Computer Science 2001F 2018 Assignment 2:

Problem Description:

Computer Science students (CSC2001F) were given an assignment to check or to compare the effieciency of an AVL tree and Binary Search Tree, an AVL tree is a binary search tree with additional balancing propertiess. The students were provided with a file, in the file there is information about dams, their names , dam level and more. We are required to only take certain columns in the file to use. The first parts of the assignment must make use of an AVL tree as a data structure and the last part of the assignment students are required to use a Binary Search Tree (BST) to store the data.

The data that is stored in these data structures is read from a csv file and only certain columns are stored in the data structures, which are dam name, FSC and dam level. There are certain functions or methods that are necessary to make use of the data provided. Two methods are supposed to be used in part one which are *printDam(String dam)* this method takes in a parameter and is used to compare the parameter to data in the data strucutures. Another method that students should make use of is printAllDams(). This method is resposnible for printing all the dams and their information stored in the data structures crated by the students.

The main goal or aim of this assignment is to be able to cpompare an AVL tree and Binary Search Tree, both of them are implemented in java and a real world application is to be used to read and provide access to dam water level data. In this assignment we have to compare two data structures, the two data strucutures are an AVL tree and a BST tree. Students are are required to create code that will also do calculations when inserting an item to the data strucutres and also when searching for an item. The calculation should tell us how long it took it took to search or find a certain data or information about dams and at the end you comapre these calculations, you calculate average and find best case, average case and also worst case.

Expiriment Design

The aim of this assesment is to compare the two data structures which are a Binary Search Tree and an AVL tree. By using these data strucutres students are are required to create a code that will also do calculations when searching for an item in the data structures, the calculation should should show how long it took to search or find a certain data or information about dams and at the end you compare these calculations.

Steps of the expiriment:

1. You will have a subset of dams to use which you will search on the data strucutres.

2. Record the time taken to find the dam name in the data strucure which you are searching in.

3. Repeat expiriment, and take average

4. Collect results or record them down so that you will be able to make tables or draw graphs.

5. Lastly compare all the dam names, make sure they are sorted and calculate time for them, this is part 6 of the assignment. Print this in a textfile or any other file.

6. Conclude based on what the findings where which of the data strucutres is the best when it comes to perfomance.

7. Discuss results or findings about the two data structures, The aim is to see which data structure is more efficient when used, for insertion and searching (finding an item in the data structure).

Design of application

Function: AVLTree.java

The class AVLTree extends Comparable and also extends BinaryTree. This class consists of multiple methods that will be used by other classes. This class is used to get more information about the AVL tree created. For example when you want to know the height of a tree, you use a method that returns an interger, the height of the AVL tree, *public int height (BinaryTreeNode<dataType> node).* Other methods are insert, which is called when inserting an item in the AVL tree. Another method is delete, which is responsible for removing node in the data structure being used. Another method which is very useful in this code is **num_comparisons(),** this method returns the number of comparisons after searching for an item.

Function: DamAVLApp.java

This class *DamAVL* is the main class that is used for part 1 and part 2 of the assignment, where students are required to make use of the AVL tree to store data and make use of the AVL tree that has been created for example printing data the way it is stored in the AVL tree also to search the AVL tree to see if a certain dam name does exist or not. This class makes of other classes, by creating an object of the class that it will use, for example it creates an object of class *AVLTree.*

Methods in DamAVLApp.java:

1. *printAllDams():* print all dams is a method inside this class that is used to print all the dams stores in the AVL tree. It does this by firstly creating an object of class AVLTree and then using the object or instance to call the printing method in the AVLTree class. This method is a void method , which means that it does not return anything it is just used to call the printing method from the AVLTree class.

2. *printing_S(String dam_name, BinaryTreeNode<String> rootNode) :* This method is used for searching an item in AVL tree created. It takes two parameters which are dam_name and rootNode. The dam_name parameter is input from the user, the user inputs a dam name that he or she would love to search in the AVL tree. If the dam name is found it will print the information about the dam that was being searched. For example if a user searhes for “Xonxa Dam” this will be the output:

number of comparisons: 7 for |Xonxa Dam 115.86 100

It will also show the number of comparisons it took to find the dam name in the AVL tree. This method of calculating comparisons is created in the AVLTree class, it simply calculates the number of comparisons. If the dam name is not found it will simply say that : *"The dam was not found"*.

3. *main(String arg[]) :*

In the main method it is where our file with data is being read, the csv file with dam names. As explained above only ceratin columns are really being used in the csv file. In the main method the file is read line by line and each line is split to select the columns that are needed then stored in the AVL tree. Arrays have also been used but only for printing purposes and comparing all the dam names from array to the dam names in the AVL tree. Arg is being used as input from the user in this case, when a user wants to search for a name , the name which is taken as input is inserted in the array called arg and if the length of arg is greater than one that simply means that there is a dam name inside, else nothing. Arg is used in the *printing_S* method and dam_name which is a parameter in the method *printing_S will be arg[0]*. If arg.length is less than 1 then *printAllDams()* will be called else *printing_S* method will be called.

Function : BinaryTreeNode.java

This class is used for DamAVLApp and also DamBSTApp, this class is being called when using insertion into the BST and AVL tree. It makes sure that rules of BST and AVL tree are being manatined as to which subtree to insert the new data in.

Methods in BinaryTreeNode.java:

1. In this class there is a constructor , which takes in several parameters and initialize the variables declared in the beginning to the data received from user (to data in the parameters).

2. getData() : this is a very short method which just returns data so that it will make it easy to print the data in the BST or AVL tree. If getData is not used when printig it will return an address instead of the actual value you would like to print.

Function : BinaryTree

BinaryTree class consist of methods used by the other classes especially DamBSTApp class. DamBSTApp class calls preOrder when printing data in the Binary Search Tree (BST).

Function : DamBSTApp

The DamBSTApp class is used for part three, four and five of the assignment. Informtion about dams is now stored in a different data structure which is a BST (Binary Search Tree). This class also makes use of multiple other classes.

Methods in class DamBSTApp:

1. *printing_S(String dam_name, BinaryTreeNode<String> rootNode)*-- This methods has two parameters which are dam name and rootNode. This method is used to search for a dam name if it exists or not. If it exists information about dam is collected this is then written to a file which will be used to create or compare the information about the two data types and their efficiency. This method returns the information about the dam searched and how many comparisons it took to find it.

2. *main method(String arg[])* -- this methods reads from file and inserts infromation of the dams in a binary search tree. It also writes to a file using FileWriter and printWriter. The parameter is seen as an input form the user, when main is exucuted it will check if arg is empty or not. If arg is not empty it will then invoke the printing_S method and insert the arg[0] as a parameter (*dam_Name*). If it is empty then main will make use of an object or instance **bt** this will invoke the method in the class *BinaryTree* ,which is *preOrder()*. This method preOrder() will print the binary search tree in preOrder().

Output screens

Below is some examples of the ouput screens which also contai data which will be very useful when drawing the graph. Firsly BST output will be shown and explained in to detail below.

Class : DamBSTApp.java

Susbet_1 BST:

Insertion

time: 3624.0 nanoseconds | Armenia Dam 12.957000000000001 10
time: 3159.0 nanoseconds | Welbedacht Dam 9.592 97.5
time: 4482.0 nanoseconds | Knellpoort Dam 130 32.4
time: 3960.0 nanoseconds | Gariep Dam 5196.04 53.9
time: 5072.0 nanoseconds | Vanderkloof Dam 3171.3 57
time: 4013.0 nanoseconds | Disaneng Dam 14.125 34.200000000000003
time: 375246.0 nanoseconds | Setumo Dam 20.718 41.7
time: 5059.0 nanoseconds | Boegoeberg Dam 19.815000000000001 93.5
time: 4118.0 nanoseconds | Bulshoek Dam 4.809 65
time: 3588.0 nanoseconds | Clanwilliam Dam 122.48 20.6
time: 4170.0 nanoseconds | Karee Dam 0.949 16.8
time: 5117.0 nanoseconds | Voelvlei Dam 158.58000000000001 22.2
time: 4203.0 nanoseconds | Wemmershoek Dam 58.71 52.3
time: 5335.0 nanoseconds | Misverstand Dam 6.439 88.8
time: 4655.0 nanoseconds | Berg River Dam 127.05 31.1
time: 4652.0 nanoseconds | Steenbras Dam Lower 33.880000000000003 48.1
time: 5214.0 nanoseconds | Eikenhof Dam 28.856000000000002 54.3
time: 5684.0 nanoseconds | Steenbras Dam - Upper 31.811 60
time: 3583.0 nanoseconds | Debe Dam 6.331 86.9
time: 3601.0 nanoseconds | De Bos Dam 5.735 71.3
time: 4819.0 nanoseconds | Brandvlei Dam 286.04000000000002 24.8

Above is a subset taken from the outputs of a Binary Search Tree, the sample or subset size (N) is 22. Above the best case , average case and worst case are based on the calculations of time. The time represents the ammount of time taken to insert a certain dam in the Binary Search Tree.

second_Subset BST:

time: 73983.0 nanoseconds | Ngotwane Dam 19.033000000000001 4.6
time: 129351.0 nanoseconds | Hartbeespoort Dam 186.44 96.5
time: 18726.0 nanoseconds | Bon Accord Dam 4.381 103
time: 11646.0 nanoseconds | Olifantsnek Dam 13.677 32.9
time: 27582.0 nanoseconds | Rietvlei Dam 12.25
time: 12705.0 nanoseconds | Buffelspoort Dam 10.183 71.400000000000006
time: 15105.0 nanoseconds | Bospoort Dam 15.798999999999999 98.1
time: 11359.0 nanoseconds | Lindleyspoort Dam 14.208 2.7
time: 10528.0 nanoseconds | Warmbad Dam 0.549
time: 9881.0 nanoseconds | Roodeplaat Dam 41.158000000000001 98.3
time: 8302.0 nanoseconds | Kosterrivier Dam 12.417 34.200000000000003
time: 12412.0 nanoseconds | Klipvoor Dam 40.734999999999999 52.4
time: 11695.0 nanoseconds | Swartruggens Dam 0.475 0.7
time: 13901.0 nanoseconds | Vaalkop Dam 51.314999999999998
time: 13888.0 nanoseconds | Roodekopjes Dam 102.33 34.5
time: 11387.0 nanoseconds | Marico-Bosveld Dam 26.963000000000001 14.3
time: 22285.0 nanoseconds | Klein Maricopoort Da 7.073 19.399999999999999
time: 12426.0 nanoseconds | Kromellenboog Dam 8.956 13.7
time: 16261.0 nanoseconds | Molatedi Dam 200.79 6.8
time: 52049.0 nanoseconds | Sehujwane Dam 3.614 74.8
time: 16693.0 nanoseconds | Madikwe Dam 15.938000000000001 24.8
time: 10494.0 nanoseconds | Pella Dam 2.111 38
time: 8210.0 nanoseconds | Mokolo Dam 145.37 77.099999999999994
time: 10401.0 nanoseconds | Doorndraai Dam 43.764000000000003
72.099999999999994
time: 19714.0 nanoseconds | Glen Alpine Dam 18.888999999999999 10.5

The above data is another subset of data collected. And the values of time measured in nanoseconds are recorded when inserting data item in the Binary Search Tree. The size of the above is 26 (N).

Example of search in BST:

you searched for Xonxa Dam comparisons for this Dam: 7 time: 2075.0 | Xonxa Dam 115.86 100

you searched for Xilinxa Dam comparisons for this Dam: 8 time: 2013.0 | Xilinxa Dam 13.823 27.2

you searched for Lindleyspoort Dam comparisons for this Dam: 1 time: 8990.0 | Lindleyspoort Dam 14.208 2.7

you searched for Tours Dam comparisons for this Dam: 7 time: 2158.0 | Tours Dam 6.084 32

you searched for Vaal Dam comparisons for this Dam: 7 time: 2297.0 | Vaal Dam 2603.4499999999998 48.6

you searched for Toleni Dam comparisons for this Dam: 8 time: 2332.0 | Toleni Dam 0.177

you searched for Sandile Dam comparisons for this Dam: 8 time: 2221.0 | Sandile Dam 29.655999999999999 84.9

you searched for Tzaneen Dam comparisons for this Dam: 6 time: 2247.0 | Tzaneen Dam 156.53 29.9

you searched for Da Gama Dam comparisons for this Dam: 9 time: 3647.0 | Da Gama Dam 13.526 57.9

you searched for Spring Grove Dam comparisons for this Dam: 7 time: 2541.0 | Spring Grove Dam 139.19999999999999 84.8

you searched for Vondo Dam comparisons for this Dam: 7 time: 2436.0 | Vondo Dam 30.446999999999999 63

In the above example it tells what the input name was and what the output will be. When you search in the BST it also shows the number of comparisons and time. Below will be examples of the AVL Tree. Time and number of comparisons are used to sketch the graph is to which one is better perfoming than the other.

Class : DamAVLApp.java

Below will be examples of data output for the AVL Tree, similar to the above but different values and more.

Output 1:

This output it shows how the dams are printed out form the AVL Tree.

Lindleyspoort Dam 14.208 2.7
 Flag Boshielo Dam 185.13 40.200000000000003
  Buffelspoort Dam 10.183 71.400000000000006
   Blyderivierpoort Dam 54.369 50
    Armenia Dam 12.957000000000001 10
     Albert Falls Dam 288.14 36.4
      Albasini Dam 28.199000000000002 69.2
      Allemanskraal Dam 174.52 13.5
     Berg River Dam   127.05 31.1
      Belfort Dam 0.413 98
       Beervlei Dam 85.778999999999996 1.5
       Bellair Dam 4.241 78.3
      Bloemhof Dam 1240.24 15.5
       Binfield Dam 36.848999999999997 96.6
    Bospoort Dam 15.798999999999999 98.1

Output 2:
This one below shows insertion time:

time: 0.0 nanoseconds | Ngotwane Dam 19.033000000000001 4.6
time: 574.0 nanoseconds | Hartbeespoort Dam 186.44 96.5
time: 2823.0 nanoseconds | Bon Accord Dam 4.381 103
time: 5400.0 nanoseconds | Olifantsnek Dam 13.677 32.9
time: 5516.0 nanoseconds | Rietvlei Dam 12.25
time: 14108.0 nanoseconds | Buffelspoort Dam 10.183 71.400000000000006
time: 6432.0 nanoseconds | Bospoort Dam 15.798999999999999 98.1
time: 15112.0 nanoseconds | Lindleyspoort Dam 14.208 2.7
time: 8179.0 nanoseconds | Warmbad Dam 0.549
time: 6244.0 nanoseconds | Roodeplaat Dam 41.158000000000001 98.3
time: 11198.0 nanoseconds | Kosterrivier Dam 12.417 34.200000000000003
time: 9120.0 nanoseconds | Klipvoor Dam 40.734999999999999 52.4
time: 11846.0 nanoseconds | Swartruggens Dam 0.475 0.7
time: 11292.0 nanoseconds | Vaalkop Dam 51.314999999999998

time: 11237.0 nanoseconds | Roodekopjes Dam 102.33 34.5
time: 12571.0 nanoseconds | Marico-Bosveld Dam 26.963000000000001 14.3

Output 3:

This one below shows time for searching and number of comoparisons:

number of comparisons: 6 time: 2251.0 for |Setumo Dam 20.718 41.7

number of comparisons: 7 time: 2491.0 for |Sterkspruit Dam 9.473000000000001

number of comparisons: 7 time: 1338.0 for |Vondo Dam 30.446999999999999
63

number of comparisons: 7 time: 2168.0 for |Vondo Dam 30.446999999999999
63

number of comparisons: 8 time: 1481.0 for |Zaaihoek Dam 184.63 58.5

number of comparisons: 7 time: 2491.0 for |Woodstock Dam 373.25
77.099999999999994

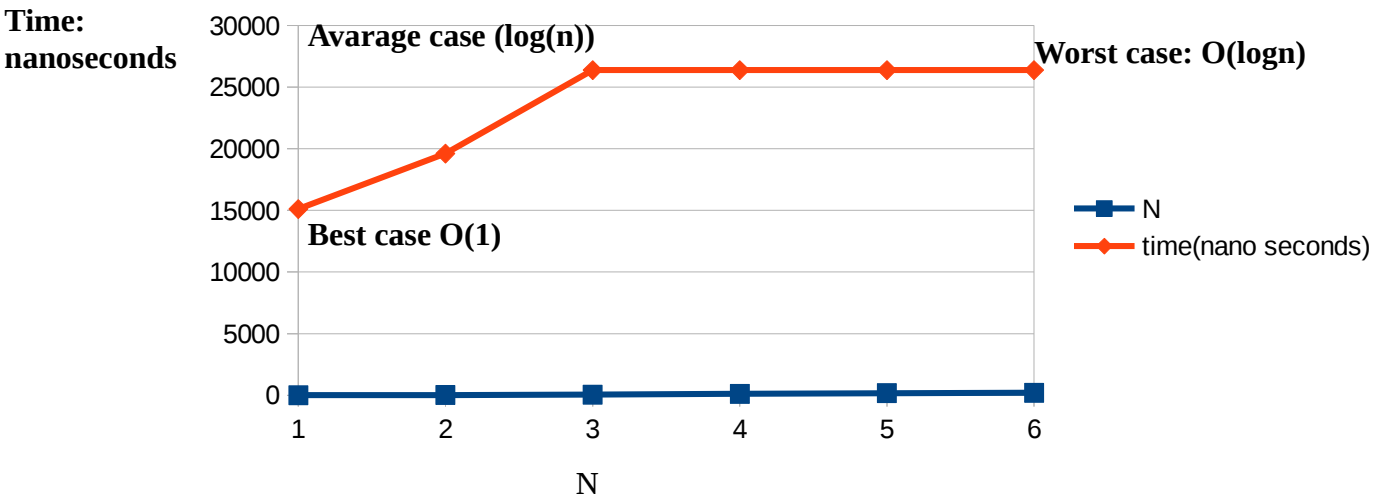
number of comparisons: 7 time: 1612.0 for |Spring Grove Dam
139.19999999999999 84.8

Comparisons Expiriment

In this expiriment below, different sizes of subsets are made for both the DamBSTApp class and DamAVLApp class. The subsets are being compared and graphs are made to make the comparisons more clear and to show the difference in efficiency between the two graphs. Time and number of comparisons is used to make sure that the data is more reliable.

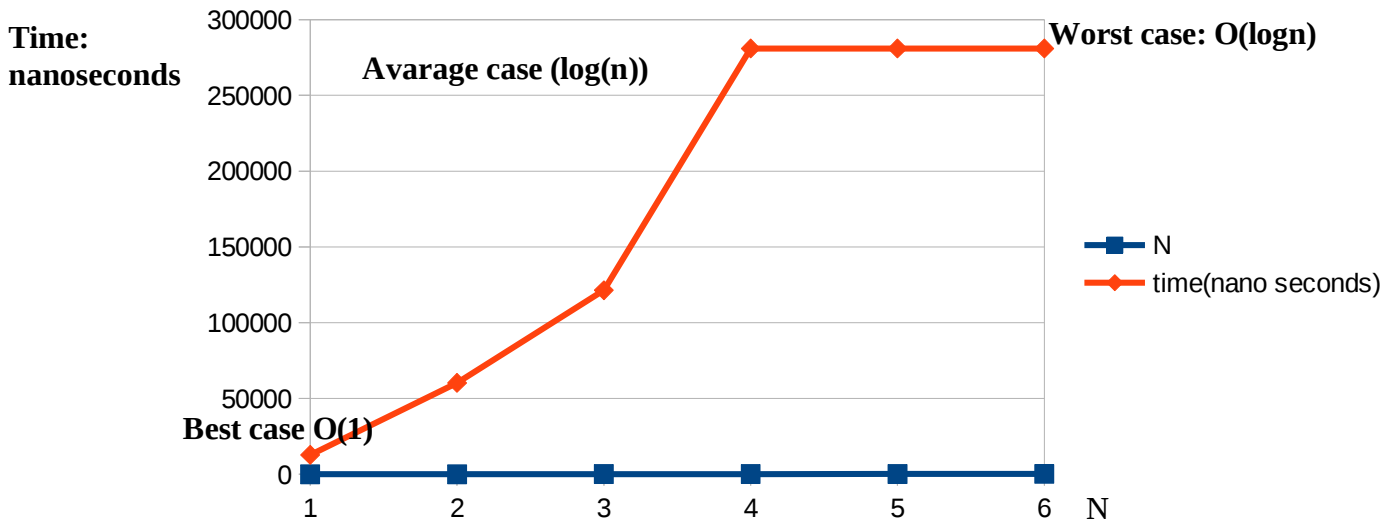
This is how the graph will look like after conducting the expiriment using different different subsets and searching them.

Insert And Find AVL ---> O(log)



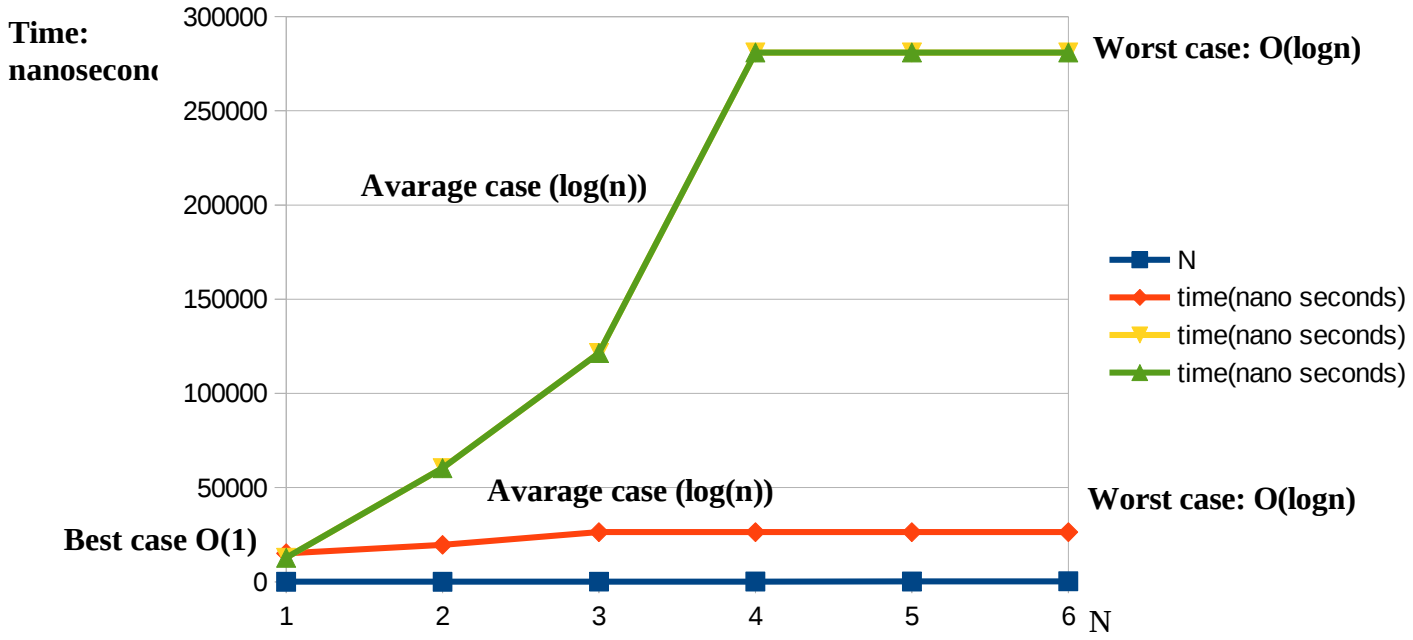
In the above graph different subsets of different numbers were taken ,the shape of the above graph is similar to that of the logs, F(x)=log(n). The worst case follows the graph of the log function. This shows for both the find function and insert function.

Insert And find BST ---> O(log)



The above graph shows the find and insert function of a BST. Both these functions follow log(n) shape. F(x)=log(n), in the worst case.

BST And AVL Tree (shapes) in one graph



Time Complexitity

Below is conlcusion of time complexity for both data structures

Data Structure	Best Case	Average Case	Worst Case
BST	O(1)	O(logn)	O(logn)
AVL Tree	O(1)	O(logn)	O(logn)

Discussion of results (part 5)

As we can see from the above results, of comparisons, the AVL comparisons are are much lower in all the subsets of data that we made and their avarage is lower compared to that of the BST comparisons and time. Both the BST and the AVL Tree seem to take the shape of the logarithmic function. But as we can see form the above graph AVL Tree grows very slow compared to the Binary Search Tree. As N increases The BST tends to grow faster and it shows that the Binary Search Tree takes more time in avarage compared to the AVL Tree. According to this expiriment the AVL is more efficient. They both have the same logarithmic function but the constant is what differs, for example F(x)=**X**logn . X is the constant and it is different for both BST and AVL Tree. Also the reason why AVL is taking less time or comparisons it is because it is a balanced Binary Search Tree. For small value of N the might be little or no difference, but as N grows, the time also grows in the direction of N, for example N increases then time also increases, the relationship is directly proportional.

Discussion of results (part 6)

The expiriment done on part six is about, sorting according to dam names and doing the expiriment ,with sorted dam names. When the dam names are sorted by their name this make it much more easier to find or search for a certain dam . The time taken to find a dam name when a user searches becomes low compared to when the dam names are not sorted at all. Sorting dam names make it more efficient to search or find a certain dam name. Still the AVL tree is better in terms of speed and comparisons as it is a more balance tree and its height is not too long.

More Examples of how the graphs would look like:

System.out.println("Examples taken from the internet");

Git usage log

commit 90bb4c14b184ad05a5fc26f1846d4f86994d62b8 (HEAD -> master)
Author: Zuqhame Skosna <SKSZUQ001@myuct.ac.za>
Date: Wed Mar 28 14:40:34 2018 +0200

Finished time and also finished comparisons, just created files to store data generated by the program.

commit 4d07cd113a1cb5b745fef28f45bdc43820cc6ed9
Author: Zuqhame Skosna <SKSZUQ001@myuct.ac.za>
Date: Wed Mar 28 03:03:36 2018 +0200

Trying to add and do the expiriment in both classes.will create one file and update it to chcek how relevent the data is.
count number of comparisons and time taken in nano seconds.

commit f1e9525169d1587bb8828df7ad510638edd34a4d
Author: Zuqhame Skosna <SKSZUQ001@myuct.ac.za>
Date: Mon Mar 26 14:41:49 2018 +0200

added a new method that calculates the time taken to calculate the number of comparisons, calculated in nano seconds, when using seconds the number is too low and is always zero since not working with a very huge file

commit 59ac65f3fbd9c921dd947bc685b3a3ce88a5f097
Author: Zuqhame Skosna <SKSZUQ001@myuct.ac.za>
Date: Mon Mar 26 13:19:04 2018 +0200

added an updated version of the class classes and currently working on doing calculations for time

commit 13e1e52e2cd2c1755e0f043347599523e8bdcd3e
Author: Zuqhame Skosna <SKSZUQ001@myuct.ac.za>
Date: Sat Mar 24 20:43:55 2018 +0200

added all classes which will be used in assignment two
just did part one and part two of the assignment, storig (insert) and searching (find) data in the AVL tree. Still has a minor error when searching for a name that does not exist.

Conclusion :

In the expiriment conducted, it is therefore an AVL tree that is more efficient, it takes less time to insert and to find an object. Both the BST and An AVL tree have have the logarithmic function. O(log n), but the BST grows quicker than the AVL Tree hence AVL is better ,it gorws slow, takes less time to perform some operations and more other things, its main advantage to the Binary Search Tree is that the AVL tree is balanced.