**EE2410 Data Structure Coding HW #4 -- Trees (Chapter 5)**

**due date 5/26/2024, 23:59**

You should submit:

(a) All your source codes (C++ file).

(b) Show the execution trace of your program, i.e., write a client main() to demonstrate all functions you designed using example data.

Submit your homework before the deadline (midnight of 5/26). Fail to comply (**late** homework) will have ZERO score. **Copy** homework will have ZERO score on both parties and SERIOUS consequences.

1. (40%) Binary tree

Develop a complete C++ template class for binary trees shown below.

**template** <**class** T> **class** Tree;

**template** <**class** T>

**class** TreeNode {

**friend** **class** Tree <T>;

**friend class** InorderIterator<T>;//inorder iterator

**private**:

T data;

TreeNode <T> \*leftChild;

TreeNode <T> \*rightChild;

};

**template**<**class** T>

**class** Tree

{

**friend class** InorderIterator<T>;//inorder iterator

**public**:

Tree(); // constructor for an empty binary tree

Tree(Tree<T>& bt1, T& item, Tree<T>& bt2);

Tree(const Tree<T>&); //copy constructor

// constructor given the root item and left subtrees bt1 and right subtree bt2

~Tree();

**bool** IsEmpty(); // return true iff the binary tree is empty

Tree<T> LeftSubtree(); // return the left subtree

Tree<T> RightSubtree();// return the right subtree

T RootData(); // return the data in the root node of \***this**

// more operations

**private**:

TreeNode <T> \*root;

**void** Visit(TreeNode<T> \*p){cout << p->data << " ";}

};

**template** <**class** T>

**class** InorderIterator{

**public**:

InorderIterator(){ currentNode = root;} // Constructor

InorderIterator(Tree<T> tree):t(tree){ currentNode = t.root; }

T\* Next();

T& **operator** \*();

**bool** **operator**!=(const InorderIterator r)

**private**:

Tree<T> t;

Stack<TreeNode<T>\*> s;

TreeNode<T> \* currentNode;

};

You must include a **constructor**, **copy constructor**, **destructor**, the traversal methods and operator overloads and the iterator class as shown below, and functions in **ADT 5.1**.

void Inorder()

void Preorder()

void Postorder()

void LevelOrder()

**void** NonrecInorder()

**void** NoStackInorder()

**bool** **operator ==** (**const** Tree& t);

TreeNode<T> \* Copy(TreeNode<T> \* p); // Workhorse

**bool** Equal(**const** Tree<T>& t);

**bool** Equal(TreeNode<T>\* a , TreeNode<T>\* b);

void setup1();

void setup2();

void output();

Write 2 setup and display functions to establish and display 2 example binary trees shown below. Then **demonstrate** the functions you wrote.



1. (20%) Threaded binary tree

Write a C++ template class for threaded binary trees: ThreadedTree according to the tree node structure as shown in Figure 5.21 in textbook. Then write C++ functions for:

1. Forward iterator by sequencing through the nodes in inorder.
2. Traverse a threaded binary tree in postorder.
3. Traverse a threaded binary tree in preorder.
4. Insert a new node r as the right child of node s in a threaded binary tree.
5. Insert a new node l as the left child of node s in a threaded binary tree.

Use binary tree (b) shown above as example to construct a threaded binary tree and demonstrate the above five functions you implemented.

1. (20%)

(a) Write a C++ class MaxHeap that derives from the abstract base class in **ADT 5.2 MaxPQ** and implement all the virtual functions of MaxPQ.

**ADT 5.2 MaxPQ**

**template** <**class** T>

**class** MaxPQ {

**public**:

**virtual** ~MaxPQ() {} // virtual destructor

**virtual** **bool** IsEmpty() **const** = 0; //return **true** iff empty

**virtual** **const** T& Top() **const** = 0; //return reference to the max

**virtual** **void** Push(**const** T&) = 0;

**virtual** **void** Pop() = 0;

};

The class MaxHeap should include a **bottom up heap construction initialization** function, the push function for inserting a new key and pop function for deleting and the max key. You should also write a client function (main()) to demonstrate how to construct a max heap from a sequence of 13 integer number: 50, 5, 30, 40, 80, 35, 2, 20, 15, 60, 70, 8, 10 by using a series of 13 pushes and by bottom up initialization. Add necessary code for displaying your result.

(b) Write a C++ abstract class similar to ADT 5.2 for the ADT **MinPQ**, which defines a min priority queue. Then write a C++ class MinHeap that derives from this abstract class and implement all the virtual functions of MinPQ.

The class MinHeap should include a **bottom up heap construction initialization** function, the push function for inserting a new key and pop function for deleting and the min key. You should also write a client function (main()) to demonstrate how to construct a min heap from a sequence of 13 integer number: 50, 5, 30, 40, 80, 35, 2, 20, 15, 60, 70, 8, 10 by using a series of 13 pushes and by bottom up initialization. Add necessary code for displaying your result.

1. (20%)

A Dictionary abstract class is shown in **ADT5.3 Dictionary**. Write a C++ class BST that derives from Dictionary and implement all the virtual functions. In addition, also implement

Pair<K, E>\* RankGet(**int** r),

**void** Split(**const** K& k, BST<K, E>& small, pair<K, E>\*& mid, BST<K, E>& big)

**ADT5.3 Dictionary**

**template** <**class** K, **class** E>

**class** Dictionary {

**public**:

**virtual** **bool** IsEmptay() **const** = 0; // return true if dictionary is empty

**virtual** pair <K, E>\* Get(const K&) **const** = 0;

// return pointer to the pair w. specified key

**virtual** **void** Insert(**const** Pair <K, E>&) = 0;

// insert the given pair; if key ia a duplicate, update associate element

**virtual** **void** Delete(**const** K&) = 0; // delete pair w. specified key

};

Use a sequence of 13 integer number: 50, 5, 30, 40, 80, 35, 2, 20, 15, 60, 70, 8, 10 as 13 key values (type int) to generate 13 (key, element) (e.g., element can be simple char) pairs to construct the BST. Demonstrate your functions using this set of records.