EECS2040 Data Structure Hw #4 (Chapter 5 Tree)
due date 5/22/2022, 23:59
by 108011235 陳昭維

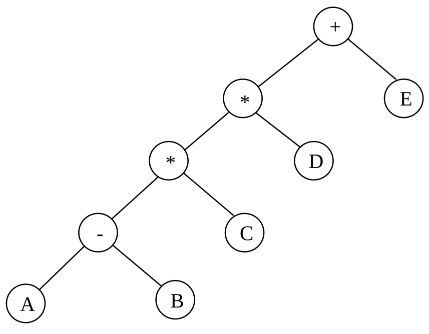
Part 2 Coding (5% of final Grade)

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..

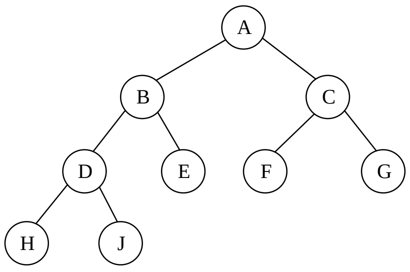
1. (30%) Develop a complete C++ template class for binary trees shown in **ADT 5.1**. You must include a **constructor**, **copy constructor**, **destructor**, the traversal methods as shown below, and functions in **ADT 5.1**.
   void Inorder()
   void Preorder()
   void Postorder()
   void LevelOrder()
   void NonrecInorder()
   void NoStackInorder()
   bool operator == (**const** BinaryTree& t) **const**

```
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
};
```

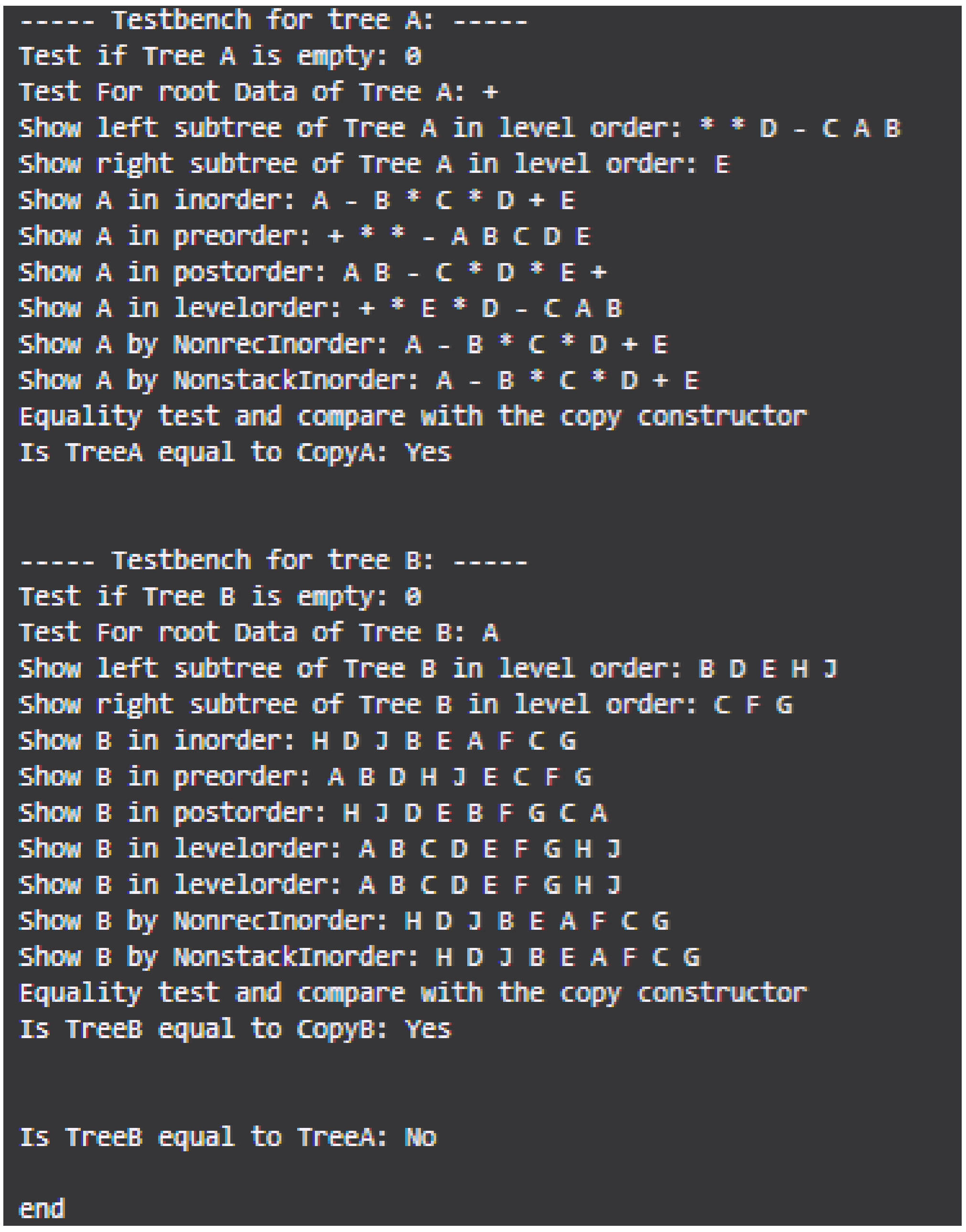
Write 2 setup and display functions to establish and display 2 example binary trees shown below. Then **demonstrate** the functions you wrote.



(a)



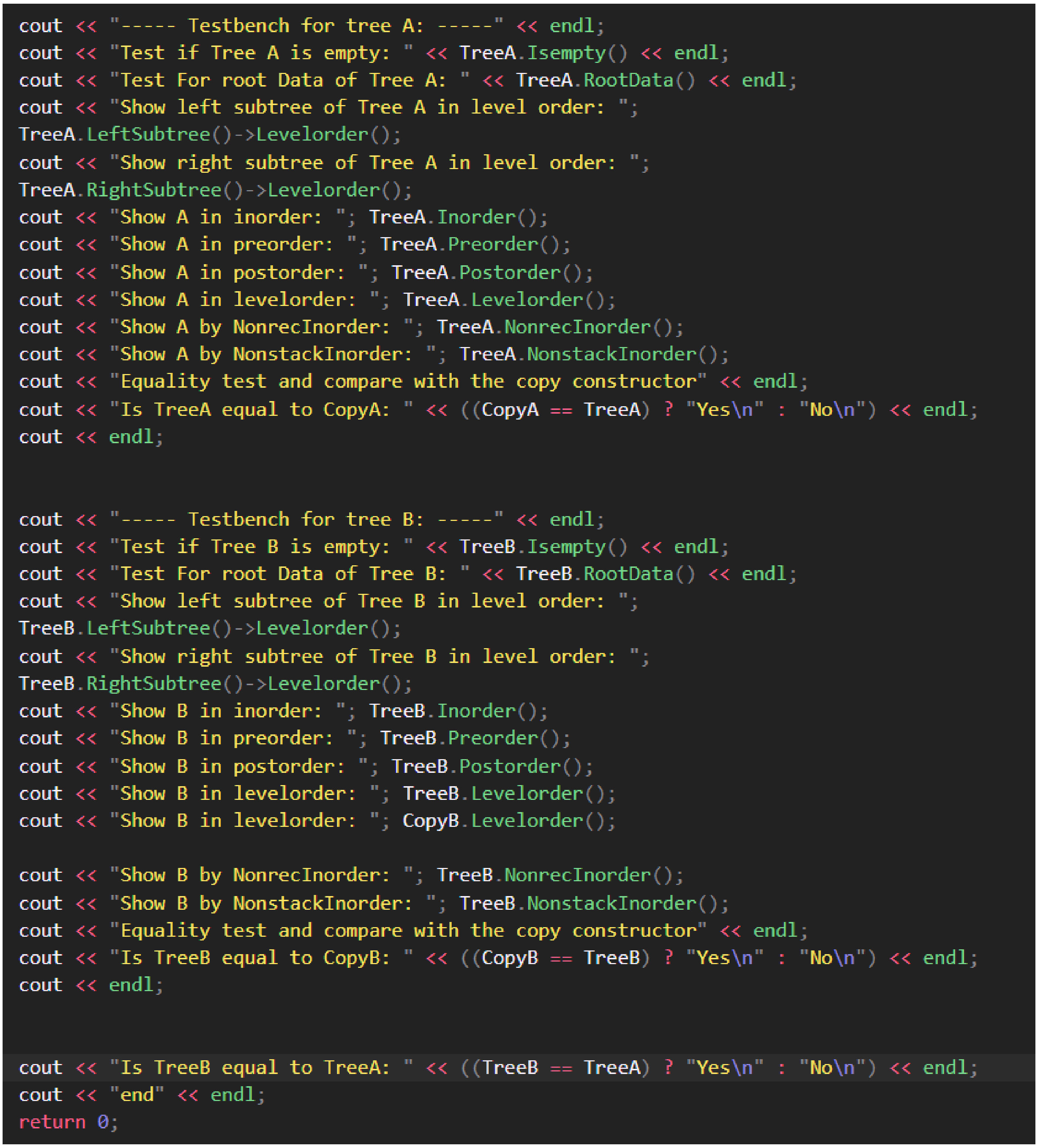
(b)



```
----- Testbench for tree A: -----
Test if Tree A is empty: 0
Test For root Data of Tree A: +
Show left subtree of Tree A in level order: * * D - C A B
Show right subtree of Tree A in level order: E
Show A in inorder: A - B * C * D + E
Show A in preorder: + * * - A B C D E
Show A in postorder: A B - C * D * E +
Show A in levelorder: + * E * D - C A B
Show A by NonrecInorder: A - B * C * D + E
Show A by NonstackInorder: A - B * C * D + E
Equality test and compare with the copy constructor
Is TreeA equal to CopyA: Yes


----- Testbench for tree B: -----
Test if Tree B is empty: 0
Test For root Data of Tree B: A
Show left subtree of Tree B in level order: B D E H J
Show right subtree of Tree B in level order: C F G
Show B in inorder: H D J B E A F C G
Show B in preorder: A B D H J E C F G
Show B in postorder: H J D E B F G C A
Show B in levelorder: A B C D E F G H J
Show B in levelorder: A B C D E F G H J
Show B by NonrecInorder: H D J B E A F C G
Show B by NonstackInorder: H D J B E A F C G
Equality test and compare with the copy constructor
Is TreeB equal to CopyB: Yes


Is TreeB equal to TreeA: No

end
```

```
  TreeNode<char> nodesOfA[10];
  TreeNode<char> nodesOfB[10];

  nodesOfA[9].SetData('B');
  nodesOfA[8].SetData('A');
  nodesOfA[7].SetData('C');
  nodesOfA[6].SetData('-');
  nodesOfA[6].SetLeftChild(nodesOfA + 8);
  nodesOfA[6].SetRightChild(nodesOfA + 9);
  nodesOfA[5].SetData('D');
  nodesOfA[4].SetData('*');
  nodesOfA[4].SetLeftChild(nodesOfA + 6);
  nodesOfA[4].SetRightChild(nodesOfA + 7);
  nodesOfA[3].SetData('E');
  nodesOfA[2].SetData('*');
  nodesOfA[2].SetLeftChild(nodesOfA + 4);
  nodesOfA[2].SetRightChild(nodesOfA + 5);
  nodesOfA[1].SetData('+');
  nodesOfA[1].SetLeftChild(nodesOfA + 2);
  nodesOfA[1].SetRightChild(nodesOfA + 3);

  nodesOfB[9].SetData('J');
  nodesOfB[8].SetData('H');
  nodesOfB[7].SetData('G');
  nodesOfB[6].SetData('F');
  nodesOfB[5].SetData('E');
  nodesOfB[4].SetData('D');
  nodesOfB[4].SetLeftChild(nodesOfB + 8);
  nodesOfB[4].SetRightChild(nodesOfB + 9);
  nodesOfB[3].SetData('C');
  nodesOfB[3].SetLeftChild(nodesOfB + 6);
  nodesOfB[3].SetRightChild(nodesOfB + 7);
  nodesOfB[2].SetData('B');
  nodesOfB[2].SetLeftChild(nodesOfB + 4);
  nodesOfB[2].SetRightChild(nodesOfB + 5);
  nodesOfB[1].SetData('A');
  nodesOfB[1].SetLeftChild(nodesOfB + 2);
  nodesOfB[1].SetRightChild(nodesOfB + 3);
```

```
BinaryTree<char> TreeA(nodesOfA + 1);
BinaryTree<char> CopyA(TreeA);
BinaryTree<char> TreeB(nodesOfB + 1);
BinaryTree<char> CopyB(TreeB);
```



```cpp
cout << "----- Testbench for tree A: -----" << endl;
cout << "Test if Tree A is empty: " << TreeA.Isempty() << endl;
cout << "Test For root Data of Tree A: " << TreeA.RootData() << endl;
cout << "Show left subtree of Tree A in level order: ";
TreeA.LeftSubtree()->Levelorder();
cout << "Show right subtree of Tree A in level order: ";
TreeA.RightSubtree()->Levelorder();
cout << "Show A in inorder: "; TreeA.Inorder();
cout << "Show A in preorder: "; TreeA.Preorder();
cout << "Show A in postorder: "; TreeA.Postorder();
cout << "Show A in levelorder: "; TreeA.Levelorder();
cout << "Show A by NonrecInorder: "; TreeA.NonrecInorder();
cout << "Show A by NonstackInorder: "; TreeA.NonstackInorder();
cout << "Equality test and compare with the copy constructor" << endl;
cout << "Is TreeA equal to CopyA: " << ((CopyA == TreeA) ? "Yes\n" : "No\n") << endl;
cout << endl;


cout << "----- Testbench for tree B: -----" << endl;
cout << "Test if Tree B is empty: " << TreeB.Isempty() << endl;
cout << "Test For root Data of Tree B: " << TreeB.RootData() << endl;
cout << "Show left subtree of Tree B in level order: ";
TreeB.LeftSubtree()->Levelorder();
cout << "Show right subtree of Tree B in level order: ";
TreeB.RightSubtree()->Levelorder();
cout << "Show B in inorder: "; TreeB.Inorder();
cout << "Show B in preorder: "; TreeB.Preorder();
cout << "Show B in postorder: "; TreeB.Postorder();
cout << "Show B in levelorder: "; TreeB.Levelorder();
cout << "Show B in levelorder: "; CopyB.Levelorder();

cout << "Show B by NonrecInorder: "; TreeB.NonrecInorder();
cout << "Show B by NonstackInorder: "; TreeB.NonstackInorder();
cout << "Equality test and compare with the copy constructor" << endl;
cout << "Is TreeB equal to CopyB: " << ((CopyB == TreeB) ? "Yes\n" : "No\n") << endl;
cout << endl;


cout << "Is TreeB equal to TreeA: " << ((TreeB == TreeA) ? "Yes\n" : "No\n") << endl;
cout << "end" << endl;
return 0;
```

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.

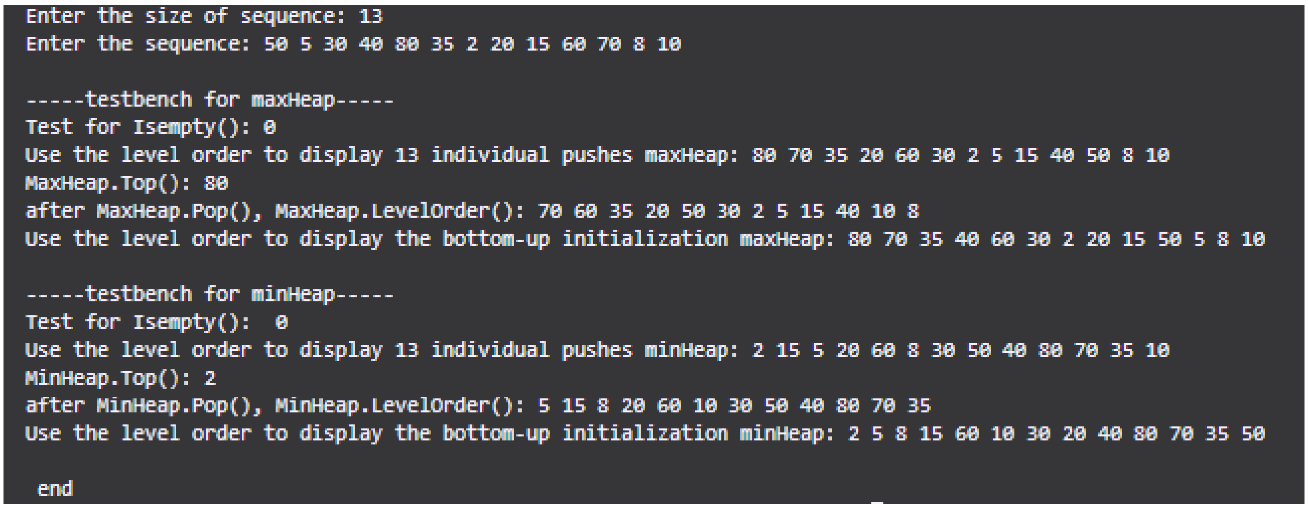
ADT 5.2 MaxPQ

```
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;
};
```

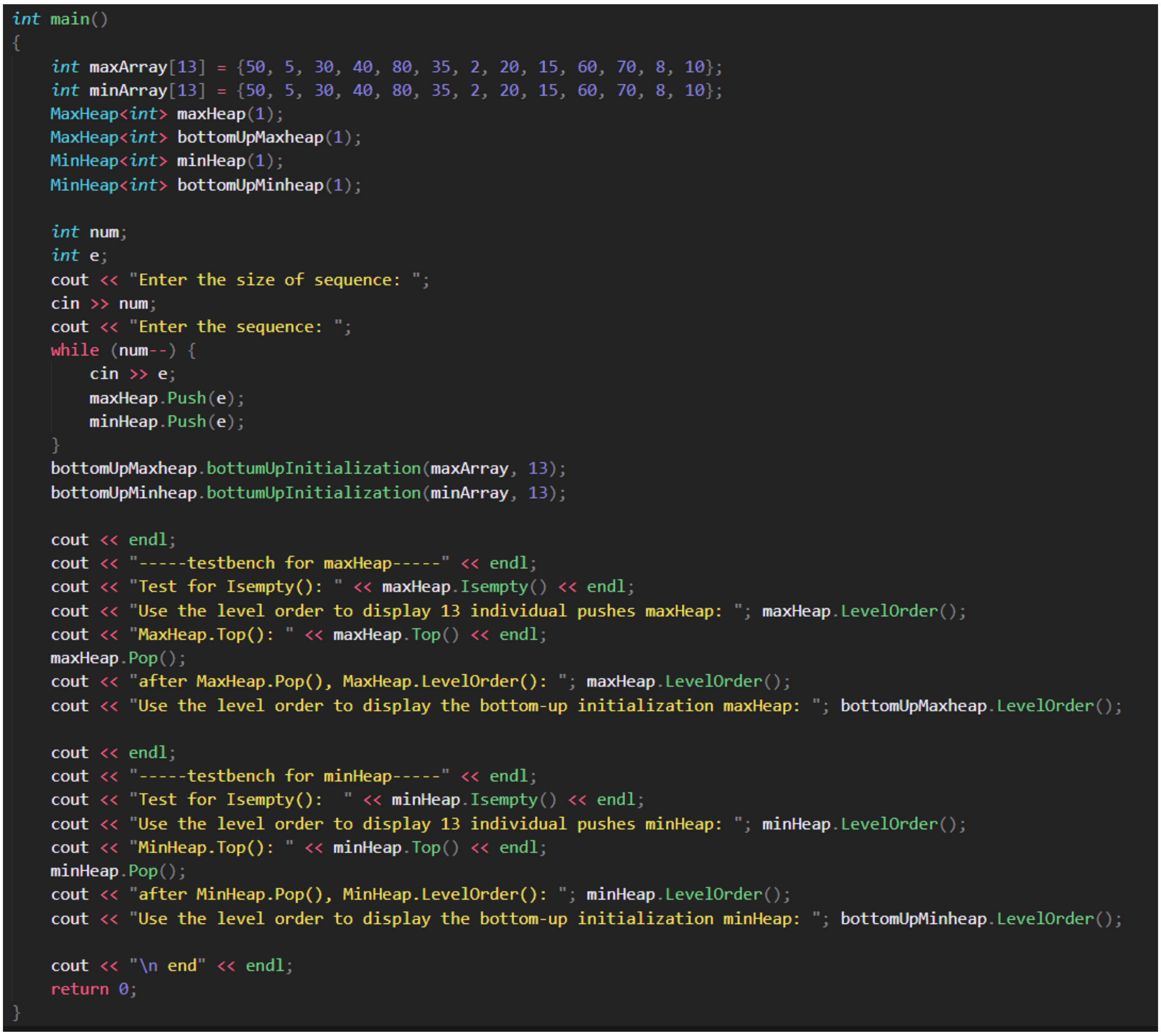
The class MaxHeap should include a **bottom up heap construction initialization** function, the push function for inserting a new key and pop function for deleting and the max key. You should also write a client function (main()) to demonstrate how to construct a max heap from a sequence of 13 integer 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.



```
Enter the size of sequence: 13
Enter the sequence: 50 5 30 40 80 35 2 20 15 60 70 8 10

-----testbench for maxHeap-----
Test for Isempty(): 0
Use the level order to display 13 individual pushes maxHeap: 80 70 35 20 60 30 2 5 15 40 50 8 10
MaxHeap.Top(): 80
after MaxHeap.Pop(), MaxHeap.LevelOrder(): 70 60 35 20 50 30 2 5 15 40 10 8
Use the level order to display the bottom-up initialization maxHeap: 80 70 35 40 60 30 2 20 15 50 5 8 10

-----testbench for minHeap-----
Test for Isempty():  0
Use the level order to display 13 individual pushes minHeap: 2 15 5 20 60 8 30 50 40 80 70 35 10
MinHeap.Top(): 2
after MinHeap.Pop(), MinHeap.LevelOrder(): 5 15 8 20 60 10 30 50 40 80 70 35
Use the level order to display the bottom-up initialization minHeap: 2 5 8 15 60 10 30 20 40 80 70 35 50

 end
```

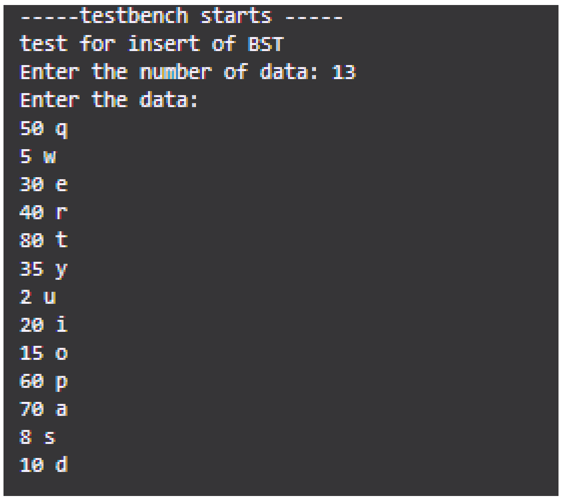


```cpp
int main()
{
    int maxArray[13] = {50, 5, 30, 40, 80, 35, 2, 20, 15, 60, 70, 8, 10};
    int minArray[13] = {50, 5, 30, 40, 80, 35, 2, 20, 15, 60, 70, 8, 10};
    MaxHeap<int> maxHeap(1);
    MaxHeap<int> bottomUpMaxheap(1);
    MinHeap<int> minHeap(1);
    MinHeap<int> bottomUpMinheap(1);

    int num;
    int e;
    cout << "Enter the size of sequence: ";
    cin >> num;
    cout << "Enter the sequence: ";
    while (num--) {
        cin >> e;
        maxHeap.Push(e);
        minHeap.Push(e);
    }
    bottomUpMaxheap.bottumUpInitialization(maxArray, 13);
    bottomUpMinheap.bottumUpInitialization(minArray, 13);

    cout << endl;
    cout << "-----testbench for maxHeap-----" << endl;
    cout << "Test for Isempty(): " << maxHeap.Isempty() << endl;
    cout << "Use the level order to display 13 individual pushes maxHeap: "; maxHeap.LevelOrder();
    cout << "MaxHeap.Top(): " << maxHeap.Top() << endl;
    maxHeap.Pop();
    cout << "after MaxHeap.Pop(), MaxHeap.LevelOrder(): "; maxHeap.LevelOrder();
    cout << "Use the level order to display the bottom-up initialization maxHeap: "; bottomUpMaxheap.LevelOrder();

    cout << endl;
    cout << "-----testbench for minHeap-----" << endl;
    cout << "Test for Isempty():  " << minHeap.Isempty() << endl;
    cout << "Use the level order to display 13 individual pushes minHeap: "; minHeap.LevelOrder();
    cout << "MinHeap.Top(): " << minHeap.Top() << endl;
    minHeap.Pop();
    cout << "after MinHeap.Pop(), MinHeap.LevelOrder(): "; minHeap.LevelOrder();
    cout << "Use the level order to display the bottom-up initialization minHeap: "; bottomUpMinheap.LevelOrder();

    cout << "\n end" << endl;
    return 0;
}
```

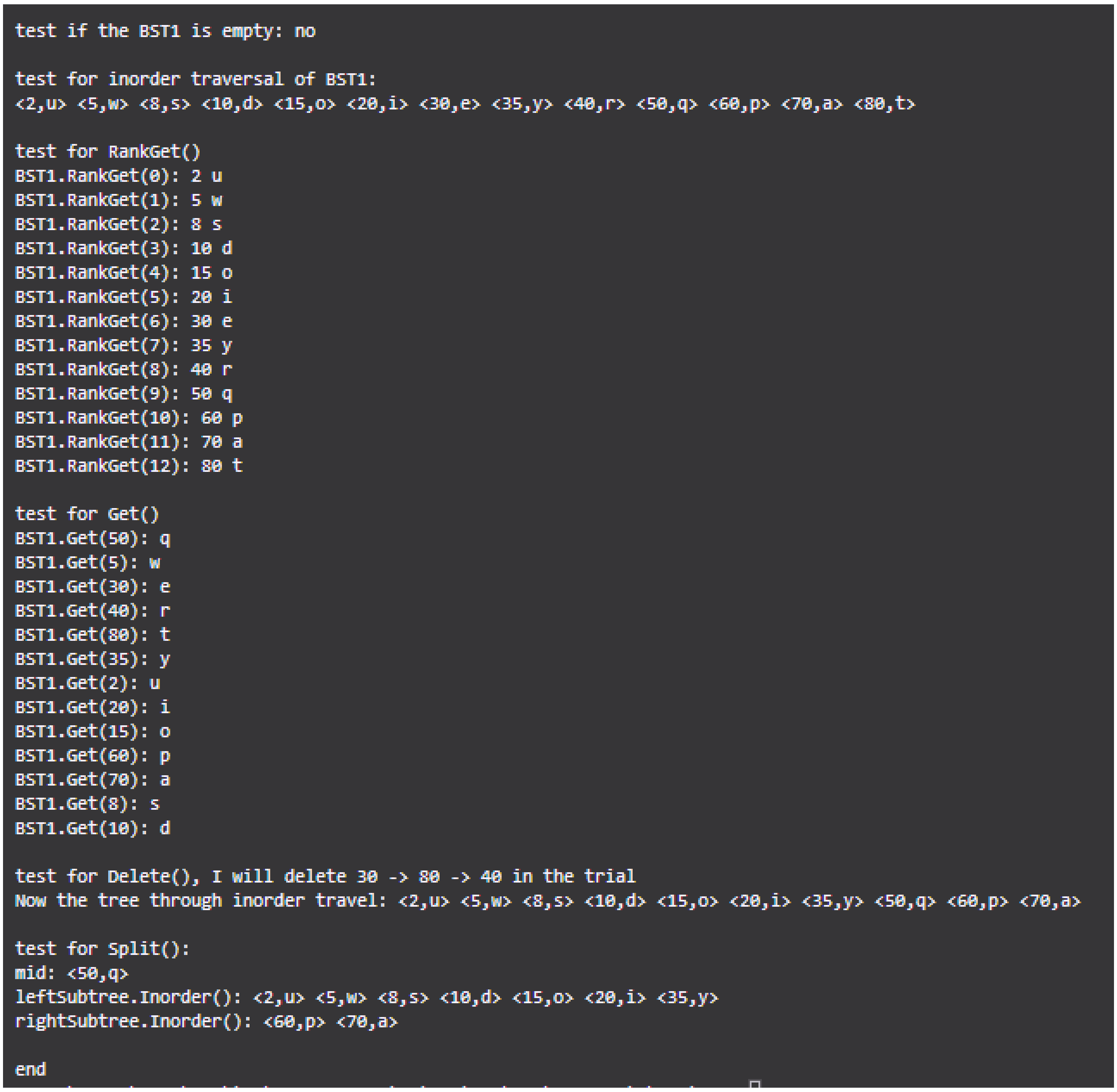
3. (35%) A Dictionary abstract class is shown in **ADT5.3 Dictionary**. Write a C++ class BST that derives from Dictionary and implement all the virtual functions. In addition, also implement
Pair<K, E>* RankGet(**int** r),
void Split(**const** K& k, BST<K, E>& small, pair<K, E>*& mid, BST<K, E>& big)

ADT5.3 Dictionary
template <**class** K, **class** E>
class Dictionary {
public:
    virtual bool 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
};

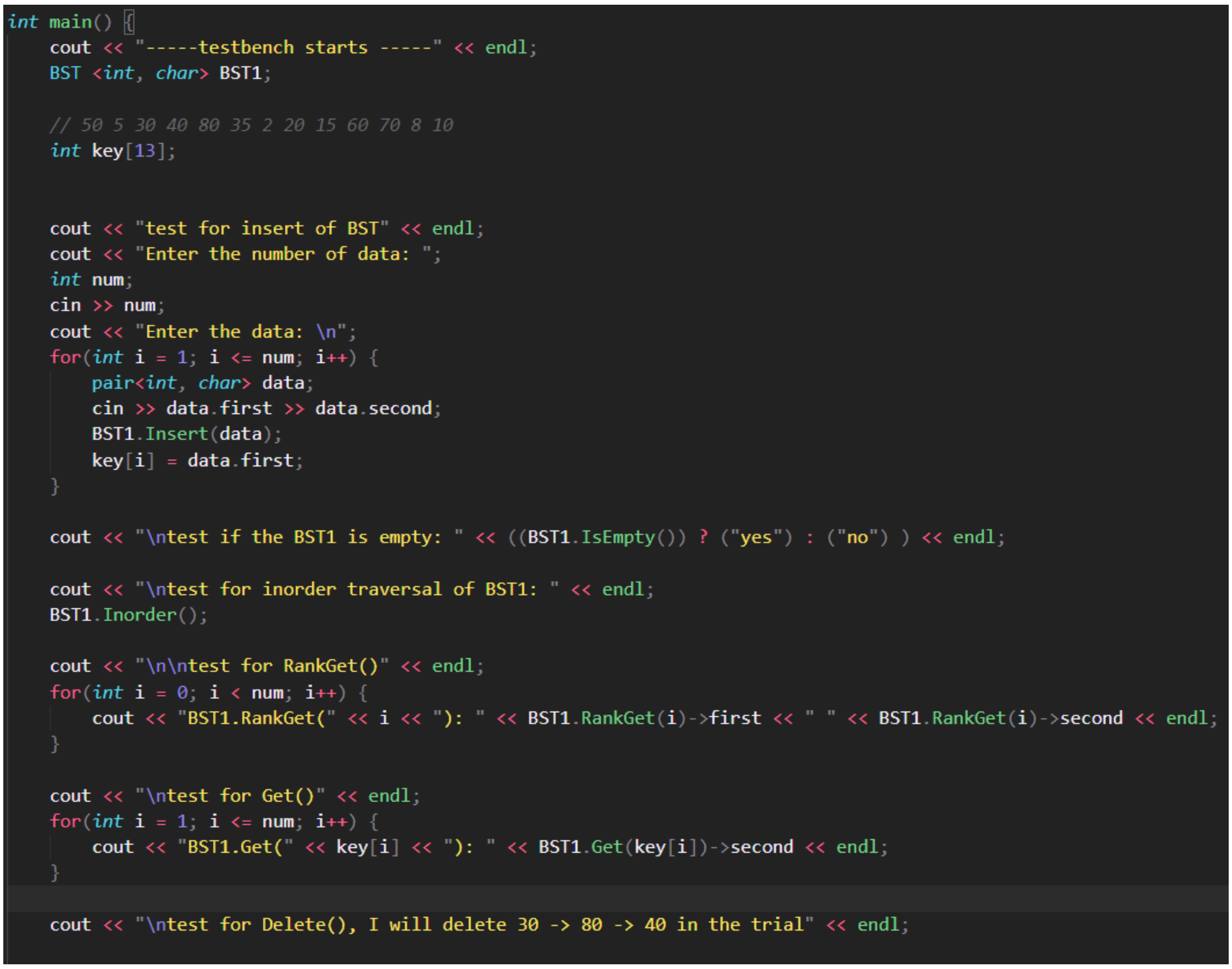
Use a sequence of 13 integer number: 50, 5, 30, 40, 80, 35, 2, 20, 15, 60, 70, 8, 10 as 13 key values (type int) to generate 13 (key, element) (e.g., element can be simple char) pairs to construct the BST. Demonstrate your functions using this set of records.



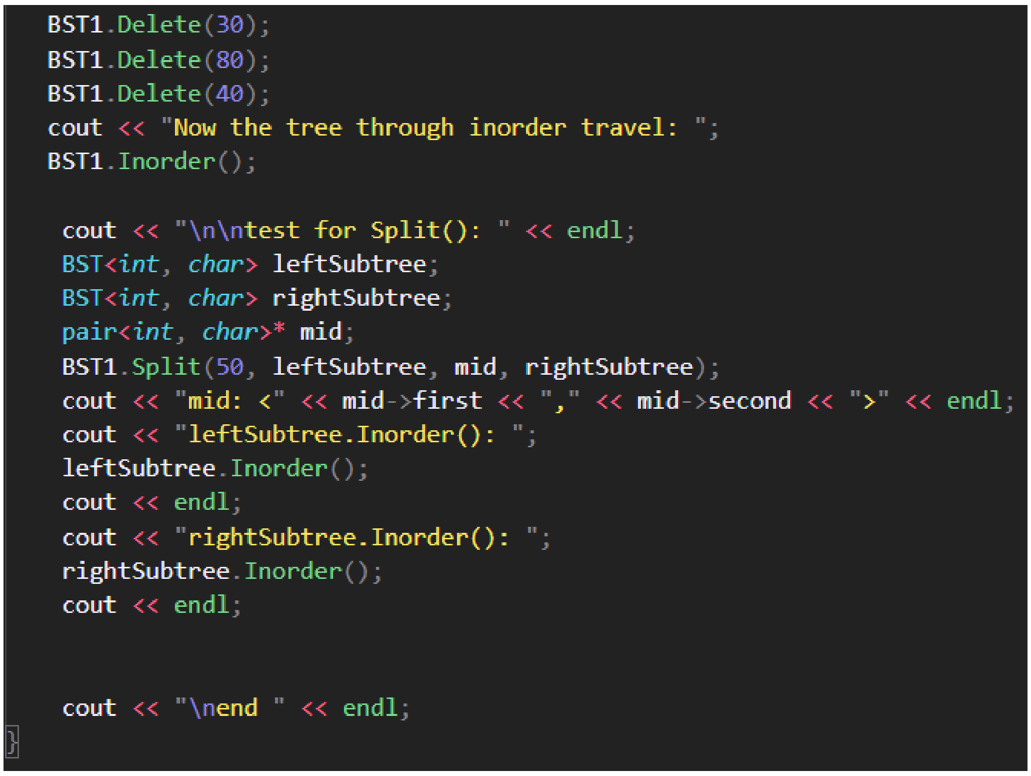
```
-----testbench starts -----
test for insert of BST
Enter the number of data: 13
Enter the data:
50 q
5 w
30 e
40 r
80 t
35 y
2 u
20 i
15 o
60 p
70 a
8 s
10 d
```



```
test if the BST1 is empty: no

test for inorder traversal of BST1:
<2,u> <5,w> <8,s> <10,d> <15,o> <20,i> <30,e> <35,y> <40,r> <50,q> <60,p> <70,a> <80,t>

test for RankGet()
BST1.RankGet(0): 2 u
BST1.RankGet(1): 5 w
BST1.RankGet(2): 8 s
BST1.RankGet(3): 10 d
BST1.RankGet(4): 15 o
BST1.RankGet(5): 20 i
BST1.RankGet(6): 30 e
BST1.RankGet(7): 35 y
BST1.RankGet(8): 40 r
BST1.RankGet(9): 50 q
BST1.RankGet(10): 60 p
BST1.RankGet(11): 70 a
BST1.RankGet(12): 80 t

test for Get()
BST1.Get(50): q
BST1.Get(5): w
BST1.Get(30): e
BST1.Get(40): r
BST1.Get(80): t
BST1.Get(35): y
BST1.Get(2): u
BST1.Get(20): i
BST1.Get(15): o
BST1.Get(60): p
BST1.Get(70): a
BST1.Get(8): s
BST1.Get(10): d

test for Delete(), I will delete 30 -> 80 -> 40 in the trial
Now the tree through inorder travel: <2,u> <5,w> <8,s> <10,d> <15,o> <20,i> <35,y> <50,q> <60,p> <70,a>

test for Split():
mid: <50,q>
leftSubtree.Inorder(): <2,u> <5,w> <8,s> <10,d> <15,o> <20,i> <35,y>
rightSubtree.Inorder(): <60,p> <70,a>

end
```



```cpp
int main() {
    cout << "-----testbench starts -----" << endl;
    BST <int, char> BST1;

    // 50 5 30 40 80 35 2 20 15 60 70 8 10
    int key[13];


    cout << "test for insert of BST" << endl;
    cout << "Enter the number of data: ";
    int num;
    cin >> num;
    cout << "Enter the data: \n";
    for(int i = 1; i <= num; i++) {
        pair<int, char> data;
        cin >> data.first >> data.second;
        BST1.Insert(data);
        key[i] = data.first;
    }

    cout << "\ntest if the BST1 is empty: " << ((BST1.IsEmpty()) ? ("yes") : ("no") ) << endl;

    cout << "\ntest for inorder traversal of BST1: " << endl;
    BST1.Inorder();

    cout << "\n\ntest for RankGet()" << endl;
    for(int i = 0; i < num; i++) {
        cout << "BST1.RankGet(" << i << "): " << BST1.RankGet(i)->first << " " << BST1.RankGet(i)->second << endl;
    }

    cout << "\ntest for Get()" << endl;
    for(int i = 1; i <= num; i++) {
        cout << "BST1.Get(" << key[i] << "): " << BST1.Get(key[i])->second << endl;
    }

    cout << "\ntest for Delete(), I will delete 30 -> 80 -> 40 in the trial" << endl;
```



```cpp
    BST1.Delete(30);
    BST1.Delete(80);
    BST1.Delete(40);
    cout << "Now the tree through inorder travel: ";
    BST1.Inorder();

     cout << "\n\ntest for Split(): " << endl;
     BST<int, char> leftSubtree;
     BST<int, char> rightSubtree;
     pair<int, char>* mid;
     BST1.Split(50, leftSubtree, mid, rightSubtree);
     cout << "mid: <" << mid->first << "," << mid->second << ">" << endl;
     cout << "leftSubtree.Inorder(): ";
     leftSubtree.Inorder();
     cout << endl;
     cout << "rightSubtree.Inorder(): ";
     rightSubtree.Inorder();
     cout << endl;


     cout << "\nend " << endl;
}
```