CSE 330 LABORATORY -- Week 4, Spring 2018

Instructor: Kerstin Voigt

In this lab, you will be spared the laborious entry of the C++ code that implements the List ADT. However, you will be asked to extend class List with additional capabilities, and in doing so, you will familiarize yourself with the details of class List.

Start by obtaining a copy of the our text book author's implementation of a doubly-linked list data structure.

- Obtain a copy of Weiss_Link.h from the instructor's directory via the Linux 'cp' command.

Next, copy the following teststub, which will put the more interesting List functions to the test:

// File: Lab4Main.cpp

```
#include <iostream>
#include <string>
#include "Weiss_List.h"
using namespace std;

class Apple
{
public:
  Apple() :color{"red"} {}
  Apple(string col) :color{col} {}
  Apple(const Apple&  x) :color{x.color} {}
  Apple(Apple && x)  :color{x.color} {}

  Apple& operator= (const Apple& x)
  {
    if (this != &x)
      color = x.color;
    return *this;
  }

  Apple& operator=(Apple&& x)
  {
    std::swap(color,x.color);
    return *this;
  }

  bool operator ==(const Apple& x) {return x.color == color;}

  string get_color() const {return color;}

private:
  string color;
};
```

```
int main()
{
  List<int> mylst;
  for (int i = 1; i <= 10; i++)
    mylst.push_back(i*i);

  List<int>::iterator itr = mylst.begin();

  for (; itr != mylst.end(); ++itr)
    cout << *itr << " ";
  cout << endl;

  List<int>::iterator third = mylst.begin();
  ++third;
  ++third;

  mylst.insert(third,123);

  List<int>::iterator fifth = third;
  fifth++;
  fifth++;

  mylst.erase(fifth);

  itr = mylst.begin();
  for(; itr != mylst.end(); ++itr)
    cout << *itr << " ";
  cout << endl;

  List<int> another(mylst);

  itr = another.begin();
  for (; itr != another.end(); ++itr)
    cout << *itr << " ";
  cout << endl;

  cout << endl;
  List<Apple> myapps;

  Apple ap1;
  Apple ap2("green");
  Apple ap3(ap2);
  Apple ap4(Apple("yellow"));

  myapps.push_back(ap1);
  myapps.push_front(ap2);
  myapps.insert(myapps.begin(),ap3);
  myapps.insert(myapps.end(),ap4);

  List<Apple>::iterator  aitr = myapps.begin();

  for (; aitr != myapps.end(); ++aitr)
    cout << (*aitr).get_color() << " ";
  cout << endl << endl;
/*--------------- uncomment for Lab4 Exercise 2 ----------------------
Apple green("green");
  Apple blue("blue");

  if ( myapps.find(green) != myapps.end())
    cout << "Green apple found" << endl;
  else
    cout << "There is no green apple" << endl;
  if (myapps.find(blue) != myapps.end())
    cout << "Blue apple found" << endl;
  else
    cout << "There is no blue apple" << endl << endl;

----------------------------------------------------------------------*/

/* ------------- uncomment for Lab4 Exercise 3 ------------------------
  cout << "Now so_finding yellow, red, yellow, yellow apple ..."
       << endl << endl;

  myapps.so_find(ap4);
  myapps.so_find(ap1);
  myapps.so_find(ap4);
  myapps.so_find(Apple("yellow"));

  aitr = myapps.begin();
  for (; aitr != myapps.end(); ++aitr)
    cout << (*aitr).get_color() << endl;
  cout << endl << endl;

  cout << "And so_finding green apple ..." << endl << endl;

  myapps.so_find(Apple("green"));


  aitr = myapps.begin();
  for (; aitr != myapps.end(); ++aitr)
    cout << (*aitr).get_color() << endl;
  cout << endl << endl;

--------------------------------------------------------------------------*/

    return 0;
}
```

Exercise 1: You will have completed this exercise when you have arrived at this point, and your program **in Lab4Main.cpp compiles and run** as intended. You should have copied this code “mindfully”, and at this point gathered a good understanding on the List operations that allow storage and access to list elements, inclusive of functions ‘insert’ and ‘erase’ which allow the efficient (why “efficient?”) removal and insertion of values at arbitrary locations.

Make sure that your programs compiles and runs before you follow the instructions of Exercise 2, ‘/*------------- uncomment for Exercise 2 ------------‘.

Exercise 2: After you have uncommented the code for this exercise, realize that you will need to **add a new member functions to class List.h:**

```
template <typename T>
class List
{
    public:
    ….

   // somewhere down in the implementation …

  iterator find(const T& x)  { …}

};
```

The **List<T>::find function** is to find value x among the items stored in the linked list. If the list instance does not contain the item, the function is to return the iterator pointing to the tail of the list. If the item is found, the returned iterator is the one pointing to the Node that holds the value.

There are several ways in which you can implement this function. Find one that works for you and test it with the provided Lab4Main.cpp with the relevant test code section uncommented.

Exercise 3: Add to class List another “find” function. This function is to accomplish two things: (1) find a target value in the list, and (2) move the found value into the first (leftmost) position in the list. **This find function is SELF-ORGANIZING;** it will be beneficial in applications where there are repeated searches for the same target (or small subset of targets). Those targets that are likely to be of interest again will have made their way to the front of the list, and thus will be more cheaply found when needed again.

Function List<T>::self_org_find is to return the iterator to the Node with the leftmost value after the target value has been found and the list has self-organized. If the target in not contained in the list, the function is to return an iterator that points to the list’s tail.

```
template <typename T>
class List
{
  …
  iterator self_org_find(const T x) { … }
  …

};
```

Test your function with the provided Lab4Main.cpp program after uncommenting the relevant lines of code for Exercise 3.

Credit for this lab: (1) Make sure to sign the **signup sheet**. (2) Nothing else to turn in, but keep your code handy for an upcoming next homework assignment.