North South University
CSE-225L(Data Structures & Algorithm)
Summer - 2018
Lab-12 (Binary Search Tree)

Header Files

bst.h

```
#ifndef BST_H_INCLUDED
#define BST_H_INCLUDED
#include <iostream>
#include "quetype.h"
using namespace std;
class BinarySearchTree
{
   private:
         struct tree_node
         {
           tree_node* left;
           tree_node* right;
           int data;
         };

tree_node* root;
void fillInOrder(QueType<int>&,tree_node*);
void fillInPreOrder(QueType<int>&,tree_node*);
void fillInPostOrder(QueType<int>&,tree_node*);
void makeEmpty(tree_node*&);
void inorder(tree_node*);
void preorder(tree_node*);
void postorder(tree_node*);
   public:
       BinarySearchTree();
      ~BinarySearchTree();
       bool isEmpty();
       void insert(int);
       void remove(int);
       void print_inorder();
       void print_preorder();
       void print_postorder();
       bool searchItem(int);
       void makeTreeEmpty();
       void getInOrder(QueType<int>&);
       void getPreOrder(QueType<int>&);
       void getPostOrder(QueType<int>&);

};
#endif // BST_H_INCLUDED
```

quetype.h

```
#ifndef QUETYPE_H_INCLUDED
#define QUETYPE_H_INCLUDED
#include <iostream>
using namespace std;

class FullQueue{};
class EmptyQueue{};

template <class DataType>
class QueType
{

      struct NodeType
      {
            DataType info;
            NodeType* next;
      };


public:
      QueType();
      ~QueType();
      void MakeEmpty();
      void Enqueue(DataType);
      DataType Dequeue();
      bool IsEmpty();
      bool IsFull();
private:
      NodeType *front, *rear;
};
#endif // QUETYPE_H_INCLUDED
```

CPP Files

bst.cpp

```
#include"bst.h"

BinarySearchTree::BinarySearchTree()
{
      root = NULL;
}

bool BinarySearchTree::isEmpty()
{
      if(root == NULL)
      {
         return true;
      }

      return false;
}

void BinarySearchTree::insert(int d)
{
 tree_node* t = new tree_node;
 t->data = d;
 t->left = NULL;
 t->right = NULL;

 tree_node* parent;
 parent = NULL;

 // is this a new tree?
 if(isEmpty())
 {
      root = t;
 }
```

```
else // inserting into a non-empty tree
 {
   //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;
     }
   }// while ends here

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

 } // insert function ends here

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;
    }
 }// while ends here

 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

  // 1. 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;
 }

//We're looking at a leaf node
if( curr->left == NULL && curr->right == NULL)
{
  if(parent->left == curr) parent->left = NULL;
  else parent->right = NULL;
  delete curr;
  return;
}

//2. Node with 2 children
// replace node with smallest value in right subtree
 if (curr->left != NULL && curr->right != NULL)
 {
   tree_node* chkr;
   chkr = curr->right;

  if((chkr->left == NULL) && (chkr->right == NULL))
  {
    curr->data = chkr->data;
    delete chkr;
    curr->right = NULL;
  }
 else // right child has children
 {
   //if the node's right child has a left child, move all the way down
  // left to locate smallest element
  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 = lcurr->data;
    delete lcurr;
    lcurrp->left = NULL;
```

```
  }

  else
  {
     tree_node* tmp;
     tmp = curr->right;
     curr->data = tmp->data;
     curr->right = tmp->right;
     delete tmp;
  }
 }
 return;
}
} // remove function ends here



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()
{
     // complete this function to print tree items in pre order
}

void BinarySearchTree::preorder(tree_node* p)
{
     // complete this function to print tree items in pre order

}
```

```
void BinarySearchTree::print_postorder()
{
     // complete this function to print tree items in post order

}

void BinarySearchTree::postorder(tree_node* p)
{
     // complete this function to print tree items in post order

}

bool BinarySearchTree::searchItem(int x)
{

    if(root==NULL)
    {
        return false;
    }
    else
    {

        tree_node* temp;
        temp = root;

        bool found = false;

        while((!found)&&(temp!=NULL))
        {
            if(x<temp->data)
            {
                temp = temp->left;
            }
            else if(x>temp->data)
            {
                temp = temp->right;
            }
            else
            {
                found = true;
            }

        }

       temp = NULL;
       return found;
    }
}
```

```
BinarySearchTree::~BinarySearchTree()
{
    makeEmpty(root);
}

void BinarySearchTree::makeEmpty(tree_node*& p)
{
    if(p!=NULL)
    {
        if(p->left) makeEmpty(p->left);
        if(p->right) makeEmpty(p->right);
        delete p;
        p = NULL;
    }
}

void BinarySearchTree::makeTreeEmpty()
{
    makeEmpty(root);
}

void BinarySearchTree::getInOrder(QueType<int>& q)
{
    if(!q.IsEmpty())
           q.MakeEmpty();
    fillInOrder(q,root);
}




void BinarySearchTree::fillInOrder(QueType<int>& q,tree_node* p)
{

  if(p!= NULL)
  {
    if(p->left) fillInOrder(q,p->left);
    q.Enqueue(p->data);
    if(p->right) fillInOrder(q,p->right);
  }

 else
    return;

}
```

```
void BinarySearchTree::getPreOrder(QueType<int>& q)
{
     if(!q.IsEmpty()) q.MakeEmpty();
     fillInPreOrder(q,root);

}

void BinarySearchTree::fillInPreOrder(QueType<int>& q,tree_node* p)
{
  if(p!= NULL)
  {
    q.Enqueue(p->data);
    if(p->left) fillInPreOrder(q,p->left);
    if(p->right) fillInPreOrder(q,p->right);
  }

 else
    return;

}

void BinarySearchTree::getPostOrder(QueType<int>& q)
{

    if(!q.IsEmpty()) q.MakeEmpty();
    fillInPostOrder(q,root);

}




void BinarySearchTree::fillInPostOrder(QueType<int>& q,tree_node* p)
{
  if(p!= NULL)
  {
    if(p->left) fillInPostOrder(q,p->left);
    if(p->right) fillInPostOrder(q,p->right);
    q.Enqueue(p->data);
  }

 else
    return;

}
```

quetype.cpp

```
#include "quetype.h"

template <class DataType>
QueType<DataType>::QueType()
{
      front = NULL;
      rear = NULL;
}


template <class DataType>
bool QueType<DataType>::IsEmpty()
{
            return (front == NULL);
}


template<class DataType>
bool QueType<DataType>::IsFull()
{
      NodeType* location;
      try
      {
            location = new NodeType;
            delete location;
            return false;
      }
      catch(bad_alloc& exception)
      {
            return true;
      }
}

template <class DataType>
void QueType<DataType>::Enqueue(DataType newItem)
{
      if (IsFull())
            throw FullQueue();
      else
      {
            NodeType* newNode;
            newNode = new NodeType;
            newNode->info = newItem;
            newNode->next = NULL;
```

```
            if (rear == NULL)
                  front = newNode;
            else
                  rear->next = newNode;

            rear = newNode;
      }
}
```

```
template <class DataType>
DataType QueType<DataType>::Dequeue()
{
            DataType item;

            if (IsEmpty())
                  throw EmptyQueue();
            else
            {
                  NodeType* tempPtr;
                  tempPtr = front;

                  item = front->info;

                  front = front->next;

                  if (front == NULL)
                        rear = NULL;
                  delete tempPtr;

                  return item;
            }
}

template <class DataType>
void QueType<DataType>::MakeEmpty()
{
      NodeType* tempPtr;

      while (front != NULL)
      {
```

```
            tempPtr = front;
            front = front->next;
            delete tempPtr;
      }
      rear = NULL;
}

template <class DataType>
QueType<DataType>::~QueType()
{
            MakeEmpty();
}


template class QueType<int>;
```