Homework 4

Due: Friday, Sept 26, 11:59 PM PT

1. Design a data structure that has the following properties (assume n elements in the data structure, and that the data structure properties need to be preserved at the end of each operation):
    a. Find-median takes O(1) time
    b. Extract-Median takes O(log n) time
    c. Insert takes O(log n) time

    Do the following:
    a) Describe how your data structure is designed. (5 points)
    b) Give algorithms that implement the Extract-Median() and Insert() functions. (8 points)

    Solution:
    For general n, we use the ceiling(n/2) smallest elements to build a max-heap and use the remaining floor(n/2) elements to build a min-heap. The median will be at the root of the max-heap for odd n, and average of the two roots for even n, and hence accessible in time O(1).
    For extraction, if n is odd, we remove the root from the max_heap in O(log n), and if n is even, we just output the median value. (Note that the 'extract' operation is not applicable for even n by default, as the median is not an element itself - one can introduce a convention as applicable, such as "simply output median value for even n" or "extract middle two for even n" etc leading to slightly different implementations - this was not specified in the question).
    For inserting an element, we do the following for even n (i.e., after insertion, n will be odd). First, we compare it with the root of the min-heap: If it is smaller, we insert it in the max-heap. This is consistent with our design to have the max-heap have one extra element for odd n. If it is bigger, we add it to the min-heap, but along with that, we extract the root of the min-heap and add it to the max-heap.
    If n is odd, we compare the element with the root of the min_heap. If it is bigger than the root of the min_heap, we remove the root, add it to the max_heap, and add the new element to the min_heap. If it is smaller, then we add the new element to the max_heap. Since all insertion and extract_min can be done in O(log n) in a heap, this operation also takes O(log n) time.

    Rubrics:
    - 5 points for using two heaps for the smaller half and one for the bigger half.
    - 3 points for the correct Extract-Median.
    - 5 points for the correct Insert.

2. Given an undirected graph $G = (V, E)$ where a set of data centers V is connected by a network of optical links E. Each link (u, v) has a positive latency cost $l(u, v)$. There is a

proposal to add a new optical link to the network. The proposal specifies a list C of candidate pairs of data centers between which the new link may be established. Your task is to choose the link that yields the greatest reduction in end-to-end latency between a given source data center s and target data center t. Design an efficient algorithm for this problem. No proof is required. Give the runtime complexity of your algorithm. (Note that your algorithm's time complexity should not be worse than Dijkstra's shortest path algorithm.) (12 points)

Solution:

- Run Dijkstra's algorithm from s (using l as the edge weights) to compute $dist(s, x)$ for every $x \in V$.
- Run Dijkstra's algorithm from t (treating the graph as undirected or equivalently run on the same graph with weights l) to compute $dist(t, x)$ for every $x \in V$.
- For each candidate link $\{u, v\} \in C$ (with proposed latency $l(u, v)$), the best possible new latency between s and t that uses that link is $(dist(s, u) + l(u, v) + dist(t, v),\ dist(s, v) + l(u, v) + dist(t, u))$.
- Choose the link in C whose insertion achieves the smallest such new latency:
    - If this minimum exceeds the original $dist(s, t)$, then no candidate link can reduce the latency.
    - Otherwise, pick the $\{u, v\}$ that attains this minimum (breaking ties arbitrarily).

Complexity: $O((|V| + |E|) \log |V|)$ for the two Dijkstra runs, plus $O(|C|)$ to scan candidates, so in total $O((|V| + |E|) \log |V| + |C|)$.

Rubric:

- 2 points for running Dijkstra algorithm from s.
- 2 points for running Dijkstra from t.
- 3 points for calculating the shortest path between s and t given two candidate data centers (u and v).
- 3 points for considering all possible cases for calculated latencies.
- 2 points for time complexity. Since the time complexity of Dijkstra depends on its implementation details, any correct time complexity for Dijkstra is accepted.

3. A network of n servers under your supervision is modeled as an undirected graph $G = (V, E)$ where a vertex in the graph corresponds to a server in the network and an edge models a link between the two servers corresponding to its incident vertices. Assume G is connected. Each edge is labeled with a positive integer that represents the cost of maintaining the link it models. Further, there is one server (call its corresponding vertex as S) that is not reliable and likely to fail. Due to a budget cut, you decide to remove a subset of the links while still ensuring connectivity. That is, you decide to remove a subset of E so that the remaining graph is a spanning tree. Further, to ensure that the failure of S does not affect the rest of the network, you also require that S is connected to

exactly one other vertex in the remaining graph. Design an algorithm that, given G and the edge costs, efficiently decides if it is possible to remove a subset of E, such that the remaining graph is a spanning tree where S is connected to exactly one other vertex and (if possible) finds a solution that minimizes the sum of maintenance costs of the remaining edges. (10 points)

Solution:
First, we need to check the possibility of node S having only one neighbor in a spanning tree of the underlying graph. The best way to check this is to remove node S and all its adjacent edges to form a graph G′. If G′ is a connected graph, then we can claim it is possible to have a spanning tree where node S has only one neighbor. To check the connectivity of G′, the simplest way is to run a DFS or BFS algorithm on G′.
Considering that G′ is connected, we need to find the spanning tree that minimizes the maintenance cost. Therefore, we need to find the MST with the additional constraint that S should be a leaf. Therefore, we remove S and all its adjacent edges to form G′. We run Prime's (or any other MST algorithm) to find the MST of G′. Among all edges adjacent to S, we find the one with the minimum maintenance cost and connect S to the MST using this edge. The resulting graph will still be a spanning tree, and in this spanning tree, S will be a leaf.
Rubric:

- 5 points for making sure that the final MST is connected
- 5 points for building your spanning tree in a way that node S is a leaf

Ungraded Problems:

1. Imagine a dynamically resizing array that supports the following operations:
   - Insert(X): Adds X to the end of the array if it is not full. If full, first double the allocated memory of the array and then insert. The doubling operation takes time proportional to the current size of the array.
   - Delete(): Deletes the most recently inserted element, and halves the allocated memory of the array if the array becomes at most half filled as a result. The halving operation takes time proportional to the current size of the array. (If the current allocated memory is an odd number, the new memory is the floor of its half)

   Suppose the array begins at size = 1.
   a) For a sequence of n insert operations, show that the amortized cost of each operation is $\Theta(1)$. Use accounting methods.
   b) For a sequence of n insertions followed by n deletions, show that the amortized cost of each operation is $\Theta(1)$. Use an aggregating method.
   c) For arbitrary n, show a sequence of n Insert/Delete operations (in a suitable order) which have total running time $\Theta(n^2)$, making the amortized cost of the involved operations $\Theta(n)$ for this sequence.

   In all the parts, you can assume n to be a power of 2 to simplify your analysis if needed. (15 points)

   Solution:
   First, we observe that the doubling and halving of the memory-allocation happen when the number of elements in the array is a power of 2. Suppose the cost of simply adding and removing elements is 1, and for doubling/halving operations, it is the amount of memory that is allocated/deleted.

   a) Suppose we charge each insertion a cost of 3. 1 gets used for adding it to the array always, and 2 gets spared for future use. Suppose we consider 2^i insertions for some i. Then, the doubling operation has happened i + 1 times, incurring a total cost of $(2^0 + 2^1 + \ldots + 2^i = 2^{i+1} - 1)$ and the total cost saved up was (2 per insert) x (2^i inserts) = 2^{i+1}, thus the balance is 1. Hence, this shows that the balance is non-negative after any expensive operation (it must be at other points as a result).
   Thus, by the accounting method, the amortized cost is 3 = $\Theta(1)$.

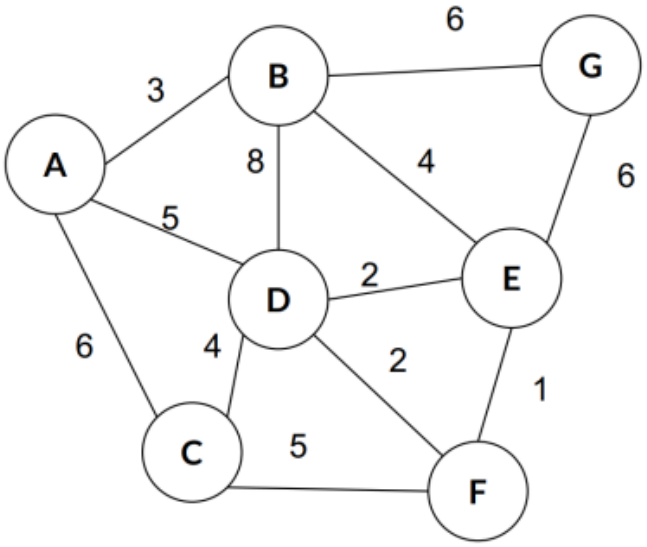
   b) Suppose $n = 2^k$. Then, across the n insertions, the cost of simply adding the elements is 2^k. There will be doubling operations whenever the array had 2^i elements for $i < k+1$, adding up to $(2^0 + 2^1 + \ldots + 2^k = 2^{k+1} - 1)$. As the n deletions follow, we will have the same amount of cost as well. In total, we have an aggregate cost of $\sim\sim 6.\ 2^k$ over $2n = 2.\ 2^k$ operations, thus an average cost of $\frac{6.2^k}{2.2^k} = 3$ per operation, which is the amortized cost.

c) Suppose $n = 2^k$. Note that we must have a sequence of operations where the doubling and halving operations must occur very frequently for the amortized cost to be much higher than the scenarios in part a) and b). To achieve this, we first have $n/2 = 2^{k-1}$ insert operations to have 2^{k-1} elements in the array. Now, the remaining n/2 operations are alternating insert and delete: each of these insertions leads to doubling since the array is full before insertion, the next deletion leads to halving since the array becomes half after deletion. Thus, each of the latter 2^{k-1} operations have an additional cost of 2^{k-1} each. In total, the cost of these operations comes out to $\sim 3 \cdot 2^{k-1}$ (for initial insertions) $+ 1 \cdot 2^{k-1}$ (for latter operations, but simply adding/removing elements) $+ 2^{k-1} \cdot 2^{k-1}$ (for doubling/halving in the latter operations). Averaged over all $n = 2^k$ operations, we get an amortized cost $2^{k-2} + 2 = n/4 + 2 = \Theta(n)$.

Rubrics:

- 5 points for each part

2. Considering the following graph G:



1. In graph G, if we use Kruskal's Algorithm to find the MST, what is the third edge added to the solution? Select all correct answers (3 points)

a. E-F b. D-E c. A-B d. C-F e. D-F

2. In graph G, if we use Prim's Algorithm to find MST starting at A, what is the second edge added to the solution? (3 points)

a. B-G b. B-E c. D-E d. A-D e. E-F

3. What is the cost of the MST in the Graph? (3 points)

a. 18 b. 19 c. 20 d. 21 e. 22

Solution:

1. c. A-B

2. b. B-E
3. c. 20

Rubrics:

- 3 points for each question

3. Prove or disprove the following:
T is a spanning tree on an undirected graph G = (V, E). Edge costs in G are NOT guaranteed to be unique. If every edge in T belongs to SOME minimum cost spanning trees in G, then T is itself a minimum cost spanning tree. (5 points)

Solution:
False. Counterexample:
Assume a graph G with three nodes a, b, and c and edges ab, bc, and ac. Where edges ab and bc have a weight of 2, and the edge ac has a weight of 1.

T = ab, bc (cost 4)
MST1 = ab, ac (cost 3)
MST2 = ac, bc (cost 3)
ab is in MST1
bc is in MST2
However, T is not an MST

Rubrics:

- 2 points for stating it is False.
- 3 points for the correct counterexample.

4. You are given a minimum spanning tree T in a graph $G = (V, E)$. Suppose we remove an edge from G to create a new graph G_1. Assume that G_1 is still connected, devise a linear time algorithm to find an MST in G_1. (8 points)

Solution:
Suppose the removed edge is named e_r.
Case 1: If e_r is not in T. The MST is T. This takes linear time.
Case 2: If e_r is in T. Once we delete an edge from T, the tree becomes disconnected. We need to find the minimum weight edge that connects two components, T_1 and T_2. Pick any component say T_2 and find all edges going to T_1. Among them, choose the one that has the smallest cost.
Runtime $O(E)$.

Rubric:

- 3 points for the first case
- 5 points for the second case

5. Imagine you are in a car and you want to travel on a road. You need to go up and down several hills on this road. Fortunately, your car is specially equipped with a nitrogen booster in addition to a regular gasoline burning engine. You can only use the booster for a limited number of times (k), and you have a certain amount of gas (m gallons) in your tank at the start of your journey.
   The car does not need to burn any gas or use the booster when you are coming down the hills or going straight on a flat road. But for each hill (with elevation gain of "h") that you want to go up, you need to use one of these two options:
      1) You can use the engine and burn 'h' gallons of gas, or
      2) You can spend one of your k special boosters (if you have any left). The booster can take the car from any height to any height.
   You are given an array of heights for the hills you need to pass hills[1..n], in the order in which you encounter the hills on your journey. Your job is to find the maximum number of hills that you can pass with the given amount of gas (m gallons) and the given number of boosters (k). In other words, you need to design an algorithm so that you can reach the furthest on this road by optimally using either your boosters or gas to climb up each hill. Your algorithm should run in O(n log k). Explain your algorithm. How much space does it take? (10 points)

   Solution:
   Intuition: We should allocate the boosters for the largest 'up-hills' and use the gas tank for the smaller ones.
   A simple approach is to start from the beginning of the array and use the boosters for the first K 'up-hills'. We will keep track of the 'up-hills' in a min-heap. After the Kth' up-hill, if we encounter an 'up-hill' that is larger than a previous 'up-hill', we would allocate gas for the previous one and use a booster for the new 'up-hill'. If we don't have enough gasoline left, it means that we have reached the maximum that we can achieve.
   Each exchange from the heap (extracting and inserting) takes O(log k) time (because the maximum number of elements in the heap is k). We are doing these operations for a maximum of 2*n time (a maximum of one time for putting into the heap and one time pulling it out of the heap). So the time complexity is O(nlog k).
   Space complexity: we are storing a heap of size K. So the space complexity is O(k)

   Rubric:
   - 4 points for creating min-heap
   - 4 points for updating the mean heap
   - 2 points for correct time and space complexity

6. Given an array of elements A[1..n], what is the tight bound on the performance of the following algorithm, which places the elements of array A into a binomial heap H? (8 points)

```
Initialize Heap H
For i=1 to n
        Insert A[i] into H
Endfor
```

Solution:
Need to find the amortized cost of Insert in a sequence of n Insert operations. This will be O(1).
The tight bound on the performance of the algorithm will be O(n)
Amortized cost analysis (Aggregate Analysis)
Observation: insert requires log n tree merge operations only if all log n trees are in the root list.
Number of insert operations that require no tree merges: n/2 (when the size of the heap is even (…..0 in binary))
Number of insert operations that require 1 tree merge: n/4 (when the size of the heap is 4k+1 (….01 in binary))
Number of insert operations that require 2 tree merges: n/8 (when size of the heap is 8k+3 (…..011 in binary))
…
Need to sum up: 0*n/2 + 1*n/4 + 2*n/8 + 3*n/16 + …
The same summation is shown in lecture notes for the bottom-up construction of binary heaps, which comes to O(n)

Amortized cost of insert will therefore be O(n)/n = O(1)

Rubric:

- 4 points for correct amortized cost
- 4 points for correct deduction on the tree merges