Data Structures

1 Dynamic Array

Sequentially stored data in a continuous chunk of memory. Double current capacity whenever capacity reached. When increasing capacity, it allocates new chunk of memory and copy over the previous values to new location.

Runtime

- indexing: *O(1)*
- searching: *O(n)*
- insertion: *O(n)*
- push: *O(n)* Note: *O(1)* Amortized
- deletion: *O(n)*

2 Linked List

Most commonly refers to singly linked list. Data stored in nodes where each node has a reference to the next node. There are doubly linked list and circular linked list as well.

Runtime

- indexing: *O(n)*
- searching: *O(n)*
- insertion: *O(1)*
- deletion: *O(1)*

Stack and queue are often implemented with linked list because linked list are most performant for insertion/deletion, which are the most frequently used operations for stacks/queues.

Stack Last in First Out (LIFO)

Queue First in First Out (FIFO)

3 Hash Table (Hash Map)

A usually unordered data structure that maps keys to values. A hash (preferably unique) is computed for a given key and its value will be stored in the corresponding bins or index according to the hash. Internally the bins can be an array. Collision can happen when multiple keys are mapped to the same hash. Common resolution is to store a list/linked-list at each bin/index location (called chaining).

Runtime

- value lookup: *O(1)*
- insertion: *O(1)*
- deletion: *O(1)*

4 Binary Search Tree (BST)

A binary tree with extra condition that each node is greater than or equal to all nodes in left sub-tree, and smaller than or equal to all nodes in right sub-tree.

Runtime

- searching: *O(log n)*
- insertion: *O(log n)*

- deletion: *O(log n)*

5 Heap (Max Heap/Min Heap)

A binary tree with the condition that parent node’s value is bigger/smaller than its children. So root is the maximum in a max heap and minimum in min heap. Priority queue is also referred to as heap because it’s usually implemented by a heap.

Runtime

- min/max: *O(1)*
- insertion: *O(log n)*
- deletion: *O(log n)*

Algorithms

1 Sorting

Bubble Sort Iterate through entire list while comparing pairs and swap positions based on their values until all elements sorted.

- *$O(n^2)$,* but fast if list is almost sorted.

Insertion Sort Iterates through unsorted list while building a sorted list. For each value encountered in unsorted list, find appropriate place in sorted list and insert it.

- *$O(n^2)$*.

Merge Sort A type of divide and conquer algorithm: 1) divides the list into two equally sized sub lists 2) sort each sub list 3) merge two sorted lists into final list.

- *O(n log n)* — needs to divide *log n* times, and for each divide, it needs to go through all *n* items to merge, thus *n* times *log n*.

Heap Sort 1) Build a heap (min or max) from the unsorted list 2)repeatedly remove the root node from the heap and put into the sorted list.

- *O(n log n)* — remove root node is *O(log n)*, and has to be repeated for each node, thus *n* times *log n*.

Quick Sort A type of divide and conquer algorithm: 1) pick an item in the unsorted list as pivot 2) divided list into 2 sub lists, one contains elements smaller than pivot while the other contains elements greater than the pivot 3) sort the sub lists, and combine the results into final list

- *O(n log n)* — need to divide *O(log n)* times, and after each divide, the partitioning has to go through all elements, thus overall runtime *n* times *log n*.

2 Searching

Linear Search *O(n)*

Binary Search *O(log n)*

Breadth-First-Search (BFS) Siblings first then children. Use queue usually.

Depth-First-Search (DFS) Children first then siblings. Use stack usually.

A* Search Goal is to find the shortest path between 2 nodes in a graph. It's a best-first search. At each iteration it finds the next node to extend the path based on the criteria *g(next) + h(next)* where *g* is the distance from next node to starting node and *h* is the heuristic (estimated) distance of next node to final node. use a heap usually.