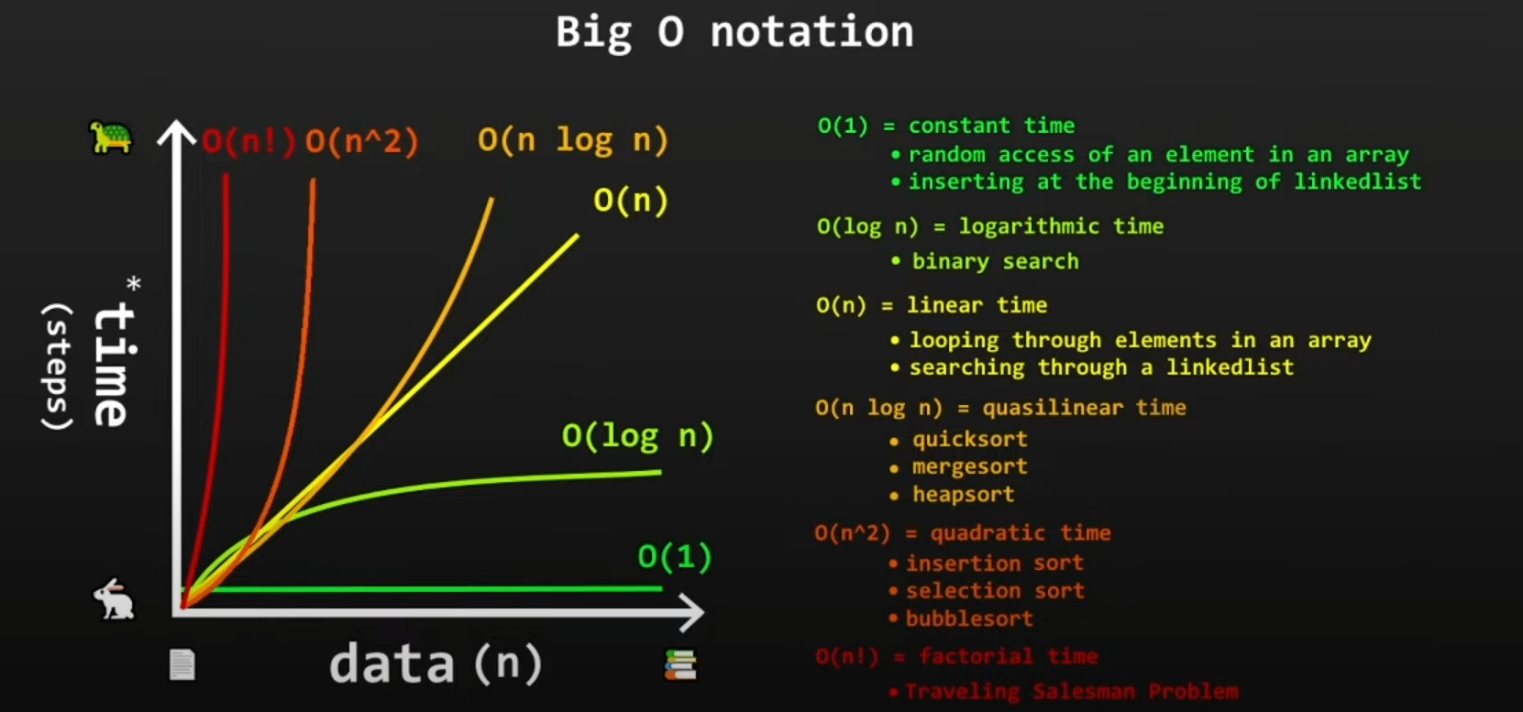
**Algorithm and Data Structure**

**Big O Notation**

1. Describes the performance of an algorithm as the amount of data increases.
2. Machine independent (# of steps to completion).
3. Ignore smaller operations O(n + 1) -> O(n).



**Data Structure:** a named location that can be used to store and organize data.

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Definition** | **Run-Time Complexity** | **Space Complexity** |
| Stack | **LIFO:** Last-In First-Out   * Stores objects into a sort of “vertical tower” * push() to add to the top * pop() to remove from the top   Use Case:  1. Undo/redo features in text editors  2. Moving back/forward through browser history  3. Backtracking algorithms (maze, file directories)  4. Calling functions (call stack) | Access/  Search:  O()  Insertion/  Deletion:  O() | O() |
| Queue | **FIFO:** First-In First-Out   * A collection designed for holding elements prior to processing * Linear data structure * add = enqueue, offer() * remove = dequeue, poll()   Use Case:  1. Keyboard Buffer  2. Printer Queue  3. Used in Linked Lists, Priority Queues, Breadth-first search | Access/  Search:  O()  Insertion/  Deletion:  O() | O() |
| Priority Queue | * **FIFO** data structure that serves elements with the highest priorities first before elements with lower priority. | Access/  Search:  O()  Insertion/  Deletion:  O() | O() |
| Linked List | * Stores Nodes in 2 parts (data + address) * Nodes are in non-consecutive memory locations * Elements are linked using pointers   Singly Linked List:  [data | address] -> [data | address]  Doubly Linked List**:**  [address | data | address] <-> [address | data | address]  Use Case:  1. Implement Stack/Queues  2. GPS navigation  3. Music playlist | Access/  Search:  O()  Insertion/  Deletion:  O() | O() |
| Dynamic Array | * In Java, known as Array List * Dynamic capacity 🡪 array size increases by factor of 3/2 every time you reach over capacity | Access/  Search:  O()  Insertion/  Deletion:  O() | O() |
| Hash Table | * A data structure that stores unique keys to values ex. <Integer, String> * Each key/value pair is known as an Entry * FAST insertion, look up, deletion of key/value pairs * Not ideal for small data sets, great for large data sets   Hashing: takes a key and computes an integer (formula will vary based on key & data type)   * In a Hash table, we use the hash % capacity to calculate an index number * key.hashCode() % capacity = index   Bucket: an indexed storage location for one or more Entries   * can store multiple Entries in case of a collision (linked similarly to a Linked list)   Collision: hash function generates the same index for more than one key 🡪 less collisions more efficiency | Best:  O()  no collision  Worst: O() | O() |
| **Graphs** | | | |
| Adjacency Matrix | * A 2D array to store 1’s/0’s to represent edges * # of rows = # of unique nodes * # of columns = # of unique nodes | O() | O() |
| Adjacency List | * An array/array list of linked lists. * Each linked list has a unique node at the head * All adjacent neighbours to that node are added to that node’s linked list | O() | O() |
| **Trees** | | | |
| Binary Search Tree | * A tree data structure, where each node is greater than its left child, but less than it’s right * Easy to locate a node when they are in this order * Unbalanced tree = worst case | Best: O()  Worst:  O() | O() |
| B-Tree | * Self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. * The B-tree generalizes the binary search tree, allowing for nodes with more than two children. | O() | O() |
| Red-Black Tree | * A kind of self-balancing binary search tree. * In addition to the user data (search key and other data) and the pointers of the binary search tree (BST), each node stores an extra bit representing "red" and "black" (called "color") which helps to keep the tree balanced during insertions and deletions. | O() | O() |

**Algorithm:** a collection of steps to solve a problem.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Definition** | **Run-Time Complexity** | **Space Complexity** |
| **Searching Algorithms** | | | |
| Linear Search | * Iterate through a collection one element at a time   Advantage:   * Fast for searches of small to medium data sets * Does not need to be sorted * Useful for data structures that do not have random access (Linked List)   Disadvantage:   * Slow for large data sets | O() | O() |
| Binary Search | * Search algorithm that finds the position of a target value within a sorted array * Half of the array is eliminated during each “step” | O() | O() |
| Interpolation Search | * Improvement over binary search best used for “uniformly” distributed data * “Guesses” where a value might be based on calculated probe results * If probe is incorrect, search area is narrowed, and a new probe is calculated | Average:  O()  Worst:  O()  values increase exp. | O() |
| Depth First Search | * A search algorithm for traversing a tree or graph data structure. * Transverses a graph branch by branch. * Utilizes a Stack. * Better if destination is on average far from the start. * Children are visited before siblings. * More popular for games/puzzles.   1. Pick a route.  2. Keep going until you reach a dead end, or a previously visited node.  3. Backtrack to last node that has unvisited adjacent neighbours. | Dependent on graph choice | Dependent on graph choice |
| Breadth First Search | * A search algorithm for traversing a tree or graph data structure. * This is done one “level” at a time, rather than one “branch” at a time. * Transverses a graph level by level. * Utilizes a Queue. * Better if destination is on average close to start. * Siblings are visited before children. | Dependent on graph choice | Dependent on graph choice |
| Tree Traversal | * In-order: left 🡪 root 🡪 right * Post-order (deletion): left 🡪 right 🡪 root * Pre-order (copy): root 🡪 left 🡪 right * Utilizes Recursion |  |  |
| **Sorting Algorithms** | | | |
| Bubble Sort | * Pairs of adjacent elements are compared, and the elements swapped if they are not in order. | O() | O() |
| Selection Sort | * Search through an array and keep track of the minimum value during each iteration. At the end of each iteration, we swap variables. | O() | O() |
| Insertion Sort | * After comparing elements to the left shift elements to the right to make room to insert a value | O() | O() |
| Recursion | * Repetition of an internal process * Apply the result of procedure to a procedure * A recursive method calls itself, can be a substitute for iteration * Divide a problem into sub-problems of the same type as the original * Commonly used with advanced sorting algorithms and navigating trees   Advantage:   * Easier to read/write * Easier to debug   Disadvantage:   * Sometime slower * Uses more memory | N/A | O() |
| Merge Sort | * Recursively divides array in 2, sorts, re-combine   Disadvantage:   * Linear space, other previous sorts use constant space | O() | O() |
| Quick Sort | * Moves smaller elements to left of a pivot and recursively divide array into 2 partitions   Disadvantage:   * Worst case if the array is already sorted * Higher space complexity due to recursion | Best: O()  Average: O()  Worst:  O() | O() |
| Heap Sort | * Heap sort is an in-place algorithm. * Its typical implementation is not stable. * Typically 2-3 times slower than well-implemented quick sort, the reason for slowness is a lack of locality of reference. | O() | O() |