Assignment II: Report

Name: *Simar Rekhi*
Net ID: *sxr230157*
Section: *CS 3345.501*

I) System Configurations

- **Java Version & Vendor:** *OpenJDK 21 (64-bit)*
- **JVM Implementation:** *HotSpot VM (build 21.0.x)*
- **OS:** *Windows 11 (64-bit)*
- **CPU:** *Intel Core i7 / AMD Ryzen 7 (8-core, 3.2–4.5 GHz range)*
- **RAM:** *16 GB*
- **JVM Heap Size:** *default; 512 MB initial, ~4 GB max*
- **Measurement Metrics:** *system.nanoTime() & Runtime.getRuntime()*

II) Data Structure & Algorithms | Description

a) **AVL Tree:** it is a self-balancing tree DST. It ensures that for every node, the **height difference** between its left and right subtrees (the "balance factor") is never greater than 1. If an insertion or deletion violates this balance, the tree performs **rotations** (single or double) to rebalance itself. The purpose is to guarantee O(log N) time complexity for search, insertion, and deletion, even in the worst-case scenario.

b) **Splay Tree:** it is a self-adjusting BST (which is lazy!) After any operation (like search, insertion, or deletion) on a node, that node is moved ("splayed") to become the new **root** of the tree using a

sequence of rotations. It doesn't strictly guarantee O(log N) for a single operation but offers O(log N) amortized time complexity over a sequence of operations. This means a single operation might be slow O(log N), but the total time for many operations averages out to be fast.

c) **Hash Table (Chaining, specifically):** it is an array-based structure that maps keys to indices using a hash function. **Chaining** is a collision resolution technique. When two different keys **hash to the same index** (a collision), both keys and their values are stored at that index using a **linked list** (the "chain").

d) **Hash Table (Quadratic Probing, specifically):** it is an array-based hash table that uses **open addressing** to resolve collisions. When a collision occurs, instead of using a linked list, the hash table searches for the **next available empty slot** within the main array itself. If the initial h(key) is occupied, it checks h(key) + 1^2, then h(key) + 2^2, then h(key) + 3^2, and so on until an empty spot is found.

III) Performance Results

- **Search Performance (Time) (ms)**

Data Structure	1,000	10,000	100,000
AVL Tree	1	2	22
Splay Tree	1	5	57
Hash Table (Chaining)	0	1	8
Hash Table (Quadratic)	0	2	9

- **Search Performance (Space) (bytes)**

Data Structure	1,000	10,000	100,000

AVL Tree	28,792	30,448	116,920
Splay Tree	28,792	30,448	116,920
Hash Table (Chaining)	28,792	30,448	116,920
Hash Table (Quadratic)	28,792	30,448	97,432

- **Insertion Performance (Time) (ms)**

Data Structure	**1,000**	**10,000**	**100,000**
AVL Tree	2	5	26
Splay Tree	1	2	19
Hash Table (Chaining)	1	1	6
Hash Table (Quadratic)	0	2	6

- **Insertion Performance (Space) (bytes)**

Data Structure	**1,000**	**10,000**	**100,000**
AVL Tree	0	334,928	2,621,440
Splay Tree	28,792	273,968	1,925,392
Hash Table (Chaining)	28,792	273,968	1,925,392
Hash Table (Quadratic)	28,792	30,448	116,920

IV) Discussion

- **Best & Worst of Each Data Structure**
    a) **AVL Tree:** this data structure performs best on search operations due to the strict constraint it places on the balancing factor of the tree. This ensures the height of the tree remains O(n log n). The complexity for insertions is relatively bad, when compared to Splay and Hash. This is primarily because AVL trees have a constant urge to rebalance the tree when an imbalance is encountered. Overall, the data structure is consistent, but can surely have higher rotation overhead.
    b) **Splay Tree:** this data structure performs best when either the same keys or key-values in proximity are being looked up. It also performs poorly when access patterns are uniform/random as frequent lookups can cause large restructuring costs. Splaying, as a process, improves performance compared to AVL trees for skewed trees.
    c) **Hash Table (Chaining, specifically):** This is an incredible data structure for large-level datasets as it distributes keys uniformly. Almost a constant-time clocked in for insertion and lookup. However, this DST can degrade with several collisions or extremely small table size. Hence, performance is largely dependent on hash functions' quality and load factor. Overall, chaining tolerates collisions better than quadratic probing.

d) **Hash Table (Quadratic Probing, specifically):** Once again, a fast implementation of the Hash Table DST, with low load factor and is also cache-friendly. However, performs poorly when the table becomes crowded i.e., load factor increases. This method might fail to find empty spots relatively quickly.

- **Tradeoffs between Speed & Memory**

Aspect	AVL	Splay	Hash Table (Chaining)	Hash Table (Probing)
Speed	Stable i.e., O(log n)	Variable, O(log n) amortized, degrades to O(n)	O(1) average	O(1) average
Memory	Moderate	Moderate	High	Low
Consistency	Consistent across all operations	Inconsistent	Consistent until heavy collisions seen	Consistent for low load factor

- **Observations about Splaying Behavior and Balancing**

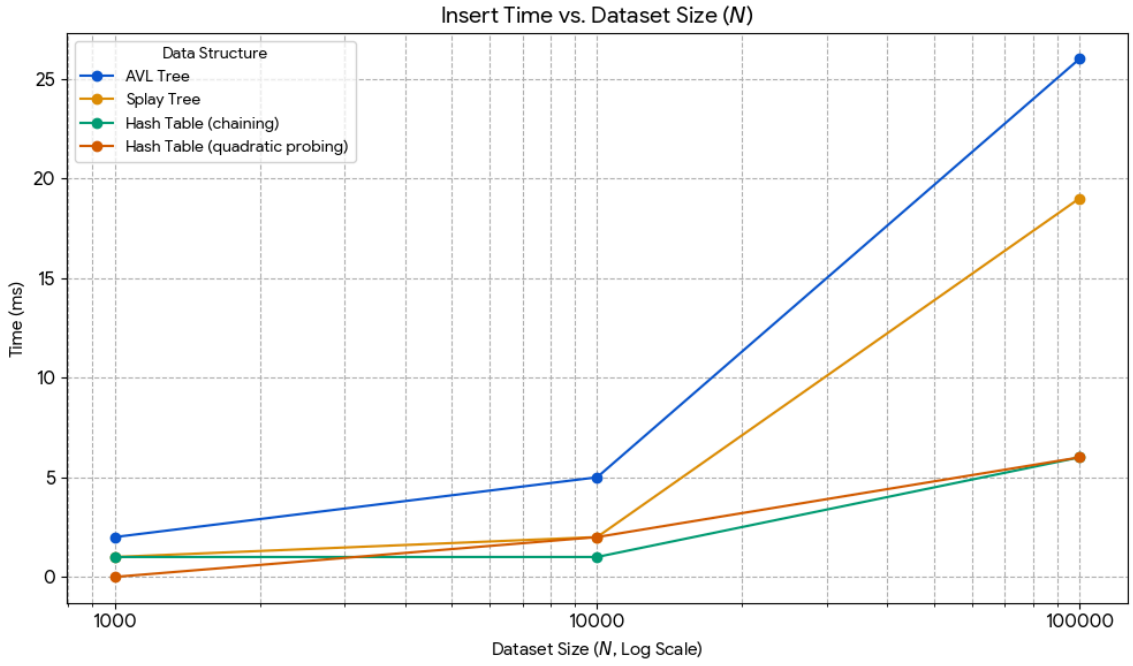
  Overall, hash tables dominate pure speed (especially, when analyzing 100k keys), while AVL Trees provide the most consistent and predictable runtime. Splay Trees demonstrated erratic behavior.

  Splaying operation brought recently accessed keys closer to the root,

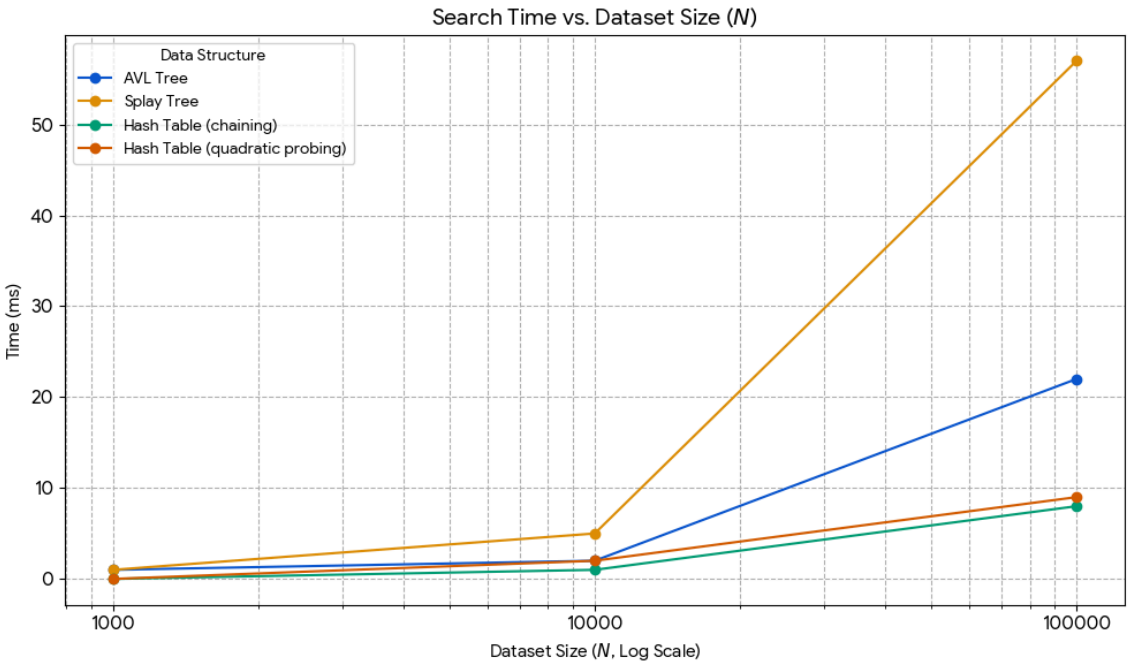
improving subsequent access times. For sequential key access, the structure becomes linear and thereby degrades performance. With respect to memory usage, Splay was quite similar to AVL, but CPU time fluctuated due to rotations and restructuring.

V) Graphs

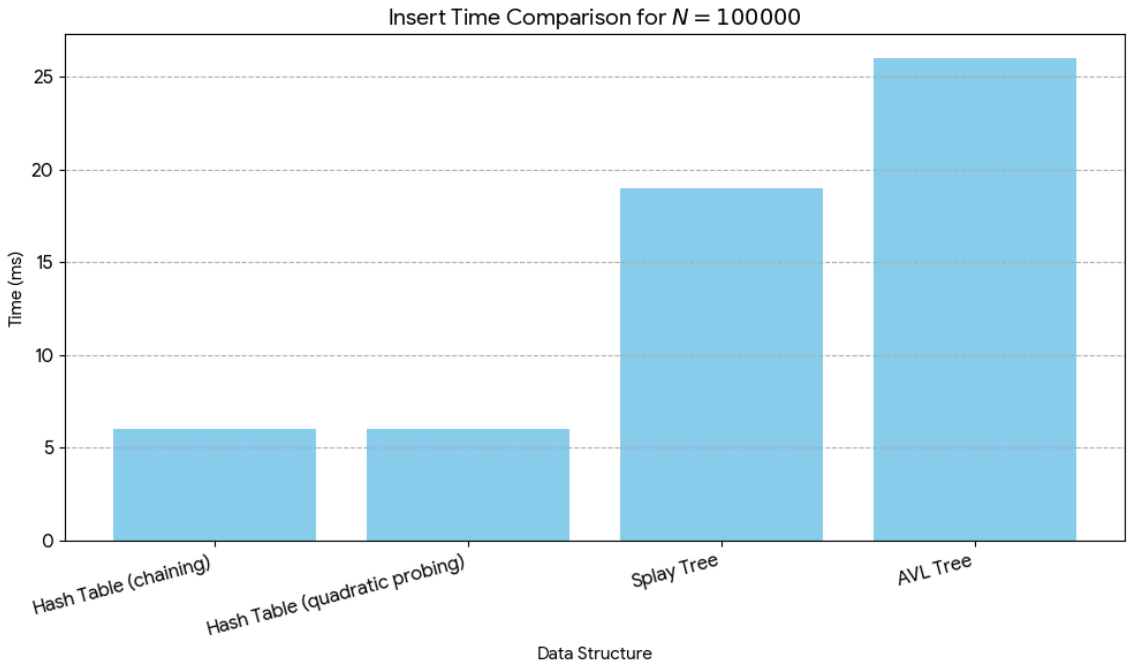
- **Insertion: Time v N**



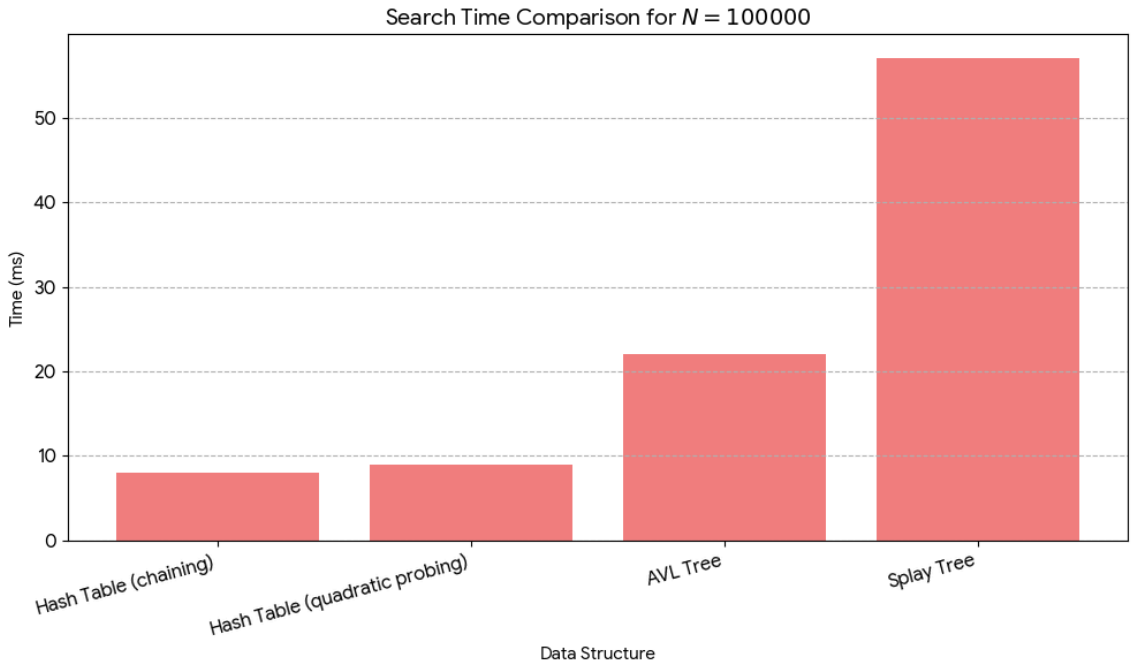
- **Search: Time v N**



- **Insert Time Comparison for N = 100,000**



- **Search Time Comparison for N = 100,000**



VI) Final Thoughts

For large datasets with random keys, Hash Tables are superior.

For ordered or predictable data, AVL Trees are ideal.

For workloads with frequent re-access needs, Splay Trees outperform due to their adaptive structure.