Computer Science 3A

Practical Assignment 7

30 March 2017

Time: 30 March 2017 13:45 – 17:00 Marks: 50

Practical assignments must be uploaded to `eve.uj.ac.za` **before** 17h00 in the practical session.

Late submissions **will not be accepted**, and will therefore not be marked. You are **not allowed to collaborate** with any other student. You **must** upload your assignment to Eve **before** it will be marked.

Remember the Producer and Consumer? Sometimes allocating a task is not about it being fair to the workers, but it is about getting the task done. *Priority Queues* allow us to give priority to tasks, thereby allowing important tasks to be pushed to the front of the queue.

For this week's practical we are going to deal with the problem allocating **Interns** to high priority **Tasks**. Interns perform tasks that are allocated to them and use up energy to perform the task (or a part thereof). You need to implement a Priority Queue-based allocator that ensures the work of the most important tasks are completed first and the Interns are not overworked.

Furthermore, Heaps and a Heap sort can be realise the ordering of a Priority Queue. The implementation of a heap using a binary tree is a natural method of creating a heap-based structure.

On the basic level, a heap is simply a binary tree where the following conditions have been placed on the tree:

- key(v) $\leq$ key(parent(v))

If the following key property holds then **heap-order** has been maintained. The other condition that we are dealing with is the **complete binary tree** property.

Insertion from a heap always occurs in the **last node**, this is the rightmost node on the lowest height level. When items are removed from the heap, the items are removed from the root. In order to maintain the binary tree as a complete binary tree, the item in the last node is swapped with the entry in the root, and then the last node is removed. In both cases, in order to maintain heap order, an `upheap` or `downheap` operation must be performed.

In the case of a heap implementation, all heaps that store more than one value must use **sentinel nodes** as the leaf nodes. This means that the definition for **isExternal** must be modified in some cases.

Part of creating the heap is locating the last node. This must be done in two places, when an item is inserted and when an item is removed.

The following algorithm can be used to locate the last node on insert:

1. If the current last node is a **left child** then the sibling on the current last node is the new last node.
2. If not then traverse up the tree until you locate a node that is a **left child** or the **root of the tree** maintain this as the **current node**
3. If the current node is **a left child** then the current node becomes the sibling of this node.
4. Traverse down the **left branch** of the tree from the current node until you reach an **External node**. This should be a sentinel node.
5. The current node should then be returned as the new last node.

When you are removing an item from the heap, the reverse of the operation is performed.

An example for the output of the program can be found in the accompanying text file named *output.txt*.

The following files must be submitted to EVE:

1. *studentnumber*_p7.zip

Marksheet

1. Heap: upheap [10]
2. Heap: downheap [10]
3. Heap:getLastNodeInsert [5]
4. PriorityQueue: removeMin [5]
5. Main:allocateInternsWithPriorityQueue [10]
6. Compilation and Correct execution. [10]