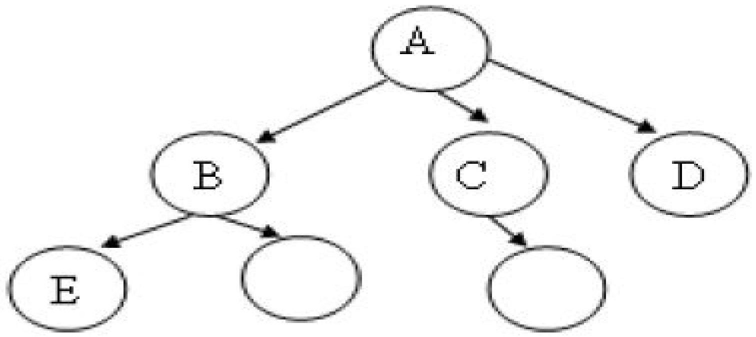
DSA(Lecture#7)

Prepared by Bal Krishna Subedi

Tree:

A tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.

- Tree is a sequence of nodes.
- There is a starting node known as root node.
- Every node other than the root has a parent node.
- Nodes may have any number of children.



A has 3 children, B, C, D
A is parent of B

Some key terms:

Degree of a node:

The degree of a node is the number of children of that node.
In above tree the degree of node A is 3.

Degree of a Tree:

The degree of a tree is the maximum degree of nodes in a given tree.

In the above tree the node A has maximum degree, thus the degree of the tree is 3.

Path:

It is the sequence of consecutive edges from source node to destination node.
There is a single unique path from the root to any node.

Height of a node:

The height of a node is the maximum path length from that node to a leaf node. A leaf node has a height of 0.

Height of a tree:

The height of a tree is the height of the root.

Depth of a node:

Depth of a node is the path length from the root to that node. The root node has a depth of 0.

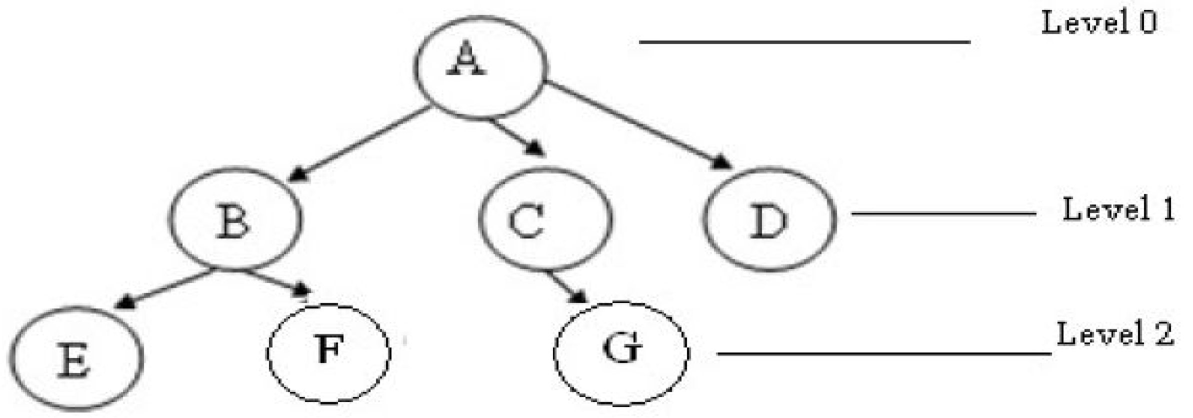
Depth of a tree:

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

Level of a node:

the level of a node is 0, if it is root; otherwise it is one more then its parent.

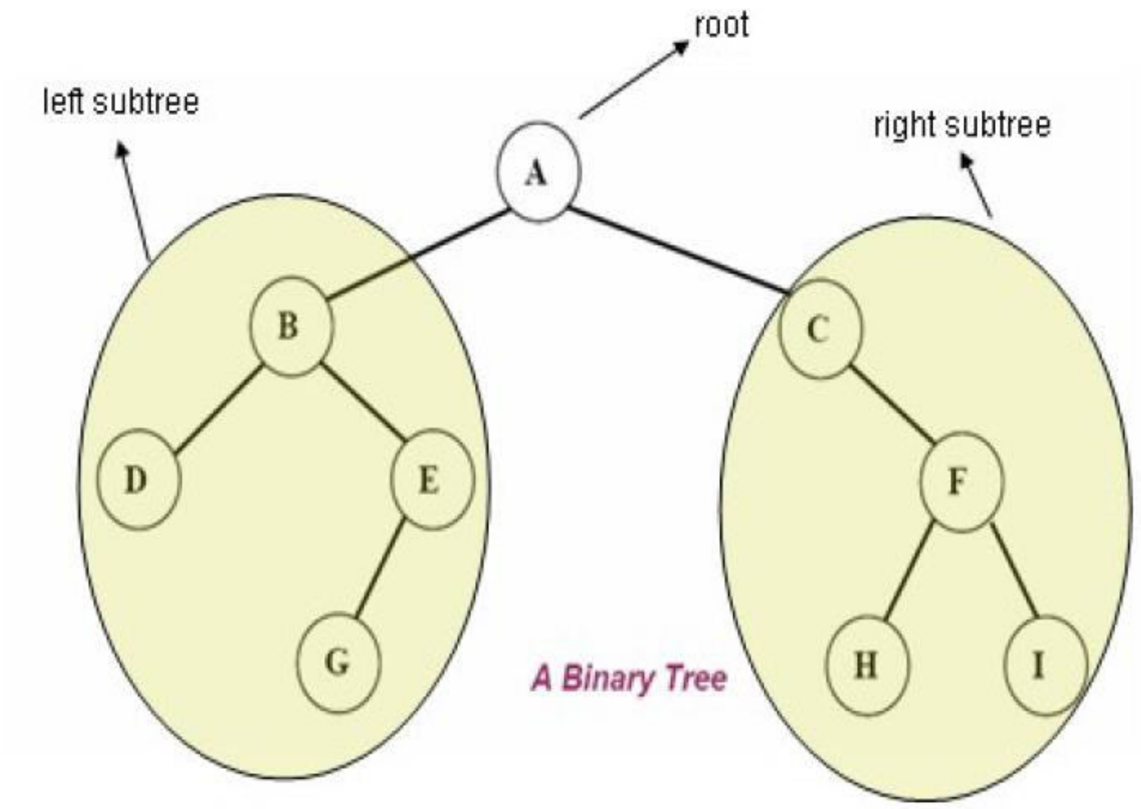
Illustration:



- ✔ A is the root node
- ✔ B is the parent of E and F
- ✔ D is the sibling of B and C
- ✔ E and F are children of B
- ✔ E, F, G, D are external nodes or leaves
- ✔ A, B, C are internal nodes
- ✔ Depth of F is 2
- ✔ the height of tree is 2
- ✔ the degree of node A is 3
- ✔ The degree of tree is 3

two subsets are themselves binary trees called the ***left*** and ***right sub-trees*** of the original tree. A left or right sub tree can be empty.

Each element of a binary tree is called a ***node*** of the tree. The following figure shows a binary tree with 9 nodes where A is the root.



•A **binary tree** consists of a **header**, plus a number of **nodes** connected by **links** in a hierarchical data structure:

Binary tree properties:

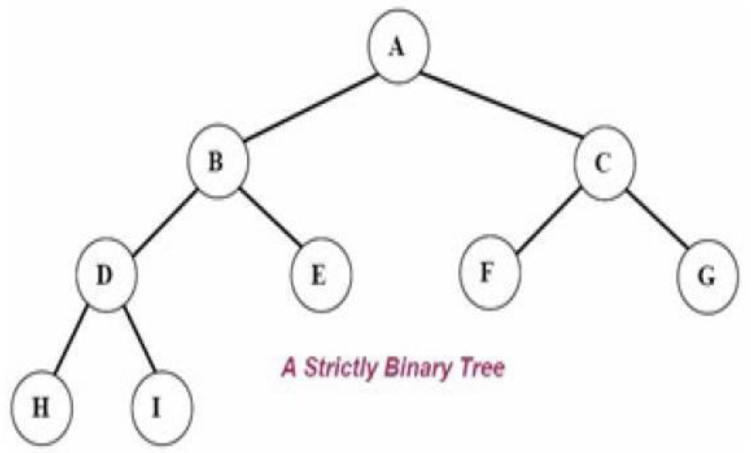
- ✓ If a binary tree contains m nodes at level l, it contains at most **2m** nodes at level **l+1**.
- ✓ Since a binary tree can contain at most 1 node at level 0 (the rot), it contains at most $\mathbf{2^l}$ nodes at level **l.**

Types of binary tree

- ✓ Complete binary tree
- ✓ Strictly binary tree
- ✓ Almost complete binary tree

Strictly binary tree:

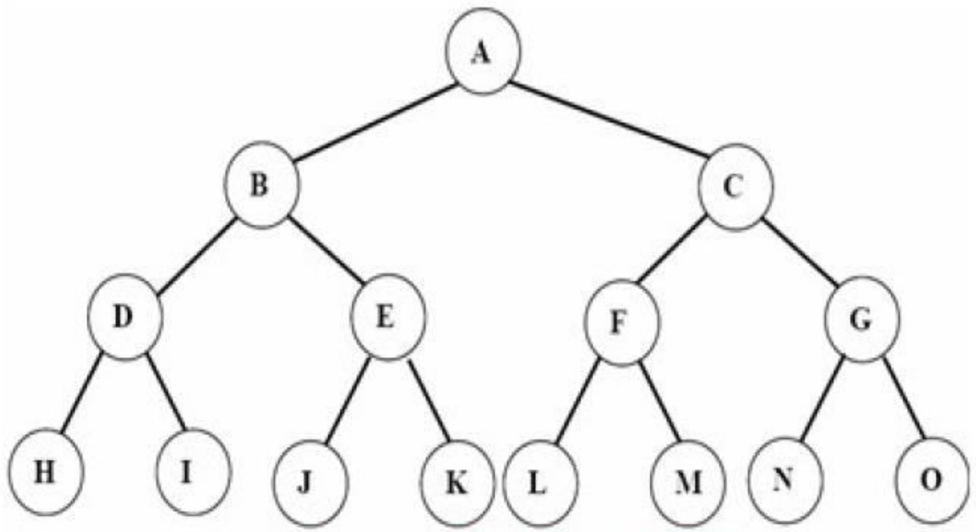
If every non-leaf node in a binary tree has nonempty left and right sub-trees, then such a tree is called a ***strictly binary tree***.



A Strictly Binary Tree

Complete binary tree:

A ***complete binary tree*** of depth d is called strictly binary tree if all of whose leaves are at level **d**. A complete binary tree with depth **d** has $\mathbf{2^d}$ leaves and $\mathbf{2^d}$ **-1** non-leaf nodes(internal)



A complete Binary Tree of depth 3

Almost complete binary tree:

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

- ✓ Any node **nd** at level less than **d-1** has two sons.
- ✓ For any nose **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.

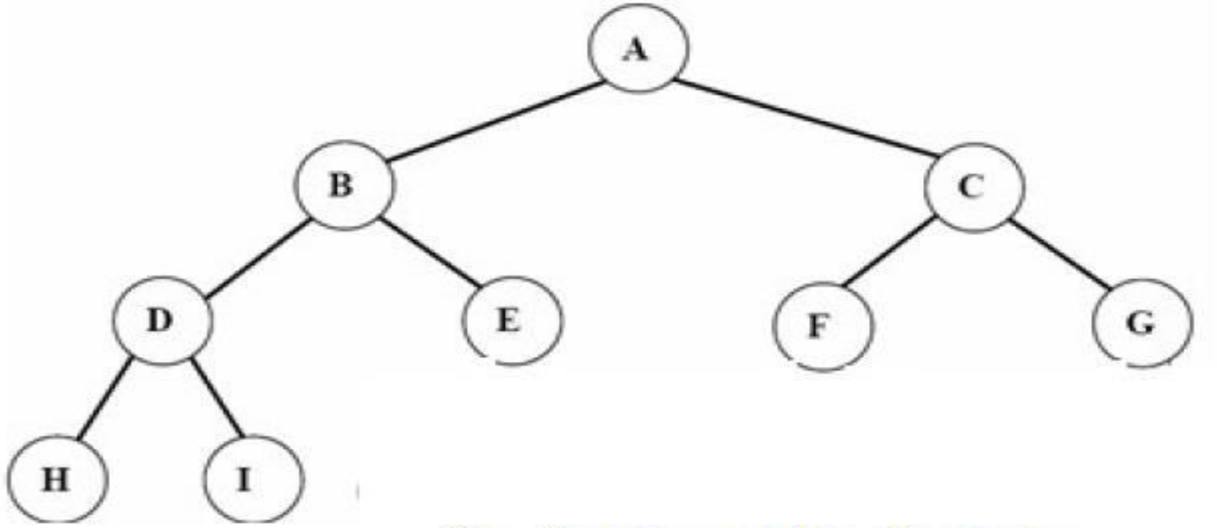


Fig Almost complete binary tree.

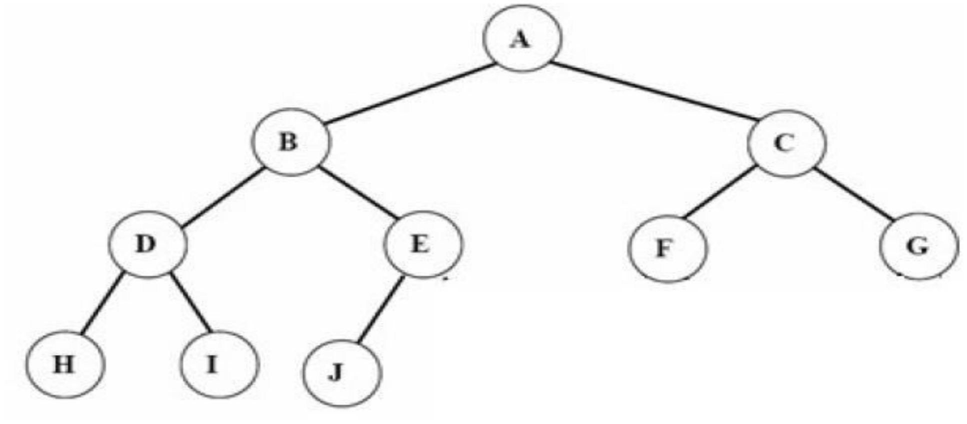


Fig Almost complete binary tree but not strictly binary tree.
Since node E has a left son but not a right son.

Operations on Binary tree:

- ✔ **father(n,T):**Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- ✔ **LeftChild(n,T):**Return the left child of node n in tree T. Return NULL if n does not have a left child.
- ✔ **RightChild(n,T)**:Return the right child of node n in tree T. Return NULL if n does not have a right child.
- ✔ **Info(n,T):** Return information stored in node n of tree T (ie. Content of a node).
- ✔ **Sibling(n,T):** return the sibling node of node **n** in tree T. Return NULL if n has no sibling.
- ✔ **Root(T):** Return root node of a tree if and only if the tree is nonempty.
- ✔ **Size(T):** Return the number of nodes in tree T
- ✔ **MakeEmpty(T):** Create an empty tree T
- ✔ **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T
- ✔ **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.
- ✔ **Preorder(T):** Traverses all the nodes of tree T in preorder.
- ✔ **postorder(T):** Traverses all the nodes of tree T in postorder
- ✔ **Inorder(T):** Traverses all the nodes of tree T in inorder.

C representation for Binary tree:

```
struct bnode
{
        int info;
        struct bnode  *left;
        struct bnode  *right;
};
struct bnode *root=NULL;
```

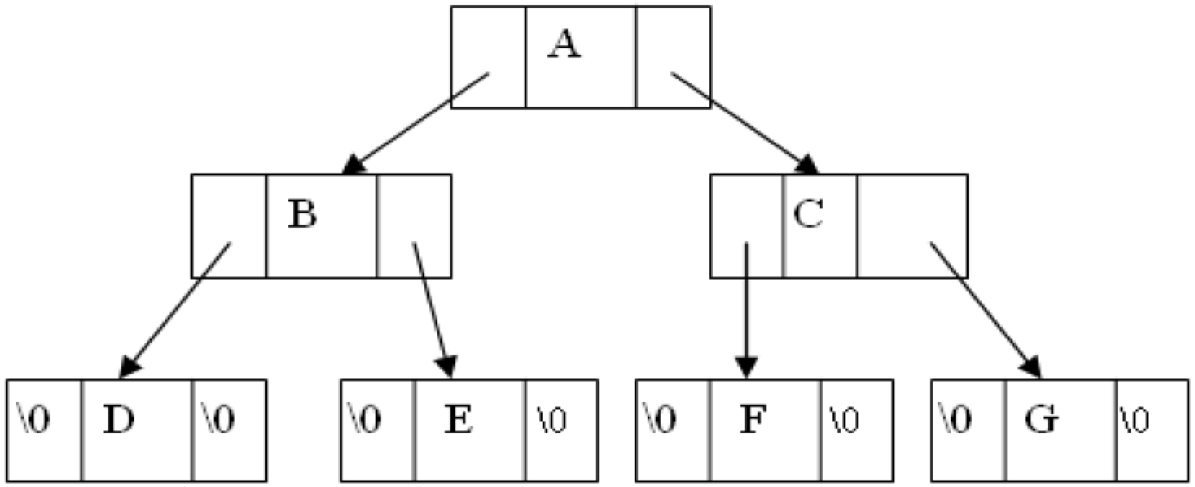


Fig: Structure of Binary tree

Tree traversal:

The tree traversal is a way in which each node in the tree is visited exactly once in a symmetric manner.
There are three popular methods of traversal

- ✔ Pre-order traversal
- ✔ In-order traversal
- ✔ Post-order traversal

Pre-order traversal:

The preorder traversal of a nonempty binary tree is defined as follows:

- ✔ Visit the root node
- ✔ Traverse the left sub-tree in preorder
- ✔ Traverse the right sub-tree in preorder

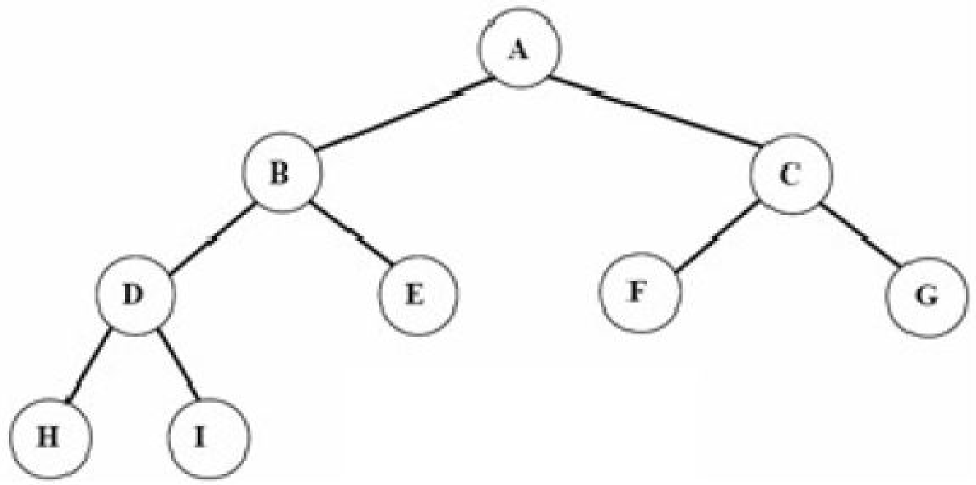


fig Binary tree

The preorder traversal output of the given tree is: A B D H I E C F G

The preorder is also known as depth first order.

C function for preorder traversing:

```
void preorder(struct bnode *root)
{
    if(root!=NULL)
    {
        printf("%c", root->info);
        preorder(root->left);
        preorder(root->right);
    }
}
```

In-order traversal:

The inorder traversal of a nonempty binary tree is defined as follows:

- ✓ Traverse the left sub-tree in inorder
- ✓ Visit the root node
- ✓ Traverse the right sub-tree in inorder

The inorder traversal output of the given tree is: H D I B E A F C G

C function for inorder traversing:

```
void inorder(struct bnode *root)
 {
     if(root!=NULL)
     {
        inorder(root->left);
        printf("%c", root->info);
        inorder(root->right);
     }
  }
```

Post-order traversal:

The post-order traversal of a nonempty binary tree is defined as follows:

- ✔ Traverse the left sub-tree in post-order
- ✔ Traverse the right sub-tree in post-order
- ✔ Visit the root node

The post-order traversal output of the given tree is: H I D E B F G C A

C function for post-order traversing:

```
void post-order(struct bnode *root)
 {
     if(root!=NULL)
     {
         post-order(root->left);
         post-order(root->right);
         printf("%c", root->info);
     }
  }
```

Binary search tree(BST):

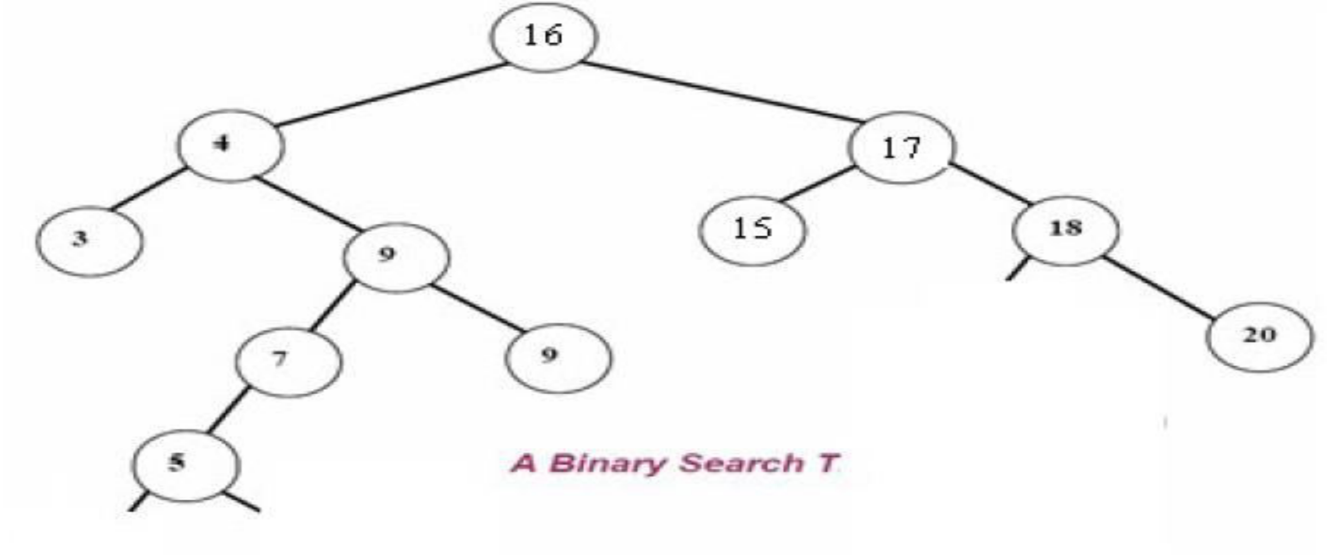
A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) ans satisfies the following conditions:

- ✔ All keys in the left sub-tree o the root are smaller than the key in the root node
- ✔ All keys in the right sub-tree of the root are greater than the key in the root node
- ✔ The left and right sub-trees of the root are again binary search trees

Given the following sequence of numbers,

14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

The following binary search tree can be constructed:



Operations on Binary search tree(BST):

Following operations can be done in BST:

- ✓ **Search(k, T):** Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
- ✓ **Insert(k, T):** Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.
- ✓ **Delete(k, T):**Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
- ✓ **FindMin(T), FindMax(T):** Find minimum and maximum element from the given nonempty BST.

Searching through the BST:

- *Problem: Search for a given target value in a BST.*
- ***Idea: Compare the target value with the element in the root node.***
  - ✓ If the target value is **equal**, the search is successful.
  - ✓ If target value is **less**, search the left subtree.
  - ✓ If target value is **greater**, search the right subtree.
  - ✓ If the subtree is **empty**, the search is unsuccessful.

BST search algorithm:

To find which if any node of a BST contains an element equal to *target*:

```
1. Set curr to the BST's root.
2. Repeat:
        2.1. If curr is null:
                2.1.1. Terminate with answer none.
        2.2. Otherwise, if target is equal to curr's element:
                2.2.1. Terminate with answer curr.
        2.3. Otherwise, if target is less than curr's element:
                2.3.1. Set curr to curr's left child.
        2.4. Otherwise, if target is greater than curr's element:
                2.4.1. Set curr to curr's right child.

2. end
```

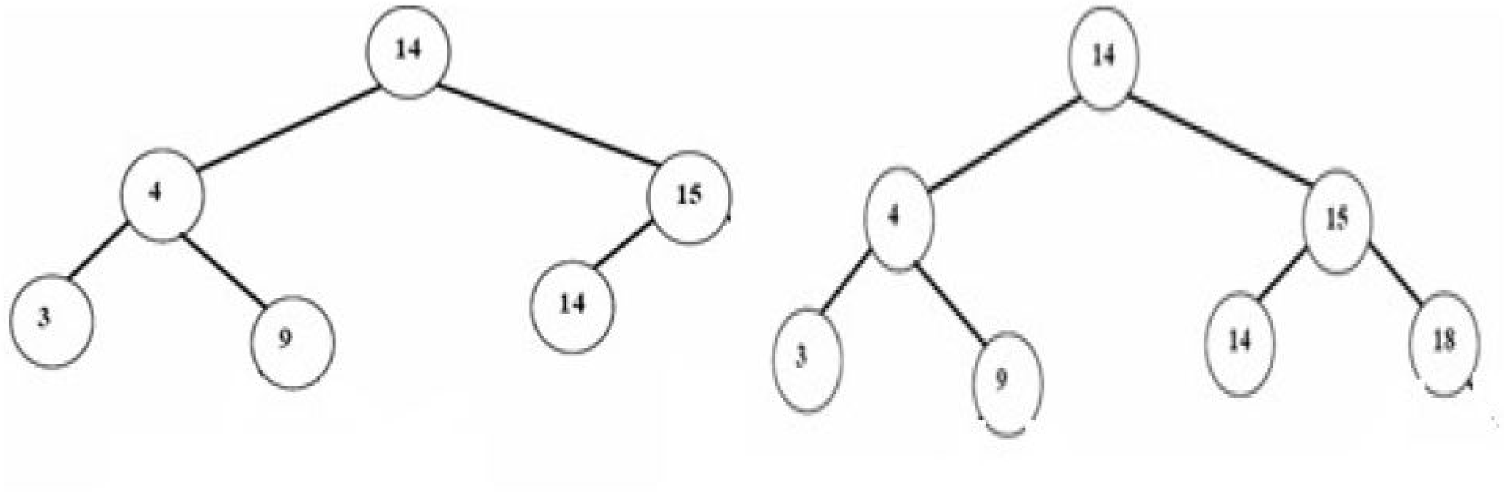
C function for BST searching:

```
void BinSearch(struct bnode *root , int key)
{
        if(root == NULL)
        {
                printf("The number does not exist");
                exit(1);
        }
        else  if (key == root->info)
        {
                printf("The searched item is found"):
        }
        else if(key < root->info)
                return BinSearch(root->left, key);
        else
                return BinSearch(root->right, key);
}
```

Insertion of a node in BST:

To insert a new item in a tree, we must first verify that its key is different from those of existing elements. To do this a search is carried out. If the search is unsuccessful, then item is inserted.

•***Idea***: To insert a new element into a BST, proceed as if searching for that element. If the element is not already present, the search will lead to a null link. Replace that null link by a link to a leaf node containing the new element.



insert(18)

BST insertion algorithm:

```
To insert the element elem into a BST:
    1. Set parent to null, and set curr to the BST's root.
    2. Repeat:
        2.1. If curr is null:
            2.1.1. Replace the null link from which curr was taken
            (either the BST's root or parent's left child or parent's right child) by a
            link to a newly-created leaf node with element elem.
            2.1.2. Terminate.
        2.2. Otherwise, if elem is equal to curr's element:
            2.2.1. Terminate.
        2.3. Otherwise, if elem is less than curr's element:
            2.3.1. Set parent to curr, and set curr to curr's left child.
        2.4. Otherwise, if elem is greater than curr's element:
            2.4.1. Set parent to curr, and set curr to curr's right child.
    3.End
```

C function for BST insertion:

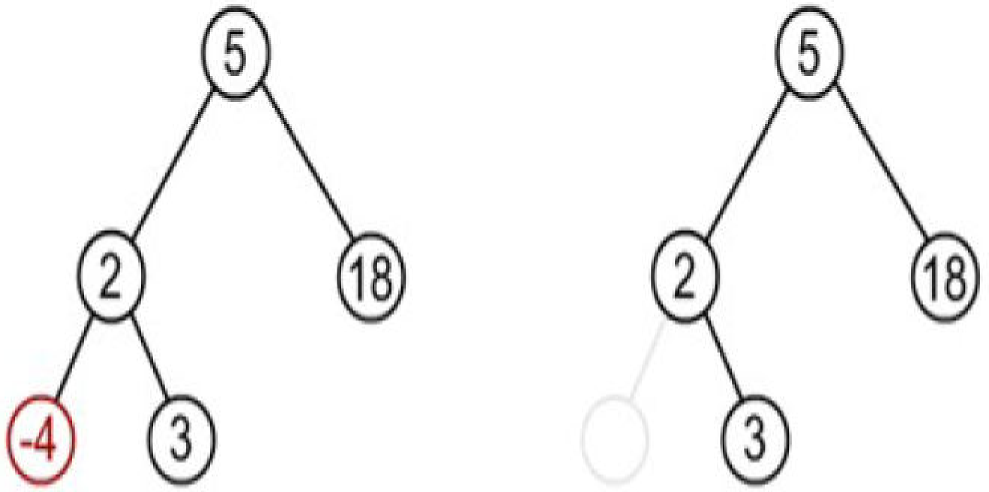
```
void insert(struct bnode *root, int item)
{
        if(root=NULL)
        {
                root=(struct bnode*)malloc (sizeof(struct bnode));
                root->left=root->right=NULL;
                root->info=item;
        }
        else
        {
                if(item<root->info)
                        root->left=insert(root->left, item);
                else
                        root->right=insert(root->right, item);
        }
}
```

Deleting a node from the BST:

While deleting a node from BST, there may be three cases:

1. ***The node to be deleted may be a leaf node:***

   In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.

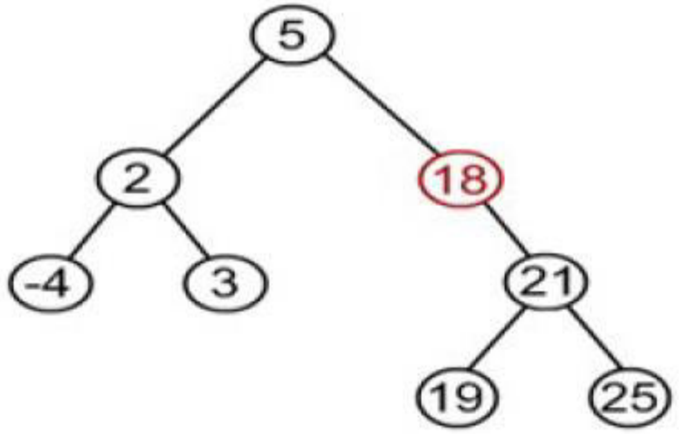


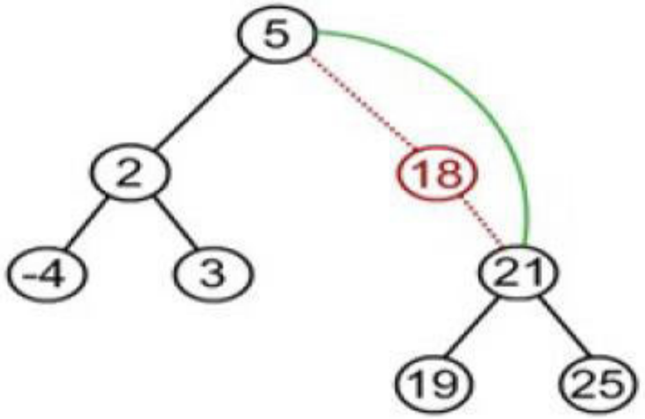
Suppose node to be deleted is -4

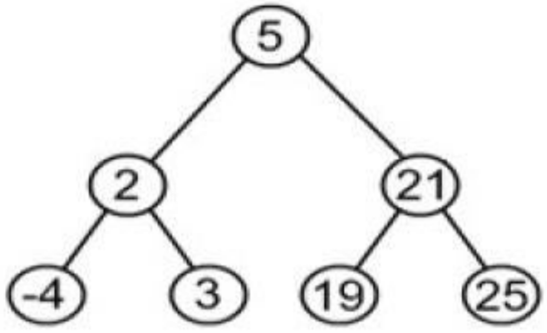
2. **The node to be deleted has one child:**

In this case the child of the node to be deleted is appended to its parent node.

Suppose node to be deleted is 18





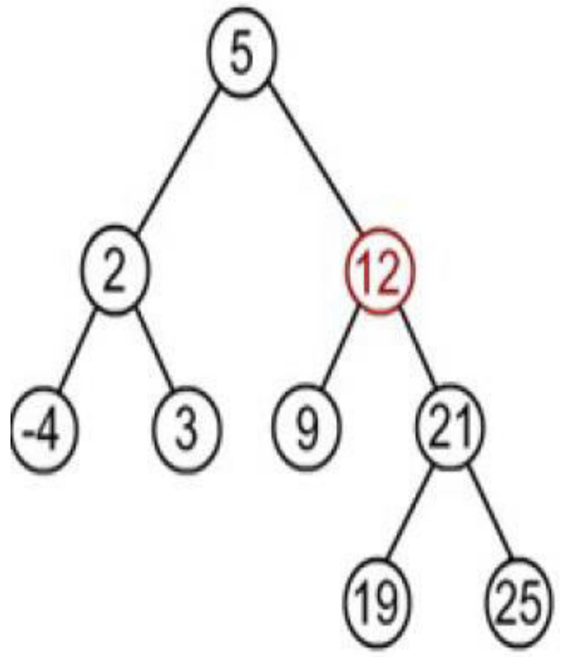


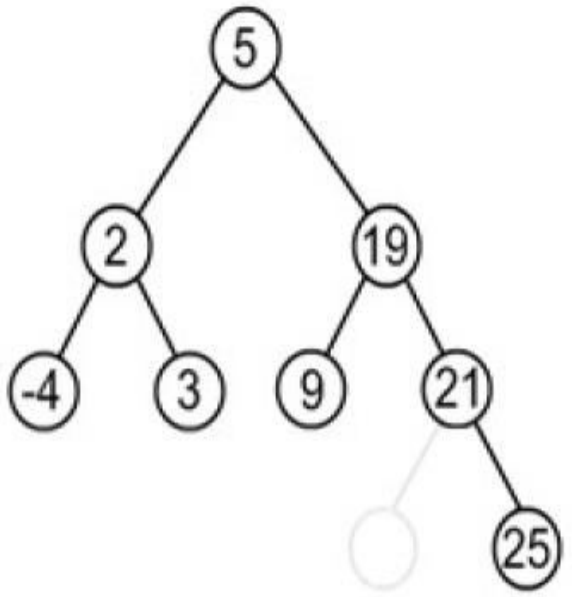
3. the node to be deleted has two children:

In this case node to be deleted is replaced by its in-order successor node.

OR

If the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.





Suppose node to deleted is 12
Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.
[illegible]

General algorithm to delete a node from a BST:

1. start
2. if a node to be deleted is a leaf nod at left side then simply delete and set null pointer to it's parent's left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer
4. if a node to be deleted has on child then connect it's child pointer with it's parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
   a. right most node of it's left sub-tree or
   b. left most node of it's right sub-tree.
6. End

The deleteBST function:

```
struct bnode *delete(struct bnode *root, int item)
{
    struct bnode *temp;
    if(root==NULL)
    {
        printf("Empty tree");
        return;
    }
    else if(item<root->info)
        root->left=delete(root->left, item);
    else if(item>root->info)
        root->right=delete(root->right, item);
    else if(root->left!=NULL &&root->right!=NULL)  //node has two child
    {
        temp=find_min(root->right);
        root->info=temp->info;
        root->right=delete(root->right, root->info);

    }
    else
    {
        temp=root;
        if(root->left==NULL)
                root=root->right;
        else if(root->right==NULL)
                root=root->left;
        free(temp);
    }
    return(temp);
}
/**********find minimum element function**********/
struct bnode *find_min(struct bnode *root)
{
        if(root==NULL)
                return0;
        else if(root->left==NULL)
                return root;
        else
                return(find_min(root->left));
}
```

Huffman algorithm:

Huffman algorithm is a method for building an extended binary tree with a minimum weighted path length from a set of given weights.

-This is a method for the construction of minimum redundancy codes .

•Applicable to many forms of data transmission

Our example: text files

-1951, David Huffman found the "most efficient method of representing numbers, letters, and other symbols using binary code". Now standard method used for data compression.

In Huffman Algorithm, a set of nodes assigned with values if fed to the algorithm. Initially 2 nodes are considered and their sum forms their parent node. When a new element is considered, it can be added to the tree. Its value and the previously calculated sum of the tree are used to form the new node which in turn becomes their parent.

Let us take any four characters and their frequencies, and *sort* this list by increasing frequency

Since to represent 4 characters the 2 bit is sufficient thus take initially two bits for each character this is called fixed length character.

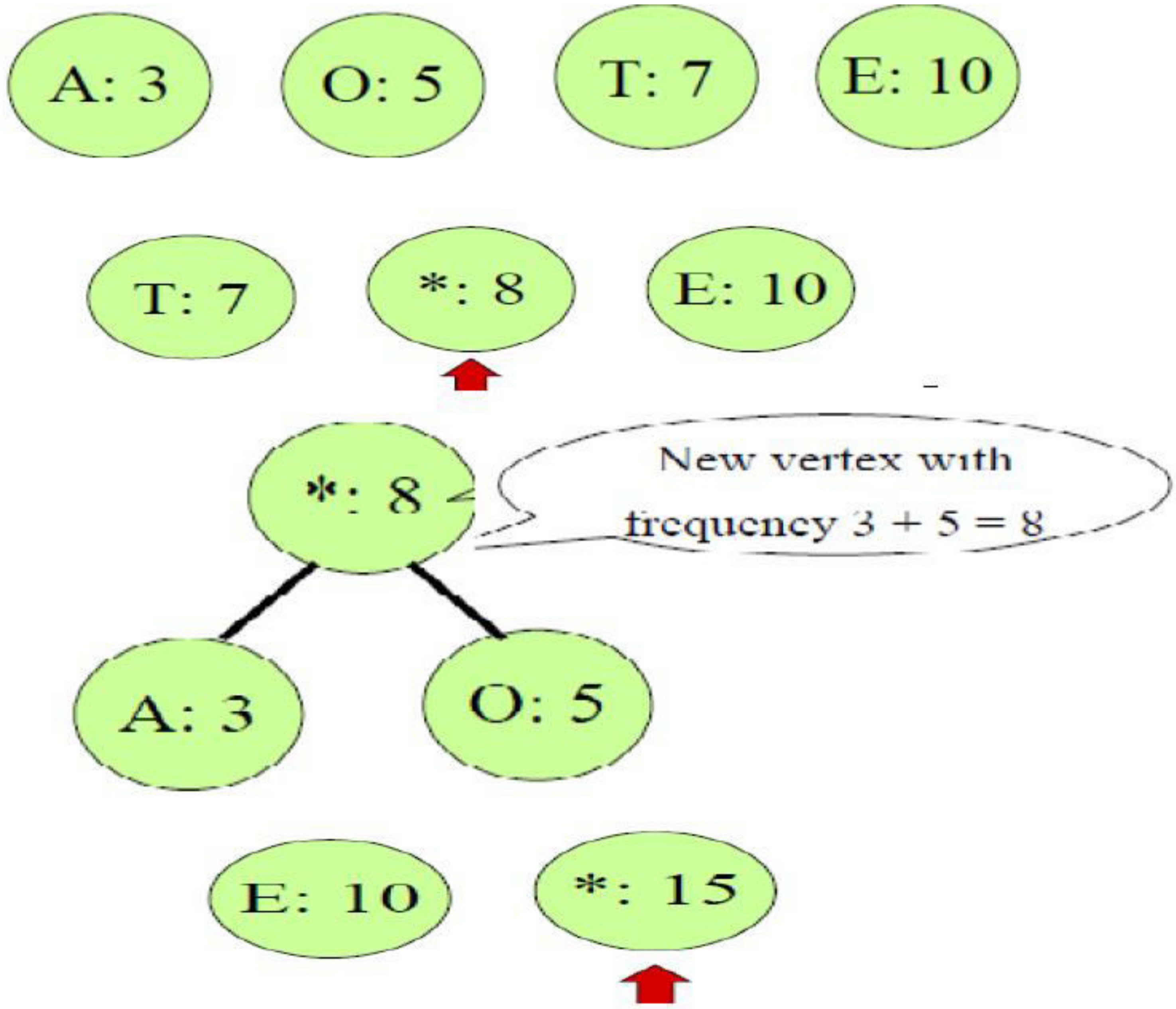
character	***frequencies***
E:	10
T:	07
O:	05
A:	03

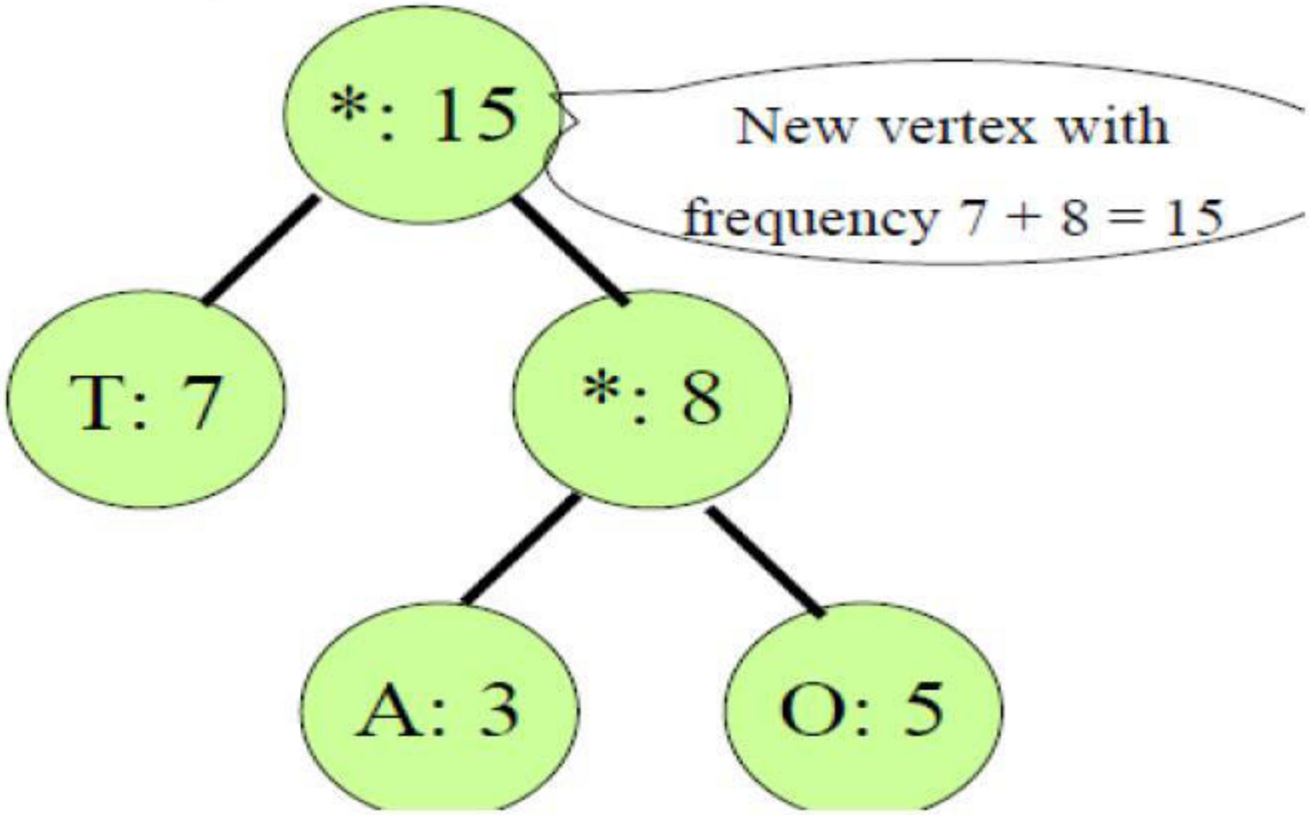
Now sort these characters according to their frequencies in non-decreasing order.

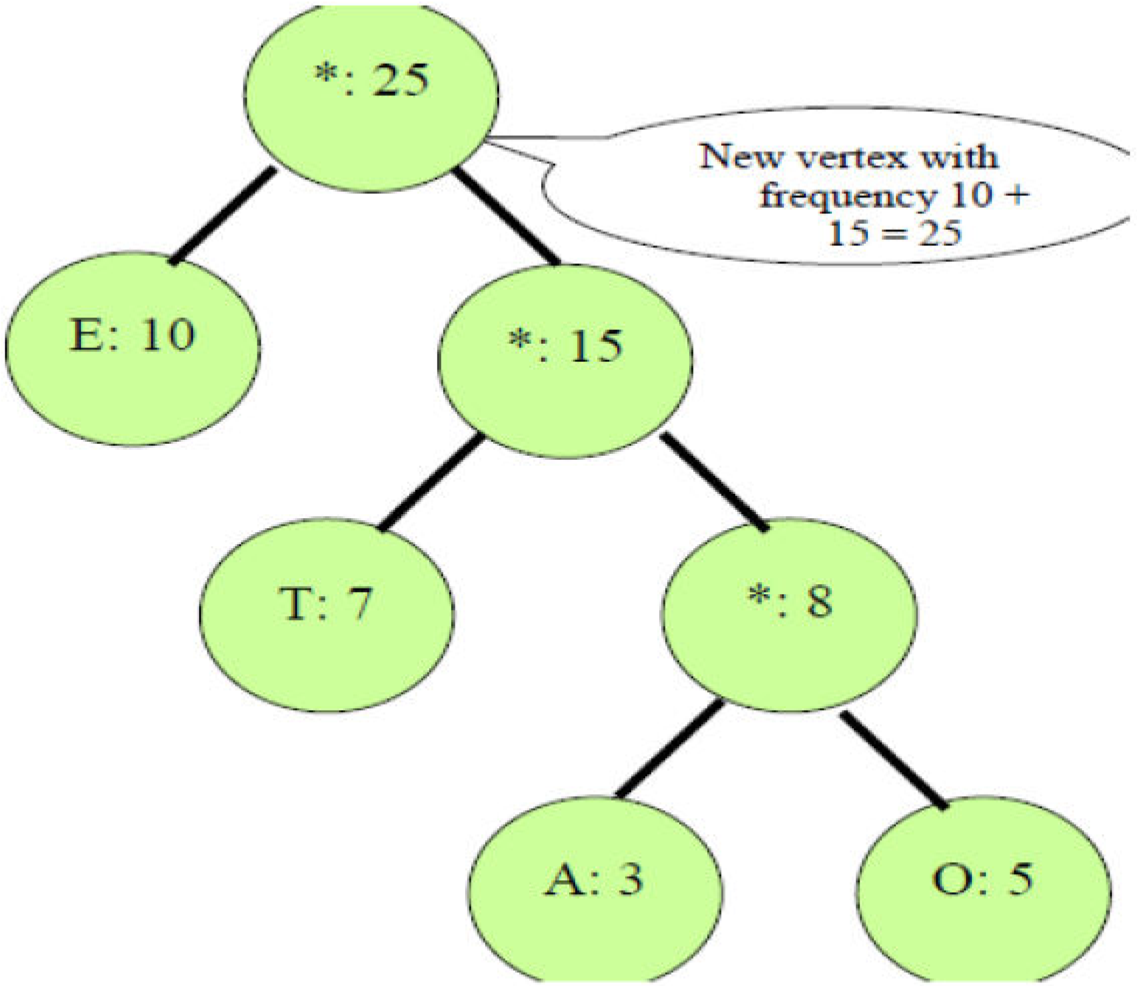
character	***frequencies***	***code***
A:	03	00
O:	05	01
T:	07	10
E:	10	11

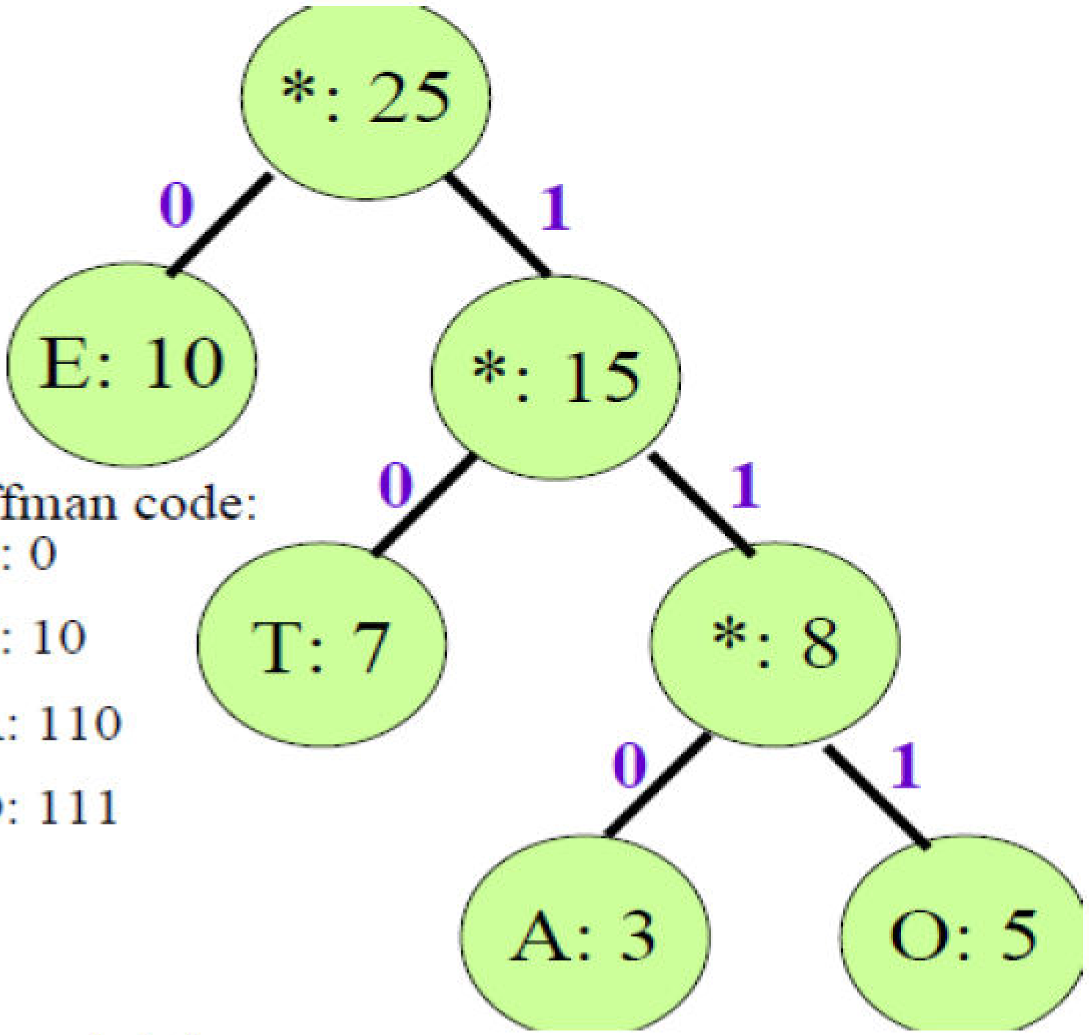
Here before using Huffman algorithm the total number of bits required is

nb=3*2+5*2+7*2+10*2=06+10+14+20=50bits


A: 3
O: 5
T: 7
E: 10
T: 7
*: 8
E: 10
*: 8
New vertex with frequency 3 + 5 = 8
A: 3
O: 5
E: 10
*: 15


*: 15
New vertex with frequency 7 + 8 = 15
T: 7
*: 8
A: 3
O: 5


*: 25
New vertex with frequency 10 + 15 = 25
E: 10
*: 15
T: 7
*: 8
A: 3
O: 5



Huffman code:

E: 0

T: 10

A: 110

O: 111

Left branch is 0
Right branch is 1

Now from variable length code we get following code sequence.

character	***frequencies***	***code***
A:	03	110
O:	05	111
T:	07	10
E:	10	0

Thus after using Huffman algorithm the total number of bits required is

nb=3*3+5*3+7*2+10*1=09+15+14+10=48bits

(50-48)/50*100%=4%

Since in this small example we save about 4% space by using Huffman algorithm. If we take large example with a lot of characters and their frequencies we can save a lot of space.

Thanks You

Any Queies