Laboratory 7: Tables and Locality of Reference

In this lab you will create a **table**. Tables are high level data structures that emphasize retrieval over insertion and deletion. If a table is implemented with a binary search tree, a retrieval in the average case will be O(logn). However, it is possible to improve the situation if you take advantage of the **locality of reference** principle. **Temporal locality of reference** states that if an item is used, it will be used again in the near future. **Spatial locality of reference** states that if an item is used, items near it will be used in the near future. Thus, when an item is retrieved, you will move it to the root while maintaining the binary search tree property, making subsequent retrievals of this item O(1) if it is retrieved again in the near future. Moving the item to the root while maintaining the BST property will require rotations, discussed below. In order to make sure that the item was moved to the root, it will be convenient to add a level order traversal to the TreeIterator interface. In this type of traversal, the root item will be displayed first.

Lab:

- TreeIterator
    - Iterator Design Pattern
    - Level Order Traversal
- Binary Search Tree
    - Rotate Left
    - Rotate Right
- Adaptable Binary Search Tree
    - Temporal Locality of Reference
- Tables
    - Adapter Design Pattern

Download the following files:

- TreeException.java
- TreeIterator.java
- TreeNode.java
- BinarySearchTree.java
- BinaryTreeBasis.java
- BinaryTreeIterator.java
- SearchTreeInterface.java
- TableException.java
- TableInterface.java
- FileIO.class
- FileIOException.class
- Song.class
- CD.class //the CD title is the search key
- KeyedItem.class //no package necessary
- AdaptableTableDriver.java
- queue.jar
- cds.txt
- make.bat

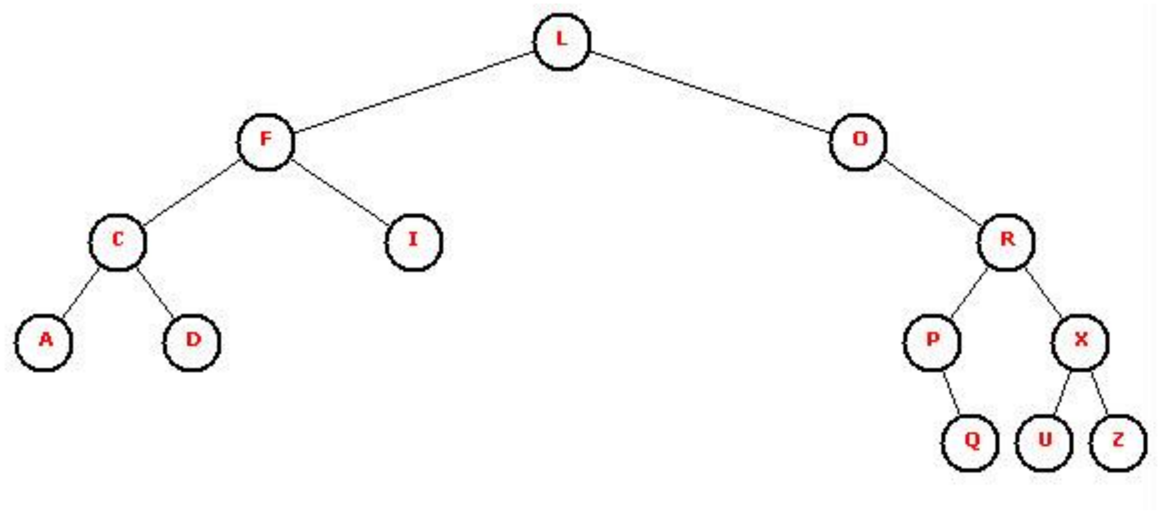
Part I: Level Order Traversal

Add the following method to TreeIterator:

- public void setLevelorder()

Implement this method in **BinaryTreeIterator**. The trick is to make use of a local queue (local to the setLevelOrder method) that will hold TreeNodes. The queues are generics enabled, so forcing the local queue to accept only TreeNodes might be useful. There is also an instance variable queue that contains the **actual items** (pulled from the TreeNodes) that will be given to the user when next() is called during traversal. Start by placing the **root** TreeNode on the local queue (if the root is null, don't do anything). Loop until the local queue is empty. When a TreeNode is pulled from the local queue, extract and place its item on the instance variable queue. In order to get a level order traversal, the left TreeNode, if present, will need to be processed first, and then the right TreeNode, if present. Do not use recursion to complete this traversal. Note that tree traversals take a **snapshot** of the contents of the tree when the specific traversal is set. This prevents changes to the iterator if items are added or removed from the tree during traversal.

The Iterator design pattern uses an "external" iterator, requiring the user to set up the loop and to perform some operation on each item in the data structure manually. In a later lab, you will investigate the Command design pattern, an "internal" iterator where the user specifies what is to be done to each item, but the data structure actually handles the loop and performs the work.



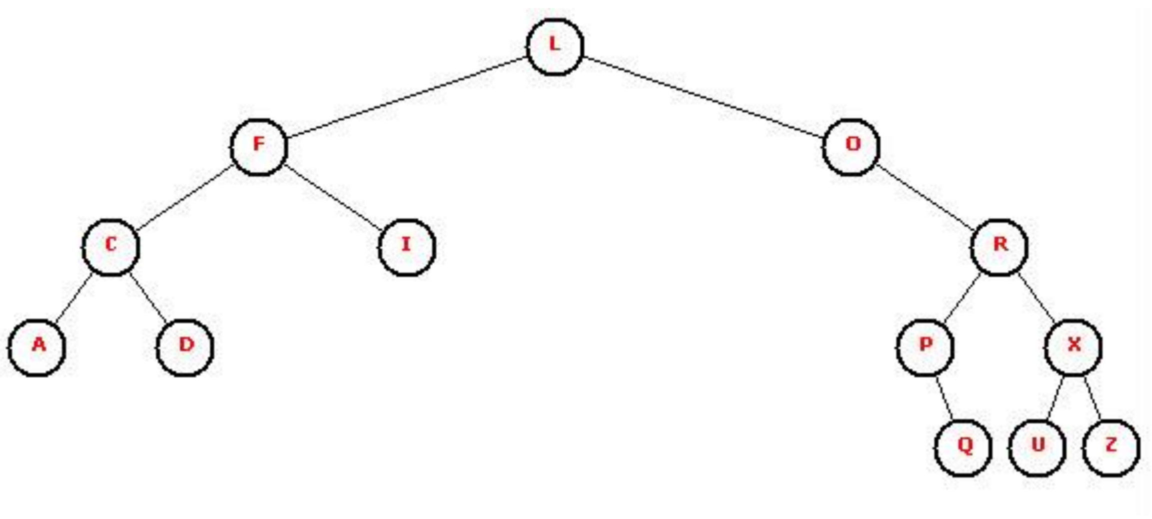
The level order traversal of the above image is L, F, O, C, I, R, A, D, P, X, Q, U, Z

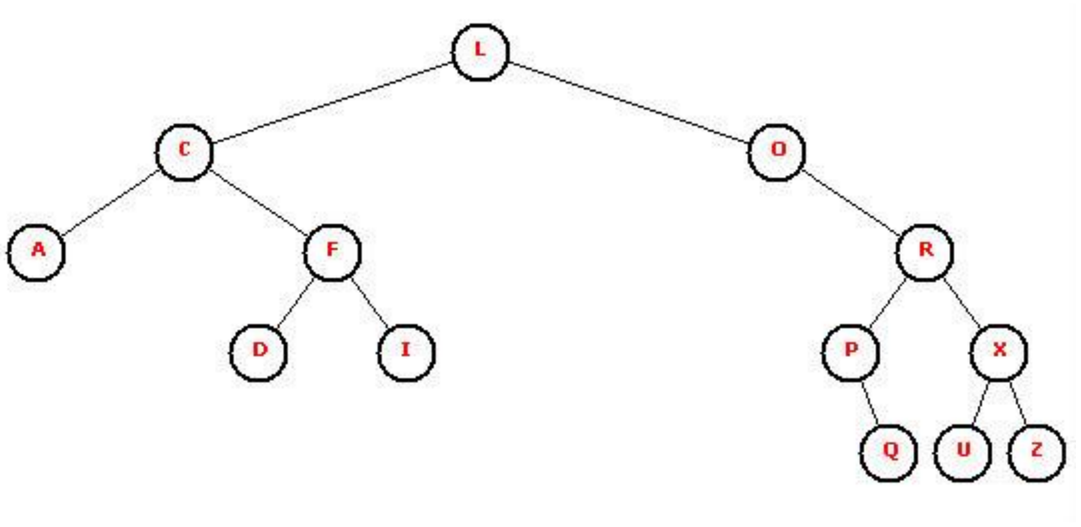
Part II: Rotations

Add two methods to **BinarySearchTree** to perform **left and right rotations**. **These methods should be short and clean**. Call your methods **rotateRight** and **rotateLeft**. Rotations shift some nodes in the tree upwards and some downwards while maintaining the binary search tree property. Have your methods accept the root of a subtree that must be rotated as a parameter, and return the new root of the subtree after the rotation.

The **rotate right** method is demonstrated in the following figure. Starting with the binary search tree shown in the previous part, the method is passed tree node F and returns tree node C. Pay attention to the root (F), root's left child (C), and root's left child's right child (D), which we will call root, left, and leftright respectively.

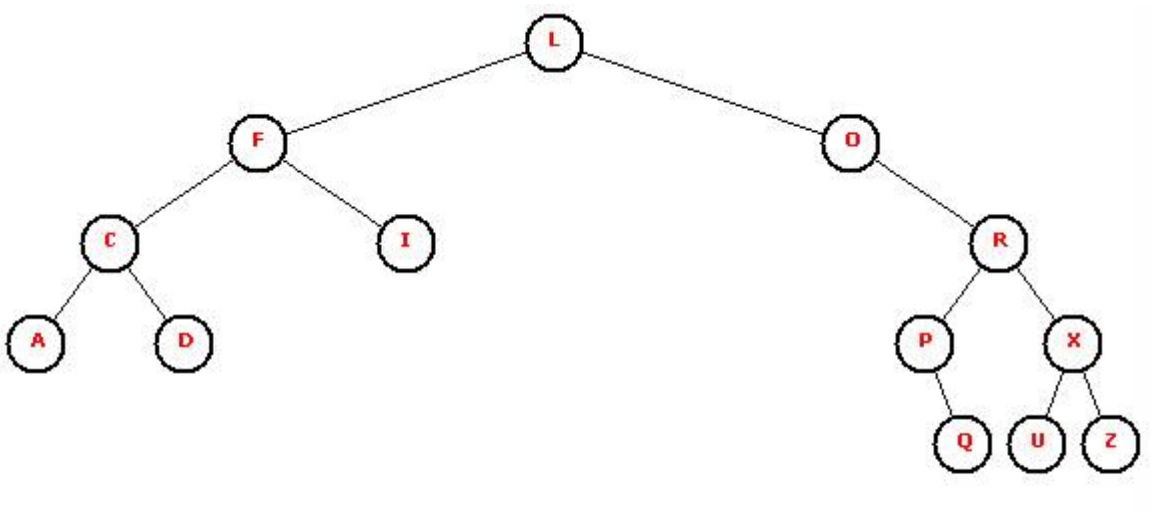
- First, left becomes the new root
- Second, root becomes left's right child
- Third, leftright becomes root's left child
- Finally, all other nodes stay attached

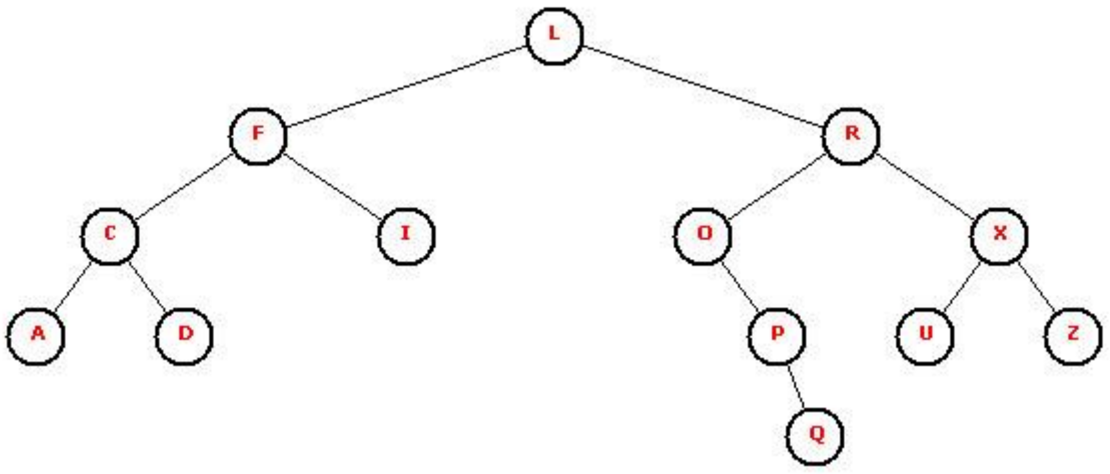




The **rotate left** method is demonstrated in the following figure. Starting with the binary search tree shown in the previous part, the method is passed tree node O and returns tree node R. pay attention to the root (O), root's right child (R), and root's right child's left child (P), which we will call root, right, and rightleft respectively.

- First, right becomes the new root
- Second, root becomes right's left child
- Third, leftright becomes root's right child
- Finally, all other nodes stay attached





Part III: Temporal Locality of Reference

Extend BinarySearchTree in a new class, **AdaptableBinarySearchTree**. When an item is retrieved, AdaptableBinarySearchTree will be adjusted to have that item moved to the root. In addition to the constructor, write two **new** methods:

- public KeyedItem retrieve(Comparable searchKey)
  - top level of recursion
  - sets the root node in case the rotation was at root level
  - catches tree exception in order to return null
- private TreeNode retrieveItemAdapt(TreeNode tNode, Comparable searchKey)
  - recursive method that adjusts retrieved items to the root

This method uses recursion and your rotate methods to adjust the links in the tree so that the requested item is moved to the root. **At the top level of recursion, set the root node** in case the rotation was at the root level. This operation will allow temporal locality of reference, but what about spatial locality of reference? Cast the TreeNode object to type KeyedItem to allow for easy comparisons.

If the item is not present, SearchTreeInterface requires that a null be returned. This is a problem since returning null up your recursive chain will cause links to be dropped. Therefore, if the item is not present, throw a TreeException (no work will be done) that will be caught at the top level of recursion (in the public retrieve method). Simply return null when a TreeException is caught.

Part IV: Tables

Implement TableInterface with a class called **TableSTBased** (**Adapter** Design Pattern). This class delegates the work to an **AdapatableBinarySearchTree** instance variable (use an interface type - SearchTreeInterface, however). Since this tree is to be used to implement a table, make sure that duplicates are not allowed into the tree (calling insert on your tree will generate a TreeException if a duplicate is found). You will also need to keep track of the number of items in the table (add a size instance variable and remember to **increase and decrease the size** as appropriate). The iterator method will set and return a level order traversal to check that the adaptable property is satisfied. You should also set and return an inorder traversal to check that the binary search tree property is satisfied (temporarily comment out setLevelorder, replacing it with setInorder).

Complete the driver to thoroughly test your work. Retrieve several items calling **displayTable** each time to check that your rotations are working. The item that you retrieved should move to the root and be the first item listed in the level order traversal. Remember to test both level order and in order traversals.

Only one submission per team is necessary, but please make sure to include both names at the top of your source code, as well as in the comments section when submitting the lab, so both people can get credit.