Parker Tennier

Jordan Corn

Data & Algorithm Analysis Report

1. For each ADT, which operation is the most efficient? Which is the slowest? Why?

   Unsorted Vector: The insert is most efficient, the lookup and remove functions are equally the slowest. Insert is O(1) because it inserts the key to the back of the vector without extra operations. Lookup and Remove must traverse the vector until it finds the key it needs to look up, or remove. The larger scale vector will make Lookup and Remove extremely inefficient.

   Sorted Vector: Lookup on a sorted vector is the most efficient and insert is the slowest. Lookup uses a binary search method which makes the search time O(log n). Remove may take slightly longer than Lookup but it still uses the binary search method to find the key value needed to be removed.

   Unsorted Linked List: Insert is the fastest and most efficient function and lookup and remove are equally the slowest. Insert just pushes the new node back and sets the head to the new node. Lookup and Remove both must traverse the linked list next values until they find the node they are searching for to do operations on.

   Sorted Linked List: The lookup and remove functions are equally as efficient and the Insert function is the slowest. The Lookup and Remove functions still require the algorithm to search from the head but they are faster than the Insert function since that requires searching for the correct place and inserting the node in between two nodes.

2. Compare the *insert* operation in an unsorted vector vs. a sorted vector. What difference do you observe, and how does it match the theoretical analysis?

   The unsorted vector is extremely faster at inserting a key value than the sorted vector. Since the sorted vector has to traverse to the correct location necessary

to place the key value. The unsorted vector is O(1) while the sorted vector is a linear search which is O(n)

3. Compare the *lookup* operation in a sorted vector vs. a sorted linked list. Are the results different and if so, why?

   The lookup operation for the sorted vector and sorted linked list are almost identical because they both require the program to start from the beginning and compare it to the key value it needs to find, making it less efficient and running at O(n).

4. Look at the *remove* operation across all four ADTs. What patterns do you see?

   The patterns in the remove function of all four ADT's mirror the lookup functions almost identically. They require the algorithm to first traverse the data type and then run another algorithm to remove the value in the correct manner, especially if it is a sorted data type.

5. Does remove behave more like insert, lookup, or some combination?

   The remove function behaves more like Lookup because both algorithms need to search through the data type in order to find the location or key value it needs to further run.

6. Compare the results for *N = 5,000 and N = 50,000*. Which operations show the steepest growth as N increases? Which ones grow the least? How do these observations connect to their theoretical run-times?

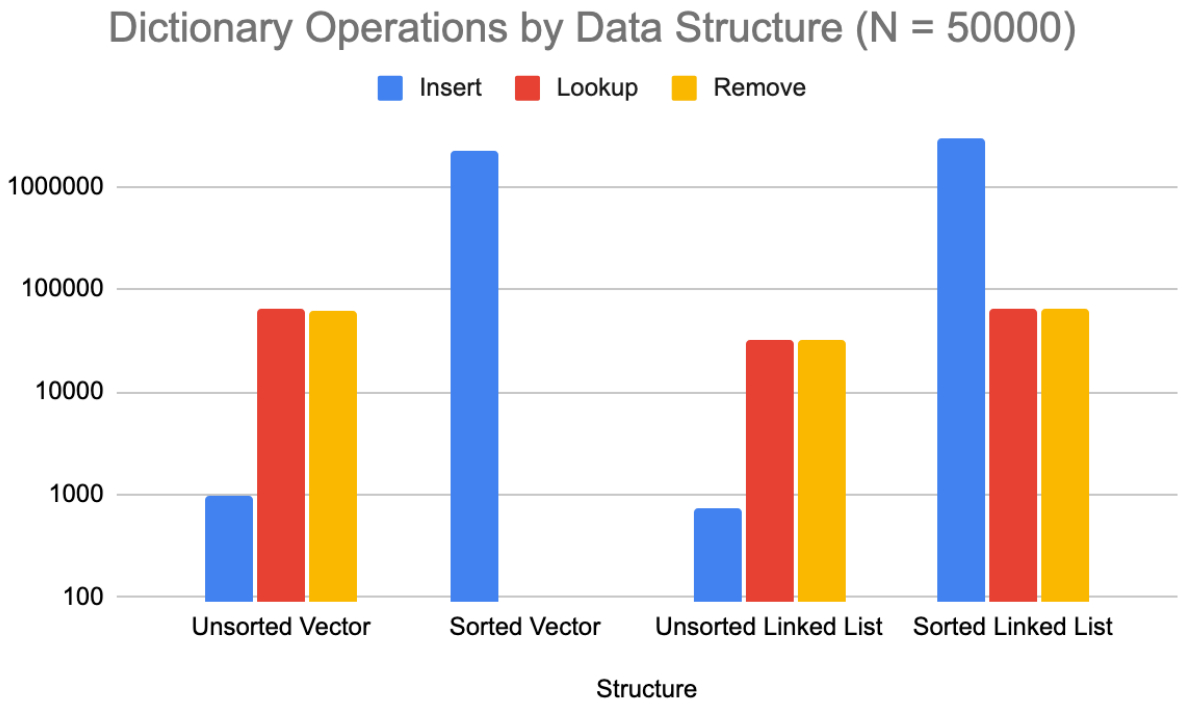
   The operation that shows the steepest growth as N increases would be Insert. The slower growths would be Lookup and Remove almost equally. For both sorted data types, insert requires a lot more numbers to traverse, making it slower and slower as N grows. While lookup and remove already are traversing the entire vector and linked list from the early N values.

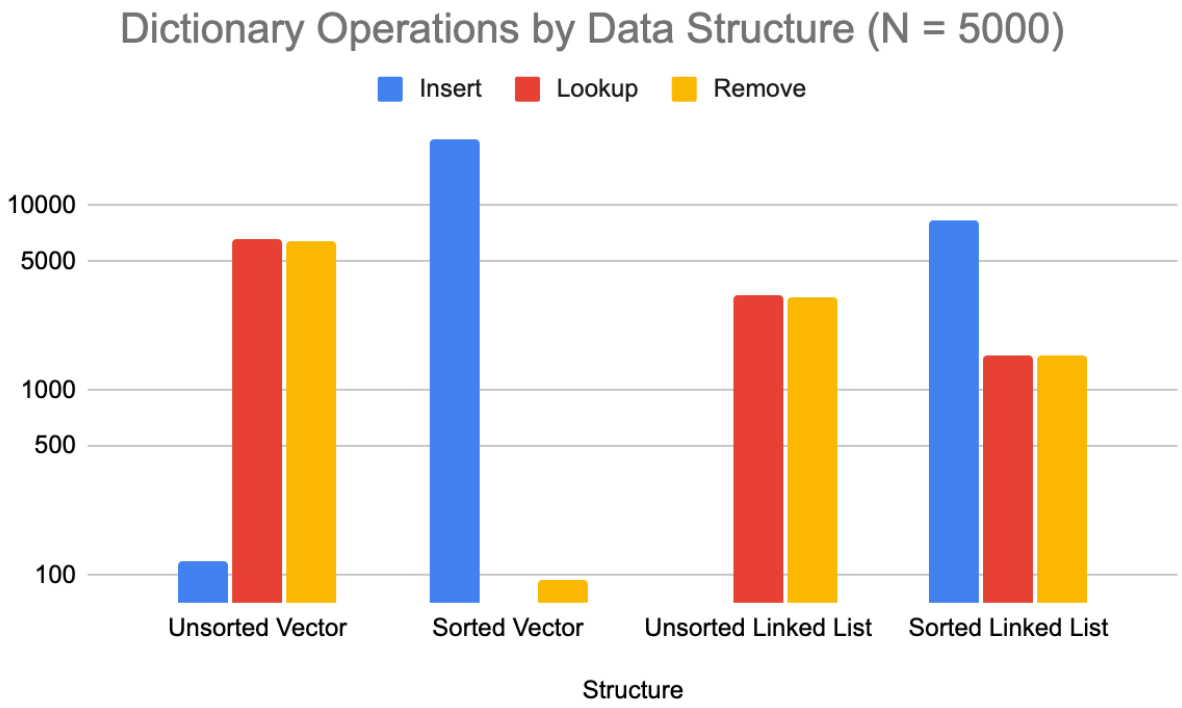
7. Based on your charts, if you were designing a system where *lookups are frequent but inserts are rare*, which ADT would you choose? What if the reverse were true?

For lookups being frequent and inserts being rare, the sorted vector has the most efficient time for Lookups, being almost instantaneous with O(log n). If it was in reverse where inserts were more common, I would use the unsorted vector.

8. *Reflection*: What surprised you most in the measurements compared to what you expected from theory? Did the charts help you see patterns that would have been harder to notice from raw numbers alone?

   What surprised me most was the time it took for the program to run the Sorted Linked List at high volume. If it were in the millions of data values needed, the sorted linked list would take hours to run the dictionary operations.The charts helped very much in seeing patterns within the data types. They can be looked at visually and almost instantly you can tell the efficiencies from each data type as well as the slowest. With raw numbers the calculations and seeing this would take much longer.


Dictionary Operations by Data Structure (N = 50000)
Insert
Lookup
Remove
1000000
100000
10000
1000
100
Unsorted Vector
Sorted Vector
Unsorted Linked List
Sorted Linked List
Structure


Dictionary Operations by Data Structure (N = 5000)
Insert
Lookup
Remove
10000
5000
1000
500
100
Unsorted Vector
Sorted Vector
Unsorted Linked List
Sorted Linked List
Structure

Runtime Analysis Problems

Problem 1: Intersection of Two Vectors

You are given two vectors of numbers and asked to find the elements they have in common (their intersection). Analyze the run-time under the following assumptions. Use **n** for the number of elements in one vector and **m** for the other vector.

1. Neither vector is sorted.

   Pseudocode:

   COMPARE-UNSORTED(A, B)

   Start at the first element of A

   For each element in vector B, one at a time

   Compare element A to B

   If element A is equal to element B, report element A is an intersection.

   If you reach the end of B, move to second element in A and repeat

   Runtime: O(nm), because it uses a nested loop that must search multiple times through each vector.

2. One vector is sorted, the other is unsorted.

   Pseudocode:

   COMPARE-UNSORTED-TO-SORTED(A, B)

   For each element of unsorted vector A

   Binary search B for element A

   If element found in vector B , report found

   If not found, report not found

   Runtime: O(n Log(m)) because it uses a binary search on the sorted vector to find each value from the unsorted vector.

3. Both vectors are sorted.

   Pseudocode:

   COMPARE-SORTED(A, B)

   For each element of sorted vector A

   Compare to first element in vector B

   If element A equals B, report found and increment A and B.

   If element A is less than B, report not found and increment A

   If element A is greater than B, report not found and increment B

   If A OR B reaches end, stop program

   Runtime: O(n+m) because you never have to check an element more than once.

   Problem 2: Element Uniqueness

   You are given a single vector of numbers (**n** elements) and asked to determine whether all elements are distinct. Analyze the run-time of different strategies for solving this problem.

   Pseudocode:

   ELEMENT-UNIQUENESS(A, B)

   For each element in vector A

   Compare element A to B

   If element A is equal to B, report NOT UNIQUE

   If element A is not equal to B, increment B

   Increment to next value of A

   If no elements are equal, report UNIQUE

   Runtime: O(nm) because worst case scenario it would need to check each element against each other.

Problem 3: Membership Test

You are given a vector of **n** numbers (the “data set”) and another vector of **q** numbers (the “queries”). For each query, determine whether it appears in the data set. Analyze the run-time under the following assumptions:

1. The data set is unsorted and the queries are unsorted.

   Pseudocode:

   LOOKUP-UNSORTED(Q, N)

   For each q in Q (query)

   Compare q to element in vector n

   If q equals element in n, report FOUND and return

   If q is not found, report not found, and move to next element in n

   If you reach end, stop

   Runtime: O(nm), because it uses a nested loop that must search multiple times through each vector.

2. The data set is sorted and the queries are unsorted.

   Pseudocode:

   LOOKUP-SORTED-TO-UNSORTED(Q, N)

   For each element of unsorted vector N

   Binary search Q for q queries

   If element found in vector Q, report found

   If not found, report not found

   If you reach end of vector Q, stop

   Runtime: O(n Log(m)) because it uses a binary search on the sorted vector to find each value from the unsorted vector.

3. Both the data set and the queries are sorted.

   LOOKUP-SORTED(Q, N)

   For each element of sorted vector Q

   Compare to first element in vector N

   If element Q equals N, report found and increment Q and N.

   If element Q is less than N, report not found and increment Q

   If element Q is greater than N, report not found and increment N

   If Q OR N reaches end, stop program

   Runtime: O(n+m) because you never have to check an element more than once.