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

## Assignment 3:
## Linked Lists, Stacks, Queues, Templates

Release date: 23-10-2023 at 06:00
Due date: 10-11-2023 at 23:59
Total Marks: 421

# 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.**
- The following libraries are allowed("<string>", "<cstddef>", "<sstream>", "<iostream>").
- 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 offense. 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.
- 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 submitted will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use c++98**
- We will be overriding your .h files, therefore do not add anything extra to them that is not in the spec

- In your student files you will find the H files supplied, do not modify these when you are programming as we will still be overriding them when you upload to FF
- If you would like to use *using namespace std*, please ensure it appears in your .cpp that you upload.

# 2 Plagiarism

The Department of Computer Science considers plagiarism as a serious offense. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding.** Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

# 3 Outcomes

The goal of this Assignment is to gain experience working with linked lists, stacks, queues, and templates.

# 4 Introduction

Before you begin, here is a good visualizer:

- `https://visualgo.net/en/list`

Note that we are using circular linked lists, however, it can help with some operations.

**NB:** In the past there have been people who do not actually implement the linked list structure. They just keep an array loaded and use that for all operations instead of implementing a linked list. We will be checking for this through FitchFork and other methods. This means for functions like insert, you cannot simply move data around, as we will be checking the node references. The only function that you are allowed to simply swap the data is *swap*().

Implement the UML diagram and functions as described on the following pages.

# 5 Class Diagram

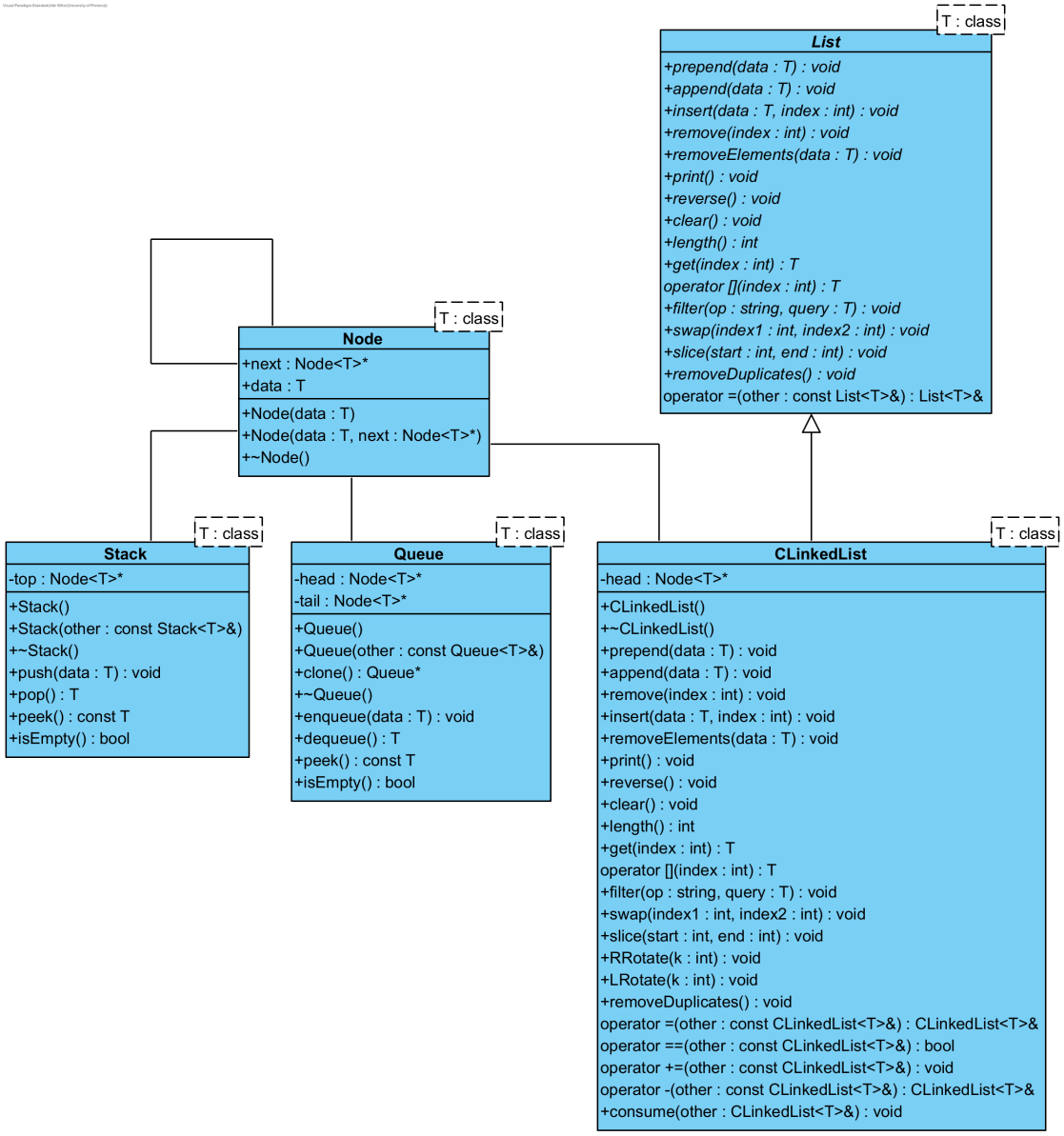


Figure 1: Class diagrams

# 6 List/ADT Classes

## 6.1 List<T>

An **ABSTRACT** List class, it contains no member variable variables. It contains the following **PURE VIRTUAL** functions:

- prepend(data: T): void
- append(data: T): void
- insert(data: T, index: int): void
- removeElements(data: T): void
- print() const: void
- reverse(): void

- clear(): void
- length() const: void
- get(index: int) const: T
- operator[](index: int) const: T
- filter(op: std::string,query: T): void
- swap(index1: int, index2: int): void
- slice(start: int, end: int)
- removeDuplicates()

This class also contains 1 function that is **NOT** virtual and should implemented in *List.cpp*:

- operator=(other: const List<T>&): List<T>&
  - A standard operator=
  - You must use the *List* class functions to achieve the copying of the list.

## 6.2 Node<T>

You will notice that the Node class has been provided and implemented in the .h file, you do not need to do this yourself and you **must** use this class. For simplicity, please see the members and functions that are available to you. Something to mention is that in this class everything is *public*.

- next: Node<T>
  - A pointer to the next node, allowing one to create a list.
- data: T
  - The data of that node.
- Node(data: T)
  - Simple constructor with only data being passed in.
- Node(data: T, next: Node<T>)
  - Constructor with data and the next node being passed in.
- ~Node
  - Simple destructor.

**A note regarding dynamic variables with Node:** You will notice in the destructor for Node we do not deallocate or delete the data variable. This is for 2 reasons. Firstly we do not know for sure that the node is holding a dynamic object. Secondly, the node acts like a holder for the class so deleting it here would mean we can no longer use it which in turn defeats the purpose of using it in a stack/queue.

When it comes to deallocating memory for Stack, Queue and CLinkedList, if the type is a dynamic object then you cannot rely on the destructor to deallocate that memory and you must do it from the calling class, such as the main.

## 6.3 CLinkedList<T> : public List<T>

**NB - THIS IS A CIRCULAR LINKED LIST**. Ensure your code works for this circular nature. When we test on FitchFork, we will not only be using your functions to create the list, but we will set the correct structure (such as List, Queue, Stack) ourselves and then run operations on it.

- Members
  - head: Node<T>
    - This is the head node for CLinkedList.
    - Remember to set this to NULL if the list is empty.
- Functions
  - CLinkedList()
    - Constructor for the CLinkedList class.
    - Should set head to NULL.
  - ~CLinkedList()
    - This is the destructor for the CLinkedList class.
    - It should delete all nodes.
  - prepend(data: T): void
    - Prepends an item to the list.
  - append(data: T) void
    - Appends an item to the list.
  - insert(data: T, index: int): void
    - Index starts at 0.
    - Inserts the passed in data at the specified index.
    - The list should grow to accommodate the new node, i.e if the list is currently:

      `[1,6,3,4,2,6,3]`

      And insert(3,1) is called, (with 1 being the index in this case), the list should look as follows:

      `[1,3,6,3,4,2,6,3]`
    - If the index is invalid, nothing should happen to the list. An invalid index for this function is an index number that is smaller than zero or an index number that is larger than the current size of the list. If the index that is sent in is the current size (such as 8 in the previous example after the insert), then it is equivalent to *append*().
  - remove(index: int): void
    - Removes a node at a specified index.
    - If the index is invalid, nothing should happen to the list.
    - An invalid index for this function is an index number that is smaller than zero or an index number that is larger than or equal to the current size of the list.

- removeElements(data: T): void
  - This function removes all nodes that are equal to the parameter that is passed in.
  - This should remove ALL instances, not just the first.
  - If no items match, then nothing should happen.
- print() const: void
  - This should print the list to the console.
  - The format should be each node on its own, separated by a " -> ".
  - Example: If the list is [1,5,9,393,0], the following should be printed:

```
1 -> 5 -> 9 -> 393 -> 0\n
```

  - If the list is empty, simply print "Empty\n".
- reverse(): void
  - This function reverses the list.
- clear(): void
  - This function clears the list.
- length() const: int
  - This function returns the number of nodes in the list.
- get(index: int) const: T
  - Return the value of the node at that index
  - If the index is invalid, then return the default value.
  - To return the default value when working with templates, us 'return T()'. [1]
  - An invalid index for this function is an index number that is smaller than zero or an index number that is larger than or equal to the current size of the list.
- operator[](index: int) const: T
  - Overloaded [] operator, behaves identical to *get*().
- filter(op: std::string, query: T): void
  - This function takes in a query and filters the list based on the query, only keeping nodes that match the query, while getting rid of the nodes that do not match.
  - This function takes in an operator and a query, with the following operators that are supported:
    (">","<","==","!=",">=","<=")
  - This operator is applied to query
  - **Example with T=int:** An operator of '>' and a query of '1' means only keeping nodes that are larger than 1 and gets rid of the rest.
  - If an invalid operator is passed in, then do nothing.
  - If an invalid query is passed in, then do nothing.

[1] An important note this is not best practice since we are now restricted to classes that have a empty constructor, however for invalid indexes normally we would just throw an exception but they are not in the scope of this assignment

    * You should use the only keep nodes that pass the operator and the query, while removing the nodes that don't.

- swap(index1: int, index2: int): void
    * If the indexes are valid, swap the data at the specified indices.
    * If invalid indices are passed in, then do nothing.
- slice(start: int, end: int): void
    * If *start* and *end* are valid indices, slice the list such that it only contains the elements between and including the 2 passed in indices.
    * For example, if the list is currently:

      `[1,6,3,4,23,2,6,3,0,51,9]`

      And *slice(2,5)* is called, the list should look like the following:

      `[3,4,23,2]`
    * If either index is invalid, do nothing.
    * If *start* is larger than *end*, do nothing.
    * A slice should always keep at least 1 node, i.e. *slice(3,3)* is valid, as it should only keep the element at index 3.
- RRotate(k: int): void
    * If $k$ is negative, do nothing and just return.
    * This function rotates a list towards the right $k$ times.
    * Rotating a list towards the right involves shifting each node to the right, in the case of a list, it is done by making the last node the first node, $k$ times.
    * Say we have the list [1,2,3,4,5,6,7,8,9,10,11]. Note the data does not matter, this is just for clarity. If the list is Right Rotated 3 times, the list would be [9,10,11,1,2,3,4,5,6,7,8].

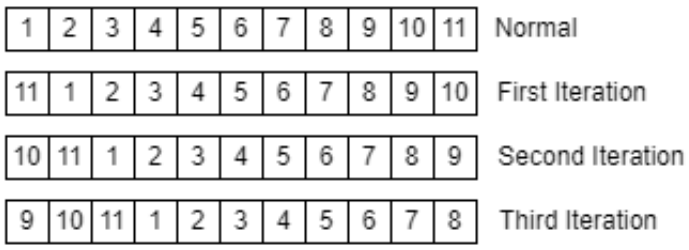


Figure 2: Right Rotate

    * There is no fixed limit to the number of times you can perform a list rotation. In other words, even if the list contains only 5 elements, you can rotate it 52 times, and it will continue to cycle. For example, if you were to rotate it 52 times, the effect would be akin to a right rotation of 2. This is because if you rotate a list of size $n$, $n$ times, the resulting list will be equivalent to the starting list.
- LRotate(k: int): void
    * If $k$ is negative, do nothing and just return.

    - This function rotates a list towards the left $k$ times.
    - Rotating a list towards the left involves shifting each node to the right, in the case of a list, it is done by making the first node the last node, $k$ times
    - Say we have the list [1,2,3,4,5,6,7,8,9,10,11]. Note the data does not matter, this is just for clarity. If the list is Left Rotated 3 times, the list would be [4,5,6,7,8,9,10,111,2,3]


Figure 3: Left Rotate

    - There is no fixed limit to the number of times you can perform a list rotation. In other words, even if the list contains only 5 elements, you can rotate it 52 times, and it will continue to cycle. For example, if you were to rotate it 52 times, the effect would be akin to a left rotation of 2. This is because if you rotate a list of size $n$, $n$ times, the resulting list will be equivalent to the starting list.

- operator=(other: const CLinkedList<T>&): CLinkedList<T>&
    - This is an operator= for the CLinkedList class.
    - Ensure you are creating a **deep** copy of the nodes.
- operator==(other: const CLinkedList<T>&) const: bool
    - This function returns true if both lists contain the same nodes.
    - The nodes do not need to be in the same order.
    - The lists need to be the same size.
    - All elements should appear the same amount of times in each list. This means that [1,2,2] != [1,1,2].
    - Essentially, the lists should be identical, barring the order.
- operator+=(other: const CLinkedList<T>&) const: void
    - This function all append all elements of *other* to the current list.
    - A list object cannot += itself, be sure to check for this.
- operator-(other: const CLinkedList<T>&) const: CLinkedList<T>&
    - This function returns a new list.
    - The list that is returned should only contain elements that are in the current list, but not in *other*.
    - For example, if the current list is [1,4,2,4,3,5,90,12] and *other* is [12,4], then the new list should be [1,2,3,5,90].
    - It is possible to return an empty list.

- removeDuplicates(): void
  - Remove any duplicates in the list, only keeping the first instance of a duplicate.
- consume(other: CLinkedList<T>&): void
  - This function behaves similarly to *operator+=*, however, instead of creating new nodes, the actual nodes of *other* should be appended to the list.
  - After this function, *other* should have an empty list, since it is now part of the current objects list.
  - Make sure this function does not make any new nodes, only appends the nodes from *other*.
  - Remember to check if either list have nodes and handle them appropriately.
  - Ensure a list cannot consume itself.

## 6.4 Stack<T>

- Members
  - top: Node<T>*
    - A Node pointer that keeps track of the top of the stack.
- Functions
  - Stack()
    - Default constructor for Stack class, simply initializes the top of the stack.
  - ~Stack()
    - Destructor for the Stack class.
    - It should delete all nodes.
  - Stack(other: const Stack<T>&)
    - Copy constructor for stack class.
  - push(data: T): void
    - Pushes data to the stack.
  - pop(): T
    - Pop and return a value off the stack.
    - The Node should be deleted.
    - If the stack is empty then return the default value, 'return T()'.
  - peek() const: const T
    - Peek the top of the stack.
    - If the stack is empty return the default value, 'return T()'.
  - isEmpty() const: bool
    - Returns true if the stack is empty, and false otherwise.

### 6.5 Queue<T>

- Members
  - head: Node<T>
    - The head of the queue.
  - tail: Node<T>
    - The tail of the queue.
- Functions
  - Queue()
    - Constructor for the Queue class, initializes all needed variables.
  - Queue(other: const Queue<T>&)
    - Copy constructor for Queue class taking in a reference to another queue.
    - Creates a deep copy.
  - ~Queue()
    - Destructor for the Queue class.
    - It should delete all nodes.
  - clone() const: Queue<T>*
    - Creates and returns a dynamic cloned copy of the queue.
  - enqueue(data: T): void
    - Enqueues an element to the queue.
  - dequeue(): T
    - Dequeues and returns the node from the queue.
    - If the queue is empty then return the default value, 'return T()'.
  - peek() const: const T
    - Peek the head of the queue.
    - If the queue is empty then return the default value, 'return T()'.
  - isEmpty(): bool
    - Returns true if the queue is empty, and false otherwise.

# 7 Memory Management

Memory management is a core part of COS110 and C++, so this assignment memory management is extremely important due to the scope. Therefore 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.

# 8 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the practical 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 the 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 [2] 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 main
```

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 | Final mark |
|---|---|---|
| 0%-5% | 0% | 0 |
| 5%-20% | 20% | 8 |
| 20%-40% | 40% | 16 |
| 40%-60% | 60% | 24 |
| 60%-80% | 80% | 32 |
| 80%-100% | 100% | 40 |

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 function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the Instructor Provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

# 9 Upload checklist

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

- main.cpp
- List.cpp
- Stack.cpp
- Queue.cpp
- CLinkedList.cpp

You will notice you do not need to upload a makefile or any .h files, you may if you wish, however, FitchFork will overwrite them before running your submission.

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.

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

# 10 Submission

You need to submit your source files, only the cpp files (**including the main.cpp**), on the FitchFork website (https://ff.cs.up.ac.za/). All methods must be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. There is no need to include any other files or .h files in your submission. **Ensure your files are in the root directory of the zip file**. Your code should be able to be compiled with the C++98 standard
For this practical, you will have 5 upload opportunities. Upload your archive to the Assignment 3 slot on the FitchFork website.