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

TreeMap

We've seen both an unordered map (HashMap) and an ordered set (TreeSet). Naturally, it would make sense to have an ordered map (unless you are designing the Python language, but I digress...)

Java's ordered map is a TreeMap. Just like a HashMap, the TreeMap stores key-value pairs, mapping each key entered into it to a single value. The difference is that the keys are ordered just like items are ordered in a TreeSet. Thus, for a given key, we can quickly (in O(lg n) time where n is the number of items in the data structure) retrieve the next higher key or the next lower key. We can iterate through all of the keys in sorted order.

Let's take a look at some of the more important methods in the TreeMap class:

Constructor	Description
`TreeMap<K,V>()`	Creates an empty TreeMap.

Modifier and Type	Method	Description
`E`	`ceilingKey(E e)`	Returns the least key in this TreeMap >= e or null if there's none.
`void`	`clear()`	Empties the contents of this TreeMap.
`boolean`	`containsKey(E e)`	Returns true iff e is a key in this TreeMap.
`boolean`	`containsValue(E e)`	Returns true iff e is a value in this TreeMap.
`E`	`firstKey()`	Returns the first key in this TreeMap or null if there's none.
`E`	`floorKey()`	Returns the greatest key in this TreeMap <= e or null if there's none.
`V`	`get(Object key)`	Returns the mapped value for this key in this TreeMap.
`E`	`higherKey(E e)`	Returns the least key in this TreeMap > e or null if there's none.
`Set<K>`	`keySet()`	Returns the set of keys in this TreeMap.
`E`	`lastKey()`	Returns the last key in this TreeMap or null if there's none.
`E`	`lowerKey(E e)`	Returns the greatest key in this TreeMap < e or null if there's none.
`V`	`put(K key, V value)`	Maps key to value in this TreeMap.
`V`	`remove(K key)`	Removes the mapping associated with key and returns the corresponding value.
`int`	`size()`	Returns the number of elements in this TreeMap.

Let's go through these methods, starting with methods replicated from HashMap:

`put` – This method is the same as the corresponding method in the HashMap class. It maps the given key to the given value, overwriting any previous mapping for that key.

`get` – Just like HashMap, this will retrieve the value associated with the given key.

`remove` – This removes the entry in the map with the given key.

`size` – Retrieves the number of entries currently stored in the map.

`clear` – Removes all entries from the map.

`containsKey` – Returns true iff the given key has an entry in the map.

`containsValue` – Returns true iff the given value is mapped to one of the keys. **Note: In general, don't call this method. It runs in O(n).**

Each of these methods doesn't depend on the ordering of the keys and externally works the same as HashMap. Internally, these methods work differently to make sure that the other methods can work smoothly.

The following methods are in TreeMap but NOT HashMap and use the ordering of the keys in some way shape or fashion:

`firstKey` – Returns the lowest key of all the keys in the map.

`lastKey` – Returns the lowest key of all the keys in the map.

`ceilingKey` – Returns the smallest key in the map greater than or equal to the given key.

`floorKey` – Returns the largest key in the map less than or equal to the given key.

`higherKey` – Returns the smallest key in the map strictly greater than the given key.

`lowerKey` – Returns the largest key in the map strictly less than the given key.

Finally, to return the set of keys, we can do:

`keySet` – this returns the set of the keys.

This is useful if we want to loop through the keys in order.

Voting Example

In this example, we'll read through some votes for office, and at various requested times, we'll print out the current standings, sorted by the last name of the candidates. Formally, our input will look like this:

First line will have a single integer, *n*, representing the number of commands to process.
The following *n* lines will have the instructions for one command each.
These instructions will either start with the integer 1 (new vote) or the integer 2 (current standings request). If the instruction is 1, then this will be followed by a space and a string of uppercase letters indicating the individual receiving a vote. If the instruction is 2, then no further information will be given on the line.

For each instruction of type 2, a chart should be printed out, in alphabetical order of candidate, with the candidate's name and the number of votes they've received so far. Follow the output for each of these requests with a blank line.

Consider the following input case:

```
10
1 BERRY
1 ADAMS
2
1 ADAMS
1 SMITH
2
1 SMITH
1 SMITH
1 SMITH
2
```

The output for this input should be:

```
ADAMS 1
BERRY 1

ADAMS 2
BERRY 1
SMITH 1

ADAMS 2
BERRY 1
SMITH 4
```

Now, let's design a program to solve this problem. We'll use a TreeMap mapping String to Integer, since we want the names of the candidates to be sorted and each candidate should be mapped to their current number of votes.

When reading in a name, if no entry in the map exists for that candidate, we should map their name to 1 vote. If the name is already in the map, we should change their mapping to be one more than it was previously.

Finally, for printing, we can just loop through the names in order as follows:

```
for (String candidate: myMap.keySet())
    System.out.println(candidate+" "+myMap.get(candidate));
```

Let's take a look at the full program:

```
import java.util.*;

public class votes {

  public static void main(String[] args) {


    Scanner stdin = new Scanner(System.in);
    int numCmd = stdin.nextInt();

    TreeMap<String,Integer> myMap = new TreeMap<String,Integer>();

    for (int loop=0; loop<numCmd; loop++) {

      int type = stdin.nextInt();

      if (type == 1) {

        String name = stdin.next();

        if (myMap.containsKey(name))
          myMap.put(name, myMap.get(name)+1);

        else
          myMap.put(name, 1);
      }

      else {

        for (String candidate: myMap.keySet())
          System.out.println(candidate+" "+myMap.get(candidate));
        System.out.println();
      }
    }
  }
}
```

We create our map before the loop. When processing each command, we use containsKey to check if the current person previously got a vote. If so, we add 1 to their total, otherwise, we add them to the map to the first time with 1 vote. Finally, for status updates, we use keySet() and an iterator loop through the keys.

bst Example

This is a more advanced example, but relevant to students who have had or are taking Computer Science 1 (COP 3502). The full problem text is included here:

https://open.kattis.com/problems/bst

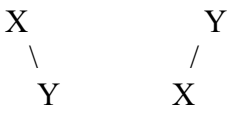
In the problem you are given a sequence of integers inserted into a binary search tree. These integers will range from 1 to a given positive integer n, and there will be no duplicates. For each insert, you have to compute the depth (how far from the root node) the node storing the value is placed. In addition, you have to report the sum of these depths upto that point in time.

The tree itself might be too big to fully simulate the process because n could be as large as 300,000 and a skewed three of size 300,000 could lead to $300{,}000^2/2$ simple operations, which is too much. Thus, we require a way to compute the depth of each node in the tree without actually ever inserting it!

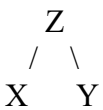
A couple of key observations:

1. Whenever a node is interested into a BST, either it's inserted directly to the right of the item that is the largest item smaller than it currently in the tree, OR it's inserted directly to the left of the item that is the smallest item greater than it currently in the tree.

2. Between those two possibilities, it's guaranteed that the one that will always happen is the one that is more deep into the tree. To see this, note that the two values in question (greatest item less than the newly inserted value and least item greater than newly inserted value) must be in a direct line in the tree like this (X is the item less, Y is the item greater than the inserted value):

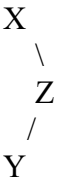


It's not possible for something like this to happen with X and Y being the two designated values:



because if this were the design, then X and Y could NOT be the two closest values, instead, our inserted value must be in between X and Z, or in between Z and Y.

Alternatively, we could have some sort of "elbow bend" like so:



In this picture the item to be inserted is greater than X and less than Y. As we can see, if this is the case, during the insertion process, the item will first pass through X, going right, and then later hit Y below. Y may have a right child, but can not have a left child in this scenario.

Either way, we see that both of these "closest" values to the inserted value must lie on the same path down the tree from the root, so it must always be the case that we insert below the deeper node between the two.

This leads us to our solution where we map each item in the tree to its depth. Whenever we insert an item directly below a previous item, it's depth is 1 more than the previous item's depth.

To get a fast run time on Kattis, fast I/O is used: both the BufferedReader class and the StringBuffer class are utilized.

We'll present the full solution on the next page, followed by its explanation.

```
import java.util.*;
import java.io.*;

public class bst {

	public static void main(String[] args) throws Exception {

		// Faster input because this problem has lots of it.
		BufferedReader stdin = new
				BufferedReader(new InputStreamReader(System.in));
		int n = Integer.parseInt(stdin.readLine());

		// Here's our map.
		TreeMap<Integer,Integer> map = new TreeMap<Integer,Integer>();

		// For faster output; we'll learn about this soon.
		StringBuffer sb = new StringBuffer();

		// Get first item.
		int root = Integer.parseInt(stdin.readLine());
		map.put(root, 0);
		sb.append("0\n");
		long res = 0;

		// Go through the rest.
		for (int i=0; i<n-1; i++) {

			// Next value to insert into the tree.
			int value = Integer.parseInt(stdin.readLine());

			Integer low = map.lowerKey(value);
			Integer high = map.higherKey(value);
			int newd = -1;

			// high is parent.
			if (low == null)
				newd = map.get(high) + 1;

			// low is parent
			else if (high == null)
				newd = map.get(low) + 1;

			// Get node further down.
			else
				newd = Math.max(map.get(low), map.get(high)) + 1;

			// Update result and add to the output buffer.
			res += newd;
			map.put(value, newd);
			sb.append(res+"\n");
		}

		// Ta da!
		System.out.print(sb);
	}
}
```

We first read in the number of nodes and set up our map, which will map the value of a node to its depth in the tree.

We set up a StringBuffer to store our output without actually printing it out as we go.

We read in the root of the tree and set its depth to 0 and set our running tally of sums to 0 as well.

Then we go through the following n – 1 insertions.

For each insertion, after we read in the value, we look in the map to get the lower and higher keys. If either is null, this means we must insert below the other (non-null) node. Which means its depth will be 1 more than the depth of the only non-null node. Alternatively, if both lower and higher keys exist, we want the maximum of the two associated depths and our node's depth will be 1 more than that.

Finally there is some basic bookkeeping at the end of each loop iteration.

Once we are done, we output the whole StringBuffer!