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 8 Specifications

Release Date: 23-10-2023 at 06:00

Due Date: 27-10-2023 at 23:59

Total Marks: 250

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 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 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 apart from the provided functions.
- Skeleton files have been provided for you with correct linking. Do not change the linking provided, as this may lead to compilation errors.
- Do not change any of the provided functions. This is used by FitchFork to print out your linked lists and nodes.
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.

2 Overview

For this practical, you will be implementing a basic singly linked list in C++. This will be used to reinforce the content covered in class by inserting, removing and traversing a linked list. You will also be expected to sort the linked list.

3 Background

3.1 Singly Linked Lists

Singly-linked lists are the most basic form of a linked list that can be encountered. The linked list is an alternative data structure for an array. A linked list consists of two primary components, namely a linked list container and a linked list node. The container usually contains the algorithms for the linked list and the nodes form the links in the chains. The linked list node contains two member variables, namely a data member that stores the data for the node and a next pointer to the next node in the linked list.

Figure 1 illustrates an unordered singly-linked list where each blue shape contains the data of the node and the arrow binding two shapes are the *next* pointers. The black dot represents the *head* member variable and the black dot with the ring represents NULL.

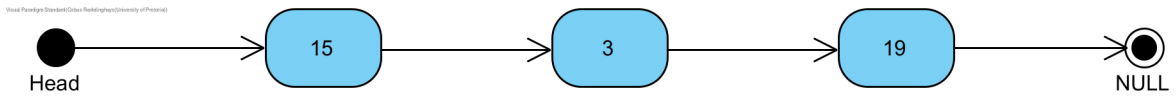

Figure 1: Example of an unordered singly linked list

4 Your Task:

You are required to implement the following classes. Pay close attention to the function signatures as the .h files will be overwritten, therfore failure to comply with the UML will result in a mark of 0.

Skeleton files have been provided for you with correct linking. Do not change the linking provided, as this may lead to compilation errors. Also, do not change any of the provided functions. This is used by FitchFork to print out your linked lists and nodes.

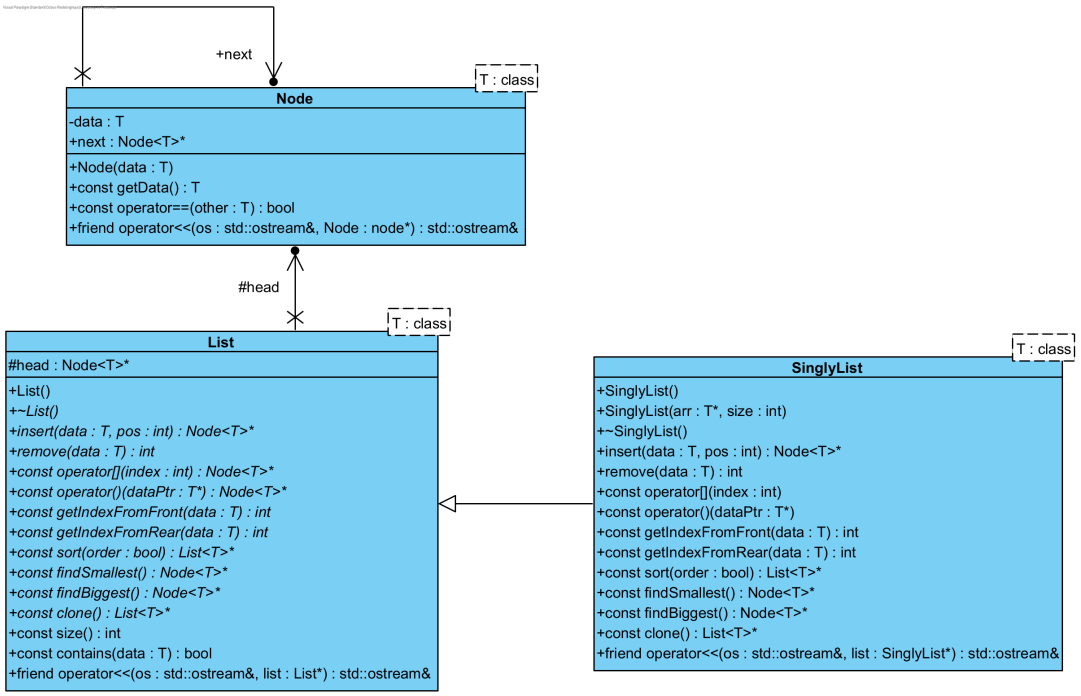

Figure 2: Class diagrams

4.1 Node

This class will represent a single node in the linked list. Node is a template class of type T.

- Members:
  - data: T
    - This member is the data of the linked list node.
  - next: Node<T>*
    - This member is a pointer to the next node in the linked list.
- Functions:
  - Node(data: T)
    - This is the constructor for the Node class and should initialize the member variables appropriately.
    - See the slides for a hint on what next should be initialized to.
  - const getData(): T
    - This is a constant function that returns the data of the node.
  - const operator==(other: T): bool
    - This is a constant function that determines if the passed-in data parameter is equal to the left-hand side of the operator member.
    - Use template type T's == operator.
    - It can be assumed that any type T utilized in this practical will have == implemented.
  - friend operator«(os: std::ostream&, node: Node*): std::ostream&
    - This function has been provided for you.
    - Do not change this function.

Hint: Think back to default destructors to explain the lack of a destructor.

4.2 List

This is the class that will contain the abstract definition of the List hierarchy. List is a template class of type T.

- Members:
  - head: Node<T>*
    - This member is the head of the linked list.
    - Head indicates the start of the linked list.
- Functions:
  - List()

    - This is the constructor of the List class.
    - *Hint: see the slides for a hint on what head should be initialized to.*
- ~List()
    - This is a **virtual** destructor for the List class.
    - The destructor should be empty as the derived class will be responsible for deallocation.
    - **Although the function will be empty it should still be present in the implementation file.**
- insert(data: T, pos: int): Node<T>*
    - This is a **pure virtual** function.
    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function is responsible for inserting into the linked list.
    - The function should return the node that was inserted.
- remove(data: T): int
    - This is a **pure virtual** function.
    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function is responsible for removing from the list.
    - The function should return the index of the node that was removed.
- const operator[](index: int): Node<T>*
    - This is a **pure virtual const** function.
    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function is responsible for retrieving the node at a specified index.
- const operator()(dataPtr: T*): Node<T>*
    - This is a **pure virtual const** function.
    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function is responsible for retrieving the node with the passed-in data.
    - Note you will need to dereference the dataPtr when comparing.
- const getIndexFromFront(data: T): int
    - This is a **pure virtual const** function.

    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function will return the index of the data member, starting at the front of the list.
- const getIndexFromRear(data: T): int
    - This is a **pure virtual const** function.
    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function will return the index of the data member, starting from the back of the list.
- const sort(order: bool): List<T>*
    - This is a **pure virtual const** function.
    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function will be used to return a *new* sorted list based on the passed-in parameter.
- const findSmallest(): Node<T>*
    - This is a **pure virtual const** function.
    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function will return the node containing the smallest data value.
- const findBiggest(): Node<T>*
    - This is a **pure virtual const** function.
    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function will return the node containing the biggest data value.
- const clone(): List<T>*
    - This is a **pure virtual const** function.
    - Since this is a pure virtual function, it must **not** be implemented. A short description of what you can expect the derived class should do has been added for the sake of clarity.
    - This function will return a populated deep copy of the current list.
- const size(): int
    - This is a **const** function.

    - This function should count the number of nodes in the list.
    - If the list is empty, the function should return 0.
  - const contains(data: T): bool
    - This is a **const** function.
    - This function should return true if the passed-in parameter is contained in the list and false otherwise.
    - If the list is empty, the function should return false.
  - friend operator«(os: std::ostream&, node : List*): std::ostream&
    - This function has been provided for you.
    - Do not change this function.

Remember the **pure virtual** functions will be implemented by the derived class. The descriptions above are just to aid in your understanding of what the function can be used for.

4.3 SinglyLinked

This is the class that will contain the derived definition of the List class. SinglyLinked publically inherits from List.

- Functions:
  - SinglyLinked()
    - Using the theory taught in the lectures, determine if it is required for this constructor to initialize anything. If required, do so, else leave the function empty in the implementation file.
  - SinglyLinked(arr: T*, size: int)
    - This constructor should populate the linked list with the values in the array.
    - The order of elements in the passed-in array should be maintained in the linked list.
    - If the array or the size is invalid (null or negative), the list should remain empty.
    - WARNING: This function will be used to populate the linked list in most of the test cases. Ensure this function works as expected.
  - ~SinglyLinked()
    - This is the destructor for the SinglyLinked class.
    - This function should destroy any remaining nodes in the list and deallocate any allocated memory.
  - insert(data: T, pos: int): Node<T>*
    - This function is responsible for inserting into the linked list.
    - This function should insert the passed in data into the list at the specified position and return the newly created node after linking.

    - If the passed-in position is smaller or equal to zero the node should be prepended to the list.
    - If the passed-in position is greater than the number of nodes in the list the node should be appended.
    - Else insert the node at the correct position where indexing starts at 0.
    - *Hint: the first node is at index 0, the second node at index 1 and so on.*
    - For example:
        - Using Figure 1, if a new node is inserted at position 1 it will be inserted between 15 and 3.
- remove(data: T): int
    - This function is responsible for removing the passed-in data from the list.
    - This function should remove the first occurrence of the data if duplicates exist.
    - This function will return the index of the node that was removed, with the indexing starting at 0.
    - If the passed-in parameter is not in the list, the function should return -1.
    - For example:
        - Using Figure 1, if 3 is removed the function will return 1.
- const operator[](index: int): Node<T>*
    - This is a **const** function.
    - This function is responsible for retrieving the node at a specified index starting at the front of the list with an index of 0.
    - If a negative index is passed in as a parameter the function should return the node at the specified index starting from the rear and -1. Thus the last node in the list will have an index of -1. The second last an index of -2 .
    - If the index and so for this out of the bounds of the list, the function should return NULL.
    - For example:
        - Using Figure 1, if index 1 is passed in, a node with data 3 is returned.
        - Using Figure 1, if index -1 is passed in a node with data 19 is returned.
- const operator()(dataPtr: T*): Node<T>*
    - This is a **const** function.
    - This function is responsible for retrieving the node with the passed-in data.
    - The function should return the first occurrence of the data.
    - If the data is not in the list, the function should return NULL.
    - Note: you will need to dereference the dataPtr when doing comparisons.
- const getIndexFromFront(data: T): int
    - This is a **const** function.

    * This function should return the index of the data member starting from the front of the list.
    * Return the index of the first occurrence of the data.
    * The indexing starts at 0.
    * If the data is not in the list the function should return -1.
    * For example
        · Using Figure 1, the node with data 15 has a index of 0, starting from the front.
- const getIndexFromRear(data: T): int
    * This function should return the index of the data member from the back of the list.
    * Return the index of the first occurrence of the data.
    * The indexing starts at 0.
    * If the data is not in the list the function should return -1.
    * For example
        · Using Figure 1, the node with data 15 has an index of 2 starting from the rear.
- const sort(order: bool): List<T>*
    * This is a **const** function.
    * This function must return a newly created sorted list based on the passed-in parameter.
    * If the passed-in parameter is **true**, the list should be sorted in *ascending* order. If the passed-in parameter is **false**, the list should be sorted in *descending* order.
    * The original list should not be altered.
    * Do some research on different sorting algorithms, namely: bubble sort, insertion sort and selection sort. A smart choice in the chosen algorithm can simplify this function while a poor choice can add unnecessary complexity. Choose wisely.
- const findSmallest(): Node<T>*
    * This is a **const** function.
    * This function will return the node containing the smallest data value.
    * Return the first occurrence of the smallest data.
    * If the list is empty, the function should return null.
- const findBiggest(): Node<T>*
    * This is a **const** function.
    * This function will return the node containing the biggest data value.
    * Return the first occurrence of the biggest data.
    * If the list is empty, the function should return null.
- const clone(): List<T>*
    * This is a **const** function.

    - This function should return a newly created list that is a populated deep copy of the current list.
    - The original list should not be altered.
    - If the original list is empty, the resulting list should also be empty.
- friend operator«(os: std::ostream&, list: SinglyLinked*): std::ostream&
    - This function has been provided for you.
    - Do not change this function.

5 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-allocate *all* the memory that was allocated.

6 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 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%	4

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

20%-40%	40%	8
40%-60%	60%	12
60%-80%	80%	16
80%-100%	100%	20

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.

7 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.
- Do not include using namespace std in any of the files.
- You may only use the following libraries:
  - IOStream
  - String

8 Upload Checklist

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

- Node.h
- Node.cpp
- List.h
- List.cpp
- SinglyLinked.h
- SinglyLinked.cpp
- main.cpp

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.

Skeleton files have been provided for you with correct linking. Do not change the linking provided, as this may lead to compilation errors. Also, do not change any of the provided functions. This is used by FitchFork to print out your linked lists and nodes.

9 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 8 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**