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:

The destructor enters a loop that continues as long as fRoot is not equal to nullptr. This loop iterates through the list, freeing one node at a time.

Inside the loop, it checks if fRoot (the first element in the list) is different from its previous element, which is accessed through &fRoot->getPrevious(). This is a check to determine if there is more than one node in the list. If there's only one node left (i.e., the last node), there is no previous node, and the destructor handles this case separately.

If there is more than one node in the list, it enters the if block and creates a temporary pointer, lTemp, which is a non-const version of the previous node. This is done using const_cast to cast away the const qualification.

The lTemp->isolate() method is called on the previous node. The purpose of this method is to disconnect the current node from the previous node and the next node (if any), effectively isolating it from the list.

After the current node is isolated, it is deleted using delete lTemp, freeing the memory occupied by that node.

The loop then continues to the next iteration.

When there is only one node left in the list (the last node), the else block is executed. In this case, the last node (represented by fRoot) is deleted directly using delete fRoot.

The loop continues to iterate, and this time, it recognizes that there are no more nodes left because fRoot is set to nullptr, and the loop exits.

```
#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.

1.The method starts by initializing a pointer lNode to the first element in the list, which is fRoot.
2. It enters a loop that continues as long as lNode is not nullptr, indicating that there are still elements in the list to search.
3. Inside the loop, it checks if the payload of the current node pointed to by lNode is equal to the target element aElement. The comparison is performed using the **lNode syntax, which dereferences the current node and accesses its payload.
4. If a match is found, it breaks out of the loop, and lNode points to the node containing the element to be removed.
5. If no match is found in the current node, it checks whether there's another node to explore (not the last node). If there is, it updates lNode to point to the next node by using lNode = const_cast<Node*>(&lNode->getNext()).
6. If there are no more nodes to explore, meaning that the target element is not found in the list, lNode is set to nullptr, indicating that the search is unsuccessful.
7. After the loop, it checks whether lNode is still nullptr. If it's not nullptr, it means the element was found and needs to be removed.
8. If the list contains more than one element (i.e., fCount is not 1), it further checks whether the element to remove is the first element in the list, which is indicated by lNode == fRoot. If it is the first element, it updates fRoot to point to the next node, effectively removing the first element from the list.
9. If there's only one element in the list (i.e., fCount is 1), it sets fRoot to nullptr to indicate that the list is now empty.
10. The lNode->isolate() method is called to disconnect the node pointed to by lNode from its neighboring nodes. This isolates the node and prepares it for deletion.
11. Finally, it deletes the isolated node using delete lNode, which frees the memory associated with that node, and decrements fCount to reflect the reduced size of the list.

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;
}
```

- List(): This is the constructor's name, and it matches the class name. It's a constructor for creating objects of the List class.
- fRoot(nullptr): This is an initializer for a member variable called fRoot. In this line, the fRoot member is being set to nullptr. fRoot is typically a pointer that points to the first element in the list. By initializing it to nullptr, you are indicating that this list is empty because there are no elements to point to.
- fCount(0): Similarly, this is initializing another member variable fCount to 0. fCount typically stores the number of elements in the list. By setting it to 0, you are stating that this list has zero elements initially.

- fRoot is a member variable of the class, typically used to point to the first element in the list (or nullptr if the list is empty). nullptr is a special null pointer value in C++ that indicates a pointer does not point to any valid memory location.
- The purpose of this method is to check whether the list is empty. It does so by comparing the value of fRoot to nullptr. If fRoot is equal to nullptr, it means that there are no elements in the list, and the method returns true to indicate that the list is indeed empty.

- fCount is a member variable of the class. It appears to represent the number of elements currently in the list.
- The purpose of this method is to provide the current size of the list, which is the count of elements it contains.

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
```

- { if (isEmpty()) { ... } else { ... } }: This block of code checks whether the list is empty or not by calling the isEmpty method, which is not provided in the code snippet but is expected to return a boolean value indicating whether the list is empty or not.

- fRoot = new Node(aElement);: If the list is empty, this line of code creates a new node of type Node and initializes it with the value aElement. This newly created node becomes the root of the list.

- else { Node* lNode = new Node(aElement); fRoot->push_front(*lNode); fRoot = lNode; }: If the list is not empty, this part of the code creates a new node lNode and initializes it with the value aElement. It then calls the push_front method on the existing root node (fRoot) and passes *lNode as an argument. This means the new node is added to the front of the list by attaching it to the existing root. Finally, the root pointer (fRoot) is updated to point to the newly added node, making it the new root.

- ++fCount;: After adding an element, the fCount member variable is incremented to keep track of the number of elements in the list.

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;
}
```

- { if (isEmpty()) { ... } else { ... } }: This block of code checks whether the list is empty or not by calling the isEmpty method. If the list is empty, it creates a new node at the back. If the list is not empty, it appends a new node to the back of the list.

- fRoot = new Node(aElement);: If the list is empty, this line of code creates a new node of type Node and initializes it with the value aElement. This newly created node becomes the root of the list.

- else { Node* lastNode = const_cast<Node*>(&fRoot->getPrevious()); lastNode->push_back (*new Node(aElement)); }: If the list is not empty, this part of the code retrieves the last node in the list. It does this by accessing the previous node of the current root node (fRoot->getPrevious()), which is a doubly-linked list. Then, it calls the push_back method on the last node and passes *new Node(aElement) as an argument. This means the new node is added to the back of the list by attaching it to the last node.

- ++fCount;: After adding an element, the fCount member variable is incremented to keep track of the number of elements in the list.

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;
}
```

- { if (aIndex >= size()) { throw std::out_of_range("Index out of bounds"); } }: This part of the code checks whether the provided index aIndex is within the valid range of the list. If aIndex is greater than or equal to the size of the list (as determined by the size () method), it means the index is out of bounds, and an std::out_of_range exception is thrown.

- Iterator lIterator = begin();: Here, a local variable lIterator of type Iterator is created and initialized with the result of the begin() method. This iterator is used to traverse the list and access elements.

- for (size_t i = 0; i < aIndex; i++) { ++lIterator; }: This loop iterates through the list by incrementing the lIterator aIndex times. This effectively advances the iterator to the position corresponding to the desired index.

- return *lIterator;: Finally, the element at the specified index is accessed by dereferencing the iterator (*lIterator), and it is returned as a constant reference to type T.

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
```

lIterator:

1. Initialization: Before you can use lIterator to traverse the list, you need to initialize it. In the code you provided, initialization occurs with the statement Iterator lIterator = begin();. begin() is a member function of your List class that returns an iterator pointing to the first element in the list. So, after this initialization, lIterator is set to the beginning of the list.
2. Dereferencing (*lIterator): To access the element that the iterator is currently pointing to, you can use the dereference operator *. For example, *lIterator will give you access to the element at the current position.
This allows you to read or modify the content of the element that lIterator is pointing to.
3. Iterating to the Next Element (++lIterator): To move to the next element in the list, you can use the increment operator ++. For example, ++lIterator advances lIterator to the next element in the list.
After this operation, lIterator is now pointing to the next element, and you can again use *lIterator to access the content of the new element.
You can repeat this operation in a loop or as needed to traverse the list, moving from one element to the next.

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
```

- this copy constructor first initializes the new List object with default values and then copies the contents of the existing aOtherList into the new object. This allows you to create a deep copy of the List while ensuring that the new object is independent of the original one.

1. Initialization: It starts by initializing the data members fRoot and fCount. In this case, both are set to initial values: fRoot is set to nullptr, and fCount is set to 0.

2. Assignment: After the initialization, the copy constructor proceeds to perform the copy operation. It assigns the content of the existing list, aOtherList, to the newly created - - List object: The assignment is done using the copy assignment operator, which you have already implemented for your List class. It ensures that the elements of aOtherList are correctly copied into the new list, so they share the same values.
- The assignment operator handles the allocation of memory for the new list and the copying of elements from aOtherList into the new list. This allows you to reuse the logic for copying elements that you have already implemented in the assignment operator.

1. Self-Assignment Check: It starts with a check to see if the source List, aOtherList, is the same as the current object (this). Self-assignment should be avoided since it doesn't involve copying from one object to another. If self-assignment is detected, the operator simply returns without making any changes.

2. Destructor Call: If aOtherList is not the same as the current object (this), the operator first explicitly calls the destructor for the current object using this->~List(). This is done to release any resources held by the current object. It ensures that any existing elements and memory are properly deallocated before performing the assignment.

3. Copying Elements: If aOtherList is not empty (i.e., its fRoot is not nullptr), the operator then proceeds to copy the elements from aOtherList into the current object. It initializes fRoot and fCount to their default values, which is nullptr for fRoot and 0 for fCount. Then, it iterates over the elements in aOtherList and uses the push_back method to copy each element into the current object.

4. The copy process ensures that the new List has a deep copy of the elements, so it's not sharing the same data as aOtherList. This is important for data separation between the two objects.

5. Return Reference: After the assignment is complete, the operator returns a reference to the current object (*this) to allow for method chaining or further operations.

In summary, this copy assignment operator takes care of releasing any existing resources, creates a deep copy of the elements from the source list (aOtherList), and ensures that the current object becomes an independent copy of the source list. This is a common pattern for implementing copy assignment operators in C++.

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;
  }
```

1. Input Argument: The move constructor takes an rvalue reference to another List object, aOtherList. The use of an rvalue reference (&&) indicates that the constructor is designed to work with temporary or movable objects.

2. Move Assignment: The primary operation performed in the move constructor is the move assignment. It uses std::move(aOtherList) to cast aOtherList to an rvalue. This allows the move constructor to take ownership of the resources held by aOtherList. In other words, it transfers the internal data (in this case, the linked list) from aOtherList to the newly constructed object (the one pointed to by this).

This operation is very efficient because it typically involves changing pointers rather than copying data. The move constructor effectively "steals" the data from the source object, leaving the source object in a valid but unspecified state.

3. Resource Transfer: After the move assignment, the current object (the one pointed to by this) effectively takes over the linked list and the ownership of its elements.

The aOtherList is left in a state where it has no elements and is safe to destroy or be reassigned.

4. Result: This move constructor doesn't allocate new memory for the linked list's elements. Instead, it reassigns pointers to existing elements in the linked list. This is a key characteristic of move semantics: it's about efficiently transferring ownership of resources without unnecessary copying.

```
  // 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
```

- Input Argument: The move assignment operator takes an rvalue reference to another List object, aOtherList. The use of an rvalue reference (&&) indicates that the operator is designed to work with temporary or movable objects.
- Self-Assignment Check: It starts with a self-assignment check using if (&aOtherList != this). Self-assignment would occur if someone tried to assign a List object to itself. In that case, it's not necessary to do anything, and the function returns without further action.
- Destruction: The line this->~List(); is used to destroy the current object. This step ensures that any existing resources held by the current object are properly released. This is an essential part of move assignment because it prepares the object to receive the resources from aOtherList.
- Resource Transfer: If aOtherList is not empty (i.e., it has a non-null fRoot), the move assignment operator transfers ownership of the linked list and its elements from aOtherList to the current object (*this).
It reassigns fRoot to point to the linked list in aOtherList.
It transfers the count of elements (fCount) from aOtherList to the current object.
It sets aOtherList.fRoot to nullptr to make aOtherList effectively empty.
It sets aOtherList.fCount to 0 to indicate that aOtherList now has no elements.
- Result: After the move assignment, the current object (the one pointed to by this) takes over the linked list and the ownership of its elements, while aOtherList is left in a state where it's empty and safe to destroy or be reassigned.

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

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

push_front(T&& aElement): This function is used to add an element to the front (beginning) of the list.

If the list is empty (isEmpty() check), it creates a new Node by moving the provided element, std::move (aElement), and assigns it as the new fRoot. This means the list becomes a single-node list with the provided element.

If the list is not empty, it also creates a new Node by moving aElement and inserts this new node at the front of the existing list. The existing fRoot is then updated to point to this new node, making it the new front of the list. This effectively prepends the element to the list.

Finally, the element count fCount is incremented by one to reflect the addition of a new element.

push_back(T&& aElement): This function adds an element to the back (end) of the list.

If the list is empty (isEmpty() check), it behaves similarly to push_front. It creates a new Node by moving the provided element, std::move(aElement), and assigns it as the fRoot. This makes the list a single-node list with the provided element.

If the list is not empty, it retrieves the last node in the list using const_cast<Node*>(&fRoot->getPrevious()), ensuring that we are working with the last node (since the list is doubly-linked). Then, it creates a new Node by moving aElement and appends this new node to the list by invoking lastNode->push_back. This operation extends the list by attaching the new node to the previous last node.

The element count fCount is incremented to indicate the addition of a new element to the list.