

# AED

## Índice

AED .....	1
Análise da Complexidade de Algoritmos.....	2
Melhor Caso, Pior Caso e Caso Médio .....	2
Ordens de Complexidade .....	4
Algoritmos de Procura e de Ordenação .....	7
Linear Search – Procura sequencial num array não ordenado.....	7
Linear Search – Procura sequencial num array ordenado.....	8
Binary Search – Procura binária .....	10
Selection Sort – Ordenação por Seleção .....	14
Bubble Sort – Ordenação por Troca Sequencial.....	17
Insertion Sort – Ordenação por Inserção .....	20
Algoritmos recursivos .....	22
Decomposição em subproblemas .....	22
Calcular $x_n$ .....	24
Procura Binária – Versão recursive.....	25
Mergesort – Ordenação por Fusão .....	28
Programação Dinâmica .....	32
Memoization .....	33
Listas Ligadas .....	35
Árvores Binárias .....	39
Travessias Recursivas.....	40
Árvores Binárias de Procura (ABP) – Binary Search Trees (BST).....	41
Árvores equilibradas em altura – Balanced Trees .....	42
Filas com Prioridade - Priority Queues.....	45
MIN-Heaps.....	47
Dicionários / Tabelas de Dispersão.....	48
Hash Tables – Tabelas de Dispersão .....	48
Hash Tables – Open Addressing .....	49
Grafos .....	51
Grafos Orientados .....	53

Redes .....	54
Travessia em Profundidade .....	56
Travessia por Níveis .....	59
Ordenação Topológica.....	60

## Análise da Complexidade de Algoritmos

Melhor Caso, Pior Caso e Caso Médio

---

### Best case, Worst case

- $D_n$  = conjunto de instâncias de dimensão n
- I é uma instância de  $D_n$
- $t(I)$  = tempo de execução ou nº de operações para a instância I

$$B(n) = \min_{I \in D_n} t(I) \quad W(n) = \max_{I \in D_n} t(I)$$

## Average case

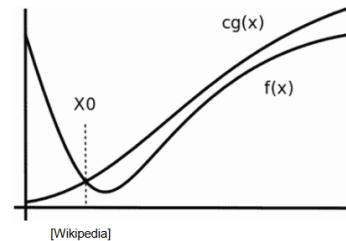
- $D_n$  = conjunto de instâncias de dimensão n
- I é uma instância de  $D_n$
- $p(I)$  = probabilidade de ocorrência da instância I
- $t(I)$  = tempo de execução ou nº de operações para a instância I

$$A(n) = \sum_{I \in D_n} p(I) \times t(I)$$

## Ordens de Complexidade

Big-Oh :  $t(n) \in O(g(n))$

- Majorante / Limite superior



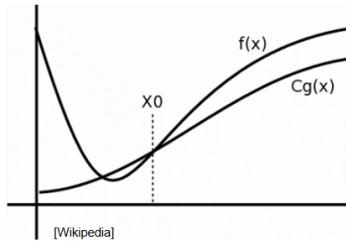
- $O(g(n))$  : conjunto de todas as funções com a mesma ordem de crescimento que  $g(n)$  ou com uma ordem de crescimento inferior

- $t(n) \leq c g(n)$ , para todo  $n \geq n_0$ ,  $c$  é uma constante positiva

- $t(n), g(n)$  : funções não negativas sobre o conjunto dos números naturais

Big-Omega :  $t(n) \in \Omega(g(n))$

- Minorante / Limite inferior

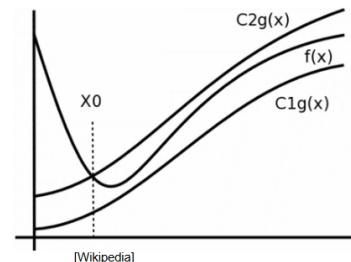


- $\Omega(g(n))$  : conjunto de todas as funções com a mesma ordem de crescimento que  $g(n)$  ou com uma ordem de crescimento superior

- $t(n) \geq c g(n)$ , para todo  $n \geq n_0$ ,  $c$  é uma constante positiva

## Big-Theta : $t(n) \in \Theta(g(n))$

- Enquadramento



- $\Theta(g(n))$  : conjunto de todas as funções com a mesma ordem de crescimento que  $g(n)$
- $c_1 g(n) \leq t(n) \leq c_2 g(n)$ , para todo o  $n \geq n_0$ ,  $c_1, c_2$  constantes positivas
- $t(n) \in O(g(n))$  e  $t(n) \in \Omega(g(n))$  ←

## Notação – Exemplos

notation	provides	example	shorthand for	used to
<b>Big Theta</b>	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2}N^2$ $10N^2$ $5N^2 + 22N\log N + 3N$ ⋮	classify algorithms
<b>Big Oh</b>	$\Theta(N^2)$ and smaller	$O(N^2)$	$10N^2$ $100N$ $22N\log N + 3N$ ⋮	develop upper bounds
<b>Big Omega</b>	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2}N^2$ $N^5$ $N^3 + 22N\log N + 3N$ ⋮	develop lower bounds

[Sedgewick & Wayne]

# Ordens de Complexidade/Classes de Eficiência

- $O(n^k)$  : polinomial (quadrático, cúbico, etc.)
  - k ciclos encastelados
- $O(2^n)$  : exponencial
  - Gerar todos os subconjuntos de um conjunto com n elementos
- $O(n!)$  : fatorial
  - Gerar todas as permutações de um conjunto com n elementos

# Ordens de Complexidade/Classes de Eficiência

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$	
1	constant	<code>a = b + c;</code>	statement	add two numbers	1	
$\log N$	logarithmic	<code>while (N &gt; 1) {   N = N / 2; ... }</code>	divide in half	binary search	$\sim 1$	
$N$	linear	<code>for (int i = 0; i &lt; N; i++) { ... }</code>	loop	find the maximum	2	
$N \log N$	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	$\sim 2$	
$N^2$	quadratic	<code>for (int i = 0; i &lt; N; i++)   for (int j = 0; j &lt; N; j++)     { ... }</code>	double loop	check all pairs	4	
$N^3$	cubic	<code>for (int i = 0; i &lt; N; i++)   for (int j = 0; j &lt; N; j++)     for (int k = 0; k &lt; N; k++)       { ... }</code>	triple loop	check all triples	8	
$2^N$	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$	[Sedgewick & Wayne]

## Algoritmos de Procura e de Ordenação

Linear Search – Procura sequencial num array não ordenado

---

### Procura sequencial – Comparações ?

```
int search( int a[], int n, int x ) {  
    for( int i=0; i<n; i++ ) {  
        if( a[i] == x ) { ← B(n) = 1  
            return i;          W(n) = n  
        }                      A(n) = ?  
    }  
    return -1;  
}
```

### 2 – Valor procurado pode não pertencer !

Casos possíveis		Nº de comparações	Probabilidade
$I_0$	É o 1º elemento	1	$p/n$
$I_1$	É o 2º elemento	2	$p/n$
$I_2$	É o 3º elemento	3	$p/n$
...	...	...	...
$I_{n-1}$	É o último elemento	$n$	$p/n$
$I_n$	Não encontrado !	$n$	$(1-p)$

$$A(n) = \sum_{i=0}^{n-1} \frac{p}{n} \times (i+1) + (1-p) \times n = \frac{p \times (n+1)}{2} + (1-p) \times n$$

## Procura sequencial – Comparações ?

```
int search( int a[], int n, int x ) {  
    int stop = 0; int i;  
    for( i=0; i<n; i++ ) {  
        if( x <= a[i] ) {  
            stop = 1; break;  
        }  
    }  
    if( stop && x == a[i] ) return i; // Ordem da conjunção !!  
    return -1;  
}
```

## Caso Médio

Casos possíveis		Nº de comparações	Probabilidade
Sucesso 0	É o 1º elemento	2	1 / (2 n +1)
Sucesso 1	É o 2º elemento	3	1 / (2 n +1)
...	...	...	...
Sucesso (n – 1)	É o último elemento	n + 1	1 / (2 n +1)
Insucesso 0	Menor do que o 1º	2	1 / (2 n +1)
Insucesso 1	Entre o 1º e o 2º	3	1 / (2 n +1)
...	...	...	...
Insucesso (n – 1)	Entre o penúltimo e o último	n + 1	1 / (2 n +1)
Insucesso n	Maior do que o último	n	1 / (2 n +1)

- $A(n) = ?$

## Caso Médio

$$A(n) = \frac{1}{2n+1} \left\{ \left( \sum_{i=0}^{n-1} (i+2) \right) + \left( \sum_{i=0}^{n-1} (i+2) \right) + n \right\}$$

$$A(n) = \frac{1}{2n+1} \left\{ \frac{n \times (n+3)}{2} + \frac{n \times (n+3)}{2} + n \right\}$$

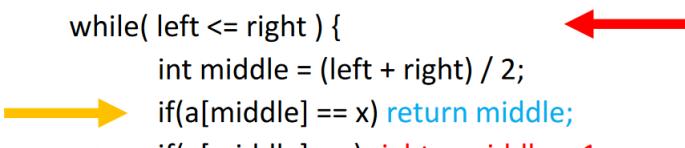
$$A(n) \approx \frac{n}{2}$$

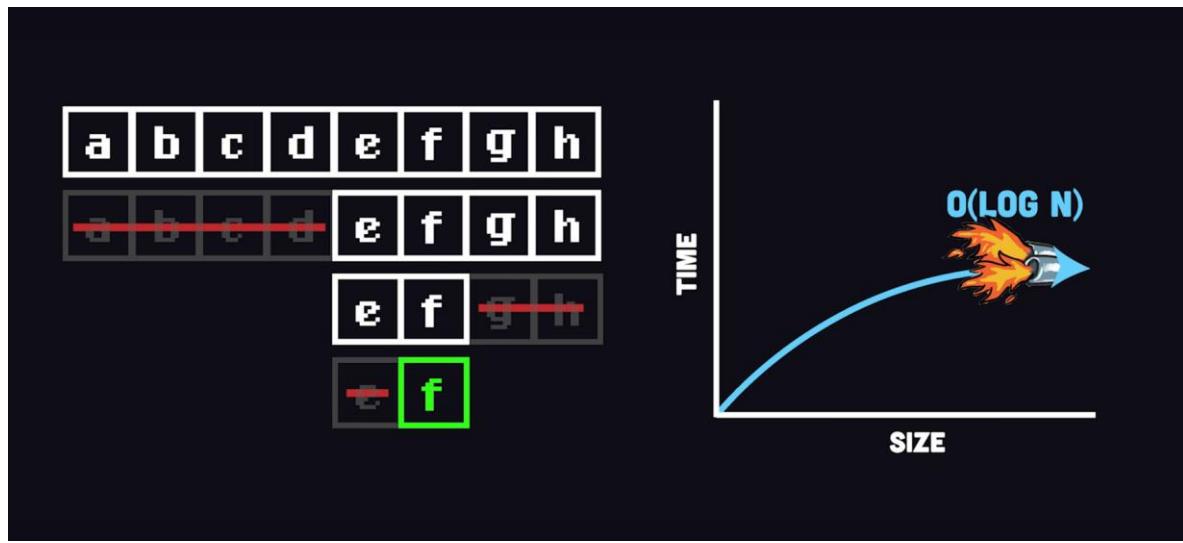
- Comparar com o cenário do array não ordenado
- É melhor ou pior ?

## Binary Search – Procura binária

### Procura binária – Nº de iterações do ciclo ?

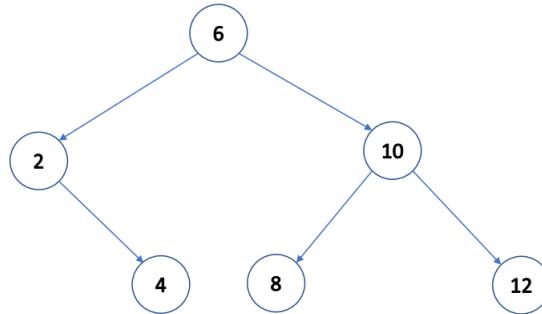
```
int binSearch( int a[], int n, int x ) {  
    int left = 0; int right = n - 1;  
    while( left <= right ) {  
        int middle = (left + right) / 2;  
        if(a[middle] == x) return middle;  
        if(a[middle] > x) right = middle - 1;  
        else left = middle + 1;  
    }  
    return -1;  
}
```





## Árvore binária

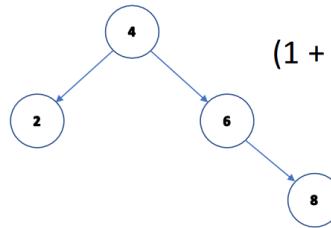
- A representação gráfica como árvore binária auxilia a compreensão do funcionamento do algoritmo



## Caso Médio – Situações mais simples

- $n = 4$

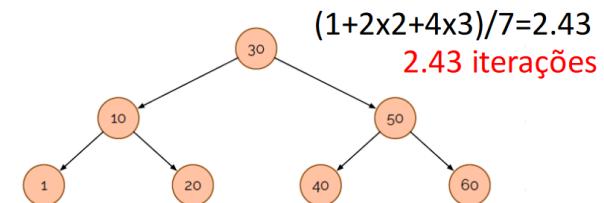
0	1	2	3
2	4	6	8



- ...

- $n = 7$

0	1	2	3	4	5	6
1	10	20	30	40	50	60



## Caso Médio – Generalização

- Valor procurado pertence ao array – **Equiprobabilidade**
- Caso particular :  $n = 2^k - 1$ ,  $k = \log_2(n + 1)$
- **Nº de níveis** da árvore binária =  $k$

$$A(n) = \sum_{i=0}^{k-1} \frac{1}{n} \times 2^i \times (i + 1) = \frac{1}{n} \left[ \sum_{i=0}^{k-1} i \times 2^i + \sum_{i=0}^{k-1} 2^i \right]$$

↑      ↑

# Caso Médio – Generalização

- Expressões auxiliares

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1 \quad \sum_{i=0}^{k-1} i \times 2^i = 2^k(k-2) + 2$$

$$A(n) = \frac{1}{n} [2^k(k-1) + 1] = \frac{k \times 2^k - (2^k - 1)}{n} = \frac{k \times (n+1)}{n} - 1$$

$$A(n) = k + \frac{k}{n} - 1 = k - \left(1 - \frac{k}{n}\right) = \log_2(n+1) - \left(1 - \frac{\log_2(n+1)}{n}\right)$$

## Desempenho – Nº de comparações

Nº de elementos	Procura Sequencial		Procura Binária	
	A(n)	W(n)	A(n)	W(n)
$2^9 - 1 = 511$	256	512	17	18
$2^{10} - 1 = 1023$	512	1024	19	20
$2^{14} - 1 = 16383$	8192	16384	27	28
$2^{17} - 1 = 131071$	65536	131072	33	34
$2^{20} - 1 = 1048575$	524288	1048576	39	40
	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$

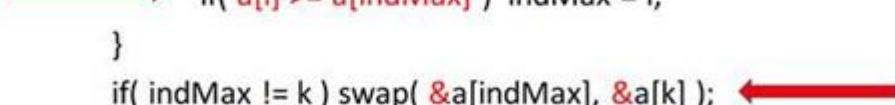
- P. Binária : nº de comparações  $\approx 2 \times$  nº de iterações
- Caso médio : valores aproximados
- Valores procurados podem não estar no array (Cenário 2)

## Trocar

```
void swap( int* a, int* b )      // Call-by-pointer
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

## Selection Sort

```
void selectionSort( int a[], int n ) {
    for( int k = n - 1; k > 0; k-- ) {
        int indMax = 0;
        for( int i = 1; i <= k; i++ ) {
            if( a[i] >= a[indMax] ) indMax = i;
        }
        if( indMax != k ) swap( &a[indMax], &a[k] );
    }
}
```



## Exemplo

0	1	2	3	4
7	2	6	4	3

7	2	6	4	3
3	2	6	4	7
3	2	4	6	7
3	2	4	6	7
2	3	4	6	7

- 1 + 1 + 0 + 1 trocas

Terminado !!

## Nº de Comparações

- Número **fixo** de comparações ! --- Algoritmo “pouco inteligente”
- Mesmo que o array já esteja ordenado, continuamos a comparar !!

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

**O(n<sup>2</sup>)**

## Nº de Trocas – Caso Médio

- $p(I_j)$  é a probabilidade de o elemento  $a[j]$  estar na posição correta
- Simplificação : **Equiprobabilidade** :  $p(I_j) = 1 / (j + 1)$
- $(1 - p(I_j))$  é a probabilidade de ser necessária **uma troca** para o elemento  $a[j]$  ficar na posição correta

$$A_t(n) = \sum_{j=1}^{n-1} (1 - p(I_j)) \times 1 = \sum_{j=1}^{n-1} 1 - \sum_{j=1}^{n-1} p(I_j)$$

## Bubble Sort

```

void bubbleSort( int a[], int n ) {
    int k = n; int stop = 0;
    while( stop == 0 ) {
        stop = 1; k--;
        for( int i = 0; i < k; i++ ) {
            if( a[i] > a[i + 1] ) { ←
                swap( &a[i], &a[i + 1] ); →
                stop = 0;
            }
        }
    }
}

```

## Nº de Comparações – Melhor Caso e Pior Caso

- Melhor Caso ? - Array ordenado
- $B_c(n) = n - 1$   $O(n)$
- Pior Caso ? - Array pela ordem inversa, sem elementos repetidos
- $W_c(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$   $O(n^2)$

## Nº de Trocas – Melhor Caso e Pior Caso

- Melhor Caso ? - Array ordenado

- $B_t(n) = 0$   $O(1)$

- Pior Caso ?
  - Array pela **ordem inversa, sem elementos repetidos**
  - 1 troca para cada comparação

- $W_t(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$   $O(n^2)$

## Nº de Comparações – Caso Médio

- Probabilidade ? Simplificação...

Nº de iterações do ciclo while	Nº de comparações realizadas	Probabilidade
1	$(n - 1)$	$1 / (n - 1)$
2	$(n - 1) + (n - 2)$	$1 / (n - 1)$
...	...	...
j	$(n - 1) + (n - 2) + \dots + (n - j)$	$1 / (n - 1)$
...	...	...
n - 2	$(n - 1) + (n - 2) + \dots + 2$	$1 / (n - 1)$
n - 1	$(n - 1) + (n - 2) + \dots + 2 + 1$	$1 / (n - 1)$

- Habitualmente, para **arrays aleatórios**, o número de iterações do ciclo while é próximo do seu número **máximo**

---

## Nº de Comparações – Caso Médio

$$A_c(n) = \sum_{k=1}^{n-1} \frac{1}{n-1} C(k) = \dots = \frac{1}{2(n-1)} \sum_{k=1}^{n-1} [(2n-1)k - k^2]$$

Expressão auxiliar:  $\sum_{k=1}^n k^2 = \frac{1}{6} n(n+1)(2n+1)$

$$A_c(n) = \frac{1}{3} n^2 - \frac{1}{6} n \quad \textcolor{red}{O(n^2)}$$

- Façam o desenvolvimento e confirmem o resultado

## Função Auxiliar – Inserção Ordenada

```
void insertElement( int sorted[], int n, int elem ) {  
    // Array sorted está ordenado  
    // Há espaço para acrescentar mais um elemento  
    int i;  
    for( i = n - 1; (i >= 0) && (elem < sorted[i]); i-- )  
        sorted[i + 1] = sorted[i]; ← blue arrow  
        sorted[i + 1] = elem; ← blue arrow  
    } ← yellow arrow
```

- Deslocamentos para a direita, para abrir espaço e inserir

## Insertion Sort

```
void insertionSort( int a[], int n ) {  
    for( int i = 1; i < n; i++ )  
        if( a[i] < a[i - 1] ) ← red arrow  
            insertElement( a, i, a[i] ); ← red arrow  
    }
```

- Deslocamentos (i.e., atribuições) são efetuados pela função auxiliar
- Contar as comparações feitas pela função auxiliar !!

## Comparações – Melhor Caso e Pior Caso

- Melhor Caso
  - $B_c(n) = n - 1$   $O(n)$
  - Array ordenado : A função auxiliar nunca é chamada
- 
- Pior Caso
  - $W_c(n) = \sum_{i=1}^{n-1} (1 + i) = \frac{n-1}{2} \times (n + 2)$   $O(n^2)$
  - A função auxiliar é **sempre chamada !!**
  - E tem **sempre o comportamento de pior caso !!**

---

## Comparações – Caso Médio

- Análise simplificada !
- Considera-se que em cada iteração a função auxiliar tem **sempre o comportamento do caso médio**

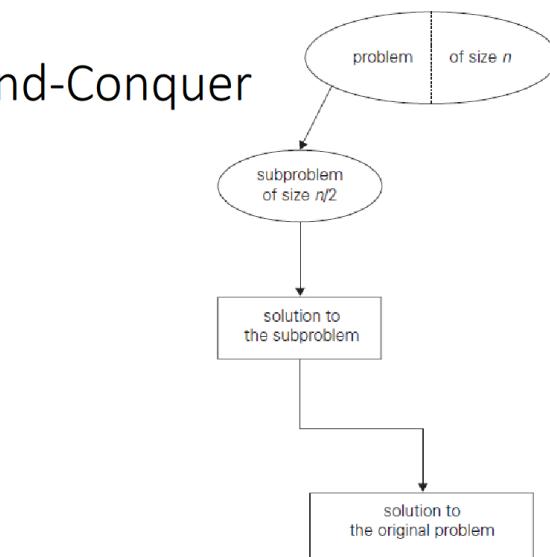
$$A_c(n) \approx \sum_{i=1}^{n-1} \left[ 1 + \left( \frac{i}{2} + 1 \right) \right] = \left[ 2(n - 1) + \frac{n(n - 1)}{4} \right]$$
$$A_c(n) \approx \frac{n^2}{4} + \frac{7n}{4}$$

- Comparar com o pior caso !

## Decomposição em subproblemas

- Diminuir-para-Reinar / **Decrease-and-Conquer**
  - Resolver **1 só subproblema** em cada passo do processo recursivo
  - Lista / cadeia de chamadas recursivas
- Dividir-para-Reinar / **Divide-and-Conquer**
  - Resolver **2 ou mais subproblemas** em cada passo do processo recursivo
  - Árvore de chamadas recursivas

### Decrease-and-Conquer



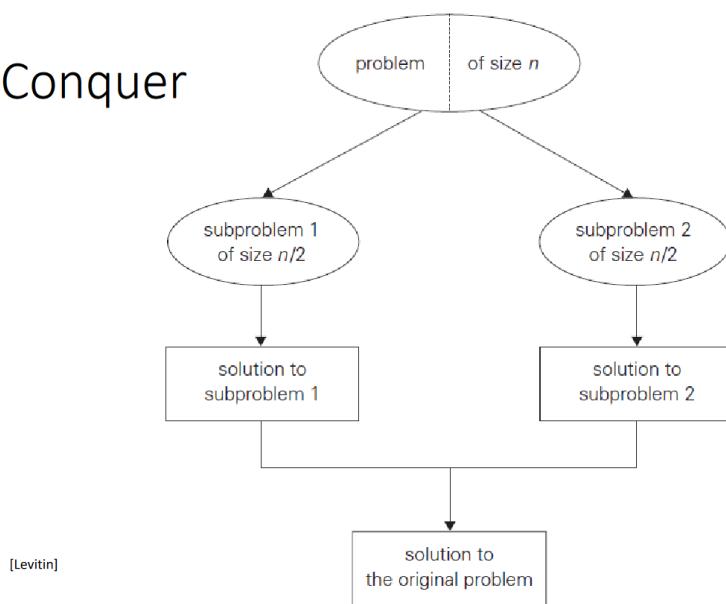
[Levitin]

Decrease-(by half)-and-conquer technique.

# Recursividade simples

- Diminuir-para-Reinar
- Executada **1 só chamada recursiva** em cada passo
- Fatorial, mdc, travessia de um array / uma lista, procura binária, ...
- Facilmente transformável num algoritmo iterativo, usando um ciclo
- Sugestão: implementar alguns destes algoritmos

## Divide-and-Conquer



[Levitin]

Calcular  $x^n$

---

## Calcular $x^n$

$$\begin{aligned}x^n &= x \times x^{n-1}, n > 0 \\x^0 &= 1\end{aligned}$$

```
double p(double x, unsigned int n) {  
    if(n > 0) return x * p(x, n - 1);  
    return 1;  
}
```

## Contar o número de multiplicações

$$M(0) = 0$$

$$M(n) = 1 + M(n - 1), n > 0$$

- Desenvolvimento telescópico – Quando parar ?

$$M(n) = 1 + M(n - 1) = 2 + M(n - 2) = \dots = k + M(\textcolor{red}{n - k})$$

$$M(n) = n + M(\textcolor{red}{0}) = n$$

$$\mathbf{M(n)} \in \mathcal{O}(n)$$

Procura Binária – Versão recursive

---

## Procura Binária

- Dado um array **ordenado** com n elementos : A[**left..right**]
- Procurar valor / chave X : índice ?
- Estratégia
  - Comparar A[**middle**] com X
  - Se **iguais**, devolver middle
  - Se **maior**, procura recursiva em A[**left..middle - 1**]
  - Se **menor**, procura recursiva em A[**middle + 1..right**]

## Procura Binária

```
int pesqBinRec(int* v, int esq, int dir, int valor) {
    unsigned int meio;

    if (esq > dir) return -1;

    meio = (esq + dir) / 2;

    contadorComps++;
    if (v[meio] == valor) {
        return meio;
    }
    contadorComps++;
    if (v[meio] > valor) {
        return pesqBinRec(v, esq, meio - 1, valor);
    }

    return pesqBinRec(v, meio + 1, dir, valor);
}
```

## Procura Binária – Caso particular

- $n = 2^k$
- $\text{esq} = 0 \quad \text{dir} = 2^k - 1 \quad \text{meio} = 2^{k-1} - 1$
- **Pior caso:** escolher sempre a **partição da direita**
  - É a maior das duas !!

$$W(1) = 2$$

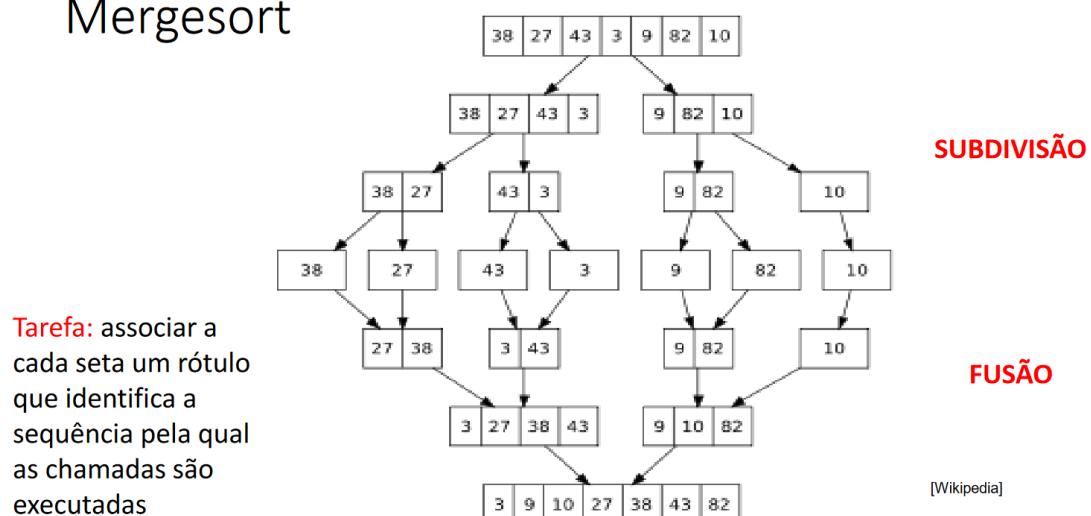
$$W(n) = 2 + W(n/2) = 4 + W(n/4) = 6 + W(n/8) = \dots$$

$$W(n) = 2 \times k + W(1) = 2 + 2 \log n \quad \mathbf{W(n)} \in \mathcal{O}(\log_2 n)$$

## Mergesort

- Ordenar um array / lista
  - Se o tamanho é **0 ou 1**, já está ordenada
  - Caso contrário, **subdividir** em duas “metades”
    - Aprox. do mesmo tamanho !!
  - **Ordenar recusivamente** cada “metade”
  - **Fundir** as duas “metades” ordenadas num só array / lista
- Questão : usar ou não um **array / lista adicional** ?

## Mergesort



```

mergesort (array a)
  if ( n == 1 )
    return a

  arrayOne = a[0] ... a[n/2]
  arrayTwo = a[n/2+1] ... a[n]

  arrayOne = mergesort ( arrayOne )
  arrayTwo = mergesort ( arrayTwo )

  return merge ( arrayOne, arrayTwo )

```

```

merge ( array a, array b )
  array c

  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a

  // At this point either a or b is empty

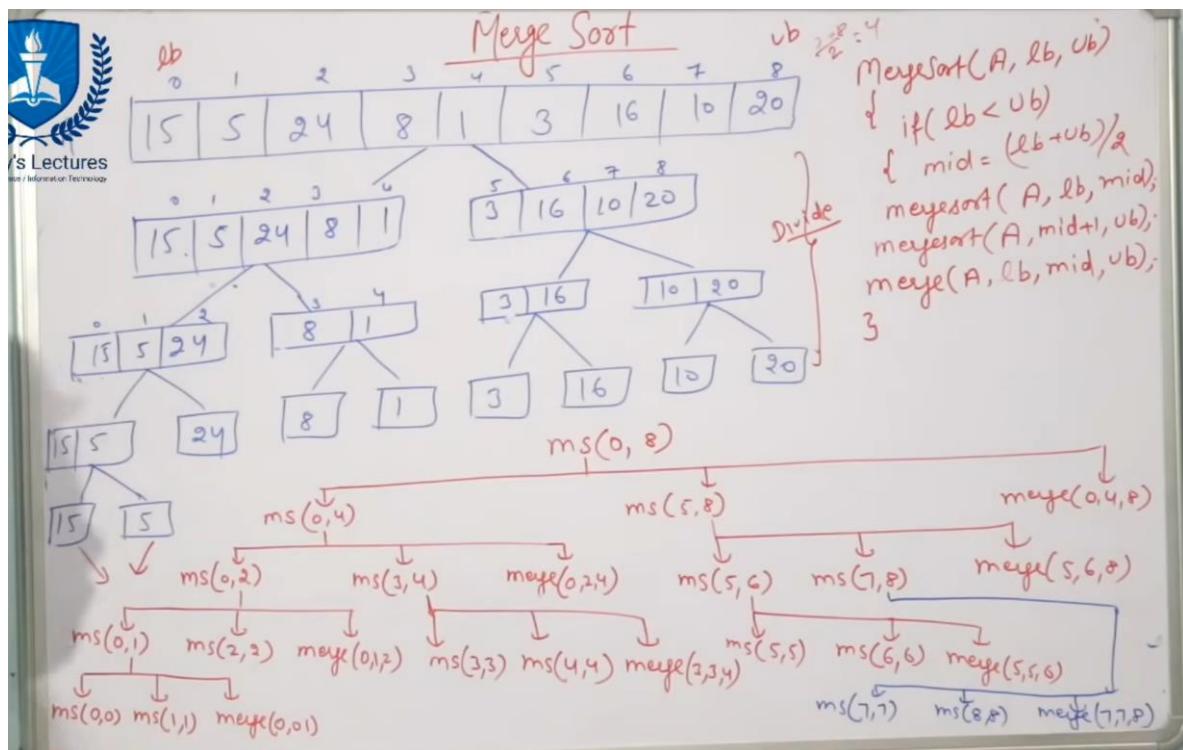
  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b

  return c

```

# $O(n \log n)$



---

## Eficiência – $C_{\text{merge}}(n)$ – Melhor Caso

- Todos elementos de um dos sub-arrays são copiados primeiro
- Apenas  $n / 2$  comparações para fazer isso !!
- $C(n) = 2 \times C(n / 2) + n / 2$
- Teorema Mestre :  $\Theta(n \log n)$
- Construir um exemplo ! 

## Eficiência – $C_{\text{merge}}(n)$ – Pior Caso

- Os elementos de um dos sub-arrays são copiados de modo intercalado, um a um !
- Necessárias  $(n - 1)$  comparações !
- $C(n) = 2 \times C(n / 2) + (n - 1)$
- Teorema Mestre :  $\Theta(n \log n)$
- Construir um exemplo ! 

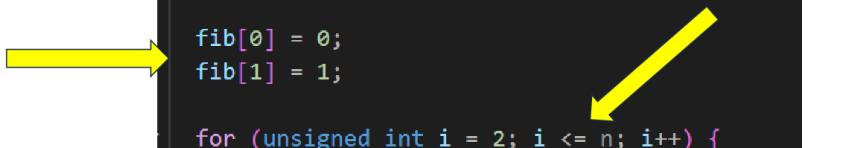
## Eficiência – $C_{\text{merge}}(n)$ – Caso Médio

- Podemos assumir que ocorre o nº médio de comparações
  - $\sim 3 \times n / 4$  comparações
- $C(n) = 2 \times C(n / 2) + 3 \times n / 4$
- Teorema Mestre :  $\Theta(n \log n)$
- Construir um exemplo ! 

## Programação Dinâmica

- Estratégia algorítmica genérica
- Aplicável a
  - Cálculo de **recorrências**
  - Resolução de **problemas de otimização combinatória**
- Ideia : **armazenar e reutilizar resultados “anteriores”** usando um **array local**
  - Array 1D / 2D / ...

## Números de Fibonacci – Prog. Dinâmica



```
long unsigned int fibonacci(unsigned int n) {
    long unsigned int fib[n + 1];

    fib[0] = 0;
    fib[1] = 1;

    for (unsigned int i = 2; i <= n; i++) {
        N_SOMAS++;
        fib[i] = fib[i - 2] + fib[i - 1];
    }

    return fib[n];
}
```

## Memoization

- Resultados de uma função são **memorizados** para uso futuro
- I.e., **evita-se** o cálculo de resultados (intermédios ou finais) obtidos no processamento de inputs anteriores
- Usar **um array global / uma cache** para armazenar os resultados calculados
  - Como inicializar ?
- **Recursividade + Programação Dinâmica**

## Números de Fibonacci – Memoization

```
#define SIZE 100
long int Fib_Cache[SIZE];
void initializeCache(void) {
    for (size_t i = 0; i < SIZE; i++) {
        Fib_Cache[i] = -1;
    }
}
```

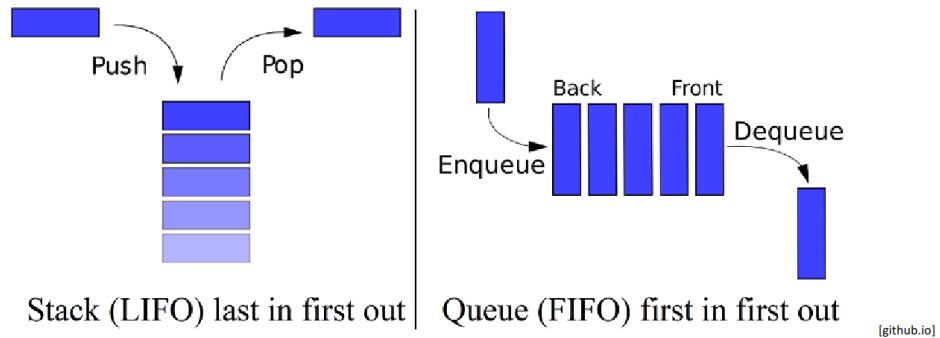
## Números de Fibonacci – Memoization

```
long int fibonacci(unsigned int n) {
    if (Fib_Cache[n] != -1) return Fib_Cache[n];

    long int r;
    if (n == 0)
        r = 0;
    else if (n == 1)
        r = 1;
    else {
        N_SOMAS++;
        r = fibonacci(n - 2) + fibonacci(n - 1);
    }
    Fib_Cache[n] = r;

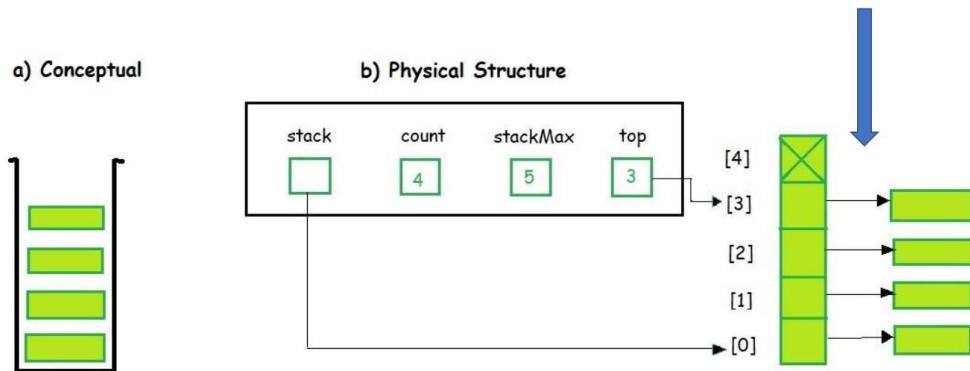
    return r;
}
```

## STACK e QUEUE

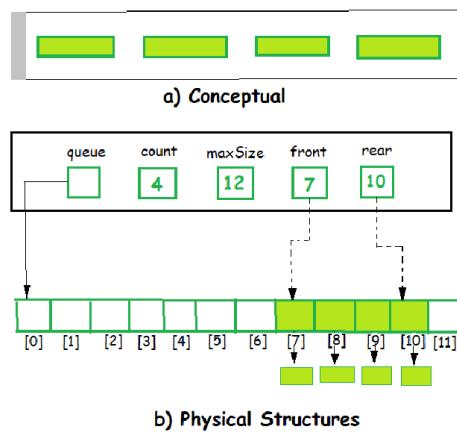


- Num próximo **guião prático** iremos usar / aplicar

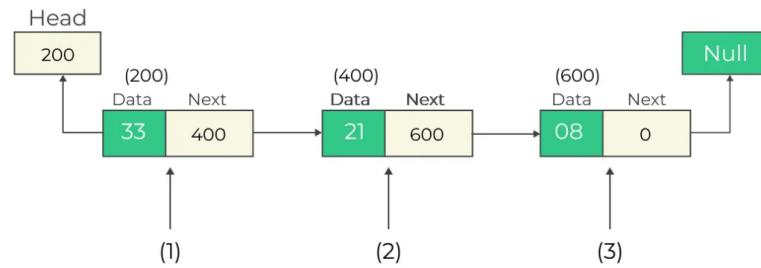
## O TAD Stack – Array de ponteiros



## O TAD QUEUE – Array circular de ponteiros

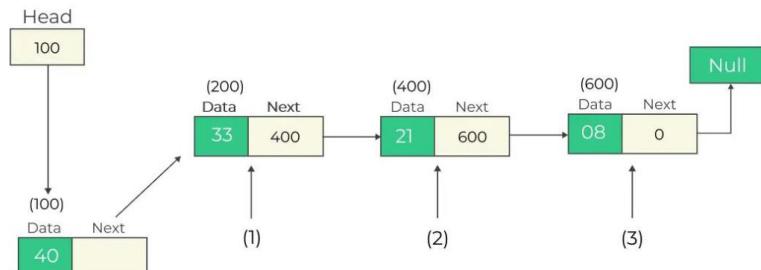


## Stack usando uma lista ligada



[prepinsta.com]

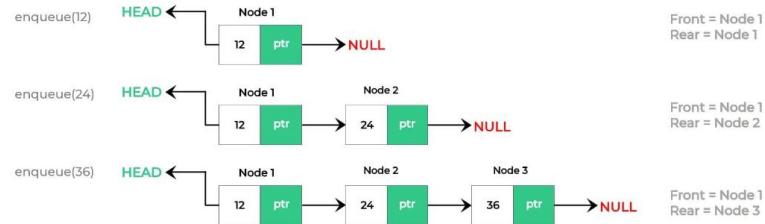
## Stack usando uma lista ligada – push(40)



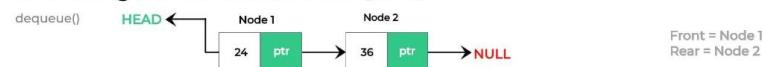
[prepinsta.com]

# Queue usando uma lista ligada

## Adding the elements into Queue



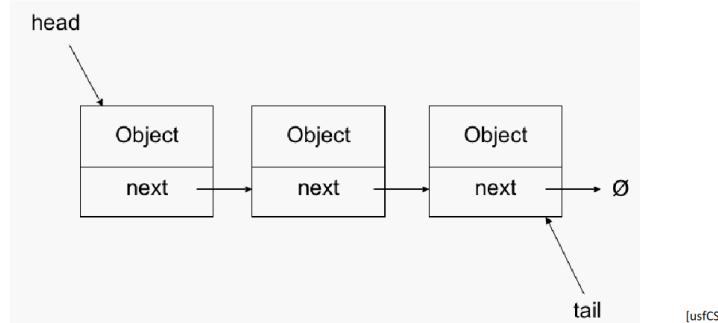
## Removing the elements from Queue



## Printing the Queue



# Lista Ligada – Acesso ao **início** e ao **fim** da lista



## Algumas propriedades das árvores binárias

- Questões simples !!

- Nº **máximo** de nós no **nível i** ?  **$2^i$**

- Nº **máximo** de nós numa árvore de **altura h** ? Quando ?  **$2^{h+1} - 1$**

- Nº **mínimo** de nós numa árvore de **altura h** ? Quando ?  **$h + 1$**

$$h + 1 \leq n \leq 2^{h+1} - 1$$

## Travessias recursivas

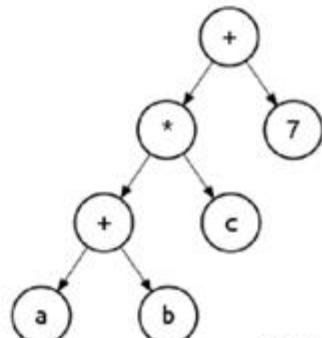
- Travessia em **pré-ordem (NLR)**

$$+ * + a b c 7$$

- Travessia em **em-ordem (LNR)**

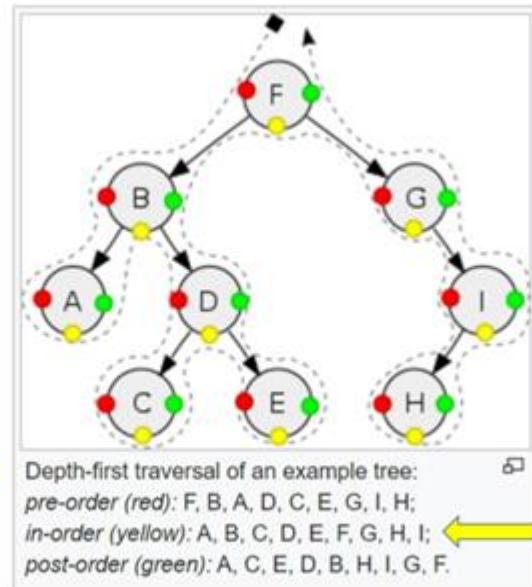
$$a + b * c + 7$$

- Travessia em **pós-ordem (LRN)**

$$a b + c * 7 +$$


[Wikipedia]

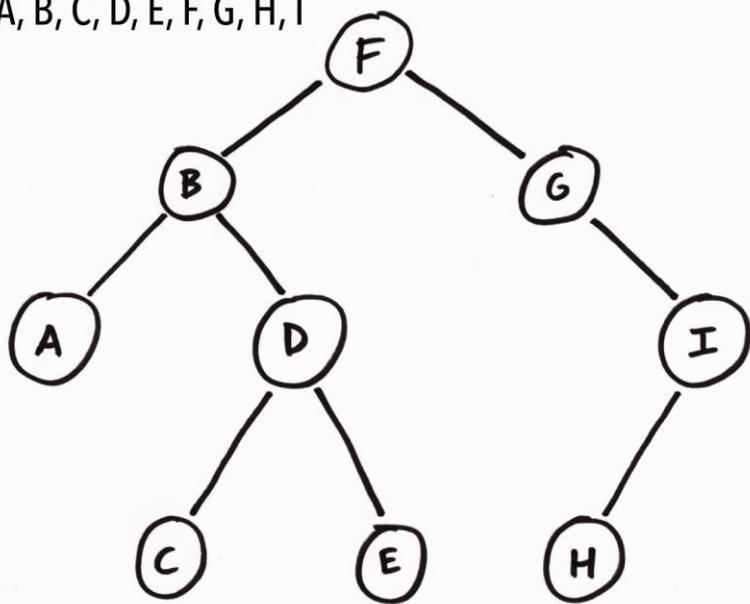
## Travessias



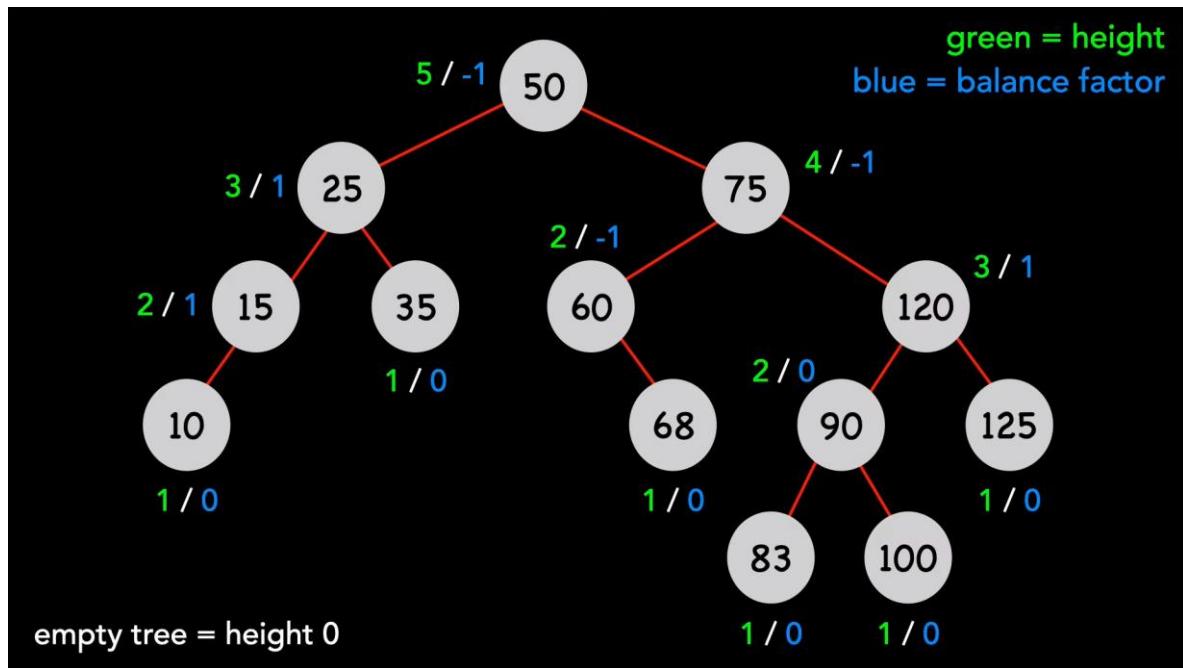
[Wikipedia]

Árvores Binárias de Procura (ABP) – Binary Search Trees (BST)

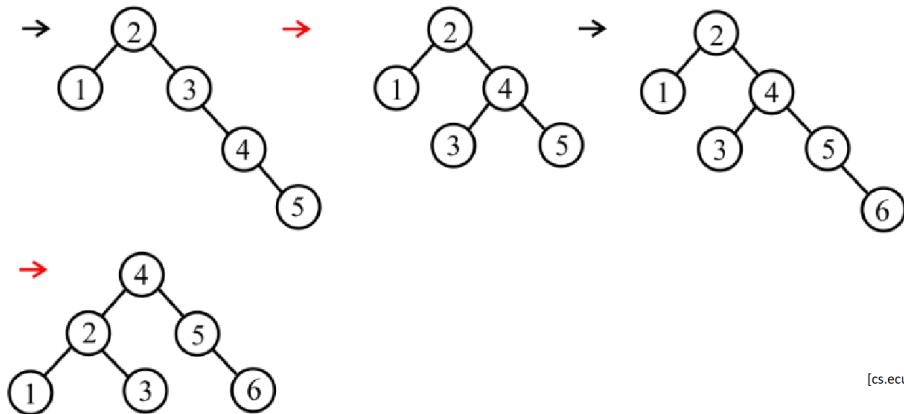
Output: A, B, C, D, E, F, G, H, I



## Árvores equilibradas em altura – Balanced Trees



## Árvore AVL – Inserir + Equilibrar, se necessário



---

## Fator de equilíbrio

- Para cada nó
- As duas sub-árvorestêm a mesma altura
- **Ou** a sua altura difere de 1
- $F = \text{Altura}(\text{SADireita}) - \text{Altura}(\text{SAEsquerda})$
- $F = -1, 0, 1$
- Se uma árvore estiver equilibrada, a **adição/remoção** de um nó pode forçar **F** a tomar o valor **+2 ou -2**
- Podemos usar para **identificar** os nós “**desequilibrados**” !!

## Array não-ordenado vs array ordenado

<i>operation</i>	<i>argument</i>	<i>return value</i>	<i>size</i>	<i>contents (unordered)</i>	<i>contents (ordered)</i>
<i>insert</i>	P		1	P	P
<i>insert</i>	Q		2	P Q	P Q
<i>insert</i>	E		3	P Q E	E P Q
<i>remove max</i>		Q	2	P E	E P
<i>insert</i>	X		3	P E X	E P X
<i>insert</i>	A		4	P E X A	A E P X
<i>insert</i>	M		5	P E X A M	A E M P X
<i>remove max</i>		X	4	P E M A	A E M P
<i>insert</i>	P		5	P E M A P	A E M P P
<i>insert</i>	L		6	P E M A P L	A E L M P P
<i>insert</i>	E		7	P E M A P L E	A E E L M P P
<i>remove max</i>		P	6	E M A P L E	A E E L M P P

A sequence of operations on a priority queue

## Critérios de ordem

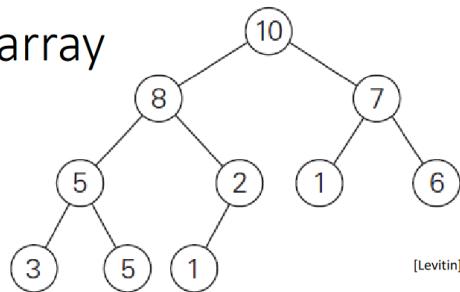
- **MIN-HEAP** : O valor/chave de um **nó não é superior** ao do seus **filhos**
- A **sequência de valores** em qualquer **caminho** da raiz da árvore até uma folha é **não-decrescente**
- **MAX-HEAP** : O valor/chave de um **nó não é inferior** ao do seus **filhos**
- A **sequência de valores** em qualquer **caminho** da raiz da árvore até uma folha é **não-crescente**
- Podem existir elementos/chaves **repetidos** !

---

## Representação usando um array

0	1	2	3	4	5	6	7	8	9
10	8	7	5	2	1	6	3	5	1

folhas

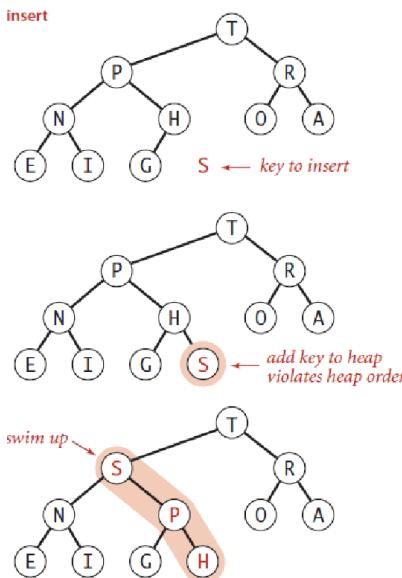


- Armazenar de modo contíguo, da esquerda para a direita, num array
- $\text{LeftChild}(i) = 2 \times i + 1$ , se existir
- $\text{RightChild}(i) = 2 \times (i + 1)$ , se existir
- $\text{Parent}(i) = (i - 1) \text{ div } 2$ , se  $i > 0$

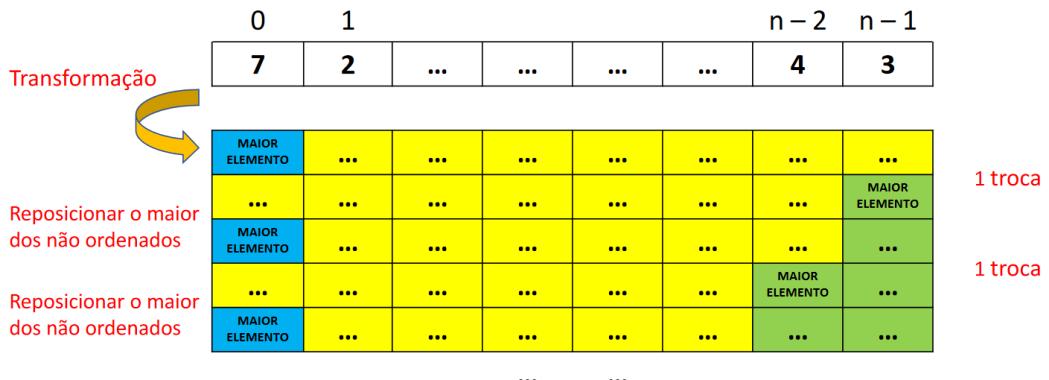
## Eficiência computacional

- Consultar o elemento de maior/menor prioridade **O(1)**
- Adicionar um elemento
  - Pode ser necessário reorganizar a heap**O(log n)**
- Apagar o elemento de maior/menor prioridade **O(log n)**
  - Pode ser necessário reorganizar a heap
- No pior caso, é necessário percorrer o caminho mais longo definido na heap !!

## Adicionar um elemento



## T&amp;C – Como ordenar um array ?



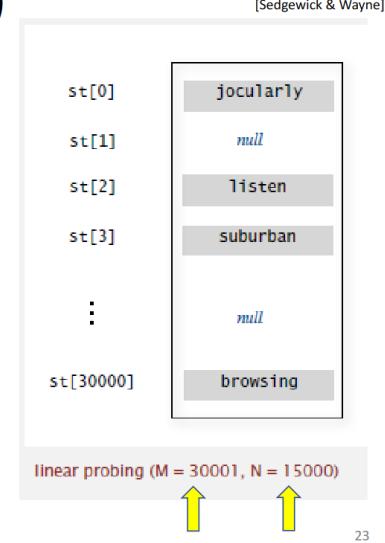
## Tabelas de Dispersão

- Estrutura de dados para armazenar pares (**chave, valor**)
- Sem **chaves duplicadas**
- **Sem uma ordem implícita !!**
- MAS, com **acesso rápido !!**



## Open Addressing (IBM, 1953)

- Quando há uma colisão, procurar o **espaço vago seguinte** e armazenar o item – (chave, valor)
- **Linear Probing** – Sondagem Linear
- O **tamanho da tabela (M)** tem de ser **maior** do que o **número de itens (N)** !!
- Quantas vezes maior ??



## Inserir na tabela – **Linear Probing**

- Guardar na **posição i**, se estiver disponível
- Caso contrário, tentar  $(i + 1) \% M, (i + 2) \% M$ , etc.

- Inserir L -> índice = 6

st[]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$M = 16$	P	M			A	C	S	H		E			R	X		

- Colisão !!

• ...

st[]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$M = 16$	P	M			A	C	S	H	L	E			R	X		

## Procurar na tabela – Linear Probing

- Procurar na posição  $i$
- Se estiver ocupada, verificar se as chaves são iguais
- Se forem diferentes, tentar em  $(i + 1) \% M, (i + 2) \% M$ , etc.
- Até encontrar a chave procurada ou chegar a um espaço vago
- Procurar  $H \rightarrow$  índice = 4                            -> 4 comparações

st[]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	P	M		A	C	S	H	L		E			R	X		
$M = 16$																
H																
search hit (return corresponding value)																

[Sedgewick & Wayne]

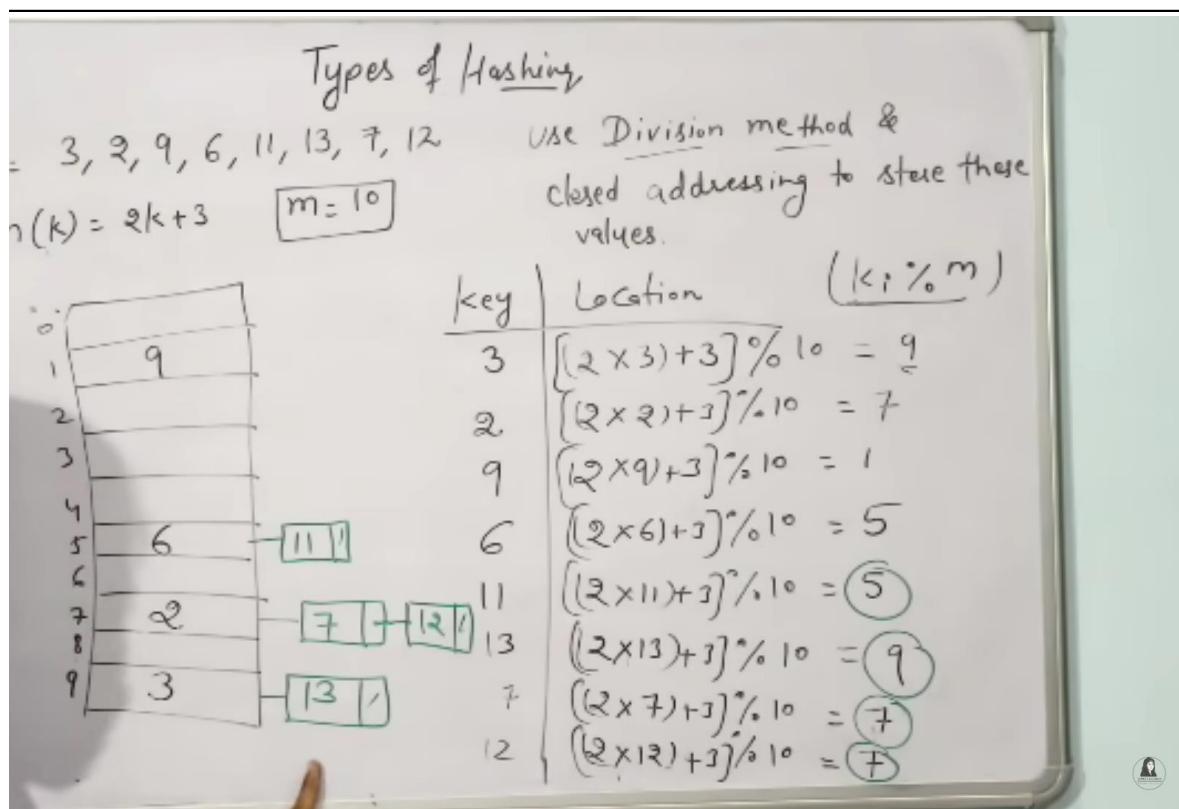
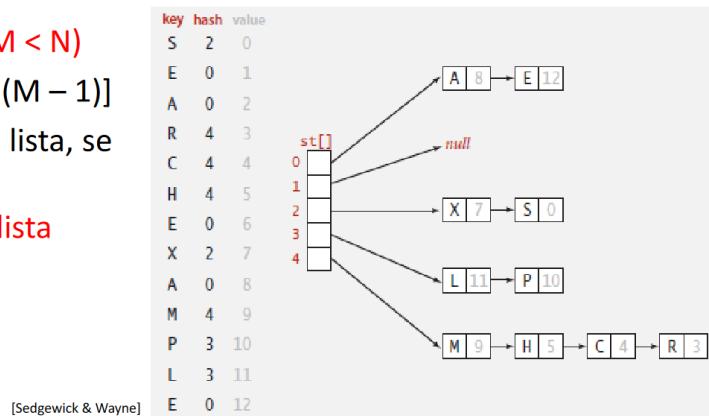
sent  $k_i$  at first free location from  $(u+i)\%m$  where  $i = 0 \text{ to } (m-1)$

$= 3, 2, 9, 6, 11, 13, 7, 12$       use Division method &  
 $h(k) = 2k+3$        $m=10$       Open addressing to store these values.

	key	Location ( $u$ )	Probes
0	13	3	1
1	9	$(2 \times 3) + 3 \% 10 = 9$	1
2	12	$(2 \times 2) + 3 \% 10 = 7$	1
3		$(2 \times 9) + 3 \% 10 = 1$	1
4		$(2 \times 6) + 3 \% 10 = 5$	1
5	6	$(2 \times 11) + 3 \% 10 = 5$	2
6	11	$(2 \times 7) + 3 \% 10 = 9$	2
7	2	$(2 \times 13) + 3 \% 10 = 7$	2
8	7	$(2 \times 12) + 3 \% 10 = 7$	6
9	3		
	12, -, -, 6, 11, 2, 7, 3		

## Separate Chaining (IBM, 1953)

- Array de **M** ponteiros ( $M < N$ )
- Mapear a chave em  $[0..(M - 1)]$
- Inserir no **início** de uma lista, se não existir
- Procurar **apenas numa lista**



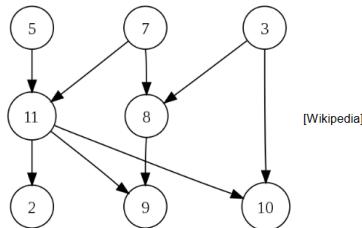
---

# Grafos

- Número máximo de arestas =  $V \times (V - 1) / 2$ 
  - Grafo completo :  $K_V$
- **Grau** de um vértice
  - Número de arestas incidentes nesse vértice
  - Grau máximo ?
- **Grafo regular**
  - Todos os vértices têm o mesmo grau  $k$  : grafo *k-regular*

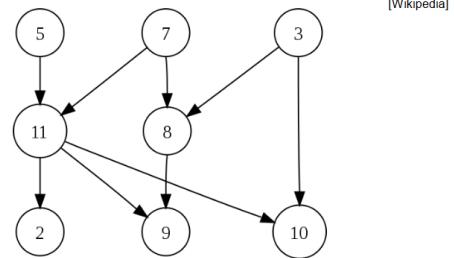
## Grafos orientados

- $G(V, E)$
- Grafo **orientado**
  - As **arestas orientadas** definem uma adjacência unidirecional
- $e_i = (v_j, v_k)$ 
  - $v_j$  é o vértice **origem** e  $v_k$  o vértice **destino**
  - $v_k$  é **adjacente** a  $v_j$
  - $e_i$  é **incidente** em  $v_k$



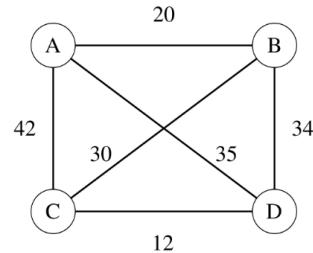
## Grafos orientados acíclicos

- Directed Acyclic Graphs (**DAGs**)
- Um grafo orientado que **não contém qualquer ciclo** !
  - Relações de **precedência**



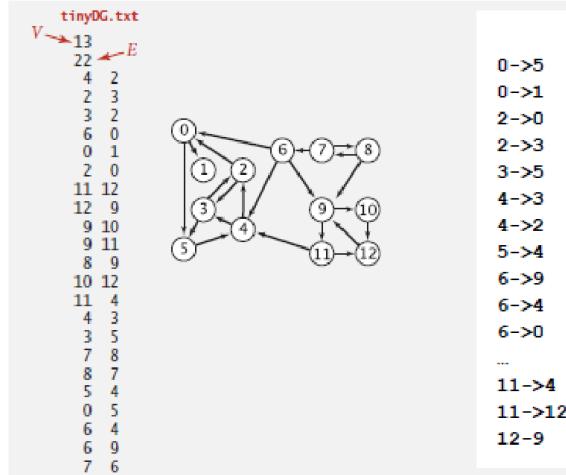
## Rede

- Uma rede é um grafo / grafo orientado com “pesos” associados às suas arestas
  - Weighted graph / digraph
  - Associar **um ou mais valores** a cada aresta
  - Custo, distância, capacidade, ...



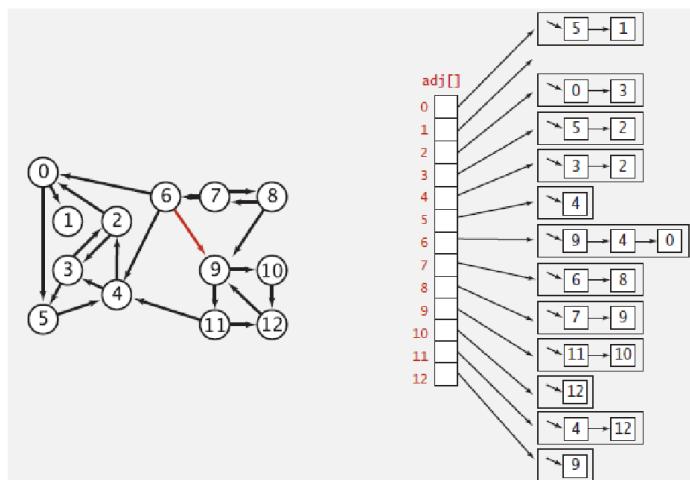
[Wikipedia]

## Representação em ficheiro – Lista de arestas



[Sedgewick/Wayne]

## Representação – Listas de adjacências



[Sedgewick/Wayne]

## Desempenho

- Na prática: usar a representação em **listas de adjacências**
- Os **grafos orientados** do mundo real são habitualmente **esparsos** !!
- Algoritmos iteram sobre os vértices adjacentes a um vértice dado

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices adjacent from v?
list of edges	E	1	E	E
adjacency matrix	V <sup>2</sup>	1 †	1	V
adjacency list	E + V	1	outdegree(v)	outdegree(v) ↙

† disallows parallel edges

[Sedgewick/Wayne]

Travessia em Profundidade

---

## Travessia em profundidade – Depth-First

- Algoritmo **idêntico** ao da travessia em profundidade de uma árvore binária
- Versão **recursiva** / Versão **iterativa** com **PILHA/STACK**
- **DIFERENÇAS :**
  - Há um vértice inicial – **start vertex – s**
  - O número de vértices adjacentes é **variável**
  - Podem haver **ciclos** e/ou **mais do que um caminho** para cada vértice
  - Para não entrar em ciclo, **marcar os vértices visitados !!**

## Algoritmo recursivo

Travessia em Profundidade (vértice **v**)

    Marcar **v** como visitado

    Para cada vértice **w** adjacente a **v**

        Se **w** não está marcado como visitado

            Então efetuar a Travessia em Profundidade (**w**)

- Resultado ?

- Ficam **marcados todos os vértices alcançados**

# Algoritmo iterativo – A mesma ordem ?

Travessia em Profundidade (vértice  $v$ )

Criar um STACK vazio

$\text{Push}(\text{stack}, v)$

Marcar  $v$  como visitado

Enquanto  $\text{NãoVazio}(\text{stack})$  fazer

$v = \text{Pop}(\text{stack})$

Para cada vértice  $w$  adjacente a  $v$

Se  $w$  não está marcado como visitado

Então  $\text{Push}(\text{stack}, w)$

Marcar  $w$  como visitado

## Travessia por níveis – Breadth-First

- Algoritmo **idêntico** ao da travessia por níveis de uma árvore binária
- **Versão iterativa** com **FILA/QUEUE**
- **Idêntico** à travessia em profundidade iterativa de um grafo
- **MAS**, usando um estrutura de dados auxiliar distinta
- A **ordem** pela qual os **vértices** são **visitados** é **diferente !!**
- Progressão em **círculos concêntricos** a partir do vértice inicial
- **APLICAÇÃO** : determinar **caminhos mais curtos !!**

## Algoritmo iterativo

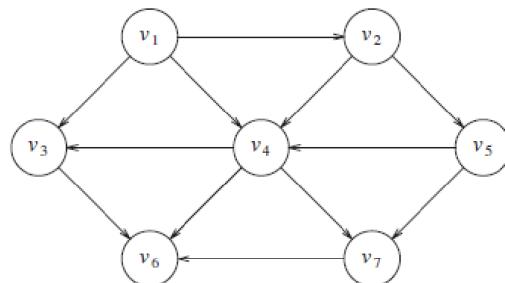
Travessia por Níveis(vértice  $v$ )

```
Criar FILA vazia  
Enqueue(queue, v)  
Marcar v como visitado  
Enquanto NãoVazia(queue) fazer  
    v = Dequeue(queue)  
    Para cada vértice w adjacente a v  
        Se w não está marcado como visitado  
            Então Enqueue(queue, w)  
            Marcar w como visitado
```

## Ordenação Topológica

- Podemos desenhar um dado **grafo orientado** de maneira a que **todas as arestas apontem para o mesmo lado** ?
- Dado um conjunto de **tarefas** a realizar, e as respetivas **precedências**, qual a **ordem** pela qual devem ser **escalonadas** ?
  - Usar **BFS** ou **DFS** !
  - Representar a solução com um **grafo orientado acíclico** !
- Aplicação : **verificar** se um **grafo orientado** é **acíclico** ou não

## Exemplo



- Possíveis sequências de vértices ?
- v1, v2, v5, v4, v3, v7, v6   OU   v1, v2, v5, v4, v7, v3, v6
- Como determinar ?

---

## 1º algoritmo – Cópia do grafo G

Criar  $G'$ , uma cópia do grafo G

Enquanto for possível

    Selecionar um vértice sem arestas incidentes

    Imprimir o seu ID

    Apagar esse vértice de  $G'$  e as arestas que dele emergem

- Usar o InDegree de cada vértice
- Ineficiência : cópia + sucessivas procuras através do conjunto de vértices

## 2º algoritmo – Array auxiliar



Registrar num array auxiliar numEdgesPerVertex o InDegree de cada vértice

Enquanto for possível

    Selecionar vértice v com  $\text{numEdgesPerVertex}[v] == 0$  E não marcado

    Imprimir o seu ID

    Marcá-lo como pertencendo à ordenação

    Para cada vértice w adjacente a v

$\text{numEdgesPerVertex}[w]--$

- Ineficiência : sucessivas procuras através do conjunto de vértices

---

## 3º alg. – Manter o conjunto de candidatos

Registrar num array auxiliar `numEdgesPerVertex` o InDegree de cada vértice

Criar FILA vazia e inserir na FILA os vértices `v` com `numEdgesPerVertex[v] == 0`

Enquanto a FILA não for vazia

`v` = retirar próximo vértice da FILA

    Imprimir o seu ID

    Para cada vértice `w` adjacente a `v`

`numEdgesPerVertex[w] --`

        Se `numEdgesPerVertex[w] == 0` Então Inserir `w` na FILA



- PROBLEMA : o que acontece se existir um ciclo ??