CS3460: Data Structures
Lab 06: Priority Queues
Total Points: 25

Problems

Notes on Grading: Unless otherwise stated, all programs will receive input via `System.in` and will output solutions via `System.out`.

To simplify the grading process, all grading will be automated. When applicable, you will be provided with sample input/output files for testing. You can ensure that your program will receive full marks by testing it with these provided files.

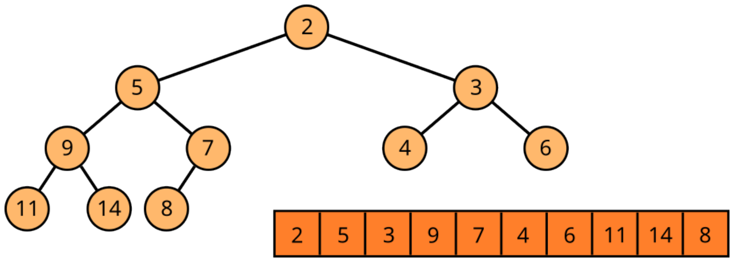
```
$ java YourProgram < input.txt > output.txt
$ diff output.txt correct.txt
```

The first line executes the Java program, redirecting input from a file `input.txt` and writing the output to a file `output.txt`. The second line compares your program's output (now stored in `output.txt`) with the correct answer (stored in `correct.txt`). If these files match exactly, the `diff` program will print nothing. Otherwise, it will list the differences.

Important Note: You are not allowed to use any classes or code from the Java Collections library. While the classes defined in that library would not be an ideal fit for most of our tasks, the purpose of these assignments is to build these data structures from first principles. Programs which import any of these libraries will receive zero points.

Submission: Please submit any `.java` files necessary to compile/run your project. Please do not include any other files (such as input/output files).

1. **Building a Binary Heap (15 points):** A *binary heap* is an almost-complete binary tree that satisfies the heap property, which is that the value stored at every node greater than or equal to the value stored in its parent, or key(parent(e)) $\leq$ key(e). All levels are filled to capacity before moving to the next level, and each level is filled from left to right. While this isn't necessary for the heap property to be maintained, this added constraint allows us to embed our tree into an array.



An array embedding of a binary heap.

You will implement a binary min-heap (meaning lower key = higher priority) in a file called `BinaryHeap.java`. This class should provide the following operations:

(a) `public void insert(int key)` : insert an integer `key`.

(b) `public int remove_min()` : remove the minimum key and return it.

(c) `public int find_min()` : return the minimum key (without removing).

(d) `public int size()` : return how many elements are in the priority queue

To implement these methods, you will want to create methods for `sift_up(i)` and `sift_down(i)` to maintain the heap property. With these, we can implement *insert* and *remove-min* by

(a) `public void insert(int key)` : placing element at the end of the array (may need to resize the array), then `sift_up(n)` .

(b) `public int remove_min()` : swapping the minimum (at $i = 0$) with the last element of the array, then `sift_down(0)` .

Create a driver to test these operations thoroughly for correctness and efficiency.

2. **Online Median Finding (10 points):** The *median* of a collection of values is the middle value in a sorted output. This simple definition for the median gives a straightforward way to find it: sort the list and look at the value at index $n/2$. This takes $O(n \lg n)$. There are other algorithms for finding the median, such as *quickselect*, but sorting is easy!

These solutions, however, assume that you have the entire list. What if you knew you needed to be able to find the median, while new values were potentially arriving? How would you do that?

Incidentally, another way to define the median of a dataset is as the value m for which 50% of the values are less than m and 50% of the values are greater than m. This suggests another possible solution for finding the median: What if we maintained two priority queues, one with the **large** half of the dataset, and one with the **small** half of the dataset? If these priority queues were balanced, the median would either be the max of **small** or the min of **large**.

Unfortunately, we implemented a min-heap, which won't give us what we are looking for in **small**. Really, the **small** values should be using a max-heap. How can we use a min-heap as a max-heap? Store the inverse (negative) of every value.

Therefore, the strategy looks like this:

(a) When a new element arrives, choose whether to insert into **large** or **small** by calling *find-min* on **large** to decide.

(b) If the priority queues become imbalanced as a result (their sizes differ by 2), move an element from the one with more elements to the one with fewer elements (keep in mind that one is storing negative values and one is storing positive values). This will restore the balance.

(c) When asked for the median, *find-min* on the priority queue with more elements. If they are the same size, *find-min* on both of them and compute the arithmetic mean on the two values.

In this problem, the input is considered to be "online", meaning you will be answering queries about the data before all of it has arrived. Each line of input will begin with either an `i`, indicating a value that needs to be *insert*ed, or `q`, indicating a query for the current median value. Each time a `q` command is received, the current median should be printed to standard output. Further `i` commands may follow, however. A single `e` command will represent the end of input and that no further values need to be inserted, nor are further queries arriving.

Each line will look like the following:

- `i 4` – insert the value 4. Each `i` command will be followed by an integer that is to be inserted into the dataset.
- `q` – print the current median of the dataset. If there are an odd number of values in the dataset, the median is a single unique value. If there are an even number of values, the median is the arithmetic mean of the two middle values.

- `e` – end of input