Assignment 2

My PlayStation Friends

SP2

Data Structures

April 2020

1 Introduction

Needing a way to manage all your PlayStation friends, you decide to build a back-end system for adding, removing and maintaining them. The idea is to organise your friends so you can search for individuals, search for players who have the same games as you, determine rankings, and view player information and trophies. At the same time, you'd like your search queries to be fast, ruling out basic structures like arrays and linked lists. Deciding that the most important factor for ordering your friends is level, you build a binary search tree (BST) structure, using the level (actually a function on level, see section 4) as the key.

In this assignment we will build a BST structure, with each node representing a PlayStation friend. Each friend node contains information about that player, including their username, level and date of birth, along with attached data structures for their games (single linked-list) and trophies (ArrayList). In accordance with the rules of BSTs, each friend has a parent node and two child nodes, left and right. From any one node, all nodes to the left are less (lower level) and all nodes to the right are greater (higher level). Due to this, searching for higher or lower-levelled players is, on average, a O(log*n*) process.

This assignment consists of a number of parts. In part A you will setup the basic class structure, ensuring that the test suite is able to run without error. In part B you will implement the basic structures needed by `User` to hold multiple `Trophy` and `Game` objects. In part C you will create your BST-based friend database. Finally, in part D you will improve the efficiency of your tree by implementing AVL balancing. You are free to add your own methods and fields to any of the classes in the `Database` package, but do not change any existing method prototypes or field definitions. A testing suite has been provided for you to test the functionality of your classes. These tests will be used to mark your assignment, but with altered values. This means that you cannot hard-code answers to pass the tests. It is suggested that you complete the assignment in the order outlined in the following sections. Many of the later-stage classes rely on the correct implementation of their dependencies.

1.1 Importing into eclipse

The Assignment has been provided as an eclipse project. You just need to import the project into an existing workspace. See Figure 1 for a visual guide. Make sure that your Java JDK has been set, as well as the two jar files that you need for junit to function. This can be found in Project → Properties → Java Build Path → Libraries. The jar files have been provided within the project; there is no need to download any other version and doing so may impact the testing environment.

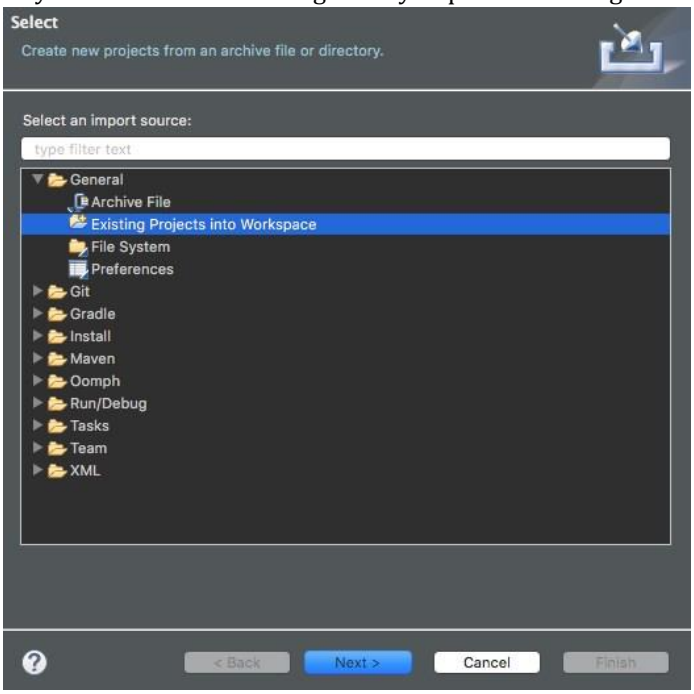

Figure 1: Importing the project through File → Import

2 Part A

If you run the testing suite, you will be lovingly presented with many errors.
Your first task is to complete the class implementations that the tests expect (including instance variables and basic methods) to remove all errors from the testing classes.

2.1 Game

The Game class represents one PlayStation game and includes general information, namely the title, release date, and total number of available trophies. In addition, it holds a reference to another Game object. This will be important in section 3 where you will make a single-linked list of games. The Game class requires the following instance variables:

```
private String name; private
Calendar released; private Game
next; private int totalTrophies;
```

The toString method should output a string in the following format (quotation marks included):

```
"Assassin's Creed IV: Black Flag", released on: Nov 29, 2013
```

Hint: A printed Calendar object may not look as you might expect. Take a look at APIs for java date formatters.

You should also generate the appropriate accessor and mutator methods. GameTester will assign marks as shown in Table 1:

Table 1: GameTester mark allocation

Test	Marks
testConstructor	2
toStringTest	3
Total:	**5**

2.2 Trophy

The Trophy class represents one PlayStation trophy and includes information about its name, date obtained and the game from which it was earnt. Additionally, its rank and rarity values are set from finite options available through enumerator variables. The Trophy class requires the following instance variables:

```
public enum Rank {
    BRONZE, SILVER, GOLD, PLATINUM
} public enum Rarity {
    COMMON, UNCOMMON, RARE, VERY_RARE, ULTRA_RARE
}
```

```
private String name; private Rank
rank; private Rarity rarity; private
Calendar obtained; private Game
game;
```

The toString method should output a string in the following format (quotation marks included):

"What Did You Call Me?", rank: BRONZE, rarity: RARE, obtained on: May 04, 2014

You should also generate the appropriate accessor and mutator methods. GameTester will assign marks as shown in Table 2:

Table 2: TrophyTester mark allocation

Test	Marks
testConstructor	2
toStringTest	3
Total:	**5**

2.3 User

The User class represents a PlayStation user and, more generally, a tree node. Most importantly when using as a tree node, the class must have a key on which the tree can be ordered. In our case, it is a double named key. This key is a simple function based on the combination of a user's username and level. As levels are whole numbers and likely not unique, a simple method (see calculateKey snippet below) is used to combine the two values into one key whilst preserving the level. For example, imagine that the hashcode for username "abc" is 1234 and the user's level is 3. We do not want to simply add the hash to the level as that would not preserve the level and would lead to incorrect rankings. Instead, we calculate 1234/10000 to get 0.1234. This can then be added to the level. As the usernames must be unique, our node keys are now also unique [1] and the user level is preserved.

```
private double calculateKey() { int hash =
    Math.abs(username.hashCode()); // Calculate
    number of zeros we need int length =
    (int)(Math.log10(hash) + 1);
```

[1] A string's hash value can never be guaranteed to be unique, but for the purposes of this assignment we will assume them to be.

```
        // Make a divisor 10^length double divisor =
        Math.pow(10, length);
        // Return level.hash return level +
        hash / divisor;
}
```

The User class requires the following instance variables:

```
private String username; private int level;
private double key; private
ArrayList<Trophy> trophies; private
GameList games; private Calendar dob;
private User left; private User right;
```

An ArrayList type was chosen for variable trophies as you figured it would be easier to add a new trophy to a list than a standard array, and you probably would mostly just traverse the list in order. A GameList object (see section 3) was chosen as the structure for storing games as a custom single linked-list is more appropriate for writing reusable methods.

The toString method should output a string in the following format:

```
User: Pippin

Trophies:
"War Never Changes", rank: BRONZE, rarity: COMMON, obtained on: Mar 26, 2017
"The Nuclear Option", rank: SILVER, rarity: UNCOMMON, obtained on: Mar
    26, 2017
"Prepared for the Future", rank: GOLD, rarity: UNCOMMON, obtained on: Mar 26, 2017
"Keep", rank: SILVER, rarity: RARE, obtained on: Mar 26, 2017

Games:
"Yooka-Laylee", released on: May 11, 2017
"Mass Effect Andromeda", released on: Apr 21, 2017
"Persona 5", released on: May 04, 2017

Birth Date: May 23, 1980
```

You should also generate the appropriate accessor and mutator methods. UserTester will assign marks as shown in Table 3:

3 Part B

In this section you will complete the GameList single linked-list for storing Game objects.

Table 3: UserTester mark allocation

Test	Marks
testConstructor	2
toStringTest	3
Total:	**5**

3.1 GameList

The GameList class provides a set of methods used to find Game objects that have been linked to form a single-linked list as shown in Figure 2. The head is a reference to the first Game node, and each Game stores a reference to the next Game, or null if the Game is at the end.

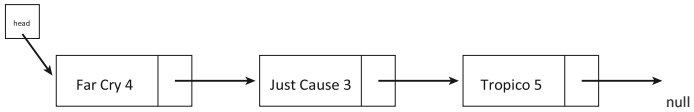

Figure 2: The single-linked list structure built in GameList

The GameList class requires only one instance variable:

public Game head

There are a number of methods that you must complete to receive marks for this section. They can be completed in any order. Your tasks for each method are outlined in the following sections.

3.1.1 void addGame(Game game)

This method should add the provided game to the end of your linked list. It should search for the first available slot, and appropriately set the previous game's next variable. All games must be unique, so you should check that the same game has not already been added. Note that the tests require that the provided Game object is added, not a copy. If the GameList head variable is null, head should be updated to refer to the new game. If the provided Game object is null, an IllegalArgumentException should be thrown.

3.1.2 Game getGame(String name)

getGame should traverse the linked list to find a game with a matching name. If the game cannot be found, the method should return null. If the name provided is null, the method should throw an IllegalArgumentException.

3.1.3 void removeGame(String name) — void removeGame(Game game)

There are two overloaded removeGame methods with one taking as an argument a String, the other a Game. Both methods should search the linked list for the target game and remove it from the list. You should appropriately set the previous node's next variable or set the head variable, if applicable. Both methods should throw an IllegalArgumentException if their argument is null.

3.1.4 String toString()

This method should output a string in the following format:

"Assassin's Creed IV: Black Flag", released on: Nov 29, 2013
"Child of Light", released on: May 01, 2014

3.2 GameListTester

GameListTester will assign marks as shown in Table 4:

Table 4: GameListTester mark allocation

Test	Marks
getGameNullArg	1
getGame	2
addGame	2
addGameExists	1
addGameNullArg	1
removeGameNullArg	1
removeGameString	2
removeGameObject	2

toStringTest	3
Total:	**15**

4 Part C

In this section you will complete your binary search tree data structure for storing all your friends' information.

4.1 BinaryTree

Now that all the extra setup has been completed, it can all be brought together to form your tree structure. The BinaryTree class provides a set of methods for forming and altering your tree, and a set of methods for querying your tree. The goal is to form a tree that adheres to BST rules, resulting in a structure such as shown in Figure 3.

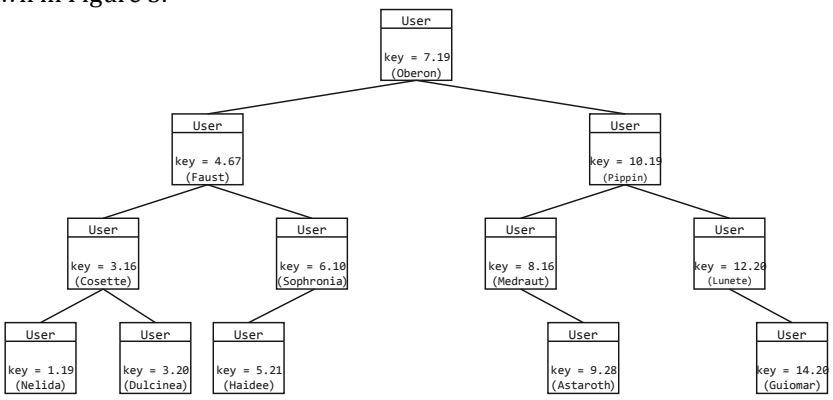

Figure 3: The binary search tree structure built in BinaryTree

The BinaryTree class requires only one instance variable:

public User root

There are a number of methods that you must complete to receive marks for this section. They can be completed in any order. Your tasks for each method are outlined in the following sections. Remember that you can add any other methods you require, but do not modify existing method signatures.

4.1.1 boolean beFriend(User friend)

The beFriend method takes as an argument a new User to add to your database. Adhering to the rules of BSTs, you should traverse the tree and find the correct position to add your new friend. You must also correct set the left, right and parent

variables as applicable. Note that the tests require that you add the provided User object, not a copy. If the User key is already present in the tree, this method should return false. If the User argument is null, this method should throw an IllegalArgumentException. As an example, adding a User with key 6 into Figure 3 results in the tree shown in Figure 4.

4.1.2 boolean deFriend(User friend)

The deFriend method takes as an argument a User to remove from your database. This method should search the tree for the target friend and remove them. This should be achieved by removing all references to the User and

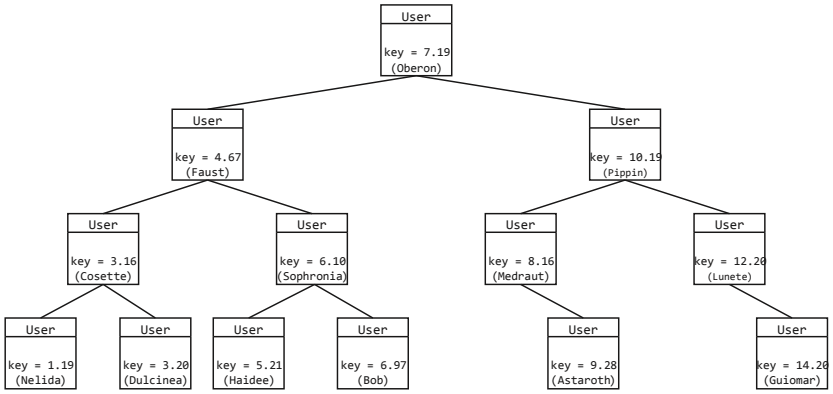

Figure 4: The BST after adding a friend with key 6

updating the left, right and parent values as applicable. deFriend should return true if the friend is successfully removed, false if not found or some other error case. If the friend object is null, an IllegalArgumentException should be thrown. As an example, removing the User with key 4 from Figure 3 results in the tree shown in Figure 5.

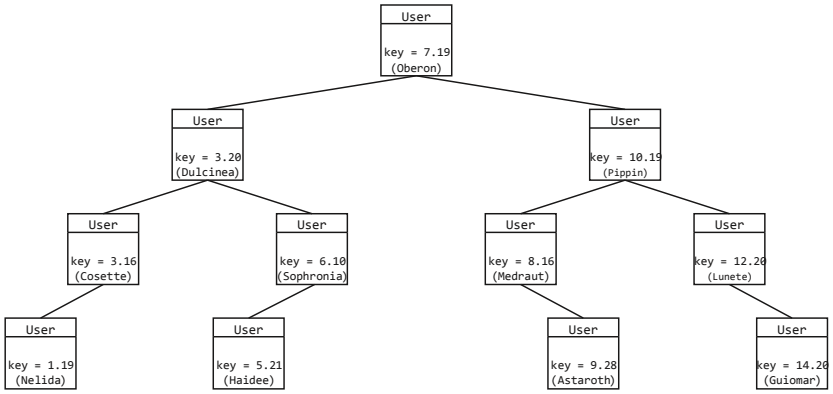

Figure 5: The BST after removing the friend with key 4

4.1.3 int countBetterPlayers(User reference)

The countBetterPlayers method takes as an argument a User from which you should search for players with higher rank. This method should search from the reference user and increment a counter of better players to return. You should return the number of better players, 0 if there are none. Note that a greater key value does not necessarily equal a higher level. If the User argument is null, this method should throw an IllegalArgumentException. As an example, using User with key 7 from Figure 3, this method should return 5.

4.1.4 int countWorsePlayers(User reference)

The countWorsePlayers method takes as an argument a User from which you should search for players with lower rank. This method should search from the reference user and increment a counter of worse players to return. You should return the number of worse players, 0 if there are none. Note that a lower key value does not necessarily equal a lower level. If the User argument is null, this method should throw an IllegalArgumentException. As an example, using User with key 7 from Figure 3, this method should return 4.

4.1.5 User mostPlatinums()

The mostPlatinums method should search the database and return the player who has the most platinum-level trophies. If there are multiple players with the same number of platinum trophies, gold trophies should be used to break the tie. If there are no users with platinum trophies this method should return null.

4.1.6 void addGame(String username, Game game)

The addGame method takes two arguments, a String username and Game game. You should search your database for a matching user and add the new game to their GameList. You should also check that they do not already have that game in their collection. If either argument is null, this method should throw an IllegalArgumentException.

4.1.7 void addTrophy(String username, Trophy trophy)

The addTrophy method takes two arguments, a String username and Trophy trophy. You should search your database for a matching user and add the new trophy to their trophies. You should also check that they do not already have the trophy to be added and that they do not already have all available trophies for the trophy's game. If either argument is null, this method should throw an IllegalArgumentException.

4.1.8 void levelUp(String username)

The levelUp method takes as an argument a String username that you should use to search for the matching user in the database. You should then increment that user's level by one. If this breaches any BST rules you should make the necessary adjustments to the tree. As an example, Figure 6 shows an invalid tree after a level-up and Figure 7 shows the correct alteration. If the username argument is null, this method should throw an IllegalArgumentException.

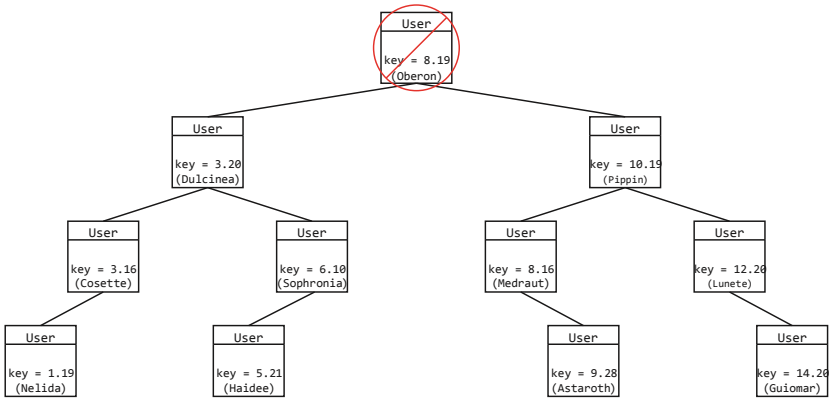

Figure6:InvalidBSTafterlevel-upapplied. Thekeygeneratorallowsfor multiple level-8 users.

4.2 BinaryTreeTester

BinaryTreeTester will assign marks as shown in Table 5.

5 Part D

In this final section you will implement the AVL tree balancing algorithm. This will give your tree more efficiency as it will maintain a perfect balance as different values are added.

5.1 boolean addAVL(User friend)

The addAVL methods takes as an argument a User friend that you should add to the tree. AVL rules should apply, which means that if the tree becomes unbalanced, rotations should be performed to rectify. This excellent visualisation may help you understand how to implement any rotations that may be necessary: https://www.cs.usfca.edu/ galles/visualization/AVLtree.html The problem is broken into stages in the testing file. Tests are only provided for ascending values, meaning they only test left rotations. Alternate tests will also test right rotations so be sure to test adding descending values. If the friend argument is null, this method should throw an IllegalArgumentException.

5.2 AVLTester

Marks for AVLTest will be assigned as shown in Table 6.

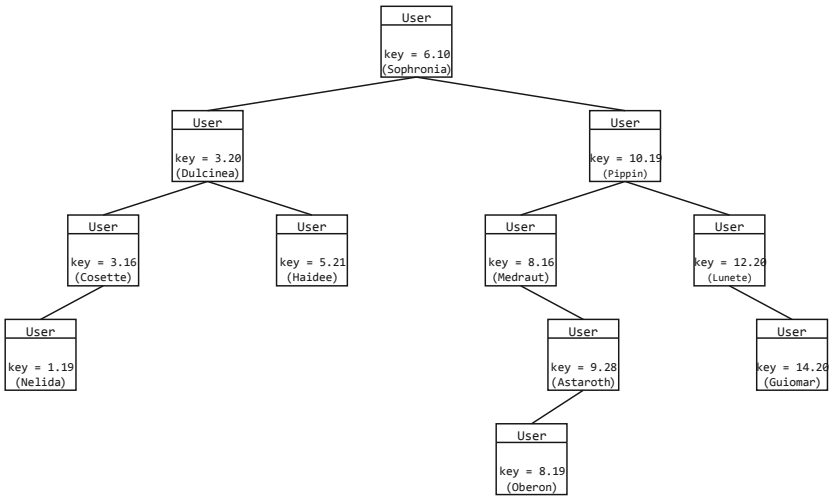

Figure 7: Correct operations applied after level-up to preserve BST structure.

Table 5: BinaryTreeTester mark allocation

Test	Marks
beFriendNullArg	1
beFriendDuplicate	1
beFriend	6
deFriendNullArg	1
deFriendNonExistent	1
deFriend	7
countBetterPlayersNullArg	1
countBetterPlayersNonExistent	1
countBetterPlayers	4
countWorsePlayersNullArg	1
countWorsePlayersNonExistent	1
countWorsePlayers	4
mostPlatinums	4
addGameNullArg	1
addGame	4
addTrophyNullArg	1
addTrophy	4
levelUpNullArgs	1
levelUp	7
toStringTest	3
Total:	**54**

Table 6: avlTester mark allocation

Test	Marks
avlStage1	6
avlStage2	5
avlStage3	5
Total:	**16**