Homework 4

● Graded

Student

Abhishek Soundalgekar

Total Points

40 / 40 pts

Question 1

Data Structure **10** / 10 pts

+ 0 pts Incorrect

+ 3 pts Data structure

+ 4 pts Insert()

+ 3 pts Find-Median()

✔ **+ 10 pts** Correct!

Question 2

Grocery Store

10 / 10 pts

✔ **+ 10 pts** Full points

+ 0 pts Incorrect

+ 4 pts Algorithm

+ 3 pts Correctness argument

+ 3 pts Time Complexity.

Question 3

Unique

5 / 5 pts

– 5 pts Incorrect

✔ **– 0 pts** Correctness

Question 4

MST **10** / 10 pts

+ 0 pts Incorrect

✔ **+ 3 pts** Time Complexity

✔ **+ 4 pts** Algorithm

✔ **+ 3 pts** Time Complexity of BFS.

Question 5

Maximum Bandwidth

Resolved **5** / 5 pts

+ 5 pts Correctness

+ 0 pts Incorrect

✔ **+ 1 pt** **Change1:** Replace the min-priority queue with a max-priority queue

✔ **+ 1 pt** **Change 2:** Initialization *band[other_v] <= 0*

✔ **+ 1 pt** **Change 3:** Initialization *band[s]=+inf*

✔ **+ 2 pts** **Change 4:** sum of weights to minimum of weights

Regrade Request **Submitted on: Feb 20**

I believe there may have been a misunderstanding regarding the initialization steps in my solution.

On the second page, under Steps- in my first step I have initialized

bandwidth of other vertices as 0 and the bandwidth[s] = infinity, on the third page in another way of representing my steps, I have mentioned the same. The initialization matches the requirements stated.

I kindly request you to please review my solution.
Thank you very much for your time and consideration.
Name: Abhishek Soundalgekar
USC Id: 2089011000
Email: soundalg@usc.edu

Updated

Reviewed on: Feb 27

Question assigned to the following page: 1

CSCI 570 Spring 2025

Homework 4

Name : Abhishek Soundalgekar
USC ID : 2089011000

Question 1
Answer 1

→ Find Median takes $O(1)$ time.
→ Insert takes $O(\log n)$ time

* We maintain two heaps
→ Max-Heap : contains the $\lceil n/2 \rceil$ smallest elements. Its root is the largest of these, which by design is the median when the total number of elements is odd (or in the even element case, the $n/2^{th}$ smallest element).

→ Min Heap : contains the $\lceil n/2 \rceil$ largest elements. Its root is the smallest element in the upper half.

The heaps are kept balanced so that the max-heap has either the same number of elements as the min-heap or one extra element.

Question assigned to the following page: 1

Algorithm : Find-Median
we need to return the median value in O(1) time.

1. Determine the total number of elements :
1.1 Let total = (number of elements in max heap) + (number of elements in min heap)

2. Step to compute the median :
2.1 If total is even :
2.1.1 Average Definition : Return the average of the root of the max-heap and the root of the min-heap
2.1.2 Given specification : Return the root of the max-heap (which is defined as the $n/2^{th}$ smallest element)
2.2 If total is odd :
2.2.1 Since the max-heap is maintained with one extra element when the count is odd, return the root of the max-heap.

Since we are only returning the top or root elements this algorithms time complexity will be O(1).

Question assigned to the following page: 1

Algorithm for Insert (x)
→ Inserting a new element x while maintaining the heap balance in $O(\log n)$ time.

1. If the data structure is empty:
   1.1 Insert x into the max-heap
   1.2 Update the count for the max-heap
   1.3 Stop.

2. Current median determination:
   2.1 We will use the Find-Median algorithm written before to retrieve the current median value.

3. Place the new element into the appropriate heap:
   3.1 If x is less than or equal to the current median:
      3.1.1 Insert x into the max-heap
   3.2 Else:
      3.2.1 Insert x into the min-heap

Each heap insertion takes $O(\log n)$ time.

Question assigned to the following page: 1

4. Rebalance the heaps if required
4.1 If the max-heap has more than one extra element compared to the min-heap
(size(max-heap) > size(min-heap) + 1):
4.1.1 Remove the root from the max-heap (this is the largest of the lower half).
4.1.2 Insert the extracted element into the min-heap.
4.2 Else if the min-heap has more elements than the max-heap
(size(min-heap) > size(max-heap)):
4.2.1 Remove the root from the min-heap (this is the smallest of the upper half).
4.2.2 Insert the extracted element into the max-heap.
4.3 Each extraction & subsequent insertion takes $O(\log n)$ time.
5 End.

Question assigned to the following page: 1

Or we can have an insert() algorithm like :

1. Compare the number to be inserted with top (max-heap) and top (min-heap)
2. If num < top (min-heap) then insert at end of max-heap.
3. Else insert at end of min-heap.
4. After insertion arrange the elements by pushing them up or down to maintain binary heap properties.
5. If the len (min-heap) > len(max-heap) Insert root of min heap into max-heap and repeat step 4.
   This would give the len(min-heap) = len(max-heap) or len (max-heap)+(1)
6. Else if len (max-heap) > len (min-heap) + 1 then insert root of max-heap [top max-heap] into min-heap and repeat step 4 so that len (max-heap) == len (min-heap) + 1

Question assigned to the following page: 1

Time Complexities:

Find-Median: This operation takes O(1) time. Retrieves the median by simply examining the roots of the two heaps.

Insert: Places the new element in one of the two heaps (based on a comparison with the median) and then rebalances the heap if needed. Each insertion and rebalancing operation takes logarithmic time. O(log n).

Question assigned to the following page: 2

Question 2
Answer 2

* Algorithm to merge k sorted lanes into a single queue with O(n log k)

→ The optimal way is to use a min-heap (priority queue) to track the smallest available customer across all lanes. This ensures that at each step, the next customer with the earliest arrival time is selected, maintaining the sorted order in the merged queue.

1. Initialize a min-heap to store tuples:
   customer_arrival_time, lane_index, customer_index.
2. Insert the first customer from each lane into the heap. This ensures the heap initially contains the smallest element from each of the k lanes. If a lane is empty, skip it.
3. Build the merged queue:
   3.1 Extract the minimum element from the heap (the customer with the earliest arrival time).
   3.2 Add this customer to the merged queue.
   3.3 Insert the next customer from the same lane into the heap (if there are remaining customers in that lane).
   3.4 Repeat until the heap is empty.

Question assigned to the following page: 2

Correctness Justification

→ The algorithm correctly merges all customers from k sorted lanes into a single sorted queue.

1. Invariant Maintenance:
→ At every step, the min-heap contains the smallest unprocessed customer from the front of each lane.
→ Since all lanes are pre-sorted by arrival time, the next candidate for the merged queue must be one of the k front-most customers (one from each lane).

2. Initialization: Inserting first customer from each lane into min-heap ensures that heap initially contains all potential candidates for the earliest arrival time across all lanes

3. Extraction & Insertion cycle
→ Extracting the root of the min-heap guarantees that this customer is the next earliest to be added to the merged queue.

Question assigned to the following page: 2

After extraction, the next customer from the same lane (if available) is inserted into the heap. This ensures the heap always reflects the current front of all non-empty lanes.

4. Termination:

→ The process continues until the heap is empty, meaning all n customers have been added to the merged queue.

→ Since every extraction strictly follows the order of increasing arrival times, the merged queue is guaranteed to be sorted.

TIME COMPLEXITY

Algorithm achieves $O(n \log k)$ time complexity

1. Heap operations: Insertion & Extraction each take $O(\log k)$ time because the heap size is bounded by k (one entry per lane).

2. Total operations: Heap initialization: Inserting k elements takes $O(k \log k)$

Main loop: for all n customers:

n extractions $O(n \log k)$, n insertions (one per extraction) $O(n \log k)$

Total time = $O(k \log k) + O(n \log k)$. Since $k \leq n$ = $O(n \log k)$

Optimality: This complexity matches the lower bound $\Omega(n \log k)$. No algorithm can do better without leveraging additional structure.

Question assigned to the following page: 3

Question 3
Answer 3

* The Minimum-cost Road Network solution is Unique.

→ The town's goal is to build roads (edges) connecting all neighborhoods (nodes) at the lowest total cost, where each road has a unique construction cost. This is equivalent to finding the minimum spanning tree (MST) of a graph with distinct edge weights.

→ If all edge costs are unique, the MST is guaranteed to be unique.
This follows from fundamental properties of MSTs and the behaviour of algorithms like Prim's and Kruskal's when applied to graphs with distinct edge weights. Below is a detailed justification

Question assigned to the following page: 3

1. Role of Prim's and Kruskal's Algorithms.

Both algorithms construct an MST by greedily selecting edges in a way that avoids cycles and minimizes total cost.

→ Prim's Algorithm : Start at an arbitrary node and iteratively adds the smallest-weight edge adjacent to the current tree.

→ Kruskal's Algorithm : Sorts all edges by weight and add them in increasing order, skipping edges that create cycles.

* Reason for Producing the same MST when weights are unique:

→ With unique edge weights, there is no ambiguity in selecting the next edge. At every step, both algorithms will choose the same smallest available edge that does not form a cycle.

→ Thus both algorithms will lead to identical MSTs.

Question assigned to the following page: 3

Proof of Uniqueness by Contradiction

Let's assume there are two different MSTs, MST_1 and MST_2, with distinct edge sets.

1. Identify a Divergent Edge:
   → Let e be an edge in MST_1 but not in MST_2. Add e to MST_2. This creates a cycle C.

2. Cycle Property of MSTs:
   → In cycle C, there must be at least one edge f that is in MST_2 but not in MST_1.
   → Since all edge weights are unique, either $w(e) < w(f)$ or $w(e) > w(f)$.

3. Swap Edges to Derive a Contradiction:
   Case 1: If $w(e) < w(f)$:
   → Remove f from $MST_2 \cup \{e\}$. The new tree has cost $cost(MST_2) - w(f) + w(e)$.
   → Since $w(e) < w(f)$, this tree is cheaper than MST_2, contradicting its minimality.

Question assigned to the following page: 3

Case 2: If $w(e) > w(f)$:

→ Remove e from $MST_1 \cup \{f\}$. The new tree has cost $cost(MST_1) - w(e) + w(f)$.

→ Since $w(e) > w(f)$, this tree is cheaper than MST_1, contradicting its minimality.

4. Conclusion: Both cases lead to contradictions. Therefore, no two distinct MSTs can exist when edge weights are unique.

The edge uniqueness matters otherwise if edge weights are not unique, then multiple MSTs can exist. For this reason it should be unique.

* Therefore the solution is always unique.

→ When all road construction costs are distince, there is exactly one MST that connects all neighborhoods at the minimal total cost. This is guaranteed by the cycle property and the deterministic behaviour of MST algorithms under unique edge weights. Any assumption of multiple MSTs leads to contradictions in total cost, proving uniqueness.

Question assigned to the following page: 4

Question 4
Answer 4

Part a] Prim's algorithm in $O(n \log n)$ time

Prim's algorithm with a binary heap achieves $O(n \log n)$ time for a connected graph with edge weights 1 or 2, "where n is the number of edges"

Steps:

1. Prim's Algorithm:
→ Maintain a priority queue (binary heap) to track the minimum-weight edge connecting the growing MST to unvisited nodes.
→ For each node, the key (minimum edge weight to the MST) is initialized to ∞ and updated as edges are relaxed.

2. Key observations for Edge weights 1 or 2:
→ Each node's key can be updated at most twice.
⇒ From $\infty \to 2$ (first encountered via a weight-2 edge).
⇒ From $2 \to 1$ (later discovered via a weight-1 edge).
→ Total decrease_key operations: $O(m)$, where "m is the number of nodes"

Question assigned to the following page: 4

3. Time complexity :

→ Extract-Min : n operations, each $O(\log n)$.
total = $O(n \log n)$.

→ Decrease-Key : $O(m)$ operations, each $O(\log n)$
total = $O(m \log n)$

→ Edge Relaxation : All m edges are checked once.
total = $O(m)$.

4. For connected Graph.
$m = O(n)$, $\log(m) = O(\log n)$
total time : $O(n \log n)$

— — — —

Part b] Improving to $O(n)$ time

A modified algorithm using two queues achieves $O(n)$ time.

Algorithm :

1. Modified Method :
1.1 Queue 1 : Nodes reachable via edges of weight 1 (processed first)
1.2 Queue 2 : Nodes reachable via edges of weight 2.
1.3 Key step : Prioritize weight 1 edges to avoid a priority queue.

Question assigned to the following page: 4

2. Initialize with an arbitrary node in Queue.
3. Process all nodes in Queue first (BFS-like traversal for weight-1 edges).
4. Once Queue 1 is empty, process Queue 2 (weight-2 edges).

Time complexity :
→ Each edge is processed once : $O(m)$
→ Each node is enqueued/dequeued once : $O(n)$.
Total time : $O(n+m)$
In connected graphs $m = O(n)$
So final total time : $O(n)$

Edge cases :
→ All edges weight 1 : Processed in $O(n)$.
→ All edges weight 2 : Processed in $O(n)$.
→ Mixed weights : weight-1 edges are prioritized, ensuring correctness.

The modified algorithm achieves $O(n)$ time, strictly better than $O(n \log n)$.

Question assigned to the following page: 4

In this Answer for Question 4 I have taken n as the Number of Edges and m and the Number of Nodes.

In the next answer for Question 4, I have followed the general nomenclature and taken n as the Number of Nodes and m as the number of Edges.

Question assigned to the following page: 4

Question 4
Answer 4

* MST Problem with Edge Weights 1 or 2

1. Prim's Algorithm achieves $O(n \log n)$ time

→ Prim's Algorithm traditionally uses a priority queue (a binary heap) to select the minimum weight edge connecting the growing MST to a new node. For a graph with m edges and n nodes, the time complexity is $O(m \log n)$. However, when all edge weights are either 1 or 2, we can optimize:

(i) Each node's key (the smallest edge weight connecting it to the MST) can be only 1 or 2

(ii) Priority Queue Operations:

→ Extract-Min: Dominates the runtime. With n nodes, there are n extract-min operations, each taking $O(\log n)$ time.

→ Decrease-Key: For m edges, updating keys (to 1 or 2) still takes $O(m \log n)$, but since keys can only decrease once per node (from 2 to 1), the total cost reduces to $O(n \log n)$.

Therefore: Prim's runs in $O(n \log n)$ time for this graph.

Question assigned to the following page: 4

2. Yes, we can do better than O(nlogn)

→ Algorithm:

Use a modified Prim's algorithm with two queues:

- Queue 1 - Nodes reachable via edges of weight 1.
- Queue 2 - Nodes reachable via edges of weight 2.

Steps:

1. Start from any node v.
2. For each node, add its adjacent unvisited nodes to Queue 1 (if edge weight = 1) or Queue 2 (if edge weight = 2).
3. Process all nodes in Queue 1 first (since weight 1 edges are cheaper).
4. Once Queue 1 is empty, process Queue 2.

Time Complexity:

→ Each edge and node is processed once.
→ Queue operations (enqueue/dequeue) take O(1) time.
→ Total time: O(m+n) which is linear in the graph's size.

Question assigned to the following page: 4

3. Proof of correctness and Optimality

Correctness :

→ The algorithm prioritizes weight 1 edges first, ensuring the MST is built with the smallest possible total cost.

→ All nodes are processed, guaranteeing connectivity.

Optimality :

→ The $O(m+n)$ runtime is asymptotically better than $O(n \log n)$, especially for sparse graphs ($m = O(n)$).

→ For dense graphs ($m = O(n^2)$), $O(m)$ is still optimal since all edges must be examined.

Therefore

→ Prim's algorithm can run in $O(n \log n)$

→ A modified Prim's algorithm using two queues achieves $O(m+n)$ time, which is strictly better than $O(n \log n)$

Question assigned to the following page: 5

Question 5
Answer 5

* Modified Dijkstra's Algorithm for Maximum bandwidth Path
→ Instead of finding paths with minimum total cost, we want to maximize the minimum edge capacity along the path. For each vertex v, we maintain a value bandwidth(v) that represents the maximum bandwidth (the highest bottleneck value) from s to v found so far. When relaxing an edge (u,v) with capacity w(u,v) the candidate bandwidth for v is

new_bandwidth = min(bandwidth(u), w(u,v))

If this value is greater than the current bandwidth(v), we update it and adjust the vertex's key in the max-heap.

To find the maximum bandwidth path between a source vertex s and a destination vertex t in a weighted graph G = (V, E) we can modify Dijkstra's algorithm:

Question assigned to the following page: 5

Steps

1. Initialize all vertices v in V with bandwidth(v) = 0, except for the source s where bandwidth(s) = ∞.

2. Use a max-heap priority queue Q instead of a min-heap, initialized with all vertices in V.

3. While Q is not empty:
   3.1 Extract the vertex u with maximum bandwidth from Q.
   3.2 For each adjacent vertex v of u:
      3.2.1 Calculate new_bandwidth
      new_bandwidth = min(bandwidth(u), w(u,v)), where w(u,v) is the weight (bandwidth capacity) of edge (u,v).
      3.2.2 If new_bandwidth > bandwidth(v):
         3.2.2.1 Update bandwidth(v) = new_bandwidth
         3.2.2.2 Increase the key of v in Q to new bandwidth value

4. End when Q is empty or t is extracted.

The final bandwidth(t) will be the maximum bandwidth of the path from s to t.

Question assigned to the following page: 5

Another way of steps :

1. Using a max-heap instead of a min-heap
2. Initialising bandwidth(s) = ∞ and all other bandwidth(v) = 0
3. Relaxing each step edge(u,v) via
   new_bandwidth = min(bandwidth(u), w(u,v))
   and updating bandwidth(v) if new_bandwidth is greater.

Time Complexity :

$O(m \log n)$, where m is the number of edges and n is no. of vertices, due to the heap operations.

Correctness :

When a vertex is extracted from the max-heap, its bandwidth value represents the maximum possible bottleneck value for any path from s to that vertex.