

CSCI2100: Regular Exercise Set 8

Prepared by Yufei Tao

Problem 1. Prove: A tree with n nodes has $n - 1$ edges.

Solution. We will prove the claim by induction. The case of $n = 1$ is obviously true. Now suppose that the claim is true for all $n \leq k$, we now proceed to prove that it is also true for $n = k + 1$. Let T be a tree with $k + 1$ nodes. Remove an arbitrary edge of T . The fact that the original T has no cycle implies that now T has been divided into two trees T_1 and T_2 . If T_1 has x nodes, then T_2 has $k + 1 - x$ nodes. Our inductive claim shows that T_1 has $x - 1$ edges and T_2 has $k + 1 - x - 1 = k - x$ edges. Therefore, T has $(x - 1) + (k - x) + 1 = k$ edges. This completes the proof.

Problem 2 (Max Heap). The binary heap we discussed in the class is called the *min-heap* because of the `delete-min` operation. Conversely, a *max-heap* on a set S of integers aims to support insertions and the following `delete-max` operation:

- **Delete-max:** Reports the largest integer in S , and removes it from S .

Describe how a min-heap can be used to implement a max-heap *without* changing its structure and algorithms. Your max-heap must still use $O(|S|)$ space, and support an insertion and a delete-max operation in $O(\log |S|)$ time.

Solution. To perform an insertion of e , simply insert $-e$ to a min-heap. To perform a delete-max, simply perform a delete-min from the min-heap, and then return the fetched value after negating it.

Problem 3* (Priority Queue with Attrition). Let S be a dynamic set of integers. At the beginning S is empty. We want to support the following operations:

- **Insert-with-Attrition(e):** First removes all integers in S that are greater than e , and then adds e to S .
- **Delete-Min:** Removes and returns the smallest integer of S .

For example, suppose we perform the following sequence of operations:

1. `Insert-with-Attrition(83)`
2. `Insert-with-Attrition(5)`
3. `Insert-with-Attrition(10)`
4. `Insert-with-Attrition(15)`
5. `Insert-with-Attrition(12)`
6. `Delete-Min`
7. `Delete-Min`

After Operation 3, $S = \{5, 10\}$ (note that 83 has been deleted by Operation 2). After Operation 5, $S = \{5, 10, 12\}$. After Operation 6, $S = \{10, 12\}$.

Describe a data structure with the following guarantees:

- At all times, the space consumption is $O(|S|)$.

- Any sequence of n operations (each being an `insert-with-attrition` or `delete-min`) is processed with $O(n)$ time, i.e., $O(1)$ amortized time per operation.

Solution. We simply maintain all the elements of S in a queue Q , where they are arranged in the same order by which they enter S . Given an `Insert-with-Attrition(e)` operation, we keep walking back from the tail of Q until either seeing the first element smaller than e or having exhausted the entire Q . Delete all the elements that (i) are already seen, and (ii) are larger than e . It is important to observe that at this moment all the remaining elements in Q are sorted in ascending order.

To perform a `delete-min`, simply remove the first element of Q .

The cost of `delete-min` is clearly $O(1)$. The cost of `Insert-with-Attrition` equals $O(1 + x)$ where x is the number of elements removed. The total cost of all the `Insert-with-Attrition` operations is $O(n)$ because every element can contribute to the x -term only once.

Problem 4 (Textbook Exercise 6.5-9). Suppose that we have k arrays A_1, A_2, \dots, A_k of integers, such that each array has been sorted in ascending order. Let n be the total number of integers in those arrays. Describe an algorithm to produce an array that sorts all the n integers in ascending order (you may assume that no integer exists in two arrays). Your algorithm must finish in $O(n \log k)$ time.

For example, suppose that $k = 3$, and that the three arrays are $(2, 23, 32, 35, 37)$, $(5, 10)$, and $(33, 58, 82)$. Then you should produce an array containing $(2, 5, 10, 23, 32, 33, 35, 37, 58, 82)$.

Solution. Insert the smallest element of each array into a binary heap H . This takes $O(k \log k)$ time. Then, repeat the following until H is empty:

- Perform a `delete-min`. Let e be the element fetched.
- Append e to the output array.
- If e comes from A_i (for some i), obtain the next element from A_i , and insert it into H . If A_i has been exhausted, then do nothing.

Each `delete-min` and insertion require $O(\log k)$ time because H has at most k elements. There are n `delete-min` and n insertions. So the total cost is $O(n \log k)$.