CS2230 Computer Science II:Data Structures
Homework 5
Queries on binary trees

In this assignment, you will build part of a database for hierarchical data. Specifically, you will write code that can be used to query (i.e., ask questions about) data that is stored as a binary tree. Although you'll use family tree data as a case study, you will write your database using generic types so that it can be used on any data type. You'll also use the BiFunction interface so that the database can accept custom queries.

Learning objectives for this assignment

- Write code that uses a binary tree built from linked Nodes
- Write both recursive and iterative algorithms for trees
- Use higher order functions and generic types with a new data structure
- Produce evidence that your code is correct by writing your own JUnit tests

Submission Checklist

All files provided in HW6 must be present in your GitHub repository. The following must also include your changes:

- TreeFunctions
  - Method sumAtDepthIterative, sumAtDepthRecursive
- AbstractBinaryTree.java
  - specifically, the methods reduceAtDepthIterative, reduceAtDepthRecursive, and suchThatIterative
- LinkedBinaryTree.java
  - specifically, the method insertNode
- TreeTest.java
  - Additional tests for all the reduce and suchThat methods to have enough evidence that they work for different types of trees and different BiFunctions
- FamilyRecordQuery2.java
  - complete the query
- FamilyRecordQuery3.java
  - complete the query
- FamilyRecordQuery4.java
  - complete the query

- ✓ Do the tests pass?

- ✓ Did you write additional tests as directed?
- ✓ Does the code at https://research-git.uiowa.edu/cs2230-fa19/hw5-hawkid reflect the version of the files you intended to turn in?
    - the minimum here is to double check your files in the web browser...
    - but for the most certainty we recommend that you clone your repo as a fresh copy, import it into IntelliJ as a new project, and re-run all the tests

Source repository

https://research-git.uiowa.edu/cs2230-assignments/binarytreequeriesgtg-fa19.git

Your submission repository

https://research-git.uiowa.edu/cs2230-fa19/hw5-hawkid

Part 0

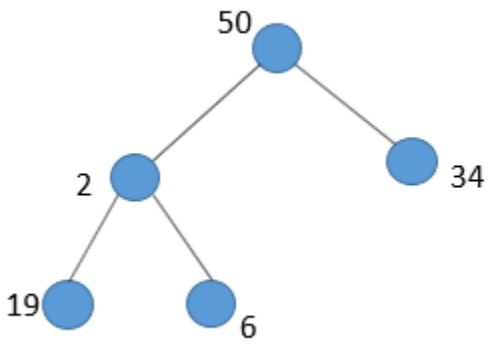
Open *familyrecord.csv* and briefly look at the data. Each row (i.e., record) is intended to be one Node in a binary tree representing a family lineage. Let's assume we fill the tree top-to-bottom and left-to-right. An example is shown in Part 1. On paper, draw the binary tree that results from inserting these names into a binary tree in this way. (No submission of Part 0 necessary)

Part 1: Insert nodes into a tree

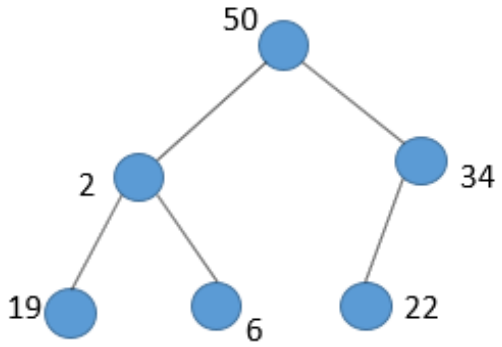
The LinkedBinaryTree has methods for inserting a node into a specific position (root(), addLeft(), addRight()), but what if the user doesn't want to handle keeping track of all the Positions themselves when they are building up a tree?

The method `LinkedBinaryTree.insertNode()` inserts a node in the leftmost available ***free spot***. That is, if the method is called in the following sequence:

`insertNode(50), insertNode(2), insertNode(34), insertNode(19), insertNode(6)`, then the tree will look like:



And, if we call `insertNode(22)`, the tree will look like:



Notice that the ***breadth-first traversal order*** of the resulting tree is the same order in which nodes were inserted.

Implement this method, `insertNode`. Notice that LinkedBinaryTree class has a helpful instance variable: a Queue called `nodesThatHaveOneOrZeroChildren`. This Queue should contain the Nodes who are missing 1 or 2 children. Your insert method will use `nodesThatHaveOneOrZeroChildren` to know which Node to add a child to. For example, in the first tree above, `nodesThatHaveOneOrZeroChildren` = [Node(34),Node(19),Node(6)] and in the second tree, `nodesThatHaveOneOrZeroChildren` = [Node(34),Node(19),Node(6),Node(22)].

When finished, `testInsertionAndtoArray` test should pass.

You may assume that for a given LinkedBinaryTree, either the user will build the tree using only insertNode, or they will build the tree using only addLeft, addRight, addRoot; they won't mix the two approaches. That said, your implementation of insertNode of course relies on calling addLeft, addRight, and addRoot.

Sanity Check:

- Do the following tests in TreeTest pass?
  - `testInsertionAndtoArray`
  - `testInsertionAndtoArray2`
- Run FamilyRecordQuery.java. Does it print out the breadth-first traversal of the Tree you drew in Part 0?

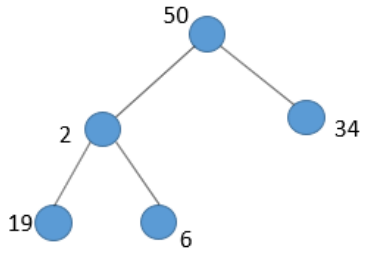
STOP: You should make at least 1 commit with the message "Part 1". Push it to your repository and double check on the web browser that it is in https://research-git.uiowa.edu/cs2230-fa19/hw5-hawkid

Part 2: sumAtDepthIterative & sumAtDepthRecursive

In TreeFunctions, finish these two methods, which sum up all the nodes at a given depth.

Example

If the tree looks like



Then `sumAtDepth(depth=0)` is 50, `sumAtDepth(depth=1)` is 36 and `sumAtDepth(depth=2)` is 25. If the depth is greater than the maximum depth of the tree, just return 0. So in our example, `sumAtDepth(depth=3)` is 0.

You'll implement two versions:

- `sumAtDepthIterative()` must be iterative
  - use a loop
  - use a data structure to keep track of unvisited nodes
- `sumAtDepthRecurisve()` must use recursion
  - **We recommend** that you define a `private` helper method where all the actual work happens. The reason is that the given function parameters may not be sufficient to keep track of where you are in the tree. See Lab 12 for examples.

Your method ought to have a **worst-case running time in O(N),** where N is the number of elements in the tree. That means you'll have to calculate depth as you go instead of relying on calling the depth() method over and over, which already has a worst case running time in O(N) ***for a single call***.

Testing

You should see if the following tests pass in TreeTest.java:

- `sumAtDepthTest1, sumAtDepthTest2`
- `sumAtDepthTreeWithHolesTest1, sumAtDepthTreeWithHolesTest2`

STOP: You should make at least 1 commit with the message "Part 2". Push it to your repository and double check on the web browser that it is in https://research-git.uiowa.edu/cs2230-fa19/hw5-hawkid

Part 3: ReduceAtDepthIterative & ReduceAtDepthRecursive

Let's generalize the idea of sumAtDepth so that it works for Trees of any data type (not just Integer) and for any notion of "sum". The name we'll use for this generalization of "sum" is "reduce". Reduce means combining all the elements of a data structure to come to a single value.

To generalize, we'll use an interface called BiFunction. This interface declares a single method called apply. A class that implements BiFunction needs to define apply. Apply can do anything you want as long as it takes 2 arguments and returns a value. See IntegerSum and DoubleMax in TreeTest for two examples.

The same requirements apply to reduceAtDepthIterative/Recursive as for the sumAtDepthIterative/Recursive. Find stubs for the two methods in AbstractTree.java. Notice that the reduce methods take two extra parameters compared to the sum methods. One is the BiFunction and the other is the initialValue. The initialValue is our generalization of zero, that is the value for which for any x, apply(initialValue, x) = x. Depending on what your BiFunction is, the concept of zero will be different. What is "zero" for max?

Testing

You should see if the following tests pass in TreeTest.java:

- `reduceAtDepthTest1, reduceAtDepthTest2`
- `reduceAtDepthTreeWithHolesTest1, reduceAtDepthTreeWithHolesTest2`

You must write additional test cases to ensure your code is correct. Your methods will be graded on several additional hidden tests. Some examples of things to test, an empty tree, a tree with only one node, different tree shapes etc.

- **TESTING HINT 1:** The provided tests include only one BiFunction implementation, IntegerSum. The hidden tests will use other BiFunctions, so test your code with DoubleMax and other BiFunctions that you define yourself. Make sure you pass in an appropriate initialValue! (see discussion above).
- **TESTING HINT 2:** The provided tests include only LinkedBinaryTree<Integer>. The hidden tests will use other types for LinkedBinaryTree<E>, so test your code with other types, such as LinkedBinaryTree<String> and LinkedBinaryTree<Double>.
- **TESTING HINT 3:** The provided tests only test the method on two tree shapes and one depth. The hidden tests will test other trees and other depths.

To write a new test case, we recommend that you copy/paste an existing one and change the name of the test method and tweak its code. Draw your test tree on paper so you know what your assertions' expected values are.

STOP: You should make at least 1 commit with the message "Part 3". Push it to your repository and double check on the web browser that it is in https://research-git.uiowa.edu/cs2230-fa19/hw5-hawkid

Part 4: Queries that use ReduceAtDepth*

The word ***query*** in computer science typically means "a question about the data in a database". In this homework, our example database is the family tree data that comes from familyrecord.csv.

Our first query is *"what people were in generation X of the family tree?".* Take a look at the tree that is printed out in breadth first by FamilyRecordQuery2. Notice that each level of the tree is a generation of the family. At depth 0 is the child Roger, depth 1 is the parents of Roger, and depth 2 the grandparents and so forth.

Write a method that returns a concatenated string of all the names in a generation **separated by one space**.

For example, generation 1 would be "Ryan Jisoon". There is starter code in the file. The code queries the treeOfNames by using your `reduceAtDepthRecursive` to concatenate names at that depth. For this query, please fix the `ConcatenateNames` class in FamilyRecordQuery2.java.

To check your work, see if the names printed are correct for the given generation (depth).

STOP: You should make at least 1 commit with the message "Part 4". Push it to your repository and double check on the web browser that it is in https://research-git.uiowa.edu/cs2230-fa19/hw5-hawkid

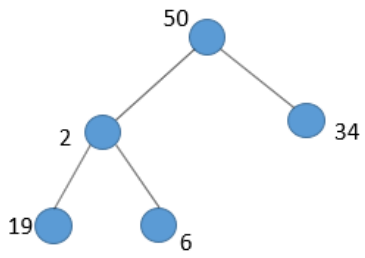
Part 5: SuchThat

A "SuchThat query" finds the elements that are desired and returns them in depth-first **pre-order**. Write a method to find the nodes with ***desired*** values and return them in a list. To determine what data is ***desired***, we'll use the BiFunction interface again (but in a different way!)

The SuchThat methods in take a BiFunction<BinaryTree<E>, Position<E>, Boolean>. This BiFunction's apply method takes two arguments – a tree and a position in that tree – and returns true or false[1].

Example

If the tree contains...



[1] In computer science, we often call such a function that returns true or false a ***predicate***.

and our predicate asks "is the element even?", then the answer will be the list [50,2,6,34]. That is, the pre-order traversal is [50,2,19,6,34] but we remove the integers where "is element even?" is false to get [50,2,6,34].

The “is the element even?” predicate is defined as IsEven in TreeTest. IsEven is a simple example. The fact that SuchThat takes BiFunctions whose two arguments include the tree and the position makes SuchThat very powerful: not only can it express true/false functions over the contents of the tree’s nodes, but also over the structure of the tree! You can find two such examples in TreeTest: HasTwoChildren and InOrder.

AbstractBinaryTree contains suchThatIterative and suchThatRecursive methods. The recursive one is already implemented for you so that you can see an example of how to use the BiFunction to determine if an element should be included in the results.

Testing

TreeTest.java contains two test cases; see if they pass.

- `suchThatIterativeTest`
- `suchThatIterativeTreeWithHolesTest`

You must write additional test cases for *both* `suchThatIterative` and `suchThatRecursive`. Notice that `suchThatRecursive` is already completed. You are to write test cases for both `suchThatRecursive` and `suchThatIterative` to make sure the code really works. Your methods will be graded on several additional hidden tests and on the quality of your tests for both methods.

HINT 3: The provided tests include only one of the predicate implementations, HasTwoChildren. The hidden tests will use other predicates, so test your code with InOrder, IsEven, and any other predicates you come up with.

To write a new test case, we recommend that you copy/paste an existing one and change the name of the test method. Then draw your test tree on paper so you know what your assertions’ expected values are.

STOP: You should make at least 1 commit with the message “Part 5”. Push it to your repository and double check it is in https://github.uiowa.edu/cs2230-sp19/hw6-hawkid

Part 6: Queries that use suchThat

In FamilyRecordQuery3.java uncomment the code in the main method and the classes. You might have noticed the family has a lot of people named Roger and a lot of Engineers. You will write 3 queries

- return all Rogers in the tree
- return all Engineers in the tree
- return all children of Rogers in the tree (note again that there are multiple Rogers)

You will use your `suchThatIterative` and/or `suchThatRecursive` to find these people in the family tree.

- Finish the class `NameIs` so that its apply returns true if the name field of the FamilyRecord matches the **exact** String that was given to NameIs's constructor.
- Finish the class `JobIs` so that its apply returns true if the job field of the Family Record **contains** the String that was given to JobIs's constructor.
- Finish the class `IsChildOf` so that its apply returns true if the Position is a ***parent*** of a Position whose Family Record has a name field that exactly matches the String that was given to IsChildOf's constructor. **IMPORTANT TIP**: What?? Why does IsChildOf need the Position to be a *parent* of a Roger position rather than the *child*? Notice that a family tree stored as a BinaryTree, reverses the parent-child relationship, since a bio-child has exactly two bio-parents, who each have two bio-parents, etc. For example, in this 3-node family tree

  Joe (Jr)

  / \

Joe (Sr) Louise

IsChildOf("Joe")'s apply method would return true for Joe (Jr) and false for the other two Positions.

The main method includes three queries, one that uses NameIs, one that uses JobIs, and one that uses IsChildOf. The first should find all records whose name is "Roger". The second query should find all records whose job title *includes* "Engineer", including Software Engineers[2], Mechanical Engineers, and others. The third query should find all records who are a child of a Roger.

Feel free to test different names or jobs with these queries. To check your work, look over the family tree.

STOP: You should make at least 1 commit with the message "Part 6". Push it to your repository and double check it is in https://github.uiowa.edu/cs2230-sp19/hw6-hawkid

[2] Fun fact: People born in 1920 couldn't have been "Software Engineers" because that term was coined later by Margaret Hamilton while she was leading the development of the flight software for NASA's Apollo Project.

Part 7: One more query

Uncomment the code in the main method of FamilyRecordQuery4.java. Write the code that will return and print out all the people in the family tree that today are under the age of 50. Follow the same structure of the previous queries. Once again, check your answer by looking over the family tree.

STOP: You should make at least 1 commit with the message "Part 7". Push it to your repository and double check it is in https://github.uiowa.edu/cs2230-sp19/hw6-hawkid