COL106 : Data Structures and Algorithms, Semester II 2024-25
Practice Programming Questions on Priority Queues & Heaps

February 2025

Instructions

- Please use the following questions as practice questions for learning about Priority Queues and Heaps.
- The questions with * next to them should be attempted during the lab sessions, and your solutions must be uploaded on moodlenew. Note that your submissions will not be evaluated, but will be used as a mark of your attendance. We will filter out all the submissions that are not from the lab workstations. So do not use your laptops for submitting the programs.

1 Questions

1. *** Implement a Priority Queue from scratch**

   Implement all methods of the class signature for a priority queue below using heaps as taught in class. Avoid using LLMs for this problem :) since this is the first non trivial data structure implementation you have encountered in this course and so should try your best to complete this yourself.

   Also try to use this class in remaining problems instead of `java.utils.PriorityQueue` .

```java

public class PriorityQueue {

    /**
     * Constructor: Initializes an empty priority queue (max heap).
     * Time Complexity: O(1)
     */
    public PriorityQueue();

    /**
     * Inserts a new key into the heap
     * Time Complexity: O(log N)
     */
    public void insert(int key);

    /**
     * Extracts and returns the maximum element
     * Time Complexity: O(log N)
     * @return the maximum value in the heap
     * @throws IllegalStateException if the heap is empty
     */
    public int extractMax();

```

```java
    /**
     * Returns the maximum element  without removing it.
     * Time Complexity: O(1)
     * @return the maximum value in the heap
     * @throws IllegalStateException if the heap is empty
     */
    public int getMax();

    /**
     * Deletes a specific key from the heap and maintains the heap property.
     * Time Complexity: O(N) (for searching) + O(log N) (for deletion)
     * @param key the key to delete
     * @throws IllegalArgumentException if the key is not found
     */
    public void delete(int key);

    /**
     * Updates the value of a specific key and maintains the heap property.
     * Time Complexity: O(N) (for searching) + O(log N) (for updation)
     * @param oldKey the existing key in the heap
     * @param newKey the new value to update the key with
     * @throws IllegalArgumentException if the key is not found
     */
    public void updateKey(int oldKey, int newKey);

    /**
     * Builds a max heap from an unsorted array.
     * Time Complexity: O(N)
     * @param array an array of integers to build the heap from
     */
    public void buildHeap(int[] array);

}
```

2. *** Implement HeapSort from scratch**

   Implement a method `void heapSort(int arr[])` for performing heapsort inplace on an input array.

   Example:
   Input: arr = [5, 4, 3, 2, 1]
   Output: arr = [1, 2, 3, 4, 5]

   Time Complexity: $O(NlogN)$.

3. *** K Smallest Elements**

   Implement a method `ArrayList<Integer> kSmallestElements(int[] arr, int k)` to find the k smallest elements in an array of integers and return them (order of returned k elements can be arbitrary). $(n >= k >= 1)$

   Example:
   Input: arr = [3, 4, 5, 2, 10] , k = 3
   Output: [3, 2, 5]

   Time Complexity: $O(Nlogk)$, where N is the size of the array.

4. **Find the Median of an input stream**

   The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values

   Given an input stream of integers (in form of an array) `arr`, return an array `med` where $med_i = Median(arr[:i])$, where $arr[:i]$ is the subarray of elements upto index i.

   Implement a method with signature `ArrayList<Double> streamMed(int A[])`.

   Example:
   Input: arr = [5 ,6, 9, 10, 1, 2]
   Output: med = [5, 5.5, 6, 7.5, 6, 5.5]

   Time Complexity: $O(NlogN)$.

5. **K Weakest Rows in a Matrix**

   You are given an $m \times n$ binary matrix `mat` of `1`'s, representing soldiers, and `0`'s, representing civilians. The soldiers are positioned in front of the civilians, that is, all the `1`'s will appear to the left of all the `0`'s in each row.

   A row i is weaker than a row j if one of the following is true:

   - The number of soldiers in row i is less than the number of soldiers in row j.
   - Both rows have the same number of soldiers and $i < j$.

   Return the indices of the k weakest rows in the matrix ordered from weakest to strongest.

6. *** Sweet Cookie Magic**

   Your cookies don't taste as sweet as you want them to :(

   Suppose your cookies have varying sweetness. You want your cookies to have sweetness $\geq k$ units.

   Luckily, a fairy bestows a magical power upon you :)

   Mixing two cookies with the least sweetness yields a cookie having

   $$\text{sweetness} = (\text{sweetness of least sweet cookie} + 2 \times \text{sweetness of second-least sweet cookie})$$

   If the power is enough to yield sufficiently sweet cookies, that is, all cookies have sweetness $\geq k$, then return the minimum number of times the power has to be exploited, else return -1.

   Example:
   Input: `mat = [[1, 1, 1], [1, 1, 0], [0, 0, 0]], k = 2`
   Output: `[2, 1]`

7. **Sequentially Ordinal Rank Tracker**

   A location is represented by its `name` and magnificence `score`, where `name` is a unique string among all locations and `score` is an integer. Thus, locations can be ranked from the best to the worst: the higher the score, the better the location. If the scores of two locations are equal, then the location with the "lexicographically smaller" name is better.

   You are building a system that tracks the ranking of locations with the system initially starting with no locations. Therefore, implement the `SORTracker` class such that:

   - `SORTracker()` initializes the tracker system.
   - `void add(string name, int score)` adds a location with `name` and `score` to the system in $\mathcal{O}(\log n)$ time complexity, if there are n stored locations.

- `String get()` queries and returns the ith best location in $\mathcal{O}(\log n)$ time complexity, where i is the number of times this method has been invoked (including the current invocation). Assume that $i <$ number of stored locations $= n$.

8. **Most Used Meeting Room**

   There are n rooms **numbered** from 0 to $n-1$. You are given a 2D integer array `meetings` where `meetings[`i`] = [start`$_i$`, end`$_i$`]` means that a meeting will be held during the half-closed time interval [`start`$_i$, `end`$_i$) such that `start`$_i$ is unique $\forall\ i$. The elements of `meetings` are sorted in increasing order w.r.t. the starting times `start`$_i$.

   Meetings are allocated to rooms in the following manner:

   - Each meeting will take place in the unused room with the lowest number.
   - If there are no available rooms, the meeting will be delayed until a room becomes free. The delayed meeting should have the same duration as the original meeting.
   - When a room becomes unused, meetings that have an earlier original start time should be given the room.

   Return the **number** of the room that held the most meetings in $\mathcal{O}(k \log n)$ time complexity where k is the number of meetings. If there are multiple rooms, return the room with the lowest **number**.

   Example:
   Input: `meetings = [[1, 5], [2, 4], [3, 5]], n = 2`
   Output: `1`

9. *** Merge K Sorted Linked Lists**

   You are given an array of K sorted linked lists. Merge them into one linked list in $\mathcal{O}(N \log K)$ time complexity and $\mathcal{O}(K)$ space complexity using priority queues, where N is the total number of nodes.

   Input: `ListNode[] listHeads`
   Output: `ListNode sortedListHead`

10. **Minimise Average Waiting Time**

    You own a fast-food restaurant that can complete only one order at a time. A simple strategy to serve customers would be first-come, first-serve. However, this is not optimum w.r.t. the average waiting time of customers. A customer who ordered earlier but has a very long order completion time as compared to the one who ordered right after, if served earlier, unnecessarily increases the waiting time of the other customer.

    For example, consider the following scenario of customers A, B, C who ordered at $t = 0$, $t = 1$, $t = 2$, respectively. Let their order completion times be 4, 10, 5 units of time respectively. In case of first-come, first-serve, the waiting times are 4, 13 $(= 4 + 10 - 1)$, 17 $(= 4 + 10 + 5 - 2)$ units, respectively. The average waiting time is then 11.33 units. Now, consider serving C before B, after serving A. In this case, the waiting times are 4, 18 $(= 4 + 5 + 10 - 1)$, 7 $(= 4 + 5 - 2)$ units, respectively. The average waiting time is then 9.67 units. This strategy achieves minimum average waiting time, unlike first-come, first-serve, in this case.

    Suppose that the i^{th} customer orders at $t =$ `T`$_i$ and their order takes `L`$_i$ units of time to complete. Return the non-fractional part of minimum average waiting time achievable in $\mathcal{O}(n \log n)$ time complexity, where n is the number of customers.

    Input: `List<List<Integer>>`, where `List.get(`i`) = [T`$_i$`, L`$_i$`]`

    Output: `long t`, the non-fractional part of minimum average waiting time achievable.