Trees الاشجار

Trees: Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

Tree Vocabulary: The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, a is a child of f and f is the parent of a. Finally, elements with no children are called leaves.

```
      tree
      ----
       j    <-- root
     /   \
    f     k
  /   \     \
 a     h     z    <-- leaves
```

Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

```
file system
-----------
     root
  /      \
...       home
      /          \
   upgrade      course
    /       /     |     \
  ...      cs101  cs112  cs113
```

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).

3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).

4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Main applications of trees include:

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.

Binary Tree: A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Binary Tree Representation in C++: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.
A Tree node contains following parts.
1. Data
2. Pointer to left child
3. Pointer to right child
In C++, we can represent a tree node using structures. Below is an example of a tree node with an integer data.

Definition:

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

Create Tree in c++:

```
node* newnode(int data)
{

   node* node = (struct node*)malloc(sizeof(struct node));
   node->data = data;

  node->left  = NULL;
  node->right = NULL;
  return(node);
}
```

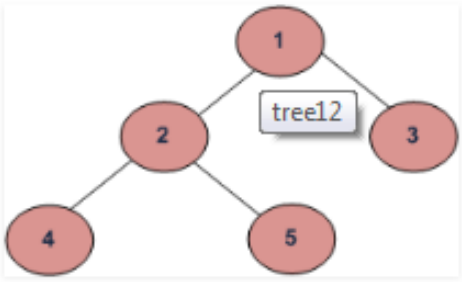
Size of Tree:

To computes the number of nodes in a tree:

```
int size( node* node)
{
  if (node==NULL)
    return 0;
  else
    return(size(node->left) + 1 + size(node->right));
}
```

Tree Traversals (Inorder , Preorder and Postorder):

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. Following are the generally used ways for traversing trees.



Example Tree

Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

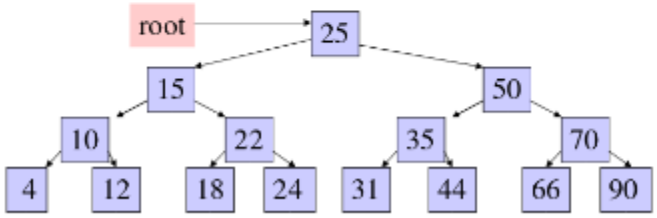
Breadth First or Level Order Traversal : 1 2 3 4 5

Example:

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25




```
void printPostorder( node* node)
{
     if (node == NULL)
        return;

     printPostorder(node->left);

     printPostorder(node->right);

     cout<<"  "<< node->data;
}

/ *-------------------------------*/

void printInorder( node* node)
{
     if (node == NULL)
          return;

     printInorder(node->left);

     cout<<"  "<< node->data;

     printInorder(node->right);
}

/*-------------------------------*/

void printPreorder( node* node)
{
     if (node == NULL)
          return;
```

```
      cout<<"  "<< node->data;

      printPreorder(node->left);

      printPreorder(node->right);
}
```

Program to Create Simple Tree in C++:

Let us create a simple tree with 4 nodes in C++. The created tree would be as following.

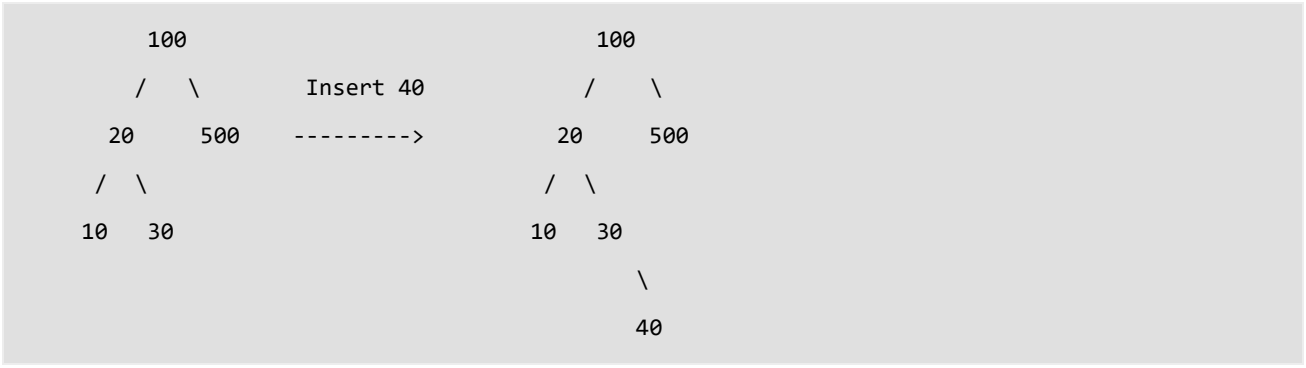
```
      tree
      ----
       1    <-- root
     /   \
    2     3
   /  \
  4     5
```

```
#include<iostream>

using namespace std;

struct node
{
    int data;
     node *left;
     node *right;
};

/*-------------------------*/
 node* newnode(int data)
{
  node* node = (struct node*)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return(node);
}

/*-------------------------*/
int size( node* node)
{
  if (node==NULL)
    return 0;
  else
    return(size(node->left) + 1 + size(node->right));
}

/*-------------------------*/
```

```
int printPostorder( node* node)
{
     if (node == NULL)
       return 0;

     printPostorder(node->left);
     printPostorder(node->right);
     cout<<"  "<< node->data;
}
/*------------------------------*/
int printInorder( node* node)
{
     if (node == NULL)
         return 0;

     printInorder(node->left);

     cout<<"  "<< node->data;

     printInorder(node->right);
}

/*------------------------------*/
int printPreorder( node* node)
{
     if (node == NULL)
         return 0;

     cout<<"  "<< node->data;

     printPreorder(node->left);

     printPreorder(node->right);
}

/*------------------------------*/
int main()
{

  node *root  = newnode(1);
  root->left  = newnode(2);
  root->right = newnode(3);
  root->left->left  = newnode(4);
  root->left->right = newnode(5);
  cout<<"Postorder:\n";
  printPostorder(root);
  cout<<"\n:Preorder:\n";
  printPreorder(root);
  cout<<"\n:Inorder:\n";
  printInorder(root);

  cout<<"\n:Size of the tree is  "<< size(root);

  return 0;
}
```

Insertion of a key:

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.




```
 node* insert( node* node, int key)
{

    if (node == NULL) return newnode(key);

    if (key < node->data)
        node->left  = insert(node->left, key);
    else  if (key > node->data)
        node->right = insert(node->right, key);

    return node;
}

int main()
{
     node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    inorder(root);
    return 0;
}
```

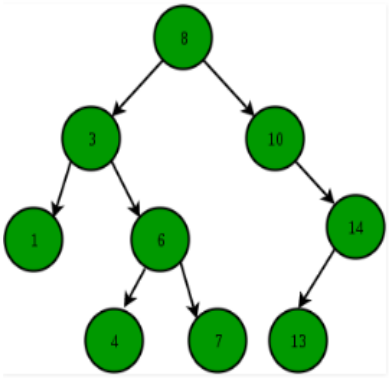
Output:

```
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80
```

Binary Search Tree :

Binary Search Tree(BST), is a node-based binary tree data structure 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 be no duplicate nodes.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Searching a key:

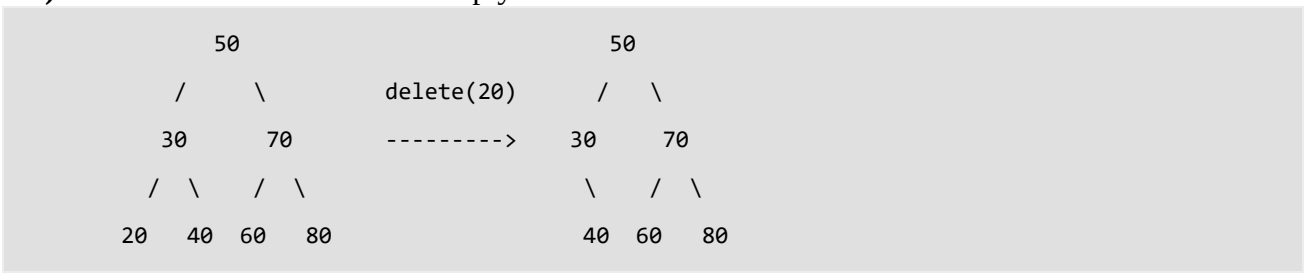
To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

```
int search( node* root, int key)
{

    if (root == NULL )
      return 0;
    else  if( root->data == key)
       return  1;

    else if (root->data < key)
       return search(root->right, key);

    else return search(root->left, key);
}
```

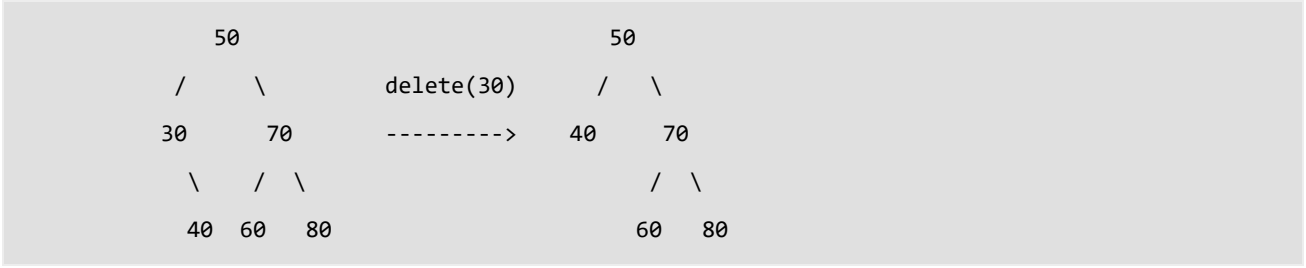
Delete node in tree:

In this post, delete operation is discussed. When we delete a node, there possibilities arise.

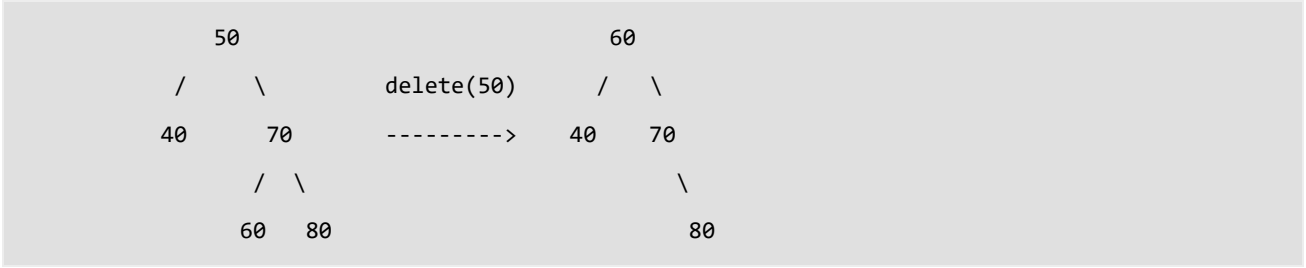
1) Node to be deleted is leaf*:* Simply remove from the tree.



2) Node to be deleted has only one child*:* Copy the child to the node and delete the child



3) Node to be deleted has two children*:* Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

```
 node * minValueNode( node* node)
{
    while (node->left != NULL)
       node = node ->left;

    return node;
}
```

```
/*  this function deletes the key and returns the new root */

 node* deleteNode( node* root, int key)
{

    if (root == NULL) return root;

    if (key < root->data)
       root->left = deleteNode(root->left, key);

    else if (key > root->data)
       root->right = deleteNode(root->right, key);


    else
    {
       // node with only one child or no child

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

       // node with two children: Get(smallest in the right subtree)

       node * temp = minValueNode(root->right);

       root->data = temp->data;

       root->right = deleteNode(root->right, temp->data);
    }
    return root;
}


Int main()

{
    cout<<"\nDelete 20\n";
    root = deleteNode(root, 20);
    cout<<"Inorder traversal of the modified tree \n";
    printInorder(root);

    cout<<"\nDelete 30\n";
    root = deleteNode(root, 30);
    cout<<"Inorder traversal of the modified tree \n";
    printInorder(root);

    cout<<"\nDelete 50\n";
    root = deleteNode(root, 50);
```

```
    cout<<"Inorder traversal of the modified tree \n";
    printInorder(root);

     return 0;
}
```

```
Inorder traversal of the given tree

20 30 40 50 60 70 80

Delete 20

Inorder traversal of the modified tree

30 40 50 60 70 80

Delete 30

Inorder traversal of the modified tree

40 50 60 70 80

Delete 50

Inorder traversal of the modified tree

40 60 70 80
```