CS 331 Lab: Binary Search Trees
Fall, 2010

1 Objectives

In this lab you will create a binary search tree. Your objectives:

- Be able to use the `find` and `add` traversal patterns.
- Be able to write `delete` for the three cases.
- Have experience using a dictionary style container.
- Use an iterator to verify the correct placement of data in the structure.

2 Binary Search Trees

As we discussed in lecture, a BST node has two pointers: `left` and `right`. During a search, we will be looking for data `x`, and looking at node `n`. If `x < n.data`, we would next go down the left side. Otherwise, we would go down the right side.

2.1 Dictionaries

The structures you've implemented in lab so far have all been (multi-)sets. The structure either contains an item, or it doesn't. This time, we are going to implement a dictionary. A dictionary node has two elements, a key of type `K` and a value of type `V`. When adding or searching for an element, the algorithm does all comparisons on the keys. When it finds an element, it returns the value. An example of this kind of structure is a phone book. The key is the person's name, and the value is the phone number.

3 Given Files

You are given 4 files for this lab. They are in the `bst-lab` directory of your repository. Do a `git pull` on to find them.

- `BST.java` — A brutally skeletal BST implementation.

- `Iterator.java`
- `Makefile` — the scripts needed to compile your files and run the tests. It supports the following targets:

  compile Compiles everything.

  tests Tests everything.

- `TestBST.java` — another skeleton test file. There are some strings and numbers predefined, but you'll almost certainly want to use more data than that.

4 Your Work

You will need to implement at least the following methods. You may add more if you feel they are necessary.

constructor This just creates an empty tree.

int size() Return the number of elements stored in the tree. This should run in $\mathcal{O}(1)$ time.

void add(K key, V value) Insert an element into the tree. There will be only one place in the tree where it should be inserted. This should run in $\mathcal{O}(\lg n)$ time. If the key already exists in the tree, replace the value with the new one.

V find(K key) Try to find key `K`, and return value `V` if it exists, `null` otherwise. This should run in $\mathcal{O}(\lg n)$ time.

void delete(K key) Delete element `k`, if it exists. This should run in $\mathcal{O}(\lg n)$ time. There are at least four kinds of cases you need to test.

K revFind(V value) This is a "reverse phone number lookup" method. You supply the value, and get the key back. The values are not in any order, so you'll have to use one of the traversal patterns to find it. For this lab, use depth-first search.

Iterator<K> mkBFSIterator() One of the things you want to do is make sure that the elements are placed correctly in the tree. If you make a breadth-first search iterator, you can iterate through the elements of the tree and see if they are in the order you expect. You may want to copy your `Queue.java` over here and make use of that. Be sure to add it to the repository if you do.

5 Tests

Of course you'll need to write tests. Tests are the deodorant of software. You can go without, but it'll just stink.