Lab 7: BSTMap

Introduction

In this lab, you'll create **BSTMap**, a BST-based implementation of the Map61B interface, which represents a basic tree-based map. You will be creating this completely from scratch, using the interface provided as your guide.

After you've completed your implementation, you'll compare the performance of your implementation to a list-based Map implementation `ULLMap` as well as the built-in Java `TreeMap` class (which also uses a BST).

BSTMap

Create a class **BSTMap** that implements the **Map61B** interface using a BST (Binary Search Tree) as its core data structure. You must do this in a file named `BSTMap.java`. Your implementation is required to implement all of the methods given in **Map61B** *except* for `remove`, `iterator` and `keySet`. For these methods you should throw an `UnsupportedOperationException`.

Your code will not compile until you create the `BSTMap` class and you implement all the methods of **Map61B**. You can implement methods one at a time by writing the method signatures of all the required methods, but throwing `UnsupportedOperationExceptions` for the implementations, until you get around to actually writing them.

Your `BSTMap` should also add an additional method `printInOrder()` (not given in the **Map61B** interface) that prints out your **BSTMap** in order of increasing Key. We will not test the result of this method, but you will find this helpful for testing your implementation!

In your implementation you should assume that generic keys `K` in `BSTMap<K,V>` extend Comparable. In other words, you can assume that generic keys `K` have a `compareTo` method. This can be enforced in Java with a bounded type parameter. Consider the example below, taken from the Oracle docs:

```
/*
 * Bounded type parameters allow you to invoke methods de
 * The `isEven` method invokes the `intValue` method defi
 * `Integer` class through `n`.
 */

public class NaturalNumber<T extends Integer> {

        private T n;

        public NaturalNumber(T n)  { this.n = n; }

        public boolean isEven() {
                return n.intValue() % 2 == 0;
        }

        // ...
}
```

We also recommend that you use a private nested `BSTNode` class to help facilitate your implementation. How you design and use this inner class is up to you!

You can test your implementation using `TestBSTMap.java`.

The following resources might prove useful:

- Lecture 16 slides.
- BST code from pages 109 and 111 of Data Structures Into Java, from our course resources page.

- BST code from our optional textbook.
- `ULLMap.java` (provided), a working unordered linked list based **Map61B** imlementation.

So… How Fast Is It?

There are two interactive speed tests provided in `InsertRandomSpeedTest.java` and `InsertInOrderSpeedTest.java`. Do not attempt to run these tests before you've completed **BSTMap**. Once you're ready, you can run the tests in IntelliJ.

The `InsertRandomSpeedTest` class performs tests on element-insertion speed of your **BSTMap**, **ULLMap** (provided), Java's built-in **TreeMap**, and Java's built-in **HashMap** (which you'll explore more in the next lab). It works by asking the user for a desired length of each String to insert, and also for an input size (the number of insertions to perform). It then generates that many Strings of length as specified and inserts them into the maps as `<String,Integer>` pairs.

Try it out and see how your data structure scales with the number of insertions compared to the naive and industrial-strength implementations. Remember that asympototics aren't representative on small samples, so make sure your inputs are sufficiently large if you are getting a confusing trend. Record your results in a file named `speedTestResults.txt`. There is no standard format required for your results, and there is no required number of data points.

Now try running `InsertInOrderSpeedTest`, which behaves similarly to `InsertRandomSpeedTest`, except this time the Strings in `<String, Integer>` key-value pairs are inserted in lexicographically-increasing order. If you observed anything interesting (hopefully you did), you should discuss it with your fellow students and/or TA.

Optional Exercises

This will not be graded, but you can still receive feedback on the autograder.

Implement the methods `iterator()`, `keySet()`, `remove(K key)`, and `remove(K key, V value)`, in your **BSTMap** class. When implementing the `iterator` method, you should return an iterator over the *keys*. Implementing `remove()` is fairly challenging. For an extra challenge implement `keySet()` and `iterator` without using a second instance variable to store the set of keys.

For `remove`, you should return null if the argument key does not exist in the **BSTMap**. Otherwise, delete the key-value pair (key, value) and return value.

Lab Debrief and Submission

At the end of lab, your TA will go over the reference solution. This will be helpful if you haven't finished the lab, since we don't want you to be stuck working on lab too much outside of lab. (This is also an incentive for you to go to lab!)

Make sure to submit your completed `BSTMap.java` and `speedTestResults.txt`, and submit through git and Gradescope as usual.

Optional Asymptotics Problems

Given `B`, a **BSTMap** with `N` key-value pairs, and `(K, V)`, a random key-value pair, answer the following questions.

Unless otherwise stated, “big-Oh” bounds (e.g. `O(N)`) and “big-Theta” bounds (e.g. Θ(`N`)) refer to the **number of comparisons** in the given method call(s).

For questions 1-7, state whether the statement is true or false. For question 8, give a runtime bound.

1. `B.put(K, V)` ∈ O(log(`N`)).
2. `B.put(K, V)` ∈ Θ(log(`N`)).
3. `B.put(K, V)` ∈ Θ(`N`).
4. `B.put(K, V)` ∈ O(`N`).
5. `B.put(K, V)` ∈ O(`N`^{2}).
6. Let `g(N)` be the average number of comparisons required to complete `N` random calls to `B.put(K, V)` followed by `B.containsKey(K)`. Then, `g(N) ~ 2(ln(N))`.

   ```
   Note: We write g(N)~f(N) to represent that g(N)/f(
   ```

7. For key `C` != `K`, running both `B.containsKey(K)` and `B.containsKey(C)` ∈ Ω(log(`N`)).
8. Let **BSTMap** `b` be comprised of a `root` Node (Key, Value pair) and two **BSTMap** subtrees called `left` and `right`. Further, assume the method `numberOfNodes(BSTMap b)` returns the number of nodes of a **BSTMap** rooted in `b.root` and has a running time of Θ`(n)`, where `n` is the number of Nodes in the **BSTMap** rooted in `b`. What is the running time (in big O notation) of `mystery(b, z)` for some positive integer `z`? Give the tightest bound you can assuming `b` has `N` nodes. Your

answer should not contain any unnecessary multiplicative constants or additive factors.

```
public Key mystery(BSTMap b, int z) {
    if (z > numberOfNodes(b) || z <= 0)
        return null;
    if (numberOfNodes(b.left) == z-1)
        return b.root.key;
    else if (numberOfNodes(b.left) > z)
        return mystery(b.left, z);
    else
        return mystery(b.right, z-numberOfNodes(b.l
}
```