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: Doubly Linked List

Release date: 30-10-2023 at 06:00
Due date: 03-11-2023 at 23:59
Total Marks: 150

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.
- You may not import any of C++'s built-in data structures. Doing so will result in a mark of zero. You may only make use of 1-dimensional native arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a mark of zero. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with C++98**

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. 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 reuse does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

On completion of this practical, you will have gained experience with the following:

- Implement a sorted doubly linked list.
- Use only a node class for all operations.

4 Introduction

Doubly linked lists are a variation of singly linked lists where all nodes have a link to the previous and next pointer in the list. This allows traversal of the list in both directions. Doubly linked lists are often implemented using a list class, where the head and tail of the list are stored, but technically all operations on a doubly linked list can be performed from any node since all nodes are available from every node. This way of defining functions integrates naturally with recursion, but since this has not been covered, you are expected to use iteration for this practical. It is recommended to come back to this practical after you learn about recursion to see how much easier this practical is using recursion.

5 Tasks

You are tasked with implementing a doubly linked list **using only a node class**. This will be used to create a sorted list of integers that does not contain duplicate values.

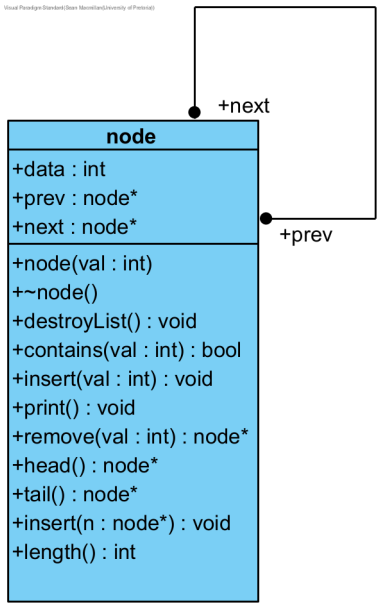

Figure 1: UML

- Members
  - prev: node*
    - This is the previous pointer.
    - This will point to the previous node in the list, if there is a previous node.
    - If there is no previous node, this will be NULL.
  - next: node*
    - This is the next pointer.
    - This will point to the next node in the list, if there is a next node.
    - If there is no next node, this will be NULL.
  - data: int
    - This is the data of the node.
- Functions
- **Note that all functions (except the constructor and destructor) can be called on ANY node in the list. A list refers to all nodes that are connected to the current node in both directions.**
  - node(val: int)
    - This is the node constructor.
    - Set the data variable to the passed-in parameter.
    - This should be an unlinked node after construction.
  - ~node()
    - This is the node destructor.
    - Set the previous and next pointers to NULL.

- destroyList(): void
  - This acts as a list destructor.
  - Calling this on any node in a list, should delete all of the nodes in the list that this node is part of.
- contains(val: int): bool
  - Returns true if the passed-in parameter is inside the current list.
  - Returns false if the passed-in parameter is not inside the current list.
  - *Hint: Make sure to check both directions.*
- insert(val: int): void
  - If the passed-in parameter is already inside the list, then do nothing.
  - This should create a new node using the passed-in parameter.
  - The newNode should be added to the list so the list is always sorted in ascending order.
- insert(n: node*): void
  - This function should combine two lists. After this is called, all nodes in both lists should be in the same list.
  - If the passed-in parameter and the current object have the same data, **DO NOT** combine the lists, as it will lead to unpredicatable behaviour if the passed-in node is deleted.
  - The resulting list should be sorted, contain no duplicates and stay sorted.
  - If there are duplicates, the node from the current list should be kept, and the node from the passed-in list should be deleted (deleted using the destructor, **not** *destroyList*()).
  - Example:
    If list one is:

    ```
    1->3->5
    ```

    1

    And list two is:

    ```
    2->3->6
    ```

    1

    Then, when we add the two lists together, the result should be (Note that the colours shows which list the object comes from):

    ```
    1->2->3->5->6
    ```

    1

    And then node:

    ```
    3
    ```

    1

    was deleted.
- print(): void
  - This will print the current list on its own line.
  - Nodes should be separated by an arrow "->".
  - The values of the nodes are printed without spaces.
  - The value of the node on which this function was called should be inside square brackets.
  - Example:
    Suppose we have a list with nodes [1,2,3,4,5] and we call *print*() on the 3 node. The print function should print:

    ```
    1->2->[3]->4->5
    ```

    1

- length(): int
  - Return how many nodes are in the current list.
- remove(val: int): node*
  - This function will remove a node from the list.
  - It should **not** delete the node, but merely unlink the node (it should not point to any nodes after removing) and return it.
  - If no node matches the passed-in parameter, then do nothing.
- head(): node*
  - This should return the first node in the list.
- tail(): node*
  - This should return the last node in the list.

6 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.

7 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 node
```

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

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

and 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 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.

8 Upload checklist

The following files should be in the root of your archive

- main.cpp
- node.cpp

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

9 Allowed libraries

- iostream
- sstream

10 Submission

You need to submit your source files, only the .cpp files, on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to 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. Your code must be able to be compiled with the C++98 standard

For this practical, you will have 10 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**