**Data Structures**

Find their good and bad at different operation. I have to note all the cases and efficiency for Java data structures.

.add (), .get(), .sort(), .search() are called big O notation. This is like measuring operational skills.

1. Linked list:
   1. Good: adding new items, deleting items
   2. Bad: retrieval, searching
2. Array:
   1. Good: retrieving items
   2. Bad: adding items (lower level language like c++, Java you have to declare the size of an array in advance, but in python or JavaScript it does it automatically)
3. Hashtable: its object in JavaScript and Dict in Python. Key has a hashing function that addresses a memory address. The memory location must not to be next to each other. They can be anywhere.
   1. Good: No problem with increasing size. Good at add, remove, retrieving
   2. Bad: collision: depends of hashing algorithm two key could hash/point out same memory location. There are different way to resolve this.
4. Stack/ Queue: Stack: LIFO (many books kept on top of others), Using .push, .pop you can add remove items., Queue: FIFO (queue in the supermarket), using .enqueue, dequeue you can add or remove items. Every language keeps track of the function that has been called which is called call stacks.
   1. Good: Efficient add, remove
   2. Bad: has limited use cases compare to other data structures.
5. Graph/Tree: Graph is kind of similar to linked list where a node is pointing to other node excepts that the pointer is called the edges. Edges can also have weights or numbers assigned to them. Imagine two cities Berlin and Hamburg, the road between them is edge and the length of the road is weight. Complicated relationships like social media networks are stored as graphs.

Hierarchical graph called tree where it has where the data expands in one direction where it can have family tree (parents, child or nested elements). Or even more specific tree called binary search tree. This tree has really specific rules but these rules allow us to things like searching really fast. The rules are follows: each node can two children left and right where left can contain less value and right can contain more. With these rules we can go and search an element really fast.

* 1. Bad: But if you add data unbalance or weird order, it can lead losing lot of the advantage.
  2. Good: There are self balancing trees but they are advanced

Make a document when is the right situation to use which one? Search when to use Linked list.

I need to make a list of many predefined or established data structure’s big O notation. I have to figure out all the calculation techniques of Big O notation with code example.

Recommendation to see

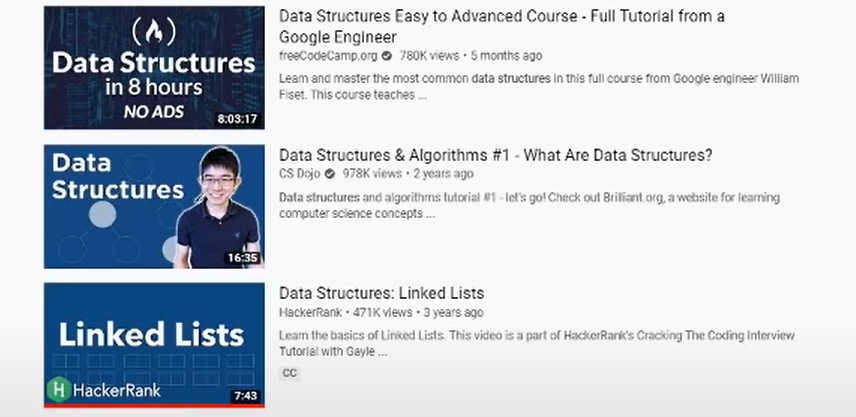
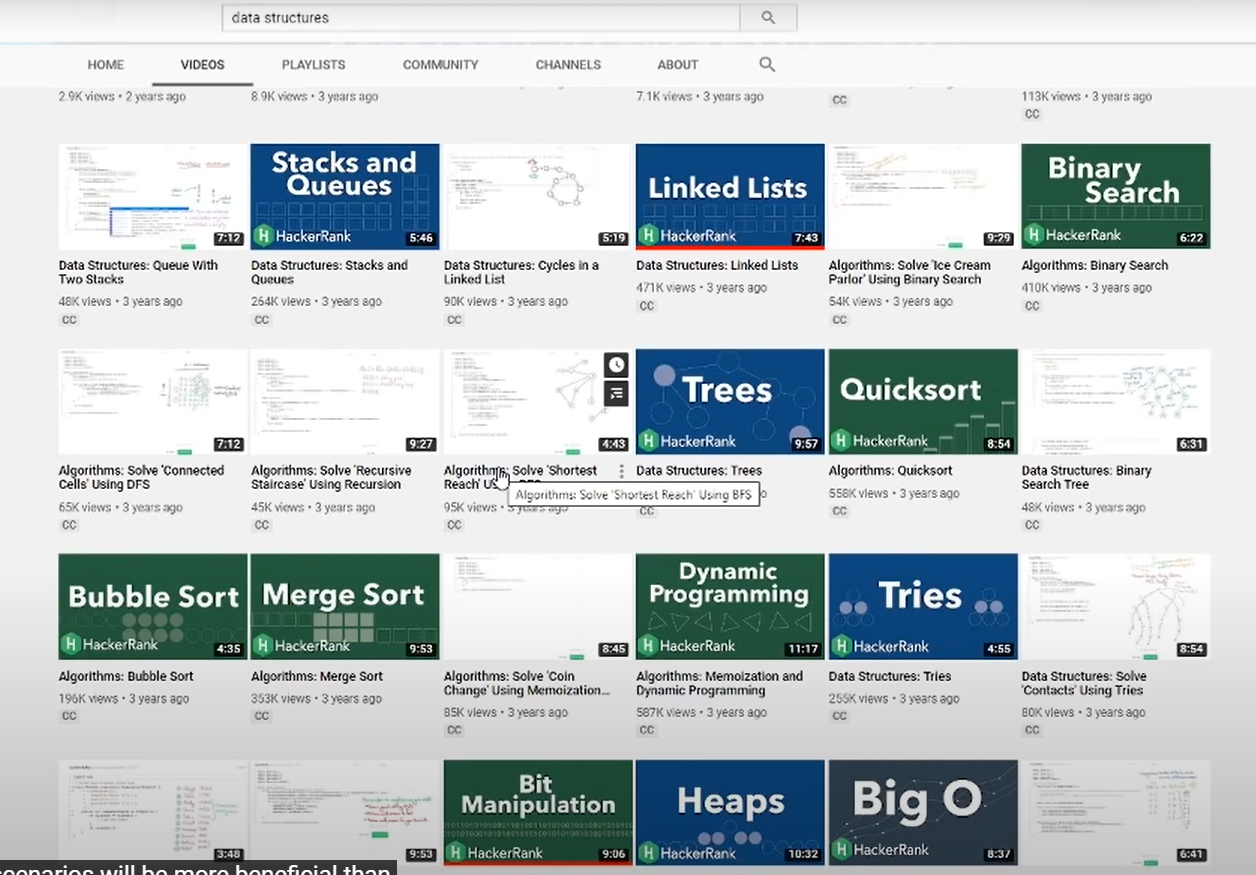
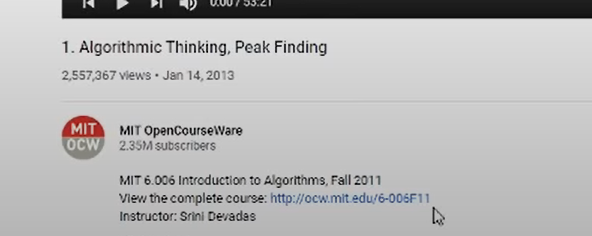


Figure out which data structures are more beneficial than others in certain scenario,



Courses by Eric Demaine are even better than Courseera course Algorithems Specialization



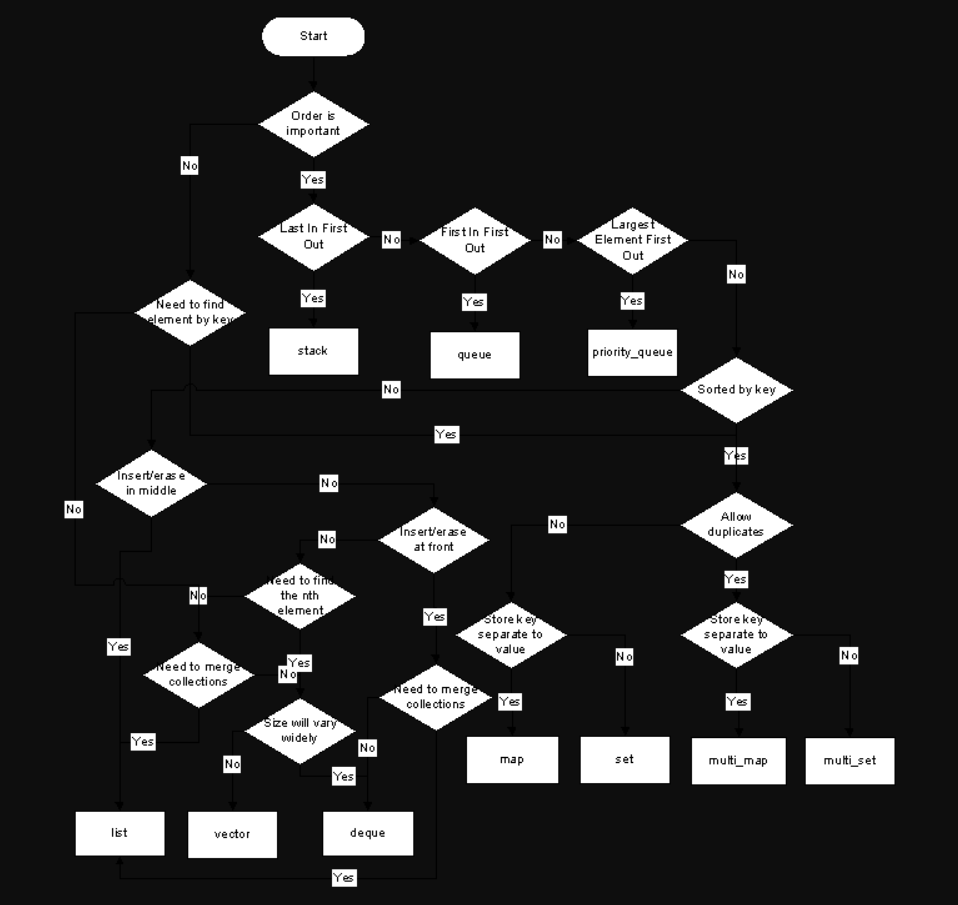
You comfort ability level should be starting text editor and start to implement a data structure like linked list.

Geekandgeeks has great image of that kind of data structures. And then you should try to compare and able to say why should I used a list over an array in that case.

Then use Pramp for random engineers.

Buy cracking the coding interview.

**Which data structure has to be used in which scenario? Make a list of scenario and make sure to demonstrate all the data structures benefit and when to use which one with big O notation.**

****

I am trying to be as simple as possible. Try Oracle Docs fro further details

**Struct:** When ever you need Object like structure, where you can group related data, use structs. Structs are very rarely used in java though(as objects are created in their place)

**Arrays:** Arrays are contiguous memory. when ever you want fixed time access based on index, unlike linkedlist, arrays are very fast and so use them.

But the backlog with arrays is that you need to know the size at the time of initialization. Also arrays does not support higher level methods such as add(), remove(), clear(),contains(), indexOf() etc.

**List:** is an interface which can be implemented using Arrays(ArrayList) or LinkedLists (LinkedList). They support all the higher level methods specified earlier.

Also Lists re-sizes themselves whenever it is getting out of space. You can specify the initial size which the underlying Arrays or LinkedLists will be created, but whenever the limit is reached, it created the underlying structure with a bigger size and then copies the contents of the initial one.

**Queue or Stack:** is an implementation technique and not really a data structure. If you want FIFO implementation, you implement Queue on either Arrays or LinkedList(yes, you can implement this technique on both these data structures) <https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues>

**HashMap:** Hashmap is used whenever you want to store key value pairs. if you notice, you cannot use arrays or linked lists or any other mentioned data structure for this purpose. a key can be any thing from String to Object(but note that it has to be an object and cannot be a primitive) and a value can also be any object

google out each data structure for more details

**Fastest Sorting Algorithm**

**Quick sort**

**Fastest way to Insert in sorted Array**

**Fastest way to Delete in sorted Array**

**Bubble Sort** (complexity O (n)2, Space complexity O(1), stable)

**Selection Sort**(divide by 2, one sorted one unsorted, complexity O(n)2, but faster than bubble sort, space complexity O(1), not stable)

**Insertion Sort** (complexity O(n)2, but faster than bubble sort and selection sort, space complexity O(1), stable, quick for small lists, can accommodate more items it receives on, it is the best sorting technique)

**2nd level of Sorting**

**Merge Sort** (complexity nlogn, stable, but Space complexity O(N))

**Quick Sort** (complexity nlogn, worst case: O(n2), faster than merge sort because it recursively partitioning input log, not stable)

**Heap Sort** (complexity nlogn, divide the input into two part, one sorted one unsorted, keep unsorted section as a heap, its slower than quick sort, but the good thing is Space complexity O(1), not stable because its arbitrary rearrangement of elements to maintain the heap during the sorting process)

**3nd level of Sorting (Hybrid Algorithms)**

**Timsort** (complexity nlogn, Space complexity: O(n), stable, It’s a highly optimized merge sort)

**Introsort** (Hybrid of Quicksort, Heapsort and Insertion sort, complexity nlogn, Space complexity: O(logn), not stable as heapsort and quicksort are not stable)

**Java Data Structure Efficiency**

### [Using ArrayList or HashMap for better speed](https://stackoverflow.com/questions/18862997/using-arraylist-or-hashmap-for-better-speed/18863041)

The **ArrayList** has O (n) **performance** for every search, so for n searches its **performance** is O (n^2). The **HashMap** has O (1) **performance** for every search (on average), so for n searches its **performance** will be O (n). While the **HashMap** will be slower at first and take more memory, it will be faster for large values of n.

**Java standard data structures Big O notation**

**Java Collections Algorithm Efficiencies:**[**Source**](http://shop.oreilly.com/product/0636920027829.do)

ArrayList

* get, set: **O(1)**
* add, remove: **O(n)**
* contains, indexOf: **O(n)**

HashMap

* get, put, remove, containsKey: **O(1)**

HashSet

* add, remove, contains: **O(1)**

LinkedHashSet

* add, remove, contains: **O(1)**

LinkedList

* get, set, add, remove (from either end): **O(1)**
* get, set, add, remove (from index): **O(n)**
* contains, indexOf: **O(n)**

PriorityQueue

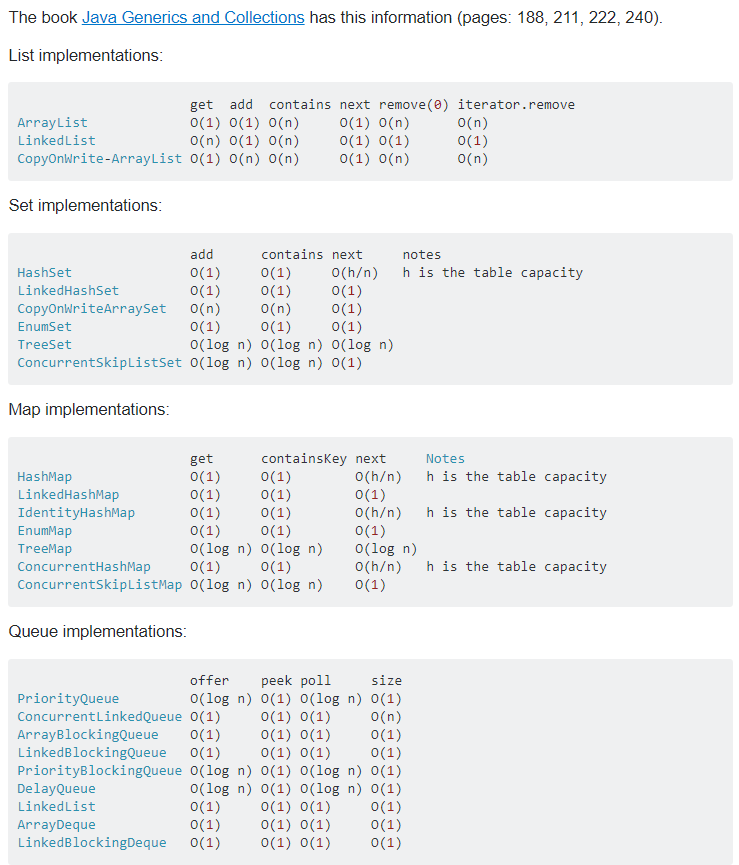
* peek: **O(1)**
* add, remove: **O(logn)**

TreeMap

* remove, get, put, containsKey: **O(logn)**

TreeSet

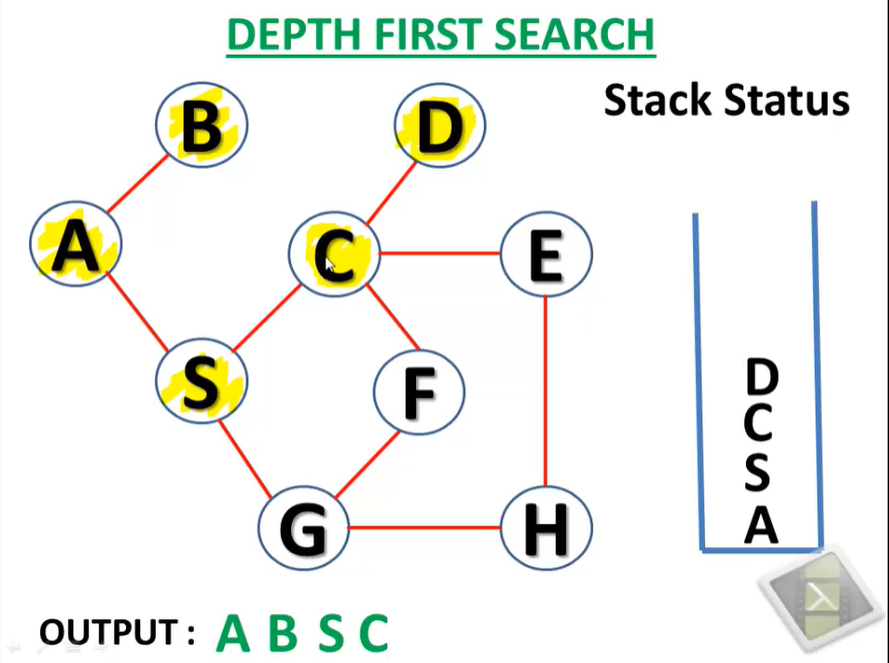
* add, remove, contains: **O(logn)**



**DFS (Depth First Search) Stack based LIFO**

It will push current node to the stack and goes alphabetically until leafs. It will pop it from the stack when it has no more connected node and the current node will came back to top node of the stack.

Starting with A, it will push A to the Stack and add in the output sequence. Alphabetically it will go to B as adjacent node and push B on the top of the Stack and add in the output sequence. B has no more adjacent node that is why it will come back to A and pop B from the stack. As a next option A will go to S and push S to the top of the Stack and add in the output sequence. From S, alphabetically C is the node to visit as next option and it will add C in the stack. C will direct to D as an next option and add D at the stack. As D has no more option to go, it will delete D from the stack.



How to store list of nodes consisting i and j,

List<Integer[]> unvisitedNeighbours = **new** ArrayList<Integer[]>();

How to keep track to visited nodes,

Boolean [][] visited = **new** Boolean[matrix.length][matrix[0].length];

Java has Stack to deal with DFS which allows push() pop(),

Stack<Integer[]> nodesToExplore = **new** Stack<Integer[]>();

**BFS (Breath First Search) Queue based FIFO**

We have to mark output sequence first and mark it visited (Let start with A). So currently working node is A. As BFS we have to check all adjacent unvisited nodes. We have B and S here, so we will en-queue B and S alphabetically. And we mark it as visited and add it to the output sequence. Now we have to check first element of the queue, we have B here. So, we updated currently working node B and de-queue B. Now, we find that we have only A is an adjacent node which we have already visited. So, we will update the pointer to S and de-queue S. Now, we see S has adjacent node C and G. So, en-queue S and G and put them in the output sequence. So C is the current working node in the queue as it came first. C will be de-queued and as for adjacent from C, DEF node will be en-queued below G and will be added to the output sequence.

