

This lab has **five (5)** checkpoints.

Learning Outcomes

Upon completing this lab, students should be able to

- Take apart an integer into its sequence of bits.
- Use JUnit tests to build a BST with insert and delete.

Introduction

1. Consider the *bits* that make up an *int*: if $\text{int } x = 5$, then x is stored as the bit sequence $1\ 0\ 1$ or $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ (yes, there are a bunch of leading zeros; we are going to ignore those for the moment).

Here is a small collection of non-negative integers and their bit patterns after the leading zeros:

0	0	only leftmost 0
5	101	
11	1011	
22	10110	
31	11111	
32	100000	

Compare 11 and 22.

Also compare 31 and 32.

You are not yet learning binary arithmetic but you can see some patterns emerging. We will look at converting a *binary* integer into a *String* of 0 and 1 characters.

The simplest algorithm to convert an *int* to a string of 0/1 characters uses the remainder modulo 2 to pick the character and divides n by 2 until it is zero.

```
String asBitString(int n) {
    String b = ""; // retVal
    if (n == 0) b = "0";
    else
        while (n > 0) {
            b = "" + (n % 2) + b; // prepend as a string
            n /= 2;
        }
    return b;
}
```

b	n	n % 2
""	11	1
"1" + "" = "1"	5	1
"1" + "1" = "11"	2	0
"0" + "11" = "011"	1	1
"1" + "011" = "1011"	0	---

- (a) Convert, showing your work (the values of b , n), each of the following non-negative integers to bit strings:
 - i. 0
 - ii. 6
 - iii. 17
 - iv. 27
 - v. 33
- (b) Since each position in a binary number is a *power of two*, you can convert the other way by adding up powers of two. Given the following powers, convert the bit strings back to decimal.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32

- i. 1010
- ii. 1111
- iii. 10
- iv. 100100
- v. 110011

✓ Show the lab instructor your work and answers.

2. Initialize a Gradle application project in a new directory. Also initialize git in the root of the project.

The name of the project: `treeTest`. Starting package: `bst`.

Go ahead and delete `bst.App`. Rather than a client, this lab will use JUnit tests to run the program.

Delete everything in `src/test/java/bst`. Don't delete the directory, just the `AppTest.java` file in it.

(If you want, you can remove the `resources` directories in the `main` and `test` trees; they will not be used but they are empty so it is not a big deal to leave or remove them.)

[This space intentionally left w/o a checkpoint.]

3. Write a new class: `main.java.bst.BST.java`. This is a *binary search tree* (bet you guessed from the name; isn't consistent naming nice?). Declare a nested `BSTNode` class: the data is of type `int`, the fields are visible to the class. No constructor, yet.

Write a default *constructor* that does *nothing*. You have no fields and no active code yet. You can make sure that your code builds with `gradle build`.

Write a new *test class*. This is a collection of test methods. The class is `src/test/java/bst/BSTTest.java`.

¹ The initial content of the file is

```

1 package bst;
2
3 import org.junit.jupiter.api.Test;
4 import static org.junit.jupiter.api.Assertions.*;
5
6 public class BSTTest {
7     @Test
8     void testConstructor() {
9         BST bst = new BST();
10        assertNotNull(bst);
11    }
12 }
```

- 3 Bring in the `Test` annotation. Note: the first time you build after this code is included, `gradle` will download a copy of the `junit.jupiter` library. This is why the name of the library is in the `settings.gradle` file.

- 4 Usually we do not use the star notation for import. Here we are getting a huge number of `assert*` functions so that admonition is relaxed. What does `static import` mean? That the functions are effectively global functions and are linked in at compile rather than at run-time.

- 7 The `@Test` annotation tells the *test runner* (part of JUnit) that this function is a function to run as part of the test.

JUnit/gradle will, by default, create an HTML test report. While this can be helpful on a large project, looking there is overkill for a project of this size. ² There is a plugin called `test-logger` that gives you almost the IDE's worth of information in the console.

Modify the plugin task in `build.gradle` to read

```

plugins {
    // Apply the application plugin to add support for building a CLI application in Java.
    id 'application'
    id 'com.adarshr.test-logger' version '4.0.0'
}
```

¹Test classes should be in a *package* matching that of the class they test. It is convention to name them something like `Test*.java` or `*Test.java`; while this is not strictly required, it documents the reason for the class to exist.

²We will not turn off the HTML report. Point a Web browser at `app/build/reports/tests/test/index.html` and you can see the fancy version of the test summary.

Build the project with `gradle build` (if it says everything was up to date, run `gradle clean` and build it again). Notice that the following should be part of the output:

```
> Task :app:test

bst.BSTTest

    Test testConstructor() PASSED

SUCCESS: Executed 1 tests in 313ms
```

This tells you the test was run and PASSED.

✓ Show the lab instructor your working tree, your test code, `gradle clean`, and `gradle build` with the passing test.

4. Add a root field to BST and initialize it in the constructor.

Set up the following *public* functions in `BST.java`:

```
class BST {
    public boolean isEmpty() { return false; }
    public int size() { return 0; }
    public void add(int n) { } // do nothing
    public void delete(int n) { }
    public String toString() { return ""; }
}
```

Do **not** implement the private paired functions, *yet*. Just have each function return `false` or `0` for the moment. (And ignore its parameters.)

You will add tests to your test class, one at a time:

```
class BSTTest {
    ...
    @Test
    void testIsEmpty() {
        BST bst = new BST();
        assertTrue(bst.isEmpty());
        bst.add(10);
        assertFalse(bst.isEmpty());
    }
    ...
}
```

Now, when you run this, it passes the constructor test but fails the new test on the first *assertion*: since `BST.isEmpty()` always returns `false`, `assertTrue` fails:

```
> Task :app:test FAILED

bst.BSTTest

    Test testIsEmpty() FAILED

org.opentest4j.AssertionFailedError: expected: <true> but was: <false>
    at app//bst.BSTTest.testIsEmpty(BSTTest.java:16)

    Test testConstructor() PASSED
```

```
FAILURE: Executed 2 tests in 324ms (1 failed)
```

Note that the *expected* and *actual* values are reported in the console.

✓ Show the lab instructor your working tree, your test code, `gradle clean`, and `gradle build` with the **failing** test.

5. Make the test pass: this requires making `BST.isEmpty()` work correctly (the first assertion will pass) **and** making `BST.add(int n)` work (since the second assertion counts on a change in state due to the call to `add`).

Go ahead and write the `add` recursive helper function, fix up `isEmpty`, and watch your test pass.

Now you can test `BST.size()` (see below), watch it *fail* (it is hardwired to return 0), and then fix it up with a helper function. The test should be something like this:

```
class BSTTest {
    ...
    @Test
    void testSize() {
        BST bst = new BST();
        int expected = 0;
        int actual = bst.size();
        assertEquals(expected, actual); // note parameter order

        bst.add(13);
        assertEquals(1, bst.size());
        bst.add(6);
        assertEquals(2, bst.size());
        bst.add(20);
        assertEquals(3, bst.size());
    }
    ...
}
```

We know that `add` updates the size correctly. We have not tested the results. Lets assume that `toString` returns the numbers, in order, with a new line after each.

```
class BSTTest {
    ...
    @Test
    void testAdd() {
        BST bst = new BST();

        bst.add(13);
        bst.add(6);
        bst.add(20);
        assertEquals("6\n13\n20\n", bst.toString());
    }
    ...
}
```

Your code should now pass four tests:

```
> Task :app:test
```

```
bst.BSTTest
```

```
Test testAdd() PASSED
Test testSize() PASSED
Test testIsEmpty() PASSED
Test testConstructor() PASSED
```

SUCCESS: Executed 4 tests in 315ms

✓ Show the lab instructor your working tree, your test code, `gradle clean`, and `gradle build` with the passing test.

6. Now you get to write a new test case on your own: Write a test case that uses `add` and `delete` to change the contents of the BST and then use the expected value of `toString` to check that it works.

Initially: add one integer, delete it. Make sure it returns `true` and that the string is empty.

Then add a couple, remove one that isn't there (since that should return `false` and leave the tree unchanged). Then remove one with a single child. Then build the tree up (you can start with an empty one, if you like), build a tree with a node with two children that is not the root. Delete the integer in it.

Yes, the testing requires that you understand the tree that is being constructed so that you test all the necessary cases.

✓ Show the lab instructor your working tree, your test code, `gradle clean`, and `gradle build` with the passing test.

7. [Not a separate checkpoint but necessary] Add and commit your full `gradle` project (if you `git add` just the root directory, `..`, `git` will add all subdirectories recursively).

Create a repository in your Gitea turn-in organization, associate it with your lab work, and push the repo. This will be done at the end of every lab.