ADSA Assignment 1

Name - Anurag Sarva
Roll No. - cs24mtech14003

Problem 0: Generating 100 random arrays each of lengths 10^4, 10^5, 10^6 and 10^7

For this problem, i am writing a code in C, to generate random permutations based on the length and store it in text file. This code generates total 400 text file.

Let's analyse the code first -

1. Random Permutation Generation:
Here i first initialise the array of size 'n', which will stores the number from 1 to n. Then using most popular shuffle algorithm which will shuffles the element of the array, which is "Fisher Yates Algorithm" it makes sure the element of the array is randomly permuted.

2. Saving Arrays to Files:
After generating array it will be stored in the text file. Here i am generating 100 array for each of size 10^4, 10^5, 10^6 and 10^7. So, there are total 400 text files are generated.

3. Main Function: In main function i am using 'srand(time(0))' to make sure that different random sequences are generated for every run. After generating all the arrays and saving all the text files, the program prints a message array generated and saved this will shows program runs successfully.

Let's analyse its running time complexity and space complexity -

Time Complexity:
Fisher Yates Algorithm will take O(n) time shuffle the the element of the array of size 'n'. There are four different array size 'x' and the program will generates 100 arrays for each, so time complexity for generating and storing arrays for a given size is 'O(100 * x)'. We have the size of different array, then the time complexity can be written as-

$$O(100 * (10000 + 100000 + 1000000 + 10000000)) = O(1111000000)$$

Space Complexity:
The space complexity for each array is ‘O(n)‘ where ‘n‘ is the size of array which is based on which size of array we are using. Here in program i am using dynamically memory allocates and deallocates for each array, so the memory usage is manageable even after using for large arrays.

Conclusions -

Fisher Yates Algorithm makes sure that each array is shuffled properly in O(n) time. This will also support for large array sizes.
This program is used for large array generation, still it perform well because of dynamically memory allocation and dellocation.
The approach which i am using here, that scales linearly with respect to both the array size and number of arrays. While it is an overhead for Input/Output for creating and saving of very large number of files, which makes the process slow for very large arrays.
How i can reduce the overhead of input/output operation instead of generating many files, what i can do is, grouping the multiple arrays into a single file. And also paralleling the work of array generation and saving of files.

“DSA” Problem No. 1

In this C code i am implementing Randomized Quick Sort Algorithm.
After implementing this algorithm, i am using this algorithm on the text file i have generated on the previous output to get average number of comparisons and to calculate the standard deviation, then saving the output in text file.
And piloting the average comparing for each different size of array using python library matplotlib.

Let’s analyse the code first -

Global Variables:
I am initialising an global variable that will keeps the record of number of comparison made during the sorting process using an Randomized Quick Sort Algorithm.

Randomized Quick Sort Implementation:
To make the Quick Sort randomised what i did, choosing pivot randomly using ‘rand()‘ function, which will acts as a key element for Randomized Quick Sort to avoiding the worst case on already sorted array.
In my code Partition function make sure that the elements smaller than the pivot element are kept to the left and the elements larger than the pivot are kept to the right i.e. placing the pivot in its correct position is its main priority

and also keep the count of number of comparison made in this process.
And the Randomized Quick Sort function apply Quick Sort recursively on the partitions.

Comparison counting and file reading: One function in my code which is Randomized Quick Sort function which will resets the comparison count for each sorting process of an array. It will makes sure that the comparisons are tracked and stored in variable for each array.
The function 'readArrayFromFile' in my code reads the array data stored in the form of array , it is pre-generated files in problem 0 using random permutations.

Calculation of Average Comparison and Standard Deviation:
For each array size, my code will calculating the mean of comparison across all 100 randomly generated arrays of same size.
My code will also calculates the value of standard deviation of comparisons by using 'customSqrt' function, which will computes the square root.

File Output:
All the results are stored in text file and also printed.

Analysis of the Performance and Result of a code

Randomized Quick Sort Efficiency:
Average time complexity for this is O(n log n), but number of comparisons will varies depending on the randomly selected pivot element and also depends on the arrangement of elements. The randomized pivot selection will some how helps to avoid the worst-case complexity which is O(n^2).

Comparisons:
For all different size of the array, the program will maintains the records of total number of comparisons for every 100 different arrays of the same size. We know that if the larger the size of array, the higher the value of expected number of comparisons, it is due to the increasing size and high complexity of sorting then smaller size array.

Standard Deviation:
Standard deviation will give me the information about insight of how spread out the comparisons are across all the 100 array. Here the low standard deviation, indicates that most of the array requires a similar number of comparison to sort the element, and vica-versa for the large value of standard deviation.

0.1 Conclusions:

Scalability of Randomized Quick Sort Algorithm:
The value of the number of comparison will rises, as the size the array increases.

This is because the Quick Sort has, average time complexity of O(n log n). As we know it is come under the Divide and Conquer approach so it scales well for large input sizes of the array.

Effectiveness of Randomly selecting Pivot element:
By randomly selecting the pivot element, the code will avoid worst case complexity on already sorted or reverse-sorted elements of array.

Performance Analysis of a code:
My code will provides meaningful relationship between array size and sorting complexity, giving it ability to understanding the computational cost of this sorting algorithm.

Potential Improvements:
The sorting process can be speed up by concurrent computation. By adding more measures like minimum and maximum comparison, would help us to give further insights, in best and worst case scenario for each array size.

As you know average compression is stored in text file.

Plotting the bar graph for average number of comparison

I am using Python to plot bar graph of a average number of comparison, by using its famous library, which is matplotlib.pyplot .

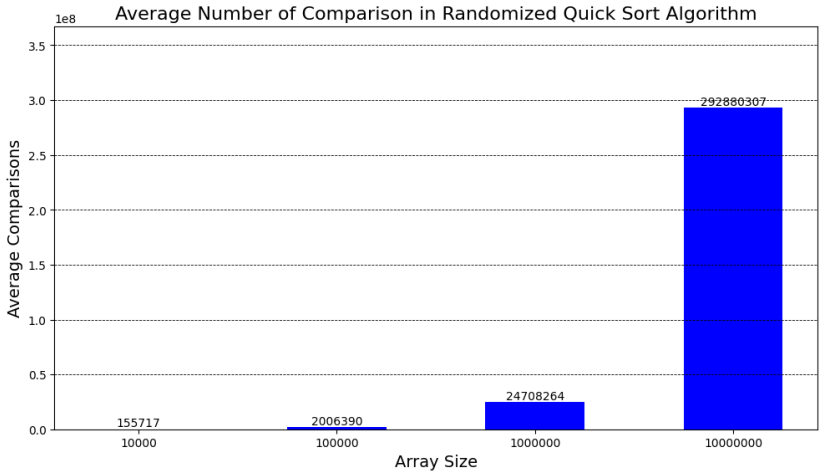


Figure 1: This is a bar graph for average comparison

"DSA" Problem No. 2

(a.) Insertion in AVL and Red-Black tree-

My code is designed to perform insertion into both Red-Black Tree and AVL Tree, during insertion it also tracks the number of rotations performed and the height for each array size. After processing for the all te array size, it computes and stores the average number of rotations and average number of heights for both tree, in separate text files.

Analysis of the Code

Node Structure:
Both AVL and the Red-Black trees are using a common user defined structure for node, it has key, left and right child, height, and color for Red-Black.

Insertion Functions:
AVL Tree:
Uses left rotations and right rotations in a tree to balance the tree on the basis of balance factor(it should be either 1, 0 and -1). And keeping the track of number of rotation perform during insertion for calculating average.
Red-Black Tree:
It maintains a red black properties of a Red-Black Trees by using color property and performs rotations either left and right when necessary. Similar to AVL, it also tracks rotations for average calculation.

Height Calculation:
Tree height is computed based on the maximum depth from root to the leaf node. It helps me to evaluate how much the tree is balanced after a series of insertions.

File Handling:
This will help me to read text files, which contains integer array.

Analysis of Result

Average Rotations:
The AVL Tree are perform more rotations on an average compared to the Red-Black Trees for the same set of data, basically when the input data is randomly arranged. This is due to the AVL Trees is more strict in balancing compared to Red-Black tree because it is a height balanced tree.

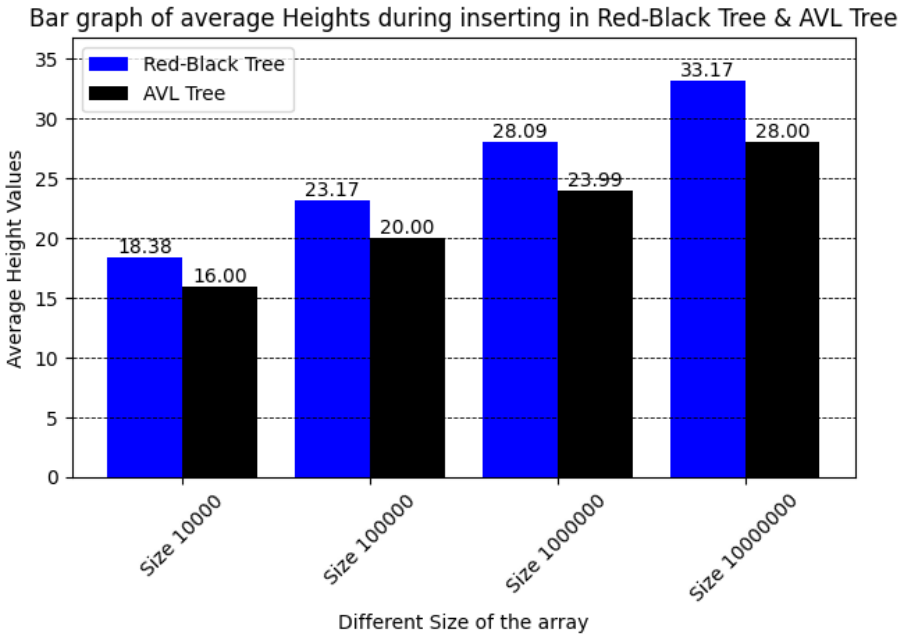
Average Height:
The height of both tree are increases as the input size increases.
AVL Trees may have slightly lower average heights compared to Red-Black Trees this is because of the AVL tree is more height balanced tree compared to the Red-Black tree.

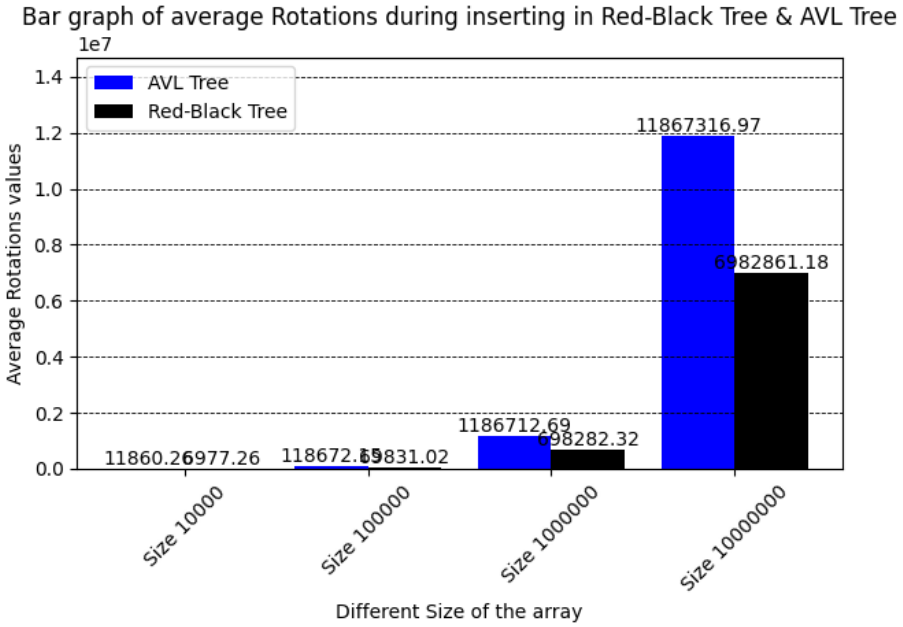
As i mentioned as size of input increase, both tree's height are also increasing, but AVL Tree have a logarithmic height i.e. smaller height than the Red-Black Tree.
Both the trees maintaining the logarithmic height with respect to the number of inserted element, but AVL Tree provides better height compared to the Red-Black tree. And it also depends on the input data, the performance may changes significantly.
I am saving the results in two different text file one for AVL tree and another for Red-Black tree.

Conclusion

My code will perfectly explains the insertion processes in both the trees, AVL Trees and Red-Black Trees.
AVL Trees gives better height (less height) but require more rotation during insertion compared to the Red-Black Trees.

Below is the plot of average rotation and height during insertion in both the tree.





(b.) Deletion in AVL and Red-Black tree-

Analysis of Deletion in AVL and Red-Black tree

Comparison

RB trees require fewer rotations during deletion but it have higher height as compared to AVL tree.
AVL tree requires more rotation while deletion of an element and have less hight compared to red-black tree.

Insertion vs. Deletion-

Both the trees will gives you more rotations, while insetin and deletion, but in AVL tree you will get higher value of average rotation as compared to the Red-Black tree.

Conclusion

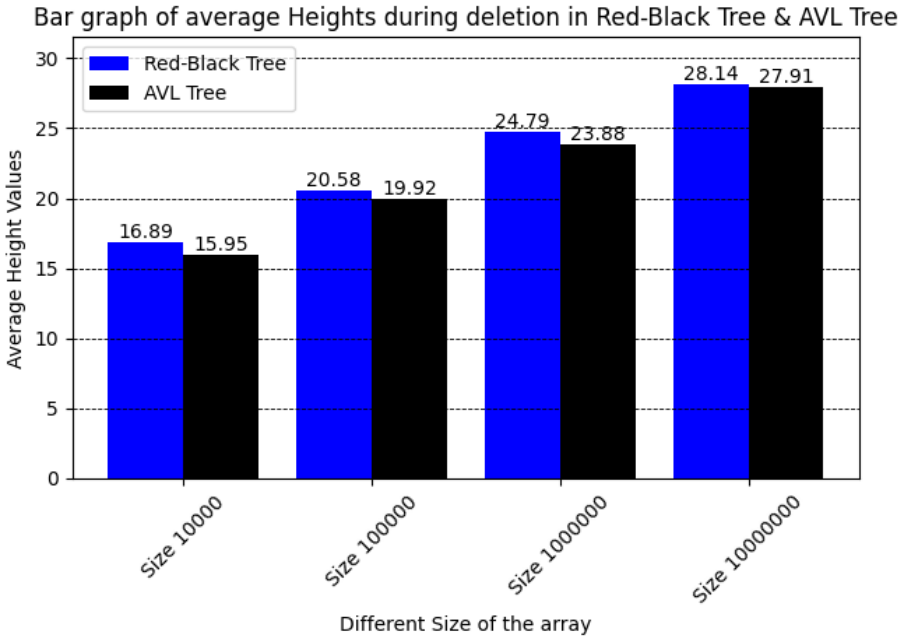
From the results obtained from my code i am come to the conclusion is that-

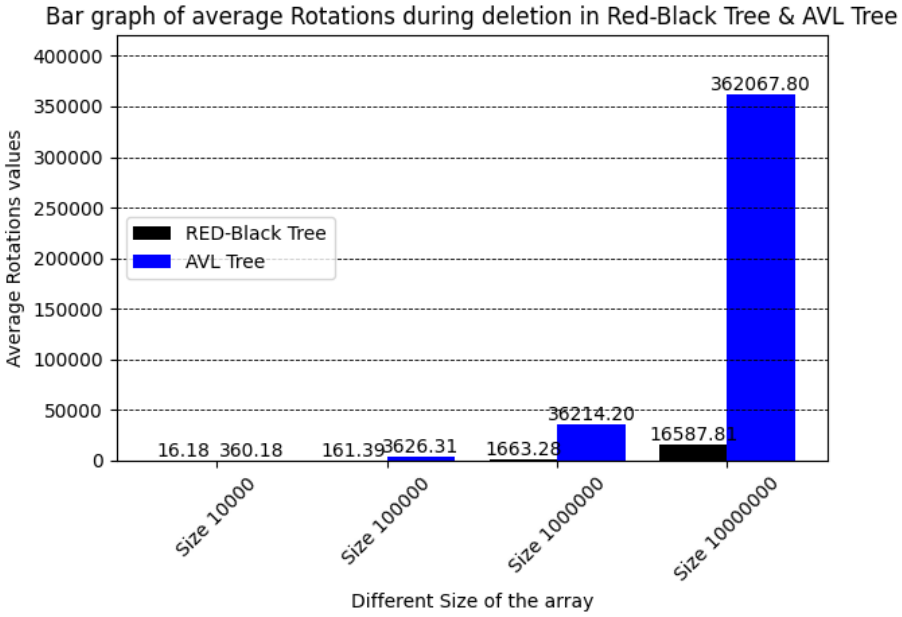
The Red-Black tree will requires very less rotations compared to the AVL tree but it required higher height of the tree for very large number of input.

AVL trees is very much strict to its balanced height that why it required more rotation compared to the red black tree but the height is less.

The choice between AVL and Red-Black to select for the work is totally depends. If the work requires that work whose height is more strict then you can prefer AVL tree and also it has less height. If you want less strict height tree then you can prefer Red-Black tree and also it requires less rotation for both insertion and deletion.

Below is the bar graph which will describe my conclusion-




Bar graph of average Rotations during deletion in Red-Black Tree & AVL Tree
Average Rotations values
400000
350000
300000
250000
200000
150000
100000
50000
0
RED-Black Tree
AVL Tree
16.18
360.18
161.39
3626.31
1663.28
36214.20
16587.81
362067.80
Size 10000
Size 100000
Size 1000000
Size 10000000
Different Size of the array