1 Heaps of fun

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

```
Heap h = new Heap(5); // Creates a min-heap with 5 as the root
h.insert(7);
h.insert(3);
h.insert(1);
h.insert(2);
h.removeMin();
h.removeMin();
```

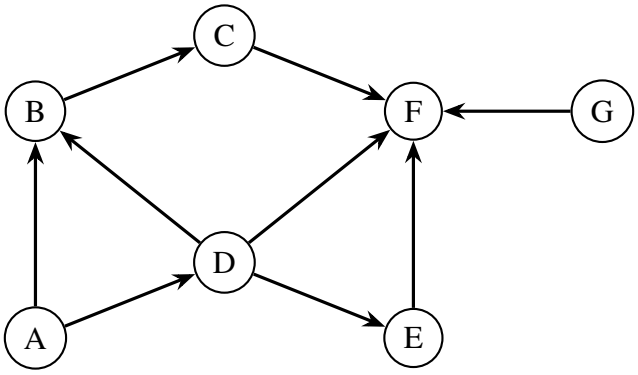
```
Heap h = new Heap(5); //Creates a min-heap with 5 as the root
// Remember that for the underlying array, we don't use index [0]. These
    solutions list the integers starting at index [1]

[5]              5

h.insert(7);
[5,7]            5
                /
               7
h.insert(3);
[3,7,5]          3
                / \
               7   5
h.insert(1);
[1,3,5,7]        1
                / \
               3   5
              /
             7
h.insert(2);
[1,2,5,7,3]      1
                / \
               2   5
              / \
             7   3
h.removeMin();
[2,3,5,7]        2
                / \
               3   5
              /
             7
h.removeMin();
[3,7,5]          3
                / \
               7   5
```

(b) Your friend Sahil Finn-Garng challenges you to quickly implement an integer max-heap data structure - "Hah! I'll just use my min-heap implementation as a template to write max-heap.java", you think to yourself. Unfortunately, two Destroyer Penguins manage to delete your `MinHeap.java` file. You notice that you still have `MinHeap.class`. Can you still complete the challenge before time runs out? Hint: 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 Graph Representations

Write the graph above as an adjacency matrix, then as an adjacency list.

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
^ start node
```

List:

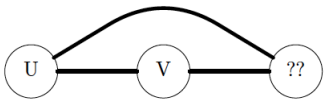
```
A: {B, D}
B: {C}
C: {F}
D: {B, E, F}
E: {F}
F: {}
G: {F}
```

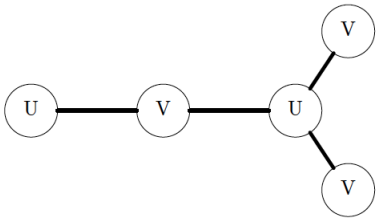
Give the DFS preorder, DFS postorder, and BFS order of the graph starting from vertex A. Break ties alphabetically.

DFS preorder: ABCFDE
DFS postorder: FCBEDA
BFS: ABDCEF

3 Graph Algorithm Design: Bipartite Graphs

An undirected graph is said to be bipartite if all of its vertices can be divided into two disjoint sets U and V such that every edge connects an item in U to an item in V. For example, the graph on the left is bipartite, whereas on the graph on the left is not. Provide an algorithm which determines whether or not a graph is bipartite. What is the runtime of your algorithm?





To solve **this** problem, we simply run a special version of DFS or BFS from any vertex. This special version marks the start vertex with a U, then each of its children with a V, and each of their children with a U, and so forth. If the DFS or BFS traverses to a node labeled U which has a visited child node already labeled U, then the graph is not bipartite (same **for if** both were labeled V).

If the graph is not connected, we repeat **this** process **for** each connected component.

If the algorithm completes, successfully marking every vertex in the graph, then it is bipartite.

4 Extra for Experts: Shortest Directed Cycles

Provide an algorithm that finds the shortest directed cycle in a graph in $O(EV)$ time and $O(E)$ space, assuming $E > V$.

The key realization here is that the shortest directed cycle involving a particular source vertex is just some shortest path plus one edge back to s. Using **this** knowledge, we can create a shortestCycleFromSource(s) subroutine. This subroutine first runs BFS on s, then checks every edge in the graph to see **if** it points at s. For each such edge originating at vertex v, it computes the cycle length by adding one to distTo(x) (which was computed by BFS).

This subroutine takes $O(E+V)$ time because it is BFS. To find the shortest cycle in the entire graph, we simply call shortestCycleFromSource() **for** each vertex, resulting in an $V * O(E+V) = O(EV+V^2)$ runtime. Since $E > V$, **this** is just $O(EV)$.

5 Extra for Experts: DFS Gone Wrong

Consider the following implementation of DFS, which contains a crucial error:

```
create the fringe, which is an empty Stack
    push the start vertex onto the fringe and mark it
    while the fringe is not empty:
        pop a vertex off the fringe and visit it
        for each neighbor of the vertex:
            if neighbor not marked:
                push neighbor onto the fringe
                mark neighbor
```

Give an example of a graph where this algorithm may not traverse in DFS order.