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: Sunday, November 05, 2023, 23:59 (VN Time)
Lecturer: Dr. Van Dai PHAM

Your name: Le Minh Kha **Your student id:** 104179506

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Thursday 10:00 (Innovation Lab)
							X

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: _____

```cpp
#pragma once
#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"
#include <stdexcept>

template<typename T>
class List
{
private:
    using Node = DoublyLinkedList<T>;
    Node* fRoot;
    Node* fTail;
    size_t fCount;

public:
    using Iterator = DoublyLinkedListIterator<T>;

    // set up
    List() : fRoot(nullptr), fTail(nullptr), fCount(0) {};

    List(const List& aOtherList) : fRoot(nullptr), fTail(nullptr), fCount(0)// copy
constructor
    {
        *this = aOtherList;
    };

    ~List()
    {
        while (fRoot != nullptr)
        {
            if (fRoot != &fRoot->getPrevious())
            {
                Node* lTemp = const_cast<Node*>(&fRoot->getPrevious());
                lTemp->isolate();
                delete lTemp;
            }
            else
            {
                delete fRoot;
                break;
            }
        }
    }

    bool isEmpty() const { return fCount == 0; }
    size_t size() const { return fCount; }

    //iterators
    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();}

    // handle methods
```

```
    void remove(const T& aElement)
    {
        Node* current_node = fRoot;  // Start at the root of the list.

        // Iterate through all nodes in the list.
        while (current_node != nullptr) {
            // Check if the current node matches the target element.
            if (*current_node == aElement)
            {
                break;
            }

            // Move to the next node if we haven't reached the end of the list.
            if (current_node != &fRoot->getPrevious())
            {
                current_node = const_cast<Node*>(&current_node->getNext());
            }
            else
            {
                current_node = nullptr;  // Reached the last node; exit the loop.
            }
        }

        // If a matching node was found during the loop:
        if (current_node != nullptr)
            {
            // Check if it's not the last element.
            if (fCount != 1)
            {
                // If the matching node is the root, update the root.
                if (current_node == fRoot)
                {
                    fRoot = const_cast<Node*>(&fRoot->getNext());  // Make the next
node the new root.
                }
            }
            else
            {
             // The list has only one element; set the root
                fRoot = nullptr;  to null.
            }

            // isolate & terminate
            current_node->isolate();
            delete current_node;
            fCount--;
        }
    }



    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;
    }


    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;
    }


///// Move semantic and movement operators///////

    // an example: int x = 10;
    //              |       |
    //            l-val  r-value (usually, not always)


    // l-value references (T&)
    // when dealing with l-value, we ususally refer to named objects
    // when you want to store data from var X to Y, a temp copy of X will be make
and give it to Y
    // this mean invoking the following methods when you want to make copies
    void push_front(const T& aElement)
    {
        //copy data of aElement to the new node
        Node* new_node = new Node(aElement);
        if (isEmpty())
        {
            fRoot = fTail = new_node;
        }
        else
        {
            fRoot->push_front(*new_node);
            // the first node is at index 0, thus should become root
            fRoot = new_node;
        }
        fCount++;
    }

    void push_back(const T& aElement)
    {
```

```
        Node* new_node = new Node(aElement);
        if (isEmpty())
        {
            fRoot = fTail = new_node;
        }
        else
        {
            fTail->push_back(*new_node);
            fTail = new_node;
        }
        fCount++;
    }


    //  r-value reference ( T&& )
    // it is usually refer to as the temporary value
    // r-value is great when you actually want to move the data without the need of
making copies
    // the standard lib std::move help us to achive this
    void push_front(T&& aElement)
    {
        // move the data of aElement to the new node
        Node* new_node = new Node(std::move(aElement));
        if (isEmpty())
        {
            fRoot = fTail = new_node;
        }
        else
        {
            new_node->fNext = fRoot;
            fRoot->fPrevious = new_node;
            fRoot = new_node;
        }
        fCount++;
    };

    void push_back(T&& aElement)
    {
        Node* new_node = new Node(std::move(aElement));
        if (isEmpty())
        {
            fRoot = fTail = new_node;
        }
        else
        {
            fTail->fNext = new_node;
            new_node->fPrevious = fTail;
            fTail = new_node;
        }
        fCount++;
    };



    // l-value operations
    List(List&& aOtherList)
    {
        fRoot = aOtherList.fRoot;
```

```
        fTail = aOtherList.fTail;
        fCount = aOtherList.fCount;

        aOtherList.fRoot = nullptr;
        aOtherList.fTail = nullptr;
        aOtherList.fCount = 0;
    };

    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;
    }

};
```

Output:

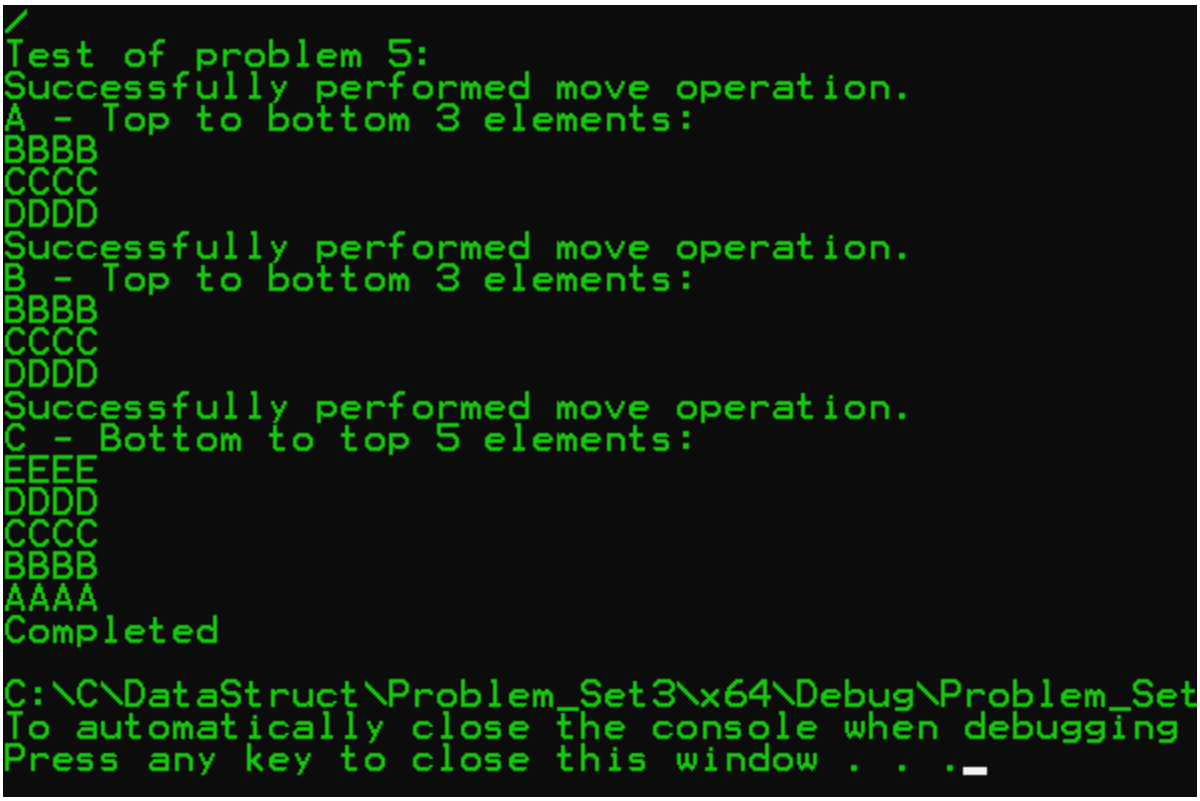
Microsoft Visual Studio Debug Console

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

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

/
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

/
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
```


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
C:\C\DataStruct\Problem_Set3\x64\Debug\Problem_Set
To automatically close the console when debugging
Press any key to close this window . . .