COP 3330 – Object Oriented Programming in Java
Java Class: TreeSet

TreeSet

A TreeSet is a set, just like a HashSet is a set. The fundamental difference between the two is that the objects in a TreeSet are ordered, so if you loop through the set, an iterator will go through those objects in order. This means that a TreeSet can only be made of a class that implements the Comparable interface.

Other than the natural iterator loop going in order through the objects in the set, here are several methods the class offers that are based on the fact that the objects in the set are ordered. The internal storage of a TreeSet is a balanced binary search tree (a Red-Black tree in particular). We'll typically use the default constructor and all the methods from the HashSet that are generic to sets are also included in TreeSet and offer the same functionality, albeit with an O(lg n) run-time where n is the number of items in the set.

Here are the important order specific methods in the TreeSet class:

```
// Returns the least element in this TreeSet greater than or
// equal to e or null if no such element exists.
E ceiling(E e)

// Returns the first element in this TreeSet or null if it does
// not have any elements.
E first()

// Returns the greatest element in this TreeSet less than or
// equal to e or null if no such element exists.
E floor(E e)

// Returns the smallest element in this TreeSet strictly greater
// than e or null if no such element exists.
E higher(E e)

// Returns the last element in this TreeSet or null if it does
// not have any elements.
E last()

// Returns the largest element in this TreeSet strictly less
// than e or null if no such element exists.
E lower(E e)

// Removes and returns the first element in this TreeSet.
E pollFirst(E e)

// Removes and returns the last element in this TreeSet.
E pollLast(E e)
```

TestTreeSet Example

Here is a short example of a TreeSet of Integers that calls the methods lower, higher, floor and ceiling. This program produces a lot of output so that output won't be reproduced here. You are encouraged to run the program yourself and observe/inspect the output. In addition, add method calls to first(), last(), pollFirst() and pollLast() to get comfortable with those methods.

```
import java.util.*;

public class TestTreeSet {

    public static void main(String[] args) {

        TreeSet<Integer> myTreeSet = new TreeSet<Integer>();
        Random rndObj = new Random();

        for (int i=0; i<20; i++) {
            int term = rndObj.nextInt(100);
            System.out.println("Inserting "+term+" into set");
            myTreeSet.add(term);
            System.out.print("set is : ");
            print(myTreeSet);
        }

        for (int i=0; i<10; i++) {

            int term = rndObj.nextInt(100);
            System.out.println("Query number is "+term);
            Integer myLower = myTreeSet.lower(term);
            Integer myHigher = myTreeSet.higher(term);
            Integer myFloor = myTreeSet.floor(term);
            Integer myCeiling = myTreeSet.ceiling(term);

            if (myLower != null) System.out.println("lower said "+myLower);
            if (myHigher != null) System.out.println("highersaid "+myHigher);
            if (myFloor != null) System.out.println("floor said "+myFloor);
            if (myCeiling != null) System.out.println("ceiling "+myCeiling);
            System.out.println();
        }
    }

    public static void print(TreeSet<Integer> set) {
        for (Integer x: set)
            System.out.print(x+" ");
        System.out.println();
    }
}
```

In the first for loop, 20 integers in between 0 and 99 are generated and added to the set. Based on probability theory, the probability that all of these are unique is quite low. (It's about an 87% chance that there will be at least one repeat. See if you can calculate this on your own!) Thus, most times this code is run, there will be at least one duplicate and when the set is printed, no changes will occur.

In the following loop, we pick 10 random integers and then show the result of running 4 of the methods discussed before that deal with order: lower(), higher(), floor() and ceiling().

A good exercise would be to add some code that removes values, looks at the smallest and largest values, and removes those as well.

Problem: Dot Game Dominator

This problem is made up based on the game agar.io. In the game, the player is a dot and there are other dots on the screen. If the player intersects with a smaller dot, it "eats" that dot and grows larger. If the player intersects with a dot of the same size or larger, the player dies.

In this problem, you are given your initial size, as well as the size of all of the dots. We can think of these sizes as areas. Each time you eat a dot, your size grows by the size of the dot you just ate. For example, if your size is 10 and you eat a dot of size 7, then right afterwards, you'll be size 17.

The problem is to determine the fewest number of dots you have to eat to become strictly bigger than the largest dot. You also need to detect if this isn't possible.

The key observation is that if we want to minimize the number of dots we eat to get to some fixed size, we may as well eat the largest dots possible. Of course, as we grow, the largest dot we could eat changes. Thus, the idea is as follows:

1. Store all of the dots in a TreeSet.
2. At each step, find the largest dot strictly smaller than me. If there's no such dot and I'm not yet the largest dot, then I can never become larger than the largest dot.
3. Eat that dot (so remove it from the list of dots) and update my size.
4. If I am bigger than the biggest dot, stop, otherwise go back to step 2.

One key observation is that two opponent dots could be the same size, so we can't just do a TreeSet of Integer. Rather, we must do a TreeSet of an object and we can use object identification numbers to distinguish between equal sized dots. (So, implement Comparable and in the compareTo, break ties by the ID number so that no two different dots could be considered equal.)

Since the TreeSet methods need to compare with like objects, your player needs to also a Dot object (or you need to temporarily create one).

The problem description is here:

https://www.cs.ucf.edu/~dmarino/ucf/introjava/DotGame-TreeSet/Dot.pdf

Let's take a look at the full solution on the next page and then we'll walk through it.

```
import java.util.*;

public class dotgame {

    public static void main(String[] args) {

        Scanner stdin = new Scanner(System.in);
        int numCases = stdin.nextInt();
        for (int loop=0; loop<numCases; loop++) {

            long me = stdin.nextLong();
            int n = stdin.nextInt();

            // -1 is so the tiebreaker works the way I want it to.
            dot myDot = new dot(me, -1);

            TreeSet<dot> dots = new TreeSet<dot>();
            for (int i=0; i<n; i++) {
                long tmp = stdin.nextLong();
                dots.add(new dot(tmp, i));
            }

            int res = 0;

            while (dots.size() > 0) {

                dot largest = dots.last();

                if (largest.compareTo(myDot) < 0) {
                    break;
                }

                dot toEat = dots.lower(myDot);
                if (toEat == null) {
                    res = -1;
                    break;
                }
                else {
                    myDot.grow(toEat);
                    dots.remove(toEat);
                    res++;
                }
            }

            System.out.println(res);
        }
    }
}
```

```
class dot implements Comparable<dot> {

    private long size;
    private int ID;

    public dot(long sz, int myID) {
        size = sz;
        ID = myID;
    }

    public void grow(dot other) {
        size += other.size;
    }

    public int compareTo(dot other) {
        if (this.size != other.size)
            return Long.compare(this.size, other.size);
        return this.ID - other.ID;
    }
}
```

Let's start with the dot class. The grow method in included since the player grows when it eats other dots. In the compareTo, we first break ties by size and then if the sizes are the same, we break ties by ID. Since all the dots have IDs 0 to n-1, and we want to be the "smallest" dot of a particular size (since when we do the lower query, we definitely want to get a dot of a different size), to be safe, we MUST assign the player to an ID less than 0. This is why we -1 is used as the ID for myDot. (Another option would be to my myDot 0 and the enemy dots 1 to n.)

Now, we first read in all of the enemy dots and just add them to our TreeSet.

In the while loop, we first check to see if we've "won" by becoming the largest dot. If so, we break out. Otherwise, we find the largest dot that we could eat. If there's no dot we could eat, then we can't win and we also break out. Finally, if there's a dot to eat, we eat it and go to the next loop iteration. One by one, dots get removed and we grow larger.