Programming Project 3
CSci 1913: Introduction to Algorithms, Data Structures, and Program Development
Fall 2014

Last revision April 18, 2016

0. Introduction.

In this programming project, you will implement a binary search tree (BST) whose keys are randomly scrambled. This may result in a BST with a small height, which can be searched more efficiently than one with a large height. This is the last project of this course. There were originally intended to be four projects, but there isn't enough time to assign another.

1. Theory.

Suppose that we build a BST by adding n key-value pairs, one at a time. Also suppose that we add pairs with the keys in increasing or decreasing order. Then we'll make a BST whose height is n. Such a *degenerate BST* requires $O(n)$ comparisons to find the value associated with a key. In contrast, a *balanced BST,* whose height is at a minimum, requires only $O(\log_2 n)$ comparisons.

We may be able to construct a better balanced BST, with a smaller height, by randomly scrambling the key-value pairs before we add them. When we are given a new key-value pair, we don't add it directly to the tree. Instead, we add it temporarily to a *buffer,* a data structure that holds objects for later processing. The buffer is made up of two arrays, called *keys* and *values*. We add the key to the array *keys,* and we add the value to the array *values,* so that each key has the same index as its corresponding value.

We continue adding keys and values to the buffer in this way, until the buffer is full, or until we want the value that is associated with a key. If either of those things happens, then we must *flush* the buffer by adding its keys and values to the BST. To flush the buffer, we first randomly scramble its keys and values. We do that by using the **Durstenfeld-Fisher-Yates** algorithm, named for its inventors. This algorithm is shown below in pseudocode, where a is the array to be scrambled, and n is its length.

```
for $i$ from 0 to $n - 2$
{
  $j$ = a random integer such that $0 \le j < n - i$
  exchange $a[i]$ and $a[i + j]$
}
```

We must scramble the *keys* and *values* arrays in the same way, so that each key still has the same index as its corresponding value. After the arrays are scrambled, we add their key-value pairs to the BST, one at a time. We empty the arrays after we've added all the pairs.

One problem remains. We'd like to implement the buffer so it's invisible to the user of the BST. That is, we want the BST to act the same way whether we use a buffer or not. Unfortunately, a BST with a buffer may handle keys differently from one without a buffer.

Suppose it doesn't have a buffer. Then we add a key k and value v directly to the BST, making a new node. Later, we add the same key k again, but with a different value v'. That changes the node so it now contains k and v'. As a result, if we ask for the value of key $k,$ then we get v'.

Now suppose it does have a buffer. Then we add a key k and its value v. Later, we add the same key k again, but with a different value v'. The buffer now has two entries for the key k: one with $v,$ and the other with v'. After the buffer is scrambled, we don't know which entry will be processed first: which one will make a new node, and which one will change the node. As a result, if we ask for the value of key $k,$ then we may get either v or v'.

There's a way to fix this, by making the buffer slightly more complex. However, to keep this project simple, we'll say that after a node is added to the BST, we aren't allowed to change its value. Any attempt to do so will throw an exception. This will prevent the situation described above from ever happening.

2. Implementation.

You must write a class called `SBST`, which stands for Scrambled Binary Search Tree. Its keys must be Java `String`'s, and their values must be instances of the generic class `Value`, so that `SBST` must look like this, with your code in place of the three dots.

```
class SBST<Value>
{
   ⋮
}
```

The class `SBST` must have a private nested class called `Node` whose instances represent the nodes of a BST. The class `Node` must have slots called `key`, `value`, `left` and `right`, like the BST nodes discussed in the lectures.

The class `SBST` must have two private arrays, called `keys` and `values`, that represent the buffer. Don't try to use only one array: that will not work. You may need more private variables in addition to these.

The class `SBST` must also implement the following methods. You may need other methods along with these. To make your project easier to grade, your methods must work as described below. You must also use the same names for methods and variables as shown here.

```
public SBST(int size)
```

Constructor. If `size` is negative, then throw an `IllegalArgumentException`. Otherwise, initialize a new `SBST` whose buffer can contain at most `size` key-value pairs.

```
private void flush()
```

Flush the buffer. First scramble the elements of the arrays `keys` and `values` as described in the previous section. Then add the key-value pairs to a BST, using the method `putting` (see below). Finally, empty the arrays.

```
public Value get(String key)
```

If the buffer is not empty, then flush it. Then search the BST for a `Node` whose `key` slot is equal to the parameter `key`. If you find such a `Node`, then return its `value` slot. If there is no such `Node`, then throw an `IllegalArgumentException`.

```
public int height()
```

If the buffer is not empty, then flush it. Then compute the height of the BST and return it.

```
public void put(String key, Value value)
```

If `key` is `null`, then throw an `IllegalArgumentException`. If the buffer is full, then flush it. Add the parameter `key` to the array `keys` and the parameter `value` to the array `values`, as discussed in the previous section.

```
private void putting(String key, Value value)
```

Add a new `Node` to the BST that contains the parameters `key` and `value`. If there is already such a `Node` in the BST, then throw an `IllegalStateException`.

To write the method `flush`, you will need a random number generator. You must use the class `Random` from the Java library. To do that, put this line at the start of your program.

```
import java.util.Random;
```

To generate random numbers, make an instance of `Random` and keep a pointer to it in a private variable. Use the same instance of `Random` throughout your code. If *r* is an instance of `Random`, then the method call *r*`.nextInt()` will return a randomly generated `int`. This `int` may be negative, and may have a large absolute value. To turn the `int` into an array index for the Durstenfeld-Fisher-Yates algorithm, you may need to use the method `Math.abs`, and the operator `%`.

3. Example.

You must also write a driver class to test your class `SBST`. You can use the following class `Scrambled`, or you can make up your own. The class `Scrambled` adds some Java reserved names to a `SBST` in increasing alphabetical order, associating each one with an `int`. Then it writes the height of the `SBST` and displays the reserved names and their `int`'s.

```
class Scrambled
{
  private final static String[] reserved =
   { "abstract",     "assert",    "boolean",     "break",
     "byte",         "case",      "catch",       "char",
     "class",        "const",     "continue",    "default",
     "do",           "double",    "else",        "extends",
     "final",        "finally",   "float",       "for",
     "goto",         "if",        "implements",  "import",
     "instanceof",   "int",       "interface",   "long",
     "native",       "new",       "package",     "private",
     "protected",    "public",    "return",      "short",
     "static",       "super",     "switch",      "synchronized",
     "this",         "throw",     "throws",      "transient",
     "try",          "void",      "volatile",    "while" };

  public static void main(String [] args)
  {
    SBST<Integer> sbst = new SBST<Integer>(30);
    for (int index = 0; index < reserved.length; index += 1)
    {
      sbst.put(reserved[index], index);
    }
    System.out.println(sbst.height());
    for (int index = 0; index < reserved.length; index += 1)
    {
      System.out.format("%02d %s", sbst.get(reserved[index]), reserved[index]);
      System.out.println();
    }
  }
```

If `sbst` were an ordinary BST, we would expect it to become degenerate, with a height of 48. However, when we run the program, we get the following (some output was omitted to save space).

```
11
00 abstract
01 assert
02 boolean
    ⋮
45 void
46 volatile
47 while
```

We get a BST whose height is only 11. This compares favorably with the theoretical minimum height of a BST with 48 keys, between 5 and 6. The height of your tree may be different, and still be correct, because your random number generator may have produced a different sequence of `int`'s than this one. This driver class does not adequately test the class `SBST`. To be safe, you should also run tests of your own.

4. Deliverables.

Unlike the lab assignments, you are not allowed to work with a partner on this project. IT MUST BE DONE BY YOURSELF ALONE. Although you can discuss the assignment in a general way with others, you are not allowed to get help from anyone, except Prof. Moen or the course TA's.

The project is worth 50 points. It will be due at midnight after the last lecture, which is May 6, 2016. You must turn in Java code for your `SBST` class. It must include a constructor (5 points), along with the methods `flush` (10 points), `get` (10 points), `height` (10 points), `put` (5 points), and `putting` (10 points). You must also turn in code for your driver class, and any output it produces, even though you don't get points for them. Put the code for both classes in one file, and put the output in a comment at the end of the file. If you have questions about how to turn in the project, then please ask your lab TA.