Departamento de
CIÊNCIA DA COMPUTAÇÃO

DCC302 - Estrutura de Dados I

Aula 01 - Conceitos Iniciais e TAD

Prof. Acauan C. Ribeiro

O que é uma Estrutura de Dados?

Uma estrutura de dados é uma forma de organizar os dados para que os mesmo possam ser utilizados de maneira eficiente.

Por que Estrutura de Dados?

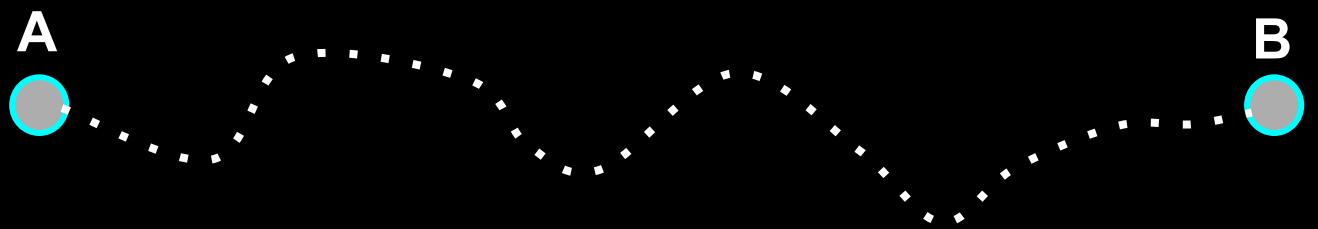
- Elas são ingredientes essenciais na criação de algoritmos rápidos e poderosos
- Podem ajudar a organizar e manipular os dados
- Eles fazem o código ficar mais limpo e fácil de entender.

Tipo Abstrato de Dados
vs.
Estrutura de dados

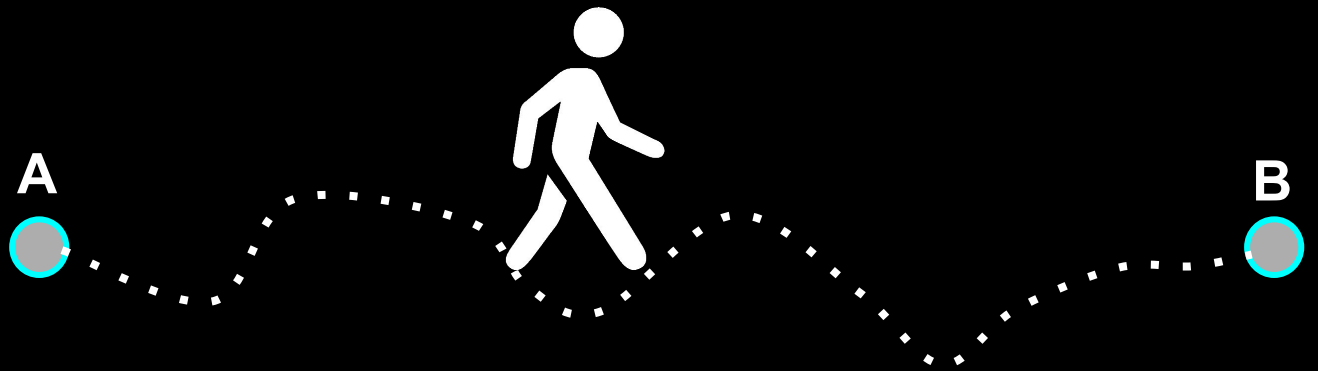
Tipo Abstrato de Dados

- Um Tipo Abstrato de Dados (TAD) é uma abstração da estrutura de dados que fornece apenas a interface à qual uma estrutura de dados deve aderir.
- A interface não fornece nenhum detalhe específico sobre como algo deve ser implementado ou em qual linguagem.

Tipo Abstrato de Dados

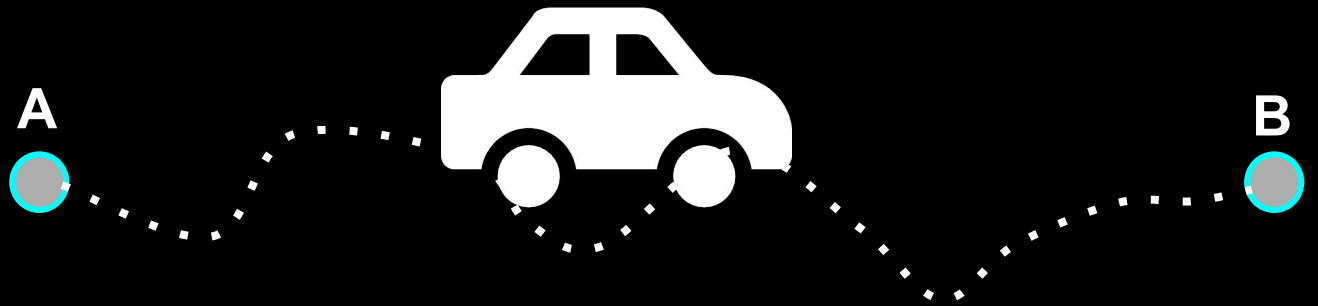


Tipo Abstrato de Dados



Tipo Abstrato de Dados

Tipo Abstrato de Dados



TAD (Exemplos)

Abstração (TAD)	Implementação
List	Array Dinâmico Lista Ligada
Queue (Fila)	Linked List based Queue Array based Queue Stack based Queue
Map	Tree Map Hash Map / Hash Table
Veículo	Carrinho de golf Bicicleta Carro

Análise de Complexidade Computacional

Análise da Complexidade

Como programadores, muitas vezes nos pegamos fazendo as mesmas duas perguntas repetidamente:

- Quanto tempo esse algoritmo precisa para terminar?
- Quanto espaço esse algoritmo precisa para seu cálculo?

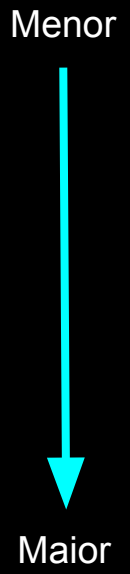
Notação Big-O

- A notação Big-O fornece um limite superior da complexidade computacional de um algoritmo no **pior** caso.
- Isso nos ajuda a quantificar o desempenho dos algoritmos conforme o tamanho da entrada torna-se **arbitrariamente grande**.

-> Exemplo de buscar um número numa lista.

Notação Big-O

n - O tamanho das Complexidades de entrada ordenadas do menor ao maior (n é o tamanho da entrada de dados)



Constant Time: $O(1)$
Logarithmic Time: $O(\log(n))$
Linear Time: $O(n)$
Linearithmic Time: $O(n\log(n))$
Quadratic Time: $O(n^2)$
Cubic Time: $O(n^3)$
Exponential Time: $O(b^n)$, $b > 1$
Factorial Time: $O(n!)$

Propriedade do Big-O

$O(n + c) = O(n)$

$O(cn) = O(n)$, $c > 0$

Seja f uma função que descreve o tempo de execução de um algoritmo específico para uma entrada de tamanho n:

$$f(n) = 7\log(n)^3 + 15n^2 + 2n^3 + 8$$

$$O(f(n)) = O(n^3)$$

Termo dominante

-> Exemplos práticos vão vir no futuro que ajudarão a entender =)

Exemplos Big-O

Os códigos seguintes rodam em tempo constante: **O(1)**

```
a = 1;
b = 2;
c = a + 5*b;
```

```
i = 0;
while i < 11 do
    i = i + 1;
```

Ninguém aqui depende do **n**. Mesmo o loop, não tem nada a ver com n.

Exemplos Big-O

Os códigos seguintes rodam em tempo linear: **O(n)**

```
i = 0;
while i < n do
    i = i + 1;
```

f(n) = n

O(f(n)) = O(n)

```
i = 0;
while i < n do
    i = i + 3;
```

f(n) = n/3

O(f(n)) = O(n)

Exemplos Big-O

Os dois códigos seguintes rodam em tempo quadrático. O primeiro faz o óbvio fazendo com que *n* se repita *n* vezes *n***n* = **O(n²),** mas o o segundo?

```
for (i = 0 ; i < n ; i++)
    for (j = 0 ; j < n ; j++)
```

f(n) = n*n = n^2, O(f(n)) = **O(n^2)**

```
for (i = 0 ; i < n ; i++)
    for (j = i ; j < n ; j++)
```

trocou-se 0 por i

Exemplos Big-O

Por um momento, concentre-se apenas no segundo loop. Como i vai de [0, n), a quantidade de looping realizada é determinada diretamente pelo valor de i. Observe que se i = 0, fazemos *n* vezes, se i = 1, fazemos *n-1* vezes, se i = 2, fazemos n-2 vezes, etc ...

Portanto, a questão passa a ser o que é:

(n) + (n-1) + (n-2) + (n-3) +... + 3 + 2 + 1?

Notavelmente, isso acaba sendo n (n + 1) / 2, então

O (n (n + 1) / 2) = O ($n^2/2$ + n / 2) = O(n^2)

```
for (i = 0 ; i < n ; i++)
    for (j = i ; j < n ; j++)
```

$$\sum_{i=1}^{n} x_i = \frac{n(n+1)}{2}$$

Exemplos Big-O

Suponha que temos um array ordenado e queremos encontrar o índice de um determinado valor no array, se ele existir. Qual é a complexidade de tempo do seguinte algoritmo?

```
low = 0
high = n-1
while low <= high do
        mid = (low + high) / 2
        if array[mid] == value return mid
        else if array[mid] < value: low = mid + 1
        else if array[mid] > value: high = mid - 1
return -1 // valor não encontrado
```

Resposta: $O(\log_2(n))$ = **O(log(n))**

Exemplos Big-O

```
i := 0
While i < n Do
    j = 0
    While j < 3*n Do
       j = j + 1
    j = 0
    While j < 2*n Do
       j = j + 1
    i = i + 1
```

$f(n) = n * (3n + 2n) = 5n^2$

$O(f(n)) = \mathbf{O(n^2)}$

Exemplos Big-O

```
i := 0
While i < 3 * n Do
    j := 10
    While j <= 50 Do
       j = j + 1
    j = 0
    While j < n*n*n Do
       j = j + 2
    i = i + 1
```

$$f(n) = 3n * (40 + n^3/2) = 3n/40 + 3n^4/2$$

$$O(f(n)) = \mathbf{O(n^4)}$$

Exemplos Big-O

Encontrando todos os subconjuntos de um conjunto - **$O(2^n)$**

Encontrando todas as permutações de uma string - **O(n!)**

Ordenar usando mergesort - **O(nlog(n))**

Iterando sobre todas as células em uma matriz de tamanho n por m - **O(nm)**