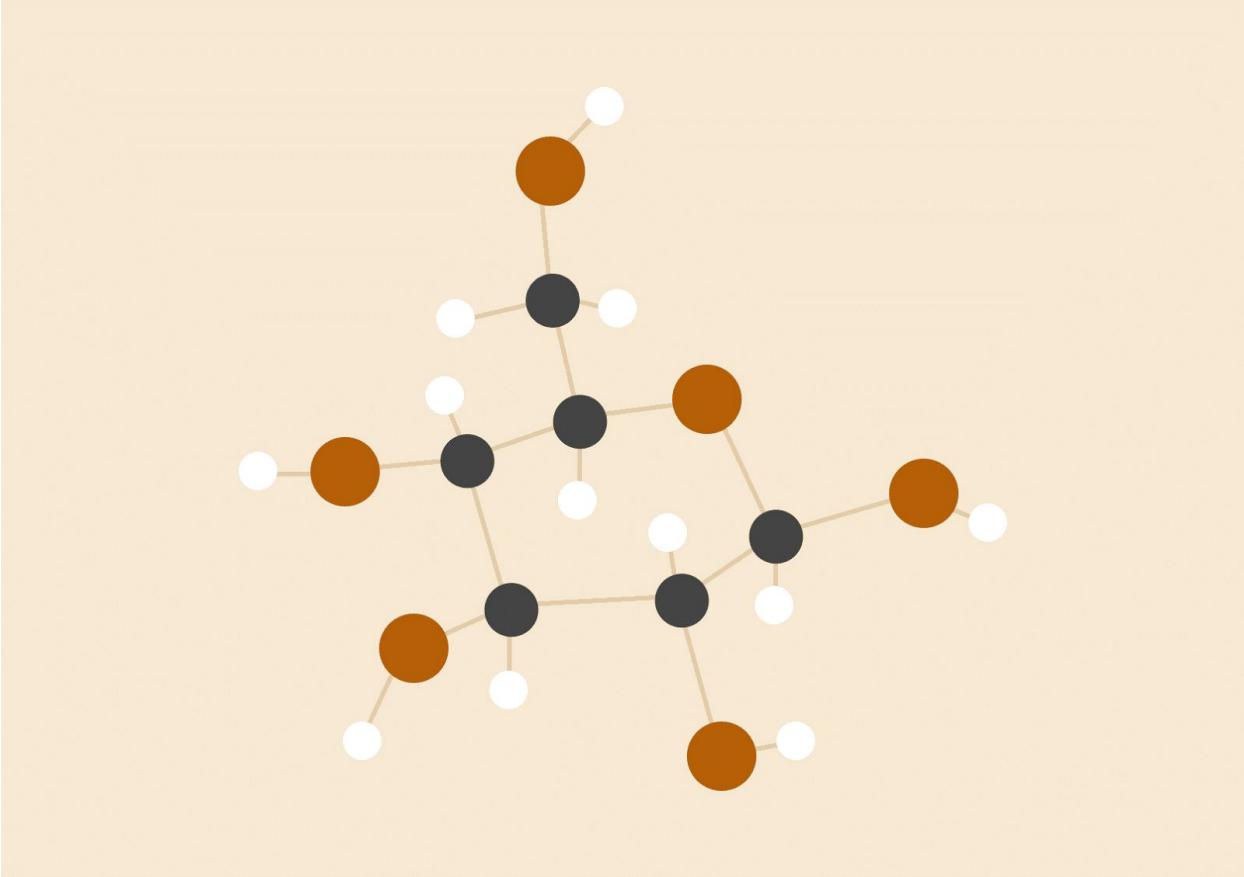
# Computer Science

*Report for Assignment 2*



**Brandon Gower-Winter**

24.04.2017

CSC2001F

## Problem

Assignment 2 tasks one to test the efficiency between two different data structures( Binary Search Tree(BST) and a AVL Tree) and determine which one of the two aforementioned data structures is more efficient at inserting, searching and removing data.

One is also tasked with understanding a AVL tree and the various algorithms associated with it; Namely the balancing algorithm that is unique to an AVL tree and the rotation algorithms associated with that balancing algorithm.

A BST is a data structure that conceptualises the idea of a Binary Search Algorithm into a Binary Tree data structure (A Tree data structure in which each node can have at most two child nodes).  A BST is unique in the way it stores data. Values that are smaller go to left of the tree and values that are greater go to the right. This happens recursively until a null node is found and the value that was to be added gets stored there. This allows for a BST to be more efficient when searching for data with an average case of log(n) for searches.

An AVL tree is a balanced Binary Search Tree. The AVL Tree balances the data stored in it to ensure that even in the worst case the search algorithm is of O(log(n)) efficiency. The tree does this by ensuring that the height of  any node's two subtrees doesn't differ by more than one. The balancing is done by applying the appropriate rotations to the node structure. As a result this means that inserts and deletions are longer than in a regular BST. This is a deliberate sacrifice to ensure efficient searches and is the main reason to use a AVL Tree instead of a BST in situations that don't require quick inserts or deletions but require fast searches.

## Application Design

There were two tasks to be done in the given assignment. 1. Write an application to search an AVL Tree data structure for names from a query file. 2. Compare the AVL Tree data structure to the BST data structure by recording the time time it takes for them to insert, search and delete data.

In order to make the application as modular and simple as possible there were specific rules I had set in place for the application.

1. Each Main Class(SearchIt, SearchAVL) will only contain a main function and will be used as application files that will make use of the other class files I created.
2. The data would be stored in a person object as to make the code neater and allow for ease of use.(I could create specific functions in the person class to handle how one object is compared to another and I could create a toString() function to allow all the data within the object to be displayed in one function call).
3. The generic BST and AVL Tree that I created must ensure that the data type it is taking in implements the Comparable interface. This is because the compareTo() function is used to determine where to store new values and find old values.

The person class acts as a data model and is used purely to store the data given from the data file. It implements the the comparable interface which allows for custom handling on how to objects are compared with one another. Two person objects are only compared by name as is the requirement.

The Binary Search Tree class is generic and ensures that any data type that it will handle implements the Comparable interface (As mentioned above). All functions within the tree are recursive for code tidiness. The BST can insert, search, and delete data within the tree. The tree doesn't use the concept of left and right node it uses less node and great node as to get rid of the concept that the tree is orientation based. Less node is used to store a value that is less than and greater node is used to store a value that is greater than.

The AVL Tree class is also generic and inherits from the BST class which means it inherits all of its functionality and ensures that only an object that implements the Comparable interface can use it. The AVL tree makes use of the same code as the BST for insert, delete and search functions the only difference is that it uses a balance function after inserting or deleting a node. This balance function will check the heights of the subtrees and determine if a balance is needed (If the height difference is greater than 1). The balancing function makes use of two rotate functions (rotateLeft and rotateRight) to move the nodes in the tree around.

SearchIt is the application class used to test the BST and was used in assignment 1.

SearchAVL is the application class used to test the AVL tree and it makes use of a scanner to parse in the data from a datafile line by line and add it into the AVL tree. Then another scanner is used to parse in query names and queries are made to the tree and an appropriate result is returned. This class as will as SearchIt make use of the java.time package in order to record the time taken for a function(eg: add()) to run. The SearchAVL class has the ability to add, delete and search for data in the AVL tree and acts as the hub for all of the testing.

## SearchAVLOutput

Mayert Cathy  791-772-8120 x42168  90125 Raven Circle #864, Downey

Gower-Winter Brandon was not found.

Hickle Leone  018-594-2935 x716  17386 Stephanie Parks, Palm Springs

West Ramon  1-702-852-5634  50773 Schinner Extensions, Zanesville

Mikazuki Augus was not found.

Setsuna F. Seiei was not found.

Langosh Matt  (682)669-6865 x500  73754 Leffler Squares, Atwater

Ondricka Luz  (522)447-5098 x1929  68014 Jermain Street, Springfield

Labadie Leanna  (896)176-7008  37773 Durgan Parkways Suite 558, San Dimas

Lockon Stratos was not found.

Jacobs Wallace  (174)976-6745 x0539  06395 Cormier Crest Suite 404, West Memphis

Orga Itsuka was not found.

Ketchum Ash was not found.

Haag Abraham  (933)453-8588 x7220  04266 Missouri Junction, Burlingame

Ryan Alicia  950-938-7050 x27619  76980 Side, Auburn

Alleluyah Haptism was not found.

Daugherty Elijah  (549)540-0126 x715  26255 Marvin Way #268, Decatur

Abernathy Houston  757.248.9579 x9418  24902 O'Conner Creek, Homer

Lehner Alberta  (460)301-1274 x351  67100 Schumm Pines, Barrow

Fahey Lane  1-220-139-2838 x649  04923 Flatley Island Suite 476, San Clemente

## Experimental Design

Insertion, Deletion and the Searching of data needed to be tested at various sizes of data N.

I decided to test at the powers of 10 for each function in both of the trees so N = {10,100,1 000,10 000}. This would allow for extreme testing(10 at low end and 10 000 at high end) for an average sized data structure. Intermediary values of N are also used to get an idea of how efficiently a function handles a rapid increase in the size of the data present. It will also show how quickly a function's process time can degenerate.

All three functions are being tested separately to allow for each function to be analysed as it's own entity.

Time recorded means different things for each function:

- For Insertion the time recorded is the average time taken for the data structure to add N elements (Including balancing for the AVL Tree)
- For Deletion the time recorded is the average time taken for the data structure to delete 20 elements(including balancing for the AVL Tree) in a data structure of N elements that have been predetermined from a query file.
- For Searching the time recorded is the average time taken for the data structure to return 20 elements in a data structure of N elements that match 20 queries sent to it from a query file.

## Results

**Insertion:**

| N | Time BST(ms) | Time AVL Tree(ms) |
|---|---|---|
| 10 | 11 | 11 |
| 100 | 43 | 45 |
| 1 000 | 242 | 349 |
| 10 000 | 713 | 14 075 |

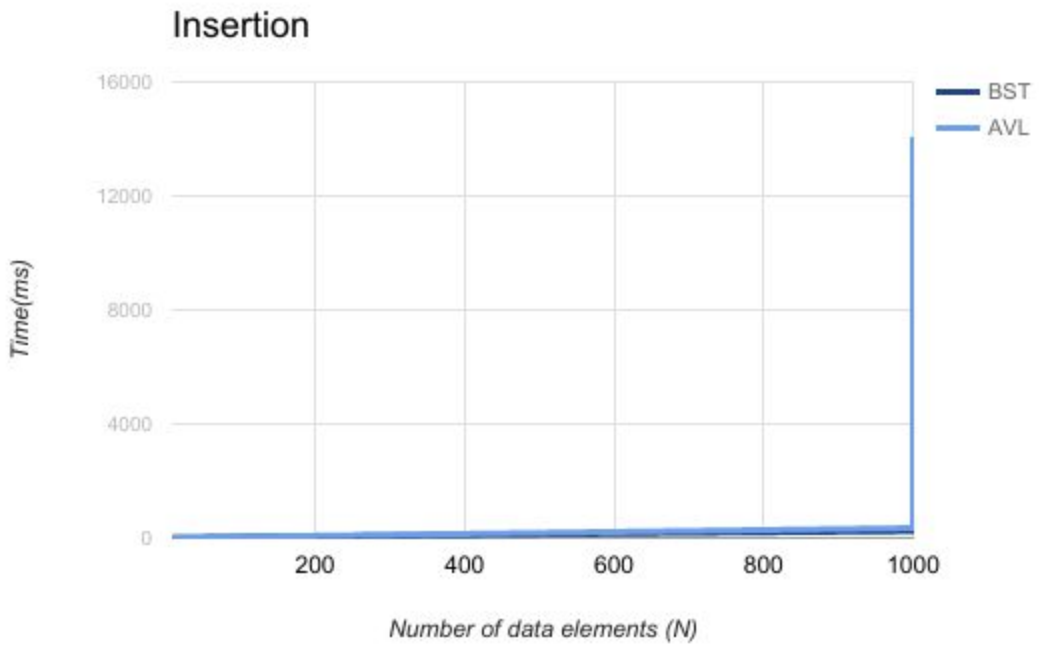
**Searching:**

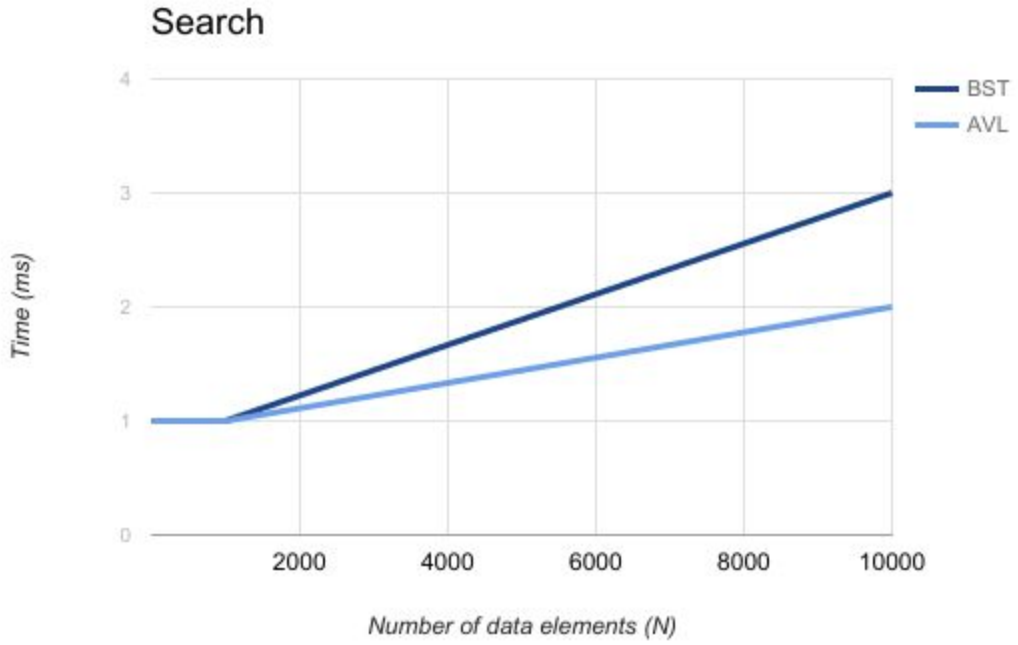
| N | Time BST(ms) | Time AVL Tree(ms) |
|---|---|---|
| 10 | 1 | 1 |
| 100 | 1 | 1 |
| 1 000 | 1 | 1 |
| 10 000 | 3 | 2 |

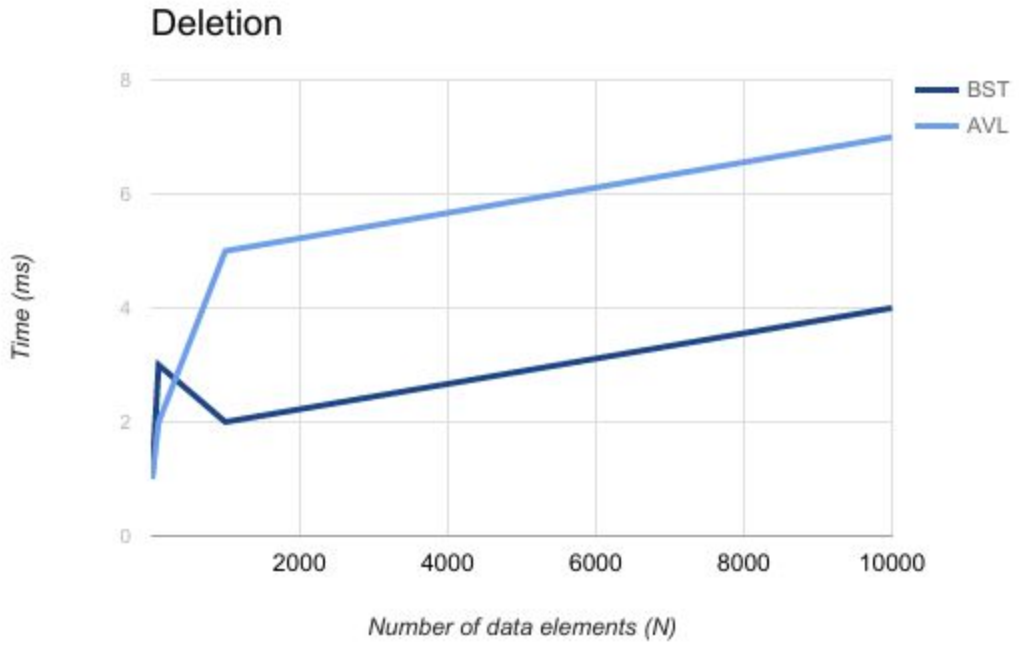
**Deletion:**

| N | Time BST(ms) | Time AVL Tree(ms) |
|---|---|---|
| 10 | 1 | 1 |
| 100 | 3 | 2 |
| 1 000 | 2 | 5 |
| 10 000 | 4 | 7 |

## Graph:


Insertion
16000
12000
8000
4000
0
Time(ms)
200
400
600
800
1000
Number of data elements (N)
BST
AVL


Search
4
3
2
1
0
Time (ms)
2000
4000
6000
8000
10000
Number of data elements (N)
BST
AVL




## Conclusion

After analysis of the data it can be concluded that the answer is a lot more complicated than saying one of the data structures is more efficient because both of them have positives and negatives. They should be compared at each individual function they were tested for.

Insertion:

The BST is quite clearly more efficient at inserting data especially at larger amounts of data. This is to be expected though because as mentioned earlier the AVL tree must spend time balancing the tree which affects its performance greatly at Insertion.

Deletion:

Deletion had a similar situation but not as drastic. The BST is more efficent for larger values of N and this to be epected because similiarly to Insertion balancing must be done after these processes  which will slow down the overall efficiency of the deletion within the AVL tree.

Searching:

Searching is where the AVL tree shines with slight efficiency over the BST and this because of the balancing done before that will pay off more and more the larger N gets and the more searches are called in the AVL tree.

Overall the BST is preferable in a situation where one wants to insert and delete data quickly and primary memory is a concern while and AVL tree can and should be used when search times are important and memory is not a scarce resource.

## GIT Log

commit 120cf1892b7c7656edc94bcd88632c633108375b

Author: BrandonGower-Winter <brandongowerwinter@gmail.com>

Date: Mon Apr 24 03:00:05 2017 +0200

Update query file

commit 46cac4ad5d7417a93ae9b2bff9e81f228ff66739

Author: BrandonGower-Winter <brandongowerwinter@gmail.com>

Date: Mon Apr 24 01:07:58 2017 +0200

Finished Testing

commit cb5520ab982b22a821e40a2f9eb86a4295b76e91

Author: BrandonGower-Winter <brandongowerwinter@gmail.com>

Date: Sun Apr 23 23:52:17 2017 +0200

Added javadocs

commit 15d6fd5a7e9ff63db6701dff2dc4a745e0ee137a

Author: BrandonGower-Winter <brandongowerwinter@gmail.com>

Date: Sun Apr 23 22:52:04 2017 +0200

Tasks complete

commit 5190037b7d5650c61bf4e97dab2a4bf46e99287c

Author: BrandonGower-Winter <brandongowerwinter@gmail.com>

Date:   Sun Apr 23 17:15:19 2017 +0200

    Finished Generic AVL tree implementation

commit 9266e69f7d2346191e58f7e2158771d9b2a286b5

Author: BrandonGower-Winter <brandongowerwinter@gmail.com>

Date:   Sat Apr 22 22:34:22 2017 +0200

    Moved data structures to package

commit 3ae1064c5439a0e3627afdecd02c1c3118120214

Author: BrandonGower-Winter <brandongowerwinter@gmail.com>

Date:   Fri Apr 14 12:16:29 2017 +0200

    Updated Documents and copied files needed from Assignment 1 to Assignment 2

commit 59b4b6f163730117bd03411e30dfdb7575d7b95c

Author: BrandonGower-Winter <brandongowerwinter@gmail.com>

Date:   Mon Apr 10 09:38:19 2017 +0200

    Managed Folders and created assignment 2

## JacocoReport

If I could figure out how to do it. It would go here. I tried though.

## JUnit Testing

Testing works. The output is deliberately supposed to return only true if all cases are true. It makes the terminal less cluttered and much easier to spot actual mistakes. Only shows details for cases that did not turn out the expected way.