Discussion 4

1. Hardy decides to start running to work in San Francisco city to get in shape. He prefers a route that goes entirely uphill and then entirely downhill so that he could work up a sweat uphill and get a nice, cool breeze at the end of his run as he runs faster downhill. He starts running from his home and ends at his workplace. To guide his run, he prints out a detailed map of the roads between home and work with k intersections and m road segments (any existing road between two intersections). The length of each road segment and the elevations at every intersection are given. Assuming that every road segment is either fully uphill or fully downhill, give an efficient algorithm to find the shortest path (route) that meets Hardy's specifications. If no such path meets Hardy's specifications, your algorithm should determine this. Justify your answer.

Solution: Use a directed graph to represent the map of the city. The edge direction will indicate the uphill direction of the road segment. Then

- Run Dijkstra's algorithm with Home as the starting point. This will find us the shortest paths and shortest distances from Home to all nodes in the graph that can be reached using uphill edges only. The node set that can be reached using such uphill paths will be called S_1.
- Run Dijkstra's algorithm with Work as the starting point. This will find us the shortest paths and shortest distances from Work to all nodes in the graph that can be reached using uphill edges only. The node set that can be reached using such uphill paths will be called S_2.
- $S = S_1 \cap S_2$
- If nodes corresponding to Work or Home fall into S remove them
- If S is null then there is no such solution, otherwise, for each node *i* in S, add up the shortest uphill distance from Home to *i* and shortest uphill distance from Work to *i*. Select the node *j* in S that gives us the smallest total distance. The path we are interested in will be the shortest uphill path from Home to *j* followed by the shortest downhill path from *j* to Work.

2. You are given a graph representing the several career paths available in industry. Each node represents a position and there is an edge from node *v* to node *u* if and only if *v* is a prerequisite for *u*. Top positions are the ones which are not prerequisites for any positions. The cost of an edge *(v, u)* is the effort required to go from one position *v* to position *u*. Ivan wants to start a career and achieve a top position with minimum effort. Using the given graph can you provide an algorithm with the same run time complexity as Dijkstra's? You may assume the graph is a DAG.

Solution: Add a new node S and connect S to all entry level positions with zero edges costs. Run Dijkstra's starting from S and find all shortest paths to all nodes in the graph. Find the top position *t* with lowest distance from S. The career path we are looking for will be on the shortest path from S to *t*.

3. You have a stack data type, and you need to implement a FIFO queue. The stack has the usual POP and PUSH operations, and the cost of each operation is 1. The FIFO has two operations: ENQUEUE and DEQUEUE. We can implement a FIFO queue using two stacks. What is the amortized cost of ENQUEUE and DEQUEUE operations.

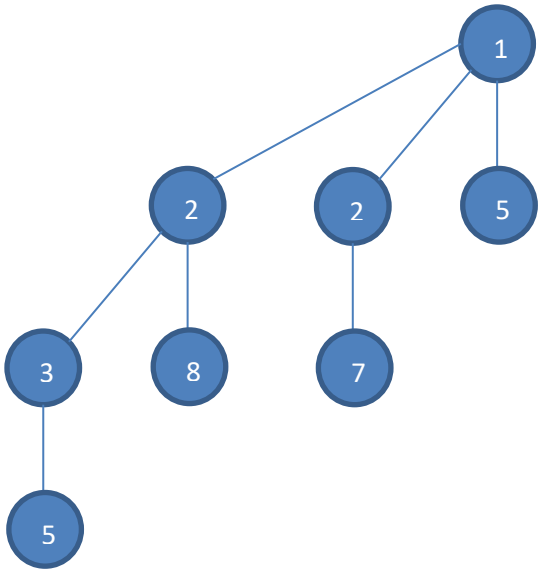
Implementation of the queue using two stacks:
Enqueue – Push element enqueued onto stack A
Dequeue – If stack B is not empty, Pop top element from B, otherwise, Pop all elements from A and Push them onto B one at a time. Then Pop the top element from B.

Solution 1 : Aggregate Method
A worst-case sequence of operations occurs when we do n Enqueues and one Dequeue. So there will be a total of n+1 operations involved.
T(n+1) = total time to Push n elements onto A + total time to Pop all elements from A + total time to Push all elements onto B + time to Pop top element from B
T(n+1) = n + n + n + 1 = 3n + 1
Average cost of the operations = T(n+1) / (n+1) = (3n+1) / (n+1)
When n -> ∞ , (3n+1) / (n+1) = 3 = O(1)
O(1) is the amortized cost of both Enqueue and Dequeue operations

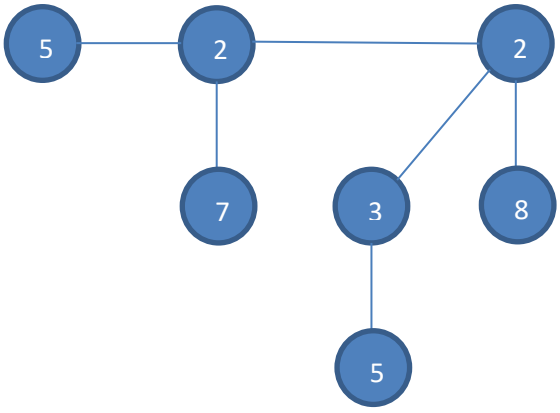
Solution 1 : Accounting Method
Charge Enqueue more than its actual cost so we can save credit for the expensive Dequeue operation. Since each element Pushed onto A needs to be Popped from A and then Pushed onto B, we can charge Enqueue 3 units instead of its actual cost (which is 1 unit). We can charge Dequeue 1 unit.
Check to see if there will be enough credit to pay for a worst case sequence of operations:
n Enqueue results in a total charge of 3n. When we subtract the actual cost of 1n, we are left with 2n units of credit. Dequeue is also charged 1 unit so we have a total to 2n+1 to pay for the Dequeue operation. Let's see if that is enough. The actual cost of Dequeue will be n + n + 1 which is exactly 2n+1. The charges are therefore sufficient.
Amortized cost of Enqueue = 3 = O(1)
Amortized cost of Dequeue = 1 = O(1)

4. Given a sequence of numbers: 3, 5, 2, 8, 1, 5, 2, 7

a. Draw a binomial heap by inserting the above numbers reading them from left to right



b. Show a heap that would be the result after the call to deleteMin() on this heap



5. (a): Suppose we are given an instance of the Minimum Spanning Tree problem on a graph G. Assume that all edges costs are distinct. Let T be a minimum spanning tree for this instance. Now suppose that we replace each edge cost c_e by its square, c_e^2 thereby creating a new instance of the problem with the same graph but different costs. Prove or disprove: T is still a MST for this new instance.

Solution: T may no longer be the MST since if there are any negative cost edges in G, then those edges will have positive costs after squaring the edge costs. So, the order of edges (based on their weight) could change.

(b): Consider an undirected graph G = (V, E) with distinct nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of G. Now suppose each edge weight is increased by 1: the new weights are $c'_e = c_e + 1$. Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

Solution: Since we are adding a constant to all edge weights the order of edges (based on their weight) will NOT change. So, for example, Kruskal's algorithm will pick edges exactly in the same order since the graph topology has not changed.