DATA STRUCTURE
(CS13217)

Lab Report

Name:	Zain ul abideen
Registration #:	SEU-F16-133
Lab Report #:	09
Dated:	4-06-2018
Submitted To:	Mr. Usman Ahmed

The University of Lahore, Islamabad Campus
Department of Computer Science & Information Technology

Experiment # 1
Implementing the Binary search tree graph

Objective
To understand and implement the binery search.

Software Tool
1.
dev c++

1 Theory

In computer science, binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of container: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations

.

2 Task

2.1 procedure: Task 1

```
#include <iostream>
#include <cstdlib>
using namespace std;

class BinarySearchTree
{
    private:
        struct tree_node
        {
           tree_node* left;
           tree_node* right;
           int data;
        };
        tree_node* root;
    public:
        BinarySearchTree()
        {
           root = NULL;
        }
        bool isEmpty() const { return root==NULL; }
        void print_inorder();
        void inorder(tree_node*);
        void print_preorder();
        void preorder(tree_node*);
        void print_postorder();
        void postorder(tree_node*);
        void insert(int);
        void remove(int);
};

// Smaller elements go left
// larger elements go right
void BinarySearchTree::insert(int d)
{
```

```
    tree_node* t = new tree_node;
    tree_node* parent;
    t->data = d;
    t->left = NULL;
    t->right = NULL;
    parent = NULL;
  // is this a new tree?
  if(isEmpty()) root = t;
  else
  {
    //Note: ALL insertions are as leaf nodes
    tree_node* curr;
    curr = root;
    // Find the Node's parent
    while(curr)
    {
        parent = curr;
        if(t->data > curr->data) curr = curr->right;
        else curr = curr->left;
    }

    if(t->data < parent->data)
       parent->left = t;
    else
       parent->right = t;
  }
}

void BinarySearchTree::remove(int d)
{
    //Locate the element
    bool found = false;
    if(isEmpty())
    {
        cout<<" This Tree is empty! "<<endl;
        return;
    }
    tree_node* curr;
    tree_node* parent;
    curr = root;
```

```
␣␣␣␣while(curr␣!=␣NULL)
␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣if(curr->data␣==␣d)
␣␣␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣found␣=␣true;
␣␣␣␣␣␣␣␣␣␣␣␣␣break;
␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣parent␣=␣curr;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣if(d>curr->data)␣curr␣=␣curr->right;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣else␣curr␣=␣curr->left;
␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣}
␣␣␣␣if(!found)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣cout<<"␣Data␣not␣found!␣"<<endl;
␣␣␣␣␣␣␣␣return;
␣␣␣␣}


␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣3␣cases␣:
␣␣␣␣//␣1.␣We're removing a leaf node
    // 2. We're␣removing␣a␣node␣with␣a␣single␣child
␣␣␣␣//␣3.␣we're removing a node with 2 children

    // Node with single child
    if((curr->left == NULL && curr->right != NULL)|| (curr->left != NULL
&& curr->right == NULL))
    {
       if(curr->left == NULL && curr->right != NULL)
       {
           if(parent->left == curr)
           {
             parent->left = curr->right;
             delete curr;
           }
           else
           {
             parent->right = curr->right;
```

```
        delete curr;
      }
  }
  else // left child present, no right child
  {
     if(parent->left == curr)
      {
        parent->left = curr->left;
        delete curr;
      }
      else
      {
        parent->right = curr->left;
        delete curr;
      }
  }
 return;
}


           if( curr->left == NULL && curr->right == NULL)
{
    if(parent->left == curr) parent->left = NULL;
    else parent->right = NULL;
                          delete curr;
                          return;
}



if (curr->left != NULL && curr->right != NULL)
{
    tree_node* chkr;
    chkr = curr->right;
    if((chkr->left == NULL) && (chkr->right == NULL))
    {
        curr = chkr;
        delete chkr;
        curr->right = NULL;
    }
```

```
        else // right child has children
        {

            if((curr->right)->left != NULL)
            {
                tree_node* lcurr;
                tree_node* lcurrp;
                lcurrp = curr->right;
                lcurr = (curr->right)->left;
                while(lcurr->left != NULL)
                {
                    lcurrp = lcurr;
                    lcurr = lcurr->left;
                }
                                                                curr->data
                delete lcurr;
                lcurrp->left = NULL;
            }
            else
            {
                tree_node* tmp;
                tmp = curr->right;
                curr->data = tmp->data;
                                                curr->right = tmp->righ
                delete tmp;
            }

        }
                return;
    }

}

void BinarySearchTree::print_inorder()
{
  inorder(root);
}

void BinarySearchTree::inorder(tree_node* p)
{
```

```
    if(p != NULL)
    {
        if(p->left) inorder(p->left);
        cout<<" "<<p->data<<" ";
        if(p->right) inorder(p->right);
    }
    else return;
}

void BinarySearchTree::print_preorder()
{
  preorder(root);
}

void BinarySearchTree::preorder(tree_node* p)
{
    if(p != NULL)
    {
        cout<<" "<<p->data<<" ";
        if(p->left) preorder(p->left);
        if(p->right) preorder(p->right);
    }
    else return;
}

void BinarySearchTree::print_postorder()
{
  postorder(root);
}

void BinarySearchTree::postorder(tree_node* p)
{
    if(p != NULL)
    {
        if(p->left) postorder(p->left);
        if(p->right) postorder(p->right);
        cout<<" "<<p->data<<" ";
    }
    else return;
}
```

```cpp
int main()
{
    BinarySearchTree b;
    int ch,tmp,tmp1;
    while(1)
    {
       cout<<endl<<endl;
       cout<<" Binary Search Tree Operations "<<endl;

       cout<<" 1. Insertion/Creation "<<endl;
       cout<<" 2. In-Order Traversal "<<endl;
       cout<<" 3. Pre-Order Traversal "<<endl;
       cout<<" 4. Post-Order Traversal "<<endl;
       cout<<" 5. Removal "<<endl;
       cout<<" 6. Exit "<<endl;
       cout<<" Enter your choice : ";
       cin>>ch;
       switch(ch)
       {
           case 1 : cout<<" Enter Number to be inserted : ";
                    cin>>tmp;
                    b.insert(tmp);
                    break;
           case 2 : cout<<endl;
                    cout<<" In-Order Traversal "<<endl;

                    b.print_inorder();
                    break;
           case 3 : cout<<endl;


                    b.print_preorder();
                    break;
           case 4 : cout<<endl;
                    cout<<" Post-Order Traversal "<<endl;

                    b.print_postorder();
                    break;
           case 5 : cout<<" Enter data to be deleted : ";
```

```
                    cin>>tmp1;
                    b.remove(tmp1);
                    break;
          case 6 :  system("pause");
                    return 0;
                    break;
      }
  }
```

3 Conclusion

In this lab we perform the binary search tree(BST) and how to display code in your screen.