Assignment 1:
AVL Trees and Tree Maps

COMP2014J: Data Structures and Algorithms 2
Lecturer: Dr. David Lillis (david.lillis@ucd.ie)
Weight: **15% of final grade**
Due Date: 23:59 Tuesday May 9th 2023 (Week 12)
Document Version: 1.0

Introduction

This assignment is intended to give you experience implementing AVL trees and using an AVL tree to implement a different type of data structure (a type of sorted Map known as a Tree Map). It is also a good exercise to gain experience about how generics, inheritance and object references work in Java.

Source code that you must start from has been posted to Brightspace in the file `Assignment1-Source.zip`. This also contains the Javadoc for the classes and interfaces provided (in the "`doc`" folder). Import this project into IntelliJ in the usual way.

You must use the interfaces and data structure implementations that are provided. Do not use any interfaces or implementations from the built-in Java Collections Framework. If you are in doubt, ask!

Tasks

The main tasks for this assignment are:

- Implement the key methods for an AVL Tree, according to the provided interfaces and base classes.
- Adapt your AVL Tree implementation to implement the key methods of a Tree Map, according to the provided `IMap` interface.
- Develop a strategy to test if your implementations are correct.

Implementation of AVL Tree Methods

The source code contains a partial implementation of an AVL Tree in a file called `AVLTree.java` in the `dsa.impl` package. All of your work in this section **must** be in this class and it **must** use the interfaces that are provided.

You must implement the following methods:

- `public boolean insert(T value)` – insert a value into the AVL tree. Returns `true` if the value was inserted (i.e. it was not already in the tree), or `false` if not.
- `public boolean remove(T value)` – remove a value from the AVL tree. Returns `true` if the value was removed, or `false` if not (i.e. the tree did not contain that value).
- `public boolean contains(T value)` – check to see if a value is contained in the AVL tree. Returns `true` if the value is in the tree, or `false` if not.
- `private void restructure(IPosition<T> x)` – trinode restructuring (the three nodes are `x`, its parent and its grandparent).

If you wish, you may create other methods that help you to complete the task (e.g. `rightRotate(IPosition<T> n)`, `leftRotate(IPosition<T> n)`, etc.).

Some hints and tips

- Remember your `AVLTree` extends several other classes, so you can use some of their helpful methods (e.g. `expandExternal(…)`).
- The `expandExternal(…)` method uses `newPosition(…)` to create all position objects, so all the positions in the tree will be `AVLPosition` instances.
- You can cast an `IPosition<T>` to an `AVLPosition` in the same way as you did in previous worksheets.
- Remember, every parent/child relationship works in **two directions**. **Every** time you change one of these references, you must change both.
- In the lectures we talk about attaching subtrees. **BUT** when we program this, we notice that the subtree structure does not change at all. We just need to put the root of the subtree in the right place.
- An `AVLPosition` object has a height attribute. You will need to efficiently calculate the height of the positions in the tree when the tree changes. Calculating the heights of all positions every time the tree changes will be at best O(n). An efficient implementation would be at worst O(h) when an `insert(…)` or `remove(…)` operation is called.
- The `TreePrinter` class has been provided, so you can print the contents of your tree and see what it contains.

Tree Map Implementation of IMap Methods

The source code contains a skeleton implementation of a map based on an AVL Tree in a file called `AVLTreeMap.java` in the `dsa.impl` package. All of your work in this section **must** be in this class and it **must** use the interfaces that are provided.

As you have learned in Data Structures and Algorithms 1, a Map is an ADT contains key/value pairs (called "entries"). Keys are used to uniquely identify values. By default, entries in a map have no particular order. The `IMap<K,V>` interface is provided (where `K` is the generic type of the keys and `V` is the generic type of the values) and contains the following methods:

- `public V put( K key, V value )` – add a new key/value pair to the map. If this key was already contained in the map, the old value associated with it is returned and the new value is stored in the map instead. Otherwise it returns `null`. (**Hint:** this is similar to the AVL Tree `insert(…)` method).
- `public V get( K key )` – get the value associated with the given key, or `null` if that key is not contained in the map (**Hint:** this is similar to the AVL Tree `contains(…)` method).
- `public V remove( K key)` – remove the entry with the given key from the map. Returns the value associated with that key if it was contained in the map, or `null` otherwise (**Hint:** this is similar to the AVL Tree `remove(…)` method).
- `public IIterator<IEntry<K,V>> entries()` – Return an iterator to iterate over all the entries contained in this map.
- `public IIterator<K> keys()` – Return an iterator to iterate over all the keys contained in this map.
- `public IIterator<V> values()` – Return an iterator to iterate over all the values contained in this map.
- `public int size()` – return the number of entries contained in this map.
- `public boolean isEmpty()` – return `true` if the map is empty, or `false` otherwise.

A Map that is implemented using any type of binary search tree (often called a "Tree Map") can be said to be a kind of sorted map, where all entries can be accessed according to the natural ordering of their keys. In your implementation, the three iterator methods (`entries()`, `keys()`, and `values()` must iterate in ascending key order).

For example, consider the following key/value entries stored in a map:
`{"zh", "Chinese"}, {"ga", "Irish"}, {"de", "German"}, {"en", "English"}`

When iterating the keys, the order would be (i.e. in alphabetical order):
- `"de", "en", "ga", "zh"`

When iterating the values, the order would be (i.e. in order of their keys):
- `"German", "English", "Irish", "Chinese"`

When iterating the entries, the order would be the same, i.e.:
- `{"de", "German"}, {"en", "English"}, {"ga", "Irish"}, {"zh", "Chinese"}`

Testing the Implementations

It is important to check whether your implementations are correct. A good way to do this is to use your implementation to perform some operations, and then check if the outcome is correct. This is best done using a program, rather than doing it manually every time.

An example is given in the `AVLTreeStructureTest` class in the `dsa.example` package. This performs some operations (only insert) on an AVL tree. To check if the final AVL tree is correct, it compares it with a Binary Search Tree that has the final expected shape (I worked this out manually).

Another example is shown in the `AVLTreeSpeedTest` class. This performs several operations on an AVL Tree and measures how quickly it runs. This is a good way to test the efficiency of your implementation.

Create some test classes for your implementations (called `Test1`, `Test2`, etc.). You can follow these examples or have your own ideas.

In your tests, you should test all the different types of restructuring that are possible (e.g. for an AVL tree, there are different types of trinode restructurings, and it will be done differently at the root compared to deeper in the tree). Similarly, for the Tree Map, you should test a number of different situations (e.g. inserting a new entry with a new key, replacing the value for an existing key, etc.).

Each test class must have a comment to explain the purpose of the test and what the outcome was.

Submission

- This is an **individual programming assignment**. Therefore, all code must be written by yourself. There is some advice below about avoiding plagiarism in programming assignments.

- All code should be well-formatted and well-commented to describe what it is trying to do.

- If you write code outside the `AVLTree.java`, `AVLTreeMap.java` and test files (`Test1.java`, `Test2.java`, etc.), it will not be noticed when grading. Write code only in these files.

- Submit a single .zip file to Brightspace.
    - This should include **only** the files you have written code in. Do not submit your entire IntelliJ project.

Assignment 1 Grading Rubric

This document shows the grading guidelines for Assignment 1 (implementation of AVL Tree and Tree Map). Below are the main criteria that will be applied for the major grades (A, B, C, etc.). Other aspects will also be taken into account to decide minor grades (i.e. the difference between B+, B, B-, etc.).

- Readability and organisation of code (including use of appropriate functions, variable names, helpful comments, etc.).
- Quality of solution (including code efficiency, minor bugs, etc.).

Passing Grades

D Grade
Good implementation of an AVL Tree, plus some basic testing (if you do not correctly implement the AVL Tree, it will not be possible to correctly implement the Tree Map).
A "good" implementation is one where all the key methods work correctly in the vast majority of cases (i.e. some occasional bugs will be tolerated, including some inefficiencies).

C Grade
Good implementation of an AVL Tree, plus comprehensive testing; OR
Excellent implementation of an AVL Tree, plus basic testing; OR
Good implementation of an AVL Tree, plus basic test testing, and a good attempt at the Tree Map.
"Comprehensive" testing should make sure that the different operations of the tree(s) are all tested (e.g. situations where different types of rotations are required, testing rotations at the root and deeper in the tree, situations with different numbers of rotations, etc.).
An “excellent” implementation is one where all methods are implemented correctly and efficiently.

B Grade
Excellent implementation of an AVL Tree, plus comprehensive testing; AND either
(Excellent implementation of Tree Map, plus basic testing OR
Good implementation of Tree Map, plus comprehensive testing)

A Grade
Excellent implementations of AVL Tree and Tree Map, with comprehensive testing of both.

Failing Grades

ABS/NM Grade
No submission received/no relevant work attempted.

G Grade
Code does not compile; OR
Little or no evidence of meaningful work attempted.

F Grade
Some evidence of work attempted, but few (if any) methods operate in the correct manner.

E Grade
AVL Tree has been attempted, but there are too many implementation errors for the implementation to be useful in practice.

Plagiarism in Programming Assignments

- This is an **individual assignment**, not a group assignment.
- This means that you must submit **your own work** only.
  If you submit somebody else's work and pretend that you wrote it, this is **plagiarism**.
- Plagiarism is a **very serious** academic offence.

Why should you not plagiarise?

- You don't learn anything!
- It is unfair to other students who work hard to write their own solutions.
- It's cheating! There are **very serious punishments** for students who plagiarise. The UCD policy on plagiarism can be found online[1].
  - A student found to have plagiarised can be exclude from their programme and not allowed to graduate.

Asking for Help

If you find things difficult, help is available.

- TAs are available.
- Your lecturer is available in the lab.
- You can post questions in the Brightspace discussion forum.
- You can email the lecturer (`david.lillis@ucd.ie`).
- You can get help from your classmates.
  - Getting help to understand something is not the same as copying a solution!

The best way to get useful answers is to ask good questions.

Don't just send a photo of your computer screen and ask "Why does this not work?" (**N.B.** images are not a good way to send code).

Do:

- Send/post your Java file(s) as an attachment. We can't run code that's in a photograph to test it out!
- Say what error message you got when you tried to run the code (if any).
- Say what the code did that you did not expect.
- Say what the code did not do that you did expect.

[1] https://www.ucd.ie/t4cms/UCD%20Plagiarism%20Policy%20and%20Procedures.pdf

How to avoid plagiarism: Helping without copying.

If you are trying to help a classmate with a programming assignment, there are two golden rules:

Never, ever give your code to somebody else.

- You don't know what they will do with it or who they will give it to.
- If somebody else submits code that is the same as yours, **you will be in trouble too.**

Don't touch their keyboard

Don't type solutions for them! It will end up looking a lot like your code. Also, they don't learn anything.

Here are some other ways you can help a friend with an assignment, without risking plagiarism:

- If their code doesn't work, it's OK to explain what is wrong with it.
- If they don't understand a concept, draw a diagram to explain.
- Tell them about useful methods that I have provided that can help achieve their goals.
- Describe an algorithm that will help.
    - Describe it in **words** or **diagrams**, not in code!
    - E.g. "You could try saving the node's right child as a variable. Then you could use a loop to keep getting that node's left child until you reach the bottom of the tree".

Problems?

If you notice any problems or errors either in this document or in the source code provided, please let me know as soon as possible via email: david.lillis@ucd.ie

Document History

v1.0, 2023/04/10, Initial Version