UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS110 Assignment 3

Linked Lists

Due date: 06 November 2016 (23h59)

Total marks: **90**

1 General instructions

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline, as **no extension** will be granted.
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence of certain functions or classes).
- Read the entire assignment thoroughly before you start coding.
- **To ensure that you did not plagiarize, your code will be inspected with the help of dedicated software.**
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `http://www.ais.up.ac.za/plagiarism/index.htm`.

2 Overview

For this assignment, you will implement a variety of linked lists. You will learn about the differences and similarities between these linked lists, and your final program will allow users to make their own choices of which implementation to use for which type of linked list at run time.

3 Background

3.1 Linked Lists

A linked list is a type of linear data structure that keeps data in a similar way to generic arrays. However, linked lists make use of a series of pointers that links memory blocks together. Because of this property linked lists use memory more efficiently. Unlike arrays, linked lists do not have an explicit index for blocks of memory. However, functions can be implemented to imitate this property. Linked lists are heavily reliant on the links between the chain of memory blocks which makes a link list's structure very volatile. If the links were not updated correctly, any blocks of memory after that link stands a chance of being lost or it may even corrupt the entire linked list. The blocks of memory that are chained together by pointers are called nodes. The collection of these nodes forms a linked list.

There are several methods in which to implement a linked list, all with its own unique qualities and structures that come with certain advantages and disadvantages. To name a few, there are Singular linked lists, Circular linked lists, Doubly linked lists, Stacks, Queues and many other data structures that are not necessarily linear in structure.

3.2 Implementing Linked lists

Linked lists may be implemented in a number of ways. For example, it can be implemented as singly linked or doubly linked lists. As discussed previously, each type of implementation comes with its own advantages, and which implementation is most suited depends on the situation.

One can easily cater for every type of implementation that one may possibly need, and do it in such a way that it is easy and convenient to add additional implementations at a later stage should it be required, without changing any code that you have already written. For example, you needn't hard code two linked lists classes where one is implemented in terms of circular structure and one a doubly structure. Instead, you can code one linked list class and instantiate it with a circular structure or doubly structure implementation (provided these conform to the same interface). It might seem redundant to code three classes (linked list, circular linked list, and doubly linked list) instead of just two (circular linked list and doubly linked list). Inheritance is a powerful way to structure your classes if you just want to add additional features. The key to making use of inheritance is to find the similarities and differences between data structures.

3.3 Previous Knowledge

To tackle this assignment you will need knowledge of the following concepts:

- Pointers (Chapter 9)
- Object Orientated Programming (OOP) (Chapter 13 and 14)
- Templates, Template classes (Chapter 16)
- Exception handling by making use of the try and catch mechanism (Chapter 16)
- Inheritance with overriding functions (virtual functions) and protected members (Chapter 15)
- Operator overloading (Chapter 14)
- LinkedLists (Chapter 17)

These concepts are explained with examples within the prescribed textbook.

Additionally, some tips have also been provided at the end of this assignment specification. **Please read them**. It may prevent you from wasting time. With that being said, **any queries that can be answered by these tips may be ignored or given a generic answer** that refers you back to this assignment specification.

4 Your Task

Download the archive `assignment3code.tar.gz` from the course website. This archive contains a number of classes which you will have to implement. The comments in the code describe how each function should be implemented. You are **not allowed** to modify the header files - these will be overwritten by Fitchfork when you submit code for marking. You are required to implement a `LinkedList` class. You are then required to implement 2 other data structures for which each of these data structures inherit from the `LinkedList` class. This breaks down to a total of 3 Tasks namely:

- Task1: LinkedList
- Task2: CircularList
- Task3: DoubleList

You are provided with the following classes:

4.1 Node and dNode

Node is a singly linked node class to be used for the linked list implementations. dNode is a variant of the Node class that is double linked, which is to be used for the DoubleList. The Node class and the dNode class has already been implemented for you.

4.2 Task 1: LinkedList (30 marks)

This class is the parent class to the other linked lists for which implementations need to conform to. `CircularList` and `DoubleList` inherit from this class. Write the implementation for a `LinkedList`. A brief description on how each function is expected to be implemented has been provided below. The detailed description is provided in the Header file as comments above each function's prototype. A Header file has been provided with the following functions:

- Constructor: this needs to initialize all relevant member variables.
- Copy Constructor: Copies all the data from the "other" `LinkedList` and stores them within this object in the same order.
- Destructor: releases all dynamically allocated memory to prevent memory leaks.
- assignment operator=: Deletes all data that this object is currently holding and copies all the data from the "other" `LinkedList` and stores them within this object in the same order.
- print: returns an ostream that can be used to print out the data in the correct order and format.
- ostream opperator«: prints out the data in the correct order.
- clone: creates and returns a deep copy of the `LinkedList`.
- get: returns the data from a specified index.
- index operator[]: returns the data from a specified index.
- size: returns the amount of nodes contained in the `LinkedList`.
- isEmpty: returns true or false of whether the `LinkedList` is empty or not.
- getLeader: returns a pointer to the head of the `LinkedList`.
- insert: inserts a value into the given index of the `LinkedList`.
- remove: deletes an entry from the `LinkedList` and returns the value of the given index.
- clear: releases any dynamically allocated memory stored in `LinkedList`.

- addition operator+: appends 2 `LinkedLists` together.

The expected output for the given `linkTest.cpp` should be:

```
-----------insert-----------
[a]
-----copy constructor---------------
[a]
-----------get-----------
ERROR: invalid index
-----------operator[]-----------
ERROR: invalid index
-----------remove-----------
ERROR: invalid index
-----------operator=-----------
[a]
-----------operator+-----------
[a,a]
```

NOTE: The test in the given `linkTest.cpp` is not a sufficient test and is not a good indication of a working `LinkedList`. This was just to give you a starting point.

See `LinkedList.h` for details. Submission details are specified in its own section below.

4.3 Task 2: CircularList (30 marks)

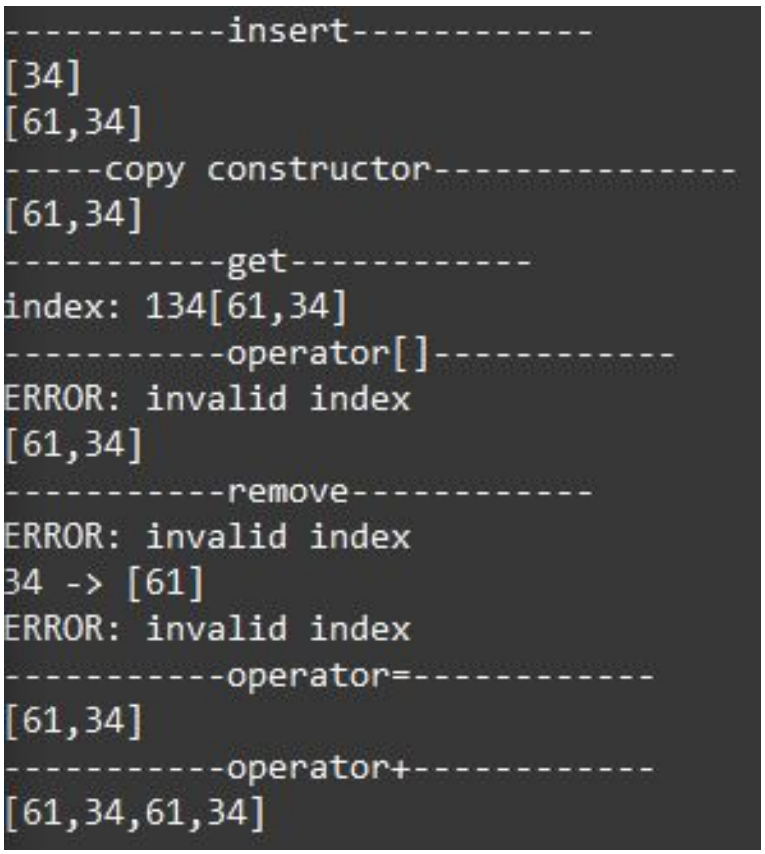
Write the implementation for a circular linked list. A brief description on how each function is expected to be implemented has been provided below. The detailed description is provided in the Header file as comments above each function's prototype. A Header file has been provided with the following functions:

- Constructor: this needs to initialize all relevant member variables.
- Copy Constructor: Creates a copy of all the data from the "other" `CircularList` and stores them within this object in the same order.
- Destructor: releases all dynamically allocated memory to prevent memory leaks
- assignment operator=: Deletes all data that this object is currently holding and copies all the data from the "other" `CircularList` and stores them within this object in the same order.

- print: returns an ostream that can be used to print out the data in the correct order and format.
- ostream opperator«: returns an ostream that can be used to print out the data in the correct order and format.
- clone: creates and returns a deep copy of the `CircularList`.
- get: returns the data from a specified index.
- index operator[]: returns the data from a specified index.
- size: returns the amount of nodes contained in the `circularList`.
- isEmpty: returns true or false for whether the `CircularList` is empty or not.
- getLeader: returns a pointer to the head of the `CircularList`.
- insert: inserts a value into the given index of the `CircularList`.
- remove: deletes an entry from the `CircularList` and returns the value of the given index.
- clear: releases any dynamically allocated memory stored in `CircularList`.
- addition operator+: appends 2 `CircularLists` together.

Be careful of infinite loop cycles for circular linkedlists, remember that the tail node points to the head. Some implementations use a tail pointer instead of a head pointer. The implementation for this task uses a head pointer because it inherits from the `LinkedList` class. There is no explicit tail pointer. The last node in the list is the tail node. Adapt your logic accordingly.

The expected output for the given `linkTest.cpp` should be:



NOTE: The test in the given `ccListTest.cpp` is not a sufficient test and is not a good indication of a working `CircularList`. This was just to give you a starting point.

See `cirularList.h` for details. Submission details are specified in its own section below.

4.4 Task 3: DoubleList (30 marks)

Write the implementation for a doubly `LinkedList`. A brief description on how each function is expected to be implemented has been provided below. The detailed description is provided in the Header file as comments above each function's prototype. A Header file has been provided with the following functions:

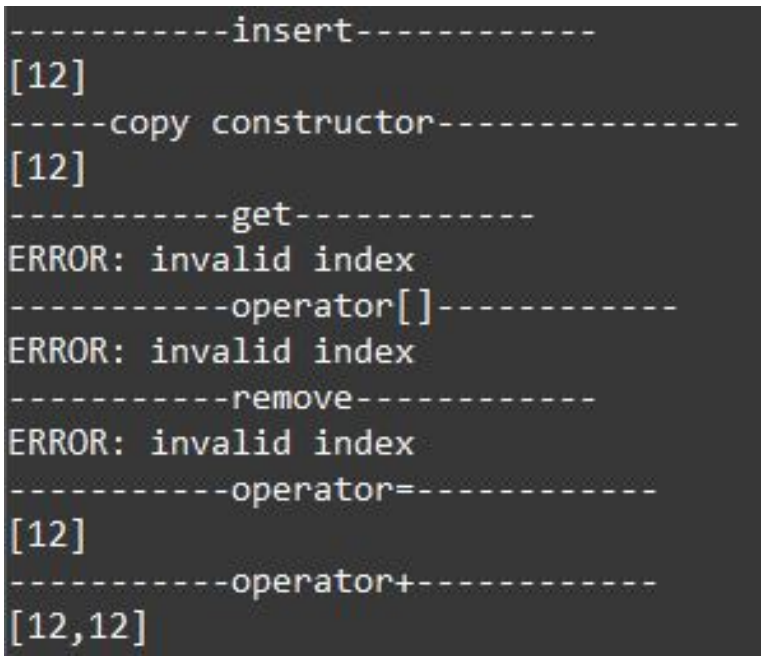
- Constructor: this needs to initialize all relevant member variables.
- Copy Constructor: Creates a copy of all the data from the "other" `DoubleList` and stores them within this object in the same order.
- Destructor: releases all dynamically allocated memory to prevent memory leaks.
- assignment operator=: Deletes all data that this object is currently holding and copies all the data from the "other" `DoubleList` and stores them within

this object in the same order.

- print: returns an ostream that can be used to print out the data in the correct order and format.
- ostream opperator«: returns an ostream that can be used to print out the data in the correct order and format.
- clone: creates and returns a deep copy of the `DoubleList`.
- get: returns the data from a specified index.
- index operator[]: returns the data from a specified index.
- size: returns the amount of nodes contained in the `DoubleList`.
- isEmpty: returns true or false for whether the `DoubleList` is empty or not.
- getHead: returns a pointer to the head of the `DoubleList`.
- insert: inserts a value into the given index of the `DoubleList`.
- remove: deletes an entry from the `DoubleList` and returns the value of the given index.
- clear: releases any dynamically allocated memory stored in `DoubleList`.
- addition operator+: appends 2 doubleLists together.

Be careful of dangling pointers for doubleLists linkedlists, remember that each node has 2 pointers (`prev`, and `next`). The implementation for this task uses a **`dhead`** pointer because the original head pointer inherited from `LinkedList` is a Node pointer. Nodes only have 1 pointer (`next`). `dhead` is a **`dNode`** pointer which has 2 pointers (`prev`, and `next`). You will notice that this class uses a `getHead` function instead of a `getLeader`. This class still has the `getLeader` function inherited from `LinkedList`, think carefully before deciding which function to use in your implementations. Adapt your logic accordingly.

The expected output for the given `dListTest.cpp` should be:



NOTE: The test in the given `dListTest.cpp` is not a sufficient test and is not a good indication of a working `DoubleList`. This was just to give you a starting point.

See `doubleLists.h` for details. Submission details are specified in its own section below.

5 Implementation Details

You are not allowed to change the header files. You will notice that the inheritance has already been implemented for you. Think carefully on how you can take full advantage of the inheritance relationship. Reuse functions to help you complete other functions's implementations where necessary.

6 Submission Details

For each task, you are **only** required to submit your **source file** (`.cpp`) that contains your implementation of the class for each task. This source file (`.cpp`) must be submitted in a tarball (compressed tar.gz file). Makefiles and Header files do not need to be submitted but you will need them when compiling and testing your code.

7 Additional Tips

7.1 Template classes

You will notice that the source files (`.cpp`) are included in the provided header files. This is one way to make linking easier with templates. **Your source files must not include the headers** - remember that template classes are only compiled for concrete types. To test your template classes, all you need to include in your main is the corresponding header.

Makefiles: template classes cannot be used to create object files before hand in the makefile. The object file for the main program can still be created but you will only include the necessary Header files to this object file. You have been given the Makefile for the first task to be used as an example.

Inheritance and protected members from a parent class: because template classes cannot be used to create object files before hand, members inherited from a parent class will not be recognised on compiling code. To fix this problem, the child class needs to make use of "this->" when referring to inherited members.

7.2 "Keep calm and ..." Segmentation fault

The infamously vague runtime error that steals the hope of a student with a successfully compiled program. You will run into a couple of these, I guarantee it. `cout` some strings around the suspected pieces of code, this will help you track it down. Common places that may cause a segmentation fault:

- a while loop with an incorrect condition to detect the end of the `LinkedList`
- a next/prev pointer that was not updated correctly during an insert/remove
- a next/prev pointer that was not updated in the correct order during an insert/remove
- referring to what a pointer is pointing to after it has been deleted
- updating a pointer of any sorts where updates were not needed

7.3 Testing and debugging

Make sure you test for all possible scenarios. The test files and expected outputs that are given to you are only good for minimal testing. Testing will not be limited to what is given and you are encouraged to make changes to these test files as you see fit. Use what has been given to you and expand the testing.

You are only done testing when your friends have tried vigorously and failed in breaking your code.

7.4 Draw!

Do not be afraid to embrace the inner artist. Make those crazy drawings of how each node points to another. You will be surprised how well this works as a way to work through your code step by step to find logical errors. Drawing a class diagram to depict all of the classes and their relationships also helps.

7.5 Tasks are independent

Although `CircularList` and `DoubleList` inherits from `LinkedList`, you do not need to submit your code for the `LinkedList` in the last 2 tasks. This was done to not have you discouraged to pursue marks in the later tasks.

Good luck!

"*May the Father of Understanding Guide Us*" - The Templar's Oath (Assassin's Creed)

"*May the Force be with you*" - Star Wars

"*... and may the odds be ever in your favor*" - Hunger Games

`The End`