Concordia University
Department of Computer Science and Software Engineering
Comp 352
Data Structure and Algorithms
Assignment 3 – Fall 2018
Due date: Monday November 12th, 2018 by midnight

Written Questions (50 marks):

Question 1

Draw a single binary tree that gave the following traversals:

Inorder: T K P C R J V I Q A L F B
Postorder: F C K T P J R A I V Q L B

Question 2

Assume that the binary tree from the above part (1) of this question is stored in an array-list as a complete binary tree as discussed in class. Specify the contents of such an array-list for this tree.

Question 3

A sequence is a list that supports the index-based operations of a List and the position-based operations of a Positional List (aka NodeList). The following questions concern sequences.
It is possible to implement the Sequence ADT using either a doubly linked lists, or an array of positions, as the underlying data structure. For each of the following cases, indicated which design is preferable and explain your answer:

i. A massive number of insertion operations are needed some of which are based on indices and some are based on insertion of positions at the start of the Sequence (addFirst).
ii. A massive number of addition before and after a position as well as adding a position at the end of the Sequence.
iii. A massive number of removal operations at positions and a massive number of setting the values at indices.

Question 4

Draw the min-heap that results from the bottom-up heap construction algorithm on the following list of values:

10, 17, 15, 25, 40, 19, 45, 16, 12, 8, 18, 14, 13, 9, 20, 11, 13

Starting from the bottom layer, use the values from left to right as specified above. Show immediate steps and the final tree representing the min-heap. Afterwards perform the operation removeMin 6 times and show the resulting min-heap after each step.

Question 5

Create again a min-heap using the list of values from the above part (1) of this question but this time you have to insert these values step by step using the order from left to right (i.e. insert 10, then insert 17, then 15, etc.) as shown in the above question. Show the tree after each step and the final tree representing the min-heap.

Note: You must submit the answers to all the questions above. However, only one or more questions, possibly chosen at random, will be corrected and will be evaluated to the full 50 marks.

模拟cpu 来安排process/job前后顺序，
模仿

Programming Questions (50 marks):

In this programming question, you need to build a program that simulates CPU (Central Processing Unit) scheduling for executing porcesses/jobs (these names will be used interchangably as they have the same meaning) on a computer system. You are going to use a priority queue to schedule the CPU jobs for the operating system. As a quick idea of what you need to do, the jobs, which are recorded in an initial array, are entred into a priority queue. The program will then keep looping, where each iteration will correspond to a time slice of the CPU where one of the jobs is partially executed, until the priroty queue is empty (which indicates that all jobs have been completed). Figure 1 illustrates the basic operation of the system. The fine details are provided below.

执行程序|任务

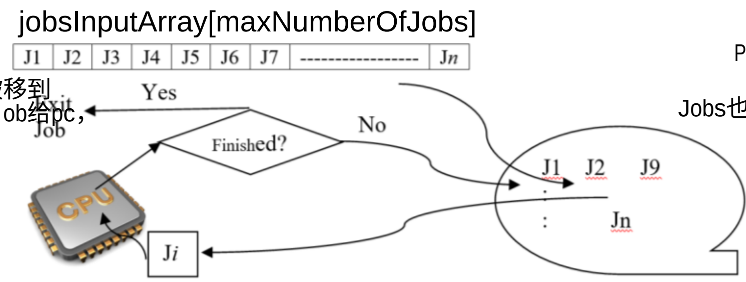

Figure 1: Basic Simulation of the System

1.所有的待执行的Job任务都会被移到pq中，然后pq会每一次会把一组job给pc，通过就就删除

PQ来给CPU的job安排先后顺序，

Jobs也就是what to do

Jobs一开始被记录在初始array中，然后enter到pq里，这个program会持续循环，

每一次迭代将对应CPU的一个time slice,其中一个Job会被执行，直到pq是空，（意味着所有job做完）

jobinput array有的几个数据

The input to the program is an array of jobs, called **jobsInputArray**, which holds a set of jobs, each consisting of **jobName** (String type) indicating the name of that process/job, **jobLength** (int type) indicating the needed CPU cycles for this job to terminate, **currentJobLength** (int type) indicating the remaining length of the job at any given time, **jobPriority** (int type) indicating the initial priority of this job, **finalPriority** (int type) indicating the final priority of the job at terminaion time, **entryTime** (long type) indicating the time this job entered the priority queue, **endTime** (long type) indicating when this job finally terminated, and **waitTime** (long type) indicating the total amount of wait time a process had to incure from the time it entered the queue until it terminates. All jobs must first be instered in the priority queue. Each insetion takes one unit of time, consequently the entryTime of a job must be set to that time. For instance, when the first job, J1, is inserted, its entryTime is recorded as 1. When J6 is insetred, its entryTime must be set as 6, since this is how much it took before this job is inserted, and so on.

job length 需要几个CPU循环完成这个job

current length,剩下的length

初始优先级

finalPriority Job终止时的优先级

每一个job必须被先输入进PQ，用了time 1
所以 第一个pq元素执行，ENTRYTIME是1,
J6,entry time是6

entrytime:这个job进入PQ几次，endtime 什么时候这个Job终止，waitTime,从这个job进入queue到种植的时间

Each job has a length (jobLength), which indicates how much time/CPU cycles the job needs to be allcated the CPU (served by the CPU) before it is finally terminated. The length of a job is always between 1 and 70 cycles. Additionally, each job has an initial priority (jobPriority), and a final prioirty (finalPriority), each is a value between 1 (highest priority) and 40 (lowest priority), inclusive. Your simulator should process the jobs from the priority queue based on the current priority of each job, and with First-Come-First-Served (FCFS) basis as a tie breaker in case these current priorities are the same. That is, if two jobs have different priorities, then the job with the higher priority will be executed first; otherwise, jobs with the same priority will be executed based on which one was first entered the queue. Your simulator must somehow keep track of that to enforce this rule. Additionally, there are also other important rules that the simulator must implement; particularly in relation to starvation avoidance, as detailed below.

优先级从1到40

当priority不同执行

Priority低的相同时，执行先进入PQ的

你的模拟器必须对谁先进入PQ有一个track

每一个Job都有1个Length，记录着他最终结束前需要进行几次cpu cycle,一定要1到70cycle

- To avoid low-priority processes from being starved for an unreasonable amount of time, the simulator must check periodically for starved jobs and change their current priorities to the highest priority, so they can finally execute. This is done periodically each time a total of 30 processes are terminated. That is, each time 30 processes are finished, the simulator must search the queue for the oldest job (recall jobs have an entry time field on them) that has NEVER been executed, then changes the current priority of that job to 1, adjusts its location in the queue if needed (depending on your implementation!), effectively resulting in this job

为了避免优先级较低的进程长时间处于starve状态（饥饿，这里理解为想要被进行进行不到），模拟器必须定期检查starving jobs并且改变他们当前优先级，这样他们才能最终执行，每有30个process被执行，模拟器必须查找到queue中最Old的job美进行过的，然后改变它的当前优先级为1，有需要的话，调整他当前位置

一旦优先级被改变了，直到这个job终结前，他的优先级都不会改变

being executed. Once the current priority is changed, it is kept as such until the job terminates. By repeating this process periodically (each time 30 jobs are finished), starvation of low priority jobs is mitigated. 通过周期性的进行这个过程，30job一次， 低优先级的starvation被减轻

So, in summary, once all jobs are inserted into the priority queue, the CPU execution process is started. The program will keep looping, where each iteration is considered a single CPU cycle. In each loop, the simulator will pickup a job from the queue (based on current priority and FCFS rule) and execute it. The execution will result in the following:

- ⇨ The current length of this job is decremented by 1.

  每一次进行循环，当前job的Length-1

- ⇨ The simulator must output (to the screen/monitor) the information of this running job: its name, its length, its current length, its initial priority and its current priority (these are often the same, unless the job has changed priority). This display should be similar to the following:

  模拟器必须输出当前Job:名字,长度，当前长度，初始优先级，现在优先级，这两个优先级，通常一样

  Now executing Job_285. Job length: 42 cycles; Current remaing length: 26 cycles; Initial priority: 22; Current priority: 1

  除非你改变了优先级成为1，

- ⇨ If the current length of the process is decremented to a value that is still greater than 0, then the job is inserted back in the queue, as illustrated in Figure 1. However, the job <u>MUST</u> be inserted *behind* all other jobs of the same current priority (*behind* here depends on how your implementation of the priority queue is made, so this may not physically mean behind). This in effect will allow other processes with the same priority to be picked-up for execution before this job is executed again. Your implementation <u>MUST</u> guarantee that this behavior is respected. <u>**If needed, you are allowed to add additional attributes to the Job class.**</u>

如果当前Process机器长度减少1以后还大于0，然后，他被插回queue，并且在所有相同priority之后进行

- ⇨ If the current length is decremented to 0, then the CPU must record end time (endTime) of that process, as well as the wait time (waitTime) of that process (explained below). The simulator must hence maintain a counter (i.e. static) to track the ***current time***. The current time is initiated to 0 at the very start of all operations, and incremented by 1 each time a job is inserted in the queue from the array, each time a job is executed, and each time an iteration is made to search for the first starved process. The timer however is NOT incremented when the priority of that starved process is finally modified or for the time it took to update the priority queue (i.e. we choose to ignore that overhead). It is very important to notice that the current time is measured in CPU cycles (not actual time). It is also very important to notice that this current time is consequently for time simulation and it does not represent the <u>actual system time (which is measured in µsec, ms, etc.)</u>. Your experiments will track that actual time as well, as explained below. current time初始是0，然后每个job被插入到pq，被进行，还有搜寻第一个starve的时候，都要加1，改变starve本身不加1，你的程序也应该track真实时间

如果当前Length等于0，cpu必须记录endtime，和waitTime,模拟器因此必须有个counter来跟踪current time，

- ⇨ The wait time of a job is calculated as the total time that the job needed to wait from the time it entered the queue until it finally terminated. You should notice that this wait time does NOT include the execution time (as the process was not waiting during that time!). For instance, assume that J56 has job length of 68, and was entered the queue at time 56, then terminated at time 3478. The wait time of that job is hence calculated as 3354 (that is: 3478 – 56 – 68).

wait time等于从被最终结束到进入q，wait time不包含进行时，因为他不在wait， 比如J56进入的时候是56，走的时候是3478，还应该减去68因为有68个time对于别人来说在wait，对于他自己来说没被wait

- ⇨ Finally, once all jobs in the queue are terminated (when the queue is empty), the simulator must record (to an output file as detailed below) a final performance report indicating final current system time (in cycles), how many jobs have been executed, average waiting time for all processes, and how many times priority had to be changed to avoid starvation. Additionally, the simulator must record the actual running system time the was spent to execute all processes. For that, you must record the system actual time at the very start (just before the jobs in the array are entered into the queue, as well as the actual system time once the execution is terminated for these processes. You can then calculate the actual amount of spent time). The report should look as follows:

最终，一旦所有Job结束，模拟器必须记录final report，写着final current system time，解决了几个job，所有过程的平均waiting time 改变了几次priority，另外，必须从一开始就记录真正时间

Current system time (cycles): 6239854
Total number of jobs executed: 100000 jobs
Average process waiting time: 2046204.3 cycles
Total number of priority chnages: 17944
Actual system time needed to execute all hobs: 682.35 ms

Part I: ADT & Implementation

In this part, you need to define the ADT and its implementation according to the following:

1. The **Job** class. The attributes of this class correspond the above description.
2. At least two priority queues one linear and one non-linear (a bonus mark will be given for implementing all 4):
   - Unsorted List, OR
   - Sorted List

     AND
   - Linked-List-based Heap, OR
   - Array-List-based Heap.

For simplicity, your priority queues can be implemented to only accept entries from the **Job** class; however, a bonus mark will be given if you implement them as generics.

Part II: Test Simulator

生成

Write a driver class called **PriorityQueueSimulatorTester**. In that class, generate an array called **jobsInputArray** of size **maxNumberOfJobs**. Use your two priority queues implementation from part I (or the 4 them if you chose to implement all of them) to run your simulator with **maxNumberOfJobs** = {100, 1000, 10000, 100000, 1000000}.

The array **jobsInputArray** holds a set of jobs (each index has an entry of type Job). When you create your job objects to fill the array, the following must be followed:

- ⇨ The **jobName** is composed of the word "**JOB_**" and the *jobNumber*. which is the index of the array where this job is inserted + 1. For instance, the entry inserted at index 0 will be set as Job_1, and the entry inserted at index 285 will be Job_286.

- ⇨ The **jobLength** must be initialed at random integer value between 1 and 70 inclusive. The currentJobLength must then be initialed to this same value.

- ⇨ The **jobPriority** must be initialed at random to an integer value between 1 (highest priority) and 40 (lowest priority), inclusive. The **finalPriority** must then be initialed to this same value.

- ⇨ The **enteryTime, waitTime, endTime** must all be initialized to zero. However, these values are updated either when the entry is inserted in the queue (entryTime as described above), or when the process terminates (endTime, and waitTime).

- ⇨ Finally, record the performance report <u>for each value</u> of **MaxNumberOfJobs** and for all two (or four) types of your priority queues. This should be something similar to the following (the timing values below are just shown for illustration and they do not reflect your actual experiments):

Here are the results when **MaxNumberOfJobs** is set to 100,000

1) Unsorted List Priority Queue

   Current system time (cycles): 6239854
   Total number of jobs executed: 100000 jobs
   Average process waiting time: 2046204.3 cycles
   Total number of priority chnages: 17944
   Actual system time needed to execute all hobs: 682.35 ms

Sorted List Priority Queue

   Current system time (cycles): 6239854
   Total number of jobs executed: 100000 jobs
   Average process waiting time: 2046204.3 cycles
   Total number of priority chnages: 17944
   Actual system time needed to execute all hobs: 682.35 ms

2) Pointer-based Heap Priority Queue

   Current system time (cycles): 4800120
   Total number of jobs executed: 100000 jobs
   Average process waiting time: 876212,45 cycles
   Total number of priority chnages: 17944
   Actual system time needed to execute all hobs: 164.77 ms

3) Vector-based Heap Priority Queue

   Current system time (cycles): 4800120
   Total number of jobs executed: 100000 jobs
   Average process waiting time: 876212,45 cycles
   Total number of priority chnages: 17944
   Actual system time needed to execute all hobs: 164.77 ms

Again, the above reports need to be shown for all values of MaxNumberOfJobs = 100; MaxNumberOfJobs = 1000; …. etc.

⇨ Save the result of your program execution in a file called *SimulatorPerformanceResults.txt* and submit it together with your other files (see below).

<u>Important Requirements :</u>

1. Make sure you use the same jobsInputArray for all four types of priority queues for each value of MaxNumberOfJobs to ensure that we are comparing the performance of each implementation against the same set of input data.
2. Reset the current time at the start of each experiment (i.e. when you start with a new array, or with another priority queue).
3. Reset the actual system time readings at the start of each experiments, so you actually track only the actual system time needed to run this particular experiment.

4. In case the operations for big N numbers take too long (e.g., more than 120s of actual system time) you may reduce the number to a smaller one or eliminate it (so that you will have a range only from, say, 100 to 1000000).
5. **Do not use any java abstract data type or packages when writing your priority queues.** You must implement your own list or heap queues.
6. Your code should, reasonably, handle boundary cases and error conditions. It is also imperative that you test your classes and all parts of your implementation.
7. For this programming questions, you are required to submit the commented Java source files, the compiled files (.class files), and the test run text files.

Part III: Analysis

- ⇨ What is the Big-O (O*(n)*) and Big-Omega (Ω*(n)*) time complexity for each of the implemented priority queues in terms of MaxNumberOfJobs? explain.
- ⇨ What is the space complexity of each of the implemented priority queues in terms of MaxNumberOfJobs? explain.
- ⇨ Is there a performance difference between the different implementations? Is the difference significant (i.e. in terms of increased % of time)? If so, explain why; if not, explain why in the end all these implementations produce comparable results.
- ⇨ Include all these explanations (**be clear but very brief**) in a text or doc file called *SimulatorComplexity.txt* and included it with the other files of your submissions

Submission Guidelines

- The **written part** must be done **individually** (no groups are permitted). The **programming part** can be done in groups of **two** students **maximum**.
- For the written part, submit all your answers in PDF. You need to be concise and brief for each question. Submit the theory part of the assignment under **Theory_A#3** Dropbox on Moodle or "**Theory Assignment 3**" via EAS.
- For the programming part: you must submit the Java programs as the source files together with the compiled files as well as any required associated files. The solutions to all the questions should be zipped together into one .zip or .tar.gz file and submitted via Moodle under **Programming_A#3** Dropbox for Moodle submission or "**Programming Assignment 3**" via EAS. You must upload at most one file. That is, if you are working in a group of 2 students, only one student will submit the programming part. Do not upload 2 copies.
  - o Create **one** zip file, containing the necessary files (.java, .html, etc.). Please name your file following this convention:
    - ➢ If the work is done by 1 student: your file should be called *a#_studentID*, where # is the number of the assignment *studentID* is your student ID number.
    - ➢ If the work is done by 2 students: The zip file should be called *a#_studentID1_studentID2*, where # is the number of the assignment *studentID1* and *studentID2* are the student ID numbers of each student.

Important Notes

- Again, the assignment must be submitted in the right DropBox for Moodle submission and right Folder for EAS submission. **Assignments uploaded to an incorrect DropBox/Folder will result in a zero mark. No resubmissions will be allowed.**
- For the programming part of the assignment, a demo is required (please refer to the course outline for full details). The marker will inform you about the demo times. **Please notice that failing to demo your assignment will result in zero mark regardless of your submission.** If working in a team, both members of the team must be present during the demo.