Project 3: Lexicographic Sorting & AVL Trees

Due: Friday 10/22/2021 at 11:59PM

General Guidelines:

The APIs given below describe the required methods in each class that will be tested. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment. Keep in mind that anything that does not need to be public should generally be kept private (instance variables, helper methods, etc.).

Unless otherwise stated in this handout, you are welcome (and encouraged) to add to/alter the provided java files as well as create new java files as needed.

In general, you are not allowed to import any additional classes or libraries in your code without explicit permission from the instructor! Adding any additional imports will result in a 0.

Note on Academic Dishonesty:

Please note that it is considered academic dishonesty to read anyone else's solution code to this problem, whether it is another student's code, code from a textbook, or something you found online. **You MUST do your own work**! You are allowed to use resources to help you, but those resources should not be code, so be careful what you look up online! **We are going to be strict if we find any implementation similar to any implementations on the internet.**

Note on implementation details:

Note that even if your solution passes all test cases, if a visual inspection of your code shows that you did not follow the guidelines in the project, you will receive a 0.

Note on grading and provided tests:

The provided tests are to help you as you implement the project. The Vocareum tests will be similar but not exactly the same. Keep in mind that the points you see when you run the local tests are only an indicator of how you are doing. The final grade will be determined by your results on Vocareum.

Project Overview:

In this project, you will work on two problems. First, you will implement a basic function to Lexicographically compare two objects. In the second part, you will be using the first part objects to implement a basic AVL tree.

Part 1: Lexicographic Compare (20 points)

For Part 1 you will implement a single `compareTo()` function. This function Lexicographically compares the two arrays. Lexicographic order is the conventional ordering used in dictionaries and encyclopedias.

All your implementation will be in the class `Tuple.java` for this part. Most of the code is implemented in the startercode. You will only need to implement 1 function for this part.

You are encouraged to have a look at the API below and the starter code before reading the handout further. Please also have a look at the comments in the starter code.

YOU ARE NOT ALLOWED TO MODIFY THE API.

Class: Tuple.java

Instance Variables	Description
`private int Item[] items`	Item array object for the Tuple

Functions/ Methods	Description
`public Tuple (Item[] items)`	Initializes the items to the parameter
`public Item[] getItems()`	Getter for items

`public void setItems(Item[] items)`	Setter for items
`public String toString()`	Returns a String representation of items
`public int compareTo(Tuple toCompare)`	Compares the current item with the item in toCompare and returns 1, -1, or 0 based on the result

There is also the main function created in the startercode for manual testing

Method to implement:

1. *public int compareTo(Tuple toCompare)*

   This method lexicographically compares the string representation of the two objects. If this is less than toCompare you will return -1, 0 if both the objects are the same, or 1 if this is greater than toCompare. **Please review the examples before you begin implementing.**

   Do not modify the toString() function, it is required for testing

Examples:

You cannot copy these examples directly, please look up TestTuple if you want to generate your own test cases.

```
Tuple t1 = new Tuple<>([a,b,c]);
Tuple t2 = new Tuple<>([d,e,f]);
t1.compareTo(t2)
Output : -1

Tuple t1 = new Tuple<>([d,e,f]);
Tuple t2 = new Tuple<>([a,b,c]);
t1.compareTo(t2)
Output : 1

Tuple t1 = new Tuple<>([a,b,c]);
Tuple t2 = new Tuple<>([a,b,c]);
t1.compareTo(t2)
Output : 0

Tuple t1 = new Tuple<>([a,a,a,a,a,a,a,a,a,a,a,a,b]);
Tuple t2 = new Tuple<>([a,a,a,a,a,a,a,a,a,a,a,a,a]);
t1.compareTo(t2)
Output : 1

Tuple t1 = new Tuple<>([1,2,3,4,5]);
Tuple t2 = new Tuple<>([1,2,4]);
t1.compareTo(t2)
Output : -1

Tuple t1 = new Tuple<>([2]);
Tuple t2 = new Tuple<>([1,0,0,0,0,0]);
t1.compareTo(t2)
Output : 1

Tuple t1 = new Tuple<>([H,a,i,l]);
Tuple t2 = new Tuple<>([H,a,i,l,P,u,r,d,u,e]);
t1.compareTo(t2)
Output : -1
```

Other Information:

1. We use a generic class for this part of the project. For more information about generic classes please refer to htttps://docs.oracle.com/javase/tutorial/java/generics/types.html

2. The name ‘Item’ is a type parameter, a symbolic placeholder for some concrete type to be used by the client. (Ex. String, Integer, etc.)

3. The class will be instantiated as follows (unchecked):
   `Tuple tuple = new Tuple<> ([1,2,3,4]);`
   You can also make it for other data types such as Strings, Arrays, and even Objects of other classes by specifying the object as well.
   `Tuple<Integer> tuple = new Tuple<> ([1,2,3,4]);`

4. Therefore, implementing this generic class will enable you to make a tuple of Integers, Strings, and other complex data types as well.

Testing:

Test class TestTuple has test cases to help you test your implementation. All tests are randomized.

Correct implementation of Tuple will give you 20 points.

You are not allowed to import any libraries. Importing any library will give you a 0.

There is NO PARTIAL CREDIT for this part of the project. This part is also REQUIRED for the implementation of part 2. As there is no partial credit the test cases on Vocareum are the SAME as the ones given to you. However, the test cases themselves are randomized, so while they are the same, they will aggressively check for operations in a randomized sequence.

Part 2: AVL Tress (80 points)

You have studied Left-Leaning Red-Black Trees in class. In this part, you will implement AVL trees insertion. AVL trees, named after inventors Adelson-Velsky and Landis (the inventors of the data structure), is a self-balancing Binary Search tree.
The lookup time of an LLRBT is

$$2\log_2(n)$$

The lookup time of an AVL tree is about

$$1.44\log_2(n)$$

The average case insertion time of an LLRBT is better than that of an AVL tree.

Let us do some math now:

The height of an arbitrary node n is given by the function $H(n)$
Where H(n)is defined as:

$$H(null\ node) = -1$$

$$H(x) = max(H_L, H_R) + 1$$

where
H_L is the height of the left subtree &
H_R is the height of the right subtree

Height of a leaf node is 0

The Balance Factor of an arbitrary node n is given by the function $B(n)$
Where $B(n)$ is defined as:

$$B(n) = H(T_L) - H(T_R)$$

For an AVL tree:

$$|B(n)| \leq 1$$

or for a valid AVL tree B(n)can only have the values -1, 0 or 1

There is a Node Class provided to aid your implementation.

Class: Node.java

Instance Variables	Description
`public Tuple data`	data stored in the node
`public Node left`	a pointer to the left child of the node
`public Node right`	a pointer to the right child of the node
`Public int height`	the height of the node

Functions/ Methods	Description
`public Node (Tuple data)`	Initializes a Node with data and height as 0, and the children as null
`public Node (Tuple data, Node left, Node right, int height)`	Initialize all parameters
`public String toString()`	Returns a String representation of Node

Do not modify the toString() function, it is required for testing. You will not need to modify the node class, but you are allowed to modify it.

Class: AVL.java

Functions/ Methods	Description
public Node insert (Node node, Tuple tuple)	Inserts a new node in the tree rooted at null
public int height (Node node)	Returns the height of the node
public int balanceFactor(Node node)	Returns the balance factor of the node
public String levelOrder(Node node)	Returns the level order traversal of the tree rooted at node
private List<Node> levelOrderHelper(Node curr, int level)	Recursive function to aid level order traversal

Do not modify the height(), balanceFactor(), levelOrder() or the levelOrderHelper() functions, they are required for testing.

Method to implement:

1. public Node insert (Node node, Tuple tuple)
   You will insert a new node in the tree. If the node with the same data already exists, you do not need to insert it. The parameter tuple is the data that is to be stored in the new node.
   The parameter node is the root of the tree. The function returns the root of the tree after insertion.
   After every insertion balance factor of every node must be balanced.

There are many insert cases that you need to account for, and they are given below in the example. Please review them carefully before you write your implementation. You are encouraged to create additional functions to aid your implementation.

HINT: Recursion is key. “To iterate is human, to recurse divine”, L Peter Deutsch. Google Recursion. Proper implementation of this project will help you understand recursion which is going to be highly beneficial for you in future classes or coding assessments.

You are not allowed to traverse the whole tree to correct the balance factor of the nodes after an insert. The insertion should be O(log(n)) operation. You will get a 0 if your insertion violates this.

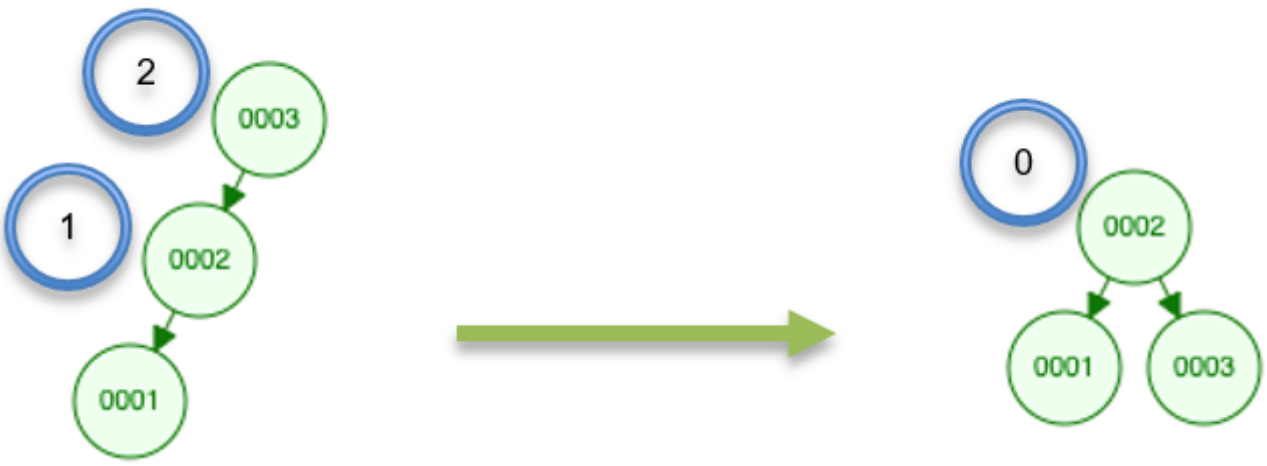
There are two imports in the startercode and you are not allowed to add any imports. Also, the imports are only to be used for testing. You are not allowed to use the ArrayList or the List class.

Insertion Cases:

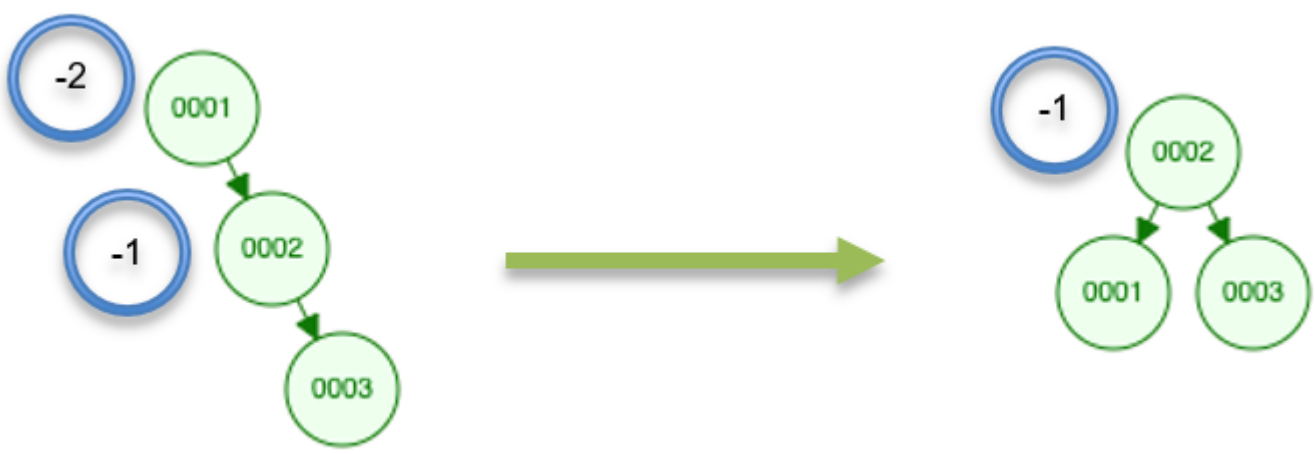
Credits: https://www.cs.usfca.edu/

The blue circles indicate the Balance Factor of the node as defined above

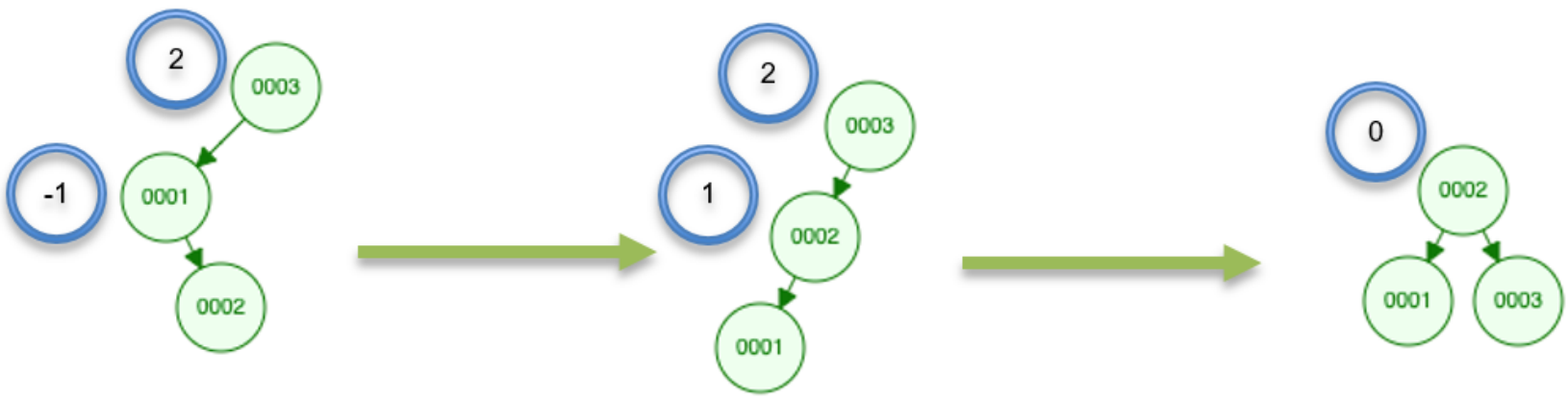
Left Rotate Example:
insert(3), insert(2), insert(1)



Right Rotate Example:
insert(1), insert(2), insert(3)

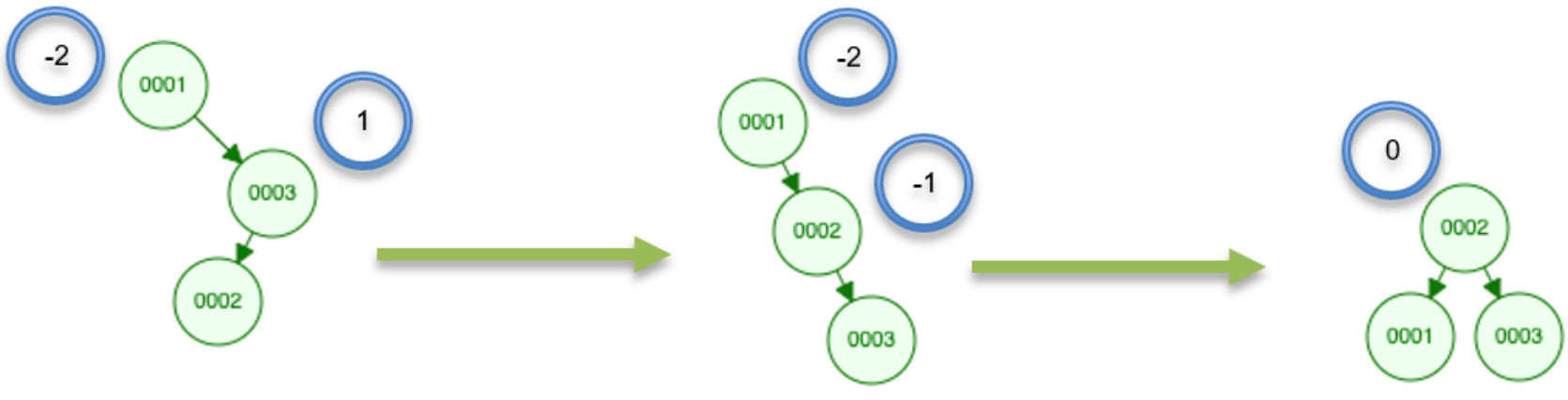


Left-Right rotate (Sometimes called Right-Left rotate):
Insert(3), insert(2), insert(1)



Right-Left rotates (Sometimes called Left-Right rotate):

Insert(3), insert(2), insert(1)



TESTING:

The first part is worth 20 points and the second part is worth 80 points. A test class for each part is provided. You can run them to get an idea of how you are progressing on the project.

Passing the local tests does NOT guarantee points on Vocareum. You will be tested against testcases other than the ones given to you. However, the hidden test cases will be similar to the ones given to you.

Please note that Vocareum is not for testing. Under no circumstances will the submission count be increased.

SUBMISSION:

Upload Tuple.java, Node.java, and AVL.java to Vocareum. You have 10 submissions.

GRADING:

Please read the grading policy carefully:
Due to the high demand for more flexible grading, there are multiple points you can get.

1. Part 1: 20 points
    a. All or nothing
    b. This part is required for part 2 if you get a zero then you will get a zero on part 2
2. Part 2: 80 points
    a. 10 points for left rotate tests
    b. 10 points for right rotate tests
    c. 15 points for left rotate first and then right rotate (The 3rd example)
    d. 15 points for right rotate first and then left rotate (The 4th example)
    e. 30 points for rigorous testing of the whole tree
