CS 61B
Spring 2025

Graphs, Heaps
Exam-Level 08: March 17, 2025

1 Graph Conceptuals

(a) 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.

   Solution: False. The graph **must** be connected.

2. Every edge is looked at exactly twice in each full run of DFS on a connected, undirected graph.

   Solution: True. Say an edge connects u and v. Both u and v will look at the other one through this edge when it's their turn.

3. 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 (recall that the fringe in BFS is a queue), $|d(u) - d(v)|$ is **always less than** 2.

   Solution: 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!

(b) 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.

Solution: We do a depth first search traversal through the graph. While we recurse, if 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, we could add the parent as another parameter in the method call. 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 $\Theta(V)$.

Pseudocode is provided below (for a disconnected graph, we should call `find_cycle` on each component).

```
find_cycle(v, parent=-1):
    visited[v] = true
    for (v, w) in G:
        if !visited[w]:
            if find_cycle(w, v):
                return True
        else if w != parent:
            return True
    return False
```

2 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 _____ and a worst case runtime of _____.
2. **`insert`** has a best case runtime of _____ and a worst case runtime of _____.
3. A _____ or _____ 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 **distinct** elements can appear in _____ places in the heap. (Feel free to draw the heap in the space below.)
5. Given a min-heap with $2^N - 1$ **distinct** elements, for an element
   - to be on the second level it must be less than _____ element(s) and greater than _____ element(s).
   - to be on the bottommost level it must be less than _____ element(s) and greater than _____ element(s).

   Hint: A complete binary tree (with a full last-level) has $2^N - 1$ elements, with N being the number of levels. (Feel free to draw the heap in the space below.)

Solution:

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 can output the elements in sorted order.
   Explanation: The smallest item of a min heap is at the top, so whatever traversal we choose must output the top element first in a complete binary tree. Only preorder and level-order have this property.
4. The fourth smallest element in a min-heap with 1000 distinct elements can appear in 14 places in the heap.
   Explanation: The 4th smallest item can be on the 2nd, 3rd, or 4th level of the heap.
5. Given a min-heap with $2^N - 1$ distinct 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). (must be greater than the elements on its branch)

   Explanation: An element on the second level must be larger than the root and less than the elements in its subtree. There are $2^{N-1} - 2$ elements in the subtree of an element on the second level: half the elements in the tree minus the root, then subtracting off the node itself.
   An element on the bottom level must be greater than all elements on the path from itself to the root. A min heap with $2^N - 1$ elements has N levels, so there are $N - 1$ items above it on a path to the root.

3 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 one, i.e. `A` is the root. Our task is to figure out the numeric ordering of the letters. Therefore, 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!

Hint: Which elements are gone? Which elements are newly added? Which elements are removed and then added back?

1. `removeMin()`
2. _____
3. _____
4. _____

Solution:
1. removeMin()
2. insert(X)
3. removeMin()
4. insert(A)

Explanation: We know immediately that A was removed. Then, after looking at the final state of the min-heap, we see that C was removed. Then, for A to remain in the min-heap, we see that A must have been inserted afterwards. And, after seeing a new value X in the min-heap, we see that X must have been inserted as well. We just need to determine the relative ordering of the `insert(X)` in between the operations `removeMin()` and `insert(A)`, and we see that the `insert(X)` must go before both.

(b) Fill in the following comparisons with either $>$, $<$, or ? if unknown. We recommend considering which elements were compared to reach the final array.

1. X _____ D
2. X _____ C
3. B _____ C
4. G _____ X

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

Reasoning:

1. X is never compared to D
2. X must be greater than C since C is removed after X's insertion.
3. B must also be greater than C otherwise the second call to `removeMin` would have removed B
4. X must be greater than G so that it can be "promoted" to the top after the removal of C. It needs to be promoted to the top to land in its new position.