



Departamento de
Computação - **UFSCar**



Departamento de Computação
Centro de Ciências Exatas e Tecnologia
Universidade Federal de São Carlos

Algoritmos e Estruturas de Dados I

Apostila sobre algoritmos de ordenação e estrutura de dados elementares

Prof. Alexandre Luis Magalhães Levada
Email: alexandre.levada@ufscar.br

Sumário

Prefácio.....	3
Complexidade de algoritmos.....	4
Recursão.....	12
Algoritmos de ordenação elementares.....	22
O algoritmo Bubblesort.....	22
O algoritmo Insertionsort.....	26
O algoritmo Selectionsort.....	28
Tipos Abstratos de Dados (TAD's).....	32
Estruturas de dados lineares estáticas.....	40
Pilhas (LIFO).....	40
Filas (FIFO).....	47
Deque: Double-Ended Queue (Fila de duas extremidades).....	53
Fila de prioridades (Priority Queue).....	58
Estruturas de dados dinâmicas.....	62
Listas encadeadas.....	63
Pilhas com estruturas dinâmicas.....	67
Filas com estruturas dinâmicas.....	68
Listas encadeadas ordenadas.....	70
Listas duplamente encadeadas.....	71
Listas duplamente encadeadas com sentinelas.....	73
Aplicações: matriz esparsa com arrays e listas encadeadas.....	76
Aplicações: listas ortogonais e matrizes dinâmicas.....	79
Árvores Binárias.....	81
Árvores binárias de busca.....	86
Heaps e filas de prioridades.....	95
O algoritmo Heapsort.....	102
O problema do casamento estável e o algoritmo de Gale-Shapley.....	105
Bibliografia.....	112
Sobre o autor.....	113

Prefácio

O estudo de algoritmos e estruturas de dados é fundamental para programadores, cientistas e engenheiros de computação, pois os capacitam a desenvolver aplicações e métodos computacionais de maneira mais eficiente. Para ajudar a alcançar esse objetivo, essa apostila foi desenvolvida com o intuito de compilar alguns tópicos introdutórios sobre algoritmos e estruturas de dados elementares.

O conteúdo da apostila está organizado como segue: o capítulo 1 apresenta uma introdução ao estudo da análise de algoritmos, com conceitos e exemplos práticos. O capítulo 2 descreve uma técnica computacional fundamental, que é a recursão, utilizada em diversas estratégias de projeto e análise de algoritmos. O capítulo 3 apresenta algoritmo de ordenação elementares: Bubblesort, Insertionsort e Selectionsort, além de discutir a análise de cada um deles no pior, melhor e médio casos. O capítulo 4 apresenta o que são Tipos Abstratos de Dados e como eles podem ser empregados na implementação de estruturas de dados. O capítulo 5 descreve estruturas de dados lineares amplamente conhecidas na computação, que são as Pilhas, Filas e Deques. O capítulo 6 introduz as estruturas de dados lineares dinâmicas, em particular as listas encadeadas e duplamente encadeadas, nem como aplicações de tais estruturas na definição de matrizes esparsas. O capítulo 6 as árvores binárias, que são estruturas dinâmicas e hierárquicas muito eficientes para armazenamento de grandes volumes de dados. O Capítulo 7 aborda os heaps binários (min-heaps e max-heaps) e como podemos utilizá-los para construir filas de prioridades de maneira eficiente. Também discutimos o algoritmo Heapsort para ordenação de dados eficiente. Como uma aplicação interessante das árvores binárias, apresentamos o algoritmo de codificação de Huffman para compactação de dados sem perdas. O capítulo 7 trata do problema do casamento estável, que é muito importante na computação devido as suas várias aplicações, e sua solução ótima pelo algoritmo de Gale-Shapley.

Bons estudos!

“Experiência não é o que acontece com você; é o que você faz com o que lhe acontece.”
(Aldous Huxley)

Complexidade de algoritmos

Na ciência da computação, mais especificamente no estudo de algoritmos, uma das primeiras perguntas que surgem é: como medir a eficiência de algoritmos, ou seja, dados dois algoritmos A₁ e A₂, como saber qual deles é mais eficiente?

Na realidade, estamos interessados em analisar 2 quantidades:

- a) tempo de execução (complexidade de tempo)
- b) quantidade de memória utilizada (complexidade de espaço)

A quantidade de memória exigida pelo algoritmo depende basicamente das variáveis alocadas, o que é razoavelmente simples de se estimar. Já o tempo de execução requer uma análise mais formal e aprofundada do algoritmo. Uma dúvida natural de diversos programadores é: porque não podemos apenas cronometrar o tempo gasto pela execução de uma implementação do algoritmo?

→ **Porque essencialmente, esse tempo medido iria depender de diversos fatores externos ao algoritmo, como a linguagem de programação utilizada (C vs Python vs Java) e o hardware da máquina (processador).**

Desejamos realizar uma análise que seja universal, ou seja, dependa de fatores externos ao algoritmo em questão.

O objetivo consiste em contar o número de operações de atribuição existentes em um algoritmo A.

Para isso, assume-se algumas hipóteses:

1. Cada comando de atribuição executa em tempo constante, ou seja, possui complexidade O(1).
2. Uma função T(n) deve retornar o número de atribuições a serem executadas quando a entrada do problema resolvido pelo algoritmo possui tamanho n.

Vejamos um simples exemplo a seguir, com uma função que soma os n primeiros inteiros.

```
sum_of_n(n) {
    soma = 0
    for i = 1 to n
        soma = soma + i
    return soma
}
```

É fácil notar que a instrução de inicialização é executada apenas uma vez e que o loop FOR executa n vezes, o que nos leva a:

$$T(n) = n + 1$$

Note que quando n cresce muito, apenas uma parte dominante da função é importante (n).

Notação Big-O

Ao invés de nos preocuparmos em contar exatamente o número de instruções de um programa, é mais tratável matematicamente analisar a ordem de magnitude da função $T(n)$, ou seja, o que acontece com a função $T(n)$ quando n cresce arbitrariamente.

Suponha de exista uma função $f(n)$, definida para todos os inteiros maiores ou iguais a zero, tal que para constantes $c, m > 0$, temos:

$$T(n) \leq c f(n)$$

para $\forall n \geq m$ (n suficientemente grande). Então, dizemos que $T(n)$ é $O(f(n))$ ou ainda:

$$T(n) = O(f(n))$$

Considere o código a seguir, de uma função que soma os elementos de uma matriz quadrada.

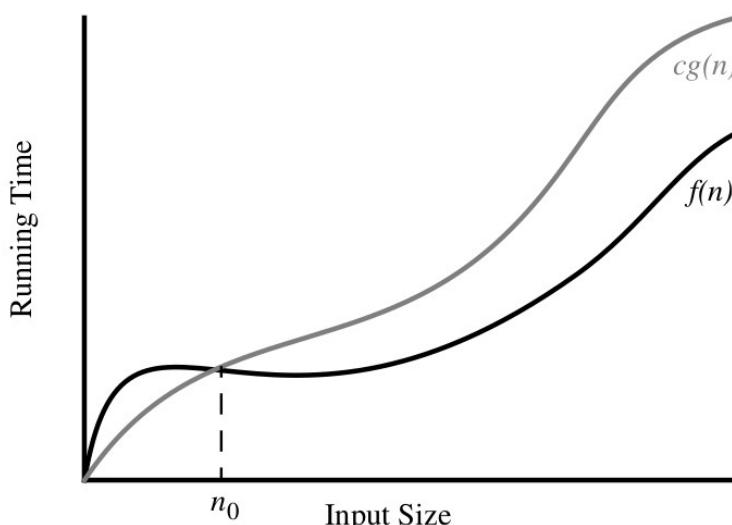
```
sum_of_matrix(M, n) {
    soma = 0
    for i = 1 to n {
        soma_linha = 0
        for j = 1 to n
            soma_linha += M[i, j]
        soma += soma_linha
    }
    return soma
}
```

Vamos calcular o número de instruções a serem executadas por esse algoritmo. Note que no loop mais interno (j) temos n iterações. Assim para cada valor de i no loop mais externo, temos $n + 2$ atribuições. Como temos n possíveis valores para i , chegamos em:

$$T(n) = n(n+2) = n^2 + 2n + 1$$

Note que se tomarmos $c=2$ e $f(n)=n^2$, temos $n^2 + 2n + 1 \leq 2n^2$ para todo $n > 2$

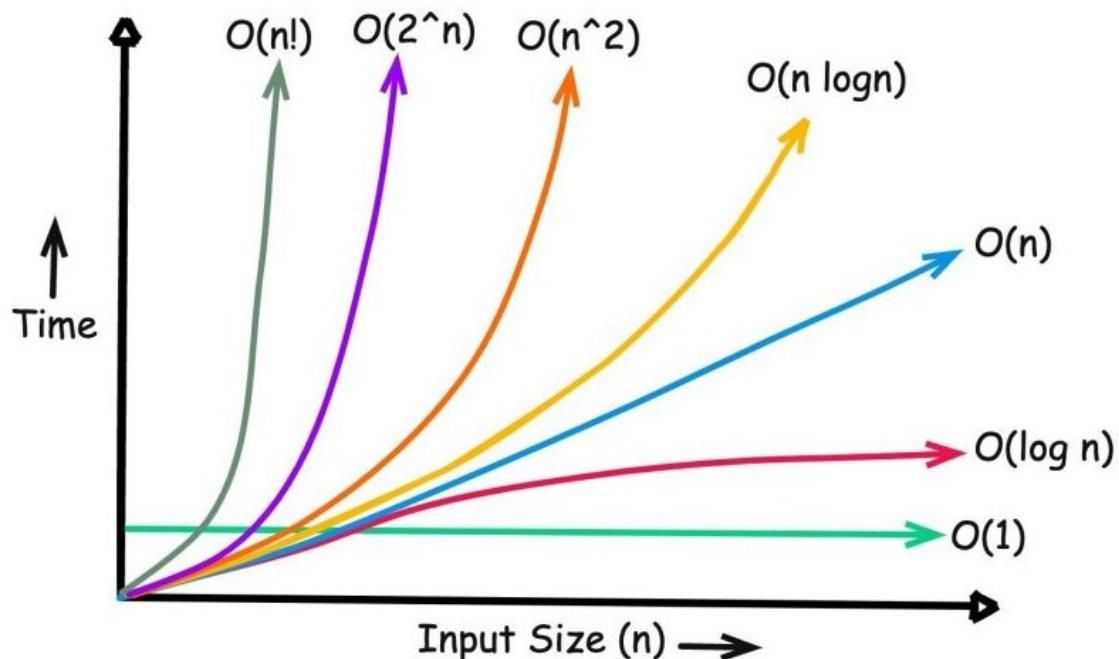
o que implica em dizer que $T(n)$ é $O(n^2)$. Poderíamos ter escolhido a função $f(n)=n^3$, mas o objetivo da notação Big-O é fornecer o limite superior mais justo possível!



Costuma-se dividir os algoritmos nas seguintes classes de complexidade:

$f(n)$	Classe
1	Constante
$\log n$	Logarítmica
n	Linear
$n \log n$	Log-linear
n^2	Quadrática
n^3	Cúbica
n^k	Polinomial
2^n	Exponencial
$n!$	Fatorial

A figura a seguir mostra a taxa de crescimento dessas funções.



Somatários

No cálculo da complexidade de algoritmos é comum termos que analisar estruturas de repetição (loops). Para isso, devemos saber resolver somatórios. Seja o somatório a seguir:

$$2 + 2^2 + 2^3 + 2^4 + \dots + 2^{10}$$

Podemos expressá-lo como:

$$\sum_{k=1}^{10} 2^k$$

A seguir veremos diversas propriedades importantes na manipulação e resolução de somatórios.

1) Substituição de variáveis

Seja o seguinte somatório:

$$\sum_{k=1}^n 2^k$$

Definindo $i = k - 1$, temos que $k = i + 1$.

Como k inicia em 1, i deve iniciar em zero (limite inferior, LI).

Como k vai até n , i deve ir até $n - 1$ (limite superior, LS).

Dessa forma, podemos expressar o somatório como:

$$\sum_{i=0}^{n-1} 2^{i+1}$$

2) Distributiva: para toda constante c

$$\sum_{k \in A} c f(k) = c \left(\sum_{k \in A} f(k) \right)$$

Em outras palavras, é possível mover as constantes para fora do somatório colocando-as em evidência.

3) Associativa: somatórios de somas é igual a somas de somatórios

$$\sum_{k \in A} (f(k) + g(k)) = \sum_{k \in A} f(k) + \sum_{k \in A} g(k)$$

4. Somas telescópicas: considere uma sequencia de números reais $x_1, x_2, x_3, \dots, x_n, x_{n+1}$. Então, a identidade a seguir é válida:

$$\sum_{k=1}^n (x_{k+1} - x_k) = x_{n+1} - x_1$$

ou seja, o valor do somatório das diferenças é igual a diferença entre o último elemento e o primeiro

Prova:

1. Pela propriedade associativa (3), temos:

$$S = \sum_{k=1}^n (x_{k+1} - x_k) = \sum_{k=1}^n x_{k+1} - \sum_{k=1}^n x_k$$

2. Por substituição de variáveis (1), temos:

$$i = k+1 \rightarrow k = i - 1$$

$$\text{LI: } k = 1 \rightarrow i = 2$$

$$\text{LS: } k = n \rightarrow i = n+1$$

$$S = \sum_{i=2}^{n+1} x_i - \sum_{k=1}^n x_k$$

3. Removendo o último termo do primeiro somatório e o primeiro termo do segundo, temos:

$$S = \sum_{i=2}^n x_i + x_{n+1} - x_1 - \sum_{k=2}^n x_k = x_{n+1} - x_1$$

A prova está concluída.

Analisando algoritmos

Considere o exemplo a seguir, com duas estruturas de repetição em série.

```
Func_A(n) {
    c = 0
    for i = 1 to n
        c += 1
    for j = 1 to n
        c += 2
    return c
}
```

Note que temos uma atribuição inicial (1) e logo dois loops com n iterações. Cada um deles, contribui com n para o total, de modo que no total temos $T(n) = 2n + 1$, o que resulta em uma complexidade $O(n)$.

Considere o algoritmo a seguir, que utiliza duas estruturas de repetição aninhadas.

```
Func_B(n) {
    c = 0
    for i = 1 to n {
        for j = 1 to n
            c = c + 1
    }
    return c
}
```

Nesse caso, o loop interno tem n operações. Como o loop externo é executado n vezes, e temos uma inicialização, o total de operações é $T(n)=n^2+1$, o que resulta em $O(n^2)$.

E se no loop interno ao invés de n fosse 10?

Teríamos $T(n)=10n+1$, o que é $O(n)$.

Vejamos a seguir mais um exemplo com estruturas de repetição aninhadas, onde o loop mais interno depende a variável contadora do loop mais externo.

```
Func_C(n) {
    count = 0
    for i = 1 to n {
        for j = 1 to i
            c = c + 1
    }
    return c
}
```

Note que quando $i = 0$, o loop interno executa uma vez, quando $n = 1$, o loop interno executa duas vezes, quando $n = 2$, o loop interno executa 3 vezes, e assim sucessivamente. Assim, o número de vezes que a variável count é incrementada é igual a: $1 + 2 + 3 + 4 + \dots + n$. Devemos resolver esse somatório para calcular a complexidade dessa função.

$$\sum_{k=1}^n k$$

Primeiramente, note que

$$(k+1)^2 = k^2 + 2k + 1$$

o que implica em

$$(k+1)^2 - k^2 = 2k + 1$$

Assim, $\sum_{k=1}^n [(k+1)^2 - k^2] = \sum_{k=1}^n [2k + 1]$. Porém, o lado esquerdo é uma soma telescópica e temos:

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

Dessa forma, podemos escrever:

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

Aplicando a propriedade associativa, temos:

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

o que nos leva a:

$$2 \sum_{k=1}^n k = n^2 + 2n + 1 - 1 - n = n^2 + n$$

Finalmente, colocando n em evidência e dividindo por 2, finalmente temos:

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

Portanto, $T(n)$ é igual a:

$$T(n) = \frac{1}{2}(n^2 + n) + 1$$

o que resulta em $O(n^2)$.

O próximo exemplo mostra uma função em que a variável contadora é dividida por 2 a cada iteração.

```
Func_D(n) {  
    c = 0  
    i = n  
    while i > 1 {  
        c = c + 1  
        i = i // 2      # divisão inteira  
    }  
    return c  
}
```

Essa função calcula quantas vezes o número pode ser dividido por 2. Por exemplo, considere a entrada $n = 16$. Em cada iteração esse valor será dividido por 2, até que atinja o zero.

$16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

A variável c termina a função valendo 4, pois $2^4 = 16$.

Se $n = 25$, temos:

$25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1$

A variável c termina a função valendo 4, pois $2^4 < 25 < 2^5$

Se $n = 40$, temos:

$40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 2 \rightarrow 1$

A variável c termina a função valendo 5, pois $2^5 < 40 < 2^6$

Portanto, o número de iterações do loop é $\log_2 n$. Dentro do loop existem duas instruções, portanto neste caso teremos:

$T(n) = 2 + 2\lfloor \log_2 n \rfloor$, onde a função piso(x) retorna o maior inteiro menor que x.

o que resulta em $O(\log_2 n)$.

O exemplo a seguir mostra como é possível ter algoritmos com complexidade log-linear.

```
Func_E(n) {  
    c = 0  
    for i = 1 to n  
        c = c + Func_D(n)  
    return c  
}
```

Note que, como a função $\text{Func_D}(n)$ tem complexidade logarítmica, e o loop tem n iterações, temos que a complexidade da função em questão é $O(n \log_2 n)$.

A seguir apresentamos duas funções para verificar se um número inteiro é primo. Analise a complexidade de cada uma delas e justifique qual delas é mais eficiente.

```
prime_A(n) {  
prime_B(n) {
```

```

if n == 1
    return False
for i = 2 to n-1 {
    resto = n % i
    if resto == 0
        return False
}
return True
}

if n == 1
    return False
if n == 2
    return True
if n % 2 == 0
    return False
i = 3
while i <= sqrt(n) {
    resto = n % i
    if resto == 0
        return False
    else
        i += 2
}
return True
}

```

Primeiramente, vamos analisar o algoritmo prime_A: note que se $n = 0, 1, 2$ ou 3 , o algoritmo tem complexidade $O(1)$, ou seja, temos o melhor caso. Caso contrário, devemos calcular o resto da divisão de n por i ($n - 1 - 2 + 1 = (n - 2)$ vezes, o que significa que o algoritmo é $O(n)$).

Agora, vamos analisar o algoritmo prime_B: note que se $n = 0, 1, 2$ ou 3 , o algoritmo tem complexidade $O(1)$, ou seja, temos o melhor caso. Caso contrário, o número de atribuições realizadas será $2 \frac{\sqrt{n}-3}{2}$, uma vez que a cada iteração do WHILE, calcula-se o resto e incrementa-se a variável contadora i , o que significa que o algoritmo é $O(n^{1/2})$, e portanto, mais rápido que o algoritmo prime_A (\sqrt{n} cresce um pouco mais rápido que $\log n$, veja a derivada!).

Portanto, o algoritmo prime_B é mais eficiente que o algoritmo prime_A. Em termos práticos, isso significa que em nenhum computador do mundo, prime_A será mais rápido que prime_B para valores de n suficientemente grandes (para n pequeno pode até ser).

"A sua força não vem de suas vitórias, mas sim da sua luta diária contra as adversidades."
Autor Anônimo.

Recursão

Dizemos que uma função é recursiva se ela é definida em termos dela mesma. Em matemática e computação uma classe de objetos ou métodos exibe um comportamento recursivo quando pode ser definido por duas propriedades:

1. Um caso base: condição de término da recursão em que o processo produz uma resposta.
2. Um passo recursivo: um conjunto de regras que reduz todos os outros casos ao caso base.

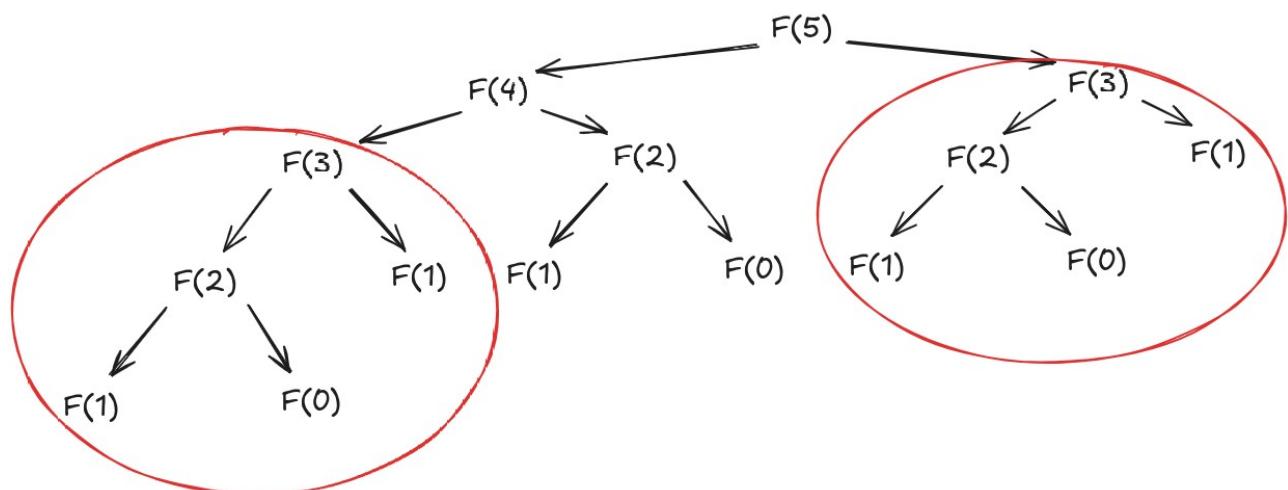
A série de Fibonacci é um exemplo clássico de recursão, pois:

$$\begin{aligned} F(0) &= 0 && \text{(caso base)} \\ F(1) &= 1 && \text{(caso base 1)} \\ F(2) &= 1 && \text{(caso base 2)} \\ \text{Para todo } n > 1, F(n) &= F(n - 1) + F(n - 2) \end{aligned}$$

A seguir ilustramos um algoritmo recursivo para geração do n -ésimo número de Fibonacci.

```
Fib(n) {
    if n == 0
        return 0
    else {
        if n == 1 or n == 2
            return 1
        else
            return Fib(n - 1) + Fib(n - 2)
    }
}
```

Problema com essa função recursiva: muito ineficiente. Suponha que deseja-se calcular o valor de $Fib(7)$. Não é muito eficiente de ponto de vista computacional. Isso ocorre pois durante o cálculo de $F(n)$, os valores de $F(n-2)$, $F(n-3)$, etc... são calculados várias vezes. O número de chamadas recursivas cresce exponencialmente. O padrão recursivo consiste na expansão de uma árvore binária (há muita repetição de cálculos).



Veja que para computar $F(3)$ são necessárias duas recursões, para $F(4)$ são 4 recursões e assim sucessivamente. Veja que o crescimento é praticamente exponencial.

	Número de recursões
F(3)	2
F(4)	4
F(5)	8
F(6)	14
F(7)	24

Quantas recursões são necessárias para calcular F(9)?

Há um padrão na sequência que deve estar óbvio agora: 2, 4, 8, 14, 24, 40, ... A pergunta que surge é: qual é o número de recursões necessárias para um n arbitrário?

Note que podemos definir a seguinte recorrência:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Como para n grande, temos que $T(n-1) \approx T(n-2)$, iremos obter uma aproximação.

$$T(n) = 2T(n-1) + O(1)$$

$$T(n) = 2(2T(n-2) + O(1)) + O(1) = 2^2T(n-2) + 2O(1) + O(1)$$

$$T(n) = 2(2(2T(n-3) + O(1)) + O(1)) + O(1) = 2^3T(n-3) + O(1) \sum_{i=0}^2 2^i$$

Prosseguindo com essas substituições até o k-ésimo termo, chega-se em:

$$T(n) = 2^k T(n-k) + O(1) \sum_{i=0}^{k-1} 2^i = 2^k T(n-k) + (2^k - 1)O(1) = 2^k T(n-k) + 2^k O(1) = 2^k (T(n-k) + O(1))$$

A condição de parada da recursão é termos $T(0)$, ou seja, $k = n$. Logo, temos:

$$T(n) = 2^n (T(0) + O(1)) = 2^n O(1) = 2^n$$

Portanto, $T(n) = O(2^n)$, mostrando que a função possui complexidade exponencial (proibitiva).

Sendo assim, a solução para esse problema para um número elevado n é inviável. Por exemplo, suponha que tenhamos $n = 100$. Então, o número de operações necessárias para resolver o problema do Fibonacci recursivo é $2^{100} = 1.2676506 \times 10^{30}$. Considerando que cada instrução é executada em 1 nanosegundo, ou seja, 10^{-9} segundos, o tempo gasto seria 1.2676506×10^{21} segundos, o que equivale a aproximadamente 4.01×10^{13} anos! Para se ter uma ideia de quão grande isso é, a idade estimada do universo é de 26.7 bilhões de anos, o que seria 26.7×10^9 anos.

Problema: Faça um algoritmo recursivo para calcular o fatorial de um inteiro n. Lembre que $n! = n(n-1)!$

```
fatorial(n) {
    if n == 1
        return 1
    else
        return n*fatorial(n-1)
}
```

Problema: Faça um algoritmo recursivo para calcular o somatório a seguir:

$$S = \sum_{i=1}^n i$$

```
somatorio(n) {
    if n == 1
        return 1
    else
        return n + somatorio(n-1)
}
```

Problema: Faça um algoritmo recursivo para calcular o valor de e^x , sabendo que:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

```
exponencial(x, n){
    if n == 0
        return 1
    else
        return (x**n)/fatorial(n) + exp(x, n-1)
}
```

Problema: O máximo divisor comum entre dois inteiros a e b é o maior inteiro n que divide tanto a quanto b. Seja a = 1071 e b = 462. Primeiramente, devemos calcular quantas vezes b cabe em a e tomar o excesso, ou seja, sabemos que $1071 // 462 = 2$ e $1071 \% 462 = 147$. O processo é então repetido, fazendo com que calculemos quantas vezes 147 cabe em 462 e tomemos o excesso. Sabemos que $462 // 147 = 3$ e $462 \% 147 = 21$. Fazendo a iteração novamente, temos que $147 // 21 = 7$ e $147 \% 21 = 0$. Como o resto é zero, o algoritmo para e temos que $\text{mdc}(a, b) = 21$.

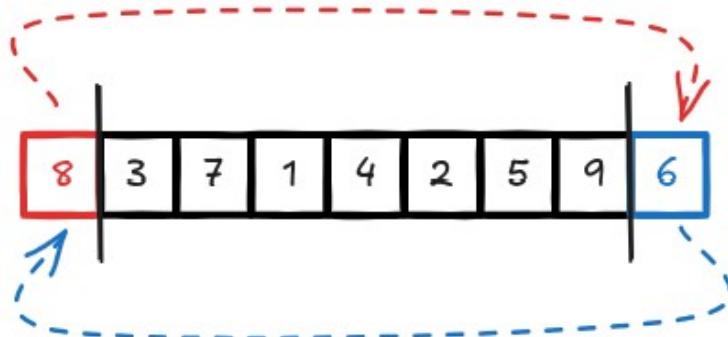
Faça duas funções para calcular o MDC entre dois inteiros: uma iterativa e outra recursiva.

```
# Função iterativa
mdc(a, b) {
    while b > 0 {
        tmp = a
        a = b
        b = tmp % b
    }
    return a
}

# Função recursiva
mdc_r(a, b) {
    if b == 0
        return a
    else
        return mdc_r(b, a % b)
}
```

Problema: Faça um algoritmo recursivo para inverter uma lista. Sugestão: utilize a seguinte ideia - inicie com inicio = 0 e fim = len(L) - 1, depois inverta o primeiro e o último elementos e chame a

função recursiva com inicio incrementado de 1 e fim decrementado de 1. A condição de parada deve ser o fim ser menor ou igual que o inicio.



Troca as 2 extremidades e
inverte o "miolo" do array

```
# Função recursiva
inverte(L, inicio, fim) {
    if fim <= inicio
        return L
    else {
        swap(L[inicio], L[fim])
        inverte(L, inicio+1, fim-1)
    }
}
```

O problema da multiplicação de inteiros

Nesta seção, iremos projetar e analisar um algoritmo mais eficiente para a multiplicação de números inteiros. Primeiramente, lembre-se que a multiplicação de 2 inteiros de n dígitos requer n^2 multiplicações e 2 adições:

$$\begin{array}{r}
 384 \\
 \times 156 \\
 \hline
 2304 \\
 1920 + \\
 384 + \\
 \hline
 59904
 \end{array}$$

3 dígitos
 3 dígitos
 São 9 multiplicações + 2 adições = $O(n^2)$

A pergunta é: podemos fazer melhor que isso? Iremos tentar.

Sem perda de generalidade, é usual assumir que $n=2^m$ (potência de 2) para simplificar as coisas. Assim, sempre podemos particionar a e b em suas metades superiores e inferiores.

Por hora, vamos considerar os números a seguir:

$$\begin{aligned}
 a &= 12345678 \\
 b &= 87654321
 \end{aligned}$$

Note que podemos escrever:

$$a = 12340000 + 5678$$

$$b = 87650000 + 4321$$

Logo, o produto entre a e b pode ser expresso como:

$$a \times b = (1234 \times 10^4 + 5678)(8765 \times 10^4 + 4321)$$

Aplicando a propriedade distributiva:

$$a \times b = 1234 \times 8756 \times 10^8 + (1234 \times 4321 + 5678 \times 8765) \times 10^4 + 5678 \times 4321$$

Usando essa notação, podemos escrever:

$$a = a_1 10^{n/2} + a_2$$

$$b = b_1 10^{n/2} + b_2$$

onde a_1 e a_2 são as partes superiores e inferiores de a e b_1 e b_2 são as partes superiores e inferiores de b. A multiplicação é dada:

$$a \times b = A \times 10^n + (B+C) \times 10^{n/2} + D$$

onde

$$A = a_1 \times b_1$$

$$B = a_2 \times b_1$$

$$C = a_1 \times b_2$$

$$D = a_2 \times b_2$$

A função a seguir utiliza uma estratégia recursiva para implementar uma solução alternativa para a multiplicação de inteiros.

```
Multiply(a, b) {
    if len(a) <= 1
        return a*b
    Particione a e b em
        a = a_1 10^{n/2} + a_2      # essa partição requer um único loop - O(n)
        b = b_1 10^{n/2} + b_2
    A = Multiply(a_1, b_1)
    B = Multiply(a_1, b_2)
    C = Multiply(a_2, b_1)
    D = Multiply(a_2, b_2)
    P = A \times 10^n + (B+C) \times 10^{n/2} + D
    return P
}
```

Note que um problema de tamanho n é dividido em 4 problemas de tamanho n/2 mais um particionamento que tem complexidade O(n). Assim, a recorrência fica:

$$T(n) = 4 T\left(\frac{n}{2}\right) + O(n)$$

Será que a complexidade é menor que $O(n^2)$? Expandindo o termo $T(n/2)$, temos:

$$T(n)=4\left[4T\left(\frac{n}{4}\right)+\frac{n}{2}\right]+n$$

Com a expansão do termo $T(n/4)$, podemos escrever:

$$T(n)=4\left[4\left[4T\left(\frac{n}{8}\right)+\frac{n}{4}\right]+\frac{n}{2}\right]+n=2^2\left[2^2\left[2^2T\left(\frac{n}{2^3}\right)+\frac{n}{2^2}\right]+\frac{n}{2}\right]+n$$

Extrapolando para um k arbitrário:

$$T(n)=(2^2)^k T\left(\frac{n}{2^k}\right)+(n+2n+2^2n+2^3n+\dots+2^{k-1}n)$$

Note que o somatório entre parêntesis é:

$$S=n \sum_{i=0}^{k-1} 2^i$$

Note que $2^{i+1}=2 \cdot 2^i=2^i+2^i$, o que nos leva a $2^i=2^{i+1}-2^i$. Então,

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

Como no caso limite temos $T(1)$, então $n=2^k$, o que nos leva a $k=\log_2 n$. Sendo assim,

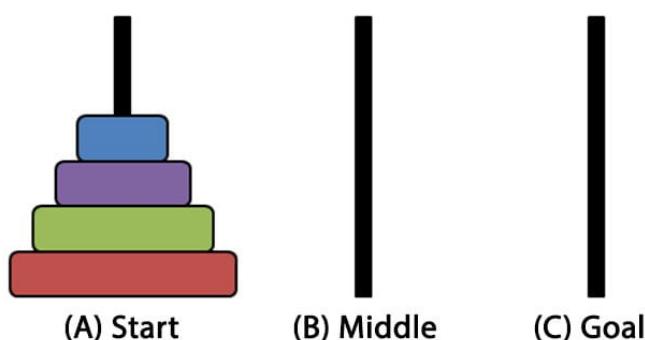
$$T(n)=(2^{2k})T(1)+n(2^k-1)=2^{\log_2 n^2}+n(2^{\log_2 n}-1)=n^2+n(n-1)=2n^2-n$$

o que é $O(n^2)$.

O problema da torre de Hanói

Imagine que temos 3 hastes (A, B e C) e inicialmente n discos de tamanhos distintos empilhados na haste A, de modo que discos maiores não podem ser colocados acima de discos menores. O objetivo consiste em mover todos os discos para uma outra haste. Há apenas duas regras:

1. Podemos mover apenas um disco por vez
2. Não pode haver um disco menor embaixo de um disco maior



Vejamos o que ocorre para diferentes valores de n (número de discos).

Se $n = 1$, basta um movimento: Move A, B

Se $n = 2$, são necessários 3 movimentos: Move A, B
 Move A, C
 Move B, C

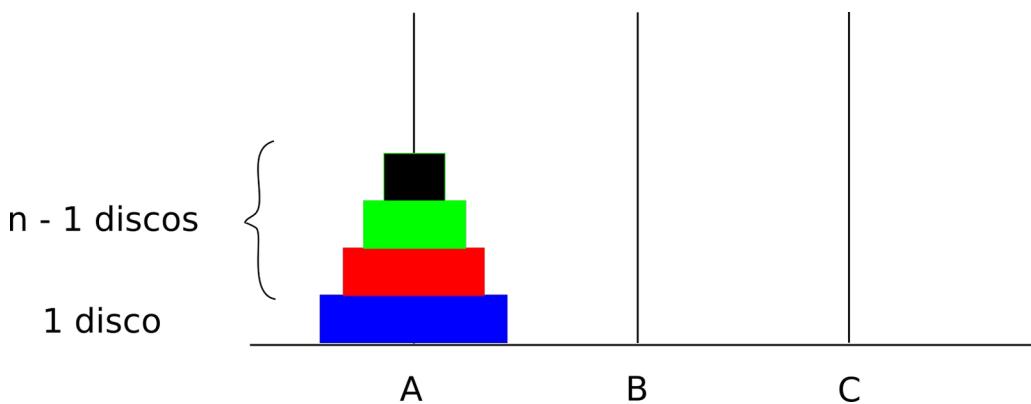
Se $n = 3$, são necessários 7 movimentos: Move A, B
 Move A, C
 Move B, C
 Move A, B
 Move C, A
 Move C, B
 Move A, B

Utilizando uma abordagem recursiva, note que são 3 movimentos para os dois menores discos, 1 para o maior e mais 3 movimentos para os dois menores

Se $n = 4$, são necessários 15 movimentos: utilizando a abordagem recursiva, temos 7 movimentos para os 3 menores discos, 1 movimento para o maior e mais 7 movimentos para os 3 menores, o que totaliza $7 + 1 + 7 = 15$ movimentos

Se $n = 5$, teremos $15 + 1 + 15 = 31$ movimentos

A essa altura deve estar claro que temos a seguinte lógica:



Para mover $n - 1$ discos menores: T_{n-1} movimentos

Para mover o maior disco: 1 movimento

Para mover de volta os $n - 1$ discos menores: T_{n-1} movimentos

```
# Move n discos de from para to usando aux
Hanoi(n, from, to, aux) {
    if n == 0
        return
    # Move recursivamente os n-1 discos superiores
    Hanoi(n-1, from, aux, to)
    # Move o maior disco
    print("Move disk", n, "from rod", from, "to rod", to)
    # Move recursivamente os n-1 discos
    Hanoi(n-1, aux, to, from)
}
```

```

# A, B, C são os nomes das hastes
# Chamando a função: mover N discos de A para C usando haste B
N = 5
Hanoi(N, 'A', 'C', 'B')

```

Como podemos calcular a complexidade desse algoritmo? Note que podemos definir a seguinte recorrência :

$$T(n) = 2T(n-1) + O(1)$$

Expandindo a recursão, podemos escrever:

$$T(n) = 2(2T(n-2) + O(1)) + O(1) = 2^2T(n-2) + O(1)$$

Novamente expandindo a recursão, temos:

$$T(n) = 2(2(2T(n-3) + O(1)) + O(1)) + O(1) = 2^3T(n-3) + O(1)$$

Prosseguindo com essas substituições até o k-ésimo termo, chega-se em:

$$T(n) = 2^k T(n-k) + O(1)$$

A condição de parada da recursão é termos $T(0)$, ou seja, $k = n$. Logo, temos:

$$T(n) = 2^n T(0) + O(1) = 2^n + O(1)$$

Portanto, $T(n) = O(2^n)$, mostrando que a função possui complexidade exponencial (proibitiva). Portanto, a solução para esse problema para um número elevado de discos é inviável. Por exemplo, suponha que tenhamos $n = 100$ discos. Então, o número de operações necessárias para resolver o problema da Torre de Hanói é $2^{100} = 1.2676506 \times 10^{30}$. Considerando que cada instrução é executada em 1 nanosegundo, ou seja, 10^{-9} segundos, o tempo gasto seria 1.2676506×10^{21} segundos, o que equivale a aproximadamente 4.01×10^{13} anos! Para se ter uma ideia de quão grande isso é, a idade estimada do universo é de 26.7×10^9 anos.

Busca sequencial x Busca binária

Uma tarefa fundamental na computação consiste em buscar um elemento em um array. A forma mais simples de buscar um elemento de um array é a busca sequencial. A função percorre todo array verificando se o elemento chave é igual ao elemento da i-ésima posição.

```

busca_sequencial(L, n, x) {
    i = 1
    while i <= n {
        if L[i] == x
            return i
        else
            i = i + 1
    }
    return False
}

```

Vamos analisar a complexidade da busca sequencial no pior caso, ou seja, quando o elemento a ser buscado encontra-se na última posição do vetor. Note que o loop executa $n - 1$ vezes a instrução de incremento no valor de i .

$$T(n) = 1 + (n - 1) = n$$

o que resulta em $O(n)$. A busca binária requer uma lista ordenada de elementos para funcionar. Ela imita o processo que nós utilizamos para procurar uma palavra no dicionário. Como as palavras estão ordenadas, a ideia é abrir o dicionário mais ou menos no meio. Se a palavra que desejamos inicia com uma letra que vem antes, então nós já descartamos toda a metade final do dicionário (não precisamos procurar lá, pois é certeza que a palavra estará na primeira metade). No algoritmo, temos uma lista com números ordenados. Basicamente, a ideia consiste em acessar o elemento do meio da lista. Se ele for o que desejamos buscar, a busca se encerra. Caso contrário, se o que desejamos é menor que o elemento do meio, a busca é realizada na metade a esquerda. Senão, a busca é realizada na metade a direita.

```
busca_binaria(L, x, ini, fim) {
    meio = (ini + fim) // 2
    if ini > fim
        return -1           # elemento não encontrado
    else {
        if L[meio] == x
            return meio
        else {
            if L[meio] > x
                return busca_binaria(L, x, ini, meio-1)
            else
                return busca_binaria(L, x, meio+1, fim)
        }
    }
}
```

Uma comparação entre o pior caso da busca sequencial e da busca binária, mostra a significativa diferença entre os métodos. Na busca sequencial, faremos n acessos para encontrar o valor procurado na última posição. Costuma-se dizer que o custo é $O(n)$ (é da ordem de n , ou seja, linear). Na busca binária, como a cada acesso descartamos metade das amostras restantes. Supondo, por motivos de simplificação, que o tamanho do vetor n é uma potência de 2, ou seja, $n = 2^m$, note que:

Acessos	Descartados
$m = 1$	$\rightarrow n/2$
$m = 2$	$\rightarrow n/4$
$m = 3$	$\rightarrow n/8$
$m = 4$	$\rightarrow n/16$

e assim sucessivamente. É possível notar um padrão? Podemos escrever a seguinte recorrência:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Expandindo a recorrência, temos:

$$T(n) = \left[T\left(\frac{n}{4}\right) + 1 \right] + 1$$

$$T(n) = \left\lceil T\left(\frac{n}{8}\right) + 1\right\rceil + 1 = T\left(\frac{n}{2^3}\right) + 3 \times 1$$

Extrapolando para um valor arbitrário de k:

$$T(n) = T\left(\frac{n}{2^k}\right) + k$$

No limite da recursão (condição de parada), temos que $\frac{n}{2^k} = 1$, ou seja, $n = 2^k$, o que implica que $k = \log_2 n$. Substituindo em T(n), chega-se a:

$$T(n) = T(1) + \log_2 n$$

o que é $O(\log n)$, ou seja, bem melhor que a busca sequencial que é $O(n)$.

A função $\log(n)$ tem uma curva de crescimento bem mais lento do que a função linear n . Veja que a derivada (taxa de variação) da função linear n é constante e igual a 1 sempre. A derivada da função $\log(n)$ é $1/n$, ou seja, quando n cresce, a taxa de variação, que é o que controla o crescimento da função, decresce. Na prática, isso significa que em uma lista com 1024 elementos, a busca sequencial fará no pior caso 1023 acessos até encontrar o elemento desejado. Na busca binária, serão necessários no pior caso apenas $\log_2 1024 = 10$ acessos, o que corresponde a aproximadamente 1% do necessário na busca sequencial! Isso porque na busca binária, a cada acesso, descartamos metade dos elementos do array.

"You will never speak to anyone more than you speak to yourself in your head. Be kind to yourself."
-- Author Unknown

Algoritmos de ordenação elementares

Ser capaz de ordenar os elementos de um conjunto de dados é uma das tarefas básicas mais requisitadas por aplicações computacionais. Como exemplo, podemos citar a busca binária, um algoritmo de busca muito mais eficiente que a simples busca sequencial. Buscar elementos em conjuntos ordenados é bem mais rápido do que em conjuntos desordenados. Existem diversos algoritmos de ordenação, sendo alguns mais eficientes do que outros. Neste tópico, iremos estudar o funcionamento de alguns deles, como Bubblesort, Selectionsort e Insertionsort. Além disso, analisaremos a complexidade de tais algoritmos para entender porque e quando devemos utilizá-los.

Problema: ordenação de dados

Entrada: uma sequência arbitrária de chaves $a_1, a_2, a_3, \dots, a_n$ (elementos em uma lista/array)

Saída: uma permutação da sequência de entrada tal que $a'_1 \leq a'_2 \leq a'_3 \leq \dots \leq a'_n$ (ordem crescente)

Uma primitiva básica utilizada nos algoritmo que iremos estudar é a função `swap(a, b)`, que realiza a troca das posições da chave a com a chave b.

```
swap(a, b) {
    temp = a
    a = b
    b = temp
}
```

Como podemos perceber, a complexidade desta operação é $O(1)$, o que é algo fundamental no cálculo das complexidades dos algoritmos.

O algoritmo Bubblesort

O algoritmo *Bubblesort* é uma das abordagens mais simplistas para a ordenação de dados. A ideia básica consiste em percorrer o vetor diversas vezes, em cada passagem fazendo flutuar para o topo da lista (posição mais a direita possível) o maior elemento da sequência. Esse padrão de movimentação lembra a forma como as bolhas em um tanque procuram seu próprio nível, e disso vem o nome do algoritmo (também conhecido como o método bolha)

Embora no melhor caso esse algoritmo necessite de apenas n operações relevantes, onde n representa o número de elementos no vetor, no pior caso são feitas n^2 operações. Portanto, diz-se que a complexidade do método é de ordem quadrática. Por essa razão, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados. A seguir veremos um pseudo-código desse algoritmo.

```
BubbleSort(L, n) {
    # Percorre cada elemento do array L
    for i = n-1 downto 1 {
        # Flutua o maior elemento para a posição mais a direita
        for j = 1 to i {
            if L[j] > L[j+1]
                swap(L[j], L[j+1])
        }
    }
}
```

O exemplo a seguir mostra o passo a passo necessário para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1^a passagem (levar maior elemento para última posição)

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6] em vermelho, não troca
[2, 5, 13, 7, -3, 4, 15, 10, 1, 6] em azul, troca
[2, 5, 7, 13, -3, 4, 15, 10, 1, 6]
[2, 5, 7, -3, 13, 4, 15, 10, 1, 6]
[2, 5, 7, -3, 4, 13, 15, 10, 1, 6]
[2, 5, 7, -3, 4, 13, 15, 10, 1, 6]
[2, 5, 7, -3, 4, 13, 10, 15, 1, 6]
[2, 5, 7, -3, 4, 13, 10, 1, 15, 6]
[5, 2, 7, -3, 4, 13, 10, 1, 6, 15]

2^a passagem (levar segundo maior para penúltima posição)

[2, 5, 7, -3, 4, 13, 10, 1, 6, 15]
[2, 5, 7, -3, 4, 13, 10, 1, 6, 15]
[2, 5, -3, 7, 4, 13, 10, 1, 6, 15]
[2, 5, -3, 4, 7, 13, 10, 1, 6, 15]
[2, 5, -3, 4, 7, 13, 10, 1, 6, 15]
[2, 5, -3, 4, 7, 10, 13, 1, 6, 15]
[2, 5, -3, 4, 7, 10, 1, 13, 6, 15]
[2, 5, -3, 4, 7, 10, 1, 6, 13, 15]

3^a passagem (levar terceiro maior para antepenúltima posição)

[2, 5, -3, 4, 7, 10, 1, 6, 13, 15]
[2, -3, 5, 4, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 1, 10, 6, 13, 15]
[2, -3, 4, 5, 7, 1, 6, 10, 13, 15]

4^a passagem

[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 1, 7, 6, 10, 13, 15]
[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]

5^a passagem

[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]
[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]
[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]
[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

6^a passagem

[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]

[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]

7^a passagem

[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

8^a passagem

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

9^a passagem

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

Análise da complexidade

Iremos considerar 3 cenários distintos: pior caso, caso médio e melhor caso.

Pior caso: vetor em ordem decrescente.

Observando a função definida anteriormente, note que no pior caso o segundo loop vai de 1 a i, sendo que na primeira vez i = n - 1, na segunda vez i = n - 2 e até i = 1. Sendo assim, o número de operações é dado por:

$$T(n) = ((n-1) + (n-2) + \dots + 1)$$

Já vimos na aula anterior que o somatório $1 + 2 + \dots + n$ é igual a $n(n + 1)/2$. Assim, temos:

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

o que nos leva a $O(n^2)$.

Melhor caso: é possível fazer uma ligeira modificação no algoritmo para torná-lo $O(n)$.

```
BubbleSort_M(L, n) {
    for i = n-1 downto 1 {
        s = 0
        # Flutua o maior elemento para a posição mais a direita
        for j = 1 to i {
```

```

        if L[j] > L[j+1] {
            swap(L[j], L[j+1])
            s = s + 1
        }
    } if s == 0      # não houve nenhuma troca
        break
}

```

No melhor caso, é possível fazer uma pequena modificação no Bubblesort para contar quantas inversões (trocas) ele realiza. Dessa forma, se uma lista já está ordenada e o Bubblesort não realiza nenhuma troca, o algoritmo pode terminar após o primeiro passo. Com essa modificação, se o algoritmo encontra uma lista ordenada, sua complexidade é $O(n)$, pois ele percorre a lista de n elementos uma única vez. Porém, para fins didáticos, a versão original apresentada anteriormente tem complexidade $O(n^2)$ mesmo no melhor caso, pois não faz a checagem de quantas inversões são realizadas.

Caso médio: devemos tirar uma média de todos os possíveis casos, supondo que são equiprováveis.

Lembre-se que na primeira iteração temos $n - 1$ trocas, na segunda iteração temos $n - 2$ e assim sucessivamente, até atingirmos 1 única troca na última iteração. Portanto, é como se tirássemos uma média do somatório do caso anterior para diversos valores de k , variando de 1 até $n - 1$.

$$T(n) = \frac{1}{n-1} \sum_{k=1}^{n-1} \left(\sum_{i=1}^k i \right)$$

Sabemos que o somatório $1 + 2 + 3 + \dots + k = k(k+1)/2$, o que nos leva a:

$$\frac{1}{n-1} \sum_{k=1}^{n-1} \left[\frac{k(k+1)}{2} \right] = T(n) = \frac{1}{n-1} \sum_{k=1}^{n-1} \left(\frac{1}{2}(k^2+k) \right) = \frac{1}{2(n-1)} \left[\sum_{k=1}^{n-1} k^2 + \sum_{k=1}^{n-1} k \right] \quad (*)$$

Vamos chamar o primeiro somatório de A e o segundo de B. O valor de B é facilmente calculado pois sabemos que $1 + 2 + 3 + \dots + n - 1 = n(n-1)/2$. Vamos agora calcular o valor de A, dado por:

$$A = \sum_{k=1}^{n-1} k^2$$

Para isso, iremos utilizar o conceito de soma telescópica. Lembre-se que:

$$(k+1)^3 = k^3 + 3k^2 + 3k + 1$$

de modo que podemos escrever

$$(k+1)^3 - k^3 = 3k^2 + 3k + 1$$

Aplicando somatório de ambos os lados, temos:

$$\sum_{k=1}^{n-1} [(k+1)^3 - k^3] = 3 \sum_{k=1}^{n-1} k^2 + 3 \sum_{k=1}^{n-1} k + \sum_{k=1}^{n-1} 1$$

Sabemos que o lado esquerdo da identidade acima é $n^3 - 1$, o que nos leva a:

$$n^3 - 1 = 3 \sum_{k=1}^{n-1} k^2 + 3 \frac{n(n-1)}{2} + n - 1$$

Rearranjando os termos:

$$\sum_{k=1}^{n-1} k^2 = \frac{n^3 - 1}{3} - \frac{n(n-1)}{2} - \frac{n-1}{3} \quad (**)$$

Substituindo (**) em (*):

$$T(n) = \frac{1}{2(n-1)} \left[\frac{n^3 - 1}{3} - \frac{n(n-1)}{2} - \frac{n-1}{3} + \frac{n(n-1)}{2} \right] = \frac{1}{2(n-1)} \left[\frac{n^3 - 1}{3} - \frac{n-1}{3} \right]$$

Como sabemos que $n^3 - 1 = (n-1)(n^2 + n + 1)$, podemos cancelar os termos comuns:

$$T(n) = \frac{1}{2} \left[\frac{(n^2 + n + 1)}{3} - \frac{1}{3} \right] = \frac{1}{6}(n^2 + n)$$

o que resulta em $O(n^2)$.

O algoritmo Insertionsort

Insertionsort, ou ordenação por inserção, é o algoritmo de ordenação que, dado um vetor inicial constrói um vetor final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadráticos, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

Podemos fazer uma comparação do Insertion sort com o modo de como algumas pessoas organizam um baralho num jogo de cartas. Imagine que você está jogando cartas. Você está com as cartas na mão e elas estão ordenadas. Você recebe uma nova carta e deve colocá-la na posição correta da sua mão de cartas, de forma que as cartas obedeçam a ordenação.

A cada nova carta adicionada a sua mão de cartas, a nova carta pode ser menor que algumas das cartas que você já tem na mão ou maior, e assim, você começa a comparar a nova carta com todas as cartas na sua mão até encontrar sua posição correta. Você insere a nova carta na posição correta, e, novamente, sua mão é composta de cartas totalmente ordenadas. Então, você recebe outra carta e repete o mesmo procedimento. Então outra carta, e outra, e assim por diante, até você não receber mais cartas. Esta é a ideia por trás da ordenação por inserção. Percorra as posições do vetor, começando com o índice um. Cada nova posição é como a nova carta que você recebeu, e você precisa inseri-la no lugar correto na sublista ordenado à esquerda daquela posição.

```
InsertionSort(L, n) {
    # Percorre cada elemento de L
    for i = 2 to n {
        k = i
        # Insere o pivô na posição correta
        while k > 1 and L[k] < L[k-1] {
            swap(L[k], L[k-1] = L[k-1])
            k = k - 1
        }
    }
}
```

O exemplo a seguir ilustra o funcionamento do algoritmo em um exemplo passo a passo.

Suponha o seguinte vetor de entrada.

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1^a passagem: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 13, 7, -3, 4, 15, 10, 1, 6]

2^a passagem: [2, 5, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 13, 7, -3, 4, 15, 10, 1, 6]

3^a passagem: [2, 5, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 7, 13, -3, 4, 15, 10, 1, 6]

4^a passagem: [2, 5, 7, 13, -3, 4, 15, 10, 1, 6] → [-3, 2, 5, 7, 13, 4, 15, 10, 1, 6]

5^a passagem: [-3, 2, 5, 7, 13, 4, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6]

6^a passagem: [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6]

7^a passagem: [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 10, 13, 15, 1, 6]

8^a passagem: [-3, 2, 4, 5, 7, 10, 13, 15, 1, 6] → [-3, 1, 2, 4, 5, 7, 10, 13, 15, 6]

9^a passagem: [-3, 1, 2, 4, 5, 7, 10, 13, 15, 6] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

Análise da complexidade

Iremos considerar 3 cenários distintos: pior caso, caso médio e melhor caso.

Melhor caso: note que quando o array já está ordenado, os pivôs já estão nas posições corretas. Assim, nenhuma troca será necessária. Logo:

$$T(n) = \sum_{i=2}^n 1 = n - 1$$

pois, dentro do loop mais externo, apenas uma instrução será executada, o que resulta em O(n).

Pior caso: note que neste caso, o primeiro pivô realizará 1 troca, o segundo pivô realizará 2 trocas, e assim sucessivamente, o que nos leva a:

$$T(n) = \sum_{i=2}^n \left(1 + \sum_{j=1}^{i-1} 2 \right)$$

pois no pior caso a posição correta do pivô será sempre em k = 1 de modo que o segundo loop vai ter de percorrer todo vetor (i-1 trocas). Expandindo os somatórios, temos:

$$T(n) = \sum_{i=2}^n 1 + \sum_{i=2}^n \left(\sum_{j=1}^{i-1} 2 \right) = (n-1) + \sum_{i=2}^n 2(i-1) = (n-1) + 2 \sum_{i=2}^n i - \sum_{i=2}^n 2$$

O valor do primeiro somatório é:

$$\frac{n(n+1)}{2} - 1$$

de modo que $T(n)$ pode ser expressa por:

$$T(n) = n - 1 + 2 \left[\frac{n(n+1)}{2} - 1 \right] - 2(n-1) = n(n+1) - 2 - (n-1) = n^2 - 1$$

o que resulta em uma complexidade $O(n^2)$.

Caso médio: podemos aplicar uma estratégia muito similar àquela adotada na análise do Bubblesort. A ideia consiste em considerar que todos os casos são igualmente prováveis e calcular uma média de todos eles. Assim, temos:

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

A soma S entre colchetes pode ser expressa como:

$$S = \sum_{i=2}^{k+1} 1 + \sum_{i=2}^{k+1} \sum_{j=1}^{i-1} 2 = k + \sum_{i=2}^{k+1} 2(i-1) = k + 2 \sum_{i=2}^{k+1} i - 2 \sum_{i=2}^{k+1} 1 = k + 2 \left[\frac{k(k+1)}{2} - 1 + (k+1) \right] - 2k$$

Simplificando a expressão, chega-se em:

$$S = k + k(k+1) + 2k - 2k = k + k^2 + k = k^2 + 2k$$

Substituindo a S em $T(n)$, temos:

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

É fácil notar que o segundo somatório resulta em $n(n-1)$. O primeiro somatório já foi calculado na análise do algoritmo Bubblesort (***) e vale:

$$\sum_{k=1}^{n-1} k^2 = \frac{n^3 - 1}{3} - \frac{n(n-1)}{2} - \frac{n-1}{3} = \frac{(n-1)(n^2+n+1)}{3} - \frac{n(n-1)}{2} - \frac{n-1}{3}$$

Voltando a $T(n)$, podemos escrever:

$$T(n) = \frac{1}{n-1} \left(\frac{(n-1)(n^2+n+1)}{3} - \frac{n(n-1)}{2} - \frac{n-1}{3} + n(n-1) \right) = \frac{1}{3} n^2 + \frac{1}{3} n + \frac{1}{3} - \frac{1}{2} n - \frac{1}{3} + n = \frac{1}{3} n^2 + \frac{5}{6} n$$

o que resulta em complexidade $O(n^2)$.

O algoritmo Selectionsort

A ordenação por seleção é um método baseado em passar o menor valor do vetor para a primeira posição mais a esquerda disponível, depois o de segundo menor valor para a segunda posição a esquerda e assim sucessivamente, com os $n - 1$ elementos restantes. Esse algoritmo compara a cada

iteração um elemento com os demais, visando encontrar o menor. A complexidade desse algoritmo será sempre de ordem quadrática, isto é o número de operações realizadas depende do quadrado do tamanho do vetor de entrada. Algumas vantagens desse método são: é um algoritmo simples de ser implementado, não usa um vetor auxiliar e portanto ocupa pouca memória, é um dos mais velozes para vetores pequenos. Como desvantagens podemos citar o fato de que ele não é muito eficiente para grandes vetores.

```
SelectionSort(L, n) {
    # Percorre todos os elementos de L
    for i = 1 to n {
        menor = i
        # Encontra o menor elemento
        for k = i+1 to n {
            if L[k] < L[menor]
                menor = k
        }
        # Troca a posição do elemento i com o menor
        swap(L[menor], L[i])
    }
}
```

O exemplo a seguir mostra os passos necessários para a ordenação do seguinte vetor:

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1^a passagem: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [-3, 2, 13, 7, 5, 4, 15, 10, 1, 6]

2^a passagem: [-3, 2, 13, 7, 5, 4, 15, 10, 1, 6] → [-3, 1, 13, 7, 5, 4, 15, 10, 2, 6]

3^a passagem: [-3, 1, 13, 7, 5, 4, 15, 10, 2, 6] → [-3, 1, 2, 7, 5, 4, 15, 10, 13, 6]

4^a passagem: [-3, 1, 2, 7, 5, 4, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6]

5^a passagem: [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6]

6^a passagem: [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 6, 15, 10, 13, 7]

7^a passagem: [-3, 1, 2, 4, 5, 6, 15, 10, 13, 7] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

8^a passagem: [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

9^a passagem: [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

Análise da complexidade

Iremos considerar 3 cenários distintos: pior caso, caso médio e melhor caso.

Pior caso: note que no pior caso o segundo loop vai de $i+1$ até n , sendo que na primeira vez $i = 1$, na segunda vez $i = 2$ e até $i = n - 1$ (a atribuição dentro do FOR é realizada toda vez). Sendo assim, o número de operações é dado por:

$$T(n) = \sum_{i=1}^n \left(2 + \sum_{j=i+1}^n 1 \right) = 2 \sum_{i=1}^n 1 + \sum_{i=1}^n \left(\sum_{j=i+1}^n 1 \right)$$

Note que:

$$\sum_{i=2}^5 1 = (5-2)+1 = 4$$

ou seja,

$$\sum_{k=i+1}^n 1 = n - (i+1) + 1 = n - i$$

Então, podemos escrever:

$$T(n) = 2n + \sum_{i=1}^n (n-i) = 2n + \sum_{i=1}^n n - \sum_{i=1}^n i = 2n + n^2 - \frac{n(n+1)}{2} = 2n + n^2 - \frac{n^2}{2} - \frac{n}{2} = \frac{1}{2}n^2 + \frac{3}{2}n$$

o que resulta em $O(n^2)$.

Melhor caso: note que mesmo que a lista já esteja ordenada, o loop FOR interno será executado todas as vezes! Uma desvantagem do algoritmo Selectionsort é que mesmo no melhor caso, para encontrar o menor elemento, devemos percorrer todo o restante do vetor no loop mais interno.

O comando FOR possui uma instrução de atribuição embutida:

<pre>for i = 1 to n print(i)</pre>	\rightarrow $i = 1$ $\text{while } i \leq n \{$ $\quad \text{print}(i)$ $\quad i = i + 1$ $\}$
--	---

Assim, pelo mesmo raciocínio do pior caso, a complexidade é $O(n^2)$.

Caso médio: como tanto o melhor caso quanto o pior caso possuem complexidade quadrática, temos que o caso médio também é $O(n^2)$.

Em resumo, a tabela a seguir faz uma comparação das complexidades dos cinco algoritmos de ordenação apresentados aqui, no pior caso, caso médio e melhor caso.

Algoritmo	Melhor	Médio	Pior
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Para os interessados em aprender mais sobre o assunto, a internet contém diversos materiais sobre algoritmos de ordenação.

Outro detalhe interessante está relacionado com a estabilidade. Um algoritmo de ordenação é considerado estável ele mantém a ordem relativa dos registros em caso de igualdade de chaves. Importante quando ordenamos strings pelo primeiro caracter (i.e., Processamento de Linguagem Natural). Dentre os algoritmos anteriores temos a seguinte classificação:

Algoritmo	Estável
Bubblesort	Sim
Selectionsort	Não
Insertionsort	Sim

Há vários recursos multimídias disponíveis na internet para ilustrar o funcionamento de algoritmos de ordenação. Dentre elas destacam-se algumas animações que demonstram o funcionamento dos algoritmos de ordenação visualmente. A seguir indicamos alguns links interessantes:

15 sorting algorithms in 6 minutes (Animações sonorizadas muito boas para visualização)
<https://www.youtube.com/watch?v=kPRA0W1kECg>

Animações passo a passo dos algoritmos
<https://visualgo.net/en/sorting>

Comparação em tempo real dos algoritmos
<https://www.toptal.com/developers/sorting-algorithms>

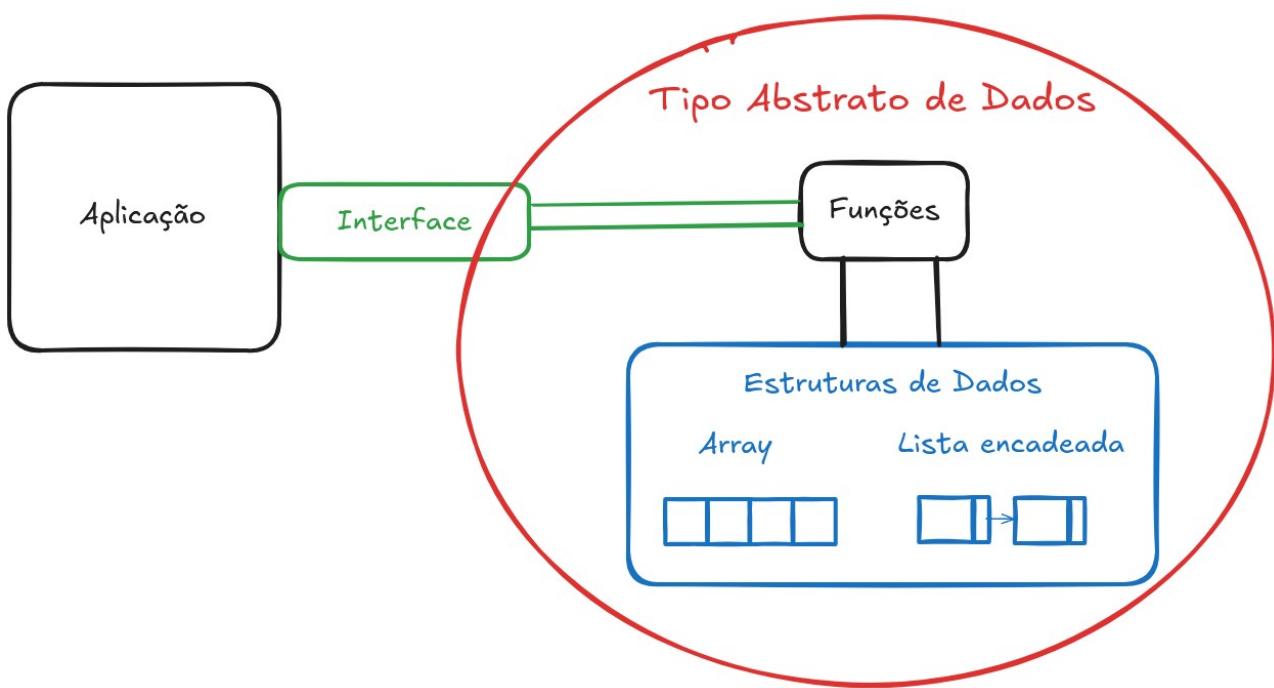
Algoritmos de ordenação como danças em grupo
<https://www.youtube.com/user/AlgoRythmics>

"If you feel like you're losing everything, remember that trees lose their leaves every year and they still stand tall and wait for better days to come."
-- Author Unknown

Tipos Abstratos de Dados (TAD's)

Na programação de computadores, o conceito de abstração é fundamental para o desenvolvimento de software de alto nível. Um exemplo são as funções, que são abstrações de processos. Uma vez que uma função é criada, toda lógica é encapsulada do usuário. Sempre que o programador necessitar, basta invocar a função, sem que ele precise conhecer os detalhes da implementação. Conforme a complexidade dos programas aumenta, torna-se necessário definir abstrações para dados. É para essa finalidade que foram criados os Tipos Abstratos de Dados (TAD's), assunto que iremos explorar em detalhes a seguir.

Um Tipo Abstrato de Dados, ou TAD, é um tipo de dados definido pelo programador que especifica um conjunto de variáveis que são utilizadas para armazenar informação e um conjunto de operações bem definidas sobre essas variáveis. TAD's são definidos de forma a ocultar a sua implementação, de modo que um programador deve interagir com as variáveis internas a partir de uma interface, definida em termos do conjunto de operações. A Figura a seguir ilustra essa ideia: a grande vantagem é que, depois de definido o TAD, não é preciso conhecer os detalhes internos de sua implementação para utilizá-lo, basta conhecer sua interface (como ativar as funções internas).



TAD's são considerados os precursores da programação orientada a objetos (POO). Um exemplo de TAD implementado nativamente pela linguagem Python via orientação a objetos são as listas. Uma lista em Python consiste basicamente de uma variável composta heterogênea (pode armazenar informações de tipos de dados distintos) utilizada para armazenar as informações mais um conjunto de operações para manipular essa variável. Não é necessário conhecer os detalhes internos da implementação de uma lista em Python. Basta conhecer as interfaces para utilizar as funções da maneira correta.

- `L.append(x)`: adiciona x no final da lista L
- `L.pop()`: remove o último elemento da lista L
- `L.pop(i)`: remove o elemento da posição i
- `L.insert(i, x)`: insere elemento x na posição i
- `L.reverse()`: inverte a lista L
- `L.sort()`: ordena os elementos da lista L

Note que para o programador, a implementação desses processos fica encapsulada, sendo que não é preciso saber os detalhes, basta conhecer a interface das funções, ou seja, quais os parâmetros necessários para invocá-las. Por exemplo, na função `L.insert(i, x)` o primeiro parâmetro deve ser o índice da posição do elemento na lista e o segundo parâmetro deve ser o valor a ser armazenado.

Existem diversas vantagens de se trabalhar com TAD's em programação, em especial na implementação de estruturas de dados:

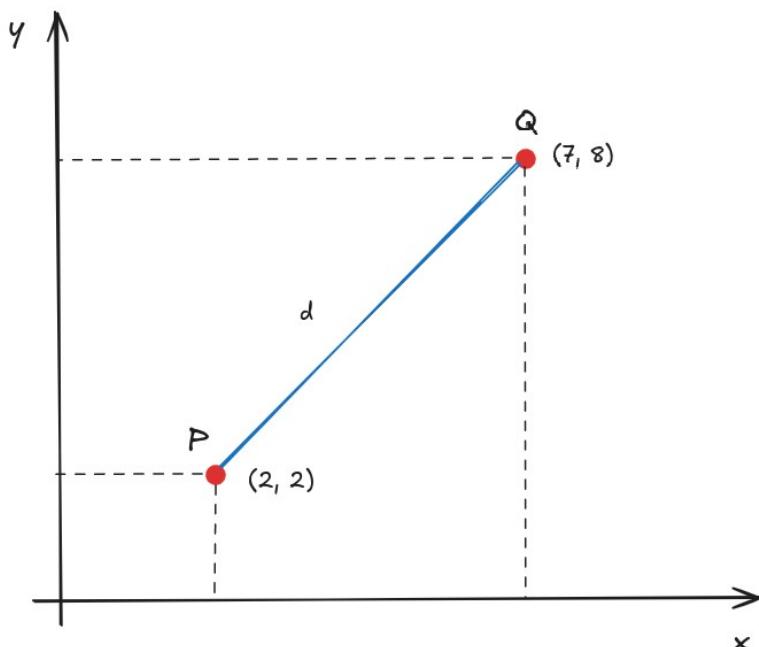
1. Foco na resolução do problema e não nos detalhes de implementação.
2. Redução de erros pelo encapsulamento de código validado.
3. Correção de bugs e manutenção de código (podemos modificar a parte interna de um TAD sem se preocupar com o programa que utiliza o TAD).
4. Redução da complexidade no desenvolvimento de software, pois é mais fácil dividir um programa muito extenso em pequenos módulos separados de modo que times possam trabalhar de maneira independente

Um TAD é uma abstração que pode ser implementado tanto em linguagens funcionais como C e Pascal, quanto em linguagens orientadas a objetos como C++, Java e Python. Em POO a implementação de um TAD gera o conceito de classe, a qual após ser instanciada torna-se um objeto. Uma classe é como um template composto por um conjunto de atributos, que são as variáveis internas utilizadas para armazenar informações, e um conjunto de métodos, que são as operações (funções) utilizadas para processar seus atributos (variáveis internas).

Exemplo: Criar um TAD Ponto, para representar um ponto no plano e algumas operações básicas.

As operações que iremos definir são:

1. cria : operação que cria um ponto com coordenadas (x, y).
2. libera : operação que libera a memória alocada por um ponto.
3. acessa : operação que retorna as coordenadas de um ponto.
4. atribui : operação que atribui novos valores às coordenadas de um ponto.
5. distancia : operação que calcula a distância entre dois pontos.



Em linguagem C, primeiramente, define-se o arquivo da interface, ponto.h como segue.

```
// TAD: Ponto (interface)

//***** Tipo exportado
typedef struct ponto Ponto;

//***** Funções exportadas
// Função cria
// Aloca e retorna a um ponto com coordenadas (x, y)
Ponto *pto_cria(float x, float y);

// Função libera
// Libera a memória de um ponto previamente criado
void pto_libera(Ponto *p);

// Função acessa (get)
// Retorna os valores das coordenadas de um ponto
void pto_acessa(Ponto *p, float *x, float *y);

// Função atribui (set)
// Atribui novos valores às coordenadas de um ponto
void pto_atribui(Ponto *p, float x, float y);

// Função distancia
// Retorna a distância entre dois pontos
float pto_distancia(Ponto *p1 , Ponto *p2 );
```

Em seguida, devemos definir a implementação do TAD no arquivo ponto.c como segue.

```
# include <stdlib.h> // malloc , free , exit
# include <stdio.h> // printf
# include <math.h> // sqrt
# include "ponto.h" // TAD ponto (interface)

//*****
// Implementação do TAD
//*****

// Definição da estrutura ponto
struct ponto {
    float x;
    float y;
};

// Função para criar ponto
Ponto *pto_cria(float x, float y) {
    Ponto *p = (Ponto *)malloc(sizeof(Ponto));
    if (p == NULL) {
        printf("Memória insuficiente!\n");
        exit(1);
    }
    p->x = x;
    p->y = y;
    return p;
}
```

```

// Função para liberar memória
void pto_libera(Ponto *p) {
    free(p);
}

// Função get
void pto_acessa(Ponto *p, float *x, float *y) {
    *x = p->x;
    *y = p->y;
}

// Função set
void pto_atribui(Ponto *p, float x, float y) {
    p->x = x;
    p->y = y;
}

// Função para calcular a distância entre 2 pontos
float pto_distancia(Ponto *p1, Ponto *p2) {
    float dx = p2->x - p1->x;
    float dy = p2->y - p1->y;
    return sqrt(dx*dx + dy*dy);
}

// Função main (usa o TAD criado)
int main(void) {
    Ponto *p = pto_cria(2.0, 1.0);
    Ponto *q = pto_cria(3.4, 2.1);
    float d = pto_distancia(p, q);
    printf("Distância entre pontos: %f\n", d);
    pto_libera(q);
    pto_libera(p);
    return 0;
}

```

Exercício: Crie um TAD círculo que contenha a definição de um tipo de dado para representar um círculo e funções para manipulá-lo.

1. cria : operação que cria um círculo com centro (x,y) e raio r.
2. libera : operação que libera a memória alocada por um círculo.
3. area : operação que calcula a área do círculo.
4. interior : operação que verifica se um dado ponto está dentro do círculo.

Note que um círculo é composto por um Ponto que indica o seu centro e por um raio.

Primeiramente, define-se o arquivo de interface circulo.h, conforme a seguir.

```

// TAD: Circulo (interface)

#include "ponto.h"

//***** Tipo exportado
typedef struct circulo Circulo;

```

```

//***** Funções exportadas
// Função cria
// Aloca e retorna um círculo com centro (x, y) e raio r
Circulo *circ_cria(float x, float y, float r);

// Função libera
// Libera a memória de um ponto previamente criado
void circ_libera(Circulo *c);

// Função area
// Retorna a área do círculo
float circ_area(Circulo *c);

// Função interior
// Verifica se um dado ponto está no interior do círculo
int circ_interior(Circulo *c, Ponto *p);

```

Em seguida, devemos definir a implementação do TAD no arquivo circulo.c como segue.

```

#include <stdlib.h> // malloc , free , exit
#include <stdio.h> // printf
#include <math.h> // sqrt
#include "circulo.h" // TAD ponto (interface)

#define PI 3.14159

//*****
// Implementação do TAD
//*****

// Definição da estrutura ponto
struct circulo {
    Ponto *p; // Centro
    float r; // Raio
};

// Função para criar Circulo
Circulo *circ_cria(float x, float y, float r) {
    Circulo *c = (Circulo *)malloc(sizeof(Circulo));
    c->p = pto_cria(x, y);
    c->r = r;
    return c;
}

// Função para liberar memória
void circ_libera(Circulo *c) {
    pto_libera(c->p);
    free(c);
}

// Função para calcular a area
float circ_area(Circulo *c) {
    return PI*(c->r)*(c->r);
}

```

```

// Função para checar se um ponto está dentro do círculo
int circ_interior(Circulo *c, Ponto *p) {
    float d = pto_distancia(c->p, p);
    return (d < c->r);
}

// Função main (usa o TAD criado)
int main (void) {
    Circulo *c = circ_cria(5, 5, 2);
    float area = circ_area(c);
    printf("Área do círculo: %f\n", area);
    Ponto *p = pto_cria(4, 4);
    if (circ_interior(c, p) == 0)
        printf("Ponto f fora do círculo");
    else
        printf("Ponto p dentro do círculo");
    pto_libera(p);
    circ_libera(c);
    return 0;
}

```

Exercício: Crie um TAD NumeroComplexo que contenha a definição de um tipo de dado para representar um número complexo e as seguintes funções:

1. Função para criar um número complexo, dados a e b (parte real e imaginária).
2. Função para liberar um número complexo previamente criado.
3. Função para somar dois números complexos, retornando um novo número com o resultado da operação. Sabe-se que: $(a + bi) + (c + di) = (a + c) + (b + d)i$
4. Função para subtrair dois números complexos, retornando um novo número com o resultado da operação. Sabe-se que: $(a + bi) - (c + di) = (a - c) + (b - d)i$
5. Função para multiplicar dois números complexos, retornando um novo número com o resultado da operação. Sabe-se que: $(a + bi)(c + di) = (ac - bd) + (bc + ad)i$
6. Função para dividir dois números complexos, retornando um novo número com o resultado da operação. Sabe-se que:

$$\frac{a + bi}{c + di} = \left(\frac{ac + bd}{c^2 + d^2} \right) + \left(\frac{bc - ad}{c^2 + d^2} \right)$$

Mas como podemos criar um TAD em Python, uma linguagem orientada a objetos? Em POO, o conceito de TAD equivale ao conceito da definição de uma classe. Para criarmos uma classe Ponto, podemos utilizar a seguinte definição no arquivo ponto.py:

```

import math

class Ponto:
    # Construtor: usado para instanciar novos objetos
    def __init__(self, coord_x, coord_y):
        self.x = coord_x
        self.y = coord_y

    # Obtém as coordenadas x e y do ponto
    def pto_acessa(self):
        return (self.x, self.y)

    # Atribui novas coordenadas x e y para ponto

```

```

def pto_atribui(self, coord_x, coord_y):
    self.x = coord_x
    self.y = coord_y

# Calcula a distância entre 2 pontos no plano
def pto_distancia(self, other_point):
    dx = other_point.x - self.x
    dy = other_point.y - self.y
    return math.sqrt(dx**2 + dy**2)

if __name__ == '__main__':
    # Cria objetos
    p = Ponto(2.0, 1.0)
    q = Ponto(3.4, 2.1)
    d = p.pto_distancia(q)
    print('Distância entre pontos: %f' %d)

```

Exercício: Desenvolva uma classe Equacao_Quadratica em Python para criar e resolver uma equação do segundo grau.

```

from math import sqrt
from math import isnan

class Equacao_Quadratica:

    # Construtor (usado para instanciar novos objetos)
    def __init__(self, a, b, c):
        self.a = float(a)
        self.b = float(b)
        self.c = float(c)
        self.delta = float('nan')
        self.x1 = float('nan')
        self.x2 = float('nan')

    # Retorna string para imprimir equação com comando print
    def __str__(self):
        return str(self.a) + 'x^2+' + str(self.b) + 'x+' + str(self.c)

    # Verifica se é equação do segundo grau (a não é zero)
    def eh_quadratica(self):
        if self.a != 0:
            return True

    # Calcula o valor de delta
    def calcula_delta(self):
        self.delta = self.b**2 - 4 * self.a * self.c

    # Raiz real: verifica se a equação possui raízes reais
    def raiz_real(self):
        if self.delta >= 0:
            return True

    # Resolve equação do 2º grau
    def resolve(self):
        # Copiando variáveis para código menor
        a = self.a

```

```

b = self.b
# Verifica se é equação quadrática
if self.eh_quadratica():
    self.calcula_delta()
    delta = self.delta
    # Verifica se as raízes são reais
    if self.raiz_real():
        self.x1 = (-b-sqrt(delta))/(2*a)
        self.x2 = (-b+sqrt(delta))/(2*a)
        return (self.x1, self.x2)
    else:
        print('A equação não admite raízes reais')
else:
    print('A equação não é do segundo grau (coef. a = 0)')

if __name__ == '__main__':
    # Cria equação do segundo grau
    equacao = Equacao_Quadratica(1, -5, 6)
    # Imprime na tela
    print(equacao)
    # Resolve a equação utilizando a fórmula quadrática
    print(equacao.resolve())

```

Exercício: Refaça o exercício criando um TAD em linguagem C.

Em resumo, TAD's serão muito úteis na implementação das estruturas de dados que iremos estudar ao longo deste curso, como as Pilhas, as Filas, as Listas Encadeadas e as Árvores Binárias. Quando implementamos uma estrutura de dados, é preciso definir uma série de operações básicas que operam sobre os dados. Por exemplo, em Pilhas e Filas, é preciso tanto inserir quanto remover chaves de uma maneira própria, o que define o funcionamento correto de cada uma delas. Sendo assim, duas estruturas de dados podem até ter as mesmas funções básicas, porém o comportamento de cada uma delas deve ser específico. Sendo assim, a utilização de TAD's nos permite implementar estruturas de dados de maneira mais clara, objetiva e modularizada.

"A hero is an ordinary individual who finds the strength to persevere and endure in spite of overwhelming obstacles."

-- Christopher Reeve

Estruturas de dados lineares estáticas

Na programação de computadores, estruturas de dados são abstrações lógicas criadas para organizarmos dados na memória de maneira eficiente para o posterior acesso e manipulação. Do ponto de vista prático, estruturas de dados são compostas por coleções de elementos, o relacionamento entre esses elementos e as funções/operações que podem ser aplicadas nesses elementos.

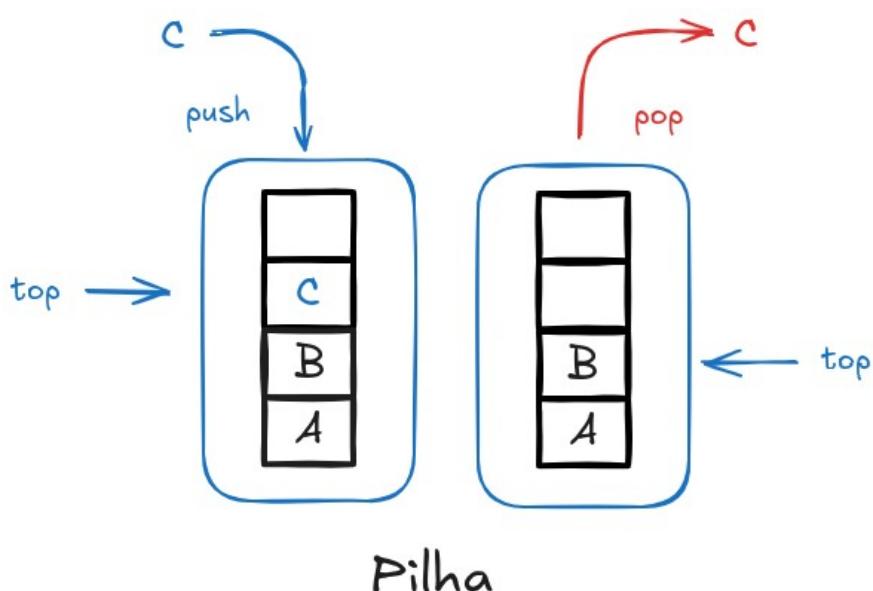
Iniciaremos nosso estudo com Pilhas e Filas, que são coleções em que os elementos são organizados dependendo de como eles são adicionados ou removidos do conjunto. Uma vez que um elemento é adicionado ele permanece na mesma posição relativa ao elemento que veio antes e o elemento que veio depois. Essa é a característica comum de todas as estruturas lineares.

Sendo bastante simplista, estruturas de dados lineares são aquelas que possuem duas extremidades: início e fim, ou esquerda e direita, ou base e topo. A nomenclatura não é relevante para nós, pois o que realmente importa é como são realizadas as inserções e remoções de elementos na estrutura.

Pilhas (LIFO)

Uma pilha (stack) é uma estrutura de dados linear em que a inserção e a remoção de elementos é realizada sempre na mesma extremidade, comumente de chamada de topo. O oposto do topo é a base. Quanto mais próximo da base está um elemento, a mais tempo ele está armazenado na estrutura. Por outro lado, um item inserido agora estará sempre no topo, o que significa que ele será o primeiro a ser removido (se não empilharmos novos elementos antes). Por esse princípio de ordenação inerente das pilhas, elas são conhecidas como a estrutura **LIFO**, do inglês, *Last In First Out*, ou seja, primeiro a entrar é último a sair. Essa intuição faz total sentido com uma pilha de livros por exemplo. O primeiro livro a ser empilhado fica na base da pilha e será o último a ser retirado. A ordem de remoção é o inverso da ordem de inserção. A figura a seguir ilustra uma pilha de caracteres.

Para que possamos implementar um TAD Pilha, devemos ter em mente quais suas variáveis internas e quais as operações que podemos aplicar sobre os seus elementos. A listagem a seguir mostra como podemos construir a pilha da figura acima, a partir de uma pilha inicialmente vazia. Note que podemos utilizar uma lista P para armazenar os elementos da pilha.



Operação	Conteúdo	Retorno	Descrição
is_empty(S)	[]	True	Verifica se pilha está vazia
push(S, 4)	[4]		Insere elemento no topo
push(S, 9)	[4, 9]		Insere elemento no topo
peek(S)	[4, 9]	9	Consulta o elemento do topo
push(5)	[4, 9, 5]		Insere elemento no topo
size(S)	[4, 9, 5]	3	Número de elementos da pilha
is_empty(S)	[4, 9, 5]	False	Verifica se pilha está vazia
push(8)	[4, 9, 5, 8]		Insere elemento no topo
pop(S)	[4, 9, 5, 8]	8	Remove elemento do topo
pop(S)	[4, 9, 5]	5	Remove elemento do topo
size(S)	[4, 9]	2	Número de elementos da pilha

Antes de implementar as funções necessárias para operar uma estrutura de dados Pilha, iremos definir a interface das funções, que são primitivas básicas utilizadas para manipular a estrutura.

```
# Inicializa pilha vazia
init(S)
# Verifica se pilha está vazia
is_empty(S)
# Verifica o topo da pilha
peek(S)
# Insere elemento no topo da pilha
push(S, key)
# Remove elemento do topo da pilha
pop(S, key)
# Verifica quantos elementos existem na pilha
size(S)
```

Uma pilha S pode ser implementada de maneira estática como um array de tamanho n, de modo que o último elemento inserido é referenciado pela variável top (topo da pilha).

TAD Stack

int array[1..n] keys	# n é o tamanho da pilha
int top	# topo da pilha

A função init(S), inicializa uma pilha S, fazendo top ser igual a zero.

```
init(S) {
    S.top = 0
}

is_empty(S) {
    if S.top == 0
        return True
    else
        return False
}

peek(S) {
    return S.keys[S.top]
}

push(S, key) {
    if S.top == n
```

```

        error('overflow')
    else {
        S.top += 1
        S.keys[S.top] = key
    }
}

pop(S) {
    if is_empty(S)
        error('underflow')
    else {
        key = S.keys[S.top]
        S.keys[S.top] = NIL
        S.top -= 1
        return key
    }
}

```

Do ponto de vista de complexidade, repare que as primitivas para inserção e remoção de chaves definidas anteriormente são O(1), ou seja, são realizadas em tempo constante.

Problema: Uma aplicação interessante que utiliza uma estrutura de dados do tipo Pilha é a verificação do balanceamento dos parêntesis de uma expressão matemática. Uma expressão matemática é bem formada se o número de parêntesis é par, sendo que para cada (deve existir um). Por exemplo, a expressão a seguir é válida:

$((1 + 2) * (3 + 4)) - (5 + 6)$

Já expressão a seguir não é válida:

$((1 + 2) * (3 + 4) - (5 + 6)$

Como desenvolver um algoritmo que verifique se uma dada expressão é válida ou não? A ideia consiste em utilizar uma pilha para empilhar cada abre parêntesis que aparece na expressão e ao encontrar um fecha parêntesis, devemos desempilhar o seu par da pilha. Se ao final do processo a pilha estiver vazia, ou seja, para cada (existe um respectivo), então a fórmula é considerada válida.

```

# Apenas parêntesis são permitidos na expressão: ( , )
verifica_expressao_simples(expressao, n) {
    # Cria pilha vazia de tamanho n
    S = Stack()
    balanced = True
    indice = 1           # primeira posição na expressão
    # Enquanto não chegar no final e estiver balanceado
    while indice <= n and balanced {
        # Obtém o símbolo atual
        simbolo = expressao[indice]
        # Se for abre parêntesis, empilha
        if simbolo == '('
            push(S, simbolo)
        # Se for fecha parêntesis, o abre tem que estar na pilha
        # Se não estiver na pilha, não está平衡ado
        elif simbolo == ')' {
            if is_empty(S)
                balanced = False
        }
    }
}

```

```

        else:          # Se abre parêntesis está na pilha, OK
            pop(S)
    }
    # Passa para o próximo caracter da expressão
    indice += 1
}
# Se ao final estiver balanceado e não sobrou nada na pilha, OK
if balanced and is_empty(S):
    return True
else:
    return False
}

```

No caso de expressões compostas, em que existem vários tipos de símbolos, como {}, [e (, ao desempilhar o fechamento, devemos nos atentar se o par é bem formado, ou seja, se temos (), [] ou {}). Como podemos validar se uma expressão composta é bem formada? Esse é um bom exercício de programação. Refaça o exercício anterior considerando agora que temos 3 tipos de símbolos de parentização: (, [, { e seus respectivos fechamentos),], }..

Problema: Outro problema interessante que pode ser resolvido com uma estrutura de dados do tipo Pilha é a conversão de um número decimal (base 10) para binário (base 2). A representação de números inteiros por computadores digitais é feita na base 2 e não na base 10. Sabemos que um número na base 10 (decimal) é representado como:

$$23457 = 2 \times 10^4 + 3 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

Veja o quanto mais a esquerda estiver o dígito, maior o seu valor. O 2 em 23457 vale na verdade 20000 pois é igual a 2×10^4 .

Analogamente, um número binário (base 2) pode ser representado como:

$$110101 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 53$$

O bit mais a direita é o menos significativo e portanto o seu valor é $1 \times 2^0 = 1$
O segundo bit a partir da direita tem valor de $0 \times 2^1 = 0$
O terceiro bit a partir da direita tem valor de $1 \times 2^2 = 4$
O quarto bit a partir da direita tem valor de $0 \times 2^3 = 0$
O quinto bit a partir da esquerda tem valor de $1 \times 2^4 = 16$
Por fim, o bit mais a esquerda tem valor de $1 \times 2^5 = 32$
Somando tudo temos: $1 + 4 + 16 + 32 = 5 + 48 = 53$.

Essa é a regra para convertermos um número binário para sua notação decimal.

Veremos agora o processo inverso: como converter um número decimal para binário. O processo é simples. Começamos dividindo o número decimal por 2:

$$53 / 2 = 26 \text{ e sobra resto } 1 \rightarrow \text{esse } 1 \text{ será nosso bit mais a direita (menos significativo no binário)}$$

Continuamos o processo até que a divisão por 2 não seja mais possível:

$$26 / 2 = 13 \text{ e sobra resto } 0 \rightarrow \text{esse } 0 \text{ será nosso segundo bit mais a direita no binário}$$

$$13 / 2 = 6 \text{ e sobra resto } 1 \rightarrow \text{esse } 1 \text{ será nosso terceiro bit mais a direita no binário}$$

$6 / 2 = 3$ e sobra resto **0** → esse 0 será nosso quarto bit mais a direita no binário

$3 / 2 = 1$ e sobra resto **1** → esse 1 será nosso quinto bit mais a direita no binário

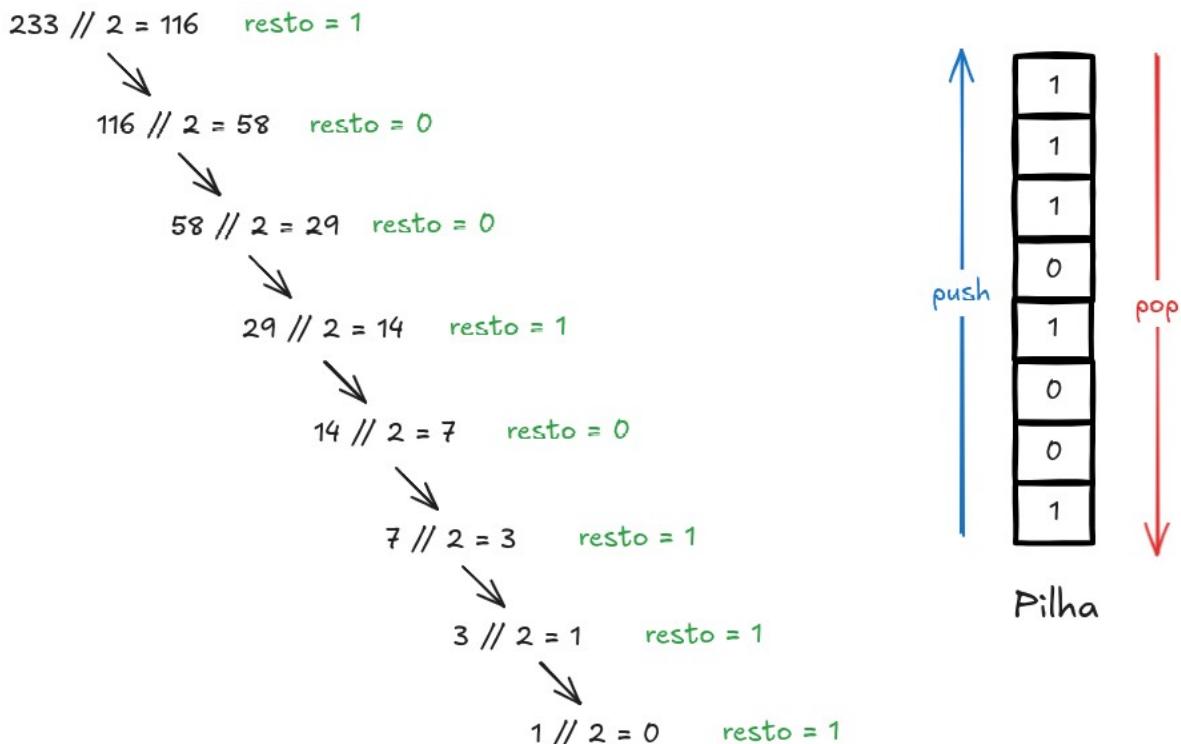
$1 / 2 = 0$ e sobra resto **1** → esse 1 será o nosso último bit (mais a esquerda)

Note que de agora em diante não precisamos continuar com o processo pois

$0 / 2 = 0$ e sobra 0

$0 / 2 = 0$ e sobra 0

ou seja a esquerda do sexto bit teremos apenas zeros, e como no sistema decimal, zeros a esquerda não possuem valor algum. Portanto, 53 em decimal equivale a 110101 em binário. Note que se empilharmos os restos em uma pilha, ao final, basta desempilharmos para termos o número binário na sua forma correta. A figura a seguir ilustra essa ideia.



O algoritmo a seguir mostra a implementação de uma função que converte um número inteiro arbitrário na base 10 (decimal) para a representação binária.

```
# Função que converte um número decimal para binário
decimal_binario(numero) {
    S = Stack()
    while numero > 0 {
        resto = numero % 2
        push(S, resto)
        numero = numero // 2
    }
    binario = ''                      # string vazia
    while not is_empty(S)
        binario = binario + str(s.pop())
    return binario
}
```

Exercício: Escreva uma função para determinar se uma cadeia de caracteres (string) é da forma: xCy onde x e y são cadeias de caracteres compostas por letras ‘A’ ou ‘B’, e y é o inverso de x. Isto é, se x = ‘AABABBA’, então y deve equivaler a ‘ABBABAA’.

Em cada ponto, você só poderá ler o próximo caractere da cadeia (use uma pilha).

Outro exemplo interessante da aplicação de Pilhas é na resolução do seguinte problema: suponha que seja dada uma Pilha S de números inteiros positivos e negativos. Desenvolva um algoritmo que organize a Pilha S de modo que todos os números positivos apareçam acima dos números negativos.

```
# Função organiza uma pilha de números inteiros
organiza_pilha(S) {
    P = Stack()      # Pilha dos positivos
    N = Stack()      # Pilha dos negativos
    # Enquanto houver chaves em S
    while not is_empty(S) {
        numero = pop(S)
        if numero >= 0
            push(P, numero)      # Se maior igual a zero, vai para P
        else
            push(N, numero)      # Senão, vai para N
    }
    # Enquanto não esvaziar N, transfere de N para S
    while not is_empty(N)
        push(S, pop(N))
    # Enquanto não esvaziar P, transfere de P para S
    while not is_empty(P)
        push(S, pop(P))
    return S
}
```

Exercício: Faça um algoritmo que receba como entrada um número inteiro maior ou igual a 10 representado na base 10 e encontre a sua representação numérica em todas as bases de 2 a 9.

Exercício (O problema da celebriade):

Suponha que uma celebridade invada uma festa local. Ele ou ela não conhece ninguém na festa, mas todos conhecem a celebridade. Suponha que temos uma lista das pessoas na festa (incluindo a celebridade) e um meio de determinar se alguma pessoa A conhece alguma pessoa B, onde A e B são pessoas na festa. Como podemos usar uma pilha para determinar eficientemente (ou seja, em O(n)) se uma celebridade está presente e -- quando uma está -- identificar quem é essa celebridade?

Dada uma matriz quadrada M de dimensões n por n, tal que $M[i, j] = 1$ se a i-ésima pessoa conhece a j-ésima pessoa, o objetivo consiste em encontrar a celebridade. Uma celebridade é uma pessoa que é conhecida por todos mas não conhece ninguém. Retorne o índice da celebridade e caso não exista uma, o algoritmo deve retornar -1.

Um exemplo de matriz seria a seguinte:

$$M = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Solução trivial: O algoritmo deve seguir os passos abaixo para resolver o problema.

1. Crie dois arrays indegree e outdegree. O valor de indegree[i] deve armazenar quantas pessoas conhecem a i-esima pessoa e o valor de outdegree[i] deve armazenar quantas pessoas a i-ésima pessoa conhece.
2. Execute um loop aninhado, o loop externo de 0 a n-1 e o loop interno de 0 a n-1.
 - a) Para cada par i, j, se i conhece j, então aumente o outdegree de i e o indegree de j.
 - b) Para cada par i, j, se j conhece i, então aumente o outdegree de j e o indegree de i.
3. Execute um loop de 0 a n-1 e encontre o id para o qual o indegree é n-1 e o outdegree é 0.

```
# Função auxiliar
conhece(i, j, M) {
    return M[i, j]
}

celebridade(M, n) {
    indegree = array(n)
    outdegree = array(n)
    # Laço principal
    for i = 0 to n-1 {
        for j = 0 to n-1 {
            x = conhece(i, j, M)
            indegree[i] += x
            outdegree[j] += x
        }
    }
    # Verific se há uma celebridade
    for i = 0 to n-1 {
        if indegree[i] == n-1 and outdegree[i] == 0
            return i
    }
    return -1
}
```

É fácil notar que essa solução trivial, sem a utilização de uma Pilha, possui complexidade $O(n^2)$.

Solução eficiente: O algoritmo deve seguir os passos abaixo para resolver o problema.

1. Crie uma pilha e insira todos os ids na pilha.
2. Execute um loop enquanto houver mais de 1 elemento na pilha.
 - a) Retire os dois elementos do topo da pilha (represente-os como A e B)
 - b) Se A conhece B, então A não pode ser a celebridade, insira B na pilha. Caso contrário, se A não conhece B, então B não pode ser celebridade, insira A na pilha.
3. Atribua o elemento restante na pilha como a celebridade.
4. Execute um loop de 0 a n-1 e encontre a contagem de pessoas que conhecem a celebridade e o número de pessoas que a celebridade conhece.
5. Se a contagem de pessoas que conhecem a celebridade for n-1 e a contagem de pessoas que a celebridade conhece for 0, então retorne o id da celebridade, caso contrário, retorne -1.

```
# Função auxiliar
conhece(i, j, M) {
    return M[i, j]
}
```

```

celebridade(M, n) {
    S = Stack()
    C = -1
    # Insere todos na pilha
    for i = 0 to n-1
        push(S, i)
    # Encontra a potencial celebridade
    while len(S) > 1 {
        A = pop(S)
        B = pop(S)
        if conhece(A, B, M)
            push(S, B)
        else
            push(S, A)
    }
    # Celebridade potencial é quem sobra na pilha
    C = pop(S)
    # Verifica se C é de fato celebridade
    for i = 0 to n-1 {
        # Se alguma pessoa não conhece C ou C conhece alguém, retorna -1
        if i ≠ C and (conhece(C, i, M) or not conhece(i, C, M))
            return -1
    }
    return C
}

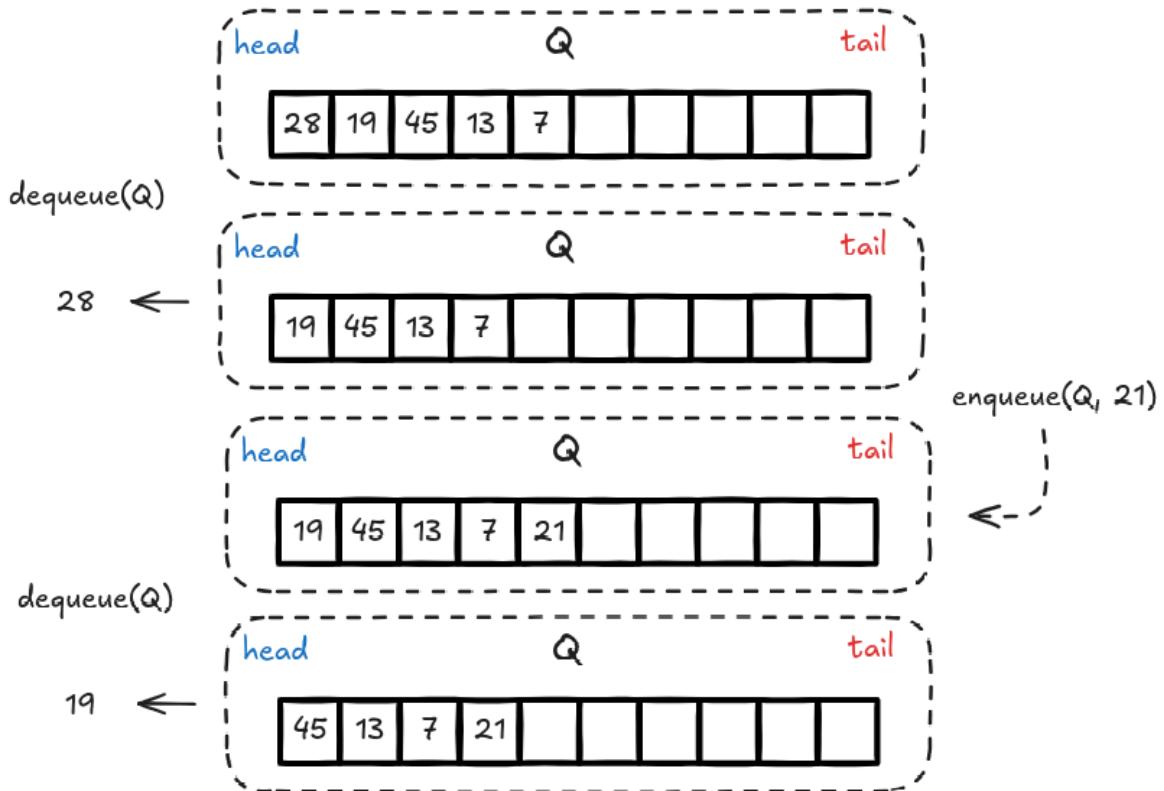
```

Note que as primitivas push e pop da Pilha possuem complexidade O(1), o que faz com que o algoritmo anterior possua complexidade O(n), ou seja, é mais eficiente do que a solução trivial em termos computacionais.

Filas (FIFO)

Uma fila (queue) é uma estrutura de dados linear em que a inserção de elementos é realizada em uma extremidade (final) e a remoção é realizada na outra extremidade (início). Assim como em uma fila de pessoas no mundo real, o primeiro a entrar será o primeiro a sair, o que define o princípio de ordenação FIFO, ou do inglês, *First In First Out*. Além disso, filas devem ser restritivas no sentido de que um elemento não pode passar na frente de seu antecessor.

Na ciência da computação há diversos exemplos de aplicações que utilizam filas para gerenciar a ordem de acesso aos recursos. Por exemplo, um servidor de impressão localizado em um laboratório de pesquisa em um departamento da universidade precisar lidar com o gerenciamento da ordem das impressões. Para isso, é usual o software da impressora criar uma fila de impressões. Assim, múltiplas requisições são tratadas sequencialmente de acordo com a ordem em que chegam. Para que possamos implementar um TAD Fila, devemos ter em mente quais suas variáveis internas e quais as operações que podemos aplicar sobre os seus elementos. A listagem a seguir mostra como podemos construir uma fila a partir de uma lista inicialmente vazia. As operações são bastante parecidas com as operações de uma pilha.



Operação	Conteúdo	Retorno	Descrição
is_empty(Q)	[]		Verifica se fila está vazia
enqueue(Q, 4)	[4]		Insere elemento no final
enqueue(Q, 1)	[1, 4]		Insere elemento no final
enqueue(Q, 7)	[7, 1, 4]		Insere elemento no final
size(Q)	[7, 1, 4]	3	Número de elementos da fila
is_empty(Q)	[7, 1, 4]	False	Verifica se fila está vazia
enqueue(Q, 9)	[9, 7, 1, 4]		Insere elemento no final
dequeue(Q)	[9, 7, 1]	4	Remove elemento do início
dequeue(Q)	[9, 7]	1	Remove elemento do início
size(Q)	[9, 7]	2	Número de elementos da fila

Para projetar uma fila, iremos utilizar a mesma estratégia utilizada com a pilha: como atributo teremos um array de inteiros, que inicia vazio. Definiremos os métodos enqueue() e dequeue() para inserção de elementos no final e remoção de elementos no início, bem como, size() e is_empty(), obter o número de elementos da fila e verificar se a fila está vazia.

```
# Inicializa fila vazia
init(Q)
# Verifica se fila está vazia
is_empty(Q)
# Insere elemento no final da fila (equivalente ao push)
enqueue(Q, key)
# Remove elemento do início da fila (equivalente ao pop)
dequeue(Q)
```

Uma fila Q pode ser implementada de maneira estática como um array de tamanho n, juntamente com 2 marcadores: head, para a remoção no início, e tail, para inserção no final.

```

TAD Queue
    int array[1..n] keys (n é o tamanho da fila)
    int head           (para remoção)
    int tail           (para inserção)
    int size           (número de elementos presentes na fila)

init(Q) {
    Q.head = 0          (ambas as extremidades são livres para mover)
    Q.tail = 0
    Q.size = 0
}

is_empty(Q) {
    if Q.size = 0
        return True
    else
        return False
}

enqueue(Q, key) {
    if Q.size == n
        error('overflow')
    else {
        if Q.tail == n      # para tornar a fila circular
            Q.tail = 1
        else
            Q.tail += 1
    }
    Q.keys[Q.tail] = key
    Q.size += 1
}

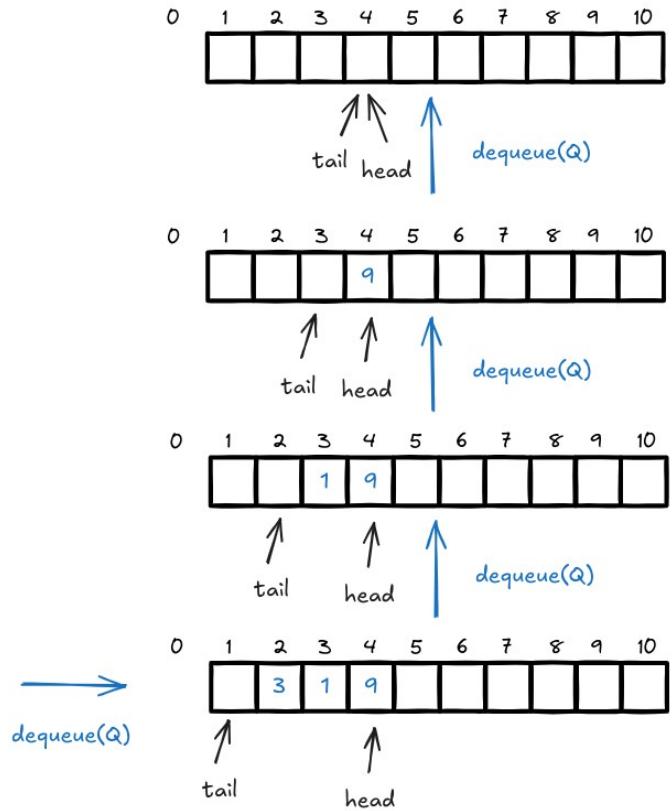
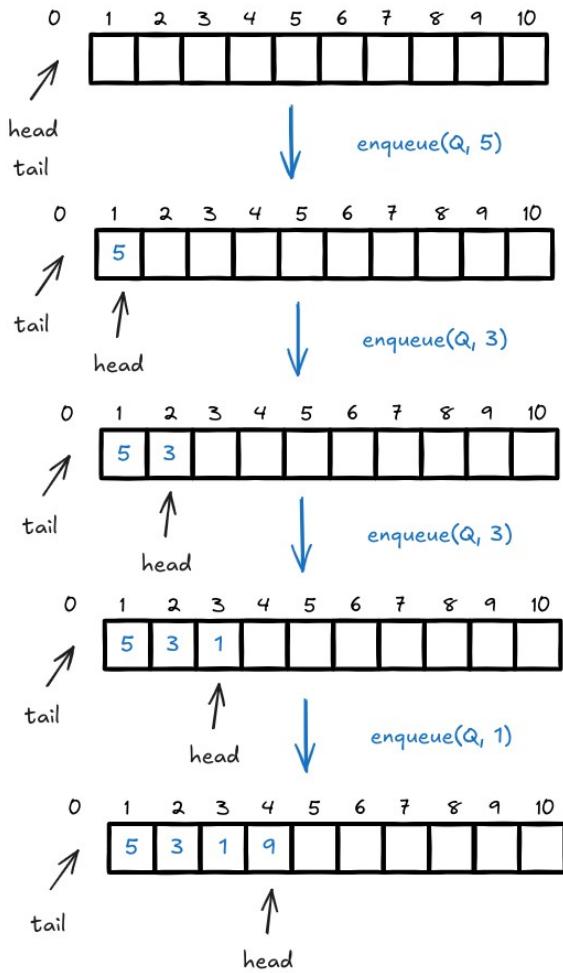
dequeue(Q) {
    if is_empty(Q)
        error('underflow')
    else {
        if Q.head == n      # para tornar a fila circular
            Q.head = 1
        else
            Q.head += 1
    }
    key = Q.keys[Q.head]
    Q.size -= 1
    return key
}

```

Note que tanto a inserção quanto a remoção em uma fila Q possuem complexidade O(1).

A figura a seguir ilustra algumas operações de inserção e remoção em uma fila Q.

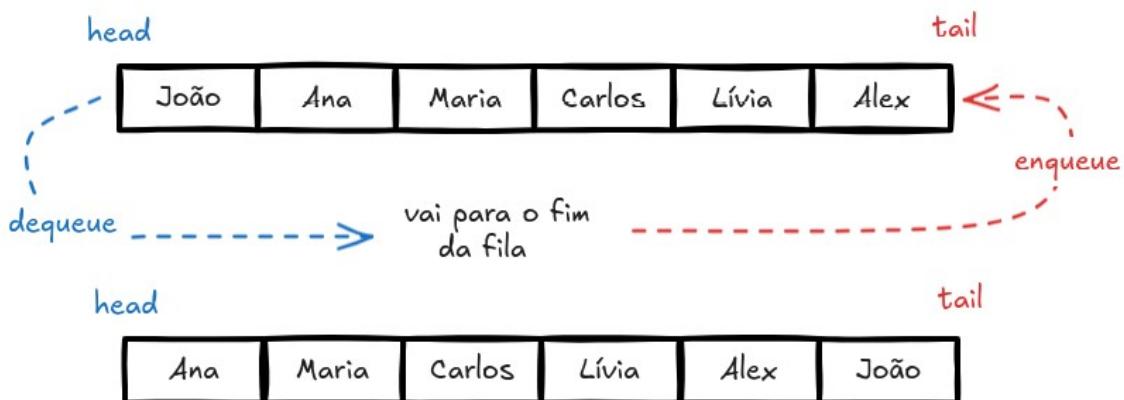
FILA Q



Para ilustrar algumas aplicações que utilizam estruturas de dados do tipo Fila. A primeira delas é uma simples simulação do jogo infantil Batata Quente. Neste jogo, as crianças formam um círculo e passam um item qualquer (batata) cada um para o seu vizinho da frente o mais rápido possível. Em um certo momento do jogo, essa ação é interrompida (queimou) e a criança que estiver com o item (batata) na mão é excluída da roda. O jogo então prossegue até que reste apenas uma única criança, que é a vencedora.

Para simular um círculo (roda), utilizaremos uma fila da seguinte maneira: a criança que está com a batata na mão será sempre a que estiver no início da fila. Após passar a batata, a simulação deve instantaneamente remover e inserir a criança, colocando-a novamente no final da fila. Ela então vai esperar até que todas as outras assumam o início da fila, antes de assumir essa posição novamente. Após um número pré estabelecido MAX de operações enqueue/dequeue, a criança que ocupar o início da fila será removida e outro ciclo da brincadeira é realizado. O processo continua até que a fila tenha tamanho um. A figura a seguir ilustra o processo.

Este problema parece simples e bobo, porém é a base para a solução de diversos problemas na computação, como o escalonamento de processos via o algoritmo Round-Robin em sistemas operacionais e métodos baseados em passagem de token em sistemas distribuídos. Nesses problemas, deseja-se que apenas o processo que contém o token seja executado, enquanto que os demais devem aguardar a sua vez.



Um algoritmo para a simulação desse jogo é apresentado a seguir.

```
# Simula o jogo batata_quente
# nomes = array com nomes de K pessoas (K <= n para caber na fila)
batata_quente(nomes, K, MAX):
    # Cria fila para simular roda
    Q = Queue()
    # Coloca os nomes em cada posição
    for i = 1 to K
        enqueue(Q, nomes[i])
    # Inicia a lógica do jogo
    while Q.size > 1 {
        # Para simular MAX passagens da batata
        for i = 1 to MAX
            # Remove o primeiro e coloca no final
            enqueue(Q, dequeue(Q))
        # Quem parar no ínicio da fila, está com batata (eliminado)
        dequeue(Q)
    }
    # Após K-1 rodadas, retorna a fila com o vencedor
    vencedor = dequeue(Q)
    return vencedor
}
```

Exercício: Modifique a simulação do jogo batata quente de modo a permitir que o número passagens MAX seja um número aleatório. Assim, em cada rodada, teremos um valor de MAX diferente, o que é menos previsível.

Exercício: Para um dado número inteiro $n > 1$, o menor inteiro $d > 1$ que divide n é chamado de fator primo. É possível determinar a fatoração prima de n achando-se o fator primo d e substituindo n pelo quociente n / d , repetindo essa operação até que n seja igual a 1. Utilizando uma das estruturas de dados lineares para auxiliá-lo na manipulação de dados, implemente uma função que compute a fatoração prima de um número imprimindo os seus fatores em ordem decrescente. Por exemplo, para $n=3960$, deverá ser impresso $11 * 5 * 3 * 3 * 2 * 2 * 2$. Justifique a escolha do TAD utilizado.

Problema: Usando uma pilha P inicialmente vazia, implemente um método para inverter uma fila Q com n elementos utilizando apenas os métodos `is_empty()`, `push()`, `pop()`, `enqueue()` e `dequeue()`.

```
# função para inverter uma fila usando uma pilha como estrutura auxiliar
inverte_fila(Q) {
    # Cria pilha vazia
```

```

S = Stack()
# Enquanto tiver elementos na fila
while not is_empty(Q)
    push(S, dequeue(Q))
# Enquanto tiver elementos na pilha
while not is_empty(S)
    enqueue(Q, pop(S))
return Q
}

```

Problema: Dado um número inteiro positivo n, faça um algoritmo que calcule todos os números de 1 até n em binário.

```

gera_binarios(n) {
    for i = 1 to n {
        s = ''          # string vazia
        temp = i
        # Enquanto tiver elementos na fila
        while temp > 0 {
            if temp & 1 == 1      # & denota o operador E lógico
                s = s + '1'
            else
                s = s + '0'
            temp = temp >> 1    # shift right uma vez
        }
        print(s)
    }
}

```

A análise do algoritmo acima revela que:

1. O número de execuções do FOR mais externo é n.
2. Como a operação de shift right é realizada nbits vezes, onde nbites é o número de bits necessários para codificar o inteiro i, temos que $nbites \approx \log_2 n$, uma vez que o número de divisões por 2 necessárias para representar um número em binário é o logaritmo de n na base 2.

Sendo assim, a complexidade total do algoritmo é $O(n \log n)$.

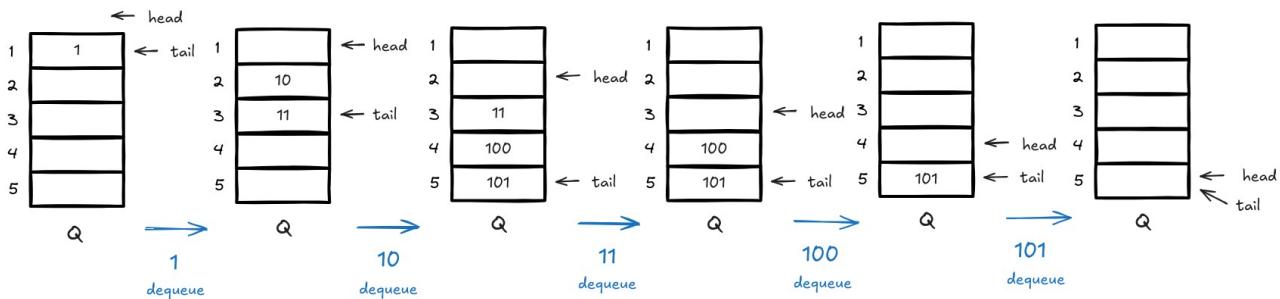
Veremos agora uma solução mais eficiente para esse problema.

```

gera_binarios(n) {
    Q = Queue()
    enqueue(Q, '1')  # insere o primeiro número na fila Q
    while n > 0 {    # Loop principal
        n = n - 1
        s = dequeue(Q)    # Remove primeiro da fila
        print(s)
        enqueue(Q, s+'0') # Insere no final da fila
        enqueue(Q, s+'1') # Insere no final da fila
    }
}

```

Vamos simular o passo a passo do algoritmo para n = 5.



A complexidade do algoritmo com a Fila pode ser calculada de maneira simples e direta. Sabemos que as funções enqueue e dequeue possuem complexidade O(1). Sendo assim, como estão dentro de um loop de tamanho n, a complexidade total será O(n). Portanto, esse algoritmo é mais eficiente do que o anterior para resolver o problema.

Deque: Double-Ended Queue (Fila de duas extremidades)

Deque (pronuncia-se deck para diferenciar da operação dequeue), ou Double-Ended Queue, nome que traduziremos como Dupla Fila, é uma estrutura de dados linear bastante similar a uma fila tradicional, porém com dois inícios e dois finais. O que a torna diferente da fila é justamente a possibilidade de inserir e remover elementos tanto do início quanto do fim da lista. Neste sentido, esse TAD híbrido prove todas as funcionalidades de filas e pilhas em uma única estrutura de dados. Apenas para deixar claro, denotaremos por start a parte da frente da fila (esquerda) e por end a parte de trás da fila (direita), ou seja:

start → [1, 2, 3, 4, 5] ← end

Operação	Conteúdo	Retorno	Descrição
d.is_empty()	[]	True	Verifica se deque está vazio
d.add_start(4)	[4]		Insere elemento na esquerda
d.add_start(7)	[7, 4]		Insere elemento na esquerda
d.add_end(2)	[7, 4, 2]		Insere elemento na direita
d.add_end(5)	[7, 4, 2, 5]		Insere elemento na direita
d.size()	[7, 4, 2, 5]	4	Retorna número de elementos
d.is_empty()	[7, 4, 2, 5]	False	Verifica se deque está vazio
d.add_start(8)	[8, 7, 4, 2, 5]		Insere elemento na esquerda
d.remove_end()	[8, 7, 4, 2]	5	Remove elemento da direita
d.remove_start()	[7, 4, 2]	8	Remove elemento da esquerda
d.size()	[7, 4, 2]	3	Retorna número de elementos

A seguir apresentamos uma TAD para a estrutura de dados Deque.

TAD Deque

```

int array[1..n] keys
int start
int end
int size

init(D) {
    D.start = 0
    D.end = 0
    D.size = 0
}

```

(n é o tamanho do deque)
(extremidade da esquerda)
(extremidade da direita)

```

is_full(D) {
    if D.size == n
        return True
    else
        return False
}

// Adiciona no início
add_start(D, key) {
    if is_full(D)
        error('overflow')
    else {
        if is_empty(D) {
            start = 1
            end = 1
        }
        else {
            if D.start == 1
                D.start = n
            else
                D.start -= 1
        }
        D.keys[D.start] = key
        D.size += 1
    }
}

// Remove no início
remove_start(D) {
    if is_empty(D)
        error('underflow')
    else {
        key = D.keys[D.start]
        D.keys[D.start] = NULL
        if D.start == D.end {
            D.start = 0
            D.end = 0
        }
        else {
            if D.start == n
                D.start = 1
            else
                D.start += 1
        }
        D.size -= 1
        return key
    }
}

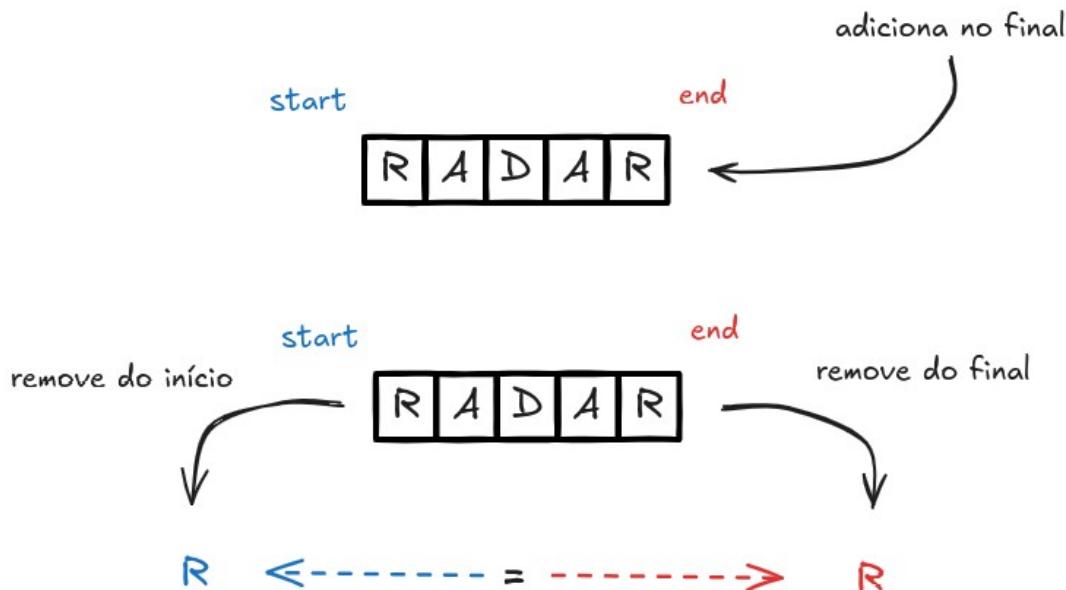
is_empty(D) {
    if D.size == 0
        return True
    else
        return False
}

// Adiciona no final
add_end(D, key) {
    if is_full(D)
        error('overflow')
    else {
        if is_empty(D) {
            start = 1
            end = 1
        }
        else {
            if D.end == n
                D.end = 1
            else
                D.end += 1
        }
        D.keys[D.end] = key
        D.size += 1
    }
}

// Remove no final
remove_end(D) {
    if is_empty(D)
        error('underflow')
    else {
        key = D.keys[D.end]
        D.keys[D.end] = NULL
        if D.start == D.end {
            D.start = 0
            D.end = 0
        }
        else {
            if D.end == 1
                D.end = n
            else
                D.end -= 1
        }
        D.size -= 1
        return key
    }
}

```

Um aspecto interessante com essa implementação que inserção e remoção de elementos no Deque possuem complexidade O(1). Um exemplo de aplicação de uma estrutura linear do tipo Deque é no problema de verificar se uma dada palavra é palíndroma, ou seja, se ela é igual a sua versão reversa. Um exemplo de palavra palíndroma é RADAR. A figura a seguir ilustra como uma estrutura Deque pode ser utilizada nesse tipo de problema.



A ideia consiste em processar cada caractere da string de entrada da esquerda para a direita adicionando cada caractere na esquerda do Deque. No próximo passo, iremos remover os caracteres das duas extremidades (esquerda e direita), comparando-as. Se os caracteres forem iguais repetimos o processo ate que não haja mais caracteres no Deque, se a string tiver um número par de caracteres, ou sobre apenas 1 caractere, se a string tiver um número ímpar de caracteres. A função a seguir implementa a solução na forma de um algoritmo.

```
# Função que verifica que palavra é palíndroma
palindroma(palavra) {
    # Cria Deque
    D = Deque()
    K = len(palavra)
    # Percorre string de entrada
    for i = 1 to K
        add_rear(D, palavra[i])
    # Enquanto houver mais de 1 letra
    while D.size > 1 {
        esquerda = remove_rear(D)
        direita = remove_front(D)
        if esquerda != direita
            return False
    }
    return True
}
```

Problema: Sliding Window Maximum

<https://www.geeksforgeeks.org/sliding-window-maximum-maximum-of-all-subarrays-of-size-k/>

Dado um array de tamanho n e um valor de k, encontrar o máximo elemento de cada subarray contíguo de tamanho k. Em outras palavras, esse problema consiste em encontrar o máximo elemento em uma janela deslizante).

Entrada: L = [1, 2, 3, 1, 4, 5], K = 3

Saída: 3 3 4 5

Explicação: Máximo de 1, 2, 3 é 3
 Máximo de 2, 3, 1 é 3
 Máximo de 3, 1, 4 é 4
 Máximo de 1, 4, 5 é 5

O primeiro algoritmo apresentado representa a solução trivial.

```
# Sliding Window Maximum v.1
sliding_window_maximum(L, n, k) {
    max = 0
    for i = 1 to n - k {
        max = L[i]
        for j = 1 to k {
            if L[i + j] > max
                max = L[i + j]
        }
        print(max)
    }
}
```

É fácil notar que o loop mais externo (i) executa $n - k$ vezes e o loop mais interno (j) executa k vezes, o que nos leva a uma complexidade $O(nk)$. Podemos melhorar a eficiência do algoritmo a partir da utilização de um Deque.

A ideia do algoritmo consiste em: crie um Deque, D de capacidade k, que armazene apenas elementos úteis da janela atual de k elementos. Um elemento é útil se estiver na janela atual e for maior que todos os outros elementos no lado direito dele na janela atual. Processe todos os elementos do array um por um e mantenha D para conter elementos úteis da janela atual e esses elementos úteis são mantidos em ordem classificada. O elemento na frente do deque D é o maior e o elemento na parte traseira/posterior do deque é o menor da janela atual.

O algoritmo é baseado nos seguintes passos:

1. Crie um deque para armazenar k elementos.
2. Execute um loop e insira os primeiros k elementos no deque. Antes de inserir o elemento, verifique se o elemento no final da fila é menor que o elemento atual, se for, remova o elemento do final do deque até que todos os elementos restantes no deque sejam maiores que o elemento atual. Então insira o elemento atual, no final do deque.
3. Agora, execute um loop de k até o final do array.
 - a) Imprima o elemento da frente do deque.
 - b) Remova o elemento da frente da fila se eles estiverem fora da janela atual.
 - c) Insira o próximo elemento no deque. Antes de inserir o elemento, verifique se o elemento no final da fila é menor que o elemento atual, se for, remova o elemento do final do deque até que todos os elementos restantes no deque sejam maiores que o elemento atual. Então insira o elemento atual, no final do deque.
 - d) Imprima o elemento máximo da última janela.

```
# Sliding Window Maximum v.2
sliding_window_maximum(L, n, k) {
    D = Deque()
    # Analisa primeira janela (primeiros k elementos de L)
    for i = 0 to k-1 {
```

```

# Elemento útil
# Enquanto L[i] maior que último elemento de D, remove último
while not is_empty(D) and L[i] >= L[D.end]
    remove_rear(D)
# Adiciona no fim
add_rear(D, i)
}
# Analisa o restante do array L
for i = k to n-1 {
    print(L[D.start])
    # Remove elementos que estão fora da janela
    while not is_empty(D) and D.start <= i - k
        remove_front(D)
    # Elemento útil
    # Enquanto L[i] maior que último elemento de D, remove último
    while not is_empty(D) and L[i] >= L[D.end]
        remove_rear(D)
    # Adiciona no fim
    add_rear(D, i)
}
print(L[D.start])
}

```

Análise da complexidade

Na análise da complexidade do algoritmo, o primeiro passo consiste em notar que as funções add_rear, remove_rear e remove_front são O(1).

1. Pode-se observar que cada elemento do array é adicionado e removido no máximo uma vez do deque D.
2. Um mesmo elemento não pode entrar duas vezes em D.
3. O primeiro FOR itera k vezes e o segundo FOR itera $n - k$ vezes, e como ambos são sequenciais, o total de iterações é n .
4. Dessa forma, há um total de $2n$ operações de custo O(1), o que resulta em complexidade O(n).

Sendo assim, o custo é menor que O(kn) da implementação trivial.

A seguir, veremos uma ilustração do funcionamento passo a passo do algoritmo em uma caso particular. A ideia é mostrar o que ocorre com o Deque a cada iteração.

Exercício: Aplique o algoritmo sliding_window_maximum no seguinte array

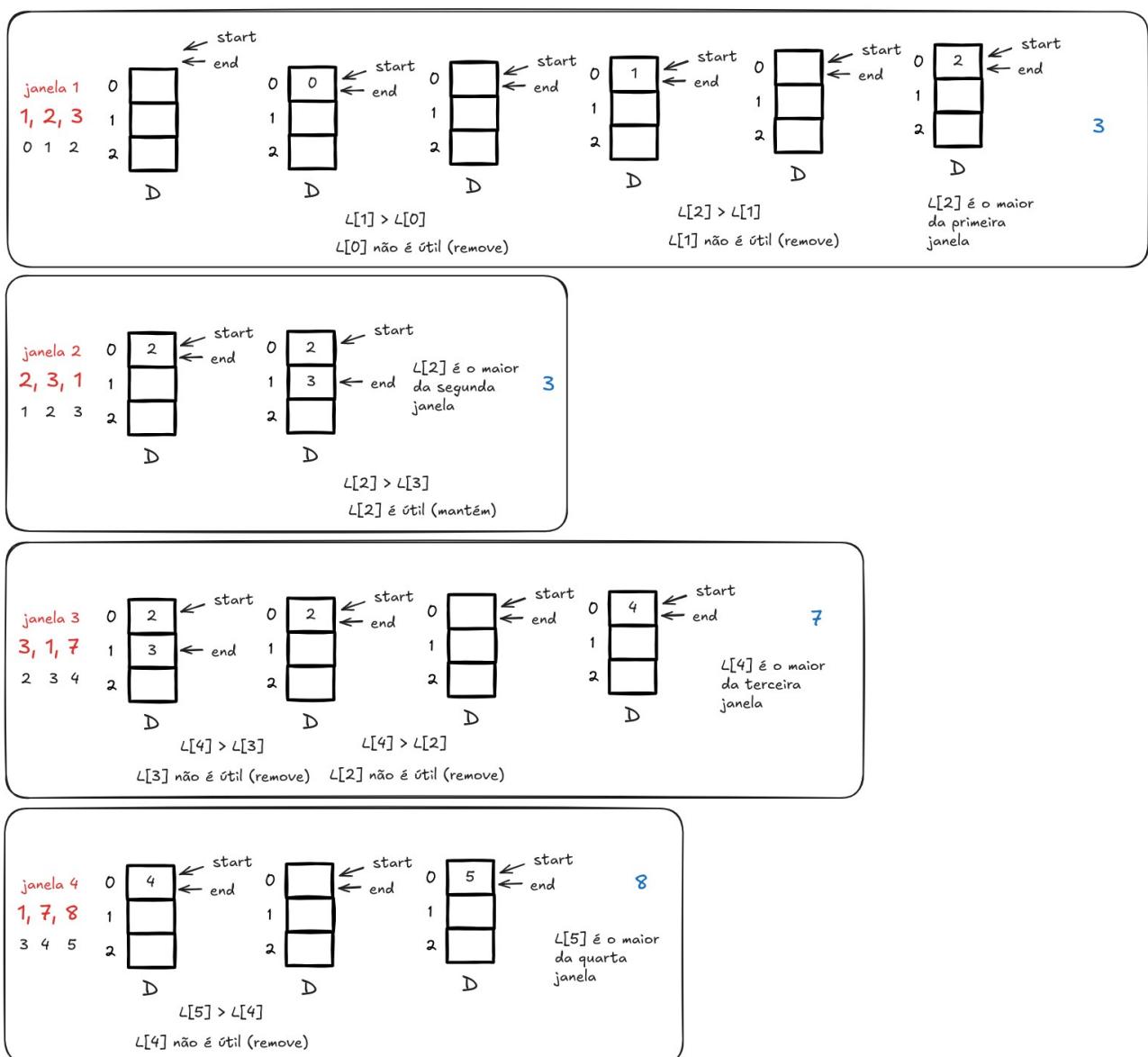
$L = [1, 2, 3, 1, 7, 8]$

considerando o tamanho da janela como $k = 3$.

$$L = [1, 2, 3, 1, 7, 8]$$

$$K = 3$$

Ideia: sempre manter o índice do maior no início do Deque



Outros problemas interessantes podem ser encontrados em:

<https://www.geeksforgeeks.org/top-50-problems-on-queue-data-structure-asked-in-sde-interviews/>

Fila de prioridades (Priority Queue)

Uma fila de prioridades é uma extensão da fila tradicional em que, para cada elemento inserido, uma prioridade p deve ser associada. Por convenção, inteiros positivos são utilizados para representar a prioridade, sendo que quanto maior o inteiro, maior a prioridade (ou seja, $p = 2$ tem prioridade sobre $p = 1$). A principal diferença em relação a fila tradicional não é o método `enqueue()`, em que todo elemento continua a ser inserido no final da fila, mas sim o método `dequeue()`, pois a remoção não é feita no início da fila e sim descobrindo o elemento que possui a maior prioridade.

```

TAD Node
    int key
    int priority

init(PQ) {
    PQ.tail = 0
    PQ.size = 0
}

# Adiciona chave com prioridade p no final da fila
enqueue(PQ, key, p) {
    if PQ.size == n
        error('overflow')
    else {
        PQ.tail += 1
        PQ.data[PQ.tail].key = key
        PQ.data[PQ.tail].priority = p
        PQ.size += 1
    }
}

# Encontra elemento de maior prioridade
findMax(PQ) {
    index = 1
    max = PQ.data[index].p
    for i = 1 to PQ.tail {
        if PQ.data[i].p > max
            max = PQ.data[i].p
            index = i
    }
    return index
}

# Remove chave de maior prioridade
dequeue(PQ) {
    if PQ.size == 0
        error('underflow')
    else {
        # encontra a posição do elemento de menor prioridade
        pos = findMax(PQ)
        key = PQ.data[pos].key
        # precisa reallocar todos elementos de pos até tail
        for i = pos to PQ.tail - 1 {
            PQ.data[i].key = PQ.data[i+1].key
            PQ.data[i].p = PQ.data[i+1].p
        }
        PQ.tail -= 1
        PQ.size -= 1
    }
    return key
}

```

Note que nesta implementação, a complexidade da função dequeue usada na remoção de um elemento da fila de prioridades, é $O(n)$. Veremos mais adiante que com a utilização de uma estrutura do tipo heap, podemos reduzir essa complexidade para $O(\log n)$.

Problema: Disk tower

Objetivo: Sua tarefa é construir uma torre em n dias seguindo estas condições:

- A cada dia você recebe um disco de tamanho distinto de 1 a n.
- O disco de maior tamanho deve ser colocado na parte inferior da torre.
- O disco de menor tamanho deve ser colocado no topo da torre.

A ordem em que a torre deve ser construída é a seguinte: Você não pode colocar um novo disco no topo da torre até que todos os discos maiores que lhe forem dados sejam colocados.

Imprima n linhas denotando os tamanhos de disco que podem ser colocados na torre no i-ésimo dia.

Formato de entrada

Primeira linha: n denotando o número total de discos que são dados a você nos n dias subsequentes.
Segunda linha: n inteiros em que o i-ésimo inteiro denota o tamanho dos discos que são dados a você no i-ésimo dia.

Observação: todos os tamanhos de disco são inteiros distintos no intervalo de 1 a n.

Formato de saída

Imprima n linhas: na i-ésima linha, imprima o tamanho dos discos que podem ser colocados no topo da torre em ordem decrescente dos tamanhos dos discos.

Se no i-ésimo dia nenhum disco puder ser colocado, deixe essa linha vazia.

Restrições: $1 < n < 10^6$
 $1 \leq \text{tamanho_do_disco} \leq n$

Entrada:	Saída
5	-
4 5 1 2 3	5 4
	-
	-
	3 2 1

Explicação

- No primeiro dia, o disco de tamanho 4 é dado. Mas você não pode colocar o disco na parte inferior da torre, pois ainda resta um disco de tamanho 5 a receber.
- No segundo dia, o disco de tamanho 5 será dado, então agora os discos de tamanhos 5 e 4 podem ser colocados na torre.
- No terceiro e quarto dia, os discos não podem ser colocados na torre, pois o disco de 3 ainda precisa ser dado. Portanto, essas linhas estão vazias.
- No quinto dia, todos os discos de tamanhos 3, 2 e 1 podem ser colocados no topo da torre.

A seguir, veremos uma solução que utiliza uma estrutura de dados do tipo fila de prioridades para resolver esse problema. A ideia consiste em ter uma fila de prioridades em que os campos key e priority são idênticos.

```

# Verifica o último elemento inserido
peek(PQ) {
    return PQ.data[PQ.tail].key
}

disk_tower(L, n) {
    max = n
    PQ = PriorityQueue()
    for i = 0 to n-1 {
        enqueue(PQ, L[i])
        while peek(PQ) == max {
            disk = dequeue(PQ)
            print(disk, '')      # imprime na mesma linha
            max -= 1
        }
        print('-\n')
    }
}

```

Como cada elemento entra na fila e sai da fila exatamente uma única vez temos:

=> o custo da operação enqueue é O(1)

=> o custo da operação dequeue é O(n)

Assim, o custo total é:

$$T(n) = nO(1) + nO(n) = O(n^2)$$

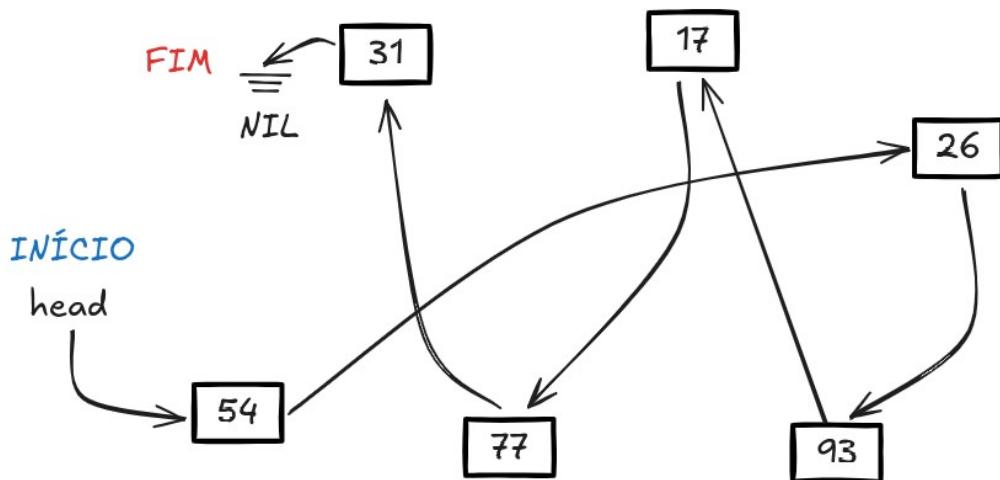
Veremos mais adiante que se implementarmos a fila de prioridades com um heap binário (min-heap ou max-heap), esse custo pode ser reduzido para $O(\log n)$.

Nesta aula, vimos as principais estruturas de dados estáticas: Pilha, Fila, Deque e Fila de prioridades. Na próxima aula estudaremos outra estrutura linear muito importante para a computação, as listas encadeadas. Listas encadeadas diferem de vetores tradicionais em um aspecto primordial: enquanto em um vetor tradicional os elementos subsequentes são armazenados de maneira contígua na memória, em uma lista encadeada, cada nó da estrutura é armazenado independente dos demais e a conexão entre os nós é realizada por um encadeamento lógico.

"You are a piece of the puzzle of someone else's life. You may never know where you fit, but others will fill the holes in their lives with pieces of you."
-- Bonnie Arbon

Estruturas de dados dinâmicas

Em programação de computadores, é possível utilizar estruturas de dados de maneira dinâmica, ou seja, projetar estruturas de dados que crescem ou diminuem de tamanho em tempo de execução, seja pela inserção ou pela remoção de elementos. Para permitir esse tipo de flexibilidade, diversas linguagens de programação oferecem mecanismos para alocação dinâmica de memória. Nesse contexto, as listas encadeadas são estruturas dinâmicas que utilizam um esquema de encadeamentos lógicos sequenciais de modo a permitir que elementos vizinhos não precisem ocupar posições contíguas da memória, como mostrado na figura a seguir.

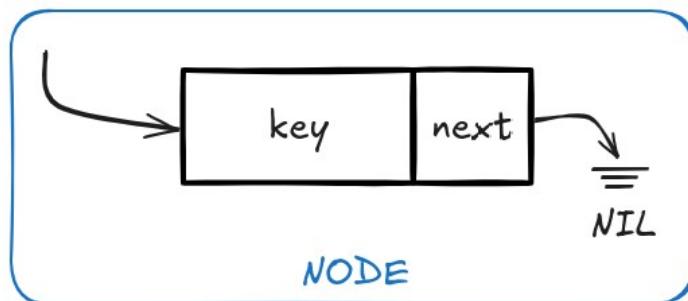


Primeiramente, antes de definirmos uma lista encadeada, devemos definir o bloco básico de construção de uma lista: o nó. Cada nó de uma lista encadeada deve conter duas informações: o dado propriamente dito e uma referência para o próximo nó da lista (para quem esse nó aponta). Podemos definir um TAD Node, que representa um nó da lista, como segue:

TAD Node

```
int key          # armazena a chave  
Node next       # referência para um outro nó (encadeamento lógico)
```

Internamente, ao criarmos um nó, temos uma representação típica como a figura a seguir.



Ao contrário das estruturas estáticas, as estruturas dinâmicas não possuem limitação de tamanho, no sentido que é possível inserir elementos indefinidamente até que a memória do computador seja totalmente ocupada. A primeira e mais básica estrutura de dados linear e dinâmica são as listas encadeadas não ordenadas.

Listas encadeadas

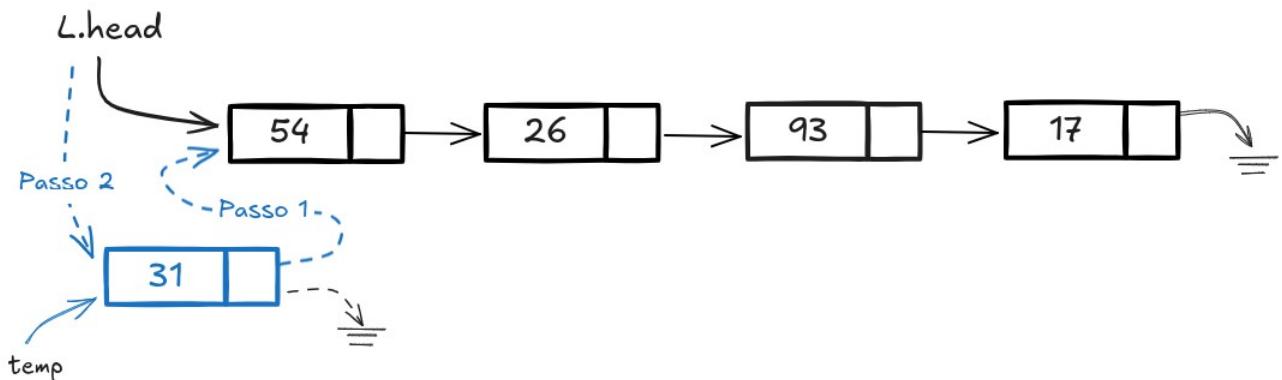
Em uma lista ordenada, a principal característica é que a inserção de um novo nó é feita sempre no início ou no final da lista, ou seja, os elementos do conjunto não encontram-se ordenados. Iniciaremos apresentando como criar uma lista encadeada não ordenada vazia. Adotaremos o seguinte construtor, em que head (cabeça) é uma referência para o primeiro nó da lista:

```
TAD Linked_List  
    Node head  
    int size
```

Durante a criação da lista encadeada L, devemos fazer a cabeça da lista referenciar NIL.

```
init(L) {  
    L.head = NIL  
    L.size = 0  
}
```

A primitiva mais básica de um TAD lista encadeada é a responsável por adicionar um elemento no início da lista. A lógica dessa operação consiste em apontar o novo nó para a cabeça da lista (head) e fazer a cabeça da lista apontar para esse novo nó recém inserido (pois ele será o primeiro elemento da lista). O diagrama a seguir ilustra o resultado da execução dos comandos a seguir:

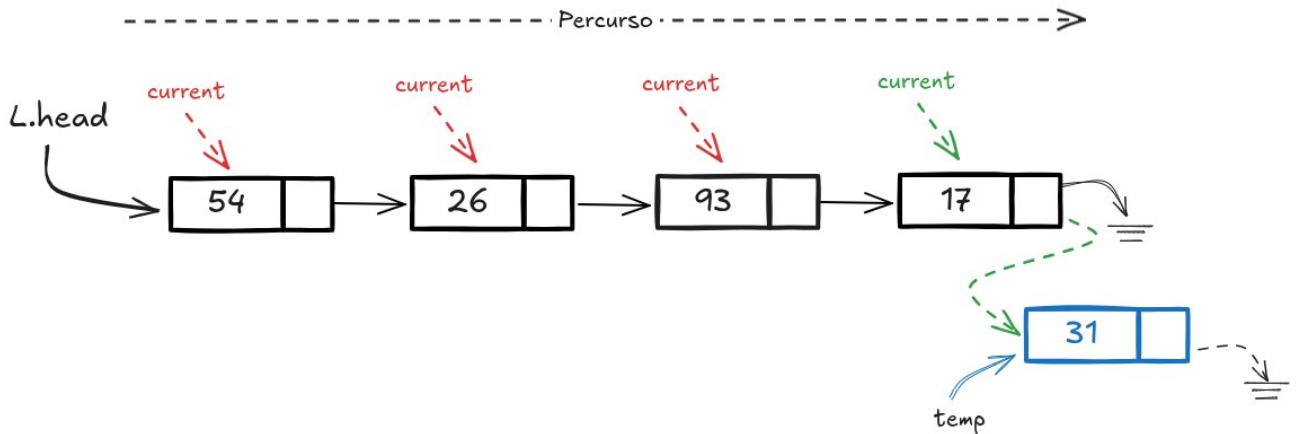


O algoritmo a seguir mostra como podemos implementar essa funcionalidade em uma lista encadeada L inicialmente vazia.

```
add_head(L, key) {  
    # Cria novo nó  
    temp = Node(key)  
    # Aponta novo nó para cabeça da lista  
    temp.next = L.head  
    # Atualiza a cabeça da lista  
    L.head = temp  
    L.size += 1  
}
```

Note que a inserção no início de uma lista encadeada possui complexidade O(1). De modo análogo, podemos realizar a inserção no final da lista. Para isso, devemos criar um novo nó e posicionar uma referência no último elemento da lista. Para isso, é preciso apontar temp para a cabeça da lista e percorrer a lista até atingir um nó tal que o próximo elemento seja definido como NIL. Isso significa que estamos no último elemento da lista. Ao percorrermos uma lista encadeada, devemos iniciar na

cabeça da lista (head) e a cada iteração fazer a referência apontar para o seu sucessor. A figura a seguir ilustra esse processo.



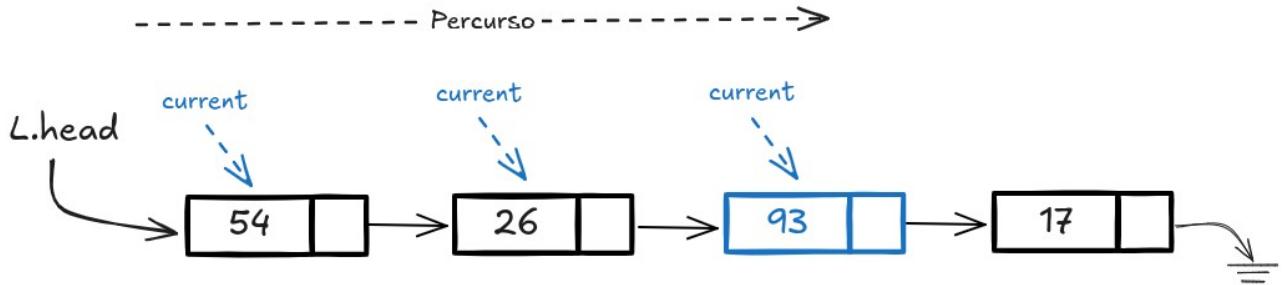
O algoritmo a seguir implementa essa funcionalidade.

```
add_tail(L, key) {
    # Cria novo nó
    tail = Node(key)
    if L.head == NIL
        L.head = tail
    else {
        # Usa referência temporária para percorrer lista (cabeça)
        temp = L.head
        # Percorre a lista até o último elemento
        while temp.next != NIL:
            temp = temp.next
        # Aponta tail (ultimo elemento) para novo nó
        temp.next = tail
    }
    tail.next = NIL
    L.size += 1
}
```

Note que a inserção no final da lista (tail) tem complexidade $O(n)$, uma vez que é preciso percorrer todos os seus nós para encontrar a posição correta. Precisamos ainda implementar uma função para retornar quantos elementos existem na lista encadeada. Note que é possível fazer isso em $O(1)$.

```
length(L) {
    return L.size
}
```

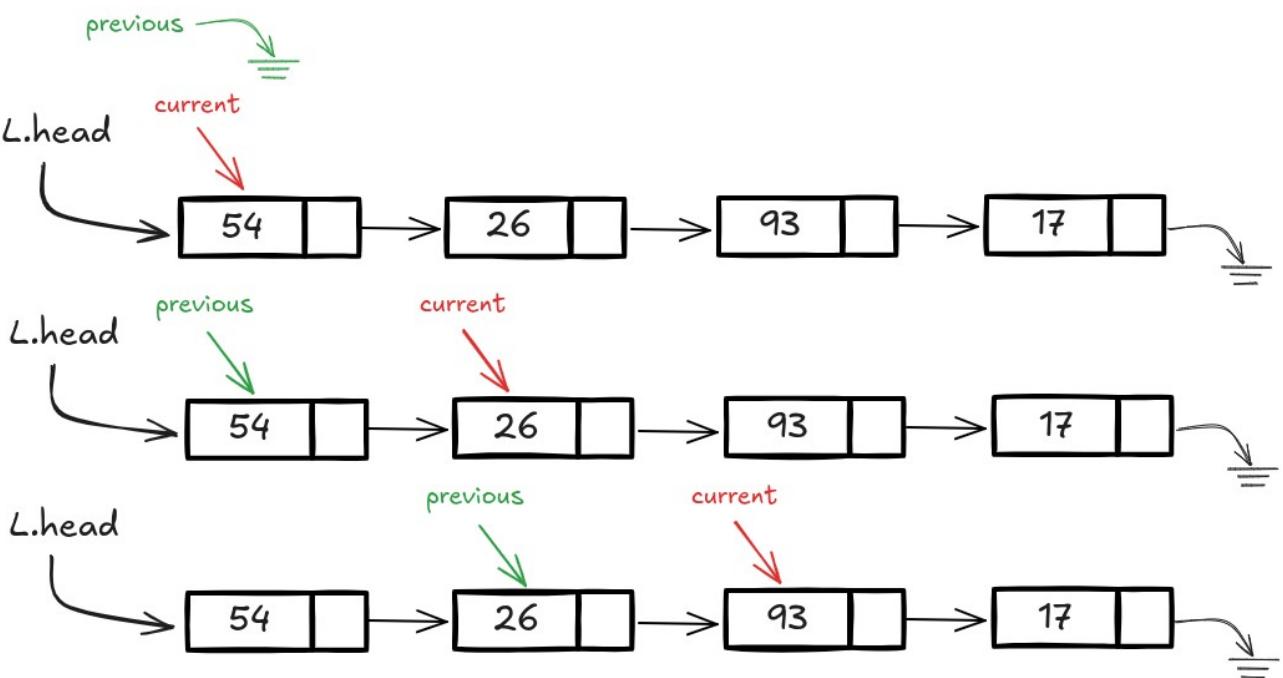
Note que, ao adotar a estratégia de incrementar ou decrementar o número de elementos da lista encadeada após cada inserção ou remoção, é possível saber o número de elementos em L com complexidade $O(1)$. Outra funcionalidade importante consiste em buscar um elemento na lista encadeada, ou seja, verificar se um dado elemento pertence ao conjunto. Para isso, devemos percorrer a lista até encontrar o elemento desejado (e retornar True, uma vez que o elemento desejado pertence a lista), ou até atingir o final da lista (e retornar False, pois o elemento não pertence ao conjunto). A figura a seguir ilustra esse processo.



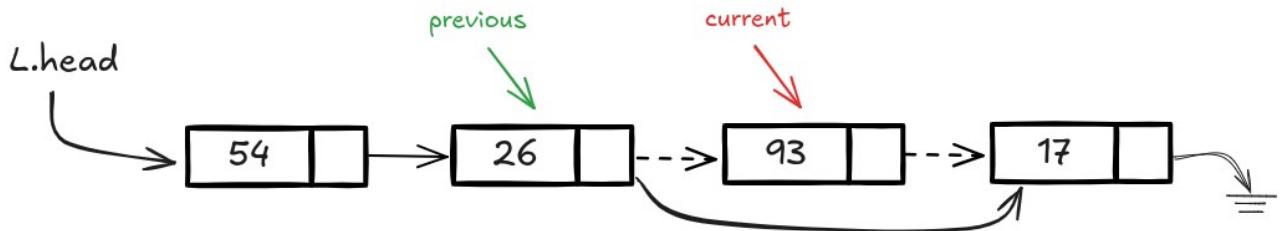
O algoritmo a seguir mostra uma implementação para essa funcionalidade.

```
# Busca pelo elemento na lista
search(L, key) {
    # Inicia na cabeça da lista
    temp = L.head
    # Percorre a lista até achar elemento ou chegar no final
    while temp != NIL {
        # Se achar atual nó contém elemento
        if temp.key == key
            return True
        else:
            temp = temp.next
    }
    return False
}
```

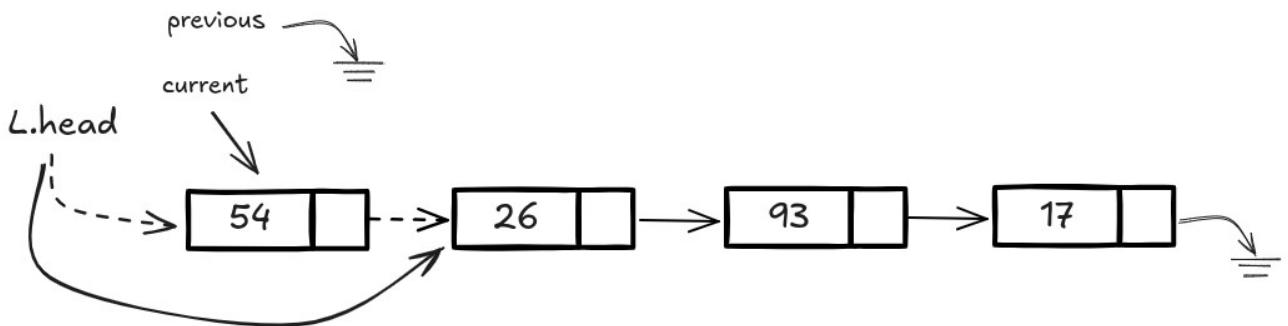
Por fim, uma operação importante é a remoção de um dado elemento da lista encadeada. Note que ao remover um nó da lista, precisamos religar o antecessor com o sucessor, de modo a evitar que elementos fiquem inacessíveis pela quebra do encadeamento sequencial. Primeiramente, precisamos ter duas referências se movendo ao longo da lista: current, que aponta para o elemento corrente da lista encadeada e previous, que aponta para seu antecessor. Eles devem se mover conjuntamente, até que current aponte diretamente para o nó a ser removido. A figura a seguir ilustra o processo.



Em seguida, devemos apontar o valor de next da referência previous para o mesmo local apontado pelo valor de next da referência corrente. Por fim, apontamos o valor de next da referência corrente para NIL, de modo a excluir completamente o nó da lista encadeada. A figura a seguir mostra uma ilustração gráfica do processo.



Porém, há um caso particular a ser considerado: se o nó que desejamos remover é o primeiro da lista. Neste caso, previous aponta para NIL, então devemos manipular a referência head, ou seja a cabeça da lista, conforme ilustra a figura a seguir.



O algoritmo a seguir apresenta uma implementação para o método remove().

```
# Remove um nó da lista encadeada
remove(L, key) {
    current = L.head
    previous = NIL
    # Enquanto não encontrar o valor a ser removido
    while current != NIL {
        # Se nó corrente armazena a chave desejada, OK
        if current.key == key
            break
        else {
            # Se no corrente não é o que buscamos
            # Atualiza o previous e o corrente
            previous = current
            current = current.next
        }
    }
    # Se nó a ser removido for o primeiro da lista
    if previous == NIL
        L.head = current.next
    else
        # Caso não seja primeiro nó, liga o previous com o próximo
        previous.next = current.next
    # Desliga nó corrente
    current.next = NIL
}
```

E assim, o TAD ListaEncadeada está completo. É interessante notar as complexidades das operações de uma lista encadeada. A função `length(L)` e a função `add-head(L, key)` são $O(1)$, enquanto as funções `add_tail(L, key)`, `search(L, key)` e `remove(L, key)` são todas $O(n)$.

Problema: Pontos críticos em uma lista encadeada

Dado uma lista encadeada de números inteiros, encontre o número de pontos críticos.

OBS: O início e o fim não são considerados pontos críticos (primeiro e último elementos).

Mínimos ou máximos locais são chamados de pontos críticos.

Um nó é chamado de mínimo local se tanto o elemento anterior quanto o seguinte forem maiores que o elemento atual.

Um nó é chamado de máximo local se tanto o elemento anterior quanto o seguinte forem menores que o elemento atual.

Entrada

$n = 8$

1 2 3 3 3 5 1 3

Saída

2 (sexto e sétimo)

$n = 7$

1 2 3 2 1 3 2

3 (terceiro, quinto e sexto)

Problema: Faça um algoritmo que, dada uma lista encadeada, retorne a lista na ordem inversa.

Entrada

1 2 3 4 5

Saída

5 4 3 2 1

Pilhas com estruturas dinâmicas

É possível definir uma pilha de forma dinâmica. Para isso, faremos uso da TAD Dynamic Stack.

TAD Dynamic_Stack
Node top
int size (número de elementos presentes na pilha)

Durante a criação da pilha DS, devemos fazer o topo referenciar NIL.

```
init(DS) {  
    DS.top = NIL  
    DS.size = 0  
}  
  
is_empty(DS) {  
    if DS.size == 0  
        return True  
    else  
        return False  
}  
  
peek(DS) {  
    return DS.top.key  
}
```

```

# Insere elemento no início do encadeamento
push(DS, key) {
    temp = Node(key)
    temp.next = DS.top
    DS.top = temp
    DS.size += 1
    return DS.top
}

# Remove elemento no início do encadeamento
pop(DS) {
    if DS.size == 0
        error('underflow')
    else {
        key = DS.top.key
        temp = DS.top
        DS.top = DS.top.next
        temp.next = NIL
        DS.size -= 1
        return key
    }
}

```

Note que assim como na versão estática, na versão dinâmica a complexidade das operações push e pop também são O(1).

Filas com estruturas dinâmicas

Assim, como uma pilha, podemos implementar uma fila de maneira dinâmica. Para isso, faremos uso da TAD Dynamic Queue.

```

TAD Dynamic_Queue
    Node head
    int size           (número de elementos presentes na fila)

init(DQ) {
    DQ.head = NIL
    DQ.size = 0
}

is_empty(DQ) {
    if DQ.size == 0
        return True
    else
        return False
}

# Insere elemento no final do encadeamento
enqueue(DQ, key) {
    # Vai até o fim da fila (tail)
    temp = DQ.head
    while temp.next != NIL
        temp = temp.next
    new = Node(key)
    new.next = NIL
}

```

```

        temp.next = new
        DQ.size += 1
        return DQ.head
    }

# Remove elemento no início do encadeamento
dequeue(DQ) {
    if DQ.size == 0
        error('underflow')
    else {
        key = DQ.head.key
        temp = DQ.head
        DQ.head = DQ.head.next
        temp.next = NIL
        DQ.size -= 1
    }
    return key
}

```

Note que nesta implementação, a função dequeue é O(1), mas a função enqueue é O(n), visto que devemos inserir um novo nó no final da fila. Porém, é possível fazer a operação enqueue ter complexidade O(1). Para isso devemos alterar a definição do TAD Dynamic_Queue para conter uma referência para o último nó da lista.

```

TAD Dynamic_Queue
    Node head
    Node tail
    int size      (número de elementos presentes na fila)

init(DQ) {
    DQ.head = NIL
    DQ.tail = NIL
    DQ.size = 0
}

# Insere elemento no final do encadeamento em O(1)
enqueue(DQ, key) {
    if DQ.head == NIL {
        new = Node(key)
        new.next = NIL
        DQ.head = new
        DQ.tail = new
    }
    else {
        # Não precisa percorrer toda estrutura
        new = Node(key)
        new.next = NIL
        DQ.tail.next = new
        DQ.tail = new
    }
    DQ.size += 1
    return DQ.tail
}

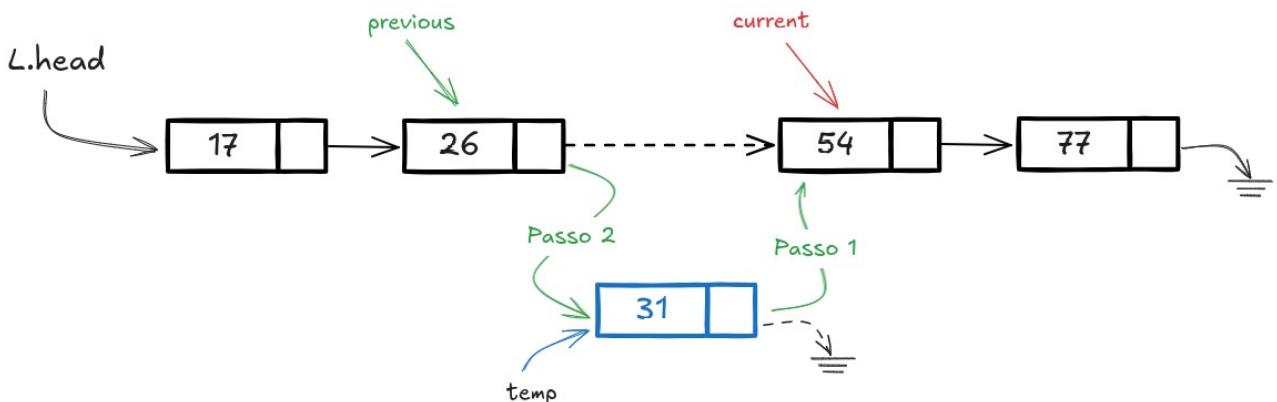
```

Listas encadeadas ordenadas

Quando trabalhamos com listas ordenadas, os dois métodos que precisam de ajustes em relação as listas encadeadas não ordenadas são `search()` e `add()`. Na inserção de um novo nó, devemos primeiramente encontrar sua posição na lista. Na busca pelo elemento, não precisamos percorrer toda a lista encadeada, pois se o elemento buscado é maior que o atual e ainda não o encontramos, significa que ele não pertence a lista. A seguir apresentamos a função que verifica se um elemento faz parte de uma lista ordenada ou não.

```
search(OL, key):
    # Inicio da lista
    current = OL.head
    # Enquanto não atingir o final da lista
    while current != NIL {
        # Se nó atual é o elemento, encontrou
        if current.key == key:
            return True
        else {
            # Se elemento atual é maior que valor buscado, pare
            if current.key > key
                break
            else
                current = current.next
        }
    }
    return False
}
```

Devemos também modificar o método `add()`, que insere um novo elemento a lista ordenada. A ideia consiste em encontrar a posição correta do elemento na lista ordenada, então para isso é mais fácil iniciar pela cabeça da lista. A figura a seguir ilustra o processo.



Para encontrar a posição correta precisamos de duas referências, assim como na remoção de um elemento. A posição correta da inserção na lista ordenada ocorre exatamente quando o valor da referência prévia é menor que o valor do novo elemento, que por sua vez é menor que o valor da referência atual. Note na figura que 31 está entre 26 e 54.

```
# Adiciona elemento na posição correta da lista
add(OL, key) {
    # Inicia na cabeça da lista
```

```

current = OL.head
previous = NIL
# Enquanto não chegar no final
while current != NIL {
    # Se valor do corrente for maior elemento desejado, pare
    if current.key > key
        break
    else {
        # Senão, move o prévio e o corrente para o próximo
        previous = current
        current = current.next
    }
}
# Cria novo nó
temp = Node(key)
# Se for primeiro elemento, prévio = NIL (muda cabeça da lista)
if previous == NIL {
    temp.next = OL.head
    OL.head = temp
}
else {
    # Senão, estamos no meio ou último
    temp.next = current
    previous.next = temp
}
}

```

A diferença em relação a complexidade da lista encadeada não ordenada é que na lista ordenada a inserção é sempre O(n). Na lista encadeada padrão, a inserção no início possui complexidade O(1) e a inserção no final possui complexidade O(n).

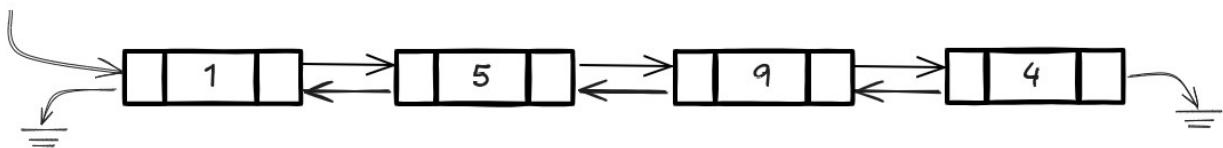
Listas duplamente encadeadas

Listas duplamente encadeadas nos permitem navegar nos dois sentidos, ou seja, tanto da esquerda para direita quanto da direita para esquerda. Isso é possível graças a inclusão de uma referência para o nó anterior. Essa pequena modificação trás algumas vantagens, como a remoção não precisar de uma referência auxiliar.

TAD Node
int key
Node prev
Node next

TAD Double_Linked_List
Node head
int size (número de elementos presentes na lista)

DL.head



A seguir apresentamos as primitivas de uma lista duplamente encadeada, lembrando que as principais diferenças em relação a uma lista encadeada tradicional estão na inserção no início, inserção no final e remoção.

```
init(DL) {
    DL.head = NIL
    DL.size = 0
}

# Adiciona no início de uma lista duplamente encadeada
add_head(DL, key) {
    temp = Node(key)
    if DL.head == NIL {
        DL.head = temp
        temp.next = NIL
    }
    else {
        # Aponta novo nó para cabeça da lista
        temp.next = DL.head
        DL.head.prev = temp
        # Atualiza a cabeça da lista
        DL.head = temp
    }
    temp.prev = NIL
    DL.size += 1
}

# Adiciona no fim de uma lista duplamente encadeada
add_tail(DL, key) {
    # Cria novo nó
    tail = Node(key)
    if DL.head == NIL {
        DL.head = tail
        tail.prev = NIL
    }
    else {
        # Usa referência temporária para percorrer lista (cabeça)
        temp = DL.head
        # Percorre a lista até o último elemento
        while temp.next != NIL:
            temp = temp.next
        # Aponta tail (ultimo elemento) para novo nó
        temp.next = tail
        tail.prev = temp
    }
    tail.next = NIL
    L.size += 1
}
```

A busca na lista duplamente encadeada pode ser feita de maneira idêntica à busca na lista encadeada tradicional. Porém, a remoção pode ser feita sem a necessidade de uma referência auxiliar, confirme ilustra o algoritmo a seguir.

```
remove(DL, key) {
    # encontra a posição do nó a ser removido
    current = DL.head
    while current != NIL {
        if current.key == key
            break
        else
            current = current.next
    }
    if current == NIL
        return False
    elif current == DL.head {
        DL.head = current.next
        DL.head.prev = NIL
    }
    else {
        current.prev.next = current.next
        current.next.prev = current.prev
    }
    # Desliga nó corrente
    current.next = NIL
    current.prev = NIL
}
```

Listas duplamente encadeadas com sentinelas

Conforme vimos anteriormente, a inserção no final de uma lista duplamente encadeada tem complexidade $O(n)$. Uma maneira de melhorar essa operação consiste na definição de listas duplamente encadeadas com sentinelas. Em listas duplamente encadeadas com sentinelas, tanto a inserção quanto a remoção no final são operações $O(1)$, pois existem duas extremidades na lista: head (início) e tail (final). São como nós que não podem ser removidos e por essa razão recebem o nome de sentinelas.

Em uma lista duplamente encadeada, cada nó possui uma informação e duas referências: uma para o nó antecessor e outra para o nó sucessor.

```
TAD Node
    int key
    Node prev
    Node prox

TAD Sentinel_Double_Linked_List
    Node header      (sentinela: nó que não pode ser removido)
    Node trailer     (sentinela: nó que não pode ser removido)
    int size         (número de elementos presentes na lista)

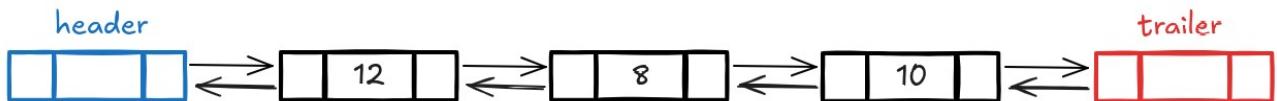
init(DL) {
    DL.header = new Node()      # cria sentinelas 1 (nó sem chave)
    DL.trailer = new Node()     # cria sentinelas 2 (nó sem chave)
    DL.header.prev = NIL
    DL.header.next = DL.trailer
```

```

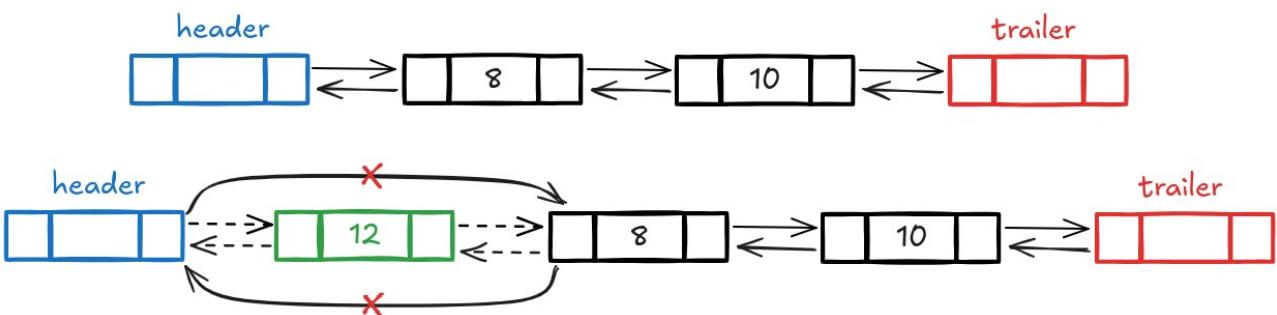
    DL.trailer.prev = DL.header
    DL.trailer.next = NIL
    DL.size = 0
}

```

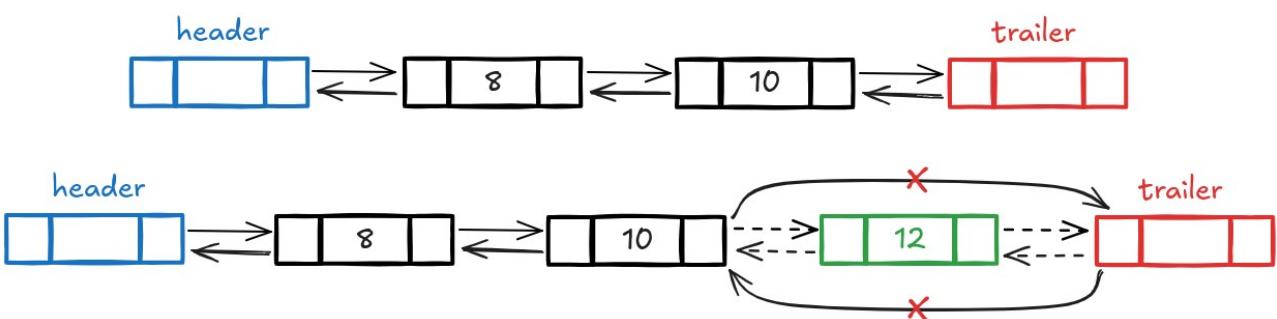
Assim como a lista encadeada possui uma cabeça (head) que sempre aponta para o início do encadeamento lógico, uma lista duplamente encadeada possui duas referências especiais: a própria cabeça, que chamaremos de header, e a cauda, que chamaremos de trailer. A figura a seguir ilustra a estrutura de uma lista duplamente encadeada.



Para essa classe, adotaremos a estratégia de a cada nó inserido, incrementar em uma unidade o seu tamanho e a cada nó removido, decrementar em uma unidade o seu tamanho, assim não precisamos de uma função para contar quantos elementos existem na lista. Com relação a operação de inserção, a principal diferença em relação a lista encadeada é que aqui devemos ligar o novo nó tanto ao seu elemento sucessor quanto ao seu elemento antecessor, conforme ilustra a figura a seguir.



A mesma observação vale para a remoção de um nó. Para desconectá-lo completamente da lista duplamente encadeada, devemos ligar o antecessor ao sucessor e vice-versa.



O algoritmo para a função auxiliar `insert_between()` é apresentado a seguir. Basicamente, ele insere um novo nó entre dois nós já existentes.

```

# Insere novo nó entre dois nós existentes
insert_between(DL, key, predecessor, successor) {
    new = Node(key)
    predecessor.next = new
    successor.prev = new
    new.prev = predecessor

```

```

        new.next = successor
        DL.size += 1
        return new
    }
}

```

De posse da função anterior, a inserção no início da lista duplamente encadeada pode ser realizada de acordo com o algoritmo a seguir.

```

# Insere elemento no início
insert_first(DL, key){
    # Nó deve ficar entre header e header.next
    insert_between(DL, key, DL.header, DL.header.next)
    DL.size += 1
}

```

De maneira análoga, a inserção no final da lista duplamente encadeada pode ser realizada de acordo com o algoritmo a seguir.

```

# Insere elemento no final
def insert_last(DL, key) {
    # Nó deve entrar entre trailer.prev e trailer
    insert_between(DL, key, DL.trailer.prev, DL.trailer)
    DL.size += 1
}

```

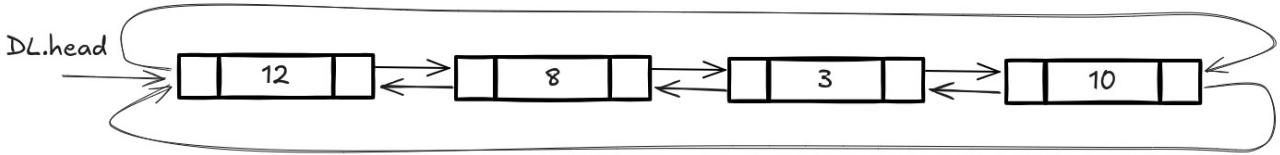
Note que diferentemente das listas encadeadas padrão, a inserção no final em uma lista duplamente encadeada possui complexidade $O(1)$, o que pode ser uma grande vantagem em diversas aplicações. A remoção de um nó da lista duplamente encadeada pode ser feita de maneira similar a inserção, usando uma função auxiliar `delete_node()`.

```

# Remove um nó intermediário da lista
# OBS: header e trailer nunca podem ser removidos!
delete_node(DL, key) {
    node = search(DL, key)
    if node == NIL
        return False
    else {
        predecessor = node.prev
        successor = node.next
        predecessor.next = successor
        successor.prev = predecessor
        DL.size -= 1
        # Armazena o elemento removido
        node.prev = NIL
        node.next = NIL
        x = node.key
        return x
    }
}

```

Outra forma de melhorar a inserção no final para que tenha custo computacional $O(1)$ é criar uma lista duplamente encadeada circular, onde o último elemento se conecta com o primeiro, conforme indica a figura a seguir.



As funções a seguir mostram como funcionam os algoritmos de inserção no início e no final de uma lista circular duplamente encadeada.

```
# Insere no início da lista circular duplamente encadeada
add_head(DL, key) {
    temp = Node(key)
    if DL.head == NIL {
        DL.head = temp
        temp.next = temp
        temp.prev = temp
    }
    else {
        # Aponta novo nó para cabeça da lista
        temp.next = DL.head
        DL.head.prev.next = temp
        temp.prev = DL.head.prev
        DL.head.prev = temp
        # Atualiza a cabeça da lista
        DL.head = temp
    }
    temp.prev = NIL
    DL.size += 1
}

# Insere no final da lista circular duplamente encadeada
add_tail(DL, key) {
    temp = Node(key)
    if DL.head == NIL {
        DL.head = temp
        temp.next = temp
        temp.prev = temp
    }
    else {
        # Aponta novo nó para cabeça da lista
        temp.next = DL.head
        DL.head.prev = temp
        temp.prev = DL.head.prev
        DL.head.prev.next = temp # não precisa atualizar cabeça
    }
    temp.prev = NIL
    DL.size += 1
}
```

Aplicações: matriz esparsa com arrays e listas encadeadas

Uma aplicação de grande relevância na ciência da computação consiste na representação eficiente de matrizes esparsas. Uma matriz é dita esparsa quando 2/3 ou mais de seus elementos são nulos. Um exemplo de matriz esparsa é ilustrado a seguir.

```

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

```

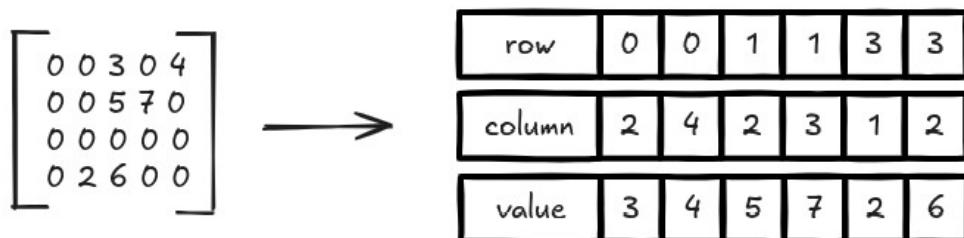
Note que dos 20 elementos, apenas 6 são não nulos, o que representa apenas 30% do total. Representar matrizes esparsas da forma tradicional, com um array bidimensional de n linhas por m colunas não é eficiente, pois há um enorme desperdício de memória.

Suponha que os elementos da matriz anterior são representados como ponto flutuante (float) e cada elemento ocupa 64 bits. Sendo assim, a matriz toda ocupa $64 \times 20 = 1280$ bits.

A seguir, veremos duas abordagens para armazenar matrizes esparsas de maneira mais eficiente.

Método 1: Arrays estáticos

Devemos criar 3 arrays: R (rows), C (columns) e V (values) de tamanho K, onde K é o número de elementos não nulos na matriz esparsa. O processo segue a lógica indicada pela figura a seguir.



onde

- * row indica a linha em que o elemento não nulo se encontra
- * column indica a coluna em que o elemento não nulo se encontra
- * value indica o valor do elemento não nulo localizado em (row, column)

Como os índices são inteiros, suponha que cada elemento dos arrays R e C ocupe apenas 32 bits de memória (menor que float). Os elementos do array V devem ser representados como float (64 bits). Sendo assim, o espaço total de memória ocupada pelos arrays R, C e V é dado por:

$$32 \times 6 + 32 \times 6 + 64 \times 6 = 192 + 192 + 384 = 768$$

Sendo assim, como $768/1280 = 0.6$, temos uma economia de 40%, o que representa uma quantidade significativa de espaço! Um algoritmo para codificar uma matriz esparsa A usando essa estratégia é dado a seguir. Repare que apesar de tudo, a representação utilizada para a codificação ainda é estática.

```

A: matriz esparsa n x m
sparse_matrix_encode(A, n, m) {
    K = 0
    for i = 1 to n {
        for j = 1 to m
            if A[i, j] != 0
                K += 1
    }
    int R[K]           # array de inteiros de tamanho K
    int C[K]           # array de inteiros de tamanho K

```

```

float V[K]           # array de floats de tamanho K
k = 1
for i = 1 to n {
    for j = 1 to m {
        if A[i, j] != 0 {
            R[k] = i
            C[k] = j
            V[k] = A[i, j]
            k += 1
        }
    }
}

```

Método 2: Listas encadeadas

Devemos criar uma lista encadeada em que cada nó contém 3 campos de informação e uma referência para o próximo no.

```

TAD Node
    int row
    int col
    float data
    Node next      # referência para um outro nó (encadeamento lógico)

```

```

TAD Sparse_Matrix
    Node head
    Node tail      # referência para último nó inserido
    int size

init(SM) {
    SM.head = NIL
    SM.size = 0
}

```

```

# Cria novo nó e conecta no final da lista encadeada
create_new_node(SM, row, col, data) {
    new_node = Node(row, col, data, NIL)
    if SM.size == 0 {
        SM.head = new_node
    } else
        SM.tail.next = new_node
    SM.tail = new_node
    SM.size += 1
}
sparse_matrix_encode(SM, A, n, m) {
    for i = 1 to n {
        for j = 1 to m {
            if A[i, j] != 0
                create_new_node(SM, i, j, A[i, j])
        }
    }
}

```

Note que em termos de espaço a estrutura dinâmica é muito similar a estrutura estática. A grande diferença porém está na capacidade de inserir/remover nós. É muito mais simples aumentar o

tamanho da lista encadeada do que de uma matriz estática. Sendo assim, a estrutura dinâmica é mais adequada para aplicações em que os dados contidos na matriz esparsa podem sofrer alterações.

Aplicações: listas ortogonais e matrizes dinâmicas

Uma lista ortogonal é uma estrutura dinâmica compostas por nós que possuem referências para 4 outros nós nas direções: cima, baixo, esquerda e direita. Da mesma forma que uma matriz é uma versão 2D de um vetor (array), uma lista ortogonal é uma generalização 2D de uma lista encadeada.

```
TAD OLNode
    int key
    Node up
    Node down
    Node left
    Node right

TAD Orthogonal_List
    OLNode head
    int size

init() {
    SM.head = build_orthogonal_list(A, n, m)
    SM.size = n*m
}

build_orthogonal_list(A, n, m) {
    OLNode array[n, m] map      # matriz auxiliar para encadeamentos
    for i = 1 to n {
        for j = 1 to m {
            # Cria um novo nó para cada entrada da matriz
            new_node = OLNode(A[i, j])
            # Armazena referência para o novo nó
            map[i, j] = new_node
            # Ajusta referências de cima e de baixo
            if i != 1 {
                new_node.up = map[i-1, j]
                map[i-1, j].down = new_node
            }
            # Ajusta referências da esquerda e direita
            if j != 1 {
                new_node.left = map[i, j-1]
                map[i, j-1].right = new_node
            }
        }
    }
    return map[0, 0]      # referência para nó inicial
}

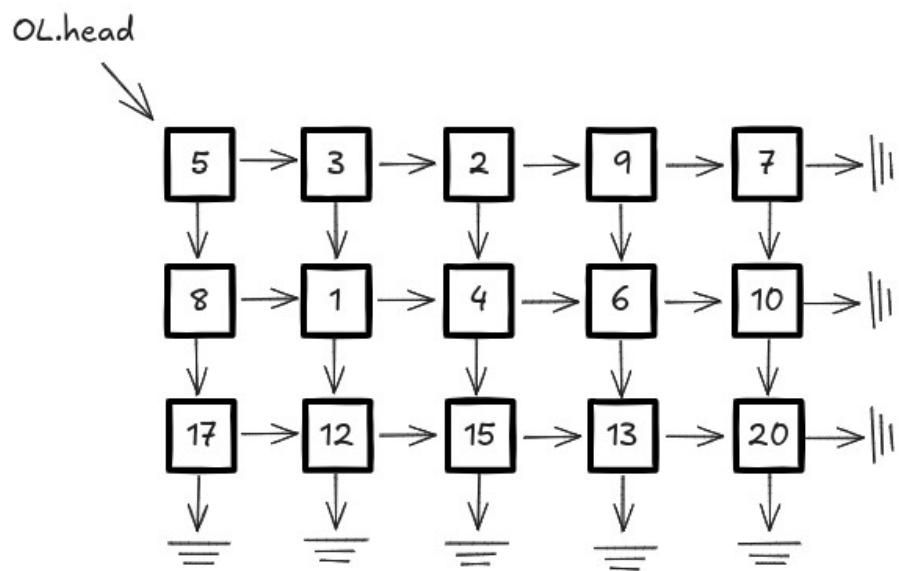
# Percorre a lista ortogonal (linha por linha)
traverse_orthogonal_list(OL) {
    current_row = OL.head
    current_col = NIL
    while current_row != NIL {
        current_col = current_row
        while current_col != NIL {
            print(current_col.key)
            current_col = current_col.right
        }
        print("\n")
    }
}
```

```

        current_row = current_row.down
    }
}

```

A figura a seguir ilustra uma representação gráfica de uma lista ortogonal.



Até o presente momento, estudamos estruturas de dados lineares, como listas, pilhas e filas. A partir de agora, iremos estudar estruturas consideradas não lineares, no sentido de que o encadeamento lógico dos elementos permite organizações mais complexas e otimizadas para a busca. Esse é o caso das árvores binárias, assunto das próximas aulas.

“Conquistar nossos objetivos requer tempo e maturação. Lembre-se que a última parte a crescer em uma árvore são os frutos.”
 (Autor anônimo)

Árvores Binárias

Árvores binárias são estruturas de dados dinâmicas baseadas em encadeamento lógico e que possuem estrutura hierárquica. Em resumo, cada nó de uma árvore binária de busca T deve conter pelo menos 4 informações:

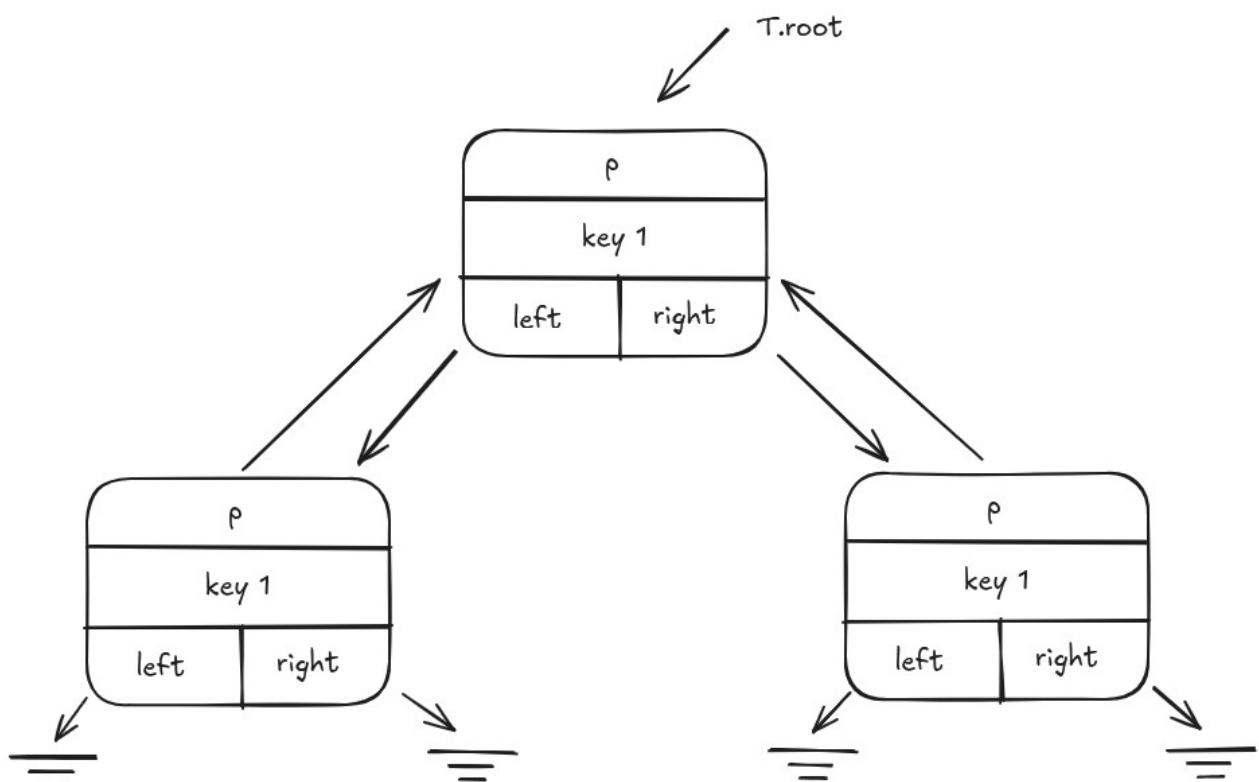
* **Chave (key)**: valor da ser armazenado em um nó de T

* **p**: referência para o nó pai

* **left**: referência para o filho a esquerda

* **right**: referência para o filho a direita

Uma árvore binária T deve sempre ter uma raiz. Denotaremos aqui por *T.root*. Também podemos ter um atributo *size* para armazenar o número de nós da árvore, mas é opcional.



Pode-se definir um TAD para o nó de uma árvore binária como:

```
TAD TreeNode
    int key
    Node p
    Node left
    Node right
```

```
TAD Binary_Tree
    TreeNode root
    int size
```

Uma pergunta natural a essa altura é: como podemos percorrer todos os nós de uma árvore binária? Utilizando uma abordagem recursiva, temos uma solução simples.

```

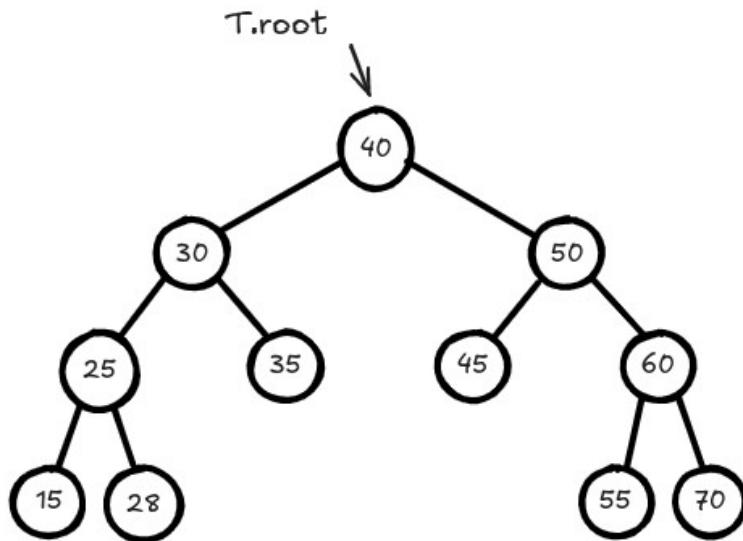
Tree_Walk_Inorder(x) {
    if x ≠ NIL {
        Tree_Walk_Inorder(x.left)
        print(x.key)
        Tree_Walk_Inorder(x.right)
    }
}

Tree_Walk_Preorder(x) {
    if x ≠ NIL {
        print(x.key)
        Tree_Walk_Preorder(x.left)
        Tree_Walk_Preorder(x.right)
    }
}

Tree_Walk_Postorder(x) {
    if x ≠ NIL {
        Tree_Walk_Postorder(x.left)
        Tree_Walk_Postorder(x.right)
        print(x.key)
    }
}

```

Supondo a árvore binária a seguir, como ficam a ordem de acesso aos nós quando utilizamos os métodos Inorder, Preorder e Postorder?



=> Percorso inorder: 15, 25, 28, 30, 35, 40, 45, 50, 55, 60, 70

=> Percorso preorder: 40, 30, 25, 15, 28, 35, 50, 45, 60, 55, 70

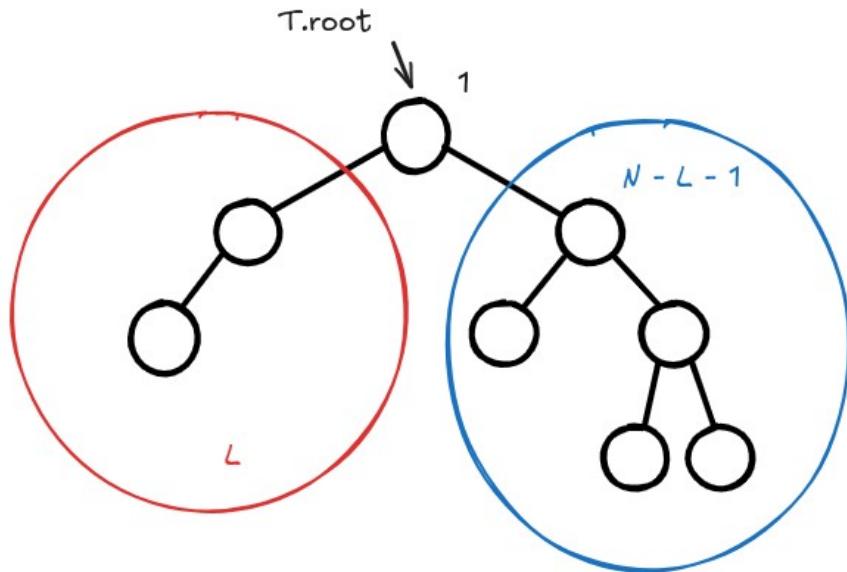
=> Percorso postorder: 15, 28, 25, 35, 30, 45, 55, 70, 60, 50, 40

A pergunta natural é: qual a complexidade das funções para percorrer os nós de uma árvore binária?

Veremos a resposta na seção a seguir.

Análise da complexidade

Seja uma árvore de N nós de modo que existam exatamente L nós na subárvore a esquerda da raiz e $N - L - 1$ nós na subárvore a direita. Veja que $(N - L - 1 + L) + 1 = N$.



Então, podemos definir a seguinte recorrência:

$$T(N) = T(L) + T(N - L - 1) + C$$

onde C é uma constante (print). Para um limite superior, vamos considerar o pior cenário: uma árvore desbalanceada para a direita (aumenta a profundidade da árvore). Isso significa ter $L = 0$, ou seja:

$$T(N) = T(0) + T(N - 1) + C$$

Expandindo recursivamente $T(N-1)$, temos:

$$T(N) = T(0) + T(0) + T(N - 2) + C + C$$

Repetindo o processo:

$$T(N) = T(0) + T(0) + T(0) + T(N - 3) + C + C + C$$

Continuando o processo até $T(1)$ teremos justamente $N - 1$ passo, ou seja:

$$T(N) = (N - 1)T(0) + T(1) + (N - 1)C = NT(0) - T(0) + T(1) + NC - C$$

Mas $T(0) = 1$ e $T(1)$ é uma constante arbitrária K , o que nos leva a:

$$T(N) = N - 1 + K + NC - C$$

o que resulta em $O(N)$.

Inserção em árvores binárias

Um aspecto complicado sobre inserção em árvores binárias é que no início, quando temos apenas a raiz da árvore, existem apenas duas posições possíveis: a esquerda ou a direita da raiz. Suponha que o novo nó x seja inserido a esquerda da raiz. Agora, note que existem 3 possibilidades para um novo nó. Suponha que um novo nó y seja inserido a direita da raiz (irmão de x). Assim, o número de possibilidades agora aumenta para 4. Esse padrão segue de modo que o número de posições possíveis para um nó é cada vez maior quanto mais nós são inseridos na árvore. Em resumo, na k -ésima inserção, temos k possíveis slots para a chave.

A princípio, em uma árvore binária arbitrária, uma chave pode ser inserida em qualquer local. Podemos assim, adotar uma estratégia bastante simples: ao atingir um nó x tente inserir uma chave k a sua esquerda, caso não seja possível, tente na direita. Caso ambos os filhos seja diferentes de NIL, faça um sorteio e prossiga para o próximo nível da árvore.

```
# Ideia: passar T.root em x
Tree_Insert(x, key) {
    # Novo nó será raiz da árvore
    if x == NIL {
        node = TreeNode(key)
        node.left = NIL          # nó inserido com certeza será uma folha!
        node.right = NIL
        node.p = NIL
    }
    elif x.left == NIL {         # há espaço na esquerda
        node = TreeNode(key)
        node.left = NIL
        node.right = NIL
        x.left = node
        node.p = x
    }
    elif x.right == NIL {        # há espaço na direita
        node = TreeNode(key)
        node.left = NIL
        node.right = NIL
        x.right = node
        node.p = x
    }
    else {                      # Ambos os filhos são diferentes de NIL
        m = random(2) # gera inteiro aleatório (zero ou um)
        if m == 0
            Tree_Insert(x.left, key)
        else
            Tree_Insert(x.right, key)
    }
}
```

Uma limitação do algoritmo acima é que ele é randomizado, ou seja, se inserirmos um mesmo conjunto de chaves em duas árvores distintas, a posição dos elementos não será a mesma. Isso traz dificuldades para a busca de uma chave key . Devemos percorrer toda a árvore em busca da chave key , utilizando uma estratégia similar a utilizada nas funções `Tree_Walk`, o que gera um custo computacional $O(n)$.

```

Tree_Search(x, key) {
    if x == NIL
        return False
    if x.key == key
        return True
    left = Tree_Search(x.left, key) # Procura na subárvore a esquerda
    # Se achou, não precisa continuar
    if left
        return True
    right = Tree_Search(x.right, key) # Procura na subárvore a direita
    return right
}

```

Remoção em árvores binárias

Um problema complexo em árvores binárias é a remoção de nós. É preciso considerar muitos casos distintos durante a remoção: devemos saber quantos filhos o nó a ser removido tem e se ele é um filho a esquerda ou a direita.

```

Tree_Delete(x, key) {
    if x != NIL { # condição de parada da recursão
        if x.key == key {
            if x.left == NIL and x.right == NIL { # nó folha
                if x == x.p.left # é filho a esquerda
                    x.p.left = NIL
                else
                    x.p.right = NIL
                x.p = NIL
            }
            elif x.left == NIL { # tem 1 filho a direita
                if x == x.p.right { # é filho a direita
                    x.right.p = x.p
                    x.p.right = x.right
                    x.p = NIL
                    x.right = NIL
                }
                else { # é filho a esquerda
                    x.right.p = x.p
                    x.p.left = x.right
                    x.p = NIL
                    x.left = NIL
                }
            }
            elif x.right == NIL { # tem 1 filho a esquerda
                if x == x.p.left { # é filho a esquerda
                    x.left.p = x.p
                    x.p.left = x.left
                    x.p = NIL
                    x.left = NIL
                }
                else { # é filho a direita
                    x.right.p = x.p
                    x.p.right = x.left
                    x.p = NIL
                    x.right = NIL
                }
            }
            else { # aqui tem os 2 filhos!
                if x.p.right == x { # é filho a direita

```

```

        x.right.p = x.p
        x.p.right = x.right
        x.p = NIL
        temp = x.left
        x.left = NIL
        # Religar subárvore a esquerda
        while x.right != NIL
            x = x.right
            x.right = temp
            temp.p = x
        }
        else {      # é filho a esquerda
            x.left.p = x.p
            x.p.left = x.left
            x.p = NIL
            temp = x.right
            x.right = NIL
            # Religar subárvore a direita
            while x.left != NIL
                x = x.left
                x.left = temp
                temp.p = x
            }
        }
    }
    else {
        Tree_Delete(x.left)      # Procura na subárvore a esquerda
        Tree_Delete(x.right)     # Procura na subárvore a direita
    }
}
}

```

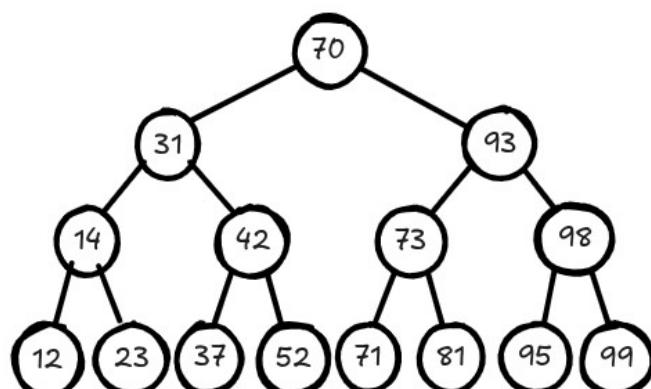
Árvores binárias de busca

Em árvores binárias de busca, toda chave possui uma localização específica dentro do conjunto, assim como uma lista ordenada. Toda árvore binária de busca satisfaz a seguinte propriedade chave.

Propriedade chave: Seja x um nó arbitrário de uma árvore binária de busca T. Se y é um nó pertencente a subárvore a esquerda de x, então $y.key \leq x.key$. Se y é um nó pertencente a subárvore a direita de x, então $y.key \geq x.key$.

Essa propriedade faz com que cada chave tenha uma posição única na árvore!

Em uma árvore binária de busca, não podemos mais inserir o nó onde desejarmos: devemos sempre manter a propriedade chave válida! A figura a seguir ilustra uma árvore binária de busca.



Note que o percurso inorder definido sempre irá imprimir as chaves dos nós da árvore em ordem crescente. Por exemplo, na árvore da figura acima, teremos como saída a seguinte sequência de chaves: 12, 14, 23, 31, 37, 42, 52, 70, 71, 73, 81, 93, 95, 98, 99. Outra primitiva importante em árvores binárias de busca consiste em encontrar a menor e a maior chave do conjunto. Note que, devido a propriedade chave de tais árvores, essa tarefa se torna trivial.

```
# Passar o nó raiz como parâmetro
Tree_Minimum(x) {
    while x.left ≠ NIL
        x = x.left
    return x
}

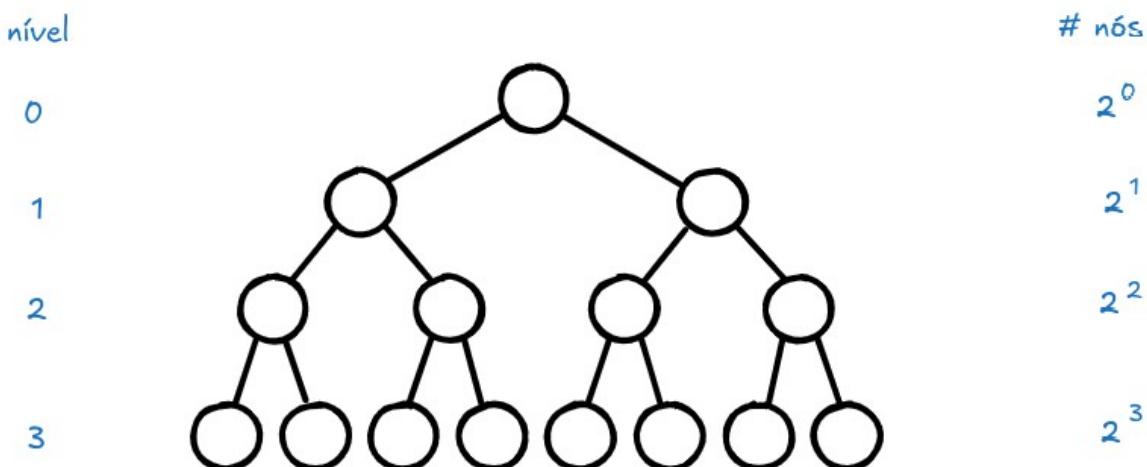
# Passar o nó raiz como parâmetro
Tree_Maximum(x) {
    while x.right ≠ NIL
        x = x.right
    return x
}
```

Busca em árvores binárias de busca

Há duas formas de pensar nesse algoritmo: recursiva ou iterativa. Iniciaremos com a recursiva.

```
Recursive_Tree_Search(x, k) {
    if x == NIL or k == x.key
        return x
    if k < x.key
        return Recursive_Tree_Search(x.left, k)
    else
        return Recursive_Tree_Search(x.right, k)
}
```

Note que a complexidade do algoritmo depende essencialmente da altura h da árvore. No melhor caso, quando a árvore encontra-se perfeitamente balanceada, ou seja, as alturas das subárvores a esquerda são iguais as alturas das subárvores a direita em todos os nós, temos uma situação com a ilustrada pela figura a seguir.



A relação entre o número de nós n e a altura da árvore vem de:

$$n = \sum_{k=0}^h 2^k = 2^0 + 2^1 + 2^2 + \dots + 2^h$$

Sabemos que $2^{k+1} = 2 \cdot 2^k = 2^k + 2^k$, o que nos leva a $2^h = 2^{h+1} - 2^h$. Sendo assim, podemos calcular n como uma soma telescópica:

$$n = \sum_{k=0}^h (2^{k+1} - 2^k) = 2^{h+1} - 1$$

Ou seja, temos que $2^h = n + 1$. Aplicando logaritmos de ambos os lados:

$$h = \log_2(n + 1) - 1$$

o que nos permite escrever que a altura da árvore é $O(\log_2 n)$. Por exemplo, suponha que $n = 15$. Qual é a menor altura da árvore?

$$h = \log_2 15 - 1 = 4 - 1 = 3$$

No pior caso, quando a árvore se degenera para uma lista encadeada, temos que a altura é igual a n. Sendo assim, a complexidade da busca em árvores binárias pode variar de $O(\log n)$ a $O(n)$. A seguir apresentamos a versão iterativa do algoritmo:

```
Iterative_Tree_Search(x, k) {
    while x ≠ NIL and k ≠ x.key {
        if k < x.key
            x = x.left
        else
            x = x.right
    }
    return x
}
```

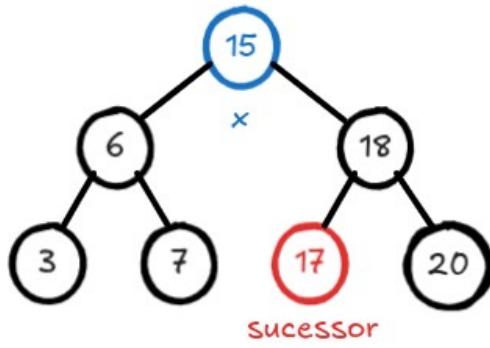
Sucessor e predecessor

Encontrar os elementos que precedem e sucedem um nó x na árvore é importante em diversos problemas. Iremos assumir a hipótese de que não existem duas chaves idênticas na árvore T por motivos de simplificação.

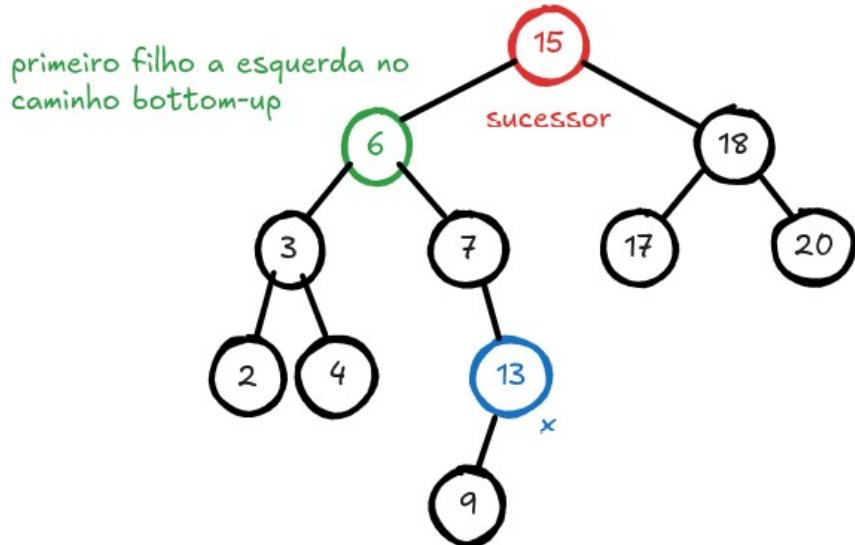
Def: O sucessor de um nó x é o nó y de menor chave tal que $y.key > x.key$, ou seja, é o próximo nó a ser visitado no percurso inorder.

Em resumo, há duas situações que podem ocorrer:

i) se a subárvore a direita do nó x não é vazia, então o sucessor é o menor elemento dessa subárvore.



ii) se a subárvore a direita de x é vazia, e x possui um sucessor y, então y é o antecessor mais baixo em T cujo filho a esquerda também é um ancestral de x.



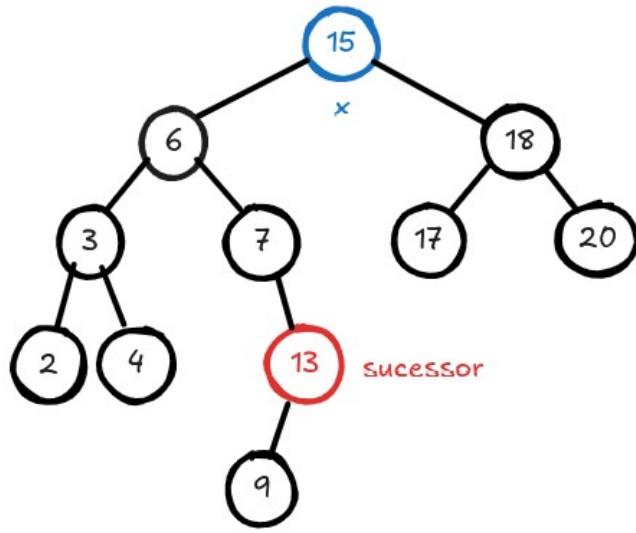
O algoritmo a seguir encontra o sucessor de um nó x em uma árvore binária de busca.

```
Tree_Successor(x) {
    if x.right ≠ NIL
        return Tree_Minimum(x.right)
    else {
        y = x.p
        while y ≠ NIL and x == y.right {
            x = y
            y = y.p
        }
        return y
    }
}
```

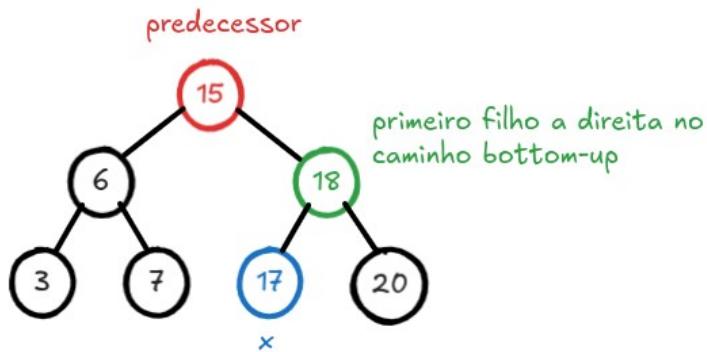
Note que a complexidade desse algoritmo também depende da altura h da árvore. Portanto, ela pode variar de $O(\log n)$ a $O(n)$.

Para encontrar o predecessor de um nó x na árvore devemos:

i) encontrar o maior elemento da subárvore a esquerda, se ela existir; ou



ii) se a subárvore a esquerda é vazia, então o predecessor y de x é o ancestral mais baixo em T cujo filho a direita também é ancestral de x;



```

Tree_Predecessor(x) {
    if x.left ≠ NIL
        return Tree_Maximum(x.left)
    else {
        y = x.p
        while y ≠ NIL and x == y.left {
            x = y
            y = y.p
        }
        return y
    }
}

```

Note que é um problema perfeitamente simétrico ao de encontrar o sucessor!

Inserção em árvores binárias de busca

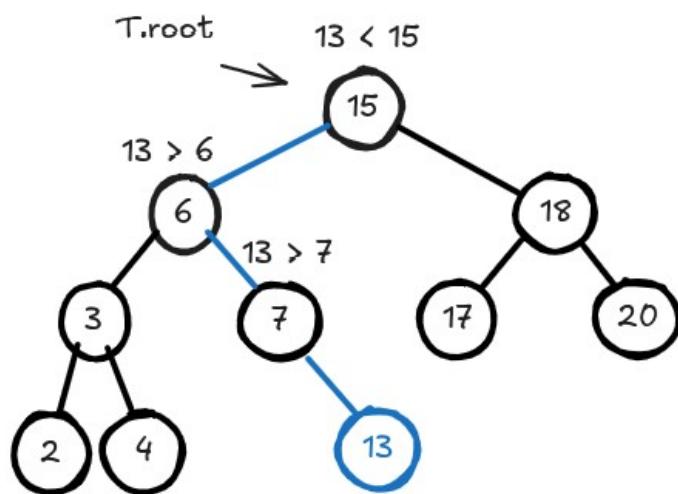
Para inserir um novo nó em uma árvore binária de busca não basta encontrar a primeira posição disponível. Devemos respeitar a propriedade chave: isso faz com que cada chave tenha sua posição correta dentro da estrutura. A função a seguir insere um novo nó z na árvore. Primeiro, devemos encontrar a posição correta de z em T, com base no valor de sua chave. Depois, realizamos o encadeamento lógico.

```

Tree_Insert(T, z) {
    x = T.root
    y = NIL
    # Encontra a posição do novo nó em T
    while x ≠ NIL {
        y = x
        if z.key < x.key
            x = x.left
        else
            x = x.right
    }
    # Realiza o encadeamento lógico
    z.p = y
    if y == NIL      # a raiz era vazia
        T.root = z
    else {
        if z.key < y.key      # adiciona a esquerda de y
            y.left = z
        else                  # adiciona a direita de y
            y.right = z
    }
}

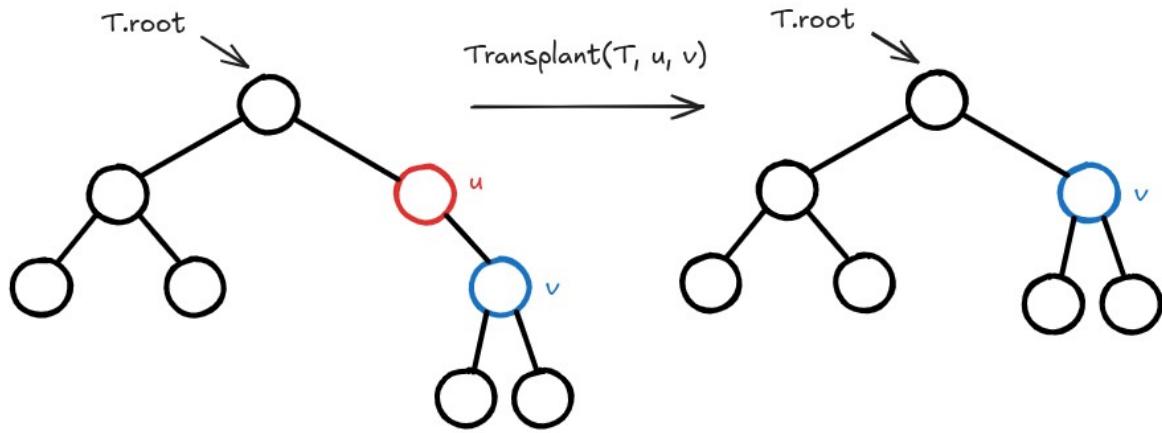
```

Assim como os algoritmos anteriores, temos que a complexidade da inserção depende da altura h da árvore: no melhor caso é $O(\log_2 n)$ e no pior caso é $O(n)$. A figura a seguir ilustra o processo de inserção da chave 13 em uma árvore binária de busca.



Remoção em árvores binárias de busca

A remoção é um processo mais complicado que a inserção. Para nos auxiliar nessa tarefa, iremos primeiramente apresentar uma primitiva auxiliar chamada Transplant. A função $\text{Transplant}(T, u, v)$ substitui a subárvore enraizada por u pela subárvore enraizada por v . A ideia é que quando a primitiva Transplant substitui uma subárvore enraizada em u por outra subárvore enraizada por v , o pai do nó u se torne o pai do nó v . É possível também que v seja NIL. A figura a seguir ilustra esse processo.



Essa operação é importante pois nos permite mover subárvores com custo $O(1)$. Em linguagens de programação modernas, após o transplante, o nó u fica solto e será desalocado da memória automaticamente pelo coletor de lixo. Caso contrário, é recomendável desalocar a memória manualmente.

O algoritmo a seguir ilustra a função `Transplant`.

```

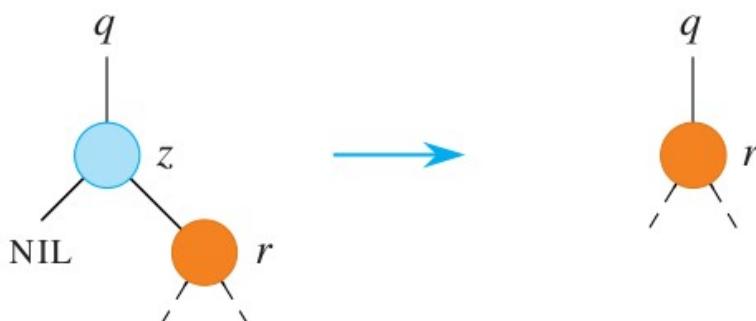
Transplant(T, u, v) {
    if u.p == NIL                      # nó u a ser substituído é a raiz de T
        T.root = v
    elif u == u.p.left {                 # u é filho a esquerda de alguém
        u.p.left = v
        u.left = NIL
    }
    else {                                # u é filho a direita de alguém
        u.p.right = v
        u.right = NIL
    }
    if v != NIL {
        v.p = u.p
        u.p = NIL
    }
}

```

Seja z o nó a ser removido de T . A estratégia para remover um nó z da árvore T baseia-se na análise de 2 casos principais:

a) z tem um único filho

a1) se z não tem filho a esquerda, substitua z pelo seu filho a direita (`Transplant`)



a2) se z não tem filho a direita, substitua z pelo seu filho a esquerda (Transplant)

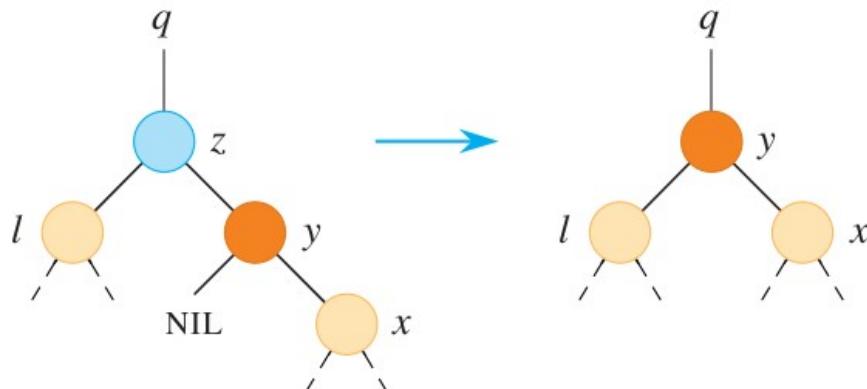


b) z tem ambos os filhos

Devemos encontrar o sucessor de z em T, denominado de y. Podem ocorrer 2 situações:

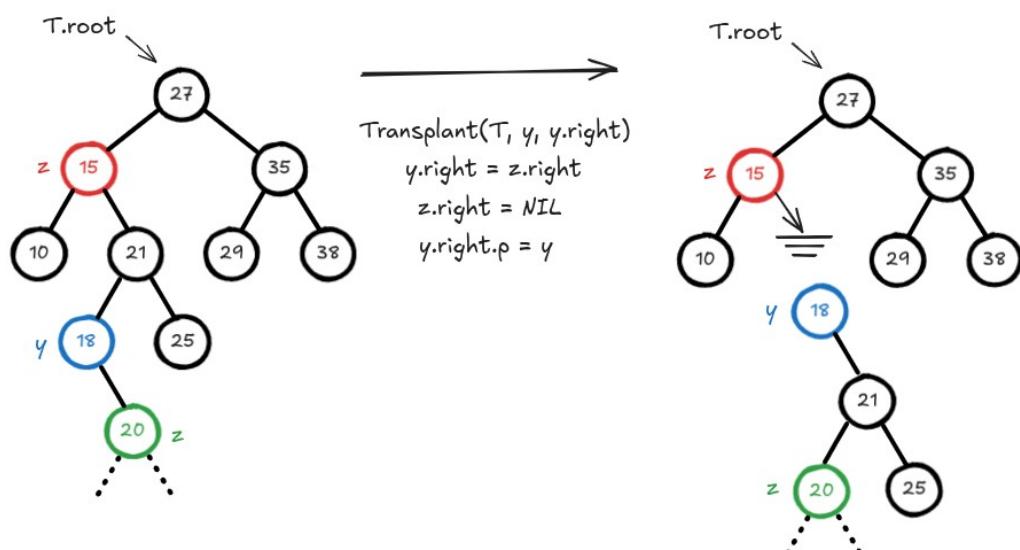
b1) y é o filho a direita de z

=> Basta transplantar y no lugar de z e fazer o filho a esquerda de z ser o filho a esquerda de y

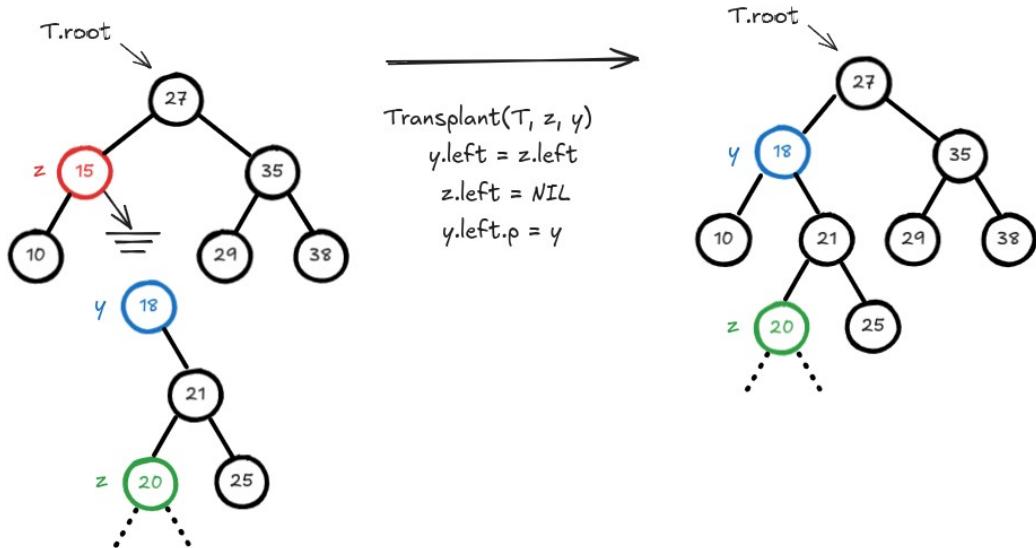


b2) y não é filho a direita de z (sucessor está lá embaixo na árvore T)

i) Substitua y por seu filho a direita (pois quando substituir z por y, tem que ter alguém no lugar do y)



ii) Substitua z por y



A seguir apresentaremos a função Tree_Delete(T, z) que remove o nó z de uma árvore binária de busca.

```

Tree_Delete( $T, z$ ) {
    if  $z.left == NIL$ 
        # substitui z pelo seu único filho a direita
        Transplant( $T, z, z.right$ )
    elseif  $z.right == NIL$ 
        # substitui z pelo seu único filho a esquerda
        Transplant( $T, z, z.left$ )
    else {
        # se entrou aqui é porque z tem 2 filhos
        # encontra a menor chave da subárvore a direita
         $y = Tree_Minimum(z.right)$     # pode ou não ser filho a direita
        if  $y \neq z.right$  {            # sucessor está lá embaixo da árvore
            # Substitui y por seu filho a direita e ajusta referências
            Transplant( $T, y, y.right$ )
             $y.right = z.right$ 
             $z.right = NIL$ 
             $y.right.p = y$ 
        }
        Transplant( $T, z, y$ )      # substitui z por y
         $y.left = z.left$           # passa filho a esquerda de z para y
         $z.left = NIL$              # ajusta referências
         $y.left.p = y$ 
    }
}
    
```

"Não diminua os outros para se sentir superior, melhore a si mesmo."
 (Autor anônimo)

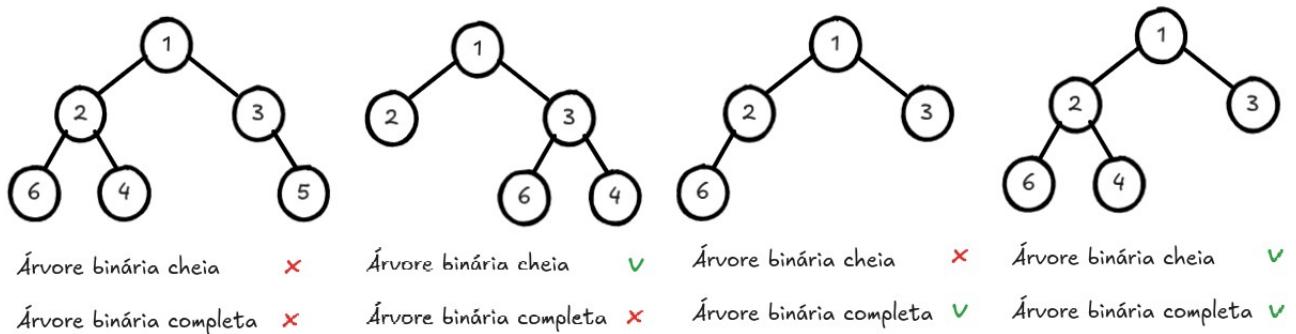
Heaps e filas de prioridades

Uma das maneiras eficientes de implementar uma fila de prioridades é através da criação de um heap binário, que é baseado na relação existente entre árvores binárias completas e arrays.

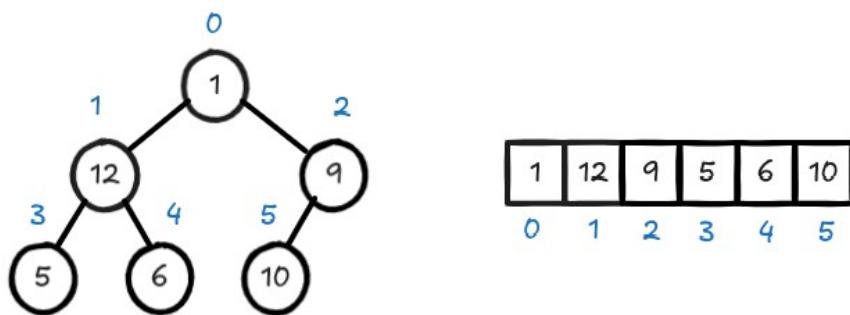
Relação entre arrays e árvores binárias completas

Uma árvore binária completa é uma árvore binária na qual todos os níveis são completamente preenchidos, exceto possivelmente o mais baixo, que é preenchido a partir da esquerda. Uma árvore binária completa é como uma árvore binária cheia, mas com duas diferenças principais

1. Todos os elementos folhas sem irmãos devem ser filhos a esquerda
2. O último elemento folha pode não ter um irmão direito, ou seja, uma árvore binária completa não precisa ser uma árvore binária cheia.



Note que para uma árvore binária ser cheia, não é necessário que todas as folhas estejam no mesmo nível, bastando que todo nó interno tenha exatamente dois filhos. Uma árvore binária perfeita é um tipo de árvore binária em que cada nó interno tem exatamente dois nós filhos e todos os nós folha estão no mesmo nível. Uma árvore binária completa possui a seguinte propriedade interessante que pode ser utilizada para encontrar os filhos e o pai de qualquer nó quando os elementos são dispostos em um vetor. Dado o elemento i do vetor, então o elemento de índice $2i + 1$ será seu filho a esquerda e o elemento de índice $2i + 2$ será o seu filho a direita. Além disso, o pai do elemento de índice i será o elemento de índice $(i - 1)/2$, onde // representa a divisão inteira. A figura a seguir ilustra essa interessante relação.



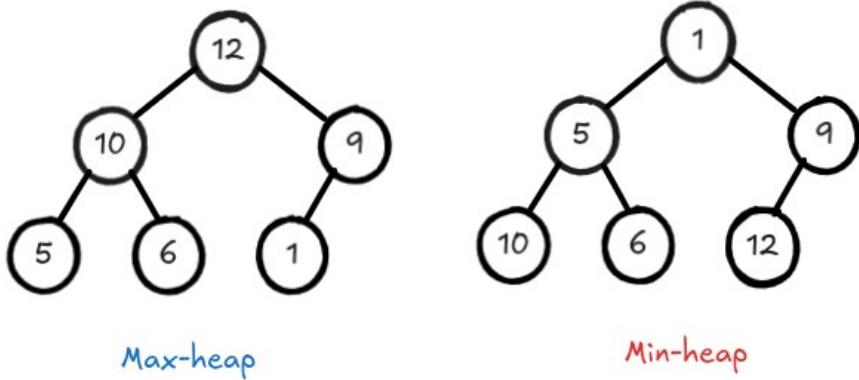
A estrutura de dados Heap

Um Heap é uma estrutura de dados especial baseada em árvore. Diz-se que uma árvore binária define uma estrutura de dados heap se:

1. É uma árvore binária completa.

2. Todos os nós na árvore seguem a propriedade de serem maiores que seus filhos, ou seja, o maior elemento está na raiz e seus filhos são menores que a raiz e assim por diante. Esse heap é chamado de max-heap. Se, em vez disso, todos os nós são menores que seus filhos, ele é chamado de min-heap.

A figura a seguir ilustra um max-heap e um min-heap.

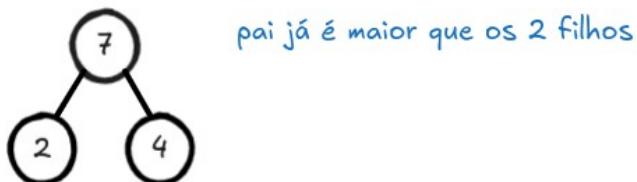


Uma pergunta natural que surge é: como transformar uma árvore binária completa arbitrária em um heap? Veremos que existe um processo de heapificação, definido pela função `heapify`.

Como heapificar uma árvore?

A partir de uma árvore binária completa, podemos modificá-la para se tornar um max-heap executando uma função chamada `heapify` em todos os elementos não-folha do heap. Para isso assume-se que temos a representação da árvore na forma vetorial. Como a função `heapify` usa recursão, pode ser um pouco complicada de entender. Então, vamos primeiro pensar em um exemplo didático de como você empilharia uma árvore com apenas três elementos. O exemplo a seguir mostra dois cenários - um em que a raiz é o maior elemento e não precisamos fazer nada. Este é o caso base da recursão, onde atingimos a condição de parada. E outro em que a raiz tem um elemento maior que o filho e precisamos trocá-los para manter o max-heap.

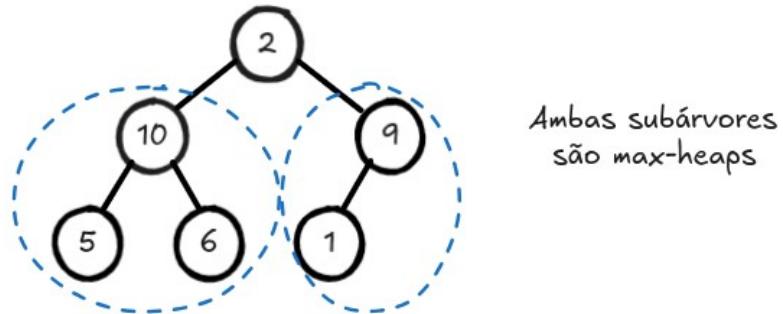
Cenário 1



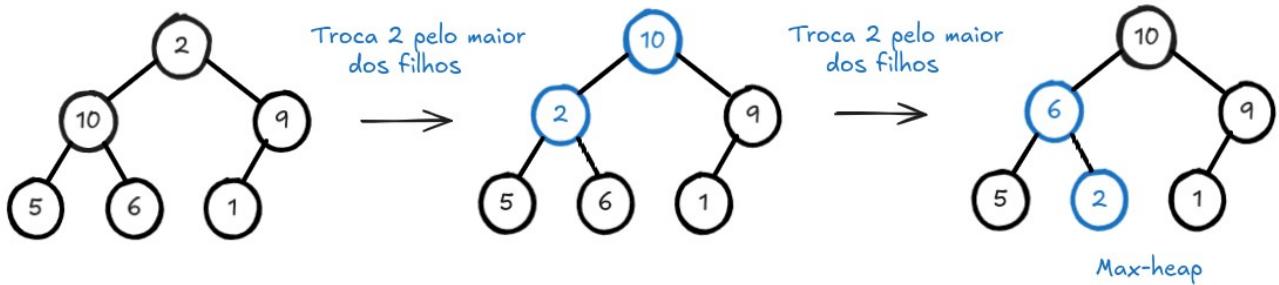
Cenário 2



Agora vamos pensar em outro cenário em que há mais de um nível no heap. Note que a árvore a seguir não é um max-heap por causa da raiz, mas todas as subárvores (a esquerda e a direita) são max-heaps.



Para manter a propriedade do max-heap para toda a árvore, temos que continuar empurrando o elemento 2 (raiz) para baixo até atingir sua posição correta na estrutura.



Assim, para manter a propriedade max-heap em uma árvore onde ambas as subárvores são max-heaps, precisamos executar heapify no elemento raiz repetidamente até que ele seja maior que seus filhos ou se torne um nó folha. A função a seguir ilustra o algoritmo.

```
heapify(L, n, i):
    # Encontra o maior entre o nó raiz i e os filhos
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and L[i] < L[l]
        largest = l
    if r < n and L[largest] < L[r]
        largest = r
    # Se nó raiz i não é maior, troca e continua heapify
    if largest != i {
        swap(L[i], L[largest])
        heapify(L, n, largest)
    }
}
```

Esta função funciona tanto para o caso base quanto para uma árvore de qualquer tamanho. Podemos, assim, mover o elemento raiz para a posição correta para manter o status do max-heap para qualquer tamanho de árvore, desde que as subárvores sejam max-heaps.

Podemos utilizar uma ideia similar a essa para implementar uma fila de prioridades. Para manter a propriedade de um max-heap, após cada inserção na fila, aplicamos uma operação heapify ao

contrário, para trazer o elemento de maior prioridade para a primeira posição da fila. Sendo assim, podemos definir um TAD Priority Queue Heap como descrito a seguir.

```

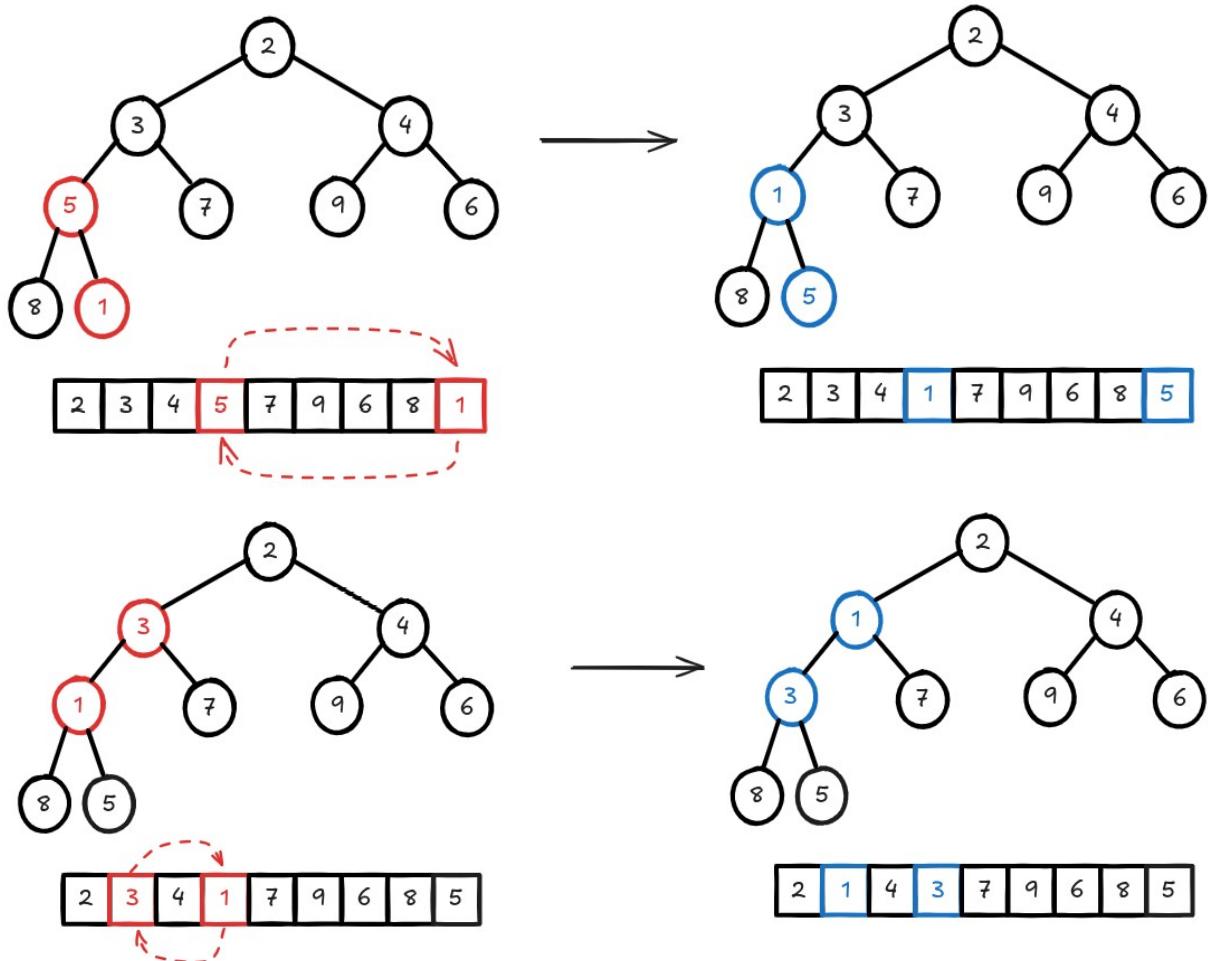
TAD Node
    int key
    int prior

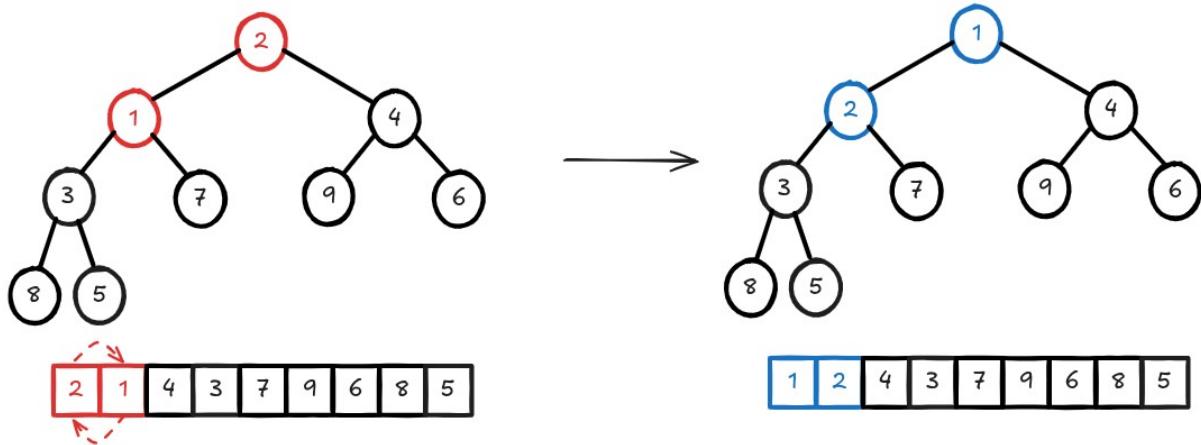
TAD PriorityQueue
    Node array[0..n-1] data
    int tail

init(PQ) {
    PQ.tail = -1      # incrementa tail e depois insere
}

# Sobe o menor para raiz: versão iterativa do heapify (bottom-up)
shift_up(PQ) {
    son = PQ.tail
    son_p = PQ.data[son].prior
    parent = (son - 1)//2
    parent_p = PQ.data[parent].prior
    while son > 0 and son_p < parent_p {
        swap(PQ.data[son], PQ.data[parent])
        son = parent
        parent = (son - 1)//2
    }
}

```

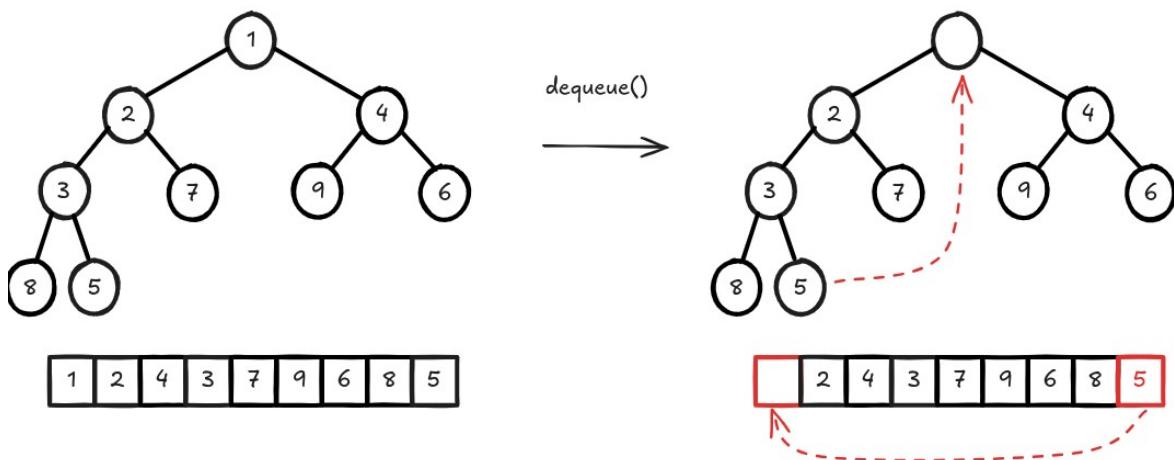


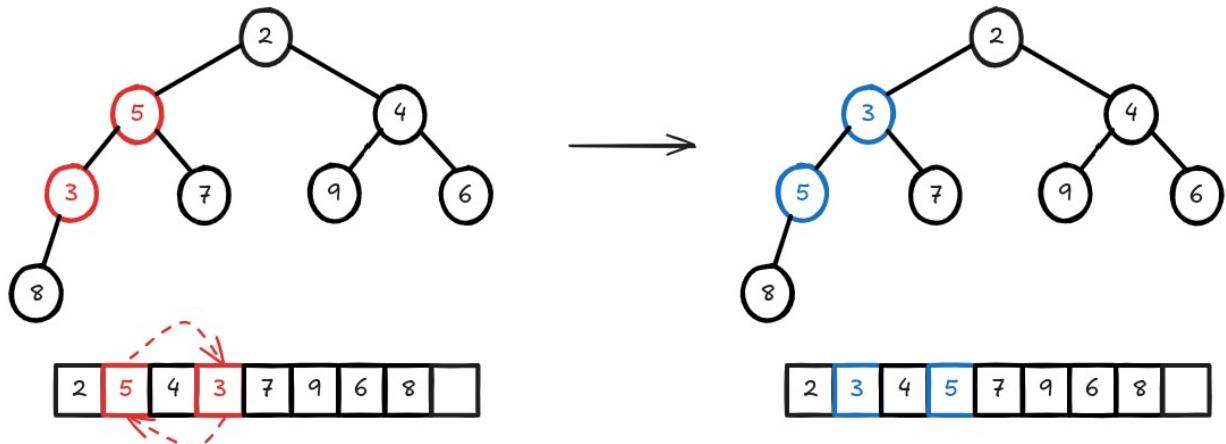
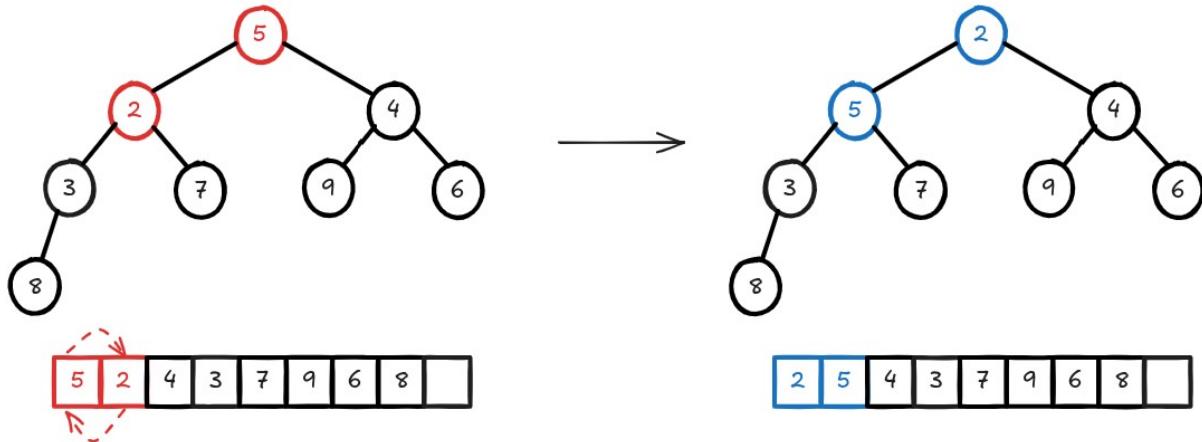


Note que após todo o processo de enqueue() e shift_up(), temos um min-heap novamente.

```
# Adiciona chave com prioridade p no final da fila e realiza um shift_up
enqueue(PQ, key, p) {
    if PQ.tail == n - 1
        error('overflow')
    else {
        PQ.tail += 1
        PQ.data[PQ.tail].key = key
        PQ.data[PQ.tail].priority = p
        shift_up(PQ)
    }
}

shift_down(PQ, i) {
    parent = i                                # i = 0 → raiz da árvore
    parent_p = PQ.data[parent].prior
    left = 2*parent + 1
    right = 2*parent + 2
    smallest = parent
    if left < n and PQ.data[parent].prior > PQ.data[left].prior
        smallest = left
    if right < n and PQ.data[smallest].prior > PQ.data[right].prior
        smallest = right
    if smallest != parent {
        swap(PQ.data[parent], PQ.data[smallest])
        shift_down(PQ, smallest)
    }
}
```



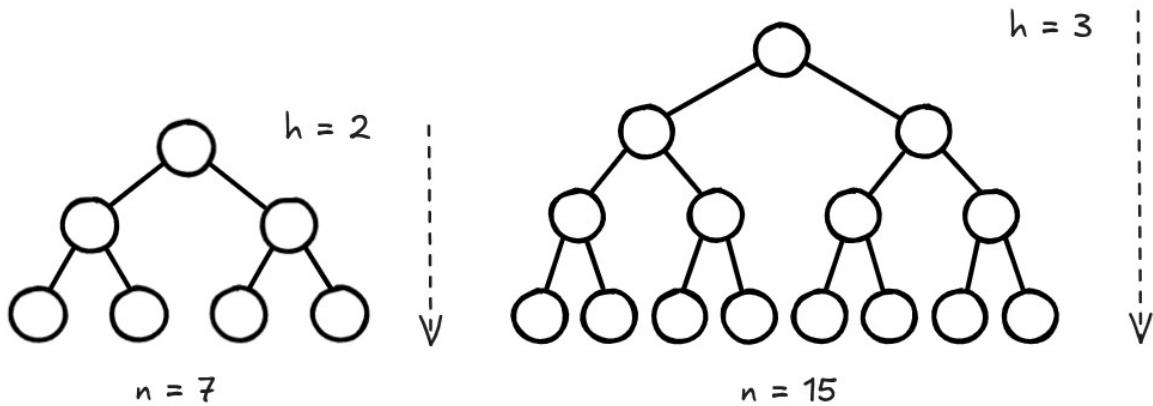


Note que após todo o processo de dequeue() e shift_down(), temos um min-heap novamente.

```
# Remove chave de menor p (menor p = maior prioridade)
dequeue(PQ) {
    if PQ.tail == 0
        error('underflow')
    else {
        key = PQ.data[0]                      # Remove a raiz
        swap(PQ.data[0], PQ.data[PQ.tail])    # Troca último com raiz
        PQ.tail -= 1
        shift_down(PQ, 0)                    # Desce nova raiz até sua posição correta
    }
    return key
}
```

Análise da complexidade

Iremos considerar inicialmente a análise do pior caso. Primeiramente, note que em uma árvore binária completa, a cada novo nível criado, o número de elementos armazenados por ela dobra. Repare que se a altura da árvore é $d = 2$ (raiz é nível 0), temos que o número máximo de elementos armazenados na árvore é $n = 2^0 + 2^1 + 2^2 = 1 + 2 + 4 = 7 = 2^{d+1} - 1$. Ao passarmos para a altura $d = 3$, temos que o número máximo de elementos é $n = 2^4 - 1 = 15$. A figura a seguir ilustra essa ideia.



Assim, no caso de uma árvore binária de altura genérica k , o número máximo de elementos armazenados é:

$$n = 2^0 + 2^1 + 2^2 + \dots + 2^k = \sum_{i=0}^k 2^i = 2^{k+1} - 1$$

No pior caso, as funções `shift_up` e `shift_down` sobe/desce toda a altura da árvore. A altura máxima da árvore em função de n é dada por:

$$2^{k+1} = n + 1 \rightarrow \log_2(n + 1) = k + 1 \rightarrow k = \log_2(n + 1) - 1$$

ou seja, $k = O(\log_2 n)$. Sendo assim, ela trocará o elemento de um nó pai com um nó filho no máximo d vezes. Portanto, a complexidade da função `heapify()` e das funções `enqueue()` e `dequeue()` da fila de prioridades é igual a $O(\log n)$.

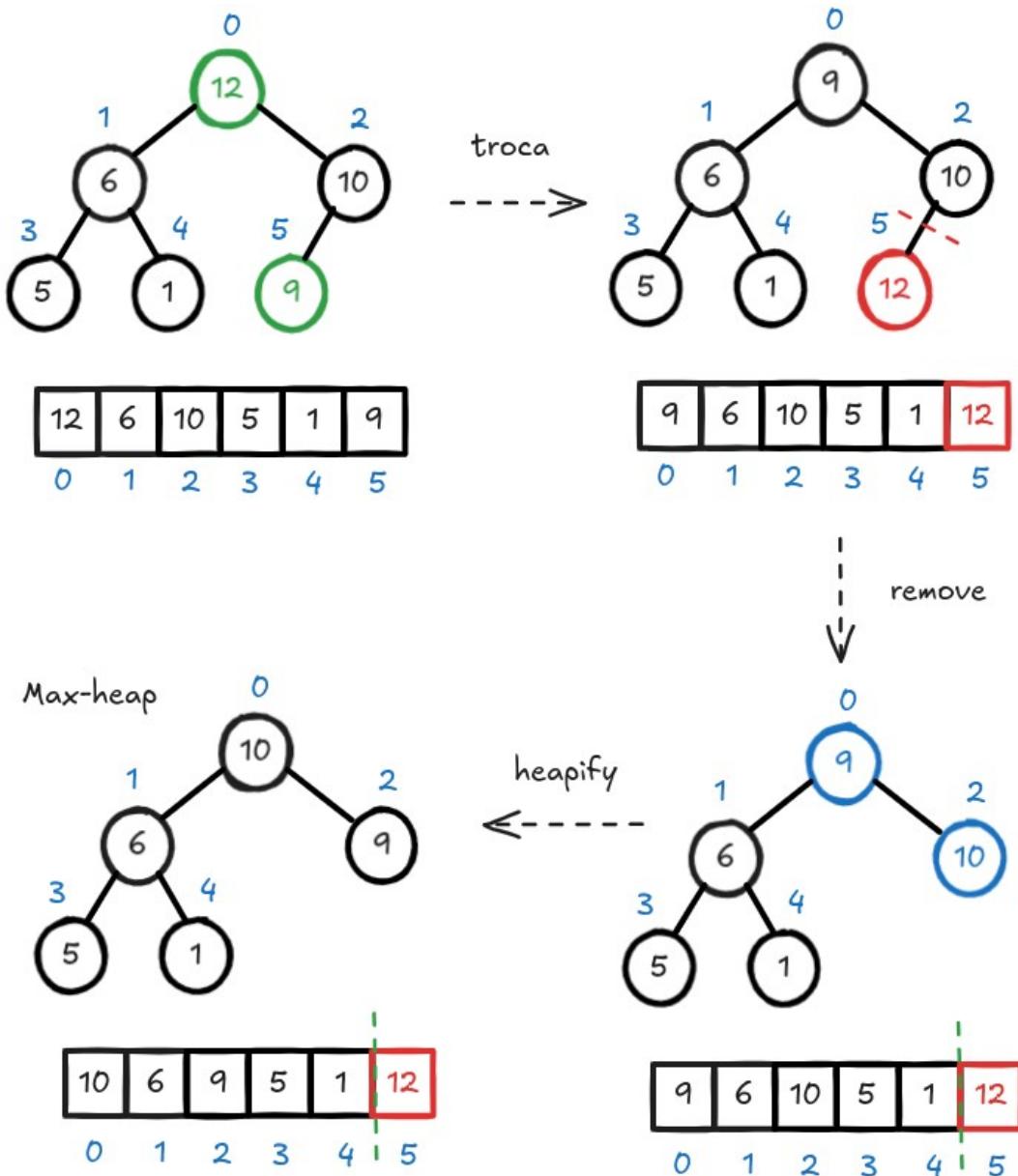
"If you can't yet do great things, do small things in a great way."
-- Napoleon Hill

O algoritmo Heapsort

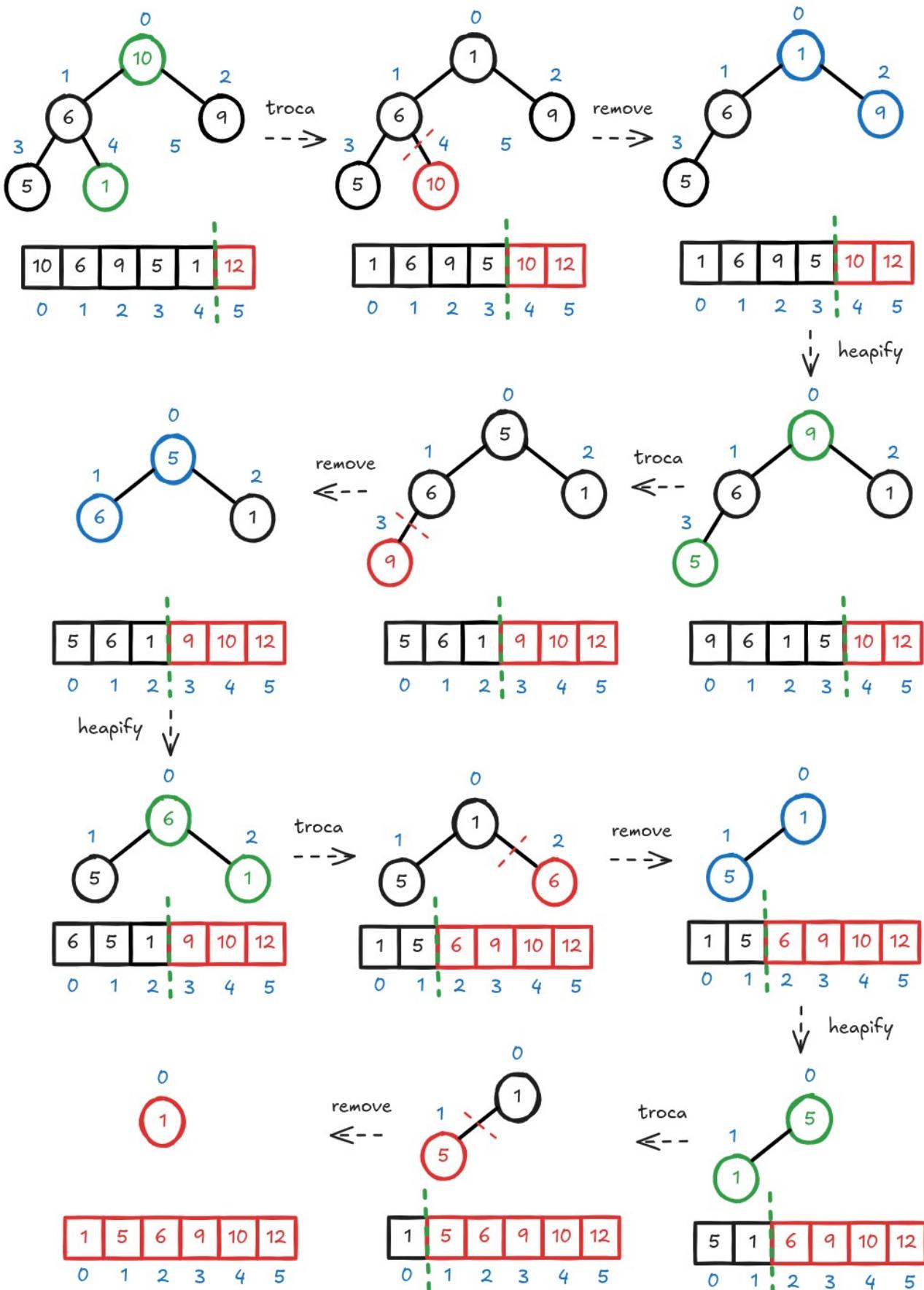
O algoritmo Heapsort pode ser entendido como uma otimização do algoritmo Insertionsort. Em resumo, ele é baseado nos seguintes passos:

1. Como a árvore em questão, na forma vetorial, satisfaz a propriedade max-heap, o maior elemento está sempre localizado na raiz (índice 0).
2. **Swap:** Coloque o elemento raiz e na última posição do vetor, trocando sua posição com o último elemento.
3. **Remove:** Reduza o tamanho do max-heap em uma unidade.
4. **Heapify:** Aplique a função heapify para trazer o maior elemento para a raiz da árvore.
5. Repita os passos de 2 a 4 até que o vetor esteja completamente ordenado.

As figuras a seguir ilustram o funcionamento do algoritmo em um exemplo didático.



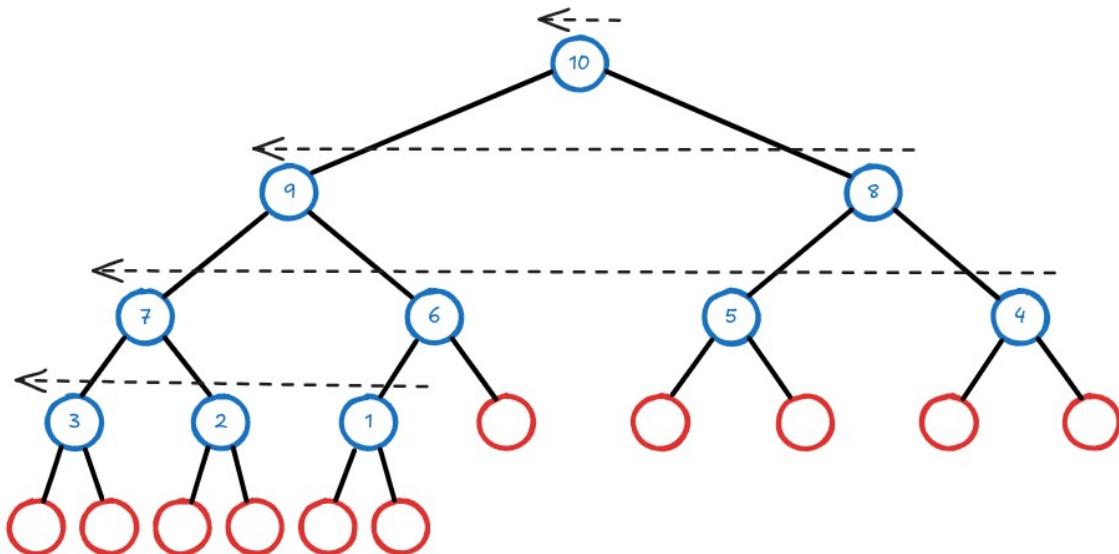
Considerando o max-heap criado anteriormente, iremos aplicar o algoritmo Heapsort no vetor inicial. As figuras a seguir ilustram o passo a passo do método.



A função a seguir ilustra o pseudocódigo do algoritmo Heapsort, que faz uso das funções auxiliares descritas anteriormente.

```
heapsort(L, n) {
    # Constrói max heap
    for i = n//2-1 downto 0
        heapify(L, n, i)
    for i = n-1 downto 1 {
        # Troca
        swap(L[i], L[0])
        # Heapifica o elemento raiz
        heapify(L, i, 0)
    }
}
```

A complexidade da função `heapify()` é $O(\log n)$. Com base nisso, agora podemos computar a complexidade necessária para criar o max-heap. Lembre que para isso devemos percorrer os nós que não são folha, como ilustra a figura a seguir.



Vimos que uma árvore de n nós possui no máximo $n/2$ nós intermediários (não folhas), de modo que a complexidade da construção do max-heap é $O(n/2 \log_2 n) = O(n \log_2 n)$. Portanto, a complexidade de pior caso do Heapsort é $O(n \log_2 n)$, de modo que ele pode ser considerado um algoritmo de ordenação eficiente.

Pode-se mostrar que no melhor caso, a complexidade do algoritmo Heapsort também é $O(n \log_2 n)$, porém os cálculos matemáticos não são de fácil compreensão. Para os leitores interessados, recomenda-se a seguinte referência:

Bollobás, B., Fenner, T. I., Frieze, A. M. On the Best Case of Heapsort, Journal of Algorithms, v. 20, pp. 205-217, 1996.

Disponível em: <https://www.math.cmu.edu/~af1p/Texfiles/Best.pdf>

Dessa forma, se tanto o pior caso como o melhor caso possuem complexidade log-linear, então é evidente que o mesmo deve valer para o caso médio. Em comparação com o Quicksort, sua grande vantagem é ser melhor no pior caso e ocupar menos memória. Por fim, pode-se mostrar que o algoritmo Heapsort não é estável.

O problema do casamento estável e o algoritmo de Gale-Shapley

Área de interface entre algoritmos e teoria dos jogos.

Objetivo: Dado um conjunto de n homens e outro conjunto de n mulheres, cada um com uma lista de preferências, encontrar dentre todos os casamentos (matchings) possíveis, o mais estável (no sentido de evitar futuros rompimentos)

Aplicações reais em diversas áreas:

- Sistemas de recomendação (candidatos/vagas)
- Alocação de alunos a universidades
- Alocação de residentes a hospitais
- Sistema para doação de órgãos

Prêmio Nobel em 2012 na área de ciências econômicas (Loyd Shapley) por contribuições e profundo impacto no estudo, caracterização e regulamentação de sistemas econômicos (mercados) não controlados pelo capital. (a ideia básica é que nas situações em que o algoritmo Gale-Shapley se aplica não é possível barganhar para obtenção de privilégios e vantagens impostas por escolher primeiro que os outros).

O problema do casamento estável

É essencial definirmos o que é um casamento estável, ou stable matching.

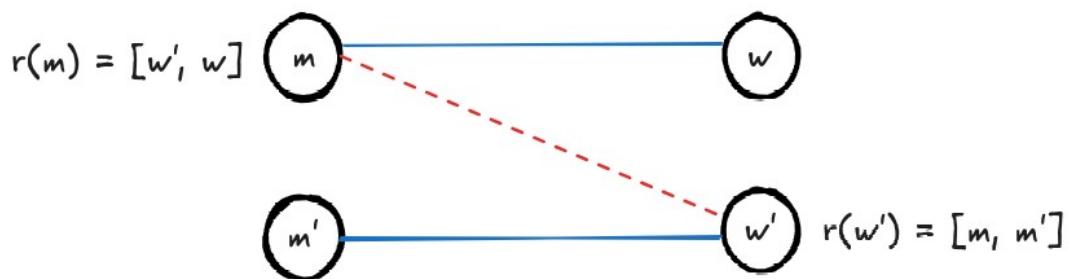
O conceito de estabilidade

- Listas de preferências: cada homem deve ter uma lista em que rankeia todas as mulheres e vice-versa.

$$\begin{aligned} \forall m \in M \quad \exists r(m) = [w_1, w_2, \dots, w_n] & \quad (\text{M: men}) \\ \forall w \in W \quad \exists r(w) = [m_1, m_2, \dots, m_n] & \quad (\text{W: women}) \end{aligned}$$

Definiremos um predicado ternário $P(m, w, w')$ para denotar que m prefere w a w' , ou seja, w vem antes de w' na lista $r(m)$

Def (Estabilidade): Seja S o emparelhamento a seguir:



S é estável?

$\exists (m, w) \in S \wedge \exists (m', w') \in S$ tal que $P(m, w', w) \wedge P(w', m, m')$, portanto o par (m, w') define uma instabilidade em S pois ele não existe em S mas nada impede que venha existir. Ambos se desejam mas não estão juntos. Portanto, S não é estável.

Nada impede que m ou w' rompa o relacionamento atual de maneira unilateral na tentativa de melhorar seu ganho, ou seja, nada impede que ocorra uma ruptura e após isso o par (m, w') espontaneamente surja. Na teoria dos jogos dizemos que tanto m quanto w' podem romper unilateralmente na expectativa de maximizar o ganho e portanto a configuração atual não satisfaz o equilíbrio de Nash.

Obs: Note que $(m, w') \notin S$ (uma instabilidade nunca é um par que existe, mas sim um par que nada impede que ele possa aparecer no futuro)

Def: S é estável se S é perfeito e $\nexists(m, w)$ que provoque instabilidade

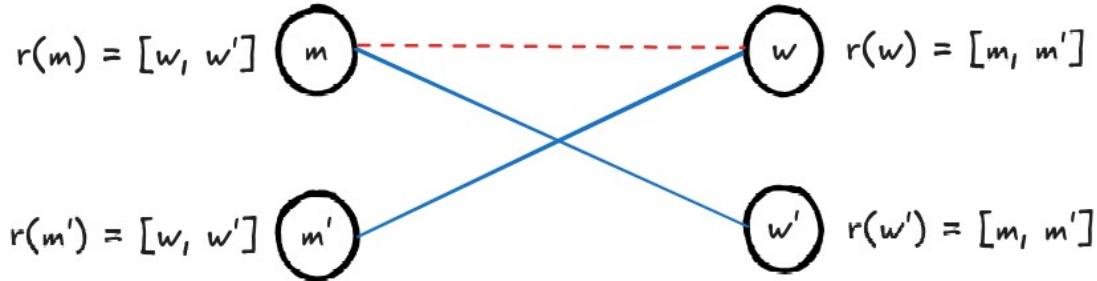
Ex: S é estável?



Sabemos que $P(m, w, w')$ e $P(w, m, m')$ e como o par (m, w) existe em S , eles não podem tentar nada melhor. Suponha que m' rompa unilateralmente com w' . Nesse caso, w vendo que m' está livre não irá largar de m para ficar com ele, pois m é o primeiro da lista. Da mesma forma, suponha que w' rompa unilateralmente com m' . O homem m não vai romper com w para ficar com w' pois prefere a parceira atual. Dessa forma, não há instabilidade que possa surgir e portanto S é estável.

Obs: Estabilidade não é agradar a todos com primeira opção!

E se S for assim?

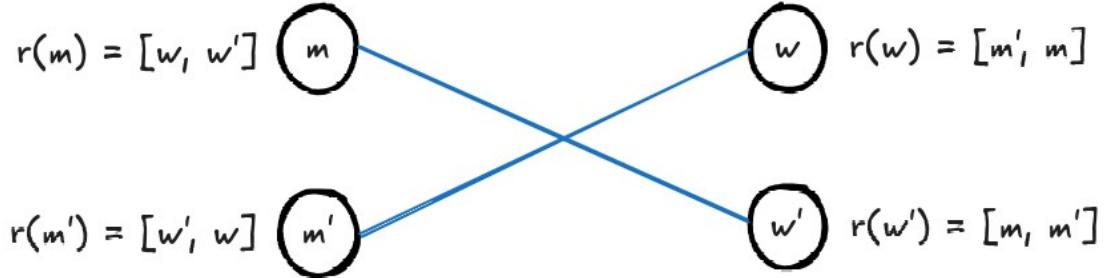


S é instável (tende a voltar para a configuração de cima). O que impede m e w de ficarem juntos no futuro? Instabilidade

Ex:



Note que para os homens está OK (perfeito), mas para mulheres está trocado. Isso significa instabilidade? Não, pois S é estável. Veja que a mulher w' deseja melhorar e pode romper unilateralmente com m' . Porém, m não se interessa por w' e a restante (m, w') nunca irá se formar. O mesmo vale para o caso de w romper unilateralmente com m . A aresta (m', w) nunca irá se formar pois m' não quer w . E o que dizer desse outro S ?



S também é estável. Porém agora satisfaz plenamente as mulheres. Da mesma forma que o caso anterior, não há como as arestas (m, w) ou (m', w') aparecerem e portanto não há instabilidade. Conceito: conjunto dominante – é o conjunto que propõe a ligação (casamento), quem toma a iniciativa

Questionamentos:

- 1) $\exists S$ estável para $\forall r(m_i)$ e $\forall r(w_i)$? Ou seja, é garantido que vai haver um S estável, independente de quais forem as listas de preferências? Pode-se mostrar que sim (teoria dos jogos)
- 2) Dados $r(m_i)$ e $r(w_i)$ $\forall i$, como construir S estável? Algoritmo de Gale-Shapley
- 3) S é único dado $r(m_i)$ e $r(w_i)$ $\forall i$ e um conjunto dominante? SIM

Essa é a grande propriedade do algoritmo de Gale-Shapley. O algoritmo GS não fornece vantagem para qual nó será o primeiro a escolher, não se pode barganhar. Independente da ordem que o conjunto dominante faz as escolhas, sempre resulta no mesmo emparelhamento. Do ponto de vista da economia, é o que proporciona a regulação de mercados.

Algoritmo de Gale-Shapley (conj. M dominante)

Enquanto $\exists m_i$ livre (que ainda não tentou $\forall w_i \in W$)
 Seja m_i esse homem
 Seja w_i a mulher mais bem rankeada em $r(m_i)$
 # (para a qual m_i ainda não propôs!)
 Se w_i está livre
 (m_i, w_i) “engaged” (adiciona em S temporariamente)
 Senão
 # significa que $\exists (\bar{m}, w_i) \in S$
 Se $P(w_i, \bar{m}, m_i)$
 m_i continua livre (vai continuar tentando)
 Senão
 # significa que $P(w_i, m_i, \bar{m})$
 (m_i, w_i) “engaged”
 \bar{m} fica livre (vai tentar outras parceiras)

Propriedades do algoritmo:

- i) Algoritmo guloso: cada homem sempre propõe para sua primeira opção
- ii) Ponto de vista de $m_i \in M$: a cada troca, sequencia de parceiras piora
Para que está no conjunto dominante, sempre que houver uma troca será para pior
- iii) Ponto de vista de $w_i \in W$: a cada troca, sequencia de parceiros melhora
Para quem está no conjunto passivo, sempre que houver uma troca será para melhorar

Análise da complexidade

Note que como o número de elementos no conjunto de homens é igual ao número de elementos no conjunto das mulheres, podemos afirmar:

a) Há n homens livres no início e todos devem terminar com um par.

b) No pior caso, cada homem m deve propor para n mulheres distintas.

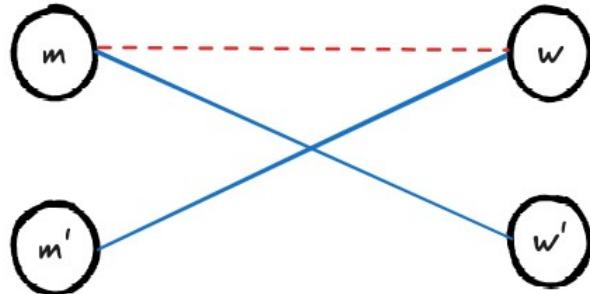
Portanto, a complexidade do algoritmo de Gale-Shapley é quadrática, ou seja, $O(n^2)$.

O resultado a seguir garante que o algoritmo de Gale-Shapley retorna um emparelhamento estável.

Teorema: O emparelhamento S retornado pelo algoritmo de Gale-Shapley é estável.

Prova: por contradição

1. Suponha que S retornado pelo algoritmo GS não é estável. Então, existe uma instabilidade (m, w)



2. Isso implica que $P(m, w, w') \wedge P(w, m, m')$

3. Mas se m finalizou o algoritmo com w' , significa que a última proposta de m foi para w'

4. Então, m propôs a w antes de w'

Isso implica que na lista de w , m' deve vir antes de m , ou seja, $P(w, m', m)$
Caso contrário, w teria ficado com m'

5. Assim, chegamos em $P(w, m, m') \wedge P(w, m', m)$, que é falso sempre pois se um dos termos é verdade o outro não pode ser verdade também. Em outras palavras, não é possível para uma mulher m preferir m a m' e m' a m simultaneamente.

6. Portanto, a suposição inicial não é válida e a instabilidade não pode existir em S .

Dessa forma, não existe S instável gerado pelo algoritmo GS.

Def: valid partner (vp)

Dizemos que w é $vp(m)$ se existe S estável tal que $(m, w) \in S$

Def: best valid partner (bvp)

Dizemos que w é $bvp(m)$ se w é $vp(m)$ e $\nexists w' \neq w$ que seja $vp(m)$ tal que $P(m, w', w)$

Teorema: Toda execução do algoritmo Gale-Shapley (com M dominante) resulta no conjunto $S^* = \{(m, bvp(m)), \forall m \in M\}$

Esse resultado é de fundamental importância pois garante que não importa a ordem de escolha dos homens, o resultado final será sempre o mesmo. Esse fato do ponto de vista da teoria dos jogos significa que nesses tipos de sistemas não existe barganha, ou seja, não é possível barganhar e comprar o direito de escolher primeiro, pois o resultado será sempre o mesmo.

Prova: por contradição

1. Suponha que S é um emparelhamento estável gerado pelo algoritmo GS em que m está emparelhado com alguém que não é sua $bvp(m) = w$

2. Então, m deve ter sido rejeitado por $w = bvp(m)$, uma vez que ele começa a propor em ordem decrescente de preferência

3. Seja m o primeiro homem para o qual isso ocorre (ser rejeitado). Ele pode ter sido rejeitado por:

a) m propôs a w , mas foi negado pois ela preferiu ficar com o atual m'

b) w rompeu com m por uma proposta melhor de um homem m'

Em qualquer caso, é certo que w está emparelhada a m' , a quem ela prefere: $P(w, m', m)$

4. Pela definição de $bvp(m)$ existe um outro casamento estável S' que contém o par (m, w) .

5. A pergunta é: Com quem m' estaria casado em S' ?

Com uma mulher $w' \neq w = bvp(m)$

6. Note que m' não pode ter sido rejeitado por ninguém quando se engajou com w em S , pois m foi o primeiro a ser rejeitado (de acordo com o passo 3). Como m' propõe em ordem decrescente de preferência, então m' prefere w a w' (senão em S ele teria escolhido w' e não w). Logo, temos que $P(m', w, w')$.

7. Assim, como temos $P(w, m', m) \wedge P(m', w, w')$ e o par $(m', w) \notin S'$ (pois em S' m está com w), temos que (m', w) define uma instabilidade em S' (contradição pois S' era suposto estável)

Portanto, S deve obrigatoriamente associar todo homem m a sua $bvp(m)$.

Ex: Suponha que num site de relacionamentos existam as seguintes listas de preferências entre um grupo de rapazes e garotas.

Man	1	2	3	4	5
A1	B2	A2	D2	E2	C2
B1	D2	B2	A2	C2	E2
C1	B2	E2	C2	D2	A2
D1	A2	D2	C2	B2	E2
E1	B2	D2	A2	E2	C2

Woman	1	2	3	4	5
A2	E1	A1	B1	D1	C1
B2	C1	B1	D1	A1	E1
C2	B1	C1	D1	E1	A1
D2	A1	E1	D1	C1	B1
E2	D1	B1	E1	C1	A1

Utilizando o algoritmo de Gale-Shapley encontre:

a) um emparelhamento estável M sendo os rapazes o conjunto dominante. Mostre a sequencia de propostas até o emparelhamento estável.

b) um emparelhamento estável M' sendo as garotas o conjunto dominante. Mostre a sequencia de propostas até o emparelhamento estável.

i	m	w	engaged
1	A1	B2	yes
2	B1	D2	yes
3	C1	B2	yes
4	A1	A2	yes
5	D1	A2	no
6	D1	D2	yes
7	B1	B2	no
8	B1	A2	no
9	B1	C2	yes
10	E1	B2	no
11	E1	D2	yes
12	D1	C2	no
13	D1	B2	no
14	D1	E2	yes

$$S^{(0)} = \emptyset$$

$$S^{(1)} = \{A1, B2\}$$

$$S^{(2)} = \{A1, B2\}, (B1, D2\}$$

$S^{(3)} = \{C1, B2\}, (B1, D2)\}$
 $S^{(4)} = \{C1, B2\}, (B1, D2), (A1, A2)\}$
 $S^{(5)} = \{C1, B2\}, (B1, D2), (A1, A2)\}$
 $S^{(6)} = \{C1, B2\}, (D1, D2), (A1, A2)\}$
 $S^{(7)} = \{C1, B2\}, (D1, D2), (A1, A2)\}$
 $S^{(8)} = \{C1, B2\}, (D1, D2), (A1, A2)\}$
 $S^{(9)} = \{C1, B2\}, (D1, D2), (A1, A2), (B1, C2)\}$
 $S^{(10)} = \{C1, B2\}, (D1, D2), (A1, A2), (B1, C2)\}$
 $S^{(11)} = \{C1, B2\}, (E1, D2), (A1, A2), (B1, C2)\}$
 $S^{(12)} = \{C1, B2\}, (E1, D2), (A1, A2), (B1, C2)\}$
 $S^{(13)} = \{C1, B2\}, (E1, D2), (A1, A2), (B1, C2)\}$
 $S^{(14)} = \{C1, B2\}, (E1, D2), (A1, A2), (B1, C2), (D1, E2)\}$

b) Solução

i	w	m	engaged
1	A2	E1	yes
2	B2	C1	yes
3	C2	B1	yes
4	D2	A1	yes
5	E2	D1	yes

$S^{(0)} = \emptyset$
 $S^{(1)} = \{A2, E1\}$
 $S^{(2)} = \{A2, E1\}, (B2, C1)$
 $S^{(3)} = \{A2, E1\}, (B2, C1), (C2, B1)$
 $S^{(4)} = \{A2, E1\}, (B2, C1), (C2, B1), (D2, A1)$
 $S^{(5)} = \{A2, E1\}, (B2, C1), (C2, B1), (D2, A1), (E2, D1)$

"Solutions are not found by pointing fingers; they are reached by extending hands."
-- Aysha Taryam

Bibliografia

- W. CELES, R. CERQUEIRA, J. RANGEL. Introdução a Estrutura de Dados, 2^a edição, Elsevier, 2016.
- CORMEN, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 4^a edição, The MIT Press, 2022.
- KARUMANCHI, N. Data Structures and Algorithms Made Easy, CareerMonk Publications, 2010.
- LAMBERT, K. A. Fundamentals of Python: Data Structures, 2^a edição, Cengage Learning, 2019.
- ALTHOFF, C. The Self-Taught Computer Scientist: The Beginners Guide to Data Structures & Algorithms, Wiley, 2022.
- BHARGAVA, A. Y. Grokking Algorithms: A Illustrated Guide for Programmers and Other Curious People, Manning Publications, 2016.
- FELICE, M. C. S. Material e aulas das disciplinas Algoritmos e Estruturas de Dados I e II. Disponíveis em: <http://www.aloc.ufscar.br/felice/#teaching>
- KLEINBERG, J.; TARDOS, E. Algorithmic Design, Addison Wesley, 2005.
- U. AGARWAL, Algorithms Design and Analysis, Dhanpat Rai Publications, 2012.
- R. SEDGEWICK, K. WAYNE. Algorithms, 4th. ed., Addison-Wesley, 2011.
- LEVITIN, A. Introduction to the Design and Analysis of Algorithms, 3^a edição, Pearson, 2012.
- SKIENA, S. S. The Algorithm Design Manual, 2^a edição, Springer, 2008.
- S. DASGUPTA, C.H. PAPADIMITRIOU, U.V. VAZIRANI. Algorithms, McGraw-Hill, 2007.
- N. ZIVIANI. Projetos de algoritmos: com implementações em Pascal e C. 3. ed. rev. e ampl. São Paulo: Cengage Learning, 2012.
- R. SEDGEWICK. Algorithms in C, Parts 1-4: fundamentals, data structures, sorting, searching. 3rd. ed., Addison-Wesley, 1998.
- Geeks for Geeks – A computer science portal for geeks. <https://www.geeksforgeeks.org/>

Sobre o autor

Alexandre L. M. Levada é bacharel em Ciências da Computação pela Universidade Estadual Paulista “Júlio de Mesquita Filho” (UNESP), mestre em Ciências da Computação pela Universidade Federal de São Carlos (UFSCar) e doutor em Física Computacional pela Universidade de São Paulo (USP). Atualmente é professor associado no Departamento de Computação da Universidade Federal de São Carlos e seus principais interesses em pesquisa são reconhecimento de padrões e processamento de sinais e imagens.

*"Thousands of candles can be lit from a single candle, and the life of the candle will not be shortened.
Happiness never decreases by being shared."*
-- Buddha