UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science
COS110 - Program Design: Introduction
Practical 9

1 Introduction

Deadline: 30th 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 two data structures: the doubly linked list and the circular linked list.

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**
storage	10
cLL	10
dLL	10
ditem	5
citem	5
Total	**40**

2 Practical

2.1 Linked List Variants

Aside from the standard kind of linked list, there are a number of variants of the standard linked list that provide additional functionality by changing the way the linked list is structured, ordered or linked. Two of these variants are the doubly linked list which have two-way connections between each node and the circular linked list which has a connection between the last node and the first node.

With regards to linking of the various classes, it is important to know the correct method. Specifically, 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

This task will consist of a class **storage** that will contain two classes **dLL** and **cLL**. This class encapsulates the two linked list classes and will be used as the primary interface for accessing the underlying list classes in terms of providing data to the lists.

2.2.1 storage

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

```
storage<T>
-clist: cLL<T>*
-dlist: dLL<T>*
-randomSeed:int
---------------------------
+storage(rS:int)
+~storage()
+storeData(data:T):void
+printCList():void
+printDList():void
```

The class variables are as follows:

- clist: A pointer to hold a cLL object.
- dlist: A pointer to hold a dLL object.

The class methods are as follows:

- storage(rS:int): The constructor for the class. It should initialise the two list objects as empty lists and initialise the random number generator.
- storage(): The destructor for the class. It should deallocate all of the memory of the class, with the clist being deallocated first.
- storeData(data:T): This function receives an argument of type T. It will add this argument into the cList if it is even and if the argument is odd, it will go into the dList. You can assume T will only refer to types that are numeric in nature.
- printCList(): This prints out the contents of the cList in the format described in the cLL class.
- printDList(): This prints out the contents of the dList in the format described in the dLL class.

2.2.2 dLL

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

```
dLL<T>
-head: ditem<T>*
-tail: ditem<T>*
-size: int
---------------------------
+dLL()
+~dLL()
+getHead(): ditem<T>*
+getTail(): ditem<T>*
+push(newItem:ditem<T>*):void
+pop():ditem<T>*
```

```
+getItem(i:int):ditem<T>*
+minNode():T
+getSize():int
+printList():void
```

The class variables are as follows:

- head: The head pointer of the doubly linked list.
- tail: The tail pointer of the doubly linked list.
- size: The current size of the doubly linked list. This starts at 0 and increases as the list grows in size.

The class methods are as follows:

- dLL: The class constructor. It starts by setting the variables to null and 0 respectively.
- ∼dLL: The class destructor. It will deallocate all of the memory in the class, starting with the head.
- getHead: This returns the head pointer of the doubly linked list.
- getTail: This returns the tail pointer of the doubly linked list.
- push: This adds a new ditem to the doubly linked list. If the value of the item is smaller than the smallest of the ditems in the list, then it should be added at the front of the list. Otherwise it should be added at the back.
- pop: This returns the top ditem of the linked list. The ditem is returned and removed from the list.
- getItem: This returns the ditem of the linked list at the index specified by the argument but without removing it from the list. If the index is out of bounds, return null.
- minNode: This returns the value of the ditem that has the lowest value in the linked list.
- getSize: This returns the current size of the linked list.
- printList: This prints out the entire list in order, from head to tail. Each ditem's data value is separate by a comma. For example: 3.1,5,26.6,17.3. Remember to add a newline to the end of the output.

2.2.3 ditem

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

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

The class has the following variables:

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

The class has the following methods:

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

2.2.4 cLL

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

```
cLL<T>
-head:citem<T> *
-size:int
------------------------
+cLL()
+~cLL()
+isEmpty():bool
+getSize():int
+push(newItem: citem<T>*):void
+pop():item<T>*
+removeAt(x:T):citem<T> *
+printList():void
```

The class variables are as follows:

- head: The head pointer for the linked list.
- size: The current size of the circular linked list. It starts at 0 and grows as the list does.

The class methods are as follows:

- cLL: The constructor. It will set the size to 0 and initialise head as null.
- ∼cLL: The class destructor. It will iterate through the circular linked list and deallocate all of the memory assigned for the items. The head should be deleted last.
- isEmpty: This function returns a bool. If the circular linked list is empty, then it will return true. Otherwise, if it has citems then it will return false.
- getSize: This returns the current size of the circular linked list. If the list is empty the size should be 0.
- push: This receives a new citem and adds it to the circular linked list. If the value of the item is larger than the largest value of the citems in the list, then it should be added at the front of the list, in front of the head. Otherwise it should be added at the back.
- pop: This receives, and returns, the first element in the list. The first element is removed from the list in the process. If the list is empty, return null. The first element referring to the head pointer in this case.
- removeAt: This will remove an citem from the linked list based on its value. If the value is not found, nothing should be removed. Also note that in the event of multiple values being found, the first one in the list, from head, should be removed. Note that nothing is deleted. Instead the node must be removed through relinking and then returned.
- printList: This will print out the entire list from head onwards. The output consists of a single comma delimited line, with a newline at the end. For example:

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

2.2.5 citem

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

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

The class has the following variables:

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

The class has the following methods:

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

You will be allowed to use the following libraries by class:

- citem: iostream, string
- ditem: iostream, string

To guide your testing and implementation, the main should have the following includes:

- storage.h
- storage.cpp
- dLL.h
- dLL.cpp
- cLL.h
- cLL.cpp

You will have a maximum of 10 uploads for this task. Your submission must contain **citem.h,citem.cpp, cLL.cpp, cLL.h, ditem.h, ditem.cpp, dLL.cpp, dLL.h, storage.h, storage.cpp, main.cpp** and a makefile.