Department of Computer Science
COS110 - Program Design: Introduction
Practical 7

1 Introduction

Deadline: 12th November, 19:30

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 the new data structure, the linked list, in addition to other previously covered topics.

1.2 Submission

All submissions are to be made to the **ff.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 else's work without consent, copying a friend's 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**
Task 1	62
Total	**62**

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.

3 Task 1

This task will consist of 4 classes. Each class is described in its own section below. There are two linked list classes called dList and train. Each has their own separate node class, dispatch and trainCar respectively.

3.1 Dispatch

```
dispatch
-order:string
+next:dispatch*
-----------------------------------------------
```

```
+dispatch(o:string):
+~dispatch():
+getOrder():string
```

The variables are as follows:

- order: A string representing a comma delimited order for a train. An example of this order is

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

- next: A pointer to the next node in the linked list.

The class methods are as follows:

- dispatch(o:string): The constructor for the dispatch class. The string is used to initialise the order variable. The next variable should be set to NULL.
- ∼dispatch(): The destructor for the dispatch class. When called this should print out the message ”X processed” with a new line at the end. X refers to the order variable.
- getOrder(): A getter for the order variable.

3.2 DList

```
dList
-head:dispatch *
-size:int
-rS:int
------------------------------------------------
+dList(rS:int):
+~dList():
+getHead() const:dispatch *
+getSize()const:int
+getItem(i:int):dispatch *
+firstOrder():string
+friend operator<<(output:ostream &,t:const dList &):ostream &
+addOrder(newOrder:string):void
+validateTrain():void
+shuffleOrders():void
```

The variables are as follows:

- head: The head variable of the linked list.
- size: The current number of dispatches in the linked list. When new orders are added, this variable should increase by 1.

- rS: A random seed variable.

The class methods are as follows:

- dList(rS:int): The constructor for the linked list class. It takes an argument which is used to instantiate the rS variable. The head pointer should be set to NULL and the size should be set to 0. The rS variable should be used to seed the srand generator here.
- ∼dList(): The destructor for the class. It should deallocate all of the memory of the class.
- getHead() const: This returns the head of the linked list. It is a constant function.
- getSize()const: This returns the size of the linked list. It is a constant function.
- getItem(i:int): This returns a dispatch indicated by the int index value passed in as an argument. If the index provides is invalid, this should return NULL or if the list is empty. The list is 0 indexed so the head will be at index 0.
- firstOrder(): This should return the order variable of the least dispatch in the linked list. This order is determined by lexicographical string order, with the < operator.
- operator<<: This prints out a list of the orders in the linked list. Each order should be separated by a comma with a newline placed at the end. For example

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

- addOrder(newOrder:string): This adds a new node into the linked list. The node should be initialised with the string provided as an argument. If the string 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. The order is based on the string order established by string comparison with the < operator.
- validateTrain(): This function removes a dispatch from the head of the link list. This should be removed from the list and the head updated accordingly. Once removed, this should be used to instantiate a new train object and the total cost of that train should be calculated. If this cost is less than 50, then output "Valid" with a newline at the end. Otherwise, output "Invalid" with a newline at the end. The constructed train object should be deleted appropriately at the end of the function. Remember to delete the dispatch before deleting the train. The calcCost function should be called before any deletes which should happen at the end of the function.
- shuffleOrders(): 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. Using the rand() function in conjunction with the size of the list is advised. 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.

3.3 TrainCar

```
trainCar
-cost:int
+next:trainCar *
------------------------------------------------
+trainCar(c:int):
+~trainCar():
+getCost():int
```

The variables are as follows:

- cost: An integer representing the cost of each train car.
- next: A pointer variable to the next train car in the linked list.

The class methods are as follows:

- trainCar(c:int): A constructor for the class that instantiates the cost variable with the provided argument. The next variable should be set to NULL.
- ∼trainCar(): The destructor for the trainCar class. When called this should print out the message ”X removed” with a new line at the end. X refers to the cost variable.
- getCost(): A getter for the cost variable.

3.4 Train

```
train
-head:trainCar *
-size:int
------------------------------------------------
+train(config:string):
+~train():
+calcTotalCost():int
+addCar(newCost:int):void
```

The variables are as follows:

- head: A pointer for the head of the linked list.
- size: The current size of the list. When a trainCar is added into the linked list, it should increase the size variable by 1.

The class methods are as follows:

- train(config:string): The constructor for the train class. It will receive a string argument that is going to be a comma delimited list of integers. This string must be used to instantiate a linked list of the train cars. For example, a string

  ```
  1,2,3
  ```

  would translate to a train linked list with 1 as the head.
- ∼train(): A destructor for the train class. When called it should deallocate all of the memory of the class.
- calcTotalCost(): This calculates the total sum cost of all of the train cars in the linked list. It will print the following message:

  ```
  Total Train Cost: X
  ```

  where X is the total cost of all the train cars in the linked list. Remember to add the new line at the end.
- addCar(newCost:int): This function adds a new trainCar into the linked list. The size of the list should be increased by 1 when the new node is added into the linked list.

4 Submission Guidelines

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

- dispatch: iostream, string
- dList: cstdlib, train, trainCar, dispatch
- trainCar: cstdlib, iostream, string
- train: sstream, trainCar.h

You will have a maximum of 10 uploads for this task. Your submission must contain **dispatch.h, dispatch.cpp, dList.h, dList.cpp, trainCar.h, trainCar.cpp, train.h, train.cpp, main.cpp** and a makefile.