Tech Interview Study Sheet

A study guide for technical intervies.

Daniel Rubinstein

December 16, 2019

Data Structures

Arrays

Arrays store a collection of items in contiguous memory locations. Normal arrays have a set length (as one must know how many consecutive memory spots must be allocated for the given array) although there are other methods by which one can increase the size of an array (See Doubling Arrays). Some facts to know about the time complexity of array actions:

Action	Worst Case	Best Case	Average Case
Element Access	O(1)	O(1)	O(1)
Insert / Change / Delete	O(1)	O(1)	O(1)

Note:

- Insert is assumed to mean setting the initial value of a variable
- Delete is assumed to mean changing the value of an element to some temporary invalid value.

Other interpretations of inserting and deleting will have a varied time complexity.

Doubling Arrays

This is simply an array that has a dynamic size. Once the array is filled, a new array of double the size is created and all the contents of the original array are copied over.

Linked Lists

This recursively defined data structure stores a collection of objects (not necessarily in consecutive memory spots) by defining a node in the list as a structure holding some form of data as well as a pointer to the next node in the list.

Some useful information about the time complexity of various actions:

Action	Worst Case	Best Case	Average Case
Element Access	O(N)	-	O(N/2)
Insert / Change / Delete	O(N)	-	O(N/2)

Doubly Linked Lists

These are simply linked lists that have pointers both to the next and previous node.

Stacks

This data structure stores the pointer to a "stack of information" following the First in Last Out principle. One can either pop or push from the stack.

- Pop allows one to take the top item off of the stack.
- Push allows one to put a new item onto the stack.

Implementations:

- Using arrays
  - Must track the top of the stack manually.
  - Can be fixed capacity.
- Using linked lists
  - Add / remove from the stack by insertin / deleting nodes.
  - No fixed capacity.

Queues

This data structure works similarily to a line up of people at the grocery store. Data is inserted and taken out using the First in First Out principle. One can either enqueue or dequeue from the queue.

- Enqueue adds a new data item to the queue.
- Dequeue takes the "first in line" data item out of the queue.

Implementations:

- Array
- Linked list

Sets / Bags

Sets are collections of unique items. Order is irellevant and so is multiplicity as only one copy of each item is stored. A bag is similar to a set only multiplicity matters as multiple copies of each item can be stored.

Priority Queues

Similar to queues only things are taken out based on their priority level (by maximum or minimum value) instead of the order they came in. There are various implementations of such a data structure:

Unordered Arrays

The relevant time complexities are as follows:

Action	Time Complexity
Find Max	O(N)
Remove Max	O(N)
Insert	O(1)

Ordered Arrays

The relevant time complexities are as follows:

Action	Time Complexity
Find Max	O(1)
Remove Max	O(1)
Insert	O(N)

Binary Heaps

What is a heap? A heap is a structure such similar to a binary tree such that both children have values less than their parent. Properties (given that an array (id) stores the nodes in the heap):

- Children of id[x] are found at indexes 2*x and 2*x + 1
- The height of the heap is LogN

There are two actions that are used to construct and manipulate heaps:

- Sink takes a node and moves it lower into the tree until heap condition is satisfied.
- Swim takes a node and moves it higher in the tree until heap condition is satisfied.

How to do the three main PQ actions:

- Find max: Simply look at the root.
- Remove max: Swap the root with the last leaf node, remove it, and sink the new root.
- Insert: Add a new leaf node, swim it up.

The relevant time complexities are as follows:

Action	Time Complexity
Find Max	O(1)
Remove Max	O(LogN)
Insert	O(LogN)

D-Ary Heaps

Work just like binary heaps, only these have d children. Binary heaps have time complexities of O(LogN), d-ary heaps have time complexities of O(cLog(base c)N)where c is the number of children that can exist in the d-ary heap.

Trees

Binary Tree

A tree with 2 children

Complete Binary Tree

A tree such that all levels are completely filled.

Binary Search Tree

Useful for representing symbol tables. Properties:

- Left child is smaller than parent
- Right child is greater than parent

For a symbol table represented as BST the relevant time complexities are:

Action	Time Complexity
Search Garuntee	O(N)
Search Average	O(logN)
Insert Garuntee	O(N)
Insert Average	O(logN)

How do we tackle deletion?

- Tombstone method - Make some nodes "removed" or tombstone nodes
- Hibbard deletion - Based on the case, find a way to delete the node and rebuild the rest of the tree.

Problem is that tree shape can vary How do we fix this problem?

- Balanced Search Trees

Balanced Search Trees

We can improve the balance of a BST by using various balancing techniques. One such technique is to implement a 2-3 tree. This tree has nodes that are either:

- 2 Nodes: 1 Key, 2 Children
- 3 Nodes: 2 Keys, 3 Children

Actions for such a tree include:

- Search: Done like a BST based on the values of the nodes.

- Insert into 2 Node: Transform the node into a 3 Node
- Insert into 3 Node: Transform the node into a temporary 4 Node, then move the middle key up and restructure the tree.

How can we represent such a tree?

- As a BST? No
- As a BST with glue nodes? Yes but this is tedious.
- As a Red Black Tree? Perfect!

The relevant time complexities for this structure are:

Action	Time Complexity
Search Garuntee	O(CLog(base C)N)
Search Average	O(CLog(base C)N)
Insert Garuntee	O(CLog(base C)N)
Insert Average	O(CLog(base C)N)

Where C is dependant on the types of nodes in the tree.

Red Black Trees

This is a BST that now has a colour value attached to it's various node connections. There are a few properties that always hold:

- No nodes can have 2 red connections going into / out of them.
- Every path from the root to a leaf has an equal number of black connections.
- Every red connection is left leaning

Red connections signify 3-nodes from Balanced Search Trees above.

There are three actions that are useful to know for constructing such a tree:

- Rotate left
- Rotate right
- Change colours

The relevant time complexities for this structure are:

Action	Time Complexity
Search Garuntee	O(LogN)
Search Average	O(LogN)
Insert Garuntee	O(LogN)
Insert Average	O(LogN)

B-Trees

TLDR: Like 2-3 trees, only they can store a variable amount of keys in each node. I'll add more on this later.

Hash Tables

Used for symbol table implementation. Determines where to store info based on a hash function. Hash function determines the memory location where the data is to be stored.

We run into an issue: What happens if we continually hash to the same place?

- Solution 1: Seperate Chaining
  - Store linked lists of data at each hash location. Then you only have to search through the linked list to find the data.
    - On average given M locations to hash to, and N values, it takes O(M/N) to find a data segment.
- Solution 2: Linear Probing
  - If the hash location is currently occupied, find the next available location to hash to.
    - Another problem: This gets worse the more we hash to the same location (i.e. problem called clustering)

Other improvements:

- Double probing: Hash to two different locations, place data into the open one
- Double hashing - Instead of finding the next available memory location like in linear probing, skip a variable amount of memory blocks and try there.
- Cuckoo hashing - Hash to two locations, reinsert some of the data there if both are filled. (I'll add more on this later).

Graph

A series of vertices connected (possibly) by edges. Graph Terminolog:

- Path - A set of connections between two vertices.
- Cycle - A path that loops.

Possible representations:

- Adjacency matrix - 2D matrix with booleans to explain if two vertices are connected.
- Adjacency lists - Series of linked lists stored in an array to describe which vertices are connected.

DFS:

- Depth first search.
- See implementation examples to be uploaded soon.
- Can be used to solve connectivity problem.
- Can be used to find cycles within DAGs.

BFS:

- Breadth first search
- See implementation examples to be uploaded soon.

Undirected vs Directed:

- Undirected has no direction to the edges in the graph.
- Directed graphs have edges with a direction.

Strongly Connected Components:

- V and W are strongly connected if there is a path from V to W and a path from W to V.
- Strong component is maximal subset of strongly connected vertices.

Edge Weighted Graphs

These are trees with weights given to each edge. It brings up the interesting idea of finding shortest paths and minimum spanning tree.

- MST A minimum spanning tree (MST) is a tree that:
  - Is connected (Has all vertices connected)
  - Is acyclic
  - Includes all vertices

  The greedy logic behind calculating the tree revolves around making cuts. The general logic states: Given any two sets of unconnected vertices ("cut" sets), the minimum edge that connects them is in the MST.

  To formalize this we have two different algorithms: Kruskall's and Prims

  Kruskall's algorithm works as follows:

  - Continually add the minimum weight edge if it makes no cycles in the MST.

  This is difficult to do as determining if a cut makes a cycle is difficult.

  Prim's (Lazy Approach)

  - Use a minimum PQ that stores every edge weight that connects a vertice W (not in the MST) to the MST.
  - Remove the minimum edge from the PQ and add that to the MST (if it is still a valid edge to add).
  - Update the PQ with new edges that are available that fit the desired conditions above.

  The problem with this is that our PQ is not efficient. We store tons of edges that eventually become invalid.

  Prim's (Eager approach)

  - Use a minimum PQ that stores only one edge weight for every vertice W (not in the MST) that can be connected to the MST.
  - Remove the minimum edgr from the PQ and add that to the MST.

  - Update the PQ with new edges that are available that add new possible vertices to connect to, lower the weight to add any given vertices, etc.

  More efficient with our memory, but a bit harder to keep track of.

- Shortest Path This section will be refined later. For now, here's the two algorithms I know for shortest path problems:

  Djikstra:

  - Consider vertices in increasing order of distance from the origin point.
  - Relax edges as you go.

  Works well if there's no cycles and no negatives.

  Bellman-Ford:

  - Initialize distance to origin as 0 and infinite for every other vertice.
  - Go through the list of vertices V-1 times and relax every distance possible.

  More to be added later.

Algorithms

Searching

Binary Search

TODO

Sorting

Selection Sort

Selection sort works by:

- Iterate through the list with a counter c
- Find the minimum value in any index >= c
- Swap the minimum value with the value at c
- Increase c

The time complexity is as follows:

Worst Case	Average Case	Best Case
$O(N^2)$	$O(N^2)$	$O(N^2)$

Insertion Sort

Insertion sort works by:

- Iterate through the list with counter c
- Swap the element at c with the item to it's left until it is in it's sorted position
- Increase c.

The time complexity is as follows:

Worst Case	Average Case	Best Case
$O(N^2)$	$O(N^2)$	$O(N)$

Shell Sort

TODO

Merge Sort

This sorting algorithm works by:

- Splitting the array in half recursively until it can no longer be split
- Building up an array out of the split pieces in sorted order.

The time complexity is as follows:

Worst Case	Average Case	Best Case
O(NLogN)	O(NLogN)	O(NLogN)

Quick Sort

This sorting algorithm works by:

- Picking a pivot point
- Rearanging the list such that the pivot is placed in it's sorted position

  - Meaning everything to the left is $<$ the pivot, everything to the right is $>$ the pivot.

- Recursing on the left and right sub-arrays

The time complexity is as follows:

Worst Case	Average Case	Best Case
$O(N^2)$	O(NLogN)	O(NLogN)

Some improvements:

- 3-way Partitioning improves quick sort with duplicate values.
- Applying insertion sort on smaller sub-arrays can also work well.

Heap Sort

Heap sort works on a heapified array by:

- Remove the maximum value by swapping it with the last leaf node, removing the value, and swimming the new root down.

The relevant time complexities are as follows:

Worst Case	Average Case	Best Case
O(LogN)	O(LogN)	O(LogN)

Topological Sort

Used on Directed Acyclic Graphs. Good for dependency problems. Topological sort returns a sorted list by:

- Run DFS on all vertices (using some way to track which vertices are visited)
- Return reverse order of visited points.

Multiple possible topological sort results can be valid.

Union Find Problem

The union find problem begs the question of how nodes are connected within a graph like strucutre. There are three primary operations that are of interest when it comes to algorithms that solve this problem:

- Are two nodes connected?
- What connected component is a node in?
- How do we connect two nodes?

Quick-Find

Given an array (id) of N objects we chose to state that the value at id[x] is the id of the component that node x is in.

The time complexity for the three main actions are as follows:

Action	Time Complexity
Connected?	O(1)
Find?	O(1)
Union?	O(N)

Quick-Union

Given an array (id) of N elements. We state that id[x] is the parent of node x (almost like building a little tree).

The time complexity for the three main actions are as follows:

Action	Time Complexity
Connected?	O(N)
Find?	O(N)
Union?	O(N)

Some improvements to this could be:

- Weighting
  - Put smaller trees as children of larger trees.
  - This garuntees a height of LogN thus making O(lgN) time complexity on all actions.
- Path compression
  - As one traverses the tree, move children up in the hierarchy to create shorter paths to traverse from the parent to the leaf node.

Language Knowledge