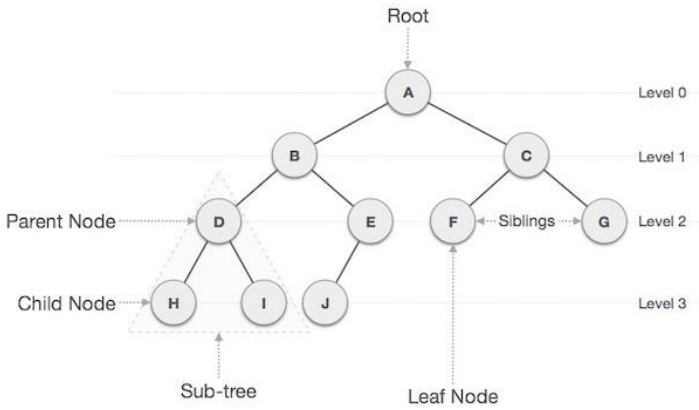
CHAPTER 6 TREE

Definition:

A tree is a nonlinear data structure in which items are arranged in a sorted Sequence, It is used to represent hierarchical relationship existing among several data items. Each node of a tree may or may not pointing more than one node.

It is a finite set of one or more data items (nodes) such that there is a special data item called the root. Its remaining data items are portioned into no. of mutually exclusive subsets, each of which is itself a tree and they are called subtree.

It represents the nodes connected by edges:



Tree terminology:

Root: - The first node in a hierarchical arrangement of data items is root.

Node: - Each data item in a tree is called a node.

Degree of a node: - It is the no. of subtrees of a node in a given tree from fig. Here,

- Degree of node A = 2
- Degree of node D = 2
- Degree of node F = 0

Degree of a tree: - It is the maximum degree of nodes in a given tree here degree of tree is 2.

Terminal node / Leaf node: - A node with degree 0 is terminal node.

Non terminal node / Parent Node: - Any node except the root whose degree is not zero is call non terminal node.

Keys: - it represents a value of a node based on which a search operation is to be carried out for a node

Siblings: - The children nodes of a given parent nodes are called siblings. F and G are siblings of parent node C & so on.

CHAPTER 6 TREE

Edge: - It is a connecting line of two nodes i.e. line drawn from one node to another.

Path: - It is a sequence of consecutive edge from the source node to the destination node.

Traversing: -It means passing through the nodes in certain order.

Level: - The entire tree structure is leveled in such a way that the root rode is always at level zero. Then its immediate children are at level 1 and there immediate children are at level 2 and so on up to the terminal nodes. Here, the level of a tree is 3.

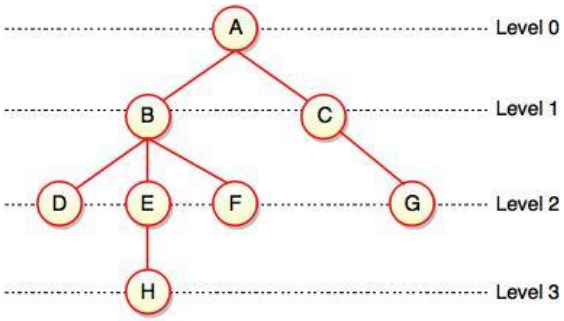
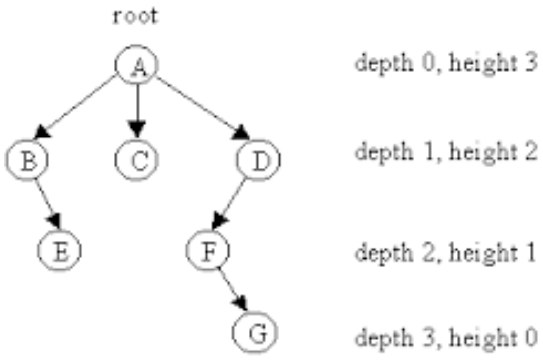


Fig. Levels of Tree

Depth or height: - Height of a node represents the number of edges on the longest path between that node and the leaf node. Depth of a node represents the number of edges from the tree's root node to the node. It is the maximum label of any node in a given tree here, the depth of a tree is 3.



A tree of height 3

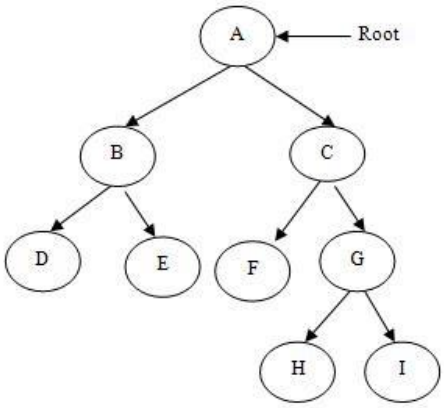
Binary tree:

A binary tree is a finite set of data items which is either empty or consist of a single item called the root and two disjoint binary trees the left subtree and right subtree. In binary tree the maximum degree of any node is at most 2 i.e. each node having 0, 1, or 2 degree node.

CHAPTER 6 TREE

Strictly Binary tree:

- Strictly binary tree is a tree where every node other than the leaves has the two children or, all of the nodes in a strictly binary tree are of degree zero or two, never degree one.
- A strictly binary tree with n leaves always contains 2n -1 nodes.

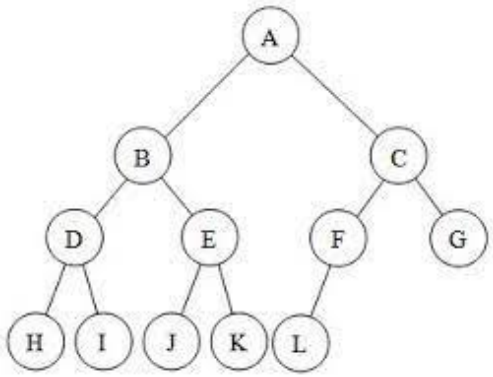


For 1 leaf no. of nodes =1
For 2 leaf no. of nodes =3
For 3 leaf no. of nodes =5
.
.
For n leaf no. of nodes =1+ (n-1)*2
=2n-1

$\because$ *The* n^{th} *term of AP is*
$= a + (n-1)d$

Almost completely Binary Tree:

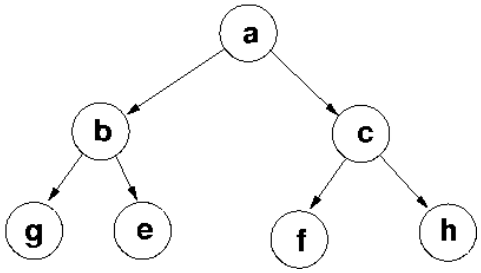
- A Binary Tree is almost complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible. A binary tree of depth d is an almost binary tree if:
  - Any node '*nd*' at level less than'*d-1'* has two sons.
  - For any node *'nd'* in the tree with a right descendant at level *d, nd* must have a left son and every left descendant of *nd* is either a leaf at level *d* or has two sons.



CHAPTER 6 TREE

Complete /Perfect Binary Tree:

- Every node except the leaf nodes have two children and every level (last level too) is completely filled. If 'm' nodes are present at level 'k' then there are '2m' nodes at level 'k+1'. There are 2^k nodes at level 'k'.
- If 'h' be the height, then total no. of leaves $\mathbf{2^h}$, and total no. of nodes $\mathbf{2^{h+1}-1}$



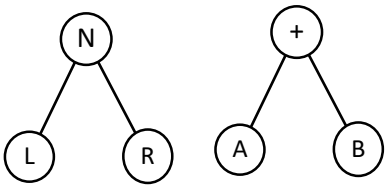
Binary Tree Traversal:

Traversal is a process to visit all the nodes of a tree and may print their values too. Unlike linear data structures (Array, Linked List, Queues, Stacks, etc.) which have only one logical way to traverse them, trees can be traversed in different ways. There are two types of traversal.

I. Depth-First Traversal:

Visit nodes in order of increasing depth. It starts with the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. While traversing a tree root is denoted by 'N' left subtree as 'L' and right subtree as 'R'. Following are the generally used ways for traversing trees.

- **In-order Traversal: LNR**
  1. Traverse the left subtree in Inorder
  2. Visit the root.
  3. Traverse the right subtree in Inorder



E.g. A + B

Algorithm:

1. ptr=root
2. Inorder (ptr);
3. If (ptr != NULL)
   a. Inorder (ptr->left);
   b. Print " ptr->info";
   c. Inorder (ptr->right);

- **Pre-order Traversal: NLR**
  1. Visit the root.
  2. Traverse the left subtree in Preorder
  3. Traverse the right subtree in Preorder

  E.g. + AB

  Algorithm:

  1. ptr=root
  2. Preorder (ptr);
  3. If (ptr != NULL)
     a. Print " ptr->info";
     b. Preorder (ptr->left);
     c. Preorder (ptr->right);
- **Post-order Traversal: LRN**
  1. Traverse the left subtree in Postorder
  2. Traverse the right subtree in Postorder
  3. Visit the root.

  E.g. AB+

  Algorithm:

  1. ptr=root
  2. Postorder (ptr);
  3. If (ptr != NULL)
     a. Postorder (ptr->left);
     b. Postorder (ptr->right);
     c. Print " ptr->info";

CHAPTER 6 TREE

Constructing the tree from preorder and in order traversal:

- In pre order traversal, scan the nodes one by one and keep them inserting in tree.
- In order traversal, put the cross mark over the node which has been inserted.
- To insert a node in its proper position in the tree we look at that node the in order traversal & insert it according to its position with respect to the crossed nodes:-

Inorder: $EACKFHDBG$ (LNR)

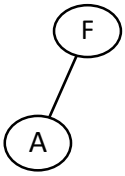
Preorder: $FAEKCDHGB$ (NLR)

Insert F

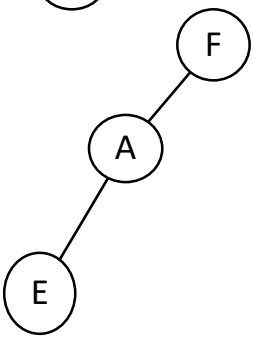
Inorder: EACK**F**HDBG (LNR)

Preorder: $FAEKCDHGB$ (NLR)

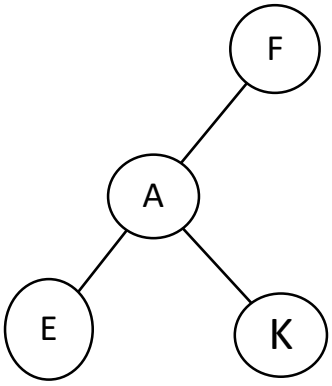
Insert A



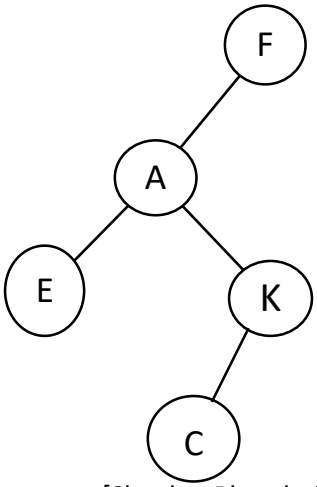
Insert E



Insert K

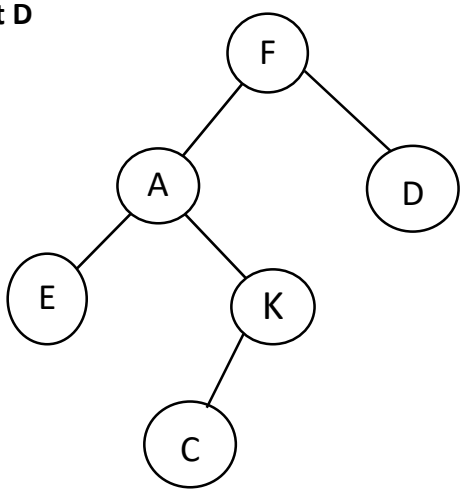


Insert C

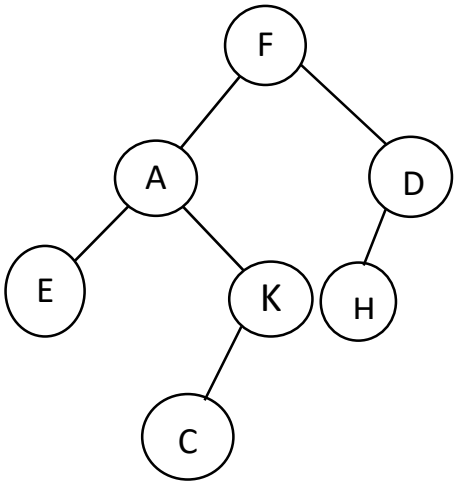


CHAPTER 6 TREE

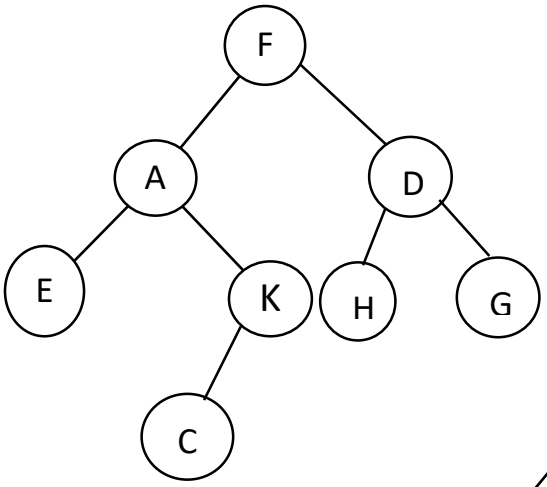
Insert D



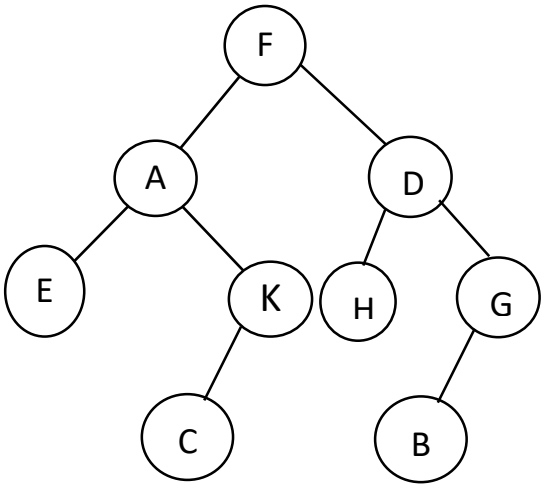
Insert H



Insert G



Insert B



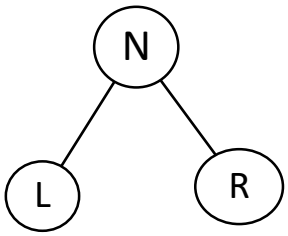
CHAPTER 6 TREE

Binary search tree: -

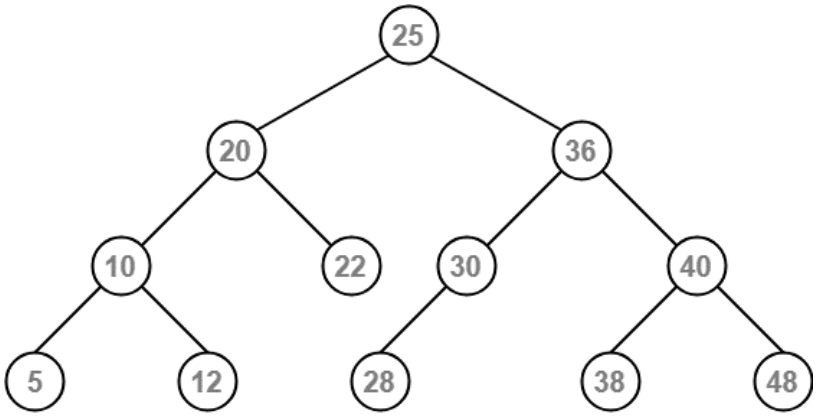
A binary search tree is a node-based binary tree in which each node has value greater than every node of left sub tree and less than every node of right sub tree which has the following properties:

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

There must not be duplicate nodes and R > N > L



E.g.



Algorithm for inserting a new node in BST:

Insert function is used to add a new element in a binary search tree at appropriate location.

1. Allocate memory for node

```
newnode = (struct node *) malloc(sizeof (struct node));
```

2. Set the data part to the value and set the left and right pointer of node, point to NULL.

```
newnode -> info = data;
newnode -> left = NULL;
newnode -> right = NULL;
```

3. Assign pointer 'tree' as 'root' node of the tree.

```
tree = root;
```

4. If (tree == NULL)

   Set `tree = newnode;`

5. Else if (data < tree -> info)
   i. If(tree -> left == NULL) then

   ```
   tree -> left = newnode;
   ```

   ii. else

   ```
   tree = tree -> left;
   ```

   iii. Repeat step from 4
6. Else if (data > tree -> info)
   i. If (tree -> right == NULL) then

   ```
   tree -> right = newnode;
   ```

   ii. Else

   ```
   tree = tree -> right;
   ```

   iii. Repeat step from 4
7. Else if (data == tree->info)
8. Display " Duplicate Data"
9. End

Example 1:

Construct a Binary Search Tree (BST) for the following sequence of numbers-

50, 70, 60, 20, 90, 10, 40, 100

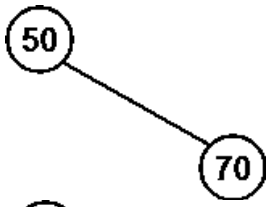
When elements are given in a sequence,

- Always consider the first element as the root node.
- Consider the given elements and insert them in the BST one by one.

Insert: 50

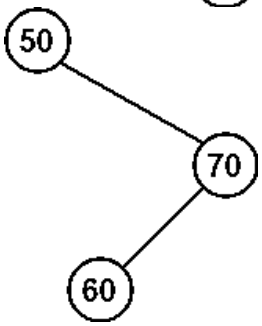
Insert: 70

As 70 > 50, so insert 70 to the right of 50.



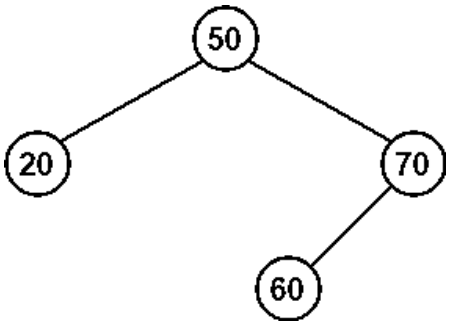
Insert: 60

As 60 > 50, so insert 60 to the right of 50.
As 60 < 70, so insert 60 to the left of 70.



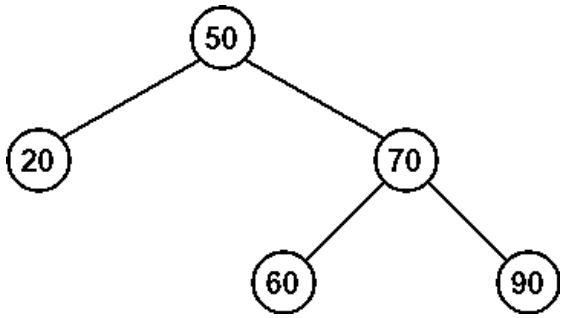
Insert: 20

As 20 < 50, so insert 20 to the left of 50.



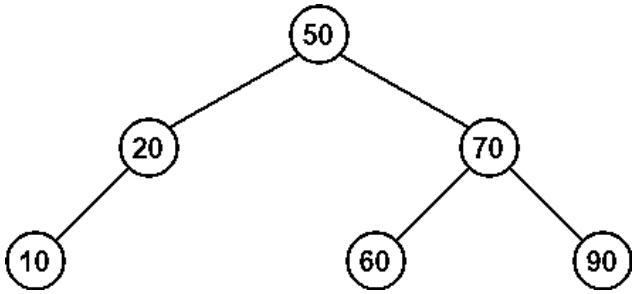
Insert: 90

As 90 > 50, so insert 90 to the right of 50.
As 90 > 70, so insert 90 to the right of 70.



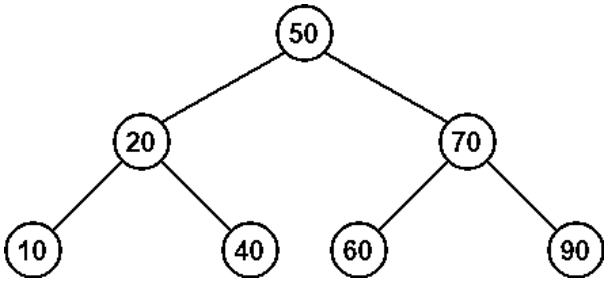
Insert: 10

As 10 < 50, so insert 10 to the left of 50.
As 10 < 20, so insert 10 to the left of 20.



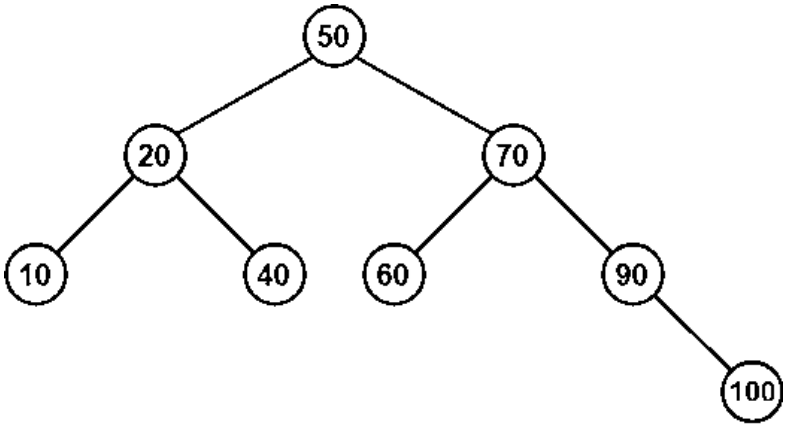
Insert: 40

As 40 < 50, so insert 40 to the left of 50.
As 40 > 20, so insert 40 to the right of 20.



Insert: 100

As 100 > 50, so insert 100 to the right of 50.
As 100 > 70, so insert 100 to the right of 70.
As 100 > 90, so insert 100 to the right of 90.

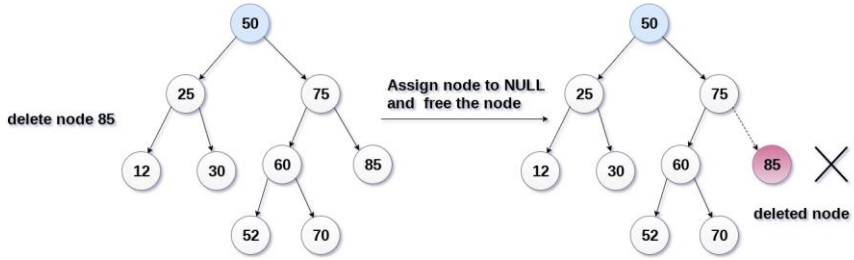


Deletion:

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

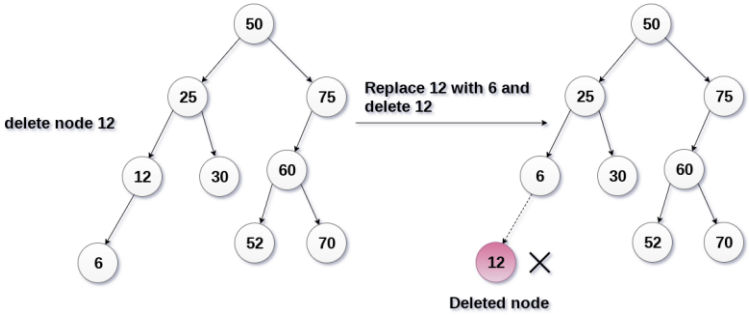
The node to be deleted is a leaf node:

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.



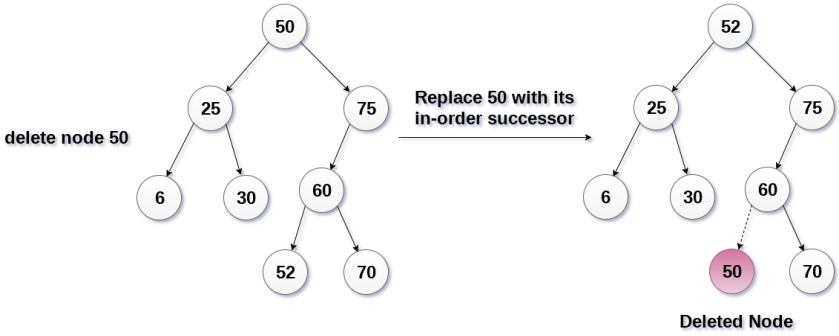
The node to be deleted has only one child:

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.



The node to be deleted has two children:

The node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.



Algorithm:

For function deletenode (root, info)

1) If (root == NULL)

   Item not found in the tree and return NULL

2) if (data < root -> info)

   Set root->left = deletenode (root->left, info);

3) else if (data > tree-> info)

   Set root->right =deletenode (root->right, info);

4) else

   // for no child

   1. if (root-> left == NULL && root -> right == NULL)
      i. Set free (root)
      ii. Return NULL

   // node with one child

   2. Else if (root -> left == NULL) then
      i. ptr= root -> right
      ii. free (root)
      iii. return ptr;
   3. else if (root -> right == NULL) then
      i. ptr = root -> left
      ii. free (root)
      iii. return ptr

   // node with two children

   4. Get the Inorder successor (smallest in the right subtree)
      i. Ptr= root -> right;
         // to find leftmost leaf
      ii. While (ptr -> left != NULL)
         ptr = ptr -> left;

      // copy the Inorder successor's content to this node

   5. root -> info = ptr -> info
   6. root -> right = deletenode (root -> right, ptr -> key);

5) Return root and End

CHAPTER 6 TREE

AVL balanced trees and balancing algorithm:

AVL tree is a self-balancing binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one which is termed as balance factor. The technique of balancing the height of binary trees was developed by Adelson, Velskii, and Landi and hence given the short form as AVL tree or Balanced Binary Tree. Every sub-tree is itself an AVL tree. If the difference is more than one, then the tree is rebalanced by applying certain rule of rotation.

Balance Factor (BF) = $H_L - H_R$

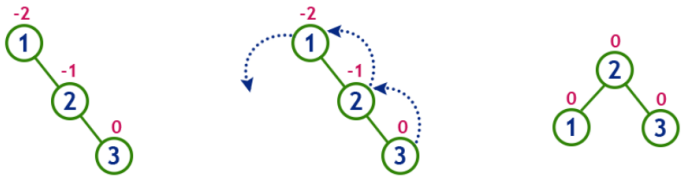
And for AVL tree, $\mathbf{|\ H_L - H_R\ |} \leq \mathbf{1}$

If BF is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations:

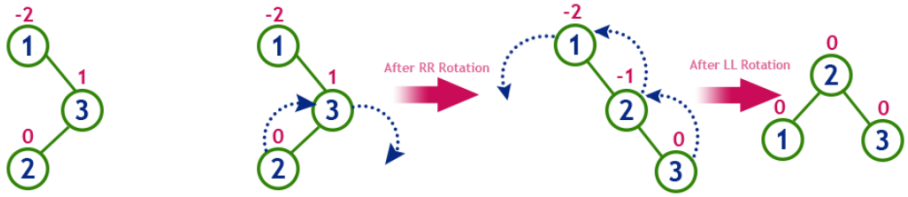
1. Right-Right Rotation:

   - If BF of node is -2 and the BF of the right child id < 0. A single 'left' rotation

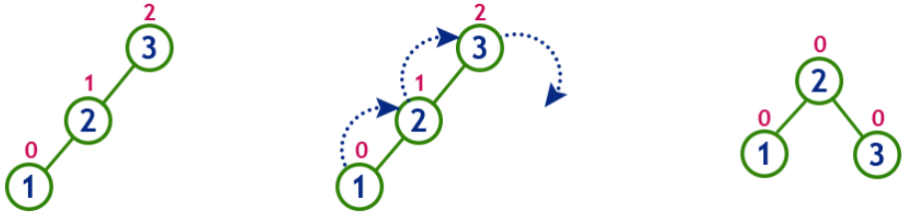


2. Right-Left rotation:

   - If BF of node is -2 and the BF of the right child id > 0 A 'right rotation followed by a 'left' rotation

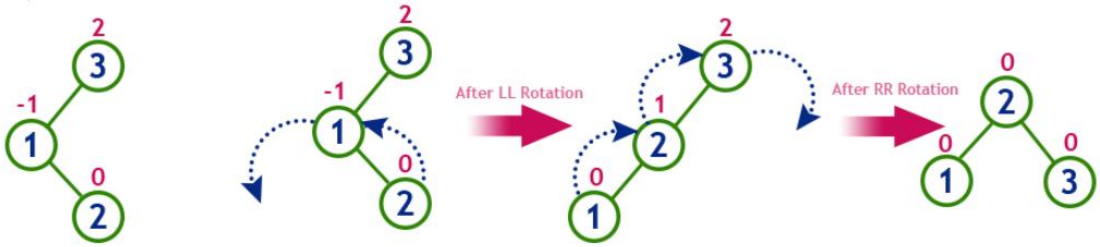


3. Left-Left rotation:
   - If BF of node is 2 and the BF of the left child id > 0. A single 'right' rotation



4. Left-Right rotation:

- If BF of node is 2 and the BF of the left child id < 0. A 'left rotation followed by a 'right' rotation

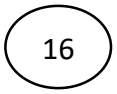


Example 1:

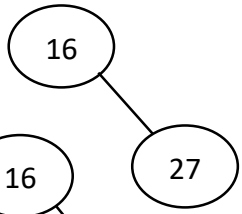
Construct AVL tree using following sequence of data:

$$16, 27, 9, 11, 36, 54, 81, 63$$

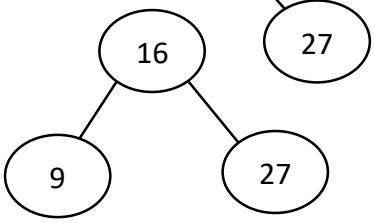
Insert 16



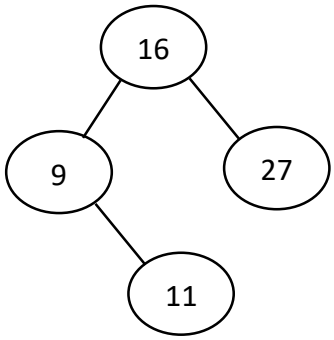
Insert 27



Insert 9

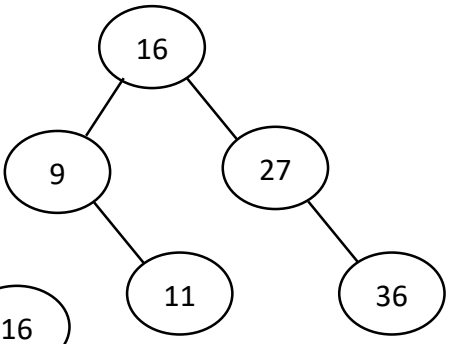


Insert 11

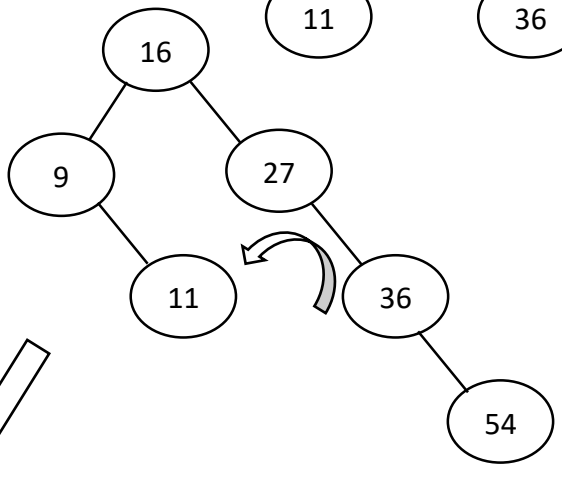


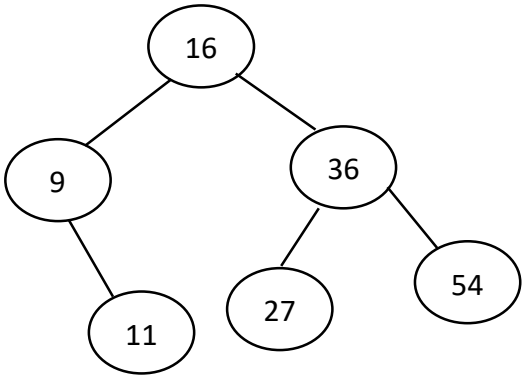
CHAPTER 6 TREE

Insert 36

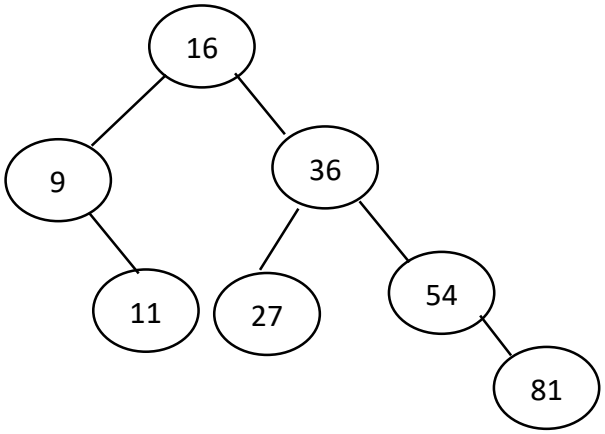


Insert 54

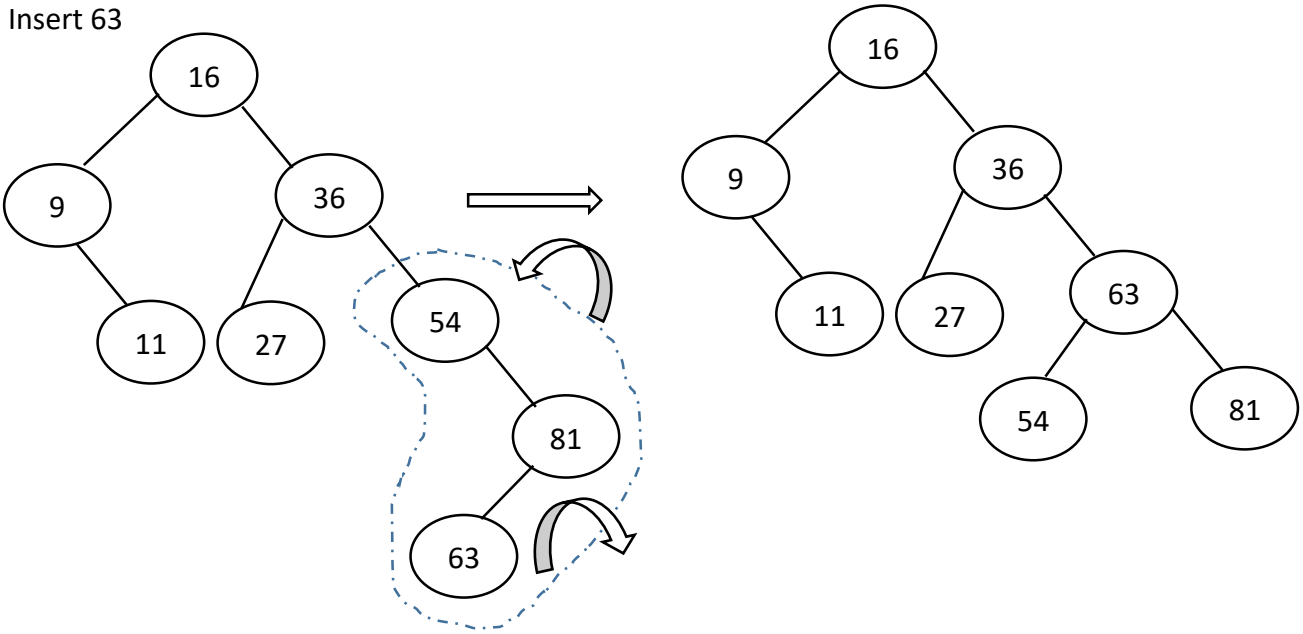




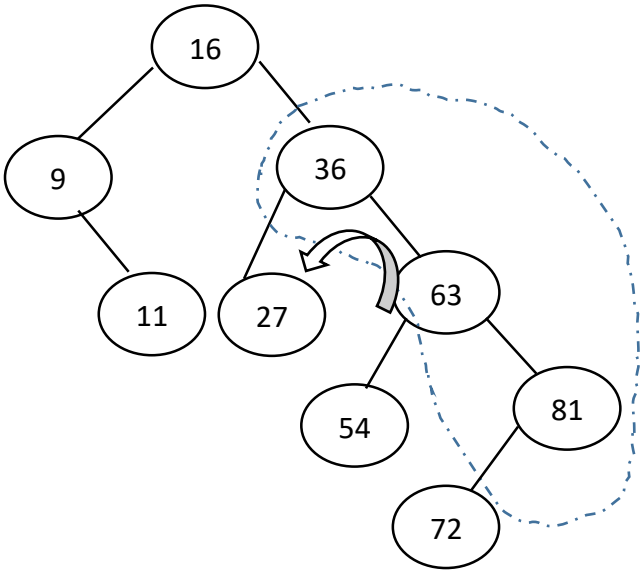
Insert 81

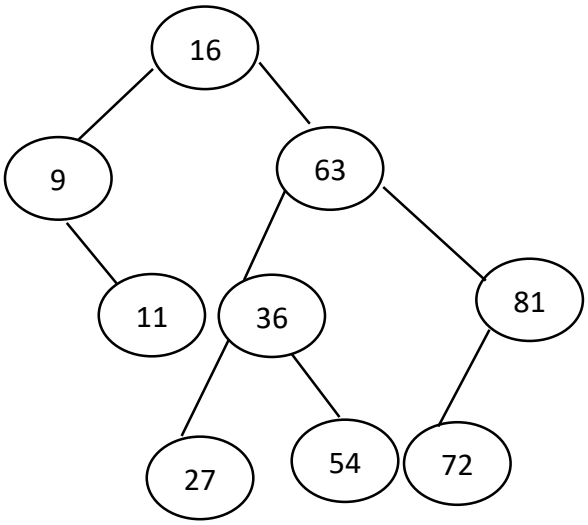


Insert 63



Insert 72



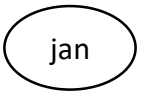


Example 2:

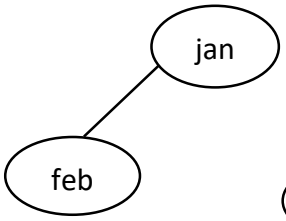
Create the AVL tree for the following data:

jan, feb, mar, apr, may, jun, jul, aug, sep, oct, noc, dec.

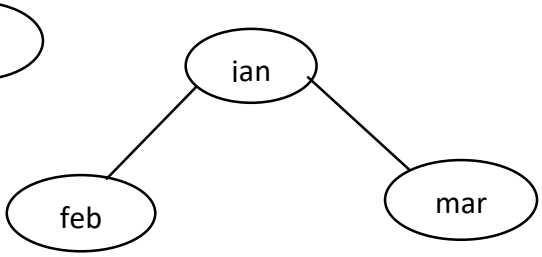
Insert: jan



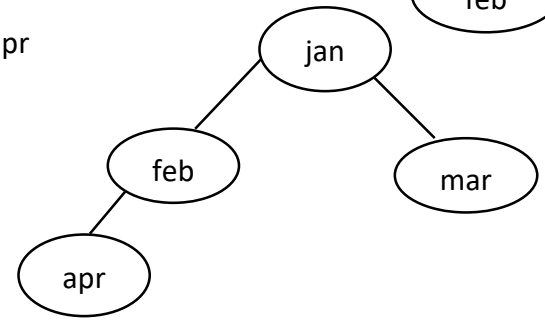
Insert: feb



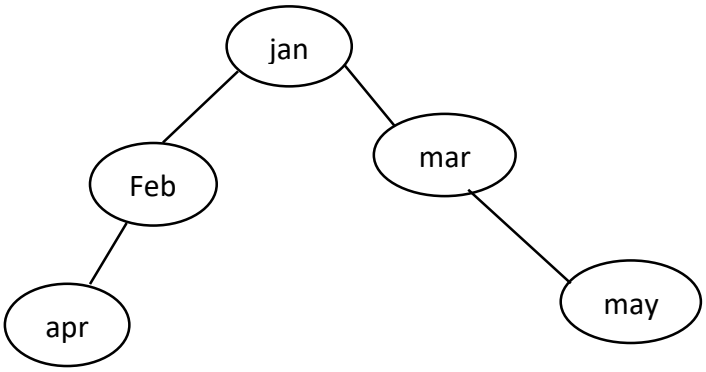
Insert: mar



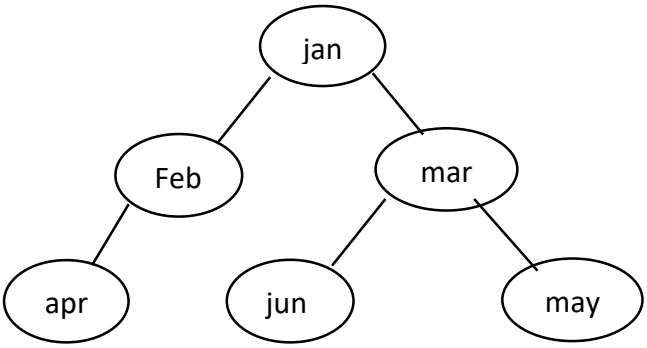
Insert: apr



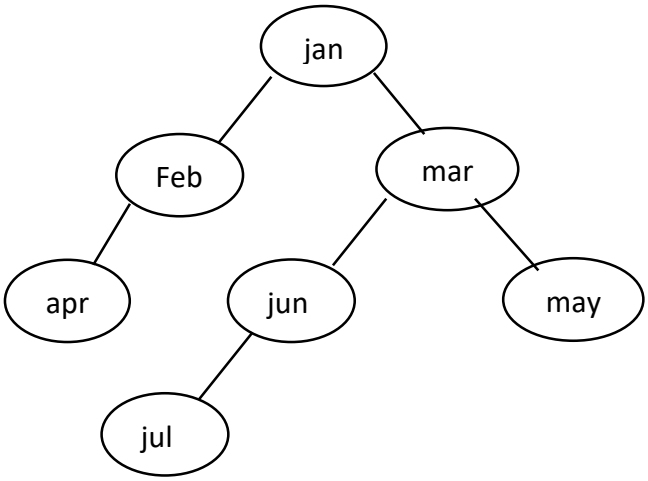
Insert: may



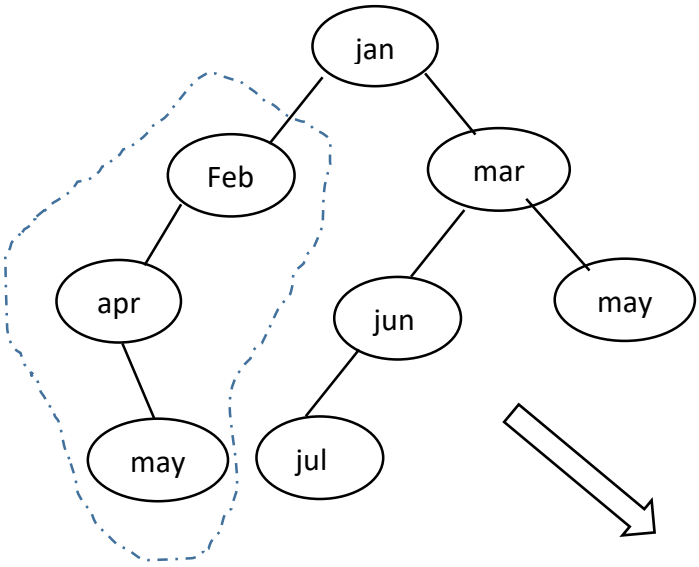
Insert: jun

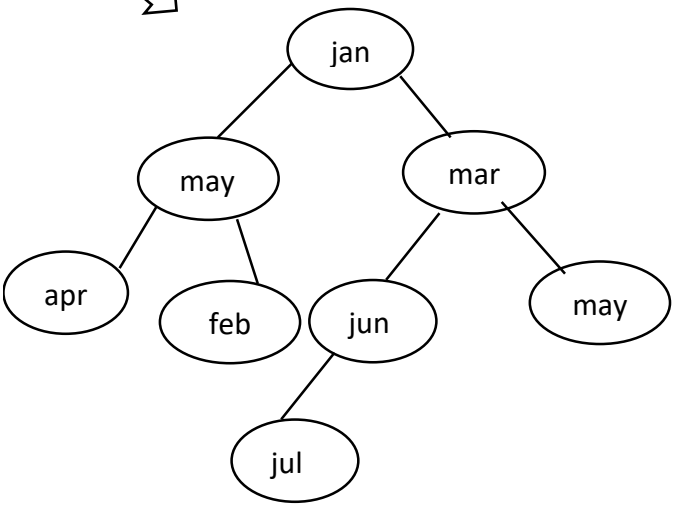


Insert: jul

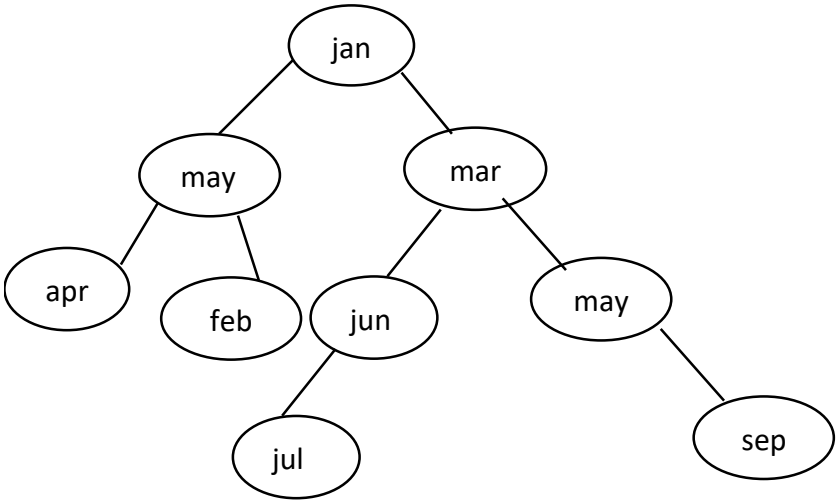


Insert: aug

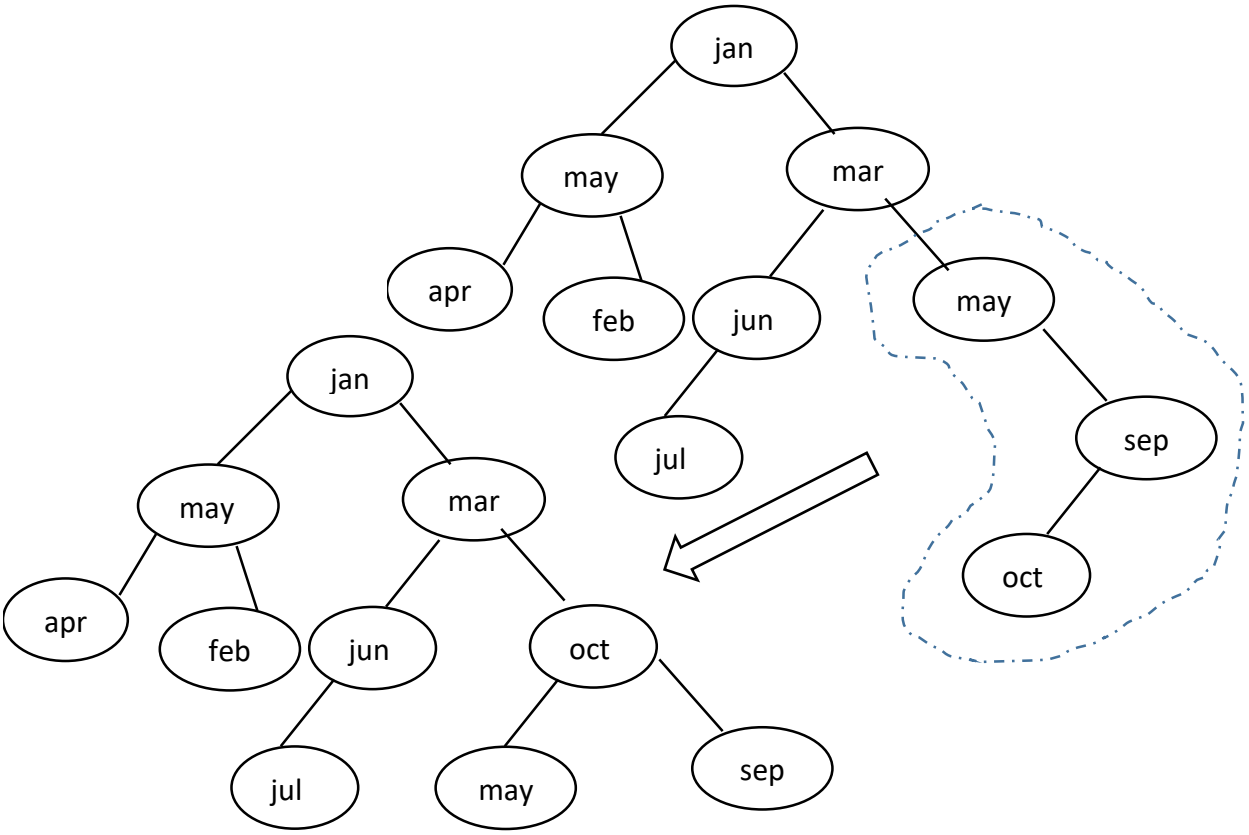




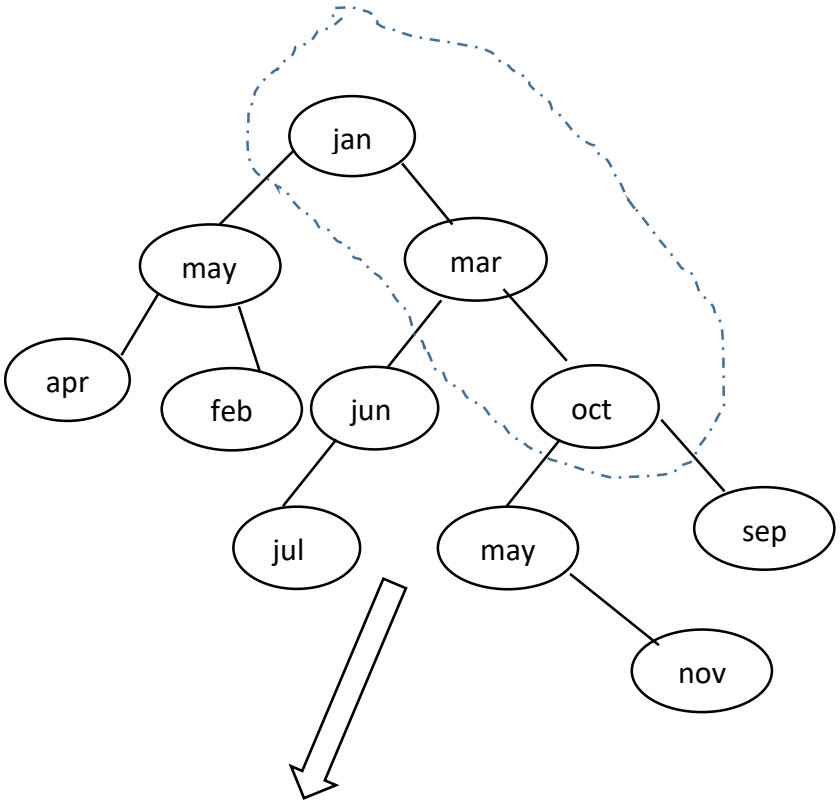
Insert: sep



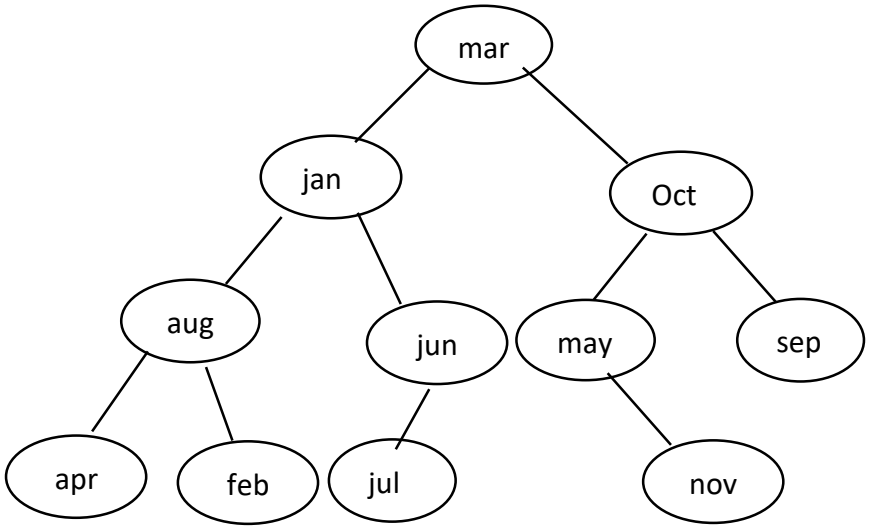
Insert: oct



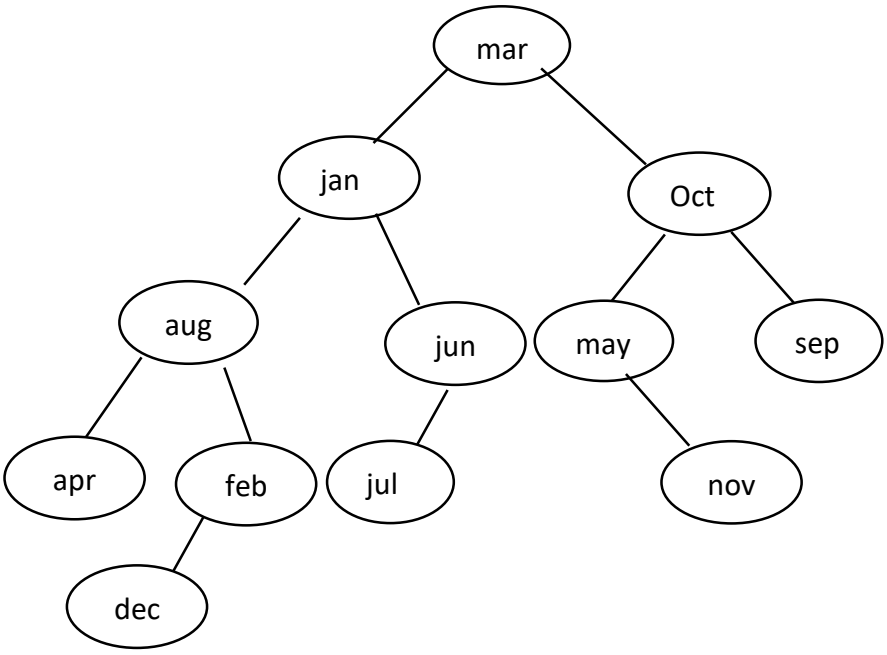
Insert: nov



CHAPTER 6 TREE



Insert: dec



CHAPTER 6 TREE

The Huffman algorithm:

Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman in 1951. Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

Properties:

- Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights and a tree-like data structure with minimum weighted path length from root is formed which can be used for generating the binary codes.
- It is a famous algorithm used for lossless data encoding.
- It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code.
- Generally, bit '0' represents the left child and bit '1' represents the right child.

Algorithm:

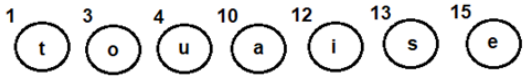
1. Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)
2. Repeat Steps 3 to 5 while heap has more than one node
3. Extract two nodes, say x and y, with minimum frequency from the heap
4. Create a new internal node z with x as its left child and y as its right child. Also **frequency(z)= frequency(x)+frequency(y)**
5. Add z to min heap
6. Last node in the heap is the root of Huffman tree

Example 1:

Create Huffman Tree for the following characters along with their frequencies.

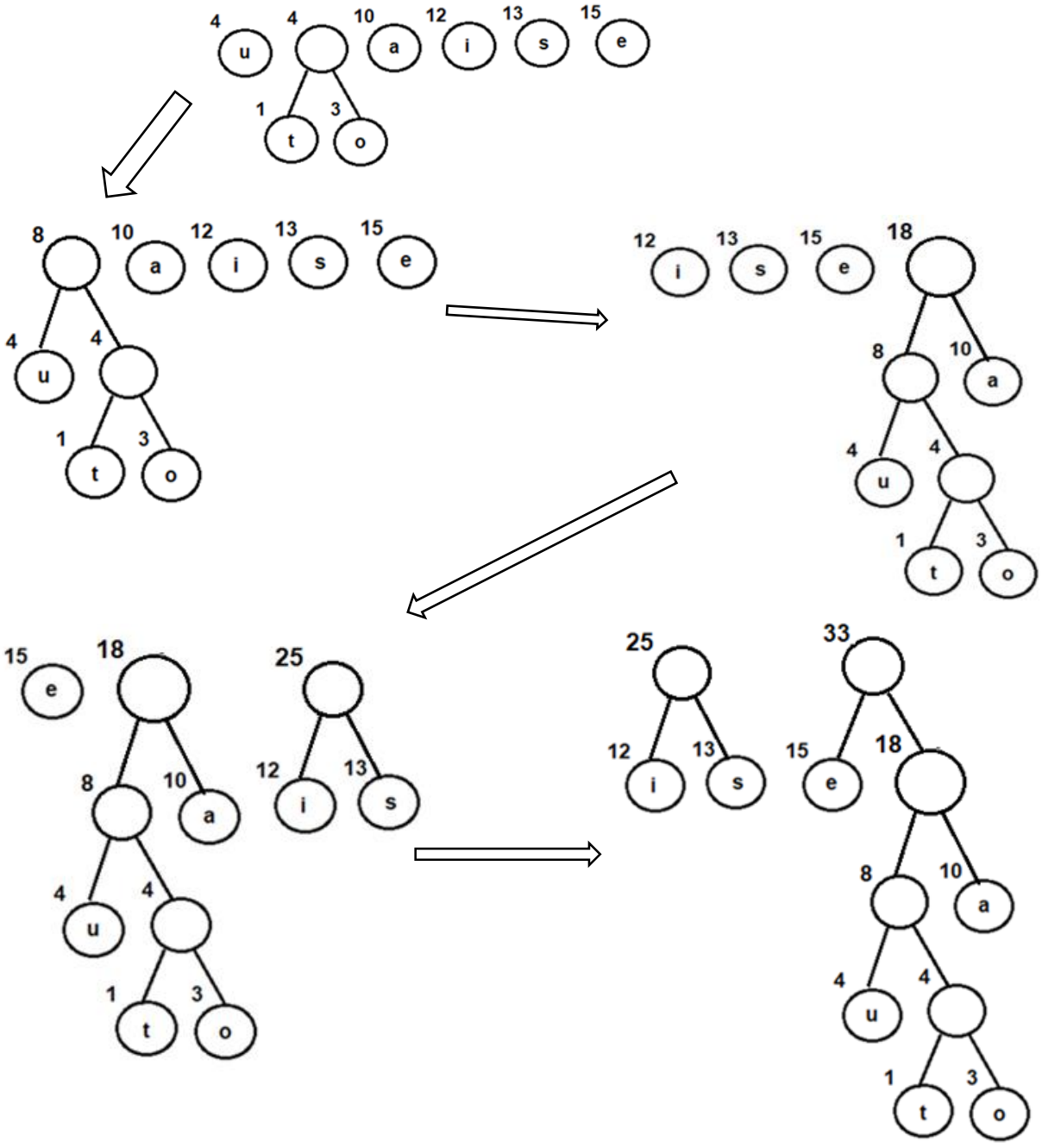
Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

Create leaf nodes for all the characters and add them to the min heap.

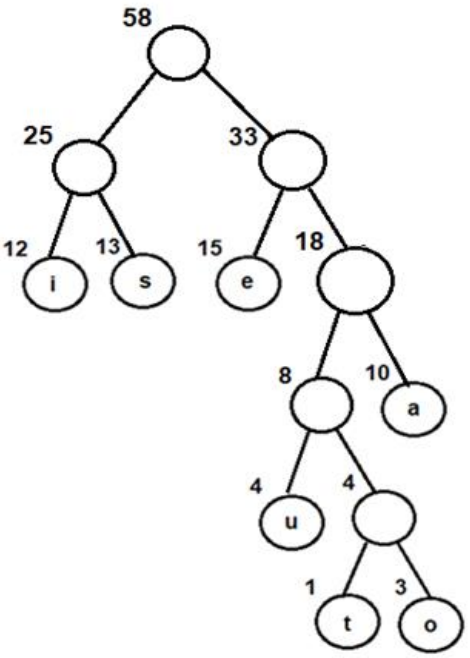


Extract two nodes, with minimum frequency from the heap and create a new internal node by adding.

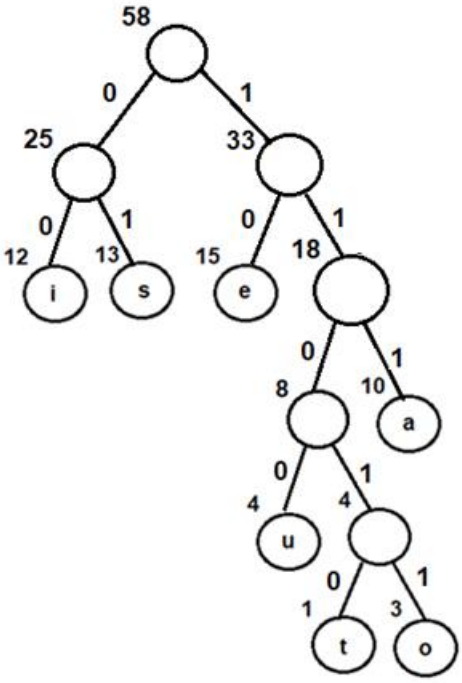
Add the new internal node to the min heap.



Last node in the heap is the root of Huffman tree.



Traverse the tree starting from root node. Add 0 to array while traversing the left child and add 1 to array while traversing the right child. Assigning binary codes to Huffman tree



We get prefix-free and variable-length binary codes with minimum expected code word length.

Characters	Binary Codes
i	00
s	01
e	10
u	1100
t	11010
o	11011
a	111

CHAPTER 6 TREE

B-Tree: "Balanced Tree"

- It is also known as balanced sorted tree.
- The height of the tree must be kept to a minimum.
- The leaves of the tree must all be the same level.
- The root has at least two subtree unless it is the only node in the tree.
- All nodes except the leaves must have at least some minimum number of children.
- Every node has maximum m children, if a B- tree has the order m.
- Every node has maximum (m-1) keys.
- Min Children:
  - For leaf: 0
  - For root: 2
  - Internal nodes: [m/2]
- Min keys:
  - Root node: 1
  - All other nodes: [m/2]-1

Insertion:

Example 1:

Consider another example for B-Tree of order 5

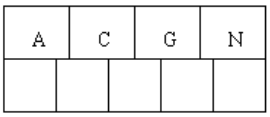
C N G A H E K Q M F W L T Z D P R X Y S

Order 5 means that a node can have a maximum of 5 children and 4 keys

All nodes other than the root must have a minimum of 2 keys

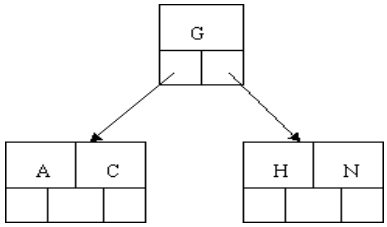
Step 1:

The first 4 letters get inserted into the same node



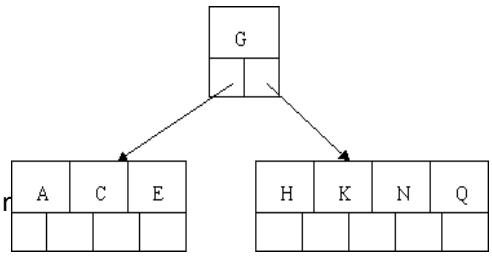
Step 2:

When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node



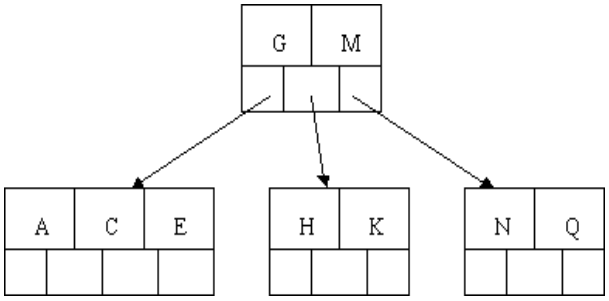
Step 3:

Inserting E, K, and Q proceeds without requiring any splits



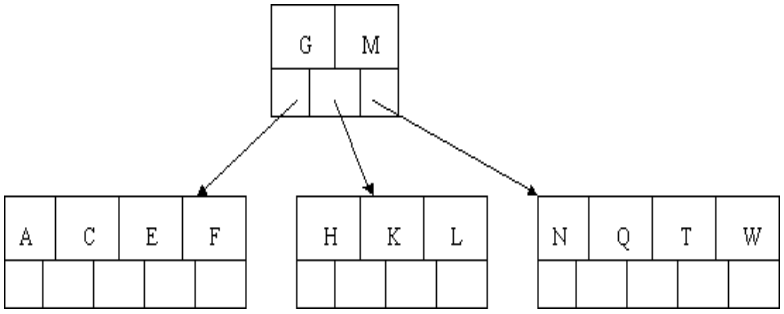
Step 4:

Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



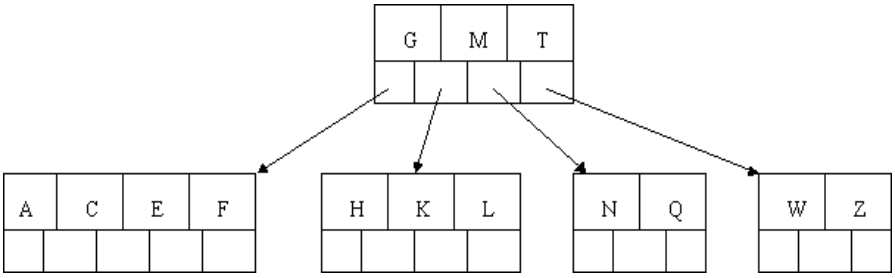
Step 5:

The letters F, W, L, and T are then added without needing any split.



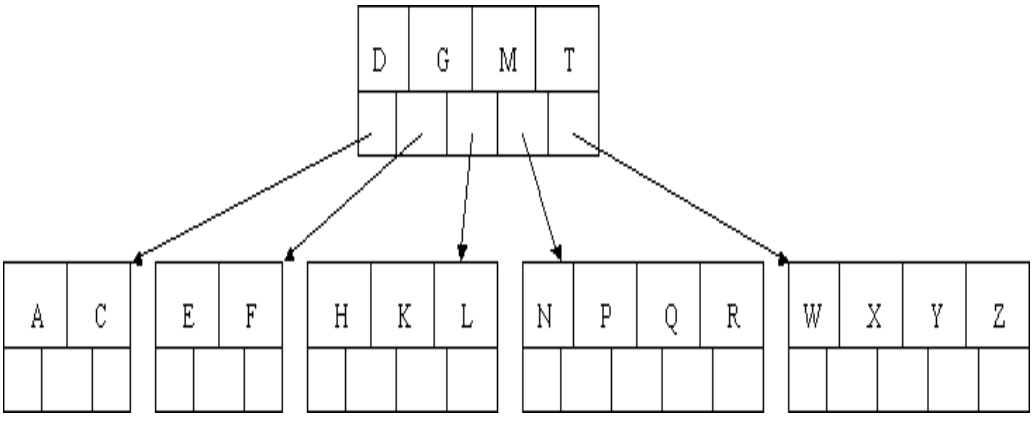
Step 6:

When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node.



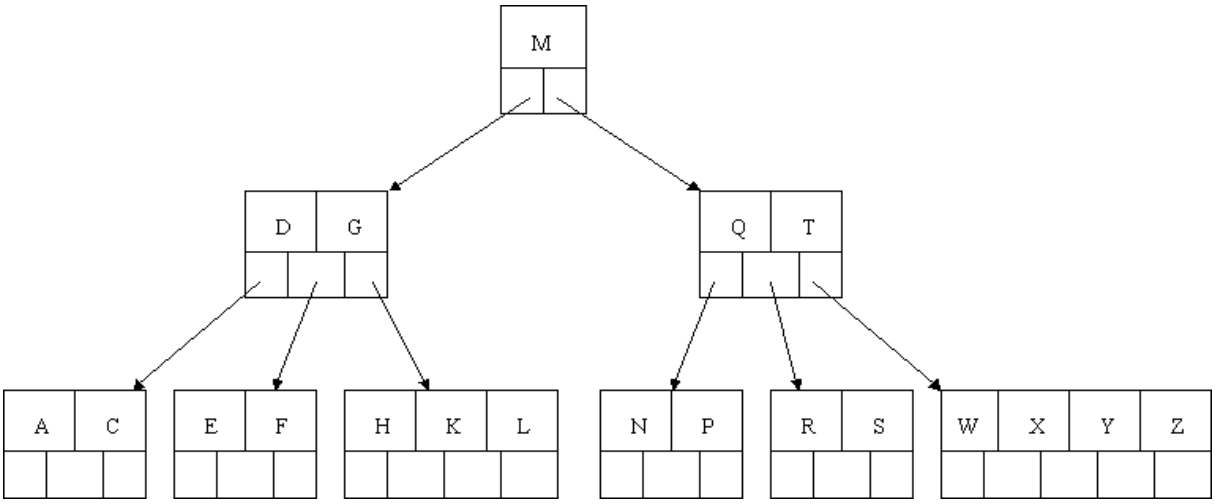
Step 7:

The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting.



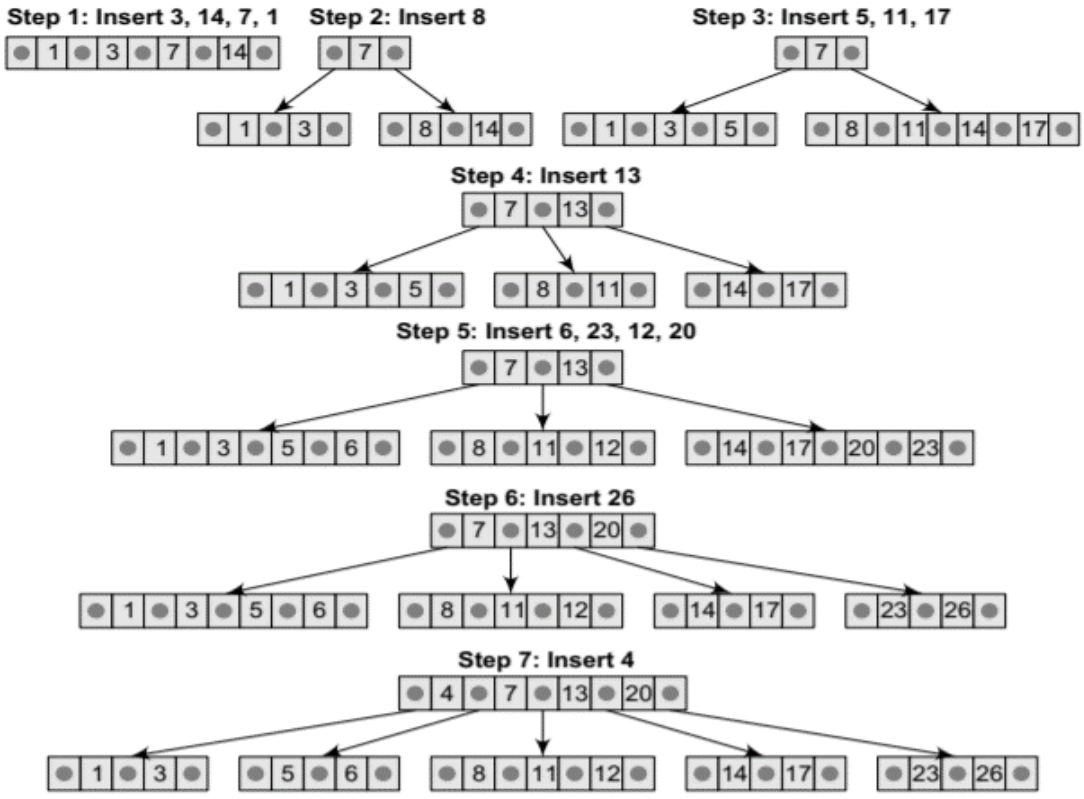
Step 8:

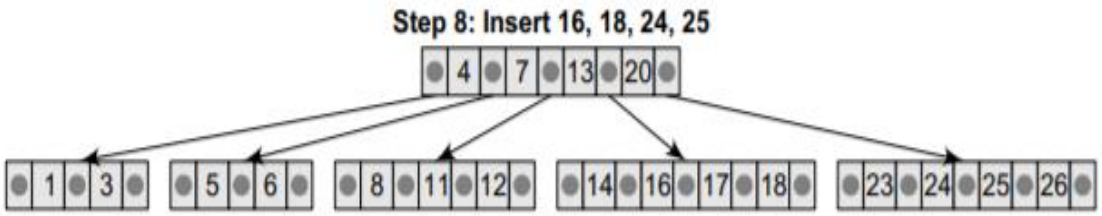
- Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Final B-Tree look like

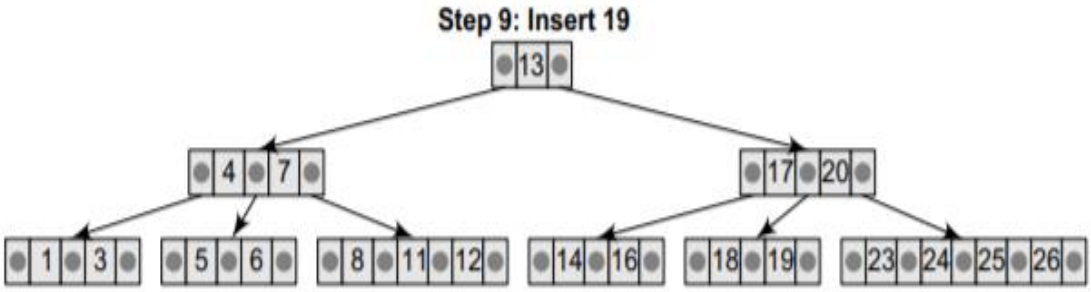


Example 2. Create a B tree of order 5 by inserting the following elements:

$3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and\ 19.$







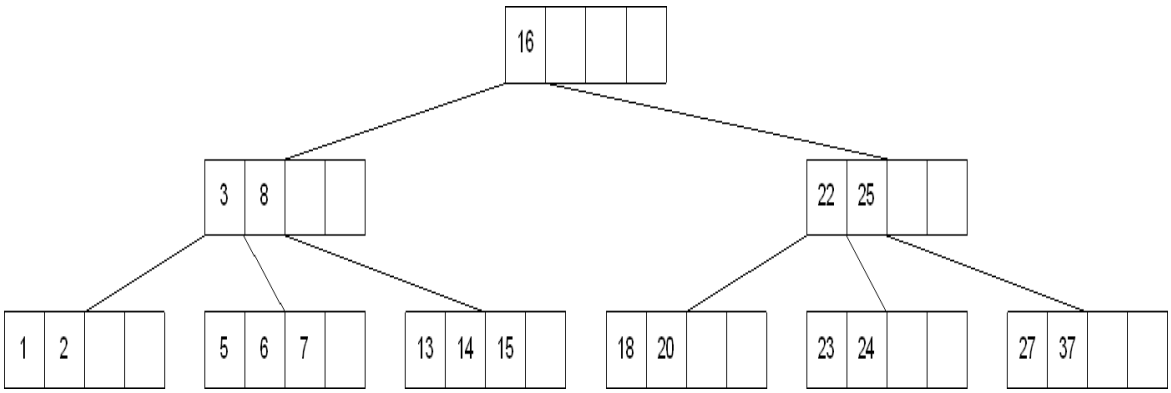
Deletion:

There are two main cases to be considered:

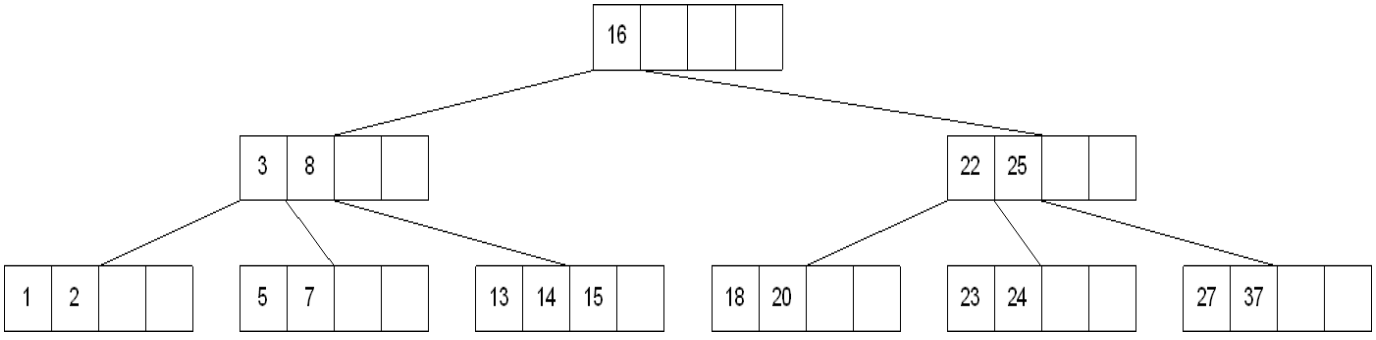
- Deletion from a leaf
- Deletion from a non-leaf

Case 1: Deletion from a leaf

a. If the leaf has at least (m-1)/2 data after deleting the desired value, the remaining larger values are moved to "fill the gap".

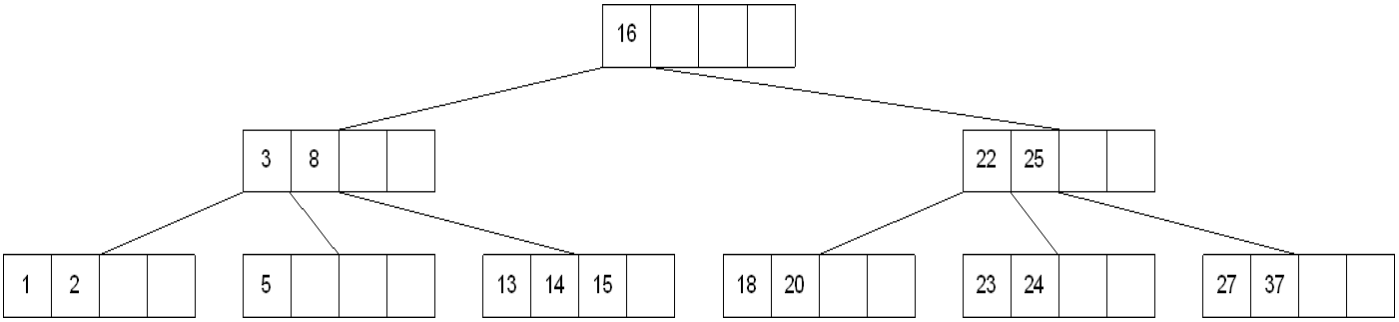


Deleting 6 from the above tree



b. If the leaf is less than (m-1)/2 after deleting the desired value (known as underflow), two things could happen

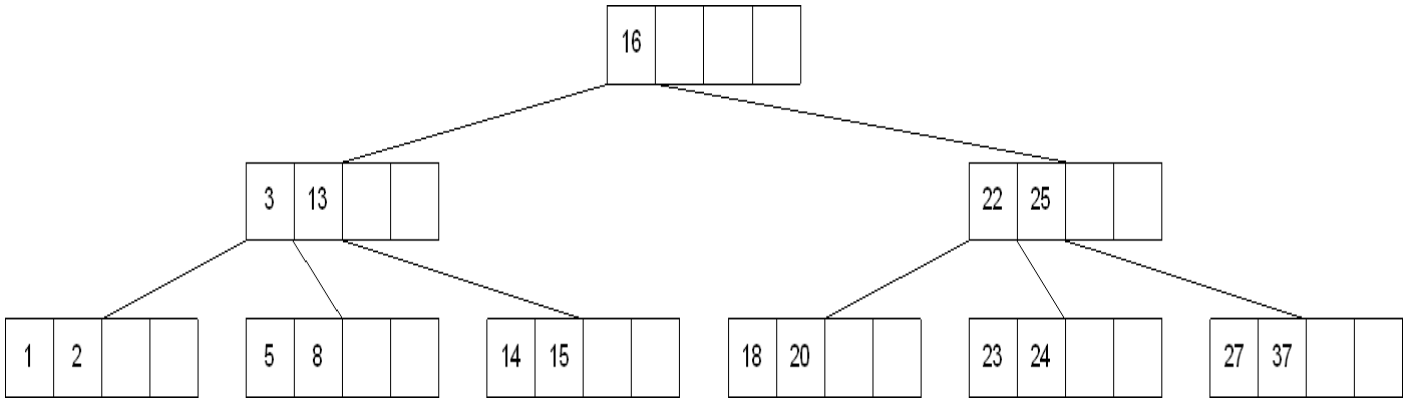
Deleting 7 from the tree above results in



i) If there is a left or right sibling with the number of keys exceeding the minimum requirement,

a) all of the keys from the leaf and sibling will be redistributed between them by moving the separator key from the parent to the leaf and

b) moving the middle key from the node and the sibling combined to the parent



ii) If the number of keys in the sibling does not exceed the minimum requirement,

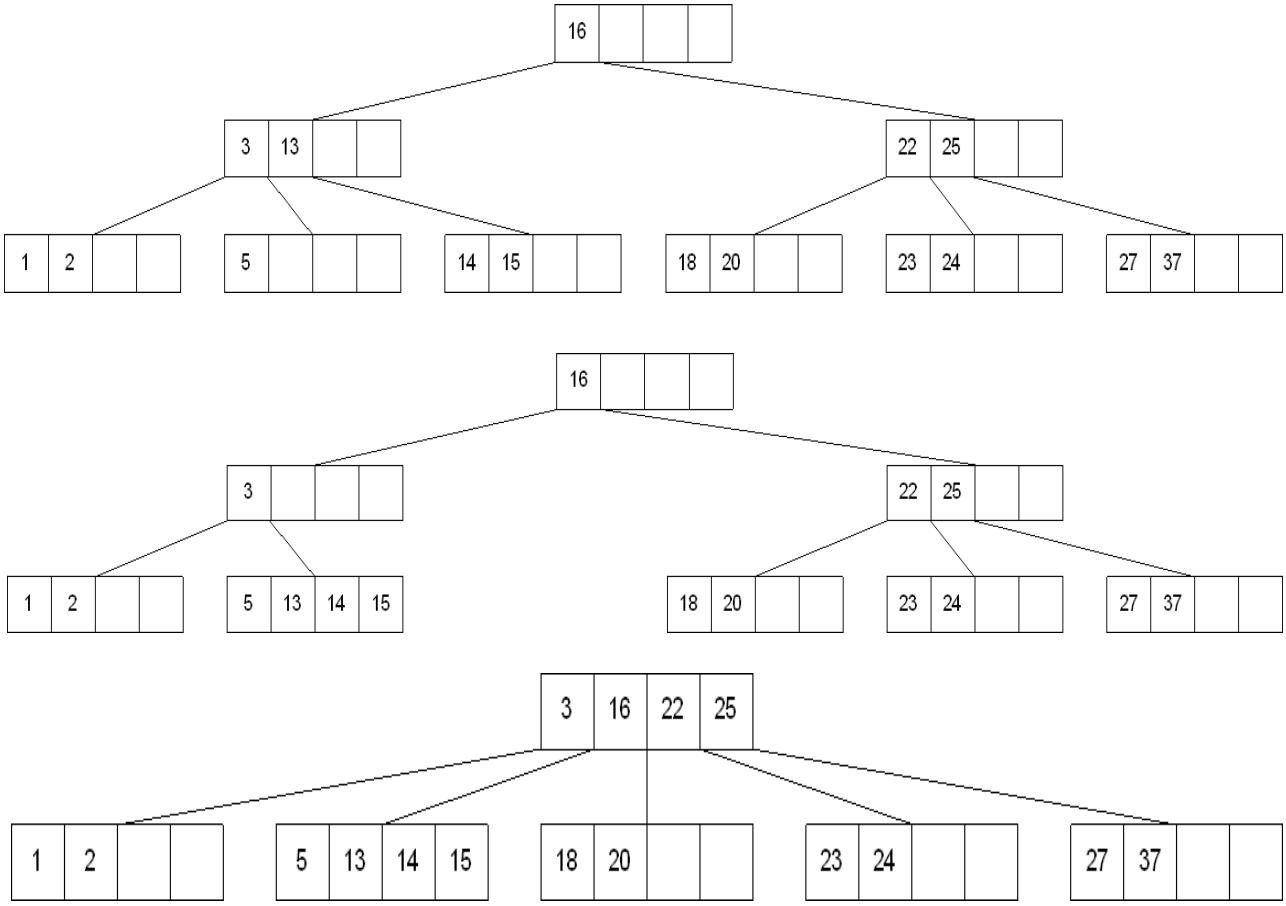
a) The leaf and sibling are merged by putting the keys from the leaf, the sibling, and the separator from the parent into the leaf.

b) The sibling node is discarded and the keys in the parent are moved to "fill the gap".

c) If the parent itself is underflow, treat the parent as a leaf and continue repeating from step a) until

The minimum requirement is met or the root of the tree is reached.

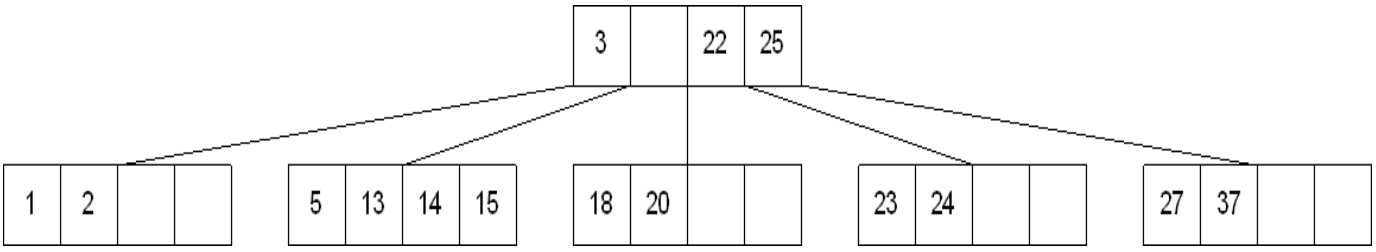
Deleting 8 from the tree above results in



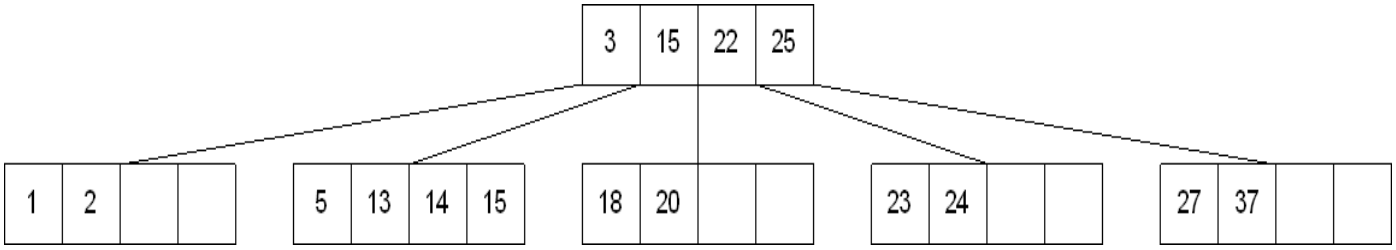
Case 2: Deletion from a non-leaf node

a. The key to be deleted will be replaced by its immediate predecessor (or successor)
b. Then the predecessor (or successor) will be deleted since it can only be found in a leaf node.

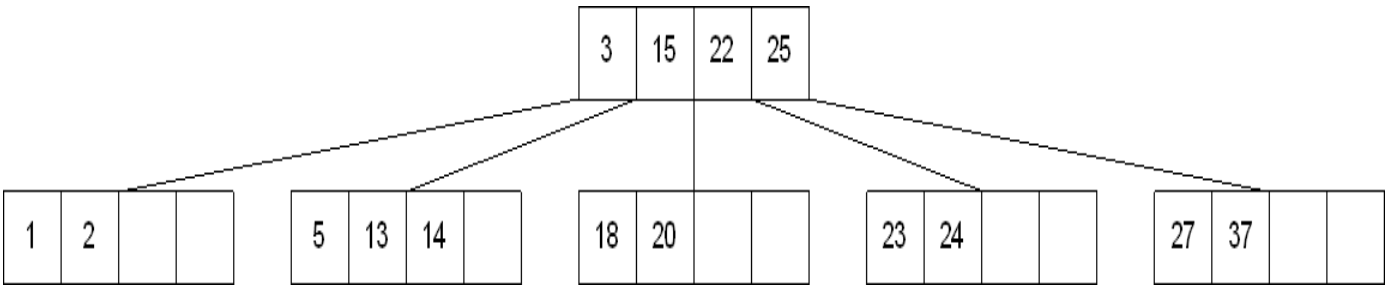
Deleting 16 from the tree above results in



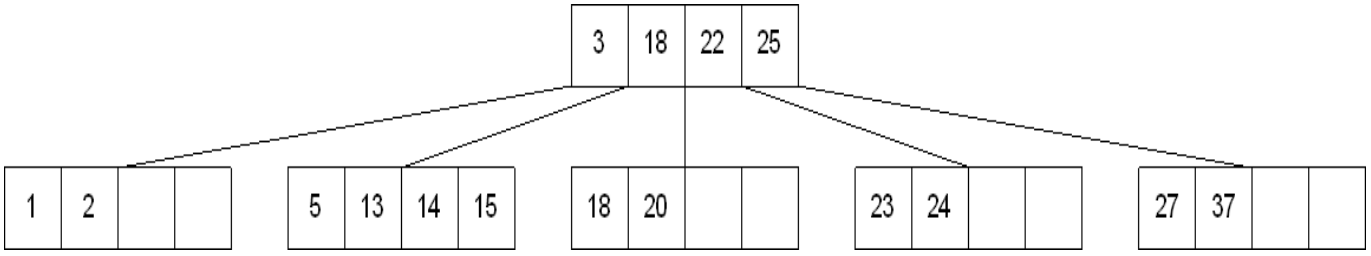
The "gap" is filled in with the immediate predecessor



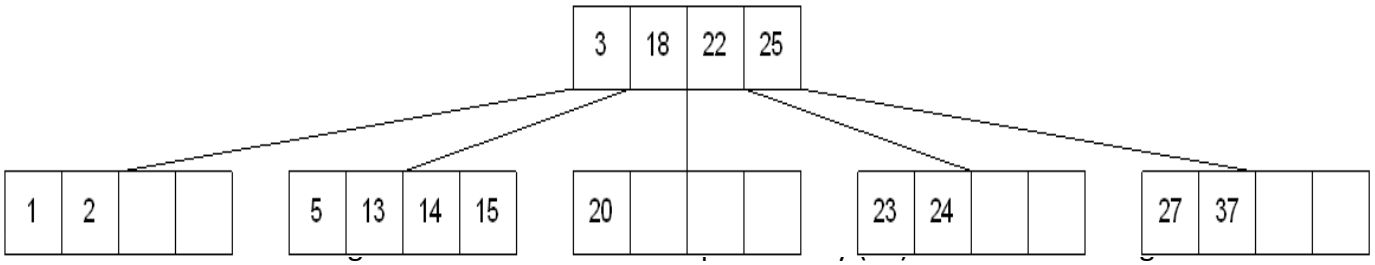
And then the immediate predecessor is deleted



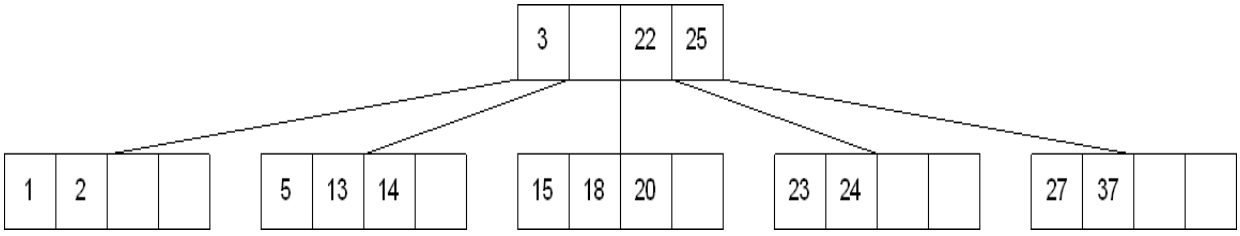
If the immediate successor had been chosen as the replacement



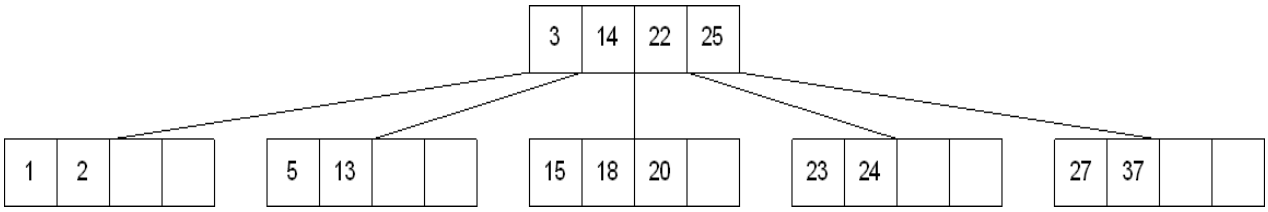
Deleting the successor



values. They are divided between the 2 nodes



And then the middle value is moved to the parent

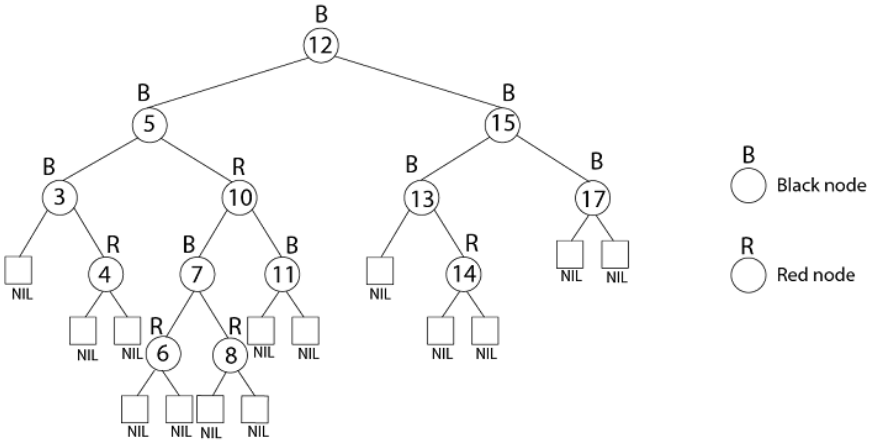


Red Black Tree:

A Red Black Tree is a type of self-balancing binary search tree, in which every node is colored with a red or black. The red black tree satisfies all the properties of the binary search tree but there are some additional properties which were added in a Red Black Tree.

Properties of Red Black Tree:

1. The root node should always be black in color.
2. Every nil child of a node is black in red black tree.
3. The children of a red node are black. It can be possible that parent of red node is black node.
4. All the nil have the same black depth. Every simple path from the root node to the (downward) nil node contains the same number of black nodes.

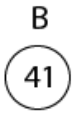


Insertion:

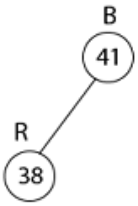
- Insert the new node the way it is done in Binary Search Trees.
- Color the node red
- If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

Example: Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.

Insert 41

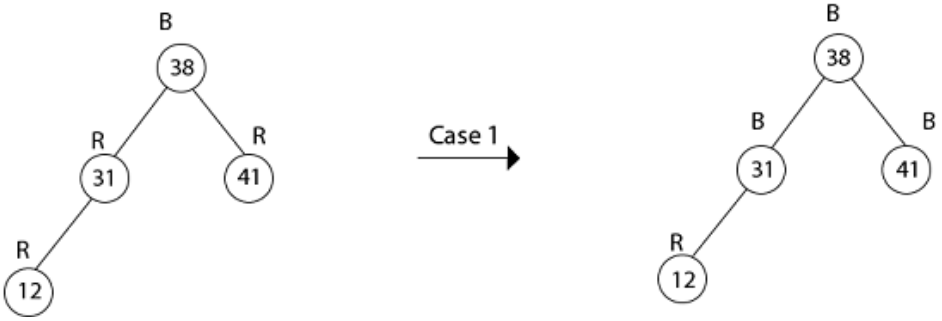


Insert 38

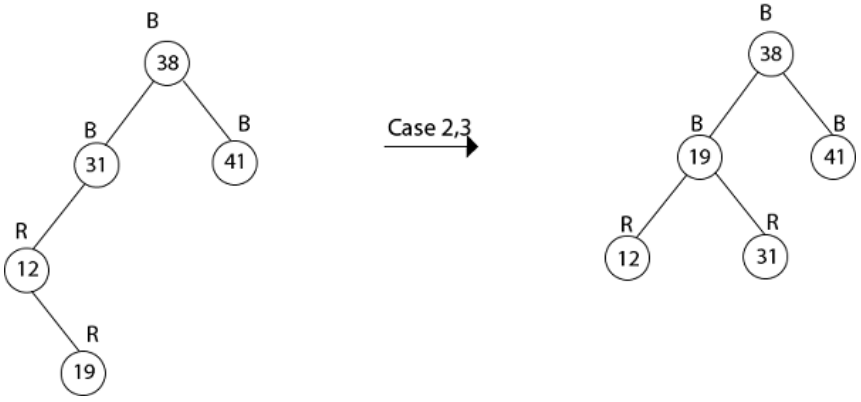


Insert 31

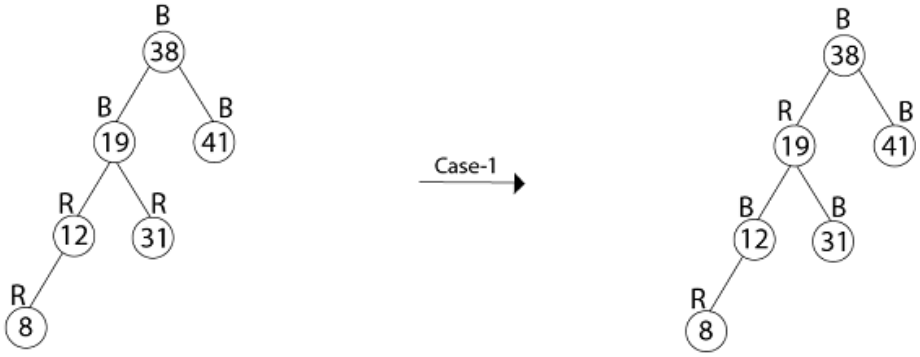
Insert 12



Insert 19



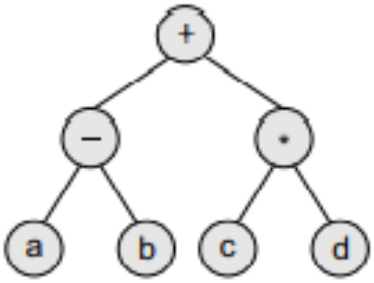
Insert 8



Binary Expression Tree:

The expression tree is a binary tree which is used to store algebraic expressions in which each internal node corresponds to the operator and each leaf node corresponds to the operand.

Expression: $(a - b) + (c * d)$



Expression: $a + (b * c) + d * (e + f)$