Kristianstad University

SE-291 88 Kristianstad

+46 44-250 30 00

www.hkr.se

**Data Structure**

Seminar3

Ahmed Sabaawi

# TASK 0

# This research paper examines various methods of storing and managing large sets of integers in hash tables, with a focus on the intricacies and efficiencies of array hash tables, open-address hash tables, and different adaptations of cuckoo hashing. It evaluates these structures meticulously by looking at their performance metrics, spatial economy, and cache utilization behaviors. The paper also explores advanced techniques such as dynamic array expansion and clustering that are aimed at optimizing cache efficiency while minimizing space requirements. One notable aspect of this study is its comprehensive comparison of different cuckoo hashing variants that carefully evaluate their implementation details in the context of integer storage. It goes further to cover more complex issues like cache-oblivious data structures, SIMD (Single Instruction, Multiple Data) instructions integration and parallelization aspects within cuckoo hashing domain thus giving a complete picture about these approaches and their performance implications. The results presented in the paper show the inherent trade-offs between different hash table configurations. It shows that linear probing is the fastest in terms of raw speed but has limitations under worst-case performance scenarios.

# TASK 1

-D

The following Table shows the average running time in nanoseconds for building a heap either by insert one element at a time up to N elements or by an existing array of N elements:

| **Input Size** | **One at a Time (ns)** | **All at a Time (ns)** |
| --- | --- | --- |
| 10 | 52,036 | 100,665 |
| 100 | 11,146 | 15,448 |
| 1,000 | 59,118 | 97,714 |
| 10,000 | 336,408 | 594,624 |
| 100,000 | 3,113,915 | 4,608,289 |
| 1,000,000 | 24,983,475 | 38,639,672 |

Table 1, Running time in nanoseconds for building a heap.

The chart below illustrates the difference between both Algorithms:

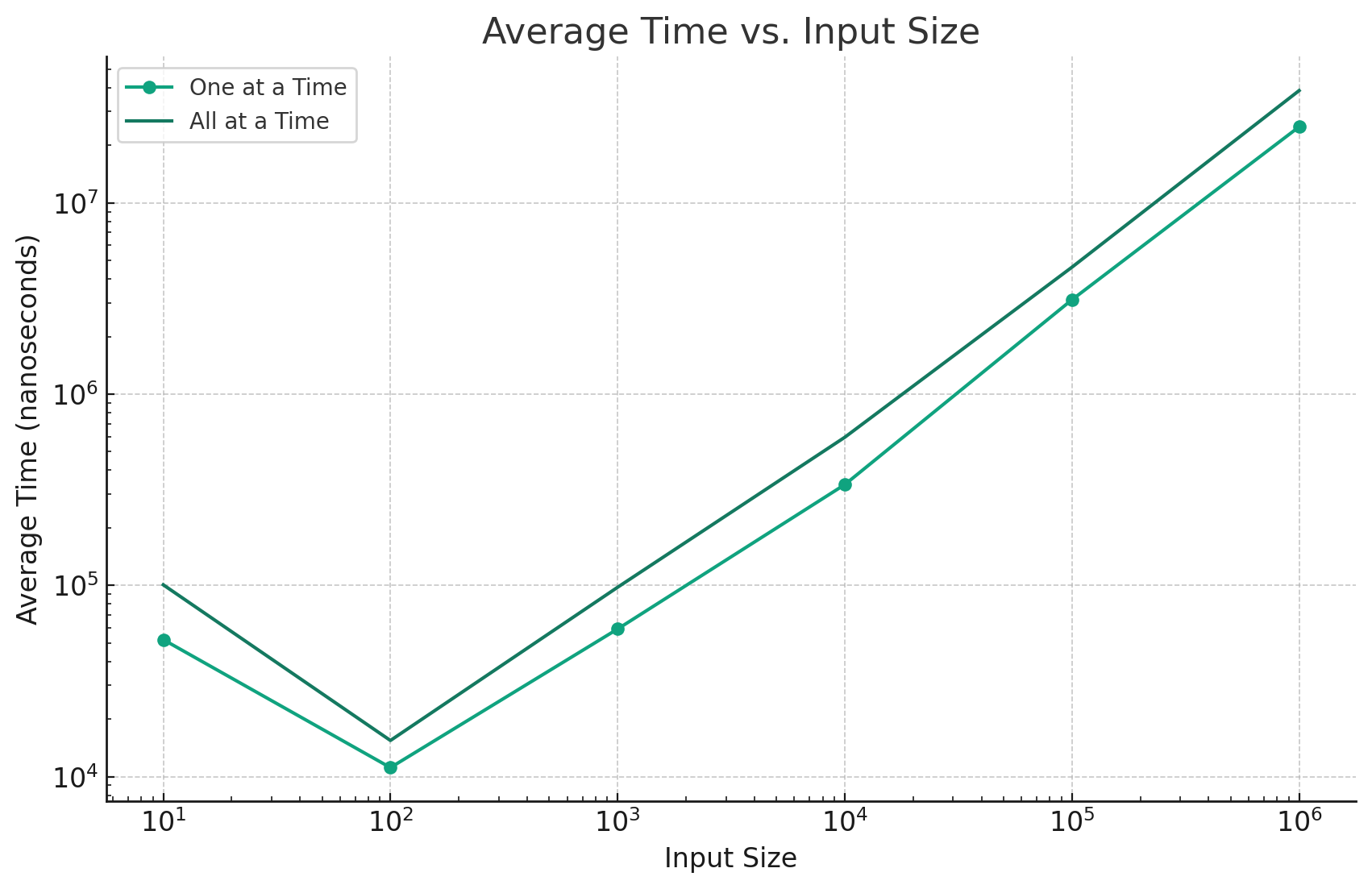


Figure 1, Running time in nanoseconds for building a heap.

-E

**Investigate in priority queues, which is more expensive operation: deleting the minimum element, or the insertion by providing these operations on both trees.**

The table below shows the average running time for inserting and deleting a specific element from an existing tree that has the high N:

| **Input Size** | **Average Time for Insertion (ns)** | **Average Time for Deletion (ns)** |
| --- | --- | --- |
| 10 | 56,332 | 1,299 |
| 100 | 11,710 | 26,805 |
| 1,000 | 57,476 | 109,943 |
| 10,000 | 228,473 | 1,006,340 |
| 100,000 | 2,218,563 | 9,695,712 |
| 1,000,000 | 16,574,684 | 106,104,093 |

Table 2, Insertion, and deletion running time.

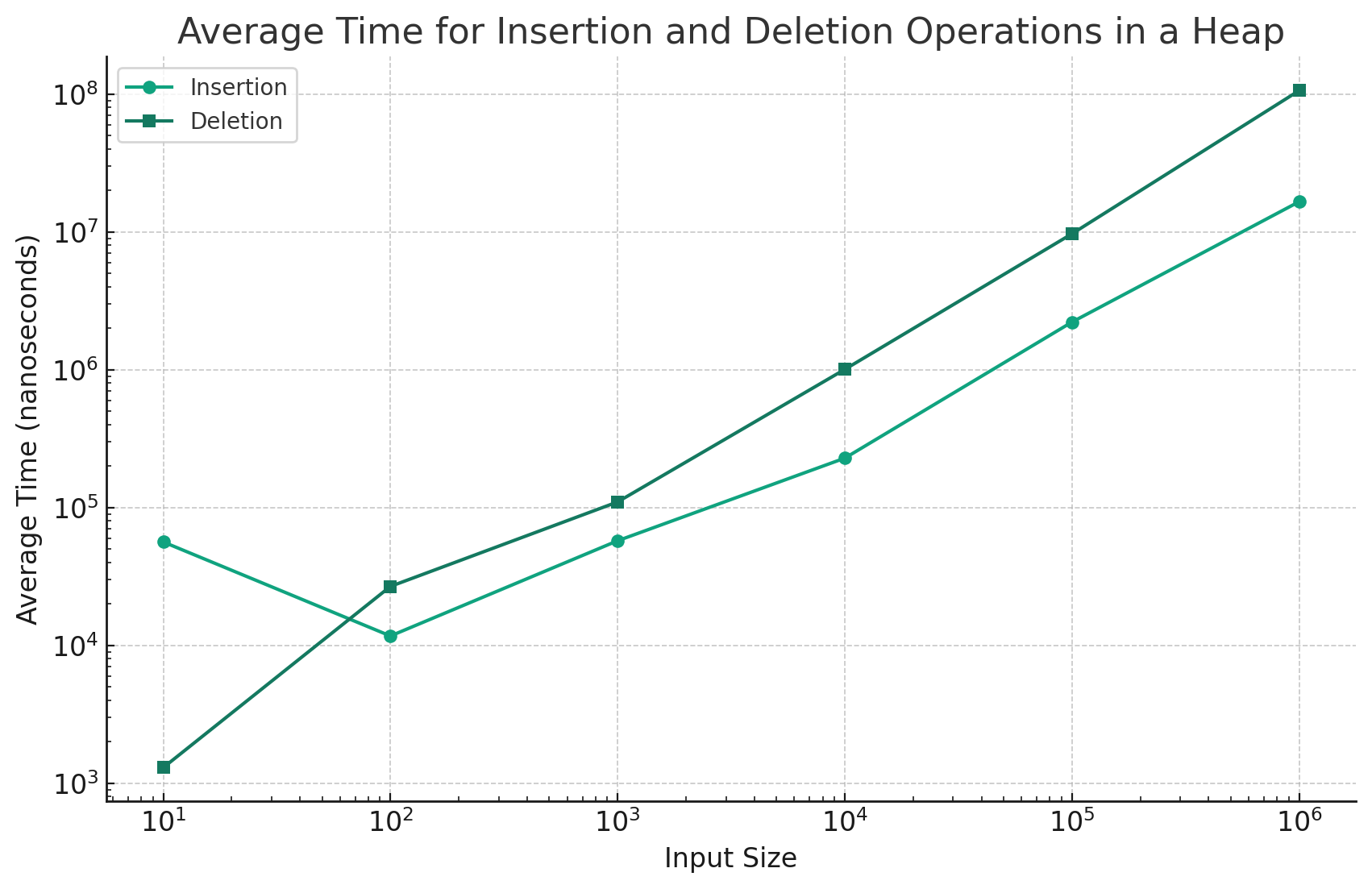


Figure 2, Insertion, and deletion running time.

The graph illustrates these values, showing how the average time for both insertion and deletion operations scales with the increase in input size.

# TASK 2

**d) Complexity Discussion**

1. **Separate Chaining Hash Table**:
   * **Average Case**: O(1 + α), where α is the load factor (number of elements/number of slots). The average case assumes that the hash function distributes keys uniformly.
   * **Worst Case**: O(n) in the situation where all keys hash to the same slot, turning the hash table into a linked list.
2. **Linear Probing Hash Table**:
   * **Average Case**: O(1/(1 - α)) for insertions and successful searches, assuming uniform hashing. The performance deteriorates as the load factor α approaches 1.
   * **Worst Case**: O(n) for insertions and searches if the table becomes full or nearly full, leading to long probe sequences.
3. **Quadratic Probing Hash Table**:
   * **Average Case**: Better than linear probing in terms of handling clustering, but still O(1) under the assumption of uniform hashing and a low load factor.
   * **Worst Case**: O(n) for insertions and searches, similar to linear probing, especially when the table is nearly full.

**f) Other Rehashing Functions**

Rehashing is the process of resizing a hash table and reinserting all elements when the load factor reaches a certain threshold. This is crucial for maintaining the efficiency of hash table operations. Here are some alternative rehashing strategies:

1. **Double Hashing**:
   * A popular method in open addressing schemes.
   * Uses a secondary hash function to calculate the probe sequence.
   * Reduces clustering compared to linear and quadratic probing.
   * The rehash function is of the form: **(hash1(key) + i \* hash2(key)) % table\_size**, where **i** is the ith probe.
2. **Increasing Size by a Prime Number**:
   * Resizing the hash table to a prime number can often lead to a more uniform distribution of hash values.
   * Especially beneficial in open addressing and double hashing.
3. **Universal Hashing**:
   * Involves selecting a hash function at random from a family of hash functions during the rehashing process.
   * This approach is effective in ensuring a more uniform distribution of keys and reducing the likelihood of clustering.
4. **Exponential Growth**:
   * Doubling the size of the hash table is a common strategy.
   * This approach balances between too frequent rehashing and having a too large table.
5. **Load Factor Management**:
   * The decision to rehash can also be based on the load factor, which is the ratio of the number of elements to the table size.
   * A common practice is to rehash when the load factor exceeds 0.7 or 0.75.

## TASK 4

In this task a comparison between the trees, Binary search tree, AVL tree, Red & Black tree, and heap tree, is illustrated. To fairly compare the running time for a specific operation, the insert operation is used, since the delete operation is not fair because of that heap only removes the root, which make it unfair to compare with other trees. All the trees have 71 nodes in the beginning then a node is added to each tree and the running time for the insertion is calculated.

| **Operation** | **AVLTree (ns)** | **BST (ns)** | **MinHeap (ns)** | **RBTree (ns)** |
| --- | --- | --- | --- | --- |
| Insert | 29062 | 3116 | 41787 | 101680 |
| Delete | 7776 | 10913 | 10359 | 12733 |
| Search | 12237 | 10134 | 269210 | 9993 |

Table 3, Insertion, deletion and searching running time.

* **Insert Operation**: The RBTree takes the longest time for insertion, significantly higher than the other structures. This could be due to the complex balancing operations in RBTree. MinHeap and AVLTree follow, with BST being the most efficient. The efficiency of BST in insertion suggests minimal balancing overhead, but this could lead to less optimal search times.
* **Delete Operation**: RBTree and MinHeap show comparable times, with AVLTree being slightly faster. BST stands out for its relatively efficient deletion process. The complexity of deletion in AVLTree and RBTree, including rebalancing, likely contributes to their increased times.
* **Search Operation**: MinHeap shows an exceedingly high search time compared to the tree structures. This is expected, as heaps are not optimized for search operations. AVLTree, RBTree, and BST show more competitive search times, with RBTree being slightly faster than AVLTree and BST. The balanced nature of AVLTree and RBTree usually results in more efficient search operations compared to BST, which can become unbalanced.

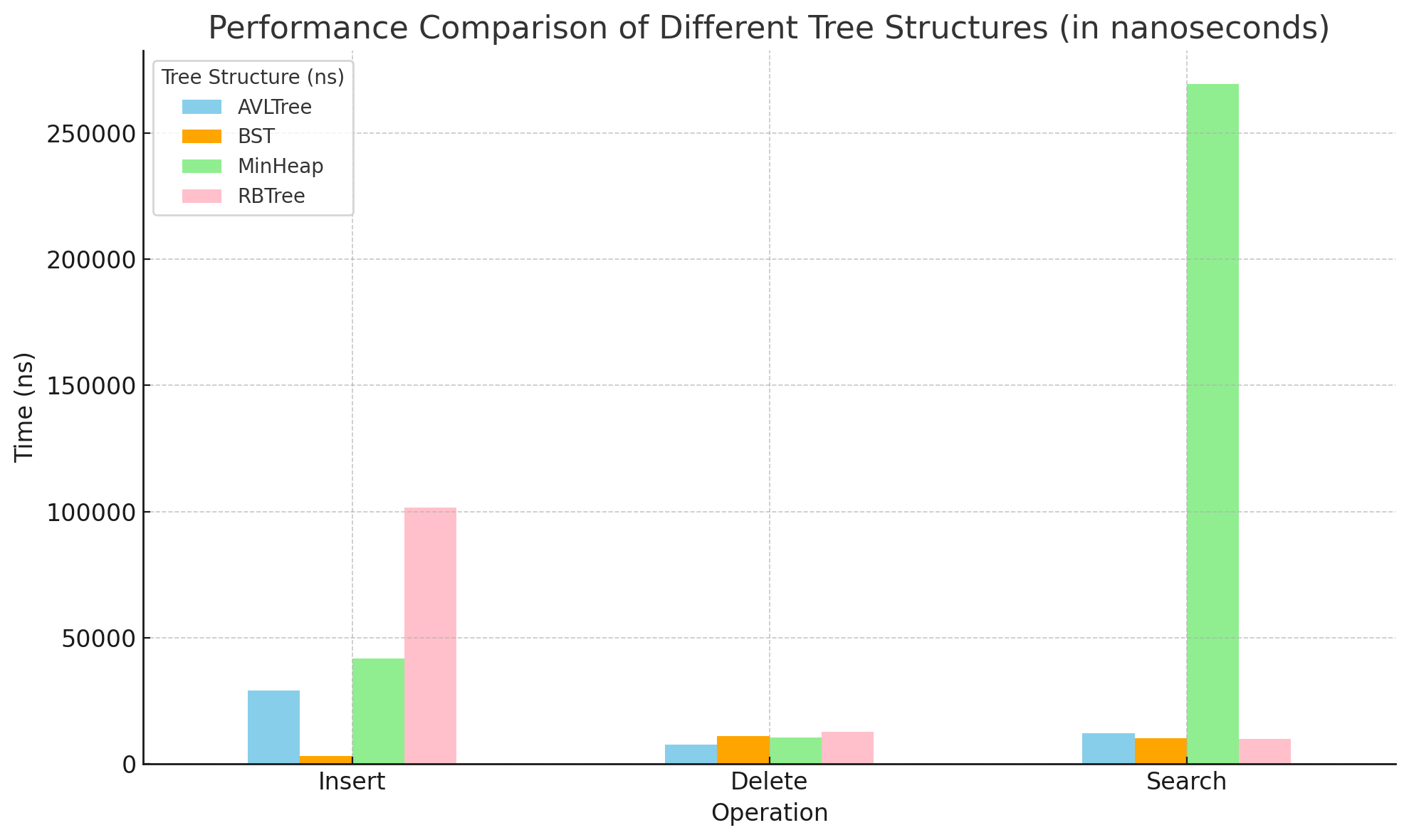


Figure 3, Insertion, deletion and search running time.

**Application of Each Tree Structure:**

1. **Binary Search Tree (BST):** Best suited for efficient searching, especially when the tree structure is relatively static and doesn't undergo frequent modifications. It's extensively used in databases and file systems for quick data retrieval. The BST's simple structure makes it ideal for applications where the data remains mostly unchanged over time.
2. **Heap:** Primarily used in implementing priority queues, which are crucial in task scheduling algorithms. In these scenarios, maintaining the order of elements is paramount. Heaps are particularly effective in applications like heap sort, bandwidth management, and in systems where managing prioritized tasks or data is essential.
3. **AVL Tree:** Designed for situations requiring a balanced tree with efficient search, insert, and delete operations. Its self-balancing nature makes it valuable in databases where the efficiency of search operations is a priority. AVL Trees are used in systems where balanced tree structure is necessary to ensure consistently efficient operations.
4. **Red-Black Tree (RBTree):** Suitable for environments where insertions and deletions are more common than searches, maintaining a balanced structure throughout. Its application is significant in implementing associative containers, as seen in the C++ Standard Template Library (STL). RBTrees are chosen for their ability to provide relatively balanced performance across different operations and are used in complex data structures where the frequency of insertions and deletions is high.