COMP 352: Data Structure and Algorithms
Fall 2020

Assignment # 3

Due date and time: November 13th 11:59 PM

Department of Computer Science and Software Engineering
Concordia University

Part 1: Written Questions (50 marks):

Question 1

a) Given a tree T, where *n* is the number of nodes of *T*. Give an algorithm for computing the depths of all the nodes of a tree T. What is the complexity of your algorithm in terms of Big-O?

b) We say that a node in a binary search tree is full if it has both a left and a right child.
Write an algorithm called *Count-Full-Nodes(t)* that takes a binary search tree rooted at `node t`, and returns the number of full nodes in the tree. What is the complexity of your solution?

Question 2

a) Draw the min-heap that results from the bottom-up heap construction algorithm on the following list of values:
20, 12, 35, 19, 7, 10, 15, 24, 16, 39, 5, 19, 11, 3, 27.

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.

b) Create again a min-heap using the list of values from the above part (a) 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 20, then insert 12, then 35, etc.) as shown in the above question. Show the tree after each step and the final tree representing the min-heap.

Question 3

Assume a hash table utilizes an array of 13 elements and that collisions are handled by separate chaining. Considering the hash function is defined as: *h(k)=k mod 13*.

i) Draw the contents of the table after inserting elements with the following keys:

32, 147, 265, 195, 207, 180, 21, 16, 189, 202, 91, 94, 162, 75, 37, 77, 81, 48.

ii) What is the maximum number of collisions caused by the above insertions?

Question 4

To reduce the maximum number of collisions in the hash table described in Question 3 above, someone proposed the use of a larger array of 15 elements (that is roughly 15% bigger) and of course modifying the hash function to: *h(k)=k mod 15*. The idea is to reduce the *load factor* and hence the number of collisions.

Does this proposal hold any validity to it? If yes, indicate why such modifications would actually reduce the number of collisions. If no, indicate clearly the reasons you believe/think that such proposal is senseless.

Question 5

Assume an *open addressing* hash table implementation, where the size of the array is $N = 19$, and that *double hashing* is performed for collision handling. The second hash function is defined as:

$$d(k) = q - k \bmod q,$$

where k is the key being inserted in the table and the prime number q is = 7. Use simple modular operation ($k \bmod N$) for the first hash function.

i) Show the content of the table after performing the following operations, in order:
put(25), put(12), put(42), put(31), put(35), put(39), remove(31), put(48), remove(25), put(18), put(29), put(29), put(35).

ii) What is the size of the longest cluster caused by the above insertions?

iii) What is the number of occurred collisions as a result of the above operations?

iv) What is the current value of the table's *load factor*?

Question 6

Draw a single binary tree that gave the following traversals:

Inorder: T N C K V A S M W Q B R L
Postorder: T C N V S A W M B Q L R K

Programming Questions (50 marks):

In this programming question, you need to build a program that simulates CPU (Central Processing Unit) scheduling for executing processes/jobs (these names will be used interchangeably 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 entered 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 priority 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.

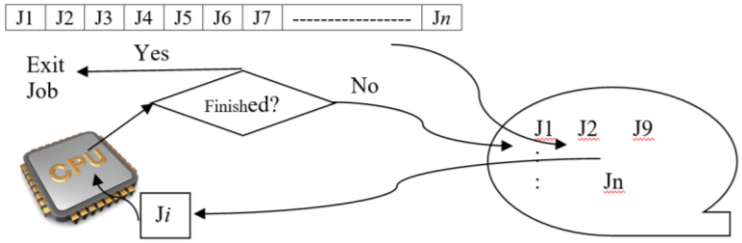

Figure 1: Basic Simulation of the System

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 termination 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 incur from the time it entered the queue until it terminates. All jobs must first be inserted in the priority queue. Each insertion 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 inserted, its entryTime must be set as 6, since this is how much it took before this job is inserted, and so on.

Each job has a length (jobLength), which indicates how much time/CPU cycles the job needs to be allocated 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 priority (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.

- ⇨ 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 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.

  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.

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:

Now executing Job_285. Job length: 42 cycles; Current remaing length: 26 cycles; Initial priority: 22; Current priority: 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 MUST 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 MUST guarantee that this behavior is respected. **If needed, you are allowed to add additional attributes to the Job class.**

⇨ 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 actual system time (which is measured in μsec, ms, etc.). Your experiments will track that actual time as well, as explained below.

⇨ 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).

⇨ 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 that 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:

Current system time (cycles): 6239854
Total number of jobs executed: 100000 jobs
Average process waiting time: 2046204.3 cycles
Total number of priority changes: 17944
Actual system time needed to execute all jobs: 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 to 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):
    - o Unsorted List, OR
    - o Sorted List

    AND
    - o Linked-List-based Heap, OR
    - o 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**

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 of 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 to a random integer value between 1 and 70 inclusive. The currentJobLength must then be initialed to this same value.

- ⇨ The **jobPriority** must be initialed to a random 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 for each value 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 changes: 17944
    Actual system time needed to execute all jobs: 682.35 ms

2) 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 changes: 17944
Actual system time needed to execute all jobs: 682.35 ms

3) 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 changes: 17944
Actual system time needed to execute all jobs: 164.77 ms

4) 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 changes: 17944
Actual system time needed to execute all jobs: 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).

Important Requirements:

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

- ⇨ 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 questions, submit all your answers in PDF (no scans of handwriting; this will result in your answer being discarded) or text formats only. Please be concise and brief (less than ¼ of a page for each question) in your answers. Submit the assignment under Theory Assignment 3 directory in EAS or the correct Dropbox/folder in Moodle (depending on your section).**
- **For the Java programs, you must submit the source files together with the compiled files. The solutions to all the questions should be zipped together into one .zip or .tar.gz file and submitted via Moodle/EAS under Programming 3 directory or under the correct Dropbox/folder. You must upload at most one file (even if working in a team; please read below). In specific, here is what you need to do:**
- 1) Create **one** zip file, containing the necessary files (.java and .html). 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.
- 2) If working in a group, only one of the team members can submit the programming part. Do not upload 2 copies.

Very Important: Again, the assignment must be submitted in the right folder of the assignments. Depending on your section, you will either upload to Moodle or EAS (your instructor will indicate which one to use). **Assignments uploaded to an incorrect folder will not be marked and result in a zero mark. No resubmissions will be allowed.**

- ⇨ Additionally, 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.