Common Data Structures

A screenshot of a cell phone

Description automatically generated

String data structure is immutable. Once value assigned, it is permanent can’t be changed. Say if we want to swap two characters in string , we have to create a new string. Whereas arrays are mutable, we can swap array elements and store in the initial array and don’t need to create a new one.

Arrays are stored in consequent memory locations.

Linked list has nodes which can be stored anywhere in memory and nodes connected to each other by pointers. Linked list can be underlying data structure for array, stack and queue.(object prototype chain is a linked list)

Javascript function calls follow Stack data structure. Message/ Events follow Queue data structure. Javascript engines have a call stack and message queue that executes our code in runtime.

Document object model follow tree data structure.

Focus on which element comes out first whether the last inserted or the first inserted. For stack the last inserted comes out first and for queue first inserted comes out first.

Stack – LIFO (push and pop)

Queue – FIFO (enqueue/push and dequeue/shift)

Array shift – remove element from beginning of array

Array unshift – add element to beginning of array

Array with push and pop is stack.

Array with push and shift is queue. You can keep track of these operations by a head and tail pointer node.

Stack, queue access element has constant time operation and that’s why they are super-fast.

Typically array unshift is linear time operation but its optimized in modern browsers.(details below)

Array shift – removes element from beginning of array then shifts all elements to one index before. So removing element is constant time operation as it is accessing 0th element but shifting index for elements is linear time operation as number of elements in array is equal to the index shifts. Most recent Browsers do the index shifting.

Back button in browser or undo button in text editor follow stack data structure.

So stack, queue and array are contiguous data structures.

Javascript has dynamic arrays, we don’t care about the length of array when we create it. We don’t care about memory management in javascript. But in other languages, we have to do memory management and that’s when linked list is very helpful.

A screenshot of a cell phone

Description automatically generated

Linked list can be used to implement a dynamic array. In linked list we traverse by next property as not all nodes are saved in contiguous memory location. The pointer concept is very important.

When we assign an object to a variable, the workflow is :

* There is an object in a memory location.
* The variable is created in another memory location
* The variable points to the memory location where the object is
* So the object does not sit inside the memory location where the variable is
* Another variable can be created and pointed to the same object in that memory location
* You can pass the object into a function as parameter but then also there will be a pointer to the memory location where the object is [Important concept Call by Reference(Memory location)]

When we assign a string/ number or any primitive data type to a variable, the workflow is:

* The variable is assigned a location in memory
* The variable is assigned a string or number value
* Both the value and the variable sit in the same memory location [Important Concept Call by Value]

So linked list is a sequential data structure.

Previously for shift in array, stack and queue we have to shift over all elements to the right of the deleted elements which can be a costly operation. But with linked list this is quite fast as we don’t have specific indices and we have pointers connecting all elements in memory. So to delete a node from beginning which can be costly for shift, is fast for linked list as we just change pointer link. Same for addition at the beginning.

Deleting node from middle or end can be costly(linear time) in linked list also if we don’t have a reference to the to be deleted node object. If we have the reference then it is a constant time operation. In these situations hash table is very efficient.

Searching in linked list can be slower operation(linear). If you know the reference to the node you want to delete or add or the node before or after it then its constant time operation.

So searching in linked list is a cons.

Similarly in array if you are searching for a value you have to loop through and mostly it’s a linear time operation. (Linear search O(n))

If the array is sorted then there can be more faster ways. (binary search O(log n))

A node(object) in linked list has two properties “value” & “next”. This is for a singly linked list which can only go in one direction. So it is a nested object starting from head object whose properties are value and next, which is another object having properties value and next, which is another object. The last object has a null value in next.

Linked lists are often the underlying data structures for stack and queue.

So the doubly linked list has a previous property in a node object. In a singly linked list you can never know which node is previous. So to traverse backwards we need a doubly linked list. The previous of head node in a doubly linked list should be null. Null should be assigned as this needs to explicitly defined to let other programmers know whereas undefined is set at runtime.

Linked list practical application is getting recent searches/ viewed from a cache.

Hash tables are not ordered but can be used for fast lookup. In array keys can be numbers only whereas in hash table keys can be strings, functions etc. Object, Set and map are hash table data structures in javascript.

So a lookup/ search operation can be constant time if we know which memory location our data is stored. But either we can search that memory location by traversing which is slow or we can mathematically calculate the location of the element in memory by a formula and look at that location directly which is a constant time operation.

This formula can be abstracted out which is a hash function.

A hash function converts a key to an integer within a predefined range which becomes the index of that key. It gives same output for the same input. This predefined range is the size of hash table. Generally if you define the initial size of the hash table keeping your data set size in mind then there is very rare chance of collision.

If your hash table is 50% full, then you should double the size of the hash table. There is a mathematical research stating this is the most efficient way. Determining when should we resize the hash table is depending on the number of collisions you are having. So a hash table collision is getting same indices from a hash function for more than one element. As the number of collision increases the efficiency of lookup for a particular element decreases as then it is not constant time and it is like a linked list/ array lookup depending on how you store collision elements. But generally with complex hash functions and resizing, this worst scenario where all items get hashed to one index and we have to loop through them is very rare. So this is called amortized time complexity(bad time complexity only in rare conditions).

<https://medium.com/@satorusasozaki/amortized-time-in-the-time-complexity-of-an-algorithm-6dd9a5d38045>

Object property lookup kind of follows hash table data structure and so it is fast and constant time operation. When we assign a value to a property of an object, the property goes through a hash function and then the integer returned becomes the index in the hash table in memory and the value we assign to the property is stored in the hash table in that particular index. When we intend to retrieve the value of the object property, the property again goes through a hash function and returns the same integer which is actually the index in the hash table and then by item lookup with that specific index, we can retrieve the property value in constant time operation.

So not only property assignment and lookup in object but deletion of property under the hood also uses hash table. Similar for map and set in ES6.

Whenever you need fast lookup you should always think about hash table data structure and hash table data structure in javascript or es6 is object/set/map. Depending on the use case you should use either normal objects or set or map.

Key-value pair where keys can be string => objects

Key-value pair where keys can be numbers/functions => map

Only need keys => set

Strings are immutable data types which means if you do any operation on string for example slicing, you are actually copying to a new string means new memory allocation and so you are increasing space complexity. So if we have to maintain constant space complexity we should use mutable data types like arrays where swapping of elements is done instead of creating new arrays .

Array have fast lookup(constant time operation) if you know the index of the looked up element. Appending data to end is constant time operation.

Inserting data in middle is slow as all elements to the right should be shifted over so time of operation depends on how many elements should be shifted over. Similarly for deleting data from middle or front. In these scenarios linked list is good.

Array is a contiguous blocks of memory and in javascript array is dynamic i.e. we don’t care about the memory allocation if array size grows.

Shallow copy/ deep copy javascript objects = <https://medium.com/javascript-in-plain-english/how-to-deep-copy-objects-and-arrays-in-javascript-7c911359b089>

Best case : Ω (Omega notation)

Average case: θ (Theta notation)

Worst case: O (Big O notation)

A screenshot of a cell phone

Description automatically generated

For every pointer in linked list a space is taken so doubly linked list takes more space.

In linked list important concept is pointer. All pointer objects initially should point to a single linked list node in a memory location. Then move the pointers to the other node of that same linked list so that reference does not change.

Space Time Complexity Cheat sheet:

<https://www.bigocheatsheet.com/>

Resources to prepare for interview :

<https://leetcode.com/>

<https://www.interviewbit.com/>

<http://www.crackingthecodinginterview.com/>

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Linked List insert at end is in general constant time operation but if you don’t have the reference to previous node then it is linear time operation as by using a while loop you have to loop through at the end.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

beginning and end in queue diagram is opposite

A screenshot of a cell phone

Description automatically generated

Non Linear data structure:

Trees:

A picture containing clock

Description automatically generated

It starts with a root object/node and then child objects/nodes. A child node which does not have a child is called leaf node. Let’s say root is a question and based on yes / no you have 2 child questions and so on. These question nodes go on until there is a recommendation. So this seems to be some problem happening recursively.

Examples of real life application of tree:

Document object model is a tree of element nodes.

Autocomplete feature in google uses a special kind of tree called trie.

Parsers uses tree(Abstract syntax tree) to model the structure of code.

Time complexity of traversing a tree is linear as the number of nodes in tree and number of hops to go to a particular node is equal in worst case.

Similar inserting a child and deleting a child is also linear as they need traversing if you don’t have reference to the node you are deleting or the parent node you are inserting into.

Similar linear complexity for counting leaf nodes as then also we have to traverse.

A tree with only two child is called binary tree. A binary search tree is a special type of binary tree where left node value is less than root and right node value is greater than root.

Binary tree traversal:

PRE ORDER: LEFT => ROOT => RIGHT

INORDER: ROOT => LEFT => RIGHT

POST ORDER: LEFT => RIGHT => ROOT

A graph has nodes called vertices and lines between vertices called edge.

A vertex has 2 main properties

Value : any primitive type

Edges []

How to represent a graph?

**Adjacency matrix:**

Relationship between vertices are maintained using a matrix.

A close up of a clock

Description automatically generated

We can add a new edge and vertex to a graph data structure.

Even if there is no edge a vertex can be added to a graph.

Adding an edge and removing an edge in adjacency matrix representation is constant time as you are only inserting/ deleting in/from a 2D array.

Adding a vertex here is linear as we have to add it like this

A close up of a keyboard

Description automatically generated

A new row in 2D array with 0s and a zero at the end of all existing rows to denote that this new vertex has no edge. So its linear time complexity.

**Adjacency List:**

A vertex has a list/ array of adjacent vertices.

A close up of a keyboard

Description automatically generated

All vertices can be objects.

Adding an edge means push the vertex in one end to the list of edges of the other vertex in the end of the edge and vice versa. So array pushing is constant time.

For removing an edge between 2 vertex, we need to find the index of both the vertices and delete the other vertex from its list. So here is searching going on and its linear time.

Array splice is deleting a portion of array so deleting is constant time but then all the elements indices need to move over. So it is linear time operation. In these scenarios linked list would have been better. So adjacency list needs to be a linked list. Then splicing is constant time operation.

Adding a vertex here means add an element in the existing object with empty array denoting no edges. So constant time operation.

Comparison:

Matrix takes up more space as it has lots of zeros in arrays which we don’t need. List takes up as much space as we need.

Adding a vertex in graph is linear time in matrix but constant time in list.

Tree is a type of graph.

Depth first search uses a stack and breadth first search uses a queue.

Shortest path is breadth first search as you are traversing in levels whereas in depth first search you are covering one side first so unless the path you are finding is in first iteration, it is slow.

Graph interview application:

A screenshot of a cell phone

Description automatically generated

Other types of graphs:

A screenshot of a cell phone

Description automatically generated

Linear search is normal array loop so time complexity is linear O(n).

Binary search on array always can be applied if the array is sorted, as then only if we cut the array in half we will know if our search is in left or right. So this cutting in half will go on and each time the array will become smaller until we find our search. Time complexity of binary search is O(log n) with base 2 as we are cutting in half. Kind of generic practice if our data structure gets halved then that algorithm has a logarithmic with base 2 time complexity.

Divide and conquer Sorting algorithms have a time complexity of O(n log n)

Binary search tree is similar as binary tree data structure where a tree can have only 2 children. Binary search tree has one extra feature which is all child nodes to the left of root should be less in value than root whereas all child nodes to the right of root should be greater than root.

Binary search on tree has a time complexity of O(h) where h is the height of the tree. The inserting order in binary tree will determine the binary search complexity as if we insert in order , one subtree is going to be very long so height of the tree will be huge resulting in almost same time complexity as linear search. This type of binary search tree where heights of subtrees differ much is called unbalanced binary search tree.

Insert and delete in a binary search tree would be same as search O(h) where h is the height of the tree.

Binary search tree is most useful if it is balanced and order is important for us. If it is not balanced then we can use hash table as its kind of becomes a linked list.

**ALGORITHMS:**

Space complexity is how much memory used.

Time complexity is how many primitive operations taken place to solve the problem with respect to input size. We always consider worst case scenario for complexity measurement. Worst case scenario is denoted by Big O notation. There are also average case and best case scenario. [Other Notations](https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/) .

[Time complexity Cheat Sheet](https://www.bigocheatsheet.com/) .

If there is a list of prices and we have to find the min and max, let’s say we compare each price with every other price value so here the time complexity is quadratic(O(n2)). This is nested loop.

Second approach if we do a loop for min and a separate loop for max price. Then the time complexity will be n + n = 2n. It is linear time O(n) complexity.

Now 3rd approach if the list of prices is sorted, then max and min would be constant time operation as we can directly access the min and max by array indices. So time complexity is O(1).

A screenshot of a cell phone

Description automatically generated**A screenshot of a cell phone

Description automatically generated**

A close up of text on a white background

Description automatically generated

**For loop : O(n)** => as the input size grows iteration increases proportionally

**Pop: O(1)** => always take out the last one. In javascript since it is dynamic array so we don’t need to allocate space in memory for a newly created array or resizing when the array size reaches a predefined size. If we would have been doing these then it would not be a constant time operation.

**Array For each , map, reduce**: **O(n)** => underlying they are all loops so linear but may change depending on what operation is happening in the callbacks.

**Accessing any element in array if we know the index e.g.: a[0] : O(1)** => constant time if we know the index.

**Property lookup in object e.g.: const object = {a: “1”}; object.a : O(1)** => constant time operation. Computer is aware in which memory location it is stored. [These temporary data stored in memory. But database stored in hard disk. ]

**Array shift: O(n)** => As we delete the first element of the array then all elements should be shifted over by 1 place to the left. These shifting operations number is equal to the number of elements to be shifted. So Linear time operation.

**Array unshift: O(n)** => As we insert before the first element of the array then all elements should be shifted over by 1 place to the right. These shifting operations number is equal to the number of elements to be shifted. So Linear time operation.

**Sort : O(n logn) =>** if it is a divide and conquer sort

Logarithmic time complexity:

Log2n = x

2x = n

|  |  |  |
| --- | --- | --- |
| base | n(number of i/p) | log n |
| 2 | 1 | 0 |
|  | 2 | 1 |
|  | 4 | 2 |
|  | 8 | 3 |
|  | 16 | 4 |
| 3 | 1 | 0 |
|  | 3 | 1 |
|  | 6 | 2 |
|  | 9 | 3 |

Log can have different bases.

As your data set size increases, the number of operations done on that data set increases by a fraction which is slow.

Calculate overall time complexity:

If operations happening one after another then add up complexity.

If one operation happening inside another multiply complexity.

Another point to consider is what is happening inside a loop say for recursion inside a loop, then also multiply the time complexity of recursive method with number of times the loop will run (n).

When you have an array and with operation you are cutting array in half or rather say you have to operate on smaller data set, the time complexity is logarithmic O(log n). Binary search on array is O(log n) with base 2. So in each recursion we divide the array by half(2). If it would have been divided by 3 then complexity will be O(log3n).

When you have a loop and in each traversal you are cutting your array in to half then the time complexity is n \* log n = O(n log n).

A screenshot of a cell phone

Description automatically generated

It is not necessary that always input will be length of array or similar, there can be multiple input which may change size so to calculate time complexity we should take all these in consideration.

Space complexity is how much memory your program is taking up. For example if for every iteration in an array we are copying to a new array, so a new array reference is created in memory for each iteration means space complexity is O(n) linear.

Call stack in recursion takes up memory.

What is the time complexity of this?

A screenshot of a cell phone

Description automatically generated

If it would have been the case that we have to loop through the string [array of chars] then it would be linear but javascript is smart to keep track of the length property when chars are inserted/deleted from array. So length is just a property of the object string. And property lookup in object is constant time operation. O(1).

Unique Sort => convert an array with duplicates to unique array

**Approach 1:** compare each item with another => time complexity: Quadratic O(n2)

**Approach 2:** do a loop and save each element in an object with value true assigned, and in each iteration check that object if it has the element and return false to denote array is not unique and if not in object add it and continue iterating. So this is linear time operation O(n). We are caching values in object to reuse later. Memoization is a type of caching. Only difference in memoization is if this value that we are storing in the cache object had been returned from a function then this concept would have been called memoization.

Objects can only store keys as strings so have to careful while using object data structure but in es6 can use map which can store any data type in key.

While we are caching we are creating a trade-off between space complexity and time complexity. But if we are not working with big data then lowering time complexity should be the primary target.

*// Memoization with closure - previous function refactored to avoid updating variable in global scope*

*// we are storing the big return in memoTimes10*

*// the returned function is closing over variable cache.*

*// memoTimes10Func is called once and then cache is defined*

*// rest of the time we call the function we are getting returned*

*// Since the returned function was defined in a scope where it has another variable cache,*

*// it remembers the cache variable until that scope is not destroyed.*

const **memoTimes10Func** = () => {

const cache = new **Map**();

return ((n) => {

if(cache.**get**(n)) {

console.**log**('getting from cache');

return cache.**get**(n)

} else {

console.**log**('calculating');

cache.**set**(n, **times10**(n));

return **times10**(n);

}

});

}

const memoTimes10 = **memoTimes10Func**();

console.**log**('Task 2 calculated value:', **memoTimes10**(9));*// calculated*

console.**log**('Task 2 cached value:', **memoTimes10**(9));*// cached*

console.**log**('Task 2 cached value:', **memoTimes10**(9));*// cached*

console.**log**('Task 2 cached value:', **memoTimes10**(9));*// cached*

console.**log**('Task 2 cached value:', **memoTimes10**(9));*// cached*

This caching technique has no improvement in this example because times10 is doing a constant time operation. But let’s say if times10 would have been doing an operation which has exponential time complexity then doing it multiple times would have a bad time complexity. In that case first time it is exponential when we don’t have the result in cache. But next time onwards if the input is same, instead of recalculating the exponential operation we can look-up in the cache map which is a constant time operation.

This type of caching is underlying using hash table as objects/ map/ set in javascript are hash table data structure. These caching optimizations are called dynamic programming concepts.

Recursion is an abstract technique where a function calls itself to avoid making your code WET(we enjoy typing/ write everything twice) and make your code DRY(Do not repeat yourself) until it hits an exit criteria. So recursion is just another cool way of looping.

Recursion is best understood by call stack.

How to populate your call stack?

A screenshot of a cell phone

Description automatically generated

If we don’t have a return or exit criteria in recursion then this call stack can be full and then we get the stack overflow error.

If a function does not have a return statement it implicitly returns undefined.

Recipe of recursion:

A screenshot of a cell phone

Description automatically generated

There is not much difference between recursion and loop from time and space complexity perspective but in some complex data structures like tree / graph where you don’t know how many levels you have to go; it is easy to use recursion than loops. If you have to solve these problems using loop, you use auxiliary data structure(stack/queue) where you keep track of all elements. This is more complicated than doing recursion and in recursion, automatically you stack your outputs in a stack.

Recursion has 2 patterns –

Wrapper function & accumulator

All recursive function calls will have a separate execution environment on their own so all private scoped variables will be newly created in that execution environment but all parameters passed to that function will still be accessible from parent scope.

In javascript whenever a function is created a closure is created combining of the function and the lexical environment within which the function is defined. So while executing that function, it still remembers the lexical environment from the closure while it was created. This is not the case for all languages.

Implementing any expensive function with memoization

e.g.: factorial

First we write a generic memoize function(using closure) which initially checks the cache if the output for the specific input exists then return it or else call factorial method and store the output for the specified number which we did factorial in cache. So next time if we do the factorial for that same number we can directly extract from cache.

Another type of caching can be done where if we do factorial of 8 for first time and second time do factorial of 10 then f(10), f(9) should be calculated and f(8) retrieved from cache.

Divide and conquer technique is we divide the data set in half and in the middle position we take a decision whether to go left or right. We continue doing this until we get the value we are searching for. For this to work the data structure should be sorted. Binary search is one example of divide and conquer. We can do binary search on array or a tree. Binary search has an O(log n) time complexity.

Sorting is a technique where at some point of time we have to look at each item. So the time complexity can never be less than linear.

2 types of sort high level  
Naive sorts – compare each item with other (Bubble sort, insertion sort, selection sort)O(n2)  
Divide and conquer sorts -divide the data set and do the operation (merge sort, quick sort)O(n log n)

These are all comparison sorts but also we have non comparison sorts. (search in internet)

Merge sort is the best sort amongst all sorting techniques. Javascript under the hood uses quick sort / insertion sort(for small array) in chrome’s v8 however Mozilla/Firefox uses merge sort.

Greedy algorithm is which gives a locally optimized solution instead of thinking the bigger picture. Greedy algorithm output is locally optimized so may not work for all types of input and that’s when people do brute force which is trying every possibility without any optimization and then it can have exponential time complexity. Brute force is not a realistic solution and should be optimized.

So comes the dynamic programming which a little optimization on the brute force. It is not recommended to do brute force due to its time complexity.

Any type of optimization related to caching is called dynamic programming.

Memoization was an optimization.

There are 2 optimization techniques of caching repeated values- top down recursive approach or bottom up iterative approach. (kind of related to factorial solution with memoization)

Array, stack, queue, linked list, hash table are linear data structures whereas tree and graph are non-linear. For non-linear we have to take into consideration the height of the data structure to calculate time complexity.

Tree: value, children[] traversal: breadth first, depth first

Binary Tree/ Binary search tree: value, left, right (nested) traversal: in-order, pre-order, post-order

Graph(adjacency list representation): value, edges traversal: breadth first, depth first

A tree can be represented in a graph as tree is a directed graph. Binary Tree => graph adjacency representation can be done by pushing left and right to an array.

In binary tree we insert like in a breadth first order for e.g. we check root left if empty insert otherwise check root right if empty insert. In each step we push child nodes to a queue and while checking a small tree we delete from beginning of queue. So this is kind of going level by level and inserting. While searching also kind of this logic of using a queue to track nodes and another hash map to track of visited nodes be used for breadth first traversal.

In Graph,

Depth-first-search uses a hash map named “visited” and a stack to push all nodes till max depth and pop out from last once visited.

Breadth-first-search uses a hash map named “visited” and a queue to push a smallest tree from top and delete from beginning once visited.

Merkle Tree in Blockchain:

<https://blockonomi.com/merkle-tree/>

Codility lessons:

<https://app.codility.com/programmers/lessons/1-iterations/>

Codility challenges:

<https://app.codility.com/programmers/challenges/>

Amazon codability Feedback: <https://app.codility.com/c/feedback/FNVKAE-2BK/>

Couldn’t understand output of this question in amazon Codility challenge:

<https://medium.com/@itoohue/breadth-first-search-by-example-5423be5778ec>