Data Structure

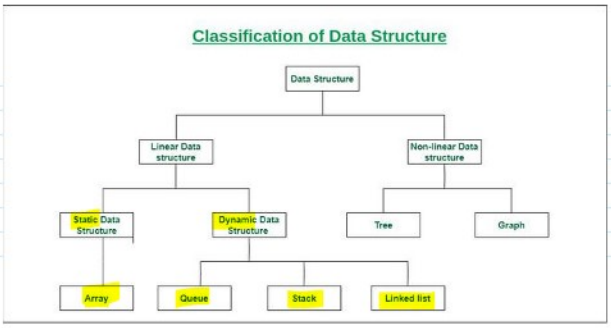
quarta-feira, 16 de outubro de 2024 12:43

Data Structure

Repo with a lot of references https://github.com/tayllan/awesome-algorithms

→ Define how data is organized, stored and manipulated
↳ Used to store and organize data

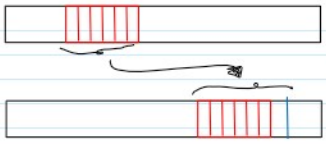
→ Data structures is a collection of data types
time complexity plays an important role



1. Linear Data Structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
   Example: Array, Stack, Queue, Linked List, etc.
2. **Static Data Structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
   Example: array.
3. **Dynamic Data Structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
   Example: Queue, Stack, etc.
4. **Non-Linear Data Structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.
   Examples: Trees and Graphs.

① Arranged sequentially

↳ A block of data → that has a limited amount of space but can't grow indefinitely without moving every thing over and over



In order to add one more item here, it needs to copy the hole memory

↳ Search time is faster but insertion time don't

Back To Basics: C++ Containers

→ This video explains it in more details

Watch it again!!!

→ COMPLEXITY → time x space

→ 3 ways to measure the complexity
↳ But the worst case is the most important one!!

CALCULUS → Count of statements that needs to execute

↳ Constant Complexity (O(1))

```
function askInfo() {
  console.log('Enter your name:');
```

→ 4 statements = O(4)
↳ Constant because it doesn't

```
function askInfo() {
  console.log('Enter your name:');
  console.log('Enter your age:');
  console.log('Enter your location:');
  console.log('Enter your profession:');
}
```

→ 4 statements = O(4)
↳ Constant because it doesn't depend on the input or process anything ↳ Call it O(1)

Linear Complexity (O(n))

```
function printItems(items) {
  for (let i = 0; i <= items.length; i++) {
    console.log(items[i]);
  }
}
```

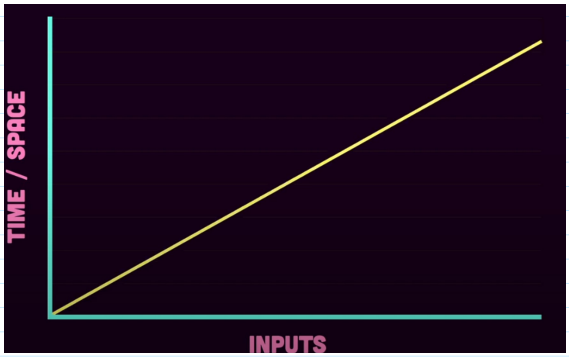
→ Complexity = ~~O(1)~~ + O(~~n~~N)
↳ O(n)
↳ Depends on the input length because the statements will repeat N times → Adding complexity

COMMON COMPLEXITIES

LINEAR	O(N) ③
QUADRATIC	O(N²) ④
CONSTANT	O(1) ①
EXPONENTIAL	O(2^N) ⑤
LOGARITHMIC	O(LOG N) ②

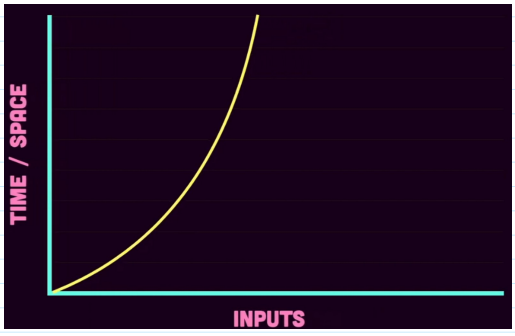
LINEAR O(N)

→ Grows linearly



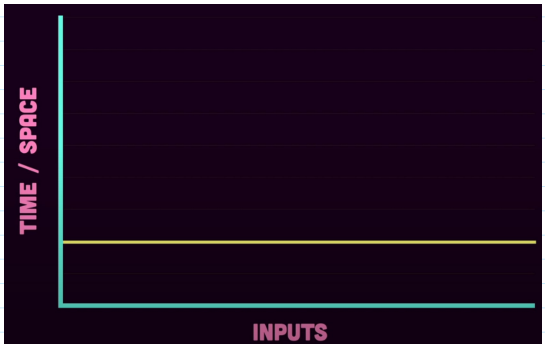

```
for (let i = 0; i <= n; i++) {
  // do something
}
```

QUADRATIC O(N²)




```
for (let i = 0; i <= n; i++) {
  for (let j = 0; j <= n; j++) {
    // do something
  }
}
```

CONSTANT O(1)



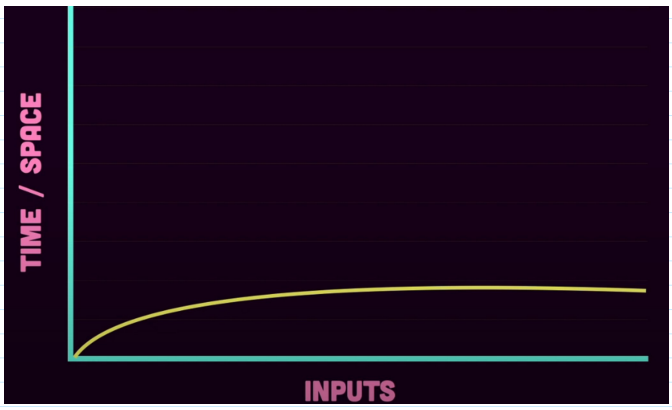

```
let names = ['John Doe', 'Jane Doe', 'Baby Doe'];
console.log(names[0]);
```

EXPONENTIAL O(2^N)


```
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 2) + fibonacci(n - 1);
}
```

função recursiva?

LOGARITHMIC O(LOG N)




```
for (let i = 1; i <= n; i = i * 2) {
  // do something
}
```

Binary search is also a good example

O-Notation → worse case (most important)

↳ Upper bound on the asymptotic behavior

↳ function grows no faster than a certain rate

$7n^3 + 100n^2 - 20n + 6 \rightsquigarrow n^3 \rightsquigarrow O(n^3)$

↳ Considers only the highest order terms because they are the most important ones

Ω notation → Best case

↳ Lower Bound on the asymptotic behavior

↳ function grows at least as fast as certain rate

$7n^3 + 100n^2 - 20n + 6 \rightsquigarrow n \rightsquigarrow \Omega(n)$

Θ notation → Average case

↳ Tight bound on the asymptotic behavior

↳ Grows precisely at a certain rate

$7n^3 + 100n^2 - 20n + 6 \rightsquigarrow O(n^3)$, $\Omega(n^3)$ → Can be !

"Grows no faster than"

$\Theta(n^3)$

```
INSERTION-SORT(A, n)
1  for i = 2 to n                          → O(n)
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key          → O(n)
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

$O(n^2)$

↳ The worse case would be 2 loops with N iterations

→ Nested loops