TREE

Introduction to Trees

A tree is a collection of elements called “nodes”,one of which is distinguished as a root say r ,

The root can have zero or more nonempty subtrees T_1, T_2,… T_k each of whose roots are connected by a directed edge from r.

The root of each subtree is said to be a child of r and r is parent of each subtree root.

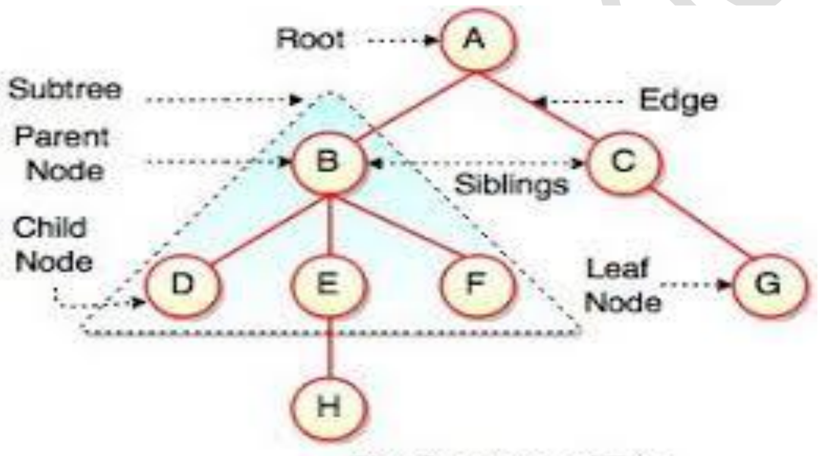


Fig. Structure of Tree

Root: it is a special node in a tree structure and the entire tree is referenced through it. This mode does not have a parent.

Parent: it is an immediate predecessor of a node ,. In the fig A is the parent of B,C,D.

Child: All immediate successor of a node are its children. In the fig B,C and D are the children of A.

Siblings: nodes with the same parent are siblings H,I and J are siblings as they are children of the same parent D.

Path: it is number of successive edges from source node to destination node.

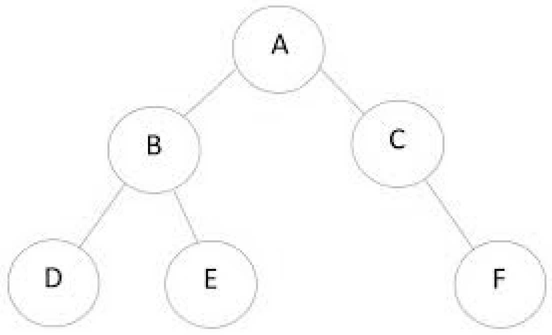
Degree of a node:
The degree of a node is a number of children of a node
A node of degree 2 and B node of degree 3.

Leaf node: A node of degree 0 is also know as a leaf node. A leaf node is a terminal node and it has no children.

Level: The level of a node in a binary tree is defined as:
The root of the tree has level 0

Depth: The Depth of a binary tree is the maximum level of any node in the tree. This is equal to the node in the tree. This is equal to the longest path from the root to any leaf node.

Binary Tree
A tree is binary if each node of the tree can have maximum of two children.
Children of a node of binary tree are ordered. One child is called the “left” child and the other is called “right” child.



Linked Representation of a Binary Tree

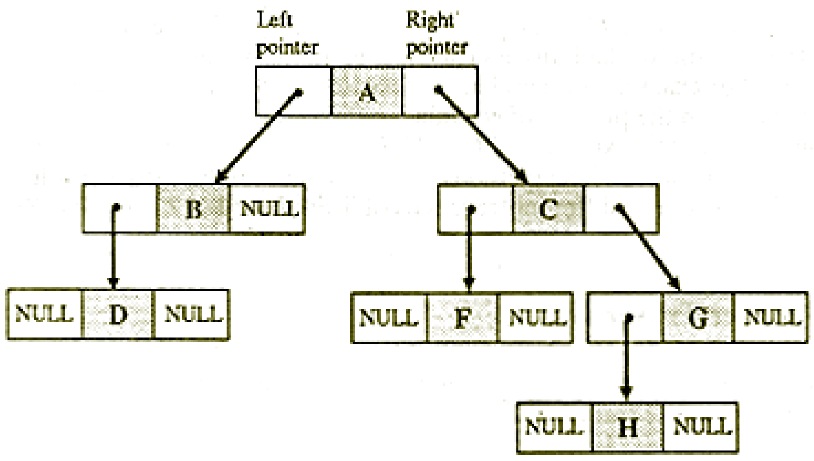
Linked representation of a binary tree is more efficient than array representation. A node of a binary tree consist of three fields as shown below

- Data
- Address of the left child
- Address of the right child

Left	Data	Right

Syntax :

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



Types of Binary Tree

1. **Full Binary Tree**

   A binary tree is said to be full binary tree if each of its node has either two children or no child at all.
   Every level is completely filledup. Number of node at any level i in a full binary tree is given by 2^i .

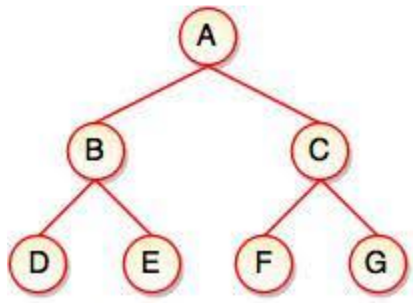
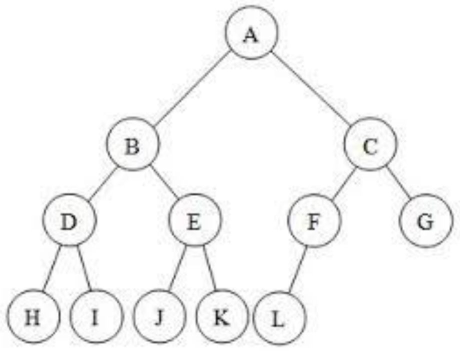


Fig. Full Binary Tree

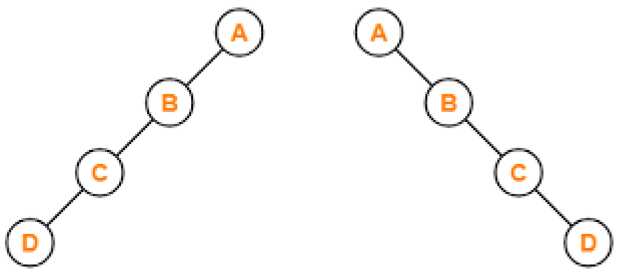
2. **Complete Binary Tree**

   A complete binary tree is defined as a binary tree where
   All leaf nodes are on level n or n-1 Levels are filled from left to right.
   heap is an example of complete binary tree



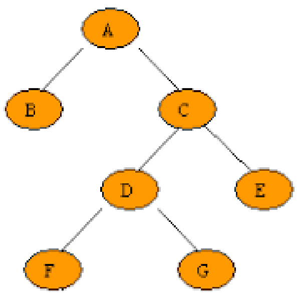
3. **Skewed Binary Tree**

   Skewed binary tree could be skewed to the left or it could be skewed to the right.
   In a left skewed binary tree, most of the nodes have the left child without corresponding right child.
   Binary search tree could be an examples of a skewed binary tree.



Left Skewed Binary Tree Right Skewed Binary Tree

4. **Strictly Binary Tree**
   If every non-terminal node in a binary tree consists of non-empty left subtree and right subtree, then such a tree is called strictly binary tree.
   In other words, a node will have either two children or no child at all.



<u>**Binary Tree Traversal**</u>

- Most of the tree operations require traversing a tree in a particular order.
- Traversing a tree is a process of visiting every node of the tree and exactly once.
- For example, to traverse a tree, one may visit the root first, then the left subtree and finally traverse the right subtree.
- If we impose the restriction that left subtree is visited before the right subtree then three different combination of visiting the root, traversing left subtree, traversing right subtree is possible.

1. Visit the root, traverse, left subtree, traverse right subtree.

2. Traverse left subtree, visit the root, traverse right subtree.
3. Traverse left subtree, traverse right subtree, visit the root.

These three techniques of traversal are known as preorder, inorder and postorder traversal of a binary tree.

Preorder traversal (recursive)

- First Visit the root
- Next, Traverse left subtree
- At last, traverse the right subtree.

```
void preorder(node* T ) /* address of root node is passed in T */
{
      if(T!=NULL)
      {
            printf("\ n% d",T→data);
            preorder (T→ left);
            preorder(T→ right ):
}
```

Inorder traversal (Recursive)

- Firstly, traverse the left subtree in inorder
- Next, visit the root node
- At last, traverse the right subtree in inorder.

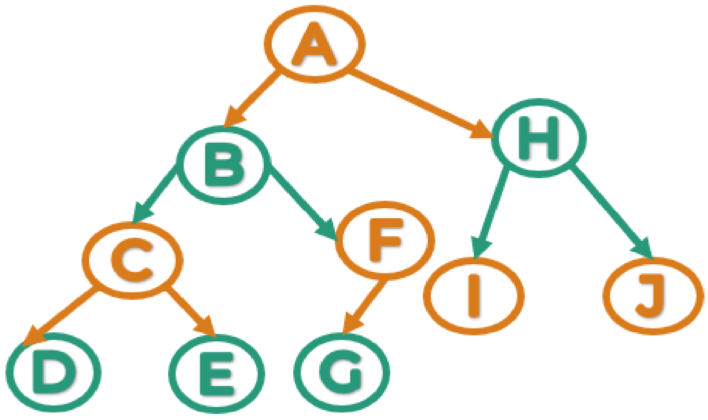
```
void inorder(node*T) /* address of the node is passed in T */

{

    if (T!=NULL)

    {

        inorder (T→left);

        printf('\ n% , T→data);

        inorder (T→ right);
        }
 }
```

Postorder Traversal (recursive)

- Firstly, traverse the left subtree in postorder
- Next, traverse the right subtree in postorder
- At last, visit the root node.

```
void postorder(node*T) /* address of the root node passed in T*/
{
        if(T!= NULL)
        {
                postorder(T→left);
                postorder (T→right);
                print (“\ n% d”,T→data);
        }
}
```

1. **Perform inorder, preorder and postorder traversal of the following binary tree**

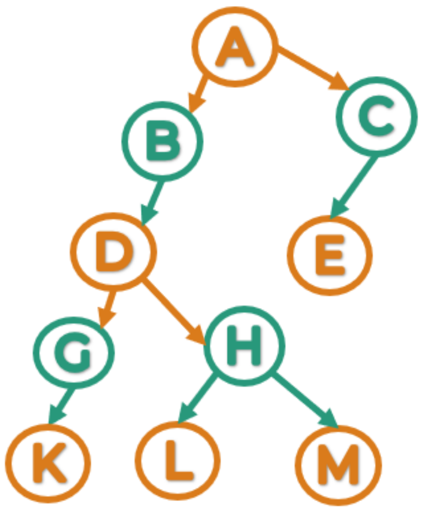


Preorder sequence : A B C D E F G H I J

Inorder sequence : D C E B G F A I H J

Postorder sequence : D E C G F B I J H A

2. For the given binary tree, perform inorder, preorder and postorder traversal.



Postorder sequence : A B D G K H L M C E

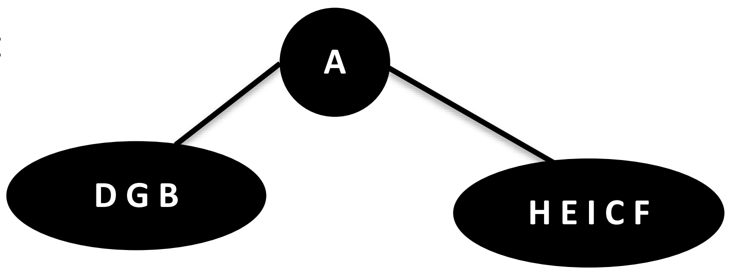
Inorder sequence : K G D L H M A E C

Preorder sequence : K G L M H D B E C A

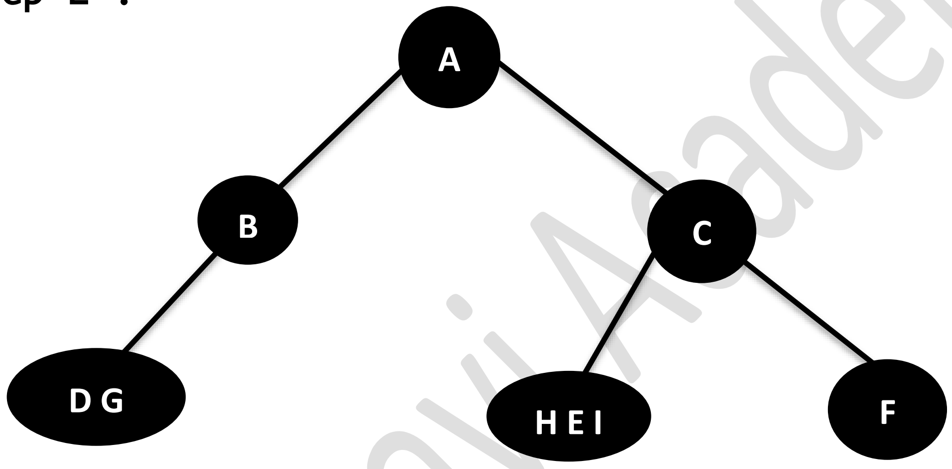
Construction of binary tree from preorder and inorder traversal sequence

Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F

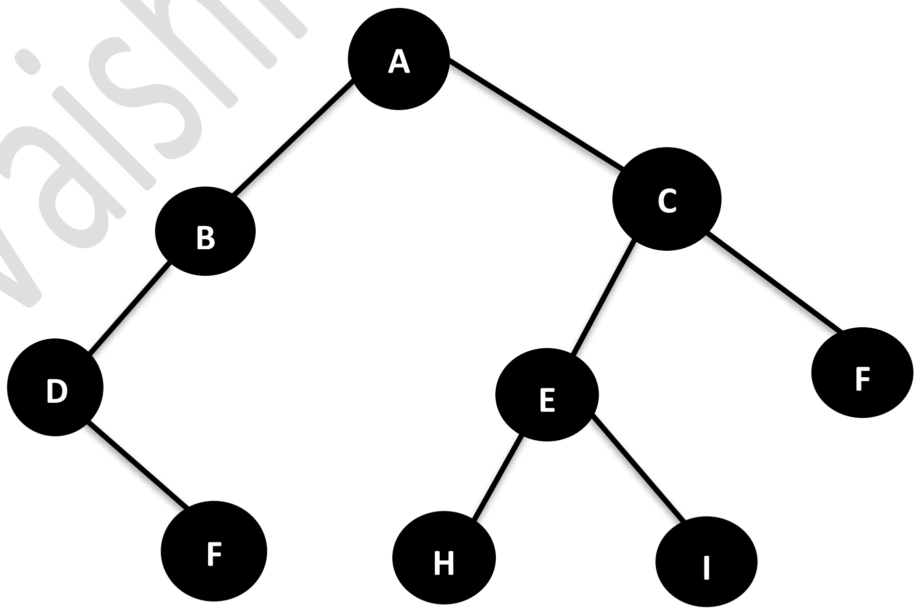
Step 1 :



Step 2 :



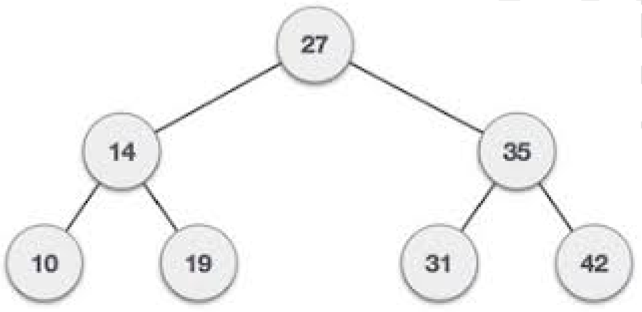
Step 3 :



Binary Search Tree (BST)

A binary search tree is a binary tree, which is either empty or in which each node contains a key that satisfies the following conditions

1. All keys are distinct.
2. For every node, X in the tree, the values of all the keys in its left subtree are similar than the key value in X.
3. For every node, X in the tree, the of all the keys in its right subtree are larger than the key value in X.



The basic operations that can be performed on a binary search tree data structure, are the following -

- **Find** - Searches an element in a tree.
- **Insert** - Inserts an element in a tree/create a tree.
- **Delete**- Delete an element in a tree.
- **FindMin** -Find minimum element in a tree
- **Findmax** – find maximum element in a tree
- **Create** – create BST tree.

Find 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.

Recursive

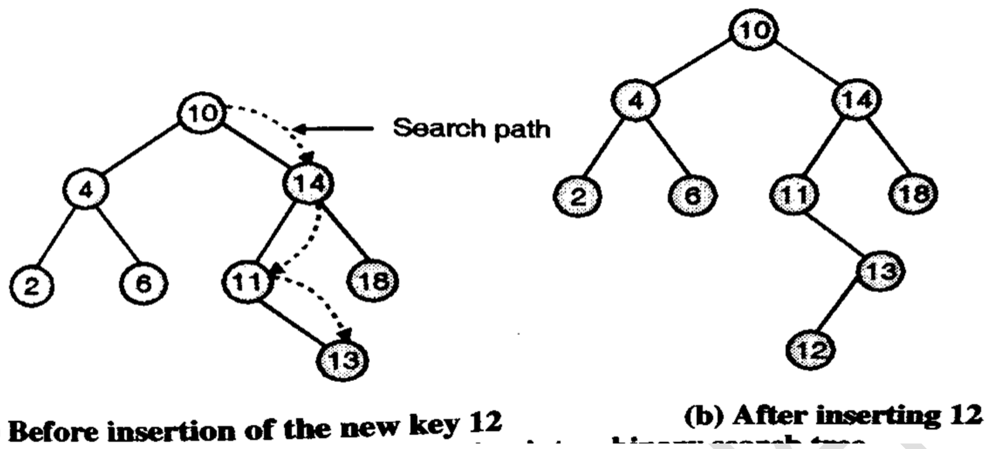
```
BSTnode * Find(BSTnode *root, int x)

{

        if (root == NULL )
        {
                return(null);
        }
        if(root→data == x)
                return root;
        if (root->key < key)
                return search(root→right, x);
        else
                return search(root→left, x);
}
```

Non recursive

```
BSTnode * Find(BSTnode *root, int x)

{
        while(root != null)
        {
                If(x==root→data)
                        return (root);
                if(x>root→data)
                        root=root->right;
                else
                        root=root→left;
        }
        return(NULL);
}
```

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.



(a) Before insertion of the new key 12 (b) After inserting 12

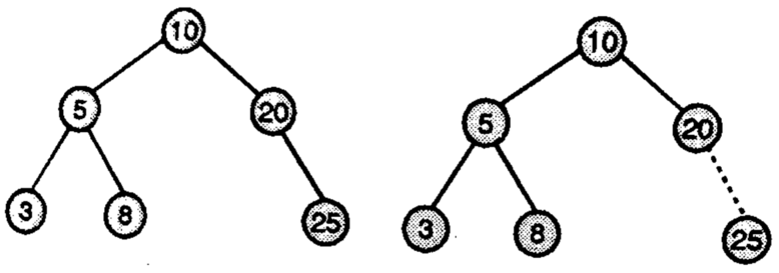
```
BSTnode* insert(BSTnode* T, int X)
{
   /* If the tree is empty, return a new node */
   if (T== NULL)
   {
       T=(BSTnode *)malloc(sizeof(BSTnode));

       T->data = x;

       T->left = T->right = NULL;

       return temp;

   }
   /* Otherwise, recur down the tree */
   if (X <T->data)
      T->left  = insert(T->left, X);
   else if (X> T->key)
      T->right = insert(T->right, key);
   return (T);
}
```

Delete Operation

1. A Leaf node
2. A node with one child.
3. A node with two child.

1. <u>A Leaf node</u>

If node is a leaf node, it can be deleted immediately by setting the corresponding parent pointer to NULL.

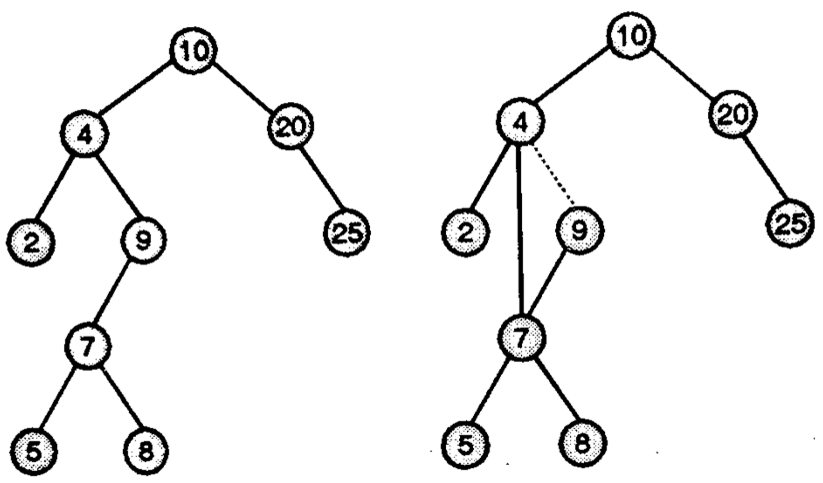


2. <u>A node with one child.</u>

When node to be deleted has one child, it can be deleted easily.

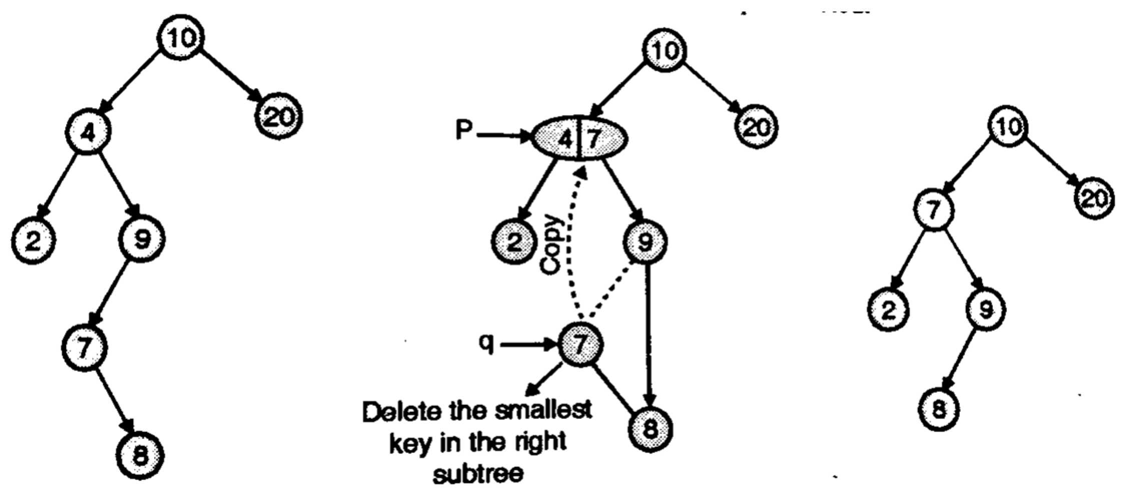
If node q is to be deleted and it is right child of its parent node p. the only child of q will become the right child of p after deletion of q.

Similarly if a node q is to be deleted and it is left child of its parent node p, only child of q will become the left child of p after deletion.



3. **A node with two child.**

When node to be a deleted has two child is to replace the data of this node with smallest data of right subtree. Smallest data can be either leaf node or a node with degree 1.

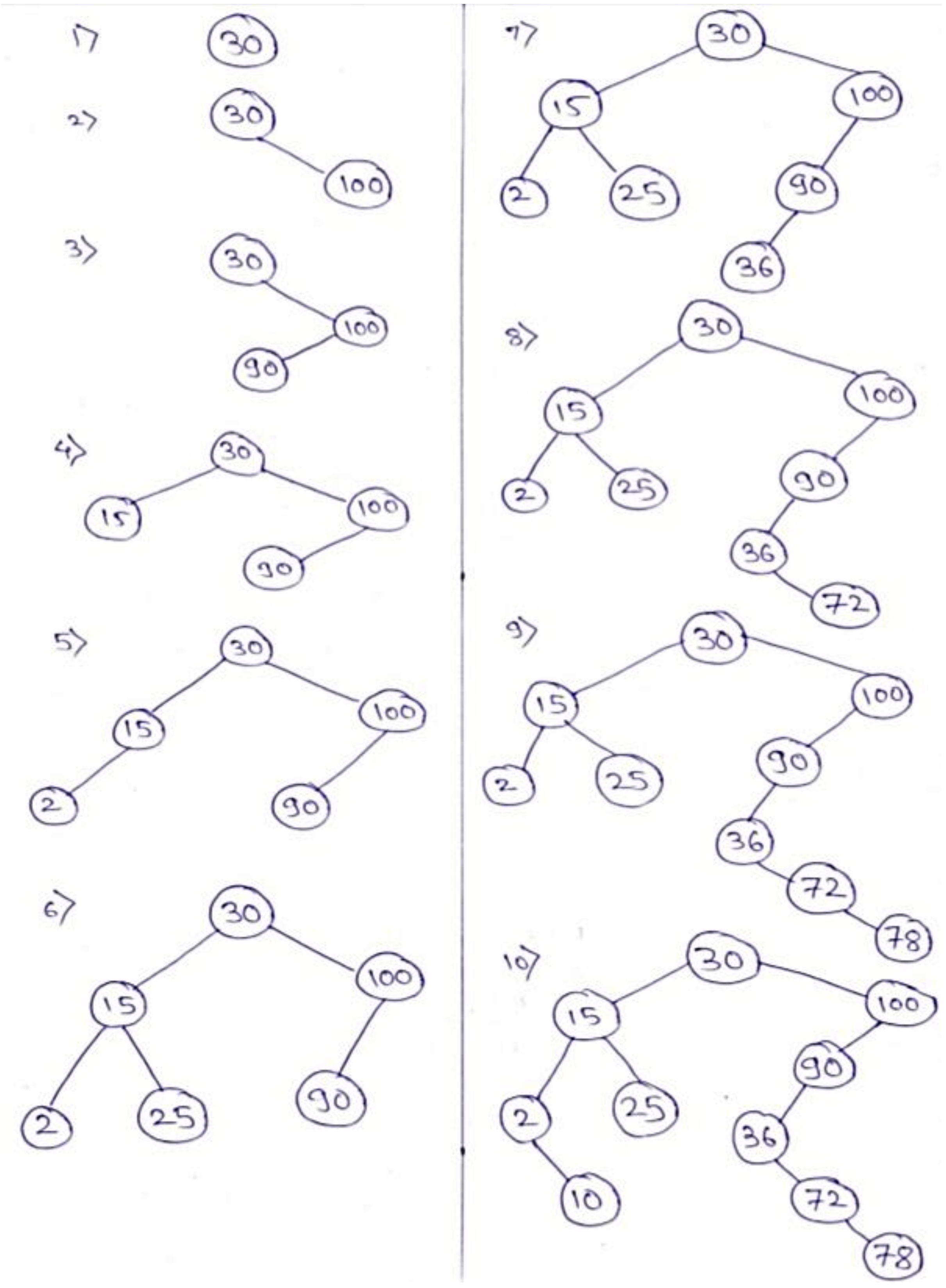


```
BSTnode* delete(BSTnode *root, int x)
{
   //searching for the item to be deleted
   if(root==NULL)
   {
      return NULL;
     }
   if(x>root->data)
   {
      root->right = delete(root->right, x);
   }
   if(x<root->data)
   {
      root->left = delete(root->left, x);
   }
   //No Children
      if(root->left ==NULL && root->right ==NULL)
   {
         free(root);
         return NULL;
   }

   //One Child
   if(root->left ==NULL || root->right ==NULL)
```

```
    {
      BSTnode *temp;
        if(root->left ==NULL)
            temp = root->right;
         else
            temp = root->left;
         free(root);
         return temp;
    }

    //Two Children
    else
    {

         BSTnode *temp = find_min(root->right);
         root->data = temp->data;
         root->right = delete(root->right, temp->data);
      }
    }
    return root;
}

BSTnode* find_min(BSTnode *root)
{
    While(root->left != NULL)
    {
          root=root->left; // left most element will be minimum
    }
    return root;
}

BSTnode* find_max(BSTnode *root)
{
    While(root->right != NULL)
    {
          root=root->right; // left most element will be minimum
    }
    return root;
}
```

Stepwise construction of Binary search tree for following elements:

30,100,90,15,2,25,36,72,78,10



Threaded Binary Tree

In linked representation of tree, there are more number When a binary tree is represented using linked list representation.

If any node is not having a child we use a NULL pointer. These special pointers are threaded and the binary tree having such pointers is called a threaded binary tree.

Thread in a binary tree is represented by a dotted line

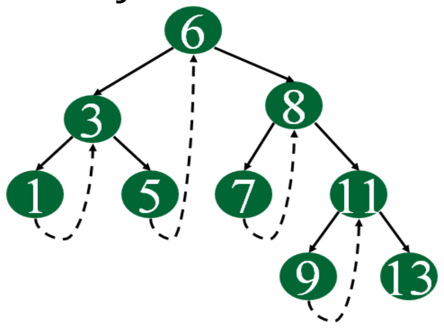
This NULL pointer does not play any role except indicating there is no child.

There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

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

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



Why do we need Threaded Binary Tree?

- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals.
- Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal

Huffman Coding

Huffman coding is a lossless data encoding algorithm

A text message can be converted into a sequence of 0's and 1's by replacing each character of the message with its code.

It uses variable length encoding where variable length codes are assigned to all the characters depending on how frequently they occur in the given text.

The character which occurs most frequently gets the smallest code and the character which occurs least frequently gets the largest code.

Huffman's algorithm

1. Each character is converted into a single node tree.
2. Each node is labelled by its frequency.
3. Arrange all the nodes in the increasing order of frequency value contained in the nodes.
4. Select first two nodes having minimum frequency.
5. Combine these two nodes into one weight of combined tree= sum of weight of two nodes.
6. Keep repeating Step-04 and Step-05 until all the nodes form a single tree.

A file contains the following characters with the frequencies as shown. If Huffman coding is used for data compression, determine-

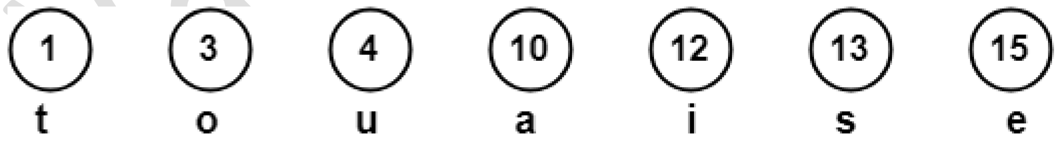
1. Huffman code for each character
2. Average code length
3. Length of Huffman encoded message (in bits)

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

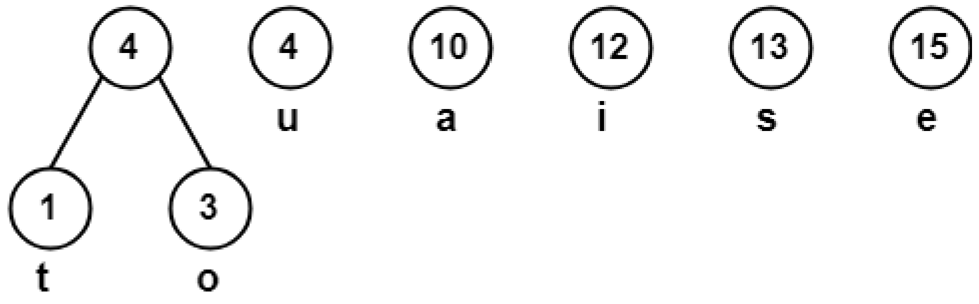
Solution-

First let us construct the Huffman tree using the steps we have learnt above-

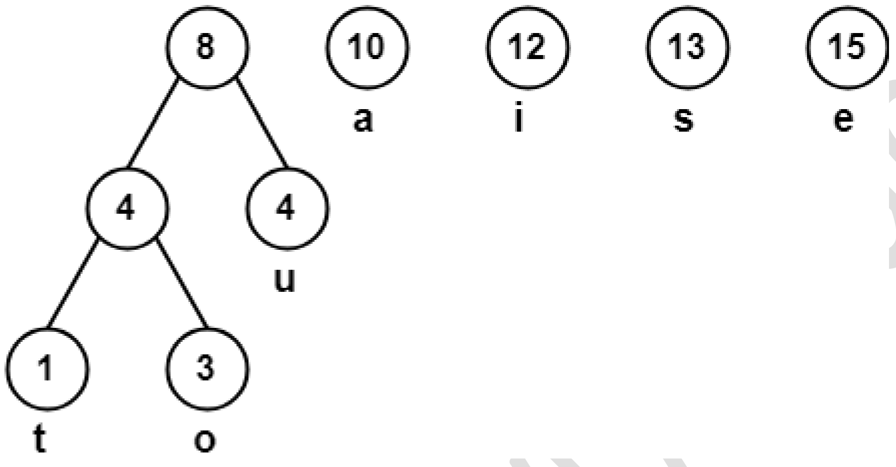
Step-01:



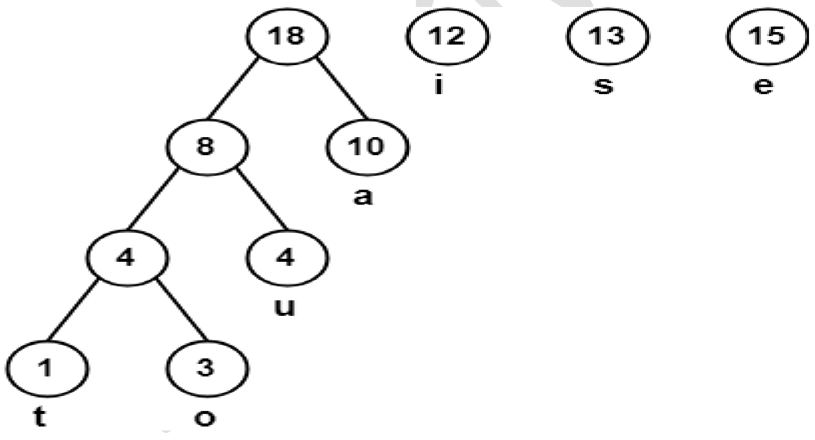
Step-02:



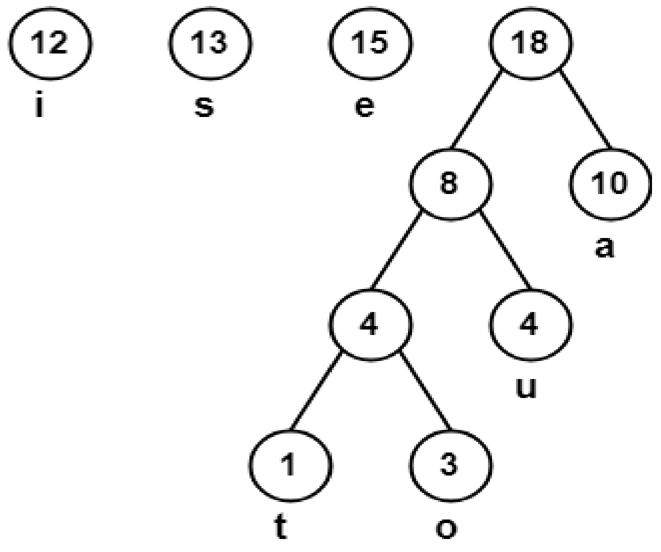
Step-03:

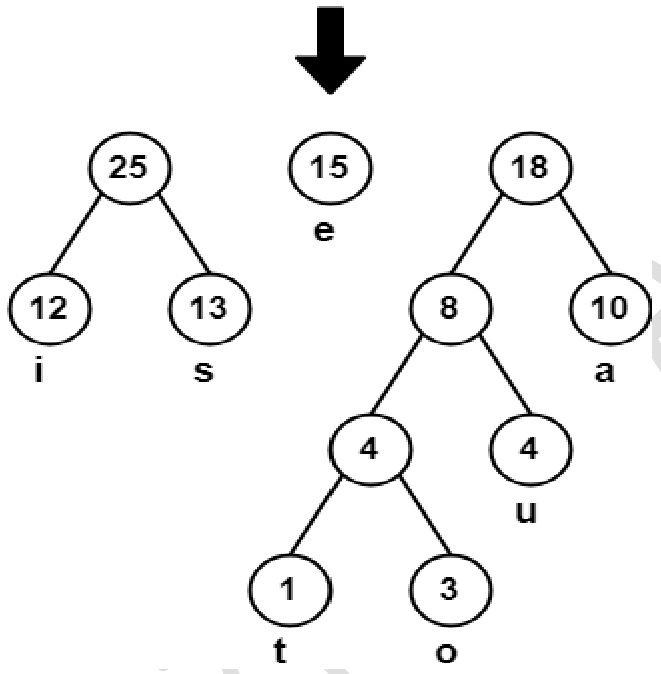


Step-04:

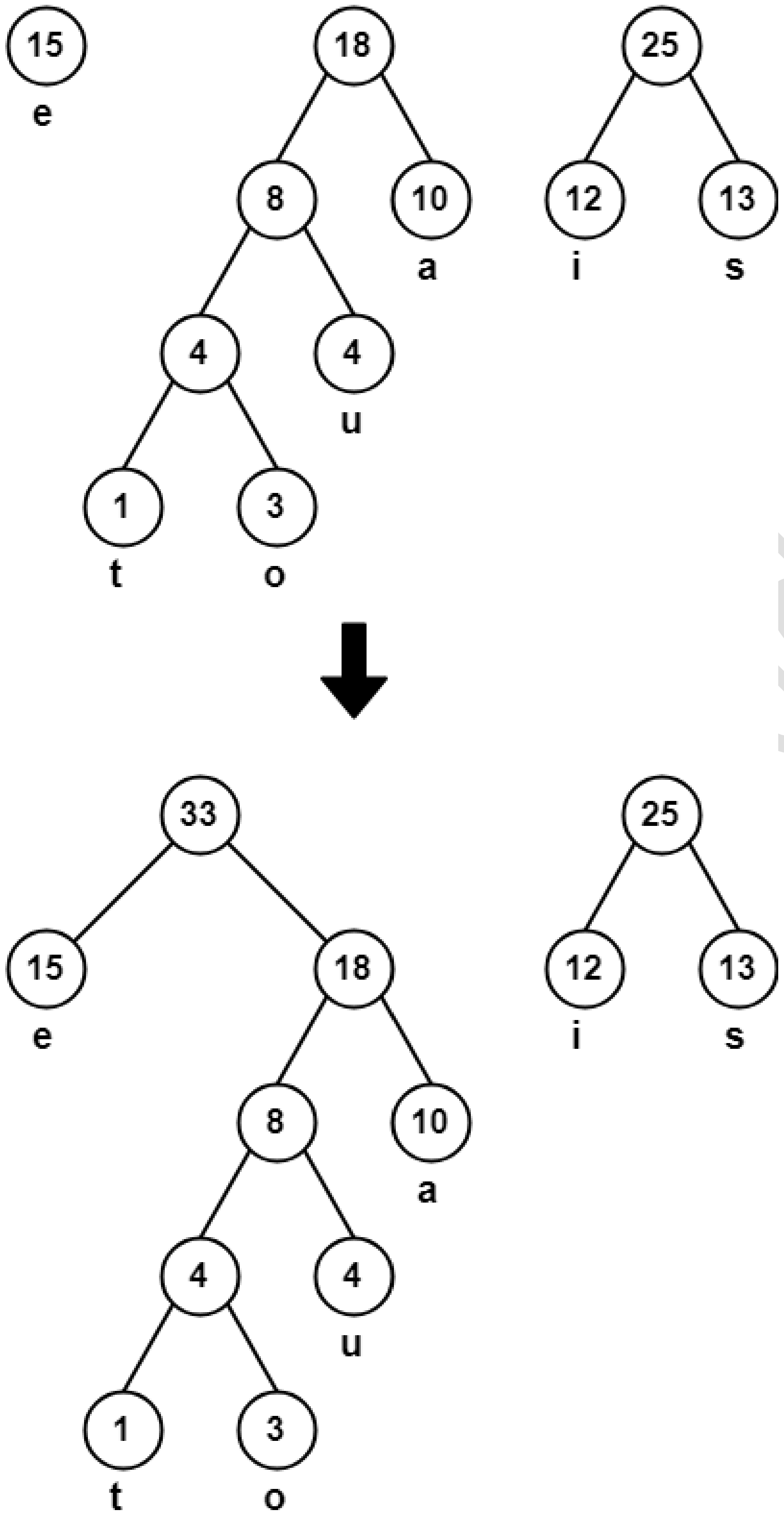


Step-05:

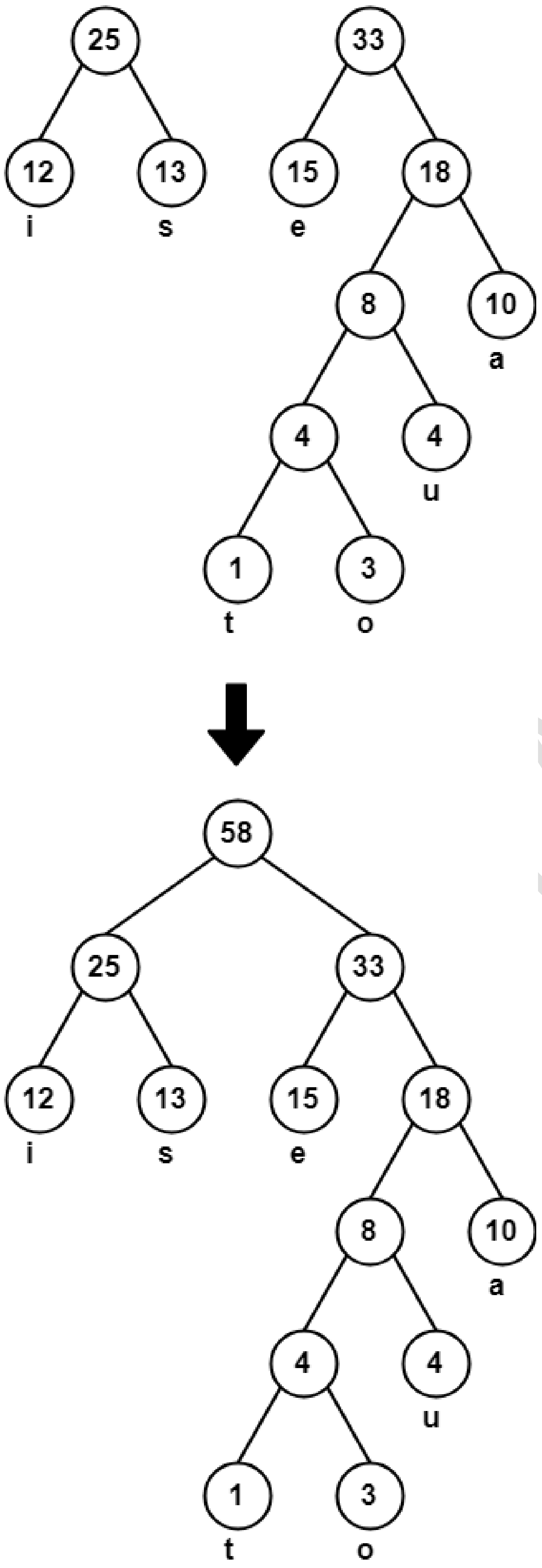

12
i
13
s
15
e
18
8
10
a
4
4
u
1
t
3
o


25
12
i
13
s
15
e
18
8
10
a
4
4
u
1
t
3
o

Step-06:

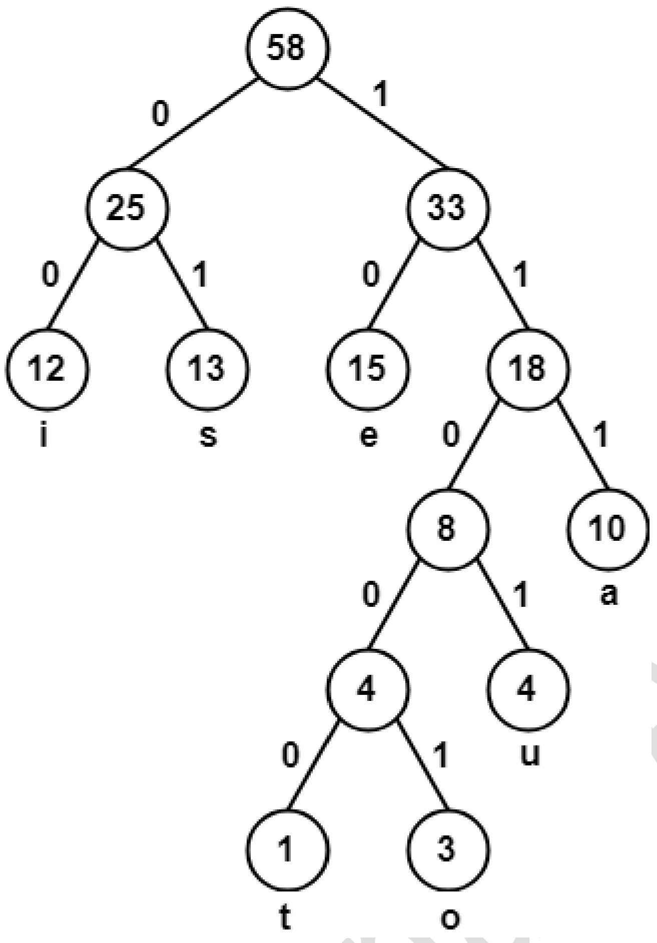


Step 7



Huffman Tree

After we have constructed the Huffman tree, we will assign weights to all the edges. Let us assign weight ‘0’ to the left edges and weight ‘1’ to the right edges



1. Huffman code for the characters-

We will traverse the Huffman tree from the root node to all the leaf nodes one by one and and will write the Huffman code for all the characters-

- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000

2. Average code length-

Average code length

= $\sum$ (frequency_i x code length_i) / $\sum$ (frequency_i)

= { (10 x 3) + (15 x 2) + (12 x 2) + (3 x 5) + (4 x 4) + (13 x 2) + (1 x 5) } / (10 + 15 + 12 + 3 + 4 + 13 + 1)

= 2.52

Length of Huffman encoded message-

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

= 58 x 2.52

= 146.16

AVL TREE

Adelson, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

BalanceFactor=height(leftsutree)-height(right sutree)

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

AVL Rotations

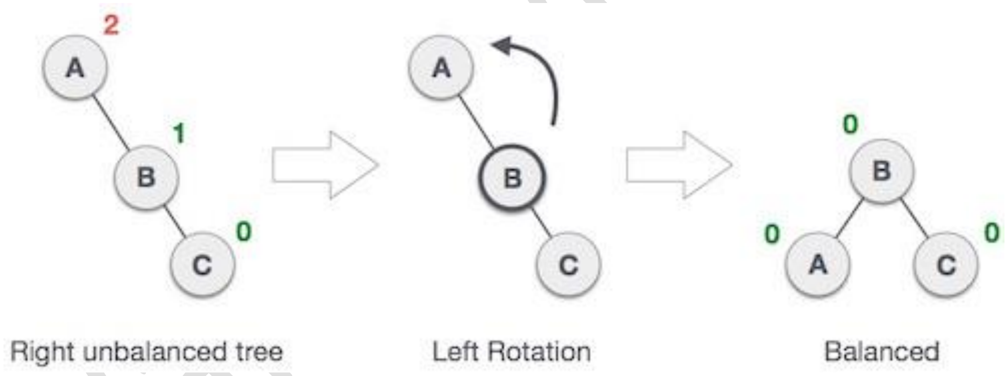
To balance itself, an AVL tree may perform the following four kinds of rotations −

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

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

Left Rotation

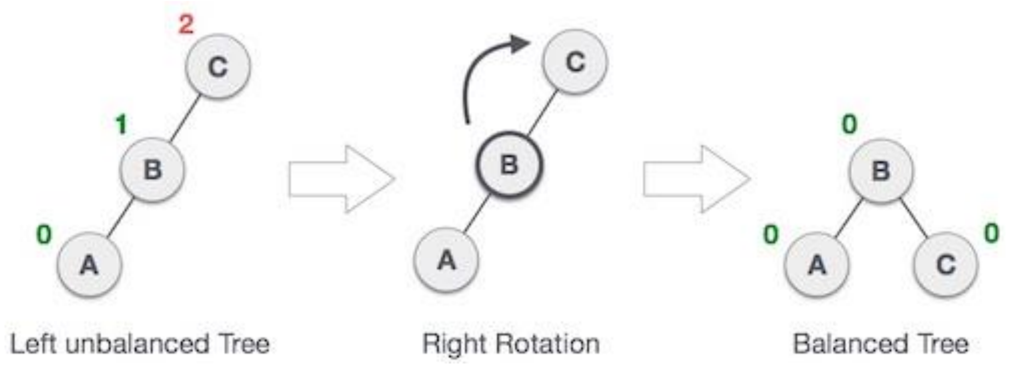
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



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

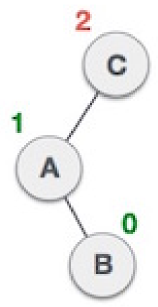
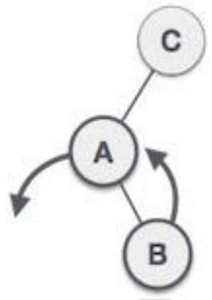
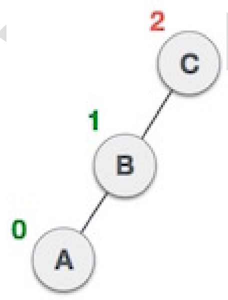
Right Rotation

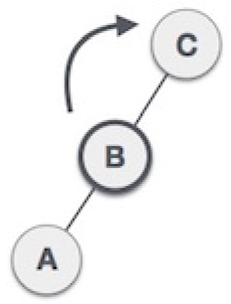
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



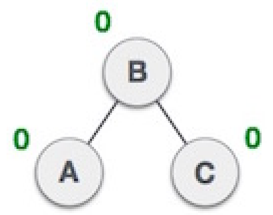
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.



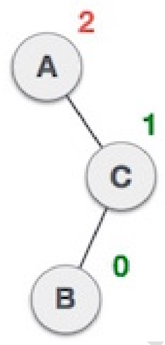
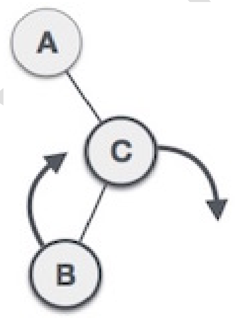
We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

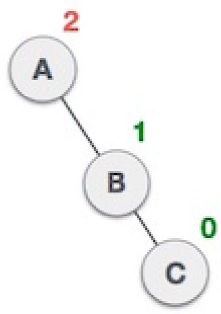


The tree is now balanced.

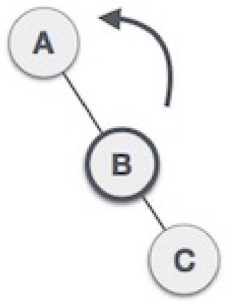
Right-Left Rotation

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

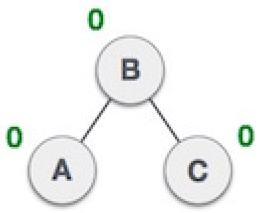
State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.



Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.



A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.



The tree is now balanced.

Expression tree

When expression is represented through a tree, it is known as an expression tree.

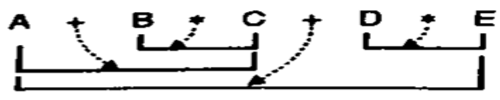
The leaves of an expression tree are operands, such as constants or variables and all internal nodes contains operations.

The preorder traversal produces the prefix representation, the in-order traversal produces the infix representation, and the post-order traversal produces the postfix representation of the expression.

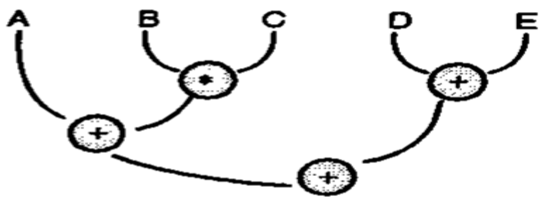
Construct binary tree for the following expression

TREE A + B * C + D * E

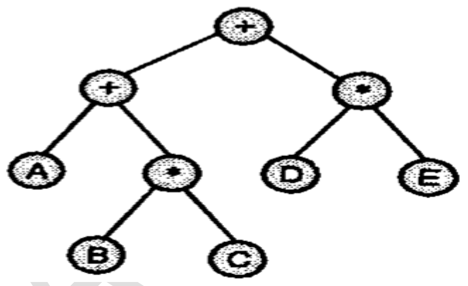
Step 1 : Group elements as per the sequence of evaluation. [This step is s an expression].



Step 2 : Move the operator at the center of the group.



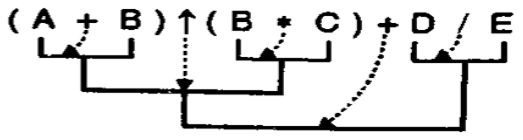
Step 3 : Invert the structure.



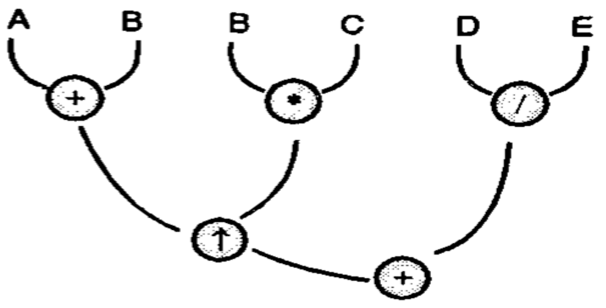
Construct binary tree for following expression.

(A+B)↑(B*C) + D/E

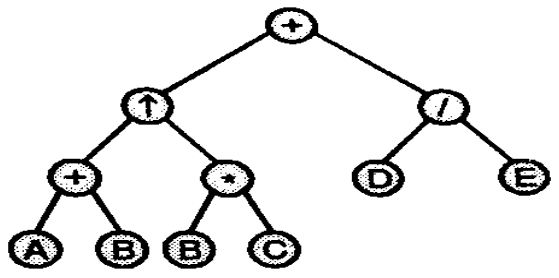
Step 1 : Group elements as per the sequence of evaluation. [This step an expression].



Step 2 : Move the operator at the center of the group.



Step 3 : **Inverting the structure, we get**



B-Trees

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children

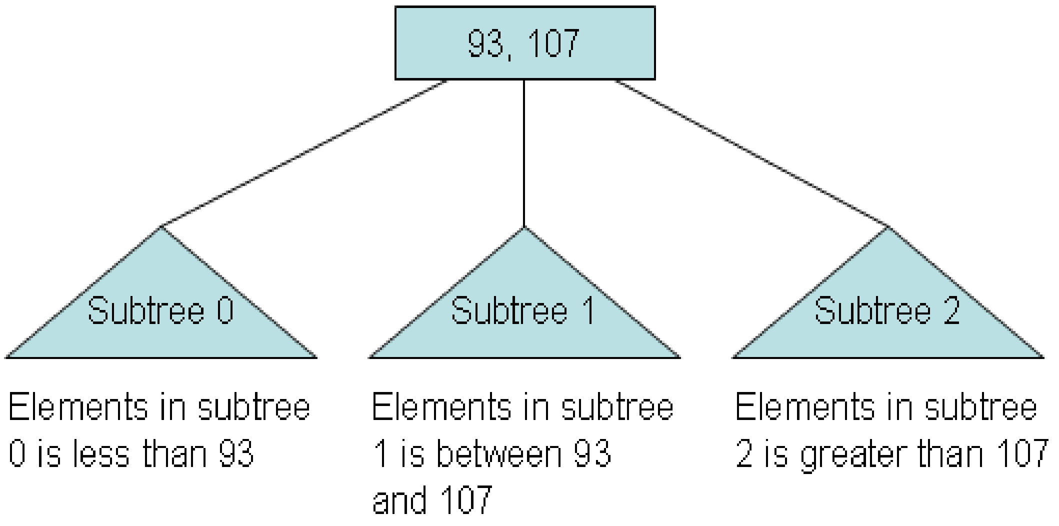
The **B-tree** is a generalization of a binary search **tree** in that a node can have more than two children.

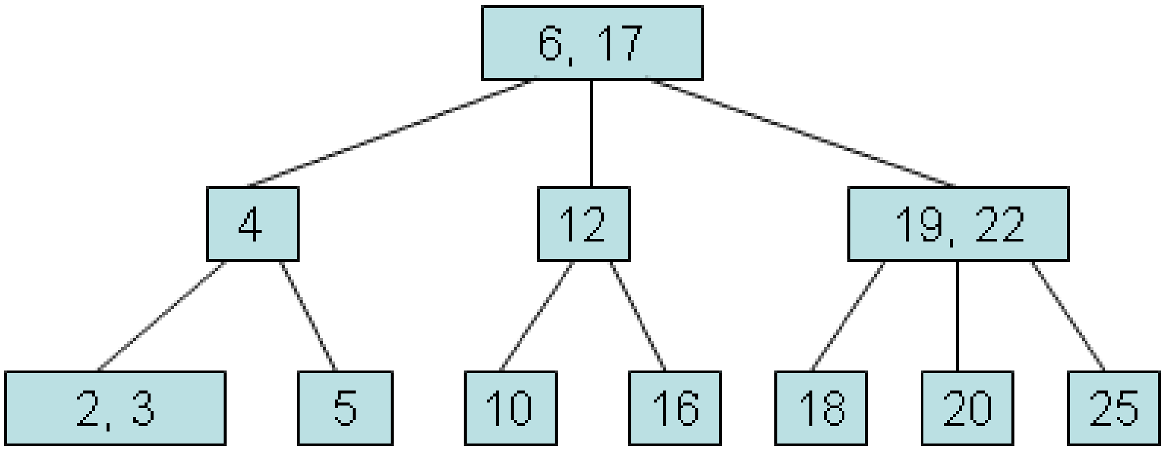
When large volume of data is handled, the search tree can not accommodated in primary, b tree primarily meant for secondary storage.

B-Tree is a M-way tree. And M-way tree can have maximum of M children.

Properties of B-Tree

1. All **leaf nodes** must be **at same level**.
2. All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
3. All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
4. If the root node is a non leaf node, then it must have **atleast 2** children.
5. A non leaf node with **n-1** keys must have **n** number of children.
6. All the **key values in a node** must be in **Ascending Order**.


93, 107
Subtree 0
Subtree 1
Subtree 2
Elements in subtree 0 is less than 93
Elements in subtree 1 is between 93 and 107
Elements in subtree 2 is greater than 107


6, 17
4
12
19, 22
2, 3
5
10
16
18
20
25