1. [20 marks] Implement and augment AVL trees to support all these operations in worst case $O(\log(n))$ time ($n$ refers to the number of keys in the tree).
   - $s$.find($k$): whether $k$ is in $s$.
   - $s$.insert($k$): insert $k$ into $s$; no change if $k$ is already present in $s$.
   - $s$.remove($k$): remove $k$ from $s$; no change if $k$ is already absent in $s$.
   - $s$.from($k$, $i$): return the $i$th smallest key greater than $k$; if there is no such key, throw the exception NotFound. More details and clarifications:
     - $i$ is a natural number (I will not give you negative numbers)
     - $k$ should be already in $s$; if it is not, throw NotFound
     - “the 0th smallest key greater than $k$” means $k$ itself, despite my imperfect wording
   Example: if $s$ consists of these keys: 37, 46, 71, 85:
     - $s$.from(46, 0) returns 46
     - $s$.from(46, 1) returns 71
     - $s$.from(46, 2) returns 85
     - $s$.from(46, 3) throws NotFound
     - $s$.from(55, $i$) throws NotFound regardless of $i$
   Prove that your insert and from both take $O(\log(n))$ worst-case time.
   The template code is on Blackboard. You likely have to add more fields to the node class to help with from. You may also like to add your own helper functions.
   My tests are partly blackbox and partly not: some tests just perform the above operations and check final answers, some others actually check that your trees are really AVL trees. Although the latter break your abstraction, they may be helpful.

2. [5 marks] Start with an empty heap and insert these priorities in the given order. Use a min-heap, i.e., smaller priorities are closer to the root. Hand in both the final binary tree form and the final array form.
   38, 15, 76, 21, 40, 32, 59, 77, 48, 57.

3. [5 marks] In class and in the textbook, the method of storing a heap in an array starts at index 1. Suppose I really like to start at index 0, for example:

   ![Diagram]

   Give formulas for the indexes of:
   - the left child of the node at index $i$
**4.** [20 marks] An \(m \times n\) sliding-block puzzle is an \(m \times n\) grid holding \(m-n-1\) square blocks and 1 empty square. Each square block comes with a different label or picture. You may exchange the empty square with the block above, below, on the left, or on the right by sliding that block. The goal is to use a sequence of such moves to reach a target state. Example:

\[
\begin{array}{c|c|c}
\text{start} & \rightarrow & \text{target} \\
A & B & C \\
D & H & E \\
G & F \\
\end{array}
\begin{array}{c|c|c}
A & B & C \\
D & H & E \\
G & H & F \\
\end{array}
\begin{array}{c|c|c}
A & B & C \\
D & E & G \\
H & F \\
\end{array}
\begin{array}{c|c|c}
A & B & C \\
D & E & F \\
G & H \\
\end{array}
\]

Write a program to solve 3 \(\times\) 3 sliding puzzles like the above. The start state is given as a String of 9 characters. Each character is a letter A–H for a block, or space for the empty square. Indexes correspond to grid positions as:

\[
\begin{array}{ccc}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8 \\
\end{array}
\]

Example:

"ABCDHE GF" means

\[
\begin{array}{c|c|c}
A & B & C \\
D & H & E \\
G & F \\
\end{array}
\]

The target state is always "ABCDEFGH ".

Every way to reach the target state is not equal; some take fewer moves than others. Your program must find a way that takes the fewest moves. Your program must also finish in a reasonable amount of time.

The program must fill in this template code:

```java
import java.util.Iterator; // you may change this to java.util *
public class Sliding {
    public static Iterator<String> solve(String start) {
    }
}
```

If the target is unreachable, the function returns null.

If the target is reachable, the function returns an iterator that does this: The first time I call its next(), it gives me the state after the first move. In general, the \(i\)th time I call next(), it gives me the state after the \(i\)th move. After the fewest number of moves, it gives me the target state. So, exclude the start state, but include the target state.

You may add your own classes and helper functions, in the same file or in additional files. My tester will be simply calling Sliding.solve with various parameters. I will not give you invalid strings.

**5.** Mr. and Mrs. Smith, a married couple, invited 9 other married couples to a party. (So the party consisted of 10 couples.) There was a round of handshaking, but no one shook hand with his or her spouse. Afterwards, Mrs. Smith asked everyone except herself, “how many persons have you shaken hands with?” All 19 answers were different. With this information, we can already determine:

(a) [5 marks] What were the answered numbers?
(b) [10 marks] What was the answer from Mr. Smith?

Suggestion: Solve smaller problems (fewer invited couples) to see the pattern. Then solve the general problem by induction: Mr. and Mrs. Smith invited \(n\) other married couples. (So the party consisted of \(n+1\) couples.) All \(2n+1\) answers were different.