“What is a HashTrieStackFilterMap?” A Data Structures & Algorithms Refresher For Everyone

**Abstract:**

- Are you about to go into job interviews where you'll be asked technical questions?

- Have you found you don't really know the standard data structures or algorithms, or you have holes in your knowledge?

- Has it been a long time since your computer science classes and you want a review session?

In this workshop, we'll go over all the basics of standard data structures and algorithms. We'll start at the beginning with different types of lists and build up from there. We'll also cover the basics of popular algorithms and how to analyze operations of the data structure to assess a “Big O” runtime. We'll break into teams to solve some technical problems. You will leave with a wealth of knowledge and resources to help you!

(While this won't replace a whole CS degree, it's designed to give you a boost in your CS learning journey.)

**Short Abstract:**

In this workshop, we'll go over all the basics of standard data structures and algorithms. We'll start at the beginning with different types of lists and build up from there. We'll also cover the basics of popular algorithms and how to analyze operations of the data structure to assess a “Big O” runtime. We'll break into teams to solve some technical problems. You will leave with a wealth of knowledge and resources to help you!

**Ideas (via twitter):**

* This may be terrible but I’ve made it through a pretty good career with almost exclusively Arrays and Hashes, and using pre-built sort functions. I know about Linked Lists and B-Trees and O-notation but TBH I’ve never really needed them. FWIW I do mostly biz stuff and apps. […] I did once read a book on Algorithms and Data Structures, but I’ve found I never had to use much of it. The most influential books for me were The Pragmatic Programmer, the first third of Code Complete, and about 1/5 of Design Patterns.
* At school!!? Stacks, queues and linkelist 😀😀
* General ones which come up in interviews a lot and I tend to use for development: lists, arrays, maps, hash maps especially, various trees (rarely use actual trees for development). Never used in development, but came up in interviews: red black trees.
* I don't know if I would call this a data structure or an algorithm, but the "Read Write" mutexes that (hopefully) come bundled in <your favourite language> have become increasingly more important to me, and I remain surprised at how infrequently I see them used. […] I think just infrequently used. Not all code needs to be concurrent (thank goodness), and concurrency is hard, but there is a lot of code that can be made more concurrent. I think I'm just substituting "Concurrent code" for "RW Mutexes"
* Lots of hash and array manipulation, primarily map and reduce. I’ve also needed to use graph traversal, specifically creating and iterating through directed acyclic graphs. I suspect this is rare though
* I think e.g. std::map is normally implemented as a red black tree, so typically you don't write your own. I'd say it's more important to understand why red black exists (rebalancing the tree to avoid worst case performance) than memorising the implementation
* bloom, cuckoo, and time decay bloom filters are some weird ones that ive found useful.
* Those f\* sort algorithms. E.g. mergesort in place. Using arrays of arrays of arrays in C, aka pointerpointerpointers ... Not that I ever used that in real life. But thinking about us the key.
* Singly linked lists and binary trees! They're everywhere in Haskell.
* This paper on folds: [cs.nott.ac.uk/~pszgmh/fold.p…](https://t.co/3IPq6t47Rv)
* State machines are everywhere, whether you realise it or not. The hairiest of logic can be deciphered in an approachable way with a state machine. Very useful for web UIs in the age of asynchronous requests!
* You mean like trying to use binary sort when matching socks after washing them? First make two big piles, one for whites/white-ish and the other for colors. Then separate the colors into black/black-ish and the rest of colors and just continue like that.
* Most of the times I've seen algorithms solving in 3 aspects of my dev exp: 2% of my day to day work, interviews, and practicing interviews. Although, learning (some) Dynamic Programming helped me to be a better developer. 😉 Similar problems are solved similarly.
* Traversing a linked list. As a visual thinker these really challenged me and taught me how to visualize pointers and abstract data in general.

**Outline:**

Introductions

* Introduce me
* Introduce group
* Why talk about data structures/algorithms, especially if most people say they don’t use them on a daily basis?
  + Sometimes theory helps us understand what we do
  + Sometimes things come up that make us feel like outsiders or not smart enough
  + Because programming and computer science theory are different
  + Because sometimes efficiency *does* matter
* Talk about outline, exercises regularly mixed in, work together
* Open workshop, ok to get up, move around, talk to each other, ask questions, etc.
* No stupid questions, only unanswered ones
* I’ll try to do frequent breaks because I know too much sitting and learning is hard
* DISCLAIMER: I do NOT recommend using anything here as a method to hire new employees. I’m just teaching it for your knowledge and in case anyone does ask you. (Find me later to ask why)

Part 1: The basics

* Data Structure: A way of storing and organizing data in a computer so it can be used effectively
  + You want this to reduce time and/or space complexities for different tasks
  + These are usually built into most languages (sometimes through libraries), but we’re covering concepts
* Arrays/Linked Lists
  + Arrays/lists:
    - Collection of single values (like a bunch of variables)
    - Tied together in one continuous chunk of memory
    - Some languages may use these interchangeably
    - Usually start at index 0 (though some languages can use 1)
      * Usually math languages like FORTRAN, MATLAB, Julia, Lua, etc.
    - [picture]
  + Linked Lists:
    - Collection of single values (like a bunch of variables)
    - Tied together through links (pointers) as they’re not continuous in memory
    - Some languages implement linked lists behind the scenes for a “list” type structure
    - [picture]
    - Singly-linked means they point to the next item in the list
    - Doubly-linked means they point to the next and previous item in the list
    - Circularly-linked means the end points back to the beginning
  + Both are basically the exact same thing, but they function differently and have different efficiencies
  + Creating them
    - Empty arrays and lists are made in a single step
    - Arrays need to know the size so they can grab that solid chunk of memory (fixed size)
    - Linked list always starts off with a pointer to nothing, so they take up no room (variable size)
  + Adding/deleting
    - Adding/deleting from beginning/end of either is easy
    - Adding to middle of an array is costly (you have to shift elements over or copy to make a new one)
    - Adding to middle of a linked list is cheap (you just make a new point and change links around)
    - Deleting in array is easy, but dealing with the hole in the middle is not
    - (could move last item into hole to speed it up a bit if you don’t care about order)
    - Deleting in a linked list is easy as you just remove the node
  + Accessing/Searching:
    - To access an exact element in an array, you can go straight to it (arr[5])
    - You cannot access an exact element in a LL
    - To find something in an array or linked list, you can loop through it
  + Space:
    - Arrays will always stay the same size (unless you add items, but you have to create a new one and copy over)
    - Linked Lists will always be the size of the items it has
  + [insert chart/table comparing these]
  + So, when do I use them?
    - Both if you’re storing simple lists
    - Arrays are super easy to implement. Linked Lists involve some code
    - Arrays if you need fast seek times
    - Either for searching for an item
    - Linked Lists if you are rearranging elements a lot or adding in the middle often
    - Linked lists for really large data sets (since you can’t allocate one solid chunk at once)
  + Problems:
    - How might you store….
      * A phone book (for lookups): probably an array
      * List of all odd integers above 10: array
      * List of library book titles: probably an array
      * List of library books checked out at the moment: probably a linked list
      * Names of sessions at a conference in alphabetical order (considering people can change names of talks before the conference): probably a linked list
    - How would you add item to a linked list?
      * [show example]
    - How might you merge two lists together? (array or LL)
* Stacks/queues
  + Two applications of lists
  + They follow a particular order of how things are added/removed
  + Stacks:
    - Like books!
    - First in is the last out (FILO) or Last in, first out (LIFO)
    - Insert/delete happen on the same end (just like with real stacks)
    - Add is “push”, remove is “pop”
  + Queues:
    - Think cashier lines
    - First in is the first out (FIFO) or Last in is the last out (LILO)
    - Insert and delete happen on opposite ends
    - Add is “enqueue”, remove is “dequeue”
  + If implemented right, push/pop are both single steps
  + Often “peek”/”front”/”top” is implemented to just look at the next item to pop without popping it
  + As arrays:
    - Stacks/queues have a limited size
    - If you also track the beginning (“head”/”bottom”/”front”) or end (“tail”/”bottom”/”top”), you can keep push/pop quick and not slide elements around
  + As LLs:
    - This is easier than arrays
    - Stacks are easy with singly-linked lists, add at the front/head and pop there too
    - Queues are easy too. Add to the back/tail, remove from front/head.
  + So, when do I use them?
    - Stacks:
      * Balancing things (math symbols, HTML tags, etc.)
      * Anything with backtracking (like undo/redo, forward/backward on browser, maze traversals, sudoku solvers, etc.)
      * Math parsing
      * Recursive things (function call stacks, listing files and directories)
    - Queues:
      * When something is shared in order (ex: scheduling disk access, print documents)
      * Asynchronous data (any kind of input/output buffer)
      * Load balancing (you have X processes, as they complete you push into queue to be pulled out again when needed)
  + Example problems:
    - What would you use for a “pull a number” system for people returning products? (queue)
    - What would you use to reverse a word or string? (stack)
    - What would you use to determine if a string had the right number of open/close parenthesis? (stack)
    - What would you use to route telephone calls in a customer support center?
    - What would you use to determine the order of boxes to remove from a shipping truck? (stack, first in, last out)
* STRETCH!
* Practice problems in teams (one per team):
  + Merge two Linked Lists algorithmically (A-B-C, D-E-F => A-D-B-E-C-F)
  + How would you implement a queue as an array?
  + Merge two arrays into another in sorted order (A-C-E-F, B-D-G-H-I => A-B-C-D-E-F-G-H-I)
  + Reverse an array using a stack, then a queue
  + Check if a string is a palindrome with a stack
  + How can you change a queue to make it so important items get moved to the front?
* BREAK!
* Trees
  + What if you had a list, and instead of it being linear, it was hierarchical?
  + Ex: Children have parents, but parents also have parents. Children can also have children
  + In a linked list, if each item has a child pointer, you can make it like a singly-list tree. If something points to its parent, it can be like a doubly-linked list
  + [Picture]
  + Why trees?
    - Can store hierarchical things easier
    - With some ordering built in (discuss in a moment), they can be quicker to search than linked lists (though sometimes not as fast as arrays)
    - Fairly quick insertion (faster than arrays, slower than unordered linked lists)
    - No upper limit on storage (they work like linked lists)
    - Can be really fast to search through
  + What do they look like?
    - [picture, node+data + child nodes]
  + Types of trees
    - Binary tree: each node has up to 2 children (hence “binary”, usually named left/right)
    - Binary search tree: binary tree where each node is inserted in such a way to make things easy to find alphabetically/numerically (left is less than, right is greater than)
    - Full binary tree: each node has 0 or 2 children (not 1)
    - Compete binary tree: every level is full except last level, but all nodes are as far left as can be
    - Perfect binary tree: all nodes have 2 children (except bottom nodes)
    - Pathological tree: all notes have 1 child. (it’s basically a linked list)
    - AVL tree: A tree that makes sure the levels between left and right subtrees are no more than 1.
      * (if anyone asks you to build this in a job interview, just leave)
    - Expression tree: Each node corresponds to a math operator (ex: + with 3 to left and 5 to right)
    - Decision tree: Each node is a decision point where children are results of the decision
    - Heap: a self-balancing and sorting tree where maximum or minimum values rise to the top
    - Trie: (tree or “try”) Special tree used to quickly search for things
  + So, when do I use them?
    - Anything that forms a hierarchy
    - With ordering, trees can provide some pretty fast searching and access
    - Anything with nested layers
    - Anything where different actions occur though choices (multi-stage decision making)
* Heaps
  + A complete binary tree
  + Usually two types: max-heap and min-heap
    - Both present the root node to the be max/min elements compared to its children
    - This works recursively throughout
    - When a new item is added, it’s likely not in the right place. “Reheaping”
  + Why?
    - It always maintains a balanced structures
    - Easy to find the largest or smallest item in a set
    - Can be used for sorting
    - Can be used to implement storing things with different priority levels
    - This is how memory allocation works in a computer
  + Con: when you insert/delete something, that reheaping has to happen, which takes some time to rearrange the elements.
* BREAK (?)
* HashMaps/Hash Tables/Map
  + What’s a hash?
    - A formula that, given some input, returns a (hopefully) unique value
    - Examples include MD5, SHA-1, SHA-256, etc.
    - Super simple example might be n % 10
  + What’s a map/set/pair?
    - Ties a “key” to a “value” so you can easily find that value later
    - Think word→definition, or address→resident, or phone number→owner
  + What’s a hashmap?
    - It’s a list of key-value pairs where the keys are hashed values
    - Often an array of some sort
  + Why?
    - Super fast (with a good hash, instantaneous lookups)
    - Should eliminate searching for an item and instead just almost give it to yourself
  + Wait, what if a hash returns duplicates?
    - Then that has to be implemented in code. Usually they’re stored in the next available slot instead, or each slot is a linked list
  + Examples
    - Let’s suppose people enter a lottery, and winners are notified of the amount by phone number
      * I can hash the phone number to get a unique value
      * I store the amount they won to that value
      * Now I can easily look up their phone number (something normally time consuming to search for) and pull up their winnings really fast
  + When might I use this?
    - Counting things. (number of words? Hash the word, store the count)
    - Find duplicates
    - Finding subsets of things. (throw hashes into table, can sort table, then quickly look for different elements)
    - Quite possibly could be used for a good amount of things where our natural instinct is to use another structure
  + Dont use this…
    - If you need to add a huge number of items (as the table will grow too big)
    - with a poor hash function
    - If something is going to make items collide in the same spot a lot
* Graphs (directed and undirected, weighted and unweighted)
  + a non-linear data structure with nodes (vertices) and edges
  + they represent some sort of network
    - network of roads/cities
    - network of people/relationships
    - computers/locations
  + Directed/Undirected:
    - Undirected just connects nodes/verticies together
    - Directed means traversing can only go one way (ex: a one-way road would be a directed edge)
    - Bi-directional between nodes means you can go either way (and usually has two direction lines)
    - (think one-way roads vs. two-way roads)
  + Weighted/unweighted:
    - Unweighted just connects nodes/vertices together
    - Weighted puts a value to each node, which can affect how it’s traversed
    - (think miles between cities)
  + How are they stored?
    - You might link some form of linked list or tree, but there’s even easier
    - Use a table of nodes vs nodes. For each intersection, record it in the table. Intersections of row/col that don’t have a value don’t connect
  + Why?
    - Trees only work so far, sometimes you need a more larger or non-linear structure
    - Traversing them can help determine paths through networks
    - Trees don’t let you backtrack
    - trees have a starting point, graphs do not
* BREAK!

Part 2: The algorithms

* Big O notation
  + https://www.bigocheatsheet.com/
  + It’s a measure of complexity, but has no units
  + Just tells us relatively how long an algorithm will take if it ran infinitely
    - Some say it’s the “cost” of an algorithm
  + Why “Big O”? O = order of a functions
  + (There’s others, like Little o, Theta, Omega, etc. but we won’t cover those)
  + For the *most* part: this doesn’t matter in day to day work. It will only matter for large data sets (think sorting 1000+ things)
  + It’s based around the “number” n
    - Every statement is 1 step → O(1)
    - Multiple statements are counted like 1 step → O(1)
    - A loop is iterative, so it’s counted like n steps → O(n)
    - Two loops in a row: n + n, or 2n, rounds to n → O(n)
    - …
  + Just drop the constants
    - 3n^2 + 4n + 6 should just be n^2 + n
  + Ex: Fibonacci
    - Recursively: it’s usually what people teach recursion with, but it’s terrible
      * public int fibonacci(int n) {
      * if(n == 0)
      * return 0;
      * else if(n == 1)
      * return 1;
      * else
      * return fibonacci(n - 1) + fibonacci(n - 2);
      * }
    - Array/list/loop: it’s easy AND efficient
      * loop forward
* Searching
  + Binary search (with binary search tree)
    - Go left for smaller, go right for larger
    - O(lg n)
  + Breadth-first search
  + Depth-first search
  + Which one?
    - Depends on how the tree is stored and what you are looking for
* Sorts
  + Why learn these? Most computers and libraries do this for you
  + Bubble
    - Super easy, 2 for loops
  + Merge
    - Split into many tiny lists, merge them together in the correct order
  + Quick
    - Pick a pivot element, then split into two lists
  + Insertion
    - Simple, tries to insert the current element in the right order
  + Selection
    - Goes through and “selects” the smaller item to put in its place. Keeps a sorted/unsorted split in list
  + Heap
    - Basically uses a heap to sort.
* Problems: (and Big O if you know it)
  + Remove a middle node out of a linked list
  + Swap two nodes in a linked list
  + How could you modify a stack that could add a function to tell you the minimum item in it?
  + How could you make animal shelter that’s based on a first in, first out adoption strategy, but they could pick which animal they wanted (dog/cat)?
  + How might you try to find the shortest distance between two nodes on a graph (assuming direct connections)?
  + How might you find a common ancestor between two people in a tree?
  + How might you find a way to traverse through a maze? (use a brute force method)
  + Given a string, how might you count the maximum number of consecutive characters?
  + Find min, max, and average of linked list of numbers
  + Think through how one might do undo and redo in an application
  + How would you “zip” a linked list? (A-B-C, D-E-F => A-D-B-E-C-F)
  + What kind of structure would you use to implement a Choose Your Own Adventure game and what would it look like?
  + How would you build something to determine what each dot and dash of Morse Code turns into a letter?
  + Using a Stack or Queue, how might you remove a letter out of a string?
  + With a linked List of numbers, move all of one particular number to the end (ex: move 1 to end of 3-5-1-6-1-2 so it’s 3-5-6-2-1-1)
  + Given a list of cities that connect, how might you find if you can drive from one to another?
  + Given a family tree, how would you determine who someone’s cousin is (if they have one)?
  + Count the number of times a word appears in a file, then print out if any words appear the same number of times. (ex: “to be or not to be” would show that to and be both appear twice)

Part 4: Job interviews and strategies

* We know job interviews suck
* We know code interviews can suck because they can be simple (merge two arrays) or complex (find the prime factorization of a large number)
* The most important thing: Talk through *everything!* Even if you don’t know what you’re doing, ramble what’s floating around in your head. It at least helps prevent the illusion you’re 100% stuck, even if you are 100% stuck.
* Run through data structures in your head. (the point of this talk!)
* Problem solving strategies (via Cracking the Coding Interview):
  + Listen to the problem.
    - Make sure you heard it correctly by trying to write out the basic things they’re looking for.
    - Find any of the issues you see (missing info, conflicting info, etc.).
    - Look for any missing details or any confusing points that don’t seem right.
    - Try to write out the basic things they’re looking for.
  + Draw an example:
    - They probably provide an example. If not, come up with one.
    - Walk through it to see if it makes sense.
    - Make sure the example is big enough to make sense and test a few things.
  + Find that brute force solution:
    - this might be the easiest way to find an algorithm.
    - Work it through in your mind to find how to do something on an example problem.
    - Sometimes this may be the ugly solution, but if it works, start with that.
  + Optimize (depending on time)
    - If there’s unused info, try to incorporate that. (ex: is data sorted?)
    - Use a fresh example to try to see if it changes how you think about the problem.
    - You could even try “incorrectly” solving the problem (which could result in correctly solving it).
    - And can you use a hash table?
  + Walk through:
    - Look at the steps again to see if it makes sense and if your understanding of the algorithm is clear. (Especially since whiteboarding is so slow.)
  + Implement it:
    - Code it up!
    - For whiteboards, start in the top left and work your way down.
    - If you can, add in error checks (or a comment to deal with it in a bit.)
    - And try to use good variable/function/class names.
    - Bonus: Try to cut down some writing, like make a fake function that would initialize a data structure, for example.
  + Test.
    - Make sure it works before “turning it in.”
    - Look at code to see if conceptually it makes sense.
* Job hunting strategies:
  + geekygirlsarah.com/job-hunt
  + Remember your self care
  + Remember interviews work both ways (you’re trying them out to see if they are a fit too)
  + No one’s perfect, and interviews rarely go perfect (which is fine)
  + Interview methods often reflect how the teams work too. (Bad interview process? The team process might be bad too)
  + Prep yourself properly
  + Give yourself time to reflect
  + Timebox what you’re working on.

Resources:

* All these notes: geekygirlsarah.com/algorithms/
* Cracking the Coding Interview (Gayle Laakmann McDowell)
* The Imposter’s Handbook – A Primer for Self-Taught Developers (Rob Conery, <https://bigmachine.io/products/the-imposters-handbook>)
* Geeks for Geeks (<https://www.geeksforgeeks.org/>) (for algs, DS, more)
* Big-O Cheat Sheet (<https://www.bigocheatsheet.com/>)