UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science
COS110 - Program Design: Introduction
Practical 8

1 Introduction

Deadline: 23rd of October, 18:00

1.1 Objectives and Outcomes

The objective of this practical is to test your understanding of the programming concepts covered in the theory classes. In particular, this practical will test your understanding of templates and a new data structure, the linked list, in addition to other previously covered topics.

1.2 Submission

All submissions are to be made to the **assignments.cs.up.ac.za** page under the COS 110 page, and for the correct practical slot. Submit your code to Fitchfork before the closing time. Students are **strongly advised** to submit well before the deadline as **no late submissions will be accepted**.

1.3 Plagiarism

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone elses work without consent, copying a friends work (even with consent) and copying textual material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to **http://www.ais.up.ac.za/plagiarism/index.htm** (from the main page of the University of Pretoria site, follow the *Library* quick link, and then click the *Plagiarism* link). If you have questions regarding this, please ask one of the lecturers, to avoid any misunderstanding.

1.4 Implementation Guidelines

Follow the specifications of the practical precisely. For each practical, you will be required to create your own makefile so pay attention to the names of the files you will be asked to create. If the practical requires you to submit additional files of your own, follow the file structure and format exactly. Incorrect submissions will use up your uploads

and no extensions will be given. In terms of C++, unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the practical, will not be allowed. Some of the appropriate files that you submit will be overwritten during marking to ensure compliance to these requirements. If the specification makes use of text files, for providing input information, be sure to include blank text files with the specified names.

1.5 Mark Distribution

Activity	**Mark**
LL	35
Item	10
Total	**45**

2 Practical

2.1 Linked Lists

Linked lists are an example of an unbound data structure. Whereas the array is based around having a fixed size per array creation, there is no logical limit on the ability of a linked list to grow. Physical limitations aside, linked lists are capable of growing largely without any limitations. To achieve this, they trade easier access to their individual elements. This is because linked lists have to be traversed from their root node to any node in question, unlike arrays which are capable of supporting random access to any index without traversing the others.

If a linked list uses another class, such as item, for the nodes it has, and both classes are using templates, then the linked list .cpp should have an include for the item .cpp as well to ensure proper linkages. You should compile all of your classes as you have previously done as individual classes then linked together into the main.

Additionally, you will be not be provided with mains. You must create your own main to test that your code works and include it in the submission which will be overwritten during marking.

2.2 Task 1

You are going to implement a linked list with some additional features beyond that of a standard linked list. This will consist of implementing two classes: **LL** and **item**.

2.2.1 LL

The class is described according to the simple UML diagram below:

```
LL<T>
-head: item<T>*
```

```
-size: int
-randomSeed:int
----------------------------
+LL(rS:int)
+~LL()
+getHead() const: item<T>*
+addItem(newItem:item<T>*):void
+randomShuffleList():void
+randomShuffleList(i:int):void
+pop():item<T>*
+getItem(i:int) const:item<T>*
+minNode():T
+maxNode():T
+getSize() const:int
+printList():void
```

The class variables are as follows:

- head: The head pointer of the linked list. It refers to the latest node to be added into the list.
- size: The current size of the list. This starts at 0 and increases as the list grows in size by 1 each time a node is added.
- randomSeed: An int representing a random seed that is used for generating random values by the class.

The class methods are as follows:

- LL(rS:int): The constructor for the class. It receives an int that represents random seed and should use that seed to set the random value generator.
- LL(): The destructor for the class. It should deallocate all of the memory of the class. Deletes should happen from the head of the list and progress from there.
- getHead(): This returns the head node of the list.
- addItem(newItem:item<T> $*$): This function adds the given new item into the linked list. If the value of the new item is lower than or equal to the smallest value in the list, it should be added at the end of the list. Otherwise it should be added in front of the current head of the list.
- randomShuffleList(): This function shuffles the current list by picking one of the nodes in the list at random. This node then becomes the new head of the list. All of the nodes that before this node, should be moved to the end of the list. For example (the values are for demonstration and not indicative of an actual list), given a list that look as

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

  The current head is at 1 and the new head is 3, then the list will look like:

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

You should not create or delete any of the nodes; just reorder them and their links.

- randomShuffleList(i:int): This function is an overload of the randomShuffleList function. It shuffles the current list by picking one of the nodes in the list at random. However, the difference is that it receives the index of the new head in the current list. This node then becomes the new head of the list. All of the nodes that before this node, should be moved to the end of the list. If the index is invalid, do nothing. For example (the values are for demonstration and not indicative of an actual list), given a list that look as

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

  The current head is at 1 and the new head is 3, then the list will look like:

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

  You should not create or delete any of the nodes; just reorder them and their links.

- pop(): This removes and returns head node in the list. The head should be correspondingly updated. If the list is empty, return null.

- getItem(i:int): This returns the node specified at the index in the list. The head node is index 0 and each node after increases its index by one. If the index is out of the bounds of the list, null should be returned.

- minNode(): This returns the smallest value found in the list. If the list is empty, it should return null.

- maxNode():This returns the largest value found in the list. If the list is empty, it should return null.

- getSize(): This returns the current size of the list.

- printList(): This function prints out the values stored in all of the nodes, from the head. The format of this output should be a single comma delineated list with a newline at the end. For example (the values are for demonstration and not indicative of an actual list)

  ```
  1,2,3,4,5
  ```

  as an example of the output of the list.

2.2.2 item

The class is described according to the simple UML diagram below:

```
item <T>
-data:T
--------------------
+item(t:T)
+~item()
+next: item*
+getData():T
```

The class has the following variables:

- data: A template variable that stores some piece of information.
- next: A pointer of item type to the next item in the linked list.

The class has the following methods:

- item: This constructor receives an argument and instantiates the data variable with it.
- ∼item: This is the destructor for the item class. It prints out "X Deleted" with no quotation marks and a new line at the end. X refers to the data stored in the item.
- getData: This returns the data element stored in the item.

You will be allowed to use the following libraries for each class:

- item: iostream, string
- LL: cstdlib

You will have a maximum of 10 uploads for this task. Your submission must contain **item.h, item.cpp, LL.cpp, LL.h, main.cpp** and a makefile.