Precept Outline

- Binary Search Trees
  - Rank and Selection
  - Deletion
- Midterm Review

A. Rank, Selection and Deletion

In this section, let n be the number of nodes in a (not necessarily balanced) binary search tree. Feel free to modify the `Node` class if it helps implement the required methods.

Part 1: Rank and Selection

First, as a warm-up: describe how to implement `Key min()` and `Key max()`, which return the minimum and maximum keys in the tree, respectively.

Now, describe how to implement `int rank(Key key)`, which returns the rank (the number of nodes with smaller keys) of key `key`.

Finally, describe how to implement `Key select(int rank)`, which returns the key of rank `rank`.

Part 2: Deletion

Describe how to implement the method `deleteMin()`, which deletes the node with minimum key in the BST.

Describe how to implement the method `delete(Key key)`, which deletes the node with key `key`, if any, in the BST. *Hint: consider the node of rank* `rank(key) + 1`.

B. Midterm Review

Part 1: Data Structure Design

Suppose that there are two teams of players that can play each other in head-to-head matches. Each player has a certain rating, which is an integer value, and two players can play each other if they have the same rating (otherwise it isn't a balanced match). Design a data structure that computes the maximum number of distinct matches that the two teams can play at any point. The data structure should support 2 operations. The first, `addPlayer(rating, team)` adds a new player of rating equal to `rating` and adds it to team `team`, which can be 1 or 2. The second operation, `numberOfMatches()`, returns the maximum number of distinct matches that can be played by elements of team 1 versus elements of team 2 (see the example below for more information).

public class MatchMaker

`MatchMaker()`	*creates two empty teams*
`void addPlayer(int rating, int team)`	*adds player of rating to team*
`int numberOfMatches()`	*returns the maximum number of matches that can be played*

Full credit: The `addPlayer()` method should run in $O(\log n)$ time in the worst case and the `numberOfMatches()` method should run in $\Theta(1)$ time.
Partial credit: The `addPlayer()` method should run in $O(n)$ time in the worst case and the `numberOfMatches()` method should run in $\Theta(1)$ time.

Example

```
MatchMaker mm = new MatchMaker();
mm.addPlayer(100, 1); // Team 1: {100: 1}
mm.addPlayer(200, 1); // Team 1: {100: 1, 200: 1}
mm.addPlayer(100, 2); // Team 2: {100: 1}, Matches = 1
mm.addPlayer(200, 2); // Team 2: {100: 1, 200: 1}, Matches = 2
mm.addPlayer(100, 1); // Team 1: {100: 2, 200: 1}, Matches = 2
StdOut.println(mm.numberOfMatches()); // Output: 2
mm.addPlayer(100, 2); // Team 2: {100: 2, 200: 1}, Matches = 3
StdOut.println(mm.numberOfMatches()); // Output: 3
```

The first call of `numberOfMatches()` outputs 2 since we can match the two 200 rating players together as well as the one 100 rating player from team 2 with one of the 100 rating players from team 1. The second call can match each player to a different player, resulting in 3 total matches.

In the space provided, give a concise English description of your solution to the constructor, the `addPlayer()` *and the* `numberOfMatches()` *methods. You may use any of the algorithms and data structures that we have considered in this course (e.g., lectures, precepts, textbook, assignments) as subroutines. If you modify any of them, be sure to describe the modification. Feel free to use code or pseudocode to improve clarity.*

Part 2: Algorithm Design

A length-n integer array `a[]` is *single-peaked* if there exists $0 \leq k \leq n-1$ such that the subarray from index up to (and including) k is strictly increasing, and the subarray from index k until $n-1$ is strictly decreasing. The entry `a[k]` is called the *peak*. For example, the array `a[] = {3, 6, 7, 10, 4, 1}` is a single-peaked with peak `10`, but the array `a[] = {3, 6, 7, 10, 4, 5}` is not.

Design an algorithm that receives as input a single-peaked array with n *distinct elements*, and outputs the peak of the array. *Specify the running time of your solution.*

Full credit: The running time of the algorithm must be $O(\log n)$.
Partial credit: The running time of the algorithm must be $O(n)$.