Com S 228 Spring 2020

Project 5 : Video Store Transactions

```
Due at 11:59pm, Sunday, Apr 26
```

1. Project Overview

In this project, you will implement a self-adjusting binary search tree called the splay tree. It carries out self-optimization based on the heuristic that recently accessed elements are quick to access again. With *n* nodes, the tree has the amortized running time of *O(logn)* per operation. (This is the time averaged over a worst-case sequence of operations.) Next, you will use a splay tree to simulate customer rental and return transactions at a video store.

The class *SplayTree* implements the data structure of splay tree. The class *Video* represents a video with rental information, while the class *VideoStore* makes use of a *SplayTree* object to simulate video transactions. You also need to implement the class *Transactions* to simulate video rental and return activities. To these four classes you may add new instance variables and methods. You cannot, however, rename or remove any existing ones, or change any from public to protected (or private), or vice versa.

2. Splay Tree

The *SplayTree* class has been completed for you. You **do not** need to implement *SplayTree*.

If you would like to learn more about the inner workings of splay trees, please refer to the provided code, the included javadocs, and the attached *splayTrees.pptx*. A description of parts of the the classes functionality follows.

The splay tree is implemented by the following generic class:

public class SplayTree<E extends Comparable<? super E>> extends AbstractSet<E>

A node in the tree is fully implemented by the public class Node within *SplayTree*.

2.1 Tree Construction, Methods, and Iterator

Three constructors are provided for the SplayTree class:

public SplayTree()

public SplayTree(E data)

public SplayTree(SplayTree<E> tree)

- The first constructor creates an empty tree (with *null* root).
- The second constructor merely creates a tree root to store given data. The rest of the tree needs to be constructed by repeatedly calling the following method which performs a binary search tree addition:

public boolean addBST(E data)

- For efficiency of tree initialization, the method does not splay at the newly created node.
- The third constructor copies over the structure of an existing splay tree.

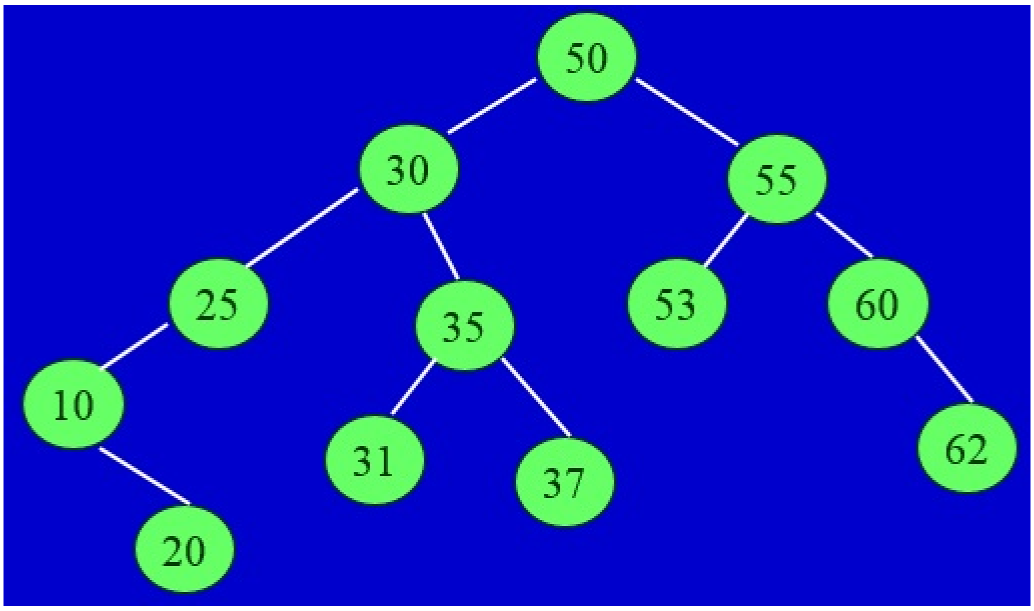
The class has a private iterator class *SplayTreeIterator*. **No iterator method splays** at a node.

2.2 Tree Display

The *toString()* method has been overridden to display the configuration of the splay tree under the following rules:

- Every node of the tree occupies a separate line with its data written out only. (Assume that the *toString()* method for the data class *E* has been properly overridden.)
- The data stored at a node at depth *d* is printed with indentation *4d* (i.e., preceded by *4d* blanks).
- Starting at the root (at depth 0), it displays the nodes in a preorder traversal. More specifically, suppose a node *n* is shown at line *l*. Then, starting at line *l+1* , it
    - recursively prints all nodes in the left subtree (if any) of *n*;
    - recursively prints all nodes in the right subtree (if any) of *n*.
- If a node has a left child but no right child, prints its right child as null.
- If a node has a right child but no left child, prints its left child as null.
- If a node is a leaf, it is printed with no further recursion.

The *toString()* method for the class E is assumed to be properly overridden. Shown next is a splay tree with 12 nodes to store integers (*E* instantiated as *int*). What follows is the expected output that would be generated from calling the *toString()* and *System.out.println()*.



```
50
    30
        25
            10
                null
                20
            null
        35
            31
            37
    55
        53
        60
            null
            62
```

3. Video Store

The class *VideoStore* simulates rental and return transactions at a video store. Videos are objects of the Video class such that a **single** Video object represents all video copies of the same film.

```
public class Video implements Comparable<Video>

{

private String film;

private int numCopies;

private int numRentedCopies;

public Video(String name, int n) throws IllegalArgumentException

{ ... }

...

}
```

Suppose the store keeps one copy of the film *The Godfather*. Then, a constructor call *Video("The Godfather", 1)* will create an entry to be put on the store's inventory. Videos are compared **by film title** in the alphabetical order.

3.1 Inventory

The class VideoStore employs one splay tree to store its inventory:

protected SplayTree<Video> inventory;

Every node in the tree is an object of the Node class.

3.2 Constructors and Video File

Two constructors are provided for the class:

public VideoStore()

public VideoStore(String videoFile) throws FileNotFoundException

The default constructor initializes the inventory to be an empty splay tree. It is expected to later call the method *setUpInventory()* to complete the inventory construction.

The second constructor builds inventory over a *video file*. Each line of the file lists a film title followed by one or more blanks and then the number of video copies within a pair of parentheses.

1. A negative number following a film title is treated as zero.
2. If a line has a film title only, then the number of copies defaults to one.
3. A film **cannot** appear on multiple lines in the file.

A sample video file videoList1.txt has the content below:

```
The Godfather
Forrest Gump
Brokeback Mountain (1)
A Streetcar Named Desire (1)
Slumdog Millionaire (5)
Taxi Driver (1)
Psycho (1)
Singin' in the Rain (2)
```

(In case you are momentarily short on film titles to compose a video file for testing, just check out American Film Institute's list of 100 Greatest Movies of All Time at http://www.afi.com/100years/movies.aspx)..)

The name of a video file is always provided by the parameter *videoFile* in various methods in the class *VideoStore*. The file format is assumed to be **correct** so you do **not** need to worry about checking it.

3.3 Video Acquisition

Whenever new videos are acquired by the store, their records will be added to the tree inventory. Additions are carried out by three methods:

public void addVideo(String videoName, int n)

public void addVideo(String videoName)

public void bulkImport(String videoFile) throws FileNotFoundException

The last method adds multiple videos of the films from a video file whose format was given in Section 3.2. Please refer to their javadocs when you implement the above methods.

3.4 Video Query, Rental and Return

Methods are provided to simulate video queries and transactions. The stores supports single and bulk rentals and returns.

public boolean available(String videoName) *public void videoRent(String videoName, int n) throws IllegalArgumentException, FilmNotInInventoryException, AllCopiesRentedOutException*

public void bulkRent(String videoFile) throws FileNotFoundException, IllegalArgumentException, FilmNotInInventoryException, AllCopiesRentedOutException

public void videoReturn(String videoName, int n) throws IllegalArgumentException, AllCopiesRentedOutException

public void bulkReturn(String videoFile) throws FileNotFoundException, IllegalArgumentException, FilmNotInInventoryException

In case a method may throw multiple exceptions, use the following decreasing order of importance:

1. FileNotFoundException
2. IllegalArgumentException
3. FilmNotInInventoryException
4. AllCopiesRentedOutException

The method *bulkRent()* and *bulkReturn()*, accepting a parameter *videoFile*, should always **read through the entire video file. Concatenate the messages of all the exceptions into one string**, which is then thrown with the exception ranked the **highest**. In the concatenated message, the exceptions should appear in the **same order** as the corresponding films in the video file.

For example, if *bulkRent()* is executed with a video file which requests sequentially 1 copy of *The Silence of the Lambs* (not in inventory), -2 copies of *Forrest Gump*, and 1 copy of *The Godfather* (rented out), then it will throw an *IllegalArgumentException* and print out the following message:

```
Film The Silence of the Lambs is not in inventory
Film Forrest Gump has an invalid request
Film The Godfather has been rented out
```

4. Simulation of Video Transactions

Suppose a VideoStore object has been constructed over the file *videoList1.txt* given in

Section 3.2. A second video file *videoList2.txt* has the content below:

```
The Godfather (1)

Forrest Gump (1)

Slumdog Millionaire (4)
```

A third video file *videoList3.txt* has the content:

```
Forrest Gump
Slumdog Millionaire (1)
```

Below is a sample simulation scenario executed by the *main()* method in the class *Transactions*. (User keystrokes are shown in bold and a larger font.)

```
Transactions at a Video Store
keys: 1 (rent)      2 (bulk rent)
      3 (return)    4 (bulk return)
      5 (summary)   6 (exit)

Transaction: 1
Film to rent: The Godfather

Transaction: 2
Video file (rent): videoList2.txt
Film The Godfather has been rented out

Transaction: 1
Film to rent: Brokeback Mountain

Transaction: 3
Film to return: Slumdog Millionaire (2)

Transaction: 1
Film to rent: The Silence of the Lambs
Film The Silence of the Lambs is not in inventory

Transaction: 1
Film to rent: Singin' in the Rain (2)

Transaction: 4
Video file (return): videoList3.txt

Transaction: 5
Rented films:

Brokeback Mountain (1)
Singin' in the Rain (2)
Slumdog Millionaire (1)
The Godfather (1)

Films remaining in inventory:

A Streetcar Named Desire (1)
Forrest Gump (1)
Psycho (1)
Slumdog Millionaire (4)
Taxi Driver (1)
```

```
    Transaction: 6
```

Your code needs to print out text messages in the same format for user interactions. Please note the following:

- To request for the rental or return of a single video, simply enter the film title. You may also enter "(1)" after the title though it is redundant.
- To request for multiple copies of the same film, enter the number of copies within a pair of parentheses after the film title.
- The transaction numbered 5 in the above scenario lists rented films and unrented films in the alphabetical order decided by the *compareTo()* method for *String* objects.

5. Submission

Write your classes in the *edu.iastate.cs228.hw5* package. **Turn in the zip file not your class files**. Please follow the guideline posted under Documents & Links on Canvas.

You are not required to submit any JUnit test cases. Nevertheless, you are encouraged to write JUnit tests for your code. Since these tests will not be submitted, feel free to share them with other students.

Include the Javadoc tag @author in every class source file you have made changes to. Your zip file should be named *Firstname_Lastname_HW5.zip*.