Problem 5 (Binary trees): Analysis document:

Big O of AVL tree:

Let N(h) represents the minimum number of nodes in an AVL tree of height h. The Recurrence relation for N(h) is:

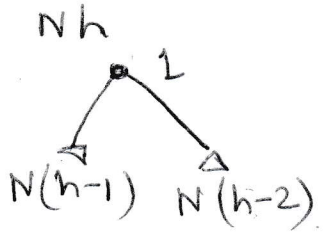
$$N(h) = 1 + N(h-1) + N(h-2)$$

Where, the root contribution 1 node
the left subtree has height (h-1) and the right subtree has height (h-2)

Here, $N(h-1) \geq N(h-2)$

total Nodes,

$$N(h) > 2N(h-2)$$
$$> 2^2 N(h-4)$$
$$> 2^3 N(h-6)$$
$$\vdots$$
$$N(h) > 2^i N(h-2 \cdot i) \rightarrow \text{general formula}$$



The power i can be expressed in terms of h as

$$i = \frac{h}{2} - 1$$

So, $N(h) \geq 2^{\frac{h}{2}-1} \cdot N\left(h-2\left(\frac{h}{2}-1\right)\right)$

$$\Rightarrow N(h) > 2^{h/2-1} \cdot N(2)$$
$$\Rightarrow N(h) \geq 2^{h/2-1}$$

Take log in both sides,

$$\Rightarrow \log(N(h)) \geq h/2 - 1$$
$$\Rightarrow h \leq 2\log(N) + 2$$
$$\Rightarrow h = O(\log(N))$$ The AVL tree is, $O(\log(N))$

Big O for Hash table:

When all n elements hash to the same bucket. The bucket degenerate into a data structure like a Linked list with n elements. For worst case search, insert / deletation requires total n elements traversal:

The load factor, α for hash table,

$$\alpha = \frac{n}{m}$$

Where, n = number of keys
m = number of buckets

In worst case, all n keys fall into 1 bucket, effectively making $m = 1$. Traversing this bucket involves checking n keys.

So total time $T(n) = (cn)$, where c is a constant for processing each element

So, it is $O(n)$

Comparison Between Hashing and AVL Tree

= Hashing offers an average-case search time of O(1), making it extremely fast when there are no collisions. However, in the worst case (when all elements hash to the same bucket), the search time can degrade to O(n).
AVL Trees consistently maintain a search time of O(logn), providing more predictable performance across all cases.

= In hashing, insertion has an average-case time complexity of O(1), but in the worst case (due to collisions), it can take O(n) to traverse a bucket.
AVL Trees require O(logn) for insertions, which may involve additional overhead for rebalancing the tree when needed.

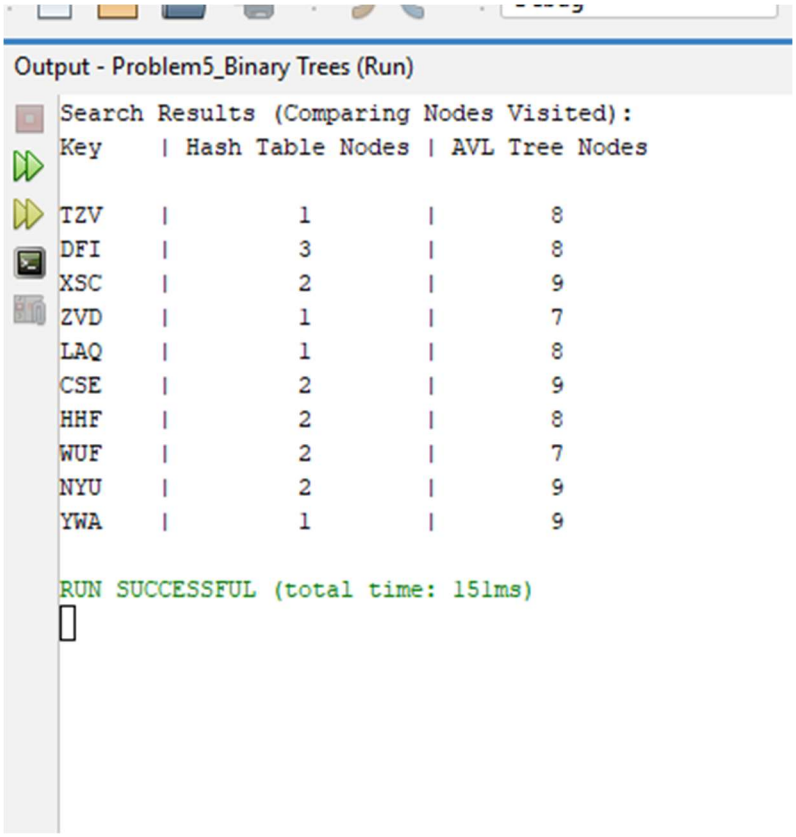
= Hashing can waste space due to unused filled buckets, especially when the hash table is not well-utilized.
AVL Trees use more memory per node to store balancing information, but the memory usage is directly proportional to the size of the data.

= Hashing does not maintain any order among the elements, so retrieving data in a sorted manner is inefficient.
AVL Trees keep the data sorted, allowing in-order traversal in O(n) time, which is useful for ordered data access.

= In hashing, collisions can significantly degrade performance and require additional techniques like chaining or open addressing to manage.
AVL Trees use rotations to maintain a balanced structure, ensuring consistent performance and avoiding degeneration.

= Hashing is ideal for applications requiring fast lookups on unsorted data, such as caches, hash maps, or dictionaries.
AVL Trees are better suited for applications needing sorted data access, range queries, or operations where consistent performance is critical.

Here is the program output:



In the program output observation, the hash table shows better performance compared to AVL Tree.