Data Structures: Assignment 2

Pokemon Unite Friend Tracker

AnDe 2024

1- Introduction

A MOBA recently got released in China called Pokemon Unite. As you enjoy playing various MOBA-style games you decide to give it a try. While playing, you've decided you want to track your skills and progression in comparison to your friends. To accomplish this, you decide to build a back-end system for adding, removing and maintaining them. The idea is to organise your friends so that you can search for individuals, search for players who play certain Pokemon, determine rankings based on account level and view player information and the ribbons they've earnt. 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 their account level, you build a binary search tree (BST) structure, using the level (actually represented via a method, see Section 4) as the key.

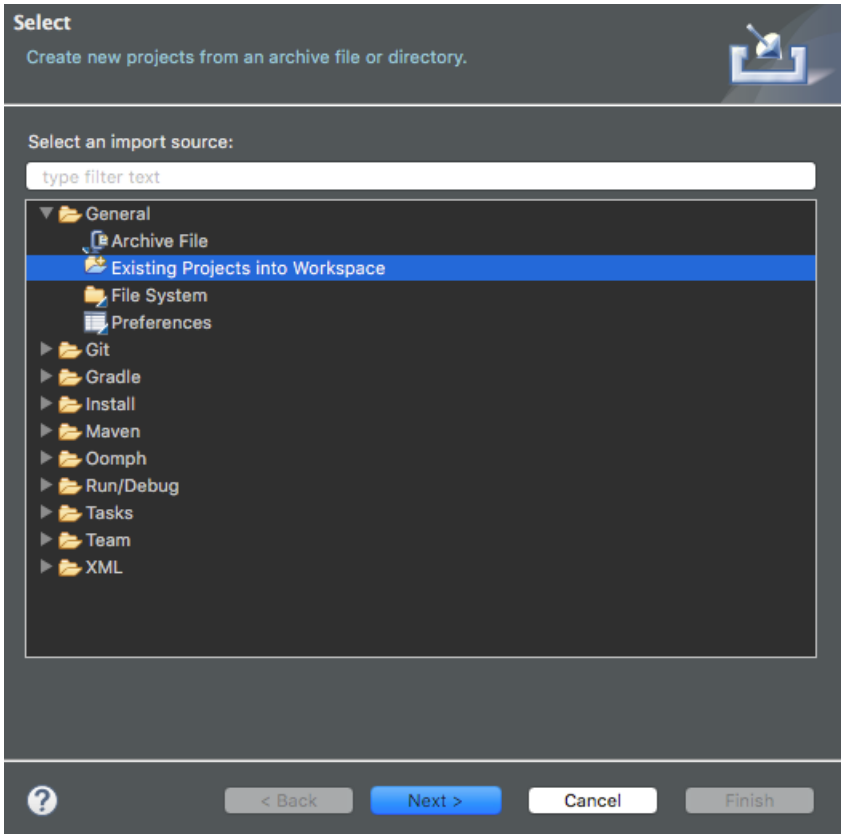
In this assignment we will build a BST structure, with each node representing a friend. Each friend node contains information about that player, including their username and level, along with attached data structures for the Pokemon they play (single linked-list) and ribbons (ArrayList). In accordance 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.

- Part A - you will setup the basic class structure, ensuring that the test suite is able to run without error.
- Part B - you will implement the basic structures needed by Account to hold multiple Ribbon and Pokemon objects.
- Part C - you will create your BST-based friend database.
- 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 1 the order outlined in the following sections. Many of the later-stage classes rely on the correct implementation of their dependencies.

Importing into eclipse

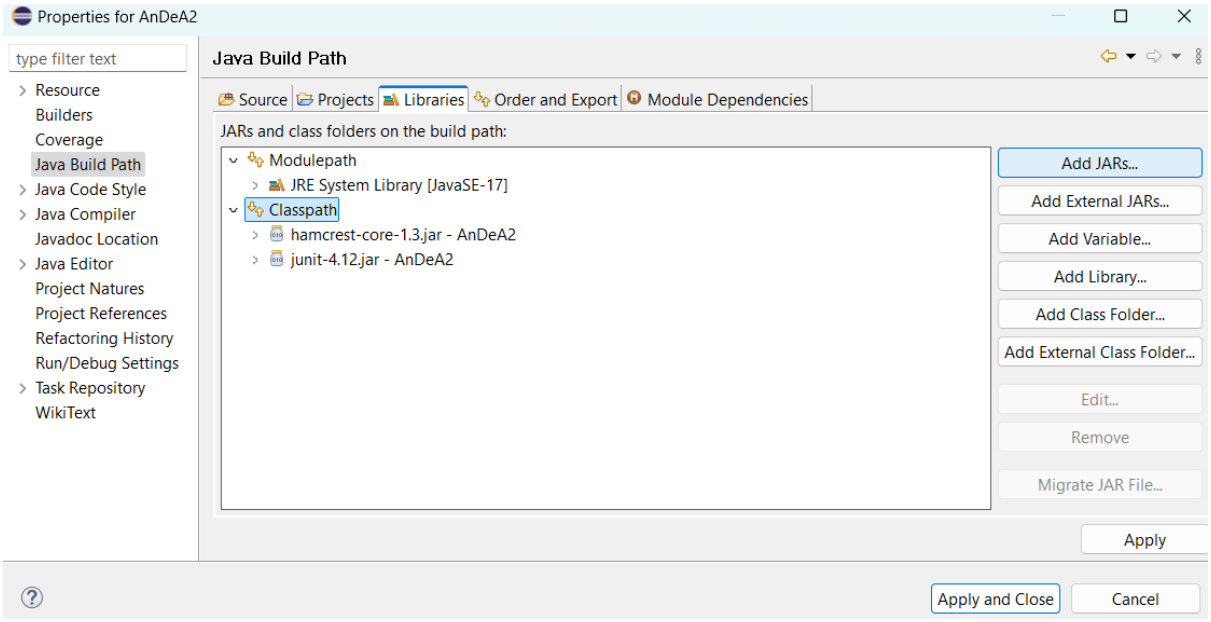
The Assignment has been provided as an eclipse project. You just need to import the project into an existing workspace.



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.

You might have to add the JAR files to the Classpath (see figure below). 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.



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 Pokemon

The Pokemon class represents one playable Pokemon and includes general information: the name, role (represented as an enum: speedster, all-rounder, attacker, defender, supporter) and number of obtainable ribbons. In addition, it holds a reference to another Pokemon object. This will be important in section 3 where you will make a single-linked list of Pokemon. The Pokemon class requires the following instance variables:

```
public enum Role {
SPEEDSTER, ALLROUNDER, ATTACKER, DEFENDER, SUPPORTER
}

private String name;
private Role role;
private Pokemon next;
```

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

Absol, Role: SPEEDSTER

You should also generate the appropriate accessor and mutator methods.

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

2.2 Ribbon

The Ribbon class represents an achievement level. This includes information about the name, rarity and the date it was obtained. The rarity can only be from a set of finite options available through enumerator variables. The Ribbon class requires the following instance variables:

```
public enum Rarity {
GREEN, BLUE, GOLD
}

private String name;
private Rarity rarity;
private Calendar obtained;
private Pokemon pokemon;
```

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

“Pursuited”, rarity: GOLD, obtained on: May 04, 2024

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. PokemonTester will assign marks as shown below:

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

2.3 Account

The Account class represents a 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 and the user level is preserved.
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.

```
private double calculateKey() {
int hash = Math.abs(username.hashCode());
// Calculate number of zeros we need
int length = (int)(Math.log10(hash) + 1);
4
// Make a divisor 10^length
double divisor = Math.pow(10, length);
// Return level.hash
return level + hash / divisor;
}
```

The Account class requires the following instance variables:

```
private String username;
private int level;
private double key;
private ArrayList<Ribbon> ribbons;
private PokemonList pokemon;
private Account left;
private Account right;
```

An ArrayList type was chosen for variable ribbons as you figured it would be easier to add a new ribbon to a list than a standard array, and you probably would mostly just traverse the list in order. A PokemonList object (see

section 3) was chosen as the structure for storing owned Pokemon 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: Harry

Ribbons:
"Snow Warning", rarity: BLUE, obtained on: Mar 26, 2024
"Sacred Sword", rarity: GREEN, obtained on: Mar 26, 2014

Pokemon Owned:
Ninetales, Role: ATTACKER
Absol, Role: SPEEDSTER
Aegislash, Role: ALLROUNDER
Tyranitar, Role: ALLROUNDER
Charizard, Role: ALLROUNDER

You should also generate the appropriate accessor and mutator methods.
AccountTester will assign marks as shown below:

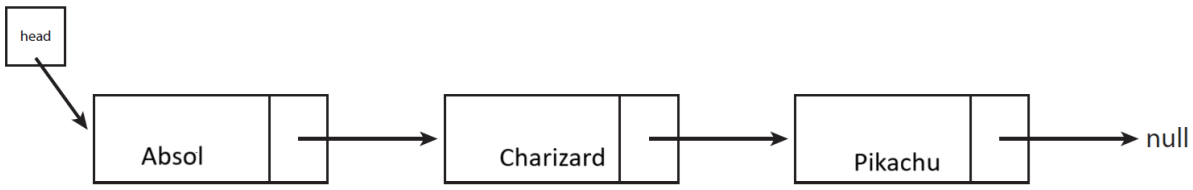
Test	Marks
testConstructor	2
toStringTest	3
Total:	**5**

3 Part B

In this section you will complete the PokemonList single linked-list for storing playable Pokemon objects.

3.1 PokemonList

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



The PokemonList class requires only one instance variable:
public Pokemon 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 addPokemon(Pokemon pokemon)

This method should add the provided pokemon to the end of your linked list. It should search for the first available slot, and appropriately set the previous pokemon's next variable. All pokemon must be unique, so you should check that the same pokemon has not already been added.

Note that the tests require that the provided Pokemon object is added, not a copy. If the PokemonList head variable is null, head should be updated to refer to the new pokemon. If the provided Pokemon object is null, an IllegalArgumentException should be thrown.

3.1.2 Pokemon getPokemon(String name)

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

3.1.3 void removePokemon(String name) | void removePokemon(Pokemon pokemon)

There are two overloaded removePokemon methods with one taking as an argument a String, the other a Pokemon. Both methods should search the linked list for the target pokemon 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:

Ninetales, Role: ATTACKER
Absol, Role: SPEEDSTER

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.

Note: To keep things simple, we have limited the Pokemon and Ribbon data structures to preset ones in the marker file. In an actual application you would not have these presets made at the start, and instead add them as they're created.

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 1 below.

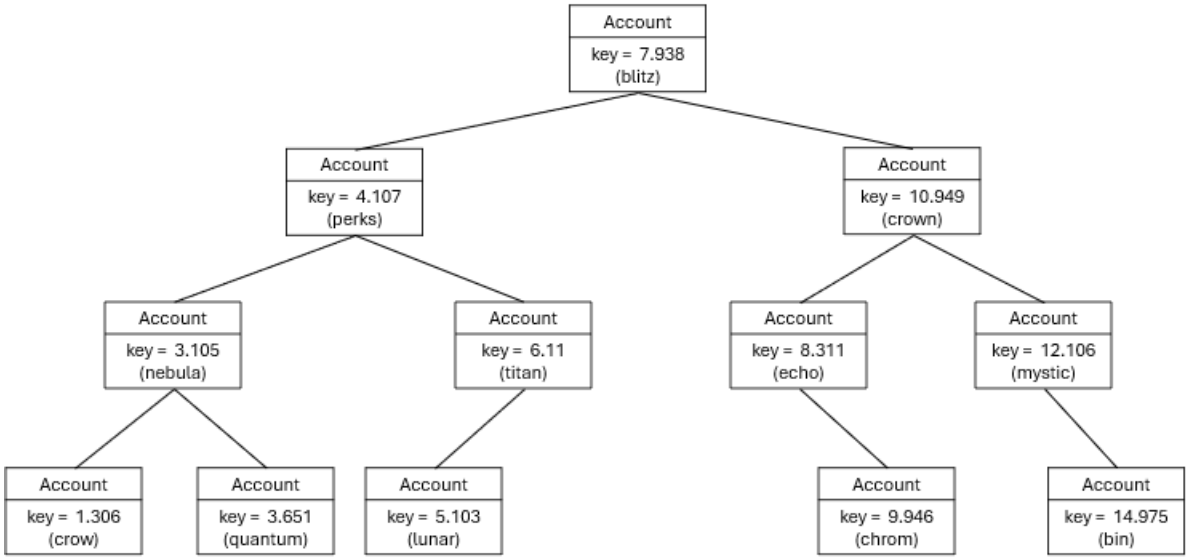


Figure 1 – initial BST structure

The BinaryTree class requires only one instance variable:

public Account 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(Account friend)
The beFriend method takes as an argument a new Account 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 Account object, not a copy. If the Account key is already present in the tree, this method should return false. If the Account argument is null, this method should throw an IllegalArgumentException. As an example, adding an Account with key 6 into the Figure above results in the tree shown in Figure 2 below.

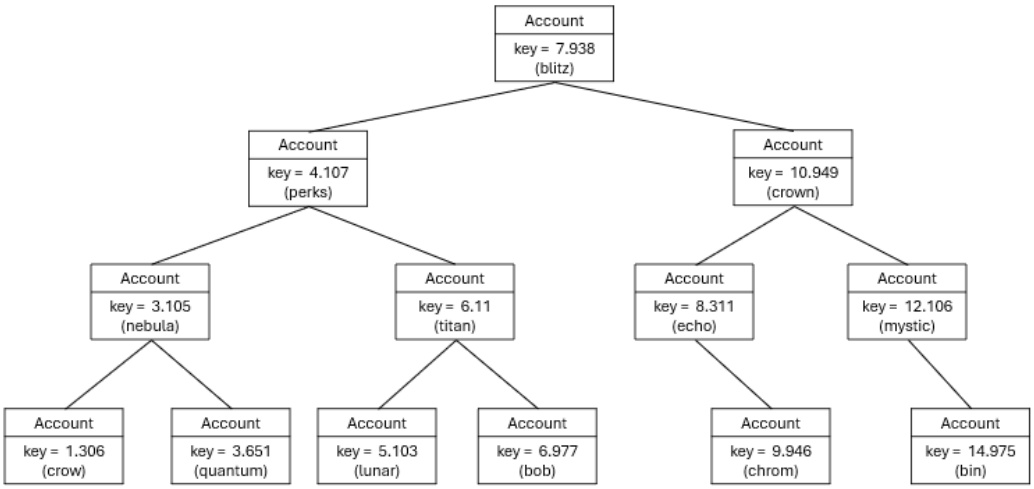


Figure 2 – BST with an account added with key value of 6

4.1.2 boolean removeFriend(Account friend)
The removeFriend method takes as an argument an Account 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 Account and updating the left, right and parent values as applicable. removeFriend 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 Account with key 4 from Figure 1 above. Results are shown in Figure 3 below.

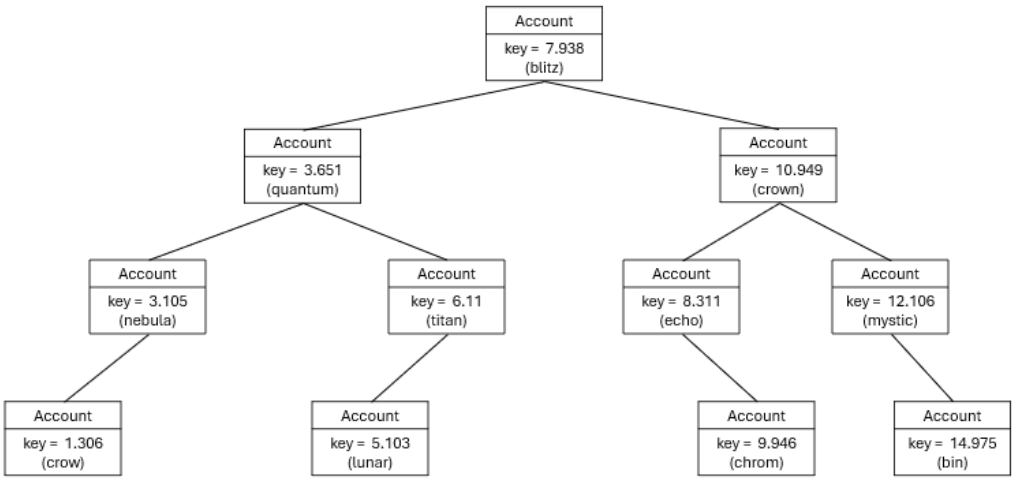


Figure 3 – initial BST with node removed containing key value of 4

4.1.3 int countBetterPlayers(Account reference)
The countBetterPlayers method takes as an argument an Account from which you should search for players with higher rank. This method should search from the reference account 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 Account argument is null, this method should throw an IllegalArgumentException. As an example, using User with key 7 from Figure 1 above, this method should return 5.

4.1.4 int countWorsePlayers(Account reference)
The countWorsePlayers method takes as an argument an Account from which you should search for players with lower rank. This method should search from the reference account 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 1, this method should return 6.

4.1.5 Account highestRibbonScore()
The highestRibbonScore method should search the tree and return the player who has the highest ribbon score. If there are no users with ribbons, this method should return null.
Ribbon score is calculated by adding the ribbons together with the following points:
GREEN = 1, BLUE = 2, GOLD = 3.

Hint: You may want to add a method in the Account class for determining ribbon points of an account.

4.1.6 void addPokemon(String username, Pokemon pokemon)
The addPokemon method takes two arguments, a String username and Pokemon pokemon.
You should search your database for a matching user and add the new pokemon to their PokemonList. You should also check that they do not already have that pokemon in their collection. If either argument is null, this method should throw an IllegalArgumentException.

4.1.7 void addRibbon(String username, Ribbon ribbon)
The addRibbon method takes two arguments, a String username and Ribbon ribbon. You should search your database for a matching user and add the new ribbon to their ribbons. You should also check that they do not already have the ribbon to be added and that they do not already have all available ribbons for the pokemon. 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 4 shows an invalid tree after a level-up and Figure 5 shows the correct alteration. If the username argument is null, this method should throw an IllegalArgumentException.

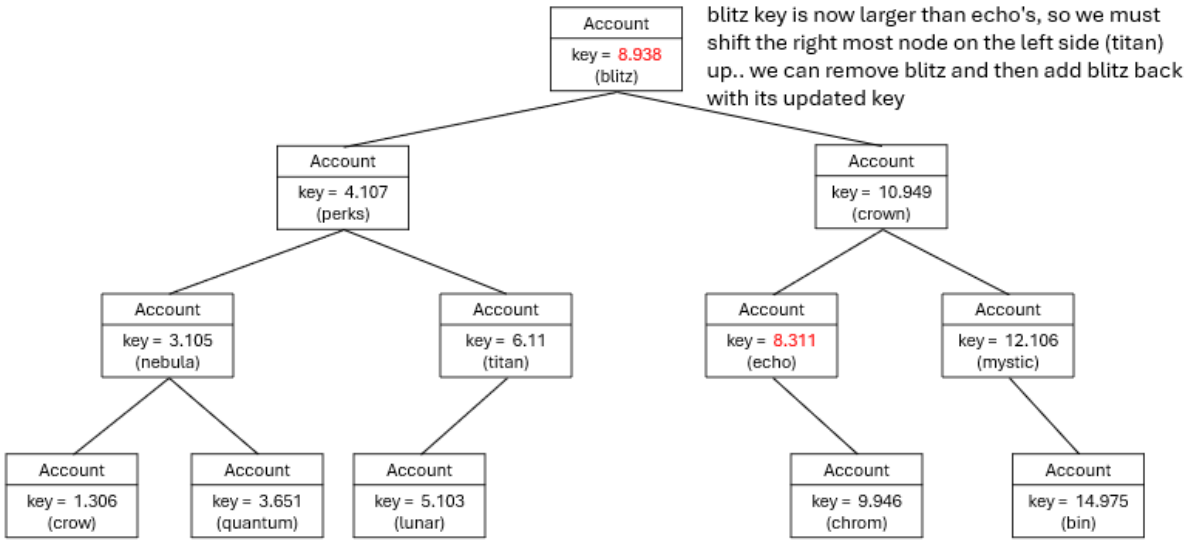


Figure 4 – Level up the root node causes issues with the BST structure, we need to adjust the tree

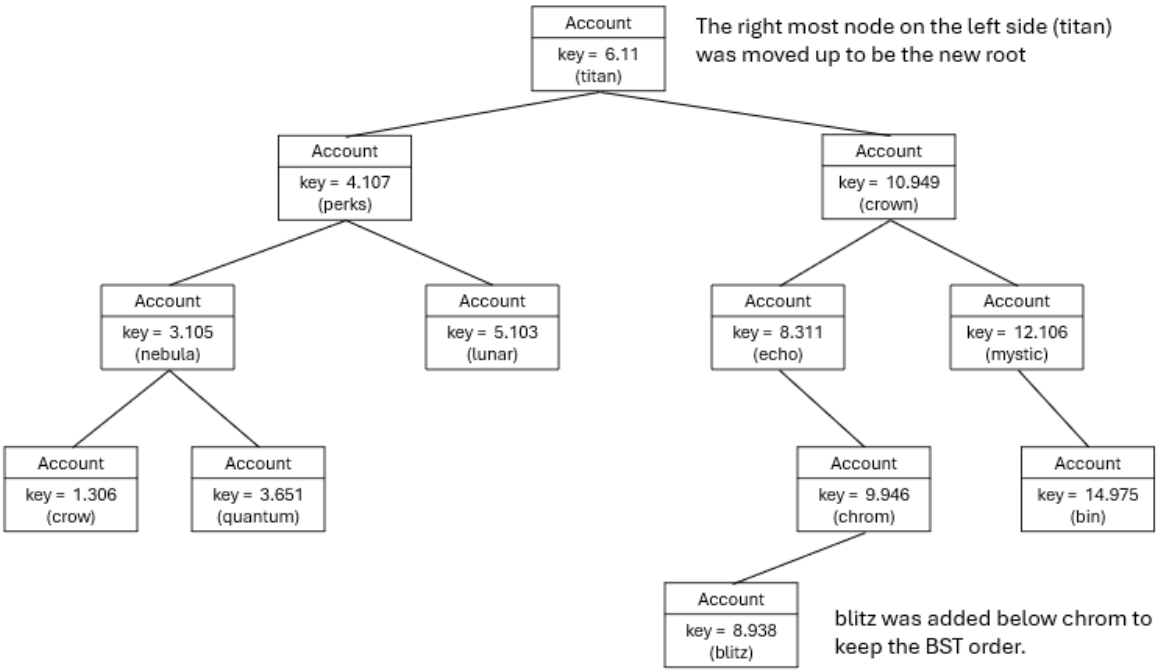


Figure 5 – Level up with correct rotations in place. [titan was shifted up and blitz was removed then re-added to its correct position in the BST]

Test	Marks
beFriendNullArg	1
beFriendDuplicate	1
beFriend	6
removeFriendNullArg	1
removeFriendNonExistent	1
removeFriend	7
countBetterPlayersNullArg	1
countBetterPlayersNonExistent	1
countBetterPlayers	4
countWorsePlayersNullArg	1
countWorsePlayersNonExistent	1
countWorsePlayers	4
highestRibbonScore	4
addPokemonNullArg	1
addPokemon	4
addRibbonNullArg	1
addRibbon	4
levelUpNullargs	1
levelUp	7
toStringTest	3
Total:	**54**

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(Account friend)

The addAVL methods takes as an argument an Account friend that you should add to the tree. AVL rules should apply, which means that if the tree becomes un-balanced, rotations should be performed to rectify. 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.

This link may be useful for visualising the AVL rotations required.

https://www.cs.usfca.edu/%7Egalles/visualization/AVLtree.html

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