NAME: JABULANI MAVODZE
STUDENT NO: MVDJAB001
MODULE: CSC 2001F
TASK:ASSIGNMENT 2

TITLE:AVL Tree vs Binary search tree

•DESIGN

I am going to be applying the object oriented programming methodology so to archive a clean and easy to understand code to model the respective data structures, I am going to have four classes namely **BstNode** , **Entry** , **AVLTree1** , **BST** *and four additional classes that will run the main method and perform different functions using objects of the* **AVLTree1** *and* **BST** *the classes are* **LSAVLTreeApp , LSBSTApp , TestApp1** *and* **TestApp.**

***BstNode class**:

The BSTNode class is going to used to create the node objects that are going to create the Binary Search tree and AVL tree ,they're are going to be predominantly used in the AVLTree and BST classes.

•Instance variables:

(+) Entry data -> *The instance data of type Entry is going to store data from the load shedding file(line by line I.e 1_1_00 1, 5), the instance data holds the content of the node object and it is public.*

(+) BstNode left -> *The instance left of type BstNode is the reference to the left child of the node and it is public.*

(+)BstNode right -> *The instance right of type BstNode is the reference to the right child of the node and it is public.*

(+) height ->

•Methods:

- ***NB:****This class has only one method the constructor.*

(+)BstNode(data, left, right) -> *Is the constructor used to initialize the class instance variables (I.e data , left and right).*

***Entry class:**

The Entry class is going to be used to create the fragmented data of the original data from the load shedding file, the data will be fragmented in this manner the **theKey** *will hold the first part of the data prior to the empty space(I.e original data -> 1_1_00 1, 4 ,* **theKey -> 1_1_00** *excluding {1, 4}) , and the* **Areas** *will hold the last part of data after the empty space(I.e original data -> 1_1_00 1, 4,* **Areas-> 1, 4** *excluding {1_1_00}).The fragmented data is going to be manipulated in conjunction with the original data to create the Binary search tree and AVL tree respectively.The other reason I need the fragmented data it is because I am going to use fragmented data from the terminal particularly the* **theKey** *for other objectives like instrumentation and when conducting the experiment.*

•Instance variables:

(-) String theKey -> *The instance theKey of type String is going to store the first part of fragmented data (I.e* **theKey -> 1_1_00)** *and it is private.*

(-) String Areas -> *The instance Areas of type String is going to store the latter part of the fragmented data(I.e* **Areas -> 1, 4)** and it is private.

•Methods:

- **NB:** *This class has three methods a constructor and because of encapsulation it has two accessor methods.*

(+) Entry(String theKey, String Areas) -> *Is the constructor used to initialize the class variables(I.e theKey and Areas), ant it takes only two parameters of type string.*
(+) getTheKey() -> *Is the accessor method used to access the* **theKey** *variable, thus it returns a string.*
(+) getAreas() -> *Is the accessor method used to access the* **Areas** *variable ,thus it returns a string.*

***BST Class:**
The BST class is going to be used to create the **BST** *object in the BSTApp class.*

•Instances Variables:
(-) int count -> *The instance count of type int ,is going to be used to count the number of comparison operations(I .e < ,> , =) performed in this class ,the comparisons will be performed in two methods namely* **find()** *and* **insert().**

•Methods:

- **NB:***This class has six methods a constructor, a mutator method for the count, two helper methods , an insert method and a find method.*

(+) setCount(int x)-> *Is a mutator method for the count, it takes only one parameter of type int and it returns an integer .*
(+) insert(Entry data)-> *Is an overloaded helper method to the primary insert method ,it takes only one parameter of type Entry.*
(+) insert(Entry data, BstNode node)->*Is the primary insert method used to insert a node into a binary search tree, takes two parameters of type Entry and BstNode.*
(+) find(String data)-> *Is an overloaded helper method to the primary find method, it takes only one parameter of type String.*
(+) find(String data,BstNode) -> *Is the primary find method, it takes two parameters of type String and type BstNode respectively.*

***AVL TREE CLASS:**
The BST class is going to be used to create the AVLTree1 object in the LSAVLTreeApp class.
• Instances Variables:
(-) int count -> *The instance count of type int ,is going to be used to count the number of comparison operations(I .e < ,> , =) performed in this class ,the comparisons will be performed in two methods namely* **find()** *and* **insert().**

•Method:

Methods:

- **NB:***This class has six methods a constructor, a mutator method for the count, two helper methods , an insert method and a find method.*

(+) setCount(int x)-> *Is a mutator method for the count, it takes only one parameter of type int and it returns an integer .*
(+) insert(Entry data)-> *Is an overloaded helper method to the primary insert method ,it takes only one parameter of type Entry.*
(+) insert(Entry data, BstNode node)->*Is the primary insert method used to insert a node into a binary search tree, takes two parameters of type Entry and BstNode.*
(+) find(String data)-> *Is an overloaded helper method to the primary find method, it takes only one parameter of type String.*
(+) find(String data,BstNode) -> *Is the primary find method, it takes two parameters of type String and type BstNode respectively.*
(+) height(BstNode node)-> *Is the method used to get the height of the node provide*

(+)balanceFactor(BstNode->*Is the method used to calculate the balance Factor*
(+)fixHeight(BstNode node)->*Is the method used to check if the height difference of the sub trees for not violet the AVL tree property.*
(+)rotateLeft(BstNode)->*Is the method used to rotate to the left ,if tree is unbalanced.*
(+)rotateRight(BstNode)->*Is the method used to rotate to the right, if tree is unbalanced.*

***EXPERIMENT**
***INSTRUMENTATION:**
Instrumentation for the BST
***Trial test values and outputs (Part 2).**
***Three valid inputs for the find method:**
-input 1 : 2_18_08
The Areas are/is: [1, 9] and The count is:{55}
-input 2 : 3_23_20
The Areas are/is: [12, 4, 16] and The count is:{113}
-input 3 : 8_23_10
The Areas are/is: [7, 15, 3, 11, 8, 16, 4, 12] and The count is:{369}

***Three invalid inputs for the find method:**
-input 1 : 4_10
Error enter valid entry: and The count is:{0}
-input 2 : 5_20_@
Entry not found : and The count is:{382}
-input 3 : 6;_10_12
Entry not found : and The count is:{352}

Instrumentation for the AVL Tree
***Trial test values and outputs (Part 4).**
***Three valid inputs:**
-input 1 : 2_18_08
The Areas are/is: [1, 9] and The count is:{5}
-input 2 : 3_23_20
The Areas are/is: [12, 4, 16] and The count is:{6}
-input 3 : 8_23_10
The Areas are/is: [7, 15, 3, 11, 8, 16, 4, 12] and The count is:{7}

***Three invalid inputs :**
-input 1 : 4_10
Error enter valid entry: and The count is:{0}
-input 2 : 5_20_@
Entry not found : and The count is:{10}
-input 3 : 6;_10_12
Entry not found : and The count is:{10}

- **The AIM:**

The aim of this experiment is to compare the time complexities of the best case, average case and worst case for both the binary search tree and AVL tree using real world data(The Cape Town load shedding data), so to deduce which of the two data structures is more efficient.

- **The METHOD:**

- *The first step is to create the binary search tree class and the AVL tree class respectively using the object oriented programming methodology.*
- *The second step is to create a TestApp1 class and TestApp2 class ,TestApp1 class is going to be used to generate the best case ,average case and worst case data of the the insert function of each data structure and TestApp2 class is going to be used to generate the best case , average case and worst case data of the find function for each data structure ,they will do this for different files(10 files to be precise) of different sample sizes.The data generated by each class will be stored in two txt files respectively to be used for analysis.*
- *The third step is to perform analysis using Microsoft excel, in this step I will generate graphs using data I got after executing step two.*

• **Discussion:**

I am going to compare the asymptotic time complexities of the binary search tree and AVL tree for their find functions and insert functions .

- **<u>Find functionAnalysis:</u>**
  -NB: *The figures being discussed here can be found in* **annexure** *1(**Please*** **open file :: Find_Analysis_1)**

*After plotting the insertion data for both data structures on excel, I acquired two functions for each case, (***Figure** *1.1) for the average case ,* **(Figure 1.2)** *for the best case and (***Figure 1..3***) for the worst case.The two functions on each figure depict the relationship between the count and the number of data elements in a file.*

•Average case analysis:

- **Binary search tree:**
  The equation of the function is : ***f(n)*** *= 0.0299n– 1.8793*
  The coefficient of determination is : **R^2 = 0.995**
  Which means **99.5%** *of the outcomes(count) can be explained by the inputs(number of data elements in a file) ,this shows a very strong* ***Linear relationship*** *between the two variables.*
  The big oh notation of this function is **:O(n),** *we only include n because it's the dominant term.*
- **AVL TREE:**
  The equation of the function is ***: f(n) =1.3923 log(n)- 1.0307***
  *The coefficient of determination is :***R^2 = 0.9372**
  Which means with ***93.7%*** *of the outcomes (count) can be explained by the inputs (number of data elements in a file), this shows a very strong* ***logarithmic relationship*** *between the two variables.*
  *The big Oh notation of this function is :***O(log(n)),***we only include log(n) because it's the dominant term.*

•Worst case analysis:

- **Binary search tree:**
  *The equation of the function is:****f(n) = 0.0537n -3.4255***
  *The coefficient of determination is:****R^2 = 0,985***
  Which means **98.5%** *of the outcomes(count) can be explained by the inputs, this shows a very strong* ***Linear relationship*** *between the two variables.*
  *The big oh notation of this function is:***O(n),** *we online include n because it's the dominant term.*

- **AVL TREE:**

*The equation of the function is **: f(n) =1.3022 log(n)- 4.4812***
*The coefficient of determination is :***R^2 = 0.9216**
*Which means with **92.1%** of the outcomes (count) can be explained by the inputs (number of data elements in a file), this shows a very strong **logarithmic relationship** between the two variables.*
*The big Oh notation of this function is :***O(log(n)),***we only include log(n) because it's the dominant term.*

•Best case analysis:

- **Binary search tree:**
  *The equation of the function is: **f(n) = 1***
  *The coefficient of determination is:**N/A (**No variations).*
  *The big oh notation of the function is:**O(1) (constant).***
- **AVL Tree:**
  *The equation of the function is: **f(n) = 1***
  *The coefficient of determination is:**N/A (**No variations).*
  *The big oh notation of the function is:**O(1) (constant).***

•Insert function analysis:

NB:** The figures being discussed here can be found in* **annexure** *2(**Please **open file :: Insert_Analysis_1)**

*After plotting the insertion data for both data structures on excel, I acquired two functions for each case, (***Figure** *2.1) for the average case ,* **(Figure 2.2)** *for the best case and (***Figure 2.3***) for the worst case.The two functions on each figure depict the relationship between the count and the number of data elements in a file.*

Average case analysis:

- **Binary search tree:**
  *The equation of the function is : **f(n) = 0537n – 3.3822***
  The coefficient of determination is : **R^2 = 0.9893**
  Which means **98.9%** *of the outcomes(count) can be explained by the inputs(number of data elements in a file) ,this shows a very strong **Linear relationship** between the two variables.*
  The big oh notation of this function is **:O(n),** *we only include n because it's the dominant term.*
- **AVL TREE:**
  *The equation of the function is **: f(n) =1.3023 log(n)- 4.4825***
  *The coefficient of determination is :***R^2 = 0.9213**
  *Which means with **92.1%** of the outcomes (count) can be explained by the inputs (number of data elements in a file), this shows a very strong **logarithmic relationship** between the two variables.*
  *The big Oh notation of this function is :***O(log(n)),***we only include log(n) because it's the dominant term.*

•Worst case analysis:

- **Binary search tree:**
  *The equation of the function is:**f(n) = 0.97n -9.1754***
  *The coefficient of determination is:**R^2 = 0,9894***
  Which means **98.8%** *of the outcomes(count) can be explained by the inputs, this shows a very strong **Linear relationship** between the two variables.*
  *The big oh notation of this function is:***O(n),** *we online include n because it's the dominant term.*

- **AVL TREE:**
  *The equation of the function is **: f(n) =1.3923 log(n)- 1.0307***
  *The coefficient of determination is :***R^2 = 0.9372**
  *Which means with **93.7%** of the outcomes (count) can be explained by the inputs (number of data elements in a file), this shows a very strong **logarithmic relationship** between the two variables.*
  *The big Oh notation of this function is :***O(log(n)),***we only include log(n) because it's the dominant term.*

•Best case analysis:

- **Binary search tree:**
  *The equation of the function is: **f(n) = 1***
  *The coefficient of determination is:**N/A (**No variations)*
  *The big oh notation of the function is:**O(1) (constant)***
- **AVL Tree:**
  *The equation of the function is: **f(n) = 1***
  *The coefficient of determination is:**N/A (**No variations)*
  *The big oh notation of the function is:**O(1) (constant)***

•CONCLUSION

The time complexities for the insert function for the **Binary Search Tree are :*
-Best case: **O(1)**
Worst case : **O(n)**
Average case: **O(n)**
*The time complexities for the insert function for the **AVL Tree are :***
-Best case: **O(1)**
Worst case : **O(log(n))**
Average case: **O(log(n))**

The time complexities for the find function for the **Binary Search Tree are :*
-Best case: **O(1)**
Worst case : **O(n)**
Average case: **O(n)**
*The time complexities for the find function for the **AVL Tree are :***
-Best case: **O(1)**
Worst case : **O(log(n))**
Average case: **O(log(n))**

*A logarithm function is a slowly growing function compared to a linear function, thus we can conclude that the AVL tree is more efficient compared to the binary search tree, because it has an upper bound of **f(n)** =***log(n)** *for both the insert and find functions compared to the upper bound* **f(n) = n** *for the binary search tree.*