CS 345 (Fall 18): Computer Organization

Project #3

BSTs and Generics

due at 5pm, Thu 18 Oct 2018

1 Introduction

In this project, you'll be implementing a BST, continuing to practice with generating `.dot` files, and getting some experience with Java generics. And of course, you're probably not surprised: I'm going to have you write it in the `x=change(x)` style.

The data structure that you will write is a simple BST. It will **not** include any self-balancing for this project, but it **will** include delete (including case 3) and also **manual** rotations. You will also be maintaining a simple metadata field for each node, which is the count of all of the nodes in that subtree; you'll have to update it every time that you insert, delete, or rotate a node.

2 Java Generics

We've discussed Java generics in class - but now, you are going to make use of them. The key idea of Java generics is that classes and interfaces can have 'type parameters,' which don't change **how** the code works, but instead change **what** it works on. Type parameters are declared in angle brackets, next to the type name, like this:

```
public class Foo<T>
```

A class can have multiple type parameters, and the `extends` keyword can be used to limit what sorts of types are allowed. Take a look at the file `Proj03_BST.java`, which I've provided for this project. The declaration for this interface is as follows:

```
public interface Proj03_BST<K extends Comparable, V>
```

This means that there are two parameters, `K,V`. The `K` parameter is limited - the type that you supply there **must** implement the `Comparable` interface. But the `V` parameter can be **any** Java class.

Of course, `Proj03_BST` is an interface which represents a BST, and the type parameters represent the key and value types. The key must be comparable (so that we can use it to sort the values into the tree), but the value type can be anything at all - all we need is to be able to call `toString()` to print it out.

Now, look through the file, and example each of the required methods. The types `K,V` are used throughout: `K` is used to represent keys for searching through the tree, and `V` is the type of the value that is returned when we perform a search.

Next, look at the `BSTNode` class, which I've also provided. This is a basic BST node, with key and value fields, a count, and left and right pointers. Look more closely at the left and right pointers: this shows you how to **use** a generic class:

```
Proj03_BSTNode<K,V> left,right;
```

You see that we can provide the various type parameters in angle brackets, as though it's a funny sort of function. Thus,

```
Proj03_BSTNode
```

names a generic class, while

```
Proj03_BSTNode<K,V>
```

names a particular instantiation of that class, with certain type parameters. In this case, since we have nodes pointing to other nodes (all with the same type parameters), we simply pass along the type parameters.

But look at the `main()` method in `TestDriver`, which I've provided for you. It builds three different trees: one that uses `Integer` keys and `String` values, one `Integer/Integer`, and one `String/Integer`. The variables are declared like this:

```
Proj03_BST<Integer,String>  is = null;
Proj03_BST<Integer,Integer> ii = null;
Proj03_BST<String, Integer> si = null;
```

We initialize these references by building new Java objects. The `new` operator needs us to provide the full type (with parameters), like this:

```
is = new Proj03_SimpleBST_example<Integer,String>("debug_example_is");
```

Finally, look at the file `Proj03_SimpleBST_student.java`, which is the skeleton I've provided of the code you need to write. I've given you an example of the `x=change(x)` style, using `BSTNode` objects which include all of the generic parameters:

```
private Proj03_BSTNode<K,V> set_helper(Proj03_BSTNode<K,V> oldRoot,
                                       K key, V value)
{
    // TODO: implement me
}
```

Notice how the first parameter `oldRoot` is a `BSTNode` reference, with the `K,V` type parameters indicating the type of the keys and values stored in this type of tree. The next two parameters are the actual key that we're searching for, and the value to store once we find the proper location. And what does it return? Exactly the same type as the `oldRoot` parmeter.

There are lots more details about Java generics that are worth studying - but this basic introduction should be enough to let you implement a basic tree.

3 How These Testcases Work

For this project, I have provided three critical pieces that allow you to test your code (in addition to the grading script itself):

- `Proj03_TestDriver.java`

  This provides the `main()` functions for all testing. It will build three trees, with different type parmeters. If you pass it the `example` command-line argument, then it will build the trees using the example code that I've provided; if you don't pass that argument, then it will build the trees from your code.

  After it builds the three trees, it reads the testcase file (reading from `System.in`, parses it, and performs whatever operations the testcase requires.

  If you want to run this yourself (without the grading script), compile it, and then run it like this to run my example code:

  ```
  java Proj03_TestDriver example < testcaseFilename
  ```

  (To run your code for comparison, simply omit the `example` argument.)

- `Proj03_SimpleBST_example.class`

  This is my example code, pre-compiled. It should pass all of the requirements of the spec. Your goal is to make your code work exactly the same way, on every testcase you can come up with.

- `test_01`

  This is a simple testcase, which shows you the basic format of a testcase and most of the available commands.

3.1 Testcase Format

The format of the testcase is a text file, with one command per line. Each line starts with a "tree name," which is one of `is,ii,si`. This tells the code which of the three trees will be accepting this line. This is followed by a command and then zero, one, or two parameters. The various commands are very similar (sometimes identical) to the methods in the `Proj03_BST` interface.

The available commands are:

- `set <key> <val>`
- `get <key>`
- `remove <key>`
- `rotateLeft <key>`
- `rotateRight <key>`

- `getSize`
- `inOrder`
- `postOrder`
- `dotFile`

3.2 Writing Your Own Testcases

I've provided a single example testcase, `test_01`. However, if you want to test your code well, you **must** come up with more testcases! You should write your own, and I **encourage** you to share with on Piazza, so that the whole class can benefit.

4 Miscellanous Requirements

4.1 No Global Variables

Hopefully, you're used to this by now!

4.2 BST Delete Case 3

When in Delete Case 3, you must pull up the **predecessor.**

4.3 `O(1)` cost to update the count

First of all, since the count is stored in every node, you must keep it up-to-date every time that you modify the tree. **You must** do this as part of the normal `x=change(x)` code; don't return a tree to your caller until the count field is up-to-date.

Second, the cost of updating the count in **one** node must be $O(1)$. If you are doing something more complex (such as traversing the tree to calculate the count), then you must rethink your strategy.

Third, you must **only** update the count in nodes that are part of the path you updated. You **must not** traverse other parts of the tree!

4.4 `x=change(x)`

You must use the `x=change(x)` style to implement the following operations in your BST:

- `set()`
- `remove()`
- `rotateLeft()`

- `rotateRight()`

You are **not** required to use this style for the other methods - it's up to you. In some cases (like `get()`) I think it's actually easier to do it other ways.

REMEMBER: Not every programmer likes the `x=change(x)` style - even some of the faculty don't! And even I think that it doesn't work well for all types of trees. But, for many trees, it will make your recursive code **easier to write.** So I require it now, for some of the projects - after this class is over, you can choose whether or not to use it in the future.

5 Base Code

Download all of the files from the project directory

`http://lecturer-russ.appspot.com/classes/cs345/fall18/projects/proj03/`

If you want to access any of the files from Lectura, you can also find a mirror of the class website (on any department computer) at:

`/home/russelll/cs345_website/`

6 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

6.1 Testcases

You can find a set of testcases for this project at
`http://lecturer-russ.appspot.com/classes/cs345/fall18/projects/proj03/`

See the descriptions above for the precise testcase format, and also for information about how to run the two test driver classes.

6.2 Other Testcases

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcaes of your own, in order to better test your code.** You are also encouraged to share your testcases on Piazza!

6.3 Automatic Testing

We have provided a testing script (in the same directory), named `grade_proj03`. Place this script, all of the testcase files, and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

7 Turning in Your Solution

Turn in the following file(s):

`Proj03_SimpleBST_student.java`

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.