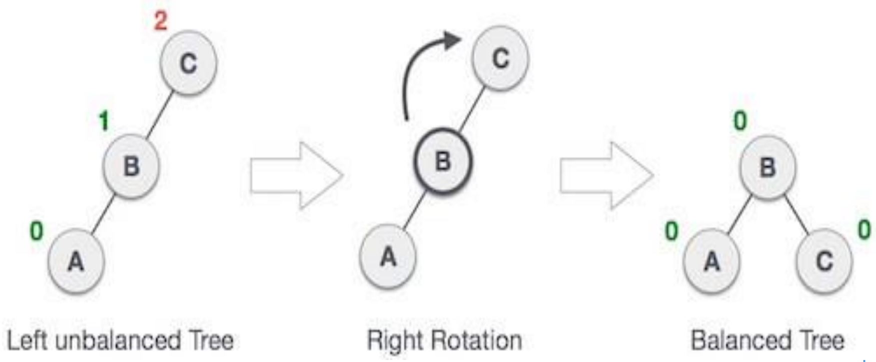
❑RR Rotation:

- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation
- is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



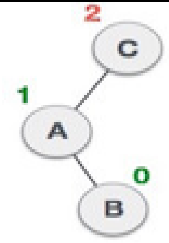
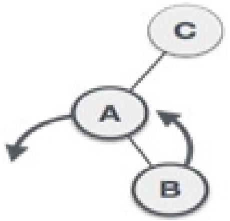
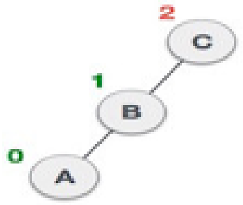
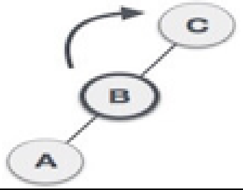
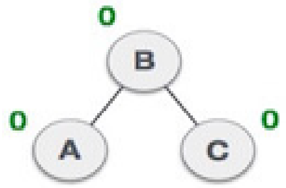
❑LL Rotation:

- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation
- is clockwise rotation, which is applied on the edge below a node having balance factor 2.



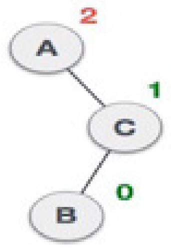
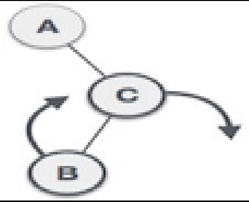
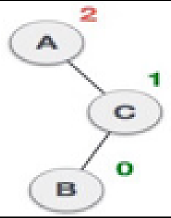
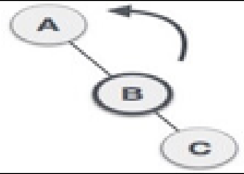
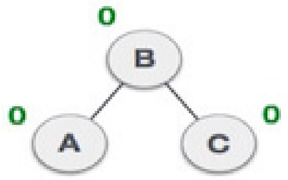
❑LR Rotation:

- LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

❑RL Rotation:

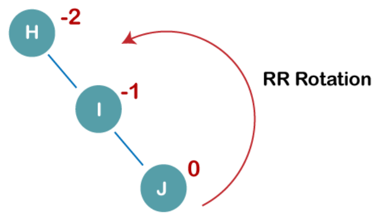
- RL rotation= LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

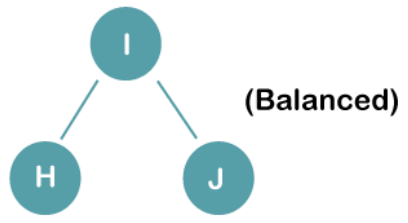
- Construct an AVL tree having the following elements:

  H, I, J, B, A, E, C, F, D, G, K, L

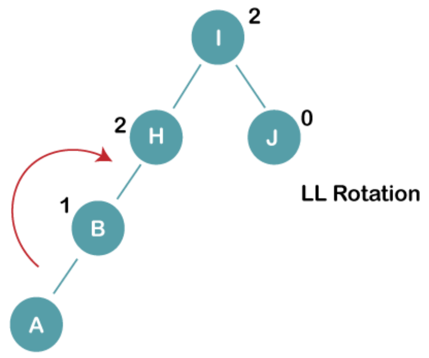
1. Insert H, I, J



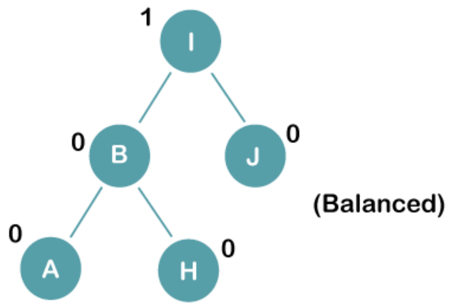
Resultant balanced tree is:



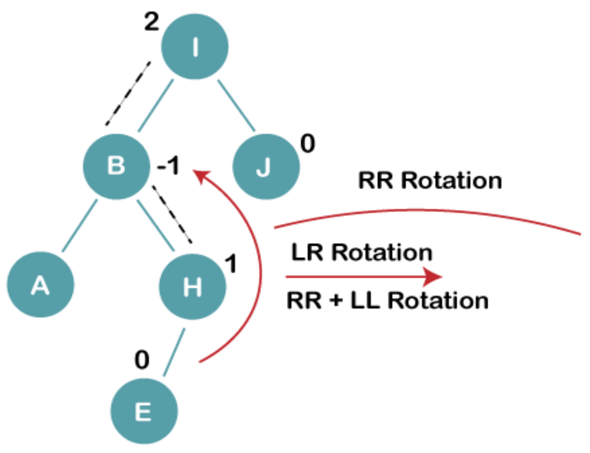
2. Insert B, A



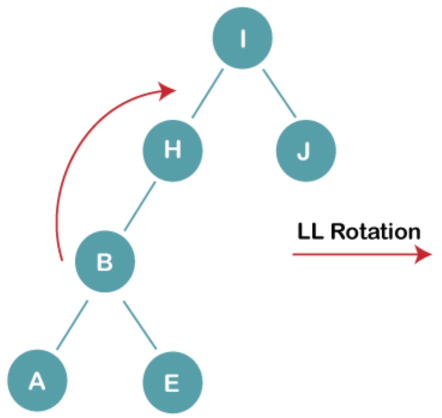
The resultant balanced tree is:



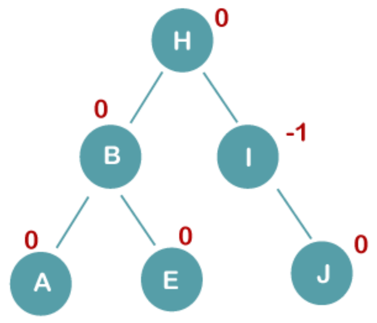
3. Insert E



i) Perform LL rotation on node B

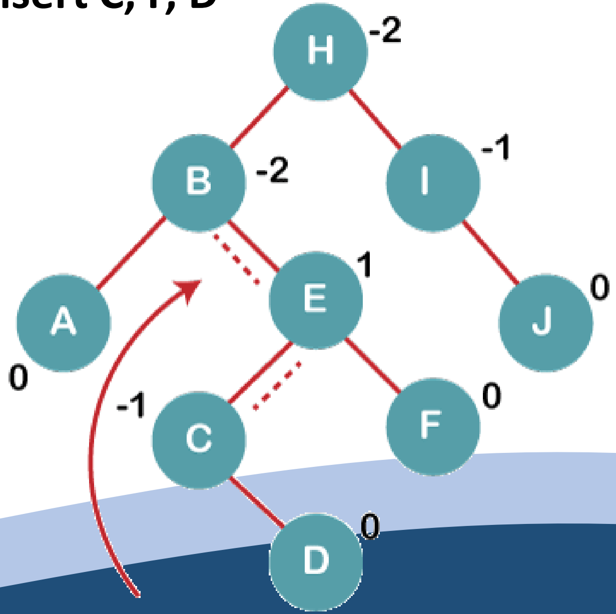


ii) First, perform LL rotation on node I

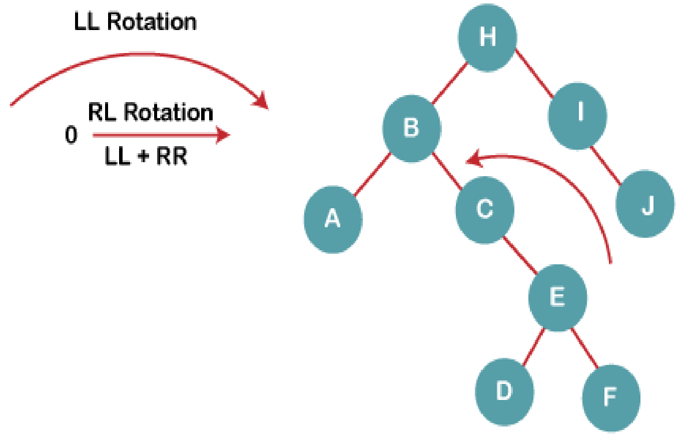


(Balanced)

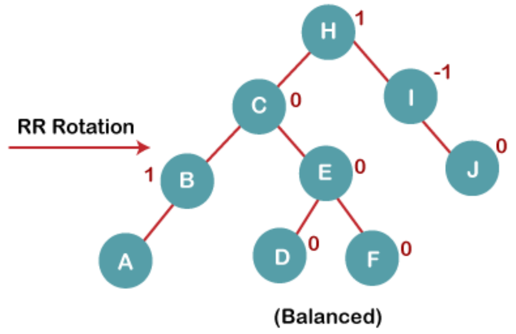
4. Insert C, F, D



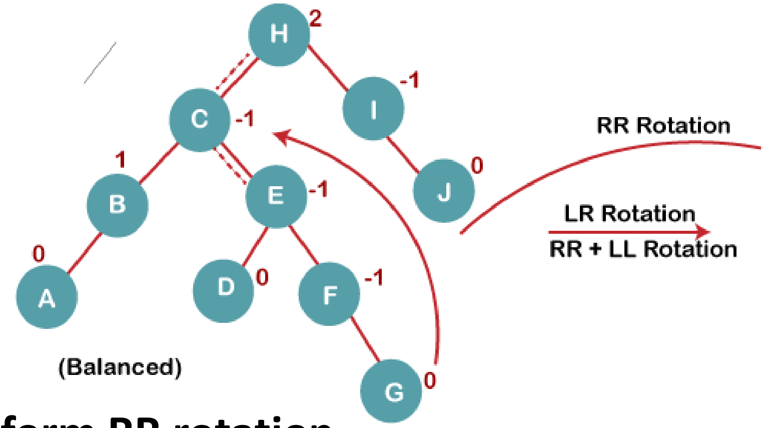
i) perform LL rotation on node E



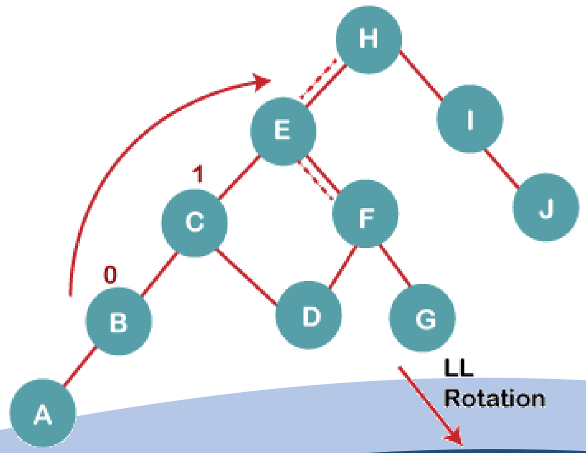
ii)Perform RR rotation on B



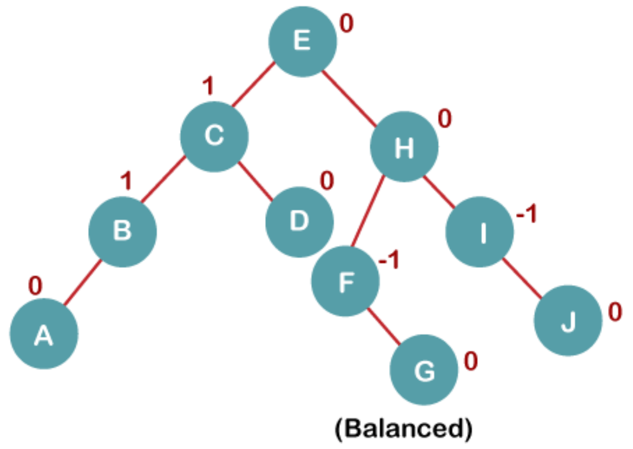
5. Insert G



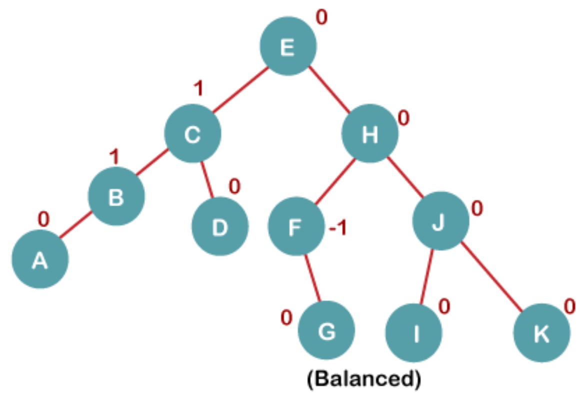
i) Perform RR rotation



5 ii) perform LL rotation on node H

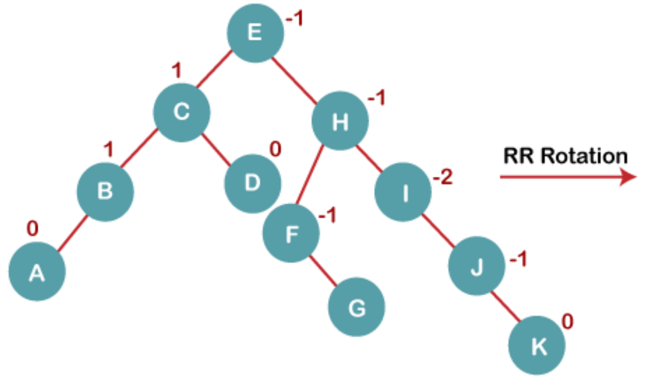


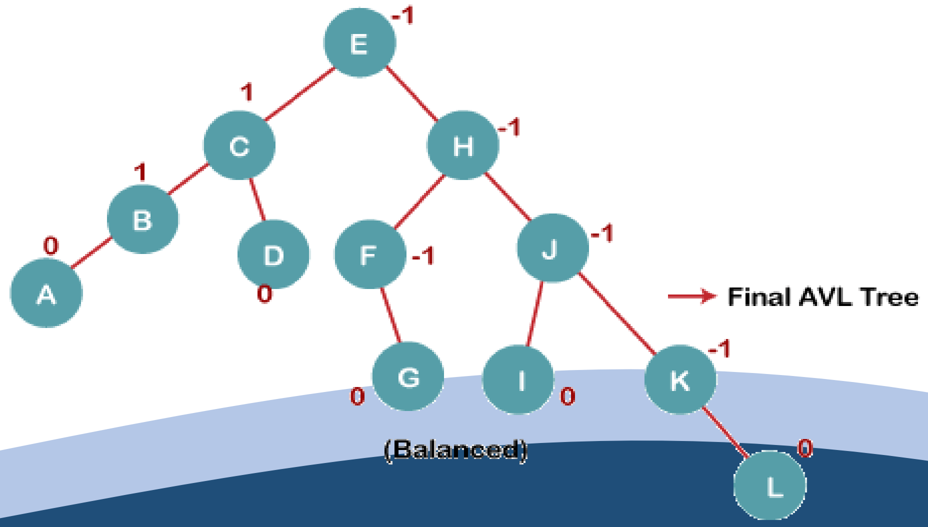
perform RR rotation on node I



7. Insert L

6.Insert K





Practice: Construct AVL tree for the insertion of following elements

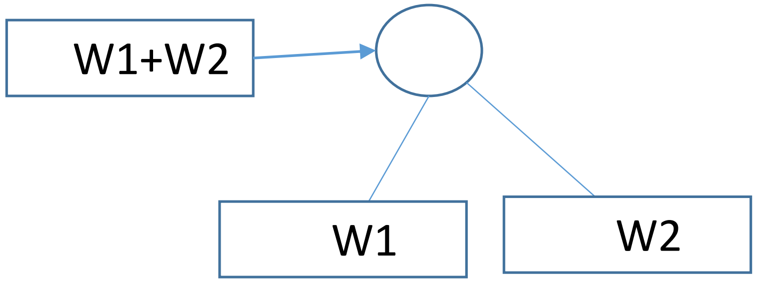
i) 2,26,1,20,3,30,25,15,12

ii) 1,26,5,24,8,10,3,7,20

Huffman Algorithm

Step 1: Suppose there are n weights, w1,w2,…wn

Step2: Take two minimum weights among the n given weights. Suppose w1,w2 are first two minimum weight then sub tree will be: w*=w1+w2



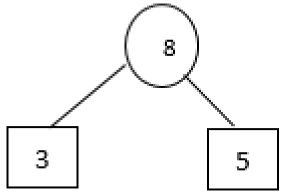
Step 3: Now the remaining weights will be: w*,w3,w4,….wn

Step 4: Create all sub tree at the last weight.

Example: Suppose A,B,C,D,E,F,G are elements with weights as follows: Construct Huffman tree.

item	A	B	C	D	E	F	G
weight	15	10	5	3	7	12	25

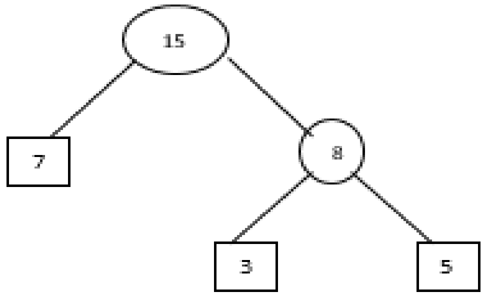
Step 1: Taking two nodes with minimum weights i.e, 3 and 5.



Now elements in the list are:

15,10,8,7,12,25

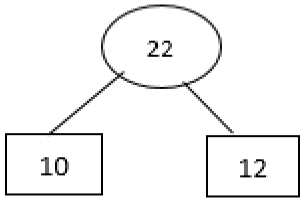
Step 2: Taking two minimum weight nodes which are 8 and 7.



Now elements in the list are:

15,10,15,12,25

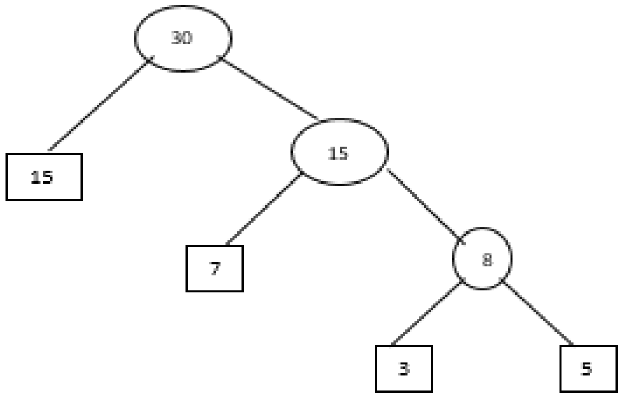
Step 3: Taking two minimum weight nodes which are 10 and 12.



Now elements in the list are:

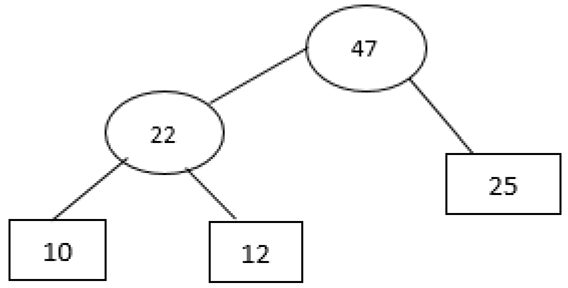
15,22,15,25

Step 4: Taking two minimum weight nodes which are 15 and 15.



Now elements in the list are: 30,22,25

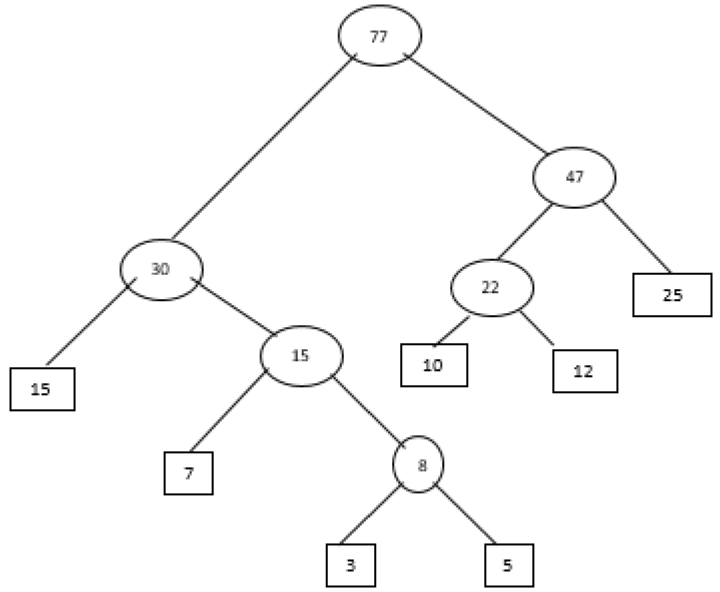
Step 5: Taking two minimum weights i.e, 22and 25.



Now elements in the list are:

30,47

Step 6: Taking two minimum weight nodes which are 30 and 47

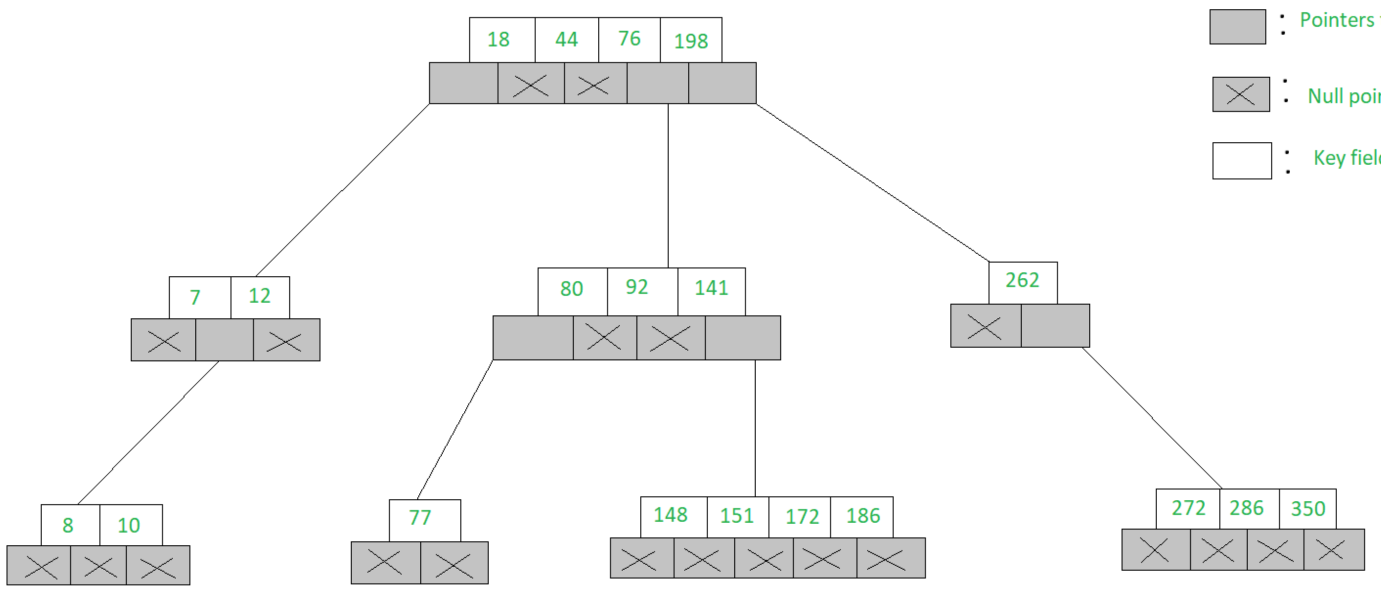


Practice: Construct a Huffman tree for ALLIANCE.

M way search tree

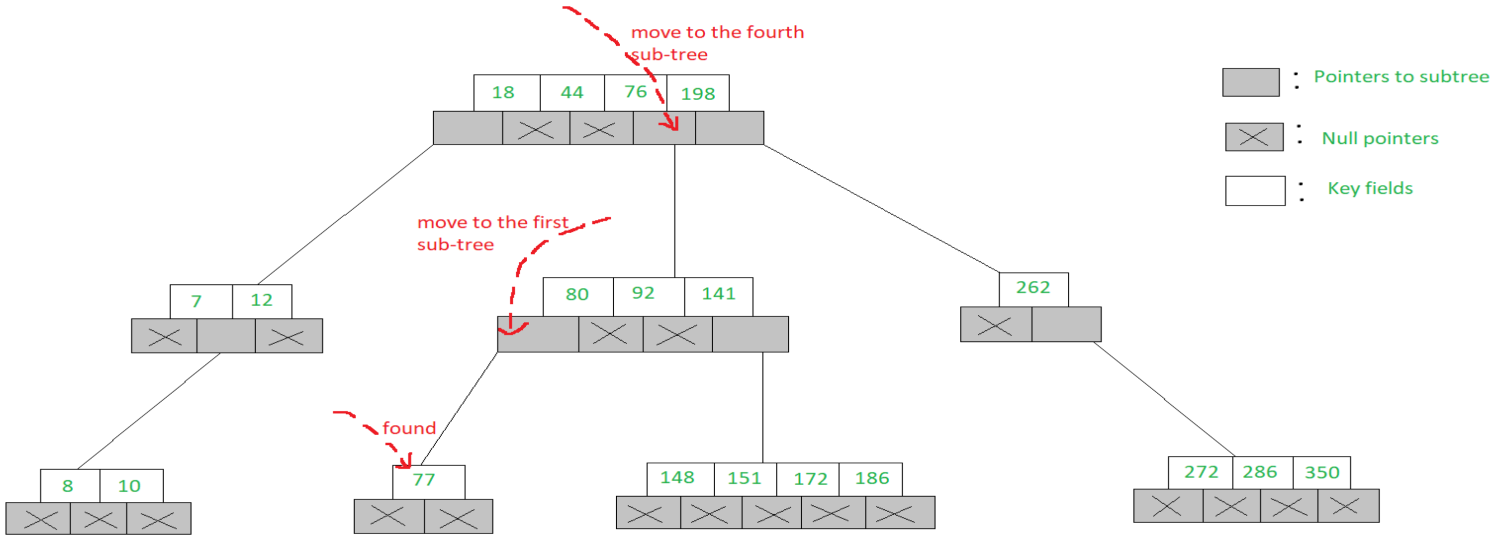
- The m-way search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple elements.
- In an m-Way tree of order m, each node contains a maximum of m – 1 elements and m children.
- The goal of m-Way search tree of height h calls for *O(h)* number of accesses for an insert/delete/retrieval operation. Hence, it ensures that the height h is close to *log_m(n + 1).*
- The number of elements in an m-Way search tree of height h ranges from a minimum of *h* to a maximum of m^h *-1* .
- An m-Way search tree of n elements ranges from a minimum height of log_m(n+1) to a maximum of n

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



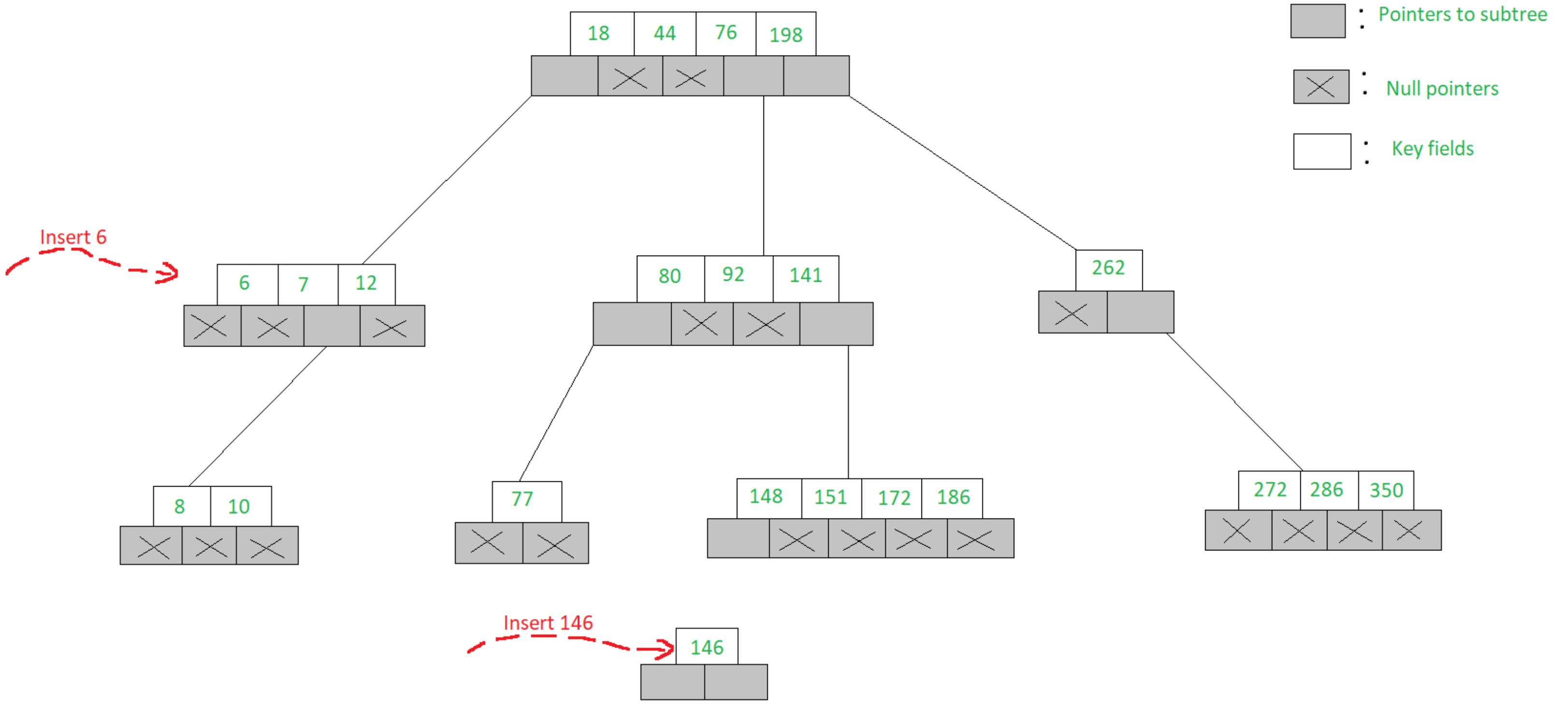
Searching in an m-Way search tree:

- 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.
  Let us see the example given below to insert an element in an m-Way search tree.
- To insert data 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



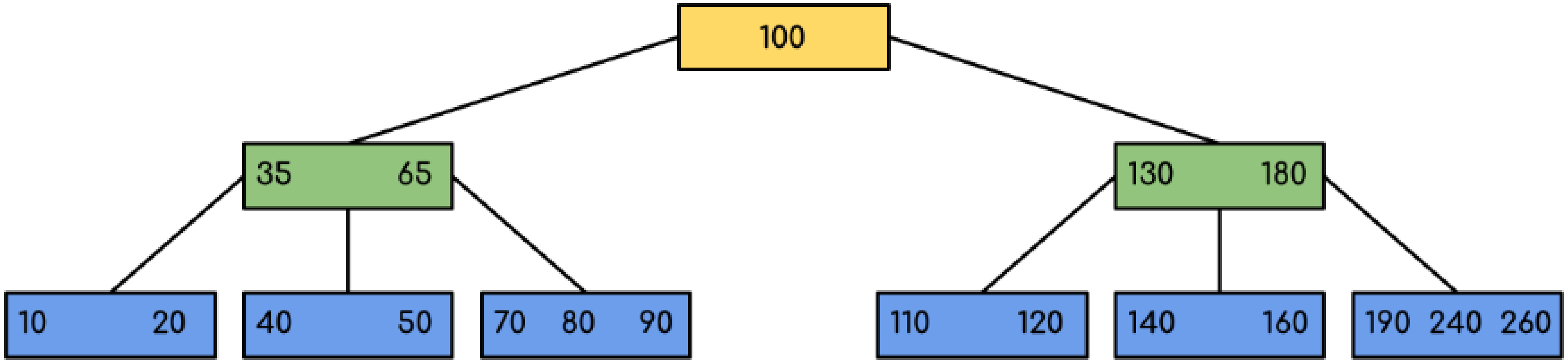
18
44
76
198
Pointers to subtree
Null pointers
Key fields
Insert 6
6
7
12
80
92
141
262
8
10
77
148
151
172
186
272
286
350
Insert 146
146

B Tree

- ❑B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

- A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.
  - Every node in a B-Tree contains at most m children.
  - Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
  - The root nodes must have at least 2 nodes.
  - All leaf nodes must be at the same level.
- It is not necessary that, all the nodes contain the same number of children but, each node must have **m/2** number of nodes.

example of a B-Tree of minimum order 5

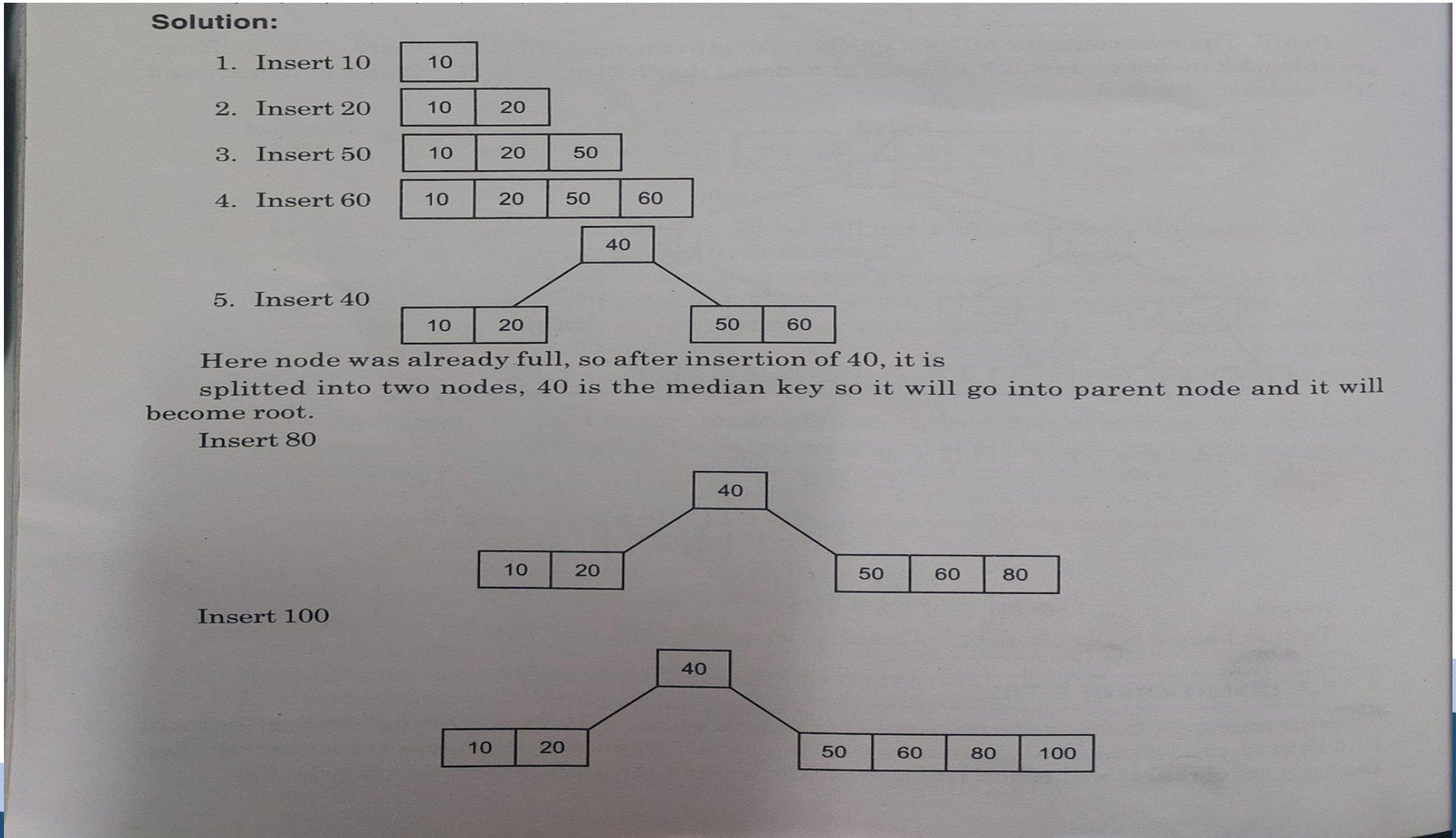


B-Tree: Insertion

- A new key is always inserted at the leaf node. Let the key to be inserted be k.
- Like BST, we start from the root and traverse down till we reach a leaf node.
- Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on the number of keys that a node can contain. So before inserting a key to the node, we make sure that the node has extra space.

Example: Create a B- tree of order 5 for the following keys:

10,20,50,60,40,80,100,70,130,90,30,120,140,25,35,160,180



Solution:

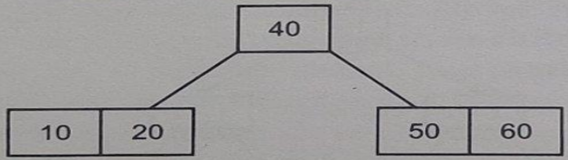
1. Insert 10 | 10 |

2. Insert 20 | 10 | 20 |

3. Insert 50 | 10 | 20 | 50 |

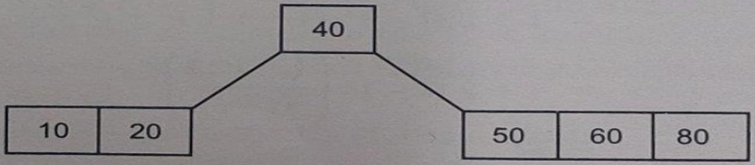
4. Insert 60 | 10 | 20 | 50 | 60 |

5. Insert 40

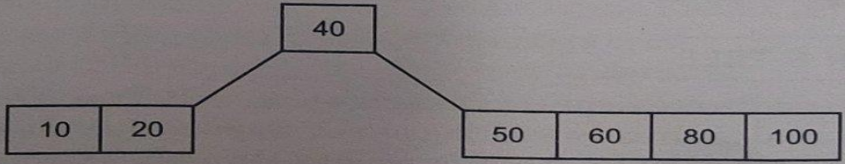


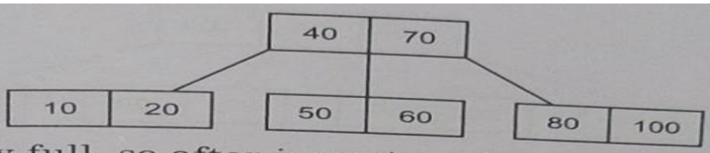
Here node was already full, so after insertion of 40, it is splitted into two nodes, 40 is the median key so it will go into parent node and it will become root.

Insert 80



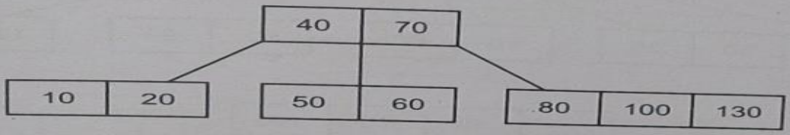
Insert 100



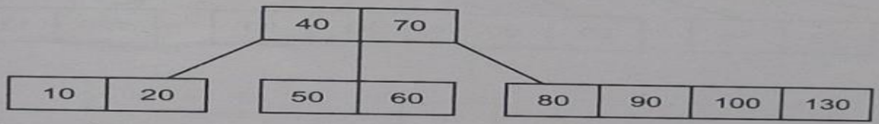


Here node was already full, so after insertion of 70 it splitted into two nodes, 70 is the median key, so it will go to the parent node.

Insert 130

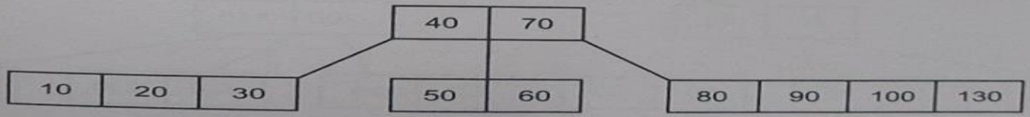


Insert 90

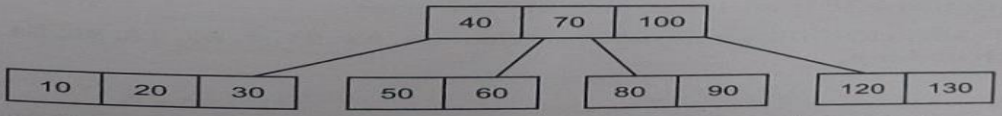


Here after insertion of 90, the keys in node will be sorted.

Insert 30

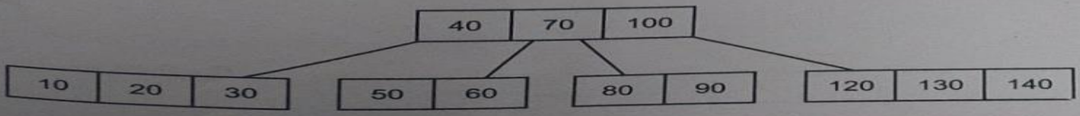


Insert 120

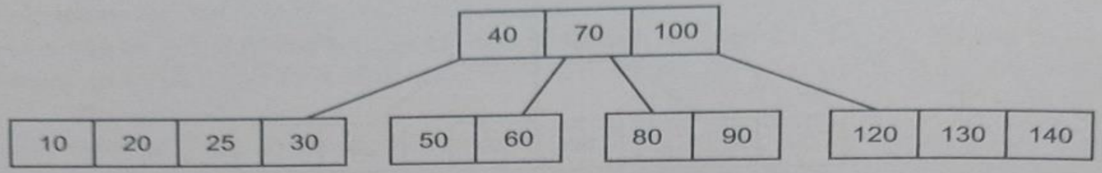


Here node was already full, so after insertion of 120 it splitted in two nodes, 100 is the median key so it will go into parent.

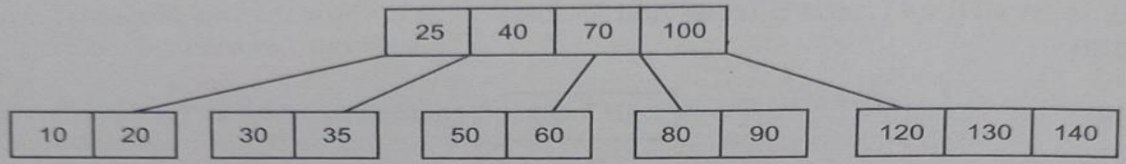
Insert 140



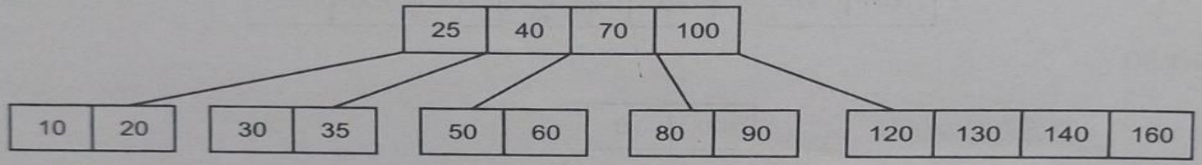
Insert 25



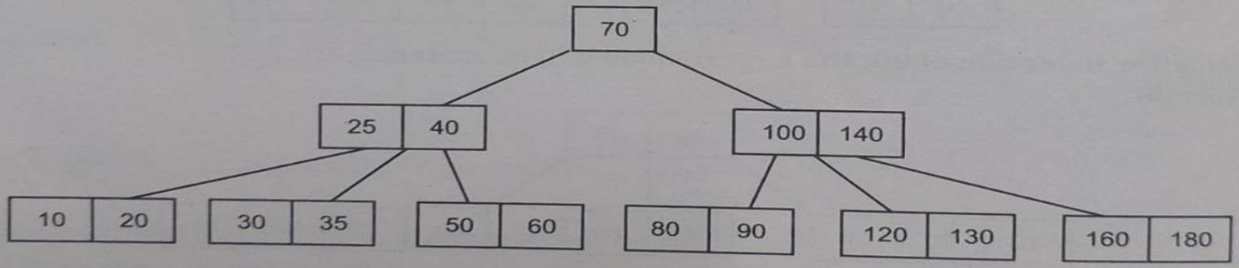
Insert 35



Insert 160



Insert 180



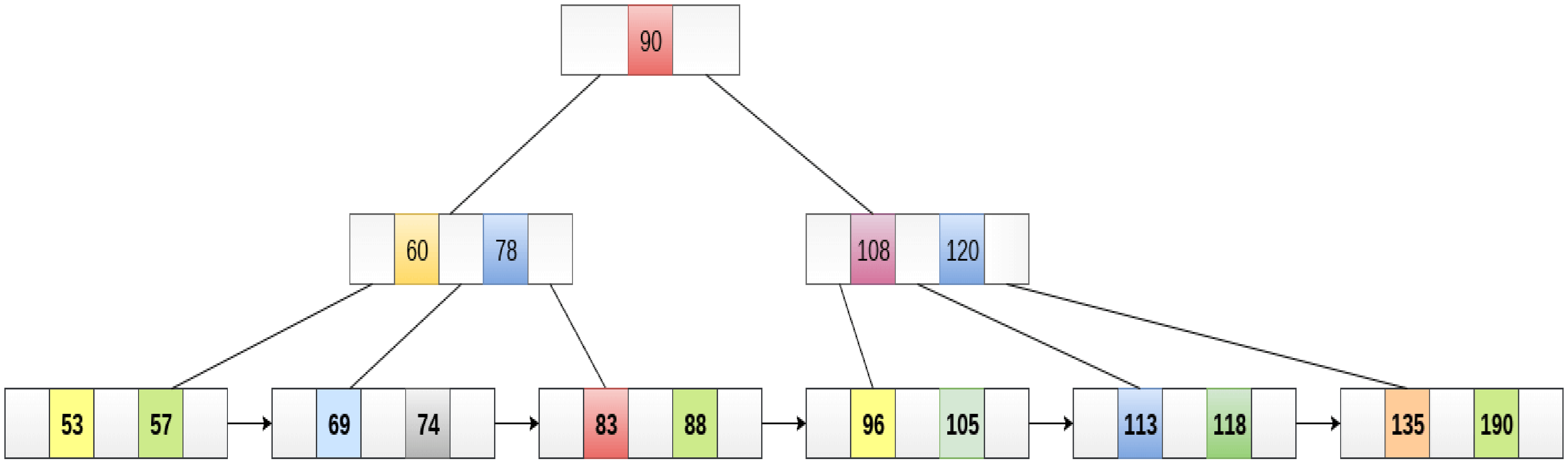
Here node was already full, so after insertion 180 it splitted in two nodes. 140 is the median key so it will go into parent node. Here parent is already full, so it splitted in two nodes and 70 is the median key so it will become the new root.

Practice: insert the following information into an empty B- tree with order 4.

B+ Tree

- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.
- B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

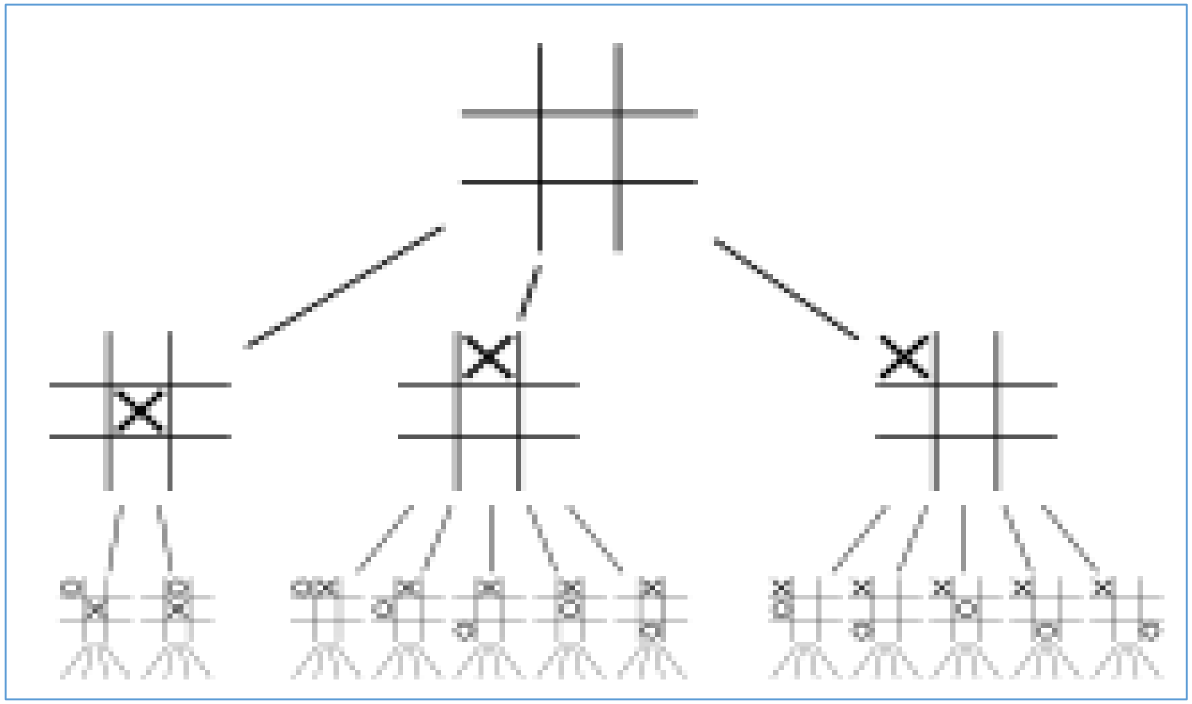
The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Game Tree

- A **game tree** is a type of recursive search function that examines all possible moves of a strategy game, and their results, in an attempt to ascertain the optimal move.
- They are very useful for Artificial Intelligence in scenarios that do not require real-time decision making and have a relatively low number of possible choices per play. The most commonly-cited example is chess, but they are applicable to many situations.
- Game trees are generally used in board games to determine the best possible move **Tic-Tac-Toe** as an example.
- The idea is to start at the current board position, and check all the possible moves the computer can make. Then, from each of those possible moves, to look at what moves the opponent may make. Then to look back at the computer's. Ideally, the computer will flip back and forth, making moves for itself and its opponent, until the game's completion. It will do this for every possible outcome (see image below), effectively playing thousands (often more) of games. From the winners and losers of these games, it tries to determine the outcome that gives it the best chance of success.
- This can be likened to how people think during a board game - for example "if I make this move, they could make this one, and I could counter with this" etc. Game trees do just that. They examine every possible move and every opponent move, and then try to see if they are winning after as many moves as they can think through.

Figure: Game tree for tic- tac- toe game



Any Query?