LINKEDIN.COM/IN/OWENPSMITH/

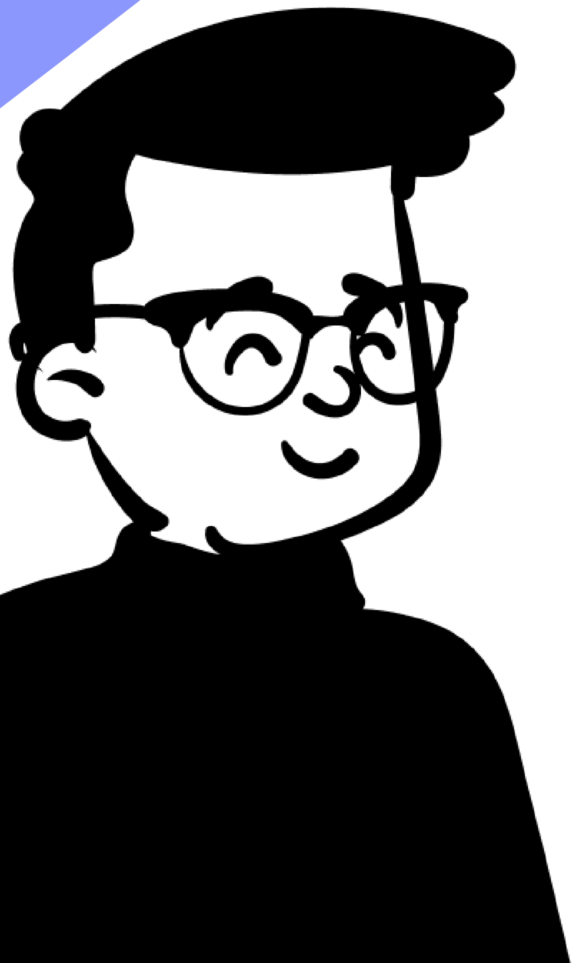
Technical Interview Crash Course

Owen Smith

An all-in-one quick guide on how to tackle technical interviews for Software Developers.

INTERN, NEW GRAD, AND JUNIOR POSITIONS

Table of Contents



Data Structures

Stack

A LIFO (Last In First Out) data structure, stacks resemble a deck of cards where we can only add and remove from the top card.

In general, there are only four operations:

- Push(item): add element to the top [O(1)]
- Pop: remove top element [O(1)]
- Peek: return top element (no remove) [O(1)]
- isEmpty: true iff stack is empty [O(1)]

Queue

A FIFO (First In First Out) data structure, queues can be thought of as a store lineup where customers first in line are the first to be served.

In general, there are only four operations:

- Enqueue(item): add item to the end [O(1)]
- Dequeue: remove first element [O(1)]
- Peek: return first item (no remove) [O(1)]
- isEmpty: true iff stack is empty [O(1)]

Searching through stacks and queues, like arrays, take linear [O(n)] time. When solving problems, look for cases where we can exploit the internal structure of stacks and queues to simplify the solution to the problem.

Hashmap

A hashmap (or hash table) is a data structure that maps keys to values. Thus we can think of elements as unique key-value pairs. In the vast majority of cases, operations on a hashmap are incredibly efficient.

Operation Effeciency

Searching, insertion and deletion take constant time [$O(1)$]. In the (rare) worst case, these operations take linear time [$O(n)$].

When to use Hashmaps

Hashmaps are generally you go-to when performing multiple lookups, or searching for specific values.

Additionally, we can also implement a set using a hashmap. That is, an unordered collection of elements in which duplicates cannot be stored. A simple implementation is to set the key and values to share the same value. In Python sets are a built-in data structure whereas in Java, we can import the HashSet from the utils package.

Linked Lists

A linked list is a data structure that represents a sequence of nodes. In a singly linked list (SLL) each node points to the next node. A doubly linked list (DLL) gives pointers to both the next and previous node.

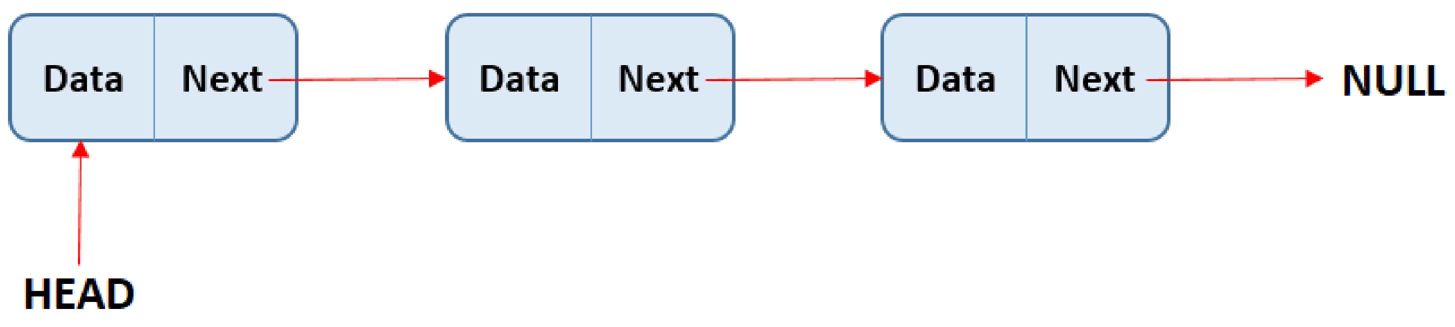
Operation Effeciency

Inserting and deleting a node from (the beginning of) a linked list (SLL and DLL) takes constant time [O(1)]. Searching still takes linear time [O(n)].

The Tortoise and the Hare Technique

This is a technique where we keep two pointers that run through the array, one is a fixed number of nodes ahead (the hare) of the other (the tortoise). It is also known as the "Runner" Technique in many books. The idea is that perhaps the hare can offer information to the tortoise by running ahead in the list.

We can use this technique to determine cycles in linked lists (Floyd's Algorithm) among many other applications.



Graphs and Trees

Trees

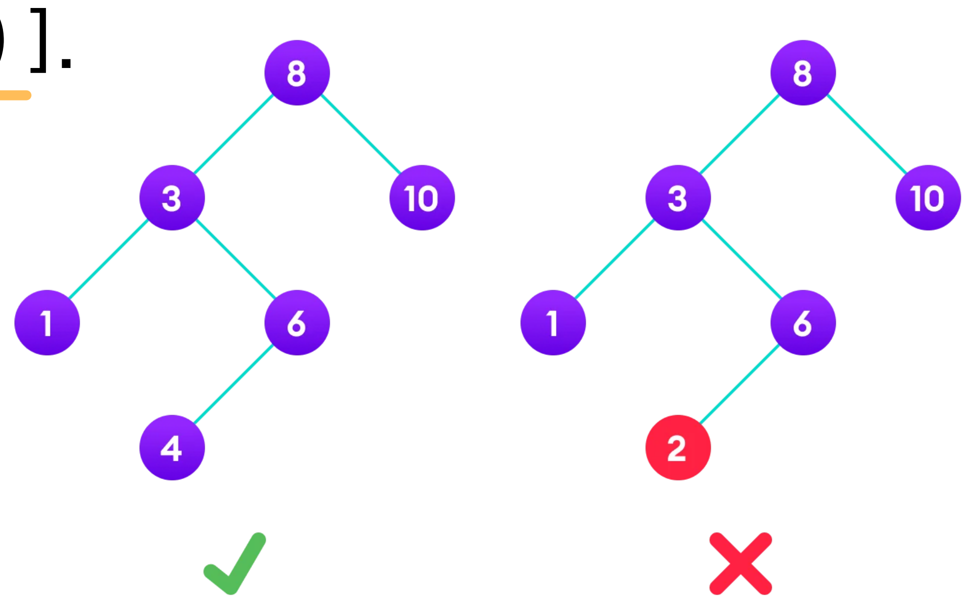
A tree is a collection of nodes which are linked through edges. We say node X is a child of node Y if there is an edge from Y to X. Nodes with no children are called "leaf" nodes. Trees by definition have no cycles.

Binary Trees

Just a regular tree except every node has at most two children. A binary tree is said to be perfect if all non-leaf nodes have two descendants

Binary Search Trees

A binary search tree is a binary tree in which every node fits a specific ordering property. For every node in the tree, the values of all left descendants must be less than all the right descendants. In the average case, Binary Search Trees have logarithmic search, insertion, and deletion time complexity [O(log n)]. However, in the worst case these operations drop to linear time [O(n)].



Binary Tree Traversal

- In-Order: visit the left branch, then the current node, then finally the right branch.
- Pre-Order: visit current node then left and right children.
- Post-Order: visit left branch, right branch, then current node.

Binary Heaps

In a min-heap the root is the minimum element in the tree (the maximum element for max-heaps). Heaps contain two operations: insert and extractMin. These operations both take logarithmic time [O(log n)]. Heaps are useful for problems where we would expect a priority queue. In fact, we often implement priority queues using heaps.

Tries (Prefix Trees)

A trie is a tree where characters are stored at each node. A path (string of characters) from the root node to a "null" node signify a word. A path that does not end in a null node signals that there contains word(s) where the path is a prefix. A trie can check if a string is a valid prefix in linear [O(n)] time with respect to the length of the string.

Graphs

A graph is a collection of nodes with edges/connections between (some or all of) them. Graphs may contain cycles.

Breadth First Graph Search

Visit all nodes at the current depth before moving on to nodes at the next depth level.

```
procedure BFS(G, root) is
    let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as explored then
                label w as explored
                Q.enqueue(w)
```

Breadth First Search commonly appears in scenarios where we explore 2D grid environments.

Depth First Graph Search

Visit as far as possible along each branch before backtracking.

```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

Depth first search is useful for finding cycles in graphs.

Dynamic Programming

In DP, we wish to identify and solve simpler subproblems and then combine them together to solve a larger problem.

Identifying DP Problems

Problems with the objective of computing or listing the "nth" of something, should be a good hint at a dynamic programming (or at least a recursive) solution.

Identifying subproblems are critical, ask yourself,

- *"How can we use smaller parts of the problem to compute the current solution?"*
- *"What changes, or choices do I have when I add an extra element?"*

Be careful about recursive approaches, recursive functions are notoriously space inefficient. However, sometimes the code to implement an iterative approach is significantly more complex. Discuss the trade-off with your interviewer.

Sorting and Searching

Sorting

Sorting is generally a technique to help you reframe the problem. Sometimes sorting a given input can provide insight into the problem structure and possible simplifications.

The time complexity of built-in sorting methods is [O (n log n)].

Searching

Binary Search is the most efficient search algorithm but relies on the list being sorted. The time complexity is logarithmic [O(log n)]. The algorithm is as follows:

- Check the middle element, if target is found return.
- If it is greater than the target, we repeat but on the subarray before the middle element.
- If it is smaller than the target, we repeat but on the subarray after the middle element.

Interestingly for problem with arrays which specifically ask for a [O(n log n)] or [O(log n)] solution, we should be expected to implement some variation of binary search.

Union-Find

Union-Find is the underlying algorithm of the Disjoint-Set data structure. It is incredibly useful for producing groups (or sets) of elements such that each group is disjoint (no overlap).

Algorithm

First we create an array called *parents*. A parent is a group representative. So initially each element is in its own group, thus all elements are their own parent. To find an element's group representative we use the "Find" function. This will recursively lookup the parents until the parent is itself.

```
function Find(x) is
  if parents[x] != x:
    parent[x] = Find(parent[x])
    return parent[x]
  else:
    return x
```

To merge two groups together, we use the "Union" function. We simply set the parent of one group representative to the other.

```
function Union(x,y) is
  // Replace nodes by roots
  x = Find(x)
  y = Find(y)

  if x == y then
    return // x and y are already in the same set

  // Make x the new root
  parent[y] = x
```

Sliding Window

Sliding Window technique is for array problems where we can convert nested for loop solution into a single loop. Thus, drastically improving the time complexity.

We are only looping through an array once. So we start at the beginning and look at the first k elements. Depending on what we see, we may decide to shrink that window (k = k-1), increase the window (k = k+1), or slide the window to the next position (now looking at index 1 to k+1).

Example: Leetcode #3

Given a string s, find the length of the longest substring without repeating characters.

For example for input "helloworld" the answer would be 5 since "world" contains five characters, none of which are duplicates.

```python
def lengthOfLongestSubstring(self, s: str) -> int:
        window = set()
        l = 0
        best = 0
        for r in range(len(s)):
            while s[r] in window:
                window.remove(s[l])
                l += 1
            window.add(s[r])
            best = max(best,len(window))
        return best
```

5 Steps To Better Solve Problems

1. Translating the Problem

It is critical that you start analyzing how you start a problem. Rushing into an implementation without completely understanding the problem and its constraints can lead you off track and cause you to waste time.

Think real hard about how you start problems. I like writing out all the given constraints, and generate some test cases. Solving a test case manually gives you a better understanding of the inner workings of building a solution. Perhaps there are some patterns you recognize that can help with your own implementation.

2. What Data Structures Do I Use?

Identifying patterns is a huge part of problem solving. Ask yourself,

- *"Does this remind me of any problem I've seen before?"*
- *"What am I trying to accomplish? And what data structure can help me with that task?"*

3. Preprocessing

Is there something you can do to the input data that can help simplify/speed-up your algorithm (i.e. sorting, eliminating duplicates, etc.)?

If you're stuck on a problem perhaps preprocessing the input can give insight to new properties and patterns.

4. Write It Down & Optimize

Don't even think about coding an implementation yet. Write down (on paper or comment pseudocode) the steps of your current algorithm. Try not to worry too much about optimization. Once you've written down a complete solution, revisit each line and try to optimize or speedup your algorithm. Make sure you find the space and time complexity of your solution.

5. Implement Your Solution

Finally, once you are absolutely sure of your algorithm, and have verified it with your interviewer, type it in code. We write things out first (even in pseudocode) so we aren't getting hung up on the syntax or small implementation bugs. Interviewers mostly care about the high level problem solving process, less about the actual syntax to implementing a solution.

Before the Interview

Don't Sleep on the Behavioural Q's

Even in technical interviews, there are opportunities (either at the beginning or end) to talk about your experience, career goals, and how they relate to the company.

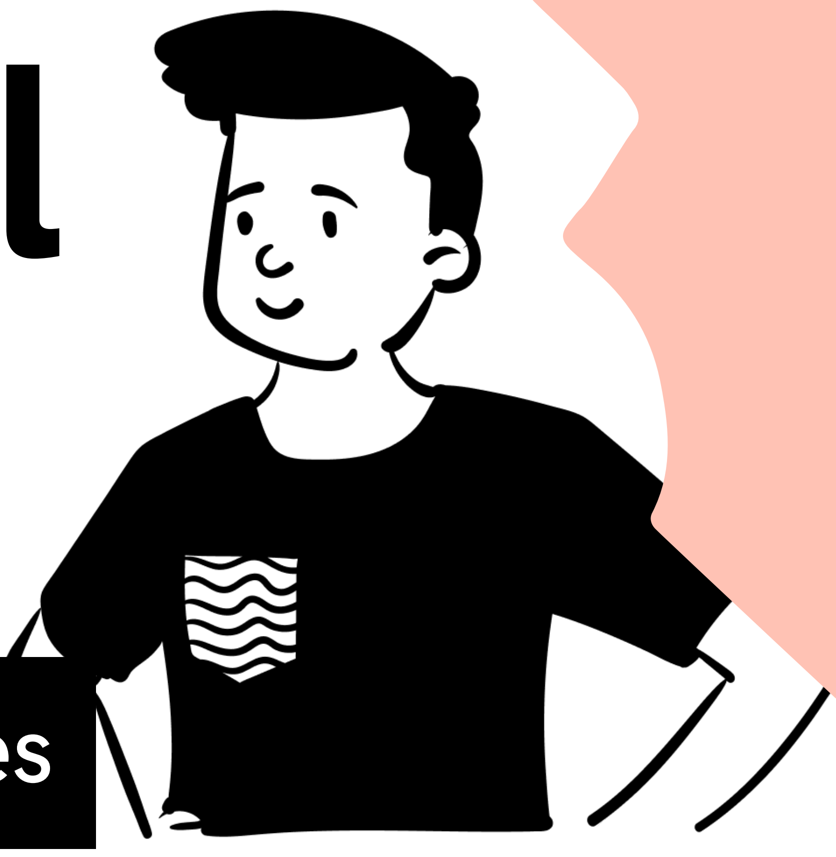
Be sure to review the ins and outs of your resume and relevant experience/projects. Try to have 2-3 stories which cover a variety of the company's principles and values.

Relax & Smile!

Remember that the interview is ultimately about selling yourself. It's not just an examination of your problem solving skills.

Hiring managers/interviewers look for people they would want to work with and who embody the company culture. Be sure to communicate that naturally through your tone and body language throughout the interview.

Personal Tips



Don't Isolate Categories

Perhaps the most important skill for solving these programming challenges is the ability to identify which techniques/data structures are needed to craft an optimal solution. People rob themselves of harnessing this skill by dedicating a study session to dynamic programming questions or graph and tree problems, etc.

Two Pointer Technique

This trick was actually was the solution to my interview problem at Amazon. In essence, when we are performing many operations at once, we can often simplify this process by keeping track of only two pointers. The two pointers are usually for start and end. They could store numerical values or point to positions in an array.

So if you are in a situation where you are iteratively performing many operations to a data structure, perhaps incorporating two pointers can offer significant speed-up opportunities.

External Resources

- Cracking the Coding Interview by Gayle Laakmann McDowell
- Leetcode.com
- Jeremy Aguilon - Ranking Interview Questions by Cram Score
- 75 Curated Questions from a Facebook Tech Lead (original article)

Thanks for Reading!

I hope you got some value out of this crash course. The best of luck on your interview!

If you enjoyed this style of content, or have ideas for future crash courses - I'd be happy to hear it 😊

LINKEDIN.COM/IN/OWENPSMITH/