

## EECS2040 Data Structure Hw #4 (Chapter 5 Tree)

due date 5/30/2021, 23:59

by 107061123 孫元駿

### Part 2 Coding

You should submit:

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

(b) Show the execution trace of your program.

1. (30%) Develop a complete C++ template class for binary trees shown in **ADT 5.1**. You must include a **constructor**, **copy constructor**, **destructor**, the four traversal methods, functions in **ADT 5.1**, ... as shown below.

```
void Inorder()
```

```
void Preorder()
```

```
void Postorder()
```

```
void LevelOrder()
```

```
void NonrecInorder()
```

```
void NoStackInorder()
```

```
bool operator == (const BinaryTree& t) const
```

Write 2 setup functions to establish 2 example binary trees (use the two trees in Part 1 Question

5). Then **demonstrate** the functions you wrote.

### ADT 5.1 BinaryTree

```
template<class T>
```

```
class BinaryTree
```

```
{ // objects: A finite set of nodes either empty or consisting
```

```
    // of a root node, left BinaryTree and right BinaryTree
```

```
public:
```

```
    BinaryTree(); // constructor for an empty binary tree
```

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

```
    BinaryTree(BinaryTree<T>& bt1, T& item, BinaryTree<T>& bt2);
```

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

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

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

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

```
};
```

```
~/Desktop/data_structure/hw4/part2/hw4_1  P master  g++ -std=c++17 main.cpp 20:36:14
~/Desktop/data_structure/hw4/part2/hw4_1  P master  ./a.out 20:36:18
Construct a new empty Tree.
The Tree is empty.

Give the Tree some value:
A-B*C*D+E

Construct a new CopyTree by copy constructor.
Tree is equal to CopyTree.

Inorder: A-B*C*D+E
Inorder(copy): A-B*C*D+E

NonrecInorder: A-B*C*D+E
NonrecInorder(copy): A-B*C*D+E

Preorder: +-+ABCDE
Preorder(copy): +-+ABCDE

Postorder: AB-C*D+E+
Postorder(copy): AB-C*D+E+

LevelOrder: ++E*D-CAB
LevelOrder(copy): ++E*D-CAB

NoStackInorder: A-B*C*D+E
NoStackInorder(copy): A-B*C*D+E

The data of root: +
The left tree of the Tree (Inorder): A-B*C*D
The right tree of the Tree (Inorder): E

~/Desktop/data_structure/hw4/part2/hw4_1  P master  20:36:30
```

The execution trace of my program:

First construct a empty tree T and show if the tree is empty by T.IsEmpty().

Then give T some value. And create a new CopyTree CT to copy T by the copy constructor  $CT = Tree(T)$  and check if T and CT are equal by  $T == CT$ .

Then, show Inorder, NonrecInorder, Preorder, Postorder, LevelOrder and NonStackInorder of T and CT.

Last, show the root of T by T.RootData(), and left child and right child by T.LeftChild() and T.RightChild() and show both tree (Inorder).

2. (35%)

(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;
};
```

(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.

Use Part1 Q6 example to demonstrate your program.

```
~/Desktop/data_structure/hw4/part2/hw4_2 g++ -Wall -std=c++17 main.cpp 23:46:27
~/Desktop/data_structure/hw4/part2/hw4_2 ./a.out 23:46:34
Max Heap is empty.
Min Heap is empty.

How many elements in the Max/Min heaps: 11
Insert 11 elements: 50 5 30 40 80 35 2 20 15 60 70

The max element of Max Heap: 80
The min element of Min Heap: 2

Max heap: 80 70 35 20 60 30 2 5 15 40 50
Min heap: 2 15 5 20 60 35 30 50 40 80 70

Pop Max heap.
Pop Min heap.

The max element of Max Heap: 70
The min element of Min Heap: 5

Max heap: 70 60 35 20 50 30 2 5 15 40
Min heap: 5 15 30 20 60 35 70 50 40 80
~/Desktop/data_structure/hw4/part2/hw4_2 23:46:39
```

The execution trace of my program:

First, construct a max heap Q and a min heap H, and show if Q and H are all empty tree by

Q.IsEmpty() and H.IsEmpty().

Then, insert some value into Q and H using Part1 Q6 example by Q.Push(<element>) and H.Push(<element>). And show Q and H by Q.show() and H.show().

Then, pop one element from Q and pop one from H, and show the max element of Q by Q.Top() and show the min element of H by H.Top().

Last, show Q and P (after pop out one element).

3. (35%) 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 IsEmpty() 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 is a duplicate, update associate element
```

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

```
};
```

Use Part1 Q6 example as 11 key values (type int) to generate 11 (key, element) (e.g., element can be simple char) pairs to construct the BST. Demonstrate your functions using this set of records.

```

sumyuanjun@KendideMBP:~/Desktop/data_structure/hw4/part2/hw4_3
~/Desktop/data_structure/hw4/part2/hw4_3 master g++ -Wall -std=c++17 main.cpp 23:24:21
~/Desktop/data_structure/hw4/part2/hw4_3 master ./a.out 23:24:27
Create a new empty BST.
BST is empty.

How many elements are in the BST: 11
Input (key,value): 50 a 5 b 30 c 40 d 80 e 35 f 2 g 20 h 15 i 60 j 70 k

BST PostOrder: (2,g) (15,i) (20,h) (35,f) (40,d) (30,c) (5,b) (70,k) (60,j) (80,e) (50,a)
Get(70) = (70,k)
RankGet(3) = (15,i)

Delete(50).
Delete(5).
Delete(60).

BST PostOrder: (2,g) (20,h) (35,f) (40,d) (30,c) (15,i) (80,e) (70,k)
Get(70) = (70,k)
RankGet(3) = (20,h)

Split(key = 35)
Mid: (35,f)
Small BST PostOrder: (2,g) (20,h) (30,c) (15,i)
Big BST PostOrder: (40,d) (80,e) (70,k)
~/Desktop/data_structure/hw4/part2/hw4_3 master 23:24:33

```

The execution trace of my program:

First, construct a empty BST call T and check if it is empty by T.IsEmpty().

Then, insert some elements(key, value) to T by T.Insert(<key>, <value>) and print the tree by postorder type by T.Postorder(). Then get the node of key is 70 by T.Get(70) and get the node that it is rank 3 by T.RankGet(3).

Then, delete the elements that keys are 50, 5 and 60 by T.Delete(50), T.Delete(5) and T.Delete(60). After that show the postorder and the node that its key is 70 and the node that it is rank 3 like above.

Last, split the tree at key equal to 35 by T.Split(35, Small, Mid, Big), and store to different BST tree, and show the mid element, and the Small BST and Big BST.