

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/393884890>

Manual Completo de Algoritmos e Estruturas de Dados

Preprint · July 2025

DOI: 10.13140/RG.2.2.11875.87847

CITATIONS

0

READS

10,443

1 author:



Alexandre L M Levada
Federal University of São Carlos

157 PUBLICATIONS 492 CITATIONS

[SEE PROFILE](#)



Departamento de
Computação - **UFSCar**



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

Manual Completo de Algoritmos e Estruturas de Dados

Apostila sobre complexidade de algoritmos, métodos de ordenação,
estrutura de dados e algoritmos em grafos

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

Sumário

Prefácio.....	3
Complexidade de algoritmos.....	5
Recursão.....	13
Algoritmos de ordenação.....	23
Bubblesort.....	23
Insertionsort.....	27
Selectionsort.....	29
Shellsort.....	31
Quicksort.....	36
Mergesort (ordenação por intercalação).....	42
Limitante Inferior para Ordenação Baseada em Comparações.....	47
Ordenação Não Baseada em Comparações.....	48
Countingsort.....	49
Radixsort.....	50
Bucketsort.....	51
Tipos Abstratos de Dados (TAD's).....	56
Estruturas de dados lineares estáticas.....	64
Pilhas (LIFO).....	64
Filas (FIFO).....	71
Deque: Double-Ended Queue (Fila de duas extremidades).....	77
Fila de prioridades (Priority Queue).....	82
Estruturas de dados dinâmicas.....	86
Listas encadeadas.....	87
Pilhas com estruturas dinâmicas.....	91
Filas com estruturas dinâmicas.....	92
Listas encadeadas ordenadas.....	94
Listas duplamente encadeadas.....	95
Listas duplamente encadeadas com sentinelas.....	97
Aplicações: matriz esparsa com arrays e listas encadeadas.....	100
Aplicações: listas ortogonais e matrizes dinâmicas.....	103
Árvore Binárias.....	105
Árvores binárias de busca.....	110
Estruturas de Dados: Árvores AVL (Balanceadas).....	119
Estruturas de dados: Heaps binários.....	131
O algoritmo Heapsort.....	137
Estruturas de Dados: Árvores de Busca Digitais (Tries).....	143
Estruturas de Dados: Árvores Rubro-Negras.....	148
Estruturas de Dados: Skip Lists.....	161
Estruturas de Dados: Tabelas de Espalhamento (Hash Tables).....	169
Estruturas de Dados: O Filtro de Bloom.....	190
Grafos: Fundamentos Básicos.....	195
Grafos: Busca em Grafos (BFS e DFS).....	200
Busca em Largura (Breadth-First Search – BFS).....	202
Busca em Profundidade (Depth-First Search – DFS).....	205
Grafos: Caminhos Mínimos e o Algoritmo de Dijkstra.....	211
Grafos: Árvores geradoras mínimas.....	225
O algoritmo de Kruskal.....	228
Algoritmo de Prim.....	233
Grafos: Fluxo em redes e o algoritmo de Ford-Fulkerson.....	239
O Problema do Caixeiro Viajante (<i>Travelling Salesman Problem – TSP</i>).....	253
Bibliografia.....	269

Prefácio

A eficiência computacional é a espinha dorsal da tecnologia moderna. Em um mundo onde os volumes de dados crescem exponencialmente e a demanda por soluções rápidas e eficazes é constante, o domínio de algoritmos e estruturas de dados se torna essencial para qualquer profissional da computação. Esta apostila foi elaborada com o objetivo de oferecer um guia didático e abrangente, cobrindo desde conceitos básicos até tópicos mais avançados, sempre com foco na compreensão prática e teórica.

Ao longo dos capítulos, o leitor encontrará explicações detalhadas, exemplos ilustrativos e análises de complexidade que irão fundamentar uma formação sólida em programação eficiente. O conteúdo está organizado de forma progressiva, permitindo uma construção gradual do conhecimento.

Conteúdo abordado nesta apostila:

- **Tópico 1 – Complexidade de Algoritmos:** apresenta as bases para medir e comparar a eficiência de algoritmos, introduzindo conceitos fundamentais como a notação Big-O, análise de tempo e espaço, e modelos clássicos como o RAM.
- **Tópico 2 – Recursão:** explora o paradigma da recursão, abordando sua aplicação em problemas clássicos como fatorial, Fibonacci, Torres de Hanói e técnicas de otimização com análise de complexidade.
- **Tópico 3 – Algoritmos de Ordenação:** discute métodos de ordenação básicos e avançados, como Bubblesort, Insertionsort, Selectionsort, Quicksort, Mergesort, além de técnicas não baseadas em comparação, como Countingsort, Radixsort e Bucketsort, incluindo a análise de seus limites inferiores.
- **Tópico 4 – Tipos Abstratos de Dados (TADs):** introduz o conceito de abstração em estruturas de dados, preparando a base para a construção de estruturas robustas e eficientes.
- **Tópico 5 – Estruturas Lineares Estáticas:** aborda a implementação e uso de pilhas, filas, deque e filas de prioridade, com exemplos práticos em arrays.
- **Tópico 6 – Estruturas Lineares Dinâmicas:** cobre listas encadeadas, listas duplamente encadeadas, listas com sentinelas e aplicações como matrizes esparsas e listas ortogonais.
- **Tópico 7 – Árvores Binárias:** introduz a estrutura de árvores binárias, detalhando árvores de busca e operações básicas para armazenamento e recuperação eficiente de dados.
- **Tópico 8 – Árvores Balanceadas e Heaps:** explora árvores AVL, heaps binários, heapsort, árvores de busca digitais (tries), árvores rubro-negras, skip lists e estruturas probabilísticas como filtros de Bloom.
- **Tópico 9 – Hash Tables:** discute tabelas de espalhamento, técnicas de hashing, colisões e aplicações práticas.
- **Tópico 10 – Grafos:** apresenta fundamentos básicos, algoritmos de busca (BFS e DFS), algoritmos clássicos como Dijkstra para caminhos mínimos, Prim e Kruskal para árvores geradoras mínimas, além de fluxo em redes e o problema do caixeiro viajante (TSP).

Ao final, o leitor terá percorrido uma jornada completa pelos principais pilares da ciência da computação relacionados a algoritmos e estruturas de dados, desenvolvendo uma capacidade analítica que será útil tanto na vida acadêmica quanto no mercado profissional.

Espero que este material ajude a consolidar seus conhecimentos e desperte o entusiasmo por este fascinante campo da computação. 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 A_1 e A_2 , como saber qual deles é mais eficiente?

Na realizada, estamos interessados em 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 determinar. 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).**

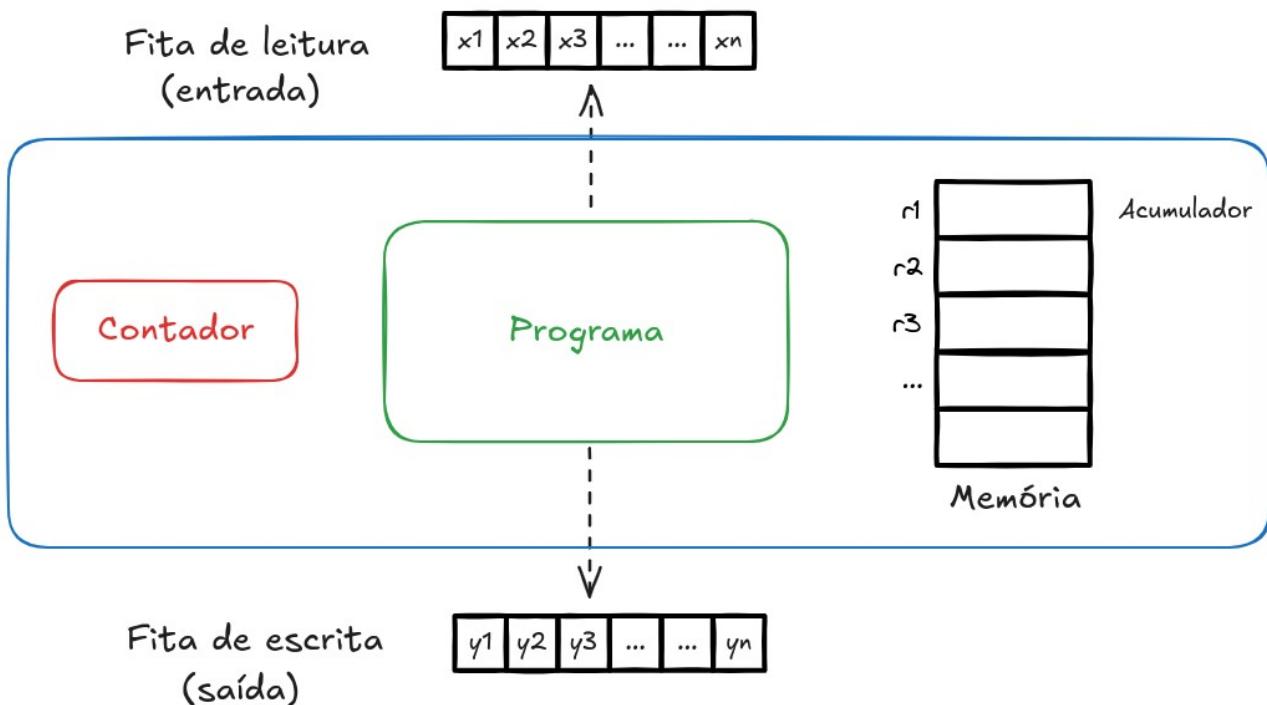
Desejamos realizar uma análise que seja universal, ou seja, dependa de fatores externos ao algoritmo em questão.

Complexidade computacional é um conceito antigo. Foi formulada sistematicamente no início da década de 1960 e, desde então, tem sido universalmente usada como função de custo para projetar algoritmos.

As operações elementares de uma CPU são chamadas de instruções e seus “custos” são chamados de latências. As instruções são armazenadas na memória e executadas uma a uma pelo processador, que possui algum estado interno armazenado em vários registradores. Um desses registradores é o ponteiro de instrução, que indica o endereço da próxima instrução a ser lida e executada. Cada instrução altera o estado do processador de uma certa maneira (incluindo mover o ponteiro da instrução), possivelmente modifica a memória principal e leva um número diferente de ciclos de CPU para ser concluída antes que a próxima possa ser iniciada. Para estimar o **tempo real** de execução de um programa, é necessário somar todas as latências de suas instruções executadas e dividir pela frequência do clock, ou seja, o número de ciclos que uma determinada CPU faz por segundo. Esse processo além de muito complicado, é praticamente inviável.

O modelo RAM (Random Access Machine)

O modelo **RAM** (*Random Access Machine*) é uma abstração de uma CPU, sendo o modelo teórico de computação mais básico já proposto. É um banco potencialmente não vinculado de células de memória, cada uma das quais pode conter um número ou caractere arbitrário. As células da memória são numeradas e leva tempo para acessar qualquer célula da memória, ou seja, todas as operações (leitura/gravação da memória, aritméticas padrão e operações booleanas) levam uma unidade de tempo. **RAM** é um modelo teórico padrão de computação com memória infinita e custo de acesso uniforme (todo acesso a memória tem o mesmo custo). O modelo **RAM** é fundamental para a análise da complexidade de algoritmos. O diagrama a seguir ilustra o modelo **RAM**.



O modelo de computação **RAM** (*Random Access Machine*) mede o tempo de execução de um algoritmo somando o número de etapas necessárias para executar o algoritmo em um conjunto de dados. O modelo **RAM** opera segundo os seguintes princípios:

- Operações lógicas ou aritméticas básicas (+, *, =, if, call) são consideradas operações simples que ocorrem em um intervalo de tempo.
- Loops e sub-rotinas são operações complexas compostas por múltiplos intervalos de tempo.
- Todo acesso à memória ocorre exatamente em um intervalo de tempo.

Na prática, esses princípios são equivalentes a dizer que a CPU possui as instruções LOAD, STORE, READ, WRITE, ADD, SUBTRACT, MULTIPLY, DIVIDE, TEST, JUMP, e HALT e que todas elas são realizadas em tempo constante (mesmo intervalo de tempo fixo e pré-determinado). Este modelo encapsula a funcionalidade central dos computadores modernos, mas não os imita completamente. Por exemplo, uma operação de adição e uma operação de multiplicação possuem um único passo de tempo no modelo RAM; no entanto, na realidade, serão necessárias mais operações para uma máquina calcular um produto versus uma soma. Apesar disso, na prática, o modelo RAM funciona muito bem para o cálculo da complexidade de algoritmos, pois ele é focado na análise assintótica. Como forma de simplificar ainda mais os cálculos matemáticos, **o objetivo consiste em contar o número de operações de atribuição** existentes em um algoritmo A (número de acessos a memória).

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
}

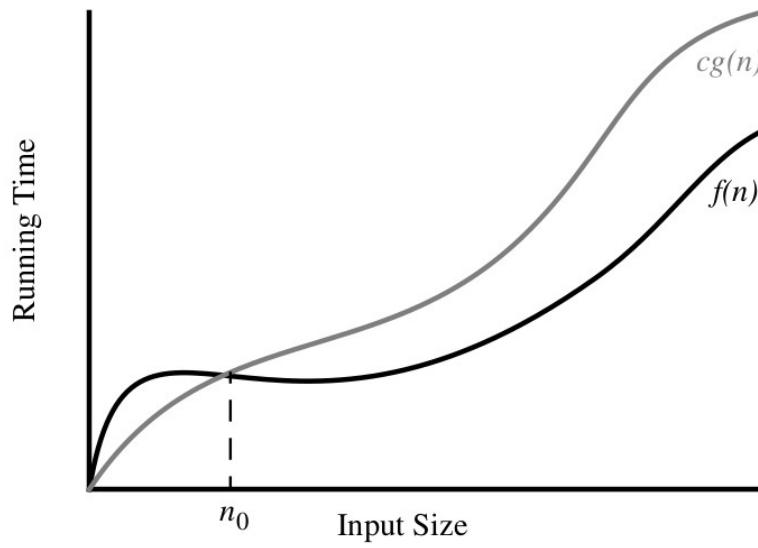
```

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 $n > 2$

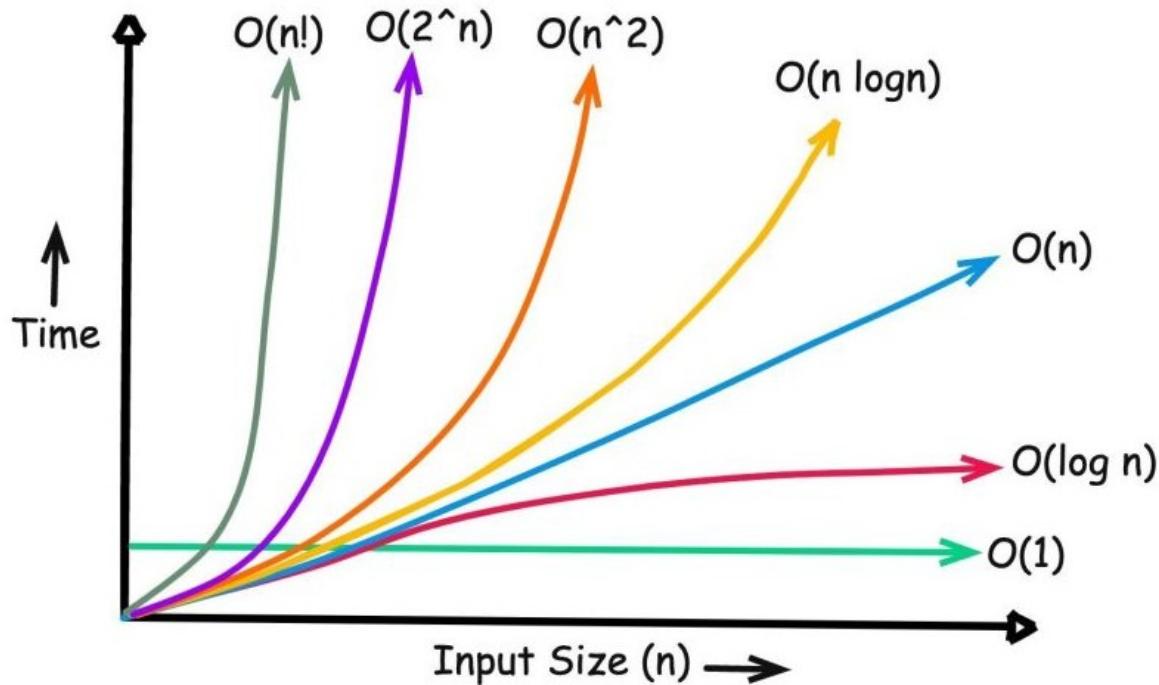
o que implica em dizer que T(n) é $O(n^2)$. Poderíamos ter escolhido a função $f(n)=n^2$, mas o objetivo da notação Big-O é fornecer o limite superior mais justo possível!



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 de algumas dessas funções.



Convém ressaltar que a análise exata de algoritmos é uma tarefa difícil. É da natureza da análise de algoritmo ser independente da máquina e da linguagem. Por exemplo, se o seu computador ficar duas vezes mais rápido após uma atualização recente, a complexidade do seu algoritmo ainda permanecerá a mesma. Além disso, a análise de algoritmos paralelos requer outros modelos de computação mais avançados que o modelo **RAM**.

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:

$T(n) = 1 + 2\lfloor \log_2 n \rfloor$, 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.

<pre> prime_A(n) { if n == 1 return False for i = 2 to n-1 { </pre>	<pre> prime_B(n) { if n == 1 return False if n == 2 </pre>
---	--

```

resto = n % i
if resto == 0
    return False
}
return True
}

        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 ($(n - 1 - 2 + 1) = (n - 3=2)$) 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, menor_A será mais rápido que menor_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.

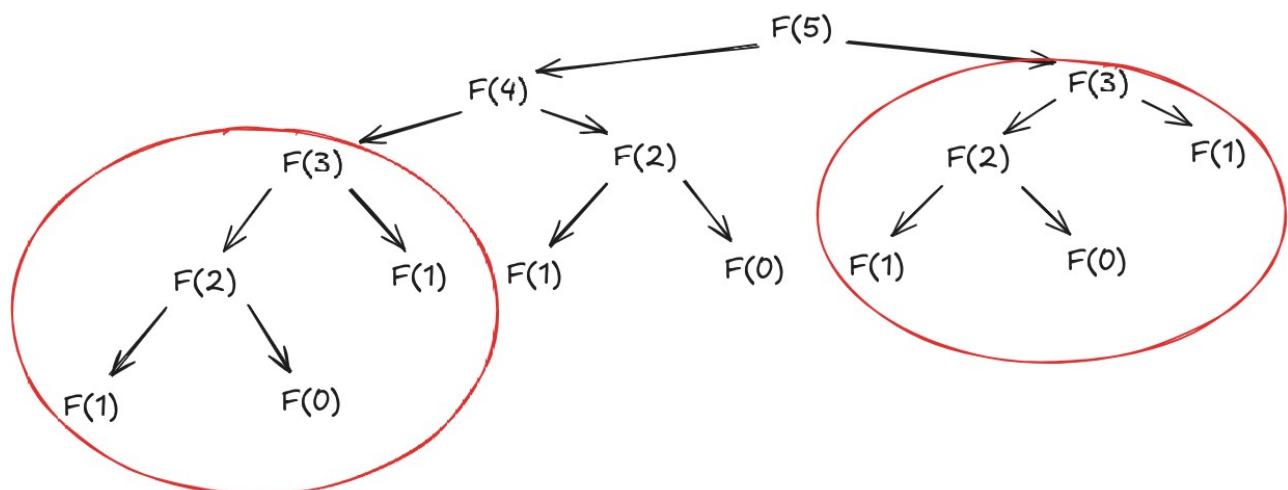
A série de Fibonacci é um exemplo clássico de recursão, pois:

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

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 $\text{mdc}(a, b) = 21$.

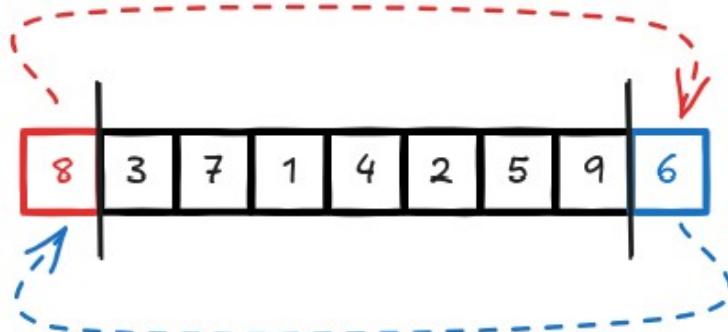
Faça duas funções para calcular o MDC entre dois inteiros: uma iterativa e outra recursiva.

```
# 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:

$$\begin{array}{r}
 384 \\
 \times 156 \\
 \hline
 2304 \\
 1920 + \\
 384 + \\
 \hline
 59904
 \end{array}
 \quad \begin{array}{l}
 \text{3 dígitos} \\
 \text{3 dígitos} \\
 \text{9 multiplicações} \\
 + \\
 \text{2 adições} \\
 \hline
 O(n^2)
 \end{array}
 \quad (n = 3)$$

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:

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

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:

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

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:

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

$$A = a_1 \times b_1$$

$$B = a_2 \times b_1$$

$$C = a_2 \times b_2$$

$$D = a_1 \times b_2$$

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

```
function 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(n²)? Expandindo o termo T(n/2), temos:

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

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

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

Extrapolando para um k arbitrário:

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

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,

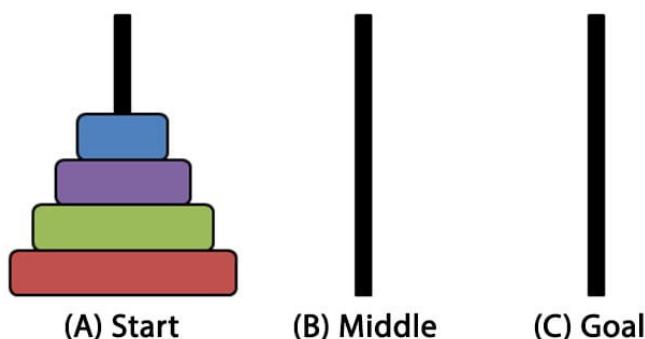
$$T(n) = (2^{2k})T(1) + n(2^k - 1) = 2^{\log_2 n^2} + n(2^{\log_2 n} - 1) = n^2 + n(n-1) = 2n^2 - n$$

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

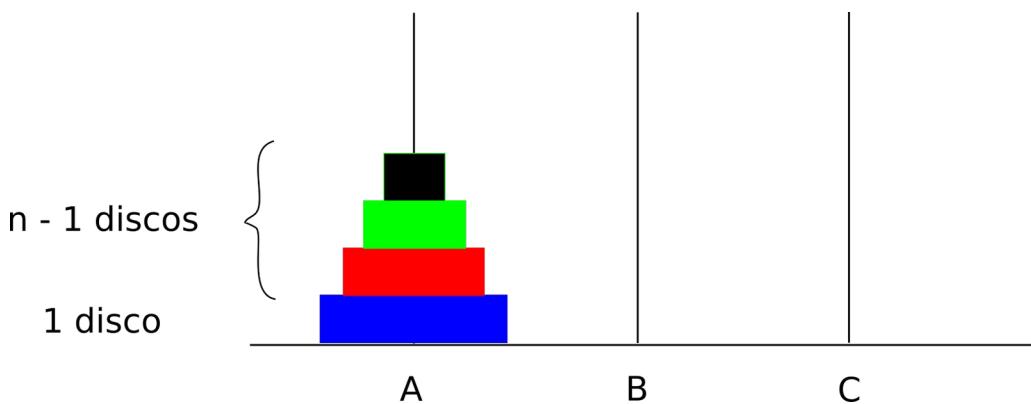
Se $n = 3$, são necessários 7 movimentos: Move A, B
 Move A, C
 Move B, C
 Move A, B
 Move C, A
 Move C, B
 Move A, B

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

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, os mais conhecidos, mas de forma alguma iremos esgotar o assunto. Além disso, analisaremos a complexidade de tais algoritmos para entender porque e quando devemos utilizá-los.

Problema: ordenação de dados

Entrada: uma sequência arbirtária 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.

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 $O(n^2)$.

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])
            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 $O(n^2)$.

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	<pre>i = 1 while i <= n { print(i) i = i + 1 }</pre>
--	---------------	---

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 $O(n^2)$.

Shellsort

O algoritmo Shellsort foi proposto originalmente por Donald Shell como uma otimização do método Insertionsort. A grande limitação do Insertionsort é a quantidade de comparações que devemos fazer até encontrar a posição correta do elemento. Por exemplo, quando o menor elemento está localizado na posição mais à direita do vetor (pior caso), são necessárias $n - 1$ comparações e trocas, o que o torna ineficiente para valores de n grande. A ideia por trás do algoritmo Shellsort é permitir trocas de elementos distantes um do outro no vetor, introduzindo o conceito de gap.

Os itens separados por um gap de h posições são rearranjados, gerando sequências ordenadas. Tais sequências são ditas estarem h -ordenadas. A cada iteração, o valor do gap h é reduzido progressivamente até atingir o valor 1, que resultará no vetor completamente ordenado. Quando a sequência está 1-ordenada, temos o final do algoritmo.

Pode-se mostrar que o Insertionsort é o Shellsort com $h = 1$. Sendo assim, no Shellsort, a ideia é que quando chegamos no $h = 1$, apenas algumas poucas trocas serão necessárias, pois algumas delas já foram feitas com valores maiores de h , tornando o método mais rápido e eficiente do que o Insertionsort.

Portanto, a lógica utilizada é exatamente a mesma do Insertionsort: inserir um elemento na sua posição correta dentro do conjunto. Vejamos um exemplo prático para ilustrar o funcionamento do método. Suponha que temos como entrada o seguinte vetor de inteiros:

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

Utilizaremos para este exemplo a sequência de gaps original proposta por Shell que consiste de na 1^a iteração usar $n/2$, na 2^a iteração usar $n/4$, na 3^a iteração usar $n/8$, e assim sucessivamente. Como no caso em questão $n = 10$, temos:

=> 1^a iteração (vermelho já está na posição correta, não mexe)

$h = 5$

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [4, 2, 13, 7, -3, 5, 15, 10, 1, 6] (troca)

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [4, 2, 13, 7, -3, 5, 15, 10, 1, 6] (não troca)

[4, 2, 13, 7, -3, 5, 15, 10, 1, 6] → [4, 2, 10, 7, -3, 5, 15, 13, 1, 6] (troca)

[4, 2, 10, 7, -3, 5, 15, 13, 1, 6] → [4, 2, 10, 1, -3, 5, 15, 13, 7, 6] (troca)

[4, 2, 10, 7, -3, 5, 15, 13, 1, 6] → [4, 2, 10, 1, -3, 5, 15, 13, 7, 6] (não troca)

=> 2^a iteração

$h = 2$ (divisão inteira de 10 por 4)

[4, 2, 10, 1, -3, 5, 15, 13, 7, 6] → [-3, 2, 4, 1, 7, 5, 10, 13, 15, 6] (leva para posições corretas)

[-3, 2, 4, 1, 7, 5, 10, 13, 15, 6] → [-3, 1, 4, 2, 7, 5, 10, 6, 15, 13] (leva para posições corretas)

=> 3^a iteração (Insertionsort, mas em um vetor praticamente ordenado)

$h = 1$

[-3, 1, 4, 2, 7, 5, 10, 6, 15, 13] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15] (leva para posições corretas)

A seguir é apresentado o algoritmo Shellsort na sua versão padrão.

```
ShellSort(L, n) {
    # Iniciamos com um gap grande, depois reduzimos
    gap = n//2
    # Realiza um Insertionsort utilizando o gap atual
    # Os primeiros elementos de cada sequência já estão ordenados
    # Adiciona demais em suas posições corretas
    while gap > 0 {
        # não precisa iniciar em 0, pois já está na posição correta
        for i = gap to n {
            # iteração do Insertionsort com gap no lugar de 1
            # encontra posição correta para pivô
            j = i
            while j >= gap and L[j-gap] > L[j] {
                swap(L[j], L[j-gap])
                j = j - gap
            }
        }
    }
}
```

```

    # diminui o valor de gap
    gap = gap//2
}

```

Análise da complexidade

Como o algoritmo Shellsort depende essencialmente da sequência de gaps empregada durante a ordenação dos dados, não existe apenas uma complexidade para o algoritmo, mas sim várias. Inclusive, a análise do algoritmo Shellsort no caso médio ainda é um problema em aberto e sem solução até hoje para diversas sequências de gap!

Para efeitos de simplificação dos cálculos, nesta explicação iremos considerar a sequência de gaps de Shell (padrão).

Melhor caso: lista já ordenada. Note que neste caso o loop WHILE interno não é executado nenhuma vez. O que nos resta são os seguintes comandos:

```

gap = n//2
while gap > 0 {
    for i = gap to n
        j = i
    gap = gap//2
}

```

Note que temos aqui o padrão de sucessivas divisões por 2. Logo, temos que o número de iterações do WHILE é $\log_2 n$, o que nos leva a:

$$T(n) = n \log n + \log n$$

o que é $O(n \log n)$.

Pior caso: lista em ordem decrescente. Neste caso, o número máximo de trocas depende do valor do gap. Inicialmente, temos $n/2$ sequências de tamanho 2, depois $n/4$ sequências de tamanho 4, depois $n/8$ sequências de tamanho 8 e assim sucessivamente. Portanto, podemos aproximar $T(n)$ (contando apenas o número de trocas) como:

$$T(n) = \left(n - \frac{n}{2} \right) 1 + \left(n - \frac{n}{4} \right) 3 + \left(n - \frac{n}{8} \right) 7 + \dots + \left(n - \frac{n}{2^k} \right) (2^k - 1)$$

$$T(n) = \frac{1}{2} 1n + \frac{3}{4} 3n + \frac{7}{8} 7n + \dots + \left(\frac{2^k - 1}{2^k} \right) (2^k - 1)n = n \left[\frac{1^2}{2^1} + \frac{3^2}{2^2} + \frac{7^2}{2^3} + \frac{15^2}{2^4} + \dots + \frac{(2^k - 1)^2}{2^k} \right]$$

Iremos denotar por S o somatório entre colchetes:

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

o que nos leva a:

$$S = \sum_{i=1}^k 2^i - 2 \sum_{i=1}^k 1 + \sum_{i=1}^k \frac{1}{2^i}$$

Denominando o primeiro somatório de A, sabe-se que $2^{i+1} = 2 \cdot 2^i = 2^i + 2^i$, o que nos leva a seguinte identidade:

$$2^i = 2^{i+1} - 2^i$$

Assim, temos a seguinte soma telescópica:

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

É fácil perceber que o segundo somatório em S, denominado de B, é igual a:

$$B = 2 \sum_{i=1}^k 1 = 2k$$

Por fim, o terceiro somatório em S, denominado de C, é uma PG de razão igual a 1/2, cujo primeiro termo é igual a 1/2:

$$C = \frac{a_1(1-q^k)}{(1-q)} = \frac{1/2[1-(1/2)^k]}{(1-1/2)} = 1 - (1/2)^k$$

Logo, podemos expressar S como:

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

Note que a condição de parada do Shellsort é o gap ser igual a 1, ou seja, quando $\frac{n}{2^k} = 1$, o que nos leva a $n = 2^k$, o que implica em $k = \log_2 n$. Sendo assim, voltando para S, temos:

$$S = 2(n-1) - 2\log_2 n + 1 - \frac{1}{n}$$

Mas como $T(n) = nS$, chegamos em:

$$T(n) = 2n^2 - 2n - 2n\log_2 n + n - 1 \quad \text{o que é } O(n^2).$$

Caso médio: utilizando uma abordagem similar ao pior caso, ao invés de realizar todas as trocas, realiza metade delas (aproximadamente metade do tamanho da lista).

$$T(n) = \left(n - \frac{n}{2}\right)1 + \left(n - \frac{n}{4}\right)2 + \left(n - \frac{n}{8}\right)4 + \dots + \left(n - \frac{n}{2^k}\right)2^{k-1}$$

Aplicando a distributiva, temos:

$$T(n) = \frac{n}{2}(1+3+7+15+\dots+(2^k-1))$$

Sabemos que o somatório entre parêntesis vale $A = 2(n-1) - \log_2 n$, o que nos leva a:

$$T(n) = \frac{n}{2}[2(n-1) - \log_2 n] = n^2 - n - \frac{1}{2} \log_2 n$$

o que resulta em $O(n^2)$ (como no pior caso).

Sequências de gap

Após a proposta original do algoritmo Shellsort, diversos estudos propuseram novas formas de definir os gaps utilizados nas iterações do algoritmo. O consenso geral é de que a cada passo o valor do gap h deve ser diminuído, até que na última iteração, atinja o menor valor possível. Sendo assim, diversas estratégias alternativas para a sequência de gaps mostraram que é possível melhorar o desempenho do algoritmo em termos de complexidade computacional. A seguir mostramos algumas das sequências de gaps propostas posteriormente à descoberta do algoritmo.

1. Sequência de Shell: $N/2, N/4, \dots, 1$ (sucessivas divisões inteiras por 2)

2. Sequência de Hibbard: $1, 3, 7, \dots, 2^k - 1$ (iniciando com $k = 1$)

A ideia aqui consiste em gerar a sequência até o k -ésimo elemento que seja menor ou igual a $N/2$. Os valores dos gaps h são tomados sempre do maior para o menor - $O(n^{3/2})$ no pior caso

3. Sequência de Knuth: $1, 4, 13, \dots, (3^k - 1) / 2$ (iniciando com $k = 1$ e não maior que $n/3$)

A ideia aqui consiste em gerar a sequência até o k -ésimo elemento que seja menor ou igual a $N/2$. Os valores dos gaps h são tomados sempre do maior para o menor - $O(n^{3/2})$ no pior caso

4. Sequência de Pratt: $1, 2, 3, 4, 6, 8, 9, 12, \dots$

A ideia aqui consiste em gerar a sequência até o k -ésimo elemento que seja menor ou igual a $N/2$. Os valores dos gaps h são tomados sempre do maior para o menor.

Sucessivos números da forma $2^p 3^q$. São definidos como os inteiros cujos fatores primos são todos menores ou iguais que 3 - $O(n(\log n)^2)$ no pior caso.

5. Sedgewick's increments: $1, 5, 19, 41, 109, \dots$

A ideia aqui consiste em gerar a sequência até o k -ésimo elemento que seja menor ou igual a $N/2$. Os valores dos gaps h são tomados sempre do maior para o menor - $O(n^{4/3})$ no pior caso.

$$\begin{cases} 9(2^k - 2^{k/2}) + 1, & \text{se } k \text{ par} \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1, & \text{se } k \text{ ímpar} \end{cases}$$

Existem diversas outras sequências de gaps na literatura, apresentamos aqui as mais conhecidas.

É por isso que diversos autores propuseram utilizar outras sequências de gap. Porém, com essas sequências mais complexas, em geral, não é possível obter expressões analíticas para $T(n)$.

É interessante notar que no pior caso, diferentes sequências de gap são capazes de produzir complexidades muito melhores que $O(n^2)$. Por exemplo, se considerarmos a sequência de Hibbard, pode-se mostrar que a complexidade de pior caso é $O(n^{3/2})$.

No caso da sequência de Pratt, pode-se mostrar que a complexidade de pior caso é $O(n \log^2 n)$.

Na sequência de Sedgewick, pode-se mostrar que a complexidade de pior caso é dada por $O(n^{4/3})$

Conforme mencionado anteriormente, calcular a complexidade do algoritmo Shellsort no caso médio para outras sequências que não a original de Shell, é um problema em aberto cuja solução

ainda é desconhecida. Porém, há resultados que comprovam que ela varia entre $O(n^{5/4})$ e $O(n \log^2 n)$.

Os dois próximos algoritmos de ordenação que iremos estudar são exemplos de abordagens recursivas, onde a cada passo, temos um subproblema menor para ser resolvido pela mesma função. Por isso, a função é definida em termos de si própria.

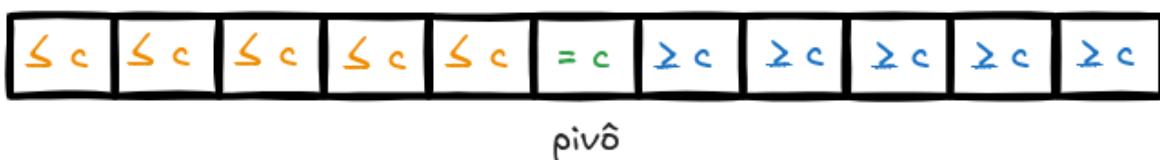
Quicksort

O algoritmo Quicksort é baseado na estratégia “Dividir para Conquistar”, pois ele quebra o problema de ordenar um vetor em subproblemas menores, mais fáceis e rápidos de serem resolvidos. Primeiramente, o método divide o vetor original em duas partes: os elementos menores que o pivô (tipicamente escolhido como o primeiro ou último elemento do conjunto). O método então ordena essas partes de maneira **recursiva**. O algoritmo pode ser dividido em 3 passos principais:

1. Escolha do pivô:

- a) Primeiro elemento da lista
- b) Último elemento da lista
- c) Mediana entre primeiro, central e último elementos (melhor)

2. Particionamento: reorganizar o vetor de modo que todos os elementos menores que o pivô apareçam antes dele (a esquerda) e os elementos maiores apareçam após ele (a direita). Ao término dessa etapa o pivô estará em sua posição final (existem várias formas de se fazer essa etapa)



3. Ordenação: recursivamente aplicar os passos acima aos sub-vetores produzidos durante o particionamento. O caso limite da recursão é o sub-vetor de tamanho 1, que já está ordenado.

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

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

1º passo: Definir pivô = 6 (último elemento)

2º passo: Particionar vetor (menores a esquerda e maiores a direita)

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

3º passo: Aplicar 1 e 2 recursivamente para as metades

a) 2 metades

Metade 1: [5, 2, -3, 4, 1] → pivô = 1

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

Metade 2: [13, 7, 15, 10] → pivô = 10

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

b) 4 metades

Note que a metade 1 possui um único elemento: [-3] → já está ordenada

Metade 2: [5, 2, 4] → pivô = 4

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

Note que a metade 3 possui apenas um único elemento: [7] → já está ordenadas

Metade 4: [15, 13] → pivô = 13

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

c) 4 metades: Note que cada uma das 4 metades restantes contém um único elemento e portanto já estão ordenadas.

Fim.

A seguir apresentamos o algoritmo Quicksort juntamente com uma função auxiliar para o particionamento da lista em duas sublistas menores.

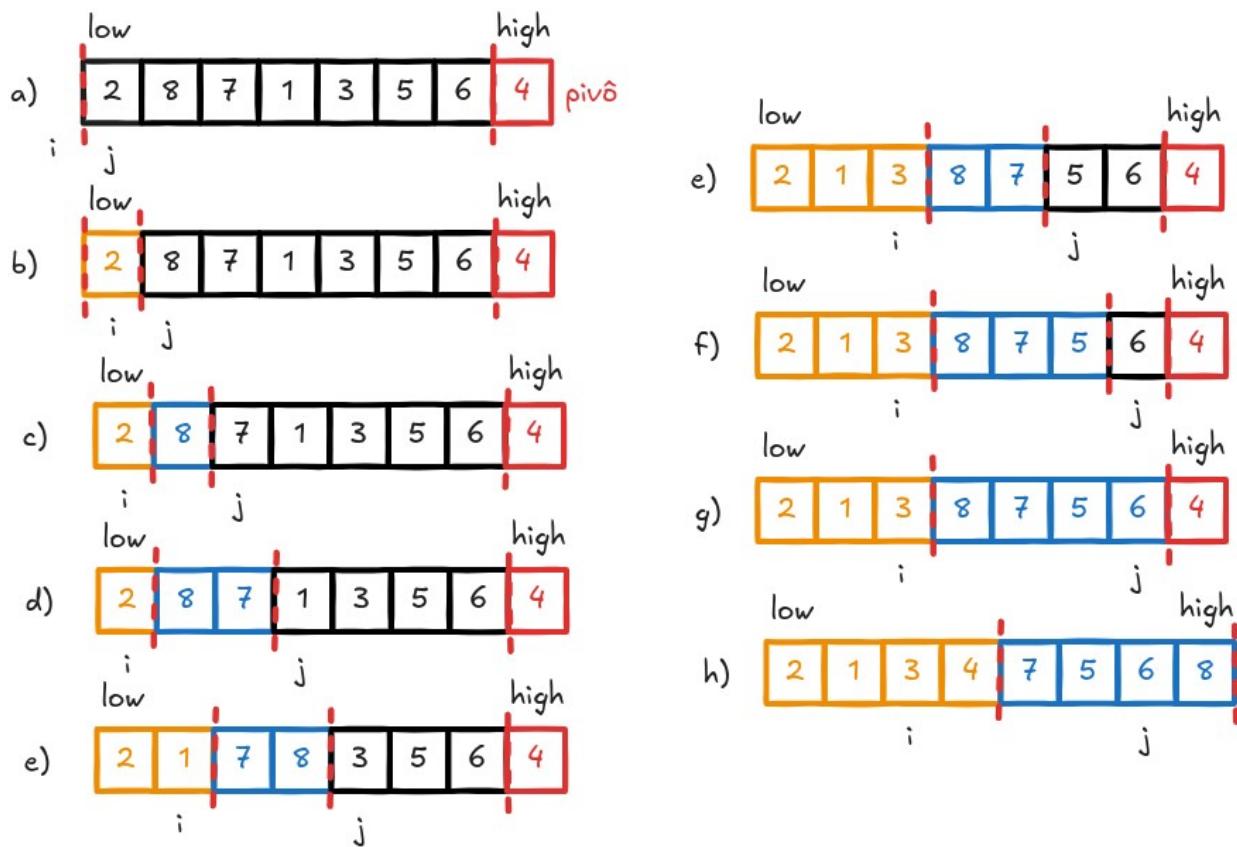
```
QuickSort(L, low, high) {
    if low < high {
        q = partition(L, low, high)
        QuickSort(L, low, q-1)
        QuickSort(L, q+1, high)
    }
}
```

Para a eficiência do algoritmo Quicksort é fundamental termos como particionar a lista com custo computacional $O(n)$. A seguir veremos uma função eficiente para particionar a lista usando como pivô o último elemento de L.

```
partition(L, low, high) {
    x = L[high]
    i = low - 1
    for j = low to high - 1 {
        if L[j] <= x {
            i = i + 1
            swap(L[i], L[j])
        }
    }
    swap(L[i+1], A[high])
    return i+1
}

# pivô
# maior índice do lado esquerdo
# processa todo elemento (- pivô)
# pertence ao lado esquerdo?
# novo índice para lado esquerdo
# adiciona ele no lado esquerdo
# pivô vai no fim do lado esquerdo
# índice do pivô
```

A figura a seguir ilustra o funcionamento do algoritmo para particionar uma lista em duas sublistas.



Análise da complexidade

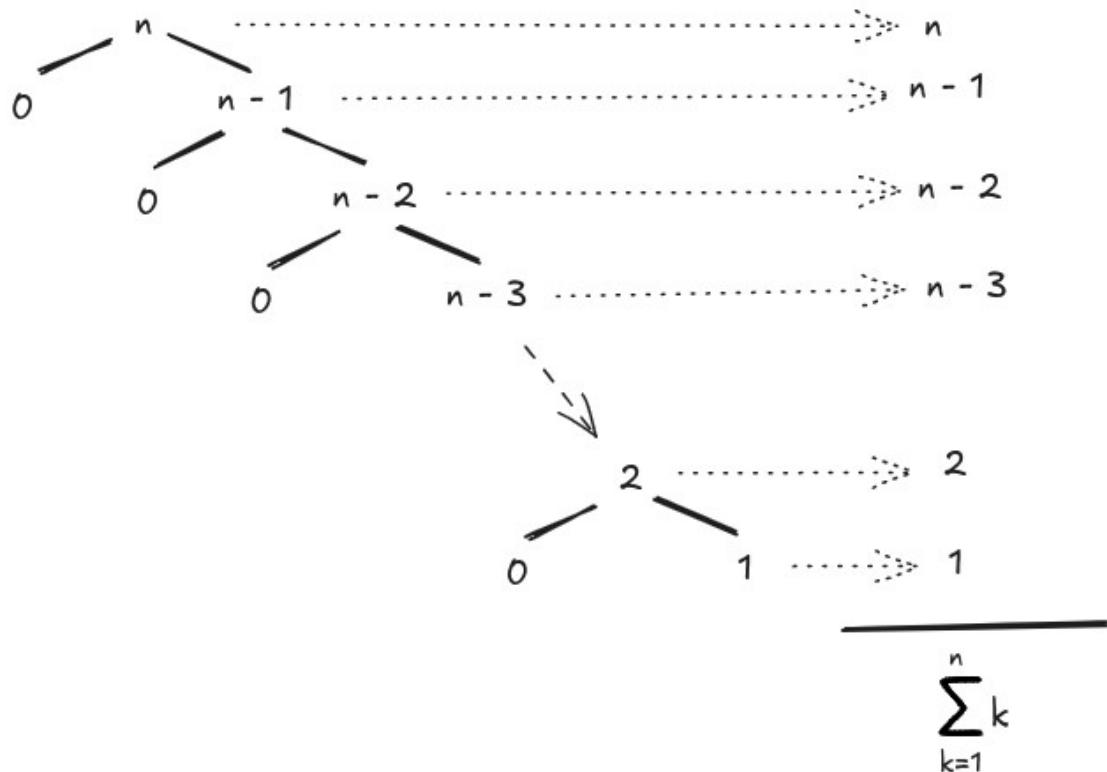
Iremos considerar 3 cenários distintos: pior caso, caso médio e melhor caso.

Pior caso: quando o pivô é sempre o maior ou menor elemento, o que gera partições totalmente desbalanceadas:

Na primeira chamada recursiva, temos uma lista de tamanho n , então para criar as novas listas L , teremos $n - 1$ elementos na primeira lista, o pivô e uma lista vazia. Isso nos permite escrever a seguinte relação de recorrência:

$$T(n) = T(n-1) + T(0) + n = T(n-1) + n$$

pois estamos decompondo um problema de tamanho n em um de tamanho zero e outro de tamanho $n - 1$, mas para realizar a divisão da lista em sublistas, utilizamos n operações (uma vez que a lista L tem n elementos)



Somando todas os custos em cada nível, temos o somatório:

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

cuja resposta é:

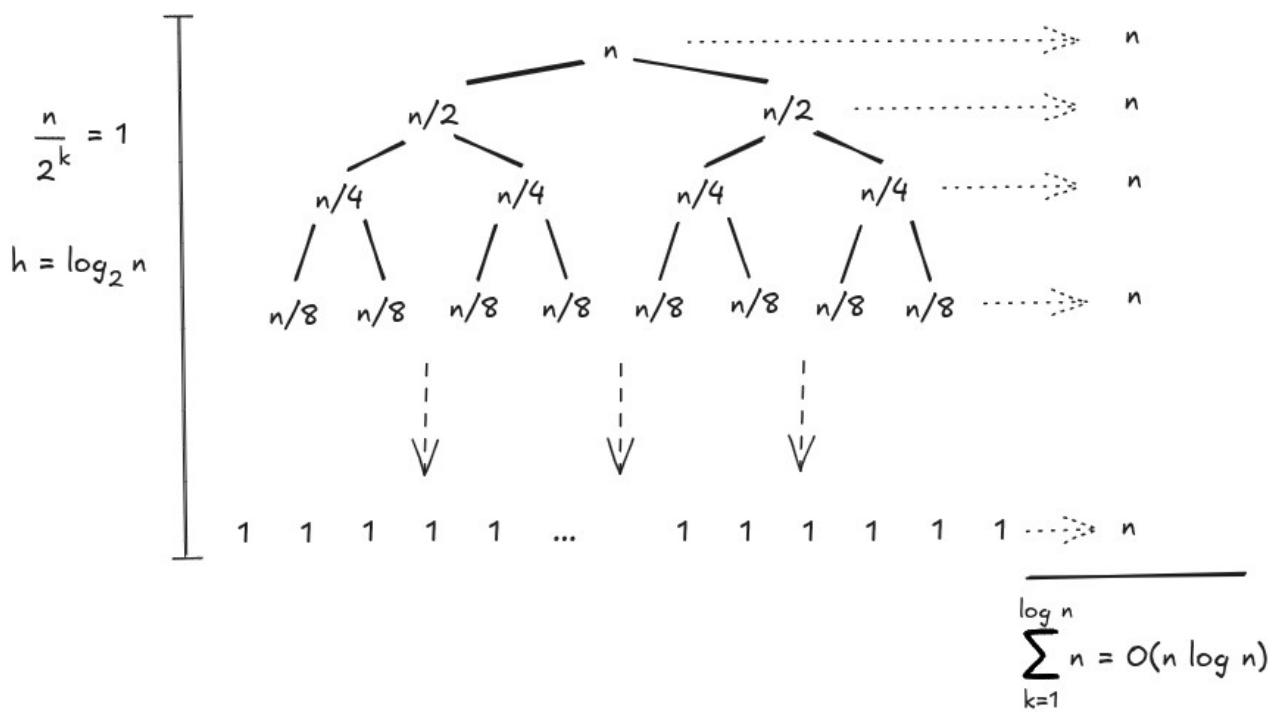
$$T(n) = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

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

Melhor caso: ocorre quando as duas partições tem exatamente o mesmo tamanho, ou seja, são $n/2$ elementos, o pivô e mais $n/2$ elementos. Podemos escrever a seguinte relação de recorrência:

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

pois estamos decompondo um problema de tamanho n em dois problemas de tamanho $n/2$, mas para realizar a divisão da lista L em sublistas, utilizamos n operações na função partition.



Como divide-se 1 problema de tamanho n em dois de tamanho $n/2$, a altura da árvore é $h = \log_2 n$ e como em cada nível da árvore de recursão o custo é n , temos que o custo total do algoritmo Quicksort no melhor caso é $O(n \log_2 n)$.

Caso médio: iremos utilizar uma estratégia de calcular a média para todas as possíveis partições.

Os tamanhos possíveis de partição são: $(1, n - 1); (2, n - 2); (3, n - 3); \dots; (n - 1, 1)$. Assim, temos:

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

Separando os somatórios, temos:

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

Note que os dois somatórios são de fato idênticos (os termos ocorrem na ordem inversa um do outro). Dessa forma, podemos escrever:

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

Multiplicando ambos os lados por n , temos:

$$nT(n) = 2 \sum_{i=0}^{n-1} T(i) + cn^2 \quad (\text{I})$$

Reescrevendo a identidade (I) fazendo $n = n - 1$:

$$(n-1)T(n-1)=2\sum_{i=0}^{n-2} T(i)+c(n-1)^2 \quad (\text{II})$$

Calculando a diferença (I) – (II) e eliminando constantes:

$$nT(n)-(n-1)T(n-1)=2T(n-1)+2cn$$

Isolando o termo $T(n)$, chega-se em:

$$nT(n)=2T(n-1)+nT(n-1)-T(n-1)+2cn$$

Agrupando os termos:

$$nT(n)=(n+1)T(n-1)+2cn$$

Dividindo ambos os lados por $n(n+1)$:

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

Utilizando a recursão, podemos expandir $T(n-1)$:

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

Repetindo o processo para $T(n-2)$:

$$\frac{T(n)}{n+1}=\frac{T(n-3)}{n-2}+\frac{2c}{n-1}+\frac{2c}{n}+\frac{2c}{n+1}$$

Continuando o processo até $T(1)$, temos a seguinte sequência:

$$\frac{T(n)}{n+1}=\frac{T(1)}{2}+\frac{2c}{3}+\frac{2c}{4}+\frac{2c}{5}+\dots+\frac{2c}{n+1}$$

Como $T(1) = O(1)$, podemos escrever o somatório a seguir:

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

Para resolver o somatório em questão, utilizaremos uma aproximação do cálculo. Sabemos que:

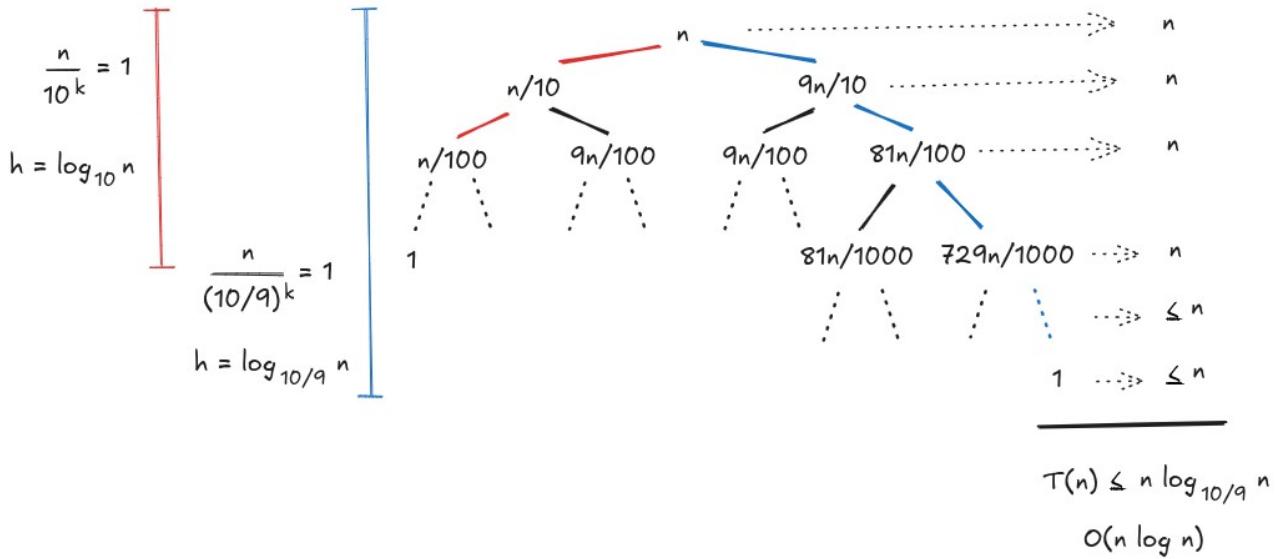
$$\log_e a = \ln a = \int_1^a \frac{1}{x} dx$$

Assim, podemos escrever:

$$T(n)=(n+1)(O(1)+2cO(\log_e n))=(n+1)+2cnO(\log_e n)+2cO(\log_e n)$$

o que resulta em complexidade $O(n \log n)$. Uma estratégia empírica que, em geral, melhora o desempenho do algoritmo Quicksort consiste em escolher como pivô a mediana entre o primeiro elemento, o elemento do meio e o último elemento (regra mediana de 3).

A grande vantagem do algoritmo Quicksort é que ele opera no modo caso médio, praticamente sempre. Isso porque mesmo com partições do tipo 90%-10%, pode-se verificar que a complexidade é $O(n \log n)$.



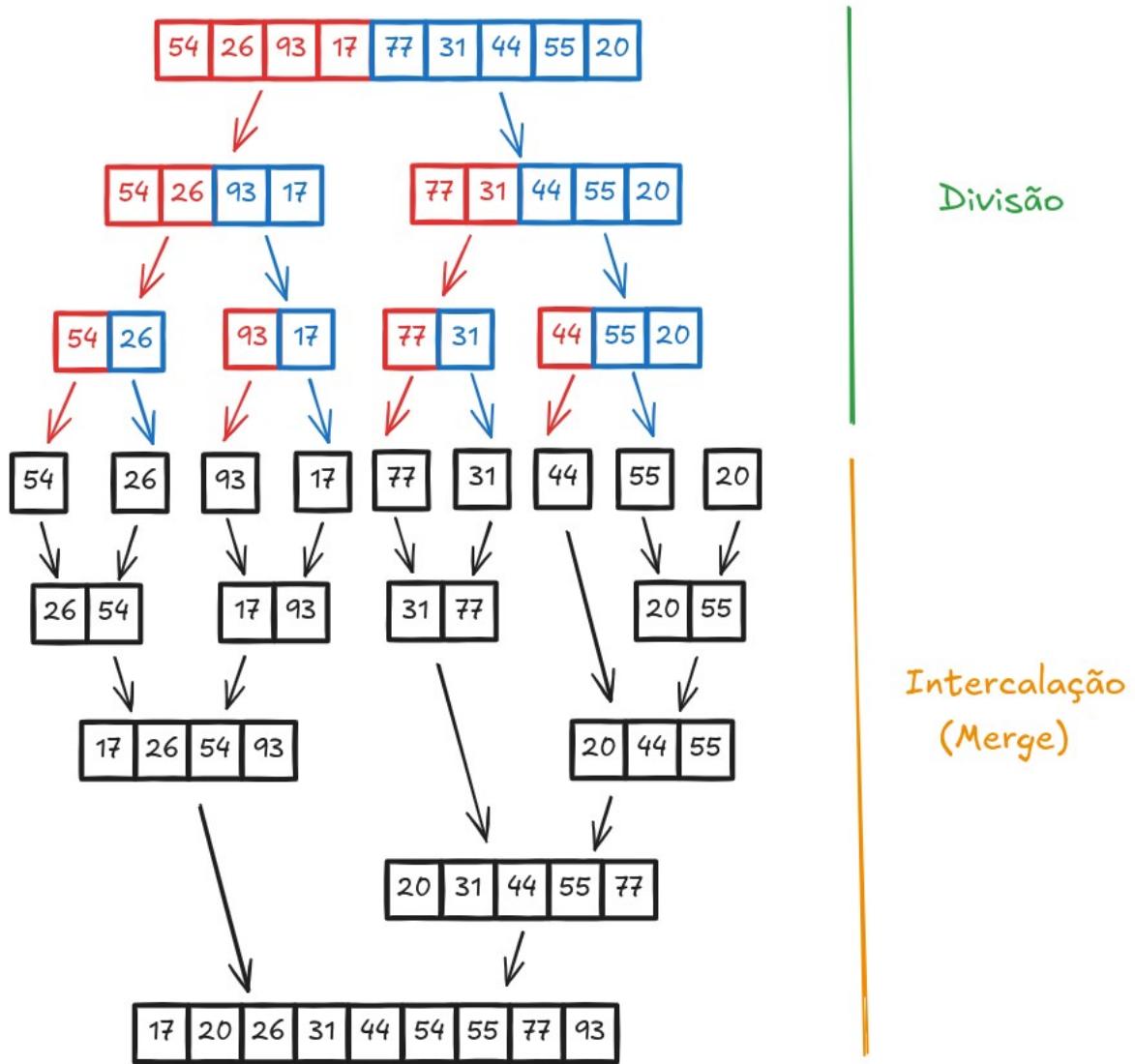
Neste caso, note que a altura do ramo mais à esquerda é a menor possível, sendo igual a $h_L = \log_{10} n$. Já a altura do ramo mais à direita é a maior possível, sendo igual a $h_R = \log_{10/9} n$.

Sendo assim, o custo dentro de um nível até h_L será igual a n , e o custo dentro de um nível após h_L será menor que n . Somando todos os custos, chegamos em $T(n) \leq cn \log_{10/9} n$. Sabendo que

$$\log_{10/9} n = \frac{\log_2 n}{\log_2 10/9} \approx \frac{\log_2 n}{0.152} \approx 6.578 \log_2 n$$

Mergesort (ordenação por intercalação)

O algoritmo Mergesort utiliza a abordagem Dividir para Conquistar. A ideia básica consiste em dividir o problema em vários subproblemas e resolver esses subproblemas através da recursividade e depois conquistar, o que é feito após todos os subproblemas terem sido resolvidos através da união das resoluções dos subproblemas menores. Trata-se de um algoritmo recursivo que divide uma lista continuamente pela metade. Se a lista estiver vazia ou tiver um único elemento, ela está ordenada por definição (o caso base). Se a lista tiver mais de um elemento, dividimos a lista e invocamos recursivamente um Mergesort em ambas as metades. Assim que as metades estiverem ordenadas, a operação fundamental, chamada de **intercalação**, é realizada. Intercalar é o processo de pegar duas listas menores ordenadas e combiná-las de modo a formar uma lista nova, única e ordenada. A figura a seguir ilustra as duas fases principais do algoritmo Mergesort: a divisão e a intercalação.



O exemplo a seguir mostra o passo a passo para a ordenação da seguinte lista:

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

Passo 1: Dividir em subproblemas

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

$$\text{meio} = 10 // 2 = 5$$

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

$$\text{meio} = 5 // 2 = 2$$

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

$$\text{meio} = 2 // 2 = 1 \quad \text{ou} \quad \text{meio} = 3 // 2 = 1$$

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

$$\text{meio} = 2 // 2 = 1$$

Passo 2: Intercalar listas (Merge) – as últimas a serem divididas serão as primeiras a fazer o merge

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

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

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

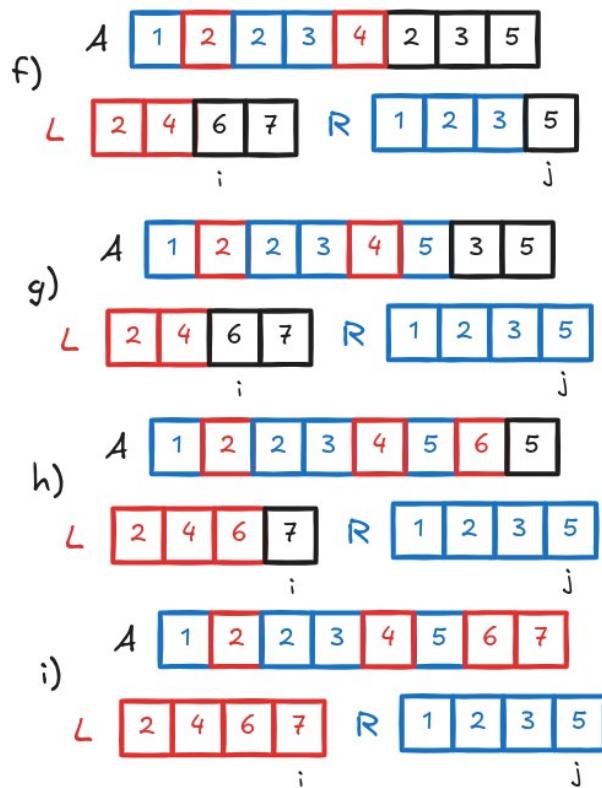
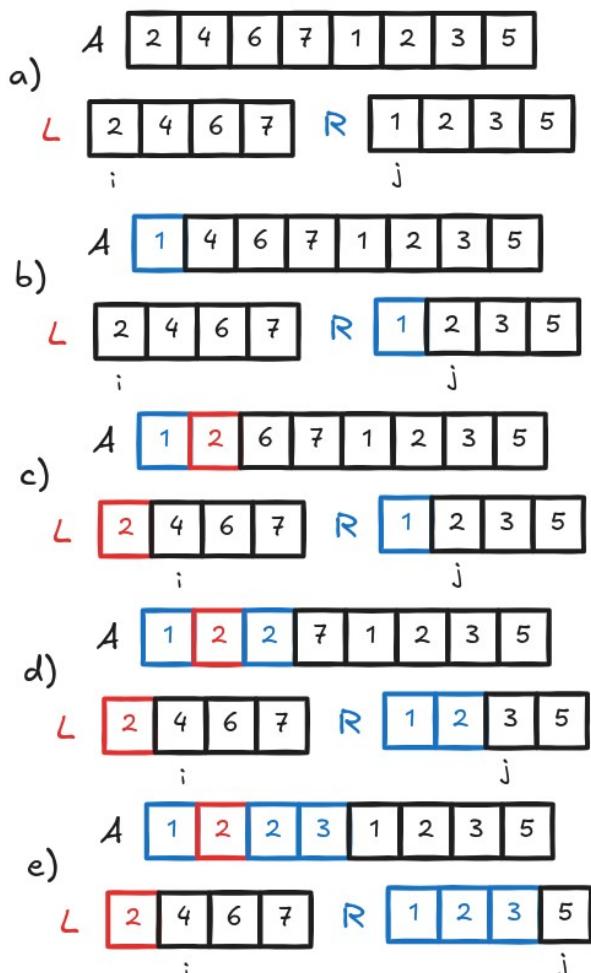
[-3, 2, 5, 7, 13, 1, 4, 6, 10, 15]

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

A seguir apresentamos o algoritmo Merge, que utiliza é responsável por realizar a intercalação de duas sublistas com custo computacional O(n).

```
merge(A, low, q, high) {           # q é o ponto médio
    n_L = q - low + 1             # número de elementos da sublista esq
    n_R = high - q               # número de elementos da sublista dir
    Let L[0..n_L - 1] be an array
    Let R[0..n_R - 1] be an array
    for i = 0 to n_L - 1
        L[i] = A[low + i]
    for j = 0 to n_R - 1
        R[j] = A[(q+1)+j]
    i = 0
    j = 0
    k = 0
    while i < n_L and j < n_R {
        if L[i] ≤ R[j] {
            A[k] = L[i]
            i = i + 1
        }
        else {
            A[k] = R[j]
            j = j + 1
        }
        k = k + 1
    }
    while i < n_L {                 # se sobrar elementos em L
        A[k] = L[i]
        i = i + 1
        k = k + 1
    }
    while j < n_R {                 # se sobrar elementos em R
        A[k] = R[j]
        j = j + 1
        k = k + 1
    }
}
```

A figura a seguir ilustra o funcionamento da função merge.



A seguir é apresentada a função mergesort, que realiza a ordenação de uma lista.

```
mergesort(A, low, high) {
    if low ≥ high # lista tem zero ou um elemento
        return
    q = ⌊(p+q)/2⌋
    mergesort(A, low, q)
    mergesort(A, q+1, high)
    merge(A, low, q, high)
}
```

A função Mergesort mostrada acima começa perguntando pelo caso base. Se o tamanho da lista for menor ou igual a um, então já temos uma lista ordenada e nenhum processamento adicional é necessário. Se, por outro lado, o tamanho da lista for maior do que um, então devemos recursivamente aplicar o mergesort nas metades da esquerda e direita. É importante observar que a lista pode não ter um número par de elementos. Isso, contudo, não importa, já que a diferença de tamanho entre as listas será de apenas um elemento.

Quando a função Mergesort retorna da recursão (após a chamada nas metades esquerda, LE, e direita, LD), elas já estão ordenadas. O resto da função é responsável por intercalar as duas listas ordenadas menores em uma lista ordenada maior. Note que a operação de intercalação coloca um item por vez de volta na lista original (L) ao tomar repetidamente o menor item das listas ordenadas.

Análise da complexidade

É interessante notar que o algoritmo Mergesort se comporta da mesma forma para o caso médio, melhor caso e pior caso.

Os três passos úteis dos algoritmos de dividir para conquistar que se aplicam ao MergeSort são:

1. Dividir: Calcula o ponto médio do sub-arranjo, o que demora um tempo constante $O(1)$;
2. Conquistar: Recursivamente resolve dois subproblemas, cada um de tamanho $n/2$, o que contribui com $T(n/2) + T(n/2)$ para o tempo de execução;
3. Combinar: Unir os sub-arranjos em um único conjunto ordenado, que leva o tempo $O(n)$;

Assim, podemos escrever a relação de recorrência como:

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

Trata-se da mesma recorrência resolvida no algoritmo Quicksort. Note que expandindo a recorrência, temos:

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

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

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

$$T\left(\frac{n}{8}\right) = 2T\left(\frac{n}{16}\right) + n$$

...

Voltando com as substituições, podemos escrever:

$$T(n) = 2 \left[2 \left[2 \left[2T\left(\frac{n}{16}\right) + \frac{n}{8} \right] + \frac{n}{4} \right] + \frac{n}{2} \right] + n = 2^4 T\left(\frac{n}{2^4}\right) + 4n$$

Generalizando para um valor k arbitrário, podemos escrever:

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

Para que tenhamos $T(1)$, é preciso que $n = 2^k$, ou seja, $k = \log_2 n$. Quando $k = \log_2 n$, temos:

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

Como $T(1)$ é $O(1)$, temos que a complexidade do Mergesort em qualquer caso é $O(n \log_2 n)$.

Uma das maiores limitações do algoritmo MergeSort é que esse método passa por todo o longo processo mesmo se a lista L já estiver ordenada. Por essa razão, a complexidade de melhor caso é idêntica a complexidade de pior caso, ou seja, $O(n \log n)$. Para o caso de n muito grande, e

listas compostas por números gerados aleatoriamente, pode-se mostrar que o número médio de comparações realizadas pelo algoritmo MergeSort é aproximadamente αn menor que o número de comparações no pior caso, onde:

$$\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k + 1} \approx 0.2645$$

Limitante Inferior para Ordenação Baseada em Comparações

Os algoritmos de ordenação vistos até o momento são todos baseados em comparações, ou seja, eles obtêm informação sobre a sequência através de uma função que recebe dois elementos e retorna qual deles é o maior, para saber se deve trocá-los ou não de posição.

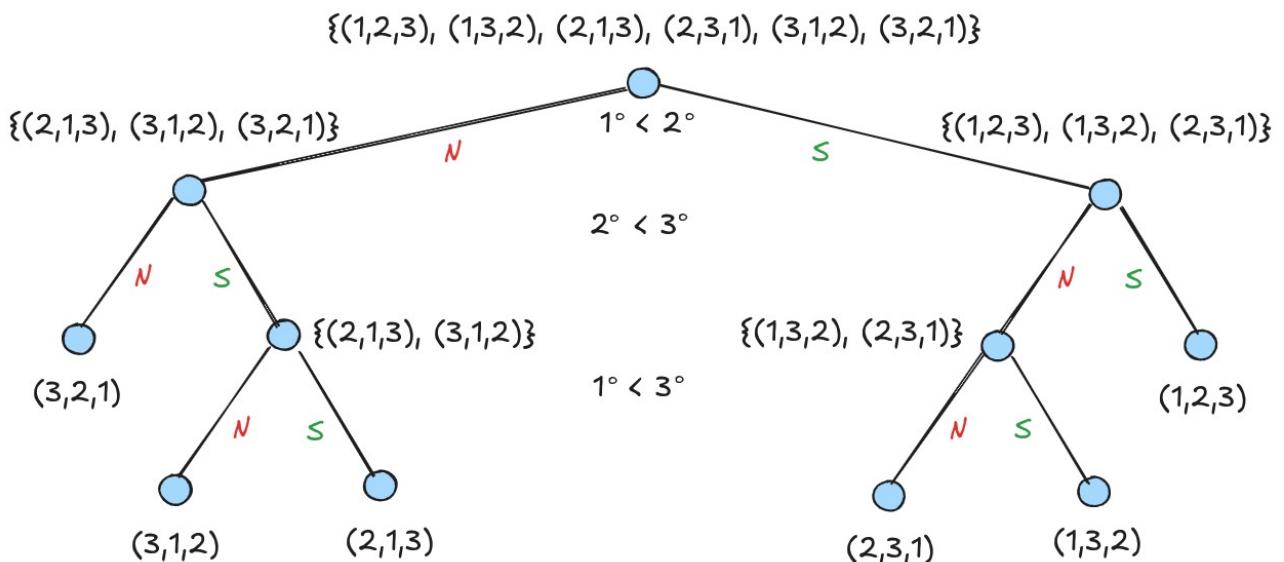
Para obter um limitante inferior para o número de trocas, vamos comparar duas grandezas:

- i) número de permutações que sequência de tamanho n pode apresentar;
- ii) número de sequências que um algoritmo consegue distinguir após k comparações;

Com relação ao primeiro ponto, é fácil notar que esse número é $n!$

$\overline{n} \quad \overline{n-1} \quad \overline{n-2} \quad \overline{n-3} \quad \dots \quad \overline{1}$

Com relação ao segundo ponto, note que inicialmente todas as sequências são iguais para o algoritmo.



Note que no melhor das hipóteses, cada comparação divide o espaço de busca em duas metades, ou seja, dobra o número de sequências identificadas.

Sabemos que em uma árvore binária de altura k (número de decisões), o número máximo de nós armazenados (sequências diferentes) é igual a 2^k . Sendo assim, a pergunta é: quantas sequências distintas “cabem” na árvore binária? No máximo 2^k .

Pelo princípio da casa dos pomos, se temos $n+1$ pomos para serem colocados em n casas, uma casa deverá conter 2 ou mais pomos. Nesse cenário, temos:

* Pombos: $n!$ permutações.

* Casas: 2^k slots na árvore.

Assim, se $n! > 2^k$ não é possível distinguir todas as possíveis sequências. Então, para que seja possível é preciso ter:

$$2^k \geq n!$$

Primeiramente, note que pela definição de fatorial, temos:

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

Sendo assim, podemos escrever:

$$n! \geq \left[\underbrace{\frac{n}{2} \times \frac{n}{2} \times \frac{n}{2} \times \dots \times \frac{n}{2}}_{n/2 \text{ termos}} \right] \times \left[\underbrace{1 \times 1 \times 1 \dots \times 1}_{n/2 \text{ termos}} \right]$$

o que nos leva a:

$$n! \geq \left(\frac{n}{2} \right)^{\frac{n}{2}}$$

Assim, temos a seguinte desigualdade:

$$2^k \geq n! \geq \left(\frac{n}{2} \right)^{\frac{n}{2}}$$

Aplicando o logaritmo, finalmente chega-se a:

$$k \geq \frac{n}{2} \log_2 \frac{n}{2}$$

o que nos permite escrever que $k = \Omega(n \log n)$. Portanto, precisamos ter pelo menos $n \log n$ comparações (e trocas).

Ordenação Não Baseada em Comparações

Nem todos os algoritmos de ordenação são baseados em comparações. Existem diversos métodos que ordenam listas sem a necessidade de comparar e trocar elementos. Alguns exemplos são:

- Countingsort, Bucketsort, Radixsort e Pigeonholesort.

Veremos a seguir os algoritmos Countingsort e Bucketsort.

Countingsort

O algoritmo Countingsort ordena os elementos de uma sequência pela contagem do número de ocorrências de cada elemento. Para mostrar a intuição por trás do algoritmo, iremos descrever seus passos ilustrando seu funcionamento em um exemplo ilustrativo. O algoritmo Countingsort é composto pelos seguintes passos:

Passo 1. Encontrar o maior elemento da lista.

$$L = [4, 2, 2, 8, 3, 3, 1] \max = 8$$

Passo 2. Criar um vetor de tamanho ($\max + 1$) composto por zeros.

$$H = [0, 0, 0, 0, 0, 0, 0, 0]$$

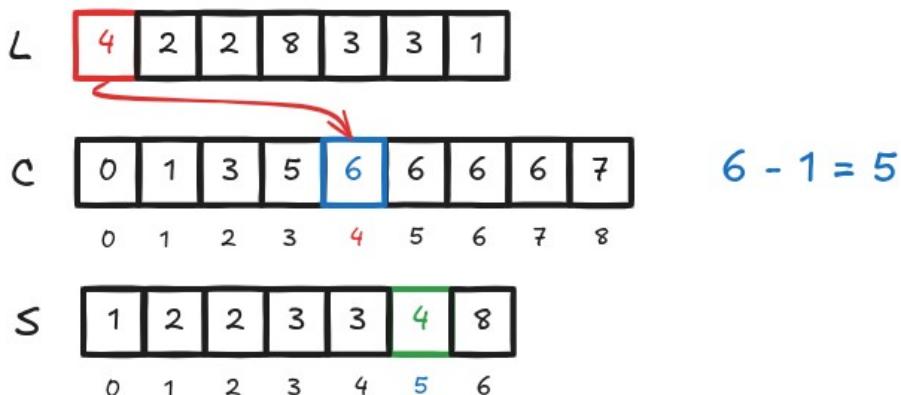
Passo 3. Armazene o número de ocorrências de cada elemento em seu respectivo índice em H. Note que poderão sobrar vários elementos nulos em H.

$$H = [0, 1, 2, 2, 1, 0, 0, 0, 1]$$

Passo 4. Calcule a soma cumulativa dos elementos de H.

$$C = [0, 1, 3, 5, 6, 6, 6, 6, 7]$$

Passo 5. Encontre o índice de cada elemento da lista L no vetor C, e coloque o respectivo elemento no índice $C[L[i]] - 1$ de uma lista S com o mesmo número de elementos de L.



Passo 6. Após colocar cada elemento na sua posição correta, diminua seu valor associado em C de uma unidade (ou seja, o 6 em C referente ao 4 inserido na posição 5, vira 5)

A seguir apresentamos o algoritmo Countingsort em pseudocódigo para que seja possível analisarmos sua complexidade.

```
Countingsort(L, n) {
    m = max(L)                                # maior elemento de L
    C = zeros(m+1)                             # vetor de 0 a m
    S = zeros(n)                               # vetor com mesmo tamanho de L
    for i = 0 to n - 1
        C[L[i]] = C[L[i]] + 1                # conta número de ocorrências
```

```

        for i = 1 to m
            C[i] = C[i] + C[i-1]  # soma acumulada
        for i = 0 to n - 1 {
            C[L[i]] = C[L[i]] - 1
            S[C[L[i]]] = L[i]
        }
    return S
}

```

Análise da complexidade

Primeiramente, note que encontrar o maior elemento de L possui complexidade $O(n)$. Analisando cada uma das 3 estruturas de repetição (FOR), podemos escrever a função $T(n)$ como:

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

Em geral, o algoritmo funciona muito bem quando o maior elemento da lista não é tão grande (pois isso reduz m). Por exemplo, quando $m < n$, como n domina m , a complexidade se torna $O(n)$. Porém, esse algoritmo tem limitações. A principal delas é quando o maior elemento é muito grande, o que torna a complexidade $O(m)$: isso faz com que o vetor count seja muito maior que L ($m \gg n$). Mesmo que n seja pequeno, ou seja, uma lista com apenas 100 elementos, o algoritmo pode levar um tempo elevado!

Radixsort

Pode ser considerado uma generalização do algoritmo Countingsort.

É um método baseado no agrupamento de elementos por dígitos, iniciando pelo dígito menos significativo.

Primeiro, ordenamos pela unidade. Depois, ordenamos pela dezena, centena, milhar, etc... até o número de dígitos do máximo elemento na lista. Por exemplo, considere a seguinte lista de inteiros:

[121, 432, 564, 23, 1, 45, 788]

Como, o máximo elemento é 788, temos que o número máximo de dígitos é $k = 3$.

1. Unidade: [121, 001, 432, 023, 564, 045, 788]
2. Dezena: [001, 121, 023, 432, 045, 564, 788]
3. Centena: [001, 023, 045, 121, 432, 564, 788]

Uma boa opção consiste em adotar uma versão modificada do Countingsort para o Radixsort, uma vez que estaremos lidando apenas com dígitos de 0 a 9 (m é pequeno e o algoritmo CountingSort é muito eficiente!). A seguir apresentamos o pseudocódigo do algoritmo Countingsort_R, usado no Radixsort.

```

Countingsort_R(L, n, d) {      # qual dígito é selecionado: uni, dez, ...
    S = zeros(n)                # vetor com mesmo tamanho de L
    m = (L[0]/d) % 10           # dígito d do primeiro elemento

```

```

for i = 1 to n - 1 {
    if (L[i]/d) % 10 > m
        m = L[i]           # m é o maior dos dígitos d
}
C = zeros(m+1)
for i = 0 to n - 1
    C[(L[i]/d)%10] += 1   # número de ocorrências
for i = 1 to m
    C[i] += C[i-1]         # soma acumulada
for i = 0 to n - 1 {
    C[(L[i]/d)%10] -= 1
    S[C[(L[i]/d)%10]] = L[i]
}
for i = 0 to n - 1
    L[i] = S[i]
}

```

A seguir apresentamos o algoritmo Radixsort. A ideia consiste em executar d vezes o Countingsort_R, onde d é o número de dígitos do maior elemento da lista L .

```

Radixsort(L, n) {
    m = max(L)
    d = 1                 # d começa com 1 (unidade)
    while m/d > 0 {
        Countingsort_R(L, n, d)
        d = d*10          # a cada iteração multiplica por 10
    }
}

```

Para analisar a complexidade do algoritmo Radixsort, primeiramente, devemos analisar a complexidade da versão modificada do Countingsort, denominada Countingsort_R. É possível notar que temos 5 estruturas de repetição (FOR) sequenciais. Sendo assim, a função $T(n)$ referente ao Countingsort_R é dada por:

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

Porém, neste caso, o valor de m é no máximo 9, o que faz com que n seja dominante, levando a complexidade $O(n)$.

Prosseguindo para a função Radixsort, podemos escrever a função $T(n)$ como:

$$T(n) = O(n) + \sum_{i=1}^k [O(n)+1] = O(n) + kO(n) + k$$

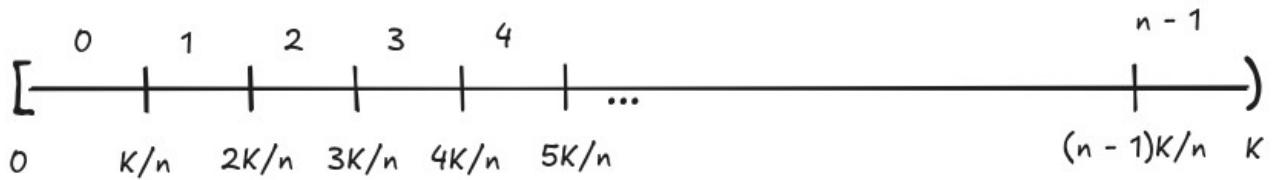
onde k denota o o número de dígitos do maior inteiro em L . Como em geral k é uma constante muito menor que n , temos que a complexidade resultante é $O(n)$.

Bucketsort

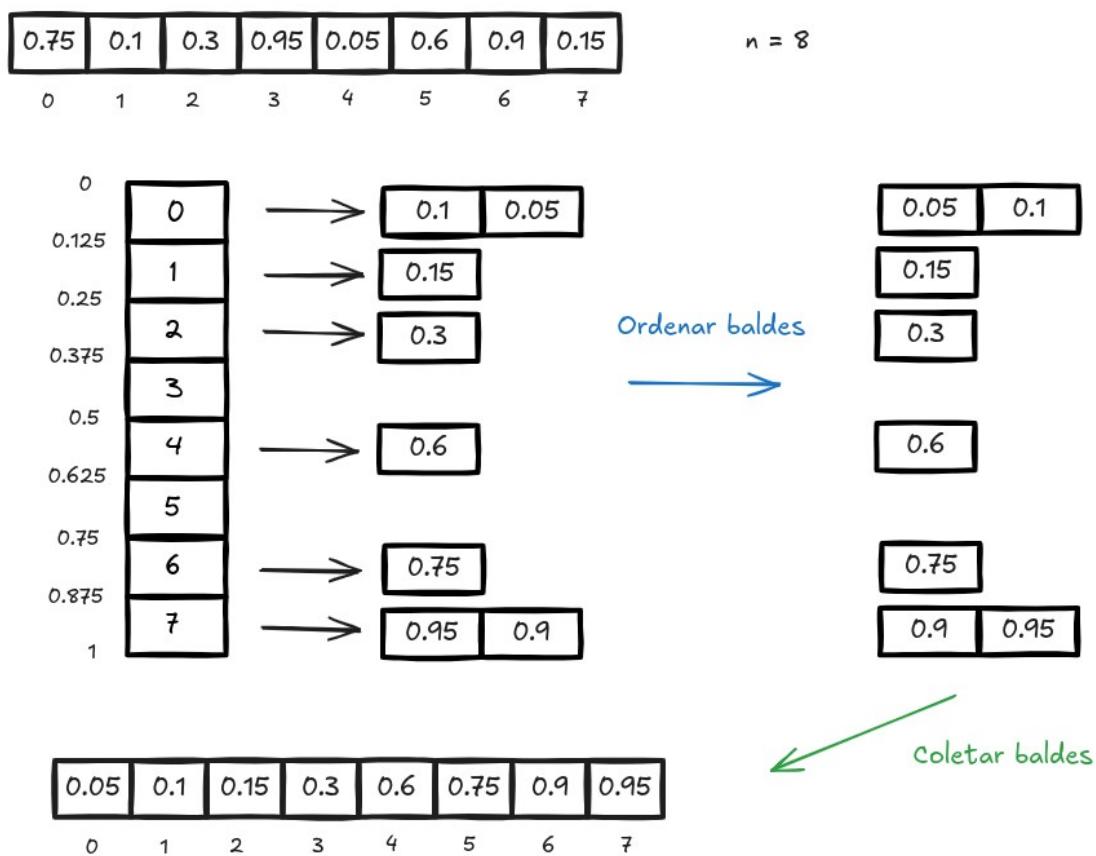
É um algoritmo de ordenação eficiente para ordenar um conjunto de n chaves distribuídas uniformemente em um intervalo (hipótese). Não é baseado em comparações, por isso não precisa

realizar trocas de posições. Seja $I = [0, K]$ um intervalo arbitrário. O primeiro passo do algoritmo consiste em dividir o intervalo de tamanho K em n baldes (buckets), cada um com tamanho K/n .

A ideia é simples: devemos colocar cada número da lista L a ser ordenada em seu respectivo balde. Sob a hipótese de uniformidade, o número de elementos por balde será pequeno e constante, fazendo com que o custo computacional da ordenação de cada balde seja $O(1)$.



Por fim, percorremos os baldes na ordem em que aparecem copiando os elementos ordenados de cada balde para a lista final. A figura a seguir ilustra um simples exemplo.



A seguir apresentamos o algoritmo Bucketsort escrito como um pseudo-código para que seja possível analisar sua complexidade.

```
Bucketsort(L, n) {
    for i = 0 to n - 1
        B[i] = NIL # Cria n baldes vazios
    for i = 0 to n - 1 {
        if L[i] == 1
            index = n - 1
        else {
            index = floor(n*L[i])
            insert(B[index], L[i])
```

```

        }
    }
    for i = 0 to n - 1
        sort(B[i])
    S = concatenate(B[0], B[1], ..., B[n-1])
    return S
}

```

Uma observação relevante é que, se os dados estão uniformemente distribuídos no intervalo $[a, b]$, se calcularmos:

$$y_i = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$$

então os dados terão distribuição uniforme no intervalo $[0, 1]$. De modo similar, ao fazer

$$x_i = x_{\min} + (x_{\max} - x_{\min}) y_i$$

devolvemos os dados para o intervalo original $[a, b]$.

Análise da complexidade

Note que utilizamos 3 primitivas básicas no algoritmo Bucketsort.

- i)** insert(p, x): insere x na lista apontada pela referência p . Possui complexidade $O(1)$.
- ii)** sort(p): ordena a lista apontada pela referência p . Se utilizarmos um algoritmo baseado em comparações, sabe-se que a complexidade seria entre $O(n \log n)$ e $O(n^2)$.
- iii)** concatenate(B, n): retorna a lista obtida pela concatenação das listas $B[0], B[1], \dots, B[n-1]$. Possui complexidade $O(kn)$, onde k é uma constante que representa o tamanho médio dos baldes, o que resulta em $O(n)$.

O ponto crítico é a análise da primitiva sort(), sendo que devemos realizar uma análise probabilística. Seja X_i o número de elementos na lista $B[i]$. Seja ainda a variável binária:

$$X_{ij} = \begin{cases} 1, & \text{se o elemento } j \text{ de } L \text{ foi para a lista } B[i] \\ 0, & \text{se o elemento } j \text{ de } L \text{ não foi para a lista } B[i] \end{cases}$$

Note que $X_i = \sum_j X_{ij}$.

Iremos denotar por Y_i o número de comparações necessárias para ordenar a lista $B[i]$. Observe que $Y_i \leq X_i^2$, pois no pior caso a ordenação será $O(n^2)$. Logo como desejamos analisar o caso médio, podemos escrever:

$$E[Y_i] \leq E[X_i^2] = E\left[\left(\sum_j X_{ij}\right)^2\right]$$

Mas podemos desenvolver o valor esperado como:

$$E\left[\left(\sum_j X_{ij}\right)^2\right] = E\left[\left(\sum_j X_{ij}\right)\left(\sum_k X_{ik}\right)\right] = E\left[\sum_j \sum_k X_{ij} X_{ik}\right] = E\left[\sum_j X_{ij}^2 + \sum_j \sum_{k \neq j} X_{ij} X_{ik}\right]$$

onde a última igualdade é válida pois o somatório duplo inclui os termos a seguir:

(1, 1) (1, 2) (1, 3), ..., (1, n)

(2, 1) (2, 2) (2, 3), ..., (2, n)

(3, 1) (3, 2) (3, 3), ..., (3, n)

...

(n, 1) (n, 2) (n, 3), ..., (n, n)

Note que para $k = j$ temos os elementos da diagonal e para os termos $k \neq j$ temos os elementos fora da diagonal. Dessa forma, podemos escrever:

$$E[Y_i] \leq \sum_j E[X_{ij}^2] + \sum_j \sum_{k \neq j} E[X_{ij} X_{ik}]$$

Como X_{ij} é uma variável aleatória binária:

$$P(X_{ij}=0) + P(X_{ij}=1) = 1$$

Pela definição de valor esperado, temos:

$$E[X_{ij}^2] = \sum_{x \in \{0,1\}} X_{ij}^2 P(X_{ij}=x) = 0^2 P(X_{ij}=0) + 1^2 P(X_{ij}=1) = P(X_{ij}=1)$$

Assumindo que a probabilidade de um elemento cair em um dos n baldes é uniforme, isto é, todos os baldes possuem a mesma probabilidade, chega-se a:

$$E[X_{ij}^2] = P(X_{ij}=1) = \frac{1}{n}$$

Para calcular o valor esperado do produto, note que para $j \neq k$ as duas variáveis aleatórias são independentes, ou seja:

$$E[X_{ij} X_{ik}] = E[X_{ij}] E[X_{ik}] = \frac{1}{n} \frac{1}{n} = \frac{1}{n^2}$$

Isso nos leva a:

$$E[Y_i] \leq \sum_{j=1}^n \frac{1}{n} + \sum_{j=1}^n \sum_{k \neq j} \frac{1}{n^2} = n \frac{1}{n} + \sum_{j=1}^n (n-1) \frac{1}{n^2} = 1 + \frac{n(n-1)}{n} = 1 + \frac{n^2-n}{n^2} = 2 - \frac{1}{n}$$

Esse é o número médio de comparações para ordenar o balde $B[i]$. Como temos n baldes, a complexidade total do Bucketsort é:

$$E[Y] = E\left[\sum_{i=1}^n Y_i\right] = E\left[n\left(2 - \frac{1}{n}\right)\right] = E[2n - 1] = 2n - 1$$

o que indica complexidade $O(n)$. Por essa razão, a complexidade total do Bucketsort é:

$$n * O(1) + O(n) + O(n) + O(n)$$

o que finalmente resulta em $O(n)$. Vimos que o Bucketsort é um algoritmo de ordenação linear no caso médio. Mas e no pior caso? Esse cenário ocorre quando todos os elementos caem no mesmo balde. Sendo assim, se optarmos pela aplicação de um algoritmo baseado em trocas, a complexidade poderia variar de $O(n \log n)$ a $O(n^2)$. 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	$O(n^2)$	$O(n^2)$	$O(n^2)$
ShellSort	$O(n \log n)$	$O(n^{1.25})$ a $O(n \log^2 n)$	$O(n \log^2 n)$ a $O(n^{1.333})$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Countingsort	$O(n+m)$	$O(n+m)$	$O(n+m)$
Radixsort	$O(nd)$	$O(nd)$	$O(nd)$
Bucketsort	$O(n)$	$O(n)$	$O(n \log n)$

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.

Algoritmo	Estável
Bubblesort	Sim
Selectionsort	Não
Insertionsort	Sim
Shellsort	Não
Quicksort	Não
Mergesort	Sim
Heapsort	Não
Countingsort	Sim
Bucketsort	Sim

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

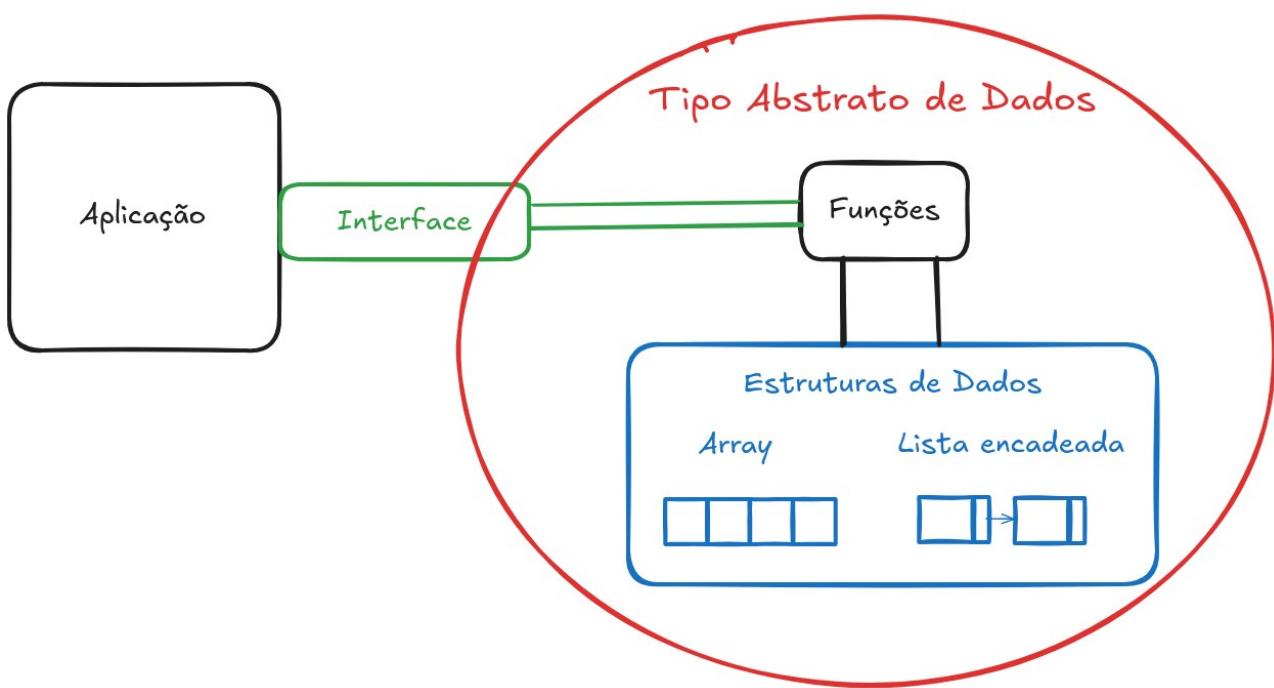
<https://www.youtube.com/user/AlgoRhythmic>

"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:

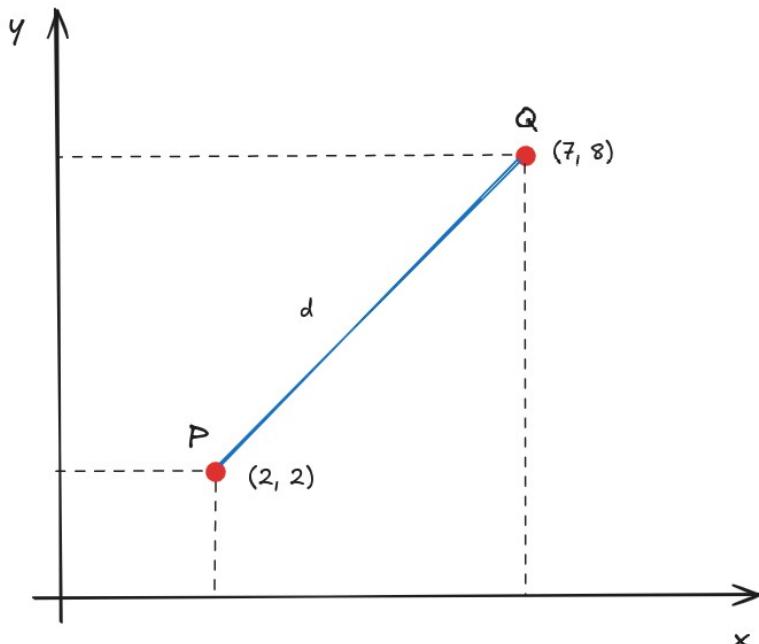
1. Foco na resolução do problema e não nos detalhes de implementação.
2. Redução de erros pelo encapsulamento de código validado.
3. Correção de bugs e manutenção de código (podemos modificar a parte interna de um TAD sem se preocupar com o programa que utiliza o TAD).
4. Redução da complexidade no desenvolvimento 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("Distancia 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. Portanto, 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.

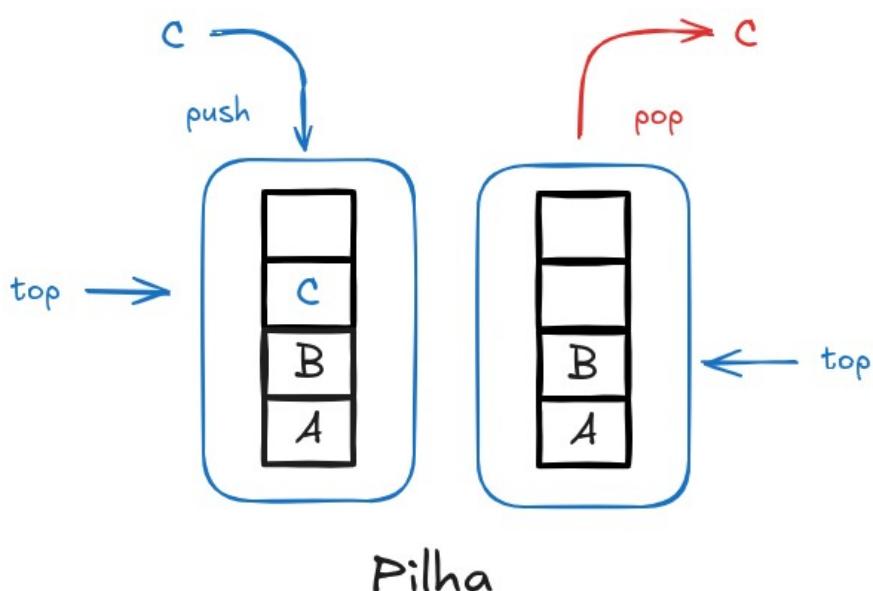
Iniciaremos nosso estudo com Pilhas e Filas, que são coleções em que os elementos são organizados dependendo de como eles são adicionados ou removidos do conjunto. Uma vez que um elemento é adicionado ele permanece na mesma posição relativa ao elemento que veio antes e o elemento que veio depois. Essa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

Pergunta: 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

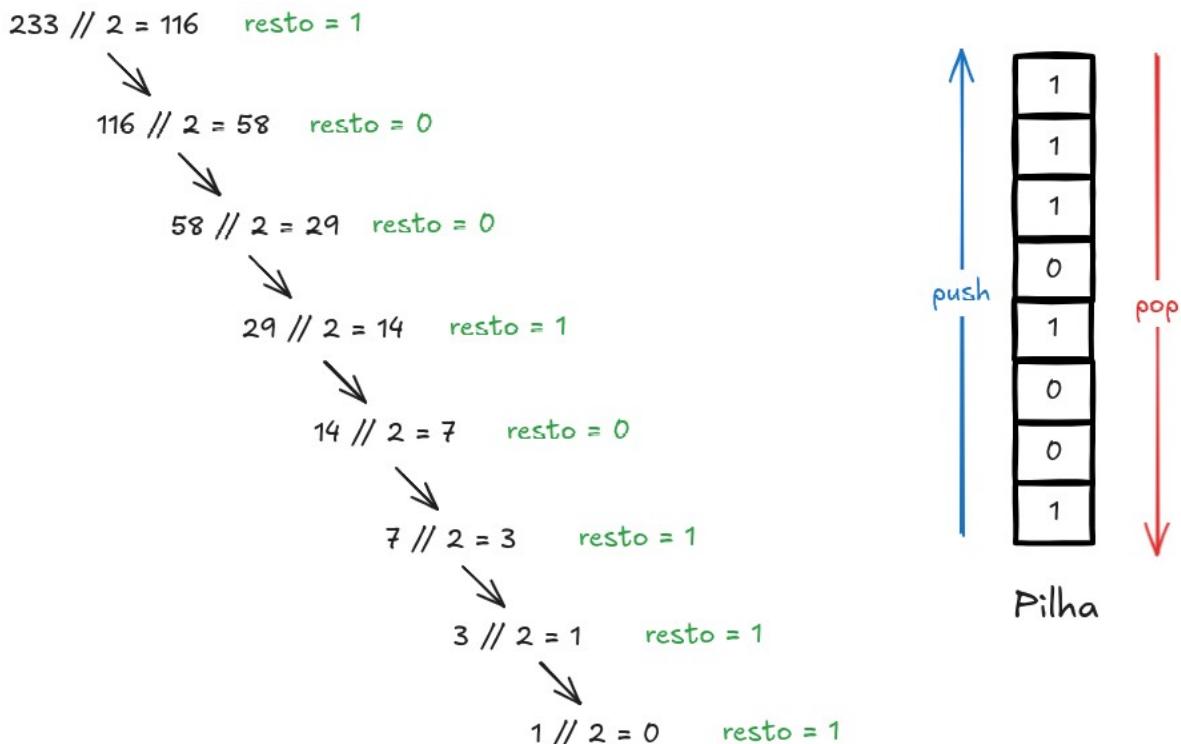
$1 / 2 = 0$ e sobra resto **1** → esse 1 será o nosso último bit (mais a esquerda)

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
}

```

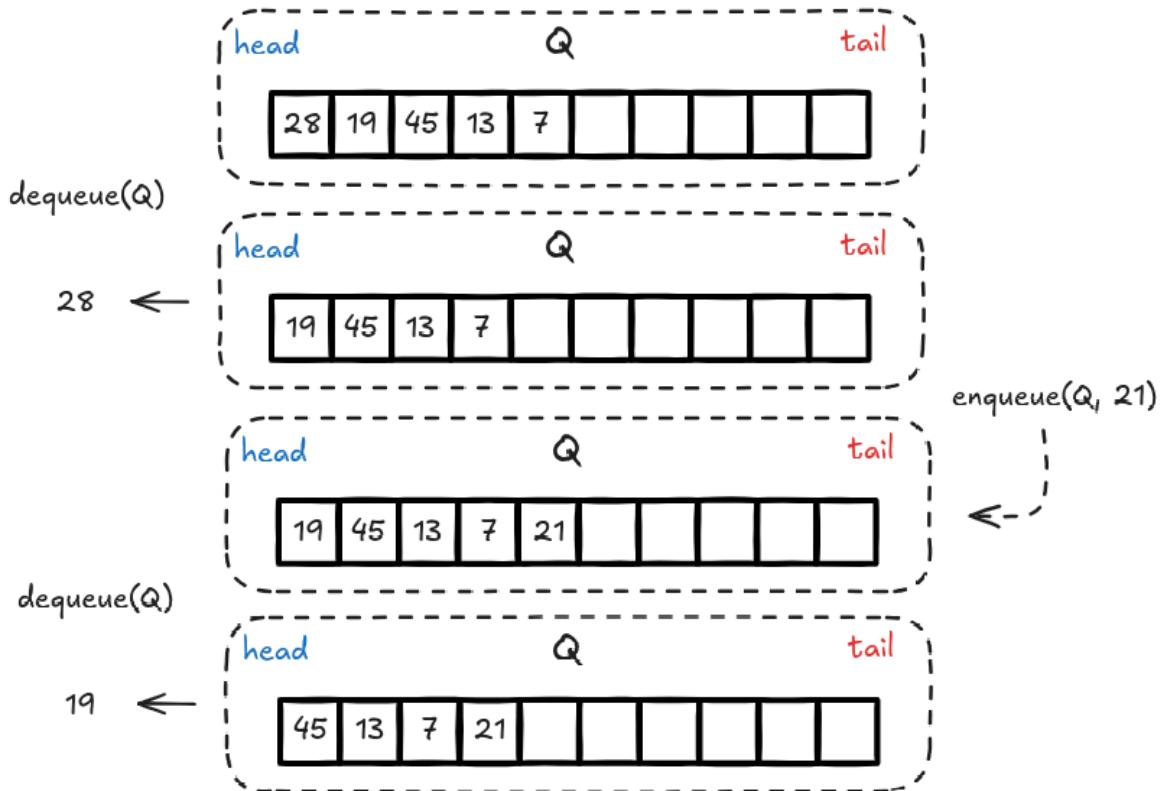
Note que as primitivas push e pop da Pilha possuem complexidade O(1), o que faz com que o algoritmo anterior possua complexidade O(n), ou seja, é mais eficiente do que a solução trivial em termos computacionais.

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
}

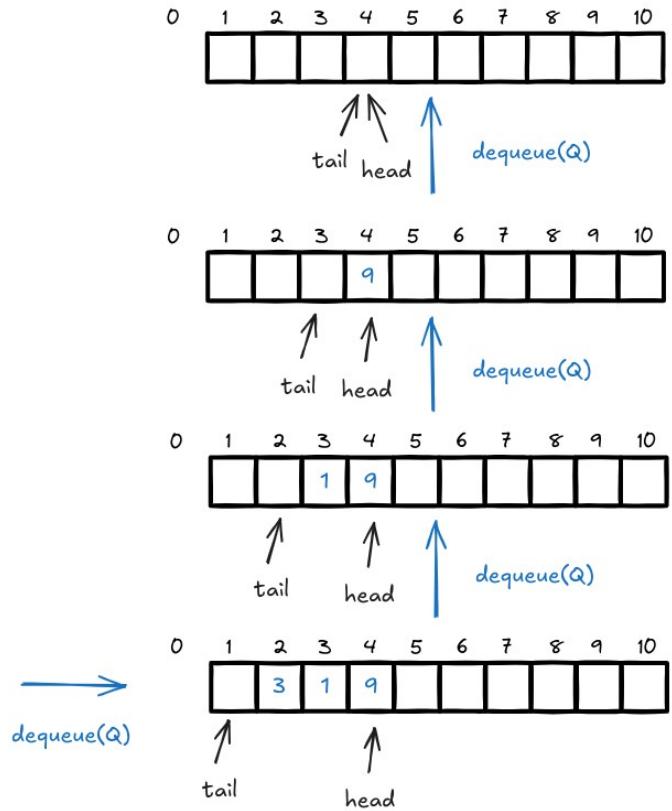
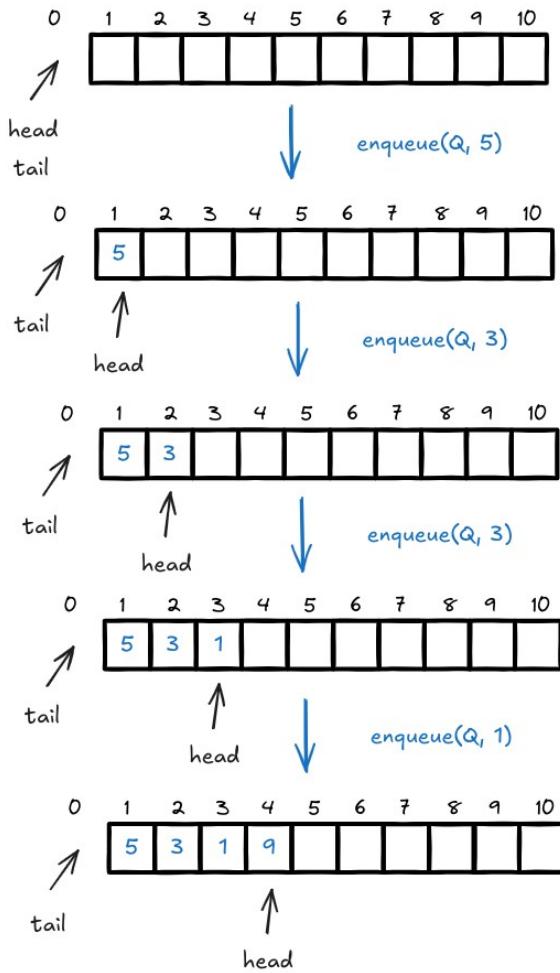
dequeue(Q) {
    if is_empty(Q)
        error('underflow')
    else {
        if Q.head == n      # para tornar a fila circular
            Q.head = 1
        else
            Q.head += 1
    }
    key = Q.keys[Q.head]
    Q.size -= 1
    return key
}

```

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 $O(n \log n)$.

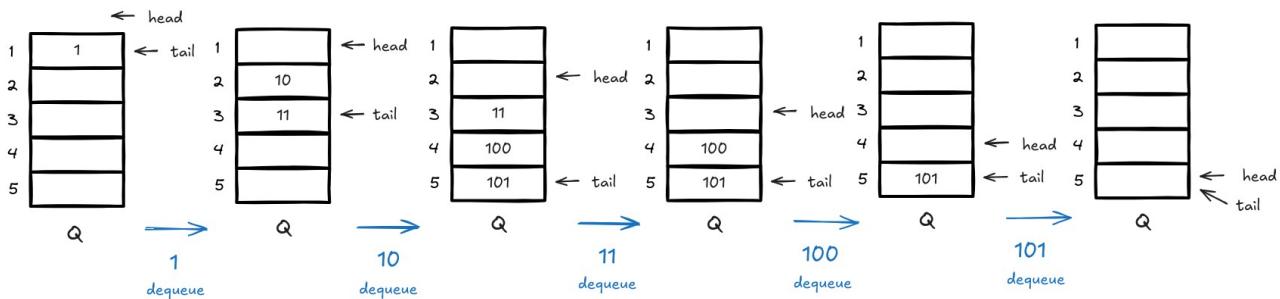
Veremos agora uma solução mais eficiente para esse problema.

```

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
    }
}

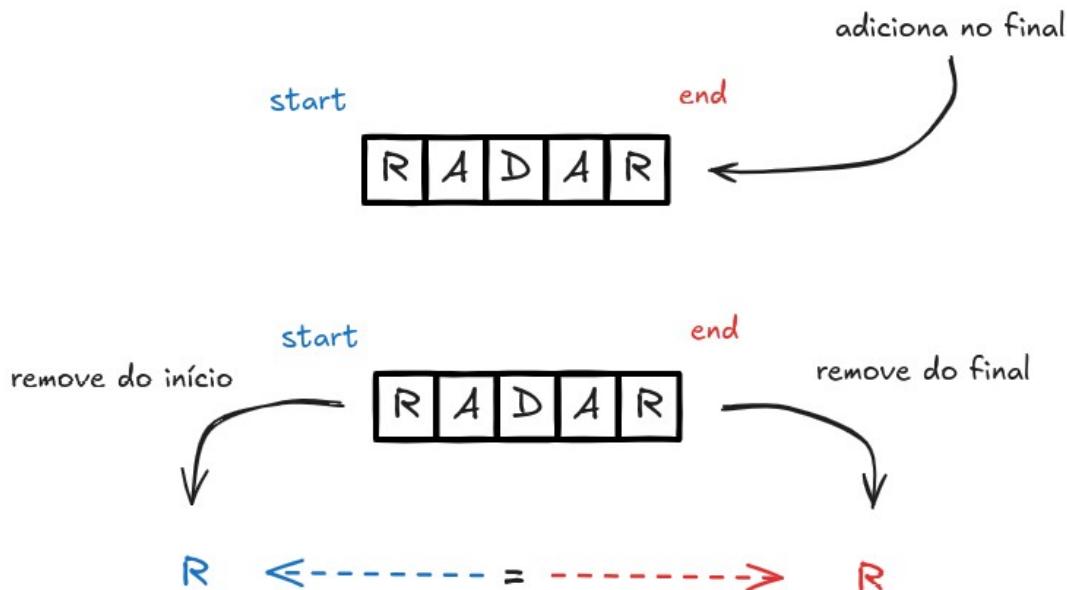
is_empty(D) {
    if D.size == 0
        return True
    else
        return False
}

// 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 até 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

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

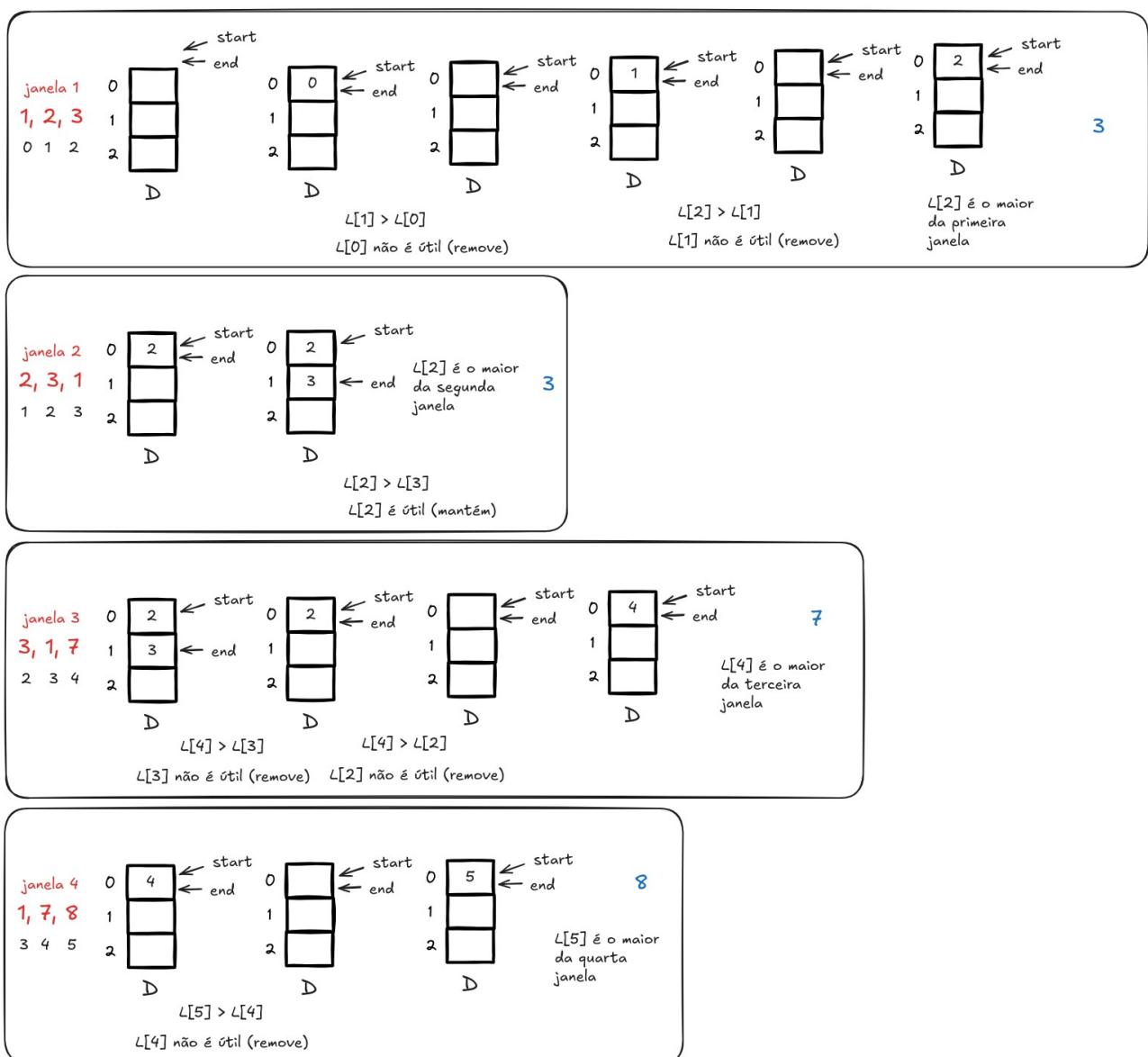
$L = [1, 2, 3, 1, 7, 8]$

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

1. 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.
2. No segundo dia, o disco de tamanho 5 será dado, então agora os discos de tamanhos 5 e 4 podem ser colocados na torre.
3. 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.
4. 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:

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

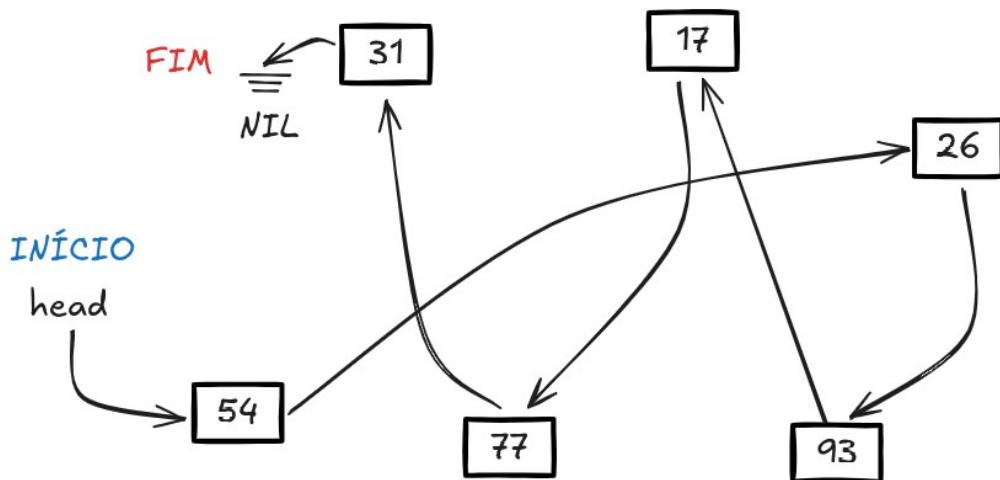
Veremos mais adiante que se implementarmos a fila de prioridades com um heap binário (min-heap ou max-heap), esse custo pode ser reduzido para $O(\log n)$.

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.

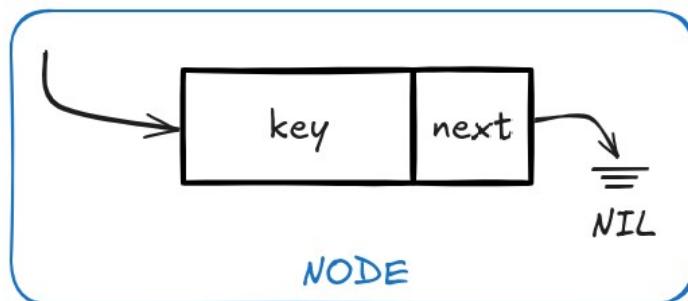


Primeiramente, antes de definirmos uma lista encadeada, devemos definir o bloco básico de construção de uma lista: o nó. Cada nó de uma lista encadeada deve conter duas informações: o dado propriamente dito e uma referência para o próximo nó da lista (para quem esse nó aponta). Podemos definir um TAD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

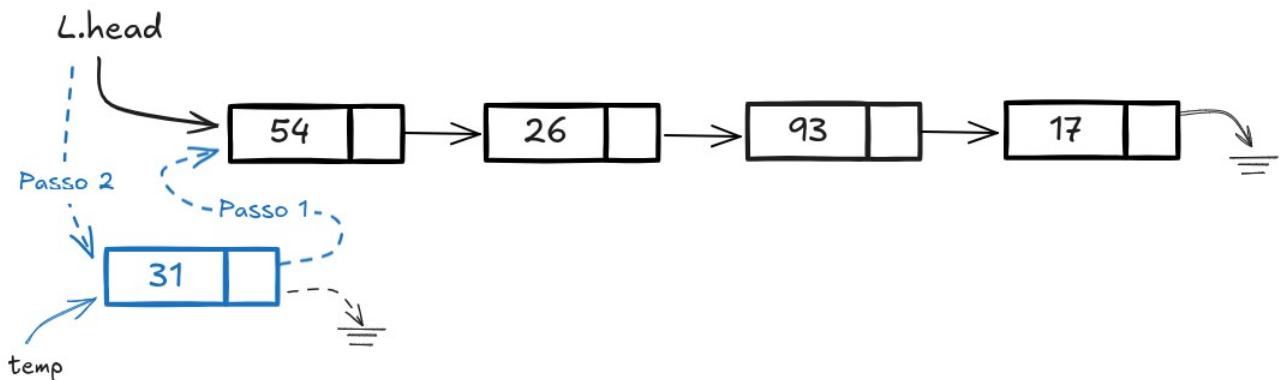
Em uma lista ordenada, a principal característica é que a inserção de um novo nó é feita sempre no início ou no final da lista, ou seja, os elementos do conjunto não encontram-se ordenados. Iniciaremos apresentando como criar uma lista encadeada não ordenada vazia. Adotaremos o seguinte construtor, em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

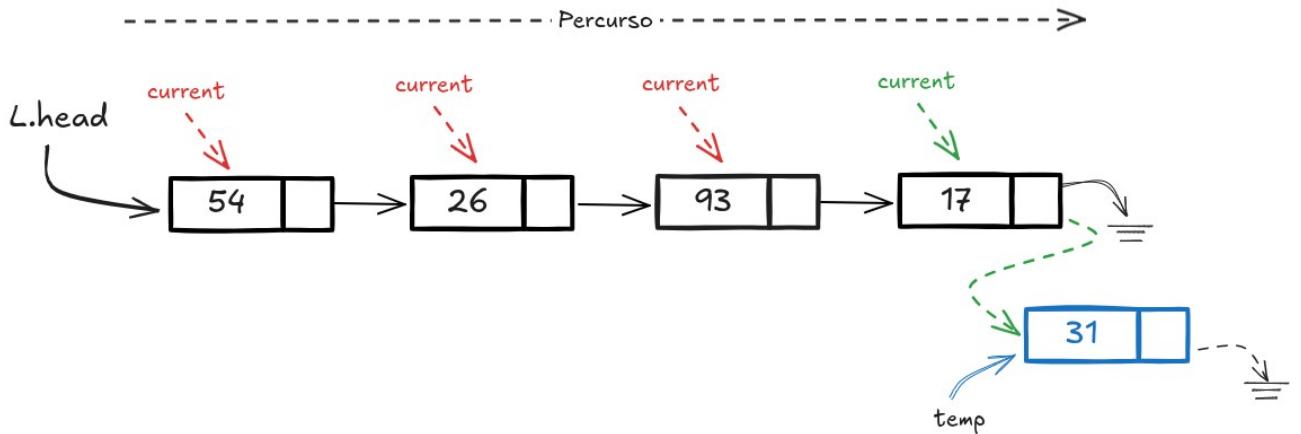


O algoritmo a seguir mostra como podemos implementar essa funcionalidade em uma lista encadeada L inicialmente vazia.

```
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.



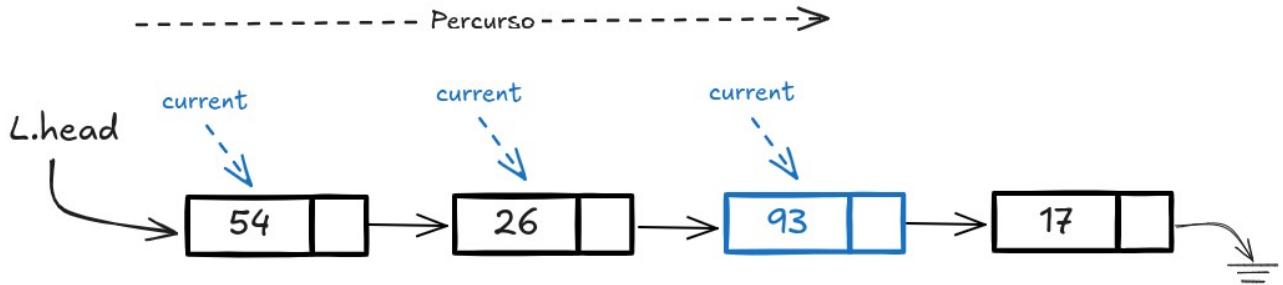
O algoritmo a seguir implementa essa funcionalidade.

```
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
}
```

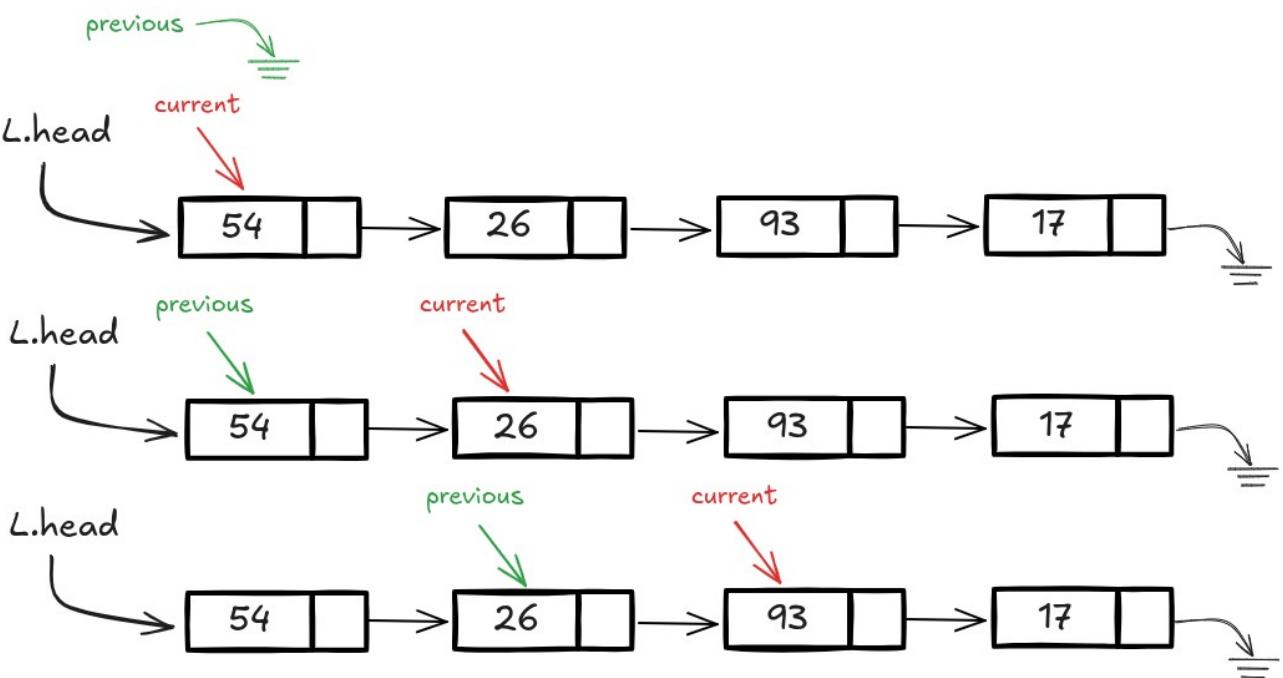
Note que, ao adotar a estratégia de incrementar ou decrementar o número de elementos da lista encadeada após cada inserção ou remoção, é possível saber o número de elementos em L com complexidade $O(1)$. Outra funcionalidade importante consiste em buscar um elemento na lista encadeada, ou seja, verificar 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.



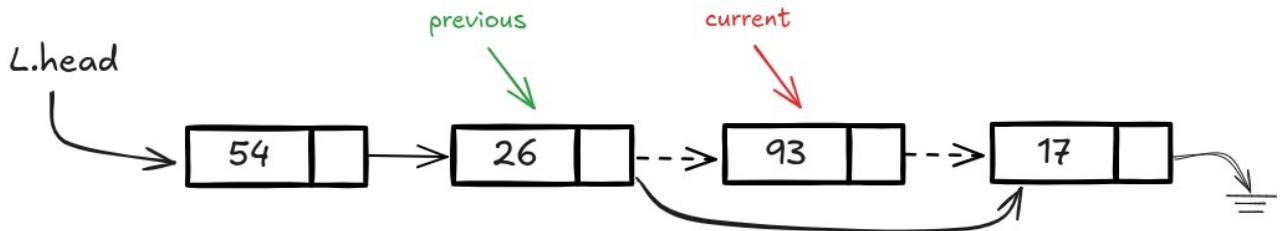
O algoritmo a seguir mostra uma implementação para essa funcionalidade.

```
# 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
}
```

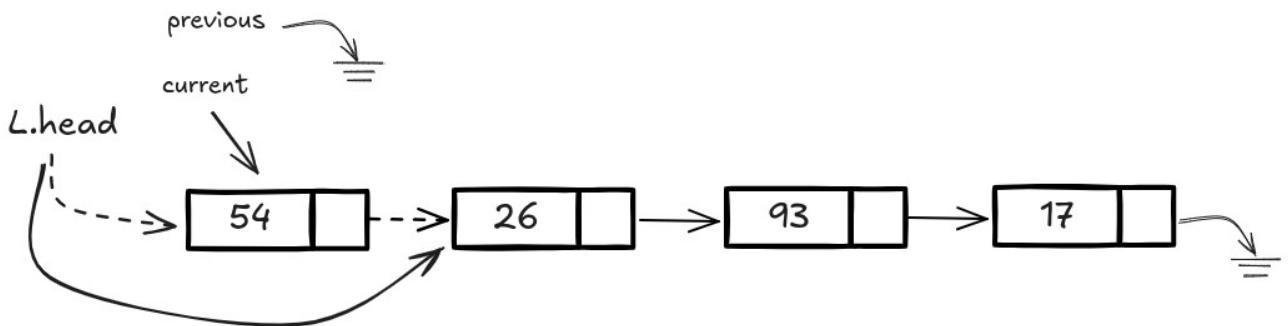
Por fim, uma operação importante é a remoção de um dado elemento da lista encadeada. Note que ao remover um nó da lista, precisamos religar o antecessor com o sucessor, de modo a evitar que elementos fiquem inacessíveis pela quebra do encadeamento sequencial. Primeiramente, precisamos ter duas referências 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
}

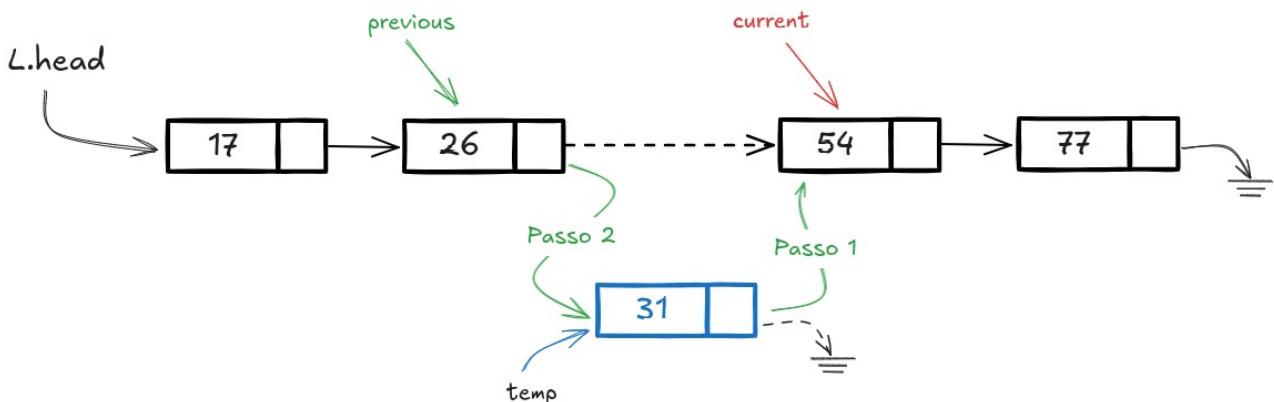
```

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).

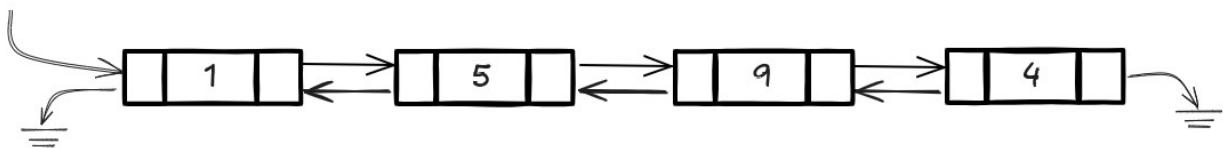
Listas duplamente encadeadas

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)

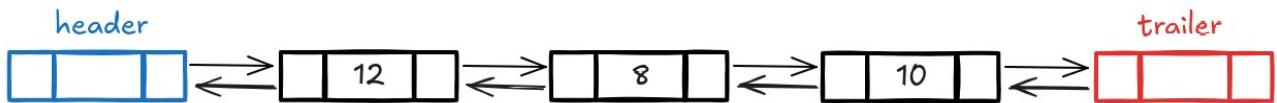
init(DL) {
    DL.header = new Node()      # cria sentinelas 1 (nó sem chave)
    DL.trailer = new Node()     # cria sentinelas 2 (nó sem chave)
    DL.header.prev = NIL
    DL.header.next = DL.trailer
```

```

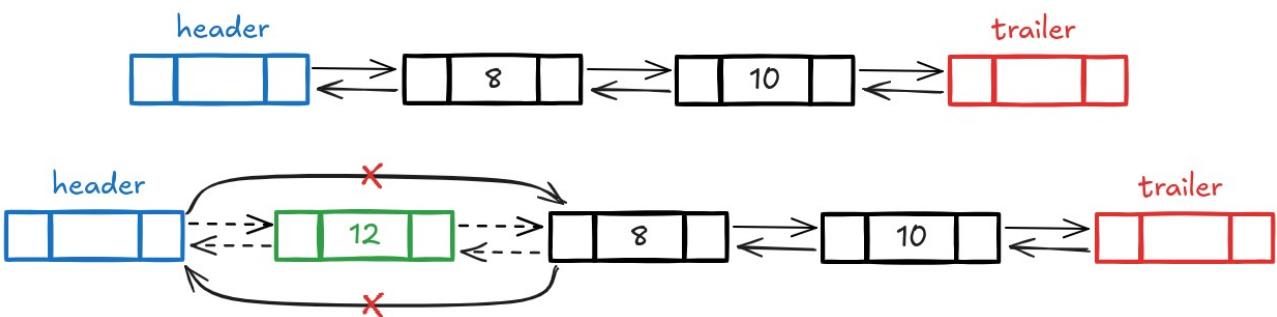
    DL.trailer.prev = DL.header
    DL.trailer.next = NIL
    DL.size = 0
}

```

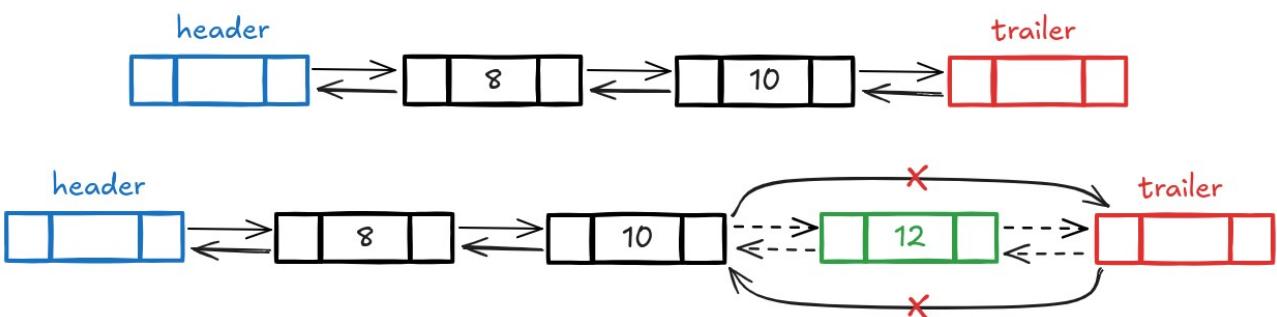
Assim como a lista encadeada possui uma cabeça (head) que sempre aponta para o início do encadeamento lógico, uma lista duplamente encadeada possui duas referências especiais: a própria cabeça, que chamaremos de header, e a cauda, que chamaremos de trailer. A figura a seguir ilustra a estrutura de uma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

```

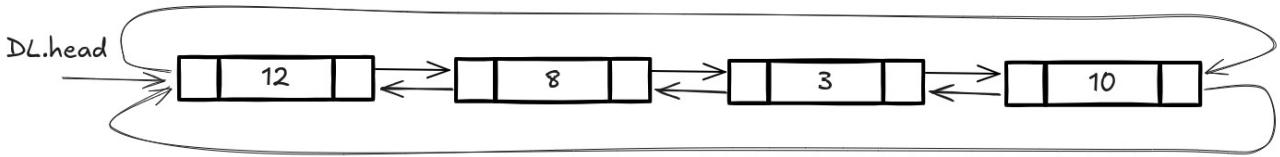
Note que diferentemente das listas encadeadas padrão, a inserção no final em uma lista duplamente encadeada possui complexidade $O(1)$, o que pode ser uma grande vantagem em diversas aplicações. A remoção de um nó da lista duplamente encadeada pode ser feita de maneira similar a inserção, usando uma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

```

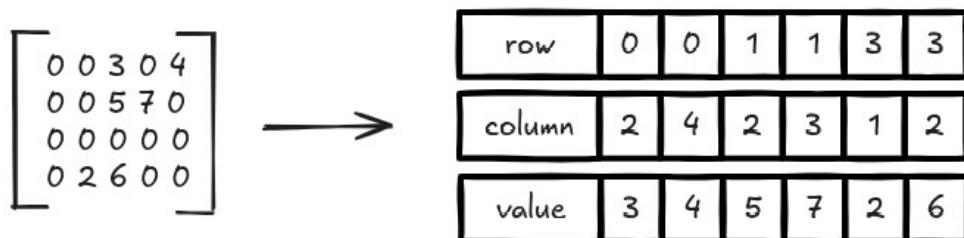
Note que dos 20 elementos, apenas 6 são não nulos, o que representa apenas 30% do total. Representar matrizes esparsas da forma tradicional, com um array bidimensional de n linhas por m colunas não é eficiente, pois há um enorme desperdício de memória.

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.

```

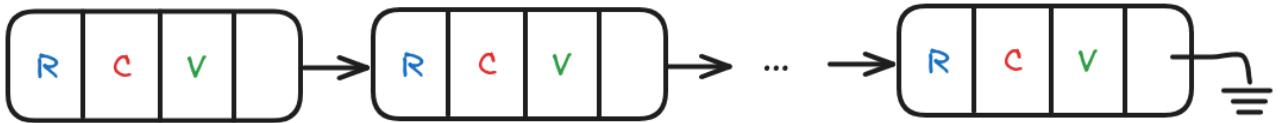
TAD Node
    int row
    int col
    float data
    Node next      # referência para um outro nó (encadeamento lógico)

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
}

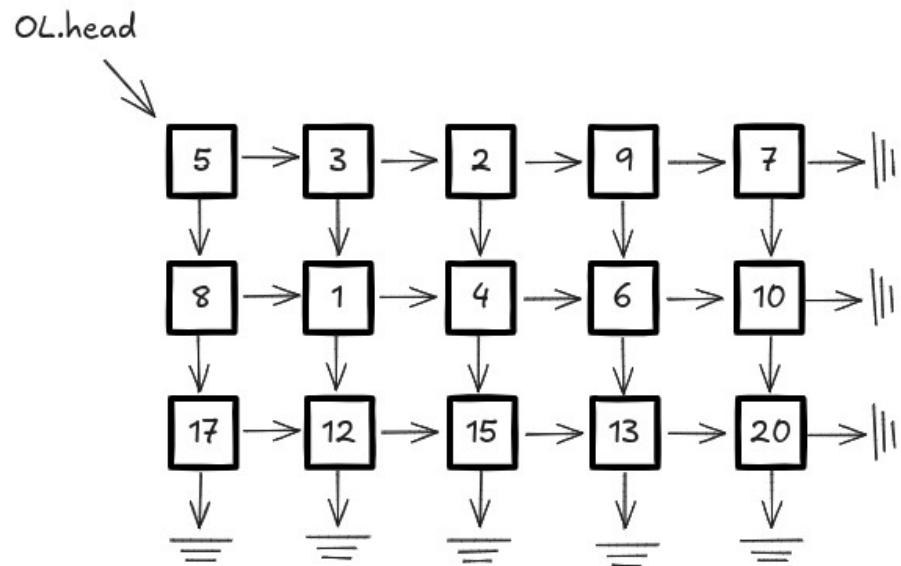
```

```

# 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 a 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:

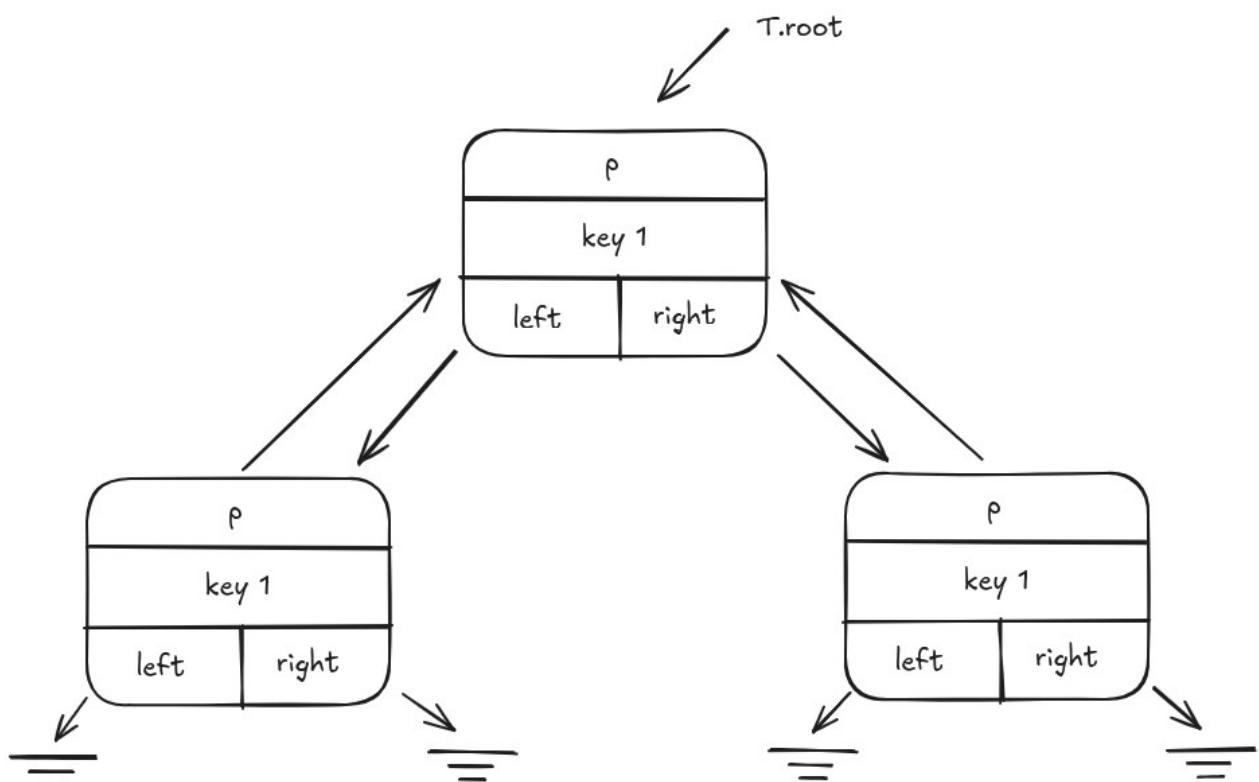
* **Chave (key)**: valor da ser armazenado em um nó de T

* **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.

```

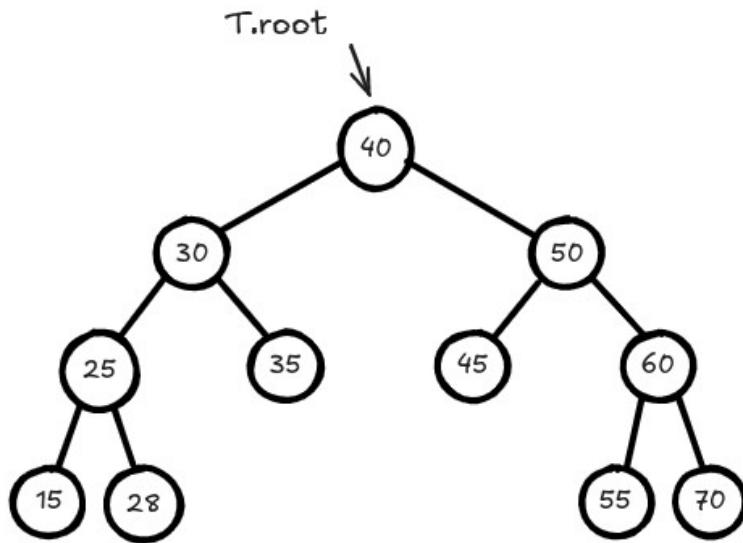
Tree_Walk_Inorder(x) {
    if x ≠ NIL {
        Tree_Walk_Inorder(x.left)
        print(x.key)
        Tree_Walk_Inorder(x.right)
    }
}

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

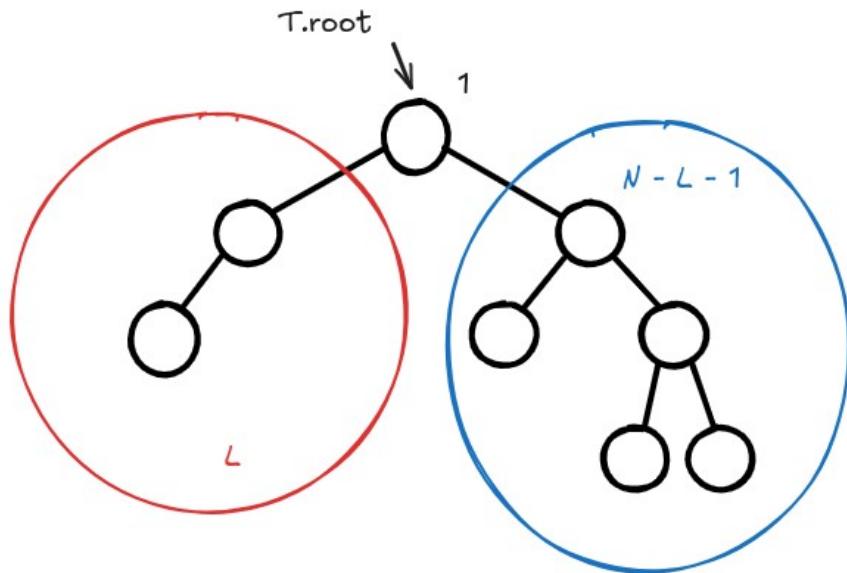
=> **Percorso postorder:** 15, 28, 25, 35, 30, 45, 55, 70, 60, 50, 40

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
}
}

```

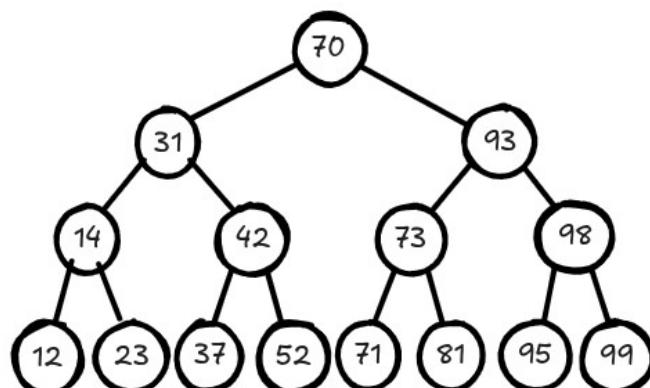
Árvores binárias de busca

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
}

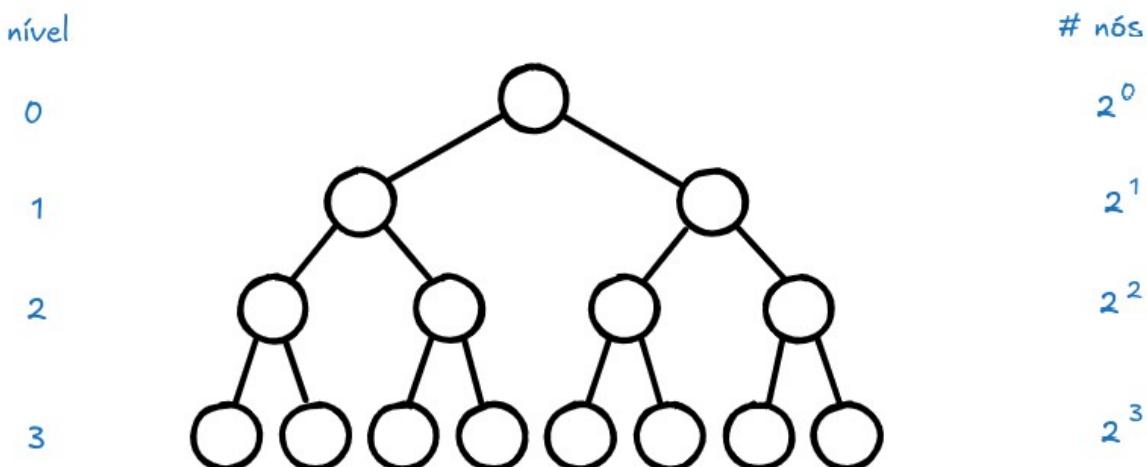
# Passar o nó raiz como parâmetro
Tree_Maximum(x) {
    while x.right ≠ NIL
        x = x.right
    return x
}
```

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
}
```

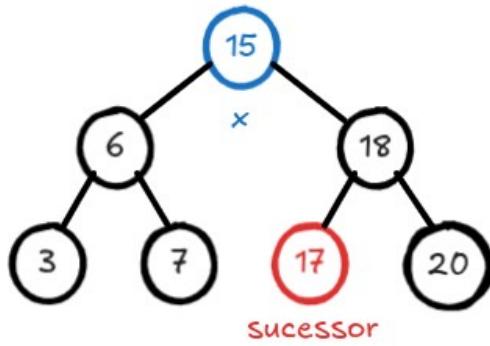
Sucessor e predecessor

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.

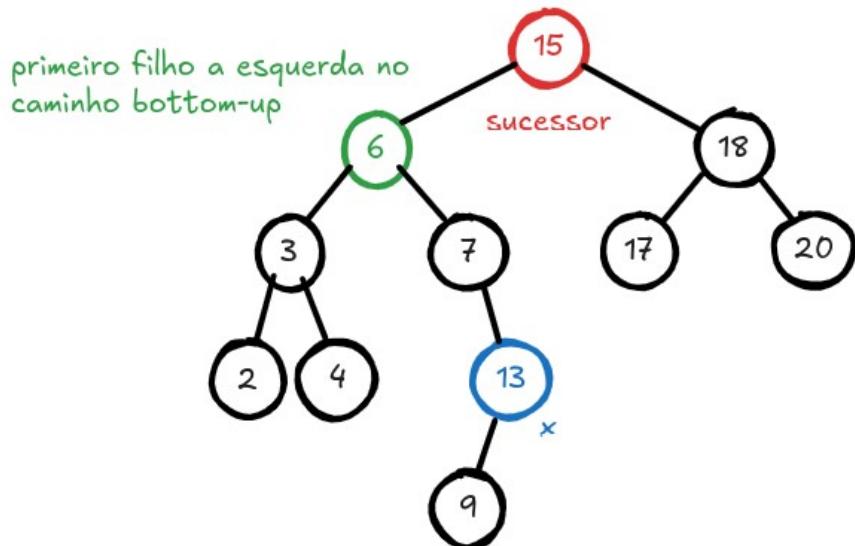
Def: O sucessor de um nó x é o nó y de menor chave tal que $y.key > x.key$, ou seja, é o próximo nó a ser visitado no percurso inorder.

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.



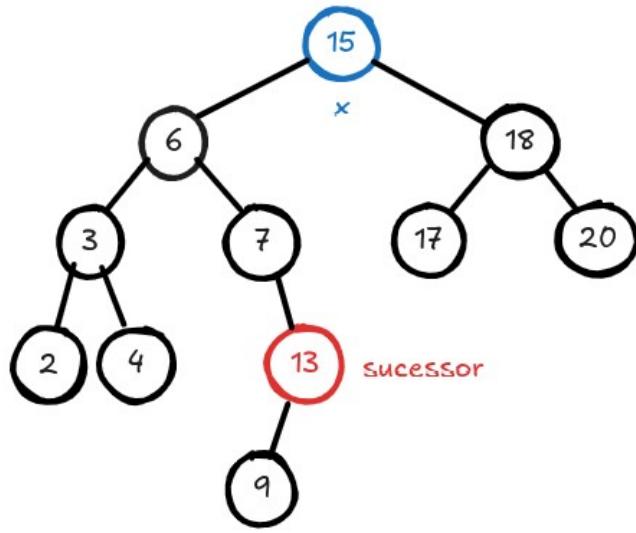
O algoritmo a seguir encontra o sucessor de um nó x em uma árvore binária de busca.

```
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
    }
}
```

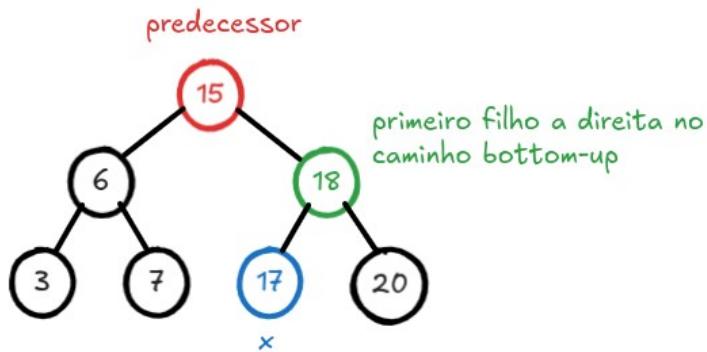
Note que a complexidade desse algoritmo também depende da altura h da árvore. Portanto, ela pode variar de $O(\log n)$ a $O(n)$.

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

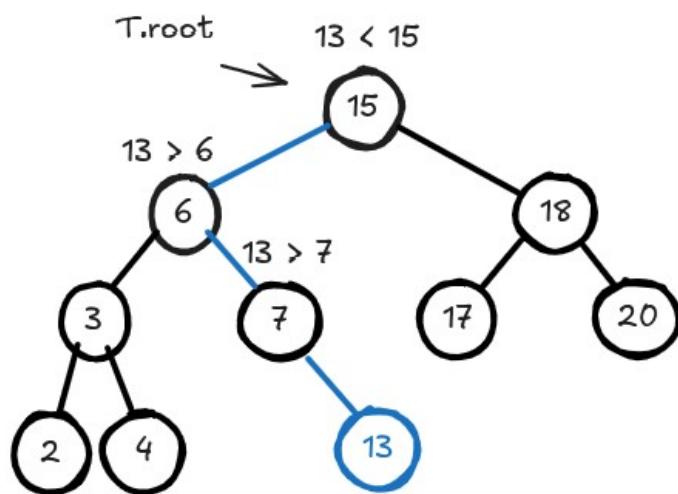
Para inserir um novo nó em uma árvore binária de busca não basta encontrar a primeira posição disponível. Devemos respeitar a propriedade chave: isso faz com que cada chave tenha sua posição correta dentro da estrutura. A função a seguir insere um novo nó z na árvore. Primeiro, devemos encontrar a posição 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                      # y aponta para o pai de 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
    }
}

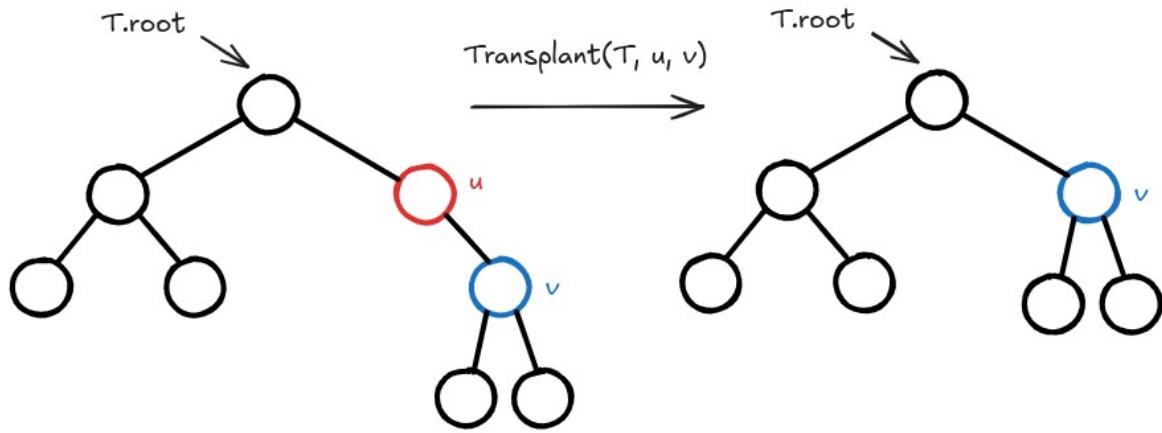
```

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.

```

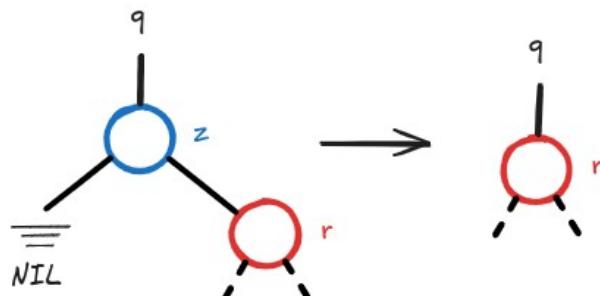
Transplant(T, u, v) {
    if u.p == NIL                      # nó u a ser substituído é a raiz de T
        T.root = v
    elif u == u.p.left {                 # u é filho a esquerda de alguém
        u.p.left = v
        u.left = NIL
    }
    else {                                # u é filho a direita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

```

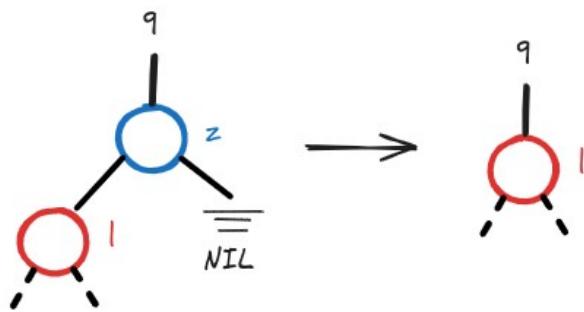
Seja z o nó a ser removido de T . A estratégia para remover um nó z da árvore T baseia-se na análise de 2 casos principais:

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)

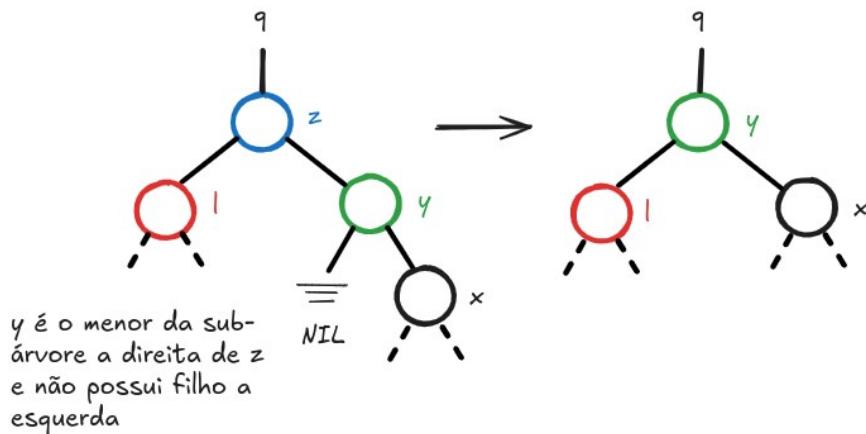


b) z tem ambos os filhos

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

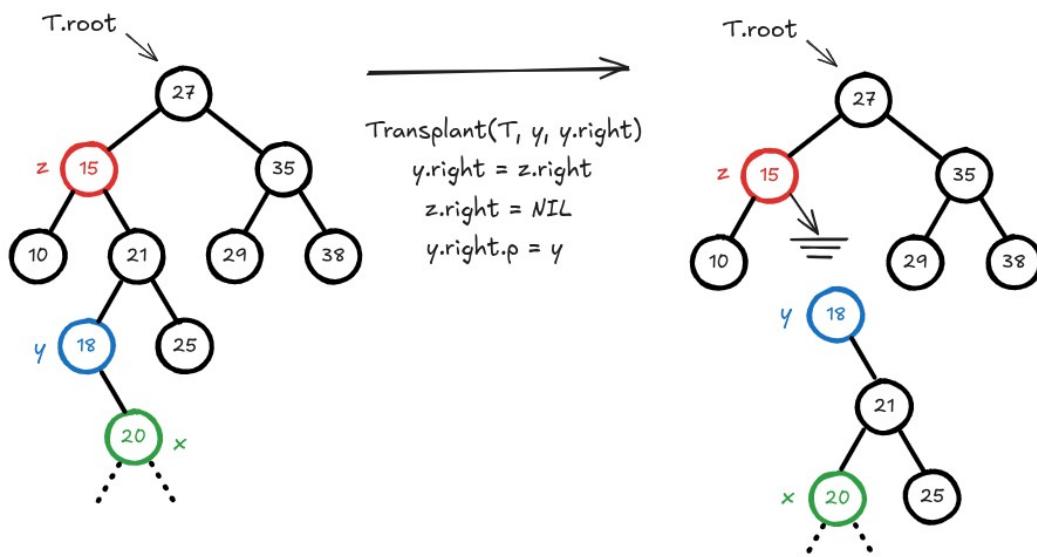
b1) y é o filho a direita de z

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

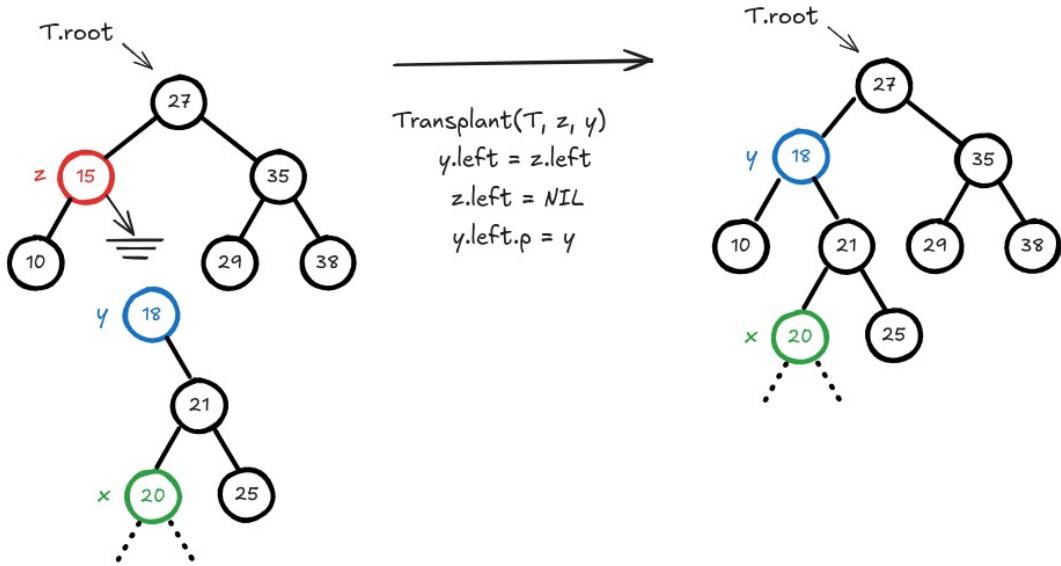


b2) y não é filho a direita de z (sucessor está lá embaixo na árvore T)

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
         $y.left.p = y$ 
    }
}

```

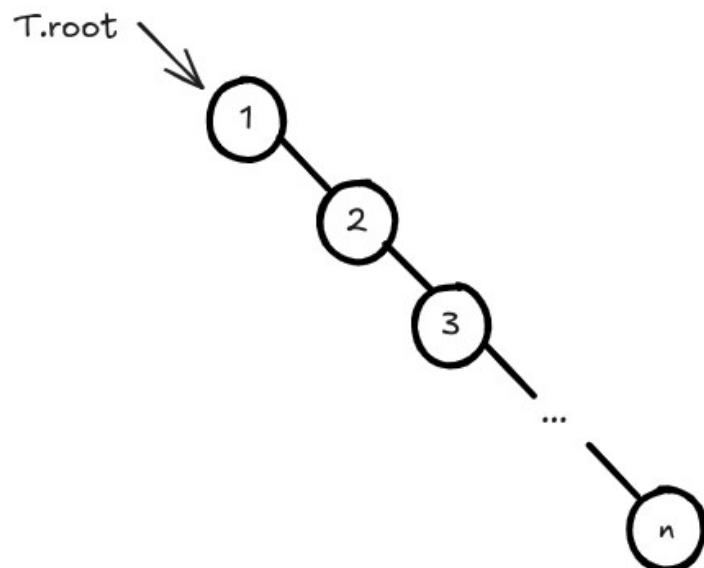
"Não diminua os outros para se sentir superior, melhore a si mesmo."
 (Autor anônimo)

Estruturas de Dados: Árvores AVL (Balanceadas)

Vimos anteriormente que as operações de busca, inserção e remoção de nós em árvores binárias de busca possuem complexidade $O(h)$, em que h é a altura da árvore T .

Um dos problemas com as árvores binárias de busca é que seu padrão de crescimento pode se tornar bastante irregular, no sentido de que, após a criação da raiz da árvore, todos os elementos inseridos à árvore são maiores que o anterior. Isso faz com que a árvore se degenera para uma lista encadeada.

Claramente, a situação descrita é um caso extremo, mas não é incomum que alguns ramos da árvore cresçam muito mais que outros, tornando a árvore completamente desbalanceada, o que faz com que as operações de inserção, pesquisa e remoção de nós sejam menos eficientes.



Para contornar esse problema, foram propostas as árvores AVL, em homenagem aos seus criadores, Georgy Adelson-Velsky e Landis. Essas árvores também são conhecidas como árvores binárias de busca balanceadas. Para cada nó x definimos o fator de balanço como:

$$x.b = h_L - h_R$$

onde h_L é a altura da subárvore a esquerda e h_R é a altura da subárvore a direita.

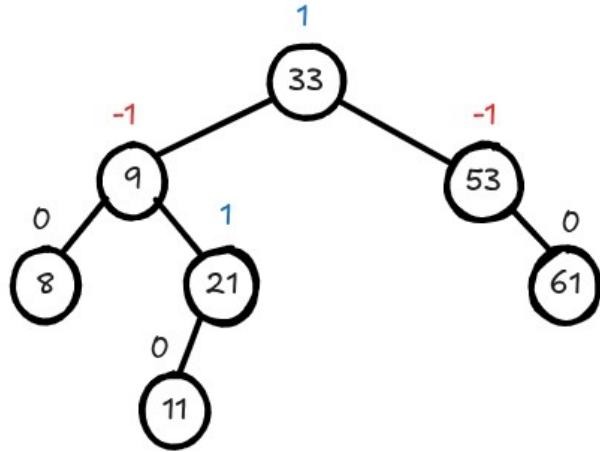
Def: Uma árvore T é dita AVL ou balanceada se:

$$\forall x \in T (x.b \in \{-1, 0, 1\})$$

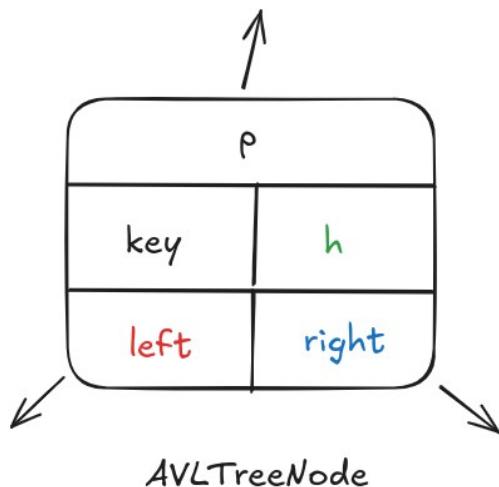
ou seja, não é permitido que nenhum nó tenha fator de balanço maior que 1 ou menor que -1.

A figura a seguir ilustra um exemplo de árvore AVL.

Para garantir que após a inserção ou remoção de um nó, a árvore em questão seja balanceada, devemos executar algumas operações de modo a garantir que o fator de balanço dos nós esteja entre -1 e +1. A seguir discutimos um pouco dessas operações.



Para otimizar os cálculos das alturas dos nós da árvore AVL é comum criarmos um campo h para armazenar essa informação e evitar o recálculo. Assim, definimos o nó de uma árvore AVL como:



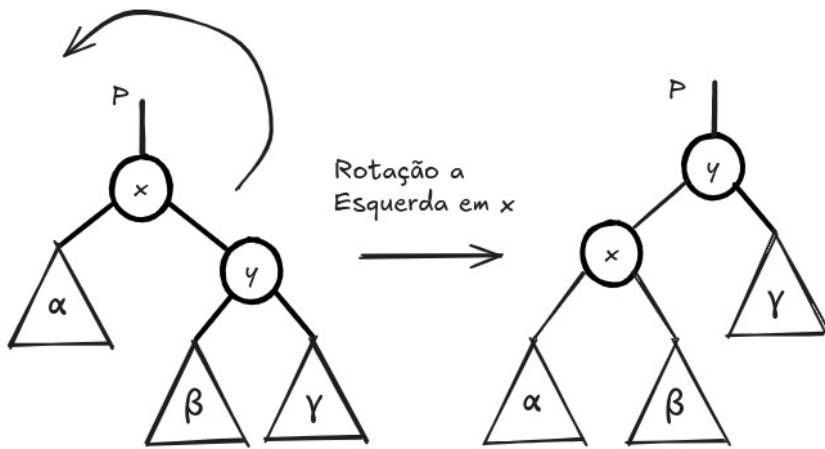
```
# Função auxiliar que retorna a altura do nó z
# Altura é o número de nós no maior caminho da raiz a um nó folha
height(z) {
    if z == NIL
        return 0
    else
        return z.h
}
```

Operações em árvores AVL

Se uma árvore AVL torna-se desbalanceada após a remoção ou inserção de um nó, devemos executar uma operação chamada rotação. Há dois tipos básicos de rotação: rotação a esquerda e rotação a direita. É importante mencionar que tais operações preservam a propriedade chave das árvores binárias de busca.

Rotação a esquerda

Seja x um nó desbalanceado com $x.b < -1$, conforme indica a figura a seguir (a altura da subárvore a direita é maior que a altura da subárvore a esquerda). Após uma rotação a esquerda em x , a altura do nó x na árvore é decrementada.

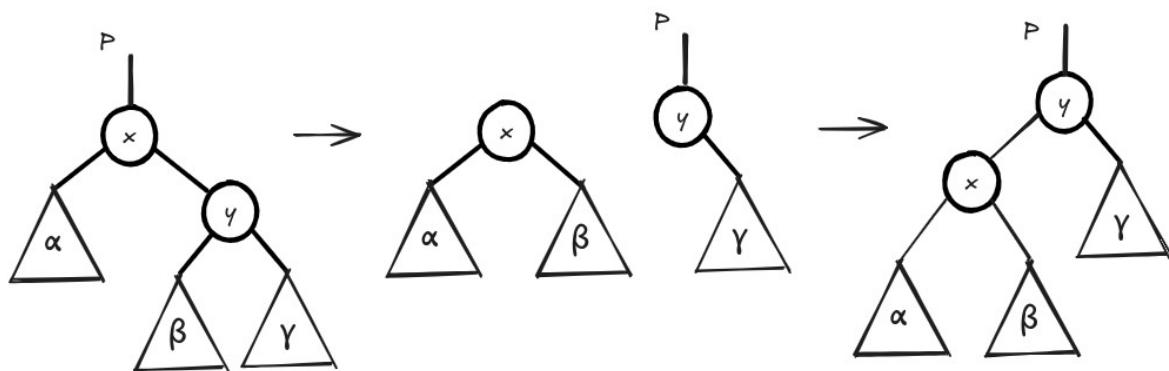


A função leftRotate a seguir mostra os passos lógicos dessa operação.

```
leftRotate(x) {
    y = x.right
    β = y.left
    x.right = β
    β.p = x
    y.left = x
    y.p = x.p
    if x == x.p.left
        x.p.left = y
    else
        x.p.right = y
    x.p = y
    x.h = 1 + max(height(x.left), height(x.right))
    y.h = 1 + max(height(y.left), height(y.right))
    return y
}
```

Note que em resumo temos os seguintes passos:

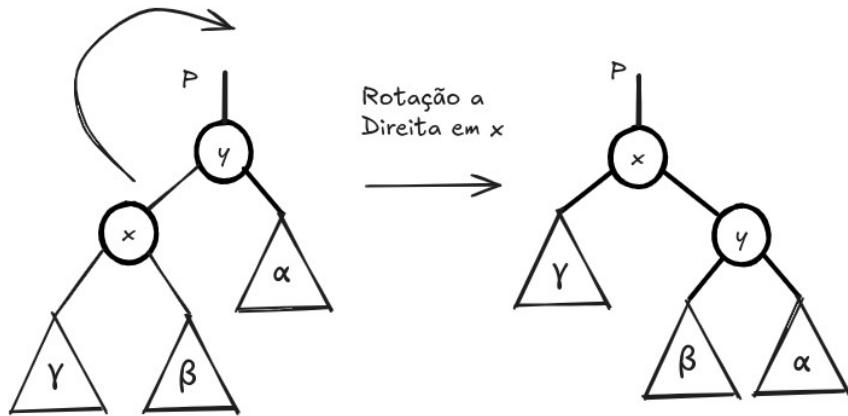
1. Se y possui uma subárvore a esquerda β , então o filho a direita de x deve apontar para β .
2. Se x é a raiz da árvore, então faça y ser a nova raiz da árvore. Senão, se x é o filho a esquerda do nó p , faça y ser o filho a esquerda de p . Senão, y deve ser o filho a direita de p .
3. Faça y ser o pai de x .



Note que as chaves em α permanecem menores que x , as chaves de β permanecem menores que y e maiores que x e as chaves de γ permanecem maiores que y (a propriedade chave é mantida!)

Rotação a direita

Seja y um nó desbalanceado com $y.b > 1$, conforme indica a figura a seguir (a altura da subárvore a esquerda é maior que a altura da subárvore a direita). Após uma rotação a direita em y , a altura do nó y na árvore é decrementada.

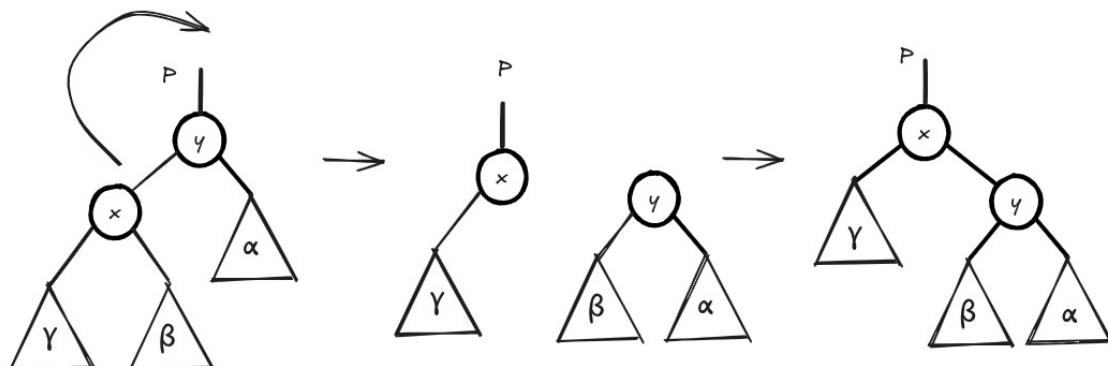


A função rightRotate a seguir mostra os passos lógicos dessa operação.

```
rightRotate(y) {
    x = y.left
    β = x.right
    y.left = β
    β.p = y
    x.right = y
    x.p = y.p
    if y == y.p.left
        y.p.left = x
    else
        y.p.right = x
    y.p = x
    y.h = 1 + max(height(y.left), height(y.right))
    x.h = 1 + max(height(x.left), height(x.right))
    return x
}
```

Note que em resumo temos os seguintes passos:

1. Se x possui uma subárvore a direita β , então o filho a esquerda de y deve apontar para β .
2. Se y é a raiz da árvore, então faça x ser a nova raiz da árvore. Senão, se y é o filho a direita do nó p , faça x ser o filho a direita de p . Senão, x deve ser o filho a esquerda de p .
3. Faça x ser pai de y .

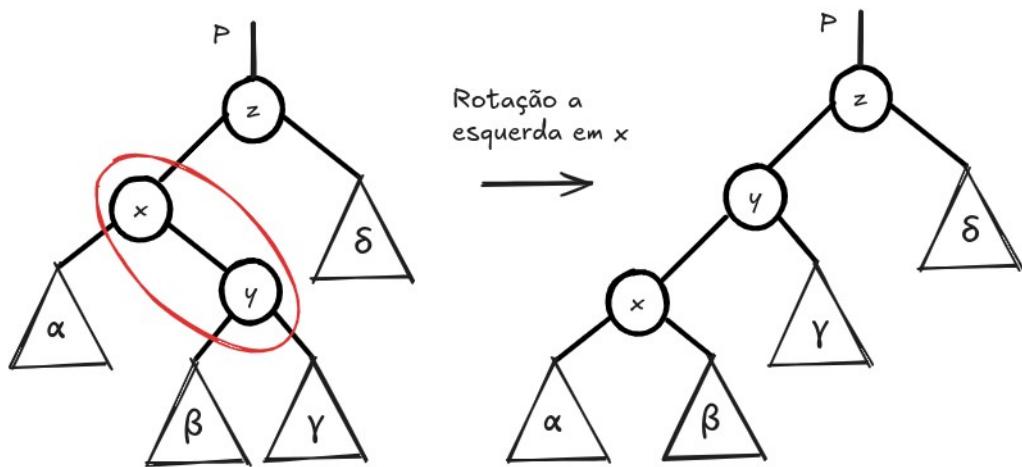


Note que após a rotação as chaves em α permanecem maiores que y , as chaves de β permanecem menores que y e maiores que x e as chaves de γ permanecem menores que y (a propriedade chave é mantida!)

Em alguns casos mais complexos, pode ser necessário realizar duas rotações básicas em sequência após a inserção ou remoção de um nó na árvore AVL: são as operações de rotação esquerda-direita e rotação direita-esquerda, que são definidas a seguir.

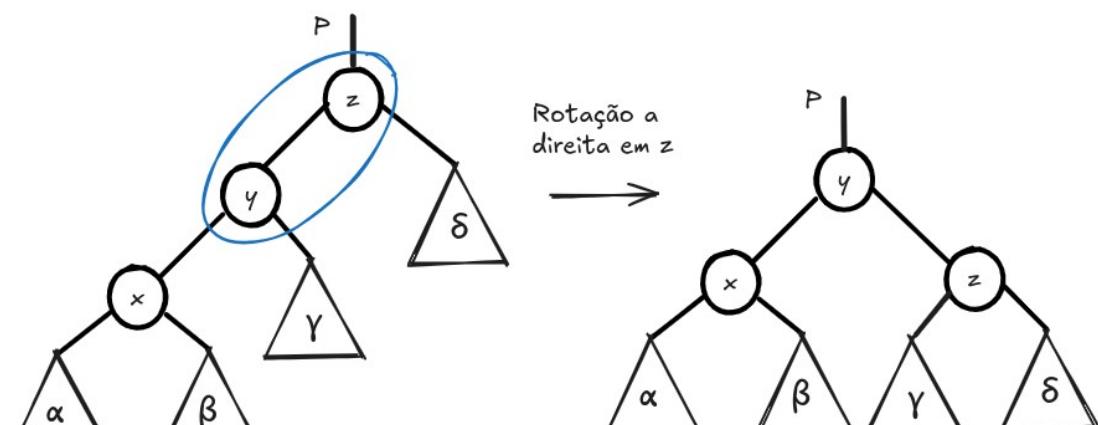
Rotação esquerda-direita

- Como o nome sugere, primeiro devemos realizar uma rotação a esquerda em x .



Puxando o y para cima de x

- Em seguida, devemos realizar uma rotação a direita em z .



Puxando y para cima de z

A função a seguir ilustra os passos para uma rotação esquerda-direita.

```

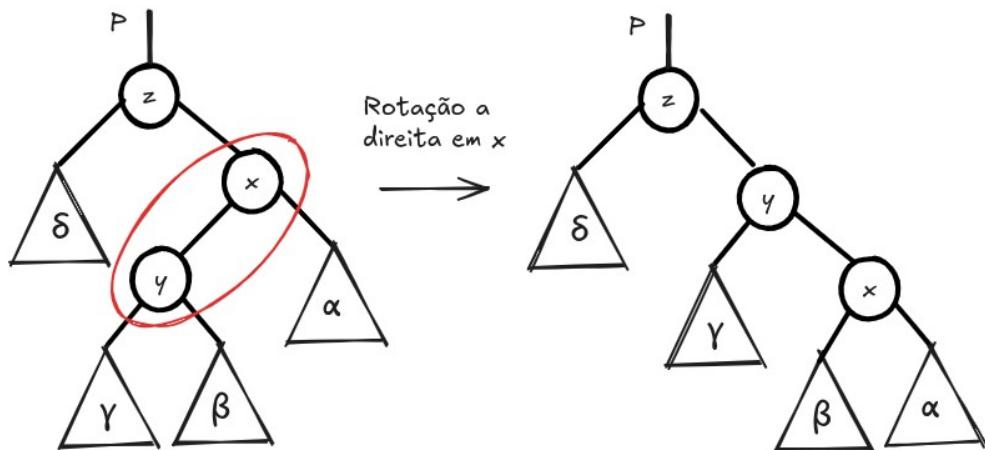
leftRightRotate(z) {
    x = z.left
    u = leftRotate(x)
    w = rightRotate(z)
    return w
}

```

De forma análoga, podemos realizar uma rotação direita-esquerda.

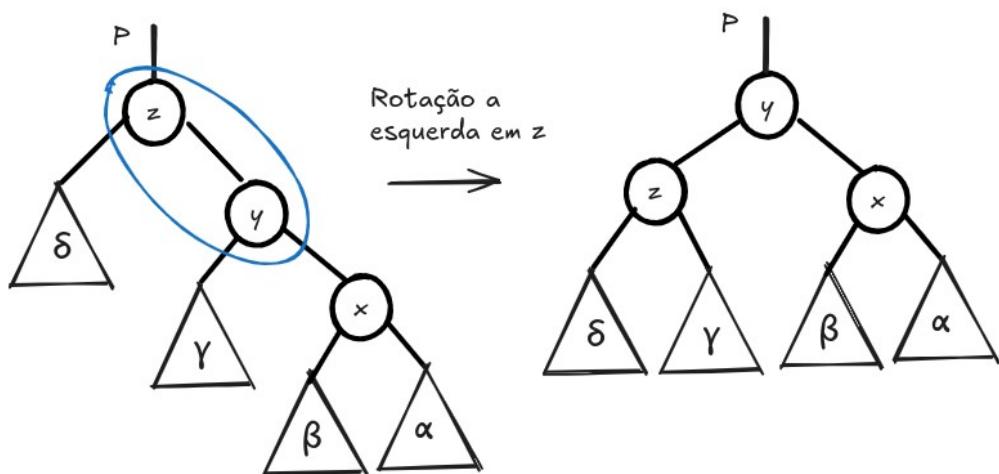
Rotação direita-esquerda

- Novamente, como o nome sugere, primeiro devemos realizar uma rotação a direita em x.



Puxando o y para cima de x

- Em seguida, devemos realizar uma rotação a esquerda em z.



Puxando y para cima de z

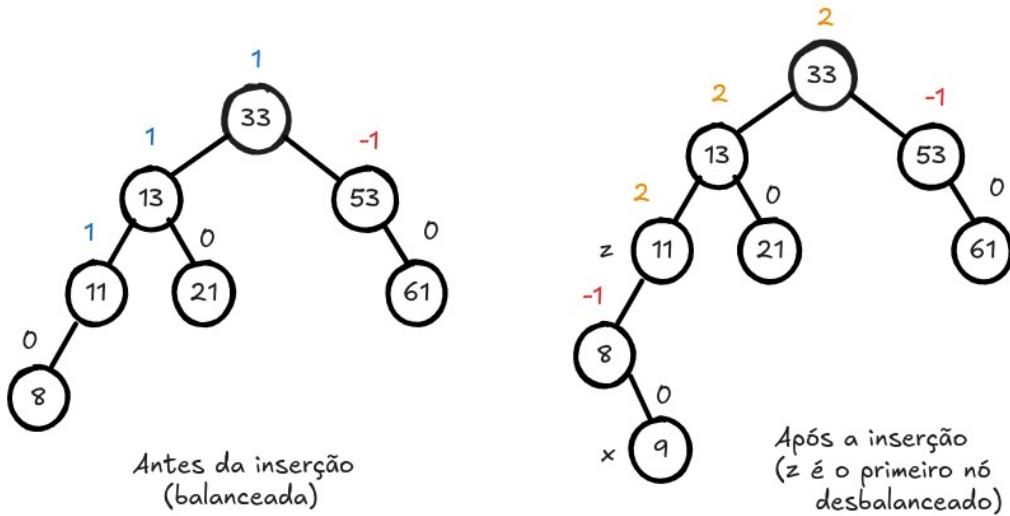
```

rightLeftRotate(z) {
    x = z.right
    u = rightRotate(x)
    w = leftRotate(z)
    return w
}

```

Inserção de nós em árvores AVL

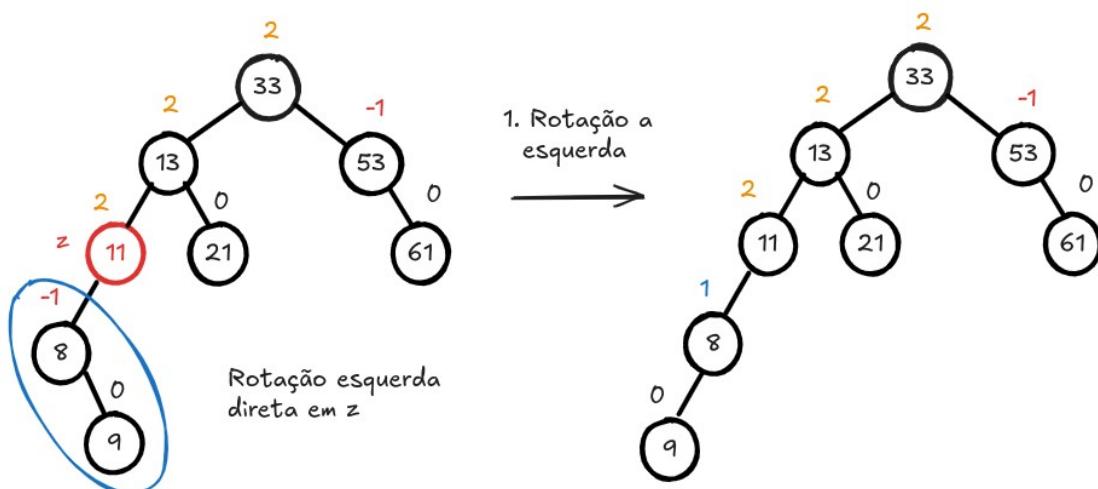
Um novo nó inserido na árvore AVL será sempre uma folha, que possui fator de balanço nulo. Porém, ele pode tornar algum outro nó desbalanceado, requerendo assim uma operação de rotação. Considere por exemplo, a inserção da chave 9 na árvore AVL a seguir.

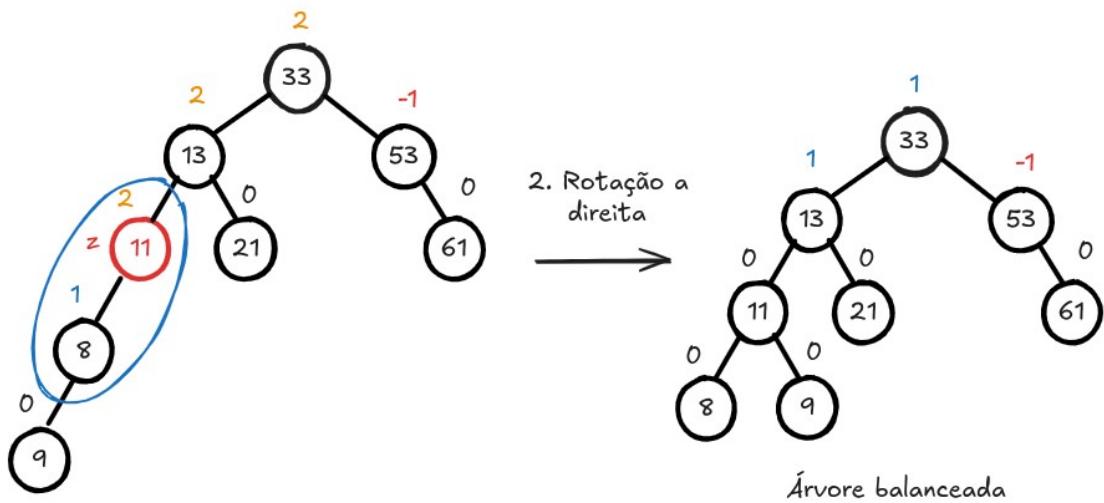


A inserção de um único nó desbalanceou vários outros nós. Como proceder? A operação de rotação deve ser aplicada no nó desbalanceado mais próximo. Partindo do princípio que todo nó inserido é uma folha da árvore (denotaremos de x), devemos percorrer um caminho bottom-up até encontrar o nó desbalanceado mais próximo ou chegar em NIL. Para o caso de fator de balanço positivo, deve-se aplicar a seguinte regra (supondo que z é o primeiro nó desbalanceado no caminho bottom-up):

```
if z.b > 1 {
    if x.key < z.left.key
        rightRotate(z)
    else
        leftRightRotate(z)
}
```

Após o balanceamento, a árvore AVL resultante encontra-se perfeitamente平衡ada.





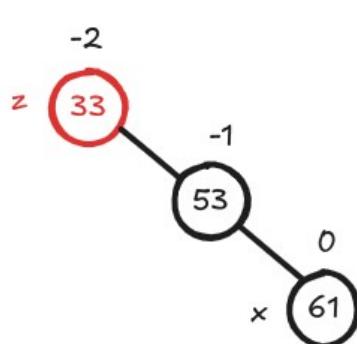
Analogamente, temos o caso simétrico, quando o fator de balanço é negativo.

```

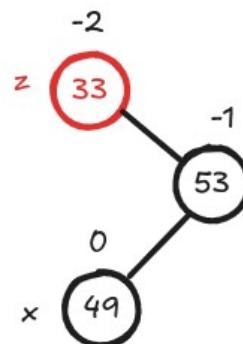
if z.b < -1 {
    if x.key > z.right.key
        leftRotate(z)
    else
        rightLeftRotate(z)
}

```

A figura a seguir ilustra um exemplo simples.



Rotação a esquerda



Rotação direta-esquerda

Sendo assim, a operação completa de rebalanceamento de uma árvore AVL pode ser realizada de acordo com o algoritmo a seguir. A função auxiliar balance retorna o fator de balanço de um nó.

```

# Função para retornar o fator de balanço de um nó
balance(z) {
    if z == NIL
        return 0
    else
        return height(z.left) - height(z.right)
}
# Função recursiva para inserir um nó em uma árvore AVL

```

```

# Deve-se passar T.root como parâmetro em node
AVL_Tree_Insert(node, key) {
    if node == NIL {
        node = AVLTreeNode(key)
        return node
    }
    # Na ida da recursão, caminho TOP-DOWN
    if key < node.key
        node.left = AVL_Tree_Insert(node.left, key)
    elif key > node.key
        node.right = AVL_Tree_Insert(node.right, key)
    else # se chave já existem em T, não insere novamente
        return node
    # Caminho BOTTOM-UP iniciando no pai do nó inserido
    # Na volta da recursão
    # atualiza as alturas dos antecessores
    node.h = 1 + max(height(node.left), height(node.right))
    # calcula o fator de balanço
    b = balance(node)
    # Temos que analisar 4 casos possíveis
    # Caso 1: RE
    if b > 1 and key < node.left.key
        return rightRotate(node)
    # Caso 2: RD
    if b < -1 and key > node.right.key
        return leftRotate(node)
    # Caso 3: RED
    if b > 1 and key > node.left.key
        return leftRightRotate(node)
    # Caso 4: RDE
    if b < -1 and key < node.right.key
        return rightLeftRotate(node)
    # Se estiver balanceado, retorna o nó sem modificações
    return node
}

```

Pode-se mostrar que, após uma inserção, são necessárias no máximo 2 rotações para restaurar o balanço da árvore AVL.

Remoção de nós em árvores AVL

Assim como vimos anteriormente, durante a remoção de um nó em uma árvore binária de busca, há 3 casos principais a serem observados:

1. O nó a ser removido é uma folha
2. O nó a ser removido possui um único filho
3. O nó a ser removido possui 2 filhos

Por conveniência, denominaremos de w o nó a ser removido da árvore.

Caso 1: após a remoção, devemos apenas checar o fator de balanço dos antecessores.

Iniciando no pai de w (denotado por y), devemos percorrer um caminho ascendente até encontrar o primeiro nó cujo fator de balanço é maior que 1 ou menor que -1 (note que tal nó pode ser o próprio pai de w, ou seja y) ou chegar em NIL. Denominaremos esse primeiro nó desbalanceado de z.

Caso 2: devemos substituir o nó removido pelo seu único filho e checar o fator de balanço dos antecessores.

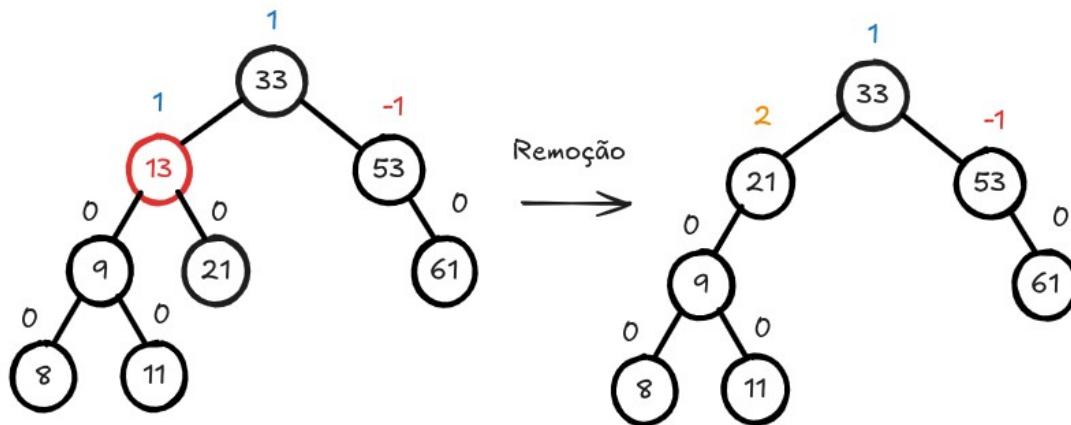
Iniciando no nó substituído, denotado por y , devemos percorrer um caminho ascendente até encontrar o primeiro nó desbalanceado z .

Caso 3: devemos encontrar o sucessor do nó na árvore e como ele sempre terá um único filho (menor chave da subárvore a direita), podemos substituir o nó removido por ele. Em seguida, devemos checar o fator de balanço dos antecessores.

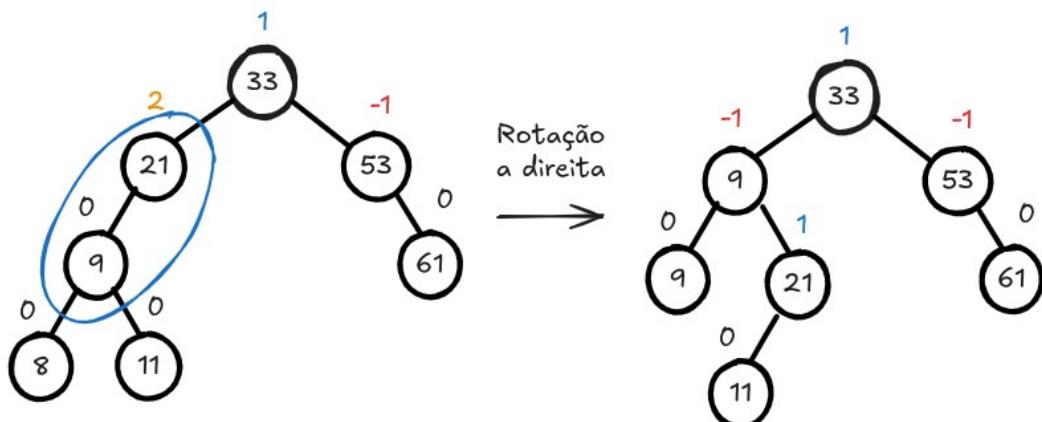
Iniciando no nó substituído, denotado por y , devemos percorrer um caminho ascendente até encontrar o primeiro nó desbalanceado z . A regra para o rebalanceamento da árvore consiste em rebalancear o ancestral mais próximo como:

```
if z.b > 1 {
    if z.left.b ≥ 0
        rightRotate(z)
    else
        leftRightRotate(z)
}
```

As figuras a seguir ilustram um exemplo didático de remoção em árvores AVL.



Sendo assim, é preciso realizar uma rotação a direita, conforme ilustra a figura a seguir.



Para o caso simétrico, temos uma regra similar.

```

if z.b < -1 {
    if z.right.b ≤ 0
        leftRotate(z)
    else
        rightLeftRotate(z)
}

```

A função a seguir mostra como podemos realizar a remoção de um nó em uma árvore AVL de maneira recursiva.

```

AVL_Tree_Delete(root, key) {
    # Caminho TOP-DOWN para encontrar nó
    if root == NIL
        return root
    if key < root.key
        root.left = AVL_Tree_Delete(root.left, key)
    elif key > root.key
        root.right = AVL_Tree_Delete(root.right, key)
    else {
        # Se entrar aqui, achou o nó a ser removido!
        if root.left == NIL or root.right == NIL {
            if root.left ≠ NIL
                temp = root.left
            if root.right ≠ NIL
                temp = root.right
            # Se nó removido é folha
            if temp == NIL
                root = NIL
            else
                # Nó a ser removido tem um único filho
                root = temp
        }
        else {      # Nó a ser removido possui 2 filhos
            # Encontrar o sucessor (menor da subárvore a direita)
            temp = Tree_Minimum(root.right)
            root.key = temp.key
            # Remove o sucessor (sempre terá 1 único filho)
            root.right = AVL_Tree_Delete(root.right, temp.key)
        }
    }
    # Condição de parada da recursão (atingiu NIL)
    if root == NIL
        return root
    # Caminho BOTTOM-UP para平衡amento (volta da recursão)
    # Na volta da recursão, atualiza as alturas dos antecessores
    root.h = 1 + max(height(root.left), height(root.right))
    # Fator de balanço dos ancestrais
    b = balance(root)
    # Temos que analisar 4 casos possíveis
    # Caso 1: RD
    if b > 1 and balance(root.left) ≥ 0
        return rightRotate(node)
    # Caso 2: RED
    if b > 1 and balance(root.left) < 0
        return leftRightRotate(node)
}

```

```

# Caso 3: RE
if b < -1 and balance(root.right) ≤ 0
    return leftRotate(node)
# Caso 4: RDE
if b < -1 and balance(root.right) > 0
    return rightLeftRotate(node)
# Se estiver balanceado, retorna o nó sem modificações na recursão
return root

}

```

Existem outros tipos de árvores aproximadamente平衡adas que não precisam de operações de rotação. Os exemplos mais conhecidos são as árvores rubro negras e as árvores de busca digitais (Tries). No próximo capítulo, apresentaremos uma breve introdução às árvores rubro-negras.

"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: Heaps binários

Para entender a lógica das estruturas de dados conhecidas como heaps, devemos primeiramente definir o que são árvores binárias cheias e árvores binárias completas.

Def: Uma árvore binária é cheia se todos os nós intermediários (que não são folhas) possuem dois filhos.

Def: Uma árvore binária é perfeita se ela é cheia e todos os nós folhas estão no mesmo nível.

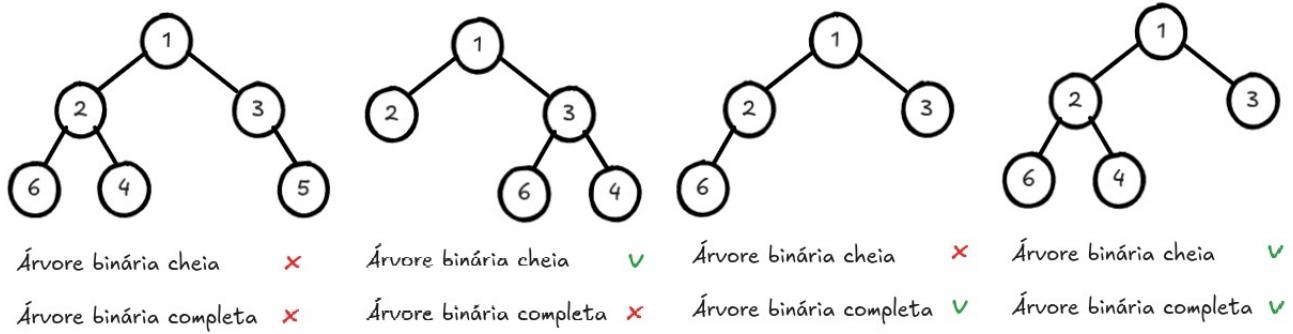
A ideia por trás de um heap é que podemos manipular uma árvore binária completa (ABC) de n nós como se fosse uma array de n elementos. Isso significa que não é necessário utilizar referências ou ponteiros para alocação dinâmica, uma vez que dado um nó da ABC, podemos facilmente encontrar seus 2 filhos e seu pai com uma simples operação matemática.

Mas afinal o que é uma árvore binária completa? Em uma árvore binária completa todos os níveis são completamente preenchidos, exceto possivelmente o último, que deve ser preenchido a partir da esquerda.

Uma árvore binária completa difere de uma árvore binária cheia em dois aspectos:

1. Todo nó folha sem irmão é filho a esquerda.
2. A última folha pode não ter um irmão direito.

Veremos adiante que 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. A seguir são apresentados exemplos de árvores binárias cheias e completas.

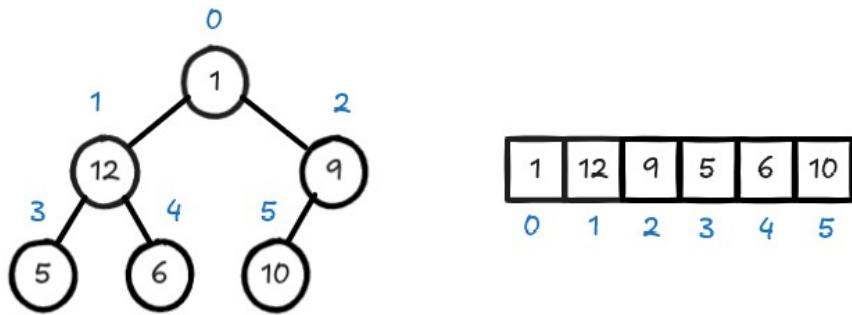


Note que para uma árvore binária ser cheia, não é necessário que todas as folhas estejam no mesmo nível, bastando que todo nó interno tenha exatamente dois filhos.

Em uma árvore binária completa de n nós, se $T[i]$ denota o i-ésimo elemento (nó), então:

- a) Filho a esquerda: $T[2i+1]$, se $2i+1 \leq n$
- b) Filho a direita: $T[2i+2]$, se $2i+2 \leq n$
- c) Pai: $T[\lfloor (i-1)/2 \rfloor]$ (onde $\lfloor . \rfloor$ denota a função piso)

Um exemplo ilustrativo pode ser encontrado na figura a seguir.

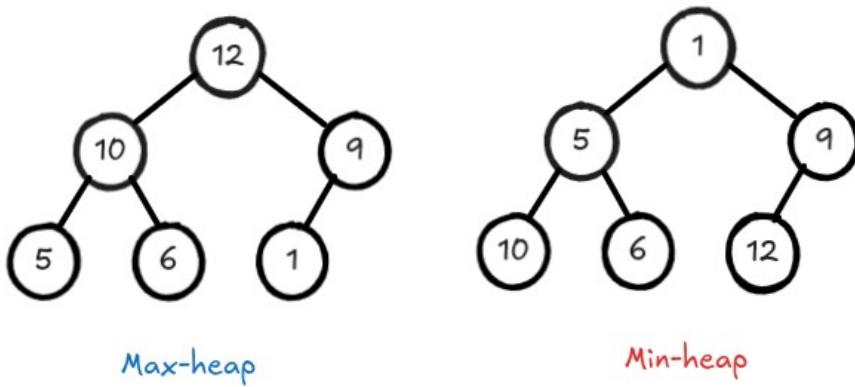


A estrutura de dados Heap

Um Heap é uma estrutura de dados especial baseada em árvores binárias completas. Uma árvore binária completa é uma estrutura de dados do tipo Heap se:

- i. Todo nó i é menor que seus dois filhos (min-heap), ou
- ii. Todo nó i é maior que seus dois filhos (max-heap)

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



Heaps e filas de prioridades

Podemos utilizar um min-heap ou max-heap para implementar uma fila de prioridades de maneira eficiente. A ideia básica é que, após cada inserção/remoção, devemos restaurar a propriedade do min-heap ou max-heap.

Em resumo, a metodologia adotada segue a lógica a seguir:

- Após cada enqueue() (inserção no final), devemos aplicar a operação auxiliar shift_up() para trazer o elemento inserido para cima no heap.
- Após cada dequeue() (remoção da raiz), trazemos a última chave para a raiz e devemos aplicar a operação auxiliar shift_down() para descer a chave para sua posição.

Sendo assim, podemos definir um TAD Priority Queue Heap como descrito a seguir.

```

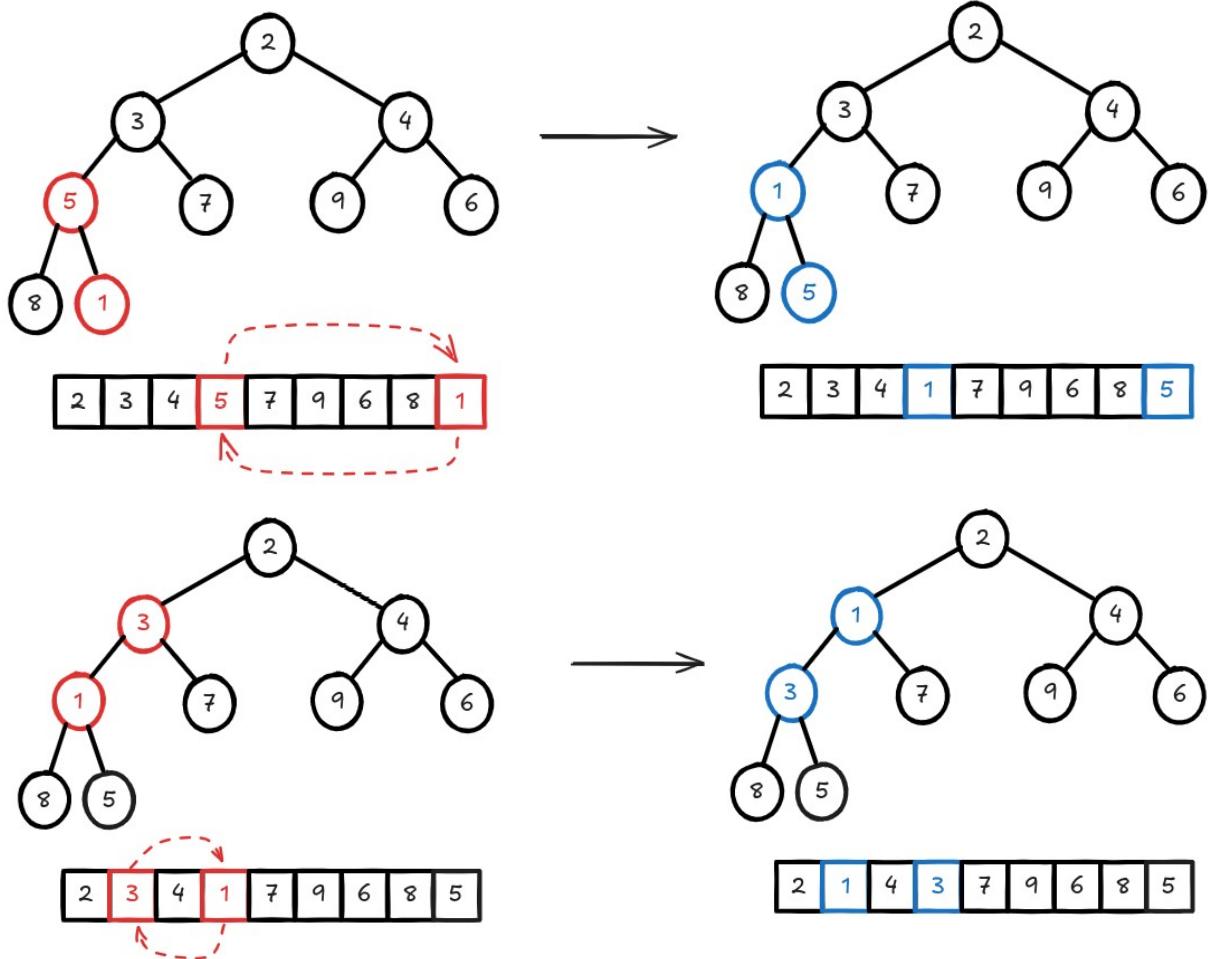
TAD Node
    int key
    int p

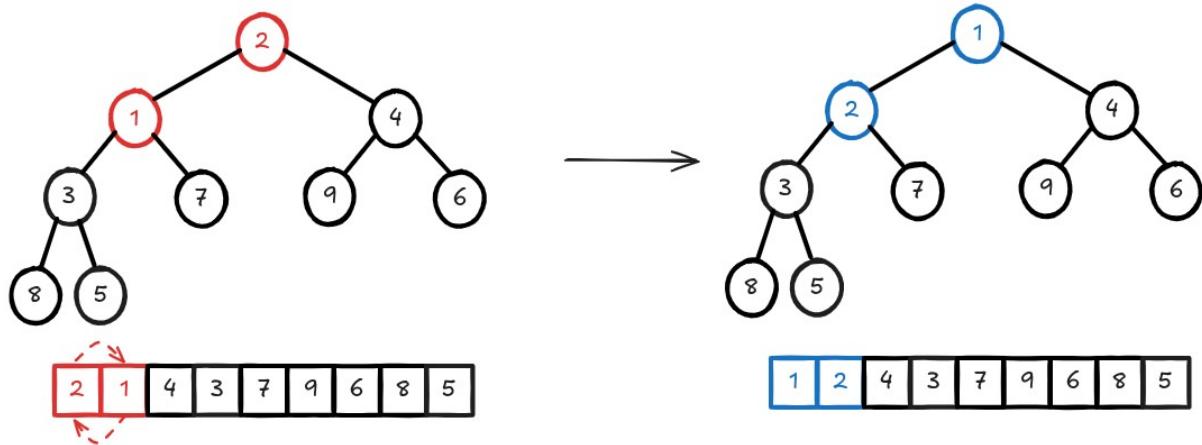
TAD PriorityQueue
    Node array[0..n-1] data
    int tail

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

# 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].p = p
        shift_up(PQ)      # sobe última chave no heap
    }
}

```

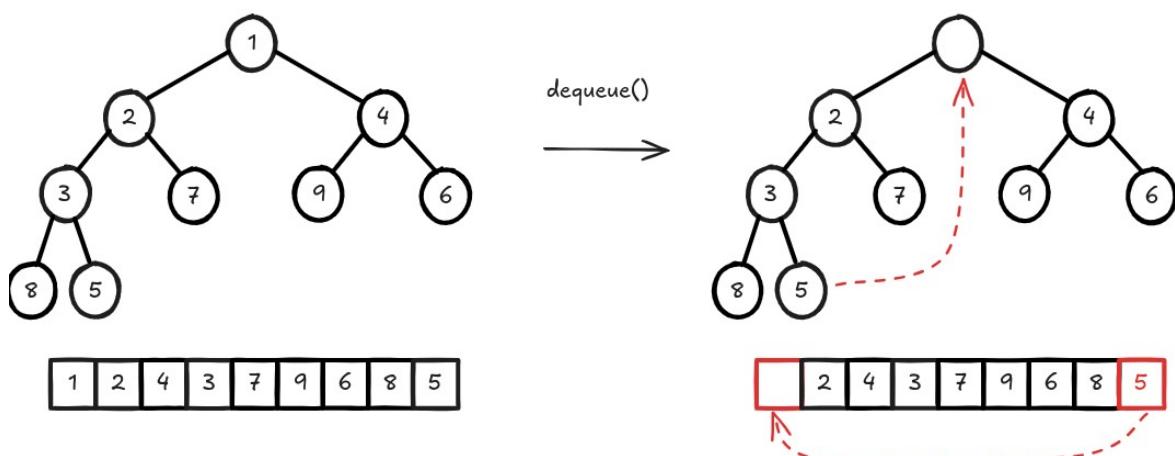


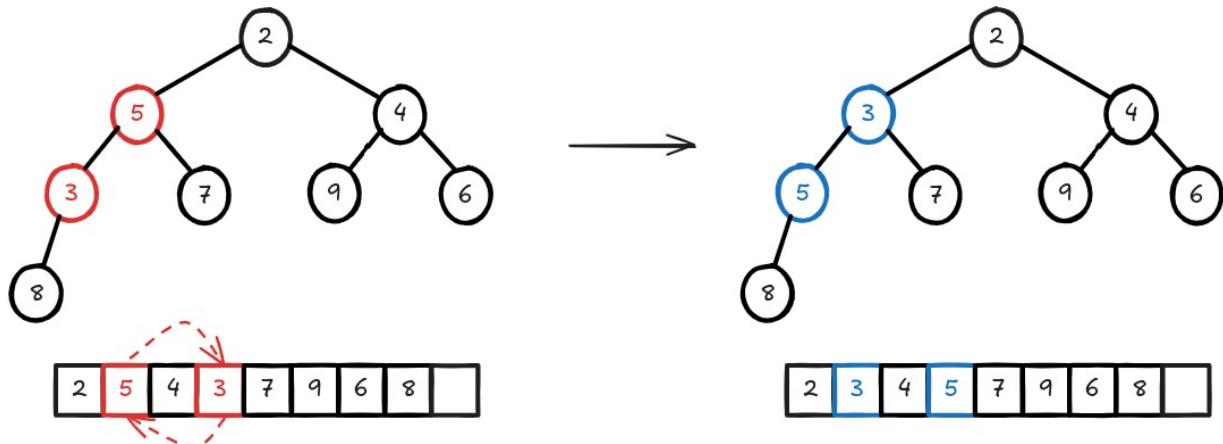
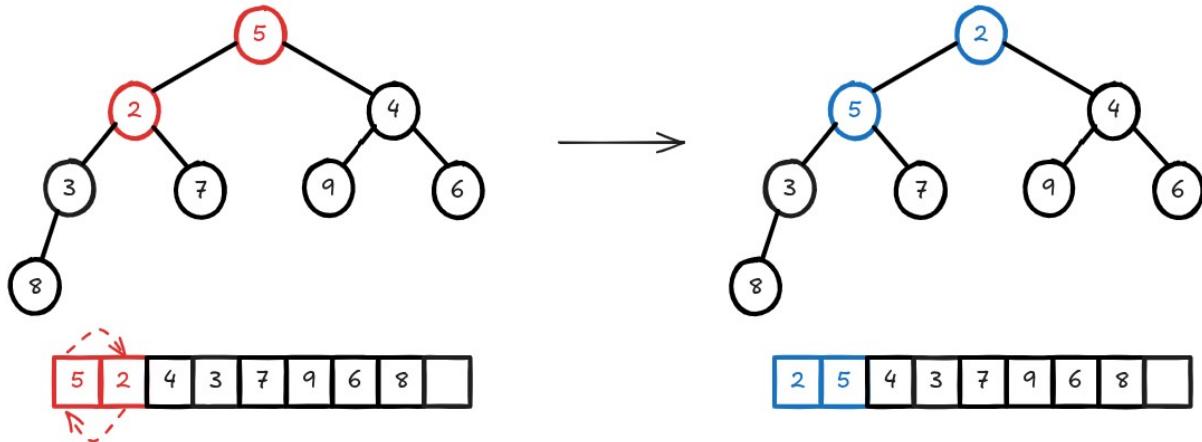


Note que após todo o processo de enqueue() e shift_up(), temos um min-heap novamente.

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

# Remove chave de menor p (menor p = maior prioridade)
dequeue(PQ) {
    if PQ.tail == -1
        error('underflow')
    else {
        key = PQ.data[0]
        swap(PQ.data[0], PQ.data[PQ.tail])      # Remove a raiz
        PQ.tail -= 1                            # Troca último com raiz
        shift_down(PQ, 0)          # desce nova raiz no heap
    }
    return key
}
```





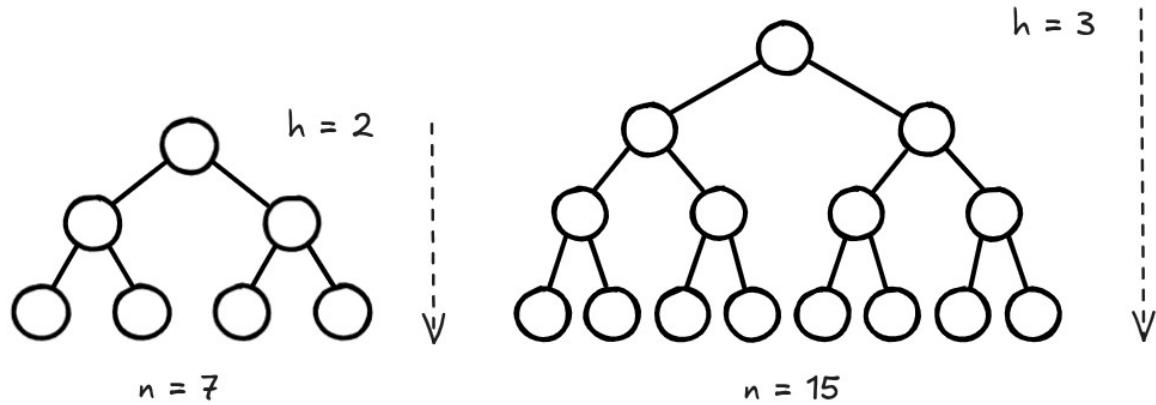
Note que após todo o processo de dequeue() e shift_down(), temos um min-heap novamente.

```
# Desce nova raiz no heap (top-down)
shift_down(PQ, i) {
    parent = i                                # i = 0 → raiz da árvore
    parent_p = PQ.data[parent].p
    left = 2*parent + 1
    right = 2*parent + 2
    smallest = parent
    if left < PQ.tail and parent_p > PQ.data[left].p
        smallest = left
    if right < PQ.tail and PQ.data[smallest].p > PQ.data[right].p
        smallest = right
    if smallest ≠ parent {
        swap(PQ.data[parent], PQ.data[smallest])
        shift_down(PQ, smallest)
    }
}
```

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

Veremos a seguir como empregar estruturas de dados do tipo heaps para projetar um algoritmo de ordenação eficiente, chamado de Heapsort.

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`.

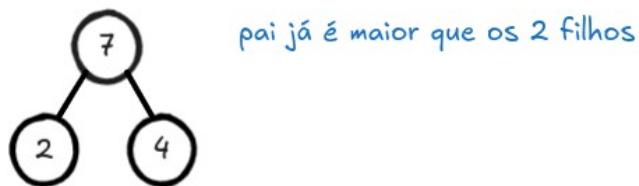
Como heapificar uma árvore?

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 a primeira vista.

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.

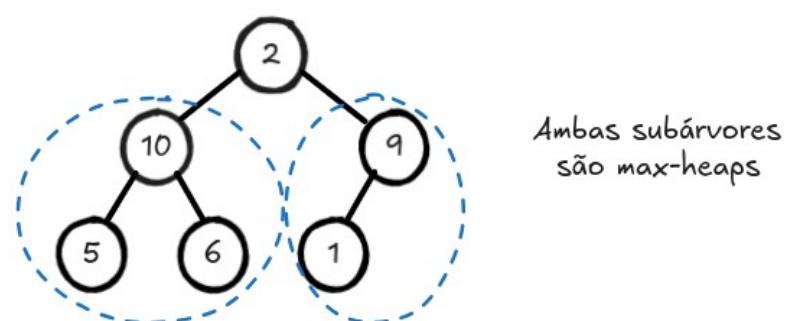
Cenário 1



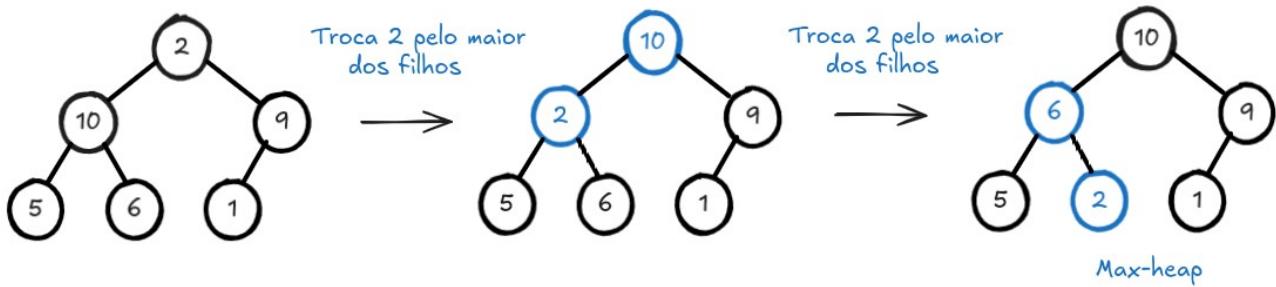
Cenário 2



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 subárvore 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.

```
# Função recursiva para heapificar um elemento da árvore
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, pois o parâmetro *i*, define qual é o elemento a ser processado. 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 subárvore sejam max-heaps.

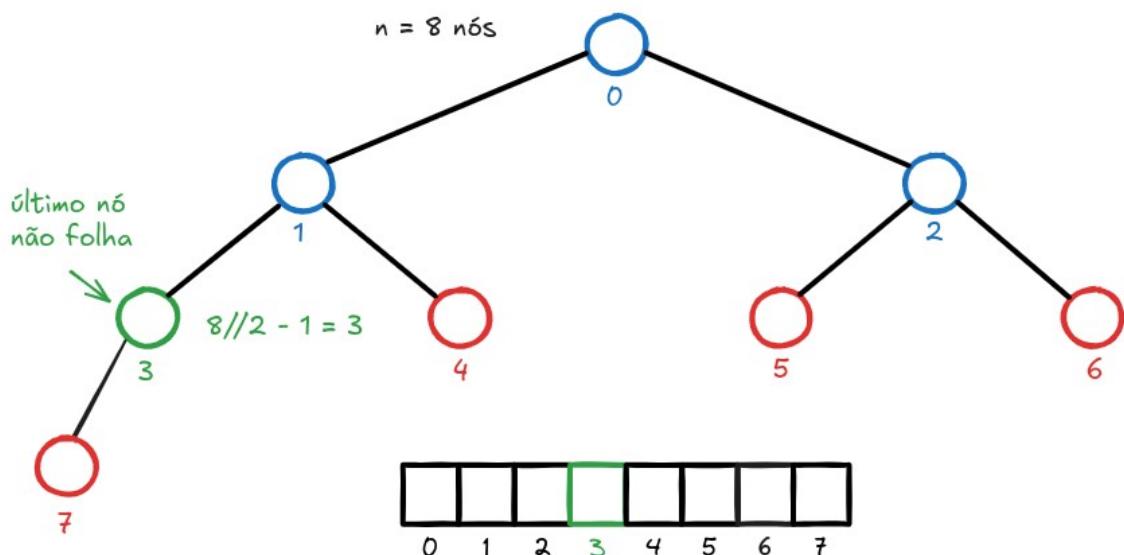
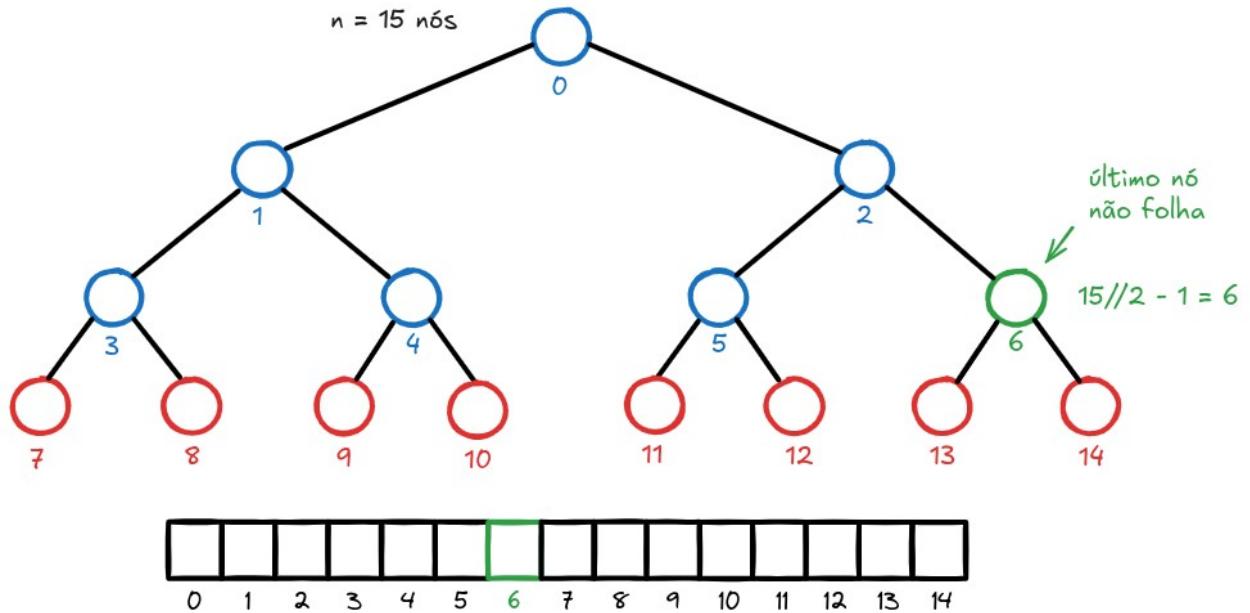
Porém, se as subárvore a esquerda ou a direita da raiz não forem max-heaps, uma única chamada da função heapify não é suficiente para criar um max-heap a partir de um array arbitrário.

Mas então como proceder para criar um max-heap a partir de um array arbitrário?

O índice do último elemento não folha em uma árvore binária completa é dado por $n//2 - 1$, pois a cada nível o número de nós em uma árvore binária dobra.

Por exemplo, em uma árvore binária completa com $n = 15$ nós, o último nó que não é folha é o nó com índice $15//2 - 1 = 7 - 1 = 6$.

Já em uma árvore binária completa de 8 nós, o índice do último nó que não é folha é $n = 8//2 - 1 = 4 - 1 = 3$. As figuras a seguir ilustram esse fato.



Dessa forma, a função para criar um max-heap a partir de um array arbitrário pode ser dada por:

```
# Constrói max heap
build_max_heap(L, n) {
    for i = n//2-1 downto 0
        heapify(L, n, i)
    return L
}
```

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

1. Inicie heapificando o array inicial para ter um max-heap. Como a árvore em questão, na forma vetorial, satisfaz a propriedade max-heap, o maior elemento está localizado na raiz (índice 0).

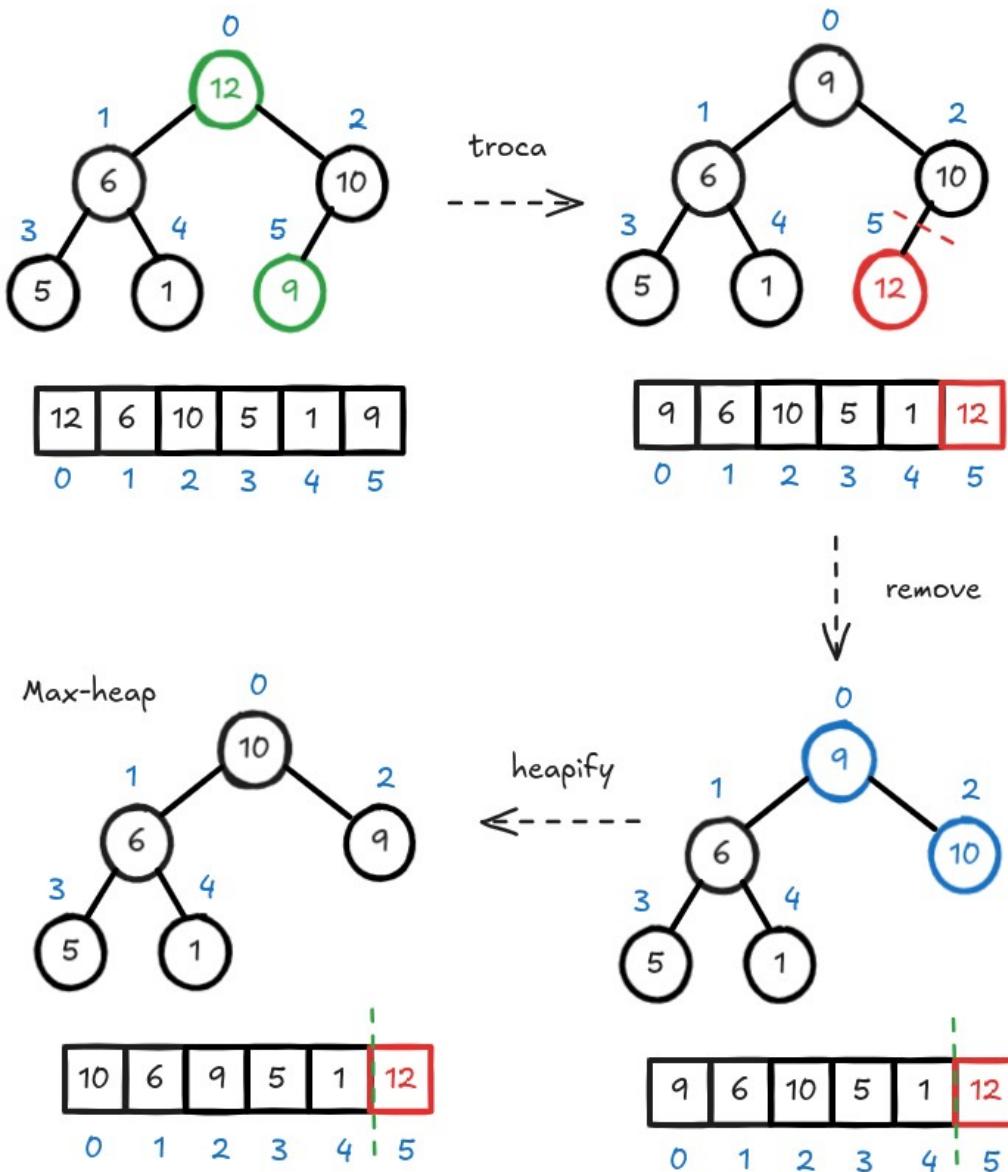
2. Swap: Coloque o elemento raiz e na última posição do vetor, trocando sua posição com o último elemento.

3. Remove: Reduza o tamanho do max-heap em uma unidade.

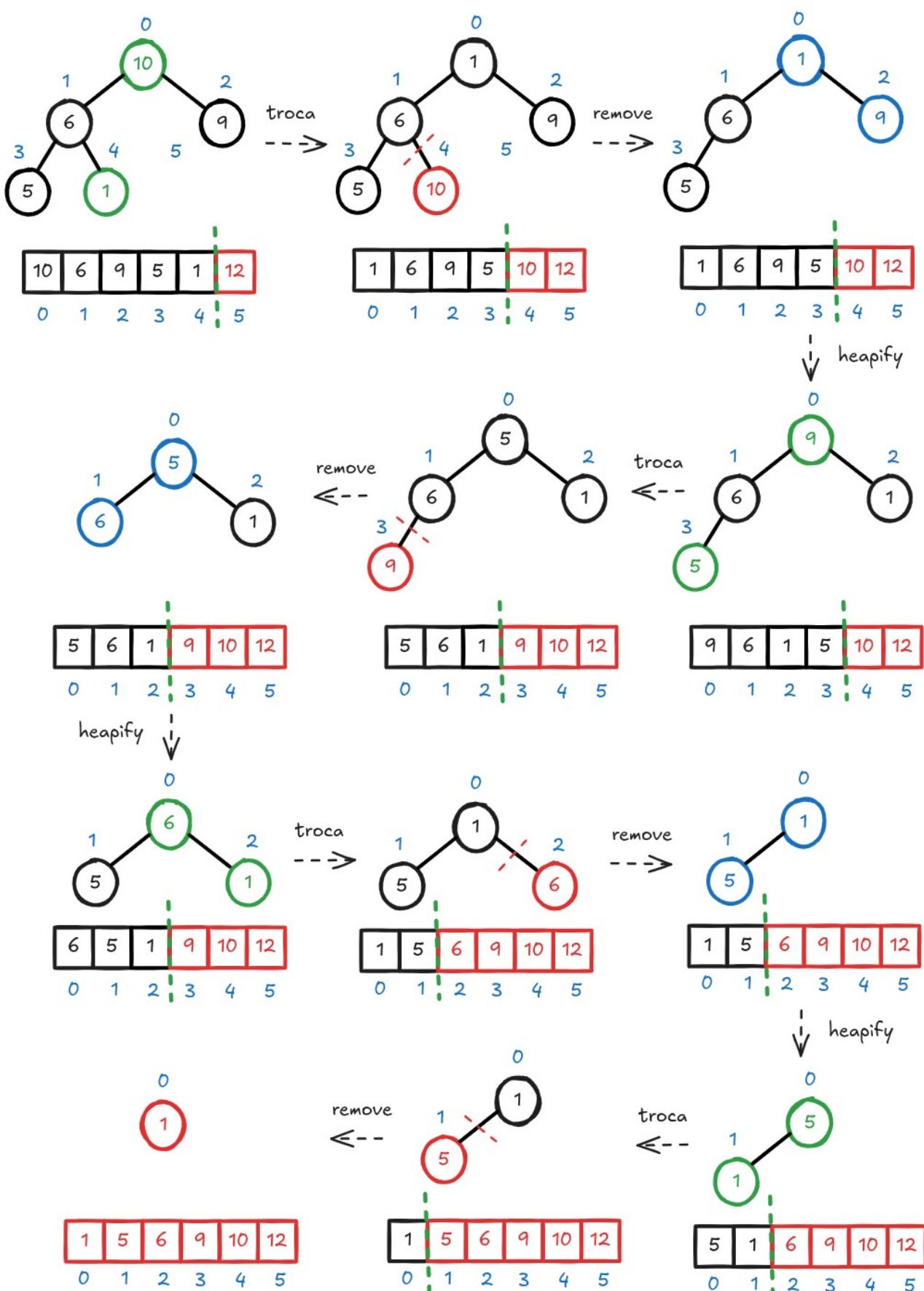
4. Heapify: Aplique a função heapify para trazer o maior elemento para a raiz da árvore.

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.



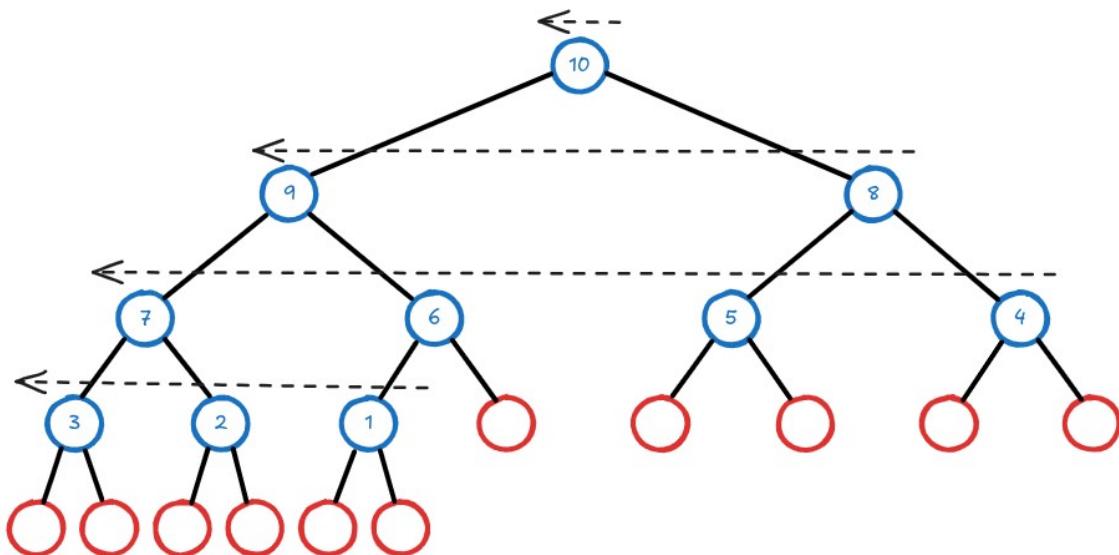
A função a seguir ilustra o pseudocódigo do algoritmo Heapsort, que faz uso das funções auxiliares descritas anteriormente.

```

heapsort(L, n) {
    # Constrói max heap
    L = build_max_heap(L, n)
    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.

"Start by doing what's necessary; then do what's possible; and suddenly you're doing the impossible."
-- Saint Francis of Assisi

Estruturas de Dados: Árvores de Busca Digitais (Tries)

Nas árvores AVL, a posição de um nó na estrutura depende essencialmente da comparação entre as chaves. Nas árvores de busca digitais (Tries), a posição de um nó depende da comparação entre os bits que compõem as chaves. Esse tipo de estrutura de dados possui vantagens e desvantagens, sendo que as principais delas são:

*Vantagens: implementação simples pois não exige operações de rotações.

* Desvantagens: desempenho depende do tamanho das chaves (quanto maior, pior).

Suponha um conjunto finito de chaves Ω dado por:

$$\Omega = \{A, B, C, D, \dots, W, X, Y, Z\}$$

Considere que as chaves são codificadas em binário de acordo com a sua posição no conjunto. Como temos 26 símbolos, são necessários 5 bits para representá-los ($2^5 = 32$).

A: 00001	B: 00010	C: 00011	D: 00100	E: 00101	F: 00110	G: 00111
H: 01000	I: 01001	J: 01010	K: 01011	L: 01100	M: 01101	N: 01110
O: 01111	P: 10000	Q: 10001	R: 10010	S: 10011	T: 10100	U: 10101
V: 10110	W: 10111	X: 11000	Y: 11001	Z: 11001		

Busca em árvores binárias digitais

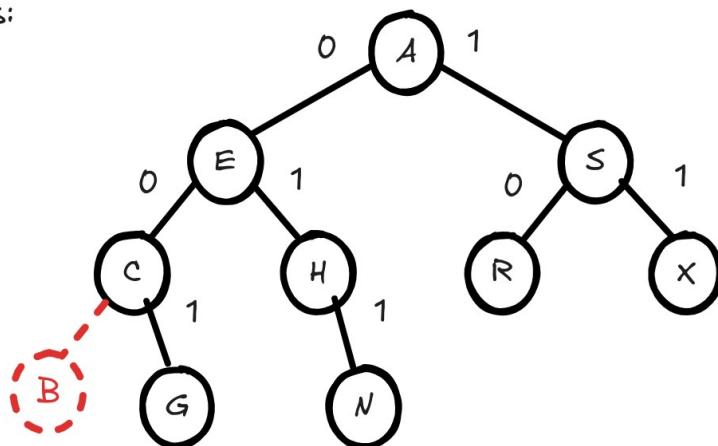
A estratégia para a busca em árvores binárias digitais consiste em comparar um bit da chave buscada com um bit do nó correto. Mas as chaves são compostas por vários bits, como saber quais deles devemos comparar? Depende da profundidade em que o nó correto se encontra! Se estamos na raiz da árvore, comparamos o primeiro bit (da esquerda para direita), se estamos no segundo nível, comparamos o segundo bit, e assim sucessivamente. Por essa razão, a profundidade da árvore está diretamente relacionada com o número de bits necessários para codificar as chaves. A figura a seguir ilustra um exemplo.

Buscar as seguintes chaves:

G: 00111 (ok)

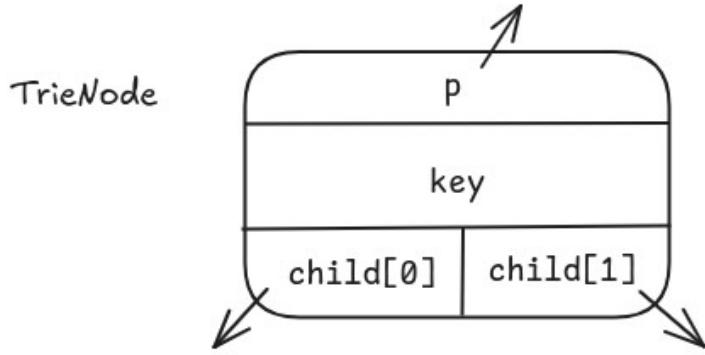
R: 10010 (ok)

B: 00010 (x)

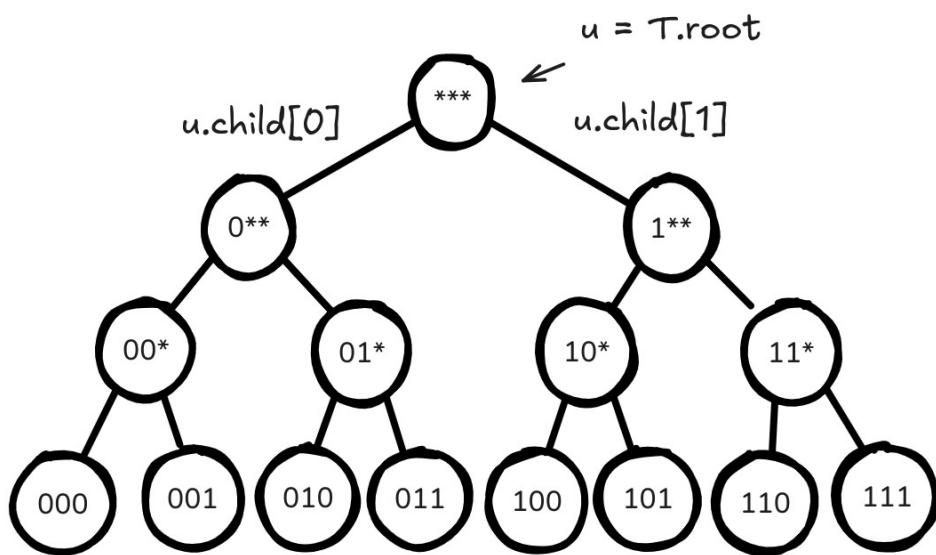


Em resumo, suponha que desejamos buscar a chave G (00111). Partindo da raiz, devemos visitar o filho a esquerda, pois o primeiro bit é zero. No segundo nível devemos novamente visitar o filho a esquerda, pois o segundo bit também é zero. No terceiro, visitamos o filho a direita pois o bit é 1.

Podemos definir o nó de uma árvore de busca digital como:



Para projetarmos um algoritmo de busca em árvores binárias digitais, considere um cenário em que as chaves possuem 3 bits. Note que nesse caso não há como ter caminhos com tamanho maior que 3, o que implica que qualquer chave deverá estar a no máximo uma distância 3 da raiz.



Com base nessa ideia, o algoritmo de busca em uma árvore digital de w bits é dado a seguir.

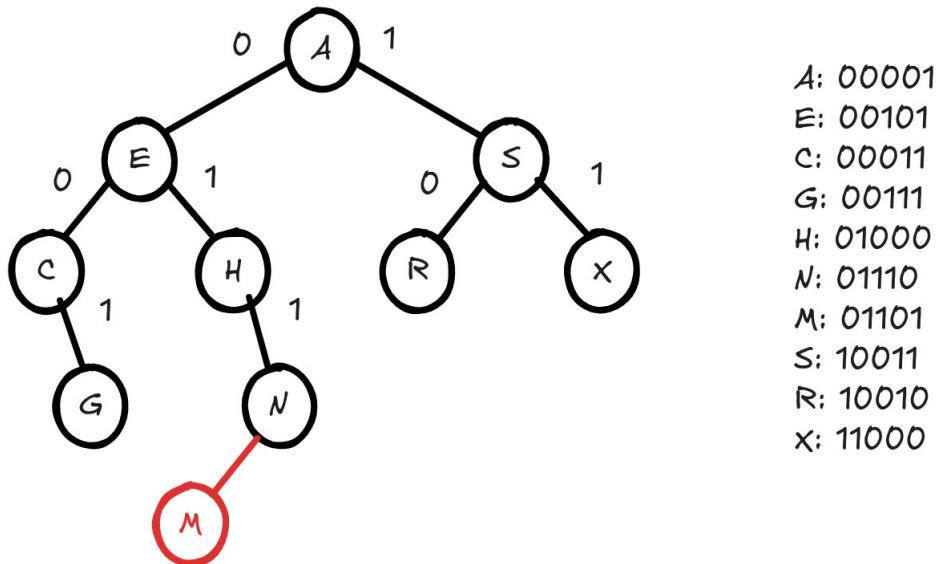
```
Trie_Search(T, x) {
    u = T.root
    for i = 0 to w-1 {
        c = (x.key >> (w-1)-i) & 1 # extrai bit a ser comparado
        if u.key ≠ x.key
            u = u.child[c]           # desce um nível
        else
            return True             # encontrou
        if u == NIL
            return False            # não encontrou
    }
}
```

O comando para extrair o bit a ser analisado usa operadores binários. Suponha $w = 3$ bits e que a chave a ser buscada é 100. A ideia é que se estamos na raiz, o bit a ser analisado é o 1. Na raiz, o valor de $i = 0$ e assim, devemos deslocar os bits de $x.key$ duas vezes para direita gerando 001 e fazer utilizar o operador lógico $\&$ (and) com o valor 1 para obter um bit 1. Antes de apresentarmos a inserção em árvores binárias digitais, veremos uma propriedade fundamental.

Propriedade chave: Toda chave de uma árvore de busca digital está em algum ponto do caminho definido pelos seus bits.

Inserção em árvores binárias digitais

Seja M a chave do nó a ser inserido, M = 01101. Logo, essa chave deverá ser inserida na primeira posição livre nesse caminho. A figura a seguir ilustra uma sequência de inserções em uma árvore de busca digital.



Note, porém, que a árvore não mantém as chaves em ordem: E > A e está na subárvore a esquerda de A. Entretanto, as chaves da subárvore a esquerda são todas menores que as chaves da subárvore a direita! Outro detalhe é que todas as chaves em uma subárvore no nível K possuem todos os K-1 bits iniciais idênticos.

A ideia é que nas árvores de busca digitais não é preciso realizar balanceamento, pois após a inserção de todas as chaves a árvore se manterá aproximadamente balanceada por conta da propriedade chave.. Dessa forma, eliminamos a necessidade de operações de rotação.

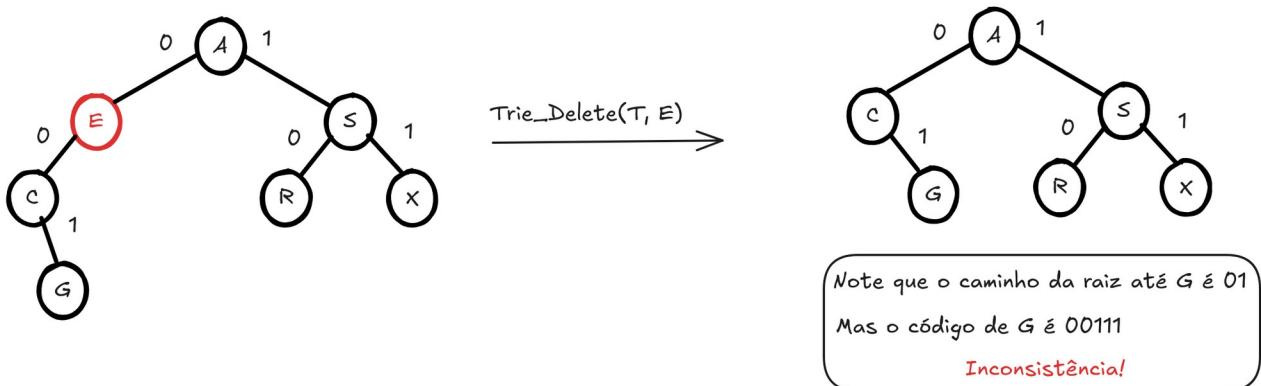
Sob hipótese de que todas as chaves são distintas e possuem w bits, o número total de elementos que podem ser armazenados é $n = 2^w$. Isso porque a altura máxima da árvore é justamente w. Se as chaves são uniformes, a altura da árvore é $h = \log_2 n = w$.

O algoritmo de inserção em uma árvore digital de w bits é dado a seguir.

```
Trie_Insert(T, x) {
    u = T.root
    for i = 0 to w-1 {
        c = (x.key >> (w-1)-i) & 1 # extrai bit a ser comparado
        if u.key == x.key
            return False                # chave já pertence a T
        if u.child[c] == NIL
            break                      # achou posição da chave
        u = u.child[c]
    }
    u.child[c] = x
    x.p = u
}
```

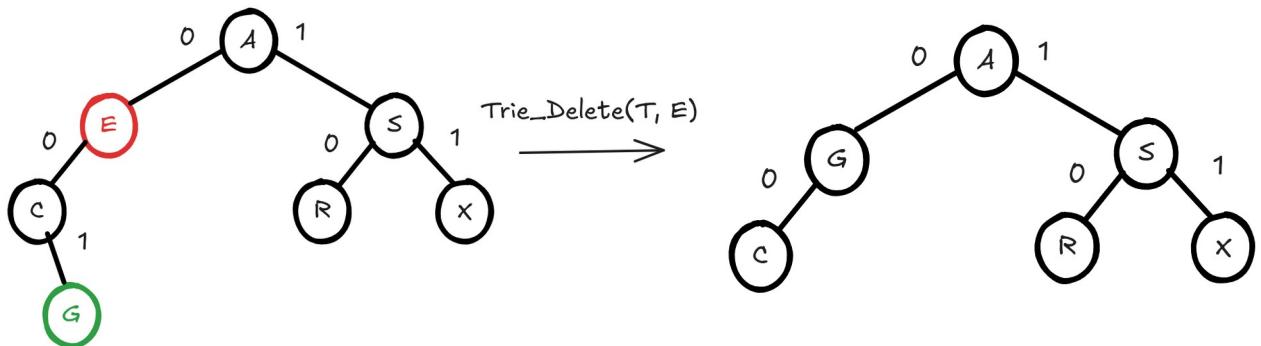
Remoção em árvores binárias digitais

A remoção de nós em árvores binárias digitais pode ser problemática, uma vez que em certos casos, a propriedade chave pode ser desfeita. O caso mais simples ocorre quando o nó a ser removido é uma folha, ou seja, não possui filhos. Neste caso, basta remover o nó desligando as referências com o nó pai. Porém, quando o nó a ser removido possui um ou dois filhos, podem surgir problemas. Vejamos um exemplo ilustrativo. Considere a árvore binária digital a seguir.



Suponha que deseja-se remover a chave E. Após a simples remoção, a chave C passa a ser filho a esquerda de A. Porém, note que de acordo com a codificação G: 00111, mas note que o caminho da raiz até G é 01. Inconsistência! O problema aqui é que 01 não é um prefixo de G, de modo que temos uma violação da propriedade chave. O que fazer?

Devemos substituir o nó removido (E) por algum descendente que seja uma folha! Depois, basta remover a folha. Isso resolve o problema, pois um nó folha pode ser colocado em qualquer posição do caminho da raiz até ele (propriedade chave). Note que G: 00111 tem o mesmo prefixo que E:00101. A árvore a seguir mostra o resultado final.



O algoritmo de remoção em uma árvore digital de w bits é dado a seguir.

```
Trie_Delete(T, x) {
    u = T.root
    for i = 0 to w-1 {
        c = (x.key >> (w-1)-i) & 1      # extrai bit a ser comparado
        if u.key == x.key
            break                         # encontrou chave a ser removida
        if u.child[c] == NIL
            return False                   # chave não pertence a T
        u = u.child[c]
    }
}
```

```

if i == w-1                                # se nó é folha
    u.p.child[c] = NIL                      # desaloca o nó
else {
    z = u
    while z.child[0] != NIL or z.child[1] != NIL {
        if z.child[0] != NIL {
            z = z.child[0]
            c = 0                         # vim da esquerda
        }
        else {
            z = z.child[1]
            c = 1                         # vim da direita
        }
    }
    u.key = z.key                          # atualiza chave
    z.p.child[c] = NIL                      # desaloca o nó
}

```

"One day you'll give someone advice you once needed and realize that it no longer applies to you. That is when you know you've grown."
-- Author Unknown

Estruturas de Dados: Árvores Rubro-Negras

Árvores rubro-negras são árvores binárias de busca平衡adas que, embora utilizem operações de rotação, seus nós não possuem fator de balanço. Mas como é possível criar uma árvore balanceada sem fator de balanço?

Veremos que, pela definição de uma árvore rubro-negra, o balanceamento é uma consequência lógica.

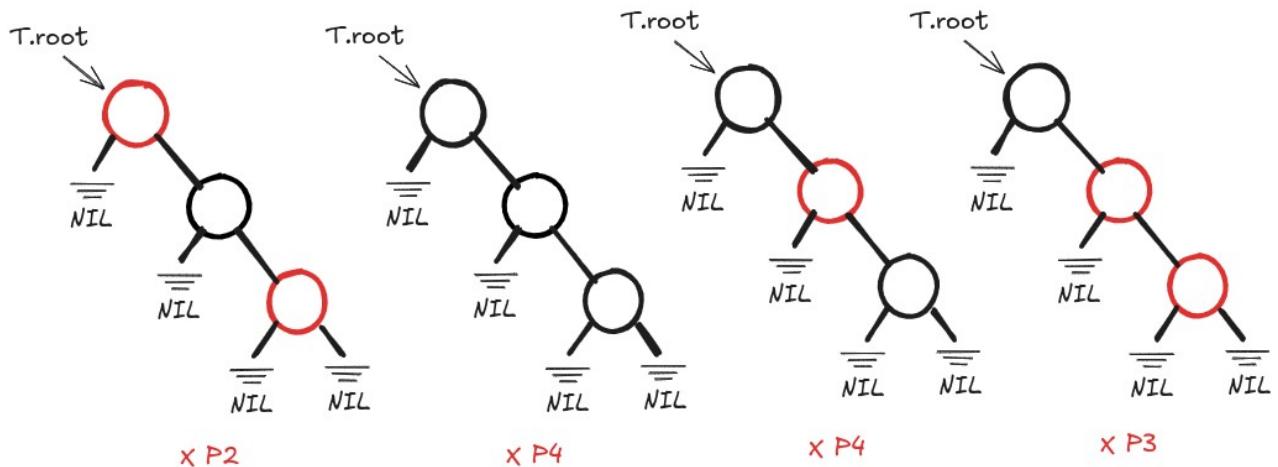
Definição: Uma árvore T é rubro-negra se:

1. Cada nó é **vermelho** ou **preto**.
2. A raiz sempre é **preta**.
3. Dois nós **vermelhos** não podem ser adjacentes (um nó **vermelho** só pode ter filhos **pretos**).
4. Todo caminho da raiz até um apontador NIL (caminho raiz-NIL) tem o mesmo número de nós **pretos** (pense nesses caminhos como aqueles percorridos em buscas mal sucedidas).

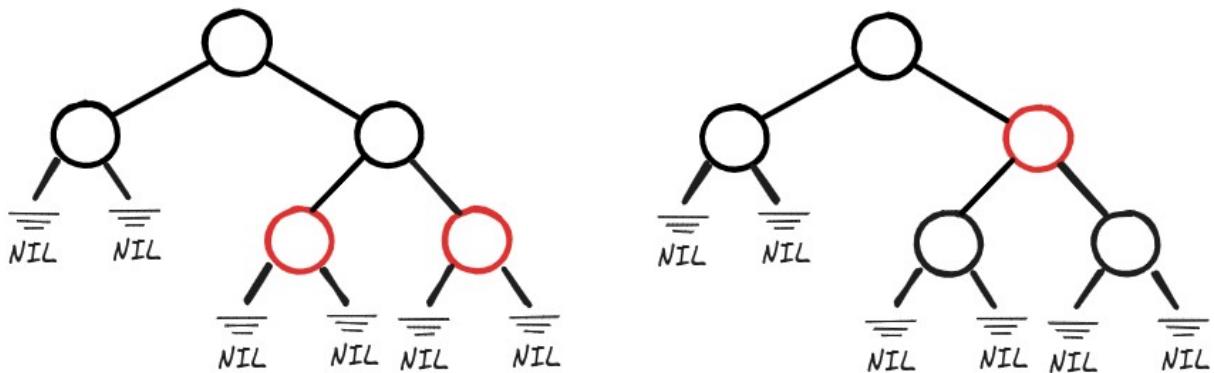
Uma conclusão lógica das premissas anteriores é: se um nó possui um único filho, ele deve ser vermelho! Caso contrário, iria ferir propriedade 4.

Iremos analisar intuitivamente porque essas propriedades nos levam a uma árvore balanceada.

Iniciamos pelas menores árvores possíveis: árvores de 3 nós.



Note que todas as árvores mostradas na figura acima ferem alguma das 4 propriedades, ou seja, nenhuma delas é uma árvore rubro-negra válida. Exemplos de árvores rubro-negras válidas são mostradas a seguir.



Note que nas duas árvores anteriores, o número de nós pretos em qualquer caminho raiz-NIL é sempre igual a 2.

Uma pergunta interessante é: qual é a altura máxima de uma árvore rubro-negra? A seguir iremos analisar como responder a essa questão.

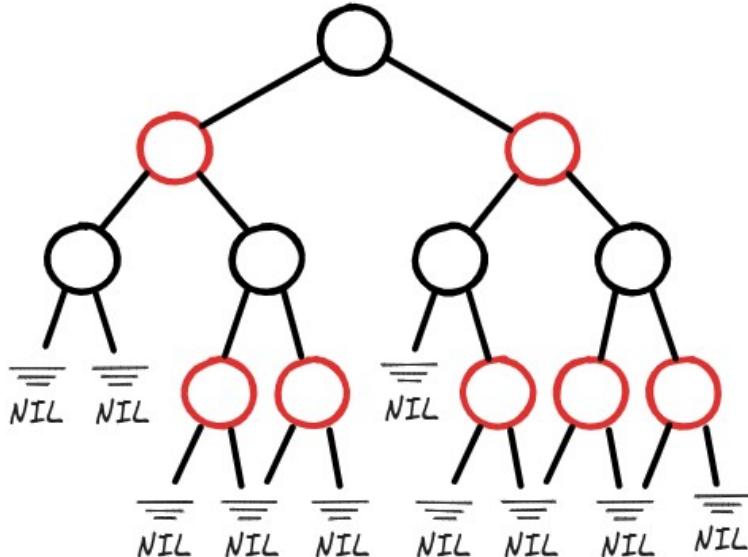
Altura máxima de árvores rubro-negras

Lembre-se que a altura de uma árvore é igual ao número de arestas do maior caminho raiz-NIL. Sendo assim, desejamos limitar superiormente o comprimento de caminhos em árvores rubro-negras.

Teorema: Se todo caminho raiz-NIL de uma árvore rubro-negra T possui pelo menos k nós, então os primeiros k níveis da árvore T devem estar completos.

É simples verificar isso pela contrapositiva: $p \rightarrow q \equiv \neg q \rightarrow \neg p$

Se os primeiros k níveis de T não estão completos, então existe um caminho raiz-NIL com menos de k nós.



Note que na árvore T acima, como o nível $k = 3$ não está completo, existem caminhos raiz-NIL de tamanho $k - 1 = 2$. Lembre-se que o comprimento de um caminho C é $|C| - 1$, onde $|C|$ denota o número de nós no caminho C.

Supondo que os k primeiros níveis da árvore estão completos, então o número de nós em T é maior ou igual o número de nós de uma árvore binária completa de altura $k - 1$, ou seja:

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

o que nos leva a

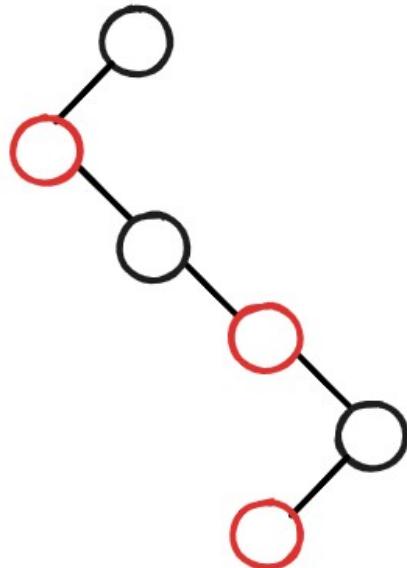
$$2^k \leq n+1$$

ou seja, $k \leq \log_2(n+1)$

o que mostra que k é $O(\log n)$, ou seja, a altura é proporcional ao log do número de nós.

Pela propriedade 4 da definição, todo caminho raiz-NIL tem o mesmo número de nós pretos. Digamos que esse número é p . Então, temos um limite superior para o número de nós pretos em qualquer caminho raiz-NIL, pois o valor de p não pode exceder a altura k , ou seja, $p \leq \log_2(n+1)$.

No entanto, desejamos encontrar um limite para o número total de nós da árvore T em qualquer caminho, já que isso limita a altura da árvore. Note que pelas propriedades 2 e 3 da definição, todo caminho tem no máximo um nó vermelho para cada nó preto, pois a raiz é preta e não são permitidos dois nós vermelhos consecutivos.



Assim, o número total de nós t em qualquer caminho raiz-NIL da árvore é no máximo duas vezes o número de nós pretos, ou seja, $t = 2p \leq 2\log_2(n+1)$.

Esse resultado é fundamental, pois nos permite caracterizar árvores rubro-negras pela sua propriedade chave:

=> **O comprimento do caminho da raiz até a folha mais distante é no máximo duas vezes o comprimento do caminho da raiz até a folha mais próxima.**

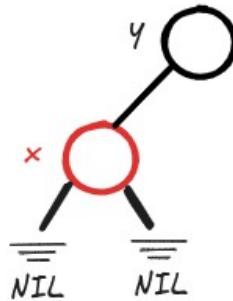
Esse resultado garante que árvores rubro-negras são balanceadas! Embora não tão balanceadas quanto as árvores AVL, mas mesmo assim garantem que as operações de busca, inserção e remoção são todas $O(\log n)$.

Inserção em árvores rubro-negras

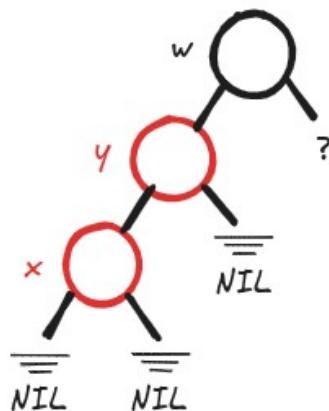
A ideia básica inserir um novo nó como uma folha **vermelha** após percorrer um caminho descendente na árvore, iniciando pela raiz. Em seguida devemos utilizar recolorações e rotações para reestabelecer as propriedades da árvore rubro-negra, percorrendo um caminho ascendente.

A seguir iremos analisar alguns casos que podem ocorrer na volta da recursão, quando percorremos o caminho ascendente, sempre considerando que o nó corrente é o nó x .

Caso 1: se y , o pai de x não for vermelho, OK. Propriedades estão mantidas!

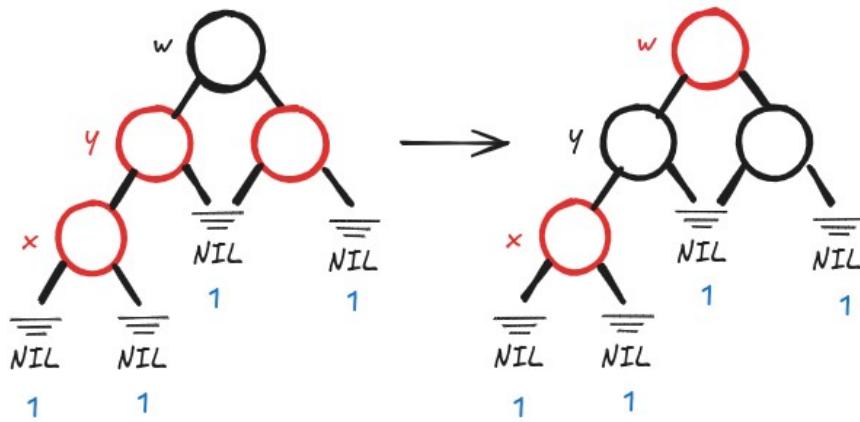


Caso 2: se y (pai de x) é **vermelho**, temos que restaurar a propriedade 3. Sabemos que, por conta da propriedade 2, y não é a raiz de T (raiz é **preta**). E que w , o pai de y , é preto, devido a propriedade 3.



Analisaremos a seguir, dois subcasos que podem ocorrer.

Caso 2.1: w tem outro filho z de cor **vermelha**



Neste caso, vamos recolorir y e z para **preto** e w para **vermelho**.

Note que a propriedade 4 está preservada, pois todos os caminhos raiz-NIL passam pelo mesmo número de vértices pretos. Porém, a propriedade 3, violada entre os nós x e y , foi restaurada.

Ao continuar a subida na árvore, será necessário analisar a situação de w (que ficou vermelho), pois essa alteração pode fazê-lo violar a propriedade 3 em relação a seu pai. Se w for a raiz da árvore,

basta mudar a cor dele para preto. Porque isso é permitido? Pois a raiz é o único nó da árvore que está presente em todos os caminhos raiz-NIL!

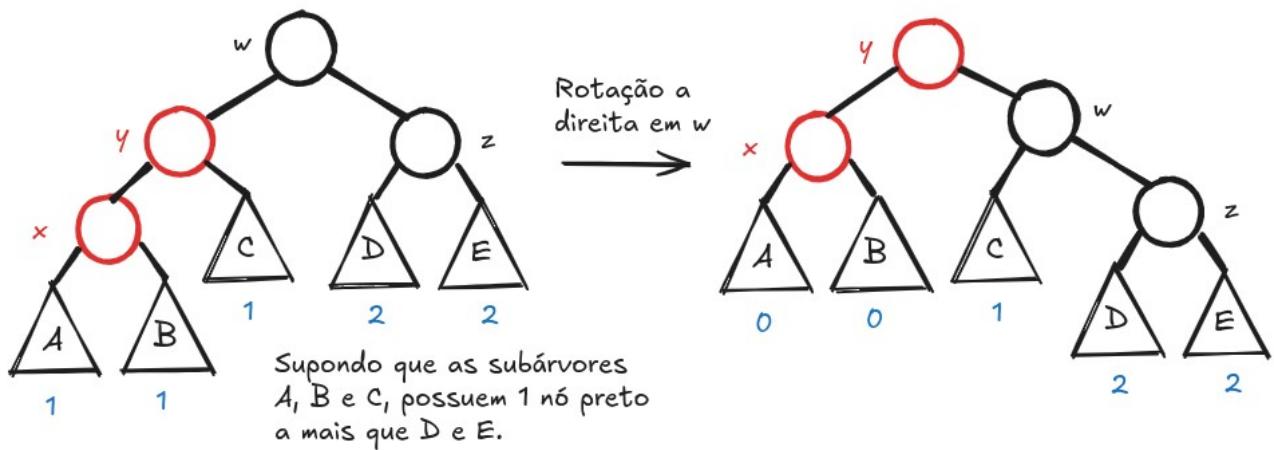
Caso 2.2: w não tem outro filho z de cor **vermelha**

Então, w pode ter outro filho **preto** z ou mesmo não ter outro filho

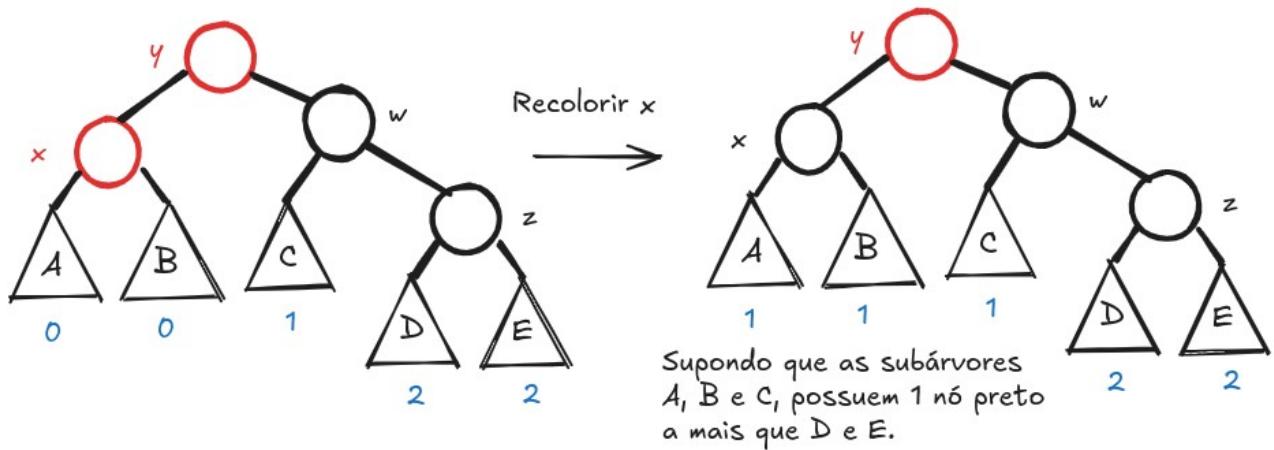
Vamos considerar a situação a) como:

a) w possui um outro filho **preto** z

Devemos realizar uma rotação a direita para inverter a relação entre w e y.



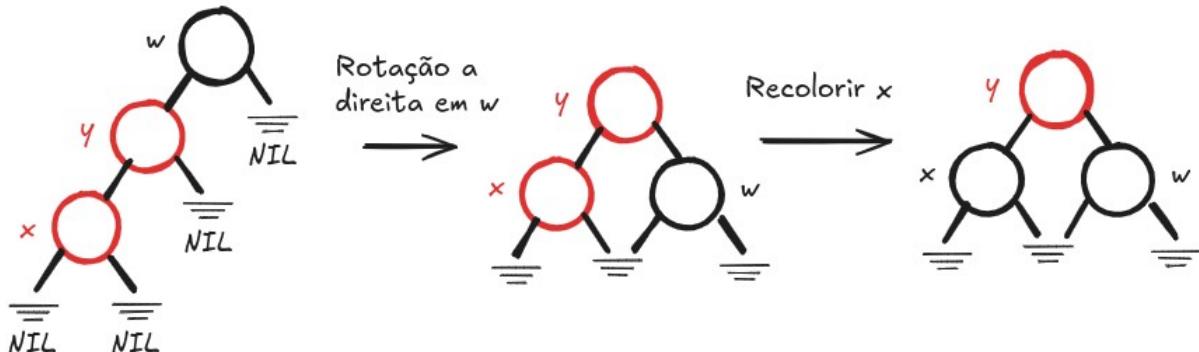
Porém, note que após a rotação, o número de nós pretos nos caminhos que vão da raiz até os filhos de x é reduzido! Para contornar esse problema, devemos mudar a cor de x para preto, o que resolve tanto o problema do número de nós pretos nos caminhos da árvore quanto os nós vermelhos adjacentes.



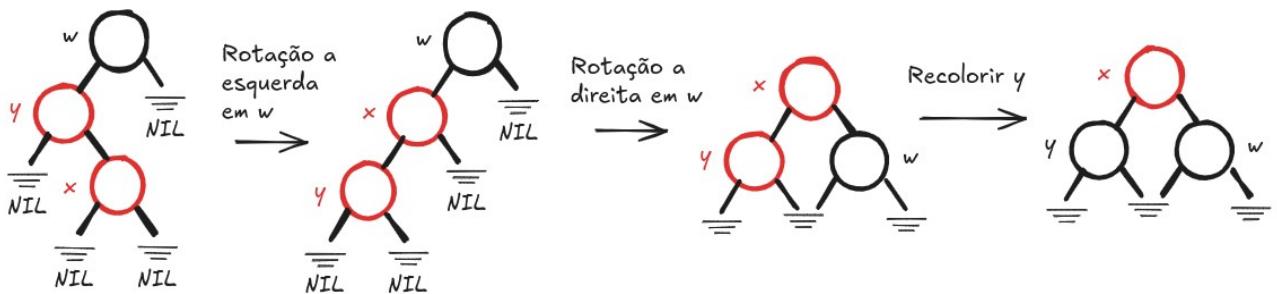
Ao continuar a subida na árvore (caminhos ascendente), será necessário analisar a situação de y, que é vermelho e se tornou a raiz da subárvore, pois ele pode violar a propriedade 3 com relação a seu pai. Se y for a nova raiz da árvore, basta recolorir y de preto (pois todo caminho raiz-NIL inclui a raiz).

b) w não possui outro filho

A mesma solução descrita anteriormente também funciona quando x é filho a esquerda de y.



Mas se x é filho a direita de y?



A seguir veremos os algoritmos necessários para a inserção em uma árvore rubro-negra. Iniciaremos relembrando a rotação a esquerda.

```
Left_Rotate(T, x) {
    y = x.right
    x.right = y.left
    if y.left ≠ NIL
        y.left.p = x
    y.p = x.p
    if x.p == NIL
        T.root = y
    elif x == x.p.left
        x.p.left = y
    else
        x.p.right = y
    y.left = x
    x.p = y
}
```

subárv esq. de y vira subárv dir. de x
se subárv esq. de y não é vazia
x torna-se o pai dessa subárvore
pai de x se torna pai de y
se x era a raiz
y se torna nova raiz
senão, se x era filho a esquerda
então y se torna filho a esquerda
senão, x era filho a direita
e y é filho a direita agora
faz x se tornar filho a esquerda de y
faz y se tornar pai de x



Com base no algoritmo acima, escreva a função para rotação a direita. A seguir apresentamos a função para inserção em uma árvore rubro-negra (percurso top-down).

```
Red_Black_Insert(T, z) {
    x = T.root           # nó que será comparado com z (buscar)
    y = NIL              # será o pai de z
    while x ≠ NIL {      # descer até a posição correta
        y = x
        if z.key < x.key
            x = x.left
        else
            x = x.right
    }
    z.p = y              # achou posição correta
    if y == NIL
        T.root = z       # ávore estava vazia
    elif z.key < y.key
        y.left = z       # z é filho a esquerda
    else
        y.right = z      # z é filho a direita
    z.left = NIL
    z.right = NIL
    z.color = RED        # todo nó inserido inicia vermelho
    Red_Black_Insert_Fixup(T, z)   # faz ajustes para preservar árvore
}
```

A função a seguir é responsável por realizar os ajustes necessários para que após a inserção de um novo nó, as propriedades da árvore rubro-negra permaneçam válidas (percurso bottom-up).

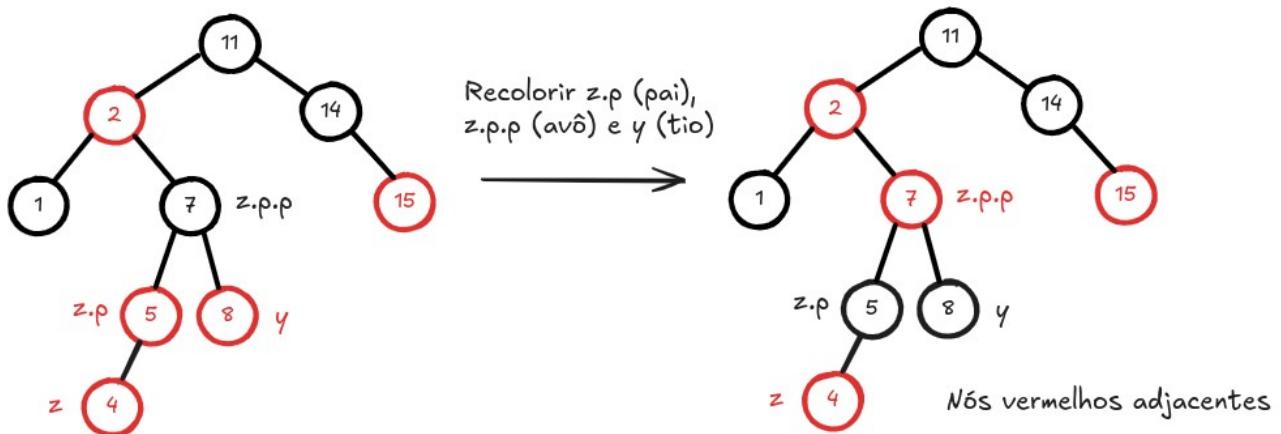
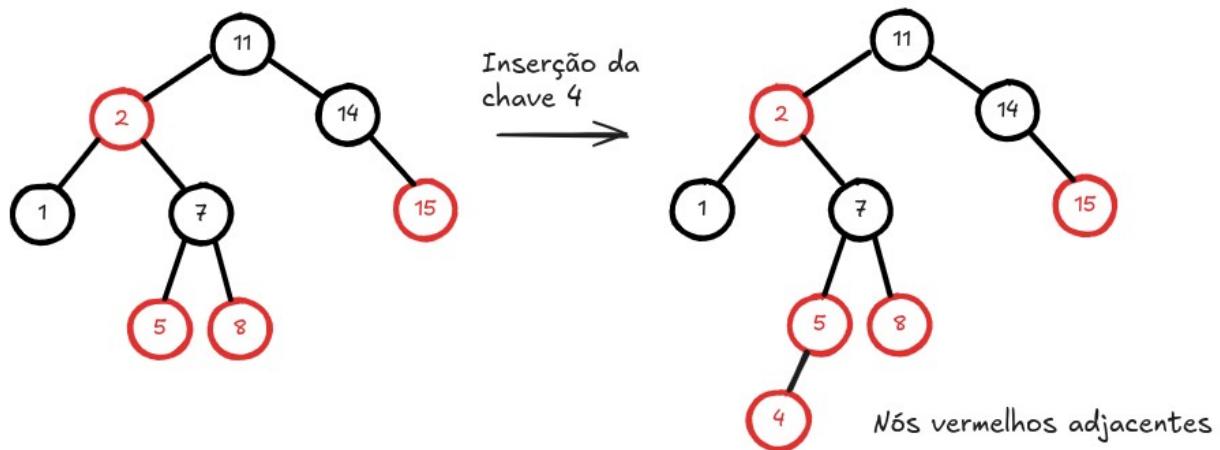
```
Red_Black_Insert_Fixup(T, z) {
    while z.p.color == RED {
        if z.p == z.p.p.left {          # o pai de z é filho a esq. ?
            y = z.p.p.right          # y é tio de z
            if y.color == RED {        # pai e tio de z são RED
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED      # raiz da subárvore fica RED
                z = z.p.p
            }
        } else {
            if z = z.p.right {         # z é filho a dir.
                z.p.color = BLACK
                z = leftRightRotate(z.p.p)
            }
            else {
                z.color = BLACK
                z = rightRotate(z.p.p)
            }
        }
    }
    else {    # o espelhamento do código anterior
        y = z.p.p.left           # y é tio de z
        if y.color == RED {        # pai e tio de z são RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED      # raiz da subárvore fica RED
        }
    }
}
```

```

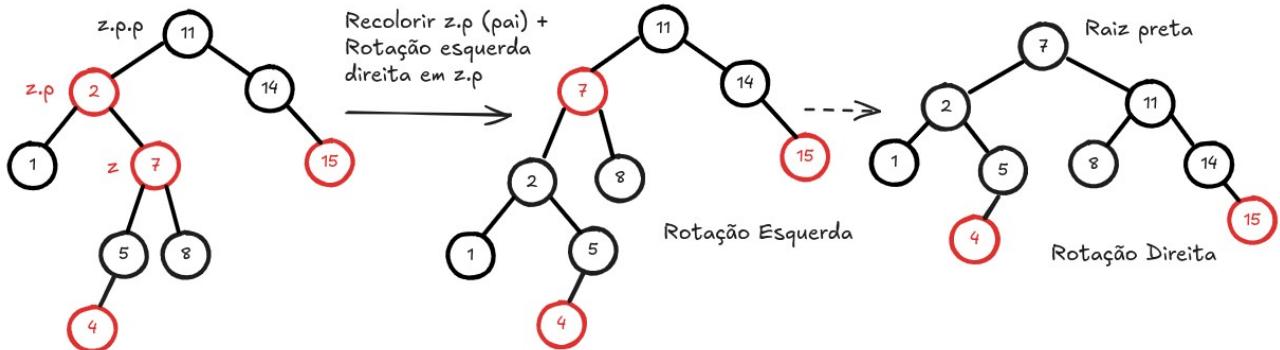
        z = z.p.p
    }
    else {
        if z = z.p.left {           # z é filho a esq.
            z.p.color = BLACK
            z = rightLeftRotate(z.p.p)
        }
        else {
            z.color = BLACK
            z = leftRotate(z.p.p)
        }
    }
}
T.root.color = BLACK
}

```

As figuras a seguir mostram um exemplo ilustrativo, em que a chave 4 é inserida em uma árvore rubro-negra. Note que após a inserção, as propriedades são violadas, mas aplicando as operações vistas anteriormente, as propriedades são restauradas.



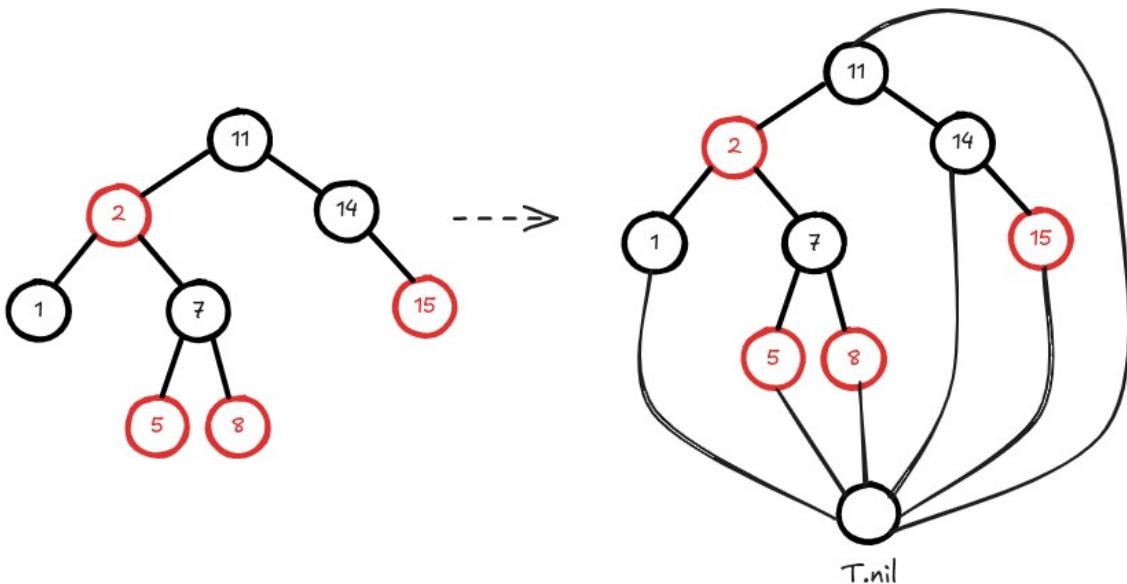
Neste ponto, devemos continuar subindo no caminho ascendente fazendo z apontar para seu avô, ou seja, fazendo $z = z.p.p$



Assim, finalmente a árvore está com todas as quatro propriedades restauradas. Note que ela encontra-se balanceada, como era de se esperar.

Remoção em árvores rubro-negras

Na remoção de nós em árvores rubro-negras, é preciso entender a organização lógica de uma árvore rubro-negra de maneira ligeiramente diferente. Ao invés de utilizarmos NIL como uma referência vazia, iremos denotar por T.nil, um nó preto na parte de baixo da árvore que representa a referência vazia. Por exemplo, a árvore a seguir ilustra esse conceito.



A remoção de um nó de uma árvore rubro-negra é mais complicado que a inserção de um novo nó. A ideia básica consiste em se inspirar na remoção de um nó em uma árvore binária de busca. Para tanto, primeiramente, iremos definir uma primitiva auxiliar RB-Transplant, que tem a mesma função do algoritmo Transplant na remoção de um nó de uma árvore binária de busca. Note que v pode ser inclusive T.nil.

```
RB-Transplant(T, u, v) {
    if u.p == T.nil
        T.root = v
    elif u == u.p.left
        u.p.left = v
    else
        u.p.right = v
    u.p = T.nil
    # nó u a ser substituído é a raiz de T
    # u é filho a esquerda de alguém
```

```

    else                      # u é filho a direita de alguém
        u.p.right = v
    v.p = u.p
}

```

A função RB-Delete é muito similar à função Tree_Delete utilizada para remover um nó de uma árvore binária de busca. A diferença principal é que após a remoção, devemos restaurar as propriedades da árvore rubro-negra, o que pode ser bastante complicado, devido aos diversos subcasos que devem ser analisados.

```

RB-Delete(T, z) {
    y = z
    y_orig_color = y.color
    if z.left == T.nil {
        x = z.right
        # substitui z pelo seu único filho a direita ou T.nil
        RB-Transplant(T, z, z.right)
    }
    elif z.right == T.nil {
        x = z.left
        # substitui z pelo seu único filho a esquerda
        RB-Transplant(T, z, z.left)
    } else {
        # se entrou aqui é porque z tem 2 filhos
        # encontra a menor chave da subárvore a direita (sucessor)
        y = Tree_Minimum(z.right) # pode ou não ser filho a dir
        y_orig_color = y.color
        x = y.right
        if y.p == z      # sucessor y é filho a direita de z
            x.p = y
        else {           # sucessor y está lá embaixo na árvore
            # Substitui y por seu filho a dir e ajusta refs
            RB-Transplant(T, y, y.right)
            y.right = z.right
            y.right.p = y
        }
        RB-Transplant(T, z, y)      # substitui z por y
        y.left = z.left            # filho a esquerda de z para y
        y.left.p = y
        y.color = z.color
    }
    if y_orig_color == BLACK   # se nó removido era preto, problema!
        RB>Delete-Fixup(T, x) # x é o nó que substitui o removido
}

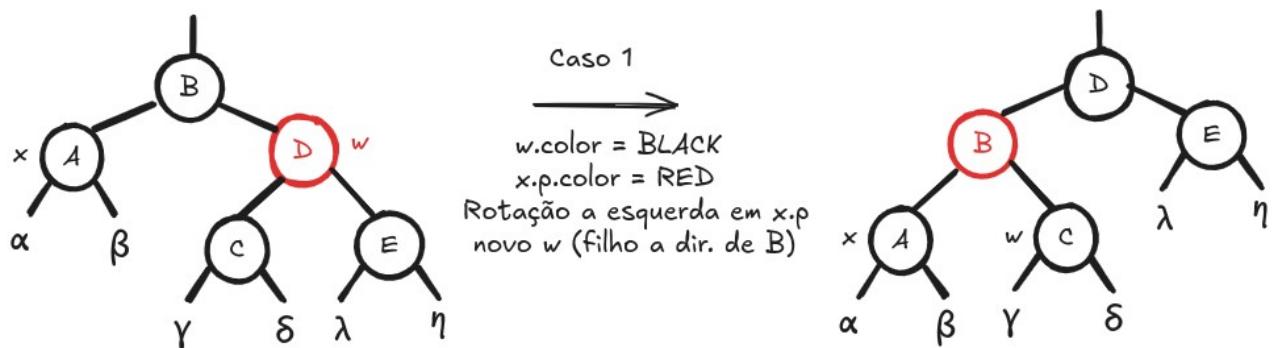
```

Note que se o nó y a ser removido é **BLACK**, uma ou mais propriedades da árvore rubro-negra podem ser violadas após sua remoção. A função RB-Delete-Fixup restaura as propriedades da árvore após a remoção. É importante salientar que se o nó a ser removido y é **RED**, as propriedades da árvore são presevadas, pois:

- i) O número de nós pretos nos caminhos raiz-NIL permanece inalterado.
- ii) Os nós vermelhos não se tornam adjacentes ao se remover um nó vermelho.

Em resumo, há 4 casos que podem ocorrer quando o nó removido y da árvore rubro-negra é substituído (por outro nó x ou por NIL). A seguir iremos ilustrar cada um deles.

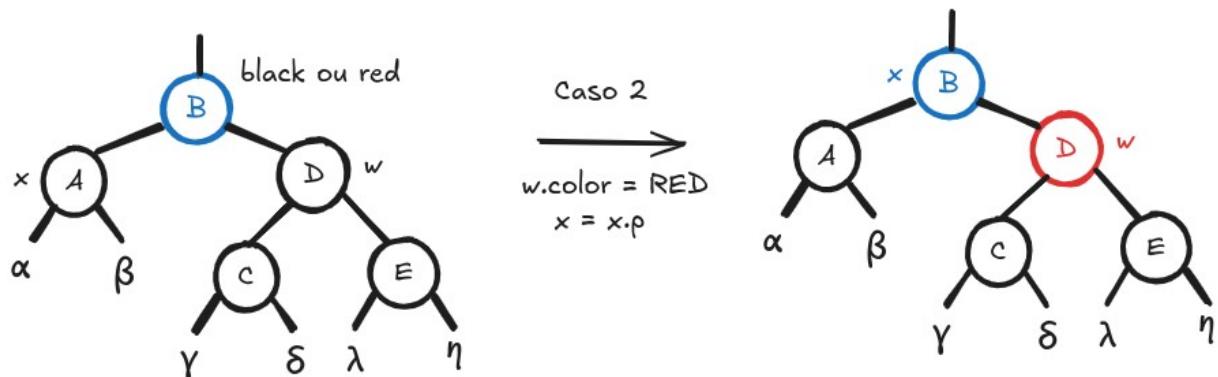
Caso 1: o nó x é filho a esquerda, é **BLACK** e possui um irmão **RED** w



- a) w fica **BLACK**
- x.p fica **RED**
- Rotação a Esquerda em x.p
novo w deve ser o filho a direita de B

Dessa forma, o caso 1 é transformado em caso 2, 3 ou 4, dependendo das cores dos vizinhos.

Caso 2: o nó x é filho a esquerda, seu irmão w é **BLACK** com os 2 filhos **BLACK**



- b) w fica **RED**
- x aponta para x.p

Caso 3: o nó x é filho a esquerda, seu irmão w é **BLACK** com o filho a direita **BLACK** e o filho a esquerda **RED**



- c) filho a esquerda de w fica **BLACK**
w fica **RED**
Rotação a Direita em w
novo w fica o filho a direita do pai de x

Caso 4: o nó x é filho a esquerda, seu irmão w é **BLACK** com o filho a direita **RED** (filho a esquerda de w pode ser qualquer cor)



- d) w fica com a cor do pai de x
o pai de x fica **BLACK**
o filho a direita de w fica **BLACK**
Rotação a Esquerda em x.p

Note que em todos os 4 casos descritos anteriormente, x é filho a esquerda de um nó arbitrário. Sendo assim, devemos tratar o caso simétrico quando x é filho a direita de alguém, o que nos leva portanto a um total de 8 casos distintos. Segue o algoritmo para a função RB-Delete-Fixup.

```
RB-Delete-Fixup(T, x) {
    while x ≠ T.root and x.color == BLACK {
        if x == x.p.left { # x é filho a esquerda
            w = x.p.right # w é irmão de x
            # Caso 1
            if w.color == RED {
                w.color = BLACK
                x.p.color = RED
                z = leftRotate(x.p)
                w = x.p.right
            }
            # Caso 2
            if w.left.color == BLACK and w.right.color == BLACK {
                w.color = RED
                x = x.p
            }
            else {
                # Caso 3
                if w.right.color == BLACK {
                    w.left.color = BLACK
                    w.color = RED
                    z = rightRotate(w)
                    w = x.p.right
                }
                # Caso 4
                w.color = x.p.color
            }
        }
    }
}
```

```

        x.p.color = BLACK
        w.right.color = BLACK
        z = leftRotate(x.p)
        x = T.root      # para sair do loop while
    }
}
else {  # x é filho a direita, igual mas troca left por right
    w = x.p.left           # w é irmão de x
    # Caso 5
    if w.color == RED {
        w.color = BLACK
        x.p.color = RED
        Right-Rotate(T, x.p)
        w = x.p.left
    }
    # Caso 6
    if w.right.color == BLACK and w.left.color == BLACK {
        w.color = RED
        x = x.p
    }
    else {
        # Caso 7
        if w.left.color == BLACK {
            w.right.color = BLACK
            w.color = RED
            Left-Rotate(T, w)
            w = x.p.left
        }
        # Caso 8
        w.color = x.p.color
        x.p.color = BLACK
        w.left.color = BLACK
        Right-Rotate(T, x.p)
        x = T.root      # para sair do loop while
    }
}
x.color = BLACK
}

```

Na teoria, as árvores AVL e rubro-negras possuem a mesma complexidade em suas operações de inserção, remoção e busca: O(log n). Em termos práticos, a árvore AVL é mais rápida na operação de busca, e mais lenta nas operações de inserção e remoção.

A principal vantagem das árvores rubro-negras é justamente nas inserções e remoções mais rápidas do que as inserções e remoções em árvores AVL. A principal desvantagem é que a busca em árvores AVL, em geral, é mais rápida do que em árvores rubro-negras, pois as árvores AVL são um pouco mais平衡adas do que as árvores rubro-negras. Portanto, se a operação prevalente for busca, árvores AVL são melhores, mas se as operações prevalentes forem inserção e remoção, árvores rubro-negras são melhores (as rotações são menos frequentes nas árvores rubro-negras).

"Once you stop learning you start dying."
Albert Einstein

Estruturas de Dados: Skip Lists

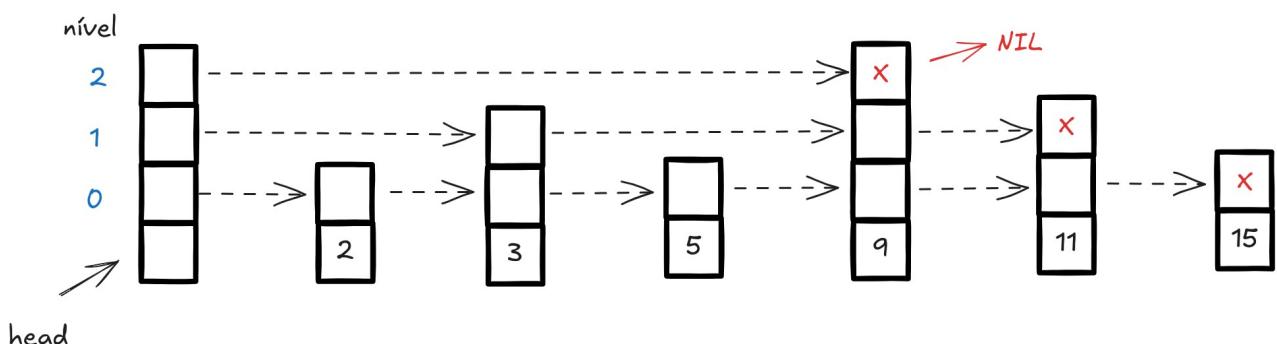
Skip Lists, ou listas com saltos, são estruturas de dados probabilísticas que permitem que as operações de busca, inserção e remoção de nós sejam realizadas com complexidade $O(\log n)$. Essas estruturas possuem as vantagens de um vetor ordenado (como na busca binária), mas com uma estrutura dinâmica similar a das listas encadeadas.

* Uma Skip List é composta por uma hierarquia de listas encadeadas de modo que quanto maior o nível, maior a esparsidade, ou seja, menos elementos aparecem.

* Uma Skip List utiliza um parâmetro conhecido como fator de dispersão ($d > 1$) para controlar a probabilidade de que um elemento do nível i seja replicado para o nível $i+1$.

Em resumo, uma Skip List é construída em camadas de modo que:

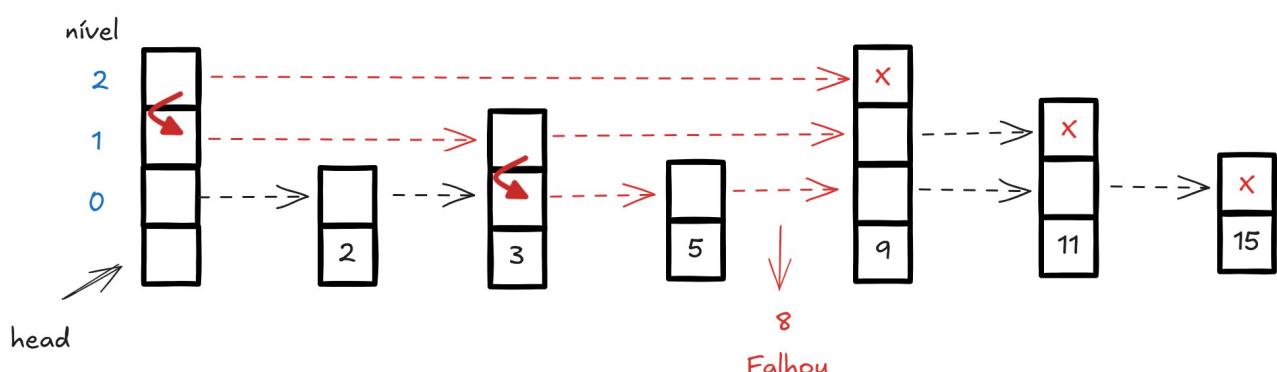
- i) a camada inferior é uma lista encadeada ordenada tradicional.
- ii) cada camada superior é uma “via expressa”, onde um elemento da camada i aparece na camada $i+1$ com probabilidade $q=1/d$. Note que se $d=2$, temos 50% de chances de aparecer (é como se jogássemos uma moeda e: se o resultado for cara o elemento da camada i será replicado na camada $i+1$, senão ele não é replicado).



Pode-se mostrar que, em média, cada elemento aparece em $d/(d-1)$ listas. O primeiro nó é um vetor de referências (ponteiros) que serve como as cabeças das diferentes listas encadeadas.

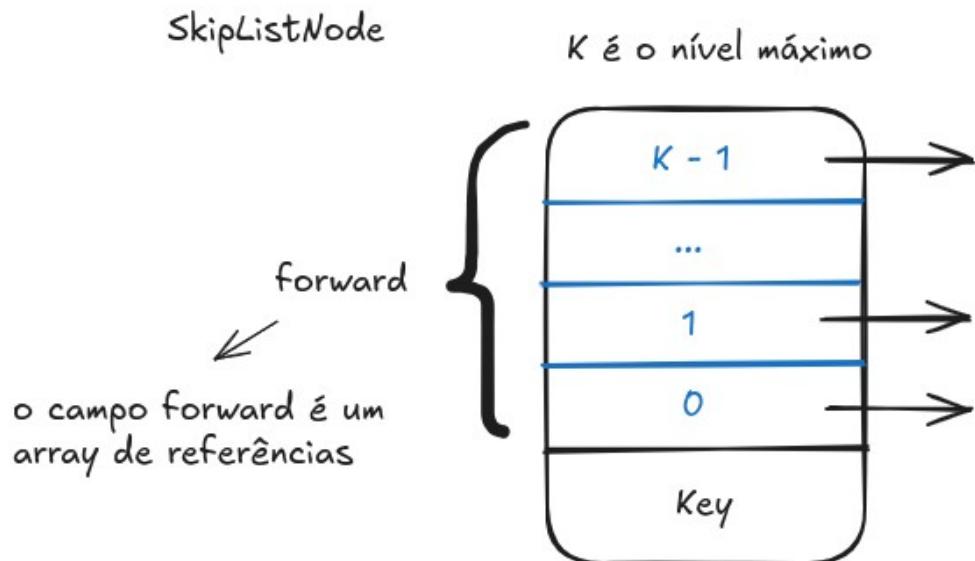
Busca em Skip Lists

Deve-se sempre iniciar a busca na lista superior (a mais esparsa) e caso não encontramos a chave, a busca deve continuar no nível abaixo. A figura a seguir ilustra a busca pela chave 8.



A lógica da busca pela chave 8 consiste em iniciar no nível 2 e percorrer a lista até que $x.key > 8$ ou chegar até o final da lista. Se não encontrou a chave, desce para o nível 1, passa pelas chaves 3 e 5 e chega na chave 9. Novamente não encontrou, então desce para o nível 0, reiniciando do último elemento visitado no nível superior (3), passa pela chave 5 e chega na chave 9. Como estamos no nível mais baixo, a busca deve retornar que a chave 8 não pertence a Skip List. A função a seguir ilustra o algoritmo de inserção em uma Skip List.

Antes de apresentar o algoritmo de busca em Skip Lists, é preciso entender como são organizadas as informações em um nó arbitrário. A figura a seguir ilustra um nó de uma Skip List. Note que além do campo `key`, temos um array de referências para outros nós, o que nos permite ligar um nó a vários outros.



A seguir apresentamos o algoritmo para buscar uma chave em uma Skip List.

```
SkipList_Search(L, key) {
    x = L.head
    for i = L.level downto 0 {
        while x.forward[i].key < key
            x = x.forward[i]
    }
    # Aqui, estamos apontando para o antecessor da chave buscada
    x = x.forward[0]
    if x.key == key
        return x.key
    else
        return False
}
```

Análise da complexidade

O fator de dispersão representa a probabilidade de um elemento do nível i ser replicado para o nível $i+1$. Em outras palavras, podemos dizer que ao passar para o próximo nível, o número de nós na lista ligada cai, em média de $q=1/d$. Note que matematicamente temos:

nível	número de nós
0	n
1	n/d
2	n/d^2
3	n/d^3
...	
k	n/d^k

Mas no último nível, temos apenas 1 elemento, ou seja, $\frac{n}{d^k} = 1$, o que implica em $n = d^k$.

Logo, $k = \log_d n$. Sendo assim, uma skip list com n itens deve ter no máximo $1 + \log_d n$ níveis.

Além disso, é esperado que para cada nó do nível $i+1$ existam d nós no nível i . Por essa razão, espera-se que na busca por um elemento, em média, sejam necessários $d/2$ saltos em um nível antes de descer para o nível seguinte.

Portanto, pelo produto entre o número esperado de níveis e o número esperado de saltos por nível, o tempo médio de busca é $\frac{d}{2}(1 + \log_d n) = O(\log_d n)$ para d constante.

Eficiência de espaço

Desejamos agora verificar, em média, quantos nós são armazenados em uma skip list. Note que no primeiro nível temos n nós, no segundo n/d , no terceiro n/d^2 , e assim sucessivamente. Portanto, a quantidade total de nós é:

$$S = n + \frac{n}{d} + \frac{n}{d^2} + \frac{n}{d^3} + \dots = n \left(1 + \frac{1}{d} + \frac{1}{d^2} + \frac{1}{d^3} + \dots \right)$$

A expressão entre parêntesis corresponde a soma de uma PG infinita com $a_0 = 1$ e razão $q = 1/d$. A soma pode ser expressa como:

$$S = n(1 + q + q^2 + q^3 + \dots)$$

Seja a soma S' dada por:

$$S' = qS = n(q + q^2 + q^3 + \dots)$$

Então, calculando a diferença $S - S'$, temos:

$$S - S' = (1 - q)S = n$$

o que nos leva a

$$S = \frac{n}{1 - q} = \frac{n}{1 - \frac{1}{d}} = \frac{n}{\frac{d-1}{d}} = \frac{nd}{d-1} = O(n)$$

Portanto, temos eficiência de espaço linear com o tamanho da entrada, o que é muito bom! Assintoticamente equivalente a uma lista encadeada tradicional.

Relação entre tempo e espaço

Vamos comparar skip lists com diferentes fatores de dispersões d.

a) Suponha que $d' = 2$

$$\text{Tempo: } \frac{2}{2} \log_2 n = \log_2 n$$

$$\text{Espaço: } \frac{2n}{(2-1)} = 2n$$

b) Suponha que $d'' = 10$

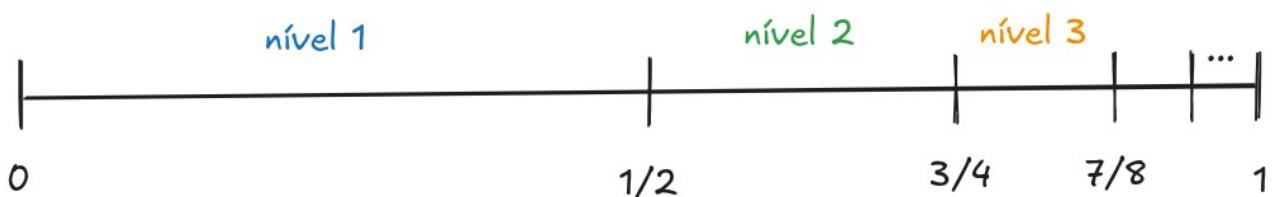
$$\text{Tempo: } \frac{10}{2} \log_{10} n = 5 \left(\frac{\log_2 n}{\log_2 10} \right) = \frac{5}{3.32} \log_2 n \approx 1.5 \log_2 n$$

$$\text{Espaço: } \frac{10n}{(10-1)} = \frac{10}{9} n \approx 1.11n$$

Portanto, podemos perceber que quanto maior o fator de dispersão d, mais lenta é a busca e menos espaço em memória ocupa a skip list.

Análise da probabilidade

A cada novo nível temos menos nós, mais especificamente $1/d$ do número de nós do nível anterior. Para isso, precisamos utilizar escolhas aleatórias, de modo que um nó pertença ao nível i com probabilidade $(1/d)^i$. Por questões didáticas, iremos considerar o caso em que $d=2$.



A ideia consiste em gerar um número uniforme entre 0 e 1 e se:

- cair na região de 0 a 1/2, vai para **nível 1**
- cair na região de 1/2 a 3/4 vai para **nível 2**
- cair na região de 3/4 a 7/8, vai para **nível 3**
- cair na região de 7/8 a 15/16, vai para **nível 4**

...

Assim, as probabilidades de um valor cair em cada subintervalo é exatamente o limite superior menos o limite inferior do intervalo.

Probabilidades

1	0.5	0.25	0.125	0.0625	...
---	-----	------	-------	--------	-----

Níveis

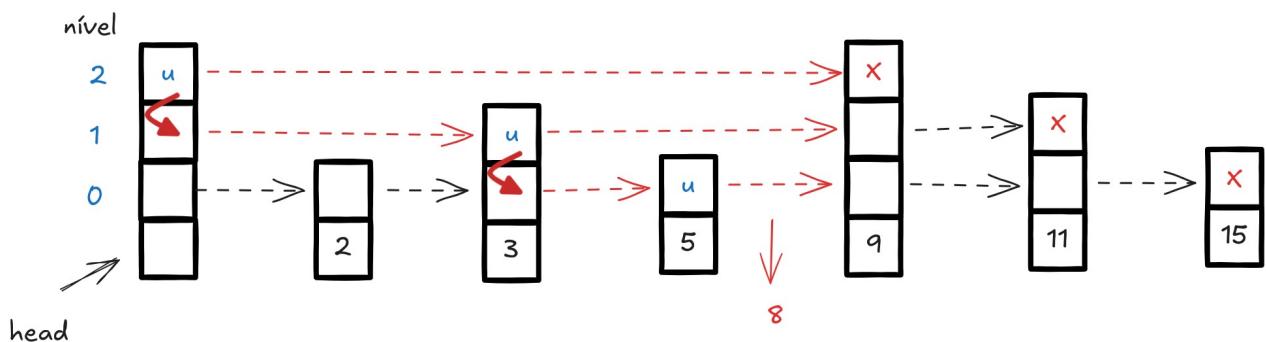
0 1 2 3 4 5 ...

Inserção em Skip Lists

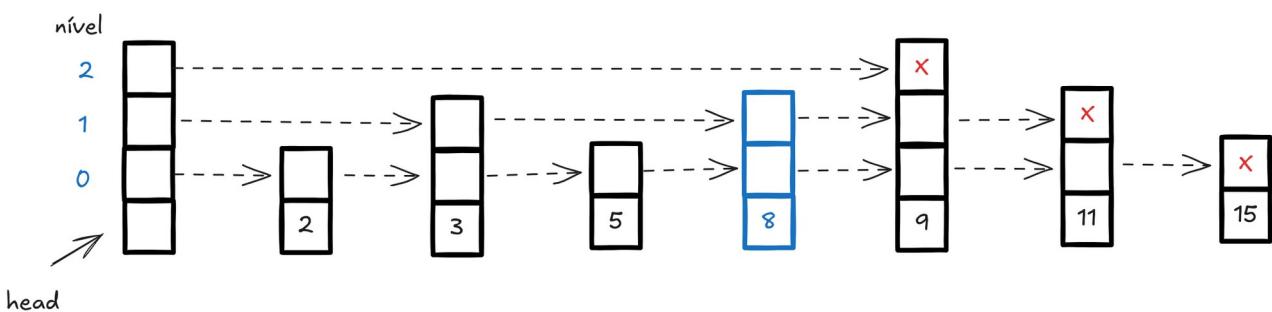
A ideia consiste em sortear um nível aleatório para o novo nó e encontrar a posição correta do nó no conjunto, o que é muito similar com o processo de busca visto anteriormente. A função a seguir ilustra o algoritmo para sortear um nível aleatório para um novo nó.

```
Random_Level(q) {      # o parâmetro q = 1/d deve ser definido previamente
    level = 1
    # É usual definir um nível máximo para evitar níveis vazios
    while random() < q and level < maxlevel
        level = level + 1
    return level
}
```

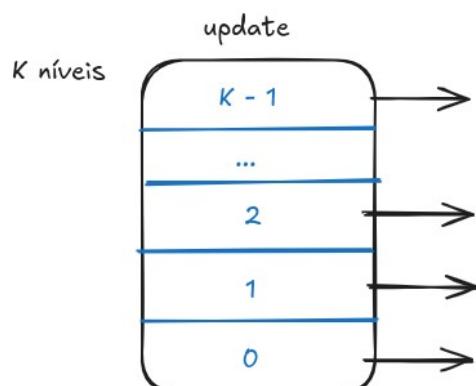
Note que se temos $q=0.5$, o processo é equivalente a: prossiga jogando uma moeda e enquanto for saindo cara, continua subindo de nível até atingir o nível máximo. A figura a seguir ilustra o processo de inserção da chave 8 em uma Skip List.



Ao encontrar a posição do novo nó, devemos ajustar as referências denotadas por u (update) para todos os níveis.



A variável `update` deve ser um array de K referências, uma para cada nível da Skip List.



A função a seguir descreve o algoritmo para a inserção de um nó em uma Skip List. O algoritmo funciona da seguinte maneira: se searchKey não existe, ela é inserida em sua posição correta. Se ela já existe, atualize-a para o valor newKey

```

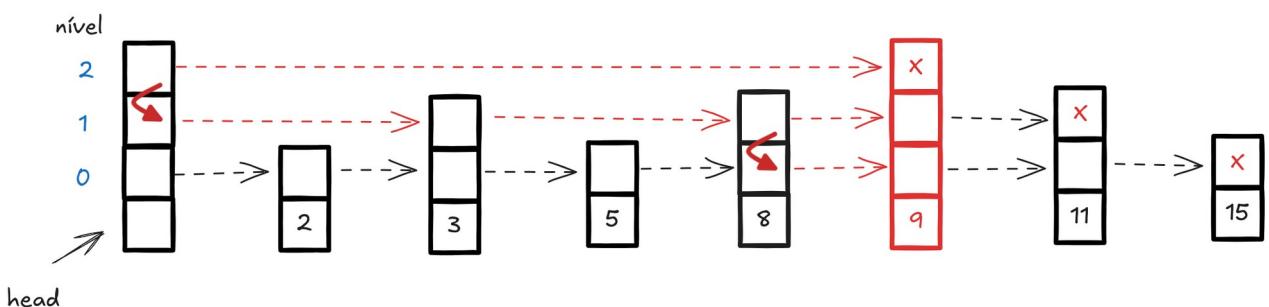
SkipList_Insert(L, searchKey, newKey) {
    Let update[0..K-1] be an array of references
    x = L.head
    # Encontra a posição correta do nó
    for i = L.level downto 0 {
        while x.forward[i].key < searchKey
            x = x.forward[i]
        update[i] = x
    }
    # Aqui, x.forward[0] aponta para o local da inserção!
    x = x.forward[0]
    # Se chave searchKey já existe, atualiza para newKey
    if x.key == searchKey
        x.key = newKey
    else {      # se chave searchKey não existe, insere
        level = Random_Level(q)      # sorteia número aleatório
        if level > L.level { # se nível sorteado é maior que atual
            for i = L.level+1 to level
                update[i] = L.head
            L.level = level
        }
        x = Make_Node(level, searchKey)
        for i = 0 to level {
            x.forward[i] = update[i].forward[i]
            update[i].forward[i] = x
        }
    }
}

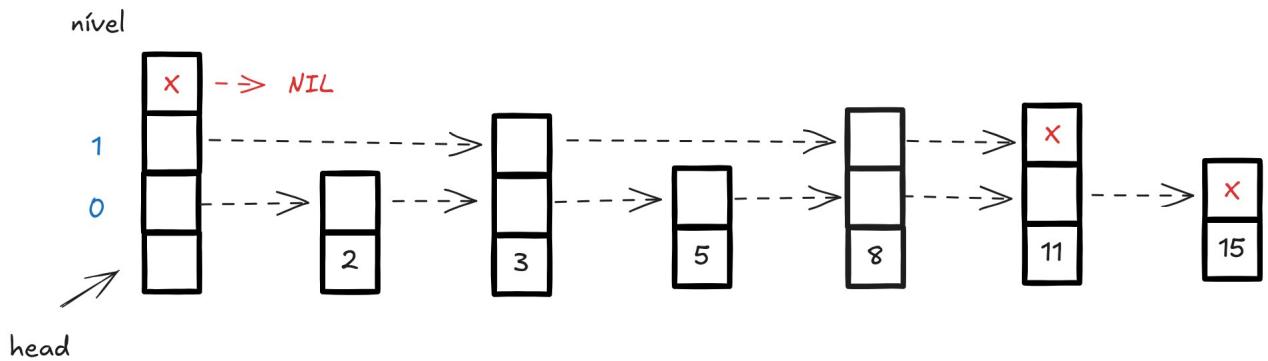
```

Como Skip Lists são estruturas de dados aleatorizadas, a inserção de um conjunto fixo de n chaves nunca será da mesma forma, ou seja, a distribuição das chaves pelos níveis muda a cada execução.

Remoção em Skip Lists

Ao remover um nó da Skip List, além de acertar as referências, devemos nos atentar se o nó removido é o último do nível k. Caso seja, devemos deletar todo o nível após sua remoção. Isso ocorre por exemplo se formos remover a chave 9 da Skip List a seguir.





Note que após a sua remoção, o último nível da Skip List é uma lista vazia. Sendo assim, ela deve ser removida, de forma que o número de níveis total é diminuído em uma unidade. A função a seguir descreve o algoritmo para a remoção de um nó em uma Skip List.

```

SkipList_Delete(L, searchKey) {
    Let update[0..K-1] be an array of references
    x = L.head
    for i = L.level downto 0 {
        while x.forward[i].key < searchKey
            x = x.forward[i]
        update[i] = x
    }
    # Aqui, x.forward[0] aponta para o nó a ser removido!
    x = x.forward[0]
    if x.key == searchKey
        for i = 0 to L.level {
            if update[i].forward[i] ≠ x      # se passa por cima do nó
                break
            update[i].forward[i] = x.forward[i]
        }
        free(x)    # desaloca o nó x
        # todo nível que estiver vazio, desconsidera
        while L.level > 0 and L.header.forward[L.level] == NIL
            L.level = L.level - 1
    }
}

```

Análise da complexidade

Tanto a inserção quanto a remoção de nós em Skip Lists realizam uma busca para encontrar a posição correta das referências a serem ajustadas, seguidas de modificações nas referências, o que leva tempo constante (pois é proporcional ao número de níveis da Skip List).

$$O(\log_d n) + O(k)$$

mas como $k \ll n$, o termo dominante é $O(\log_d n)$.

Ná prática, costuma-se observar que a inserção e a remoção em skip lists é mais rápida que a inserção e a remoção em árvores AVL, em especial a remoção, em que devemos encontrar o sucessor do nó a ser removido. Além disso, uma vantagem da skip list é armazenar as chaves ordenadas, de modo que encontrar o menor/maior elemento pode ser feito em tempo constante.

Com base em testes empíricos, observou-se que Skip List é geralmente mais rápida que árvores AVL para algumas operações, às vezes de forma significativa. Skip List é uma excelente escolha ao trabalhar com chaves já ordenadas (em ordem crescente ou decrescente). Porém, para chaves com ordem aleatória, a Skip List mostra-se muito eficiente na contagem de elementos menores que um determinado valor e na exclusão de elementos. Para chaves não ordenadas com operações de inserção e remoção, árvore AVL é preferida se a operação primária for a busca. Por outro lado, Skip List é muito mais eficiente quando há muitas exclusões e remoções na estrutura. Levando em consideração que implementar uma Skip List em uma linguagem de programação é mais fácil e rápido, pode-se dizer que Skip List supera árvore AVL em várias situações. As operações de rotação e balanceamento são bastante caras e desafiadoras de implementar em comparação com a implementação de uma Skip List, que é muito mais simples. Concluindo, implementar uma Skip List é mais simples do que uma implementar uma árvore AVL.

Aplicações

Skip Lists são muito utilizadas em sistemas distribuídos para implementar filas de prioridades concorrentes altamente escaláveis. Também são muito utilizadas em programação concorrente e paralela.

“The most frequently used implementation of a binary search tree is a red-black tree.

The concurrent problems come in when the tree is modified it often needs to rebalance.

The rebalance operation can affect large portions of the tree, which would require a mutex lock on many of the tree nodes.

Inserting a node into a skip list is far more localized, only nodes directly linked to the affected node need to be locked.”

Em sistemas operacionais, Skip Lists tem sido empregadas no kernel do Linux como alternativa a árvores AVL e rubro-negras. Skip Lists também são empregadas em motores de busca (search engines), como por exemplo o Apache Lucene (open source).

"If an egg is broken by outside force, Life ends. If broken by inside force, Life begins. Great things always begin from inside."
-- Jim Kwik

Estruturas de Dados: Tabelas de Espalhamento (Hash Tables)

Tabelas de Espalhamento, ou Hash Tables, são estruturas de dados otimizadas para operações de busca, inserção e remoção. Trata-se de uma estrutura de dados muito eficiente para a implementação de dicionários.

Pode-se mostrar que, sob um certas condições, a complexidade da busca por um elemento em uma Hash Table é $O(1)$ no caso médio.

A ideia básica de uma Hash Table generaliza a ideia de um array tradicional. Como sabemos, arrays são estruturas estáticas que permitem endereçamento direto, o que permite o acesso a qualquer elemento do conjunto com complexidade $O(1)$.

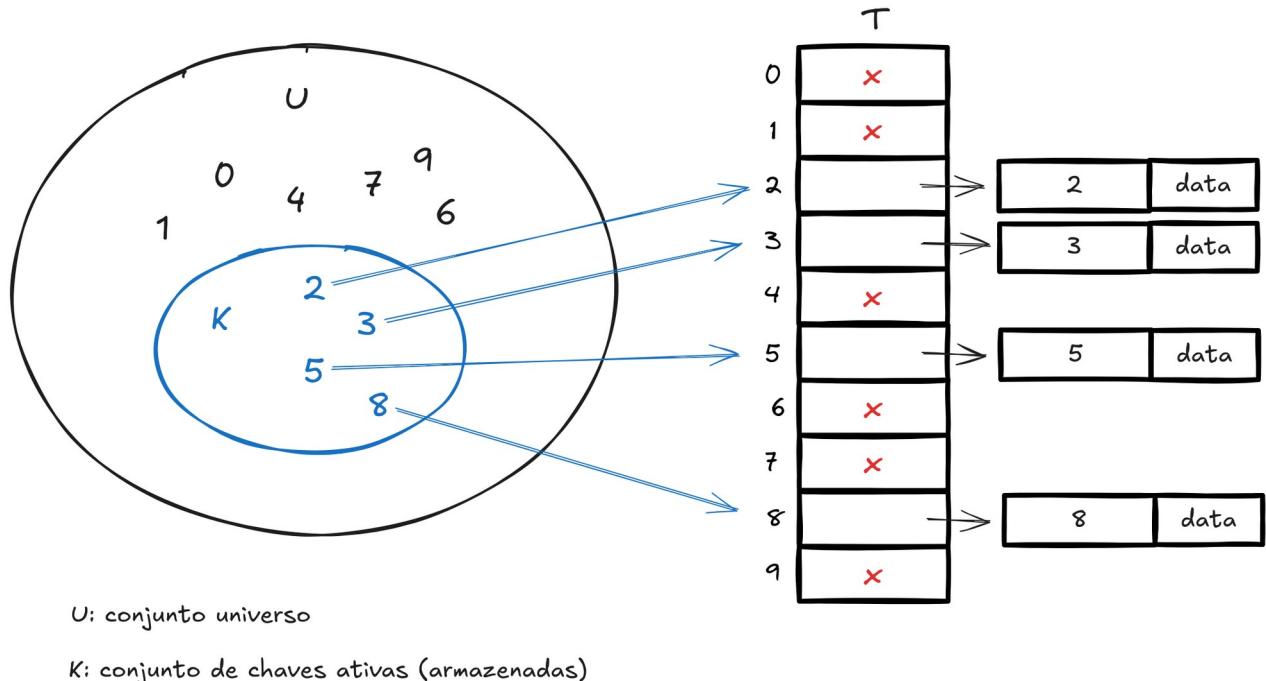
A vantagem das Hash Tables é que quando o número de chaves utilizado é relativamente pequeno em relação ao número total de possíveis chaves, essas estruturas de dados não utilizam a chave propriamente dita como índice para o acesso direto, mas sim uma função hashing.

Tabelas de acesso direto

O acesso direto é uma técnica que funciona bem quando o universo U de chaves é razoavelmente pequeno. Suponha uma aplicação que necessita de um conjunto dinâmico em que cada elemento possui uma chave retirada de um conjunto universo $U = \{0, 1, \dots, m - 1\}$, onde m não é tão grande.

Dois elementos distintos do conjunto não podem ter a mesma chave. Para representar esse conjunto dinâmico, utilizamos uma Tabela de Acesso Direto (TAD), denotada por $T[0..m-1]$, na qual cada posição ou slot, corresponde a uma chave no conjunto U .

Cada chave k é mapeada diretamente para o slot k da tabela de acesso direto. Se não há elemento com chave k , então $T[k] = \text{NIL}$.



As operações de busca, inserção e remoção são todas $O(1)$.

```
TAD-Search(T, k)
    return T[k]
```

```
TAD-Insert(T, x)
    T[x.key] = x
```

```
TAD-Delete(T, x)
    T[x.key] = NIL
```

Mas então qual é o problema?

O problema com Tabelas de Acesso Direto é simples: se o conjunto universo é muito grande, armazenar a tabela T de tamanho $|U|$ na memória do computador pode ser inviável, ou até mesmo impossível!

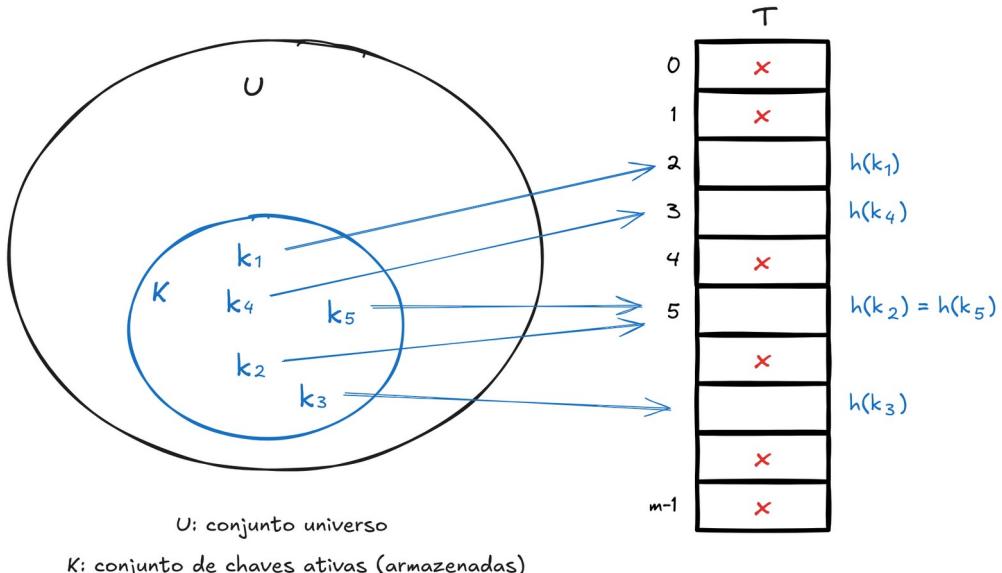
Além disso, nesse caso, o conjunto K de chaves ativas, ou seja, utilizadas de fato para o endereçamento, pode ser tão pequeno em relação ao conjunto U que a grande maioria do espaço de memória alocado para T seria um enorme desperdício.

Tabelas de Espalhamento (Hash Tables)

Quando o conjunto de chaves ativas K é muito menor que o conjunto universo de todas as possíveis chaves U, Tabelas de Espalhamento requerem muito menos espaço de armazenamento do que Tabelas de Acesso Direto, reduzindo a complexidade de espaço de $O(|U|)$ para $O(|K|)$, mantendo a complexidade da busca em $O(1)$.

Nas Tabelas de Acesso Direto o elemento com a chave k é mapeada para o slot k, enquanto que nas Tabelas de Espalhamento o elemento com a chave k é mapeada para o slot $h(k)$, onde $h(\cdot)$ é a função hash que mapeia os elementos do conjunto universo U nos possíveis slots da Tabela de Espalhamento $T[0..m-1]$, ou seja, $h: U \rightarrow \{0, 1, \dots, m-1\}$, sendo que o tamanho m da Tabela de Espalhamento é muito menor que o tamanho do conjunto universo U.

Nesse contexto, dizemos que $h(k)$ é o valor de hash da chave k. A grande vantagem é que ao contrário das Tabelas de Acesso direto, que possuem o mesmo tamanho do conjunto universo, as Tabelas de Espalhamento possuem tamanho m, o que é bem menor que $|U|$.



Problema: Podem ocorrer colisões! Duas chaves distintas mapeadas para o mesmo slot!

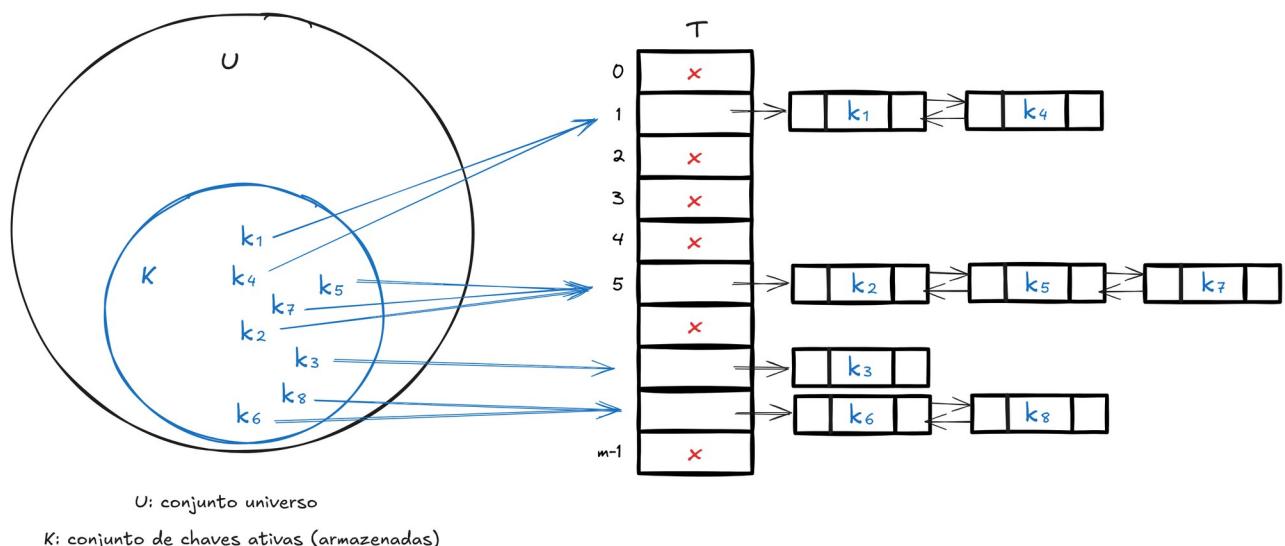
A ideia é fazer a função hash h parecer o mais “aleatória” possível, de modo que o espalhamento das chaves nos slots da tabela seja quase uniforme.

Mas como $|U| > m$, no entanto, deve haver pelo menos duas chaves que tenham o mesmo hash valor; evitar colisões por completo é, portanto, impossível. Assim, ainda precisamos de um método para resolver as colisões que ocorrem.

Tratamento de colisões por encadeamentos lógicos

A estratégia mais comum para o tratamento de colisões consiste em criar um encadeamento lógico envolvendo todos os elementos que possuam o mesmo valor hash.

Em outras palavras, devemos criar uma lista encadeada com todos os elementos cujos valores hash sejam iguais. Em resumo, o slot k contém uma referência para a cabeça de uma lista encadeada que armazena todos os elementos com valor hash $h(k)$. Se não há elemento com valor hash $h(k)$, o slot k aponta para NIL.



Assim, pode-se definir as seguintes primitivas básicas:

```
# Insira x na cabeça da lista encadeada em T[h(x.key)]
TH-Insert(T, x) {
    Insert_Head(T[h(x.key)], x)
}

# Busque por um elemento com chave k na lista encadeada T[h(k)]
TH-Search(T, k) {
    Search(T[h(k)], k)
}

# Remova x da lista encadeada T[h(x.key)]
TH-Delete(T, x) {
    Delete(T[h(x.key)], x)
}
```

Note que em termos de complexidade temos os seguintes fatos:

- A inserção de uma nova chave tem complexidade O(1) (cabeça da lista).
- A busca busca por uma chave k depende do tamanho da lista $T[h(k)]$ (analisaremos mais adiante).
- A remoção de uma chave, após ela ser localizada, também pode ser realizada em O(1), uma vez que podemos implementar listas duplamente encadeadas, o que faz com que o reajuste das referências não necessite de uma referência auxiliar (a primitiva TH-Delete assume que já temos uma referência para o nó a ser removido).

Análise da complexidade da busca

A seguir, iremos analisar a operação de busca, que é a mais complexa.

Seja T uma tabela de Espalhamento com m slots que armazena um total de n chaves.

Define-se o fator de carga de T como $\alpha = n/m$, que a média de elementos armazenados por uma lista encadeada.

As análises de complexidade serão feitas em termos de α que pode ser menor, igual ou maior que 1. O pior caso é a situação em que todos os n elementos possuem o mesmo valor hash, ou seja, são armazenados na mesma lista encadeada, o que nos leva claramente a uma complexidade O(n) para a busca.

A complexidade da busca depende de quanto bem a função hash consegue espalhar as chaves pelos m slots de T. Iremos assumir a hipótese de *hashing simples uniforme* (HSU):

$$\forall k \in U \left(\forall m \in T \left(p(k \in m) = \frac{1}{m} \right) \right)$$

ou seja, a probabilidade de uma chave cair em um slot é sempre igual a $1/m$ (mesma chance de cair em qualquer um dos m slots).

Seja n_k o comprimento da lista encadeada $T[h(k)]$, para $k = 0, 1, \dots, m-1$. Então, $n = n_0 + n_1 + \dots + n_{m-1}$. Note que o valor esperado de n_k é justamente $E[n_k] = \alpha = n/m$.

O cálculo da função hash pode ser feito em O(1), uma vez que ela requer operações matemáticos simples. Além disso, o tempo requerido para buscar um elemento com chave k depende linearmente do tamanho da lista encadeada $T[h(k)]$, ou seja, de n_k .

Devemos considerar 2 casos distintos:

- 1) quando a busca não encontra o elemento com chave k ($k \notin T$); e
- 2) a busca termina com sucesso, encontrando o elemento k em uma das listas encadeadas ($k \in T$)

Teorema: Em uma T.H. de encadeamento lógico, a busca por uma chave $k \notin T$ tem complexidade $O(1+\alpha)$ (constante), no caso médio.

Prova:

1. Sob a hipótese de hashing simples uniforme, qualquer chave k que não está armazenada na estrutura tem probabilidade igual de cair em qualquer um dos m slots.

2. O tempo esperado para a busca sem sucesso de uma chave arbitrária k é o tempo esperado de se buscar um elemento até o fim da lista $T[h(k)]$, cujo tamanho esperado é exatamente $E[n_k] = \alpha$

3. Portanto, temos uma complexidade total $O(1+\alpha)$ (pois 1 refere-se ao custo da função hash).

Note que, quanto maior o fator de carga de T , maior o custo da busca (porém, é linear no fator de carga, o que ainda é bom).

A situação para uma busca que termina com sucesso é ligeiramente diferente, uma vez que cada lista não tem a mesma probabilidade de ser buscada (o elemento pertence a uma das listas). A probabilidade de que uma lista encadeada seja pesquisada é proporcional ao número de elementos que ela contém (quanto maior, mais provável)

Teorema: Em uma T.H. de encadeamento lógico, a busca por uma chave $k \in T$ tem complexidade $O(1+\alpha)$ (constante), no caso médio.

Prova:

1. Iniciamos assumindo que o valor a ser buscado tem probabilidade igual de ser qualquer um dos n elementos armazenados na estrutura, ou seja, $1/n$.

2. O número de elementos examinados durante a busca com sucesso por uma chave k é 1 (cálculo da função hash é $O(1)$) mais o número de elementos que aparecem antes de k na sua lista encadeada $T[h(k)]$.

3. Como novos elementos são adicionados no início da lista, as chaves que vem antes de k em $T[h(k)]$, foram adicionadas depois que k foi inserida.

4. Para encontrar o número esperado de elementos examinados, nós tomamos a média sobre as n chaves na Tabela de Espalhamento, de 1 mais o número esperado de elementos adicionados à lista $T[h(k)]$ depois que k foi adicionada na lista.

5. Note que a probabilidade de uma chave ser adicionada a lista de k é $1/m$ (pela hipótese de hashing simples uniforme), o que nos leva ao seguinte número médio de operações:

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

É fácil notar que o primeiro somatório é igual a n vezes n , ou seja, n^2 . Como vimos em aulas anteriores, temos que:

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

o que nos leva a:

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

Mas como sabemos que o fator de carga é $\alpha = n/m$, temos:

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

o que é linear em α . Portanto, temos que a complexidade no caso médio da operação de busca em uma Tabela de Espalhamento é $O(1+\alpha)$.

Mas o que essa análise nos diz em termos práticos?

Se o número de chaves na Hash Table é proporcional ao número de slots, ou seja, $n=cm$, então temos que $n=O(m)$ e por consequência $\alpha=O(m)/m=O(1)$.

Portanto, no caso médio, a busca na Hash Table pode ser realizada em tempo constante!

Funções Hash

As funções hash assumem que o universo de chaves é o conjunto dos naturais $N = \{0, 1, 2, 3, \dots\}$. Assim, se as chaves não são números naturais, devemos encontrar uma maneira de interpretá-las como números naturais. Por exemplo, com caracteres e strings, podemos utilizar o código ASCII para definir uma chave numérica (inteira).

O método da divisão

No método da divisão, a ideia consiste em mapear uma chave k para um dos m slots calculando o resto da divisão de k por m (lembre-se que o resto da divisão de k por m sempre será um número entre 0 e $m - 1$). Ou seja, a função hash é dada por:

$$h(k) = k \bmod m$$

Por exemplo, se a Tabela de Espalhamento possui tamanho $m = 12$ e temos a chave $k = 100$, então ela será mapeada para o slot 4, pois:

$$\begin{array}{r} 100 \mid 12 \\ \quad \quad | \\ \quad \quad \underline{4} \quad 8 \end{array}$$

Como a função hash requer apenas uma operação matemática ela é muito rápida de ser realizada, com complexidade $O(1)$. É importante ressaltar que ao utilizar o método da divisão, devemos evitar alguns valores de m . Por exemplo, deve-se evitar escolher m como uma potência de 2. Note que se $m = 2^p$, a função hash torna-se:

$$h(k) = k \bmod 2^p$$

Note que toda potência de 2 quando expressa em binário, tem a forma:

$$\begin{aligned} 2^1 &= 10 \\ 2^2 &= 100 \\ 2^3 &= 1000 \\ 2^4 &= 10000 \\ 2^5 &= 100000 \\ 2^6 &= 1000000 \\ \dots \end{aligned}$$

o que faz com que o valor da função hash seja exatamente os p bits de ordem inferiores de k. Esse padrão gera códigos que não são igualmente prováveis, ou seja, tende-se a ter uma concentração grande em alguns valores da função hash.

Números primos são excelentes escolhas! Particularmente quando são próximos de alguma potência de 2. Lembre-se que números primos são divisíveis apenas por 1 e por ele mesmo, o que garante um bom espalhamento das chaves.

O método da multiplicação

O método da multiplicação para criar funções hash opera em dois passos principais. Primeiramente, nós multiplicamos a chave k por uma constante A no intervalo $0 < A < 1$ e extraímos a parte fracionária de kA.

Então, multiplicamos esse valor por m e tomamos o piso do resultado. Em resumo, a função hash é dada por:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

onde operação $kA \bmod 1$ denota a parte fracionária de kA, que é dada por $kA - \lfloor kA \rfloor$.

Uma possível vantagem desse método é que a escolha do valor de m (tamanho da Tabela de Espalhamento) não é crítico. Diferentemente do método da divisão, define-se m como sendo uma potência de 2, ou seja, $m = 2^p$, para p inteiro.

Uma observação importante é que, embora esse método funcione com qualquer valor de A, ele funciona melhor com alguns valores de A do que outros. A escolha do valor ótimo de A depende das características dos dados que serão armazenados (chaves). Um valor particularmente adequado e sugerido por Knuth é

$$A = \frac{\sqrt{5}-1}{2} \approx 0.6180339887\dots$$

Hashing universal

Suponha que alguém conheça a função hash de uma Tabela de Espalhamento T. Então, essa pessoa pode, de forma maliciosa, escolher chaves de forma que todos os valores hash são iguais, fazendo com que todas sejam mapeadas para o mesmo slot, o que irá tornar a complexidade da busca O(n) no caso médio. Em outras palavras, é possível “quebrar” a Tabela de Espalhamento!

Para evitar esse tipo de problema, foi proposta a estratégia de hashing universal, que basicamente consiste escolher uma função de hash aleatoriamente de modo a torná-la independente das chaves que serão armazenadas na estrutura de dados. Desse modo, tornamos a Tabela de Espalhamento mais robusta a ataques maliciosos e adversos.

Seja H uma coleção finita de funções hash que mapeiam um dado universo U de chaves no intervalo $\{0, 1, \dots, m-1\}$. Dizemos que H é universal se para cada par de chaves distintas $k, l \in U$, o número de funções hash $h \in H$ para as quais $h(k) = h(l)$ é no máximo $|H|/m$.

Em outras palavras, com uma função hash selecionada aleatoriamente de H, a probabilidade de uma colisão entre chaves distintas $k, l \in U$ não é maior que $1/m$ (o que é equivalente a hipótese de

hashing simples uniforme, ou seja, se $h(k)$ e $h(l)$ fossem escolhidos aleatoriamente do conjunto $\{0, 1, \dots, m-1\}$ a partir de uma distribuição uniforme.

A pergunta que surge é: como desenvolver uma classe universal de funções hash?

Iniciamos escolhendo um número primo p que seja grande o suficiente para que toda possível chave $k \in [0, p-1]$. Defina os conjuntos Z_p e Z_p^* como:

$$Z_p = \{0, 1, 2, \dots, p-1\}$$

$$Z_p^* = \{1, 2, \dots, p-1\}$$

Podemos definir agora uma função hash h_{ab} parametrizada para $a \in Z_p^*$ e $b \in Z_p$ utilizando uma transformação linear seguida de uma redução módulo p e outra redução módulo m :

$$h_{ab}(k) = ((ak + b) \text{ mod } p) \text{ mod } m$$

A família de todas as funções hash é definida por:

$$H_{pm} = \{h_{ab} : a \in Z_p^* \wedge b \in Z_p\}$$

Cada uma das funções hash dessa família mapeia uma chave $k \in Z_p$ para um slot $s \in Z_m$. Essa classe de funções hash tem a desejável propriedade de que o tamanho m da Tabela de Espalhamento pode ser escolhido arbitrariamente, não precisa necessariamente ser primo.

Note que como temos $(p-1)$ escolhas para a e p escolhas para b , a família de funções hash H_{pm} contém $p(p-1)$ funções hash, o que representa um número elevado, uma vez que p deve ser um número primo grande o suficiente.

Mas como o hashing universal funciona na prática?

Se a cada inserção de nova chave utilizarmos valores distintos de a e b , como iremos saber os valores no momento de efetuar uma busca? Seria totalmente caótico!

A ideia consiste em escolher uma função hash h_{ab} arbitrária e seguir utilizando essa função até que, por algum critério objetivo, seja detectado que o espalhamento das chaves não está bom o suficiente (chaves concentram-se em poucos slots). A partir desse momento, deve-se escolher outra função hash h'_{ab} arbitrariamente, realizar o rehash de todas as chaves já armazenadas na Tabela Hash (reespalhamento) e então utilizar h'_{ab} para futuras consultas.

Endereçamento aberto (open addressing)

No endereçamento aberto, todas as chaves são armazenadas diretamente na Tabela de Espalhamento, sendo que não é permitido ter múltiplas chaves mapeadas para um mesmo slot (cada chave deve ter um slot específico). Dessa forma, a Tabela de Espalhamento pode “encher-se de chaves” de modo que não seja possível fazer mais inserções. Uma consequência disso é que o fator de carga α nunca pode exceder 1, ou seja, temos $\alpha \leq 1$. Os algoritmos a seguir ilustram como são os processos de inserção e busca em Tabelas de Espalhamento de endereçamento aberto.

A ideia básica da inserção de uma chave consiste em parametrizar a função hash utilizando uma variável contadora i , de modo que na primeira tentativa atingimos um slot k , se ele estiver ocupado,

na segunda tentativa atingimos um outro slot k' , se ele estiver ocupado, na terceira tentativa atingimos um novo slot k'' , e assim sucessivamente.

```

Hash_Insert(T, k) {
    i = 0
    repeat {
        q = h(k, i)
        if T[q] == NIL {
            T[q] = k
            return q
        }
        else
            i = i + 1
    } until i == m
    error "overflow"
}

Hash_Search(T, k) {
    i = 0
    repeat {
        q = h(k, i)
        if T[q] == k
            return q
        i = i + 1
    } until T[q] == NIL or i == m
    return NIL
}

```

A questão primordial aqui é como definir a função hash $h(k, i)$. Veremos a seguir 3 métodos distintos: linear probing, quadratic probing e double hashing.

Linear probing (sondagem linear)

A ideia dessa abordagem consiste em utilizar uma função hash auxiliar $h': U \rightarrow \{0, 1, \dots, m-1\}$ na definição da função hash $h(k, i)$:

$$h(k, i) = (h'(k) + i) \bmod m$$

para $i = 0, 1, \dots, m-1$. Se uma colisão ocorre em $h(k, 0)$, então verificamos $h(k, 1)$ e assim sucessivamente. Dessa forma, sempre que uma posição está ocupada, devemos tentar a posição imediatamente posterior.

Um problema com a sondagem linear é que ela causa um efeito chamado de primary clustering (agrupamento primário), que é a formação de longas sequências de chaves, o que prejudica a inserção de novas chaves em T (aumenta o tempo gasto para percorrer toda sequência).

Quadratic probing (sondagem quadrática)

A ideia é similar a sondagem linear, mas com a utilização de incrementos não lineares em $h(k, i)$

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

onde $h'(k)$ é uma função hash auxiliar, c_1 e c_2 são duas constantes auxiliares e $i = 0, 1, \dots, m-1$. A posição inicial a ser sondada é $T[h'(k)]$ e caso ela esteja ocupada as próximas posições a serem sondadas terão offsets (deslocamentos) que dependem de forma quadrática do parâmetro i . Ou seja, ela tende a ser melhor que a sondagem linear para espalhar mais as chaves ao longo da estrutura. Porém, assim como a sondagem linear, se duas chaves tiverem a mesma sonda inicial posição, então suas sequências de sonda são as mesmas, uma vez que $h(k_1, 0) = h(k_2, 0)$ implica $h(k_1, i) = h(k_2, i)$ para todo i . Essa propriedade leva a uma forma mais branda de agrupamento, chamada secondary clustering (agrupamento secundário).

Double hashing (espalhamento duplo)

O espalhamento duplo oferece uma das melhores estratégias para o endereçamento aberto em Hash Tables porque as permutações produzidas por ele possuem muitas propriedades de permutações escolhidas aleatoriamente e conseguem atenuar os efeitos de agrupamentos primário e secundário. A função hash neste caso é definida como:

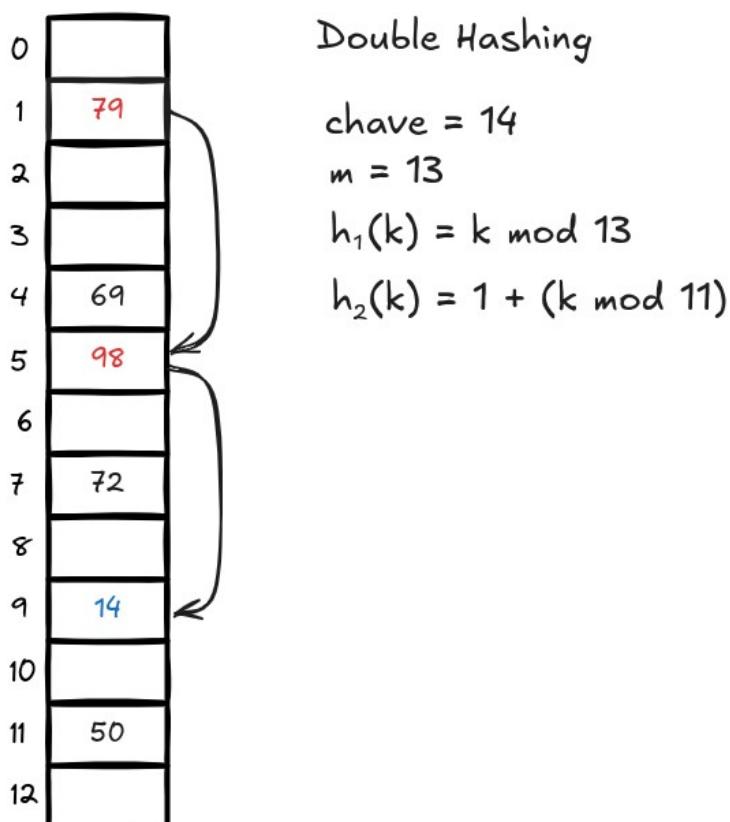
$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$$

onde temos duas funções hash auxiliares $h_1(k)$ e $h_2(k)$. A sondagem inicial é realizada na posição $T[h_1(k)]$ (pois $i=0$). As próximas posições de sondagem são deslocamentos da posição anterior de tamanho proporcional ao resto de $h_2(k)$ por m . Ou seja, diferentemente das sondagens linear e quadrática, aqui a posição inicial e os deslocamentos (offsets) são variáveis, o que contribui muito para evitar colisões. É interessante que o valor de $h_2(k)$ seja relativamente primo com o valor do tamanho da Tabela de Espalhamento, m (não tenham ou tenham poucos divisores em comum). Uma forma simples de garantir isso é fazer m igual a uma potência de 2 e definir a função $h_2(k)$ de modo que ela sempre retorne um número ímpar. Outra alternativa consiste em fixar m primo e definir $h_2(k)$ de modo que ela sempre retorne um inteiro positivo menor que m . Por exemplo:

$$h_1(k) = k \bmod m$$

$$h_2(k) = 1 + (k \bmod m')$$

com m' sendo um pouco menor que m , como $m' = m - 1$. A figura a seguir ilustra um exemplo de espalhamento duplo.



Aqui, temos:

$$\begin{aligned} m &= 13 \\ h_1(k) &= k \bmod 13 \\ h_2(k) &= 1 + (k \bmod 11) . \end{aligned}$$

Desejamos inserir a chave 14. No $i=0$, como $14 \bmod 13 = 1$, tentamos na posição 1, que está ocupada. Sendo assim, no $i=1$, o próximo slot a ser examinado será $1 + 14 \bmod 11 = 1 + 3 = 4$, o que resulta em $(1 + 4) \bmod 13 = 5$. Tentamos no slot 5, que novamente está ocupado. Assim, em $i=2$, temos $(1 + 2*4) \bmod 13 = 9 \bmod 13 = 9$. A posição está livre e a inserção finaliza com sucesso.

Exercício: Considere a inserção das chaves 10, 22, 31, 4, 15, 28, 17, 88, 59 em uma Tabela de Espalhamento de tamanho $m = 11$, utilizando endereçamento aberto com uma função auxiliar de hash $h'(k) = k$. Explique o processo de inserção ilustrando com um diagrama como fica a estrutura se utilizarmos:

- a) Sondagem linear (linear probing)
- b) Sondagem quadrática (quadratic probing)

Análise da complexidade no endereçamento aberto

A seguir, veremos resultados importantes acerca do espalhamento por endereçamento aberto.

Teorema: Dada uma Tabela de Espalhamento com endereçamento aberto e um fator de carga $\alpha = n/m < 1$, o número esperado de sondagens na busca por uma chave $k \notin T$ (não pertence ao conjunto) é no máximo $1/(1 - \alpha)$, sob hipótese de hashing uniforme.

Prova:

1. Se $\alpha < 1$, existe ao menos 1 slot vazio na Tabela Hash.
2. Assim, em uma busca sem sucesso, todos os acessos com exceção do último, são feitos em slots ocupados.
3. Seja X a variável aleatória que denota o número de sondagens em slots ocupados na busca. Defina A_i , $i = 1, 2, \dots$, como o seguinte evento: a i -ésima sondagem ocorre e retorna um slot ocupado.
4. Então, o evento $\{X \geq i\}$ é dado por $A_1 \cap A_2 \cap \dots \cap A_{i-1}$.
5. Logo, temos:

$$P(A_1 \cap A_2 \cap \dots \cap A_{i-1}) = P(A_1)P(A_2 | A_1)P(A_3 | A_1 \cap A_2) \dots P(A_{i-1} | A_1 \cap A_2 \cap \dots \cap A_{i-2})$$

6. Como há n elementos e m slots, $\frac{n}{m}$.
7. Para $j > 1$, $P(A_j | A_1 \cap A_2 \cap \dots \cap A_{j-1}) = \frac{n-(j-1)}{m-(j-1)}$ (probabilidade da j -ésima sondagem ser em um slot ocupado, dado que as $j-1$ primeiras também estavam ocupadas).

8. Então, temos:

$$P(X \geq i) = \frac{n}{m} \frac{n-1}{m-1} \frac{n-2}{m-2} \dots \frac{n-(i-2)}{m-(i-2)}$$

o que nos permite escrever a desigualdade:

$$P(X \geq i) \leq \frac{n}{m} \frac{n}{m} \frac{n}{m} \dots \frac{n}{m} = \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

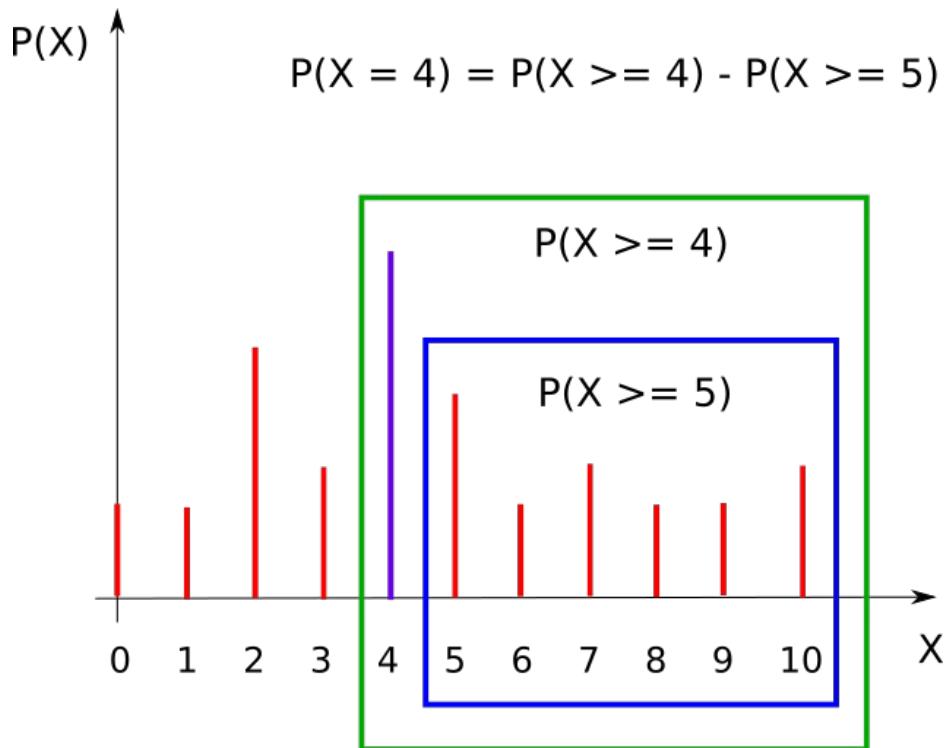
pois como $n < m$, temos $\frac{(n-j)}{(m-j)} \leq \frac{n}{m}$ para $0 \leq j \leq m$.

9. Lembre-se que o valor esperado de X é:

$$E[X] = \sum_{i=0}^{\infty} i P(X=i)$$

e como X é uma variável aleatória discreta:

$$P(X=i) = P(X \geq i) - P(X \geq i+1)$$



Então, podemos escrever o valor esperado como:

$$E[X] = \sum_{i=0}^{\infty} i [P(X \geq i) - P(X \geq i+1)]$$

Denotando $P_i = P(X \geq i)$, note que o somatório nada mais é que:

$$0(P_0 - P_1) + 1(P_1 - P_2) + 2(P_2 - P_3) + 3(P_3 - P_4) + 4(P_4 - P_5) + \dots = P_1 + P_2 + P_3 + P_4 \dots$$

ou seja:

$$E[X] = \sum_{i=1}^{\infty} P(X \geq i)$$

10. Como temos n chaves em T, há no máximo n slots ocupados, de modo que:

$$E[X] = \sum_{i=1}^{\infty} P(X \geq i) = \sum_{i=1}^n P(X \geq i) + \sum_{i \geq n+1} P(X \geq i) = \sum_{i=1}^n P(X \geq i) + 0$$

o que nos leva a:

$$E[X] = \sum_{i=1}^n \alpha^{i-1} \leq \sum_{i=1}^{\infty} \alpha^{i-1}$$

Por uma simples substituição de variáveis, seja $k = i - 1$. Então, temos que o limite inferior do somatório é $k = 0$. Dessa forma, vale a seguinte desigualdade:

$$E[X] \leq \sum_{k=0}^{\infty} \alpha^k = \alpha^0 + \alpha^1 + \alpha^2 + \dots$$

O lado direito da desigualdade é a soma de uma P. G. infinita de razão α . Logo, temos que:

$$E[X] \leq \frac{1}{1-\alpha}$$

Se α é constante, o que significa que $m = cn$, a busca possui complexidade $O(1)$. Por exemplo, se tabela está 50% ocupada, são necessários no máximo 2 acessos, se a tabela está 75% ocupada, são necessários no máximo 4 acessos, se a tabela está 90% ocupada, são necessários no máximo 10 acessos, e assim sucessivamente (mesmo que n seja um valor alto).

Corolário: A inserção de uma chave k em uma Tabela Hash com endereçamento aberto com fator de carga $\alpha < 1$ requer no máximo $\frac{1}{1-\alpha}$ sondagens, no caso médio.

Esse resultado decorre diretamente do teorema anterior. A ideia é que para um elemento ser inserido, primeiramente deve-se verificar se a chave não está na estrutura e inserí-la na posição subsequente, o que é justamente o resultado do teorema anterior.

Teorema: Dada uma Tabela de Espalhamento com endereçamento aberto e fator de carga $\alpha < 1$, o número esperado de sondagens em uma busca terminada com sucesso (encontra o elemento no conjunto) é no máximo

$$\frac{1}{\alpha} \log \frac{1}{1-\alpha}$$

assumindo hashing simples uniforme.

Prova:

A busca pela chave k é equivalente a sequência de sondagens usada na inserção da chave k.

Se k é a $(i+1)$ -ésima chave a ser inserida em T, então o fator de carga nesse momento é:

$$\alpha = \frac{i}{m}$$

o que implica que número esperado de sondagens é no máximo:

$$E[X] \leq \frac{1}{1-\alpha} = \frac{1}{1-\frac{i}{m}} = \frac{1}{\frac{m-i}{m}} = \frac{m}{m-i}$$

Tomando a média sobre todas as n chaves da tabela:

$$\frac{1}{n} \sum_{i=0}^{n-1} \left(\frac{m}{m-i} \right) = \frac{m}{n} \sum_{i=0}^{n-1} \left(\frac{1}{m-i} \right) = \frac{1}{\alpha} \sum_{i=0}^{n-1} \left(\frac{1}{m-i} \right)$$

Por uma substituição de variáveis, seja $k = m - i$. Assim, temos:

$$\begin{aligned} i=0 &\rightarrow k=m \\ i=n-1 &\rightarrow k=m-(n-1) \end{aligned}$$

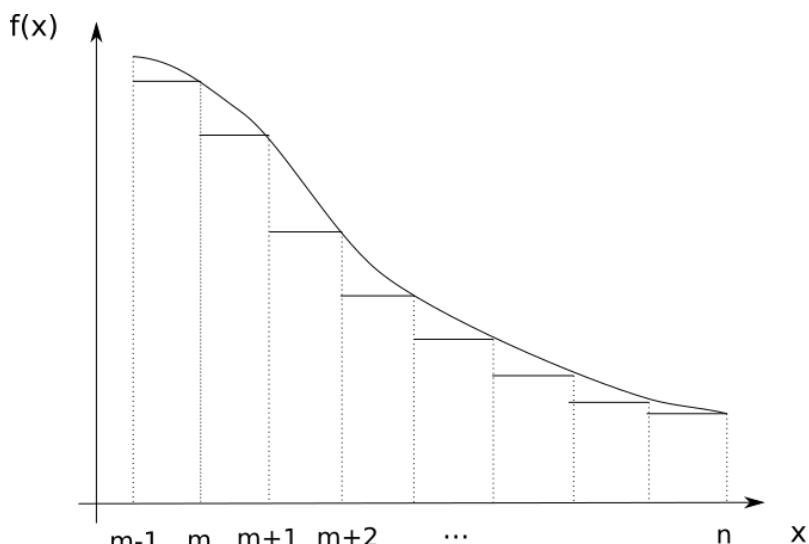
Desse forma, o somatório pode ser expresso como:

$$\frac{1}{\alpha} \sum_{k=m-n+1}^m \frac{1}{k}$$

Pode-se mostrar que para funções monotonicamente descrecentes, temos:

$$\sum_{k=m}^n f(k) \leq \int_{m-1}^n f(x) dx$$

A figura a seguir ilustra uma intuição por trás dessa desigualdade (área sob a curva é maior).



Então, podemos escrever:

$$\frac{1}{\alpha} \sum_{k=m-n+1}^m \left(\frac{1}{k} \right) \leq \frac{1}{\alpha} \int_{m-n}^m \frac{1}{x} dx = \frac{1}{\alpha} \ln x \Big|_{m-n}^m = \frac{1}{\alpha} [\ln m - \ln(m-n)] = \frac{1}{\alpha} \ln \left(\frac{m}{m-n} \right)$$

Dividindo o numerador e o denominador do logaritmo por m, finalmente chegamos em:

$$\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$$

Esse resultado é muito bom, pois é um limite ainda menor que o obtido no caso de $k \notin T$.

Por exemplo:

- Se a Tabela Hash está 50% cheia, número esperado de sondagens é aproximadamente 1.386
- Se a Tabela Hash está 75% cheia, número esperado de sondagens é aproximadamente 1.848
- Se a Tabela Hash está 90% cheia, número esperado de sondagens é aproximadamente 2.558

Na utilização de Tabelas Hash, como podemos evitar colisões? Para responder a essa questão, iremos estudar um pouco sobre probabilidades de colisão.

Calculando probabilidades de colisões

Considere uma Tabela de Espalhamento de tamanho m com n chaves.

Pergunta-se: Qual a probabilidade de que ocorra pelo menos uma colisão no conjunto de n chaves?

Para responder a essa pergunta, precisamos esclarecer um equívoco muito comum: de que colisões em uma tabela hash só acontecem quando ela está quase cheia. Por exemplo, alguns podem acreditar que se a tabela hash estiver 25% cheia, as colisões ocorreriam com uma probabilidade de 25%. No entanto, essa suposição é incorreta!

O paradoxo do aniversário

Suponha um grupo de pessoas. Qual é o número mínimo de indivíduos necessários neste grupo para que haja mais de 50% de probabilidade de duas pessoas fazerem aniversário no mesmo dia? Sabemos que deve haver $365+1=366$ pessoas para ter uma probabilidade de 100%. Então, somos levados a pensar que devem ter ao menos 183 pessoas no grupo para termos uma probabilidade maior que 50%, certo? Incorreto!

Surpreendentemente, 23 pessoas em um grupo são o suficiente para ter uma probabilidade maior que 50% de que duas pessoas compartilhem o mesmo aniversário! É difícil de acreditar? Vamos ver o porque disso.

A probabilidade de você compartilhar o mesmo aniversário com alguém é 1/365, e a probabilidade de ter aniversários diferentes é 364/365. No entanto, quando você considera todos os pares possíveis, a probabilidade cresce muito mais rápido. Lembre-se, não se trata de encontrar alguém que tenha o mesmo aniversário que você, mas sim de encontrar quaisquer duas pessoas no grupo que compartilhem o mesmo aniversário. Em um grupo de 23 pessoas, o número de pares possíveis é $(23 \times 22)/2 = 253$.

A probabilidade de duas pessoas não terem o mesmo aniversário é 364/365. Temos 253 pares, então a probabilidade de nenhum dos pares compartilhar o mesmo aniversário é exatamente $(364/365)^{253}$. Consequentemente, a probabilidade de pelo menos um par compartilhar o mesmo aniversário é:

$$1 - \left(\frac{364}{365} \right)^{253} \approx 50.05\%.$$

Vamos explorar como o paradoxo de aniversário funciona com tabelas hash e qual é a probabilidade de colisões em uma tabela hash. Será que a probabilidade de colisão de 25% ocorre quando a tabela hash está 25% preenchida? Vamos descobrir.

Seja m é o número de posições na tabela hash e n é o número de chaves a serem inseridas. Então a probabilidade de que haja pelo menos uma colisão entre n chaves inseridas aleatoriamente é

$$P_{m,n}(\text{colisão}) = 1 - P_{m,n}(\text{não colisão})$$

$$P_{m,n}(\text{colisão}) = 1 - P_{m,n}(\text{sem colisão na primeira chave}) \times \dots \times P_{m,n}(\text{sem colisão na } n\text{-ésima chave})$$

$$P_{m,n}(\text{colisão}) = 1 - 1 \times \frac{m-1}{m} \times \frac{m-2}{m} \times \frac{m-3}{m} \times \dots \times \frac{m-n+1}{m} = \frac{m!}{(m-n)! m^n}$$

Usando essa equação, se tomarmos $m = 365$ e $n = 23$, obtemos a probabilidade de colisões como 50,07%, que é a probabilidade de duas pessoas compartilharem o mesmo aniversário em um grupo de 23 pessoas. E se você tomar $n = 57$, a probabilidade é 99,01%, o que significa que é quase certo que duas pessoas podem compartilhar o mesmo aniversário em um grupo de 57 pessoas.

Agora, considere uma tabela hash com 1000 localizações, ou seja, $m = 1000$. Vamos examinar a probabilidade de colisões se a tabela estiver apenas 25% preenchida. ou seja, $n = 250$. A probabilidade de colisões é incrivelmente de 99,9999999999984%! Então é quase certo que haverá uma colisão. Isso indica que uma colisão é certa quando a tabela está apenas 25% preenchida.

Calculando alguns valores de probabilidade para diferentes valores de n , temos:

$$n = 38 \rightarrow P = 50.93\%$$

$$n = 50 \rightarrow P = 71.22\%$$

$$n = 75 \rightarrow P = 94.19\%$$

$$n = 100 \rightarrow P = 99.40\%$$

$$n = 125 \rightarrow P = 99.97\%$$

Como podemos ver, 38 chaves em uma tabela hash de tamanho 1000 produz uma probabilidade de colisão maior que 50%. Com uma ocupação de 125 chaves, o que equivale a 25% de ocupação, nos leva a uma probabilidade de colisão de 99.97%.

Número total de colisões

Inserir uma chave em uma tabela hash é como escolher um item aleatório do conjunto de n itens e jogá-lo em um conjunto de m caixas. Não podemos prever de antemão em qual caixa o item vai parar. É possível que o item que jogamos acabe em uma caixa que já tenha um item (o que significa uma colisão).

A primeira chave que inserimos acabou em alguma caixa X. A probabilidade de que a segundo segunda chave acabe na mesma caixa X que o primeiro item é $1/m$. Então a probabilidade de que os itens restantes ($n - 1$) acabem na mesma caixa X que o primeiro item é $(n - 1)/m$.

Suponha que não haja colisões na primeira caixa onde a primeira chave foi inserida. A segunda chave que inserimos acabou em alguma outra caixa Y. A probabilidade de que a terceira chave acabe na mesma caixa Y é $1/m$. Então a probabilidade de que as chaves restantes ($n - 2$) acabem na mesma caixa Y é $(n - 2)/m$.

Assim, o número médio de colisões é dado por:

$$N_m = \sum_{i=1}^{n-1} \frac{i}{m} = \frac{1}{m} \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2m}$$

Vamos ver o número total médio de colisões possíveis quando $m = 1000$, e o número de itens a serem inseridos é n .

$$\begin{aligned} n = 100 &\rightarrow \# \text{ colisões} \approx 5 \\ n = 250 &\rightarrow \# \text{ colisões} \approx 31 \\ n = 500 &\rightarrow \# \text{ colisões} \approx 125 \\ n = 750 &\rightarrow \# \text{ colisões} \approx 281 \\ n = 1000 &\rightarrow \# \text{ colisões} \approx 500 \end{aligned}$$

Então, se tivermos uma tabela hash com m slots e se o número total de itens a serem inseridos n for igual ao número total de slots m em uma tabela hash, então haverá aproximadamente $m/2$ colisões totais, o que não é um número pequeno! Então, a melhor maneira de reduzir colisões é tornar a tabela hash maior (pelo menos 1,3 vezes o número de itens a serem inseridos).

Sendo assim, precisamos pensar em formas mais eficientes de garantir de não haja colisões. Uma delas é o hashing perfeito, que será apresentado a seguir.

Perfect hashing (Espalhamento perfeito)

Vimos que as Tabelas de Espalhamento funcionam muito bem no caso médio. Porém, as técnicas anteriores não são as melhores quando analisamos o pior caso.

Há uma situação em que é possível atingir complexidade $O(1)$ para busca em Tabelas de Espalhamento mesmo no pior caso: quando o conjunto de chaves é estático (as chaves são fixas). Por exemplo, algumas aplicações em que o conjunto de chaves é estático (não muda) são: no desenvolvimento de um compilador, as palavras reservadas de uma linguagem de programação são estáticas, no processamento de dados, o conjunto de nomes de arquivos armazenados em um CD-ROM também não se altera. Na prática isso significa que, uma vez criada a Tabela de Espalhamento, não serão necessárias inserções nem remoção de chaves. O esquema em questão chama-se *Perfect Hashing*.

A ideia do *Perfect Hashing* é criar uma estratégia de hashing universal em dois níveis:

1. O primeiro nível é essencialmente um esquema de hashing com encadeamento, em que temos que espalhar n chaves em m slots usando uma função de hash h .
2. Ao invés de criar uma lista encadeada das chaves que são mapeadas para um mesmo slot j , a ideia consiste usar uma segunda, porém menor, Tabela de Espalhamento S_j , com uma outra função hash associada. Através de uma escolha adequada das funções hash, pode-se garantir que não haverá colisões no segundo nível. A única restrição é que o tamanho da Tabela de Espalhamento S_j , denotado por m_j , deve ser igual ao quadrado do número de elementos armazenados em S_j .

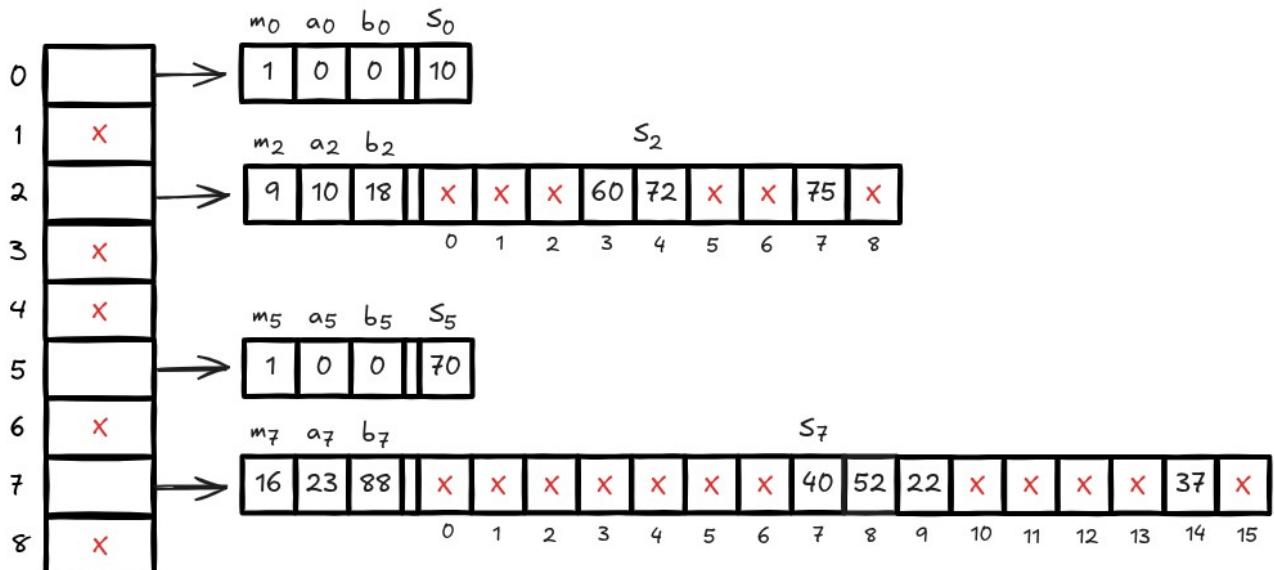
Considere o seguinte exemplo: desejamos utilizar o perfect hashing para armazenar o conjunto estático de chaves a seguir: $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. A função de hash externa (global) é dada por:

$$h(k) = ((ak+b) \bmod p) \bmod m$$

com $a=3$, $b=42$, $p=101$ e $m=9$. Como $h(75) = 2$, a chave 75 deve ser armazenada no slot 2 da tabela T. Uma Tabela de Espalhamento secundária S_j , armazena as chaves mapeadas para o slot j. Sendo assim, 75 é armazenado em S_2 . Mas onde? Aqui entra a função hash do nível 2. Cada tabela S_j deve ter tamanho $m_j = n_j^2$, além de sua própria função hash:

$$h_j(k) = ((a_j k + b_j) \bmod p_j) \bmod m_j$$

em que no caso de S_2 , os valores são $a_2=10$, $b_2=18$ e $m_2=9$. Como $h_2(75) = 7$, a chave 75 vai ser armazenada na posição 7 de S_2 . Com essa construção, é garantido que não ocorrem colisões no segundo nível de hashing, de forma que a busca tem complexidade $O(1)$ mesmo no pior caso! A Tabela Hash resultante após o Perfect Hashing é ilustrada na figura a seguir.



Um observação é que o *Perfect Hashing* possui uma limitação: antes de iniciar o processo de inserção dos elementos, primeiramente devemos saber quantos elementos cada Tabela de Espalhamento do segundo nível (S_j) terá, ou seja quais são os valores de m_j . Então, é necessária uma etapa de setup da Tabela Hash antes de populá-la com os dados.

Algoritmo básico para Perfect Hashing

Para a Tabela de Espalhamento principal, escolha $m=n$ (número de chaves), escolha um número primo p maior que a máxima das chaves. Defina os valores de a e b aleatoriamente para construir a primeira função hash: $h(k) = ((ak+b) \bmod p) \bmod m$.

Teste a função hash $h(k)$ como segue:

1. Para cada chave k , encontre seu slot $h(k)$. Mantenha um contador n_j de quantas chaves foram mapeadas para o slot j.

2. Verifique se o espaço necessário para o armazenamento é muito grande: a soma de todos os n_j^2 é maior que $2n$?

$$\sum_{j=0}^{m-1} n_j^2 > 2n$$

Se a resposta for SIM, então a função de hash $h(k)$ não é boa o suficiente. Repita o processo com outros valores de a e b para construir a função de hash primária.

Se a função de hash primária é boa o suficiente, faça:

- a.** Para cada slot S_j , defina a função de hash secundária $h_j(k)$

$$h_j(k) = ((a_j k + b_j) \bmod p_j) \bmod m_j$$

setando $p_j = p$, $m_j = n_j^2$ e escolhendo a_j e b_j aleatoriamente.

- b.** Verifique que as funções hash secundárias $h_j(k)$ não resultam em colisões nas Tabelas de Espalhamento secundárias S_j , para todo j .

Consideremos o exemplo didático a seguir.

$$K = \{8, 22, 36, 75, 61, 13, 84, 58\}$$

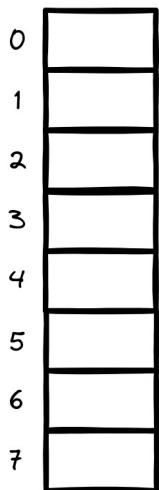
$$p = 87, a = 64, b = 5$$

$$h(k) = ((ak + b) \bmod p) \bmod m$$

$$h_j(k) = ((a_j k + b_j) \bmod p) \bmod m$$

Passo 1:

Selecione p como um número primo maior que a maior das chaves e gere a e b aleatoriamente



$$K = \{8, 22, 36, 75, 61, 13, 84, 58\}$$

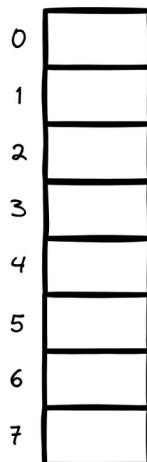
$$p = 87, a = 64, b = 5$$

$$h(k) = ((ak + b) \bmod p) \bmod m$$

$$h_j(k) = ((a_j k + b_j) \bmod p) \bmod m$$

Passo 2:

Para cada chave k , calcule $h(k)$ e conte quantas chaves caem em cada slot contagem (n_j)



0
1
2
3
4
5
6
7

0
1
2
0
1
1
1
2

$$K = \{8, 22, 36, 75, 61, 13, 84, 58\}$$

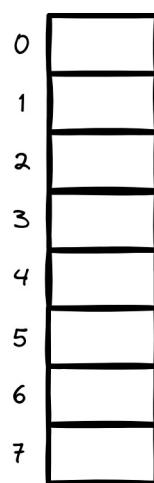
$$p = 87, a = 64, b = 5$$

$$h(k) = ((ak + b) \bmod p) \bmod m$$

$$h_j(k) = ((a_j k + b_j) \bmod p) \bmod m$$

Passo 3:

Verifique se $\sum_{j=0}^{m-1} n_j < 2n$



contagem (n_j)

0
1
2
0
1
1
1
2

$$\sum_{j=0}^{m-1} n_j = 1 + 4 + 1 + 1 + 1 + 4 = 12 < 16$$

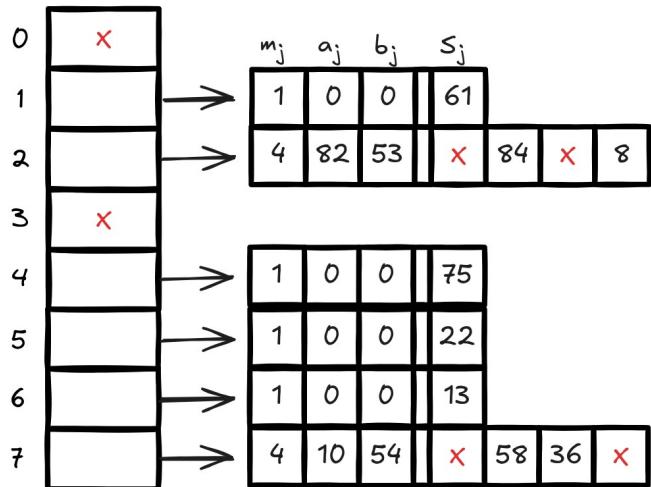
OK

$$K = \{8, 22, 36, 75, 61, 13, 84, 58\} \quad h(k) = ((ak + b) \bmod p) \bmod m$$

$$p = 87, a = 64, b = 5 \quad h_j(k) = ((a_j k + b_j) \bmod p) \bmod m$$

Passo 4:

Distribua as chaves nas tabelas hash do segundo nível



"An arrow can only be shot by pulling it backward. So, when life is dragging you back with difficulties, it means that it's going to launch you into something great. Be patient."
-- Author Unknown

Estruturas de Dados: O Filtro de Bloom

Trata-se de uma estrutura de dados probabilística para testar se um elemento pertence a um dado conjunto S (S pode ser uma estrutura de dados qualquer, i.e., lista encadeada, árvore, etc...).

Em resumo, pode ser aplicado a qualquer estrutura de dados para tornar a busca por um elemento $O(1)$.

Problema: Pode retornar falsos positivos, ou seja, uma consulta pode retornar que $x \in S$ quando de fato $x \notin S$.

Ideia geral: mapear uma chave arbitrária $x \in X$ (X é o conjunto de chaves) para um valor $y = h(x)$ usando uma função hash $h(\cdot)$

O filtro de Bloom é basicamente:

- um array de m bits em que inicialmente todos os bits possuem valor 0.
- existem k funções hash: cada uma mapeia chave x para uma posição do array.

Como escolher k e m ?

Veremos mais detalhes a seguir, mas intuição nos diz que:

- k deve ser um valor relativamente pequeno que depende da taxa de falsos positivos ϵ .
- m deve ser proporcional a k e ao número de chaves a serem armazenadas em S .

A seguir, veremos como realizar a inserção de uma chave no filtro de Bloom.

Inserção

Calcule as k funções hash para a chave x e altere os bits dessas posições para 1.

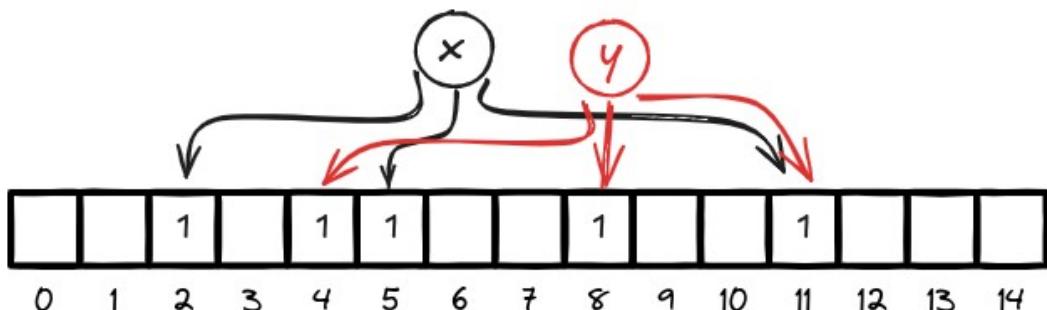
Exemplo:

Suponha um filtro de Bloom com $m = 15$ e $k = 3$. Então, temos as funções hash $h_1(x)$, $h_2(x)$ e $h_3(x)$.

Para a chave x , considere que $h_1(x) = 2$, $h_2(x) = 5$ e $h_3(x) = 11$.

Para a chave y , considere que $h_1(y) = 4$, $h_2(y) = 8$ e $h_3(y) = 11$.

Então, o filtro fica:



Note que tanto a chave x quanto a chave y apontam para o slot 11. Devido a essa característica de sobreposição, em que chaves compartilham slots, o filtro de Bloom não permite remoções. Uma vez inserida uma chave, jamais removida!

Porém, podem ocorrer faltos positivos, isto é, a busca por um elemento x pode retornar True quando na verdade x não pertence a estrutura de dados. No projeto de um filtro de Bloom, precisamos estudar como minimizar essa probabilidade de falsos positivos.

Análise da probabilidade de falso positivo

Se filtro de Bloom possui m bits, sob a hipótese de hashing simples uniforme, a probabilidade de um bit não ter valor 1 é dada por:

$$\left(1 - \frac{1}{m}\right) \quad (1/m \text{ é a probabilidade de um bit ser igual a 1})$$

Se existem k funções hash, a probabilidade de que um bit não seja modificado por nenhuma delas é:

$$\left(1 - \frac{1}{m}\right)^k$$

Sabemos que, por definição:

$$e = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n$$

o que nos permite escrever

$$e = \lim_{x \rightarrow 0} (1+x)^{1/x} \quad (\text{pois se } n \text{ tende a infinito, seu inverso tende a zero})$$

Faça $y = -\frac{1}{m}$. Então, se $m \rightarrow \infty$, temos que $y \rightarrow 0$. Sendo assim, podemos escrever:

$$\lim_{m \rightarrow \infty} \left(1 - \frac{1}{m}\right)^m = \lim_{m \rightarrow \infty} \left(1 + \left(-\frac{1}{m}\right)\right)^m = \lim_{y \rightarrow 0} (1+y)^{-1/y} = \lim_{y \rightarrow 0} [(1+y)^{1/y}]^{-1} = \frac{1}{\lim_{y \rightarrow 0} [(1+y)^{1/y}]} = e^{-1}$$

Então, para m grande o suficiente, temos que:

$$\left(1 - \frac{1}{m}\right)^k = \left[\left(1 - \frac{1}{m}\right)^m\right]^{k/m} \approx e^{-k/m}$$

Após a inserção de n chaves, a probabilidade de que um bit arbitrário ainda seja zero é dada por:

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

Assim, a probabilidade de que um bit arbitrário seja igual a 1 após n inserções é dada por:

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-kn/m}$$

Neste ponto, vamos tentar a pertinência de uma chave z que não pertence ao conjunto. Se temos k funções hash, a probabilidade de que todas os k bits sejam iguais a 1, o que faria o algoritmo erroneamente dizer que z faz parte do conjunto, é dada por:

$$\epsilon(k, m, n) \approx (1 - e^{-kn/m})^k \quad (*)$$

Exemplo: Suponha um conjunto de $n = 1000$ chaves para serem armazenadas em um filtro de Bloom com $m = 10000$ slots utilizando $k = 10$ funções hash. Calcule a probabilidade de falso positivo.

$$\epsilon(k, m, n) \approx (1 - e^{-1})^{10} \approx 0.010185$$

ou seja a probabilidade de falso positivo é aproximadamente 1%.

Desejamos encontrar o valor de k que minimiza essa probabilidade. A condição necessária para isso é:

$$\frac{\partial \epsilon(k, m, n)}{\partial k} = 0$$

Podemos reescrever (*) como:

$$\epsilon(k, m, n) \approx \exp(\ln((1 - e^{-kn/m})^k)) = \exp(k \ln((1 - e^{-kn/m})))$$

Note que minimizar $\epsilon(k, m, n)$ é equivalente a minimizar:

$$F(k) = k \ln((1 - e^{-kn/m})) \quad (\text{expoente do exp})$$

Pela regra do produto, temos:

$$F'(k) = \ln((1 - e^{-kn/m})) + \frac{\frac{kn}{m} e^{-kn/m}}{1 - e^{-kn/m}} = 0 \quad (**)$$

Seja $p = e^{-kn/m}$. Então, temos que:

$$\ln(p) = -\frac{kn}{m} \quad (\#)$$

o que nos permite escrever (**) como:

$$\ln(1-p) - \ln(p) \frac{p}{1-p} = 0$$

o que nos leva a:

$$(1-p)\ln(1-p) = p\ln(p)$$

o que é o mesmo que:

$$\ln(1-p)^{1-p} = \ln(p)^p$$

Logo, devemos ter que:

$$(1-p)^{1-p} = p^p$$

o que implica que $p=1-p$, ou seja, $p=\frac{1}{2}$. Dessa forma, voltando para (#), temos:

$$\ln\left(\frac{1}{2}\right) = -\frac{kn}{m}$$

o que finalmente nos leva ao valor ótimo de k, dado por:

$$k = \frac{m}{n} \ln 2$$

onde m é o número de slots do filtro de Bloom e n é o número de chaves.
Por exemplo, se m = 15 e n = 3:

$$k = 5 \ln 2 \approx 3.465$$

o que implica em dizer que o número ótimo de funções hash deve ser 3.

A ideia em um projeto de um filtro de Bloom consiste em fixar a probabilidade de falso positivo para obter o tamanho do filtro como função de n. Sabemos que:

$$\epsilon(k, m, n) \approx (1 - e^{-kn/m})^k$$

Substituindo k pelo valor ótimo, temos:

$$\epsilon(k, m, n) \approx (1 - e^{-\ln 2})^{\frac{m}{n} \ln 2}$$

Mas $\ln 2 = -\ln 1/2$, o que nos leva a:

$$\epsilon(k, m, n) \approx \left(\frac{1}{2}\right)^{\frac{m}{n} \ln 2}$$

Aplicando ln de ambos os lados:

$$\ln \epsilon(k, m, n) = \frac{m}{n} \ln 2 \ln(1/2) = -\frac{m}{n} \ln^2 2$$

Sendo assim, podemos escrever:

$$\frac{m}{n} = -\frac{\ln \epsilon(k, m, n)}{(\ln 2)^2}$$

o que finalmente nos leva a:

$$m = -\frac{n \ln \epsilon}{(\ln 2)^2}$$

Por exemplo, se desejamos ter probabilidade de falso positivo igual a $\epsilon=0.01$, então:

$$m \approx 9.585n \quad (\text{se } n = 100, \text{ então } m = 959 - \text{cerca de 1 KB})$$

Se desejamos ter probabilidade de falso positivo igual a $\epsilon=0.001$, então:

$$m \approx 14.377n \quad (\text{se } n = 100, \text{ então } m = 1438 - \text{cerca de 1.5 KB})$$

Apesar de parecer muito, como o filtro de Bloom é um array de bits, a complexidade de espaço é baixa.

Vantagem: Podemos incorporar o filtro de Bloom em qualquer estrutura de dados para realizar a busca por elementos em tempo constante, como por exemplo uma simples lista encadeada.

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

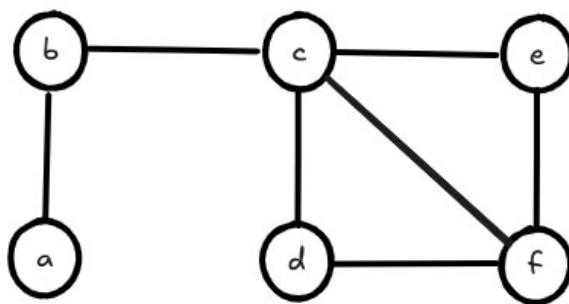
Grafos: Fundamentos Básicos

Grafos são objetos matemáticos que representam relações binárias entre elementos de um conjunto finito. Eles são utilizados na computação para representar estruturas complexas em que cada nó pode ser conectar a um número variável de vizinhos, como por exemplo, a internet e as redes sociais. Em termos gerais, um grafo consiste em um conjunto de vértices que podem estar ligados dois a dois por arestas. Se dois vértices são unidos por uma aresta, então eles são vizinhos. É uma estrutura fundamental para a computação, uma vez que diversos problemas do mundo real podem ser modelados com grafos, como encontrar caminhos mínimos entre dois pontos, alocação de recursos e modelagem de redes complexas.

Def: $G = (V, E)$ é um grafo se:

- i) V é um conjunto não vazio de **vértices**
- ii) $E \subseteq V \times V$ é uma relação binária qualquer no conjunto de vértices: conjunto de **arestas**

A figura a seguir ilustra um grafo $G = (V, E)$ arbitrário.



Denotamos por $N(v)$ o conjunto vizinhança do vértice v . Por exemplo, $N(b)=\{a, c\}$.

Def: Grau de um vértice v : $d(v)$

É o número de vezes que um vértice v é extremidade de uma aresta. Num grafo básico simples, é o mesmo que o número de vizinhos de v . Ex: $d(a) = 1$, $d(b) = 2$, $d(c) = 4$, , ...

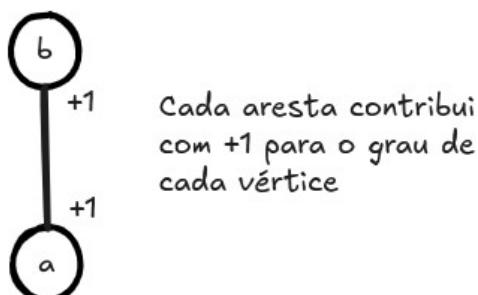
Def: A lista de graus de $G = (V, E)$ é a lista que armazena os graus dos vértices em ordem crescente.

$$L_G = (1, 2, 2, 2, 3, 4)$$

Handshaking Lema: A soma dos graus dos vértices de G é igual a duas vezes o número de arestas.

$$\sum_{i=1}^n d(v_i) = 2m \quad (\text{condição de existência para grafos})$$

onde $n = |V|$ e $m = |E|$ denotam respectivamente o número de vértices e arestas.



Prova: (por indução)

Seja $P(n)$ definido como:

$$P(n): \sum_{i=1}^n d(v_i) = 2m$$

=> BASE: Note que nesse caso, $n = 1$, o que implica dizer que temos um único vértice. Então, para todo $m \geq 0$ (número de arestas), a soma dos graus será sempre um número par pois ambas as extremidades das arestas incidem sobre o único vértice de G .

=> PASSO DE INDUÇÃO: Para k arbitrário, mostrar que $P(k) \rightarrow P(k+1)$

Note que para k vértices, temos:

$$P(k): \sum_{i=1}^k d(v_i) = 2m$$

Ao adicionarmos exatamente um vértice a mais, temos $k+1$ vértices:

$$P(k+1): \sum_{i=1}^{k+1} d(v_i) = 2m'$$

Ao passar de k para $k+1$ vértices, temos duas opções:

- a) o grau do novo vértice é zero;
- b) o grau do novo vértice é maior que zero.

Caso a): Nessa situação, temos o número de arestas permanece inalterado, ou seja, $m = m'$. Pela hipótese de indução e sabendo que o grau no novo vértice é zero, podemos escrever:

$$\sum_{i=1}^{k+1} d(v_i) = \sum_{i=1}^k d(v_i) + d(v_{k+1}) = 2m + 0 = 2m'$$

ou seja, $P(k+1)$ é válida.

Caso b): Nessa situação, temos que o número de arestas $m' > m$. Seja $m' = m + a$, onde a denota o número de arestas adicionadas ao inserir o novo vértice $k+1$. Então, pela hipótese de indução e sabendo que o grau do novo vértice será a , podemos escrever:

$$\sum_{i=1}^{k+1} d(v_i) = \sum_{i=1}^k d(v_i) + d(v_{k+1}) + 1 + 1 + 1 + \dots + 1$$

a vezes 1

pois para cada extremidade das arestas no novo vértice, haverá outra extremidade em algum outro vértice. Isso implica em:

$$\sum_{i=1}^{k+1} d(v_i) = 2m + a + a = 2m + 2a = 2(m + a) = 2m'$$

ou seja, $P(k+1)$ é valida. Note que mesmo que alguma das arestas inseridas possuam ambas as extremidades no novo vértice $k+1$, a soma dos graus também será igual a $2m+2a$, o que continuará validando $P(k+1)$. Portanto, a prova está concluída.

Teorema: Em um grafo $G = (V, E)$ o número de vértices com grau ímpar é sempre par.

Podemos partitionar V em 2 conjuntos: P (grau par) e I (grau ímpar). Assim,

$$\sum_{i=1}^n d(v_i) = \sum_{v \in P} d(v) + \sum_{u \in I} d(u) = 2m$$

Isso implica em

$$\sum_{u \in I} d(u) = 2m - \sum_{v \in P} d(v)$$

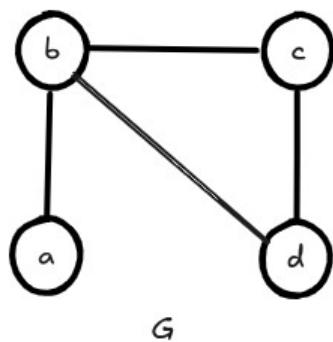
Como $2m$ é par e a soma de números pares é sempre par, resulta que a soma dos números ímpares também é par. Para que isso ocorra temos que ter $|I|$ par (número de elementos do conjunto I é par)

Representações computacionais de grafos

1. Matriz de adjacências A : matriz quadrada $n \times n$ definida como:

a) Grafos básicos simples

$$A_{i,j} = \begin{cases} 1, & j \in N(i) \\ 0, & j \notin N(i) \end{cases}$$



$$A = \begin{bmatrix} a & b & c & d \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{matrix} a \\ b \\ c \\ d \end{matrix}$$

Propriedades básicas

i) $\text{diag}(A) = 0$

ii) matriz binária

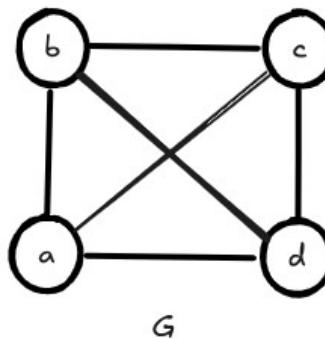
iii) $A = A^T$ (com exceção de grafos direcionados)

iv) $\sum_j A_{i,j} = d(v_i)$

v) Esparsa

vi) $O(n^2)$ em espaço – requer $\frac{n^2}{8}$ bytes para armazenamento (contíguos na memória)

2. Matriz de Incidência M: matriz $n \times m$ em que as linhas referem-se aos vértices e as colunas referem-se as arestas:

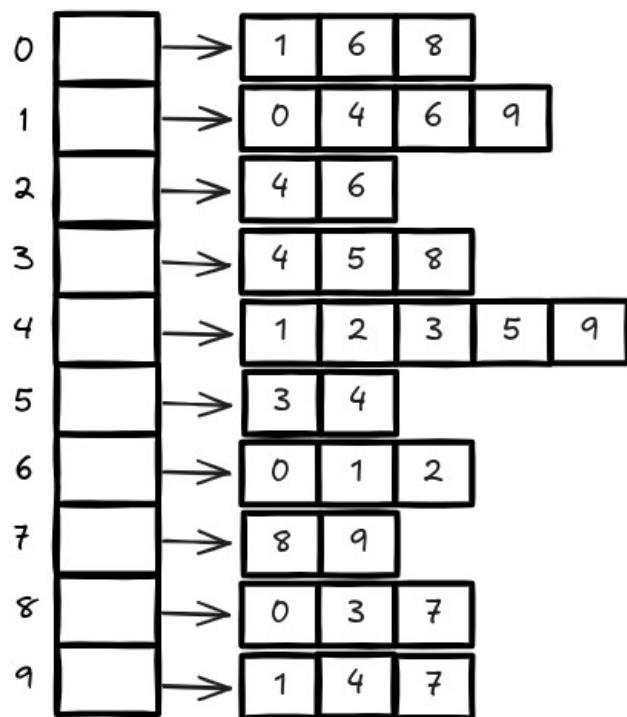


$$M = \begin{bmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \begin{array}{l} a \\ b \\ c \\ d \end{array}$$

=> $O(nm)$ em espaço – requer $\frac{nm}{8}$ bytes para armazenamento (contíguos na memória) - RUIM

3. Lista de adjacências: estrutura dinâmica em que cada nó possui uma referência para uma lista encadeada com seus vizinhos.

Lista de adjacências



Note que ao contarmos a soma dos graus dos vértices (somatório dos tamanhos de cada lista encadeada), estamos de fato contando cada aresta duas vezes. Assim, de acordo com o *Handshaking Lema*, devemos dividir esse número total por 2 para obter o número exato de arestas.

Como temos n referências e cada referência é uma lista encadeada possui $d(v_i)$ nós, temos que o número total de nós é igual a $\sum_{i=1}^n d(v_i) = 2m$, o que resulta em $O(m)$.

Pergunta: Quantos bytes são necessários para armazenar um grafo G em memória com uma lista de adjacências?

Supondo que cada inteiro ocupa 32 bits, temos que cada nó ocupa $2m\left(\frac{32}{8}\right)$ bytes, o que equivale a 8m bytes (não precisa ser contíguo na memória).

Sabendo que a densidade de um grafo é dada por $d = \frac{m}{n^2}$, quando é preferível usar uma lista de adjacências em detrimento a uma matriz de adjacências?

A lista de adjacências deve ser escolhida se:

$$8m < \frac{n^2}{8} \rightarrow \frac{m}{n^2} < \frac{1}{64} \rightarrow d < \frac{1}{64} \rightarrow d < 0.015625 \text{ (grafo esparso)}$$

Ex: Se $n = 100$ e $m = 150$, temos $d = 150/10000 = 0.015$, o que é menor que 0.01562

Lembrando que o máximo valor de densidade é dado por:

$$d = \frac{\frac{n(n-1)}{2}}{n^2} = \frac{n^2 - n}{2n^2} = \frac{1}{2} - \frac{1}{2n} = \frac{1}{2} \left(1 - \frac{1}{n}\right)$$

o que tende a 0.5 quando n cresce arbitrariamente.

"Lembre-se sempre: as últimas partes a crescer em uma árvore são os frutos."
-- Autor Anônimo

Grafos: Busca em Grafos (BFS e DFS)

Importância: como navegar em grafos de maneira determinística de modo a percorrer um conjunto de dados não estruturado, visitando todos os nós exatamente uma vez.

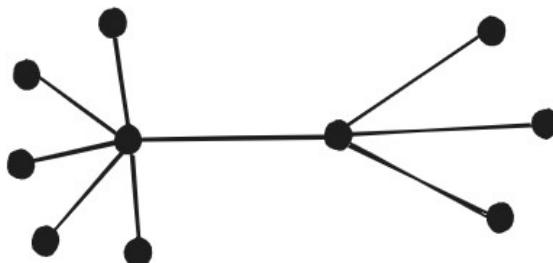
Objetivo: acessar/recuperar todos os elementos do conjunto V

Questões: De quantas maneiras podemos fazer isso? Como? Qual a melhor maneira?

Árvores e suas propriedades

Árvores são grafos especiais com diversas propriedades únicas. Devido a essas propriedades são extremamente importantes na resolução de vários tipos de problemas práticos. Veremos ao longo do curso que vários problemas que estudaremos se resumem a: dado um grafo G , extrair uma árvore T a partir de G , de modo que T satisfaça uma certa propriedade (como por exemplo, mínima profundidade, máxima profundidade, mínimo peso, mínimos caminhos, etc).

Def: Um grafo $G = (V, E)$ é uma árvore se G é acíclico e conexo.



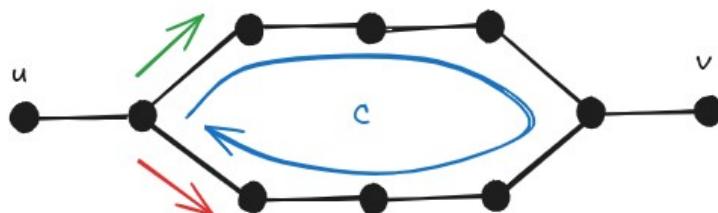
Teorema: G é uma árvore $\Leftrightarrow \exists$ um único caminho entre quaisquer 2 vértices $u, v \in V$

1. (ida) $p \rightarrow q = \neg q \rightarrow \neg p$

\nexists um único caminho entre quaisquer $u, v \rightarrow G$ não é uma árvore

a) Pode existir um par u, v tal que \nexists caminho (zero caminhos). Isso implica em G desconexo, o que implica que G não é uma árvore

b) Pode existir um par u, v tal que \exists mais de um caminho.

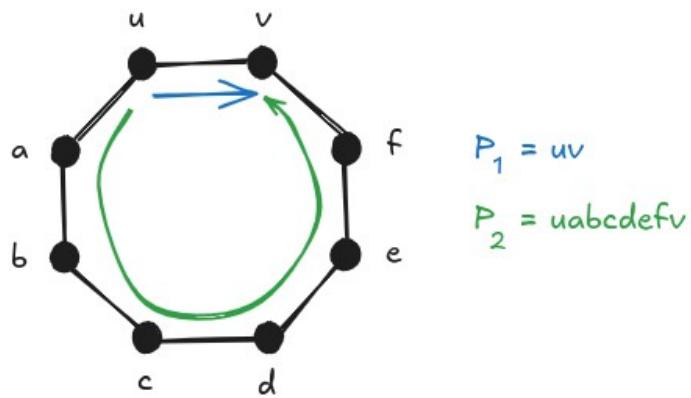


Porém neste caso temos a formação de um ciclo e portanto G não pode ser árvore.

2. (volta) $q \rightarrow p = \neg p \rightarrow \neg q$

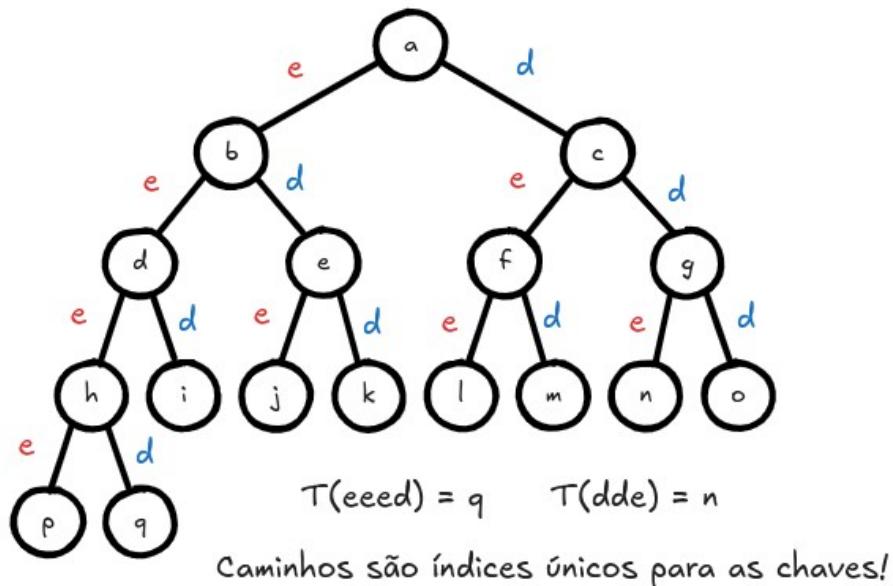
G não é árvore $\rightarrow \nexists$ único caminho entre quaisquer u, v

Para G não ser árvore, G deve ser desconexo ou conter um ciclo. Note que no primeiro caso existe um par u, v tal que não há caminho entre eles. Note que no segundo caso existem 2 caminhos entre u e v , conforme ilustra a figura

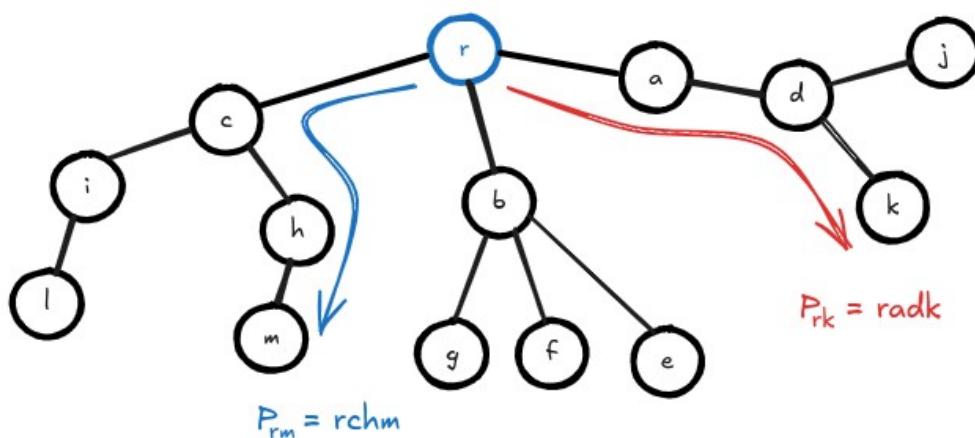


Relação entre busca em grafos e árvores

Buscar elementos num grafo G é basicamente o processo de extrair uma árvore T a partir de G. Mas porque? Como busca se relaciona com uma árvore? Isso vem de uma das propriedades das árvores. Numa arvore existe um único caminho entre 2 vértices u, v (caminho é um índice único).



Pode-se criar um esquema de indexamento baseado nos nós a esquerda e a direita. Cada elemento do conjunto possui um índice único que o recupera. No caso de árvores genéricas, o caminho faz o papel do índice único



Portanto, dado um grafo G, extrair uma árvore T com raiz r a partir dele, significa indexar unicamente cada elemento do conjunto (princípio por trás da busca).

Busca em Largura (Breadth-First Search – BFS)

Ideia geral: a cada novo nível descoberto, todos os vértices daquele nível devem ser visitados antes de prosseguir para o próximo nível

Definição das variáveis usadas no algoritmo

i) v. color : status do vértice v (existem 3 possíveis valores)

- a) WHITE: vértice v ainda não descoberto (significa que v ainda não entrou na fila Q)
- b) GRAY: vértice já descoberto (significa que v está na fila Q)
- c) BLACK: vértice finalizado (significa que v já saiu da fila Q)

ii) $\lambda(v)$: armazena a menor distância de v até a raiz

iii) $\pi(v)$: predecessor de v (onde estava quando descobri v)

iv) Q: Fila (FIFO)

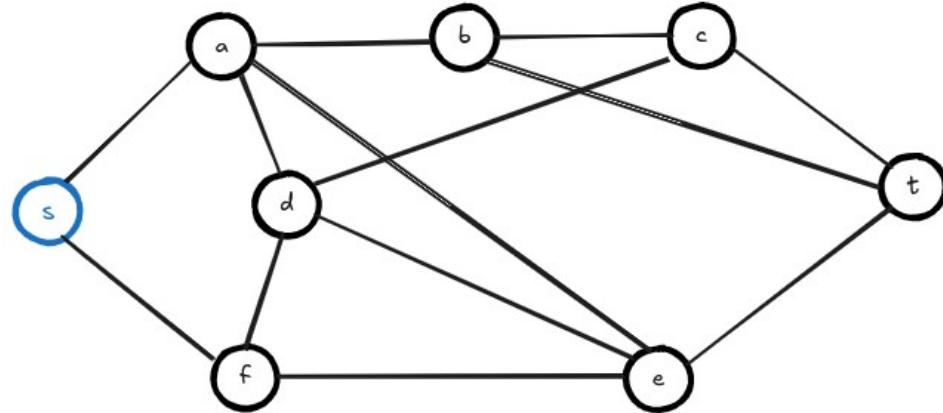
2 primitivas

- a) pop: remove elemento do início da fila
- b) push: adiciona um elemento no final da fila

ALGORITMO

```
BFS(G, s) {
    for each v ∈ V-{s} {      # inicializa as variáveis
        v.color = WHITE
        λ(v) = ∞
        π(v) = NIL
    }
    s.color = GRAY
    λ(s) = 0
    Q = ∅
    push(Q, s)                # insere raiz na fila
    while Q ≠ ∅ {              # enquanto fila não for vazia
        u = pop(Q)
        for each v ∈ N(u) {      # para todo vizinho de u
            if v.color == WHITE { # se ainda não passei aqui
                λ(v) = λ(u) + 1  # v é descendente de u
                π(v) = u
                v.color = GRAY
                push(Q, v)
            }
        }
        u.color = BLACK # após processar todo vizinho, finaliza
    }
}
```

O algoritmo BFS recebe um grafo não ponderado G e retorna uma árvore T , conhecida como BFS-tree. Essa árvore possui uma propriedade muito especial: ela armazena os menores caminhos da raiz s a todos os demais vértices de T (menor caminho de s a v , $\forall v \in V$). O exemplo a seguir ilustra o trace completo do algoritmo BFS partindo do vértice s .



Trace do algoritmo BFS

i	$u = \text{pop}(Q)$	$V' = \{v \in N(u) \mid v.\text{color} = \text{WHITE}\}$	$\lambda(v)$	$\pi(v)$
0	s	{a, f}	$\lambda(a) = \lambda(s) + 1 = 1$ $\lambda(f) = \lambda(s) + 1 = 1$	$\pi(a) = s$ $\pi(f) = s$
1	a	{b, d, e}	$\lambda(b) = \lambda(a) + 1 = 2$ $\lambda(d) = \lambda(a) + 1 = 2$ $\lambda(e) = \lambda(a) + 1 = 2$	$\pi(b) = a$ $\pi(d) = a$ $\pi(e) = a$
2	f	\emptyset	---	---
3	b	{c, t}	$\lambda(c) = \lambda(b) + 1 = 3$ $\lambda(t) = \lambda(b) + 1 = 3$	$\pi(c) = b$ $\pi(t) = b$
4	d	\emptyset	---	---
5	e	\emptyset	---	---
6	c	\emptyset	---	---
7	t	\emptyset	---	---

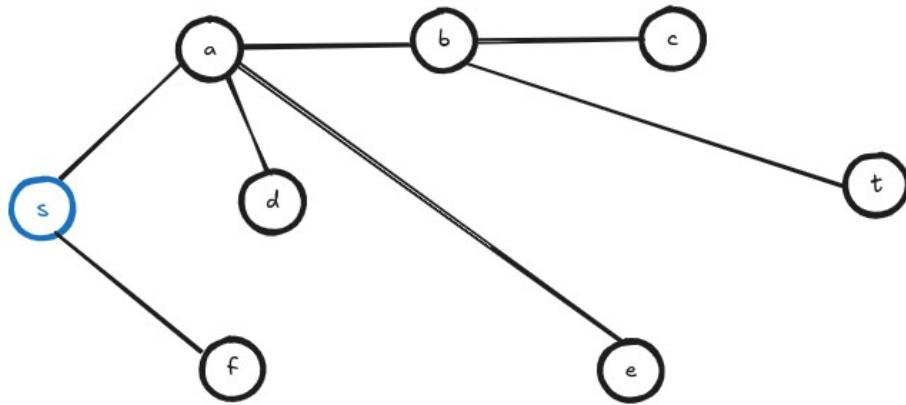
FILA

$$\begin{aligned}
 Q^{(0)} &= [s] \\
 Q^{(1)} &= [a, f] \\
 Q^{(2)} &= [f, b, d, e] \\
 Q^{(3)} &= [b, d, e] \\
 Q^{(4)} &= [d, e, c, t] \\
 Q^{(5)} &= [e, c, t] \\
 Q^{(6)} &= [c, t] \\
 Q^{(7)} &= [t] \\
 Q^{(8)} &= \emptyset
 \end{aligned}$$

A complexidade da busca em largura é $O(n + m)$, onde n é o número de vértices e m é o número de arestas. É fácil perceber que cada vértice e aresta serão acessados exatamente uma vez.

Árvore BFS

v	s	a	b	c	d	e	f	t
$\pi(v)$	---	s	a	b	a	a	s	b
$\lambda(v)$	0	1	2	3	2	2	1	3

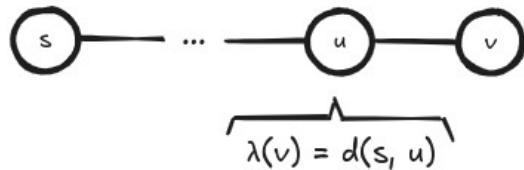


Note que a árvore nada mais é que a união dos caminhos mínimos de s (origem) a qualquer um dos vértices do grafo (destinos). A BFS-tree geralmente não é única, porém todas possuem a mesma profundidade (mínima distância da raiz ao mais distante)

Teorema: A BFS sempre termina com $\lambda(v)=d(s,v)$ para $\forall v \in V$, onde $d(s,v)$ é a distância geodésica (menor distância entre s e v).

Prova por contradição:

1. Sabemos que na BFS $\lambda(v) \geq d(s,v)$
2. Suponha que $\exists v \in V$ tal que $\lambda(v) > d(s,v)$, onde v é o primeiro vértice que isso ocorre ao sair da fila Q
3. Então, existe caminho P_{sv} pois senão $\lambda(v) = d(s,v) = \infty$ (contradiz 2)
4. Se existe P_{sv} então existe um caminho mínimo P_{sv}^*
5. Considere $u \in V$ como predecessor de v em P_{sv}^*
6. Então, $d(s,v) = d(s,u) + 1$ (pois u é predecessor de v)
7. Assim, temos



pois v foi o 1º a sair de Q com $\lambda(v) \neq d(s, v)$

$$\begin{array}{c} \lambda(v) > d(s, v) = d(s, u) + 1 = \lambda(u) + 1 \\ (2) \quad \quad \quad (6) \quad \quad \quad (5) \end{array}$$

e portanto $\lambda(v) > \lambda(u) + 1$ (*), o que é uma contradição pois só existem 3 possibilidades quando u sai da fila Q , ou seja, $u = \text{pop}(Q)$

i) v é WHITE: $\lambda(v) = \lambda(u) + 1$ (contradição)

ii) v é BLACK: se isso ocorre significa que v sai da fila Q antes de u , ou seja, $\lambda(v) < \lambda(u)$ (contradição)

iii) v é GRAY: então v foi descoberto por um w removido de Q antes de u , ou seja, $\lambda(w) \leq \lambda(u)$. Além disso, $\lambda(v) = \lambda(w) + 1$. Assim, temos $\lambda(w) + 1 \leq \lambda(u) + 1$, o que finalmente implica em $\lambda(v) \leq \lambda(u) + 1$ (contradição)

Portanto, $\nexists v \in V$ tal que $\lambda(v) > d(s, v)$.

Busca em Profundidade (Depth-First Search - DFS)

Ideia geral: a cada vértice descoberto, explorar um de seus vizinhos não visitados (sempre que possível). Imita a exploração de labirinto, aprofundando sempre que possível.

Definição das variáveis

i) $v.d$: discovery time (tempo de entrada em v)
 $v.f$: finishing time (tempo de saída de v)

ii) v . color : status do vértice v (existem 3 possíveis valores)
a) WHITE: vértice v ainda não descoberto
b) GRAY: vértice já descoberto
c) BLACK: vértice finalizado

iii) $\pi(v)$: predecessor de v (onde estava quando descobri v)

iv) Q : Pilha (LIFO)

Porém, para simular a pilha de execução, pode-se utilizar um recurso computacional: recursão!

Assim, não é necessário implementar de fato essa estrutura de dados (vantagem)

Porém, em casos extremos (tamanho muito grande), recursão pode gerar problemas (overflow).

ALGORITMO (versão recursiva)

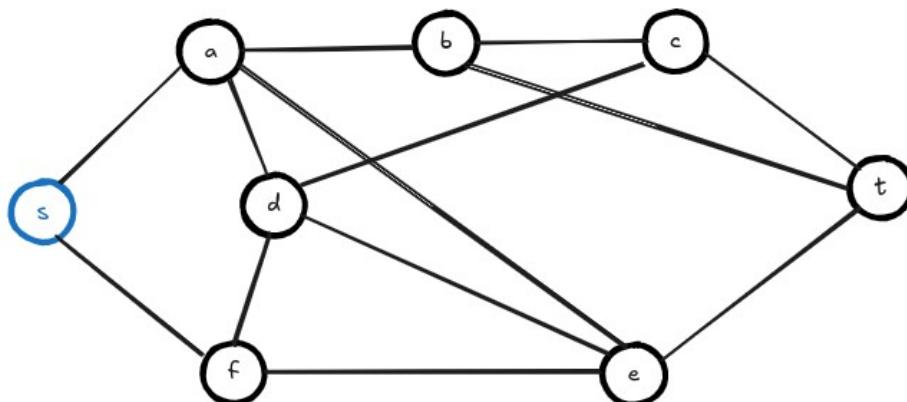
```
DFS(G, s) {
    for each u ∈ V {      # inicializa as variáveis
        u.color = WHITE
        π(v) = NIL
    }
    time = 0                # variável global para contar o tempo
    for each u ∈ V {
        if u.color == WHITE
            DFS_visit(G, u)
    }
}

# Função recursiva chamada sempre que um vértice é descoberto
DFS_visit(G, u) {
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v ∈ N(u) {
        if v.color == WHITE {
            π(v) = u
            DFS_visit(G, v)      # chamada recursiva
        }
    }
    time = time + 1
    u.f = time
    u.color = BLACK
}
```

Note que para implementar a versão iterativa (não recursiva) da Busca em Profundidade (DFS), basta usar o mesmo algoritmo da Busca em Largura (BFS) trocando a estrutura de dados fila por uma pilha.

Da mesma forma que o algoritmo BFS, esse método recebe um grafo G não ponderado e retorna uma árvore, a DFS_tree.

O exemplo a seguir ilustra o trace completo do algoritmo DFS.



u	u.color	u.d	$V' = \{v \in N(u) / v.\text{color} = \text{WHITE}\}$	$\pi(v)$	u.f
s	G	1	{a, f}	--	16
a	G	2	{b, d, e}	s	15
b	G	3	{c, t}	a	14
c	G	4	{d, t}	b	13
d	G	5	{e, f}	c	12
e	G	6	{f, t}	d	11
f	G	7	\emptyset	e	8
t	G	9	\emptyset	e	10

Diferenças entre BFS e DFS

BFS

- possui aspecto espacial
- encontrar caminhos mínimos
- estrutura de dados fila

x

DFS

- possui aspecto temporal
- vértices de corte, ordenação topológica
- estrutura de dados pilha

Propriedades da árvore da busca em profundidade (DFS_tree)

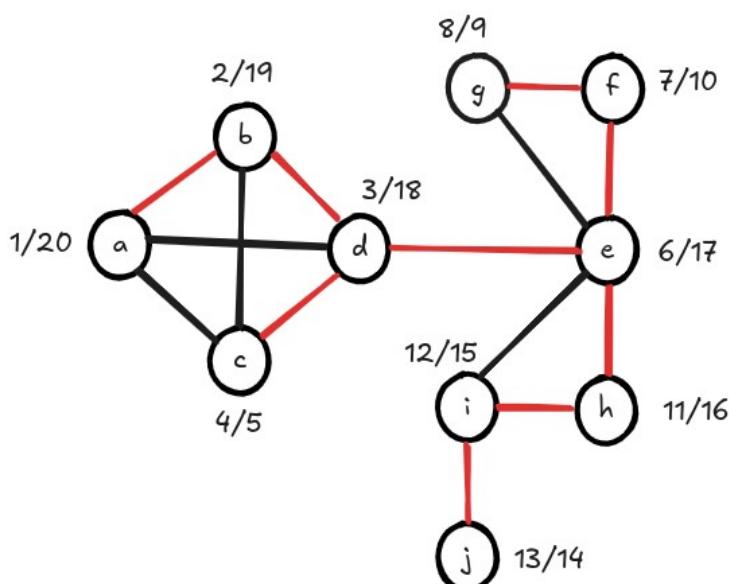
1. A rotulação tem o seguinte significado:

- a) $[u.d, u.f] \subset [v.d, v.f]$: u é descendente de v
- b) $[v.d, v.f] \subset [u.d, u.f]$: v é descendente de u
- c) $[v.d, v.f]$ e $[u.d, u.f]$ são disjuntos: estão em ramos distintos da árvore

2. Após a DFS, podemos classificar as arestas de G como:

- a) t_edges: $e \in T$
- b) b_edges (backward edges): $e \notin T$ (permitem voltar a um ancestral)

Def: Um vértice v é um vértice de corte em G, se e somente se v possui um filho s tal que \nexists b_edge ligando s ou qualquer descendente de s a um ancestral de v.



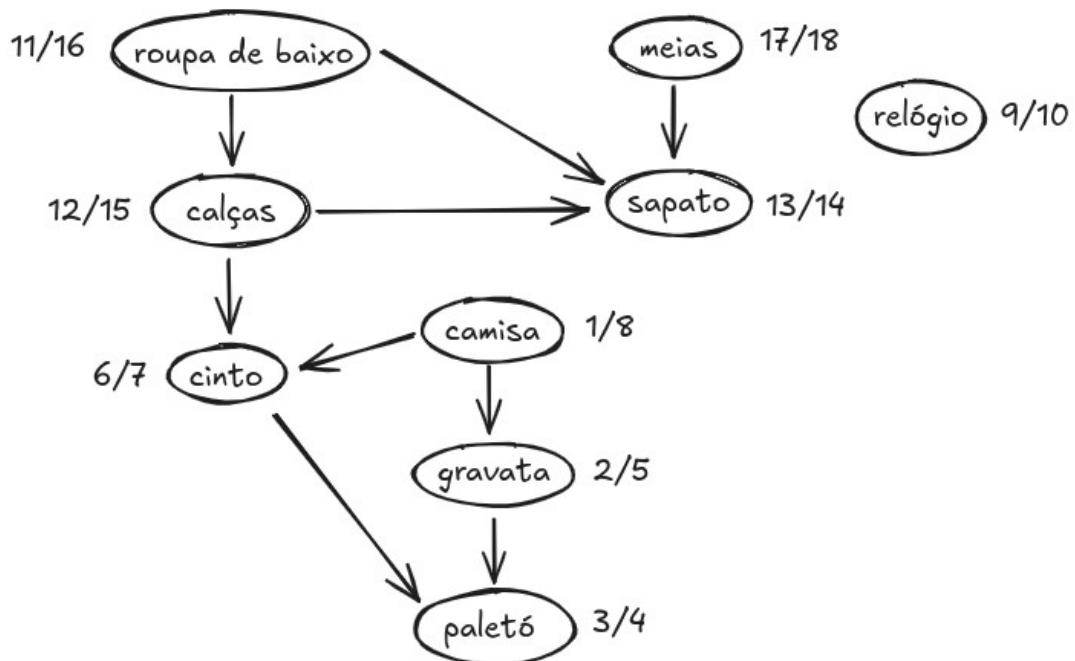
Perguntas:

- a) b é vértice de corte? Não pois b_edge (a, d) liga um sucessor a um antecessor
- b) d é vértice de corte? Sim, pois não há b_edge entre sucessor e antecessor
- c) e é vértice de corte? Sim, pois não há b_edge

Ordenação topológica

Uma ordenação topológica de um DAG (Directed Acyclic Graph) $G = (V, E)$ é uma ordenação linear de todos os seus vértices de modo que se G contém uma aresta (u, v) , então u aparece antes de v na ordenação. Podemos pensar na ordenação topológica de G como uma ordenação de seus vértices ao longo de uma reta horizontal de modo que todas as arestas apontam da esquerda para a direita.

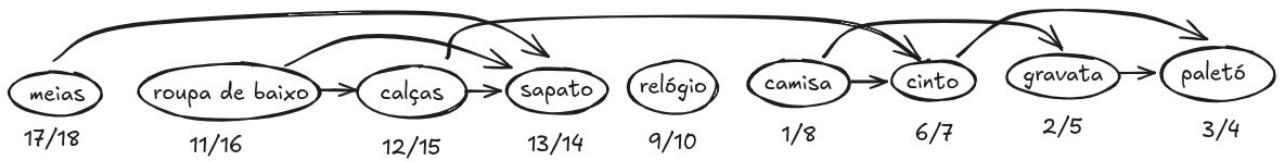
Uma aplicação da ordenação topológica seria a seguinte: imagine que você queira “ensinar” um robô humanoide a se vestir. Claramente, existe uma ordem natural que precisa ser seguida durante o processo. Por exemplo, não podemos calçar o sapato e depois colocar as meias. Suponha que o grafo acíclico direcionado que representa as dependências seja dado conforme a figura a seguir.



A rotulação obtida pela Busca em Profundidade mostra os tempos de entrada e saída para cada um dos vértices. O algoritmo Topological_Sort mostra a sequência lógica de passos necessárias para ordenar um DAG arbitrário.

```
Topological_Sort(G) {
    Execute DFS(G) para calcular v.f para todo v ∈ V
    Conforme cada vértice é finalizado, insira v.f no início de
    uma lista encadeada L
    return L
}
```

A complexidade da ordenação topológica é a mesma da Busca em Profundidade, ou seja, $O(n + m)$, onde n é o número de vértices e m é o número de arestas. O resultado da ordenação topológica do DAG anterior é ilustrada na figura a seguir.



Lema: Um grafo direcionado $G = (V, E)$ é acíclico se e somente se uma busca em profundidade em G não gera b_edges.

$$(\text{ida}) \quad p \rightarrow q \equiv \neg q \rightarrow \neg p$$

G acíclico \rightarrow DFS não gera b_edge \equiv DFS gera b_edge $\rightarrow G$ possui ciclo

1. Suponha que o algoritmo DFS gere uma b_edge (u, v) .

2. Então, por definição, ela liga u a um ancestral v .

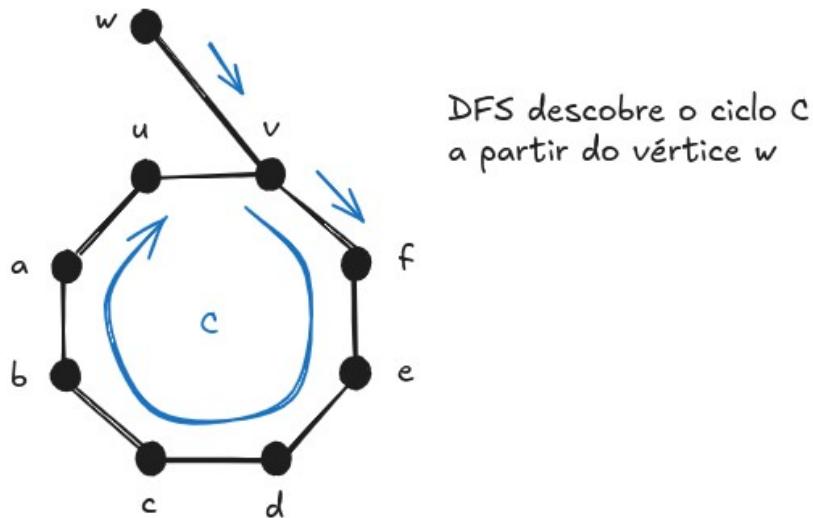
3. Mas, na DFS_tree deve existir um caminho de v até u (pois é uma árvore), o que implica na geração de um ciclo.

$$(\text{volta}) \quad q \rightarrow p = \neg p \rightarrow \neg q$$

G contém ciclo \rightarrow DFS gera b_edge

1. Suponha que G contenha um ciclo C .

2. Seja v o primeiro vértice a ser descoberto em C e seja (u, v) a aresta que precede v em C .



3. Então, no tempo v.d, os vértices do ciclo C formam um caminho de vértices WHITE de v até u .

4. Portanto, pelas propriedades do algoritmo DFS, u será um descendente de v na DFS-tree, o que implica que (u, v) é uma b_edge (pois não irá pertencer a árvore T).

A seguir veremos um resultado que garante a corretude do algoritmo Topological_Sort.

Teorema: O algoritmo Topological_Sort realiza a ordenação topológica de um grafo direcionado acíclico G .

1. Suponha que o algoritmo DFS seja executado no DAG G para determinar $v.f$, $\forall v \in V$.

2. Basta mostrar que para qualquer par de vértices $u, v \in V$, se existe uma aresta de u para v , então $v.f < u.f$

3. Considere uma aresta arbitrária (u, v) explorada pelo algoritmo DFS.

4. Note que quando essa aresta é explorada, v não pode ser GRAY, senão v seria um ancestral de u e a aresta (u, v) seria uma b_edge, gerando uma contradição ao lema anterior. Portanto, v deve ser WHITE ou BLACK.

5. Se v é WHITE, ele se torna um descendente de u e portanto temos $v.f < u.f$.

6. Se v é BLACK, ele já foi finalizado, de modo que o valor de $v.f$ já foi computado. Como a busca ainda está explorando u , o valor de $u.f$ ainda não foi calculado e será definido em um momento posterior. Logo, $v.f < u.f$, e a prova está concluída.

OBS: Uma página web muito interessante sobre a visualização de algoritmos em grafos pode ser acessada em <https://workshape.github.io/visual-graph-algorithms/>

The real voyage of discovery consists not in seeking new landscapes, but in having new eyes."
-- Marcel Proust

Grafos: Caminhos Mínimos e o Algoritmo de Dijkstra

Encontrar caminhos mínimos em grafos é um dos mais importantes problemas da computação, em grande parte por ser utilizado em aplicações nas mais diversas áreas da ciência. Vimos que a busca em largura é capaz de encontrar caminhos mínimos em grafos não ponderados. Veremos qui como resolver o problema no caso de grafos ponderados

Def: Caminho ótimo

Seja $G = (V, E, w)$ com $w: E \rightarrow R^+$ uma função de custo para as arestas. Um caminho P^* de v_0 a v_n é ótimo se seu peso é o menor possível:

$$w(P^*) = \sum_{i=0}^{n-1} w(v_i, v_{i+1}) = w(v_0, v_1) + w(v_1, v_2) + \dots + w(v_{n-1}, v_n) \quad (\text{soma dos pesos das arestas})$$

Antes de introduzirmos os algoritmos, iremos apresentar uma primitiva comum a todos eles. Trata-se da função relax, que aplica a operação conhecida como relaxamento a uma aresta de um grafo ponderado.

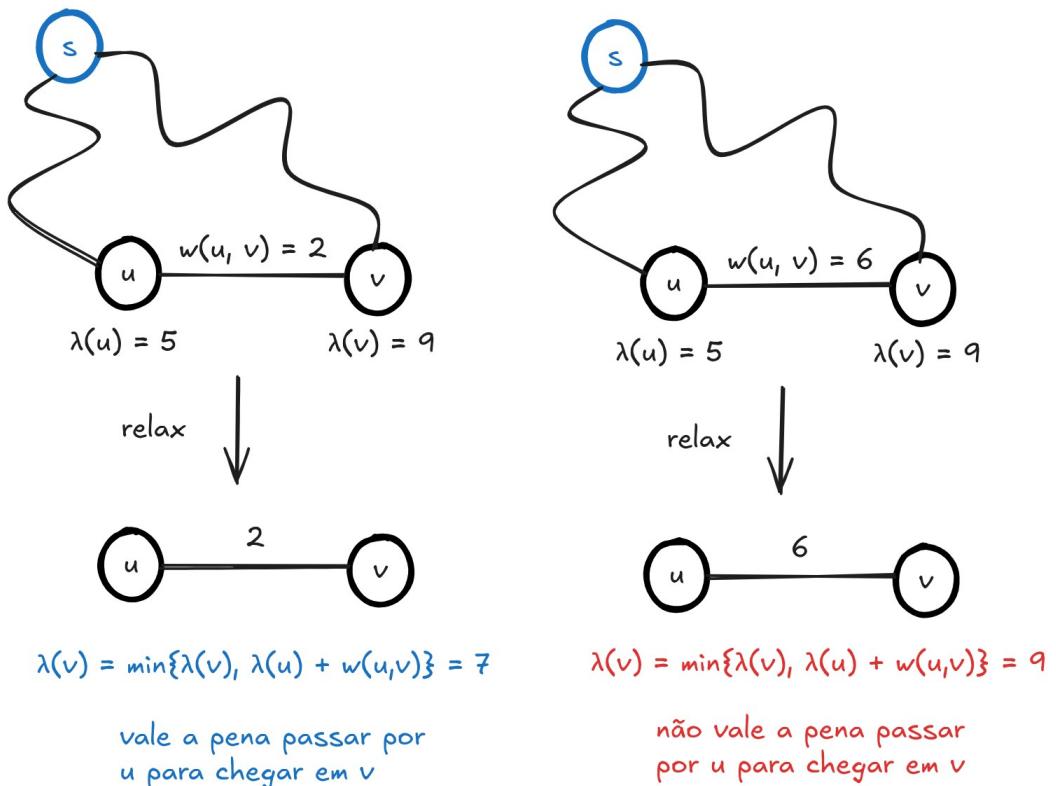
Primitiva relax

$\text{relax}(u, v, w)$: relaxar a aresta (u, v) de peso w

Para entender o que significa relaxar a aresta (u, v) de peso w , precisamos definir $\lambda(u)$ e $\lambda(v)$

Quem é $\lambda(u)$? É o custo atual de sair da origem s e chegar até u
 Quem é $\lambda(v)$? É o custo atual de sair da origem s e chegar até v

Ideia geral: é uma boa ideia passar por u para chegar em v sabendo que o custo de ir de u até v é w ?



Note que a operação $\text{relax}(u, v, w)$ nunca aumenta o valor de $\lambda(v)$, apenas diminui!

ALGORITMO

```

relax(u, v, w)
{
    if  $\lambda(v) > \lambda(u) + w(u,v)$ 
    {
         $\lambda(v) = \lambda(u) + w(u,v)$ 
         $\pi(v) = u$ 
    }
}
relax(u, v, w)
{
     $\lambda(v) = \min\{\lambda(v), \lambda(u) + w(u,v)\}$ 
    if  $\lambda(v)$  was updated
         $\pi(v) = u$ 
}

```

O que varia nos diversos algoritmos para encontrar caminhos mínimos são os seguintes aspectos:

- Quantas e quais arestas devemos relaxar?
- Quantas vezes devemos relaxar as arestas?
- Em que ordem devemos relaxar as arestas?

A seguir veremos um algoritmo muito mais eficiente para resolver o problema: o algoritmo de Dijkstra. Basicamente, esse algoritmo faz uso de uma política de gerenciamento de vértices baseada em aspectos de programação dinâmica. O que o método faz é basicamente criar uma fila de prioridades para organizar os vértices de modo que quanto menor o custo $\lambda(v)$ maior a prioridade do vértice em questão.

Assim, a ideia é expandir primeiramente os menores ramos da árvore de caminhos mínimos, na expectativa de que os caminhos mínimos mais longos usarão como base os subcaminhos obtidos anteriormente. Trata-se de um mecanismo de reaproveitar soluções de subproblemas para a solução do problema como um todo.

Definição das variáveis

$\lambda(v)$: menor custo até o momento para o caminho s-v

$\pi(v)$: predecessor de v na árvore de caminhos mínimos

Q : fila de prioridades dos vértices (maior prioridade = menor $\lambda(v)$)

A fila de prioridades Q possui 3 primitivas básicas:

- a) Insert(Q, v): insere um vértice v no fim da fila Q.
- b) ExtractMin(Q): remove da fila o vértice de maior prioridade.
- c) DecreaseKey(Q, v, $\lambda(v)$): modifica a prioridade do vértice v da fila Q, atualizando o seu valor de $\lambda(v)$ (note que sempre irá diminuir esse valor que é o tamanho do caminho mínimo)

S: conjunto dos vértices já finalizados (vértices para os quais o caminho mínimo já foi obtido).

```

Dijkstra(G, w, s) {
    for each  $v \in V$  {
         $\lambda(v) = \infty$ 
         $\pi(v) = \text{nil}$ 
    }
     $\lambda(s) = 0$ 
    S =  $\emptyset$ 
}

```

```

Q = ∅
for each v ∈ V
    Insert(Q, v)
while Q ≠ ∅ {
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v in N(u) {
        λ(v) = min{λ(v), λ(u) + w(u,v)}
        if λ(v) was updated {
            π(v) = u
            Decrease_Key(Q, v, λ(v))
        }
    }
}

```

Algoritmos em grafos e suas estruturas de dados

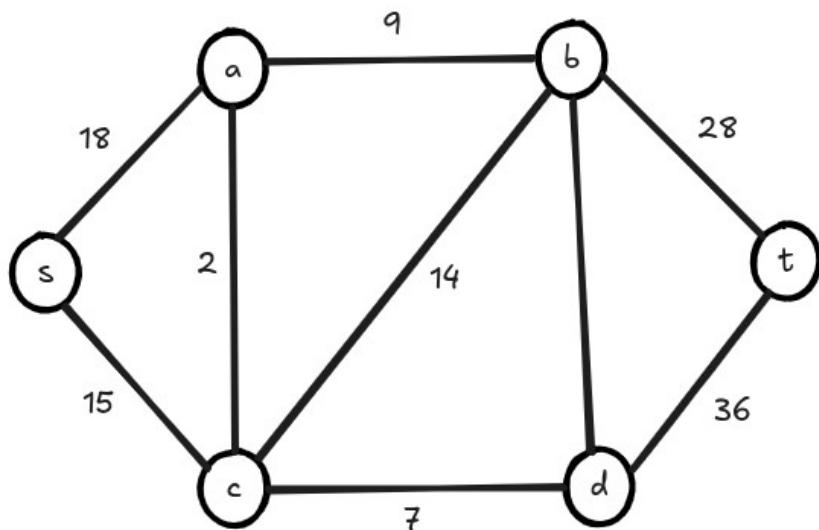
1. Busca em Largura (BFS): Fila **Q**
2. Busca em Profundidade (DFS): Pilha **S**
3. Algoritmo de Dijkstra: Fila de prioridades **Q**

Neste contexto, note que o algoritmo de Dijkstra é uma generalização da busca em largura para o caso em que os pesos das arestas não são todos iguais. Ambos crescem primeiro os menores ramos da árvore.

No algoritmo BFS toda aresta tem mesmo peso, já no algoritmo de Dijkstra esse peso é variável.

Considere o seguinte grafo ponderado. Suponha que deseja-se encontrar os menores caminhos do vértice **s** até todos os demais vértices. Para isso, basta aplicarmos o algoritmo de Dijkstra com raiz no vértice **s**.

A seguir apresentamos um trace completo (passo a passo) desse algoritmo.



Fila de prioridades

	s	a	b	c	d	t	
$\lambda^{(0)}(v)$	0	∞	∞	∞	∞	∞	
$\lambda^{(1)}(v)$		18	∞	15	∞	∞	
$\lambda^{(2)}(v)$		17	29		22	∞	
$\lambda^{(3)}(v)$			26		22	∞	
$\lambda^{(4)}(v)$			26			58	
$\lambda^{(5)}(v)$						54	

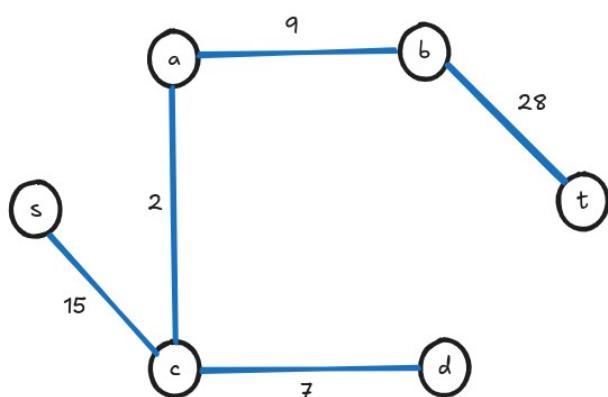
Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
s	{a, c}	$\lambda(a) = \min\{\lambda(a), \lambda(s) + w(s, a)\} = \min\{\infty, 18\} = 18$	$\pi(a) = s$
c	{a, b, d}	$\lambda(c) = \min\{\lambda(c), \lambda(s) + w(s, c)\} = \min\{\infty, 15\} = 15$ $\lambda(a) = \min\{\lambda(a), \lambda(c) + w(c, a)\} = \min\{18, 17\} = 17$ $\lambda(b) = \min\{\lambda(b), \lambda(c) + w(c, b)\} = \min\{\infty, 29\} = 29$ $\lambda(d) = \min\{\lambda(d), \lambda(c) + w(c, d)\} = \min\{\infty, 22\} = 22$	$\pi(c) = s$ $\pi(a) = c$ $\pi(b) = c$ $\pi(d) = c$
a	{b}	$\lambda(b) = \min\{\lambda(b), \lambda(a) + w(a, b)\} = \min\{29, 26\} = 26$	$\pi(b) = a$
d	{b, t}	$\lambda(b) = \min\{\lambda(b), \lambda(d) + w(d, b)\} = \min\{26, 32\} = 26$ $\lambda(t) = \min\{\lambda(t), \lambda(d) + w(d, t)\} = \min\{\infty, 58\} = 58$	$\pi(t) = d$
b	{t}	$\lambda(t) = \min\{\lambda(t), \lambda(b) + w(b, t)\} = \min\{58, 54\} = 54$	$\pi(t) = b$
t	\emptyset	---	---

Mapa de predecessores (árvore final)

v	s	a	b	c	d	t
$\pi(v)$	---	s	a	b	a	b
$\lambda(v)$	0	17	26	15	22	54

Árvore de caminhos mínimos (armazena os menores caminhos de s a todos os demais vértices)



A seguir iremos demonstrar a otimalidade do algoritmo de Dijkstra.

Teorema: O algoritmo de Dijkstra termina com $\lambda(v) = d(s, v), \forall v \in V$

Note que sempre $\lambda(v) \geq d(s, v)$ (*)

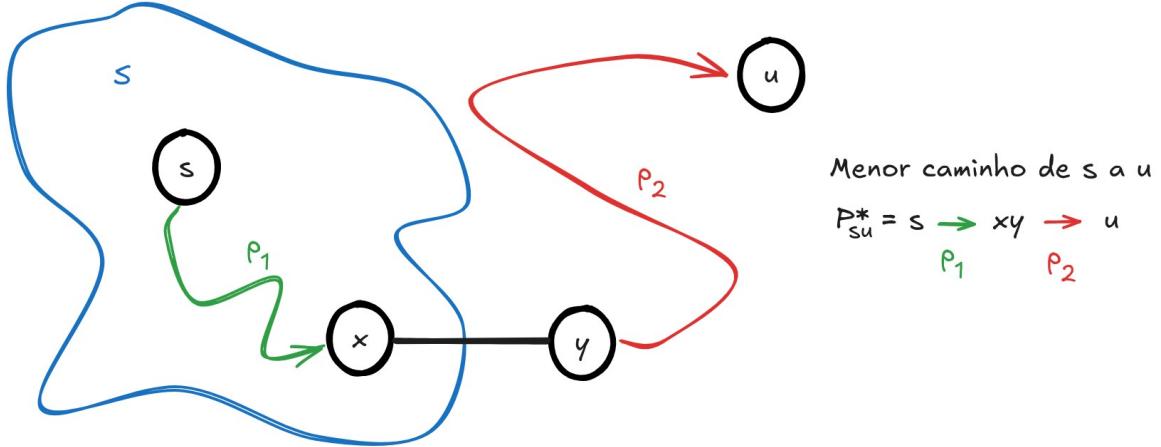
1. Suponha que u seja o 1º vértice para o qual $\lambda(u) \neq d(s, u)$ quando u entra em S .

2. Então, $u \neq s$ pois senão $\lambda(s) = d(s, s) = 0$

3. Assim, existe um caminho P_{su} pois senão $\lambda(u) = d(s, u) = \infty$. Portanto, existe um caminho mínimo P_{su}^*

4. Antes de adicionar u a S , P_{su}^* possui $s \in S$ e $u \in V - S$

5. Seja y o 1º vértice em P_{su}^* tal que $y \in V - S$ e seja x seu predecessor ($x \in S$)



Obs: Note que tanto p_1 quanto p_2 não precisam necessariamente ter arestas

6. Como $x \in S$, $\lambda(x) = d(s, x)$ e no momento em que ele foi inserido a S , a aresta (x, y) foi relaxada, ou seja:

$$\lambda(y) = \lambda(x) + w(x, y) = d(s, x) + w(x, y) = d(s, y)$$

7. Mas y antecede u no caminho e como $w: E \rightarrow R^+$ (pesos positivos), temos:

$$d(s, y) \leq d(s, u)$$

e portanto

$$\begin{aligned} \lambda(y) &= d(s, y) \leq d(s, u) \leq \lambda(u) \\ (6) \quad (7) \quad (*) \end{aligned}$$

8. Mas como ambos y e u pertencem a $V - S$, quando u é escolhido para entrar em S temos $\lambda(u) \leq \lambda(y)$

9. Como $\lambda(y) \leq \lambda(u)$ e $\lambda(u) \leq \lambda(y)$ então temos que $\lambda(u) = \lambda(y)$, o que implica em:

$$\lambda(y) = d(s, y) = d(s, u) = \lambda(u)$$

o que gera uma contradição. Portanto $\nexists u \in V$ tal que $\lambda(u) \neq d(s, u)$ quando u entra em S.

Análise da complexidade

Há duas formas de analisar a complexidade do algoritmo de Dijkstra dependendo das estruturas de dados utilizadas.

Caso 1: Fila de prioridades Q representada por um array simples de n elementos.

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$ (é $O(1)$ para cada vértice)
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = \text{ExtractMin}(Q)$ é $O(n)$ (equivale a encontrar menor elemento do array)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(1)$ (acesso direto)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n) + O(n)*O(n) + (O(1) + O(1)) * (d(v_1) + d(v_2) + \dots + d(v_n))$$

Sabendo que a multiplicação de dois termos lineares resulta em quadrático e que de acordo com o Hankshaking Lema, a soma dos graus de um grafo é igual a duas vezes o número de arestas, temos:

$$T(n) = O(n) + O(n^2) + O(1)*O(m)$$

Como em todo grafo básico simples $m < n^2$, temos finalmente que o algoritmo é $O(n^2)$.

Caso 2: Fila de prioridades Q representada por um heap binário (min-heap)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$, pois todos os pesos são iguais inicialmente (infinitos).
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = \text{ExtractMin}(Q)$ é $O(\log n)$ (dequeue), mas dentro de loop (n vezes)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(\log n)$ (pode ter que subir no min-heap até a raiz – altura da árvore)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n \log n) + (O(1) + O(\log n)) * (d(v_1) + d(v_2) + \dots + d(v_n))$$

De modo similar ao caso anterior, podemos escrever:

$$T(n) = O(n) + O(n \log n) + O(m) + O(m \log n)$$

Como o termo com logaritmo domina os demais: $T(n) = O((n+m) \log n)$

Mas como em grafos conexos $m > n - 1$, chega-se que $T(n)$ é $O(m \log n)$.

Qual das duas implementações é mais eficiente? Depende! Devemos preferir a primeira opção se $m \log n > n^2$, o que equivale a:

$$\frac{m}{n^2} > \frac{1}{\log n} \rightarrow d > \frac{1}{\log n}$$

Em grafos mais densos, o caso 1 é mais eficiente.

Em grafos menos densos, o caso 2 é mais eficiente.

Por exemplo, se $n = 1024$, temos a seguinte regra:

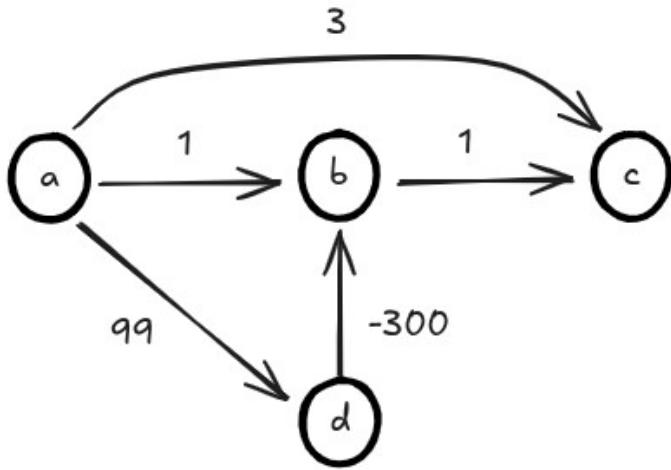
- se $d > 0.1$, opção 1 é melhor
- se $d < 0.1$, opção 2 é melhor

Para que d seja igual a 0.1, um grafo com $n = 1024$ vértices deve ter 104857 arestas.

Lembrando que o algoritmo de Djikstra tem uma limitação: em grafos com pesos negativos, sua convergência não é garantida! Ou seja, ele pode não funcionar corretamente.

Situações em que o algoritmo de Djikstra falha

O algoritmo de Djikstra pode não funcionar corretamente quando o grafo admite pesos negativos em suas arestas. Para entender o porque isso ocorre, iremos apresentar um exemplo ilustrativo. Considere o seguinte grafo ponderado. Desejamos encontrar a árvore de caminhos mínimos com raiz em A.



A seguir encontra-se a execução passo a passo do algoritmo de Djikstra.

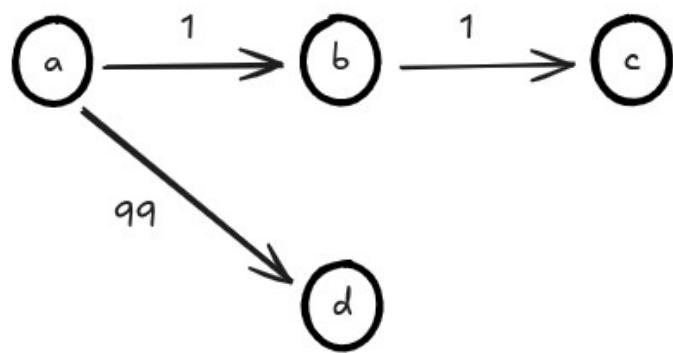
Fila de prioridades

	a	b	c	d
$\lambda^{(0)}(v)$	0	∞	∞	∞
$\lambda^{(1)}(v)$		1	3	99
$\lambda^{(2)}(v)$			1	99
$\lambda^{(3)}(v)$				99

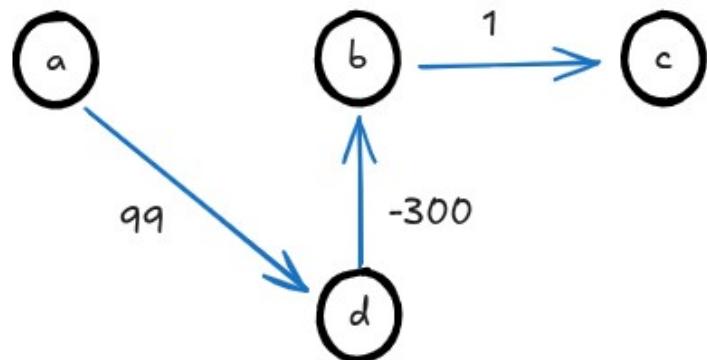
Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
a	{b, c, d}	$\lambda(b) = \min\{\infty, 1\} = 1$	$\pi(b) = a$
		$\lambda(c) = \min\{\infty, 3\} = 3$	$\pi(c) = a$
		$\lambda(d) = \min\{\infty, 99\} = 99$	$\pi(d) = a$
b	{c}	$\lambda(c) = \min\{3, 1+1\} = 2$	$\pi(c) = b$
c	\emptyset	---	---
d	\emptyset	---	---

Note que ao visitar o vértice d, o vértice b já não está mais na fila. Portanto, a árvore de caminhos mínimos retornada pelo Dijkstra é a seguinte:



o que não está correto, pois deveria ser:



A pergunta natural é: como evitar que isso ocorra?

Para isso, devemos utilizar o algoritmo de Bellman-Ford, que apesar de menos eficiente do que o algoritmo de Dijkstra, é o único capaz de encontrar caminhos mínimos em grafos em que as arestas possuem custo negativo.

Algoritmo de Bellman-Ford

Ideia: a cada passo relaxar $\forall e \in E$ em ordem arbitrária, repetindo o processo $|V| - 1$ vezes

```

Bellman_Ford(G, w, s) {
    for each v ∈ V {
        λ(v) = ∞
        π(v) = NIL
    }
    λ(s) = 0
    for i = 1 to |V|-1 {
        for each e = (u, v) in E
            relax(u, v, w)
    }
}

```

Para aplicar o algoritmo Bellman-Ford, o primeiro passo é definir uma ordem para que as arestas sejam relaxadas. Note que essa ordem pode ser arbitrária! Vamos considerar a seguinte ordem:

(a, b); (a, c); (a, d); (b, c); (d, b)

Como temos 4 vértices, o loop principal terá 3 iterações.

1^a iteração: (a, b): $\lambda(b) = \min\{\infty, 1\} = 1$

(a, c): $\lambda(c) = \min\{\infty, 3\} = 3$

(a, d): $\lambda(d) = \min\{\infty, 99\} = 99$

(b, c): $\lambda(c) = \min\{3, 1+1\} = 2$

(d, b): $\lambda(b) = \min\{1, 99-300\} = -201$

2^a iteração: (a, b): $\lambda(b) = \min\{1, 1\} = 1$

(a, c): $\lambda(c) = \min\{2, -201+1\} = -200$

(a, d): $\lambda(d) = \min\{99, 99\} = 99$

(b, c): $\lambda(c) = \min\{-200, -201+1\} = -200$

(d, b): $\lambda(d) = \min\{-201, 99-300\} = -201$

3^a iteração: (a, b): $\lambda(b) = \min\{1, 1\} = 1$

(a, c): $\lambda(c) = \min\{2, -201+1\} = -200$

(a, d): $\lambda(d) = \min\{99, 99\} = 99$

(b, c): $\lambda(c) = \min\{-200, -201+1\} = -200$

(d, b): $\lambda(d) = \min\{-201, 99-300\} = -201$

o que resulta na árvore desejada.

Dijkstra multisource

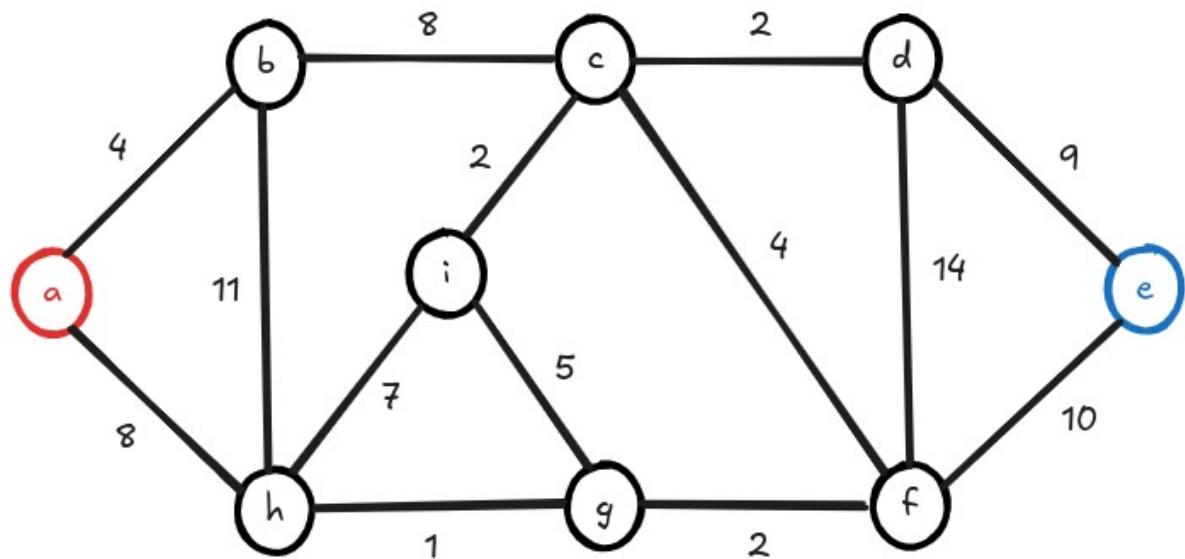
Ideia: utilizar múltiplas sementes/raízes no algoritmo de Djikstra (crescer mais de uma árvore).

Cria um processo de competição: cada vértice pode ser conquistado por apenas uma das sementes (pois ao fim, um vértice só pode estar pendurado em uma única árvore)

Durante a execução do algoritmo, nesse processo de conquista, uma semente pode “roubar” um nó de seus concorrentes, oferecendo a ele um caminho menor que o atual

Ao final temos o que se chama de floresta de caminhos ótimos, composta por várias árvores (uma para cada semente)

Cada árvore representa um agrupamento/comunidade.



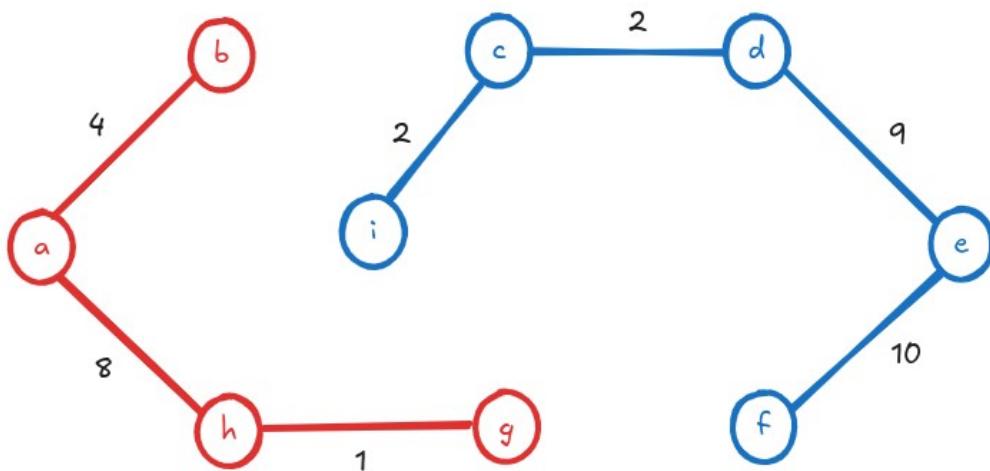
Desejamos encontrar 2 agrupamentos. Para isso, utilizaremos 2 sementes: os vértices A e E. Na prática, isso significa inicializar o algoritmo de Dijkstra com $\lambda(a)=\lambda(e)=0$

Fila

Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
a	{b, h}	$\lambda(b) = \min\{\infty, 4\} = 4$	$\pi(b) = a$
		$\lambda(h) = \min\{\infty, 8\} = 8$	$\pi(h) = a$
e	{d, f}	$\lambda(d) = \min\{\infty, 9\} = 9$	$\pi(d) = e$
		$\lambda(f) = \min\{\infty, 10\} = 10$	$\pi(f) = e$
b	{c, h}	$\lambda(c) = \min\{\infty, 4+8\} = 12$	$\pi(c) = b$
		$\lambda(h) = \min\{8, 4+11\} = 8$	-----
h	{i, g}	$\lambda(i) = \min\{\infty, 8+7\} = 15$	$\pi(i) = h$
		$\lambda(g) = \min\{\infty, 8+1\} = 9$	$\pi(g) = h$
d	{c, f}	$\lambda(c) = \min\{12, 9+2\} = 11$	$\pi(c) = d$
		$\lambda(f) = \min\{10, 9+14\} = 10$	-----
g	{f, i}	$\lambda(f) = \min\{10, 9+14\} = 10$	-----
		$\lambda(i) = \min\{15, 9+5\} = 14$	$\pi(i) = g$
f	{c}	$\lambda(c) = \min\{11, 10+4\} = 11$	-----
c	{i}	$\lambda(i) = \min\{14, 11+2\} = 13$	$\pi(i) = c$
i	\emptyset	-----	-----

Floresta de caminhos ótimos



A heurística A*

É uma técnica aplicada para acelerar a busca por caminhos mínimos em certos tipos de grafos. Pode ser considerado uma generalização do algoritmo de Dijkstra. Um dos problemas com o algoritmo de Dijkstra é não levar em consideração nenhuma informação sobre o destino. Em grafos densamente conectados esse problema é amplificado devido ao alto número de arestas e aos muitos caminhos a serem explorados. Em suma, o algoritmo A* propõe uma heurística para dizer o quanto estamos chegando próximos do destino através da modificação da prioridades dos vértices na fila Q. É um algoritmo muito utilizado na IA de jogos eletrônicos.

Ideia: modificar a função que define a prioridade dos vértices

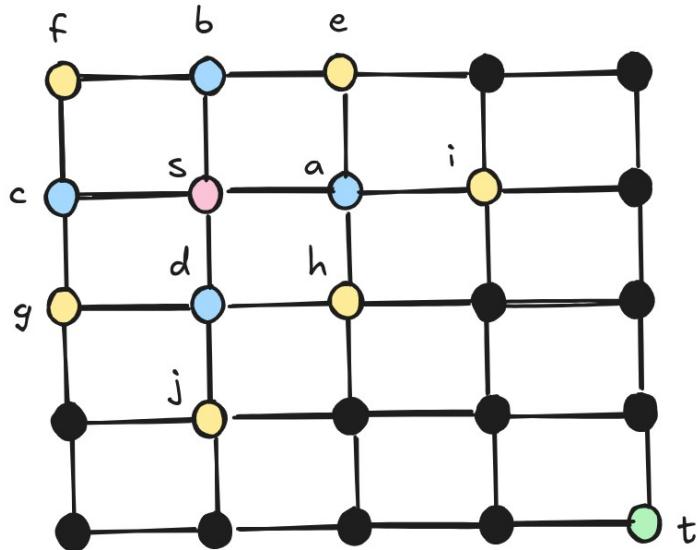
$$\alpha(v) = \lambda(v) + \gamma(v) \quad \text{onde}$$

$\lambda(v)$: custo atual de ir da origem s até v

$\gamma(v)$: custo estimado de v até o destino t (alvo)

Desafio: como calcular $\gamma(v)$?

- É viável apenas alguns casos específicos



Considere o grafo acima. O vértice s é a origem e o vértice t é o destino. Neste caso temos:

$$\lambda(a) = \lambda(b) = \lambda(c) = \lambda(d) = 1$$

o que significa que no Dijkstra, todos eles teriam a mesma prioridade. Note porém que, utilizando a distância Euclidiana para obter uma estimativa de distância até a origem, temos:

$$\gamma(a) = \gamma(d) = \sqrt{4+9} = \sqrt{13}$$

$$\gamma(b) = \gamma(c) = \sqrt{9+16} = \sqrt{25} = 5$$

Ou seja, no A*, devemos priorizar a e d em detrimento de b e c uma vez que

$$1 + \sqrt{13} < 1 + 5$$

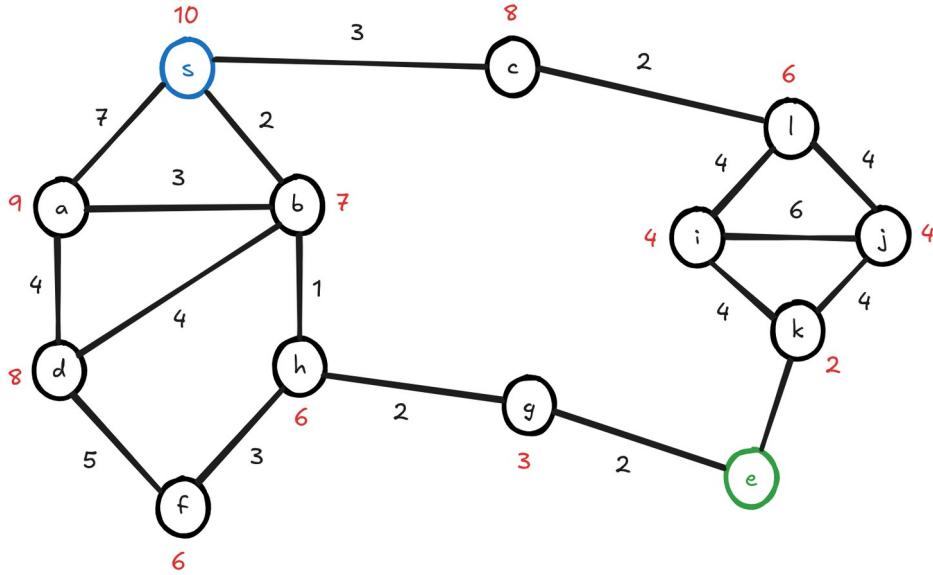
e portanto a e d saem da fila de prioridades antes. Isso ocorre pois no A* eles são considerados mais importantes. O mesmo ocorre nos demais níveis

$$\lambda(e) = \lambda(f) = \lambda(g) = \lambda(h) = \lambda(i) = \lambda(j) = 2$$

$$\gamma(e) = \sqrt{18} \qquad \gamma(j) = \sqrt{10} \qquad \gamma(h) = \sqrt{8}$$

A ideia é que o alvo t atraia o caminho. Se t se move, a busca por caminhos mínimos usando A* costuma ser bem mais eficiente que o Dijkstra em casos como esse.

Considere o seguinte exemplo: o grafo ponderado a seguir ilustra um conjunto de cidades e os pesos das arestas são as distâncias entre elas. Estamos situados na cidade s e deseja-se encontrar um caminho mínimo até a cidade e . Os números em vermelho indicam o valor de $\gamma(v)$, ou seja, são uma estimativa para a distância de v até o destino e . Execute o algoritmo A* para obter o caminho mínimo de s a e .



Fila

	s	a	b	c	d	e	f	g	h	i	j	k	l
$\gamma^{(0)}(v)$	0	∞											
$\gamma^{(1)}(v)$		$7+9$	$2+7$	$3+8$	∞								
$\gamma^{(2)}(v)$		$5+9$		$3+8$	$6+8$	∞	∞	∞	$3+6$	∞	∞	∞	∞
$\gamma^{(3)}(v)$		$5+9$		$3+8$	$6+8$	∞	$6+6$	$5+3$					
$\gamma^{(4)}(v)$		$5+9$		$3+8$	$6+8$	7	$6+6$						

Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
s	$\{a, b, c\}$	$\lambda(a) = \min\{\infty, 7\} = 7$	$\pi(a) = s$
		$\lambda(b) = \min\{\infty, 2\} = 2$	$\pi(b) = s$
		$\lambda(c) = \min\{\infty, 3\} = 3$	$\pi(c) = s$
b	$\{a, d, h\}$	$\lambda(a) = \min\{7, 2+3\} = 5$	$\pi(a) = b$
		$\lambda(d) = \min\{\infty, 2+4\} = 6$	$\pi(d) = b$
		$\lambda(h) = \min\{\infty, 2+1\} = 3$	$\pi(h) = b$
h	$\{f, g\}$	$\lambda(f) = \min\{\infty, 3+3\} = 6$	$\pi(f) = h$
		$\lambda(g) = \min\{\infty, 3+2\} = 5$	$\pi(g) = h$
g	$\{e\}$	$\lambda(e) = \min\{\infty, 5+2\} = 7$	$\pi(e) = g$

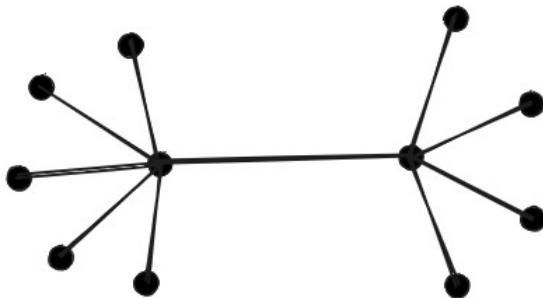
Note como a ordem de retirada dos vértices da fila é orientada ao destino. Há um algoritmo que resolve exclusivamente o problema de calcular as distâncias geodésicas entre todos os pares de vértices do grafo: o algoritmo de Floyd-Warshall, cuja complexidade computacional é $O(n^3)$.

"Solutions are not found by pointing fingers; they are reached by extending hands."
-- Aysha Taryam

Grafos: Árvores geradoras mínimas

Árvores são grafos especiais com diversas propriedades únicas. Devido a essas propriedades são extremamente importantes na resolução de vários tipos de problemas práticos. Veremos ao longo do curso que vários problemas que estudaremos se resumem a: dado um grafo G , extrair uma árvore T a partir de G , de modo que T satisfaça uma certa propriedade (como por exemplo, mínima profundidade, máxima profundidade, mínimo peso, mínimos caminhos, etc).

Def: Um grafo $G = (V, E)$ é uma árvore se G é acíclico e conexo.



Teoremas e propriedades

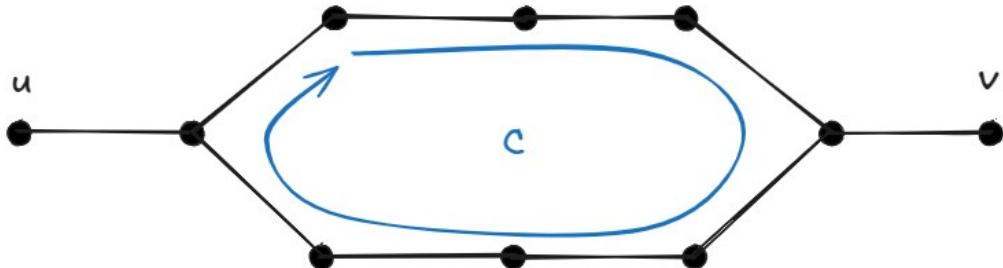
Teorema: G é uma árvore $\Leftrightarrow \exists$ um único caminho entre quaisquer 2 vértices $u, v \in V$

=> **Parte 1:** (ida) $p \rightarrow q = \neg q \rightarrow \neg p$

\nexists um único caminho entre quaisquer $u, v \rightarrow G$ não é uma árvore

a) Pode existir um par u, v tal que \nexists caminho (zero caminhos). Isso implica em G desconexo, o que implica que G não é uma árvore

b) Pode existir um par u, v tal que \exists mais de um caminho.



Porém, neste caso temos a formação de um ciclo e portanto G não pode ser árvore.

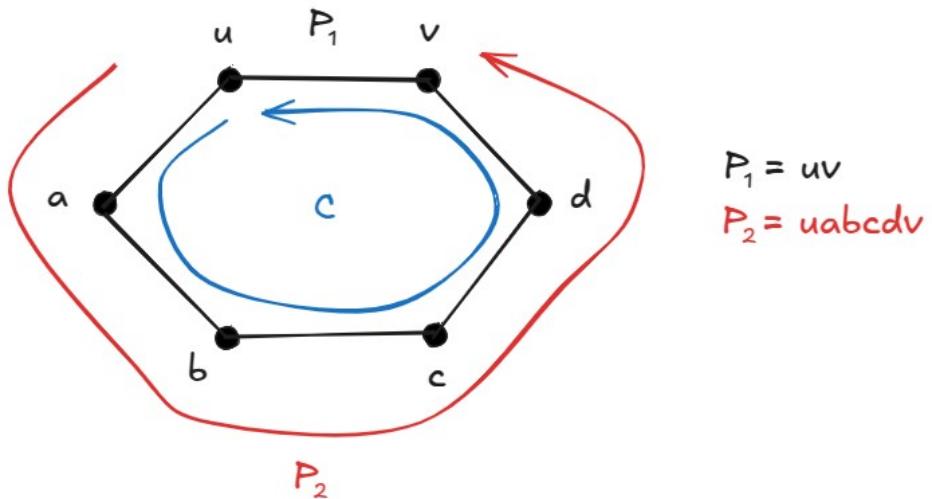
=> **Parte 2:** (volta) $q \rightarrow p = \neg p \rightarrow \neg q$

G não é árvore $\rightarrow \nexists$ único caminho entre quaisquer u, v

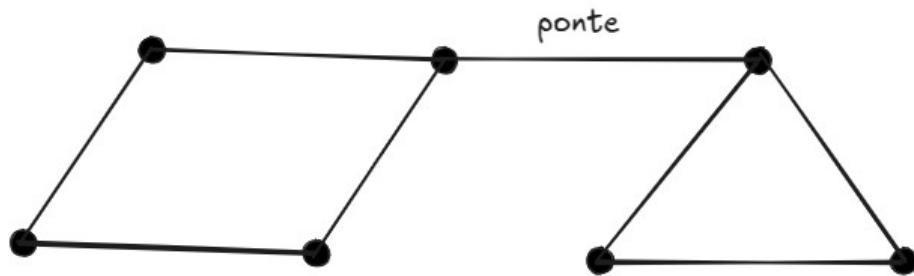
Para G não ser árvore, G deve ser desconexo ou conter um ciclo.

a) Note que no primeiro caso existe um par u, v tal que não há caminho entre eles.

b) Note também que no segundo caso existem 2 caminhos entre u e v , conforme ilustra a figura



Def: Uma aresta $e \in E$ é ponte se $G - e$ é desconexo. Ou seja, a remoção de uma aresta ponte desconecta o grafo

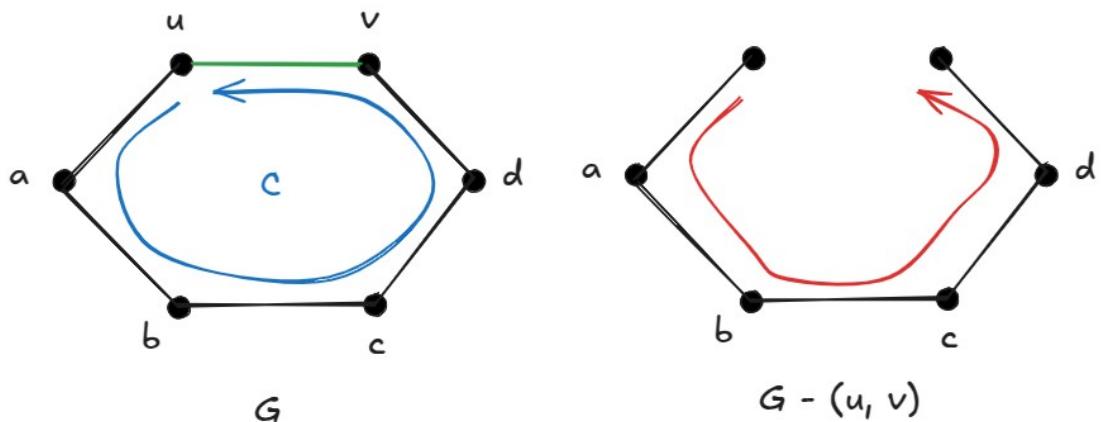


Como identificar arestas ponte?

Teorema: Uma aresta $e \in E$ é ponte \Leftrightarrow aresta não pertence a um ciclo C

=> **Parte 1:** (ida) $p \rightarrow q = \neg q \rightarrow \neg p$

$e \in C \rightarrow$ aresta não é ponte



Como aresta pertence a um ciclo C , há 2 caminhos entre u e v . Logo a remoção da aresta $e = (u,v)$ não impede que o grafo seja conexo, ou seja, $G - e$ ainda é conexo. Portanto, e não é ponte

=> **Parte 2:** (volta) $q \rightarrow p = \neg p \rightarrow \neg q$

aresta não é ponte $\rightarrow e \in C$

Se aresta não é ponte então $G - e$ ainda é conexo. Se isso ocorre, deve-se ao fato de que em $G - e$ ainda existe um caminho entre u e v que não passa por e . Logo, em G existem existem 2 caminhos, o que nos leva a conclusão de que a união entre os 2 caminhos gera um ciclo C .

Teorema: G é uma árvore \Leftrightarrow Toda aresta é ponte

=> **Parte 1:** (ida) $p \rightarrow q = \neg q \rightarrow \neg p$

\exists aresta não ponte $\rightarrow G$ não é árvore

A existência de uma aresta não ponte implica na existência de ciclo. A presença de um ciclo C faz com que G não seja um árvore

=> **Parte 2:** (volta) $q \rightarrow p = \neg p \rightarrow \neg q$

G não é árvore $\rightarrow \exists$ aresta não ponte

Para G não ser uma árvore, deve existir um ciclo em G . Logo, todas as arestas pertencentes ao ciclo não são pontes.

Teorema: Se $G = (V, E)$ é uma árvore com $|V| = n$ então $|E| = n - 1$

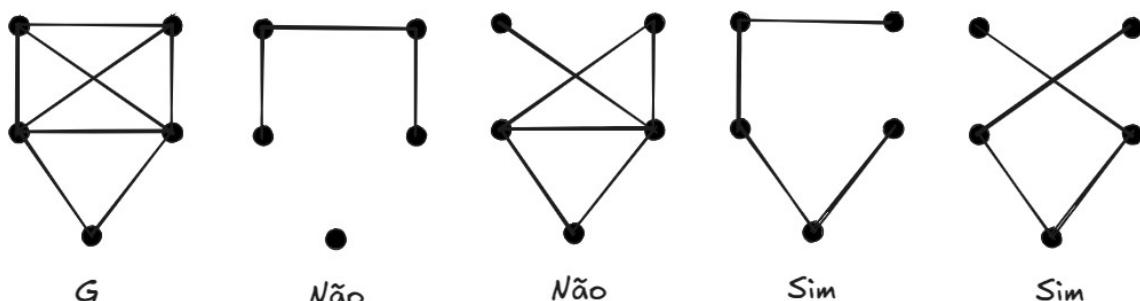
Prova por contradição:

1. Suponha uma árvore de n vértices com $m < n - 1$ arestas.
2. Isso implica em um grafo desconexo. Portanto, não pode ser uma árvore.
3. Suponha uma árvore de n vértices com $m > n - 1$ arestas.
4. Isso implica em um grafo que contém ao menos 1 ciclo C . Portanto, também não pode ser uma árvore.

Logo, toda árvore de n vértices deve ter $m = n - 1$ arestas.

Def: Árvore geradora (spanning tree)

Seja $G = (V, E)$ um grafo. Dizemos que $T = (V, E_T)$ é uma árvore geradora de G se T é um subgrafo de G que é uma árvore (ou seja tem que conectar todos os vértices)



O problema da árvore geradora mínima (MST)

Dentre todas as árvores geradoras de G , obter aquela de menor peso.

Def: Dado $G = (V, E, w)$, onde $w: E \rightarrow R^+$ (peso da aresta e), obter a árvore geradora T que minimiza o peso:

$$w(T) = \sum_{e \in T} w(e) \quad (\text{soma dos pesos das arestas que compõem a árvore})$$

Por exemplo, deseja-se conectar os bairros da cidade com fibra ótica. Qual é a interligação que minimiza o custo?

Estratégia gulosa: abordagem iterativa em que a cada passo devemos escolher a aresta de menor custo que seja segura (uma aresta é segura se, ao ser adicionada em T , T continua sendo uma árvore)

A seguir apresentamos uma função genérica para o problema da árvore geradora mínima.

```
Generic_MST(G, w) {
    T = ∅
    while T não for uma árvore geradora {
        encontre uma aresta segura (u, v)
        T = T ∪ {(u, v)}
    }
    return T
}
```

A pergunta que surge é: como podemos determinar se uma aresta é segura? Veremos a seguir dois algoritmos que utilizam critérios diferentes para definir o que são arestas seguras: os algoritmos de Kruskal e Prim.

O algoritmo de Kruskal

Objetivo: a cada passo escolha a aresta (u, v) de menor custo que não forme um ciclo em $T + (u, v)$

Ideia: iniciar adicionando cada vértice de G como raiz de uma árvore e a cada passo adicionar a aresta de menor custo com extremidades em árvores distintas.

Para isso, iremos utilizar 3 primitivas básicas:

1. Make_Set(v): cria uma árvore contendo um único vértice v (raiz)

```
Make_Set(v) {
    v.p = v           # pai do vértice v é ele mesmo (raiz)
    v.rank = 0         # altura da árvore (nível)
}
```

2. Find-Set(v): retorna qual é a árvore que o vértice v pertence (com path compression)

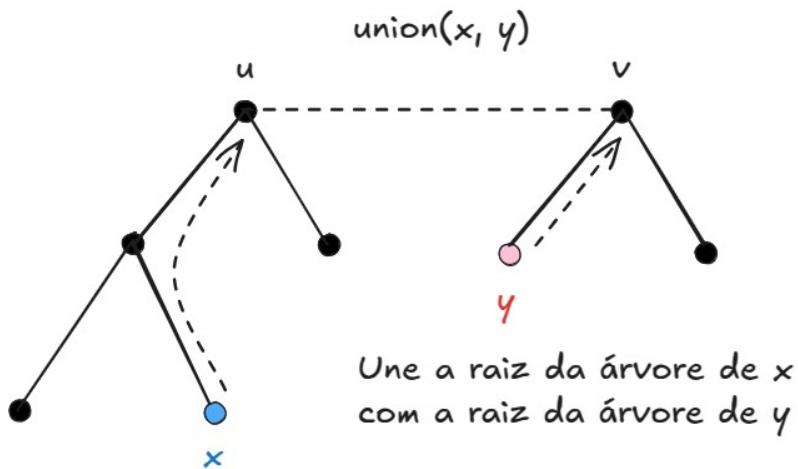
```
Find_Set(v) {
    if v ≠ v.p          # se não é raiz
        v.p = Find_Set(v.p)  # recursão: v se torna o pai
    return v.p           # retorna raiz da árvore de v
}
```

3. Union(u, v): faz a fusão das raízes das árvores de u e de v , criando uma única árvore

```
Union(u, v) {
    Link(Find_Set(u), Find_Set(v))
}

# Função auxiliar para fundir as árvores de u e de v
Link(u, v) {
    if u.rank > v.rank
        v.p = u
    else {
        u.p = v
        if u.rank == v.rank
            v.rank = v.rank + 1
    }
}
```

A figura a seguir ilustra o processo.



Ao realizar $\text{Union}(x, y)$, primeiro encontramos as raízes das árvores de x e de y , que nesse caso são u e v respectivamente. Então, a função auxiliar Link , realiza a fusão dessas duas árvores, criando uma única. Porém, a pergunta é: quem será pai de quem? Note que nesse caso u será pai de v , pois a altura da árvore de raiz u é maior! Assim, não preciso alterar a altura dela. A maior árvore domina a menor. Somente quando as alturas das duas árvores são iguais é que precisamos incrementar a altura (rank) em uma unidade. A seguir apresentamos o algoritmo de Kruskal para a obtenção de uma árvore geradora mínima.

```
MST_Kruskal(G, w) {
    T = ∅
    for each v ∈ V
        Make_Set(v)
    Crie uma lista de todas as arestas e ∈ E
    Ordene a lista pelos pesos w(e) em ordem crescente
    for each e = (u, v) na lista ordenada {
        if Find_Set(u) ≠ Find_Set(v) {
            T = T ∪ {(u, v)}
            Union(u, v)
        }
    }
}
```

Note que trata-se de um algoritmo guloso pois ele sempre tenta incluir a aresta de menor peso a cada iteração. A seguir iremos realizar um trace do algoritmo (simulação passo a passo). Para isso iremos considerar a seguinte notação:

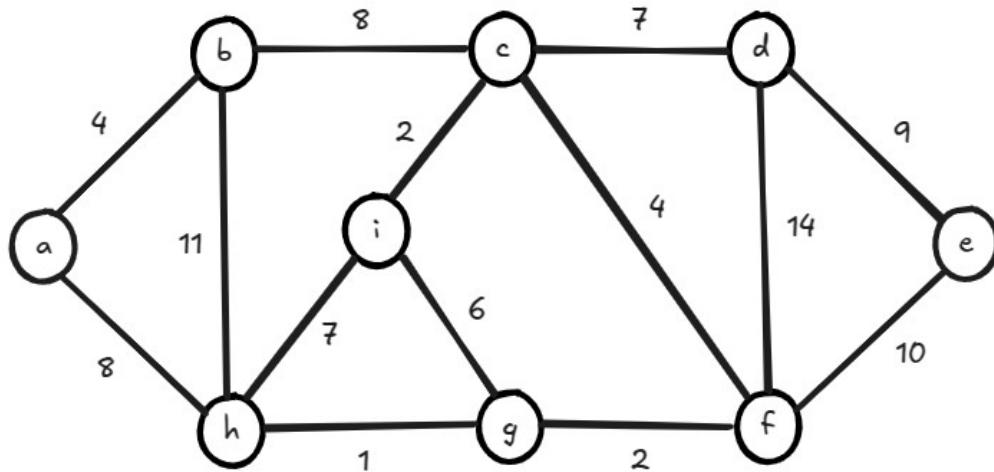
E^- : conjunto das arestas de peso mínimo não seguras ($\text{find_set}(u) = \text{find_set}(v)$)

E^+ : conjunto das arestas de peso mínimo seguras ($\text{find_set}(u) \neq \text{find_set}(v)$)

e_k : aresta escolhida no passo k

Ex: Suponha que os vértices representem bairros e as arestas com pesos os custos de interligação desses bairros com fibra ótica. Deseja saber qual é o custo mínimo de interligar todos os bairros.

Para isso, devemos encontrar a MST do grafo G, o que pode ser feito com o algoritmo de Kruskal.



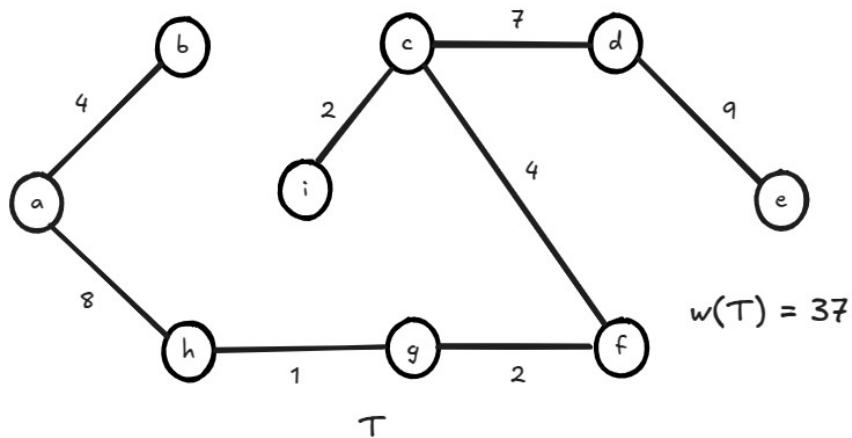
A lista das arestas ordenadas por peso é:

$$E_w = [1, 2, 2, 4, 4, 6, 7, 7, 8, 8, 9, 10, 11, 14]$$

$$E = \{(g, h), (c, i), (f, g), (a, b), (c, f), (g, i), (c, d), (a, h), (b, c), (d, e), (e, f), (b, h), (d, f)\}$$

k	E^-	E^+	e_k
1	-	$\{(g, h)\}$	(g, h)
2	-	$\{(c, i), (f, g)\}$	(c, i)
3	-	$\{(g, f)\}$	(g, f)
4	-	$\{(a, b), (c, f)\}$	(a, b)
5	-	$\{(c, f)\}$	(c, f)
6	$\{(g, i)\}$	-	-
7	$\{(h, i)\}$	$\{(c, d)\}$	(c, d)
8	-	$\{(a, h), (b, c)\}$	(a, h)
9	$\{(b, c)\}$	-	-
10	-	$\{(d, e)\}$	(d, e)

A MST resultante do algoritmo de Kruskal é ilustrada a seguir.



A seguir iremos demonstrar a otimalidade do algoritmo de Kruskal, ou seja, que o algoritmo de Kruskal sempre retorna uma MST de G .

Teorema: Toda árvore T gerado pelo algoritmo de Kruskal é uma MST de G

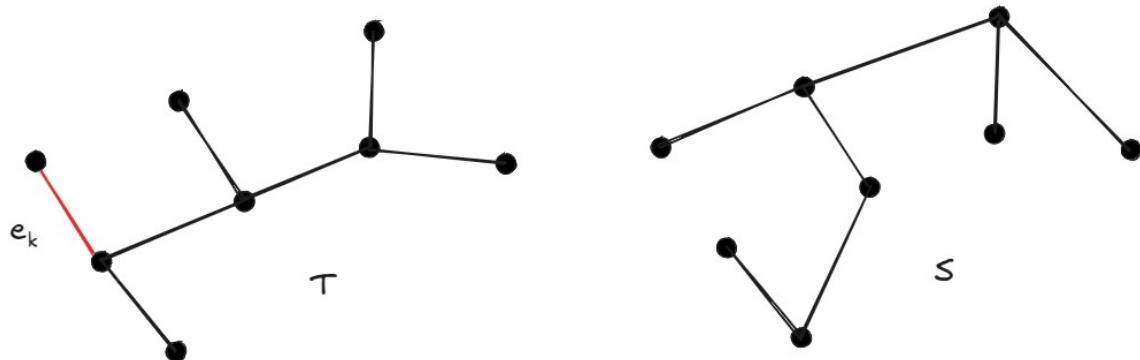
Esse resultado garante que o algoritmo sempre funciona e é ótimo (retorna sempre a solução ótima)

Prova por contradição

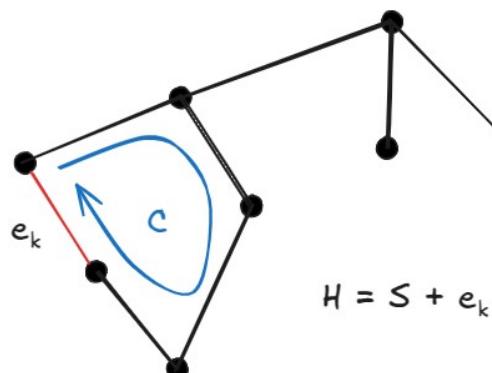
Seja T é a árvore retornada por Kruskal.

1. Suponha que $\exists S \neq T$ tal que $w(S) < w(T)$ (S é uma árvore)

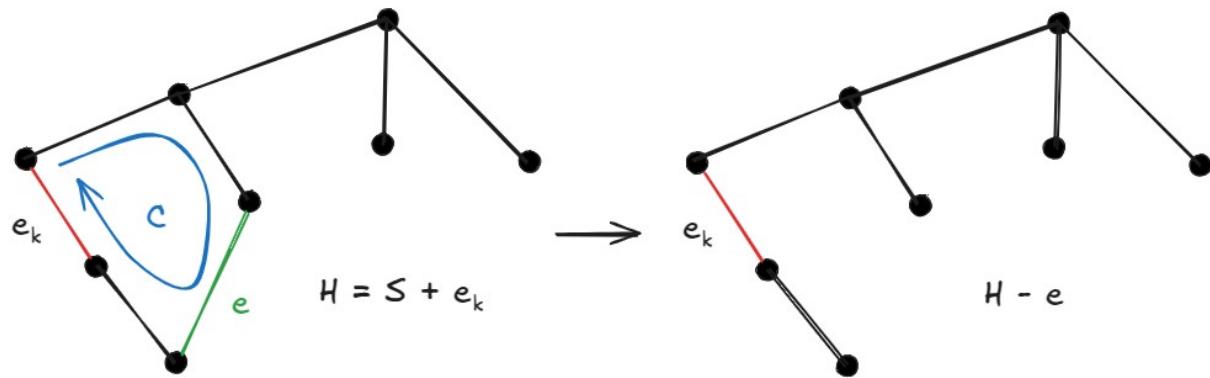
2. Seja $e_k \in T$ a primeira aresta adicionada em T que não está em S (pois árvores são diferentes)



3. Faça $H = (S + e_k)$. Note que H não é mais uma árvore e contém um ciclo



4. Note que no ciclo C , $\exists e \in S$ tal que $e \notin T$ (pois senão C existiria em T). O subgrafo $H - e$ é conexo, possui $n - 1$ arestas e define uma árvore geradora de G .



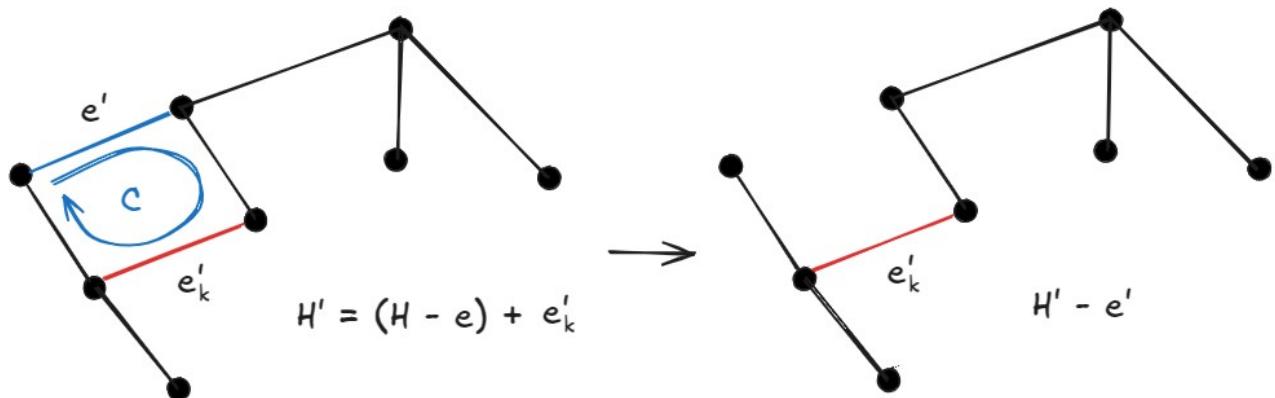
5. Porém, $w(e_k) \leq w(e)$ e assim $w(H - e) \leq w(S)$ (pois de acordo com Kruskal e_k vem antes de e na lista ordenada de arestas, é garantido pela ordenação)

Lista de arestas (após ordenação)



$$e_k \text{ precede } e \rightarrow w(e_k) \leq w(e)$$

6. Repetindo o processo usado para gerar $H - e$ a partir de S é possível produzir uma sequência de árvores que se aproximam cada vez mais de T .



Se continuarmos com esse processo, a árvore inicial S será transformada na árvore T , pois:

$$S \rightarrow (H - e) \rightarrow (H' - e') \rightarrow (H'' - e'') \rightarrow \dots \rightarrow T$$

de modo que

$$w(S) \geq w(H - e) \geq w(H' - e') \geq \dots \geq w(T) \quad (\text{contradição a suposição inicial})$$

Portanto, não existe árvore com peso menor que T , mostrando que T tem peso mínimo. (Não há como S ter peso menor que T).

Análise da complexidade

Primeiramente, devemos analisar as complexidades das primitivas básicas.

Pode-se perceber que a função `Make_Set(v)` é $O(1)$. Como ela é executada n vezes, temos que no total o custo será $O(n)$.

A ordenação das arestas pode ser realizada com o MergeSort ou QuickSort, que são $O(m \log m)$, onde m denota o número de arestas.

A primitiva `Find_Set(v)` deve retornar a raiz da árvore que v pertence. Note que para isso, dependemos da altura da árvore. No caso em que v é um nó folha da árvore binária, $h = \log n$. Portanto, a complexidade da função é $O(\log n)$. Se a árvore não for binária, muda-se apenas a base do logaritmo. Logo, a primitiva `Union(u, v)` que faz uso de `Find_Set(v)` também é $O(\log n)$. Note que `Find_Set(v)` é executada duas vezes para cada aresta (uma para cada extremidade: u e v). Assim, o custo total é $2m O(\log n)$, o que é $O(m \log n)$.

A primitiva `Union(u, v)` é executada somente quando uma aresta é adicionada a árvore. Como uma árvore tem $m = n - 1$ arestas, a complexidade é $O(n \log n)$. Portanto, o custo total do algoritmo de Kruskal é:

$$C = O(n) + O(m \log m) + O(m \log n) + O(n \log n)$$

Podemos observar que o termo dominante é $O(m \log m)$. Como $m < n^2$, temos que:

$$\log m < \log n^2 = 2 \log n = O(\log n)$$

ou seja, a complexidade do algoritmo de Kruskal é $O(m \log n)$.

Algoritmo de Prim

O algoritmo de Prim também utiliza uma abordagem gulosa para a construção da MST.

Ideia: iniciar de uma raiz r e a cada passo adicionar a aresta de menor custo com uma extremidade no conjunto dos vértices visitados e outra extremidade no conjunto dos vértices não visitados.

Definição das variáveis

$\lambda(v)$ ou $v.key$: menor custo de entrada para o vértice v até o momento

$\pi(v)$ ou $v.\pi$: predecessor do vértice v em T

Q : fila de prioridades (maior prioridade = menor $\lambda(v)$)

Veremos a seguir que esse algoritmo é muito similar ao algoritmo de Dijkstra.

```
MST_Prim(G, w, r) {
    for each v ∈ V {
        λ(v) = ∞
        π(v) = nil
    }
    λ(r) = 0
    # Insere vértices na fila de prioridades Q
```

```

Q = ∅
for each v ∈ V
    Insert(Q, v)
while Q ≠ ∅ {
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v in N(u) {
        if v ∈ Q and w(u,v) < λ(v) { # achou entrada melhor
            λ(v) = w(u,v)
            π(v) = u
            Decrease_Key(Q, v, w(u,v))
        }
    }
}

```

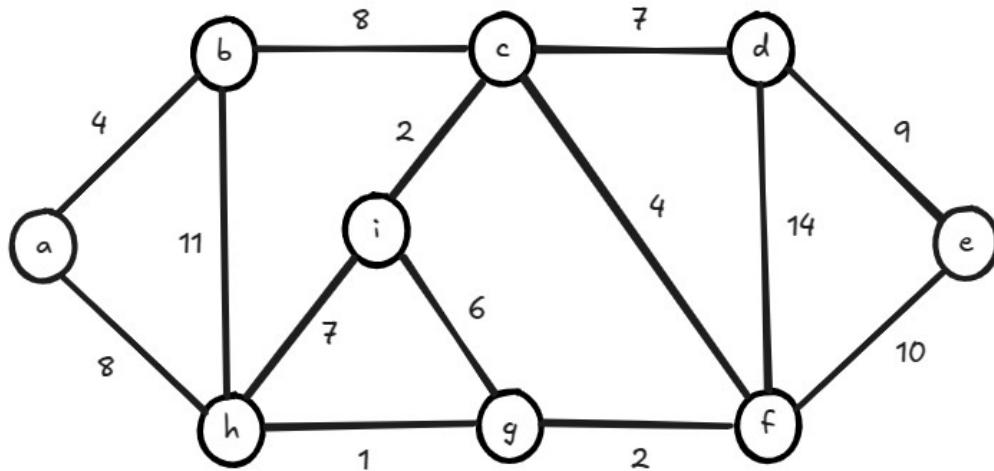
Note que o IF interno no loop FOR é equivalente a:

```

if v ∈ Q {
    λ(v) = min{λ(v), w(u,v)}
    if λ(v) was updated {
        π(v) = u
        Decrease_Key(Q, v, w(u,v))
    }
}

```

A seguir veremos um trace da execução completa do algoritmo de Prim em um simples exemplo.



Fila de prioridades

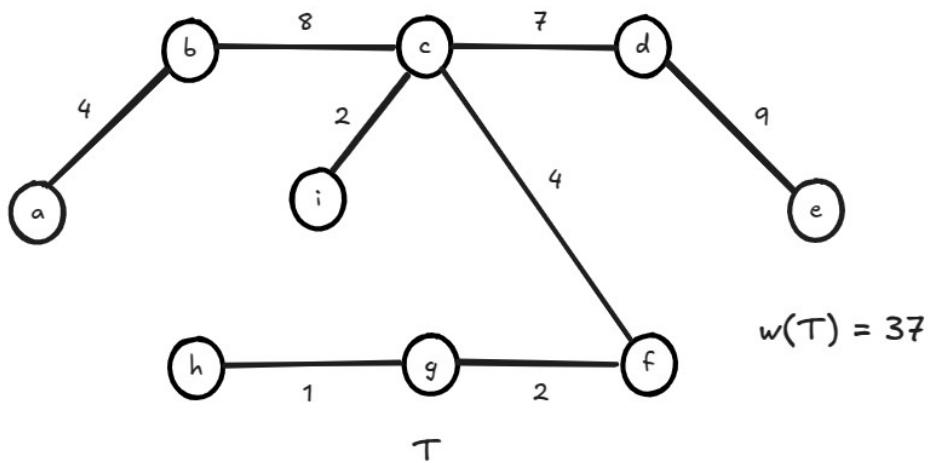
	a	b	c	d	e	f	g	h	i
$\lambda^{(0)}(v)$	0	∞							
$\lambda^{(1)}(v)$		4	∞	∞	∞	∞	∞	8	∞
$\lambda^{(2)}(v)$			8	∞	∞	∞	∞	8	∞
$\lambda^{(3)}(v)$				7	∞	4	∞	8	2
$\lambda^{(4)}(v)$					7	∞	4	6	7

$\lambda^{(5)}(v)$		7	10		2	7	
$\lambda^{(6)}(v)$		7	10			1	
$\lambda^{(7)}(v)$			9				

Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
a	{b, h}	$\lambda(b) = \min\{\lambda(b), w(a, b)\} = \min\{\infty, 4\} = 4$ $\lambda(h) = \min\{\lambda(h), w(a, h)\} = \min\{\infty, 8\} = 8$	$\pi(b) = a$ $\pi(h) = a$
b	{c, h}	$\lambda(c) = \min\{\lambda(c), w(b, c)\} = \min\{\infty, 8\} = 8$ $\lambda(h) = \min\{\lambda(h), w(b, h)\} = \min\{8, 11\} = 8$	$\pi(c) = b$
c	{d, f, i}	$\lambda(d) = \min\{\lambda(d), w(c, d)\} = \min\{\infty, 7\} = 7$ $\lambda(f) = \min\{\lambda(f), w(c, f)\} = \min\{\infty, 4\} = 4$ $\lambda(i) = \min\{\lambda(i), w(c, i)\} = \min\{\infty, 2\} = 2$	$\pi(d) = c$ $\pi(f) = c$ $\pi(i) = c$
i	{h, g}	$\lambda(h) = \min\{\lambda(h), w(i, h)\} = \min\{8, 7\} = 7$ $\lambda(g) = \min\{\lambda(g), w(i, g)\} = \min\{\infty, 6\} = 6$	$\pi(h) = i$ $\pi(g) = i$
f	{d, e, g}	$\lambda(d) = \min\{\lambda(d), w(f, d)\} = \min\{7, 14\} = 7$ $\lambda(e) = \min\{\lambda(e), w(f, e)\} = \min\{\infty, 10\} = 10$ $\lambda(g) = \min\{\lambda(g), w(f, g)\} = \min\{6, 2\} = 2$	$\pi(e) = f$ $\pi(g) = f$
g	{h}	$\lambda(h) = \min\{\lambda(h), w(g, h)\} = \min\{7, 1\} = 1$	$\pi(h) = g$
h	\emptyset	---	---
d	{e}	$\lambda(e) = \min\{\lambda(e), w(d, e)\} = \min\{10, 9\} = 9$	$\pi(e) = d$
e	\emptyset	---	---

Árvore geradora mínima



Note que, apesar da MST obtida por Prim ser diferente da MST obtida por Kruskal, ambas possuem o mesmo peso mínimo, $w(T) = 37$. Em geral, a condição para que a MST seja única é que todos os

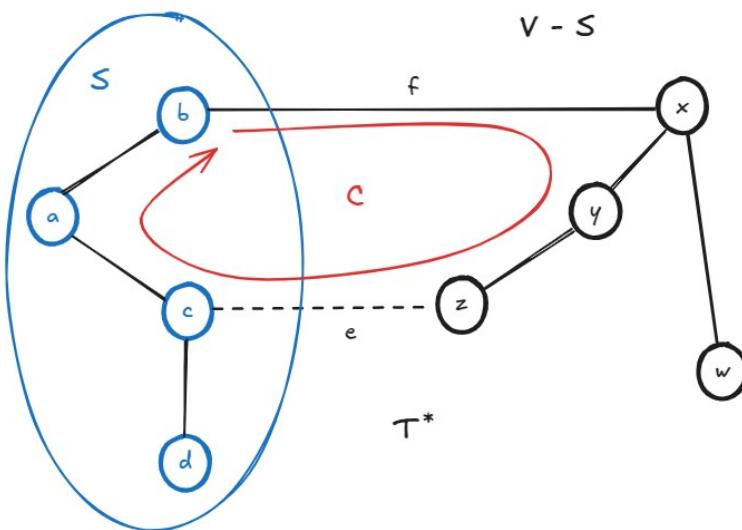
pesos das arestas do grafo $G = (V, E, w)$ sejam distintos, ou seja, não devemos ter arestas com pesos iguais no grafo de entrada.

Mapa de predecessores

v	a	b	c	d	e	f	g	h	i
$\pi(v)$	--	a	b	c	d	c	f	g	c
$\lambda(v)$	0	4	8	7	9	4	2	1	2

A seguir veremos um resultado fundamental para provar a otimalidade do algoritmo de Prim.

Propriedade do corte: Seja $G = (V, E, w)$ um grafo, S um subconjunto qualquer de V e $e \in E$ a aresta de menor custo com exatamente uma extremidade em S . Então, a MST de G contém e .

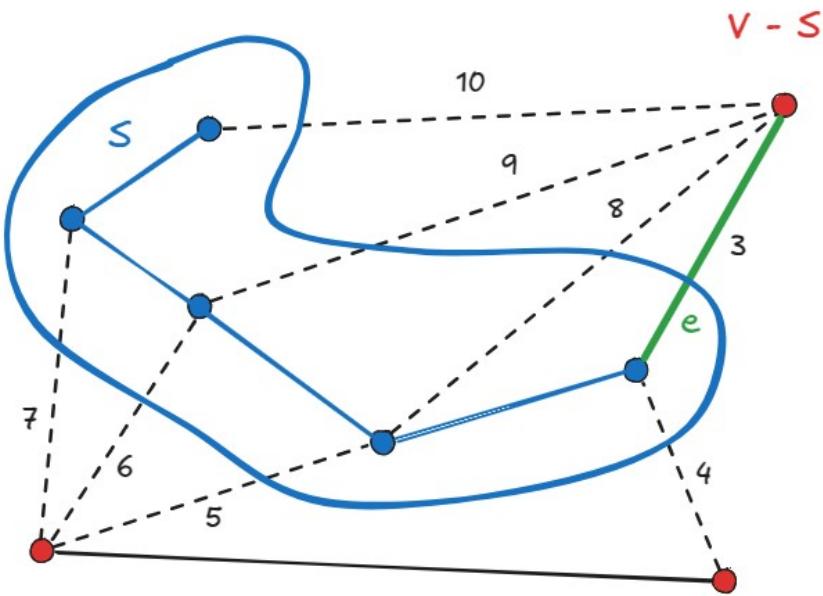


Prova por contradição:

1. Seja T^* uma MST de G .
2. Hipótese: Suponha que $e \notin T^*$ (aresta de menor peso com exatamente uma extremidade em S).
3. Ao adicionar e em T^* cria-se um único ciclo C .
4. Para que $T^* - e$ fosse uma MST, tem que haver alguma outra aresta f com apenas uma extremidade em S (senão T^* seria desconexo).
5. Então $T = T^* + e - f$ também é árvore geradora. Como, $w(e) < w(f)$, segue que T tem peso mínimo.
6. Logo, T^* não é MST de G (contradição), o que invalida a hipótese inicial de que $e \notin T^*$.

Teorema: A árvore T obtida pelo algoritmo de Prim é uma MST de G .

1. Seja S o subconjunto de vértices de G na árvore T (definido pelo algoritmo).
2. O algoritmo de Prim adiciona em T a cada passo a aresta de menor custo com apenas um vértice extremidade em S .
3. Portanto, pela propriedade do corte, toda aresta adicionada pertence a uma MST de G .



Análise da complexidade

Basicamente, há duas formas de analisar a complexidade do algoritmo de Dijkstra dependendo das estruturas de dados utilizadas.

Caso 1: $G = (V, E)$ é representado por uma matriz de adjacências
Fila de prioridades Q representada por um array simples de n elementos.

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$ (é $O(1)$ para cada vértice)
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = \text{ExtractMin}(Q)$ é $O(n)$ (equivale a encontrar menor elemento do array)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa k = $d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(1)$ (acesso direto)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n) + O(n)*O(n) + (O(1) + O(1))*(d(v_1) + d(v_2) + \dots + d(v_n))$$

Sabendo que a multiplicação de dois termos lineares resulta em quadrático e que de acordo com o Hankshaking Lema, a soma dos graus de um grafo é igual a duas vezes o número de arestas, temos:

$$T(n) = O(n) + O(n^2) + O(1)*O(m)$$

Como em todo grafo básico simples $m < n^2$, temos finalmente que o algoritmo é $O(n^2)$.

Caso 2: $G = (V, E)$ representado por uma lista de adjacências
Fila de prioridades Q representada por um heap binário (min-heap)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$, pois todos os pesos são iguais inicialmente (infinitos).
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = \text{ExtractMin}(Q)$ é $O(\log n)$ (dequeue), mas dentro de loop (n vezes)

e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)

f) Decrease_Key é $O(\log n)$ (pode ter que subir no min-heap até a raiz – altura da árvore)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n \log n) + (O(1) + O(\log n)) * (d(v_1) + d(v_2) + \dots + d(v_n))$$

De modo similar ao caso anterior, podemos escrever:

$$T(n) = O(n) + O(n \log n) + O(m) + O(m \log n)$$

Como o termo com logaritmo domina os demais: $T(n) = O((n+m) \log n)$

Mas como em grafos conexos $m > n - 1$, chega-se que $T(n)$ é $O(m \log n)$.

Qual das duas implementações é mais eficiente? Depende! Devemos preferir a primeira opção se $m \log n > n^2$, o que equivale a:

$$\frac{m}{n^2} > \frac{1}{\log n} \rightarrow d > \frac{1}{\log n}$$

Em grafos mais densos, o caso 1 é mais eficiente.

Em grafos menos densos, o caso 2 é mais eficiente.

Por exemplo, se $n = 1024$, temos a seguinte regra:

- se $d > 0.1$, opção 1 é melhor

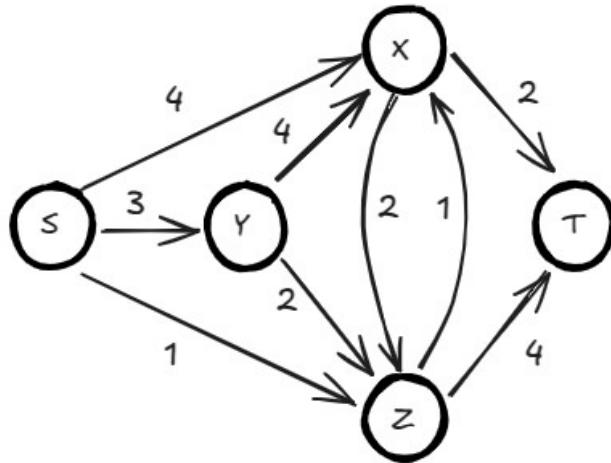
- se $d < 0.1$, opção 2 é melhor

Para que d seja igual a 0.1, um grafo com $n = 1024$ vértices deve ter 104857 arestas.

“The real voyage of discovery consists not in seeking new landscapes, but in having new eyes.”
-- Marcel Proust

Grafos: Fluxo em redes e o algoritmo de Ford-Fulkerson

Motivação: maximizar o fluxo que pode ser produzido pela fonte s e consumido pelo terminal t.



Consideraremos grafos direcionados $G = (V, E, c)$ onde $c: E \rightarrow N^+$ é a capacidade de uma aresta.
 Capacidade é a quantidade máxima de informação que pode ser transmitida por uma aresta.
 Iremos classificar os vértices de G como:

$s \in V$: source (gerador de fluxo: não entra nada, apenas sai)

$t \in V$: terminal ou sink (absorve fluxo: não sai nada, apenas entra)

$\forall v \in V - \{s, t\}$: nós internos (intermediários)

Em nossos exemplos iremos considerar apenas um único source e um único terminal

Def: Um fluxo s-t é uma função $f: E \rightarrow N^+$ (associa um inteiro a cada aresta) que satisfaz:

i) $\forall e \in E, f(e) \leq c(e)$ (restrição de capacidade)

ii) $v(f) = \sum_{e \in O(s)} f(e) = \sum_{e \in I(t)} f(e)$ (fluxo gerado na fonte é igual ao fluxo consumido no terminal)

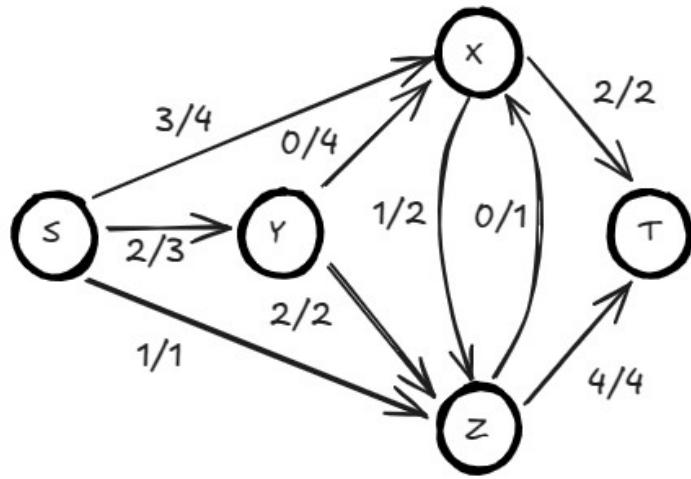
iii) $\forall v \in V - \{s, t\}$ (nó interno)

$$\sum_{e \in I(v)} f(e) = \sum_{e \in O(v)} f(e) \quad (\text{conservação do fluxo})$$

iv) Limite superior: para o valor de fluxo máximo que pode circular em G

$$v(f) = \sum_{e \in O(s)} c(e)$$

Como exemplo, verifique que um fluxo válido para o grafo anterior é representado pela figura a seguir.



Basta verificar as propriedades:

i) OK, pode-se ver facilmente que $\forall e \in E, f(e) \leq c(e)$

ii) $\sum_{e \in O(s)} f(e) = 1 + 2 + 3 = 6 = 2 + 4 = \sum_{e \in I(t)} f(e)$ (OK)

iii) Para $\{x, y, z\}$ temos

$$\sum_{e \in I(x)} f(e) = 3 + 0 + 0 = 1 + 2 = \sum_{e \in O(x)} f(e) \quad (\text{OK})$$

$$\sum_{e \in I(y)} f(e) = 2 = 0 + 2 = \sum_{e \in O(y)} f(e) \quad (\text{OK})$$

$$\sum_{e \in I(z)} f(e) = 1 + 1 + 2 = 0 + 4 = \sum_{e \in O(z)} f(e) \quad (\text{OK})$$

Portanto, temos de fato um fluxo válido.

Notação:

$$f^{\text{in}}(v) = \sum_{e \in I(v)} f(e) \quad f^{\text{out}}(v) = \sum_{e \in O(v)} f(e)$$

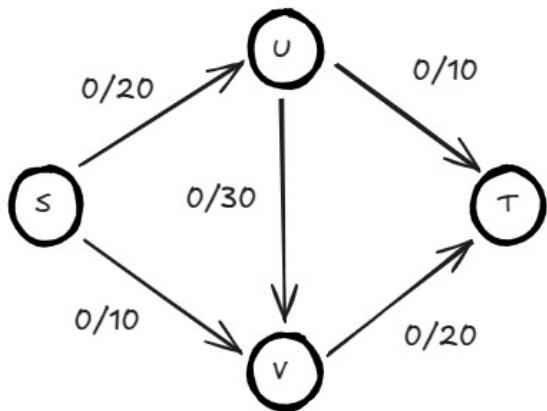
O Problema do Fluxo Máximo

Dado $G = (V, E, c)$ qual o máximo valor de $v(f)$ que pode chegar em t ?

Ideia geral

1. Condição inicial: $f(e) = 0, \forall e \in E$
2. Iteração: encontrar um caminho $s-t$ e transmitir fluxo
3. Condição de parada: Todo caminho $s-t$ encontra-se saturado

Exemplo:



Caso 1. $P = s \cup u \cup t$
 $v(f) = 20$

Caso 2. $P_1 = s \cup u \cup t \quad v(f) = 10$
 $P_2 = s \cup v \cup t \quad v(f) = 10 + 10 = 20$
 $P_3 = s \cup u \cup v \cup t \quad v(f) = 20 + 10 = 30$

Note que em cada um dos casos, o valor do fluxo obtido é diferente. Não é bom que o valor do fluxo dependa da escolha dos caminhos, pois senão o algoritmo iria chegar a resultados diferentes para um mesmo problema. O ideal é que o fluxo máximo seja obtido independente da escolha dos caminhos. Para isso iremos definir o grafo residual.

Def: Grafo Residual $G_f = (V_f, E_f)$ gerado a partir de $G = (V, E, c)$

- i) $V_f = V$ (conjunto de vértices é o mesmo)
- ii) $E_f \neq E$ pois $\forall e \in E$ pode gerar até 2 arestas em E_f
 - a) Forward-Edge (FE): na mesma direção da aresta e
 $\forall e \in E$ tal que $f(e) < c(e)$, \exists em E_f uma aresta e' com capacidade residual $c(e') = c(e) - f(e)$ (exatamente o que falta para saturar)
 - b) Backward-Edge (BE): na direção contrária a aresta e
 $\forall e \in E$ tal que $f(e) > 0$, \exists em E_f uma aresta e'' com capacidade residual $c(e'') = f(e)$ (exatamente o que já passa)

OBS: Note que no início não existem Backward-Edges pois os fluxos iniciais são nulos

RESUMO

1. $\forall e \in E$ com $f(e) = 0$ gera apenas Forward-Edge e'
2. $\forall e \in E$ com $f(e) = c(e)$ gera apenas Backward-Edge e''
3. $\forall e \in E$ com $0 < f(e) < c(e)$ gera 2 arestas: e' (FE) e e'' (BE)



Caminhos aumentados

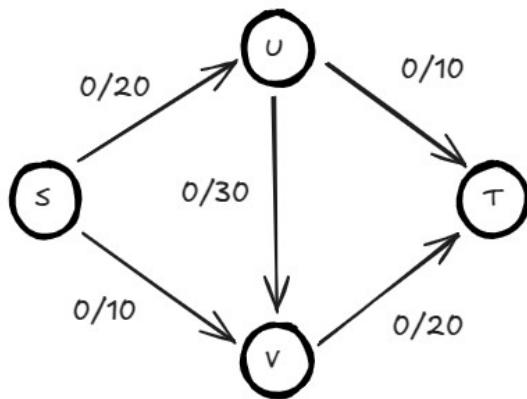
Para melhorar um fluxo inicial com valor f , devemos buscar por caminhos P_{st} no grafo residual $G_f = (V_f, E_f)$. A primitiva Augment que realiza essa operação de melhorar um fluxo f é dada por:

```

# melhora fluxo f utilizando o caminho P em G_f
Augment(f, P) {
    b = gargalo(P) # é o máximo que podemos passar por P
    for each e = (u, v) in P {
        if e está a favor do fluxo s-t em G (F.E.)
            f(e) = f(e) + b
        else (B.E.)
            f(e) = f(e) - b
    }
    return f
}

```

Exemplo:



Passo 1: Grafo residual é o próprio G

$$f = 0$$

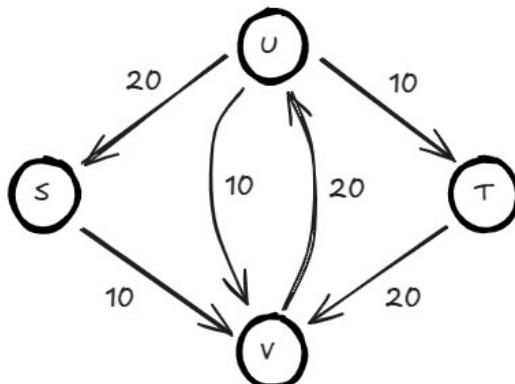
$$P = suvt$$

$$b = 20$$

$$f(su) = 0 + 20 = 20$$

$$f(uv) = 0 + 20 = 20$$

$$f(vt) = 0 + 20 = 20$$



Passo 2:

$$f = 20$$

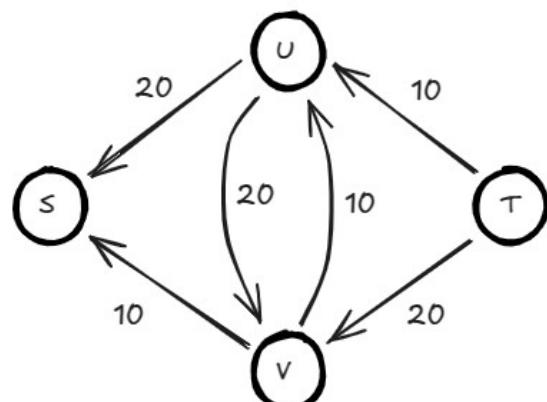
$$P = svut$$

$$b = 10$$

$$f(sv) = 0 + 10 = 10$$

Contrária a aresta (u,v)

$$f(vu) = f(uv) - 10 = 20 - 10 = 10$$



Passo 3:

$$f = 30$$

$\nexists P_{st}$ em $G_f \rightarrow$ PARE

\Rightarrow O fluxo máximo vale 30

Teorema: O resultado da operação $f' = \text{Augment}(f, P)$ é um fluxo válido.

i) f' não excede a capacidade (no caso de F.E.)

$$f'(e) = f(e) + b \leq f(e) + (c(e) - f(e))$$

pois para qualquer aresta F.E. do caminho $c(e) - f(e)$ será maior ou igual que b

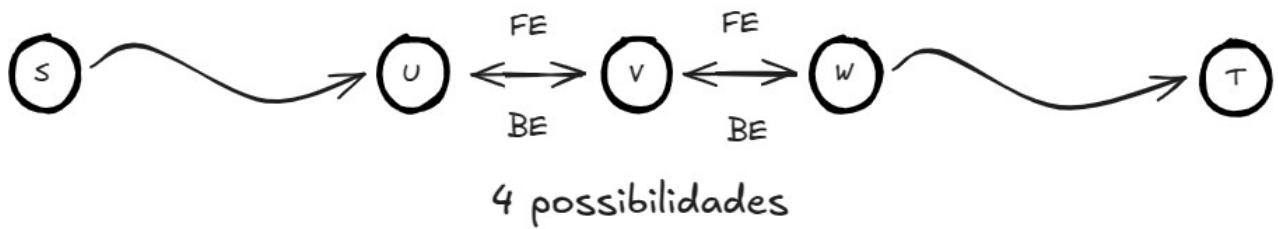
ii) f' nunca é inferior a zero (no caso de B.E.)

$$f(e) \geq f'(e) = f(e) - b \geq f(e) - f(e) = 0$$

pois para qualquer aresta B.E. do caminho $f(e)$ será maior ou igual que b

iii) conservação (tudo que entra em v sai de v)

$$f^{\text{in}}(v) = f^{\text{out}}(v) \quad \text{para todo nó interno } v$$



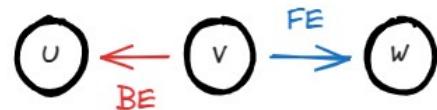
+ b na entrada
+ b na saída



+ b na entrada
- b na entrada



- b na entrada
- b na saída



- b na saída
+ b na saída

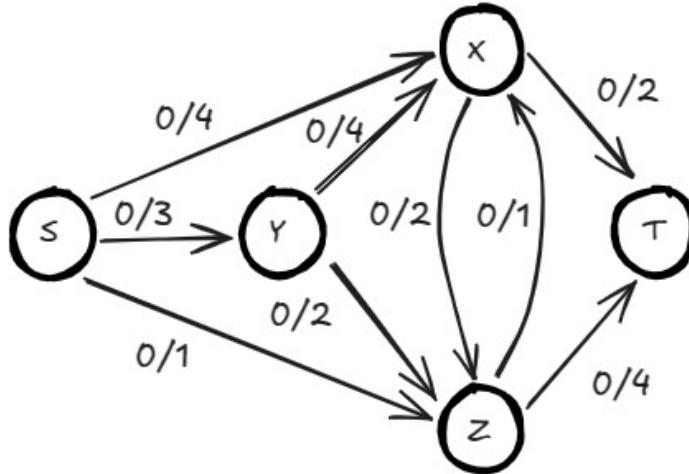
mostrando que em qualquer um dos casos, a conservação do fluxo é preservada!

Portanto, a operação $\text{Augment}(f, P)$ preserva fluxos e podemos usá-la para melhorar um dado fluxo. Veremos a seguir uma sequência lógica de passos para obter o fluxo máximo em um grafo: o algoritmo de Ford-Fulkerson, um algoritmo guloso pois tenta passar o máximo de fluxo possível em cada caminho aumentado.

```
Max_Flow(G, s, t, c) {
    f = 0
    for each e ∈ E
        f(e) = 0
    while ∃Pst in Gf {
```

Seja P_{st} um caminho P_{st} no grafo residual
 $f' = \text{Augment}(f, P_{st})$
 Atualize o grafo residual G_f
 }
 }

Exemplo: Utilizando ao algoritmo de Ford-Fulkerson, encontre o fluxo máximo



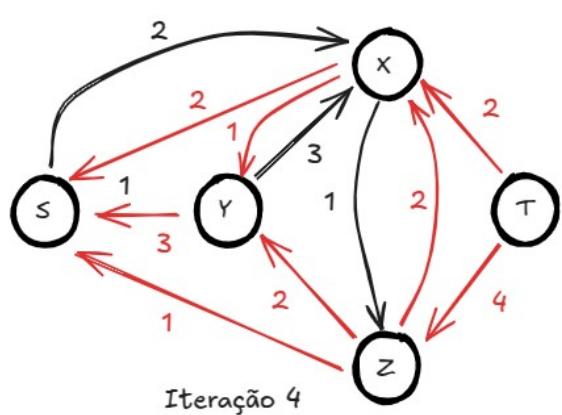
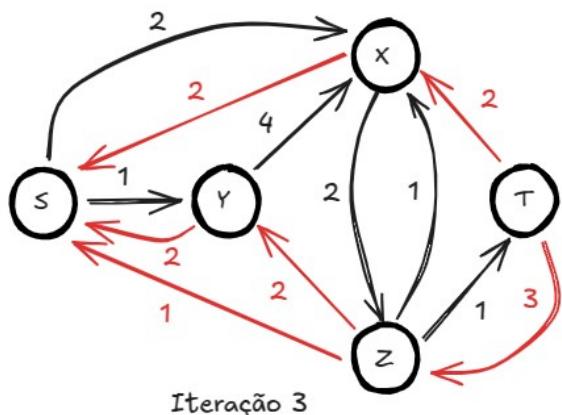
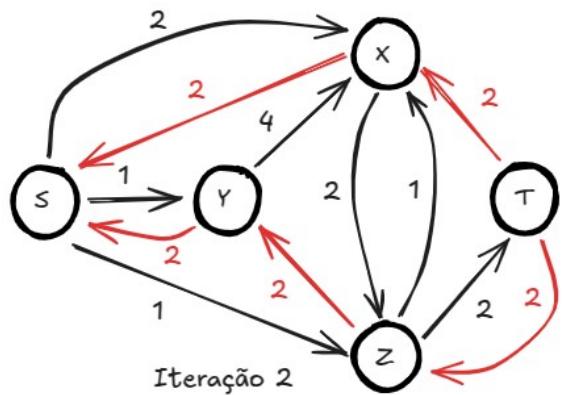
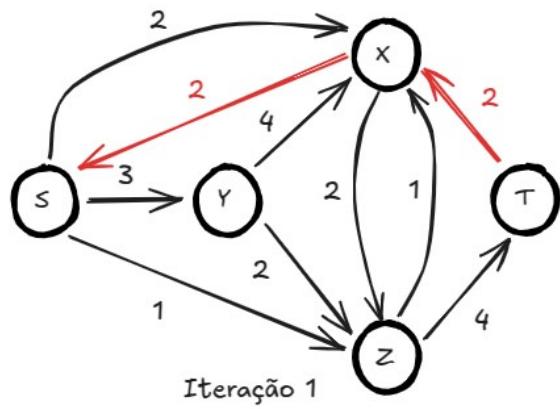
Trace do algoritmo

i	P_{st}	b	$f(e), \forall e \in P_{st}$	f
1	sxt	2	$f(sx) = 0 + 2 = 2$	$0 + 2 = 2$
2	syzt	2	$f(zt) = 0 + 2 = 2$ $f(sy) = 0 + 2 = 2$ $f(yz) = 0 + 2 = 2$	$2 + 2 = 4$
3	szt	1	$f(zt) = 0 + 2 = 2$ $f(sz) = 0 + 1 = 1$	$4 + 1 = 5$
4	syxzt	1	$f(zt) = 2 + 1 = 3$ $f(sy) = 2 + 1 = 3$ $f(yx) = 0 + 1 = 1$ $f(xz) = 0 + 1 = 1$	$5 + 1 = 6$

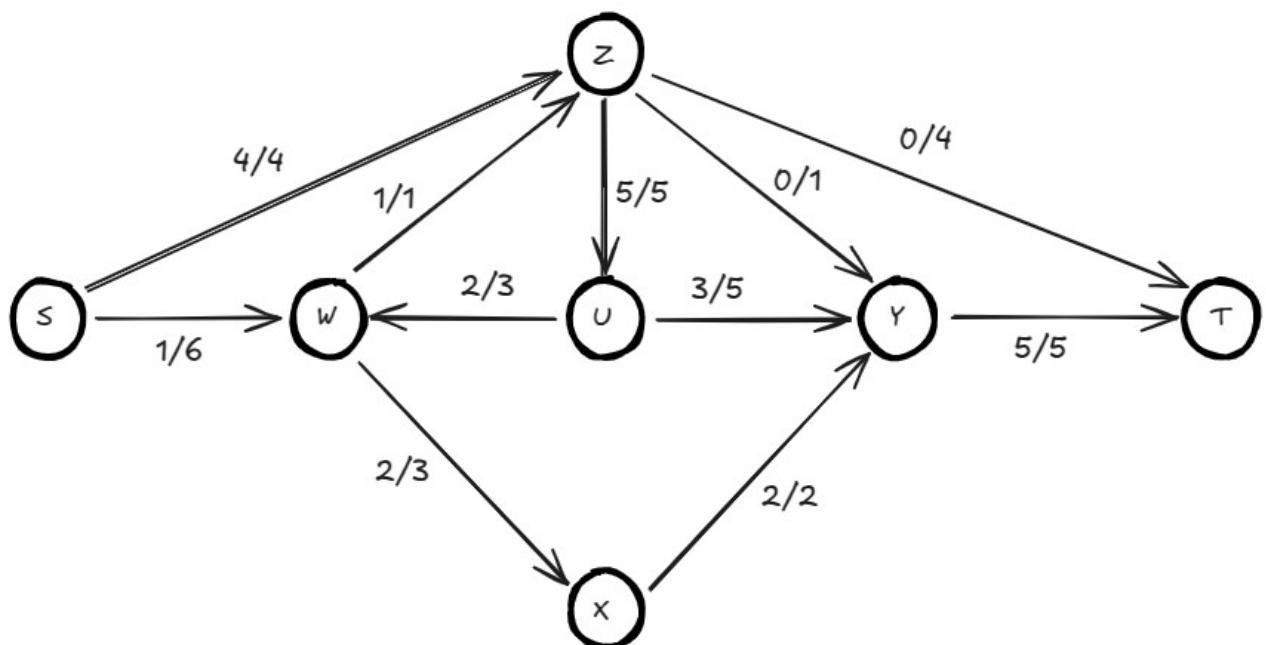
Portanto, o fluxo máximo vale 6.

A seguir encontram-se os grafos residuais após cada um dos passos do algoritmo.

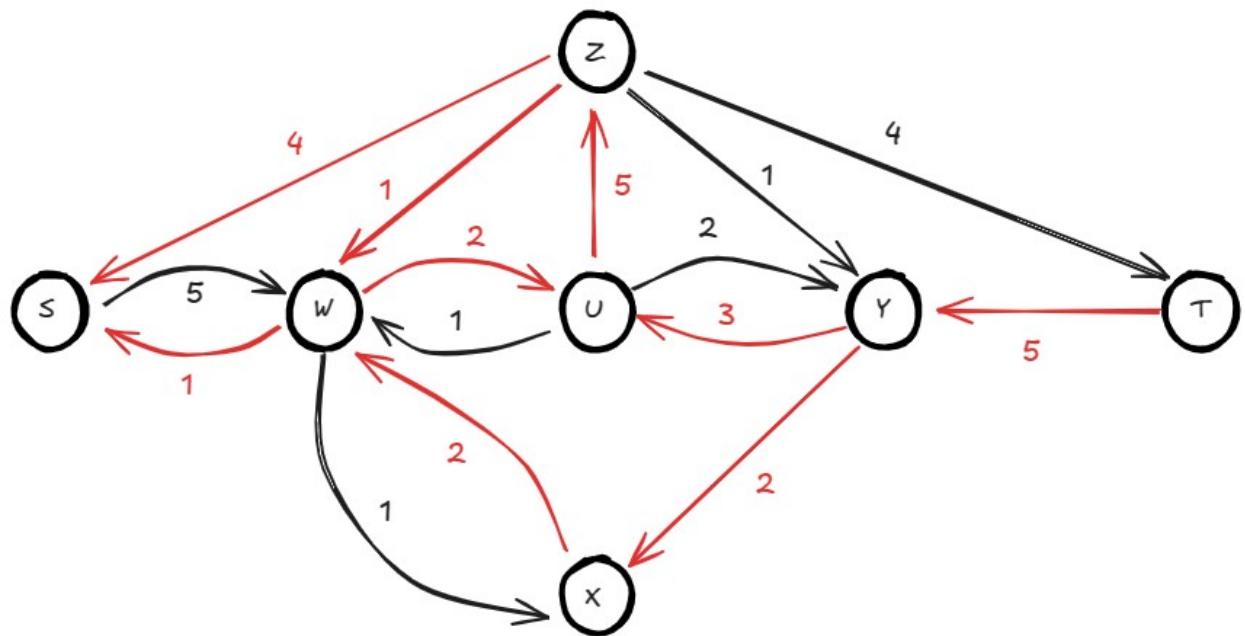
Note que a cada iteração, após passarmos um fluxo pelo caminho st , o grafo residual é modificado para refletir as novas capacidades restantes.



Exercício: Dado o grafo G a seguir, responda: o fluxo dado é máximo? Justifique sua resposta.



Para verificar se o fluxo dado é máximo, devemos observar o grafo residual. Sendo assim, o grafo residual de G fica:



Note que $\exists P_{st} = swuzt$ no grafo residual, portanto o fluxo não pode ser máximo. Como o gargalo do caminho é 2, podemos aumentar o fluxo nessas arestas de duas unidades, ou seja:

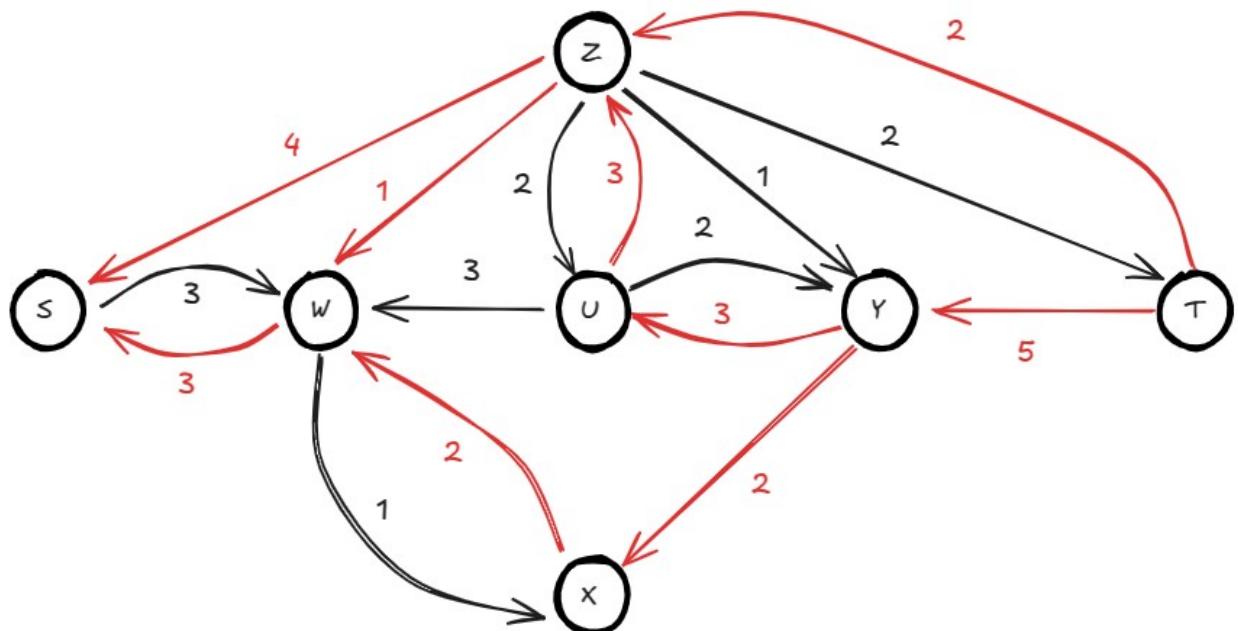
$$f(sw) = 1 + 2 = 3, \text{ pois a aresta } (s,w) \text{ está alinhada ao fluxo em } G$$

$$f(wu) = 2 - 2 = 0, \text{ pois a aresta } (w,u) \text{ está ao contrário do fluxo em } G$$

$$f(uz) = 5 - 2 = 3, \text{ pois a aresta } (u,z) \text{ está ao contrário do fluxo em } G$$

$$f(zt) = 0 + 2 = 2, \text{ pois a aresta } (z,t) \text{ está alinhada ao fluxo em } G$$

De modo que o valor total do fluxo agora é 7. O grafo residual é atualizado para:



Note que $\nexists P_{st}$ no grafo residual. Portanto, podemos concluir agora que o fluxo é máximo.

Fluxos e cortes

Aplicações: segmentação de imagens, classificação supervisionada, emparelhamentos

Motivação: cortes especificam limites superiores para $v(f)$ (valor do fluxo máximo). A relação entre fluxo e cortes nos permite resolver um problema NP-Hard (corte mínimo em grafos) em tempo polinomial.

Ideia: Ao partitionar G em 2 componentes A e B , a capacidade das arestas cortadas impõe um limite superior para o fluxo que pode sair de A e chegar em B .

Def: Um corte $s-t$ em G é qualquer partição de V em A e B tal que $s \in A$ e $t \in B$

Def: A capacidade de um corte é definida como:

$$c(A, B) = \sum_{e \in O(A)} c(e) \quad (\text{soma das capacidades das arestas que saem de } A \text{ e chegam em } B)$$

Min-cut/Max-flow: resultado que mostra a equivalência entre 2 problemas de otimização duais.

=> Encontrar o fluxo máximo corresponde a encontrar o corte de mínima capacidade (2 problemas que podem ser resolvidos com um único algoritmo)

Para provar o teorema Min-cut/Max-flow, iremos primeiramente demonstrar uma série de resultados preliminares importantes que serão utilizados depois.

Teorema: Seja f um fluxo $s-t$ e (A, B) um corte $s-t$. Então:

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \quad (\text{ou seja, o valor do fluxo depende diretamente do corte: tudo que sai de } A \text{ menos tudo que entra em } A)$$

$$v(f) = \sum_{e \in O(s)} f(e) = f^{\text{out}}(s)$$

Sabe-se que $f^{\text{in}}(s) = 0$ então podemos escrever $v(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$

Mas também sabemos que $\forall v \in V - \{s, t\}$ (nó intermediário), pela conservação do fluxo vale:

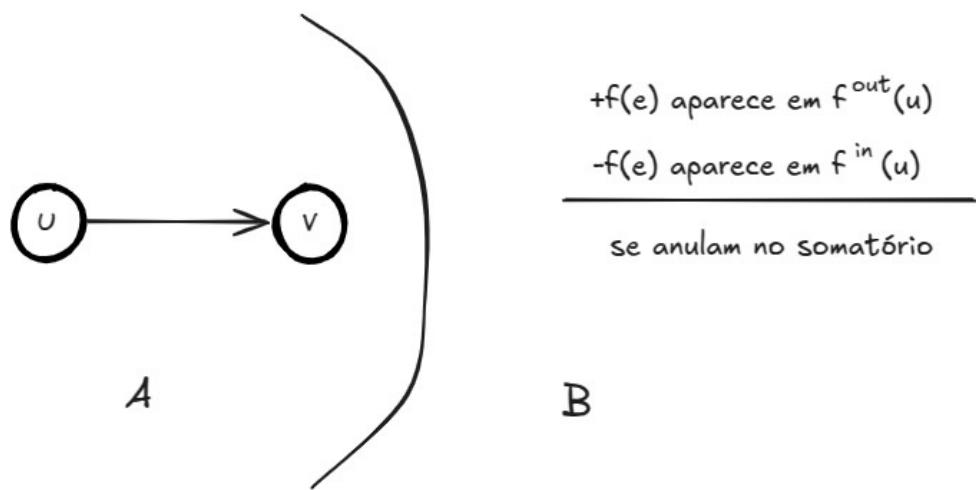
$$f^{\text{in}}(v) = f^{\text{out}}(v) \Leftrightarrow f^{\text{out}}(v) - f^{\text{in}}(v) = 0$$

Desse modo, podemos expressar $v(f)$ como:

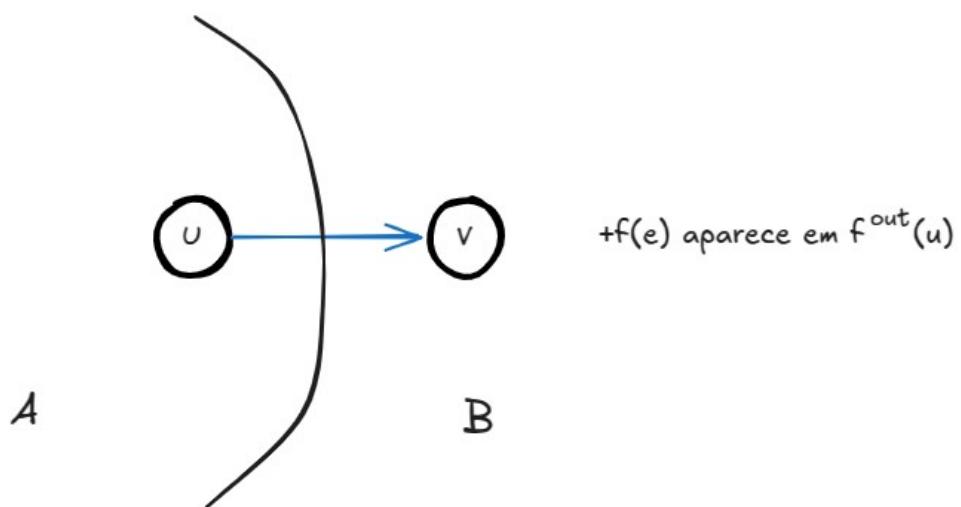
$$v(f) = \sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v)) \quad (\text{todos os termos desse somatório são nulos, exceto para } s)$$

Porém, existem apenas 4 tipos de arestas em G :

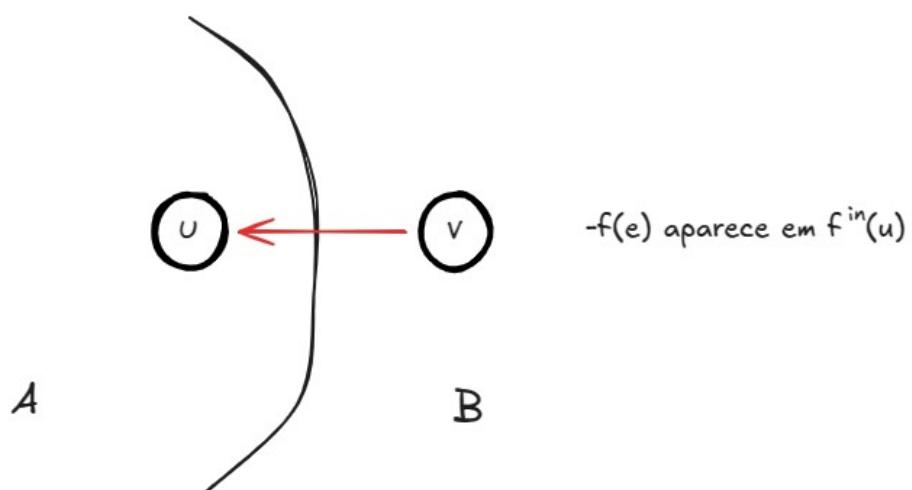
i) $e = \langle u, v \rangle$ com $u, v \in A$



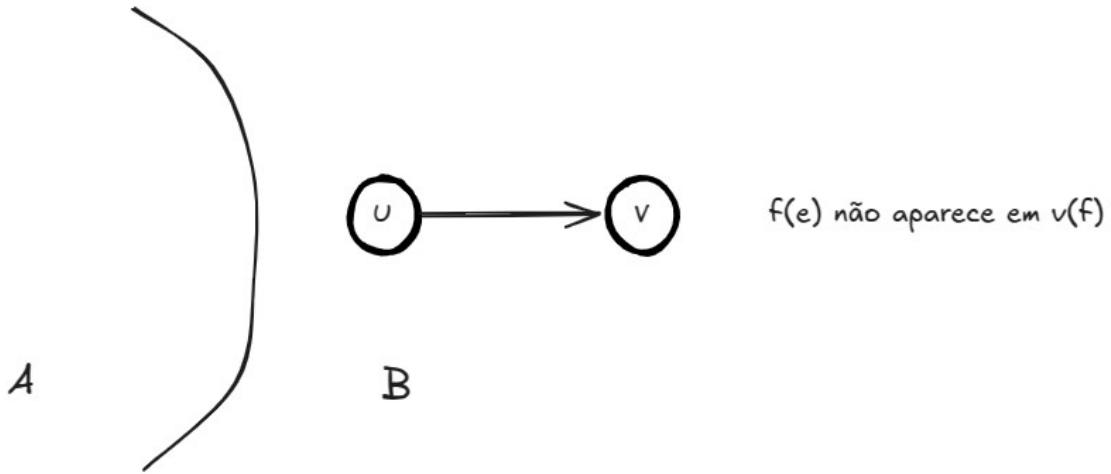
ii) $e = \langle u, v \rangle$ com $u \in A$ e $v \in B$



iii) $e = \langle v, u \rangle$ com $u \in A$ e $v \in B$



iv) $e = \langle u, v \rangle$ com $u, v \in B$



Portanto, $v(f)$ pode ser expresso por:

$$v(f) = \sum_{e \in O(A)} f(e) - \sum_{e \in I(A)} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A)$$

OBS: Note que $f^{\text{out}}(A) - f^{\text{in}}(A) = f^{\text{in}}(B) - f^{\text{out}}(B)$

Teorema: Seja um fluxo s-t qualquer e (A, B) um corte s-t. Então, $v(f) \leq c(A, B)$

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \leq f^{\text{out}}(A) = \sum_{e \in O(A)} f(e) \leq \sum_{e \in O(A)} c(e) = c(A, B)$$

OBS:

a) Se encontrarmos um fluxo s-t máximo f^* , não existe corte $c(A, B)$ com capacidade menor que $v(f^*)$.

b) Se encontrarmos um corte s-t (A, B) com capacidade mínima c^* , não existe fluxo s-t com valor maior que c^*

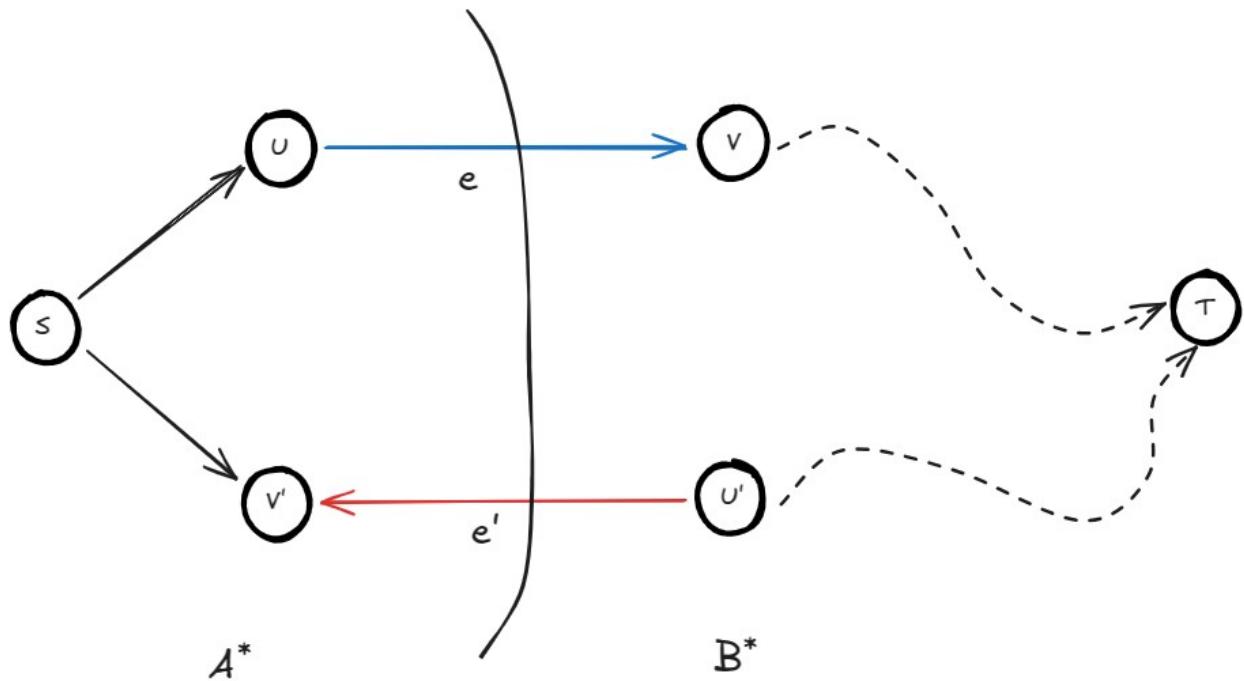
Teorema (Min-cut/Max-flow): Seja f^* o fluxo s-t tal que $\nexists P_{st}$ em G_f (fluxo gerado pelo algoritmo Ford-Fulkerson). Então, existe um corte (A^*, B^*) para qual o valor $v(f^*) = c(A^*, B^*)$. Consequentemente, f^* tem máximo valor de fluxo e o corte (A^*, B^*) tem a mínima capacidade dentre todos os possíveis. (encontrar fluxo máximo nos dá corte mínimo, o que era NP-hard).

Prova:

Seja $G_f = (V_f, E_f)$ o grafo residual de G no momento em que não existe mais caminho P_{st} . Podemos então dividir o conjunto V em 2 partições:

$$A^* = \{v \in V_f / \exists P_{sv}\} \quad (\text{atingíveis a partir de } s)$$

$$B^* = V_f - A^* \quad (\text{não atingíveis a partir de } s)$$



Note que:

a) $t \in B^*$ pois $\nexists P_{st}$ em G_f

b) Vamos analisar a aresta $e = \langle u, v \rangle$. Essa aresta certamente está saturada, ou seja, $f(e) = c(e)$, pois caso contrário haveria caminho P_{sv} uma vez que a capacidade residual da aresta F.E. correspondente em G_f seria $c(e) - f(e)$. Assim, podemos generalizar essa observação e escrever o seguinte fato:

$$\forall e \in O(A^*) \text{ tem } f(e) = c(e) \text{ (todas as arestas que saem de } A \text{ são saturadas)}$$

c) Análise da aresta $e' = \langle u', v' \rangle$. Essa aresta certamente não carrega fluxo algum ($f(e)$ nulo), pois caso contrário seria gerado uma aresta $e'' = \langle v', u' \rangle$ em G_f (B.E.) com capacidade residual $f(e')$, o que resultaria num caminho $P_{su'}$. Assim, podemos generalizar essa observação para:

$$\forall e \in I(A^*) \text{ tem } f(e) = 0 \text{ (todas as arestas que entram em } A \text{ tem fluxo nulo)}$$

Em resumo, se (A^*, B^*) é um corte, então:

$$\forall e \in O(A^*) , f(e) = c(e)$$

$$\forall e \in I(A^*) , f(e) = 0$$

Calculando o valor do fluxo temos:

$$v(f) = f^{\text{out}}(A^*) - f^{\text{in}}(A^*) = \sum_{e \in O(A^*)} f(e) - \sum_{e \in I(A^*)} f(e) = \sum_{e \in O(A^*)} c(e) = c(A^*, B^*) \quad (\text{min. capac. corte})$$

OBS: A saída do Ford-Fulkerson, $v(f)$, é de fato o valor de $c(A^*, B^*)$. Os conjuntos A^* e B^* são obtidos por uma simples busca em largura a partir de s em G_f .

Problemas NP-Completos

Há problemas computacionais que não podem ser resolvidos por algoritmos, nem que dispuséssemos de um período de tempo infinito: são os problemas incomputáveis ou indecidíveis.

Um exemplo clássico, formulado por Alan Turing, considerado o pai da computação, é o problema da parada (The halting problem). Dado um programa P e uma entrada I, devemos decidir se o programa terminará sua execução com essa entrada ou será executado para sempre. Não pode existir um algoritmo que receba P e resolva esse problema de decisão!

Veremos a seguir um breve esboço do principal argumento utilizado na prova formal.

Suponha que exista uma função computável $\text{halts}(f)$ que retorna True se a função f termina e False caso contrário (nunca termina). Seja a função g() a seguir:

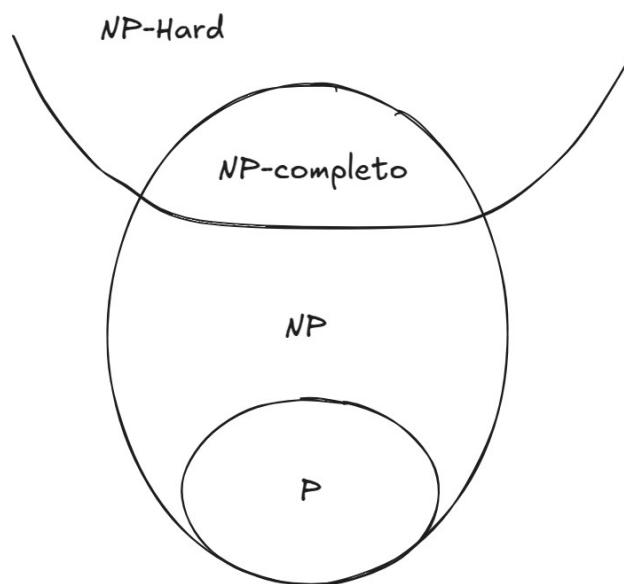
```
function g() {  
    if halts(g)  
        loop_forever  
}
```

Note que se $\text{halts}(g)$ retorna True, significa que ela termina, mas então ela entra em loop infinito, o que gera uma contradição!

Mas se $\text{halts}(g)$ retorna False, então a função g irá terminar (não entra no if), o que também gera uma contradição!

Portanto, a suposição inicial de que existe um função computável $\text{halts}()$ é FALSA!

Dentre os problemas computáveis, podemos organizar a seguinte estrutura de classes: problemas P, NP, NP-Completo e NP-Hard. A figura a seguir ilustra um diagrama.



Def: Dizemos que $p \in P$ se o problema é facilmente solucionável, ou seja, existe um algoritmo A com complexidade $O(n^k)$ (polinomial) que resolve P. Em termos matemáticos, o problema p pode ser resolvido por uma máquina de Turing determinística.

Exemplos são problemas como a ordenação, que possuem algoritmos exatos.

Def: Dizemos que $p \in NP$ se o problema é facilmente verificável, mas não facilmente solucionável. Em outras palavras, não existe algoritmo eficiente que resolva p , mas dada uma solução, é fácil verificar se ela é válida. Em termos matemáticos, pode ser resolvido por uma máquina de Turing não determinística.

Def: Os problemas $p \in NP$ para os quais $\forall q \in NP$ podem ser reduzidos a p de maneira eficiente (em tempo polinomial) são conhecidos como NP-Completos.

NP-Completos são problemas NP-Hard que estão em NP. Por isso são geralmente muito difíceis de serem resolvidos.

Def: Existem problemas $p \notin NP$ para os quais verificar se uma dada solução é válida sequer é viável: são os problemas NP-Hard (considerados os mais difíceis de todos)

Obs: A diferença é que problemas NP-Completo são problemas de decisão, enquanto que problemas NP-Hard são problemas de otimização.

A pergunta natural que surge é: o que é redução?

A redução de problemas é uma técnica fundamental na teoria da complexidade computacional para comparar a dificuldade de dois problemas. No contexto dos problemas NP-Completos, ela permite demonstrar que um problema é "pelo menos tão difícil quanto" outro, estabelecendo relações hierárquicas entre problemas. Essa abordagem é central para provar que um problema pertence à classe NP-Completo e para entender a estrutura da complexidade computacional.

Uma redução transforma instâncias de um problema A em instâncias de um problema B, de forma que:

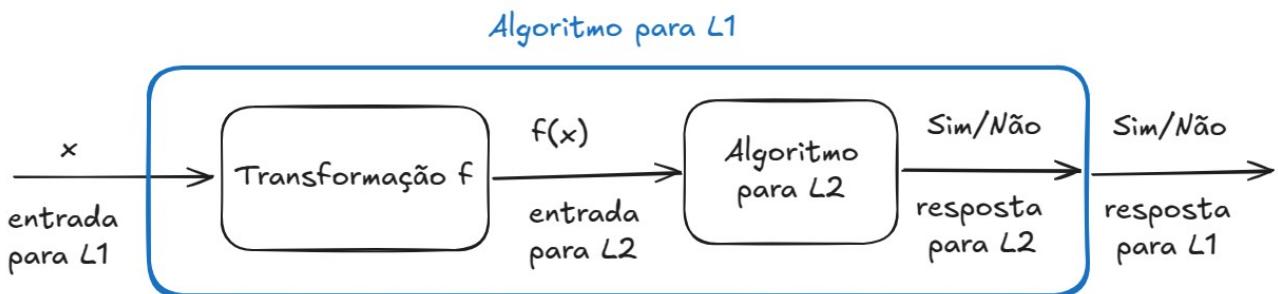
1. A resposta para a instância de B (sim/não, no caso de problemas de decisão) corresponda à resposta para A.
2. A transformação seja feita em tempo polinomial (polynomial-time reduction).

Se tal redução existe, dizemos que A é redutível a B (notação $A \leq_p B$). Isso implica que:

- Se B for resolvido eficientemente (em tempo polinomial), A também pode ser resolvido eficientemente.
- Se A for "difícil" (ex: NP-Completo), então B é pelo menos tão difícil quanto A.

Em outras palavras, sejam L_1 e L_2 dois problemas de decisão e suponha que o algoritmo A_2 resolva L_2 . Em outras palavras, se y é uma entrada para L_2 , então A_2 retornará True ou False, dependendo se $y \in L_2$ ou $y \notin L_2$.

Ideia: encontrar uma transformação de L_1 para L_2 de modo que o algoritmo A_2 possa ser parte de um algoritmo A_1 para resolver L_1 .



=> **Pergunta:** Como provar que um problema p é NP-Completo?

Parece impossível: Todo problema $q \in NP$ deve ser reduzido a p?

Ideia: tomar um problema NP-Completo conhecido e reduzí-lo a p. Se a redução for possível em tempo polinomial, prova-se que p é NP-Completo por transitividade da redução, ou seja, se um problema NP-Completo é reduzido a p em tempo polinomial, então todos os problemas em NP são reduzíveis a p em tempo polinomial.

Qual foi o primeiro problema a provado a ser NP-Completo?

SAT (Boolean satisfiability problem)

Consiste em determinar se existe uma interpretação que satisfaz uma dada expressão Booleana arbitrária. Existe alguma forma de atribuir valores lógicos as variáveis Booleanas da expressão lógica de modo que ela seja verdadeira?

Precisamos construir uma Tabela-Verdade.

Por exemplo, se tivermos uma expressão lógica com n variáveis atômicas, precisamos de 2^n linhas na Tabela-Verdade.

Claramente, a complexidade do problema é exponencial: $O(2^n)$

Um dos 7 problemas do milênio: P = NP ?

Instituto Clay dos EUA oferece um prêmio de 1 milhão de dólares para quem provar que P = NP.

Traria implicações inimagináveis na computação: se eventualmente você conseguir reduzir um problema NP-Completo para P, todos os problemas de NP virariam P (criptografia, segurança, etc.)

O Problema do Caixeiro Viajante (*Travelling Salesman Problem – TSP*)

Dado um grafo $G = (V, E, w)$ encontrar um ciclo Hamiltoniano de custo mínimo, ou seja, obter o ciclo C que minimiza:

$$w(C) = \sum_{e \in C} w(e)$$

Descrição do problema: A partir de um vértice inicial v_0 , visitar todos os demais uma única vez, voltando para v_0 , caminhando o mínimo possível. É um problema de otimização e não de decisão (NP-Hard).

Pergunta: como obter um ciclo Hamiltoniano de custo mínimo?

- Não se conhece algoritmo ótimo (não há garantias de que termos a solução ótima x^*)
- Devemos trabalhar com algoritmos aproximados (sub-ótimos).

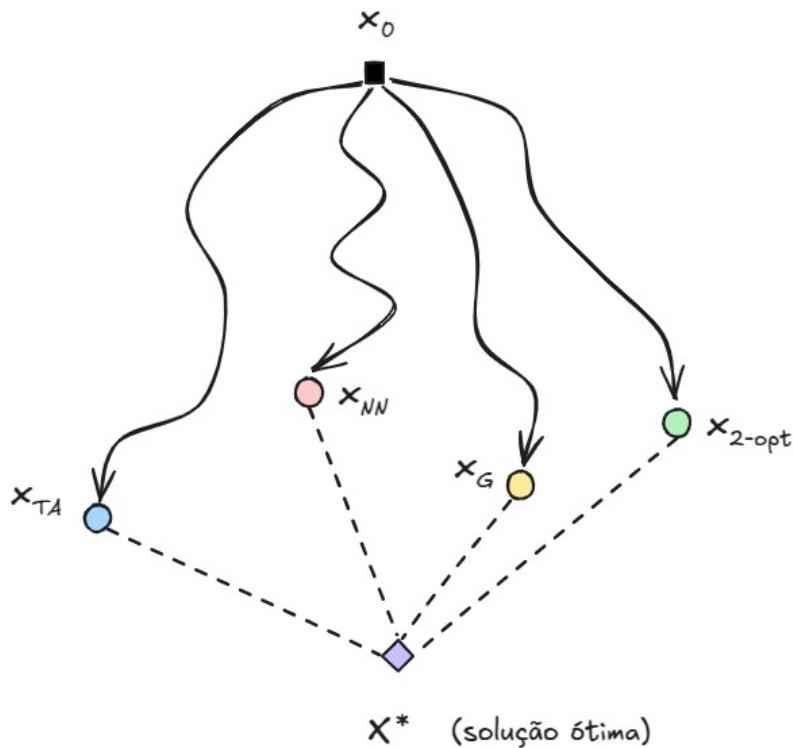
Mas como medir a qualidade da solução obtida por um algoritmo aproximado?

Devemos medir quão próximo da solução ótima é a solução aproximada. Definem-se:

$$f(x^*) : \text{custo da solução ótima (melhor possível)}$$

$$f(x_A) : \text{custo da solução obtida pelo algoritmo A}$$

Dizemos que o algoritmo A fornece uma c -aproximação para o TSP se $f(x_A) \leq c f(x^*)$



TSP métrico

Seja $G = (V, E, w)$ um grafo conexo com $w: E \rightarrow R^+$, em que w é uma função de distância. Então, a desigualdade triangular é satisfeita, ou seja:

$$\forall i, j, k \quad w(v_i, v_k) \leq w(v_i, v_j) + w(v_j, v_k)$$

Isso significa que tomar atalhos é sempre vantajoso (corta caminhos).

Uma instância do TSP em que os pesos das arestas do grafo de entrada G é uma função de distância é conhecida como TSP métrico.

Algoritmos para o TSP

A seguir apresentamos alguns dos principais algoritmos aproximados para o TSP, iniciando pelo método dos vizinhos mais próximos.

Nearest Neighbor

O algoritmo mais simples para o TSP utiliza uma abordagem gulosa: visitar sempre o vértice mais próximo do vértice atual. Porém, esse algoritmo não fornece boas soluções, em especial quando o número de vértices cresce de maneira arbitrária.

```
Nearest_Neighbor(G, w, v0) {  
    H = [v0]  
    while |H| < n {  
        x = Last(H)          # retorna o último vértice de H  
        Defina vi como o vizinho mais próximo de x  
        InsertEnd(H, vi)   # insere vi no final de H  
    }  
    InsertEnd(H, v0)  
}
```

Teorema: A solução obtida pelo algoritmo Nearest Neighbor no TSP métrico satisfaz:

$$f(x_{NN}) \leq \frac{1}{2}(\log_2 n + 1)f(x^*)$$

Note que se o número de vértices do grafo G aumenta, a qualidade da solução cai de maneira logarítmica. Em outras palavras, não é muito bom para grafos com grande quantidade de vértices.

Análise da complexidade

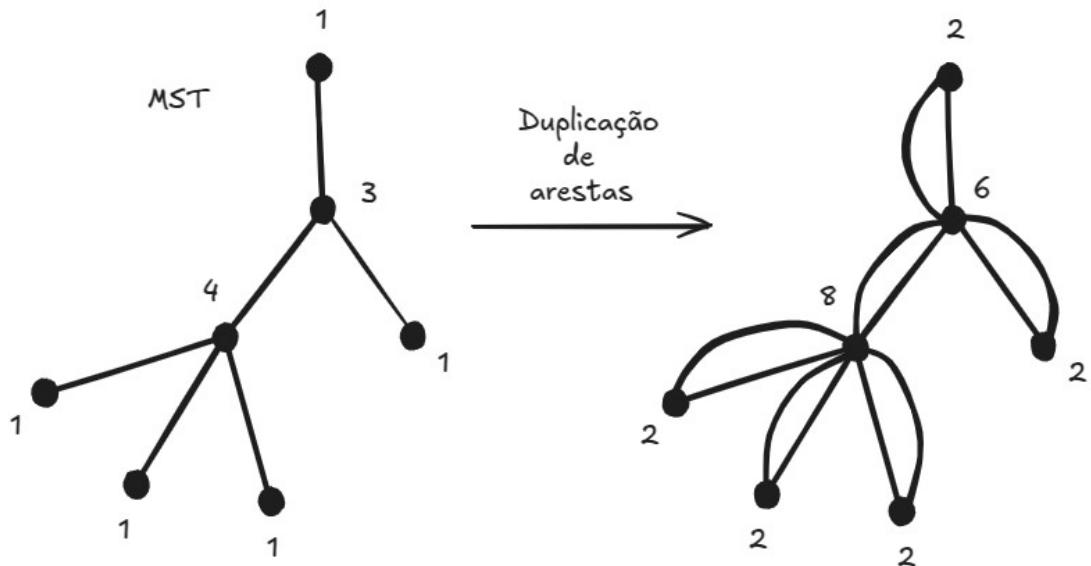
1. Note que o loop WHILE executa n vezes
2. Obter o último elemento da lista H pode ser realizado em tempo constante O(1) se mantivermos um contador para o número de elementos em H.
3. Para definir o vizinho mais próximo de x, é preciso verificar todas as distâncias da linha de x na matriz de adjacências, o que é O(n)
4. Inserção no final da lista H também pode ser feito em O(1) se mantivermos um contador para o número de elementos de H

Custo total: $n \times [O(1) + O(n) + O(1)] = O(n^2)$

Twice-Around

Este algoritmo combina a árvore geradora mínima de G e circuitos Eulerianos para construir uma solução aproximada para o TSP métrico.

Teorema: Seja $G = (V, E)$ um grafo conexo. Se T é uma MST de G , então se definirmos um supergrafo T' duplicando toda aresta de T , T' será Euleriano.



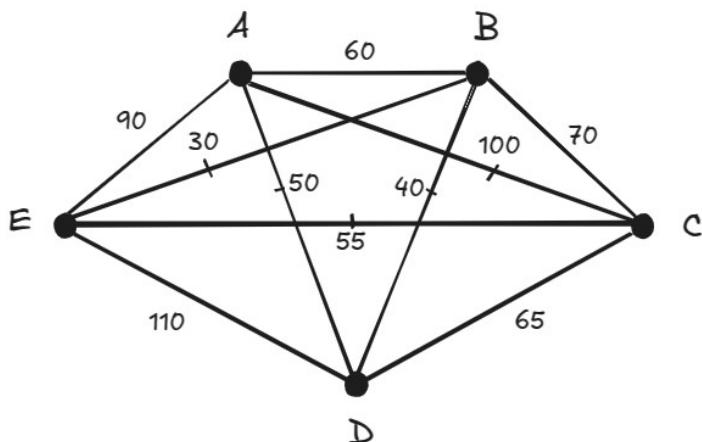
É fácil notar que ao duplicar todas as arestas da MST, estamos multiplicando os graus por 2. Assim, os nós folhas, passarão a ter grau 2 e todos os demais nós internos serão múltiplos de 2. A seguir apresentamos o algoritmo Twice_Around, para a solução aproximada do TSP métrico.

```

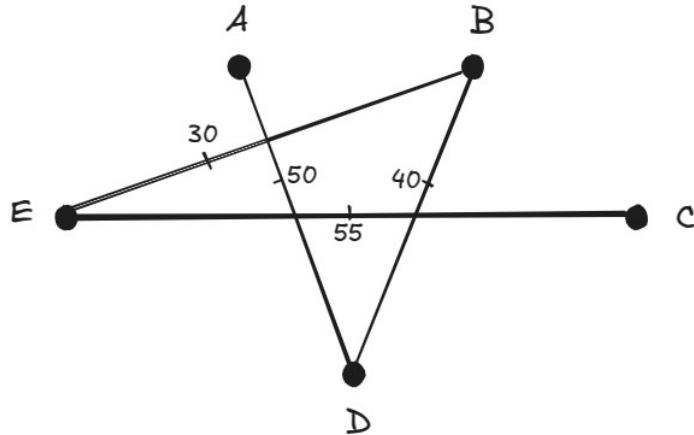
Twice_Around(G, w, v0) {
    H = ∅
    T = MST(G)
    for each e ∈ T
        T = T ∪ {e}
    L = Eulerian_Circuit(T)
    # Extrai MST de G
    # Duplica as arestas da MST
    while L ≠ ∅ {
        l = RemoveFirst(L)
        # Obtém circuito Euleriano
        if l ∉ H
            H = H ∪ {l}
        # Remove as repetições
    }
    return H
}

```

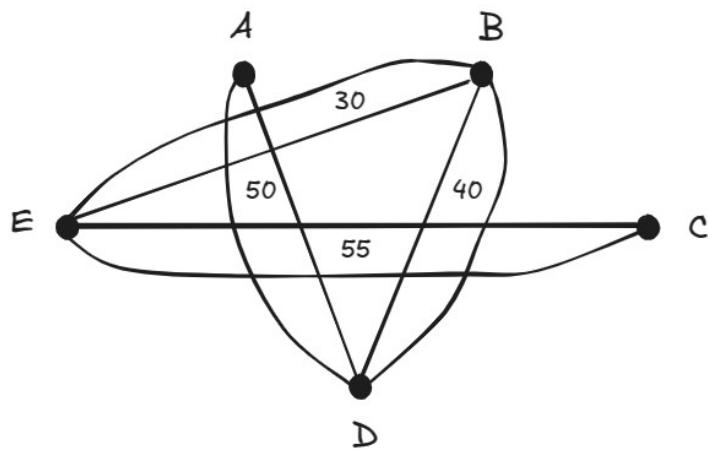
Veremos a seguir um exemplo ilustrativo da aplicação do algoritmo Twice-Around.



1º passo: Extrair MST de G



2º passo: Duplicar arestas da MST

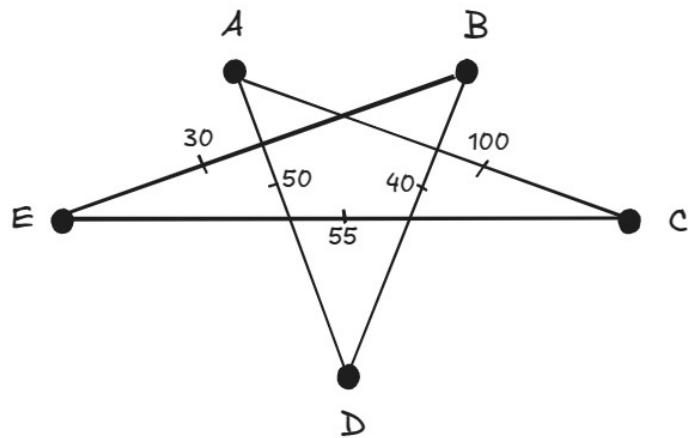


3º passo: Extrair um circuito Euleriano L

$$L = [A, D, B, E, C, E, B, D, A]$$

4º passo: Eliminar as repetições (equivale a tomar atalhos)

$$H = [A, D, B, E, C, A]$$



Note que o custo do ciclo Hamiltoniano obtido é $w(C) = 50 + 40 + 30 + 55 + 100 = 275$

Análise da complexidade

1. Obter a MST de G com Kruskal é $O(m \log n)$. Com Prim é $O(n^2)$ (estruturas estáticas) ou $O(m \log n)$ (estruturas dinâmicas).
2. A duplicação das arestas da MST pode ser feita em $O(m)$
3. A extração do circuito Euleriano L é lo algoritmo de Hierholzer é $O(m)$
4. Por fim, o WHILE é executado uma vez para cada aresta duplicada: $2m$ iterações, o que é $O(m)$
5. Portanto, o custo final é: $O(m \log n) + O(m) + O(m) + O(m) = O(m \log n)$

Ou seja, o ponto crítico (gragalo) no algoritmo Twice-Around é o cálculo da MST.

Veremos a seguir um resultado muito importante sobre a qualidade da solução obtida pelo algoritmo Twice-Around.

Teorema: A solução obtida pelo algoritmo Twice-Around no TSP métrico satisfaz:

$$f(x_{TA}) \leq 2f(x^*)$$

ou seja, a solução do TA é no máximo 2 vezes pior que a solução ótima (não importa o número de vértices de G). Melhor que Nearest Neighbor.

É fácil verificar que se x^* é a solução ótima, então se subtrairmos uma aresta e de x^* , obtemos uma árvore T, ou seja, $T = x^* - e$. Também sabemos que essa árvore T tem peso maior ou igual que a MST do grafo e por isso temos:

$$f(x^*) > w(T) \geq w(T_{MST})$$

o custo de x^* é maior que de T pois o ciclo tem uma aresta a mais

Então temos que:

$$f(x^*) > w(T_{MST})$$

o que implica em

$$2f(x^*) > 2w(T_{MST}) \quad (*) \quad (\text{é o peso do tour de Euler extraído no passo 2})$$

Se G é Euclidiano (tomar atalhos é sempre mais vantajoso devido a desigualdade triangular)

$$f(x_{TA}) \leq 2w(T_{MST}) \quad (**)$$

E portanto, combinando (*) e (**)

$$2f(x^*) > 2w(T_{MST}) \geq f(x_{TA})$$

O algoritmo de Christofides

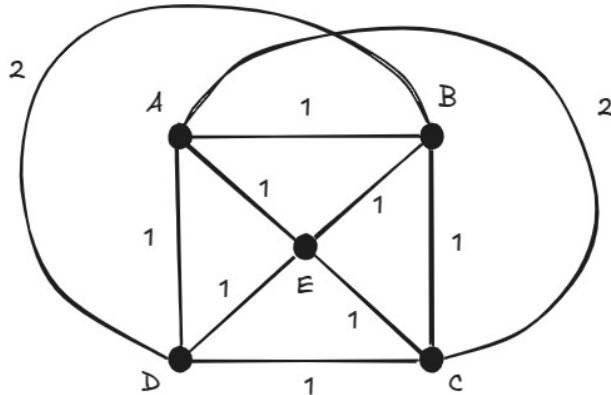
A ideia do algoritmo de Christofides consistem em melhorar ainda mais o algoritmo Twice-Around, modificando a forma com que transformamos a MST de G em um grafo Euleriano. Ao invés de simplesmente duplicar as arestas da árvore, devemos encontrar um emparelhamento perfeito de custo mínimo entre os vértices de grau ímpar na MST. Dessa forma, com menos duplicações de arestas, o custo adicional é menor.

```

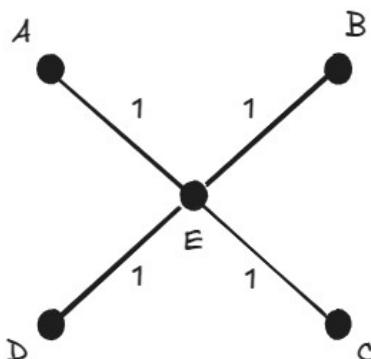
Christofides(G, w, v0) {
    H = ∅
    T = MST(G)                                # Extrai MST de G
    K = Odd_Vertices(T)                      # Constrói Kn (ímpares)
    M = Minimum_Weight_Matching(Ku)        # Emparelhamento mínimo
    T = T + M                                  # Adiciona arestas de M
    T = Eulerian_Circuit(T)                   # Obtém circuito Euleriano
    while L ≠ ∅ {
        l = RemoveFirst(L)
        if l ∉ H                               // Remove as repetições
            H = H ∪ {l}
    }
    return H
}

```

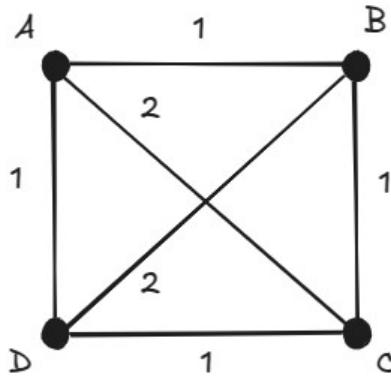
A seguir veremos um exemplo ilustrativo do funcionamento do algoritmo de Christofides. Considere o seguinte grafo G .



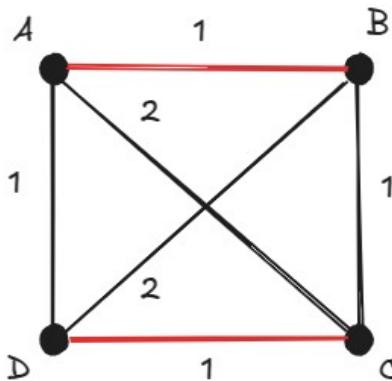
1º passo: Extrair a MST de G



2º passo: criar grafo completo K_n com todos os vértices de grau ímpar e ponderar as arestas com os caminhos mínimos entre cada par de vértices.



3º passo: encontrar um emparelhamento de custo mínimo entre os vértices de grau ímpar.



Pode-se mostrar que o número de emparelhamentos perfeitos no K_n é dado por:

$$\prod_{ímpar, i < n} i$$

No caso de $n = 4$, temos $1 \times 3 = 3$

No caso de $n = 6$, temos $1 \times 3 \times 5 = 15$

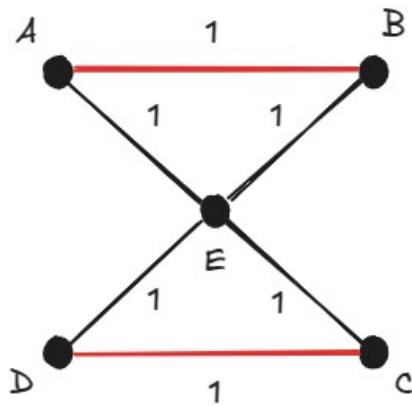
No caso de $n = 8$, temos $1 \times 3 \times 5 \times 7 = 105$

No caso de $n = 10$, temos $1 \times 3 \times 5 \times 7 \times 9 = 945$

Note que o número de possíveis emparelhamentos perfeitos cresce exponencialmente, o que inviabiliza a busca exaustiva pelo emparelhamento mínimo quando o número de vértices cresce. Existe um algoritmo com complexidade $O(n^2 m)$ que obtém o emparelhamento M de custo mínimo em um grafo arbitrário (não precisa ser bipartido). É o algoritmo de Edmonds, também conhecido como *Blossom algorithm*. Para os leitores interessados, indicamos o link a seguir:

https://en.wikipedia.org/wiki/Blossom_algorithm

4º passo: Adicione as arestas de M à MST



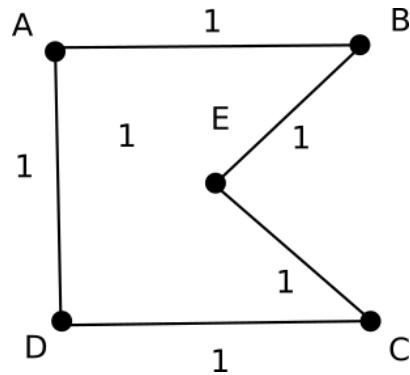
Agora, todos os vértices possuem grau par, então é possível extrair um circuito Euleriano.

5º passo: Encontre um circuito Euleriano L

$$L = [A, B, E, C, D, E, A]$$

6º passo: Eliminar as repetições (equivalente a tomar atalhos)

$$L = [A, B, E, C, D, A]$$



Análise da complexidade

1. Obter a MST de G com Kruskal é $O(m \log n)$. Com Prim é $O(n^2)$ (matriz de adjacências + array) ou $O(m \log n)$ (listas de adjacências + min-heap).

2. Obter o emparelhamento de custo mínimo M entre os vértices de grau ímpar: seja z o número de vértices ímpares. O grafo completo formado pelos vértices ímpares, denotado por K_z , terá m' arestas, onde:

$$m' = \frac{z(z-1)}{2}$$

3. Para ponderar os pesos do grafo K_z , será preciso encontrar caminhos mínimos entre cada par de vértices. Podemos executar z vezes o algoritmo de Dijkstra no grafo original, o que nos leva a uma complexidade $O(z m \log n)$

4. Para a construção do emparelhamento M de mínimo custo, o algoritmo de Edmonds possui complexidade $O(z^2 m')$

5. A extração do circuito Euleriano L é lo algoritmo de Hierholzer é $O(m)$

6. Por fim, o WHILE é executado uma vez para cada aresta duplicada: $2m$ iterações, o que é $O(m)$

7. Portanto, o custo final é:

$$O(m \log n) + O(z m \log n) + O(k^2 m') + O(m) \rightarrow O(z m \log n) + O(z^2 m')$$

Ou seja, é quadrático no número de vértices ímpares! Quando o número de vértices ímpares é grande, ou seja, $z \rightarrow n$, o custo pode ser bastante elevado em comparação com o algoritmo Twice-Around.

A seguir, iremos demonstrar um resultado fundamental sobre o algoritmo de Christofides.

Teorema: O algoritmo de Christofides fornece uma $3/2$ -aproximação para o TSP métrico.

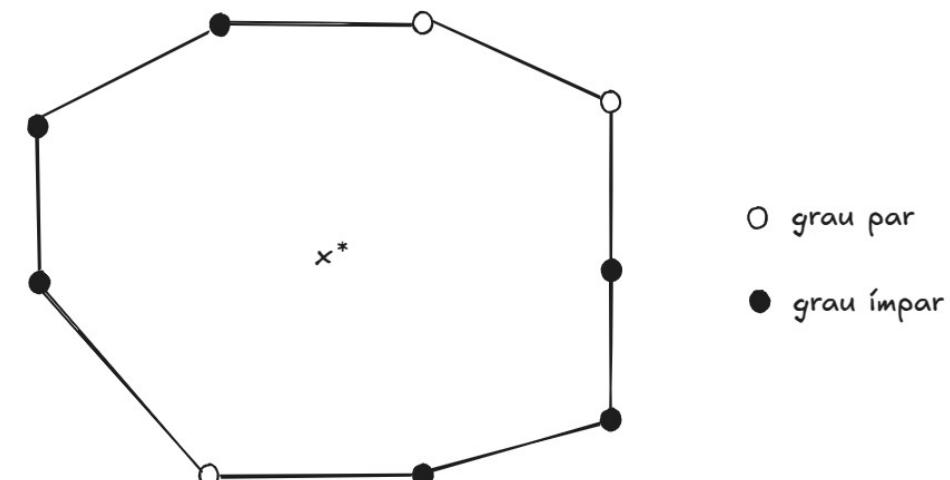
Prova: Desejamos mostrar que:

$$f(x_c) \leq \frac{3}{2} f(x^*)$$

onde x^* é a solução ótima. Sabemos que o custo da MST T é menor que x^* , ou seja:

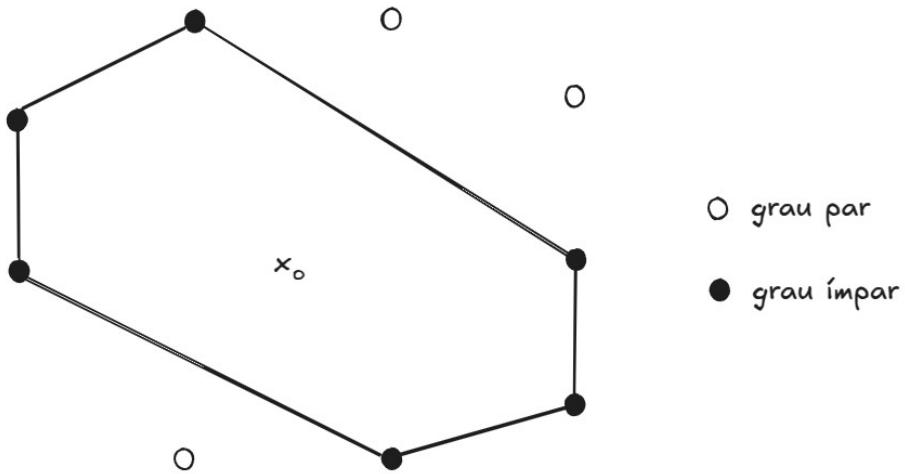
$$w(T_{MST}) < f(x^*)$$

pois $x^* - e$ é uma árvore com peso maior ou igual ao peso da MST de G . Basta agora provar que o custo do emparelhamento M entre os vértices ímpares é menor ou igual a $\frac{1}{2} f(x^*)$. Lembre-se que aqui, ao invés de duplicar todas as arestas da MST, adicionamos menos arestas a MST. Iniciamos a discussão com o ciclo Hamiltoniano de custo mínimo, cujo custo é $f(x^*)$.



onde os vértices pretos denotam os vértices de grau ímpar (subconjunto O de odd degree) e os vértices brancos denotam os vértices de grau par. Existe um ciclo Hamiltoniano de custo mínimo

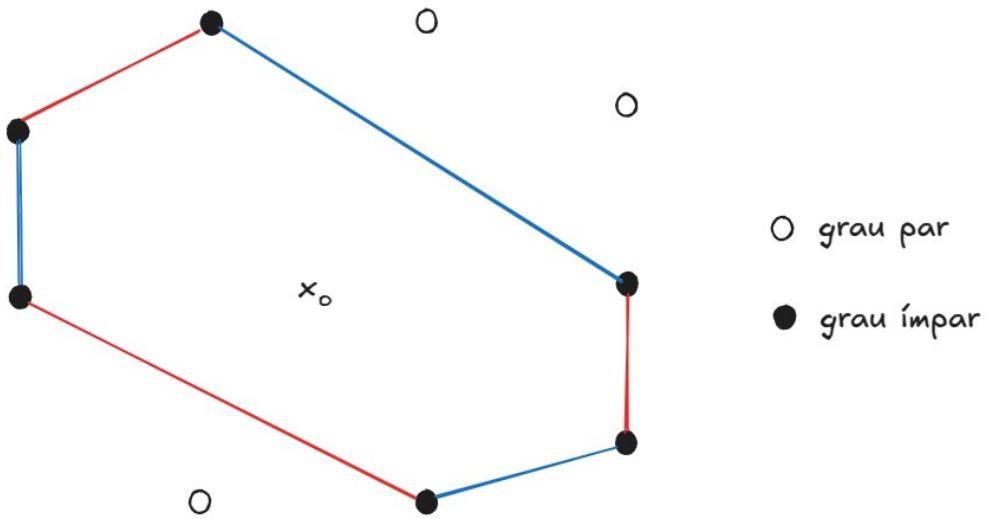
que passa apenas pelos vértices do subconjunto O , denotado por x_o , conforme ilustra a figura a seguir:



O custo do ciclo x_o é menor que o custo da solução ótima, pois ele passa por menos vértices:

$$f(x_o) \leq f(x^*)$$

Porém, note que podemos particionar o ciclo x_o em arestas alternadas de duas cores: vermelhas e azuis. Note que como o número de vértices ímpares é sempre par, o número de arestas no ciclo x_o será sempre par, e assim, temos 2 emparelhamentos distintos com o mesmo número de arestas: M' (azuis) e M'' (vermelhas).



Como o peso total dos 2 emparelhamentos é igual ao custo do ciclo x_o , temos:

$$w(M') + w(M'') = f(x_o) \leq f(x^*)$$

Se os pesos de M' e M'' forem iguais:

$$w(M') \leq \frac{f(x^*)}{2}$$

Se os pesos de M' e M'' forem diferentes, seja $\bar{M} = \min\{M', M''\}$. Então, temos:

$$w(\bar{M}) \leq \frac{f(x^*)}{2}$$

Assim, como no algoritmo de Christofides encontramos o emparelhamento M de custo mínimo entre os vértices de grau ímpar, $M = \bar{M}$ e $w(M) \leq \frac{f(x^*)}{2}$. Portanto o custo da solução final satisfaz:

$$f(x_c) = w(T_{MST}) + w(M)$$

Como $w(T_{MST}) < f(x^*)$ e $w(M) \leq \frac{f(x^*)}{2}$, temos finalmente:

$$f(x_c) \leq f(x^*) + \frac{f(x^*)}{2} = \frac{3}{2}f(x^*)$$

O algoritmo 2-optimal

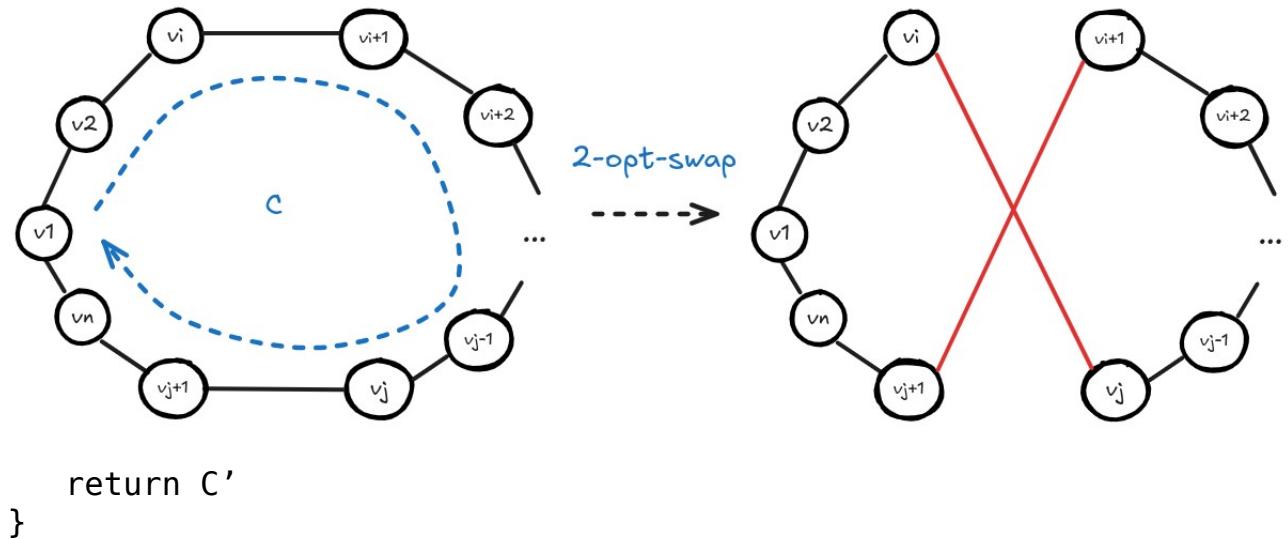
Ideia: partir de uma solução inicial arbitrária e tentar melhorá-la iterativamente a partir de buscas na vizinhança local, através de operações de religações.

Pseudocódigo

Sem perda de generalidade, seja $C = v_1 v_2 v_3 \dots v_{n-1} v_n$ a solução inicial. Definiremos a operação Two_Opt_Swap(C) conforme a seguir:

**Two-Opt-Swap(C , i , j) {
 Seja C_{ij} p ciclo obtido pela religação de v_i e v_j como segue:**

$$C' = v_1 v_2 \dots v_i v_j v_{j-1} v_{j-2} \dots v_{i+1} v_{j+1} \dots v_n v_1$$



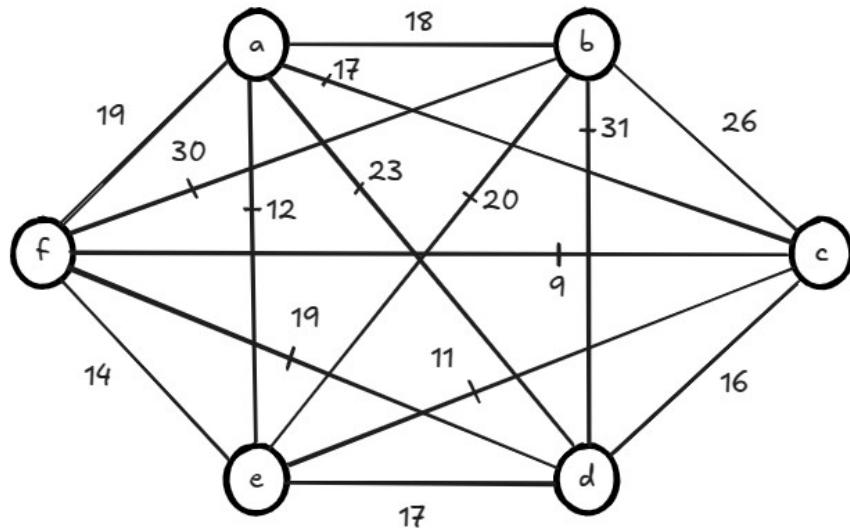
Essa operação é a base para a busca local e define como perturbamos a solução corrente na busca por uma solução melhor que esteja nas proximidades.

Note que em essencia, o que a operação Two-Opt-Swap faz é primeiramente desconectar v_i de v_{i+1} e desconectar v_j de v_{j+1} e em seguida conectar com uma aresta o par de vértices v_i e v_j , além de conectar com outra aresta o par de vértices v_{i+1} e v_{j+1} . A vantagem desta operação é que ela é eficiente, uma vez que pode ser realizada com complexidade constante $O(1)$.

A seguir é apresentado o algoritmo 2-Optimal, também conhecido como 2-Opt, que é um método iterativo para melhorar uma solução inicial dada como entrada.

```
Two-Opt(C, n) {
    best = C
    improved = True
    while improved {
        improved = False
        for i = 1 to n-2 {
            for j = i+2 to n {
                C = Two-Opt-Swap(C, i, j)
                if w(C) < w(best) {
                    best = C
                    improved = True
                }
            }
        }
        C = best
    }
}
```

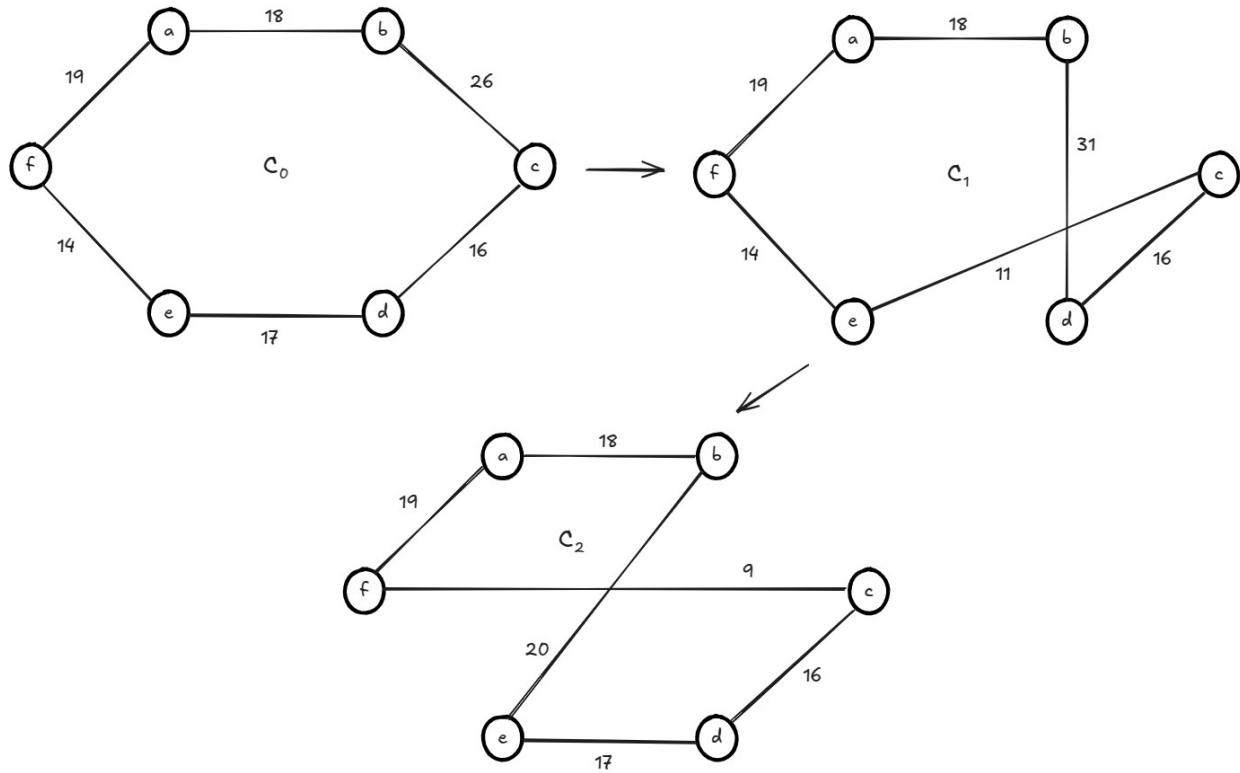
A seguir mostramos um exemplo ilustrativo da aplicação do algoritmo 2-Opt



Consideramos como solução inicial o ciclo $C = a \ b \ c \ d \ e \ f \ a$ que possui custo $w(C) = 110$.

C								W	
		1	2	3	4	5	6		
i	j	a	b	c	d	e	f	a	
1	3	a	c	b	d	e	f	a	124
1	4	a	d	c	b	e	f	a	118
1	5	a	e	d	c	b	f	a	120
1	6	a	f	e	d	c	b	a	110
2	4	a	b	d	c	e	f	a	109 *
1	3	a	d	b	c	e	f	a	124
1	4	a	c	d	b	e	f	a	117
1	5	a	e	c	d	b	f	a	119
1	6	a	f	e	c	d	b	a	109
2	4	a	b	c	d	e	f	a	110
2	5	a	b	e	c	d	f	a	103 *
1	3	a	e	b	c	d	f	a	112
1	4	a	c	e	b	d	f	a	117
1	5	a	d	c	e	b	f	a	119
1	6	a	f	d	c	e	b	a	103
2	4	a	b	c	e	d	f	a	110
2	5	a	b	d	c	e	f	a	109
2	6	a	b	f	d	c	e	a	104
3	5	a	b	e	d	c	f	a	99 *
1	3	a	e	b	d	c	f	a	107
1	4	a	d	e	b	c	f	a	114
1	5	a	c	d	e	b	f	a	119
1	6	a	f	c	d	e	b	a	99
2	4	a	b	d	e	c	f	a	105
2	5	a	b	c	d	e	f	a	110
2	6	a	b	f	c	d	e	a	102
3	5	a	b	e	c	d	f	a	103
3	6	a	b	e	f	c	d	a	100
4	6	a	b	e	d	f	c	a	100

Após toda uma varredura ao redor da solução com custo $w(C) = 99$, não foi possível encontrar nenhuma solução melhor. PARE!



Análise da complexidade

É possível realizar a operação Two_Opt_Swap(C, i, j) com complexidade $O(1)$. Sendo assim, em uma rodada do algoritmo 2-Opt, temos o seguinte número de operações:

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=i+2}^n 1 = (n-2) + (n-3) + (n-4) + \dots + (n-(n-1)) = (n+n+n+\dots+n) - (2+3+4+\dots+(n-1))$$

Note que o primeiro somatório é justamente $n(n-2)$. Já o segundo somatório é igual a:

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

Logo, temos:

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

o que é $O(n^2)$. Portanto, supondo que o loop WHILE mais externo execute K vezes, teremos uma complexidade $O(Kn^2)$. Como $K \ll n$ (K é uma constante muito menor que n), podemos dizer que a complexidade do algoritmo 2-Opt é $O(n^2)$.

Teorema: A solução obtida pelo algoritmo 2-Opt em uma instância do TSP métrico satisfaz:

$$f(x_{2-opt}) \leq (\log n) f(x^*)$$

Iremos omitir a prova deste resultado, mas de qualquer forma, note que a qualidade da solução depende do número de vértices do grafo. Quanto mais vértices o grafo possui, menos garantias de que estamos obtendo soluções próximas à solução ótima.

Uma estratégia comum consiste em utilizar os algoritmos Nearest Neighbor, Twice-Around e Christofides para gerar soluções iniciais para o 2-Opt, na tentativa de melhorá-las ao menos um pouco.

Existem algoritmos exatos para o TSP: um deles é o método da força bruta, que testa todas as combinações possíveis de ciclos de comprimento n e possui complexidade $O(n!)$, sendo inviável para a maioria dos problemas. O algoritmo de Held-Karp utiliza uma abordagem baseada em programação dinâmica para resolver o TSP e possui complexidade $O(2^n n^2)$ (exponencial), o que ainda é inviável para grafos com um grande número de vértices. Por essa razão, em geral, utiliza-se os algoritmos aproximados na solução de problemas reais.

Bibliografia

- CORMEN, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 4^a edição, The MIT Press, 2022.
- T. ROUGHGARDEN. Algorithms Illuminated: The Omnibus edition. Soundlikeyourself Publishing, LLC, 2022.
- W. CELES, R. CERQUEIRA, J. RANGEL. Introdução a Estrutura de Dados, 2^a edição, Elsevier, 2016.
- 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.
- R. THAREJA. Data Structures Using C, 2a edição, Oxford University Press, 2018.
- Y. LANGSAM, M. J. AUGENSTEIN, A. M. TENENBAUM. Data structures using C and C++. 2. ed. Upper Sadle River: Prentice Hall, 1996.
- 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.
- SZWARCFITER, J. L. Teoria Computacional de Grafos: Os algoritmos, Elsevier, 2018.

Sobre o autor

Alexandre L. M. Levada é Bacharel em Ciência da Computação pela Universidade Estadual Paulista (UNESP) em 2002. Concluiu seu Mestrado em Ciência da Computação pela Universidade Federal de São Carlos (UFSCar) em 2006 e seu Doutorado em Física Computacional pela Universidade de São Paulo (USP) em 2010. No mesmo ano, ingressou no Departamento de Computação da Universidade Federal de São Carlos. De 2010 a 2024, foi professor assistente no Departamento de Computação, onde lecionou diversas disciplinas de graduação e pós-graduação como lógica matemática, teoria dos grafos, matemática discreta, processamento digital de imagens, análise de sinais e sistemas, reconhecimento de padrões, visão computacional, algoritmos e estruturas de dados 1 e 2, projeto e análise de algoritmos, programação científica e otimização matemática. Desde 2024 é professor associado do mesmo departamento. Seus principais temas de pesquisa incluem redução de ruído em sinais e imagens, reconhecimento de padrões e aprendizado de máquina, com ênfase em algoritmos de classificação baseados em grafos, métodos de redução de dimensionalidade baseados em teoria da informação e aplicação da geometria da informação na análise da dinâmica de campos aleatórios. É autor de um livro, mais de 25 artigos em periódicos e mais de 55 artigos em conferências. Recebeu o prêmio de melhor artigo científico na categoria Inteligência Artificial, Aprendizado de Máquina e Análise de Padrões por um artigo apresentado na 26^a International Conference on Pattern Recognition (ICPR) em 2022. Recebeu uma bolsa de produtividade em pesquisa do CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) em 2025.

“Se tiver um pão e eu tiver um euro, e eu uso o meu euro para comprar o seu pão, no final da troca eu terei o pão e você o euro. Parece um equilíbrio perfeito, não? A tem um euro, B tem um pão; depois, A tem o pão e B o euro. É uma transação justa, mas meramente material. Agora, imagine que tem um soneto de Verlaine ou conhece o teorema de Pitágoras, e eu não tenho nada. Se me ensinar, no final dessa troca, eu terei aprendido o soneto e o teorema, mas você ainda os terá também. Nesse caso, não há apenas equilíbrio, mas crescimento. No primeiro, trocamos mercadorias. No segundo, compartilhamos conhecimento. E enquanto a mercadoria se consome, a cultura se expande infinitamente.”

Michel Serres, Filósofo Francês
1/09/1930 - 1/06/2019