CSE 2020 Lab 11 Maps

Lab Exercise

Implement the Map ADT in a file `Map.cpp` as shown below.

```
// Map.cpp
#ifndef MAP_H_
#define MAP_H_

#include "Pair.cpp"
#include "MapSet.cpp"     // must have insert that returns iterator!!!

using namespace std;

template <typename K, typename V>
class Map
{
public:
  Map() {}

  void printMap()
  {
    typename Set<Pair<K,V>>::iterator itr = themap.begin();
    for (; itr != themap.end(); ++itr)
    {
      cout << (*itr).getKey() << ":" << (*itr).getValue() << endl;
    }
    return;
  }

  V & operator [](K index)
  {
    typename Set<Pair<K,V> >::iterator here;
    Pair<K,V> probe(index, V());
    here = themap.insert(probe);
    return (*here).getValue();
        }

  void remove(K & key)
  {
    Pair<K,V> probe;
    probe.getKey() = key;
    themap.remove(probe);

    return;
  }

 private:

  Set<Pair<K,V> > themap;
};

#endif
```

`Map.cpp` includes two header files `Pair.cpp` and `MapSet.cpp` which are defined as below.

```
// Pair.cpp
#ifndef PAIR_H_
#define PAIR_H_
using namespace std;

template <typename K, typename V>
class Pair
{
 public:

  Pair() {}
```

```
  Pair(K thekey):key(thekey)
  {}

  Pair(K thekey, V theval):key(thekey), value(theval)
  {}

  K& getKey()
  {
    return key;
  }

  V& getValue()
  {
    return value;
  }

  bool operator == (const Pair& rhs) const
  {
    return  key == rhs.key;
  }

  bool operator != (const Pair& rhs) const
  {
    return key != rhs.key;
  }

  bool operator < (const Pair& rhs) const
  {
    return key < rhs.key;
  }

  bool operator > (const Pair& rhs) const
  {
    return key > rhs.key;
  }

 private:
  K key;
  V value;
};
#endif
```

```
// MapSet.cpp
#ifndef SET_H
#define SET_H

#include <assert.h>
#include <iostream>
#include <stack>
using namespace std;

template <typename C>
class Set
{
public:
  Set( ) : root( nullptr )
  {  }

  ~Set( )
  {
      makeEmpty();
  }

  bool isEmpty( ) const
  {
      return root == nullptr;
  }

  const C & findMin( ) const
  {
      assert(!isEmpty());
      return findMin( root )->element;
  }

  const C & findMax( ) const
  {
      assert(!isEmpty());
      return findMax( root )->element;
```

```
    }

    bool contains( const C & x ) const
    {
        return contains( x, root );
    }

    void print( ) const
    {
        if( isEmpty( ) )
            cout << "Empty tree" << endl;
        else
            print( root );
    }

    void makeEmpty( )
    {
        makeEmpty( root );
    }

    void remove( const C & x )
    {
        remove( x, root );
    }

  private:

    struct BinaryNode
    {
        C element;
        BinaryNode* left;
        BinaryNode* right;

        BinaryNode( const C & theElement, BinaryNode* lt, BinaryNode* rt )
          : element( theElement ), left( lt ), right( rt )
        { }
    };

    BinaryNode* root;

  public:
    class iterator
    {
    public:
        iterator() : current(nullptr)
        {}

        // for Map
        iterator(BinaryNode*  p) : current(p)
        {}

        C & operator *()
        {
            return current->element;
        }

        // prefix
        iterator & operator++()
        {
            if (current == nullptr)
                return *this;

            if (current->right != nullptr)
            {
                current = current->right;
                while (current->left != nullptr)
                {
                    antes.push(current);
                    current = current->left;
                }
            }
            else
            {
                if (!antes.empty())
                {
                    current = antes.top();
                    antes.pop();
                }
                else
                    current = nullptr;
            }
```

```
            return *this;
        }

        iterator operator++(int)
        {
            iterator old = *this;
            ++(*this);
            return old;
        }

        bool operator ==(const iterator & rhs) const
        {
            return current == rhs.current;
        }

        bool operator !=(const iterator & rhs) const
        {
            return !(*this == rhs);
        }

    private:
        BinaryNode * current;
        stack<BinaryNode*> antes;

        iterator(BinaryNode* p, stack<BinaryNode*> st) : current(p), antes(st)
        {}

        friend class Set<C>;
};

    iterator begin()
    {
        BinaryNode* lmost = root;
        stack<BinaryNode*> nstack;

        while (lmost->left != nullptr)
        {
            nstack.push(lmost);
            lmost = lmost->left;
        }

        return iterator(lmost,nstack);
    }

    iterator end()
    {
        stack<BinaryNode*> emptystack;
        return iterator(nullptr, emptystack);
    }

     // for map.cpp, change void to iterator
      iterator insert(const C & x)
      {
          return insert( x, root );
      }

    private:

    // Internal method to find the smallest item in a subtree t.
    // Return the pointer to the node containing the smallest item.
      BinaryNode* findMin( BinaryNode* t ) const
      {
          if( t == nullptr )
              return nullptr;
          if( t->left == nullptr )
              return t;
          return findMin( t->left );
      }

      // Internal method to find the largest item in a subtree t.
      // Return the pointer to the node containing the largest item.
      BinaryNode* findMax( BinaryNode* t ) const
      {
          if( t != nullptr )
              while( t->right != nullptr )
                  t = t->right;
          return t;
      }

      // Internal method to test if an item is in a subtree.
```

```
    // x is item to search for.
    // t is the pointer to the root of the subtree.
    bool contains( const C & x, BinaryNode* t ) const
    {
        if( t == nullptr )
            return false;
        else if( x < t->element )
            return contains( x, t->left );
        else if( t->element < x )
            return contains( x, t->right );
        else
            return true;    // Match
    }

    void print( BinaryNode* t) const
    {
        if( t != nullptr )
        {
            print( t->left);
            cout << t->element << " - ";
            print( t->right);
        }
    }

    void makeEmpty( BinaryNode* & t )
    {
        if( t != nullptr )
        {
            makeEmpty( t->left );
            makeEmpty( t->right );
            delete t;
        }
        t = nullptr;
    }

    // Internal method to insert item into a subtree.
    // x is the item to insert.
    // t is the pointer to the root of the subtree.
    // Set the new root of the subtree.
    iterator insert( const C & x, BinaryNode* & t )
    {
        if( t == nullptr )
        {
                t = new BinaryNode{ x, nullptr, nullptr };
                return iterator(t);
        }
        else if( x < t->element )
            return insert( x, t->left );
        else if( t->element < x )
            return insert( x, t->right );
        else
            return iterator(t);  // Duplicate; do nothing
    }


    // Internal method to remove from a subtree.
    // x is the item to remove.
    // t is the pointer to the root of the subtree.
    // Set the new root of the subtree.
    void remove( const C & x, BinaryNode* & t )
    {
        if( t == nullptr )
            return;   // Item not found; do nothing
        if( x < t->element )
            remove( x, t->left );
        else if( t->element < x )
            remove( x, t->right );
        else if( t->left != nullptr && t->right != nullptr ) // Two children
        {
            t->element = findMin( t->right )->element;
            remove( t->element, t->right );
        }
        else
        {
            BinaryNode* oldNode = t;
            if ( t->left == nullptr )
                t = t->right;
            else
                t = t->left;
```

```
            delete oldNode;
        }
    }

};
#endif
```

Program your own file `lab11.cpp` in which your main() function will test the Map class. The `main()` function,

- Declares an instance of Map, `Map<int,string> studentDB;` suitable to hold student id (key) and student name (value).
- Prompts user to enter a sequence of student id and name, and inserts these pairs into the map object (the entered student ids should NOT be in sorted order).
- Calls the `printMap()` member function to print out the pairs in `studentDB`.
- Prompts user to enter a student id, and then prints the name of the student.
- Prompts user to enter a student id, and changes the student name as the user input name. Prints out the pairs in `studentDB`.
- Prompts user to enter a student id, and remove this pair from `studentDB`. Prints out the pairs in `studentDB`.

The expected result:

```
using index operator to insert new pairs:
Student ID? 1006
Student Name? Jim

Student ID? 1004
Student Name? Alice

Student ID? 1008
Student Name? Fred

Student ID? 1005
Student Name? Mary

Student ID? 1001
Student Name? Tom

Student ID? 0

Content of student database:
1001:Tom
1004:Alice
1005:Mary
1006:Jim
1008:Fred

Who you want to know? 1005
The corresponding name is: Mary

Change which one? 1006
... to what name? Bill

1001:Tom
1004:Alice
1005:Mary
1006:Bill
1008:Fred

Remove which one? 1005

1001:Tom
1004:Alice
1006:Bill
1008:Fred
```

Compilation

This lab exercise should be put under `cse2020/lab11` subdirectory.

```
$g++ -c Pair.cpp
$g++ -c MapSet.cpp
$g++ -c Map.cpp
$g++ lab11.cpp -o lab11
$./lab11
...
$script lab11log.txt
...
$exit
```

Hand In

- Pair.cpp, MapSet.cpp, and Map.cpp: the implementation files of the `Pair` class template, `Set` class template, and `Map` class template.
- lab11.cpp: the test file containing `main()` funcion.
- lab11log.txt: the script file which captures the result.
