CSE 12 Spring 2020 PA7 - Binary Tree and Runtime Analysis (100 Points)

Due date: Monday, May 18th @ 11:59PM PDT
(Tuesday, May 19th @ 11:59pm PDT w/ slip day)

Useful Resources:

Throughout this assignment, you may find the following resources helpful. Refer to them BEFORE posting questions on Piazza.

- Reference guide for Linux/Vim/SSH/scp
- Connecting to the lab machines remotely
- Running bash on Windows
- Unix reference sheet
- JUnit testing tutorial

Provided Files:

- None

Files to Submit:

- BinaryTree.java
- BinaryTreeTester.java

Goal

In this Programming Assignment, you will implement a Binary Tree and create JUnit tests to verify proper operation and implementation. You will also be answering questions related to runtime analysis on Gradescope.

As you get started, please pay attention to the following:

- Please read the ENTIRE write-up before getting started.
- For this homework and likely all future homework in CSE 12, you must have the same method signatures and instance variables as defined below.

Getting Started

Compile and Execute with JUnit

Running on UNIX based systems:

Compile: `javac -cp ../libs/junit-4.12.jar:../libs/hamcrest-core-1.3.jar:. *.java`

Execute: `java -cp ../libs/junit-4.12.jar:../libs/hamcrest-core-1.3.jar:. org.junit.runner.JUnitCore <Tester File>`

Running on Windows systems:

Compile: `javac -cp ..\libs\junit-4.12.jar;..\libs\hamcrest-core-1.3.jar; *.java`

Execute: `java -cp ..\libs\junit-4.12.jar;..\libs\hamcrest-core-1.3.jar; org.junit.runner.JUnitCore <Tester File>`

Part 1 - Understanding and Testing First

Understanding a Binary Tree and its Operations

In order to write a good tester, you will need a deep understanding of how the classes and methods you are testing are supposed to work. So before you start writing your tester, you should read part 2 in order to understand the specific behavior of the binary tree you are implementing.

Test BinaryTree with BinaryTreeTester.java

- The repository will **not** contain a starter tester file. If you do not recall how to set up JUnit tests or need a refresher, refer to the previous PAs and your previous testers.
- You should make sure to also include tests to check that your method throws the correct exceptions when they are expected to throw them. There are more sophisticated ways to do this (feel free to investigate and use them), but the simple approach is to do the following (source):

```
@Test
public void testExceptionMessage() {
    try {
        new ArrayList<Object>().get(0);
        fail("Expected an IndexOutOfBoundsException to be thrown");
    } catch (IndexOutOfBoundsException anIooBException) {
        assertThat(anIooBoundsException.getMessage(),
            is("Index: 0, Size: 0"));
    }
}
```

Note: If you cannot get the assertThat to work on the lab machines (or your machine) it's sufficient to assume that any IndexOutOfBoundsException is correct, and to simply pass in that case (i.e. fail if you do NOT enter the catch block, and leave the catch block empty).

- After final submission, we will be running more extensive tests on your code. The points awarded from our tests will make up your final grade.

Part 2 - BinaryTree

Once you understand what the behavior of a binary tree is supposed to be, your next task is to create your own implementation called BinaryTree.

- Create a file named BinaryTree.java. **BinaryTree is a generic class whose type parameter extends Comparable.**

- There are various ways to implement a binary tree. To ensure that you won't lose points, please strictly follow the guidelines that we have listed.
- For this PA, you **must** use getter/setters, instead of directly accessing instance variables. Failure to do so will result in failing tests and losing points.
- **Instance variables:** There are **two** instance variables. The `root` node and the `size`. Implement them based on the specifications below.
- **Constructors:** There are **three** constructors. Implement them based on the specifications below.
- **Methods:** There are **six** methods. They are all described below.

Imports:

In previous PAs, you were asked to implement your own Stack and Queue. Now that you have a strong understanding of the behavior of these data structures, we would like you to be familiar with Java's implementation of these data structures.

You are only allowed to use these import statements in BinaryTree.java. Do **NOT** import using the wildcard (i.e. `import java.util.*`). Import these individually.

- List - java.util.List
- LinkedList - java.util.LinkedList
- Queue - java.util.Queue

Hint: Queue is an interface so we are unable to instantiate it. As a result, we must use a class, such as LinkedList, that implements the Queue interface. This means we must do the following to create a queue: `Queue<E> queue = new LinkedList<E>();`

Inner Class:

Similar to PA3, you will need to create an inner class. Inside of your BinaryTree class, create a protected class called `Node`.

This inner `Node` class should have the following instance variables, all with the default access modifier:

- `Node left`: the left child of this node
- `Node right`: the right child of this node
- `E data`: the data stored in this node

The inner `Node` class should also have the following constructors and methods:

Constructor/Method Name	Description
`public Node(E data)`	Set the `data` instance variable to the argument that was passed in. `left` and `right` should be set to `null` to represent that we are creating a terminal node (a node without children).
`public void setLeft(Node left)`	Set the `left` instance variable to the argument that was passed in.

Constructor/Method Name	Description
`public void setRight(Node right)`	Set the `right` instance variable to the argument that was passed in.
`public void setData(E data)`	Set the `data` instance variable to the argument that was passed in.
`public Node getLeft()`	Return the left child.
`public Node getRight()`	Return the right child.
`public E getData()`	Return the data stored in this node.

Instance variables:

The BinaryTree class has two instance variables. **Note: Do not make these instance variables private, they should have the default access modifier. Do not add any other instance variables and do not add any static variables (other than private static final variables to be used as constants).**

- `Node root`: The root node of our binary tree. If the binary tree is empty, this instance variable should be `null`.
- `int size`: The number of nodes in our binary tree. If the binary tree is empty, this instace variable should be 0.

Constructors:

- `public BinaryTree()`: The no-args constructor should create an empty binary tree. Empty binary trees are represented with a `root` that is `null` and a corresponding `size` that is 0.
- `public BinaryTree(E data)`: Initialize the instance variables so that we now have a root node containing `data` and a size of 1. You can assume that `data` will never be `null`.
- `public BinaryTree(List<E> list)`: Add each element in the parameter `list` to the binary tree. The first element of `list` should be added first, the second should be added second, and so on. Each element in `list` should be inserted into the BinaryTree in level order (see visual below). You can assume that `list` will never be `null` and that elements in `list` will never be `null`. **Hint: See if there is a method in BinaryTree that will help you add each element to the binary tree.**

Public Methods:

⚠⚠⚠ **When implementing this assignment, you must use the appropriate getters/setters. Do not access instance variables directly.** ⚠⚠⚠ Some of our Gradescope tests rely on using getters/setters. Make sure that you are using these getters/setters instead of accessing instance variables directly to avoid losing points.

Method Name	Description	Exceptions to Throw

Method Name	Description	Exceptions to Throw
`public void add(E element)`	Add a new `Node` containing `element` to the binary tree in level order and update`size` accordingly. Hint: Use a Queue for level order traversal.	throw `NullPointerException` when `element` is `null`
`public boolean remove(E element)`	Remove the specified element in the binary tree and replace it with the node in the rightmost position in the lowest level. Return `true` if the removal was successful, meaning that the specified element was found and removed. Otherwise, return `false` to indicate an unsuccessful removal. Update `size` if necessary. Note: Be sure that you are searching the binary tree for `element` using level order traversal. If there are multiple occurrences of `element` in the binary tree, you should remove and replace the first occurrence of `element` that you encounter when performing level order traversal. See the visualizations below for more information.	throw `NullPointerException` when `element` is `null`
`public boolean containsBFS(E element)`	Check if `element` is in the binary tree. Return `true` if `element` is in the binary tree and false otherwise. Note: You must implement this method iteratively and use BFS. When deciding which node to insert into your queue, insert the left child first, then the right	throw `NullPointerException` when `element` is `null`
`public int getHeight()`	Return the height of the binary tree. For our implementation of a binary tree, an empty binary tree has a height of 0 and a single root node has a height of 0. Hint: `Math.log()` might be useful in this method.	None
`public int getSize()`	Return the number of nodes that are in the binary tree.	None
`public E minValue()`	Return the minimum value stored in the binary tree. If the binary tree is empty, return `null`. To compare the data stored in nodes, be sure to use `compareTo()` instead of comparison operators, such as `<`, `>`, `<=`, and `>=`.	None

Clarifications

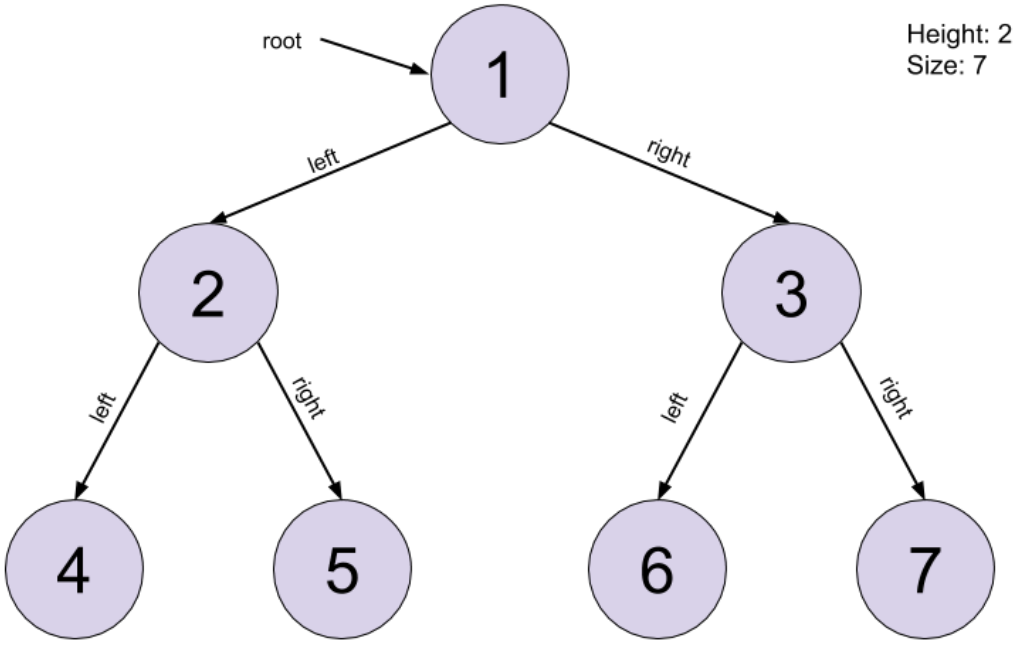
- `null` is not a valid element in the binary tree. It should not ever be in the binary tree. Therefore, be sure that you are throwing `NullPointerException` as specified in the method descriptions above.
- Duplicate elements are allowed in the binary tree.

- None of your method implementation should use recursion. Implement each method iteratively.
- In case you missed it earlier: ⚠⚠⚠ **When implementing this assignment, you must use the appropriate getters/setters. Do not access instance variables directly.** ⚠⚠⚠

Visualization of BinaryTree

Level Order Traversal (BFS)

Remember, when deciding which node to insert into your queue, insert the left child first, then the right.

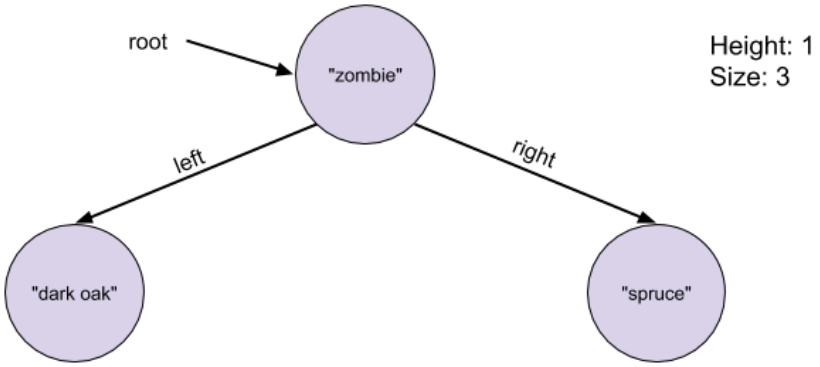


The number in each node indicates that it is the #th node to be visited.

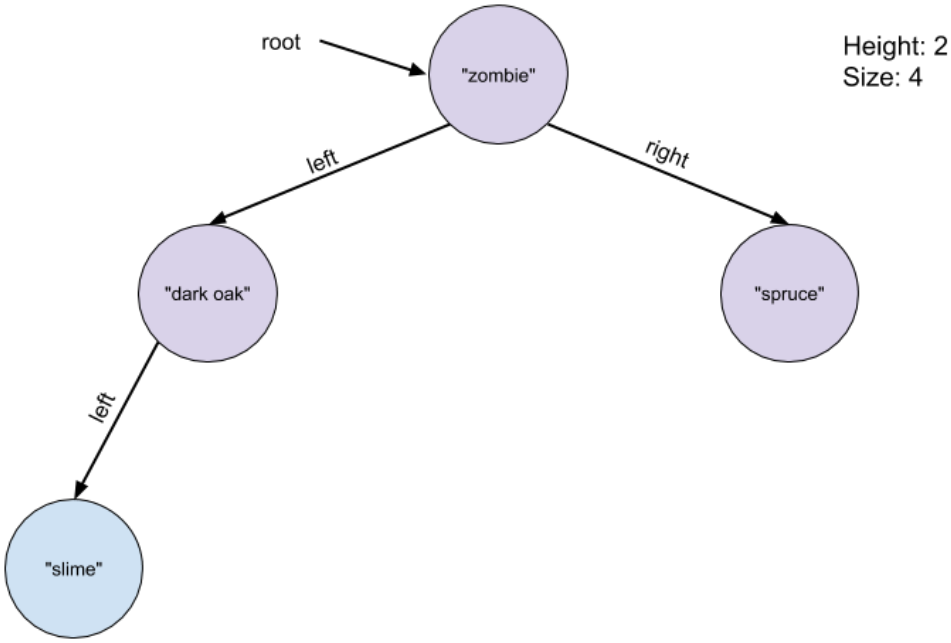
The following are visualizations of the behavior of each method.

Calling add() on a full binary tree

Imagine this is a BinaryTree object called tree.

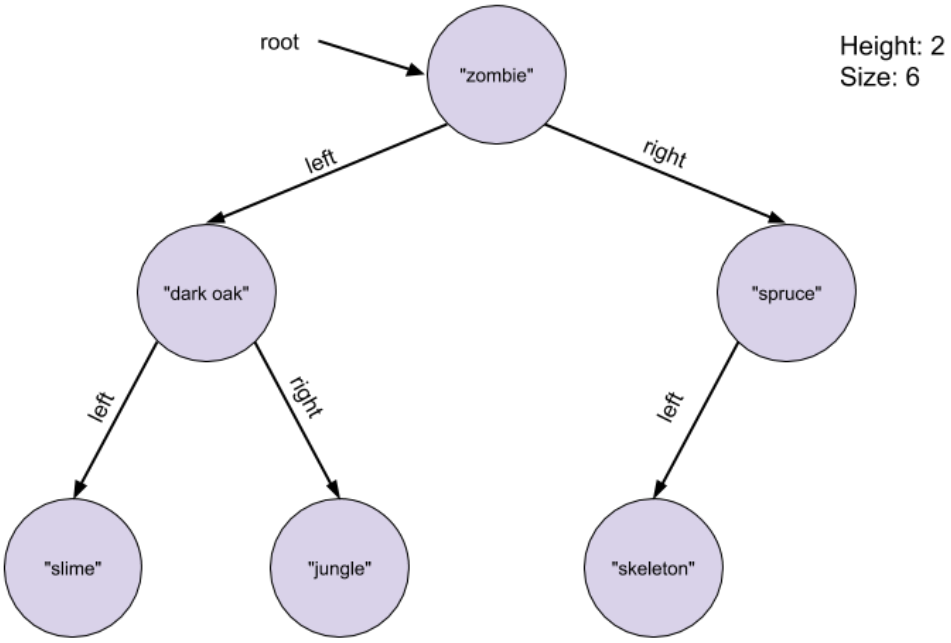


After calling tree.add("slime"), tree should look like this:

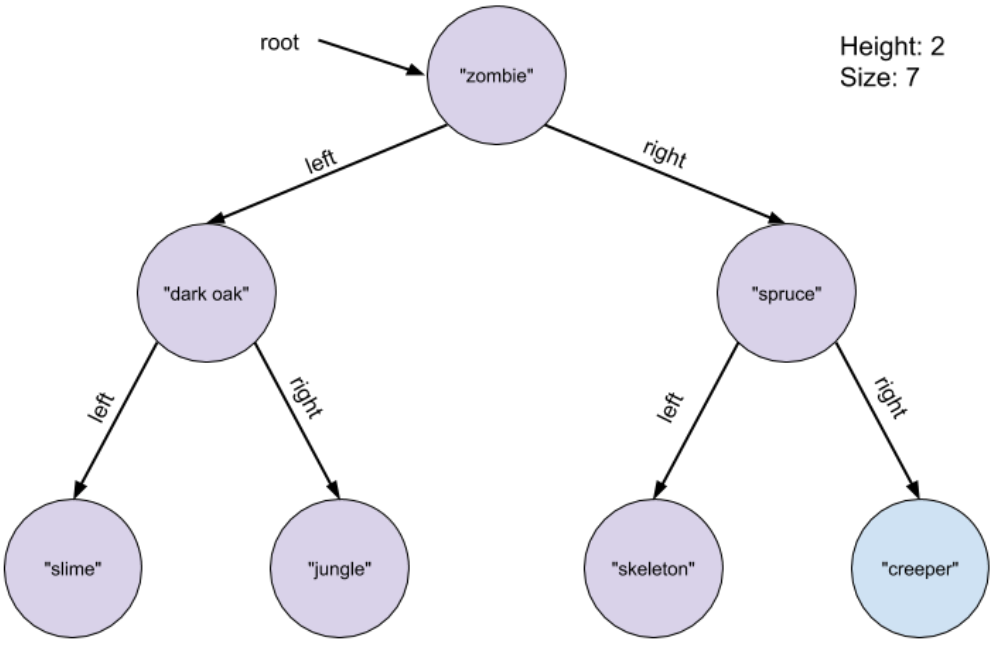


Calling add() on a complete binary tree

Imagine this is a BinaryTree object called tree.



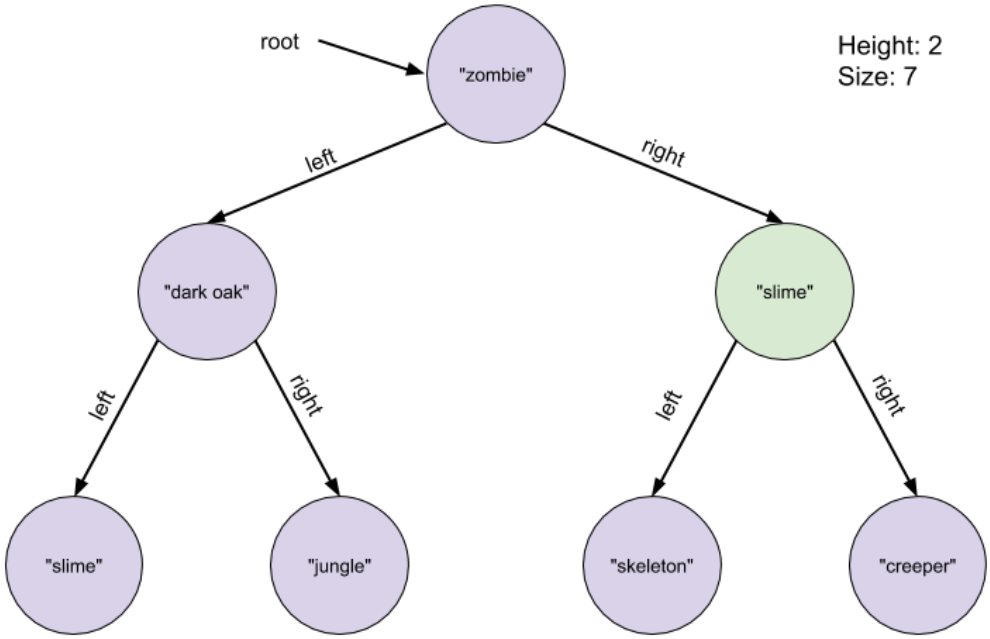
After calling tree.add("creeper"), tree should look like this:



Calling containsBFS()

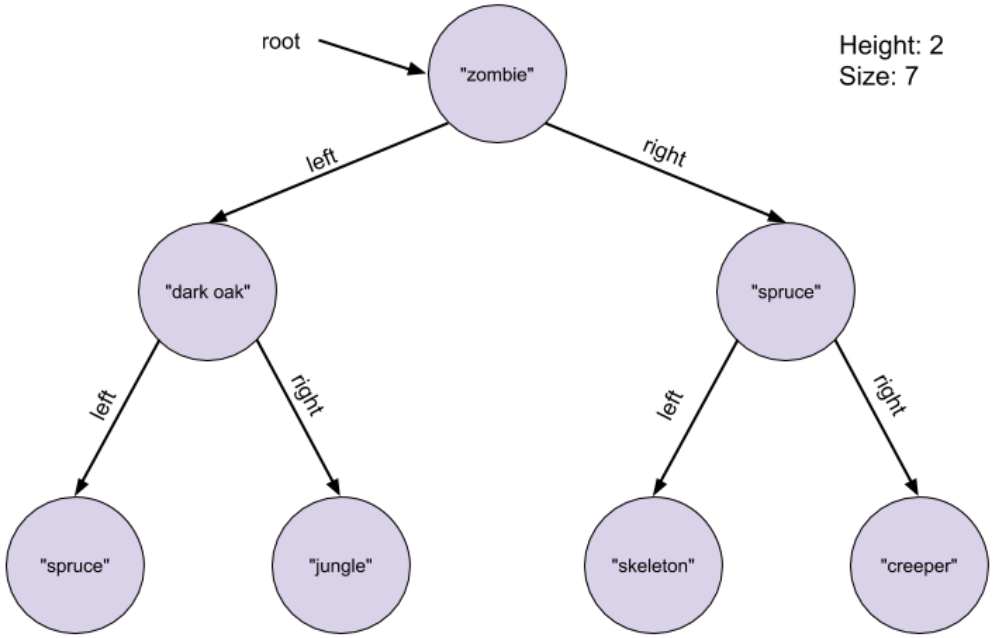
Note: The important thing here is the node that determines that the call is true.

Imagine this is a BinaryTree object called tree.
tree.containsBFS("slime") would return true because of the green node

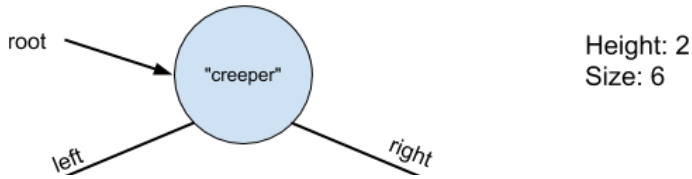


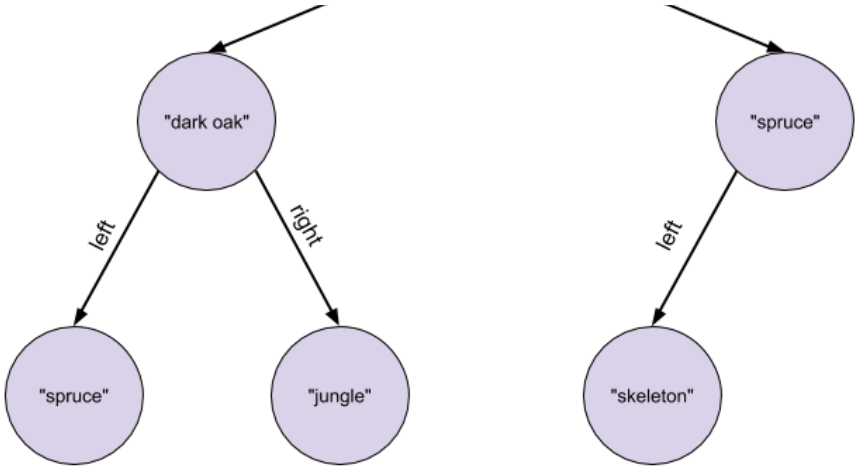
Calling remove()

Imagine this is a BinaryTree object called tree.

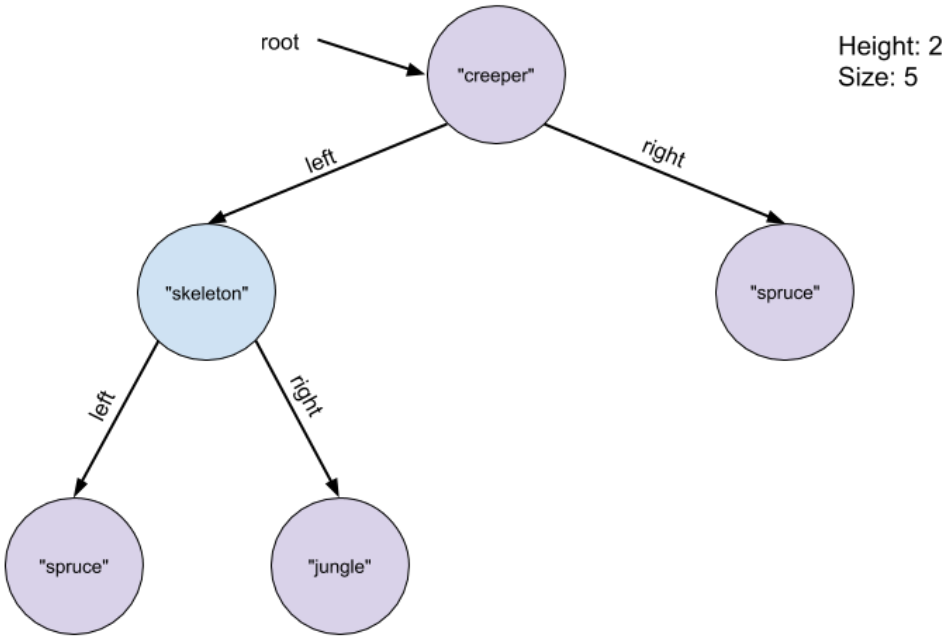


After calling tree.remove("zombie"), tree should look like this:

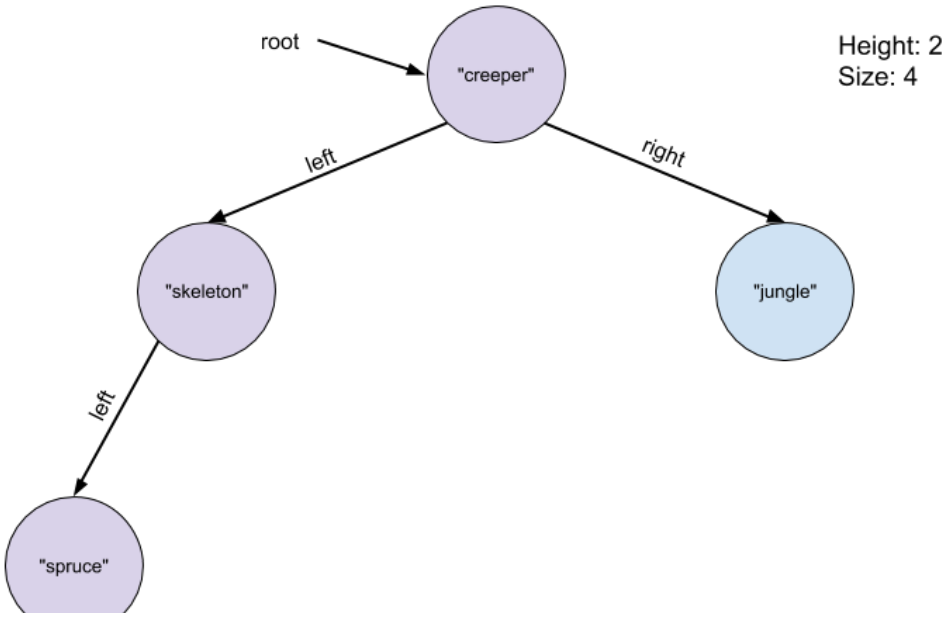




After calling tree.remove("dark oak"), tree should look like this:



After calling tree.remove("spruce"), tree should look like this:



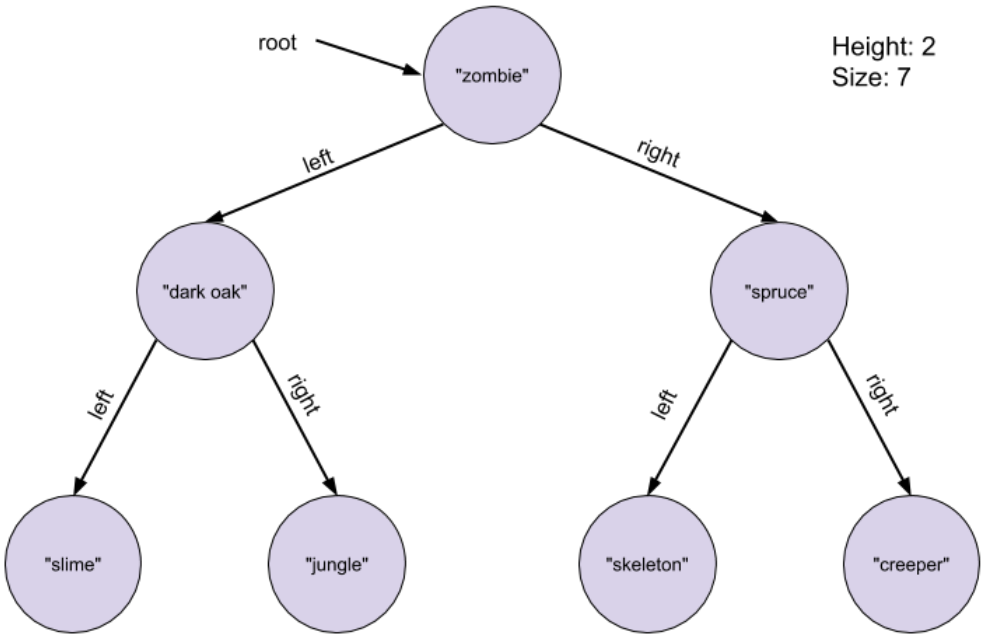
Remember, when searching for the node to remove, you must remove the first occurrence from performing a level order traversal. After you've found the node that you need to remove, there are two ways to replace the node in question: either change the data stored in the node to be the data of the last node in the tree or to update the child references of the parent and the last node. We define the last node to be the right-most node in the lowest level.

Hint: In both cases, after you have determined the node to remove and the last node, you will most likely have to re-traverse the tree to find the parent of the node that you're removing (except in the case that it's a root node). To find this node, you may find yourself comparing the node to remove with the left and right children of a parent node. In this case, == may be helpful (recall what the behavior of == is with respect to Objects).

Another hint: To make sure that your remove() function is working properly, double check **all** child references of all affected nodes to ensure that they have the correct left and right references in your tests.

Calling minValue()

Imagine this is a BinaryTree object called tree.



Calling tree.minValue() would return "creeper". Strings are ordered lexicographically (alphabet ordering).

Part 3 - Runtime Analysis

For part 3 of this assignment, you will be answering a series of questions related to runtime analysis.

The questions can be found on Gradescope under the assignment: **Programming Assignment 7 (Part 2)**

You have an unlimited number of attempts and no time limit; however, make sure to submit before the PA deadline.

Note: Follow the format guidelines carefully. You may lose points if your answer is incorrectly formatted.

For your convenience, you may also view the questions here.

This worksheet is also part of your PA so feel free to ask tutors/TAs for help! They can help you with any conceptual questions, just make sure to know what you want to ask to get the best possible help.

Testing

For this PA, we will not be providing descriptions of test cases that we will be testing your code against. Learning how to create your own test cases is an extremely useful skill. If you need inspiration for your test cases, refer back to previous assignments.

Although your tester will not be graded for style or correctness, creating your own test cases and testing your code is the only way to be sure that your code works.

Survey

You can find the weekly reflection survey here. Please fill out the survey, as it will count towards 1 point of your PA7 score.

Submission

Turning in Your Code

Submit the following files to **Gradescope**. Please make sure that you submit code to **Programming Assignment 7 (Part 1)**.

- BinaryTree.java
- BinaryTreeTester.java

Submit your runtime analysis answers to **Programming Assignment 7 (Part 2)** on **Gradescope**.

Important: Even if your code does not pass all the tests, you will still be able to submit your homework to receive partial points for the tests that you passed. ⚠ Make sure your code compiles in order to receive partial credit. ⚠

How Your Assignment Will Be Evaluated

- **Runtime Analysis** (44 points)
- **Correctness** (50 points)
    - Does your code compile? If not, you will get 0 points.
    - You are responsible for making sure that your program runs correctly under all possible situations.
- **Weekly Reflection Survey** (1 point)
- **Coding Style** (5 points) We will grade your code style in `BinaryTree.java` thoroughly. `BinaryTreeTester.java` will not be graded for style. Namely, there are a few things you must have in each file / class / method:

1. File header
2. Class header

3. Method header(s)
4. Inline comments
5. Proper indentation (do not intermingle spaces and tabs for indentation)
6. Descriptive variable names
7. No magic numbers
8. Reasonably short methods (if you have implemented each method according to specification in this write-up, you're fine). This is not enforced as strictly.
9. Lines shorter than 80 characters (Note: tabs will be counted as 4 characters toward this limit. It is a good idea to set your tab width/size to be 4)
10. Javadoc conventions (@param, @return tags, /** comments */, etc.)

A full style guide can be found here. If you need any clarifications, feel free to ask on Piazza.