PA 5: Augmented Red-Black Tree

Due Date: Friday 26 April 2024 by 11:59 PM

General Guidelines.
The instructions given below describe the required methods in each class. You may need to write additional methods or classes that will not be directly tested but may be necessary to complete the assignment.

In general, you are not allowed to import or use any additional classes in your code without explicit permission from your instructor!

Note: It is okay to use the Math class if you want.

Note on grading and provided tests: The provided tests are to help you as you implement the project and (in the case of auto-graded sections) to give you an idea of the number of points you are likely to receive. Please note that the points indicated when you run these tests locally are not your final grade. Your solution will be graded by the TAs after you submit. They may use different test cases, and they will also look at your code. Please also note that these test cases are not likely to be exhaustive and find every possible error. Part of programming is learning to test and debug your own code, so if something goes wrong, we can help guide you in the debugging process but we will not debug your code for you.

Project Overview.
In this project, you are giving a working implementation of a right-leaning red-black tree. This is just like a left-leaning red-black tree except that all the red links lean to the right instead of to the left. If you haven't already done this (due to the homework), you should take some time to work out what the insert cases would look like for this tree. You should also take some time to look through the code to make sure you understand how the transformations work and how the put and get operations work. This red-black tree is already augmented so that each node keeps track of its own height. This enables us to retrieve the height of a node in O(logN) time simply by finding the node instead of calculating the height. Your task in this project is to augment it further so that each node also knows the size of the subtree for which it is the root (size meaning the number of nodes in that subtree). This needs to be done within the insert function so that all those values are updated within the insert function as the insert function executes (not as a separate traversal). After you get that done, the second task is to implement some additional methods for the tree that use the augmented information in the nodes. This

requires some thinking about how the information in the node can help you get the necessary information efficiently. I recommend drawing some small trees and figuring it out by hand before trying to code. You must use the Node class as is. It counts the number of times you access nodes, so that the test code can give you an indication of your efficiency by counting that.

Note: It is considered academic dishonesty to try to trick the auto-grader or the human graders into thinking your solution is more efficient than it is. If you're ever unsure of what that means, ask!

Specific guidelines for augmenting the tree.

- All the nodes in an insert path need to be updated as the insert is executed. You should not be doing a separate traversal in order to update the *N* value.
- See how it is done with the *height* value and that should help you with this one.
- The tests will run *put* to make sure your augmentation is meeting the runtime requirements.

Below are the additional methods you need to implement with information about runtime requirements. There are method signatures already provided in the code. If you don't finish some of these methods, please do not remove them as that would break the test code.

`RBT.java API`

Method Signature	**Description**	**Comments**	**Points**
`int depth(int key)`	`return the depth of the node with the given key or -1 if the key does not exist`	`This should not take more time than it takes to find the node, which should be O(logN)`	4
`int size(int key)`	`return the number of nodes in the subtree rooted by the node with the given key or -1 if the key does not exist`	`This should not take more time than it takes to find the node, which should be O(logN).`	4

int min() throws EmptyTreeException	return the minimum key or throw the exception if the tree is empty	This should be O(logN).	4
int max() throws EmptyTreeException	return the maximum key or throw the exception if the tree is empty	This should be O(logN)	4
int floor(int key) throws KeyDoesNotExistException	return the largest key that is less than or equal to the parameter or throw the exception if such a key does not exist	This should be O(logN).	5
int ceil(int key) throws KeyDoesNotExistException	return the smallest key that is greater than or equal to the parameter or throw the exception if such a key does not exist	This should be O(logN).	5
int rank(int key)	return the number of keys that are less than the parameter or -1 if the key does not exist	This should be O(logN).	5
int select(int rank) throws NoSuchRankException	return the key at the given rank or throw the exception if the rank passed in does not make	This should be O(logN).	5

	sense for the tree		
int size(int lo, int hi)	return the number of keys in the range [lo…hi]	This should be O(logN).	5
String toString()	return a String representation of the tree that lists the nodes level by level	You can use the provided Deque class for this. Use the provided toString method in the Node class.	4

In addition to the tests associated with the functions above, 5 points will be allotted for testing the insert method to make sure it still works after you do the augmentation and that it still works efficiently.

Other Notes and Guidelines
Do:

- Draw some trees by hand and use them to test your functions as well as to think through the solutions to the functions.
- Implement most of these functions recursively. (This is a recommendation more than a requirement.)
- Update the *N* value of each node on the path when you insert a new node into the tree. The best way to do this is to update these values at the end of the *put* helper function so that it always gets updated for each node going back up the tree. You will also need to update some of the transformation functions.
- Write helper functions. The method signatures cannot be changed and a good recursive function will probably require more parameters.
- Make sure your submission compiles and runs with the provided test cases. Even if you don't implement a method, it still needs to exist in your file so that the test cases will not crash.
- Some of the starter code functions have dummy values being returned. This is so that the code can still compile and run even if you don't implement them. But you can obviously change the return statements when you do implement them.
- Use the Node class as is, which means you have to use the setters and getters for the left and right children in order to get an accurate node count.
- Make sure your code works with the provided test code.

- For the toString method, separate nodes in each level using the | symbol. Then move to the next line for the next level. (So each level should be on a separate line.) I recommend doing this method early because it should help with debugging.

Do not:

- Use any additional data structures except for the Queue, which is allowed for the toString method.
- Do multiple loops to update nodes.
- Change the Node class.
- Delete any of the methods even if you don't finish them.
- Mess with the *nodeCount*.

Grading.

The main part of grading will be using the autograder, which will test for both accuracy and efficiency. However, we also look at your code to make sure you are following directions. Deductions can be made based on what we see, including, but not limited to:

- bad coding style
- not following directions (could result in a 0 depending on the issue)
- inefficient code (could result in a 0 depending on the issue)

For coding style, we look for:

- understandable code
- helpful comments
- good indentation
- good method abstraction–i.e. you should be writing reusable methods rather than repeating the same code multiple times
- good use of encapsulation

Submission.

Transfer your files to lectura and make sure they compile and run there. To compile, use the *javac* command followed by the file to be compiled (or *.java for all files ending in .java). Use the *java* command followed by the *.java* filename to run (e.g. *java RBTTest.java*).
Once you are sure your code compiles and runs, submit your files with the following command:

```
turnin csc345pa5 RBT.java
```

Upon successful submission, you will see this message:

```
Turning in:
    RBT.java -- ok
All done.
```