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

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program Design: Introduction

Practical 9 Specifications

Release Date: 21-10-2024 at 06:00

Due Date: 27-10-2024 at 23:59

Total Marks: 70

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually, no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- 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/absence of certain functions or classes).
- Failure of your program to successfully exit will result in a mark of 0.
- 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 `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, the usage of c++11 or additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use c++98**
- All functions should be implemented in the corresponding `cpp` file. No inline implementation in the header file apart from the provided functions.
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.
- Note, UML notation for variable and function signatures are used in this assignment.
- Please note that there is a late deadline which is 1 hour after the initial deadline, but you will lose 20% of your mark.

2 Overview

Doubly linked lists are data structures which contains nodes, where each node has a previous and next pointer. This is a variation of a singly linked list that can be traversed in both directions. Thus, doubly linked lists have the same advantages as singly linked list, like being able to easily resize the list, but also the disadvantages like being inefficient to access a certain index in the list. The doubly linked list can also use templates to make it more generic, such that any data type can be stored in the list.

3 Your Task:

You are required to implement the following class diagram illustrated in Figure 1. Pay close attention to the function signatures as the `h` files will be overwritten, thus failure to comply with the UML, will result in a mark of 0.

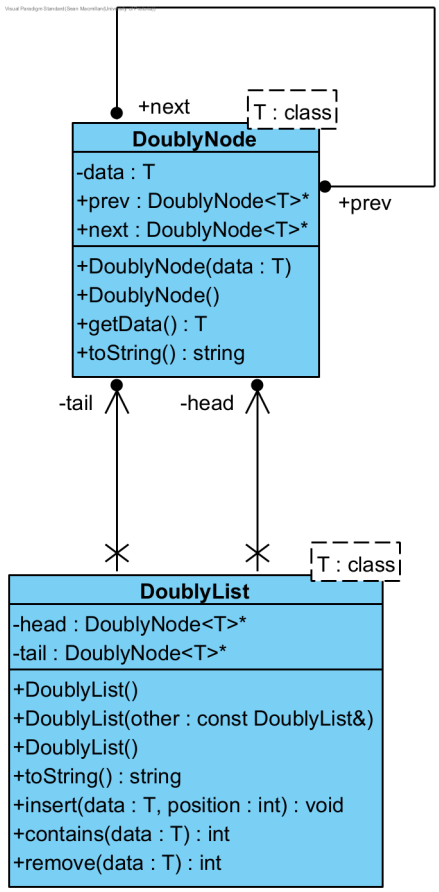

Figure 1: Class diagrams

Note, the importance of the arrows between the classes are not important for COS 110 and will be expanded on in COS 214.

3.1 Important Notice

This practical uses templates, and there are multiple ways templates can be set up and compiled. Please use the following method, as that is how it will be set up on FitchFork.

- Make sure both your header and cpp files have include guards.
- In the header file, include the cpp file at the bottom.
- In the cpp file, include the header file at the top.
- In main.cpp, include the header files.
- Don't compile the template cpp files.

3.2 DoublyNode

This class will be used to store the nodes for the doubly linked list. **This class has been given to you. Don't make any changes, as these files will be replaced on FitchFork.**

3.2.1 Members

- -data: T
  - This is the data of the node.
- +prev: DoubleNode<T>*
  - This is the previous pointer.
- +next: DoubleNext<T>*
  - This is the next pointer.

3.2.2 Functions

- +DoublyNode(data: T)
  - This is the constructor, it sets the data to the passed-in parameter, and the two pointers to NULL.
- +~DoublyNode()
  - This is the destructor.
- +getData(): T
  - This returns the data.
- +toString(): std::String
  - This returns a string representation of the node.

3.3 DoublyList

This class will be used to represent a doubly linked list.

3.3.1 Members

- -head: DoublyNode<T>*
  - This is the head of the doubly linked list.
- -tail: DoublyNode<T>*
  - This is the tail of the doubly linked list.

3.3.2 Functions

- +DoublyList()
  - This is the constructor. This should create an empty list.
- +DoublyList(other: const DoublyList<T>&)
  - This is the copy constructor. Make sure you create a proper **deep** copy of the list.
- +~DoublyList()
  - This is the destructor. All dynamic memory should be freed.
- +toString(): std::String
  - Return a string representation of the list. Use the `toString` function of the DoublyNode class to get the string representation of the nodes.
  - Between every node you should add an arrow, i.e. ->. There should **not** be an arrow after the last node.
- +insert(data: T, position: int): void
  - If the passed-in position is 0, then don't add the node.
  - If the position is positive, then you should add the node from the start of the list. Example:
    - If the position is 1, then the newNode should become the first node in the list.
    - If the position is 3, then the newNode should become the third node in the list.
  - If the position is positive, but too large to fit into the list, then add the node at the end of the list.
  - If the position is negative, then you should add the node from the back of the list. Example:
    - If the position is -1, then the newNode should become the last node in the list.
    - If the position is -3, then the newNode should become the third last node in the list.
  - If the position is negative, but too small to fit into the list, then add the node at the start of the list.
- +contains(data: T): int
  - Return the number of times the passed-in parameter appears in the list.
- +remove(data: T): int
  - This should remove all occurrences of the passed-in parameter.
  - Return the number of nodes removed.
  - Remember to free the memory of the deleted nodes.

4 Memory Management

As memory management is a core part of COS110 and C++, each task on FitchFork will allocate approximately 10% of the marks to memory management. The following command is used:

```
valgrind --leak-check=full ./main
```

Please ensure, at all times, that your code *correctly* de-allocates *all* the memory that was allocated.

5 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the `main.cpp` file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [1] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j ${files}
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

We will scale this ration based on class size.

The mark you will receive for the testing coverage task is determined using Table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the functions stipulated in this specification will be considered to determine your testing mark. Remember that your main will be testing the Instructor Provided code and as such, it can

[1] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

only be assumed that the functions outlined in this specification are defined and implemented.

As you will be receiving marks for your testing main, we will also be doing plagiarism checks on your testing main.

6 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on FitchFork.
- You may only use **c++98**.
- You may only utilize the specified libraries. Failure to do so will result in compilation errors on FitchFork.
- You may only use the following libraries:
  - sstream
  - string

7 Upload Checklist

The following c++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- `DoublyList.cpp`
- `main.cpp`
- Any textfiles used by your `main.cpp`
- `testingFramework.h` and `testingFramework.cpp` if you used these files.

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

8 Submission

You need to submit your source files on the FitchFork website (`https://ff.cs.up.ac.za/`). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ *.cpp -o main
```

and run with the following command:

```
./main
```

Remember your `h` file will be overwritten, so ensure you do not alter the provided `h` files.
You have 10 submissions and your best mark will be your final mark. Upload your archive to the Practical 9 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**

9 Accessibility

Figure 1 contains the UML diagram for this practical. It contains two classes, namely DoublyNode and DoublyList. Both of these classes are template classes of the same generic type T. There is one aggregration relationship namely DoublyList class uses the DoublyNode class.