Birla Institute of Technology & Science, Pilani

Work-Integrated Learning Programmes Division

MTech in Data Science & Engineering

S2_2022-2023, DSECLZG519- Data Structures & Algorithms Design

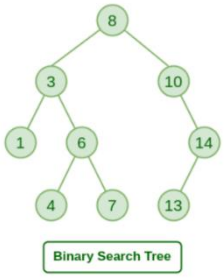
Assignment 2 – PS03 - Excursion management system

Group Number: 03(Group ID: DSAD_Group 03)

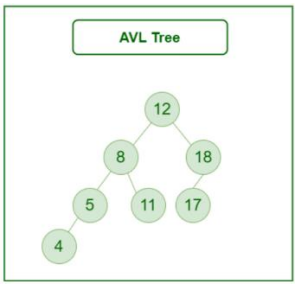
A. Data structure Model with Justifications:

In this employee Excursion management system, we need to efficiently manage employee responses(acceptances and declines),perform insertion,deletions,searches and maintain balance for optimal performance. For this two primary data structures are chosen:

1. *Binary Search Tree (BST):*
   - Justification: A BST is a suitable choice for storing employee numbers. It provides efficient searching, insertion, and deletion operations, which are required in this scenario. It allows for easy traversal, which is useful for printing pre-order traversal.



2. *AVL Tree (Balanced Binary Search Tree):*
   - Justification: An AVL tree is an extension of BST with automatic balancing, ensuring that the tree remains balanced. This provides consistent O(log N) time complexity for insertion, deletion, and searching, making it suitable for maintaining employee responses efficiently over time.



Time Complexity and Space Complexity comparison:

The efficiency of an algorithm depends on two parameters:

a)Time Complexity: Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time taken. It is because the total time took also depends on some external factors like the compiler used, processor's speed, etc.

b)Space Complexity: Space Complexity is the total memory space required by the program for its execution.

Both are calculated as the function of input size(n).

Below table gives detailed time complexity of BST and AVL Trees.

Time Complexity				
	BST -avg case	BST-Worst Case	AVL-avg case	AVL-Worst Case
insertion	O(log N)	O(N)	O(log N)	-
deletion	O(log N)	O(N)	O(log N)	-
searching	O(log N)	O(N)	O(log N)	-
Pre order traversal	O(N) to visit all nodes		O(N) to visit all nodes.	

Below table gives detailed space complexity of BST and AVL Trees also alternative unsorted Array or List.

	Space Complexity	
BST : O(N)	AVL:O(N)	Unsorted Array or List:O(N)
In the worst-case scenario, where the BST becomes completely unbalanced and degenerates into a linked list, the space complexity is O(N) because each node consumes space proportional to the number of employees (N). However, in the average case and when the tree is balanced, the space complexity is still O(N) but with better space utilization. • Space complexity for a balanced BST would be more efficient, but it's not guaranteed in all cases.	In the worst-case scenario, where the BST becomes completely unbalanced and degenerates into a linked list, the space complexity is O(N) because each node consumes space proportional to the number of employees (N). However, in the average case and when the tree is balanced, the space complexity is still O(N) but with better space utilization. • Space complexity for a balanced BST would be more efficient, but it's not guaranteed in all cases.	Unsorted arrays or lists consume space directly proportional to the number of employees (N). Each element is stored in a separate cell, resulting in an overall space complexity of O(N).

Both BST and AVL Tree have the same space complexity, O(N), in the worst case. The space consumed depends on the number of employees.

Unsorted arrays or lists also have a space complexity of O(N) because they store each employee's response in a separate element.

Algorithm to Perform the Given Task:

1. Read the Input Data:
   - Read the input Data from the file inputPS03.txt
   - Extract the initial set of employee numbers at the end of week1, the new acceptance at the end of week2 and the declines.

2. Build binary search Tree(BST):
   - Initialize an empty BST
   - Insert the employee numbers from the end of week1 into the BST
   - Output the pre-order traversal of the constructed BST

3. Build AVL Tree:
   - Initialize an empty AVL Tree
   - Insert the employee numbers from the end of week1 into AVL tree
   - Output the pre=order traversal of the constructed AVL Tree

4. Update Trees with New Acceptance:
   - Insert the new acceptance from the end of week2 into both the BST and AVL Tree.
   - Output the pre-order traversal of the updated BST and AVL Tree

5. Remove Declines from the Trees:
   - Remove the declined employee numbers from both the BST and AVL Tree

- Output the pre-ordered traversal of the updated BST and AVL Tree

6. Randomly select Three volunteers:
   - Generate random numbers to select three employees from the final list
   - Find the coordinates (level and position)of these employees in both trees.
   - Print the coordinates in the specified format

7. Measure Runtime
   - For different input data (varying sizes of employee lists),measure the runtime of search, insertion and deletion operations for both the BST and AVL tree
   - Document conclusions about the performance differences between the two data structures
8. Write the output:
   - Write the results, including tree traversals, volunteer coordinates, and performance comparisons to the file file outputPs03.txt

B. Details of Each Operations with Time Complexity and Reasons:

The table given below give an idea of how time complexity works on BST and AVL with different operations like insertion, Deletion, Searching and pre-order traversal.

Response	BST:	AVL:
Insertion of New Employee	O(log N) on average, O(N) in the worst case if the tree becomes unbalanced. Efficient for average cases but not guaranteed for worst cases.	O(log N), guaranteed. AVL trees maintain balance during insertion, ensuring consistent performance.
Deletion of Employee:	O(log N) on average, O(N) in the worst case if the tree becomes unbalanced. Efficient for average cases but not guaranteed for worst cases.	O(log N), guaranteed. AVL trees maintain balance during deletion, ensuring consistent performance.
Searching for an Employee	O(log N) on average, O(N) in the worst case if the tree becomes unbalanced. Efficient for average cases but not guaranteed for worst cases.	O(log N), guaranteed. AVL trees maintain balance, ensuring consistent O(log N) performance.
Pre-order Traversal (Printing):	O(N). In the worst case (completely unbalanced tree), it may become O(N), but it's efficient for practical purposes.	O(N). AVL trees are balanced, so traversal remains O(N).

To create new methods or improve current processes to meet that higher standard we can use Benchmarking using python API - time.perf_counter() and printed the results using -print(time.perf_counter()-start_time)

it gives most accurate result compared to other APIs - time.time(), time.clock(). time.process_time()

given below are the results :

benchmarking			
Items	operation	BST	AVL
for 10 items	Insertion	0.00019469999824650586	0.0001747999995131977
	Delete	0.00011709999671438709	0.00013610000314656645
	Search	0.00022570000146515667	0.00021279999782564119
for 20 items	Insertion	0.00023769999825162813	0.000283800000033807
	Delete	0.0001250000059371814	0.00016519999917363748
	Search	0.0003546999942045659	0.0003102999980910681

From the above table we can see insert and delete in BST is faster than AVL with increased number of data, because AVL tree has to maintain balance after each insert or delete.

However, Search operation is faster in AVL tree and with more nodes in tree this gap widens, i.e. AVL tree performance increases for search

C. Alternate Way of Modelling the Problem with Cost Implications:

Alternative: Unsorted Array or List

Instead of using binary search trees, an alternative could be to maintain an unsorted array or list to store employee numbers. Each operation would have different time complexities:

Response	Unsorted Array/ List	Cost Implication
Insertion of New Employee Response:	O(1) for appending to the end. This is the most efficient for insertions.	: Insertions are fast, but searching and deletion become inefficient with O(N) time complexity, as you would need to scan the entire list.
Deletion of Employee Response:	: O(N) since you need to search for the element and then remove it.	Deletion becomes slower, especially as the list grows larger.
Searching for an Employee Response:	O(N) since you need to scan the entire list to find the element.	Searching is less efficient, especially for large datasets.

Conclusion: While using an unsorted array or list is efficient for insertions, it becomes highly inefficient for searching and deletion operations. In contrast, the BST and AVL tree structures provide consistent and guaranteed logarithmic time complexities for these operations, making them the better choice for managing employee responses in this scenario. The cost implication of using an unsorted array or list is reduced efficiency in searching and deletion operations, which could lead to poor performance and increased computational costs as the dataset grows.