CS 61B Spring 2021 — Heaps and Graphs

Exam Prep Discussion 9: March 15, 2021

1 Fill in the Blanks

Fill in the following blanks related to min-heaps. Let `N` is the number of elements in the min-heap. For the entirety of this question, assume the elements in the min-heap are **distinct**.

1. `removeMin` has a best case runtime of $\Theta(1)$ and a worst case runtime of $\Theta(\log N)$.
2. `insert` has a best case runtime of $\Theta(1)$ and a worst case runtime of $\Theta(\log N)$.
3. A pre order or level order traversal on a min-heap *may* output the elements in sorted order. Assume there are at least 3 elements in the min-heap.
4. The fourth smallest element in a min-heap with 1000 elements can appear in 14 places in the heap.
5. Given a min-heap with $2^n - 1$ elements, for an element
   - to be on the second level it must be less than $2^{(N-1)} - 2$ element(s) and greater than 1 element(s).
   - to be on the bottommost level it must be less than 0 element(s) and greater than $N-1$ element(s).

2 Heap Mystery

We are given the following array representing a min-heap where each letter represents a **unique** number. Assume the root of the min-heap is at index zero, i.e. `A` is the root. Note that there is **no** significance of the alphabetical ordering, i.e. just because B precedes C in the alphabet, we do not know if B is less than or greater than C.

Array: [A, B, C, D, E, F, G]

Four unknown operations are then executed on the min-heap. An operation is either a `removeMin` or an `insert`. The resulting state of the min-heap is shown below.

Array: [A, E, B, D, X, F, G]

(a) Determine the operations executed and their appropriate order. The first operation has already been filled in for you!

1. `removeMin()`
2. insert(X)
3. removeMin( )
4. insert(A)

(b) Fill in the following comparisons with either >, <, or ? if unknown. Note that this question does not assume a specific ordering of operations from the previous part, i.e. we don't know which of the two possible

1. X ? D
2. X > C
3. B > C
4. G < X

3 Graph Conceptuals

Answer the following questions as either **True** or **False** and provide a brief explanation:

1. If a graph with n vertices has $n-1$ edges, it **must** be a tree.

   False. The graph must be connected.

2. The adjacency matrix representation is **typically** better than the adjacency list representation when the graph is very connected.

3. Every edge is looked at exactly twice in **every** iteration of DFS on a connected, undirected graph.

   True. The two vertices the edge is connecting will look at that edge when it's their turn.

4. In BFS, let $d(v)$ be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe, $|d(u) - d(v)|$ is **always less than** 2.

   True. Suppose this was not the case. Then, we could have a vertex 2 edges away and a vertex 4 edges away in the fringe at the same time. But, the only way to have a vertex 4 edges away is if a vertex 3 edges away was removed from the fringe. We see this could never occur because the vertex 2 edges away would be removed before the vertex 3 edges away!

True. The adjacency matrix representation is usually worse than the adjacency list representation with regards to space, scanning a vertex's neighbors, and full graph scans. However, when the graph is very connected, the adjacency matrix representation has roughly same asymtotic runtime in these operations, while "winning" in operations like hasEdge.

4 Cycle Detection

Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm. You may use either an adjacency list or an adjacency matrix to represent your graph. (We are looking for an answer in plain English, not code).

We do a depth first search traversal through the graph. While we recurse, if we visit a node that we visit a node that we visited already, then we've found a cycle. Assuming integer labels, we can use something like a visited boolean array to keep track of the elements that we've seen, and while looking through a node's neighbors, if visited gives true, then that indicates a cycle.

However, since the graph is undirected, if an edge connects vertices u and v, then u is a neighbor of v, and v is a neighbor of u. As such, if we visit v after u, our algorithm will claim that there is a cycle since u is a visited neighbor of v. To address this case, when we visit the neighbors of v, we should ignore u. To implement this in code, one idea is. using a Map to map each node to the node we took to get there, e.g. we could map v to u in the example described above.

In the worst case, we have to explore at most V edges before finding a cycle (number of edges doesn't matter). So, this runs in $\Theta(V)$.