5 different data structures were implemented to organize 10,000 integer values: a linked list, binary search tree, closed hash table with chaining, open hash table with linear probing, and open hash table with quadratic probing. After measuring runtimes for insertion and delete methods, I have concluded that the most efficient data structure for this application was the use of closed hash chaining while the least efficient data structure was the use of a linked list.

The linked list data structure suffered from a linear runtime, with the insertion and search runtimes being a linear function of n (O(n) = n) for both dataset A and B (figures 1A and 1B), which is explained by the observation that the number of node traversals for each insert incremented by 1 for each integer insert due to tail inserts without a maintained tail pointer, while the number of node traversals for each search incremented by ½ for each node insertion.

The binary search tree implementation was logarithmic for dataset A (O(n) = log(n)) (figure 2A) and linear for dataset B (O(n) = n) (figure 2B) for both insert and search methods. While logarithmic runtimes are expected for balanced binary search trees, linear runtimes may be expected for unbalanced trees. Therefore, the linear runtime of the BST for dataset B is explained by the gradually incrementing values in dataset B (figure 8B), which would result in a right-heavy tree. Dataset A, however, had random values, which resulted in a logarithmic runtime for the BST.

The runtime for the closed hashing with chaining implementation was generally constant (O(n) = 1) for insertion and search times for both datasets (figures 3A and 3B); the average insert runtime did not surpass 180 ns, which was superior to the maximum average runtime for all other data structures. The slight linearity of this structure can be explained by the linearity of the number of collisions; as the data table became more filled, collisions increased, but the number of node traversals remained relatively low compared to the BST and LL; the low runtime was also due to the constant runtime (O(1)) of indexing into an array.

The runtime for the open hashing with linear probing (figures 4A and 4B) and quadratic probing (figures 5A and 5B) were mostly constant (O(1)) then became exponential for insertion (O(n) = n^x); the insert runtimes remained quite low until around iteration 95, then suddenly and dramatically began to spike. This is explained by the difference of the average number of probes for “hit” with open hashing. The average number of probes for a hit = (1/2) \* (1 + 1 / (1 – n/m)), where n is the number of elements and m is the size of the array. With an array size of 10009 and n of 9500, the average number of probes would be 5.5; however, with an n of 10000, this number increases to 557: this would explain the sudden increase in runtime for both open hashing insert implementations.

The most efficient insert method was found to be closed hash chaining (figures 6A and 6B), likely due to the constant runtime of indexing into an array and without the exponential increase in the number of probes per insert seen around iteration 95 in open hashing insertion. However, the most efficient search method was found to be quadratic probing (figures 6A and 7B), which outperformed linear probing due to its avoidance of primary clustering. The exponential increase in inserting with open hashing, however, forces me to conclude that the most efficient data structure overall must be closed hashing with chaining.

**Linked List Plots**

*Figure 1A: Linked List Average Search and Insertion Times for Dataset A*

*Figure 1B: Linked List Average Search and Insertion Times for Dataset B*

**Binary Search Tree Plots**

*Figure 2A: Binary Search Tree Average Search and Insertion Times for Dataset A*

*Figure 2B: Binary Search Tree Average Search and Insertion Times for Dataset B*

**Hash Chaining Plots**

*Figure 3A: Hash Chaining Average Search and Insertion Times and Collisions for Dataset A*

*Figure 3B: Hash Chaining Average Search and Insertion Times and Collisions for Dataset B*

**Hash Linear Plots**

*Figure 4A: Hash Linear Average Search and Insertion Times and Collisions for Dataset A*

*Figure 4B: Hash Linear Average Search and Insertion Times and Collisions for Dataset B*

**Hash Quadratic Plots**

*Figure 5A: Hash Quadratic Average Search and Insertion Times and Collisions for Dataset A*

*Figure 5B: Hash Quadratic Average Search and Insertion Times and Collisions for Dataset B*

**Summary Insert Plots: Linked List, BST, Hash Chain**

*Figure 6A: Summary of Average Insert Times for Linked List, Binary Search Tree, and Hash Chaining for Dataset A*

*NOTE: LL IS GIVEN IN MICROSECONDS*

*Figure 6B: Summary of Average Insert Times for Linked List, Binary Search Tree, and Hash Chaining for Dataset B*

*NOTE: LL IS GIVEN IN MICROSECONDS*

**Summary Search Plots: Linked List, BST, Hash Quadratic**

*Figure 7A: Summary of Average Search Times for Linked List, Binary Search Tree, and Hash Quadratic for Dataset A*

*NOTE: LL IS GIVEN IN MICROSECONDS*

*Figure 7B: Summary of Average Search Times for Linked List, Binary Search Tree, and Hash Quadratic for Dataset B*

*NOTE: LL IS GIVEN IN MICROSECONDS*

**Data Plots**

*Figure 8A: Data Plot of Dataset A*

*Figure 8B: Data Plot of Dataset B*