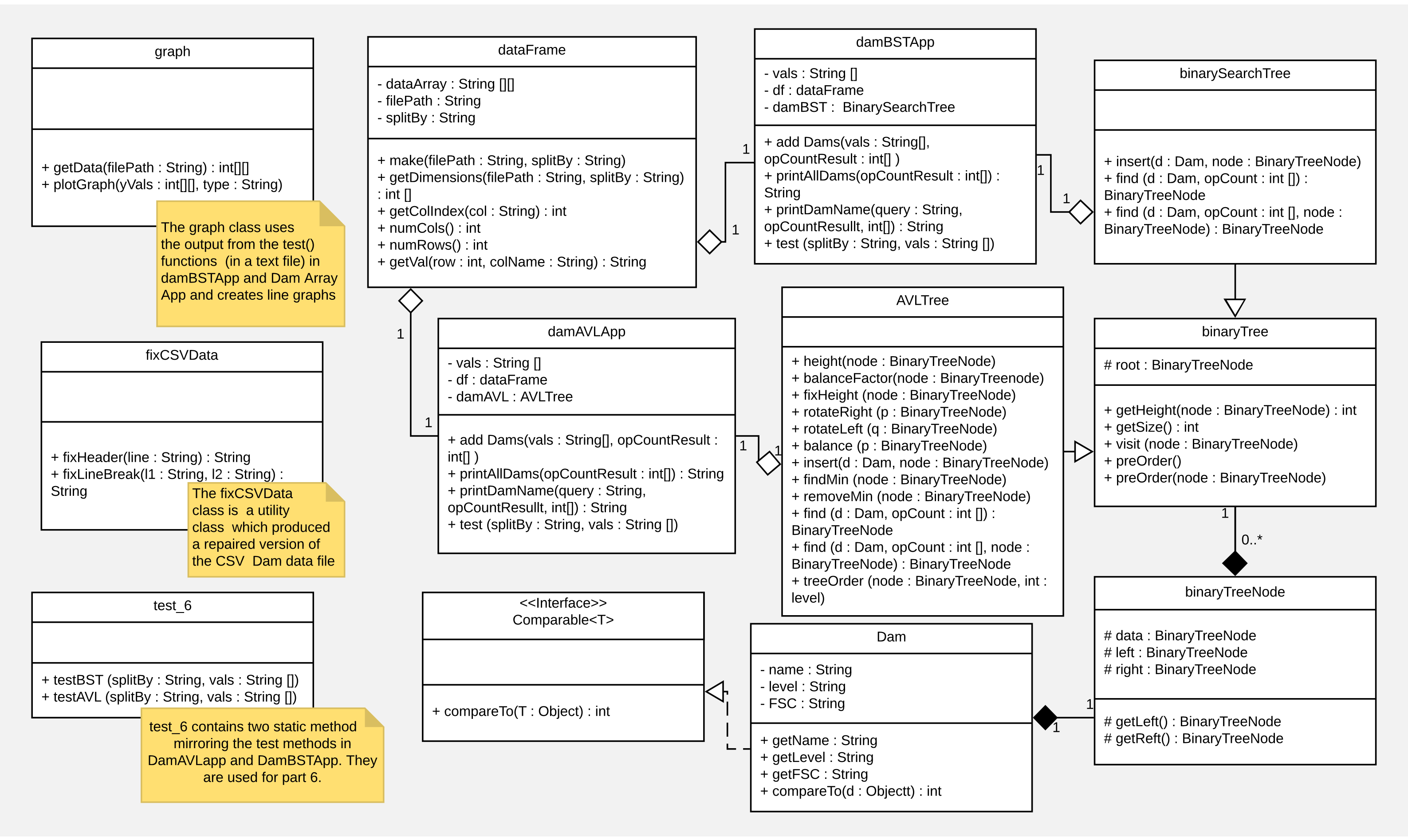
CSS2001F
Assignment 2: AVL Trees
DPLJER001
Report: Experiment Design and Results

This report details the design and results of my work on Assignment 2. The final objective of the report is to demonstrate the difference in time complexity between the AVL Tree and Binary Tree data structure search and insert algorithms.

a. Object Orientated Design (UML)


graph
+ getData(filePath : String) : int[][]
+ plotGraph(yVals : int[][], type : String)
The graph class uses the output from the test() functions (in a text file) in damBSTApp and Dam Array App and creates line graphs
dataFrame
- dataArray : String [][]
- filePath : String
- splitBy : String
+ make(filePath : String, splitBy : String)
+ getDimensions(filePath : String, splitBy : String) : int []
+ getColIndex(col : String) : int
+ numCols() : int
+ numRows() : int
+ getVal(row : int, colName : String) : String
damBSTApp
- vals : String []
- df : dataFrame
- damBST : BinarySearchTree
+ add Dams(vals : String[], opCountResult : int[])
+ printAllDams(opCountResult : int[]) : String
+ printDamName(query : String, opCountResullt, int[]) : String
+ test (splitBy : String, vals : String [])
binarySearchTree
+ insert(d : Dam, node : BinaryTreeNode)
+ find (d : Dam, opCount : int []) : BinaryTreeNode
+ find (d : Dam, opCount : int [], node : BinaryTreeNode) : BinaryTreeNode
fixCSVData
+ fixHeader(line : String) : String
+ fixLineBreak(l1 : String, l2 : String) : String
The fixCSVData class is a utility class which produced a repaired version of the CSV Dam data file
damAVLApp
- vals : String []
- df : dataFrame
- damAVL : AVLTree
+ add Dams(vals : String[], opCountResult : int[])
+ printAllDams(opCountResult : int[]) : String
+ printDamName(query : String, opCountResullt, int[]) : String
+ test (splitBy : String, vals : String [])
AVLTree
+ height(node : BinaryTreeNode)
+ balanceFactor(node : BinaryTreenode)
+ fixHeight (node : BinaryTreeNode)
+ rotateRight (p : BinaryTreeNode)
+ rotateLeft (q : BinaryTreeNode)
+ balance (p : BinaryTreeNode)
+ insert(d : Dam, node : BinaryTreeNode)
+ findMin (node : BinaryTreeNode)
+ removeMin (node : BinaryTreeNode)
+ find (d : Dam, opCount : int []) : BinaryTreeNode
+ find (d : Dam, opCount : int [], node : BinaryTreeNode) : BinaryTreeNode
+ treeOrder (node : BinaryTreeNode, int : level)
binaryTree
# root : BinaryTreeNode
+ getHeight(node : BinaryTreeNode) : int
+ getSize() : int
+ visit (node : BinaryTreeNode)
+ preOrder()
+ preOrder(node : BinaryTreeNode)
test_6
+ testBST (splitBy : String, vals : String [])
+ testAVL (splitBy : String, vals : String [])
test_6 contains two static method mirroring the test methods in DamAVLapp and DamBSTApp. They are used for part 6.
<<Interface>>
Comparable<T>
+ compareTo(T : Object) : int
Dam
- name : String
- level : String
- FSC : String
+ getName : String
+ getLevel : String
+ getFSC : String
+ compareTo(d : Objectt) : int
binaryTreeNode
# data : BinaryTreeNode
# left : BinaryTreeNode
# right : BinaryTreeNode
# getLeft() : BinaryTreeNode
# getReft() : BinaryTreeNode
1
0..*

b. Experiment: Goal and Execution

The primary goal of the assignment was to compare the time complexity between the algorithms for an AVL Tree, and a binary search tree (BST). In this assignment the number of comparison operations, used during both the insert and search methods are used as proxies for the overall time complexity of the algorithms. The high level objective was to create a set of classes to execute the experiment using a set of 211 entries of Dam information contained in a CSV file as data, comparing the variation of performance between the BST and the AVL Tree - which has the advantage of balancing during the insertion process - which is expected to allow for greater search efficiency.
My method was to aim for modularity, and to isolate as many tasks as possible as well as to create a robust framework that could potentially be used to process data in any CSV file, not just the one specific to this test. First, I created a *dataFrame* class in which I could read a CSV file, measure the dimensions and store the data in a blank, 2D String array. The class has utilities which allow me to search for single items using column names, as well as returning the dimensions. Both the *damAVLApp* and *damBSTApp* have *dataFrame* objects which are used to create *Dam* objects by each class, and store them in their respective data structures. When each data structure is created the opCounts (comparisons of Dam names) during the insertion of each dam object are recorded, the

worst, best and average case are all recorded and reported when a search is done, or when all dam info is printed. The *Dam* class implements the Comparable interface (used by both the BST and AVL) , and stores values for name, dam level and FSC. I used the toString() method inside the *Dam* class for easy printing of information during search. The *printDamName*() function is implemented in both the d*amAVLapp* and d*amBSTapp.* Both the *damBSTApp and damAVLapp* search through the nodes (containing *Dam* objects) and compare the query dam name - for which a temporary *Dam* object is created - to the names of each node and moves down the branches according to lexicographic ordering (string.compareTo(otherString)) until the queried Dam is found, or the end of a branch is reached. In both applications the opCount starts at zero when each search function is initiated, and is incremented by 1 each time a comparison is made - I used an int array to take advantage of pass-by-reference (as with the opCounts for insertion). I have written bash scripts to run both search and *printAllDam*() functions with test values, redirecting the output to text files (using echo (function()) > testFile.txt).

For the main experiment - part 5 - plotting the comparison between opCounts when inserting new nodes and searching varied subsets of data - my method was as follows:

- Create subsets of the original data file (from a set of 1, to 211), using a bash script (a loop and the unix head command, writing results to 211 new CSV files, test_1.txt… test_211.txt)
- Create *test*() functions in both the AVL and binary search tree applications to loop through each subset of the data, create a temporary *damAVLApp* or *damBSTApp* and create a *dataFrame* in the object containing the subset of data.
- The opCount statistics for insertion are recorded for each data subset and added to a string-builder, in order to be written to file.
- Then search once for each name in the given subset, storing the operation count for each search in an array.
- For a given subset, the array of operation counts will have a max and min value, which are recorded as the worst and best cases for that dataset. The average is also calculated.
- Finally, resulting curves are plotted (best, average and worst) on a two graphs, one for the AVL search algorithm, one for the binary search tree search algorithm for both search and insertion opCount statistics using the *graph* class(see part d.)

c. Test Results: Part 2 and 4

part 2: results from the Binary Search Tree Application

```
Test 1 (part 1 and 2)

Searching BST for: 3 known dam names

Result:
name: Lindleyspoort Dam
dam level: 2.7
FSC: 14.208
the size of the dataset was: 211
The number of insertion comparisons (best,
average and worst) were: 0, 7, 12
The total number of search comparisons was: 3

Result:
name: Koppies Dam
dam level: 39
FSC: 42.311
the size of the dataset was: 211
The number of insertion comparisons (best,
average and worst) were: 0, 7, 12
The total number of search comparisons was: 6
```

```
Result:
name: Woodstock Dam
dam level: 77.099999999999994
FSC: 373.25
the size of the dataset was: 211
The number of insertion comparisons (best,
average and worst) were: 0, 7, 12
The total number of search comparisons was: 9


Test 2 (part 1 and 2)
Searching BST for: 1 unknown dam name
Result:
Dam not found
query: Nothing
the size of the dataset was: 211
The number of insertion comparisons (best,
average and worst) were: 0, 7, 12
The total number of search comparisons was: 8
```

Test 3 (part 4)

```
First 10 entries:

Printing all dam entries from BST
(Pre_order Traversal)

Result:

name: Ngotwane Dam
level: 4.6
FSC: 19.033000000000001

name: Hartbeespoort Dam
level: 96.5
FSC: 186.44

name: Bon Accord Dam
level: 103
FSC: 4.381

name: Albasini Dam
level: 69.2
FSC: 28.199000000000002

name: Blyderivierpoort Dam
level: 50
FSC: 54.369
```

Test 3 (part 4)

```
name: Allemanskraal Dam
level: 13.5
FSC: 174.52

name: Albert Falls Dam
level: 36.4
FSC: 288.14

name: Bloemhof Dam
level: 15.5
FSC: 1240.24

name: Armenia Dam
level: 10
FSC: 12.957000000000001

name: Berg River Dam
level: 31.1
FSC: 127.05

…
```

Test 3 (part 4)(continued)

```
Last 10 entries:

…

name: Waterdown Dam
level: 90.1
FSC: 37.441000000000003

name: Wemmershoek Dam
level: 52.3
FSC: 58.71

name: Westoe Dam
level: 48.3
FSC: 60.094999999999999

name: Wolwedans Dam
level: 95.4
FSC: 24.626000000000001

name: Witklip Dam
level: 39.299999999999997
FSC: 12.519

…
```

Test 3 (part 4)

```
name: Xonxa Dam
level: 100
FSC: 115.86

name: Wriggleswade Dam
level: 98.7
FSC: 91.471000000000004

name: Woodstock Dam
level: 77.099999999999994
FSC: 373.25

name: Xilinxa Dam
level: 27.2
FSC: 13.823

name: Zaaihoek Dam
level: 58.5
FSC: 184.63

Printed all dam entries from binary
search tree.

the size of the dataset was: 211
The number of insertion comparisons
(best, average and worst) were: 0, 7, 12
```

Part 4: results from the AVL Tree Application

```
Test 1 (part 3 and 4)

Searching AVL Tree for: 3 known dam
names

Result:
name: Lindleyspoort Dam
dam level: 2.7
FSC: 14.208
the size of the dataset was: 211
The number of insertion comparisons
(best, average and worst) were: 0, 7, 10
The total number of search comparisons
was: 1

Result:
name: Koppies Dam
dam level: 39
FSC: 42.311
the size of the dataset was: 211
The number of insertion comparisons
(best, average and worst) were: 0, 7, 10
The total number of search comparisons
was: 6
```

```
Result:
name: Woodstock Dam
dam level: 77.099999999999994
FSC: 373.25
the size of the dataset was: 211
The number of insertion comparisons
(best, average and worst) were: 0, 7, 10
The total number of search comparisons
was: 7

Test 2 (part 3 and 4)

Searching AVL Tree for: 1 unknown dam
name

Result:
Dam not found
query: Nothing
the size of the dataset was: 211
The number of insertion comparisons
(best, average and worst) were: 0, 7, 10
The total number of search comparisons
was: 9
```

d. Test Results: Part 5

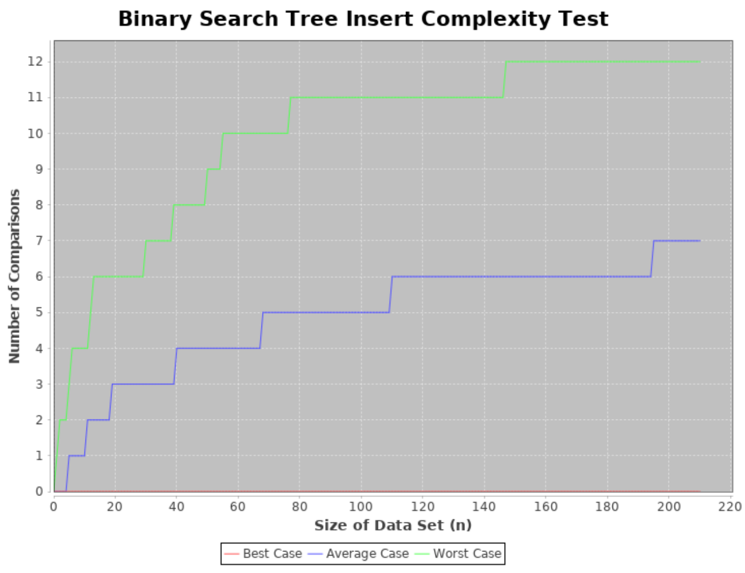
```
Test 3(part 3 and  4)
First 10 entries:

Printing all dam entries from AVL Tree

name: Lindleyspoort Dam
level: 2.7
FSC: 14.208


name: Flag Boshielo Dam
level: 40.200000000000003
FSC: 185.13


name: Buffelspoort Dam
level: 71.400000000000006
FSC: 10.183


name: Blyderivierpoort Dam
level: 50
FSC: 54.369


name: Armenia Dam
level: 10
FSC: 12.957000000000001
```

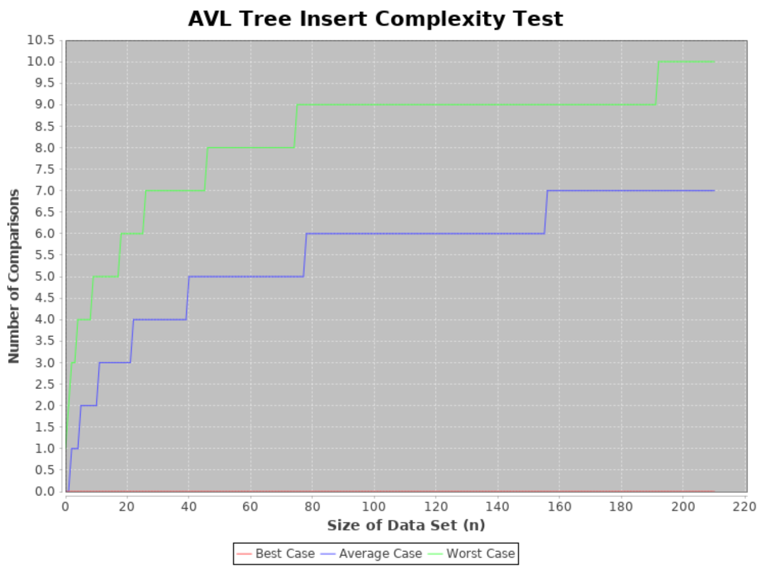
```
Test 3(part 3 and  4)

name: Albert Falls Dam
level: 36.4
FSC: 288.14


name: Albasini Dam
level: 69.2
FSC: 28.199000000000002


name: Allemanskraal Dam
level: 13.5
FSC: 174.52


name: Berg River Dam
level: 31.1
FSC: 127.05


name: Belfort Dam
level: 98
FSC: 0.413


…
```

```
Test 3 (part 4)(continued)
Last 10 entries:

…

name: Welbedacht Dam
level: 97.5
FSC: 9.592


name: Waterdown Dam
level: 90.1
FSC: 37.441000000000003


name: Witbank Dam
level: 52.2
FSC: 104.02


name: Westoe Dam
level: 48.3
FSC: 60.094999999999999


name: Witklip Dam
level: 39.299999999999997
FSC: 12.519
```

```
Test 3 (part 4)

name: Wriggleswade Dam
level: 98.7
FSC: 91.471000000000004


name: Woodstock Dam
level: 77.099999999999994
FSC: 373.25


name: Xonxa Dam
level: 100
FSC: 115.86


name: Xilinxa Dam
level: 27.2
FSC: 13.823


name: Zaaihoek Dam
level: 58.5
FSC: 184.63

Printed all dam entries from AVL tree.
the size of the dataset was: 211
The number of insertion comparisons
(best, average and worst) were: 0, 7, 10
```

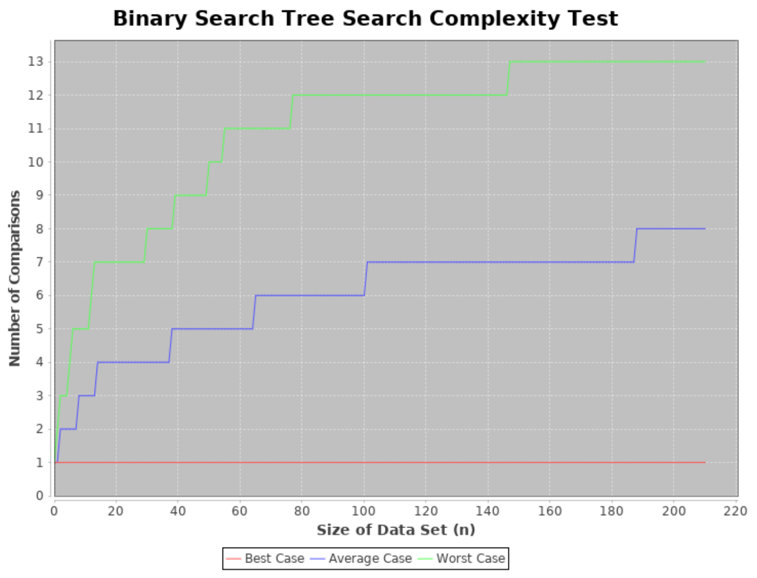
Time complexity of insert algorithms:

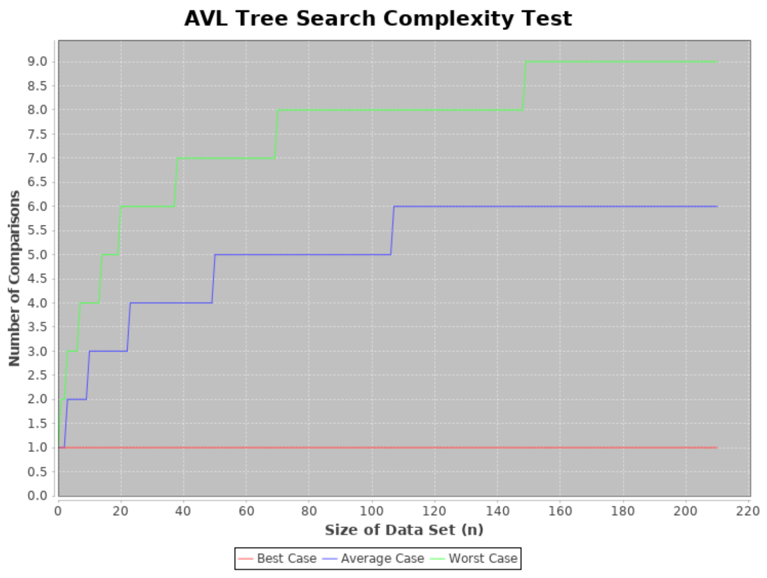




The graphs above are a graphical representation of the opCounts recorded during the insertion of data into the BS and AVL Trees, vs the size of the data set. The best, average and worst cases plotted in each case. All curves show a log(n) shape - suggesting asymptotic behaviour for large n (size dataset). By comparison the AVL tree proves to have a more efficient underlying algorithm due to the balancing of the tree during the insertion process (from the data we see an 'improvement' of anywhere between 10% and 20%, which will become increasingly significant for very large vases of n). It should be noted that the AVL tree requires additional computation which we have not taken into consideration - calculating the balance factor and height of the subtrees - but this process involve integer comparisons which are significantly faster than String comparisons, which we have measure here, so we count them as negligible. It should also be noted that the best case, in every case, is 0 which is the insertion of the first piece of data at the root.

Time complexity of search algorithms:





The results from the test of time complexity of the search algorithms show similar results to that of the insertion test. The shape is, again, log(n) for all curves (roughly). By comparison, the AVL tree both increases slower (shallower gradient) in all cases, and approaches asymptotic behaviour faster that the Binary Search Tree. The advantage for the AVL Tree clearly being the fact that the data structure is balanced according to the properties of an AVL Tree: a maximum difference of 1 in the heights of subtrees. This suggest an improvement in efficiency by nearly 25% for the AVL tree, which will become very significant for large n. The correlated results of the time complexity of insert and search for both structures are correlated which is no coincidence - the binary tree shows an increase in overall complexity in the worst / average case, while the AVL Tree shows a decrease overall, which suggests balancing effects insertion time-complexity, and even more so, search time-complexity.

e. Part 6 Results

```
AVL search opCount Results:
best: 1
average: 6
worst: 8
AVL insert opCount Results:
best: 0
average: 7
worst: 9
```

```
BST search opCount Results:
best: 1
average: 106
worst: 211
BST insert opCount Results:
best: 0
average: 104
worst: 210
```

The results above are of the test run on the sorted dataset (names of Dams in alphabetical order). They are most significant in terms of displaying the power of the balancing algorithm which forms part of the insert algorithm of the AVL tree. The Binary Tree will always exhibit linear search behaviour if the data is ordered - each node connected to only one node, all on the left or right - this O(n) performance is typical of an array, and very taxing computationally. The AVL Tree behaves nearly identically to when the data was unsorted.

f. Creative Components

Below I have listed what I feel are creative efforts on my part to go above and beyond the brief for the assignment.

- fixCSVFile utility class: Instead of simply using the unix head command I created a class to repair line breaks in CSV files, as well as replace commas.

- dataFrame class: the core of my solution to the assignment was creating a robust class to store and access information in a CSV file by column and dam name, as well as query details about its dimensions. This was used as an object in both my DamArrayApp and DamBSTApp

- I created bash scripts to run / generate test output for parts 2, 4, 5 and 6 (in the bin folder)

- test() methods inside DamAVLapp and DamBSTApp: I used an interesting technique to isolate each search case for single subsets of the data. By creating a temporary instance of the same object in which the test() method already exists (and instance within an instance of an object), I was able to effectively and cleanly isolate the set of searches run on each subset of data without creating an additional class to handle the task.

g. Git log (from most recent)

```
commit 979a7b76ce04c8cfe4d87befaf298100c5fc62ca
Author: jeremy <dpljer001@myuct.ac.za>
Date:   Tue Mar 27 17:17:46 2018 +0200

    javadoc 2

commit 4deaa8dfbed2333a3360729ed90659955a26b172
Author: jeremy <dpljer001@myuct.ac.za>
Date:   Tue Mar 27 16:57:28 2018 +0200

    Javadoc 1

commit 23832335662d71f2993d4b019b5675d13c3d3040
Author: jeremy <dpljer001@myuct.ac.za>
Date:   Tue Mar 27 15:58:26 2018 +0200

    Test 5 working for AVL tree, all 4 graphs generating

commit 618bfc0d25b739e89eb4c646791f49b9b47b0ae2
Author: jeremy <dpljer001@myuct.ac.za>
Date:   Tue Mar 27 15:18:13 2018 +0200

    Test 5 working for BST

commit d34f52f40ea0bffd18eedd1b3b247c83d4a69995
Author: jeremy <dpljer001@myuct.ac.za>
Date:   Mon Mar 26 22:41:57 2018 +0200

    Opcount algorithm for insert comparisons working for BST, results writing to file for all tests

commit c5d72c4c8b41e295209c8ef14d88581e4fe58eec
Author: jeremy <dpljer001@myuct.ac.za>
Date:   Mon Mar 26 19:26:25 2018 +0200

    tests 2, 4, 5 and graph function all working, Makefile running everything correctly

commit c4b5872abcaa5536ad1d7b4edc9774a30811abc8
Author: Jeremy du Plessis <dpljer001@myuct.ac.za>
Date:   Wed Mar 21 08:06:36 2018 +0000

    Initial commit
```