Swinburne University of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: 3, List ADT
Due date: May 12, 2022, 14:30
Lecturer: Dr. Markus Lumpe

Your name:_____ **Your student id:**_____

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30

Marker's comments:

Problem	Marks	Obtained
1	48	
2	28	
3	26	
4	30	
5	42	
Total	174	

Extension certification:

This assignment has been given an extension and is now due on _____

Signature of Convener:_____

Problem Set 3: List ADT

Review the template classs `DoublyLinkedList` and `DoublyLinkedListIterator` developed in tutorial 9. In addition, it might be beneficial to review also the lecture material regarding the construction of an abstract data type and memory management.

Start with the header files provided on Canvas, as they have been fully tested.

Using the template classes `DoublyLinkedList` and `DoublyLinkedListIterator`, implement the template class `List` as specified below:

```
#pragma once

#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"
#include <stdexcept>

template<typename T>
class List
{
private:
  // auxiliary definition to simplify node usage
  using Node = DoublyLinkedList<T>;

  Node* fRoot;      // the first element in the list
  size_t fCount;    // number of elements in the list

public:
  // auxiliary definition to simplify iterator usage
  using Iterator = DoublyLinkedListIterator<T>;

  List();                                       // default constructor
  List( const List& aOtherList );               // copy constructor
  List& operator=( const List& aOtherList );    // assignment operator
  ~List();                                      // destructor - frees all nodes

  bool isEmpty() const;                         // Is list empty?
  size_t size() const;                          // list size

  void push_front( const T& aElement );         // adds aElement at front
  void push_back( const T& aElement );          // adds aElement at back
  void remove( const T& aElement );             // remove first match from list

  const T& operator[]( size_t aIndex ) const;   // list indexer

  Iterator begin() const;                       // return a forward iterator
  Iterator end() const;                         // return a forward end iterator
  Iterator rbegin() const;                      // return a backwards iterator
  Iterator rend() const;                        // return a backwards end iterator

  // move features
  List( List&& aOtherList );                    // move constructor
  List& operator=( List&& aOtherList );         // move assignment operator
  void push_front( T&& aElement );              // adds aElement at front
  void push_back( T&& aElement );               // adds aElement at back
};
```

The template class `List` defines an "object adapter" for `DoublyLinkedList` objects (i.e., the list representation). Somebody else has already started with the implementation, but left the project unfinished. You find a header file for the incomplete `List` class on Canvas. This header file contains the specification of the template class `List` and the implementations for the destructor `~List()` and the `remove()` method. You need to implement the remaining member functions.

Problem 1

Implement the default constructor `List()`, and the methods `push_front()`, `size()`, `empty()`, as well as all iterator auxiliary methods first.

To make `List` work, we have to allocate list node elements on the heap using **`new`**. In doing so, you obtain a pointer to a `Node` object. The `DoublyLinkList` member functions, however, generally only accept references to `Node` objects. In order to satisfy this requirement, you need to deference the `Node` object pointer which gives you the `Node` object located in heap memory. This Node object is passed by reference (to a heap memory location) to the corresponding `DoublyLinkList` member function (i.e., `push_front()`).

You can use `#define P1` in `Main.cpp` to enable the corresponding test driver.

```
void testP1()
{
  using StringList = List<string>;

  string s1( "AAAA" );
  string s2( "BBBB" );
  string s3( "CCCC" );
  string s4( "DDDD" );

  cout << "Test of problem 1:" << endl;

  StringList lList;

  if ( !lList.empty() )
  {
    cerr << "Error: Newly created list is not empty." << endl;
  }

  lList.push_front( s4 );
  lList.push_front( s3 );
  lList.push_front( s2 );
  lList.push_front( s1 );

  // iterate from the top
  cout << "Top to bottom " << lList.size() << " elements:" << endl;
  for ( const string& element : lList )
  {
    cout << element << endl;
  }

  // iterate from the end
  cout << "Bottom to top " << lList.size() << " elements:" << endl;
  for ( StringList::Iterator iter = lList.rbegin(); iter != iter.rend(); iter-- )
  {
    cout << *iter << endl;
  }

  cout << "Completed" << endl;
}
```

The result should look like this. No errors should occur:

```
Test of problem 1:
Top to bottom 4 elements:
AAAA
BBBB
CCCC
DDDD
Bottom to top 4 elements:
DDDD
CCCC
BBBB
AAAA
Completed
```

Problem 2

Implement the method `push_back()`, which is just a variant of method `push_front()`. Do not reinvent the wheel. The method `push_back()` does not require a search. Remember that `fRoot` is 12 o' clock if the doubly-linked list nodes are viewed as a clock.

You can use `#define P2` in `Main.cpp` to enable the corresponding test driver.

```
void testP2()
{
  using StringList = List<string>;

  string s1( "AAAA" );
  string s2( "BBBB" );
  string s3( "CCCC" );
  string s4( "DDDD" );
  string s5( "EEEE" );
  string s6( "FFFF" );

  cout << "Test of problem 2:" << endl;

  StringList lList;

  lList.push_front( s4 );
  lList.push_front( s3 );
  lList.push_front( s2 );
  lList.push_front( s1 );
  lList.push_back( s5 );
  lList.push_back( s6 );

  // iterate from the top
  cout << "Bottom to top " << lList.size() << " elements:" << endl;
  for ( StringList::Iterator iter = lList.rbegin(); iter != iter.rend(); iter-- )
  {
    cout << *iter << endl;
  }

  cout << "Completed" << endl;
}
```

The result should look like this. No errors should occur:

```
Test of problem 2:
Bottom to top 6 elements:
FFFF
EEEE
DDDD
CCCC
BBBB
AAAA
Completed
```

Problem 3

Implement **`operator[]`**. The indexer has to search for the element that corresponds to `aIndex`. Also, `aIndex` may be out of bounds. Hence the indexer has to throw a `out_of_range` exception.

You can use `#define P3` in `Main.cpp` to enable the corresponding test driver.

```
void testP3()
{
  using StringList = List<string>;

  string s1( "AAAA" );
  string s2( "BBBB" );
  string s3( "CCCC" );
  string s4( "DDDD" );
  string s5( "EEEE" );
  string s6( "FFFF" );

  StringList lList;

  lList.push_front( s4 );
  lList.push_front( s3 );
  lList.push_front( s2 );
  lList.push_front( s1 );
  lList.push_back( s5 );
  lList.push_back( s6 );

  cout << "Test of problem 3:" << endl;

  try
  {
    cout << "Element at index 4: " << lList[4] << endl;
    lList.remove( s5 );
    cout << "Element at index 4: " << lList[4] << endl;

    cout << "Element at index 6: " << lList[6] << endl;
    cout << "Error: You should not see this text." << endl;
  }
  catch (out_of_range e)
  {
    cerr << "\nSuccessfully caught error: " << e.what() << endl;
  }

  cout << "Completed" << endl;
 }
```

The result should look like this:

```
Test of problem 3:
Element at index 4: EEEE
Element at index 4: FFFF
Element at index 6:
Successfully caught error: Index out of bounds.
Completed
```

Problem 4

Add proper copy control to the template class `List`, that is, implement the copy constructor and the assignment operator:

- `List( const List& aOtherList )`,
- `List& operator=( const List& aOtherList )`.

The copy constructor initializes an object using `aOtherList`. This process requires two steps:

- Perform default initializing of object.
- Assign `aOtherList` to this object. Remember this object is "**`*this`**".

The assignment operator overrides an initialized object. That is, the assignment operator must first free all resources and then copy the elements of `aOtherList`. Both steps are easy as you have already the necessary infrastructure. There is a convenient C++ idiom at your disposal. You can write, `this->~List()` to mean that you release all resources associated with this object, but do not delete this object itself. Remember, assignment must be secured against "accidental suicide."

You can use `#define P4` in `Main.cpp` to enable the corresponding test driver.

```
void testP4()
{
  using StringList = List<string>;

  string s1( "AAAA" );
  string s2( "BBBB" );
  string s3( "CCCC" );
  string s4( "DDDD" );
  string s5( "EEEE" );

  List<string> lList;

  cout << "Test of problem 4:" << endl;

  lList.push_front( s4 );
  lList.push_front( s3 );
  lList.push_front( s2 );

  List<string> copy( lList );

  // iterate from the top
  cout << "A - Top to bottom " << copy.size() << " elements:" << endl;

  for ( const string& element : copy )
  {
    cout << element << endl;
  }

  // override list
  lList = copy;

  lList.push_front( s1 );
  lList.push_back( s5 );

  // iterate from the top
  cout << "B - Bottom to top " << lList.size() << " elements:" << endl;

  for ( auto iter = lList.rbegin(); iter != iter.rend(); iter-- )
  {
    cout << *iter << endl;
  }

  cout << "Completed" << endl;
}
```

The result should look like this:

```
Test of problem 4:
A - Top to bottom 3 elements:
BBBB
CCCC
DDDD
B - Bottom to top 5 elements:
EEEE
DDDD
CCCC
BBBB
AAAA
Completed
```

Problem 5

Implement the move features:

- `List( List&& aOtherList )`,
- `List& operator=( List&& aOtherList )`,
- `void push_front( T&& aElement )`,
- `void push_back( T&& aElement )`.

The move features "steal" the memory of the argument. That is, calling the copy constructor or using the assignment operator leaves `aOtherList` empty. Similarly, calling `push_front()` and `push_back()`, respectively, leaves `aElement` empty.

The move variants are chosen be the compiler if the argument is a temporary or literal expression.

To force move semantics we have to use, where necessary, the `std::move()` function, which performs a type conversion on its argument that guarantees that the argument is an r-value reference.

You can use `#define P5` in `Main.cpp` to enable the corresponding test driver.

```
void testP5()
{
  using StringList = List<string>;

  string s2( "CCCC" );

  List<string> lList;

  cout << "Test of problem 5:" << endl;

  lList.push_front( string( "DDDD" ) );
  lList.push_front( std::move(s2) );
  lList.push_front( "BBBB" );

  if ( s2.empty() )
  {
    cout << "Successfully performed move operation." << endl;
  }
  else
  {
    cerr << "Error: Move operation failed." << endl;
  }

  cout << "A - Top to bottom " << lList.size() << " elements:" << endl;

  for ( const string& element : lList )
  {
    cout << element << endl;
  }

  List<string> move( std::move(lList) );

  if ( lList.empty() )
  {
    cout << "Successfully performed move operation." << endl;
  }
  else
  {
    cerr << "Error: Move operation failed." << endl;
  }

  // iterate from the top
  cout << "B - Top to bottom " << move.size() << " elements:" << endl;

  for ( const string& element : move )
  {
    cout << element << endl;
  }
```

```
  // override list
  lList = std::move(move);

  if ( move.empty() )
  {
    cout << "Successfully performed move operation." << endl;
  }
  else
  {
    cerr << "Error: Move operation failed." << endl;
  }

  lList.push_front( "AAAA" );
  lList.push_back( "EEEE" );

  // iterate from the top
  cout << "C - Bottom to top " << lList.size() << " elements:" << endl;

  for ( auto iter = lList.rbegin(); iter != iter.rend(); iter-- )
  {
    cout << *iter << endl;
  }

  cout << "Completed" << endl;
}
```

The result should look like this:

```
Test of problem 5:
Successfully performed move operation.
A - Top to bottom 3 elements:
BBBB
CCCC
DDDD
Successfully performed move operation.
B - Top to bottom 3 elements:
BBBB
CCCC
DDDD
Successfully performed move operation.
C - Bottom to top 5 elements:
EEEE
DDDD
CCCC
BBBB
AAAA
Completed
```

Submission deadline: Thursday, May 12, 2022, 14:30.

Submission procedure: PDF of printed code for **`ListPS3.h`.**