CS 61B Fall 2014 Game Trees, Quad Trees and Heaps

1 Heaps of fun®

(a) Assume that we have a binary min-heap (smallest value on top) data structue 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
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) Consider an array based min-heap with N elements. What is the worst case running time of each of the following operations if we ignore resizing? What is the worst case running time if we take into account resizing? What are the advantages of using an array based heap vs. using a BST-based heap?

```
Insert      O(log N)
Find Min    O(1)
Remove Min  O(log N)

Accounting for resizing:
Insert      O(N)
Find Min    O(1)
Remove Min  O(N)

Using a BST is not space-efficient.
```

(c) Your friend Alyssa P. Hacker challenges you to quickly implement a max-heap data structure - "Hah! I'll just use my min-heap implementation as a template", you think to yourself. Unfortunately, your arch-nemesis Malicious Mallory deletes your min-heap.java file. You notice that you still have the min-heap.class file; could you use it to complete the challenge?

```
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.
```

2 Quad Tree

Let's say you were creating a photo album, and you wanted each photo in the album to be geotagged (latitude, longitude coordinates), so that you can quickly access these photos by location. With your 1337 61b skillz, you decide that you're going to use a Quad Tree to build this application.

(a) Fill out some of the basic instance variables/inner classes you would need in your Quad Tree implementation below. While doing so, think about some design decisions you need to make based on the nature of the application that uses the QuadTree class:

```
public class QuadTree<Key extends Comparable<Key>> {
        Node root;
        QuadTree(Key central_x, Key central_y, Image img) {
                Key x = central_x;
                Key y = central_y;
                root = new Node(x, y, img);
        }

        private class Node {
                Key x, y;
                ArrayList<Image> images;
                Node NW, NE, SE, SW;

                Node(Key x, Key y, Image img) {
                        this.x = x;
                        this.y = y;
                        images = new ArrayList<Image>();
                        images.add(img);
                }
        }
}
```

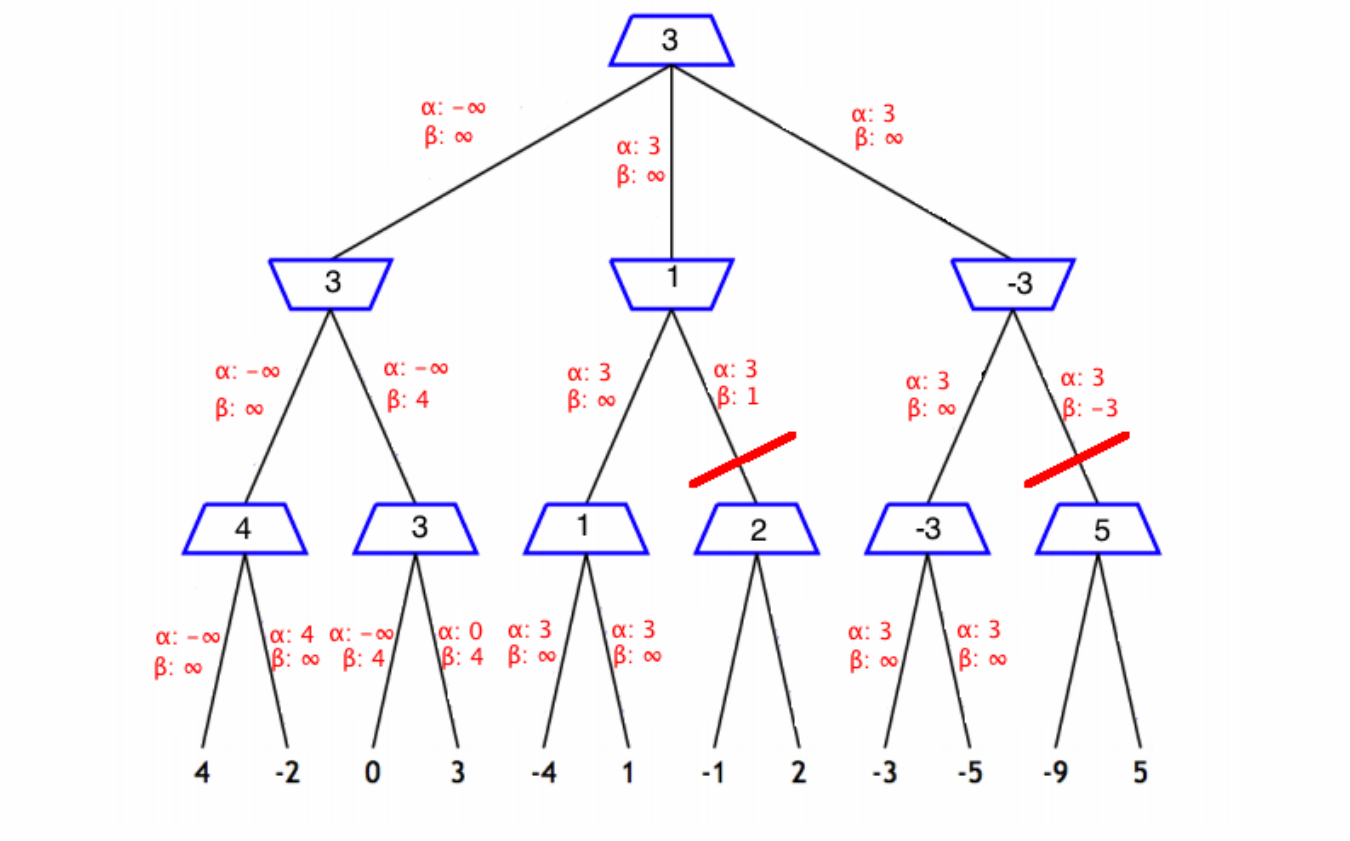
(b) Now go ahead and write an insert method for your QuadTree class:

```
public void insert(Key x,Key y, Image img) {
        root = insertToNode(root, x, y, img);
}

private Node insertToNode(Node n, Key x, Key y, Image img) {
        if (n == null) {
                return new Node(x, y, img);
        } else if (n.x.compareTo(x) <= 0 && n.y.compareTo(y) < 0) {
                n.NE = insertToNode(n.NE, x, y, img);
        } else if (n.x.compareTo(x) > 0 && n.y.compareTo(y) <= 0) {
                n.NW = insertToNode(n.NW, x, y, img);
        } else if (n.x.compareTo(x) >= 0 && n.y.compareTo(y) > 0) {
                n.SW = insertToNode(n.SW, x, y, img);
        } else if (n.x.compareTo(x) < 0 && n.y.compareTo(y) >= 0) {
                n.SE = insertToNode(n.SE, x, y, img);
        } else {
                n.images.add(img);
        }
        return n;
}
```

3 Minimax

Consider the game tree shown below. Trapezoids that 'point' up, represent the player seeking to maximize the heuristic evaluation (this is you); trapezoids that 'point' down represent the player seeking to minimize the heuristic evaluation (your opponent).



(a) Fill out the values in the 'maximizer' and 'minimizer' node for the above game tree after applying the minimax algorithm to it.

On diagram.

(b) Now show the result of Alpha-Beta pruning on the above tree. Fill out the alpha and beta values at each step.

On diagram.

(c) According to the minimax algorithm, which move should we make for the above game? Say, your opponent was a 3-year old, would you still use the minimax algorithm?

The move represented by the left-most edge at the root. No.