Lasupe Xiong

CS-300

Instructor Nathan Lebel

03/02/2025

Module Eight Journal

1. Binary Search Tree (BST)

Time Complexity Analysis:

* Insertion:  
  The height of the tree affects how long it takes to add a node to a binary search tree. The height may be O(n), where n is the number of nodes, in the worst scenario (unbalanced tree). Therefore, in the worst scenario, insertion will take O(n) time. On the other hand, it will take O(log n) time if the tree is balanced.
* Search:  
  In a similar vein, finding a node will require O(h) time, where h is the tree's height. If h is balanced, it will be O(log n); if it is unbalanced, it may be O(n).
* Deletion:  
  Because the node must be located and maybe rebalanced, deletion also requires O(h) time. Similar to insertion, this is O(log n) for a balanced tree and O(n) in the worst scenario.

Space Complexity:

* Because each node in a binary search tree holds a bid and its pointers (left and right), the space complexity of the tree is O(n), where n is the number of nodes.

Recursive calls on the stack during operations like insert, delete, and search also have O(h) space complexity, where h is the tree's height. This recursive space complexity may reach O(n) in the worst scenario, as the height of an imbalanced tree may be O(n).

Summary for BinarySearchTree:

* Time Complexity:
  + Insert: O(n) (worst case)
  + Search: O(n) (worst case)
  + Delete: O(n) (worst case)
* Space Complexity: O(n)

2. Hash Table

Time Complexity Analysis:

* Insertion:  
  Insertion involves placing the element in the appropriate bucket after calculating the hash for the given key. The time complexity is O(1) when there are no collisions. The time complexity may be O(n) in the worst scenario (with numerous collisions), where n is the number of table entries (assuming that every element hashes to the same bucket). Nonetheless, the average time complexity stays near O(1) when appropriate hashing and resizing methods are used.
* Search:  
  In a similar way, searching entails accessing the corresponding bucket and determining the hash value. The optimal time to search is O(1). In the worst scenario, searching takes O(n) if a large number of elements hash to the same bucket.
* Deletion:  
  Similar to searching, deletion involves first locating the item and then removing it out. It takes O(1) in the best scenario. In the worst scenario, where the hash function distribution is poor, it takes O(n).

Space Complexity:

* A hash table's space complexity is O(n), where n is the number of its entries. This is due to the fact that every element must be kept in a bucket, and the buckets themselves require an array. Since each bucket can hold many pieces in collision management scenarios (such as chaining), the overall size is O(n).

Summary for HashTable:

* Time Complexity:
  + Insert: O(1) (average case), O(n) (worst case)
  + Search: O(1) (average case), O(n) (worst case)
  + Delete: O(1) (average case), O(n) (worst case)
* Space Complexity: O(n)

3. Vector Sorting

Time Complexity Analysis:

* Insertion:  
  In most cases, inserting an element into a vector takes O(1) (amortized time); nevertheless, if the vector needs to reside (when it reaches its maximum size), it may take O(n).
* Sorting:  
  Usually, an algorithm like QuickSort, MergeSort, or HeapSort is used for the sorting process. If the pivot selection is subpar, the sorting time complexity is often O(n log n) in the best and average instances, and O(n²) for QuickSort in the worst case. In any situation, MergeSort ensures O(n log n).

Space Complexity:

* When n is the number of items in a vector, the space complexity for storing the vector is O(n). Temporary arrays will need O(n) more space if an external sorting method, such as MergeSort, is employed. The space complexity for the recursive stack may be O(log n) if in-place sorting is employed, such as QuickSort.

Summary for VectorSorting:

* Time Complexity:
  + Insert: O(1) (amortized)
  + Sort: O(n log n) (average and best case), O(n²) (worst case for QuickSort)
* Space Complexity: O(n)

4. Linked List

Time Complexity Analysis:

* Insertion:  
  It takes O(1) time to insert a node at the head or tail of a linked list. However, it takes O(n) time to traverse the list in order to insert at an arbitrary point.
* Search:  
  It takes O(n) time in the worst situation to search for an element in a linked list since you have to go from the head to the desired node.
* Deletion:  
  Additionally, deletion necessitates first locating the node. The deletion takes O(1) if you know the node (or whether it's the head or tail). Otherwise, finding the node first takes O(n).

Space Complexity:

* A linked list's space complexity is O(n), where n is the number of nodes. A bid and a reference to the next node are stored in each node.

Summary for LinkedList:

* Time Complexity:
  + Insert: O(1) (if at head or tail), O(n) (arbitrary position)
  + Search: O(n)
  + Delete: O(1) (if node is known), O(n) (if locating node first)
* Space Complexity: O(n)

Summary of the Four Data Structures:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Structure | Time Complexity (Insert) | Time Complexity (Search) | Time Complexity (Delete) | Space Complexity |
| Binary Search Tree | O(n) (worst case) | O(n) (worst case) | O(n) (worst case) | O(n) |
| Hash Table | O(1) (average case) | O(1) (average case) | O(1) (average case) | O(n) |
| Vector Sorting | O(1) (amortized) | O(n) | O(n log n) (sorting) | O(n) |
| Linked List | O(1) (head/tail) | O(n) | O(n) (if locating node) | O(n) |

Key Observations:

* Hash Table offers the best time complexity for insert, search, and delete (O(1) on average), but performance can degrade to O(n) with poor hash function or many collisions.
* Binary Search Tree provides O(log n) time for balanced trees but can degrade to O(n) if unbalanced.
* Vector Sorting performs well with sorting (O(n log n)), but insertion is not as efficient for large datasets.
* Linked List has efficient insertions and deletions at the head or tail but is slow for arbitrary positions (O(n)).