CS2230 Computer Science II:Data Structures
Homework 6
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 higher order functions so that the database can accept custom queries.

Learning objectives for this assignment

- Write code that uses a binary tree built from linked TreeNodes
- 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

Submit the following files to ICON with required changes:

- BinaryTree.java
- BinaryTreeTest.java
- FamilyRecordQuery.java

- ✓ Do the tests pass?
- ✓ Did you write additional tests as directed?
- ✓ Does the ICON submission reflect the version of the files I intended to turn in? (You must download the files and check).

Getting HW6

Follow the instructions in getting_hw6.pdf

Part 0

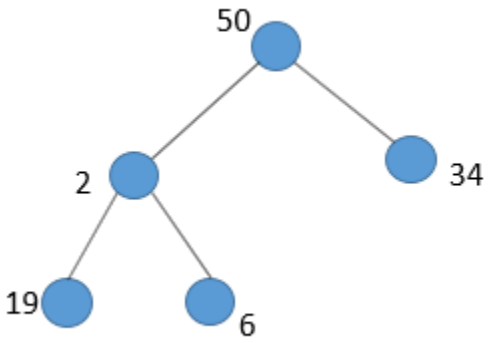
Open *familyRecord.csv* and briefly look at the data. Draw out a binary tree using this data. Start with the top record as the root of the tree. Inserting a node into the binary tree should go into the next available spot in the tree starting with the left node, an example is shown in Part 1. You will use this drawn out binary tree as reference and to check your work, in the following parts of the homework.

Part 1: Insert nodes into a tree

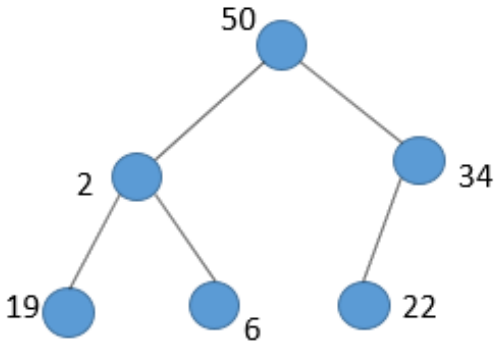
We provided a file BinaryTree.java that defines a class for a binary tree and some methods.

The method `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 order in which nodes were inserted.

Implement this method using the fields in the constructor. Notice that there is a LinkedList called `nodesThatNeedChildren`. This List should contain the TreeNodes who are missing 1 or 2 children. Your insert method will use `nodesThatNeedChildren` to know which TreeNode to add a child to. For example, in the first tree above, `nodesThatNeedChildren` = [TreeNode(34),TreeNode(19),TreeNode(6)] and in the second tree, `nodesThatNeedChildren` = [TreeNode(34),TreeNode(19),TreeNode(6),TreeNode(22)].

When finished, `testInsertionAndtoArray` test should pass.

Sanity Check:

Does `testInsertionAndtoArray` pass?

In the test file, call bt.displayTree() to print out the tree as text. Does this tree look like the example above?

Part 2: reduceAtDepth

Write a method that returns the reduced value of the nodes at a given depth. Here, "reduced" means the same as what `ReduceFunction` defines in HW5, that is, start with `ReduceFunction.initialValue()` and combine each element to the total value using `ReduceFunction.combine()`. The depth is given as a parameter. If the depth of the tree is actually less than the given height, then return `ReduceFunction.initialValue()`.

Your method must have a running time in O(N), where N is the number of nodes in the tree.

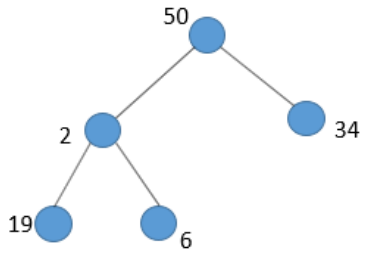
In BinaryTree.java, complete `reduceAtDepth()` and `reduceAtDepthRecurisve()`.

`reduceAtDepthRecurisve()` must use recursion and `reduceAtDepth()` must be iterative (use a loop).

- Notice that the generic data types of `reduceAtDepthRecurisve()` makes the input and output type to be the same type, T. This is just to make the recursion easier.
- Notice that there is a signature for a private helper method `reduceAtDepthHelper.` It has more arguments to facilitate the recursive calls.

Example

If the tree looks like



And the given `ReduceFunction` is plus (+), then `reduceAtDepth(0)` is 50, `reduceAtDepth(1)` is 36 and `reduceAtDepth(2)` is 25.

Testing

There is one test, `sumOfDepthTest`, in BinaryTreeTest.java. **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.

HINT 1: The provided tests include only one ReduceFunction implementation, IntegerSum. The hidden tests will use other ReduceFunctions, so test your code with other ReduceFunctions that you define.

HINT 2: The provided tests include only BinaryTree<Integer>. The hidden tests will use other types for BinaryTree<T>, so test your code with other types, such as BinaryTree<String>.

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 order to insert nodes using insertNode and what your assertions' expected values are.

Part 3: Queries that use reduceAtDepth

In the queries folder of the project, there is a file named FamilyRecordQuery.java. We created few methods that parses the CSV file of family data and creates a binary tree of the data. Open the CSV file in Excel or in a Text Editor to get a feel for the data. Notice that each level of the tree is a generation of the family. At depth 0 is the child Robert, depth 1 is the parents of Robert, 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 2: Ryan Jisoon”. There is starter code labeled as PART 3. The code queries the treeOfNames by using your *ReduceAtDepthRecursive* to concatenate names at that depth. For this query, please fix the *ConcatenateNames* class in FamilyRecordQuery.java.

We've provided the same query below it for the treeOfRecords. It uses your *reduceAtDepth* to concatenate **names of records** at that depth. All you need to do is fix the class *ConcatenateNamesFromRecord* within FamilyRecordQuery.java.

To check your work, use treeOfNames.displayTree() in the main method of FamilyRecordQuery.java and see if the names you printed are correct for the given generation.

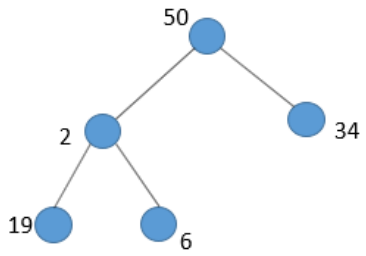
Part 4: Select

A "select 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 Predicate interface that is used by filter in HW5. Write your method iteratively.

Note that your method must run in linear time: O(n). Implement the method `selectIterative.`

Example

If the tree contains...



and our predicate is "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].

Testing

BinaryTreeTest.java contains a single test case `selectIterativeTest`. You must write additional test cases for *both* `selectIterative. Notice that selectRecursive` is already completed. You are to write test cases for both `selectRecursive` and selectIterative 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 Predicate implementation, IsEven. The hidden tests will use other Predicates, so test your code with other Predicates that you define.

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 order to insert nodes using insertNode and what your assertions' expected values are.

Part 5: Queries that use Select

In FamilyRecordQuery.java uncomment the code required for Part 5 in the main method and the classes. As you noticed, the family has a lot of people named Robert and a lot of engineers. Use your *`selectIterative`* and *`selectRecursive`* to find these people in the family tree.

To find all the Roberts, fix the `SelectName` class and search for the **exact** string “Robert” in the name field of the FamilyRecord. This must use *`selectIterative.`*

To find all the Engineers, fix the `SelectJob` class and search for any text **containing** “Engineer” in the job field of the FamilyRecord. This must use *`selectRecursive`*. This query will output a list of many engineer types (people born in 1920 couldn’t have been Software Engineers).

Feel free to test our different names or jobs with these queries. To check your work, look at the displayed tree or the CVS file to see that the output is correct.

Part 6: One more query

Uncomment the code in the main method of FamilyRecordQuery.java. Write the code that will return and print out all the people in the family tree that are under the age of 50. Follow the same structure of

the previous queries. You must use *selectIterative* or *selectRecursive* to complete this task. Once again, check the CVS file or the displayed tree to see if your output is correct.