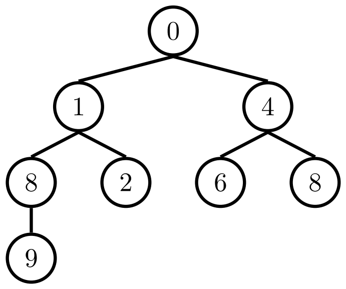
CS 61BL Summer 2021 — Heaps and Graphs

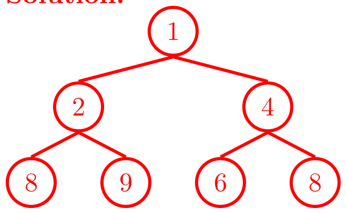
Recurring Section 10: Tuesday August 2, 2021

1 Heaps of Fun

(a) Draw the Min Heap that results if we delete the smallest item from the heap.

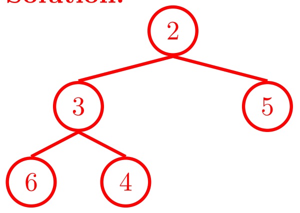


Solution:



(b) Draw the Min Heap that results if we insert the elements 6, 5, 4, 3, 2 into an empty heap.

Solution:

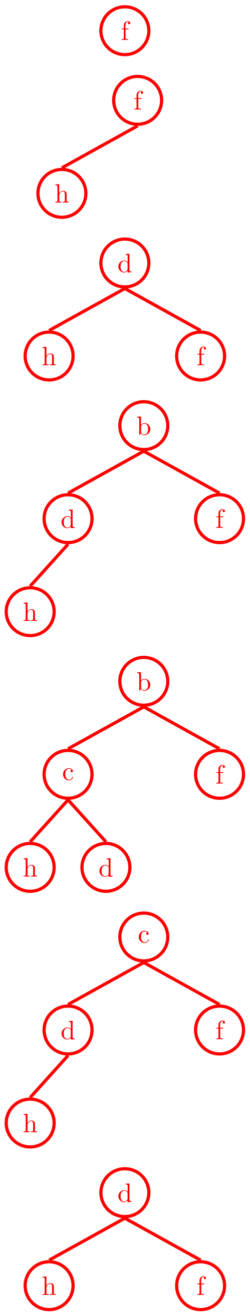


(c) Assume that we have a binary min-heap (smallest value on top) data structure called `MinHeap` that has properly implemented the `insert` and `removeMin` methods. Draw the heap and its corresponding array representation after each of the operations below:

```
MinHeap<Character> h = new MinHeap<>();
h.insert('f');
```

```
h.insert('h');
h.insert('d');
h.insert('b');
h.insert('c');
h.removeMin();
h.removeMin();
```

Solution:



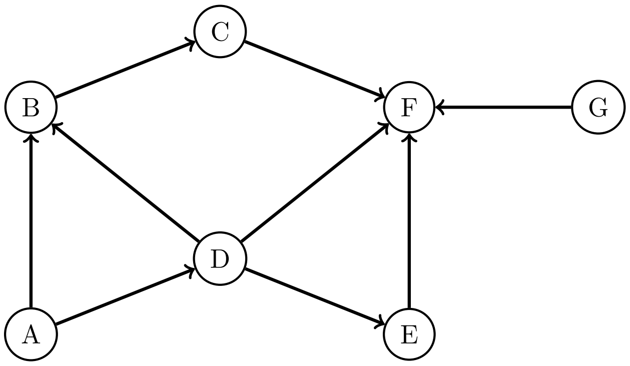
(d) Your friendly TA Sadia challenges you to quickly implement an integer max-heap data structure. However, you already have your `MinHeap` and you don't feel like writing a whole second data structure. Can you use your min-heap to mimic the behavior of a max-heap? Specifically, we want to be able to get the largest item in the heap in constant time, and add things to the heap in $\Theta(\log n)$ time, as a normal max heap should.

Hint: Although you cannot alter them, you can still use methods from `MinHeap`.

Yes. For every insert operation, negate the number and add it to the min-heap.

For a `removeMax` operation call `removeMin` on the min-heap and negate the number returned. Any number negated twice is itself (with one exception in Java, -2^{-31}), and since we store the negation of numbers, the order is now reversed (what used to be the max is now the min).

2 Graphs



(a) Write the graph above as an adjacency matrix, then as an adjacency list. What would be different if the graph were undirected instead?

Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 0 0 1 0 0 0 0
C 0 0 0 0 0 1 0
D 0 1 0 0 1 1 0
E 0 0 0 0 0 1 0
F 0 0 0 0 0 0 0
G 0 0 0 0 0 1 0
ˆ start node
```

List:

```
A: {B, D}
B: {C}
C: {F}
D: {B, E, F}
E: {F}
F: {}
G: {F}
```

For the undirected version of the graph, the representations look a bit more symmetric. For your reference, the representations are included below:

Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 1 0 1 1 0 0 0
C 0 1 0 0 0 1 0
D 1 1 0 0 1 1 0
E 0 0 0 1 0 1 0
F 0 0 1 1 1 0 1
G 0 0 0 0 0 1 0
ˆ start node
```

List:

```
A: {B, D}
B: {A, C, D}
C: {B, F}
D: {A, B, E, F}
E: {D, F}
F: {C, D, E, G}
G: {F}
```

(b) Write the order in which DFS pre-order graph traversal would visit nodes in the directed graph above, starting from vertex A. Break ties alphabetically. Do the same for DFS post-order and BFS.

```
DFS preorder: ABCFDE (G)
DFS postorder: FCBEDA (G)
BFS: ABDCEF (G)
```

Explanations

DFS preorder and postorder: To compute this, we maintain a stack of nodes, and a marked set. As soon as we add something to our stack, we note the down for preorder. The top node in our stack represents the node we are currently on, and the marked set represents nodes that have been visited. After we add a node to the stack, we visit its lexicographically next unmarked child. If there is none, we pop the topmost node from the stack and note it down for postorder. *Note that there are two ways DFS could run: with restart or without; DFS with restart is the version where if we have exhausted our stack, and still have unmarked nodes left, we restart on the next unmarked node.*

Stack (bottom-top), MarkedSet, Preorder, Postorder.

```
A. {A}. A. -
AB. {AB}. AB. -
ABC. {ABC}. ABC. -
ABCF. {ABCF}. ABCF. -
ABC. {ABCF}. ABCF. F
AB. {ABCF}. ABCF. FC.
A. {ABCF}. ABCF. FCB.
AD. {ABCFD}. ABCFD. FCB.
ADE. {ABCFDE}. ABCFDE. FCB.
AD. {ABCFDE}. ABCFDE. FCBE.
A. {ABCFDE}. ABCFDE. FCBED.
\-. {ABCFDE}. ABCFDE. FCBEDA.
```

If DFS restarts on unmarked nodes, the following happens in the last line. Otherwise, we do not proceed further.

```
G. {ABCFDEG}. ABCFDEG. FCBEDAG.
```

BFS: Start at the provided start node. Note it down, and mark it. Now, consider all nodes that are 1-hop (i.e. one edge) away from the start node. Write all of them down, and mark all of them. Next, consider all unmrked nodes that are 1-hop away from the nodes that were 1-hop away from the start (i.e., 2 hops away from the start). And so on. Note that unlike DFS, BFS uses a queue.

BFS, MarkedSet.

```
A. {A}.
A BD. {ABD}.
A BD CEF. {ABDCEF}.
```

If BFS restarts, the following happens at the end. Otherwise, we do not proceed further..

```
A BD CEF (G). {ABDCEFG}.
```

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. 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.

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, $|d(u) - d(v)|$ 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!

4. Given a fully connected, directed graph (a directed edge exists between every pair of vertices), a topological sort can never exist.

   False. Consider the graph constructed as follows: for all vertices i, j such that $i < j$, draw a directed edge from i to j. A valid topological ordering of this graph is simply enumerating the vertices: $1, 2, 3, \ldots . N$.