UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

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

1 Introduction

Deadline: 26th of 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 of two data structures: stacks and queues.

1.2 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.

1.3 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.

1.4 Mark Distribution

Activity	**Mark**
heatNode and heatStack	20
msgNode and msgQueue	20
Total	**40**

2 Practical

2.1 Stacks

A stack is a data structure that that is a container. It follows the strategy of LIFO (Last In First Out) which means that items added, chronologically sooner, than later items are taken out after items that have been added later. This can be likened to what happens when making a pile of textbooks. Starting from no books, you place one book after another, each on top of the other. After a few books, getting to the books you first placed down, requires picking up the books you placed later.

2.2 Queue

A queue is a data structure that is a container. It follows the strategy of FIFO (First In First Out) which means that the oldest item in the queue is processed first. This can be likened to the process of waiting to be served at a restaurant. Like its namesake, a queue is formed where people who enter the queue before others are served before others.

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. **The queue in this practical is a doubly-linked list.**

Additionally, you will be not be provided with mains. You must create your own main to test that your code works.

3 Part 1: Stack

Imagine that you are an engineer on a distant planet. Your ship has crash landed and you are trying to build devices and machines to send a distress signal. There are two current problems to consider. The first is the heat sink management system for the signal tower and the second is the message dispatcher. Both of these components are ready, but require code in order to function properly. **You will be implementing a stack through the use of a dynamic array.**

3.1 heatStack

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

```
heatStack<T>
-stackSize: int
-stack: heatNode<T>**
----------------------------
+heatStack()
+~heatStack()
+push(t: heatNode<T>*):void
+pop():void
+peek():heatNode<T>*
+print():void
+validateCooling(heat: int *, numSinks:int):bool
```

The class variables are defined below:

- stackSize: The number of elements on the stack. Initially zero.
- stack: A one dimensional dynamic array which contains heatNode pointers. This array represents the stack. The last element in the array represents the top of the stack. The stack array contains the exact number of elements that are on the stack (i.e. there should be no extra space in the array). The stack member variable should be set to NULL when the stack is empty.

The class methods are defined below:

- heatStack: The class constructor. It will start by initialising an empty stack.
- ∼heatStack: The class destructor. It will deallocate all of the memory assigned by the class.
- push: This will receive a heatNode to add onto the current stack. A new array should be created of the correct size to add the additional element to the end of the array. The pointers in the old array should be shallow copied into the new array. A deep copy should be created of the heatNode passed in as a parameter. The array memory for the old stack should be freed.
- pop: This will remove the top heatNode from the stack. If it is empty, print out "EMPTY" with a newline at the end and no quotation marks. When removing the node, it should be deleted. The stack should be resized to a smaller size to contain the exact number of elements (i.e. one less element than before). If the new size is zero, then set the stack to be NULL.
- peek: This will return the top node of the stack but without removing it from the stack.
- print: This will print the entire contents of the stack. For each node in the stack (from the top of the stack), print the following information out sequentially, line by line. There should be an `endl` after each printed line. If the stack is empty do nothing. The format of this is as follows:

```
Heat Sink CL: X
```

X refers to the coolant level of a given node.

- validateCooling(heat: int *, numSinks:int): This function receives two variables. The first is an array of ints representing a cooling tower configuration. The second is the number of elements in the array. The array represents the cooling requirements of a potential configuration of the signal tower. The last element represents the top of the tower and corresponds to the top of the stack. Its value, an int, indicates how much cooling power the tower needs at that level. This function returns a bool, indicating whether the current heatStack's configuration, i.e. the heatNodes in the stack, are capable of validating the requirements. It returns true if this is the case and false otherwise.

  To successfully validate, the following conditions must all be met:

  1. The heatNode stack is not empty.
  2. There are at least the same number of heatNodes in the stack as the number of sinks required.
  3. The cumulative cooling power of the stack is greater than or equal to the requirements provided by the array.
  4. No heatNode has a coolant level smaller than the number of heatNodes.
  5. No heatNode has a coolant power smaller than the corresponding requirement for that level in the given configuration. If the heatNode stack contains more elements than the provided configuration, then ignore the extra heatNodes (i.e. only check the overlap between the stacks).

3.1.1 heatNode

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

```
heatNode<T>
-coolantLevel:T
-power:int
------------------------
+heatNode(i:T,p:int)
+~heatNode()
+getCoolantLevel() const:T
+getPower() const:int
```

The class variables are defined below:

- coolantLevel: This describes the amount of coolant held in the node.
- power: A variable which describes the strength of the coolant stored in the heatNode. A larger value indicates a stronger cooling potential.

The class methods are defined below:

- heatNode: A class constructor. It receives the coolantLevel and power for that node.
- ∼heatNode: The class destructor. It should print out "Heat Sink Removed" with no quotation marks and a new line at the end.
- getCoolantLevel(): Returns the coolant level.
- getPower(): Returns the power variable.

4 Part 2: Queue

4.1 msgQueue

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

```
msgQueue<T>
-head: msgNode<T>*
-tail:msgNode<T>*
----------------------------
+msgQueue()
+~msgQueue()
+enqueue(t: msgNode <T>*):void
+dequeue():void
+peek():msgNode<T> *
+print():void
+compileMessageData():void
```

The class variables are defined below:

- head: The current top of the queue. It will start as NULL but should refer to the top of the queue.
- tail: The end node of the queue. It will also start as NULL.

The class methods are defined below:

- msgQueue: The class constructor. It will start by initialising the variables to NULL.
- ∼msgQueue: The class destructor. It will deallocate all of the memory assigned by the class. Delete nodes from the from the front of the queue.
- enqueue: This will receive a new node to add to the queue. The node can simply be added to the queue as a shallow copy.
- dequeue: This will remove the top msgNode from the queue. If it is empty, print out "EMPTY" with a newline at the end and no quotation marks. When removing the node, it should be deleted.

- peek: This will return the top node of the queue but without removing it from the queue. If the queue is empty, return NULL.
- print: This will print the entire contents of the queue,from the head. For each node print the following information out sequentially, line by line. There should be an `endl` after each printed line. If the stack is empty do nothing. The format of this is as follows:

  ```
  Message I [Size: Y]
  ```

  I refers to the message index, with the head being index 0. Y refers to the size of the message in kilobytes.
- compileMessageData(): This function compiles a summary of all of the current number of messages in the queue. It should print out the following information:
  1. Number of messages: The total number of messages
  2. Total Size: The sum of all of the message sizes. The symbol appended to the total should reflect the size of the total. If the sum is smaller than a megabyte, KB should be used. If it is smaller than a gigabyte, MB should be used. Otherwise, GB should be used. Assume that 1000 kilobytes is 1 megabyte and 1000 megabytes is 1 gigabyte.

  An example format of the output is as follows (note that there should be an `endl` after each printed line):

  ```
  Total Number of Messages: 13
  Size: 1.3MB
  ```

4.2 msgNode

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

```
msgNode<T>
-message:T
+next:msgNode*
+prev:msgNode*
-size:int
------------------------
+msgNode(i:T,s:int)
+~msgNode()
+getMessage() const:T
+getSize() const:int
```

The class variables are defined below:

- message: This is the message attached into the node. It can take a variety of different forms depending on the nature of the message.

- next: A pointer to the next node of the queue.
- prev: A pointer to the previous node in the queue.
- size: A variable which indicates the size of the message in kilobytes.

The class methods are defined below:

- msgNode: A class constructor. It receives the message and the size for that node.
- ∼msgNode: The class destructor. It should print out "Message Processed" with no quotation marks and a new line at the end.
- getMessage: This returns message variable.
- getSize: This returns the size variable.

The libraries allowed are as follows:

- iostream
- string
- cstddef

5 Submission

You need to submit your source files on the Fitch Fork website (https://ff.cs.up.ac.za/). Place all of your `.h` files and your `.cpp` files in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number. Also place your **makefile** in this archive. There is no need to include any other files in your submission. **Do not put any folders in the .zip archive**. You have 10 submissions and your best mark will be your final mark. Do not use Fitch Fork to test your code because you have limited marking opportunities. Upload your archive to the Practical 9 slot on the Fitch Fork website for COS110. Submit your work before the deadline. No late submissions will be accepted!