COS30008 Semester 1, 2022 Dr. Markus Lumpe

# 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:

TRAN QUOC DUNG

## Your student id:

103803891

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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:

1

**File: ListPS3.h**

#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() // destructor - frees all nodes

{

while (fRoot != nullptr)

{

if (fRoot != &fRoot->getPrevious() ) // more than one element

{

Node\* lTemp = const\_cast<Node\*>(&fRoot->getPrevious()); // select last

lTemp->isolate(); // remove from list

delete lTemp; // free

}

else

{

delete fRoot; // free last

break; // stop loop

}

}

}

void remove(const T& aElement) // remove first match from list

{

Node\* lNode = fRoot; // start at first

while (lNode != nullptr) // Are there still nodes available?

{

if (\*\*lNode == aElement) // Have we found the node?

{

break; // stop the search

}

if (lNode != &fRoot->getPrevious()) // not reached last

{

lNode = const\_cast<Node\*>(&lNode->getNext()); // go to next

}

else

{

lNode = nullptr; // stop search

}

}

// At this point we have either reached the end or found the node.

if (lNode != nullptr) // We have found the node.

{

if (fCount != 1) // not the last element

{

if (lNode == fRoot)

{

fRoot = const\_cast<Node\*>(&fRoot->getNext()); // make next root

}

}

else

{

fRoot = nullptr; // list becomes empty

}

lNode->isolate(); // isolate node

delete lNode; // release node's memory

fCount--; // decrement count

}

}

////////////////////////////////////////////////////////////////// STARTING PS3

// P1

List() : fRoot(nullptr), fCount(0) {} // default constructor

bool empty() const { return fCount == 0; } // Is list empty?

size\_t size() const { return fCount; } // list size

void push\_front(const T& aElement) {

if (empty())

{

fRoot = new Node(aElement);

}

else

{

Node\* lNode = new Node(aElement);

fRoot->push\_front(\*lNode);

fRoot = lNode;

}

++fCount;

}

// adds aElement at front

Iterator begin() const {

return Iterator(fRoot).begin();

} // return a forward iterator

Iterator end() const {

return Iterator(fRoot).end();

} // return a forward end iterator

Iterator rbegin() const

{

return Iterator(fRoot).rbegin();

} // return a backwards iterator

Iterator rend() const

{

return Iterator(fRoot).rend();

} // return a backwards end iterator

// P2

void push\_back(const T& aElement) {

if (empty())

{

fRoot = new Node(aElement);

}

else

{

Node\* lastNode = const\_cast<Node\*>(&fRoot->getPrevious());

lastNode->push\_back(\*new Node(aElement));

}

++fCount;

} // adds aElement at back

// P3

const T& operator[](size\_t aIndex) const

{

if (aIndex > size() - 1) throw std::out\_of\_range("Index out of bounds");

Iterator lIterator = Iterator(fRoot).begin();

for (size\_t i = 0; i < aIndex; i++) ++lIterator;

return \*lIterator;

} // list indexer

// P4

List(const List& aOtherList) : fRoot(nullptr), fCount(0)

{

\*this = aOtherList;

} // copy constructor

List& operator=(const List& aOtherList) {

if (&aOtherList != this)

{

this->~List();

if (aOtherList.fRoot == nullptr)

{

fRoot = nullptr;

}

else

{

fRoot = nullptr;

fCount = 0;

for (auto& payload : aOtherList)

{

push\_back(payload);

}

}

}

return \*this;

} // assignment operator

// P5

List(List&& aOtherList) : fRoot(nullptr), fCount(0)

{

\*this = std::move(aOtherList);

} // move constructor

List& operator=(List&& aOtherList) {

if (&aOtherList != this)

{

this->~List();

if (aOtherList.fRoot == nullptr)

{

fRoot = nullptr;

}

else

{

fRoot = aOtherList.fRoot;

fCount = aOtherList.fCount;

aOtherList.fRoot = nullptr;

aOtherList.fCount = 0;

}

}

return \*this;

}// move assignment operator

void push\_front(T&& aElement)

{

if (empty())

{

fRoot = new Node(std::move(aElement));

}

else

{

Node\* lNode = new Node(std::move(aElement));

fRoot->push\_front(\*lNode);

fRoot = lNode;

}

++fCount;

}// move push\_front

void push\_back(T&& aElement)

{

if (empty())

{

fRoot = new Node(std::move(aElement));

}

else

{

Node\* lastNode = const\_cast<Node\*>(&fRoot->getPrevious());

lastNode->push\_back(\*new Node(std::move(aElement)));

}

++fCount;

} // move push\_back

};

**File: DoublyLinkedList.h**

#pragma once

template<typename T>

class DoublyLinkedList

{

private:

T fPayload;

DoublyLinkedList\* fNext;

DoublyLinkedList\* fPrevious;

public:

// l-value constructor

explicit DoublyLinkedList(const T& aPayload) :

fPayload(aPayload),

fNext(this),

fPrevious(this)

{}

// r-value constructor

explicit DoublyLinkedList(T&& aPayload) :

fPayload(std::move(aPayload)),

fNext(this),

fPrevious(this)

{}

DoublyLinkedList& push\_front(DoublyLinkedList& aNode)

{

aNode.fNext = this; // make this the forward pointer of aNode

aNode.fPrevious = fPrevious; // make this's backward pointer aNode's

fPrevious->fNext = &aNode; // tie back to Node

fPrevious = &aNode; // this' backward pointer becomes aNode

return aNode; // last node inserted

}

DoublyLinkedList& push\_back(DoublyLinkedList& aNode)

{

aNode.fPrevious = this; // make this the backwards pointer of aNode

aNode.fNext = fNext; // make this's forward pointer aNode's

fNext->fPrevious = &aNode; // tie back to Node

fNext = &aNode; // this' forward pointer becomes aNode

return aNode; // last node inserted

}

void isolate()

{

fPrevious->fNext = fNext; // unlink previous

fNext->fPrevious = fPrevious; // unlink next

fPrevious = this; // isolate this node

fNext = this;

}

void swap(DoublyLinkedList& aNode)

{

std::swap(fPayload, aNode.fPayload); // exchange list elements

}

const T& operator\*() const // dereference operator

{

return getPayload();

}

const T& getPayload() const

{

return fPayload;

}

const DoublyLinkedList& getNext() const

{

return \*fNext;

}

const DoublyLinkedList& getPrevious() const

{

return \*fPrevious;

}

};

**File: DoublyLinkedListIterator.h**

#pragma once

#include "DoublyLinkedList.h"

template<typename T>

class DoublyLinkedListIterator

{

private:

enum class States { BEFORE, DATA, AFTER };

using Node = DoublyLinkedList<T>;

const Node\* fRoot;

States fState;

const Node\* fCurrent;

public:

using Iterator = DoublyLinkedListIterator<T>;

DoublyLinkedListIterator(const Node\* aRoot)

{

fRoot = aRoot;

fCurrent = fRoot;

if (fCurrent != nullptr)

{

fState = States::DATA;

}

else

{

// empty doubly linked list of nodes

fState = States::AFTER;

}

}

const T& operator\*() const // dereference

{

return \*\*fCurrent;

}

Iterator& operator++() // prefix increment

{

switch (fState)

{

case States::BEFORE:

fCurrent = fRoot; // set to first element

if (fCurrent == nullptr)

{

fState = States::AFTER;

}

else

{

fState = States::DATA;

}

break;

case States::DATA:

// Is current previous of root (last element forward)?

// Current cannot be nullptr as we are in state DATA.

if (fCurrent == &fRoot->getPrevious())

{

// Yes, we are done

fCurrent = nullptr;

fState = States::AFTER;

}

else

{

// No, we can advance

fCurrent = &fCurrent->getNext();

}

break;

default:

break;

}

return \*this;

}

Iterator operator++(int) // postfix increment

{

Iterator temp = \*this;

++(\*this);

return temp;

}

Iterator& operator--() // prefix decrement

{

switch (fState)

{

case States::AFTER:

fCurrent = fRoot;

if (fCurrent == nullptr)

{

fState = States::BEFORE;

}

else

{

fCurrent = &fCurrent->getPrevious(); // set to last element

fState = States::DATA;

}

break;

case States::DATA:

// Is current root (last element backwards)?

// Current cannot be nullptr as we are in state DATA.

if (fCurrent == fRoot)

{

// Yes, we are done

fCurrent = nullptr;

fState = States::BEFORE;

}

else

{

// No, we can advance

fCurrent = &fCurrent->getPrevious();

}

break;

default:

break;

}

return \*this;

}

Iterator operator--(int) // postfix decrement

{

Iterator temp = \*this;

--(\*this);

return temp;

}

bool operator==(const Iterator& aOtherIter) const

{

return

fRoot == aOtherIter.fRoot &&

fCurrent == aOtherIter.fCurrent &&

fState == aOtherIter.fState;

}

bool operator!=(const Iterator& aOtherIter) const

{

return !(\*this == aOtherIter);

}

Iterator begin() const

{

return ++(rend());

}

Iterator end() const

{

Iterator iter = \*this;

iter.fCurrent = nullptr;

iter.fState = States::AFTER;

return iter;

}

Iterator rbegin() const

{

return --(end());

}

Iterator rend() const

{

Iterator iter = \*this;

iter.fCurrent = nullptr;

iter.fState = States::BEFORE;

return iter;

}

};

**File: Main.cpp**

#include <iostream>

#include <string>

#include <stdexcept>

#include "ListPS3.h"

using namespace std;

#define P0

#define P1

#define P2

#define P3

#define P4

#define P5

#ifdef P0

void testP0()

{

cout << "Test basic setup:" << endl;

List<string> lList;

lList.remove("P0");

lList.remove(string("P0"));

cout << "Complete" << endl;

}

#endif

#ifdef P1

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;

}

#endif

#ifdef P2

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;

}

#endif

#ifdef P3

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;

}

#endif

#ifdef P4

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;

}

#endif

#ifdef P5

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(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;

}

#endif

int main()

{

#ifdef P0

testP0();

cout << "\n" << endl;

#endif

#ifdef P1

testP1();

cout << "\n" << endl;

#endif

#ifdef P2

testP2();

cout << "\n" << endl;

#endif

#ifdef P3

testP3();

cout << "\n" << endl;

#endif

#ifdef P4

testP4();

cout << "\n" << endl;

#endif

#ifdef P5

testP5();

#endif

return 0;

}