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 Specification

## More Linked Lists

Release Date: 24-10-2022 at 06:00

Due Date: 28-10-2022 at 23:59

Total Marks: 19

# Contents

**1 General instructions** **2**

**2 Overview** **3**

**3 Your Task:** **3**

3.1 SortNode . . . . . 4

3.2 SortList . . . . . 5

**4 Uploading** **7**

# 1 General instructions

- This practical 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).
- Failure of your program to successfully exit will result in a mark of 0.
- 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.
- 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**

## 2 Overview

This practical will test your ability to:

- Use doubly-linked lists in a more complex situation
- Create a list that is a template class
- Sort a list

Ensure that you are familiar with linked lists and how they work.

In the previous practical you were exposed to a single-linked list. In this prac, you get to use a double-linked list (pointers to both next **and** previous nodes).
This list and nodes will be general-purpose, so the idea is this: you create nodes that go into a list, and then they can be sorted when you want. By creating a class that inherits from this node, you can have the list sorting capabilities and then extend the child lass as you want. In this practical, you will only create the generic part of this idea, so no sub-classing. You *will* encounter something similar in the assignment, in which you have a node-type class which many things inherit from.

**Scenario** Your boss likes the idea of a generic node class that anything can inherit from. That way any random class can have some value it has used for sorting. He also insists on the following, despite your protests: the list should only be sorted at certain times, and not sorted whenever a new item is added. i.e A new item is not added at the correct place, it is added at the end of the list using the *tail* pointer, resulting in essentially immediate addition.
Lastly, the list should have the option to swap from an ascending sorting scheme, to a descending sorting scheme.

**Important:** This practical uses template classes. You will be compiling them as in Practical 4. That is,

1. Include the cpp file for a template class in the h file
2. Include the h file for the template class in the file you want to use that class (e.g a main)
3. Compile only the non-template class files

## 3 Your Task:

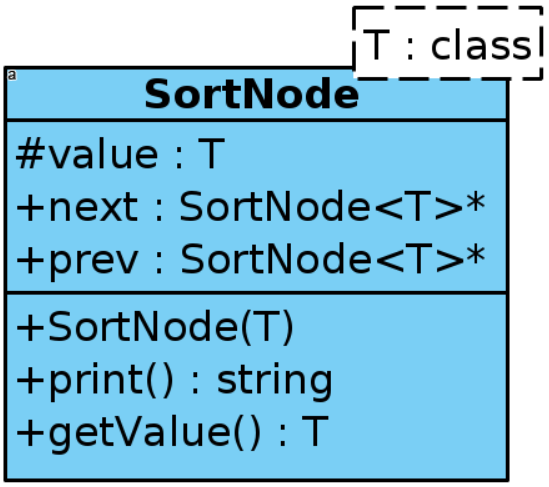
Create both generic node and double-linked list classes, that store items in given order. Then, sort either ascending or descending according to value.

You have been provided with the necessary files, in which you must complete the practical.

**Note:** Some of the UML may specify functions without named parameters. These are modeled after class definitions, which may omit the actual parameter names in the parameter lists. Obviously, when implementing functions in cpp files, the parameters should have names.

## 3.1 SortNode

The first thing to do would be to create the actual node class.



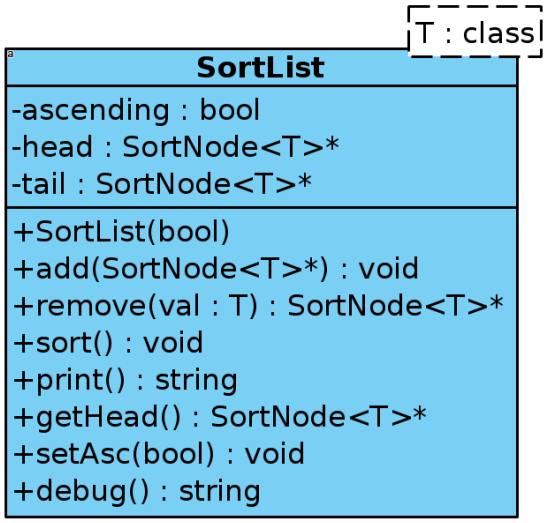

**Variables:**

- **value : T**
  This is the value of a node. It could be anything, and it's made protected so that a child class can set this value to be anything it wants, being able to even change its value as time goes on.
- **next**
  The pointer to the next node
- **prev**
  The pointer to the previous node

**Functions:**

- **SortNode(val : T)**
  Constructor, passing in a value for the node
- **print() : string**
  This returns a string representation of the value of the node. Just the value in the form of a string, nothing else (no newlines at the end or anything). In principle, this can be declared virtual (not pure virtual) and overridden in a child class to return more information. You don't need to do this for this prac (it does not matter)
- **getValue() : T**
  This returns the value in type T form

## 3.2 SortList

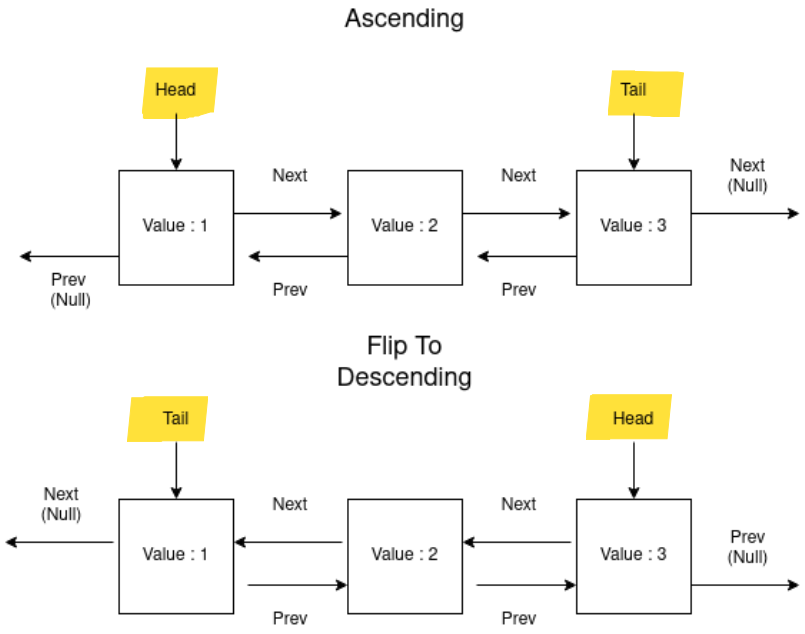



**Variables:**

- **ascending : bool**
  Whether or not the list should sort in ascending or descending. Ascending would mean this variable is *true*
- **head : SortNode<T>***
  The head of the list
- **tail : SortNode<T>***
  The tail of the list

**Functions:**

- **SortList(asc : bool)**
  Constructor, with a *true/false* indicating if it should be ascending/descending when sorted
- **add(a : SortNode<T>*) : void**
  Add a node to the end of the list
- **remove(val : T) : SortNode<T>***
  Remove the first node in the list with the passed-in value. The node should then be returned. If not found, return null.
- **setAsc(a : bool)**
  Set the *ascending* member variable. Note that the list **might just** already be sorted in one way. So, by swapping *ascending*, we are basically making the sorting of the list a worst-case scenario, since it was already sorted for the other direction. Luckily, this is a double-linked list, so we can swap the head and tail. In addition, all the links of the nodes need to be swapped, so the next/previous chain will be reversed. See below for a visual:




By doing this, we ensure that once the sorting algorithm is called, it should execute quite quickly if the items were already sorted to begin with (even though it still has some overhead)

- **sort() : void**
  Sorts the items in the list by value, either ascending or descending.
  **Hint:** For the sorting itself, you can use any sorting algorithm you want to continually place nodes in the right place. Alternatively, you can create a new temporary list, and just keep finding the smallest node in the main list, removing it and adding it to the new list. Do this until the old list is empty, then just replace it with the new list. Use whichever method you want

- **print() : string**
  Return a string with all the nodes printed, separated by commas. So, add the result for the *print* of each node, and a comma after each one (except the last one). e.g '5,6,8,9'

- **getHead() : SortNode<T>***
  Return the head of the list

- **debug() : string**
  This method will not actually be tested when you submit, and is just for your own testing purposes. You can add any output you want, in any format you want. I would recommend something similar to print, adding the output both forwards and backwards (using *prev*) to ensure your links all work as they should in both directions.

Remember to test your code well. Feel free to use and extend the given main.cpp file. When creating your own tests, remember to check edge-cases. Those would be things like removing the head/tail, sorting an empty/one-item list, removing then sorting then reversing then sorting again, etc.
There are many things that can go wrong when it comes to linked lists, so **test everything!**

Some of the more complicated linked lists can be difficult to visualise when you are doing operations on them. When creating your flipping/sorting algorithms, it helps to draw/write down the lists. From there, try to think about all the cases you need to account for, which pointers need to update, etc.

# 4 Uploading

Upload the following files in an archive:

- DLinkList.cpp, DLinkList.h
- SortNode.cpp, SortNode.h