Farhan Zaman 11/25/18
CS21200 10:00AM

HW 5 Report

This homework was designed to get us familiar with trees, and the many different representations of it, its functionality, why it's used as a data structure. We were introduced to many new terminology, and then implementation techniques with arrays or nodes, we learned how to make template functions, more than 1 class template, and even more practice with algorithms and recursion.

# For Question 1, here is my header file:

```
#ifndef __ZAMAN_FARHAN_HW5_Q1_H__
#define __ZAMAN_FARHAN_HW5_Q1_H__
#include <iostream>
#include <cassert>
#include <iomanip>
#include <cstdlib>
#include <stack>
using namespace std;

template <class Item>
class btNode
{
	private:
	    Item data_field;
	    btNode* left_field;
	    btNode* right_field;
	    btNode* parent_field;
	public:
	     btNode(const Item & init_data=Item(), btNode* init_left=NULL, btNode*
init_right=NULL, btNode* init_parent=NULL)
	     {
	            data_field=init_data;
	            left_field=init_left;
	            right_field=init_right;
	      parent_field=init_parent;
	     }
	     btNode(const btNode& source)
	     {
	            data_field=source.data_field;
	            left_field=source.left_field;
	            right_field=source.right_field;
	      parent_field=source.parent_field;
	     }
```

```
            Item& data(){return data_field;}
            btNode*& left(){return left_field;}
            btNode*& right(){return right_field;}
            btNode*& parent(){return parent_field;}
            void set_data(const Item& new_data){data_field=new_data;}
            void set_left(btNode* new_left){left_field=new_left;}
            void set_right(btNode* new_right){right_field=new_right;}
            void set_parent(btNode* new_parent){parent_field=new_parent;}
            const Item& data() const{return data_field;}
            const btNode* left() const {return left_field;}
            const btNode* right() const {return right_field;}
            const btNode*& parent() const {return parent_field;}
            bool is_leaf() const
            {
               return (left_field==NULL) && (right_field==NULL);
            }
};

template <class Item>
void print(const btNode<Item>* node_ptr, size_t depth);

template <class Item>
btNode<Item>* tree_copy(const btNode<Item>* root_ptr);

template <class Item>
size_t depth(const btNode<Item>* node_ptr);

template <class Item>
size_t numOfNodes(const btNode<Item>* node_ptr);

template <class Process, class BTNode>
void inorder(Process f, BTNode* node_ptr);

template <class Process, class BTNode>
void postorder(Process f, BTNode* node_ptr);

template <class Process, class BTNode>
void preorder(Process f, BTNode* node_ptr);


#endif
```

# For Question 1, here is my cpp file:

```
#ifndef __ZAMAN_FARHAN_HW5_Q1_CPP__
#define __ZAMAN_FARHAN_HW5_Q1_CPP__
#include "Zaman_Farhan_HW5_Q1.h"
```

```
template <class Item>
size_t numOfNodes(const btNode<Item>* node_ptr)
{
    if(node_ptr==NULL)
    {
        return 0;
    }
    else
    {
       return 1 + numOfNodes(node_ptr->left()) + numOfNodes(node_ptr->right());
    }
}

template <class Item>
size_t depth(const btNode<Item>* node_ptr)
{
    if(node_ptr==NULL)
    {
        return 0;
    }
    else
    {
       int leftDepth=1;
       int rigthDepth=1;
       leftDepth+=depth(node_ptr->left);
       rigthDepth+=depth(node_ptr->right);

       if(leftDepth>rigthDepth)
       {
        return leftDepth;
       }
       else
       {
        return rigthDepth;
       }

    }
}

template <class Item>
void print(const btNode<Item>* node_ptr, size_t depth)
{
    cout<<setw(4*depth)<<"";
    if(node_ptr==NULL)
    {
        cout<<"[Empty]"<<endl;
    }
    else if(node_ptr->is_leaf())
```

```
    {
        cout<<node_ptr->data();
        cout<<"[Leaf]"<<endl;
    }
    else
    {
        cout<<node_ptr->data()<<endl;
        print(node_ptr->left(),depth+1);
        print(node_ptr->right(),depth+1);
    }
}

template <class Item>
btNode<Item>* tree_copy(const btNode<Item>* root_ptr)
{
    btNode<Item>* l_ptr;
    btNode<Item>* r_ptr;
    if (root_ptr == NULL)
    {
        return NULL;
    }
    else
    {
        l_ptr = tree_copy(root_ptr->left());
        r_ptr = tree_copy(root_ptr->right());
        return new btNode<Item>(root_ptr->data(),l_ptr,r_ptr);
    }
}


template <class Process, class BTNode>
void inorder(Process f, BTNode* node_ptr)
{
    if (node_ptr != NULL)
    {
        inorder(f,node_ptr->left());
        f(node_ptr->data());
        inorder(f,node_ptr->right());
    }
}

template <class Process, class BTNode>
void postorder(Process f, BTNode* node_ptr)
{
    if (node_ptr != NULL)
    {
        postorder(f,node_ptr->left());
        postorder(f,node_ptr->right());
```

```
        f(node_ptr->data());
    }
}

template <class Process, class BTNode>
void preorder(Process f, BTNode* node_ptr)
{
    if (node_ptr != NULL)
    {
        f(node_ptr->data());
        preorder(f,node_ptr->left());
        preorder(f,node_ptr->right());
    }
}

#endif
```

# For Question 2, here is my header file:

```
// FILE: btClass.h
// TEMPLATE CLASS PROVIDED: binaryTree<Item> (a binary tree where each node has
//   an item) The template parameter, Item, is the data type of the items in the
//   tree's nodes. It may be any of the C++ built-in types (int, char, etc.),
//   or a class with a default constructor, a copy constructor and an assignment
//   operator.
//
// NOTE: Each non-empty tree always has a "current node."  The location of
// the current node is controlled by three member functions: shiftUp,
// shiftToRoot, shiftLeft, and shiftRight.
//
// CONSTRUCTOR for the binaryTree<Item> template class:
//   binaryTree( )
//     Postcondition: The binary tree has been initialized as an empty tree
//     (with no nodes).
//
// MODIFICATION MEMBER FUNCTIONS for the binaryTree<Item> template class:
//   void createFirstNode(const Item& entry)
//     Precondition: size( ) is zero.
//     Postcondition: The tree now has one node (a root node), containing the
//     specified entry. This new root node is the "current node."
//
//   void shiftToRoot( )
//     Precondition: size( ) > 0.
//     Postcondition: The "current node" is now the root of the tree.
//
//   void shiftUp( )
//     Precondition: hasParent( ) returns true.
//     Postcondition: The "current node" has been shifted up to the parent of
```

```
//     the old current node.
//
//   void shiftLeft( )
//     Precondition: hasLeft( ) returns true.
//     Postcondition: The "current node" been shifted down to the left child
//     of the original current node.
//
//   void shiftRight( )
//     Precondition: hasRight( ) returns true.
//     Postcondition: The "current node" been shifted down to the right child
//     of the original current node.
//
//   void change(const Item& new_entry)
//     Precondition: size( ) > 0.
//     Postcondition: The data at the "current node" has been changed to the
//     new entry.
//
//   void addLeft(const Item& entry)
//     Precondition: size( ) > 0, and hasLeft( ) returns false.
//     Postcondition: A left child has been added to the "current node,"
//     with the given entry.
//
//   void addRight(const Item& entry)
//     Precondition: size( ) > 0, and hasRight( ) returns false.
//     Postcondition: A right child has been added to the "current node,"
//     with the given entry.
//
// CONSTANT MEMBER FUNCTIONS for the binaryTree<Item> template class:
//   size_t size( ) const
//     Postcondition: The return value is the number of nodes in the tree.
//
//   Item retrieve( ) const
//     Precondition: size( ) > 0.
//     Postcondition: The return value is the data from the "current node."
//
//   bool hasParent( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a parent.
//
//   bool hasLeft( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a left child.
//
//   bool hasRight( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a right child.
//
// VALUE SEMANTICS for the binaryTree<Item> template class:
```

```
//   Assignments and the copy constructor may be used with binaryTree objects.
//
// DYNAMIC MEMORY USAGE by the binaryTree<Item> template class:
//   If there is insufficient dynamic memory, then the following functions
//   throw bad_alloc:
//   createFirstNode, addLeft, addRight, the copy constructor, and the
//   assignment operator.

#ifndef __ZAMAN_FARHAN_HW5_Q2_H__
#define __ZAMAN_FARHAN_HW5_Q2_H__
#include "Zaman_Farhan_HW5_Q1.h"

template <class Item>
class binaryTree
{
      private:
          btNode<Item>* current_ptr;
          btNode<Item>* root_ptr;
          btNode<Item>* parent_ptr;
          size_t count;
      public:
          binaryTree( );
          binaryTree(const binaryTree& source);
          ~binaryTree( );
          void createFirstNode(const Item& entry);
          void shiftToRoot( );
          void shiftUp( );
          void shiftLeft( );
          void shiftRight( );
          void change(const Item& new_entry);
          void addLeft(const Item& entry);
          void addRight(const Item& entry);
          void tree_clear(btNode<Item>*& root_ptr);
          size_t size( ) const;
          Item retrieve( ) const;
          bool hasParent( ) const;
          bool hasLeft( ) const;
          bool hasRight( ) const;
          btNode<Item>* getRoot();
};

#include "Zaman_Farhan_HW5_Q2.cpp"
#endif
```

# For Question 2, here is my cpp file:

```
#ifndef __ZAMAN_FARHAN_HW5_Q2_CPP__
#define __ZAMAN_FARHAN_HW5_Q2_CPP__
```

```
#include "Zaman_Farhan_HW5_Q2.h"

template <class Item>
binaryTree<Item>::binaryTree()
{
       current_ptr=new btNode<Item>;
    root_ptr=new btNode<Item>;
    count=0;
}

template <class Item>
binaryTree<Item>::binaryTree(const binaryTree& source)
{
    root_ptr=tree_copy(source.root_ptr);
    current_ptr=root_ptr;
    current_ptr->set_parent(NULL);
    count=source.count;
}

template <class Item>
binaryTree<Item>::~binaryTree()
{
       tree_clear(root_ptr);
}

template <class Item>
void binaryTree<Item>::createFirstNode(const Item& entry)
{
       assert(size()==0);
       root_ptr= new btNode<Item>(entry);
       current_ptr=root_ptr;
       current_ptr->set_parent(NULL);
       count=1;
}

template <class Item>
void binaryTree<Item>::shiftToRoot()
{
       assert(size()>0);
       current_ptr=root_ptr;
}

template <class Item>
void binaryTree<Item>::shiftUp()
{
       assert(hasParent());
       current_ptr=current_ptr->parent();
}
```

```
template <class Item>
void binaryTree<Item>::shiftLeft()
{
	assert(hasLeft());
	current_ptr=current_ptr->left();
}

template <class Item>
void binaryTree<Item>::shiftRight()
{
	assert(hasRight());
	current_ptr=current_ptr->right();
}

template <class Item>
void binaryTree<Item>::change(const Item& new_entry)
{
	assert(size()>0);
	current_ptr->set_data(new_entry);
}

template <class Item>
void binaryTree<Item>::addLeft(const Item& entry)
{
	assert(size()>0);
	assert(!hasLeft());
	btNode<Item>* new_left_ptr;
	new_left_ptr=new btNode<Item>(entry,NULL, NULL, current_ptr);
	current_ptr->set_left(new_left_ptr);
	count++;
}


template <class Item>
void binaryTree<Item>::addRight(const Item& entry)
{
	assert(size()>0);
	assert(!hasRight());
	btNode<Item>* new_right_ptr;
	new_right_ptr=new btNode<Item>(entry,NULL, NULL, current_ptr);
	current_ptr->set_right(new_right_ptr);
	count++;
}


template <class Item>
void binaryTree<Item>::tree_clear(btNode<Item>*& root_ptr)
```

```
{
	btNode<Item>* child;
	if (root_ptr!= NULL)
	{
		child = root_ptr->left( );
		tree_clear(child);
		child = root_ptr->right( );
		tree_clear(child);
		delete root_ptr;
		root_ptr = NULL;
	}
}


template <class Item>
size_t binaryTree<Item>::size() const
{
	return count;
}

template <class Item>
Item binaryTree<Item>::retrieve() const
{
	return current_ptr->data();
}

template <class Item>
bool binaryTree<Item>::hasParent() const
{
	if(current_ptr!=root_ptr && size()>0)
	{
		return true;
	}
	else
	{
		return false;
	}
}

template <class Item>
bool binaryTree<Item>::hasLeft() const
{
	assert(size()>0);
	if(current_ptr->left()!=NULL)
	{
		return true;
	}
	else
```

```
	{
		return false;
	}
}

template <class Item>
bool binaryTree<Item>::hasRight() const
{
	assert(size()>0);
	if(current_ptr->right()!=NULL)
	{
		return true;
	}
	else
	{
		return false;
	}
}


template <class Item>
btNode<Item>* binaryTree<Item>::getRoot()
{
	assert(size()>0);
	return root_ptr;
}

#endif
```

# For Question 3, here is my header file:

```
// FILE: btClass.h
// TEMPLATE CLASS PROVIDED: binaryTree<Item> (a binary tree where each node has
//   an item) The template parameter, Item, is the data type of the items in the
//   tree's nodes. It may be any of the C++ built-in types (int, char, etc.),
//   or a class with a default constructor, a copy constructor and an assignment
//   operator.
//
// NOTE: Each non-empty tree always has a "current node."  The location of
// the current node is controlled by three member functions: shiftUp,
// shiftToRoot, shiftLeft, and shiftRight.
//
// CONSTRUCTOR for the binaryTree<Item> template class:
//   binaryTree( )
//     Postcondition: The binary tree has been initialized as an empty tree
//     (with no nodes).
//
// MODIFICATION MEMBER FUNCTIONS for the binaryTree<Item> template class:
```

```
//   void createFirstNode(const Item& entry)
//     Precondition: size( ) is zero.
//     Postcondition: The tree now has one node (a root node), containing the
//     specified entry. This new root node is the "current node."
//
//   void shiftToRoot( )
//     Precondition: size( ) > 0.
//     Postcondition: The "current node" is now the root of the tree.
//
//   void shiftUp( )
//     Precondition: hasParent( ) returns true.
//     Postcondition: The "current node" has been shifted up to the parent of
//     the old current node.
//
//   void shiftLeft( )
//     Precondition: hasLeft( ) returns true.
//     Postcondition: The "current node" been shifted down to the left child
//     of the original current node.
//
//   void shiftRight( )
//     Precondition: hasRight( ) returns true.
//     Postcondition: The "current node" been shifted down to the right child
//     of the original current node.
//
//   void change(const Item& new_entry)
//     Precondition: size( ) > 0.
//     Postcondition: The data at the "current node" has been changed to the
//     new entry.
//
//   void addLeft(const Item& entry)
//     Precondition: size( ) > 0, and hasLeft( ) returns false.
//     Postcondition: A left child has been added to the "current node,"
//     with the given entry.
//
//   void addRight(const Item& entry)
//     Precondition: size( ) > 0, and hasRight( ) returns false.
//     Postcondition: A right child has been added to the "current node,"
//     with the given entry.
//
// CONSTANT MEMBER FUNCTIONS for the binaryTree<Item> template class:
//   size_t size( ) const
//     Postcondition: The return value is the number of nodes in the tree.
//
//   Item retrieve( ) const
//     Precondition: size( ) > 0.
//     Postcondition: The return value is the data from the "current node."
//
//   bool hasParent( ) const
```

```
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a parent.
//
//   bool hasLeft( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a left child.
//
//   bool hasRight( ) const
//     Postcondition: Returns true if size( ) > 0, and the "current node"
//     has a right child.
//
// VALUE SEMANTICS for the binaryTree<Item> template class:
//   Assignments and the copy constructor may be used with binaryTree objects.
//
// DYNAMIC MEMORY USAGE by the binaryTree<Item> template class:
//   If there is insufficient dynamic memory, then the following functions
//   throw bad_alloc:
//   createFirstNode, addLeft, addRight, the copy constructor, and the
//   assignment operator.

#ifndef __ZAMAN_FARHAN_HW5_Q3_H__
#define __ZAMAN_FARHAN_HW5_Q3_H__
#include <iostream>
#include <cstdlib>
#include <cassert>
using namespace std;

template <class Item>
class binaryTree
{
       private:
           size_t current;
           size_t root;
           size_t count;
           size_t capacity;
              Item* tree;
       public:
              static const size_t DEFAULTCAPACITY=30;
           binaryTree(size_t initCap=DEFAULTCAPACITY);
           binaryTree(const binaryTree& source);
           ~binaryTree();
           void createFirstNode(const Item& entry);
           void shiftToRoot();
           void shiftUp();
           void shiftLeft();
           void shiftRight();
           void change(const Item& new_entry);
           void addLeft(const Item& entry);
```

```
            void addRight(const Item& entry);
            void resize(size_t cap);
            size_t size() const;
            Item retrieve() const;
            bool hasParent() const;
            bool hasLeft() const;
            bool hasRight() const;
};
#include "Zaman_Farhan_HW5_Q3.cpp"
#endif
```

# For Question 3, here is my cpp file:

```
#ifndef __ZAMAN_FARHAN_HW5_Q3_CPP__
#define __ZAMAN_FARHAN_HW5_Q3_CPP__
#include "Zaman_Farhan_HW5_Q3.h"


template <class Item>
binaryTree<Item>::binaryTree(size_t initCap)
{
    capacity=initCap;
    tree=new Item[initCap];
}

template <class Item>
binaryTree<Item>::binaryTree(const binaryTree& source)
{
      current=source.current;
    root=source.root;
    count=source.count;
    capacity=source.capacity;
    tree=new Item[capacity];
    for(int i=0; i<capacity; i++)
    {
      tree[i]=source.tree[i];
    }
}

template <class Item>
binaryTree<Item>::~binaryTree()
{

}

template <class Item>
void binaryTree<Item>::createFirstNode(const Item& entry)
{
```

```
        assert(size()==0);
        current=0;
    root=0;
    count=1;
    tree[current]=entry;
}

template <class Item>
void binaryTree<Item>::shiftToRoot()
{
        assert(size()>0);
        current=0;
}

template <class Item>
void binaryTree<Item>::shiftUp()
{
        assert(hasParent());
        current=(current-1)/2;
}

template <class Item>
void binaryTree<Item>::shiftLeft()
{
        assert(hasLeft());
        current=2*current+1;
}

template <class Item>
void binaryTree<Item>::shiftRight()
{
        assert(hasLeft());
        current=2*current+2;
}

template <class Item>
void binaryTree<Item>::change(const Item& new_entry)
{
        assert(size()>0);
        tree[current]=new_entry;
}

template <class Item>
void binaryTree<Item>::addLeft(const Item& entry)
{
        assert(size()>0);
        assert(!hasLeft());
        assert(2*current<count);
```

```
	if(capacity<count+1)
	{
		capacity++;
		resize(capacity);
	}

	tree[2*current+1]=entry;
	count++;
}

template <class Item>
void binaryTree<Item>::addRight(const Item& entry)
{
	assert(size()>0);
	assert(!hasRight());
	assert(hasLeft());
	if(capacity<count+1)
	{
		capacity++;
		resize(capacity);
	}

	tree[2*current+2]=entry;
	count++;
}

template <class Item>
size_t binaryTree<Item>::size() const
{
	return count;
}

template <class Item>
Item binaryTree<Item>::retrieve() const
{
	return tree[current];
}

template <class Item>
bool binaryTree<Item>::hasParent() const
{
	if(current!=root && size()>0)
	{
		return true;
	}
	else
	{
		return false;
```

```
	}
}

template <class Item>
bool binaryTree<Item>::hasLeft() const
{
	assert(size()>0);
	if(2*current+1<count)
	{
		return true;
	}
	else
	{
		return false;
	}
}

template <class Item>
bool binaryTree<Item>::hasRight() const
{
	assert(size()>0);
	if(2*current+2<count)
	{
		return true;
	}
	else
	{
		return false;
	}
}

template <class Item>
void binaryTree<Item>::resize(size_t cap)
{
	Item* largerArr=new Item[cap];
	for (int i = 0; i < count; i++)
	{
		largerArr[i]=tree[i];
	}
	delete []tree;
	tree=largerArr;
	capacity=cap;
}

#endif
```

# For Question 4, here is my header file:

```
#ifndef __ZAMAN_FARHAN_HW5_Q4_H__
#define __ZAMAN_FARHAN_HW5_Q4_H__
#include "Zaman_Farhan_HW5_Q2.h"

template <class Item>
class binarySearchTree
{
	private:
		btNode<Item>* bstRoot;
	public:
	    binarySearchTree( );
	    ~binarySearchTree( );
	    btNode<Item>* minimum(btNode<Item>* x);
	    btNode<Item>* maximum(btNode<Item>* x);
	    btNode<Item>* search(btNode<Item>* x, Item& k);
	    btNode<Item>* get_root();
	    void set_root(btNode<Item>* x);

};

template <class Item>
void add(binarySearchTree<Item>& T, btNode<Item>* z);

template <class Item>
void remove(binarySearchTree<Item>& T, btNode<Item>* z);

template <class Item>
void transplant(binarySearchTree<Item>& T, btNode<Item>* u, btNode<Item>* v);

#include "Zaman_Farhan_HW5_Q4.cpp"
#endif
```

# For Question 4, here is my cpp file:

```
#ifndef __ZAMAN_FARHAN_HW5_Q4_CPP__
#define __ZAMAN_FARHAN_HW5_Q4_CPP__
#include "Zaman_Farhan_HW5_Q4.h"

template <class Item>
binarySearchTree<Item>::binarySearchTree( )
{
	bstRoot=NULL;
}

template <class Item>
binarySearchTree<Item>::~binarySearchTree()
{
	delete bstRoot;
```

```
}

template <class Item>
btNode<Item>* binarySearchTree<Item>::minimum(btNode<Item>* x)
{
	while(x->left()!=NULL)
	{
		x=x->left();
	}
	return x;
}


template <class Item>
btNode<Item>* binarySearchTree<Item>::get_root()
{
	return bstRoot;
}

template <class Item>
void binarySearchTree<Item>::set_root(btNode<Item>* x)
{
	bstRoot=x;
}


template <class Item>
btNode<Item>* binarySearchTree<Item>::maximum(btNode<Item>* x)
{
	while(x->right()!=NULL){
		x=x->right();
	}
	return x;
}

template <class Item>
btNode<Item>* binarySearchTree<Item>::search(btNode<Item>* x, Item& k)
{
	if(x==NULL || k==x->data())
	{
		return x;
	}
	if (k<x->data())
	{
		return search(x->left(),k);
	}
	else
	{
```

```
		return search(x->right(),k);
	}

}

template <class Item>
void add(binarySearchTree<Item>& T, btNode<Item>* z)
{
	btNode<Item>* y=NULL;
	btNode<Item>* x=T.get_root();
	while(x!=NULL)
	{
		y=x;
		if ((z->data()) < (x->data()))
		{
			x=x->left();
		}
		else
		{
			x=x->right();
		}
	}
	z->set_parent(y);
	if(y==NULL)
	{
		T.set_root(z);
	}
	else if (z->data() < y->data())
	{
		y->set_left(z);
	}
	else
	{
		y->set_right(z);
	}

}


template <class Item>
void transplant(binarySearchTree<Item>& T, btNode<Item>* u, btNode<Item>* v)
{
	if(u->parent()==NULL)
	{
		T.set_root(v);
	}
	else if (u==u->parent()->left())
	{
```

```
		u->parent()->set_left(v);
	}
	else
	{
		u->parent()->set_right(v);
	}
	if(v!=NULL)
	{
		v->set_parent(u->parent());
	}
}

template <class Item>
void remove(binarySearchTree<Item>& T, btNode<Item>* z)
{
	if(z->left()==NULL)
	{
		transplant(T,z,z->right());
	}
	else if (z->right()==NULL)
	{
		transplant(T,z,z->left());
	}
	else
	{
		btNode<Item>* y=T.minimum(z->right());
		if (y->parent()!=z)
		{
			transplant(T,y,y->right());
			y->set_right(z->right());
			y->right()->set_parent(y);
		}

		transplant(T,z,y);
		y->set_left(z->left());
		y->left()->set_parent(y);
	}
}

#endif
```

# For Question 5 .here is my header file:

```
#ifndef __ZAMAN_FARHAN_HW5_Q5_H__
#define __ZAMAN_FARHAN_HW5_Q5_H__
#include "Zaman_Farhan_HW5_Q1.h"
#include "Zaman_Farhan_HW5_Q3.h"
```

```
template <class Item>
class Heap
{
	private:
		Item* h;
		size_t capacity;
		size_t count;
	public:
	    static const size_t DEFAULTCAPACITY=30;
	    Heap(size_t initCap=DEFAULTCAPACITY);
	    ~Heap( );
	    Item minimum();
	    Item maximum();
	    void add(Item& entry);
	    Item remove();
	    void resize(size_t cap);
	    void print();

};

template <class Item>
void maxHeapify(Item arr[], size_t arrSize, size_t i);

template <class Item>
void buildMaxHeap(Item arr[], size_t arrSize);

template <class Item>
void heapsort(Item arr[], size_t arrSize);

template <class Item>
void printArray(Item arr[], size_t arrSize);

#endif
```

# For Question 5, here is my cpp file:

```
#ifndef __ZAMAN_FARHAN_HW5_Q5_CPP__
#define __ZAMAN_FARHAN_HW5_Q5_CPP__
#include "Zaman_Farhan_HW5_Q5.h"

template <class Item>
Heap<Item>::Heap(size_t initCap)
{
	capacity=initCap;
	count=0;
    h=new Item[initCap];
}
```

```
template <class Item>
Heap<Item>::~Heap()
{

}

template <class Item>
Item Heap<Item>::minimum()
{
	Item x;
	for(int i=0; i<count; i++)
	{
		if(x>h[i])
		{
			x=h[i];
		}
	}
	return x;
}

template <class Item>
Item Heap<Item>::maximum()
{
	return h[0];
}

template <class Item>
void Heap<Item>::add(Item& entry)
{
	int i;
	if(capacity<count+1)
	{
		capacity++;
		resize(capacity);
	}
	h[count]=entry;
	i=count;
	count++;
	while (h[(i-1)/2]<h[i])
	{
		Item temp=h[i];
		h[i]=h[(i-1)/2];
		h[(i-1)/2]=temp;
		i=(i-1)/2;
	}
}
```

```
template <class Item>
Item Heap<Item>::remove()
{
	int i=0;
	Item copyRoot=h[0];
	h[0]=h[count-1];
	count--;
	while(h[i]<h[2*i+1]||h[i]<h[2*i+2])
	{
		Item temp=h[i];
		if(h[2*i+1]>h[2*i+2])
		{
			h[i]=h[2*i+1];
			h[2*i+1]=temp;
			i=2*i+1;
		}

		else
		{
			h[i]=h[2*i+2];
			h[2*i+2]=temp;
			i=2*i+2;
		}
	}
	return copyRoot;
}


template <class Item>
void Heap<Item>::resize(size_t cap)
{
	Item* largerArr=new Item[cap];
	for (int i = 0; i < count; i++)
	{
		largerArr[i]=h[i];
	}
	delete []h;
	h=largerArr;
	capacity=cap;
}

template <class Item>
void Heap<Item>::print()
{
	for (int i = 0; i < count; i++)
	{
		cout<<h[i]<<",";
	}
```

```
}

template <class Item>
void maxHeapify(Item arr[], size_t arrSize,size_t i){
	int l=2*i+1;
	int r=2*i+2;
	int largest;
	if(l<arrSize && arr[l]>arr[i])
	{
		largest=l;
	}
	else
	{
		largest=i;
	}
	if(r<arrSize && arr[r]>arr[largest])
	{
		largest=r;
	}
	if(largest!=i)
	{
		swap(arr[i],arr[largest]);
		maxHeapify(arr, arrSize, largest);
	}

}

template <class Item>
void buildMaxHeap(Item arr[], size_t arrSize){
	for(int i=(arrSize/2)-1; i>=0; i--)
	{
		maxHeapify(arr, arrSize, i);
	}
}

template <class Item>
void heapsort(Item arr[], size_t arrSize)
{
	buildMaxHeap(arr, arrSize);
	for(int i=arrSize-1; i>=0; i--)
	{
		swap(arr[0],arr[i]);
		maxHeapify(arr, i, 0);
	}
}
```

```
template <class Item>
void printArray(Item arr[], size_t arrSize)
{
	for (int i = 0; i < arrSize; i++)
	{
		cout<<arr[i]<<",";
	}
}

#endif
```

Some problems of this class is that we have too many different variations, so is someone showed me an array of a tree I need to be well-versed in all traversals. There is also a need to keep track of your parent , which wasn't a first thought and took a while to figure out that that is the easier way. I don't see improvements for classes and functions as there are a myriad of ways of implement them, but I only chose this way.