CS2020: Data Structures and Algorithms (Accelerated)

Problems 8–11

Due: February 16th, 13:59

Overview. This week's problem set focuses on heaps. Lots and lots of heaps.

Collaboration Policy. As always, you are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 8. (Updating your Heap.)

Sometimes, you need to change the priority of an item that has been previously inserted into a priority queue. For example, a task was initially inserted at low priority, but some event occurs which increases the priority of the task.

Problem 8.a. Please extend the 'Heap.java' implement (written by Steven Halim and used in Lecture 6) by adding a new public method: `Heap_UpdateKey(i, k)` that will update the key stored at `A[i]` in the heap, setting its new value to `A[i] = k`. This modification may either increase, decrease, or not change the value of `A[i]`. Note that in order to maintain the heap-order property, the heap may need to be modified after the value of the key is changed. Verify that your function works for all valid values of `i` $\in$ `[1..heapsize]`.

Figure 1 contains test code that you should use to verify your implementation.(Note: you may use the main method from Figure 1, or your own.) Submit an updated version of 'Heap.java' along with your tester code in 'HeapTest1.java'.

Problem 8.b. What is the asymptotic complexity of your implementation of `Heap_UpdateKey`?

```
public static void main(String[] args) {
     int[] array = new int[]{19, 3, 17, 1, 2, 25, 7, 36, 100};
     Heap h = new Heap();
     h.Build_Heap(array);
     h.Heap_UpdateKey(1, 21); // 'decrease' max element (at index 1) from 100 to 21
     System.out.println(h.Heap_ExtractMax()); // must be 36, not 100 anymore
     System.out.println(h.Heap_ExtractMax()); // must be 25
     System.out.println(h.Heap_ExtractMax()); // must be 21 (which was originally 100)
     System.out.println(h.Heap_ExtractMax()); // must be 19

     System.out.println("-------------");
     h.Build_Heap(array); // restore the heap with the original content of 'array'
     h.Heap_UpdateKey(6, 77); // 'increase' element at index 6 (previously 17) to 77
     System.out.println(h.Heap_ExtractMax()); // must be 100 (unchanged)
     System.out.println(h.Heap_ExtractMax()); // must be 77 (which was originally 17)
     System.out.println(h.Heap_ExtractMax()); // must be 36
     System.out.println(h.Heap_ExtractMax()); // must be 25
}
```

Figure 1: Code for testing `Heap_UpdateKey`.

Problem 9. (MaxHeap, MinHeap)

Sam Stacker has implemented a very efficient and highly optimized MaxHeap (for integer keys). He has completed the assignment on time, and is about to submit his code to the SuperSpeedy Rapid-Fire Pile Competition. At the last minute, however, he re-reads the instructions: he was supposed to implement a *MinHeap*, not a *MaxHeap*!

Recall that a *MaxHeap* allows you to extract the largest element in the heap via the `Heap_ExtractMax` function. Sam was supposed to implement a data structure in which you could extract the *smallest* element.

Sam was about to go back to work on re-engineering his heap implementation, when his friend Lisa Lister pointed out that there was no need to modify his existing code. Instead, it is possible to design a MinHeap using a MaxHeap.

Without modifying any code of the MaxHeap written in 'Heap.java', show a simple way of implementing a MinHeap. (In 'Heap.java,' you may change the modifiers from *private* to *protected*, if needed. Note that there is a solution which does not require this change.) Your MinHeap implementation should support the usual heap operations: `Heap_Insert`, `Heap_ExtractMin`, and `Build_Heap`. Submit your answer as 'MinHeap.java' along with your tester code in 'HeapTest2.java'.

Problem 10. (Java Priority Queue.)

In the future, you do not have to use Steven's 'Max Heap' to solve problems that requires a MaxHeap data structure (or for problems that require a MinHeap data structure—see problem 9). As part of its standard libraries, Java provides a `PriorityQueue` implementation that can be used whereever you need a heap. In this problem, we will experiment with this Java library.

Problem 10.a. You can find the documentation on the Java PriorityQueue at:
`http://download.oracle.com/javase/1.5.0/docs/api/java/util/PriorityQueue.html`
What is the default behavior of the Java PriorityQueue: is it a MaxHeap or a MinHeap?

Problem 10.b. Write a Java method to perform `Heap_Sort` using the Java `PriorityQueue`. In addition, write a simple `main` procedure to test your HeapSort routine. Save (and submit) your solution as 'HeapSort.java'.

Problem 11. (Analyzing FastData Data Streams Fast.)

Freddy Fast works for `FastData.com`, a website dedicated to providing data as fast as possible. He hired an engineer to implement a super-fast web server, and after many months of work, the engineer builds a server that delivers web pages as fast as possible.

Freddy wants to monitor how fast the web server is working. He has hired you to build a monitoring service. Every time the server delivers a web page, it outputs the latency for loading the page. That is, it will execute the following function:

```
DataMonitor.addData(int time)
```

where the parameter `time` indicates how long it took for the web page to load. The monitoring program should store this data for future analysis. Every so often, Freddy will execute the following function:

```
DataMonitor.queryMedianData()
```

which should return the median time for a web page to load. Recall that the median for a set of data is the "middle" value. Formally, given a set of values $v_1, v_1, \ldots, v_n$, we define the **median** to be a value v that is larger than or equal to exactly $\lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$ elements. (Notice that when n is even, there are two possible median values. If n is odd, however, there is only one possible median value. For this problem, if there are two possible median values, we will always use the larger of the two values.) Freddy plans to use the median page-load time to analyze the performance of his webserver.

Freddy had previously implemented a version of the monitoring service where the data was stored in a big array, and whenever he queried the median, it simply sorted the data and returned the median. Unfortunately, as his web site grew more popular, his queries became very slow. Even with a fast implementation of QuickSort, the `queryMedianData` routine took time $O(n \log n)$. Your job is to develop a version in which `queryMedianData` runs in $O(1)$ time. However, `addData` must still be as efficient as possible in order to handle the incoming requests in real time.

Freddy suggests that you implement the monitoring service as a binary search tree. However the dataset is so large that you prefer a solution that makes more efficient use of memory. If all the data can be stored in a large array (without pointers), that will minimize the need to do an expensive (and slow) memory allocation for every `addData` operation. Instead, you propose the following idea, based on heaps, which are very efficient in their memory usage.

The monitoring system will contain two heaps: a MaxHeap A and a MinHeap B. It will maintain the following two properties:

1. Every element in A is no greater than every element in B.
2. The size of A equals the size of B, or is one less.

To implement `queryMedianData`, simply return the minimum element of B.

Problem 11.a. Argue that this is correct, i.e., that a median value is returned.

Problem 11.b. Implement the DataMonitor service. It should contain a constructor which creates an empty object, and it should support the `addData` and `queryMedianData` functionality.

For this problem, you may use Steven Halim's heap implementation (from Lecture 6), or you may use the standard Java PriorityQueue, or some combination of the two. *Note:* For this problem, you may assume that every element in the heap is distinct. For bonus credit, your data structure should work even if there are repeated elements inserted.

In addition, be sure to write tester code to verify that your solution works as expected. Submit your solution as 'DynamicMedian.java' and your tester code as 'MedianTester.java'.

Problem 11.c. Freddy is excited by your solution, and proposes the following optimization: instead of storing the entire dataset, why not just save 10 elements bigger than the median and 10 elements smaller than the median? Each of the two heaps will then only have 10 elements, which will be very fast! Whenever a new item is inserted, either the largest element in B or the smallest element in A will be deleted. You immediately realize that this won't work. Explain the problem with Freddy's solution, and give an example execution in which the wrong median is returned.

Problem 11.d. After using the monitoring system for a while, Freddy decides it would be useful to also know the maximum latency of his webserver. He asks you to implement the following additional functionality:

```
queryMaxData()
```

which returns the largest value ever inserted. Modify your previous solution in the simplest way possible to implement `queryMaxData` so that each query takes only $O(1)$ time. Submit your revised solution as 'DynamicMedian2.java' and your revised tester code as 'MedianTester2.java'.