Testing performance of an AVL Tree.

Vusi Skosana

26 April 2021

CSC2001F

Design

Two classes have been created, the other five have been obtained from UCT CSC2001F recourses. The created classes are as follows:

Students

A simple class. It has two attributes for, student number, full name and an insert-counter variable. It has get methods for student number and full name, a *compareTo* method to compare attributes of the calling object to the one in the argument and increments the insert counter, a *toString* method to display the student number and full name in one line, and lastly a tostringwithcounter method to di splay the student number, full name and the insert-count value.

AccessAVLApp

A class to process information from a file into a Binary Tree data structure, and operations can be performed to retrieve specific data. It has one attribute, a AVLTree<Students> type object, avl, which is one of the other classes, an explanation of the other classes leading to a AVLTree class:

- BinaryTreeNode class, a generic class, has four attributes, a generic type (data) of type Student, an integer height variable and two others of its type, (left and right), of BinaryTreeNode<Students> type. It has a single constructor to assign all the attributes to their values. It has two methods, *getLeft* and *getRight* that return the attributes of the object calling them.
- BinaryTree class, a generic class, has one attribute, a BinaryTreeNode<Students> root object. It has seven methods. The *getHeight* method, to obtain tree height recursively, *getSize* method, to obtain the number of nodes in the tree recursively, the *visit* method, to display the Student object attribute in that node, and lastly the *preOrder, postOrder, inOrder*, and *levelOrder* methods to display all the data in each node differently. The *inOrder* method uses BTQueue and BTQueueNode classes that help order and display nodes properly.
- AVLTree class, a generic class extending BinaryTree class, containing one static integer attribute (modified orginal AVLTree class), opCount incremented each

time after a comparison is made in the find method. The class contains thirteen methods, i.e., the *insert* method, that adds nodes to the binary tree, a *find* method, transverses through all nodes in the tree comparing them to student object. A *delete* method transvers through all nodes in the tree and deletes a node with a specific student number. A *findMin* returns node with most minimum properties in a tree, and the *removeMin* method removes node with the most minimum properties in the tree. A *height* method displays the height of the object calling it. The *balanceFactor* method displays the difference between the height of the right-hand side subtree and left-hand side subtree of the node calling it. A *fixHeight* method re-calculates the height of a node by comparing heights of left-hand subtree and right-hand subtree, then equating to the highest of the two after adding one to it. A *rotateRight* and *rotateLeft* methods adjusts nodes by shifting them to the right or left in the tree. The *balance* method utilizes the *rotateRight* and *rotateLeft* methods to shift nodes accordingly, after there is an imbalance found by the *balanceFactor* method, when it returns 2, balancing the tree according to appropriate balancing properties. A *treeOrder* method, displays the Students type attribute information on each node in a tree form.

The AccessAVLApp contains four methods. The *main* method calls the *readFile*, *printStudent*, and *printAllStudents*, and display operational counts (find and insert counts) of the BinarySearchTree attribute, avl. The *readFile* method, processes entries, in the form of student number and full name, from a file into a Students type object then place the objects in nodes of a tree. A *printStudent* method that has one parameter, a student number, this method transverses through the created tree and compares all Student type objects in the binary tree data structure to the one in the argument, if a match is found, the full name is displayed, if not “Access denied” is displayed. A *printAllStudents* method, without parameters, displays student number and full name of all objects in the binary tree data structure nodes.

Description of Experimental Setup

The experiment was performed on a desktop workstation with 8 GB RAM and Core i3 Processor.

Trial Assessments

The Program was assessed using known valid, invalid and without student numbers, where the printStudent and printAllStudents methods were invoked, the program produced the output as follows.

+ for known valid student numbers, invoking *printStudent* method.

Student Number	*Output*
WTBASE005	Asemahle Witbooi
MHLLUB026	Lubanzi Mohlala
NGCLES001	Lesedi Ngcobo

+ for known invalid student numbers, invoking *printStudent* method.

Student Number	*Output*
KHDKRD089	Access denied!
JDBFJD564	Access denied!
DHSDLF535	Access denied!

+ when assessed without a student number, *printAllStudents* method is invoked producing the following output of the first and last five lines.

AccessAVLApp
MLLNOA014 Noah Maluleke KHZOMA010 Omaatla Khoza NKNTHA021 Thato Nkuna DMSMEL001 Melokuhle Adams MGLLET011 Lethabo Mogale .... WTBASE005 Asemahle Witbooi WTBBAN012 Banele Witbooi WTBMEL022 Melokuhle Witbooi WTBROR003 Rorisang Witbooi WTBSIY016 Siyabonga Witbooi

Experiment Objective

The goal of this assignment is to test the performance of the AVL Tree, using a real-world application to check if a student is on a pre-approved list for access to campus during the lockdown.

Experiment Execution

Firstly, Instrumentation was performed on the program, a counter variable was positioned in the program where comparison of student numbers was done, when student numbers are compared, the counter variable is incremented.

The experimental procedure was as follows:

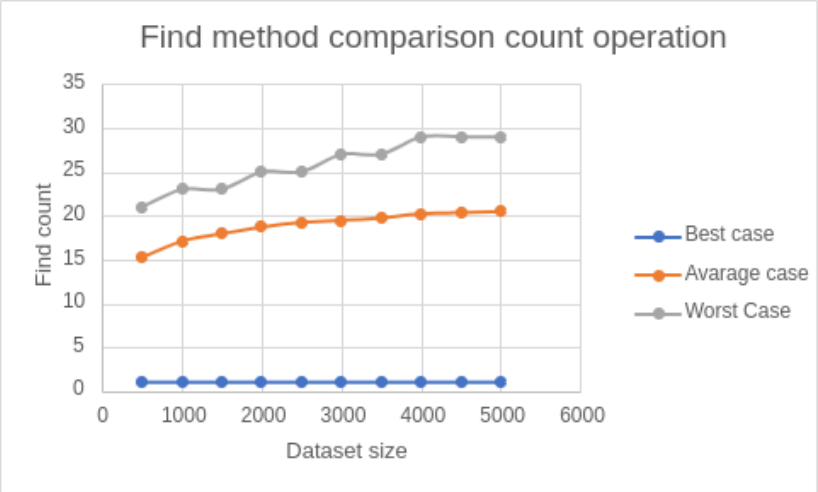
- I created ten equally spaced subsets having entries randomly selected from the oklist.txt file.
- The subsets are 1 to 10 having, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, and 5000 entries that are randomly taken from the original file, oklist.txt.
- Each entry is a student number and a full name.
- For every subset, insert all the subset entries into a the AccessAVLApp program.
- Utilize each entry of that subset by, running the program using a student number in that entry by traversing through a tree of that subset, simply put, searching for that student number against the subset that it is in., i.e., java AccessAVLApp student number, where the student number in a subset is going to be searched in its own subset.
- For a match found in the programs, that entry in that subset is replaced by the full name, the operational find and insert counter integer values of up to that student in the tree, i.e., the number of comparisons the program made before it found that student number in that subset.
- This is performed for all entries in all subsets.

This takes a lot of time to perform, I created a python script that automated the procedure, experiment.py.

Data Analysis

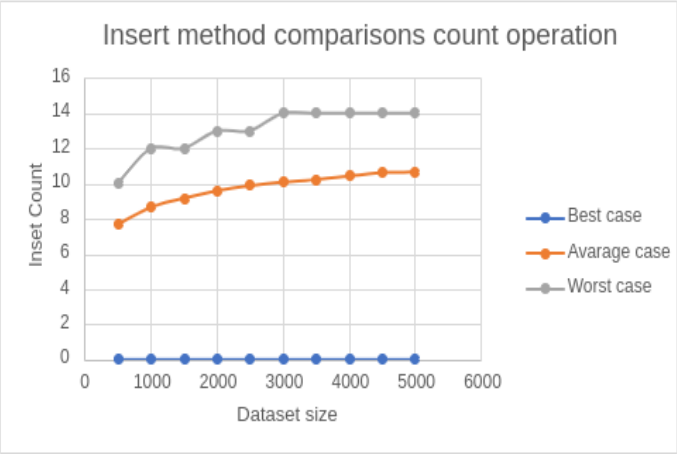
Operation find count.

Dataset size	BEST CASE	AVARAGE	WORST CASE
500	1	15	21
1000	1	17	23
1500	1	18	25
2000	1	19	25
2500	1	19	27
3000	1	20	27
3500	1	20	27
4000	1	20	27
4500	1	20	27
5000	1	20	29



Operation insert count.

DATASET SIZE	BEST CASE	AVARAGE CASE	WORST CASE
500	0	8	10
1000	0	9	12
1500	0	9	12
2000	0	10	13
2500	0	10	13
3000	0	10	14
3500	0	10	14
4000	0	10	14
4500	0	11	14
5000	0	11	14



Discussion of Results.

The theoretical Complexity Analysis of an AVLTree for worst case of operation search, insert and delete is O(log n). The obtained results confirm the theoretical predictions. Both graphs for insertions and search operations are following the logarithmic trend. There is a clear logarithmic rate of increase in operations in run time as the number of datasets get larger, then after a certain large dataset, operations become constant.

Conclusion.

My experiment indicates that one should not worry when searching or retrieving information in big datasets in the program. It can very slightly increase runtime for big datasets, but it will not be noticeable. My experiment supports the use of this Binary Data Structure as its performance is very efficient.

Creativity

I created a python program, checkStudentAccess.py. Students or any user can utilize this to check if they are in the pre-approved list or not, the program communicates directly with the AccessAVLApp program. It gives the student/user 3 options, one, to check student using the student number, two, display all students on pre-approved on list, three, to close the program.

Summary statistics of git usage

```
error@ip-not-found:~/Desktop/CSC2001F/assignment2$ git log | (ln=0; while read l; do echo $ln\: $l; ln=$((ln+1)); done) | (head -10; echo ...; tail -1
0)
0: commit 7443169d5fa7e2ac138b1717362eef6c10be9d92
1: Author: The person who changes files <ThePersonWhoChangesFiles@hahaaha.com>
2: Date: Mon Apr 26 14:49:37 2021 +0000
3:
4: added Makefile
5:
6: commit a2c749c2b829eaaeecb3d4f90323aaff5ccb27cc
7: Author: The person who changes files <ThePersonWhoChangesFiles@hahaaha.com>
8: Date: Mon Apr 26 14:44:08 2021 +0000
9:
...
85: Author: The person who changes files <ThePersonWhoChangesFiles@hahaaha.com>
86: Date: Thu Apr 15 01:58:03 2021 +0000
87:
88: AccessAVLTree - Created Class
89:
90: commit 5523ec1c5bcceb550059df12452b886a262d792f
91: Author: The person who changes files <ThePersonWhoChangesFiles@hahaaha.com>
92: Date: Thu Apr 15 01:56:38 2021 +0000
93:
94: Added All AVLTree classes
error@ip-not-found:~/Desktop/CSC2001F/assignment2$
error@ip-not-found:~/Desktop/CSC2001F/assignment2$
```