

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/367001218>

Fundamentos de Análise de Algoritmos e Estruturas de Dados

Preprint · January 2023

DOI: 10.13140/RG.2.2.29738.90567

CITATIONS

0

1 author:



Alexandre L M Levada
Universidade Federal de São Carlos

129 PUBLICATIONS 385 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Double Noise Filtering in CT: Pre- and Post-Reconstruction [View project](#)



Information geometry in random field models [View project](#)



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

Fundamentos de Análise de Algoritmos e Estruturas de Dados

Apostila sobre complexidade de algoritmos, estruturas de dados eficientes
e algoritmos em grafos

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

Sumário

Prefácio.....	3
Complexidade de algoritmos.....	4
Análise de algoritmos de ordenação.....	17
Bubblesort.....	17
Insertionsort.....	21
Selectionsort.....	23
Shellsort.....	25
Quicksort.....	30
Mergesort (ordenação por intercalação).....	35
Heapsort.....	38
Limitante inferior para ordenação baseada em comparações.....	50
Algoritmos de ordenação com complexidade linear.....	52
Estruturas de Dados: Árvores Binárias de Busca.....	59
Estruturas de Dados: Árvores AVL (Balanceadas).....	71
Estruturas de Dados: Árvores de busca digitais (Tries).....	80
Estruturas de Dados: Skip Lists.....	85
Estruturas de Dados: Tabelas de Espalhamento (Hash Tables).....	92
Grafos: Fundamentos básicos.....	111
Grafos: Busca em grafos (BFS e DFS).....	119
Busca em Largura (Breadth-First Search – BFS).....	121
Busca em Profundidade (Depth-First Search – DFS).....	124
Grafos: Caminhos mínimos e o algoritmo de Dijkstra.....	130
Bibliografia.....	143
Sobre o autor.....	144

Prefácio

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

O conteúdo da apostila está organizado como segue: o capítulo 1 apresenta uma introdução ao estudo da complexidade de algoritmos e a notação Big-O. O capítulo 2 descreve análises de complexidade para diversos algoritmos de ordenação no pior caso, caso médio e melhor caso, além de discutir limitantes inferiores para ordenação com comparações e os algoritmos com complexidade linear. O capítulo 4 apresenta as árvores binárias de busca e diversos algoritmos para manipulação de tais estruturas. O capítulo 5 trata das árvores平衡adas (AVL) e algoritmos para rotação. O capítulo 6 introduz as árvores de busca digitais (Tries) que definem a posição dos elementos na estrutura a partir dos bits de suas chaves. O capítulo 7 apresenta as skip lists, que são estruturas de dados probabilísticas eficientes baseadas em uma hierarquia de listas encadeadas. O capítulo 8 trata das tabelas de espalhamento, que são estruturas de dados eficientes por permitirem a busca por uma chave em tempo constante. O capítulo 9 apresenta algoritmos para busca em grafos: a busca em largura e a busca em profundidade. O capítulo 10 discute como encontrar caminhos mínimos em grafos ponderados a partir dos algoritmos de Dijkstra e Bellman-Ford.

Por fim, gostaria de realizar uma menção especial as aulas do Prof. Mário César San Felice do Departamento de Computação da Universidade Federal de São Carlos que serviram como um guia de referência para a elaboração do material. Para os interessados, todas as aulas estão disponíveis na internet no seguinte link: http://www.aloc.ufscar.br/felice/cursos_anteriores.php

Bons estudos!

“Experiência não é o que acontece com um homem; é o que ele 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 realidade, 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.

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

Para isso, assume-se algumas hipóteses:

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

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

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

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

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

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

Notação Big-O

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

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

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

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

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

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

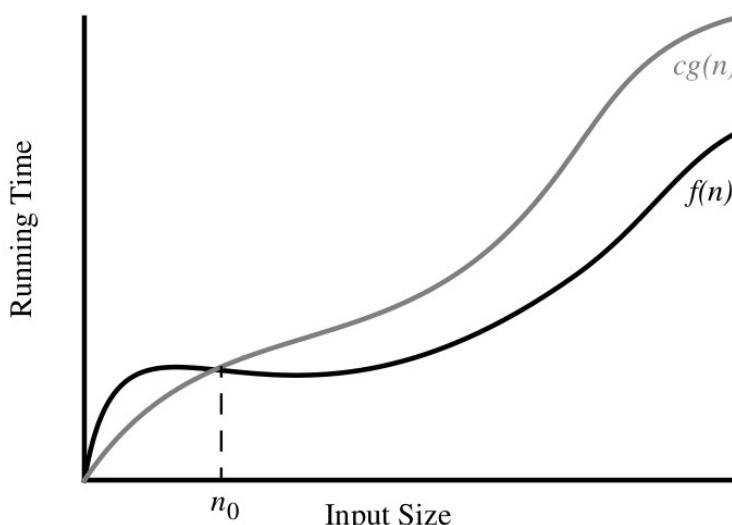
```
sum_of_matrix(M, n) {
    soma = 0
    for i = 1 to n {
        soma_linha = 0
        for j = 1 to n
            soma_linha += M[i, j]
        soma += soma_linha
    }
    return soma
}
```

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

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

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

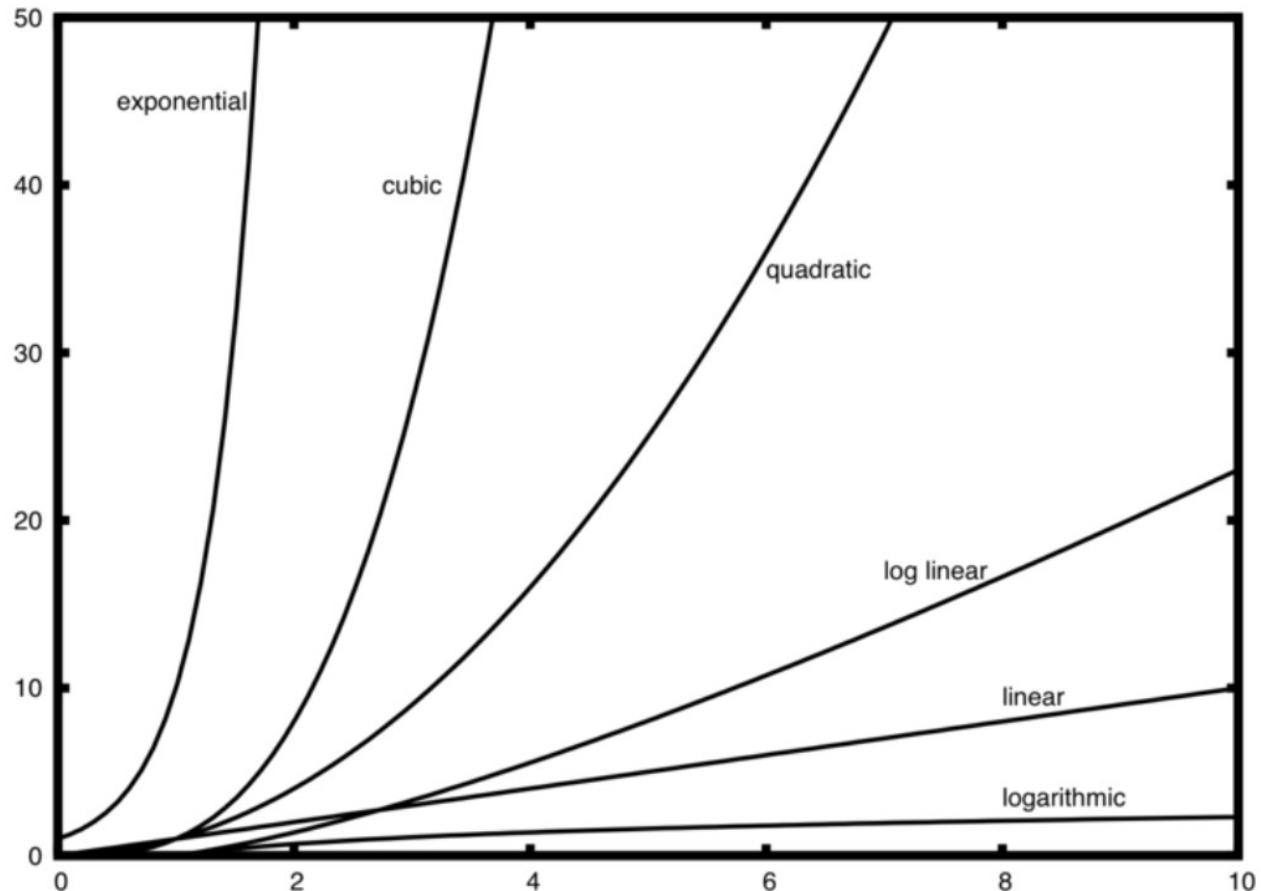
o que implica em dizer que $T(n)$ é $O(n^2)$. Poderíamos ter escolhido a função $f(n)=n^2$, mas o objetivo da notação Big-O é fornecer o limite superior mais justo possível!



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 dessas funções.



Somatários

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

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

Podemos expressá-lo como:

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

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

1) Substituição de variáveis

Seja o seguinte somatório:

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

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

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

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

Dessa forma, podemos expressar o somatório como:

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

2) Distributiva: para toda constante c

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

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

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

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

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

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

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

Prova:

1. Pela propriedade associativa (3), temos:

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

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

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

LI: $k = 1 \rightarrow i = 2$

LS: $k = n \rightarrow i = n+1$

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

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

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

A prova está concluída.

Analisando algoritmos

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

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

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

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

```
Func_B(n) {
    c = 0
    for = 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:

$$T(n) = 1 + 2\lfloor \log_2 n \rfloor , \text{ onde a função piso(x) retorna o maior inteiro menor que } x.$$

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 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 encontrar o menor elemento de um array (vetor). Analise a complexidade de cada uma delas e justifique qual delas é mais eficiente.

```
menor_A(L, n) {
    for i = 1 to n {
        x = L[i]
        menor = zeros(n)
        for j = 1 to n {
            if x <= L[j]:
                menor[j] = 1
        }
        if soma(menor) == n:
            return x
    }
}

menor_B(L, n) {
    pos = 1
    menor = L[pos]
    for i = 1 to n {
        if L[i] < menor {
            pos = i
            menor = L[i]
        }
    }
    return menor
}
```

Primeiramente, vamos analisar o algoritmo menor_A: note que, para cada elemento x da lista L, ele verifica se x é menor ou igual a todos os demais. Ele faz a marcação com o número 1 na posição de x na lista menor. Se x for menor ou igual a todos os elementos de L, teremos exatamente n 1's na lista menor, o que fará com que a soma dos elementos de L seja igual a n.

O pior caso ocorre quando o menor elemento está na última posição de L

Loop mais interno (j) tem n execuções de um comando

Loop mais externo (i) tem n execuções de 2 comandos e do loop mais interno

Inicialização de n é 1 comandos

Assim, temos:

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

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

Agora, vamos analisar o algoritmo menor_B: note que iniciamos o menor elemento como o primeiro elemento da lista L. Então, percorremos a lista verificando se o elemento atual é menor que atual menor. Se ele for, então atualizamos o menor com esse elemento.

O pior caso ocorre quando o menor elemento está na última posição de L.

Loop tem n execuções com 2 instruções

Inicialização de 2 variáveis

Assim, temos:

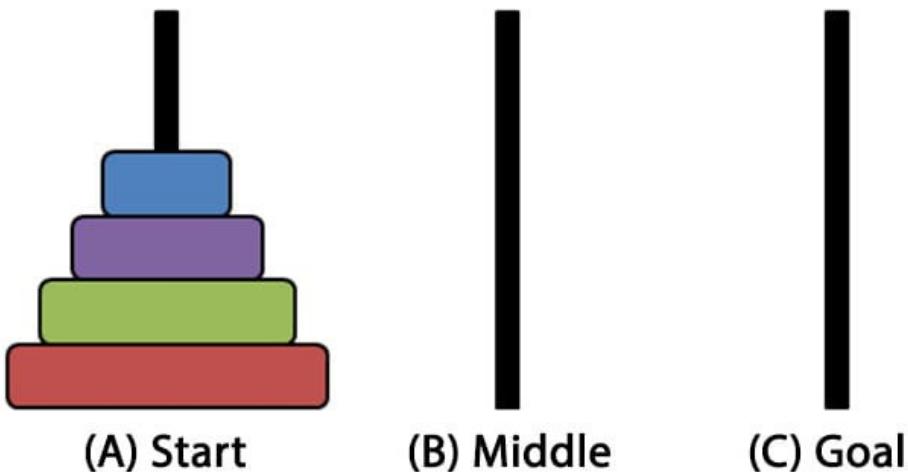
$$T(n) = 2 + 2n$$

o que resulta em $O(n)$.

Portanto, o algoritmo menor_B é mais eficiente que o algoritmo menor_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.

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:

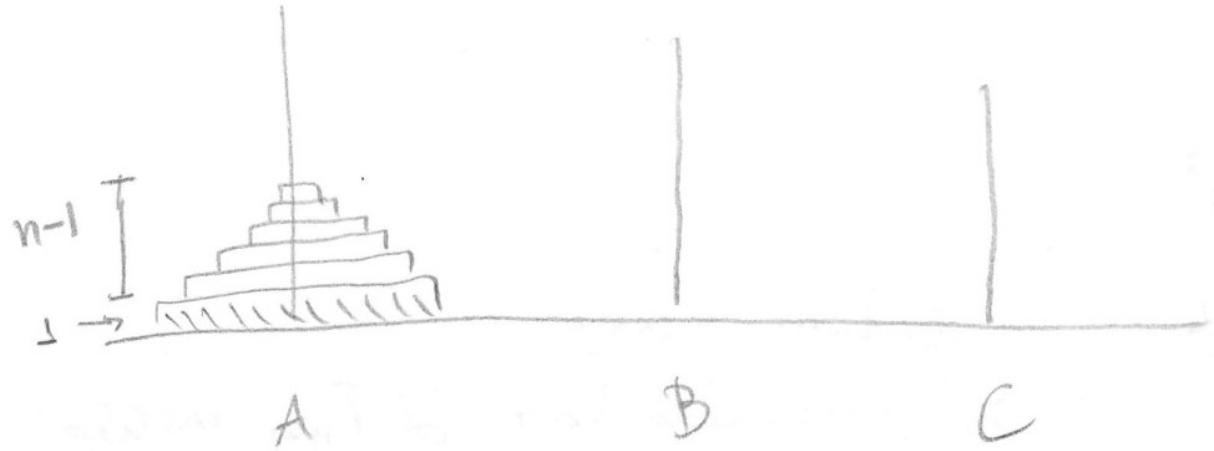
- Move A, B
- Move A, C
- Move B, C
- Move A, B
- Move C, A
- Move C, B
- Move A, B

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

Assim, a recorrência fica definida como:

$$\begin{aligned} T_n &= 2T_{n-1} + 1 \\ T_1 &= 1 \end{aligned}$$

onde T_n denota o número de instruções necessárias para resolvemos o problema das n torres de Hanói. Porém, se quisermos descobrir o número de movimentos para $n = 100$, devemos calcular todos os termos da sequência de 2 até 100.

Pergunta: Como calcular uma função $T(n)$ dada a recorrência?

Como resolver essa recorrência, ou seja, obter uma fórmula fechada? Vamos expandir a recorrência.

$$\begin{aligned} T_1 &= 1 \\ T_2 &= 2T_1 + 1 \\ T_3 &= 2T_2 + 1 \\ T_4 &= 2T_3 + 1 \\ T_5 &= 2T_4 + 1 \\ T_6 &= 2T_5 + 1 \\ \dots & \\ \dots & \\ T_{n-2} &= 2T_{n-3} + 1 \\ T_{n-1} &= 2T_{n-2} + 1 \\ T_n &= 2T_{n-1} + 1 \end{aligned}$$

A ideia consiste em somar tudo do lado esquerdo e somar tudo do lado direito e utilizar a igualdade para chegar em uma expressão fechada. Porém, gostaríamos que a soma fosse telescópica, para simplificar os cálculos. Partindo de baixo para cima, note que para o termo T_{n-1} ser cancelado, a penúltima equação precisa ser multiplicada por 2. Para que o termo T_{n-2} seja cancelado, a antepenúltima equação precisa ser multiplicada por 2^2 . E assim sucessivamente, o que nos leva ao seguinte conjunto de equações:

$$\begin{aligned}
2^{n-1}T_1 &= 2^{n-1} \\
2^{n-2}T_2 &= 2^{n-1}T_1 + 2^{n-2} \\
2^{n-3}T_3 &= 2^{n-2}T_2 + 2^{n-3} \\
2^{n-4}T_4 &= 2^{n-3}T_3 + 2^{n-4} \\
2^{n-5}T_5 &= 2^{n-4}T_4 + 2^{n-5} \\
2^{n-6}T_6 &= 2^{n-5}T_5 + 2^{n-6}
\end{aligned}$$

$$\begin{aligned}
&\dots \\
2^2T_{n-2} &= 2^3T_{n-3} + 2^2 \\
2T_{n-1} &= 2^2T_{n-2} + 2 \\
T_n &= 2T_{n-1} + 1
\end{aligned}$$

Somando todas as linhas, temos uma soma telescópica, pois os mesmos termos aparecem do lado esquerdo e direto das igualdades, o que resulta em:

$$T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + 2^{n-4} + \dots + 2^2 + 2^1 + 2^0$$

Esse somatório pode ser escrito como:

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

Note que $2^{k+1} = 2^k \cdot 2$, o que implica em $2^{k+1} = 2^k + 2^k$, e portanto, $2^k = 2^{k+1} - 2^k$.

O somatório em questão fica definido por uma soma telescópica:

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

Pela definição de somas telescópicas, temos:

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

Portanto, esse é a fórmula fechada para o número de movimentos necessários para resolver a torre de Hanói com n discos. Trata-se de um algoritmo exponencial.

Para se ter uma ideia do custo, se $n = 100$, o número de movimentos a ser executados é

$$1267650600228229401496703205376 = 1.26 \times 10^{30}$$

Supondo que 1 instrução leve aproximadamente 1 nanosegundo $= 10^{-9}$ segundos, temos um tempo total de 1.26×10^{21} segundos. Sabendo que 1 ano é aproximadamente 3.15×10^7 segundos, temos o espantoso número de 4.01×10^{13} anos!

Sabendo que a idade do planeta Terra é estimada em 4.543×10^9 anos, se esse programa tivesse sua execução iniciada no momento da criação do planeta, estaria executando até hoje. valores muito pequenos de n.

Portanto, complexidade exponencial é extremamente ineficiente (praticamente inviável).

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) {
    achou = False
    i = 1
    while i <= n and not achou {
        if L[i] == x {
            achou = True
            pos = i
        }
        else
            i = i + 1
    }
    if achou
        return pos
    else
        return achou
}
```

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 e uma vez as duas instruções para atualizar os valores de $achou$ e pos .

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

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 binary_search(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?

Quantos acessos devemos realizar para que descartemos todo o vetor? Devemos ter $n / 2^m = 1$, o que significa ter $n = 2^m$, o que implica em $m = \log_2 n$, ou seja, temos um custo $O(\log_2 n)$ o que é bem menor do que n quando n cresce muito, pois 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 apenas $\log_2 1024 = 10$ acessos, o que corresponde a aproximadamente 1% do necessário na busca sequencial! É uma ganho muito grande. Porém, na busca binária precisamos gastar um tempo para ordenar a lista! Para isso precisaremos de algoritmos de ordenação, o que é o assunto da nossa próxima aula.

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

Análise de 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, como Bubblesort, Selectionsort, Insertionsort, Shellsort, Quicksort e Mergesort. Além disso, analisaremos a complexidade de tais algoritmos para entender porque e quando devemos utilizá-los.

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 uma implementação em Python 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 n {
            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 amarelo, 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 zero a i, sendo que na primeira vez $i = n - 1$, na segunda vez $i = n - 2$ e até $i = 0$. 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 $O(n^2)$.

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 $n - 1$.

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

Sabemos que o somatório $1 + 2 + 3 + \dots + k - 1 = k(k-1)/2$, 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 (*):

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

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

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

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

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

```
BubbleSort_v2(L, n) {
    # Percorre cada elemento do array L
    for i = n-1 downto 1 {
        s = 0
        # Flutua o maior elemento para a posição mais a direita
        for j = 1 to n {
            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.

Insertionsort

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

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

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

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

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

Suponha o seguinte vetor de entrada.

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

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

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

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

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

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

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

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

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

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

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

Análise da complexidade

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

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

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

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

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

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

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

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

O valor do primeiro somatório é:

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

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

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

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

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

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

A soma entre colchetes é exatamente aquela calculada no pior caso, ou seja, temos:

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

É fácil notar que o segundo somatório resulta em $n - 1$. 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:

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

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:

```
for i = 1 to n      →      i = 1
    print(i)          while i <= n {
                           print(i)
                           i = i + 1
                       }
```

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

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

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)

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

2^a iteração

$h = 2$ (divisão inteira de 10 por 4)

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

3^a iteração

$h = 1$

$[-3, 1, 4, 2, 7, 5, 10, 6, 15, 13] \rightarrow [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]$ (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
    }
}
    
```

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 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.

3. Sequência de Knuth: 1, 4, 13, ..., $(3^k - 1) / 2$ (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.

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

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 conhecidos na matemática como números 3-suaves, definidos como os inteiros cujos fatores primos são todos menores ou iguais que 3.

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

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.

$$\begin{cases} 9 \left(2^k - 2^{\frac{k}{2}} \right) + 1 & k \text{ even, (termo par)} \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd } \quad (\text{termo ímpar}) \end{cases}$$

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

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
}
```

Dessa forma, o número de iterações do FOR na primeira iteração será $\left(n - \frac{n}{2}\right)$, na segunda iteração será $\left(n - \frac{n}{4}\right)$, na terceira iteração será $\left(n - \frac{n}{8}\right)$ e assim sucessivamente, de modo que no total, o número de instruções executadas será:

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

Mas no fim do processo temos que $\frac{n}{2^k} = 1$ que é o tamanho final do gap. Assim, temos que $n = 2^k$, o que após a aplicação do logaritmo na base 2 em ambos os lados, nos leva a $k = \log_2 n$

Note que a expressão anterior é equivalente a:

$$T(n) = \frac{1}{2}n + \frac{3}{4}n + \frac{7}{8}n + \frac{15}{16}n + \dots + \frac{2^k - 1}{2^k}n = n \left(\frac{1}{2} + \frac{3}{4} + \frac{7}{8} + \frac{15}{16} + \dots + \frac{2^k - 1}{2^k} \right)$$

A somatória entre parêntesis pode ser expressa por:

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

O primeiro somatório é trivial ver que resulta em k. Já o segundo, é a soma dos k primeiros termos de uma progressão geométrica cujo primeiro termo é 1/2 e cuja razão é 1/2. Utilizando a fórmula da soma da P.G., temos:

$$S_k = \frac{\frac{1}{2} \left[1 - \left(\frac{1}{2} \right)^k \right]}{\left(1 - \frac{1}{2} \right)} = 1 - \left(\frac{1}{2} \right)^k = 1 - 2^{-k}$$

Sendo assim, T_n pode ser expresso por:

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

Mas sabemos que $k = \log_2 n$, o que nos leva a:

$$T(n) = n(\log_2 n - 1 + 2^{-\log_2 n}) = n \left(\log_2 n - 1 + 2^{\log_2 \frac{1}{n}} \right) = n \log_2 n - n + 1$$

Portanto, temos que a complexidade 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 $T(n)$ como:

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

Aplicando a distributiva, temos:

$$T(n) = (2n - n) + (3n - n) + (8n - n) + \dots + (2^k n - n) = n(1 + 3 + 7 + 15 + \dots + (2^k - 1))$$

A soma entre parêntesis pode ser expressa como:

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

É fácil ver que o segundo somatório em A vale k. O primeiro somatório em A é a soma dos k primeiros termos de uma P.G. cujo primeiro termo vale 2 e a razão também vale 2:

$$S_k = \frac{a_1(q^k - 1)}{q - 1} = \frac{2(2^k - 1)}{2 - 1} = 2(2^k - 1)$$

Mas sabemos que $k = \log_2 n$, o que nos leva a:

$$S_k = 2(2^k - 1) = 2(n - 1)$$

Portanto, temos que $A = 2(n - 1) - \log_2 n$, o que nos leva a:

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

resultando em uma complexidade $O(n^2)$.

A complexidade de pior caso do Shellsort é variável, pois depende fortemente da sequência de gaps. Pode-se mostrar que no pior caso, o algoritmo Shellsort necessita de pelo menos $O(n(\log n)^2)$ operações¹.

Caso médio: utilizando uma abordagem similar ao pior caso, ao invés de realizar todas as trocas, realiza metade delas.

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

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).

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 $O(n \log^2 n)$. Já para a sequência de Sedgwick, 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 $O(n^{1.25})$ e $O(n \log^2 n)$.

¹ BJORN POONEN. THE WORST CASE IN SHELLSORT AND RELATED ALGORITHMS, 1992.

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.

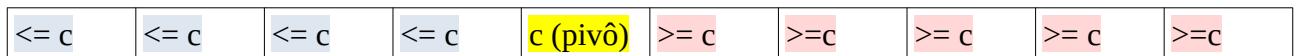
Quicksort

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] \rightarrow 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.

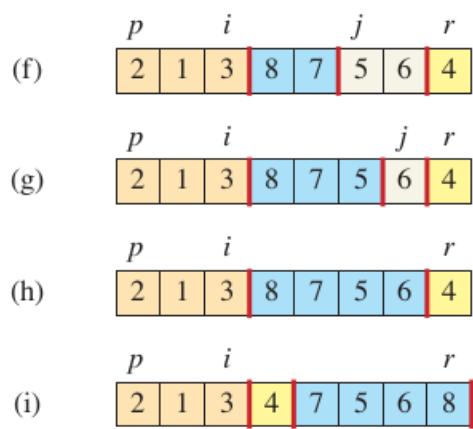
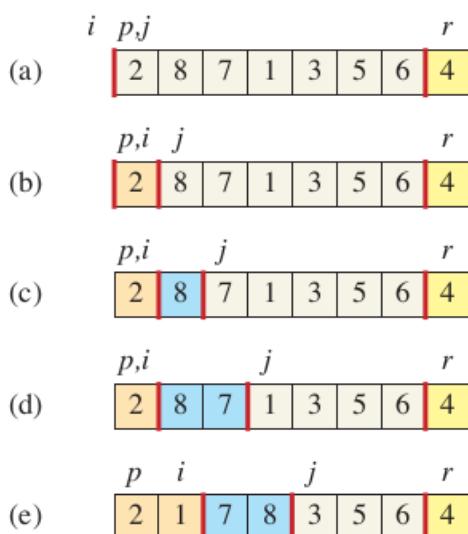
```
QuickSort(L, low, high) {
    if low < high {
        q = partition(L, low, high)
        QuickSort(L, low, q-1)
        QuickSort(L, q+1, high)
    }
}
```

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], L[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)

Note que:

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

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

$$T(n - 2) = n - 2 + T(n - 3)$$

$$T(n - 3) = n - 3 + T(n - 4)$$

...

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

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

onde $T(0) = 0$.

Somando todas as parcelas, temos o somatório $(1 + 2 + 3 + \dots + n - 1 + n)$, 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.

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

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

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

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

...

Voltando com as substituições, temos:

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

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 constante (pois não depende de n), temos finalmente que a complexidade do 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 (II) – (I):

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

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:

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

Sendo assim, podemos escrever:

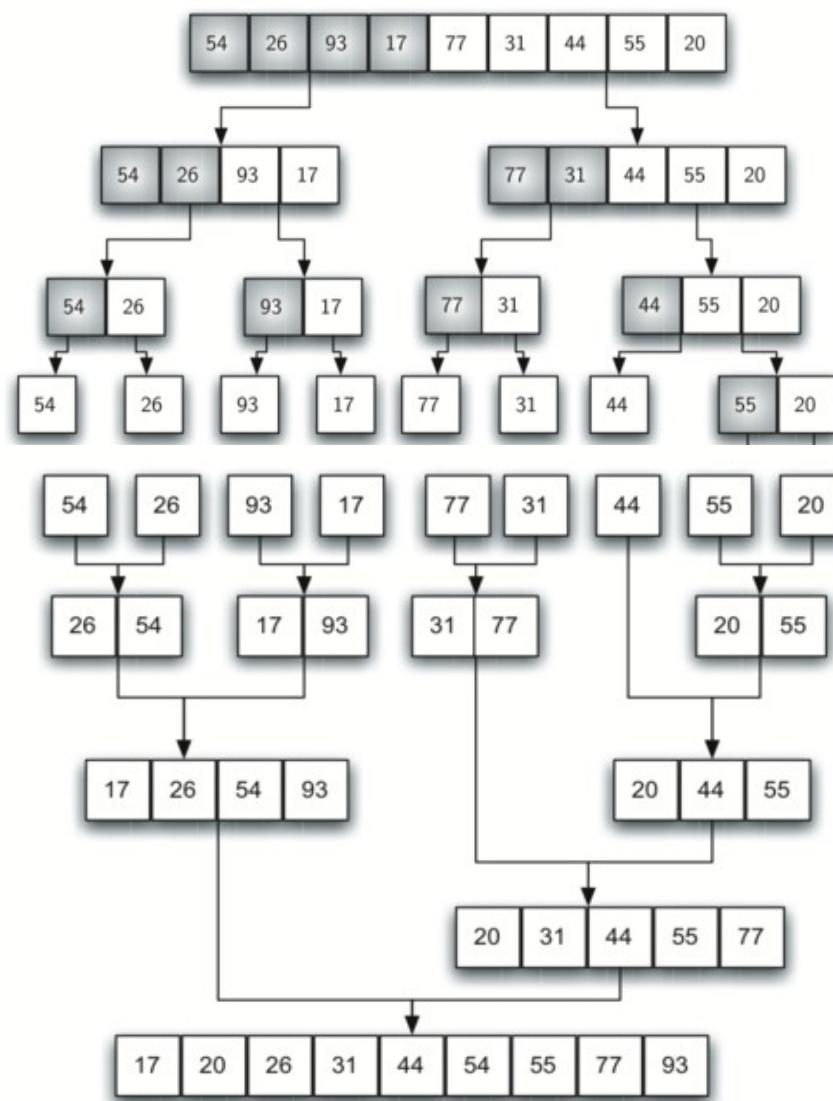
$$T(n)=(n+1)(O(1)+2cO(\log n))=(n+1)+2cnO(\log n)+2cO(\log n)$$

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.

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]

meio = 10 // 2 = 5

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

meio = 5 // 2 = 2

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

meio = 2 // 2 = 1 ou meio = 3 // 2 = 1

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

meio = 2 // 2 = 1

Parte 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 MergeSort, que utiliza uma abordagem recursiva.

```
MergeSort(L) {
    n = len(L)
    if n > 1 {
        meio = n//2
        LE = L[:meio]    # Lista Esquerda
        LD = L[meio:]   # Lista Direita
        # Aplica recursivamente nas sublistas
        MergeSort(LE)
        MergeSort(LD)
        # Quando volta da recursão inicia aqui!
        i, j, k = 0, 0, 0
        # Faz a intercalação das duas listas (merge)
        while i < len(LE) and j < len(LD) {
            if LE[i] < LD[j] {
                L[k] = LE[i]
                i += 1
            }
            else {
```

```

        L[k] = LD[j]
        j += 1
    }
    k += 1
}
while i < len(LE) {
    L[k] = LE[i]
    i += 1
    k += 1
}
while j < len(LD) {
    L[k] = LD[j]
    j += 1
    k += 1
}
}
}

```

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 extrair a 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:

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

Heapsort

Trata-se de um algoritmo criado por John Williams em 1964 e que possui complexidade $O(n \log n)$ tanto no melhor, quanto no pior e no médio casos. Mesmo tendo a mesma complexidade no caso médio que o QuickSort, o HeapSort acaba sendo um pouco mais lento que algumas boas implementações do QuickSort. Porém, além de ser mais rápido no pior caso que o QuickSort, necessita de menos memória para executar.

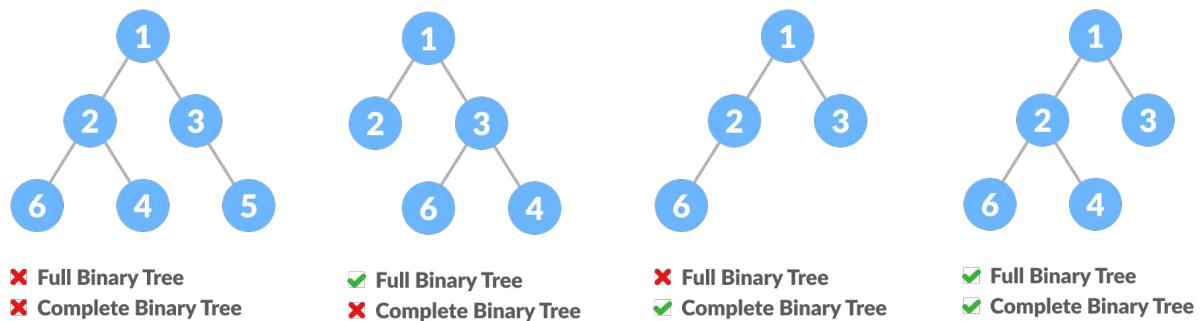
Relação entre arrays e árvores binárias

Uma árvore binária completa é uma árvore binária na qual todos os níveis são completamente preenchidos, exceto possivelmente o mais baixo, que é preenchido a partir

da esquerda. Uma árvore binária completa é como uma árvore binária cheia, mas com duas diferenças principais

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

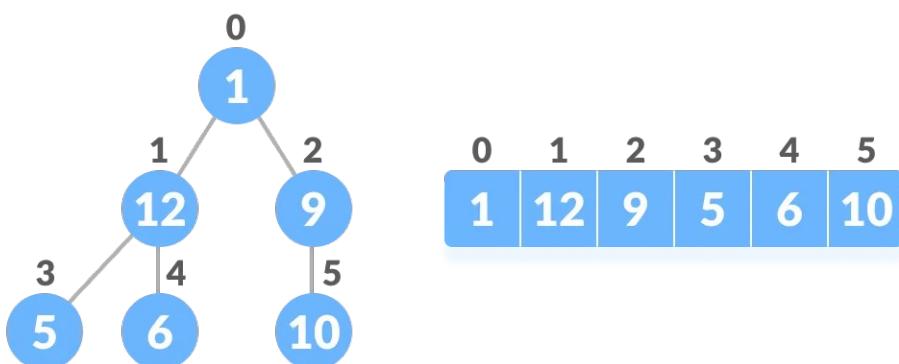
Vejamos os exemplos a seguir.



Note que para uma árvore binária ser cheia, não é necessário que todas as folhas estejam no mesmo nível, bastando que todo nó interno tenha exatamente dois filhos. Uma árvore binária perfeita é um tipo de árvore binária em que cada nó interno tem exatamente dois nós filhos e todos os nós folha estão no mesmo nível.

Uma árvore binária completa possui a seguinte propriedade interessante que pode ser utilizada para encontrar os filhos e o pai de qualquer nó quando os elementos são dispostos em um vetor. Dado o elemento i do vetor, então o elemento de índice $2i + 1$ será seu filho a esquerda e o elemento de índice $2i + 2$ será o seu filho a direita. Além disso, o pai do elemento de índice i será o elemento de índice $(i - 1)/2$, onde $//$ representa a divisão inteira.

A figura a seguir ilustra essa interessante relação. Conhecer essa relação é fundamental para entendermos o funcionamento do algoritmo Heapsort.

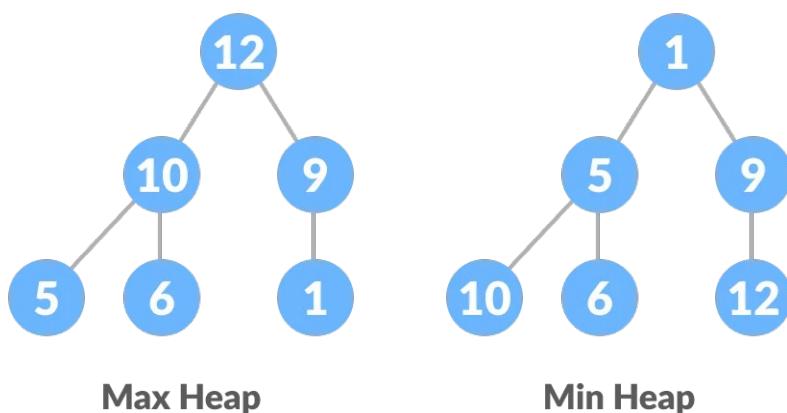


A estrutura de dados Heap

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

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

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



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

Como heapificar uma árvore?

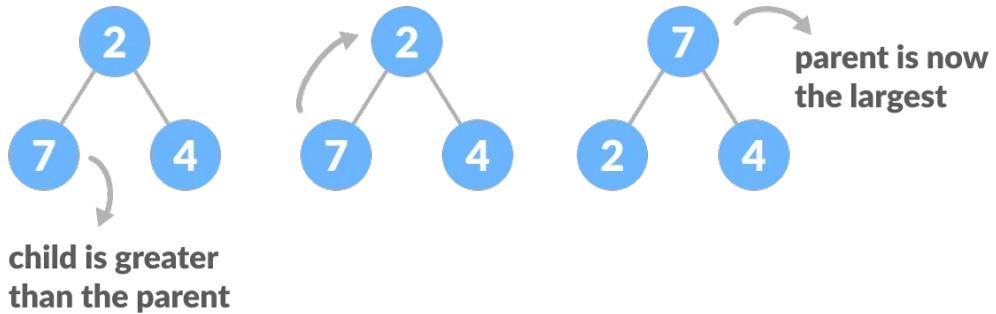
A partir de uma árvore binária completa, podemos modificá-la para se tornar um max-heap executando uma função chamada `heapify` em todos os elementos não-folha do heap. Para isso assume-se que temos a representação da árvore na forma vetorial. Como a função `heapify` usa recursão, pode ser um pouco complicada de entender. Então, vamos primeiro pensar em um exemplo didático de como você empilharia uma árvore com apenas três elementos.

O exemplo a seguir mostra dois cenários - um em que a raiz é o maior elemento e não precisamos fazer nada. Este é o caso base da recursão, onde atingimos a condição de parada. E outro em que a raiz tem um elemento maior que o filho e precisamos trocá-los para manter o max-heap válido.

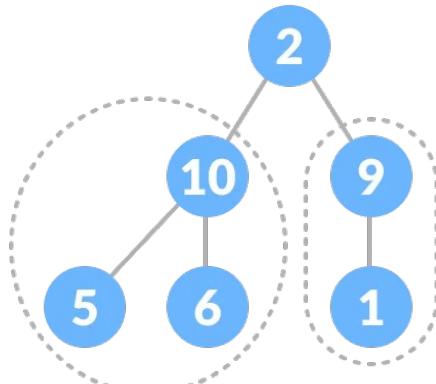
Scenario-1



Scenario-2

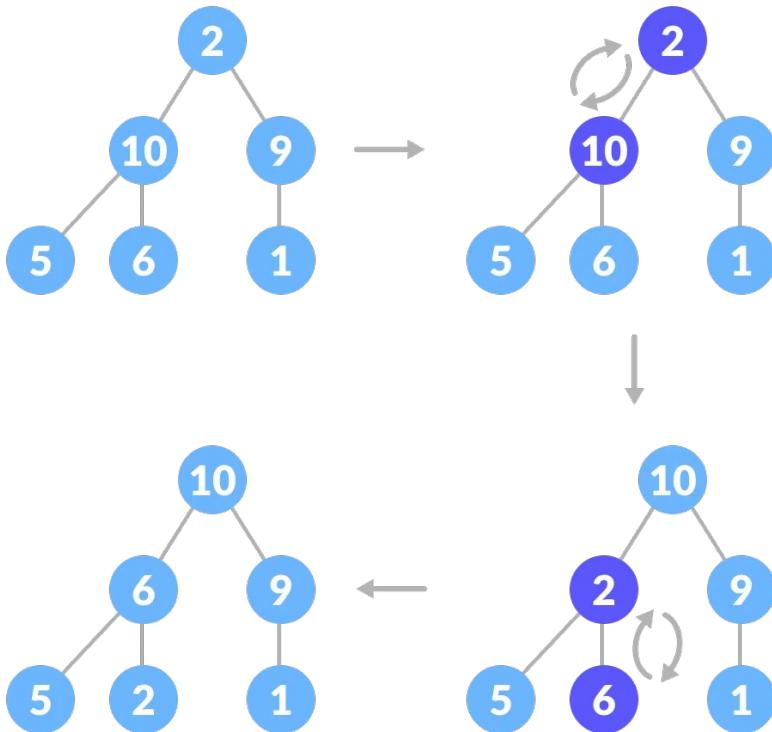


Agora vamos pensar em outro cenário em que há mais de um nível no heap. Note que a árvore a seguir não é um max-heap por causa da raiz, mas todas as subárvore (a esquerda e a direita) são max-heaps.



**both subtrees of the root
are already 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 s 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 em Python a seguir ilustra o código.

```
heapify(L, n, i):
    # Encontra o maior entre o nó raiz i e os filhos
    maior = 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 raz i não é maior, troca e continua heapify
    if largest != i {
        swap(L[i], L[largest])
        heapify(L, n, largest)
    }
}
```

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

Construindo um heap

Para construir um max-heap a partir de uma árvore qualquer, podemos começar a empilhar cada subárvore de baixo para cima e terminar com um max-heap depois que a função for aplicada a todos os elementos, incluindo o elemento raiz. No caso de uma árvore binária completa, o primeiro índice de um nó não-folha é dado por $n/2 - 1$ (divisão inteira). Todos os outros nós depois disso são folhas da árvore e, portanto, não precisam ser heapificados. O código em Python a seguir ilustra o processo.

```
# Constói um max-heap
for i = n//2-1 downto 0
    heapify(L, n, i)
```

Vamos supor que inicialmente temos o seguinte vetor:

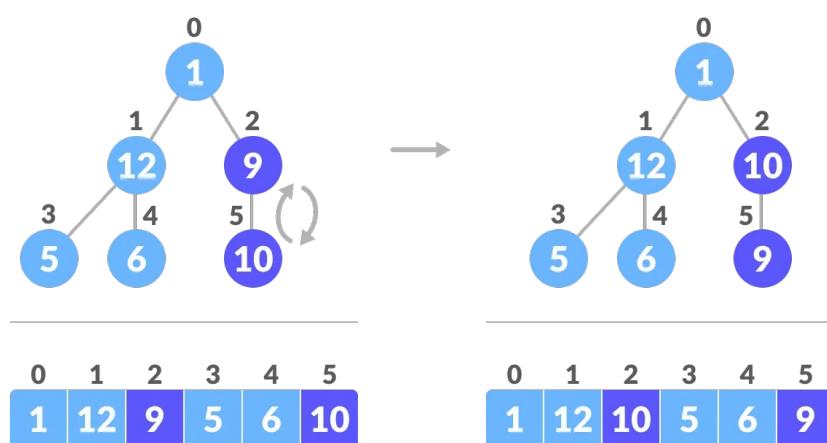
	0	1	2	3	4	5
arr	1	12	9	5	6	10

n = 6

i = 6/2 - 1 = 2 # loop runs from 2 to 0

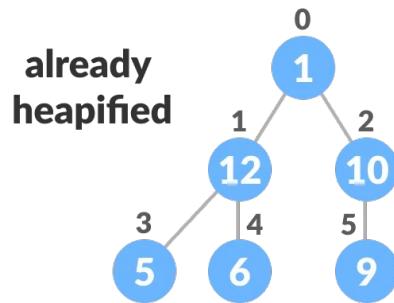
Como podemos ver, na construção do max-heap basta percorrer os elementos de índices 2, 1 e 0, nessa ordem. Primeiramente, teremos a chamada da função `heapify` para $i = 2$.

i = 2 → heapify(arr, 6, 2)



Como os filhos do nó $i=2$, são os elementos com índices 5 e 6, mas 6 não é menor que $n=6$, a função deve encontrar o maior elemento entre 9 e 10, e em seguida trocá-los de posição. A segunda iteração da construção do heap é para o nó com índice 1, ou seja o que possui valor 12. Note que esse nó já está heapificado e não há mais nada a fazer.

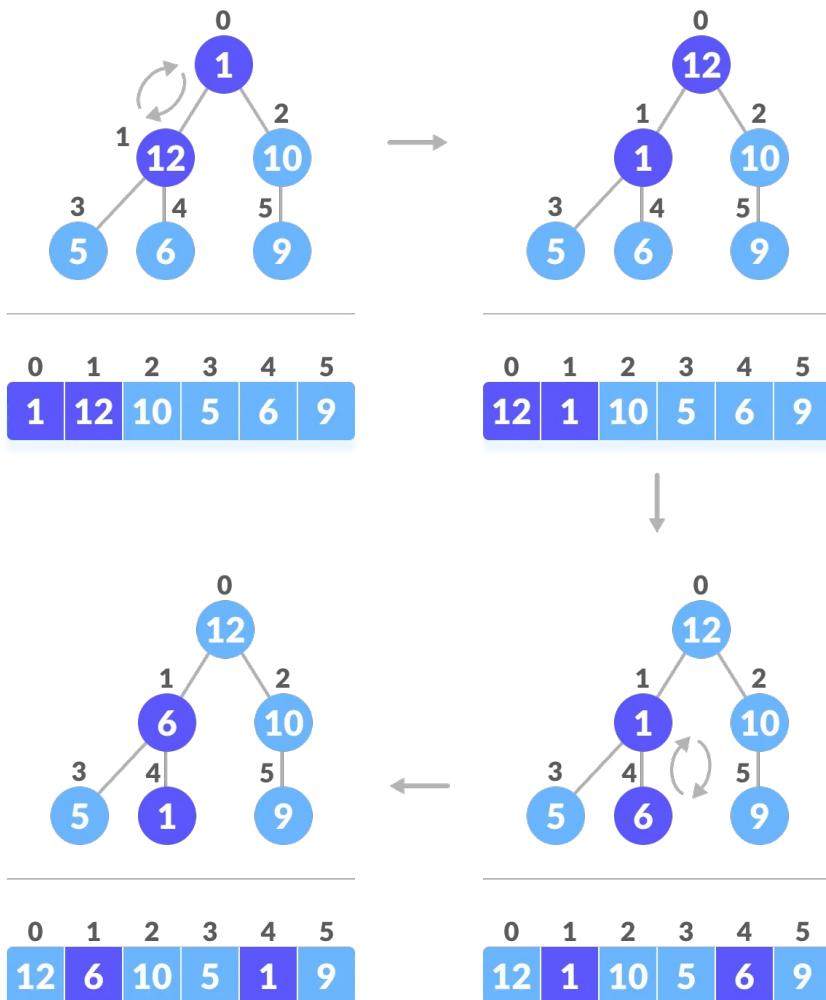
$i = 1 \rightarrow \text{heapify}(\text{arr}, 6, 1)$



0	1	2	3	4	5
1	12	10	5	6	9

Por fim, a última iteração da construção do heap refere-se ao nó de índice 0, ou seja, o nó com o valor 1.

$i = 0 \rightarrow \text{heapify}(\text{arr}, 6, 0)$

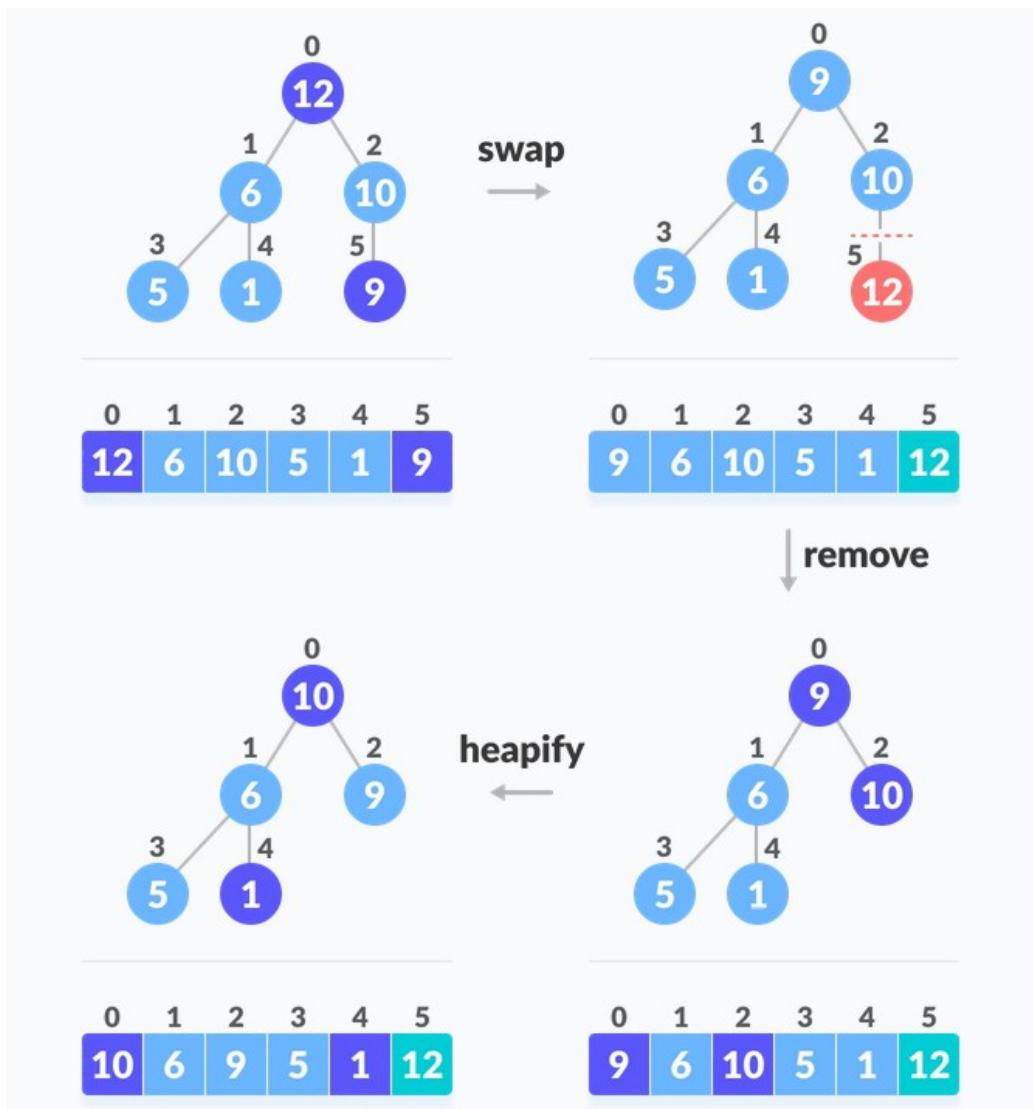


Note que primeiro devemos trocar o elemento 1 com o maior de seus 2 filhos (nós de índices 1 e 2), que no caso é o elemento 12. Em seguida, a função é chamada novamente, agora para o elemento 1. Os dois filhos desse elemento estão nos índices 3 e 4 e possuem valor 5 e 6, respectivamente. No caso o elemento 1 deverá ser trocado com o elemento 6. Como pôde ser visto, durante a construção do max-heap, iniciamos com a heapificação das menores subárvores e vamos gradualmente subindo até atingir a raiz.

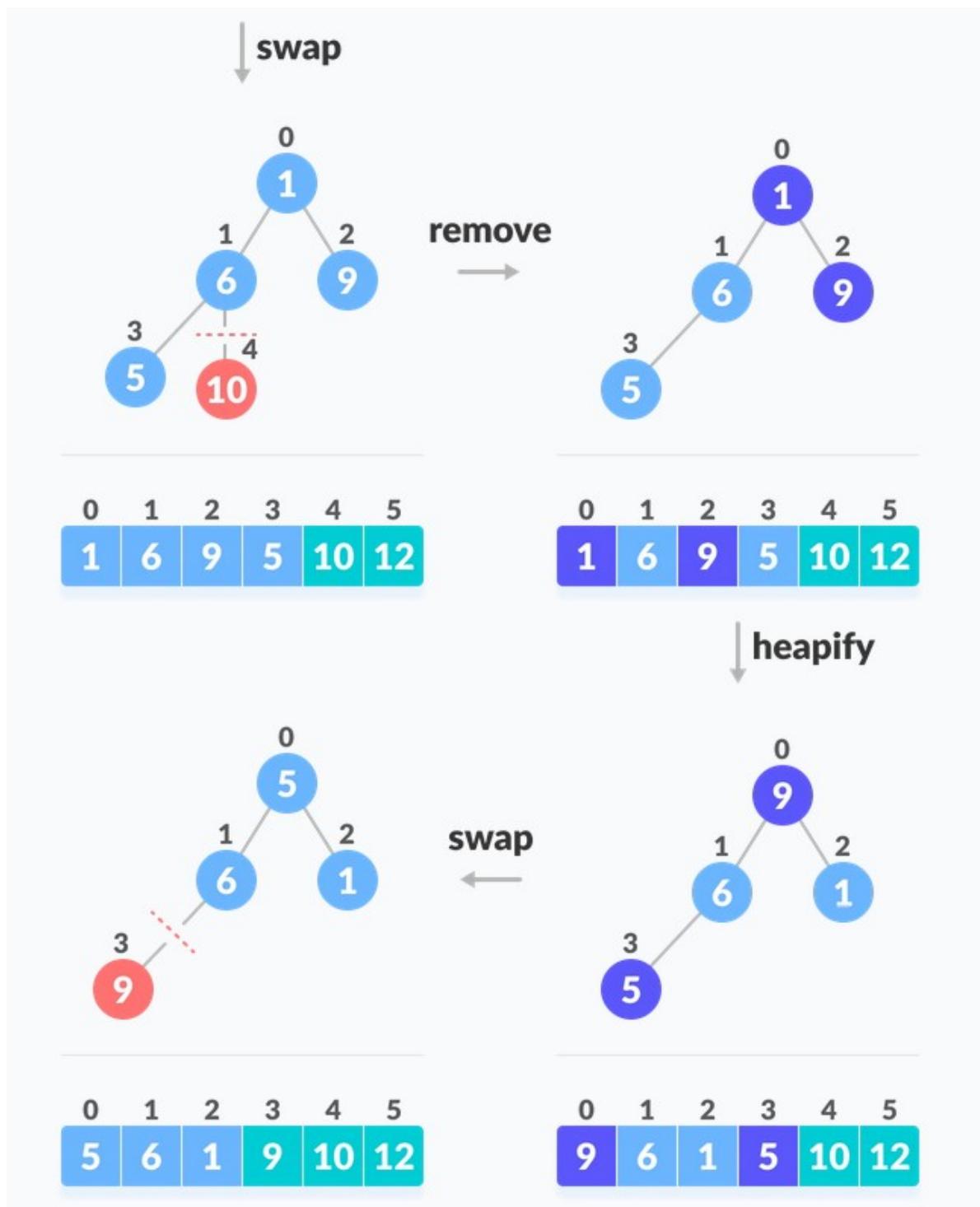
O algoritmo Hepasort

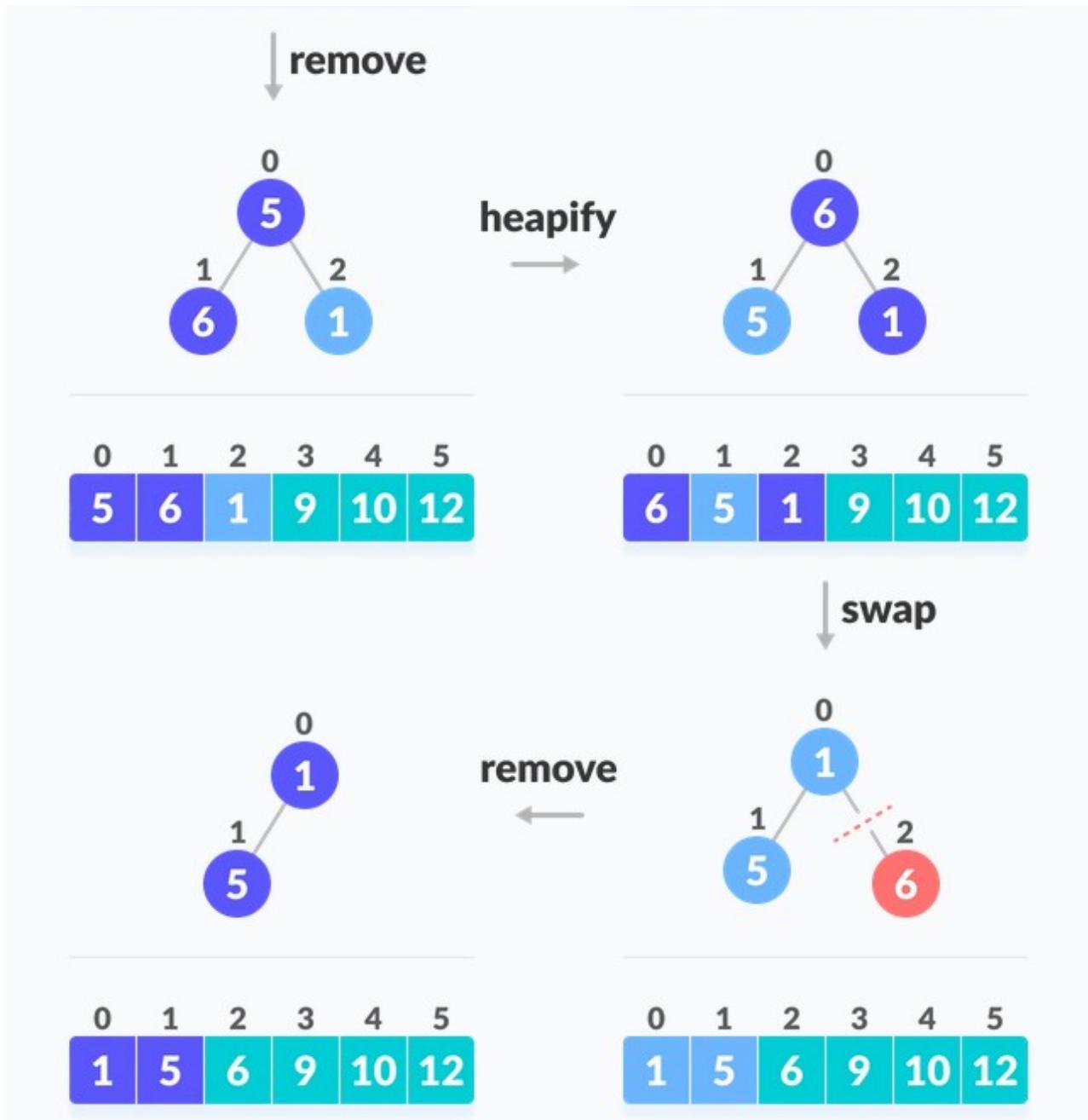
O algoritmo Heapsort é baseado nos seguintes passos:

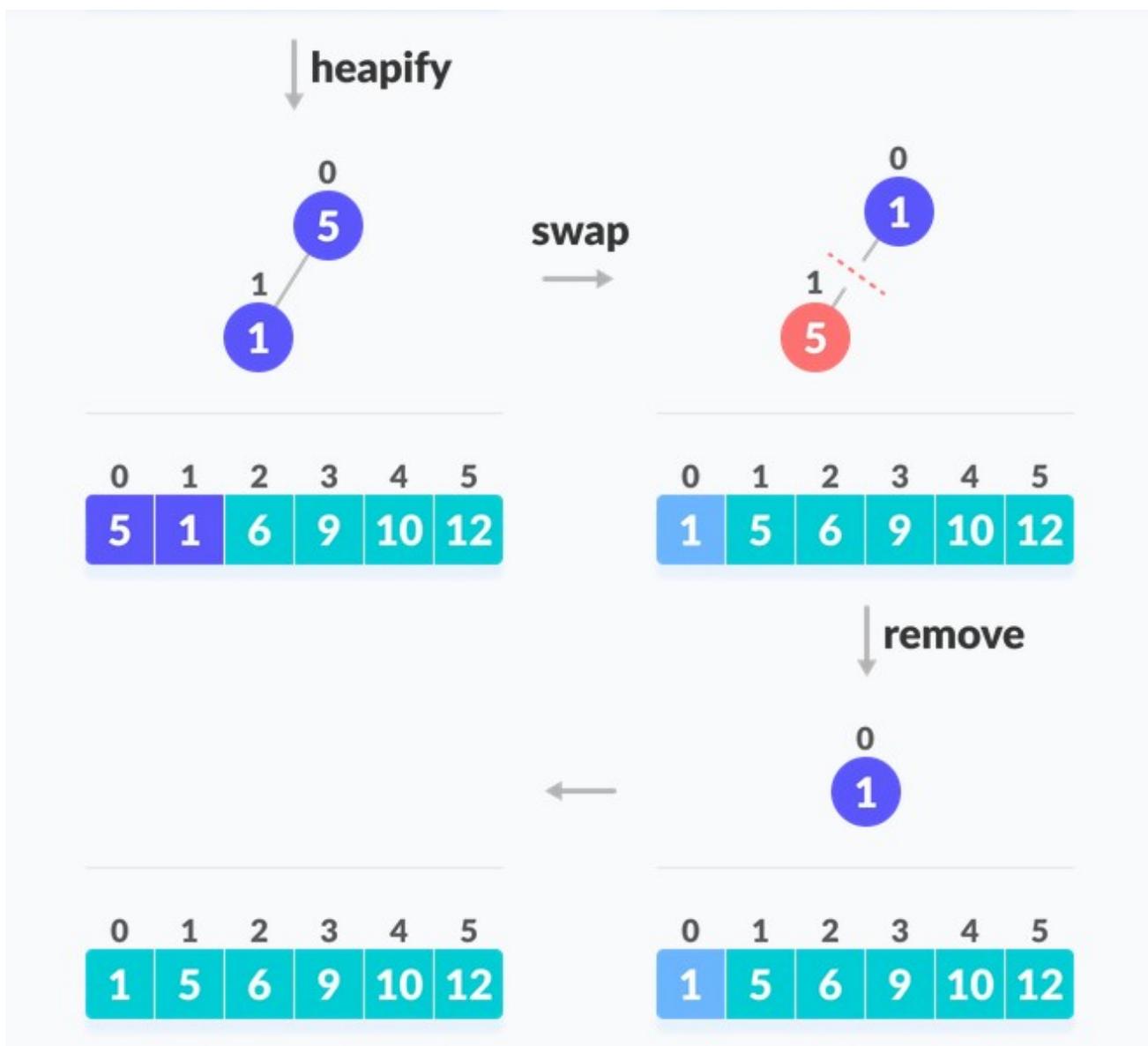
1. Como a árvore em questão, na forma vetorial, satisfaz a propriedade max-heap, o maior elemento está sempre localizado na raiz (índice 0).
2. **Swap:** Coloque o elemento raiz e na última posição do vetor, trocando sua posição com o último elemento.
3. **Remove:** Reduza o tamanho do max-heap em uma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



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.







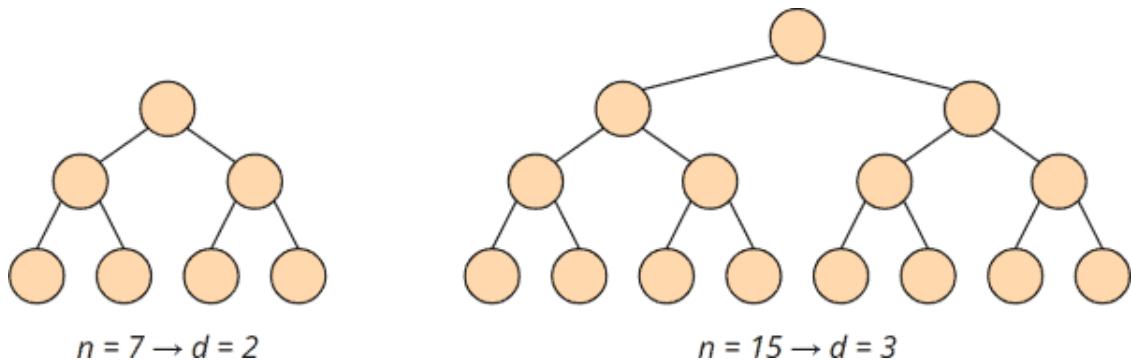
A função a seguir ilustra o algoritmo Heapsort, que faz uso das funções auxiliares descritas anteriormente.

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

A seguir iremos analisar a complexidade do Heapsort. Note que para isso devemos essencialmente analisar a complexidade da função `heapify()`.

Análise da complexidade do Heapsort

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.



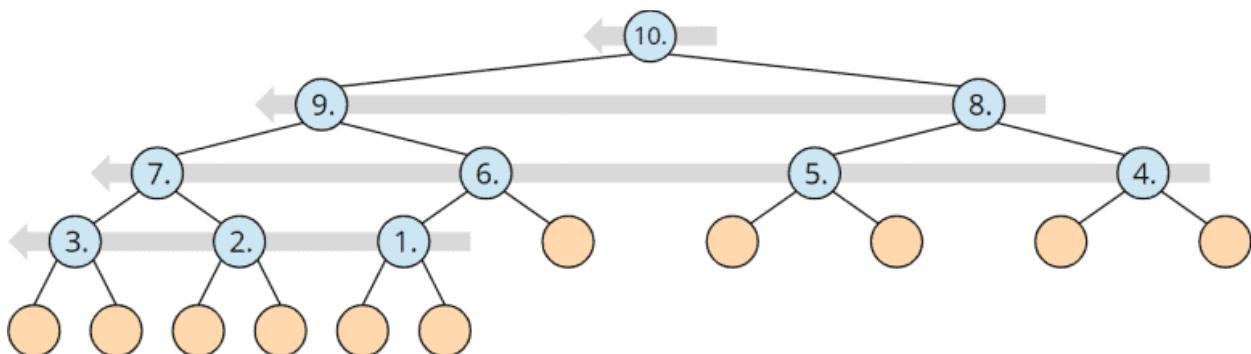
Assim, no caso de uma árvore binária de altura genérica k , o número máximo de elementos armazenados é:

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

No pior caso, a função `heapify()` basicamente desce o elemento da raiz até uma das folhas 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()` 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.

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.

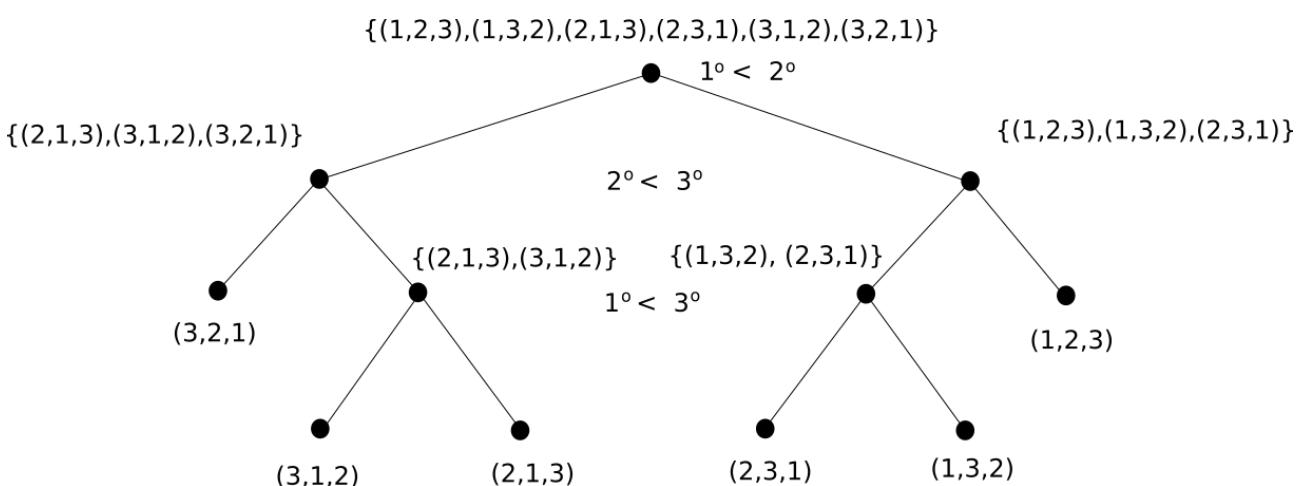
Para obter um limitante inferior para o número de trocas, vamos comparar duas grandezas:

- í) 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 pombos, se temos $n+1$ pombos para serem colocados em n casas, uma casa deverá conter 2 ou mais pombos. 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 factorial, 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:

$$n! \geq \left[\frac{n}{2} \times \frac{n}{2} \times \frac{n}{2} \times \dots \times \frac{n}{2} \right] \times [1 \times 1 \times 1 \dots \times 1]$$

$n/2$ termos $n/2$ termos

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).

Algoritmos de ordenação com complexidade linear

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
- 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.

$$L = [4, 2, 2, 8, 3, 3, 1] \quad \text{max} = 8$$

Passo 2. Criar um vetor de tamanho ($\text{max} + 1$) 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.

$$\begin{aligned} L &= [4, 2, 2, 8, 3, 3, 1] \\ &\quad \underbrace{\hspace{10em}}_{\text{C}} \\ C &= [0, 1, 3, 5, 6, 6, 6, 6, 7] \quad 6 - 1 = 5 \\ &\quad \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix} \\ S &= [1, 2, 2, 3, 3, 6, 8] \\ &\quad \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \end{aligned}$$

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

A seguir apresentamos o algoritmo Countingsort em pseudo-código para que seja possível analisarmos sua complexidade.

```
Countingsort(L, n) {
    m = max(L)
    count = zeros(m+1)
    for j = 0 to n-1
        count[L[j]] += 1
    for i = 1 to m
        count[i] += count[i-1]           # usa o mesmo vetor para H
    output = zeros(n)
    for i = 0 to n-1 {
        ind = count[L[i]-1]
        output[ind] = L[i]
        count[L[i]] = count[L[i]] - 1
    }
    return output
}
```

Análise da complexidade

Note que o primeiro loop FOR possui complexidade $O(n)$, o segundo loop FOR possui complexidade $O(m)$ e o terceiro loop FOR possui complexidade $O(n)$. Sendo assim, a complexidade do algoritmo Countingsort é:

$$O(n) + O(m) + O(n) = O(n + m)$$

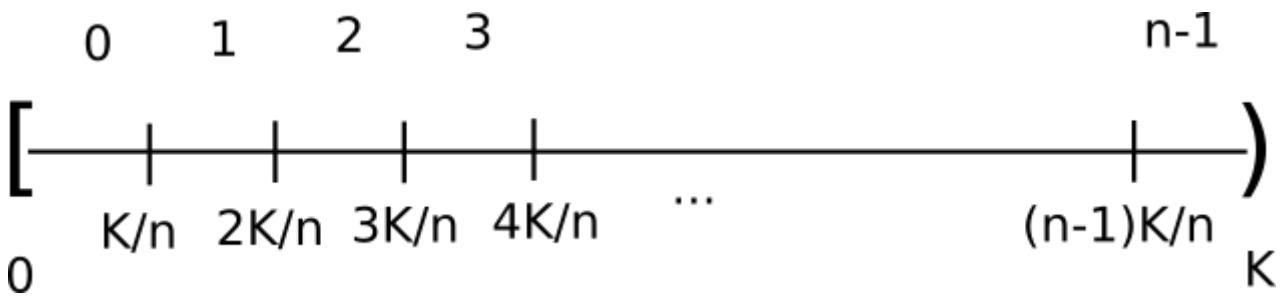
Em geral, o algoritmo funciona muito bem quando o maior elemento da lista não é tão grande (pois isso reduz m). Por exemplo, quando $m \ll n$, como n domina m, a complexidade se torna $O(n)$.

Porém, esse algoritmo tem limitações. A principal delas é quando o maior elemento é muito grande: 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!

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.

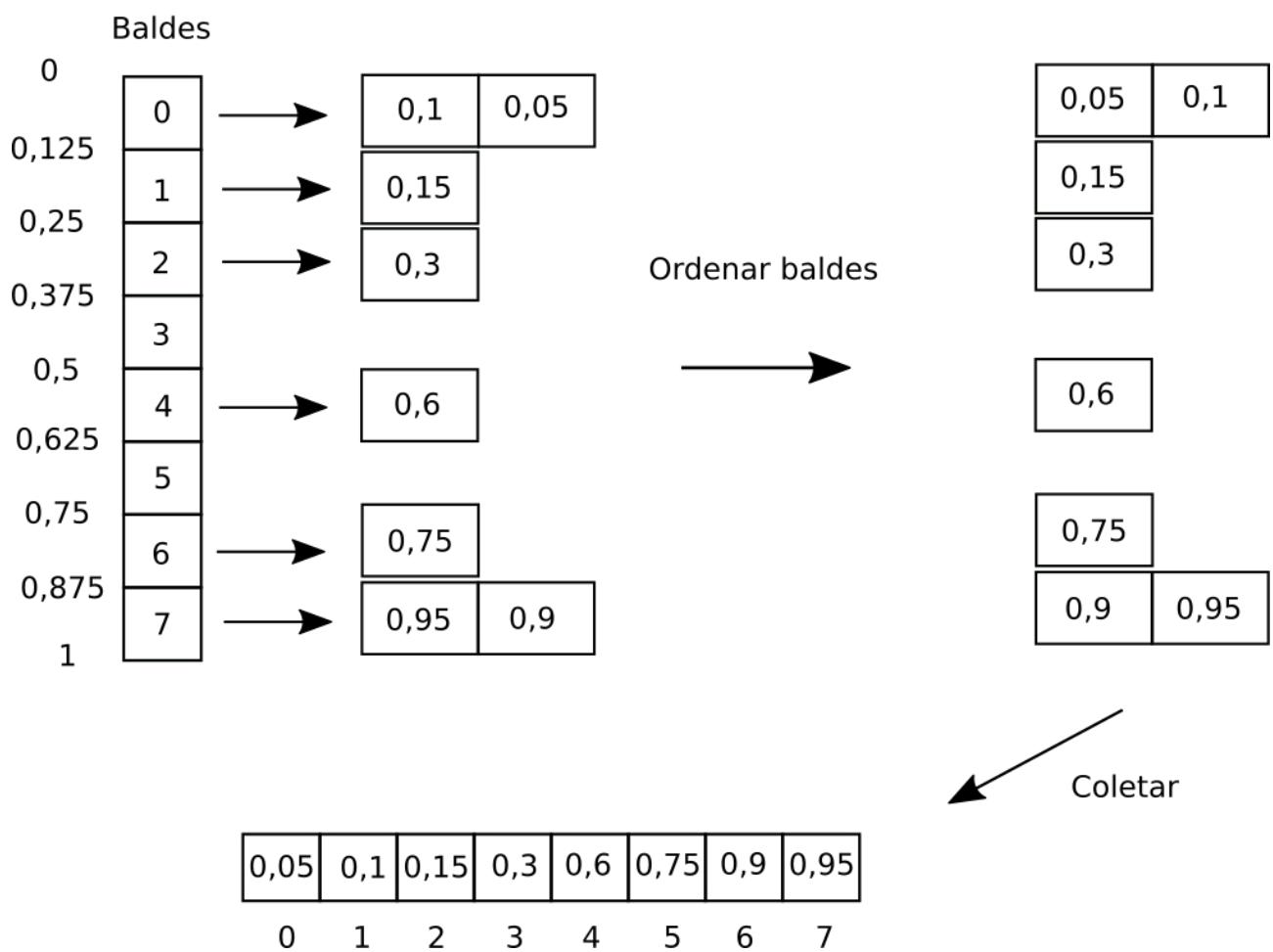
Seja $I = [0, K]$ um intervalo arbitrário. O primeiro passo do algoritmo consiste em dividir o intervalo de tamanho K em n baldes (buckets), cada um com tamanho K/n .



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.

0	1	2	3	4	5	6	7
0,75	0,1	0,3	0,95	0,05	0,6	0,9	0,15

$n = 8$



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

```

BucketSort(L, n) {
    buckets = list of n empty lists
    for i = 0 to n {
        index = floor(n*L[i])      # é usual ter K = 1 (normalizar)
        insert(buckets[index], L[i])
    }
    for i = 0 to n
        sort(buckets[i])
    output = concatenate(buckets, n)
    return output
}

```

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) $\text{insert}(p, x)$: insere x na lista apontada por uma referência p . Possui complexidade $O(1)$.
- ii) $\text{sort}(p)$: 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) $\text{concatenate}(B, n)$: 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 $\text{sort}()$. 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 Buckesort é:

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

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 varia 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
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	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Countingsort	$O(n+m)$	$O(n+m)$	$O(n+m)$
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. Importante quando ordenamos strings pelo primeiro caracter (i.e., Processamento de Linguagem Natural). Dentre os algoritmos anteriores temos a seguinte classificação:

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

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

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

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

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

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

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

-- Author Unknown

Estruturas de Dados: Árvores Binárias de Busca

Árvores binárias de busca 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:

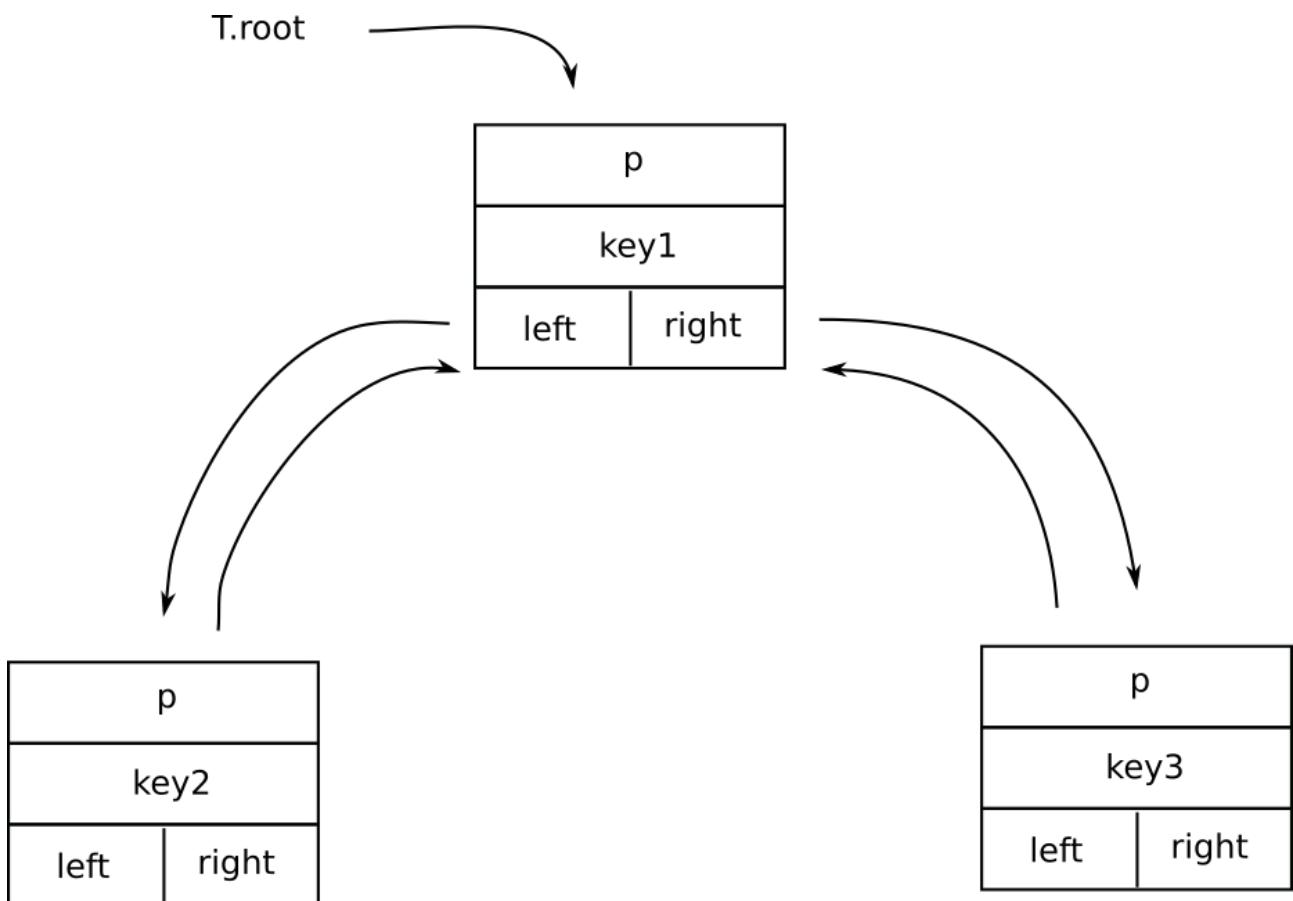
* **Chave (key)**: valor da ser armazenado em um nó de T

* **p**: referência para o nó pai

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

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

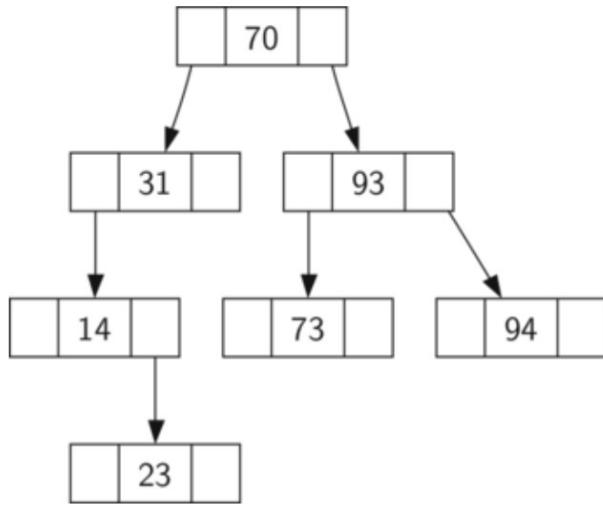
Uma árvore binária de busca T deve sempre ter uma raiz. Denotaremos aqui por $T.root$. Também podemos ter um atributo *size* para armazenar o número de nós da árvore, mas é opcional.



Toda árvore binária de busca possui uma 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 $y.key \geq x.key$.

A figura a seguir ilustra uma árvore binária de busca.



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

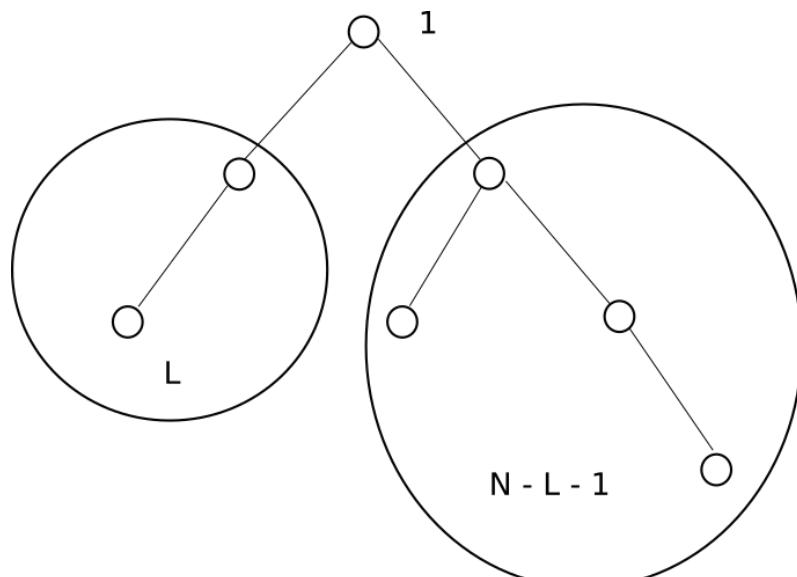
```

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

Note que o percurso inorder definido pelo algoritmo anterior sempre irá imprimir as chaves dos nós da árvore em ordem crescente. Por exemplo, na árvore da figura acima, teremos como saída: 14, 23, 31, 70, 73, 93, 94.

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:

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

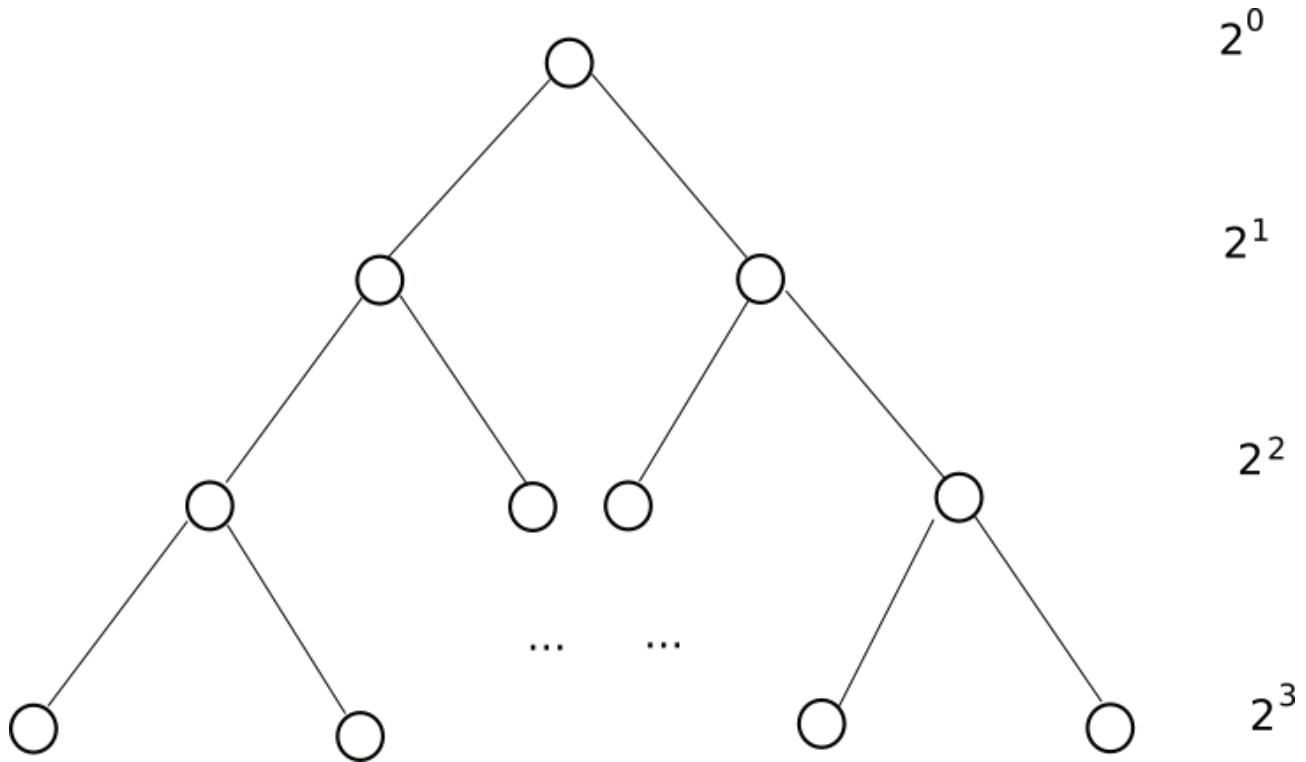
o que resulta em $O(N)$.

Busca em árvores binárias

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 balanceada, 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 $2^{h+1} = 2 \cdot 2^h = 2^h + 2^h$, 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 $2^{h+1} = n + 1$. 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?

$$h = \log_2 16 - 1 = 4 - 1 = 3$$

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
}

```

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.

```

Tree_Minimum(x) {
    while x.left ≠ NIL
        x = x.left
    return x
}

```

```

Tree_Maximum(x) {
    while x.right ≠ NIL
        x = x.right
    return x
}

```

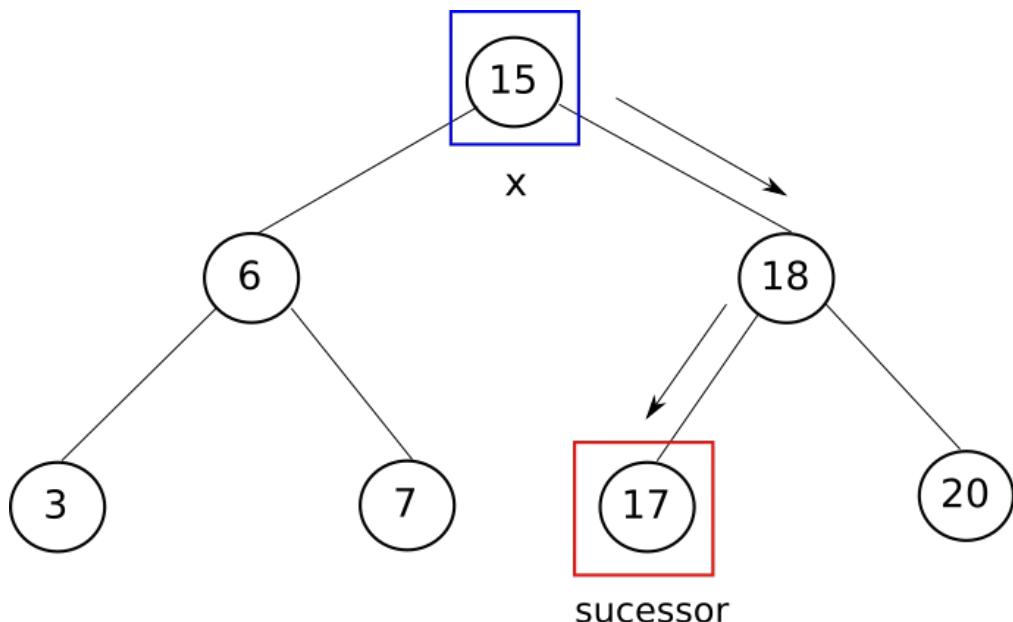
Sucessor e predecessor

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.

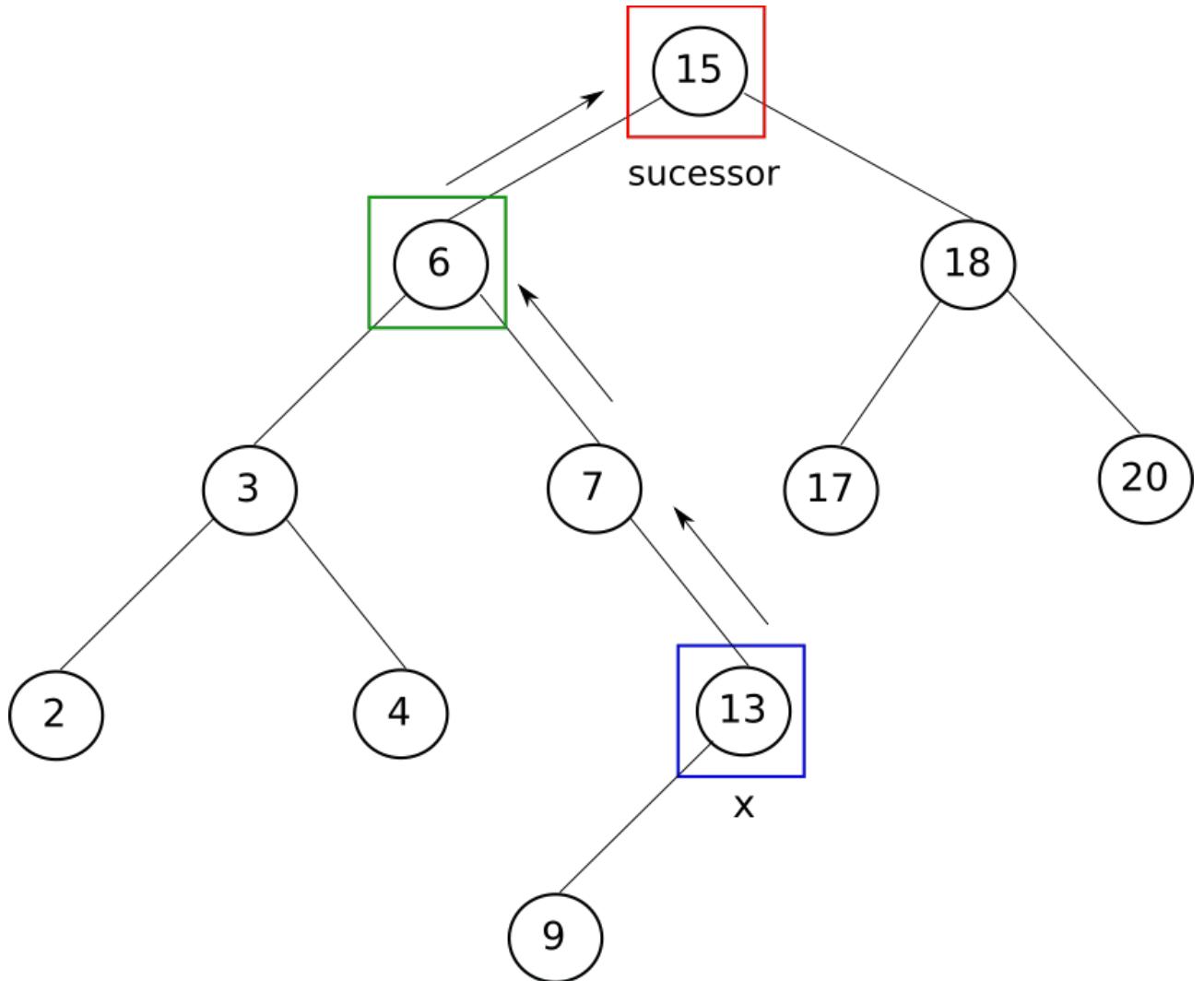
Def: O sucessor de um nó x é o nó y de menor chave tal que $y.key > x.key$, ou seja, é o próximo nó a ser visitado no percurso inorder.

Em resumo, há duas situações que podem ocorrer:

- 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 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;

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

Exercício: Escreva uma função para encontrar o predecessor de um nó x em uma árvore binária de busca.

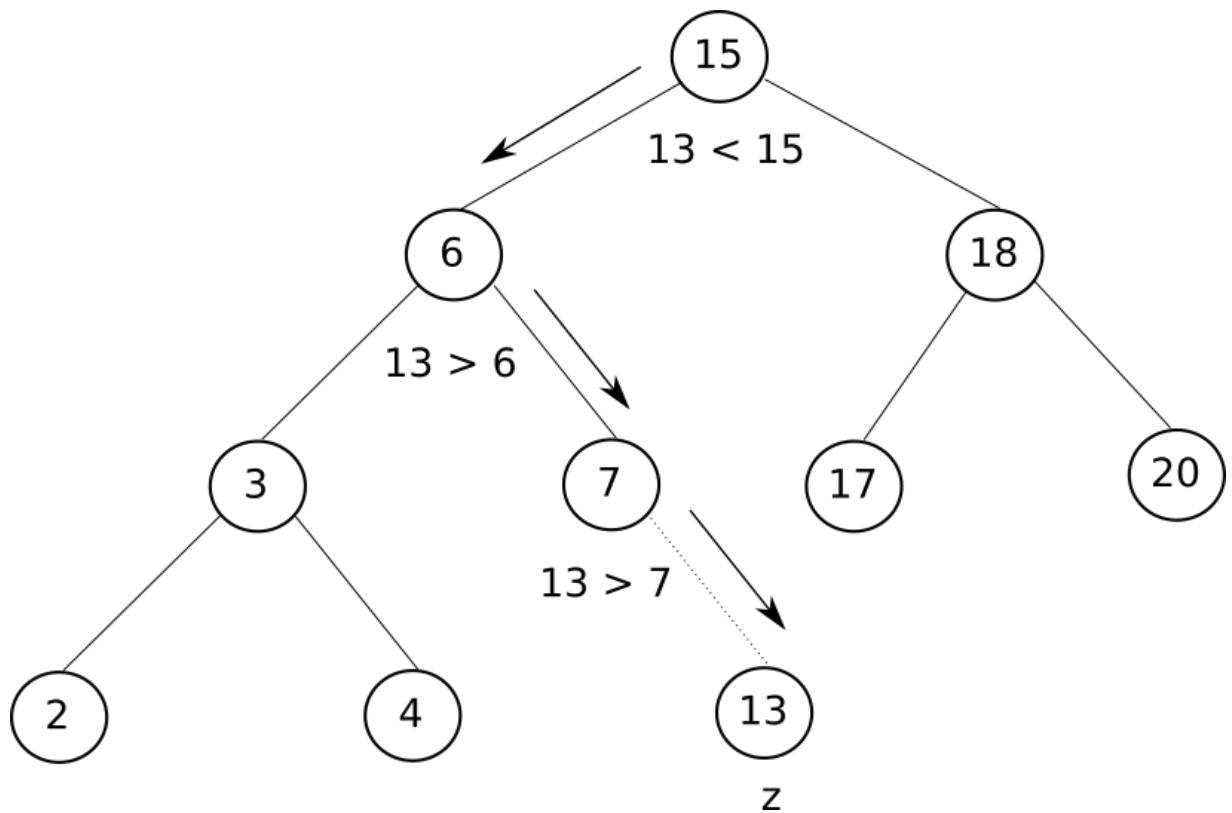
Inserção em árvores binárias de busca

Para inserir um novo nó em uma árvore binária de busca não basta encontrar a primeira posição disponível. Devemos respeitar a propriedade chave: isso faz com que cada chave tenha sua posição correta dentro da estrutura.

A função a seguir insere um novo nó z na árvore. Primeiro, devemos encontrar a posição correta de z em T, com base no valor de sua chave. Depois, realizamos o encadeamento lógico.

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

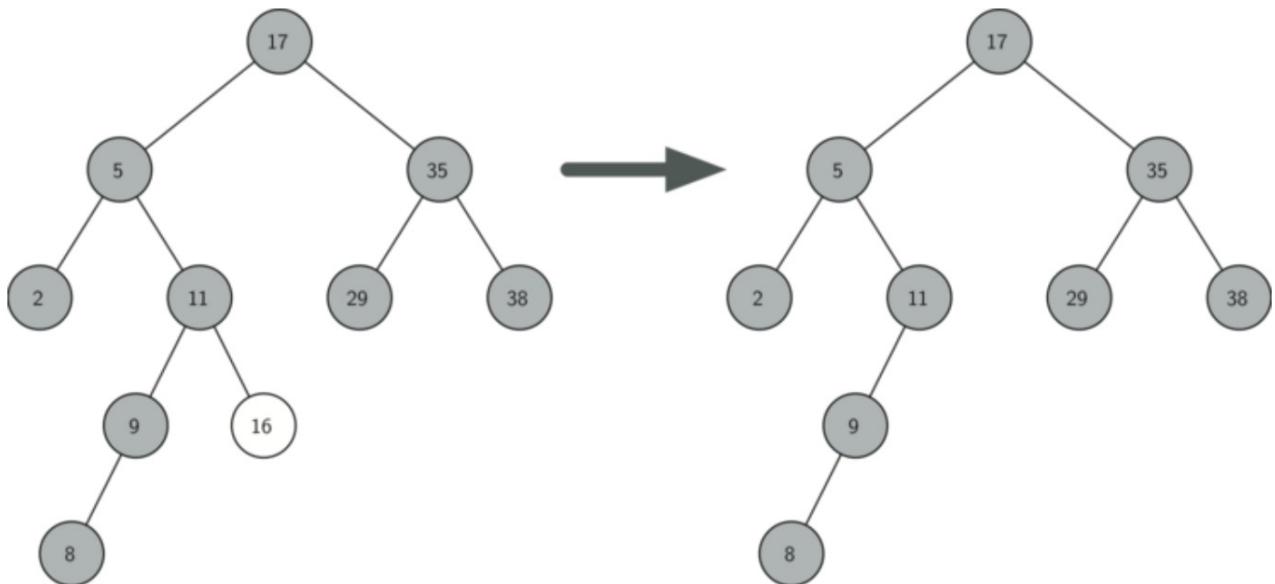
Assim como os algoritmos anteriores, temos que a complexidade da inserção depende da altura h da árvore: no melhor caso é $O(\log_2 n)$ e no pior caso é $O(n)$. A figura a seguir ilustra o processo de inserção da chave 13 em uma árvore binária de busca.



Remoção em árvores binárias de busca

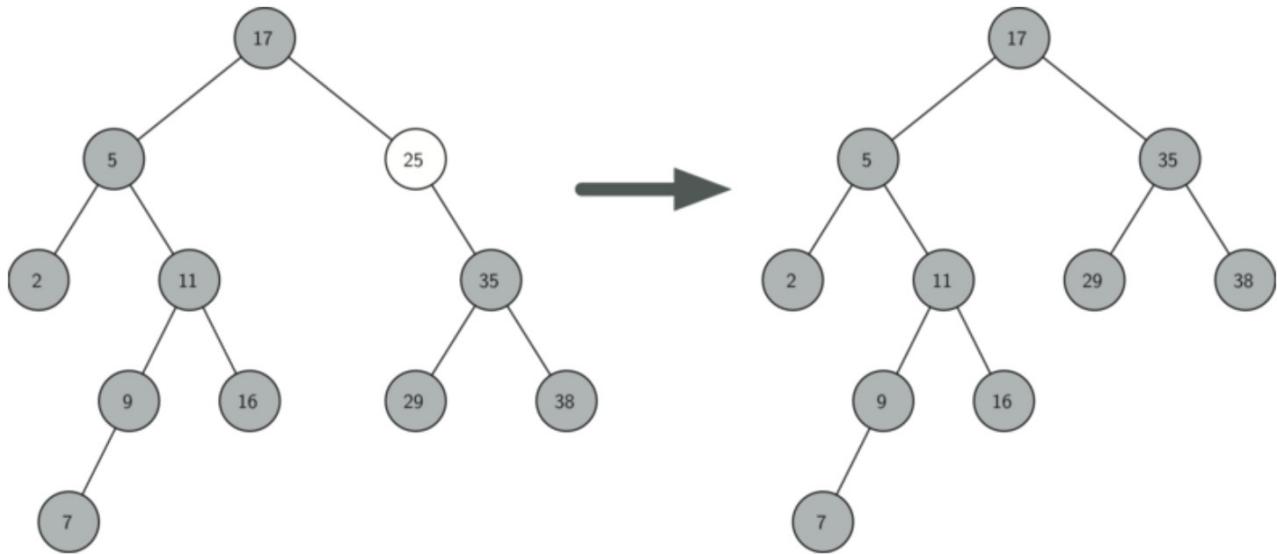
A remoção é um processo mais complicado que a inserção. A estratégia para remover um nó z da árvore T baseia-se na análise de 3 situações:

- i) Se o nó z não possui nenhum filho
- Basta remover z , modificando o seu nó pai



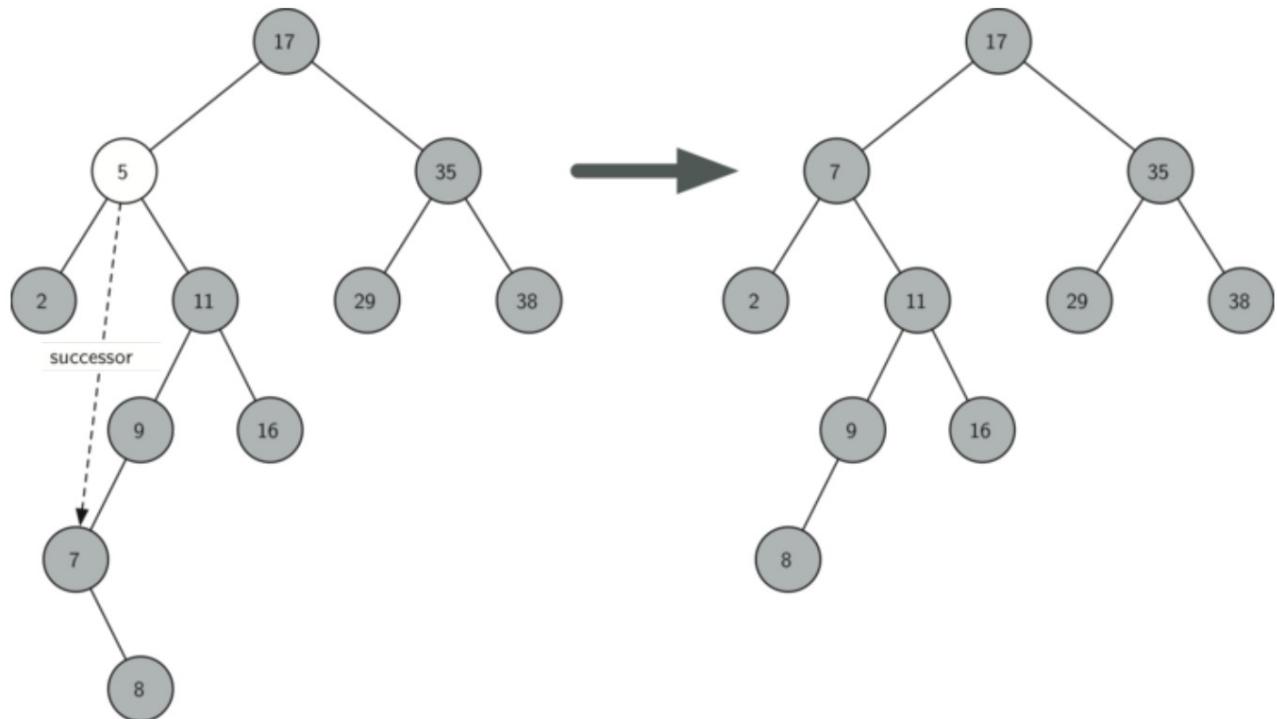
ii) Se o nó z possui apenas um filho y

- O único filho de y deve substituir z em T: o pai de z deve se ligar diretamente ao filho de z



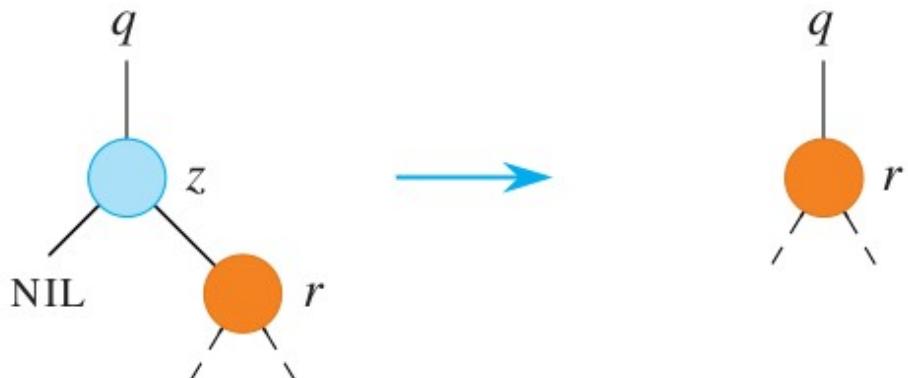
iii) Se o nó z possui dois filhos

- Devemos encontrar y, o sucessor de z na árvore T (menor chave da subárvore a esquerda). Em seguida substituímos z por y. Note que y não possui filho a esquerda, pois é o menor elemento. Assim, remover y de sua posição é um problema mais simples (caso ii)



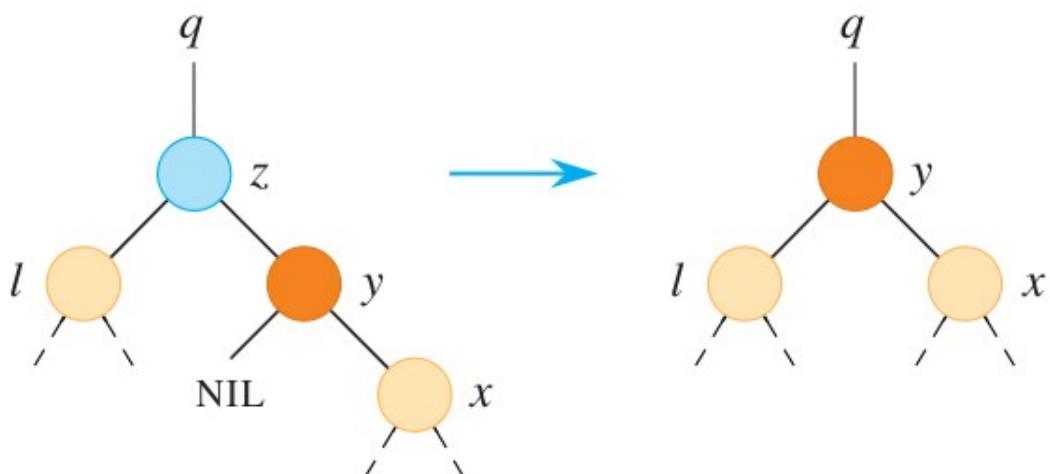
Para projetar um algoritmo para remoção de um nó z de uma árvore binária de busca T, é mais fácil organizar os casos da seguinte forma:

a) Se z não tem filho a esquerda, substitua z pelo seu filho a direita r , que pode ou não ser NIL. Quando ele é NIL, estamos lidando com o caso i) (z não tem nenhum filho). Quando ele é diferente de NIL, temos o caso de ter um único filho a direita. De modo similar, se z possui apenas um filho a esquerda l , substituímos z por l .



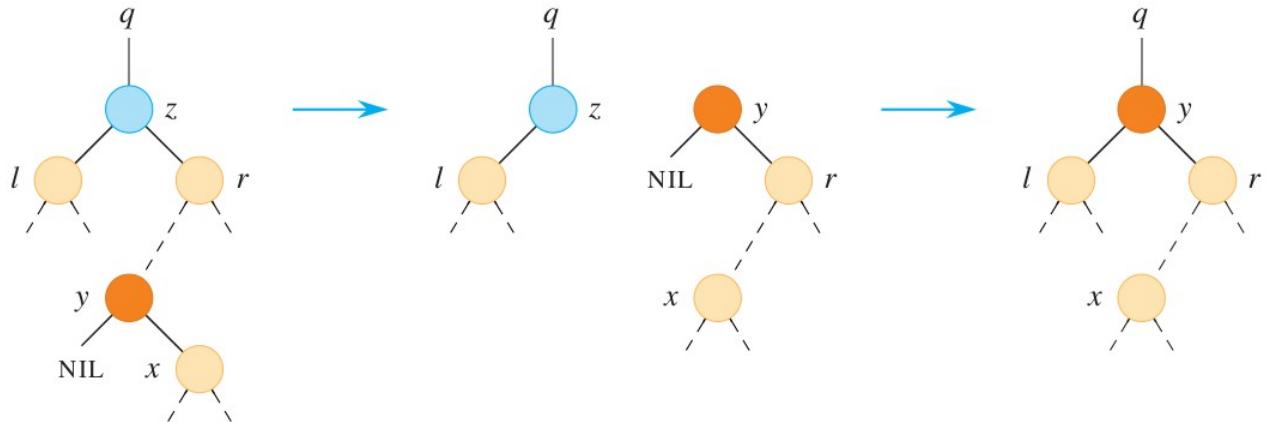
b) No caso de z ter ambos os filhos, l e r , devemos achar o sucessor de z , denotado por y . Podem ocorrer 2 casos:

i) y é o filho a direita de z ($r = y$)



Note que só existe uma coisa a fazer: transplantar l (filho a esquerda de z) para ser o filho a esquerda de y e remover z , fazendo o pai de y ser q .

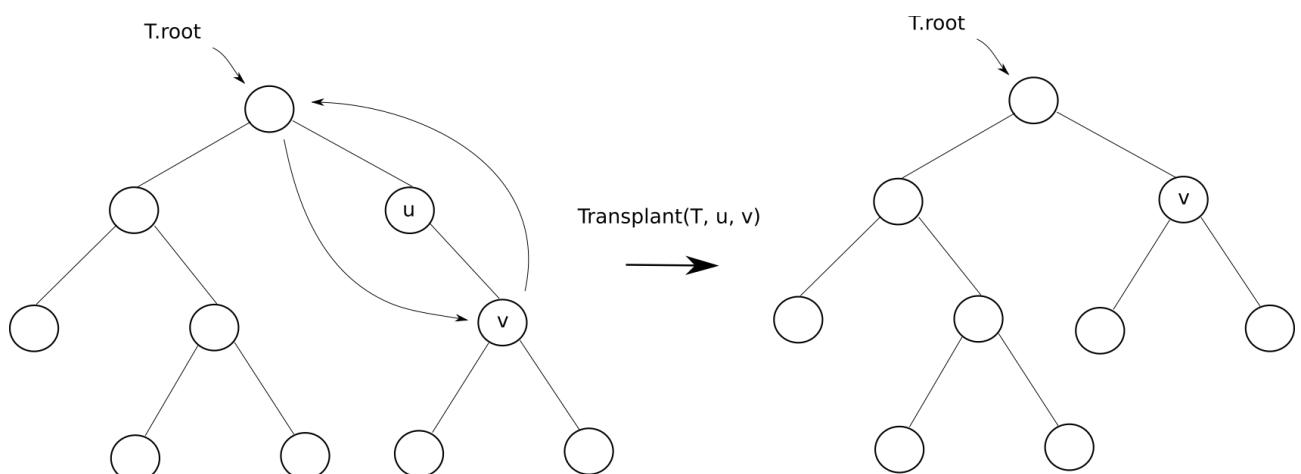
ii) Aqui, y está na subárvore a direita de z, mas $y \neq r$. Primeiro, devemos substituir y por seu filho a direita e então substituir z por y (o filho a direita de y vira filho a esquerda de r).



Como parte do processo de remoção de um nó da árvore, precisamos mover subárvore ao longo de T. A primitiva Transplant substitui uma subárvore como se fosse um filho de um nó por outra subárvore. A ideia é que quando a primitiva Transplant substitui uma subárvore enraizada em u por outra subárvore enraizada em v, o pai do nó u se torna o pai do nó v, de modo que o pai de u acaba tendo v como filho. É possível que v também seja NIL.

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

A figura a seguir ilustra o resultado da aplicação da função transplant em um exemplo.



A seguir, apresentamos uma função que realiza a remoção de um nó da árvore binária de busca utilizando a primitiva Transplant.

```
Tree_Delete(T, z) {
    if z.left == NIL
        # substitui z pelo seu filho a direita
        Transplant(T, z, z.right)
    else {
        if z.right == NIL
            # substitui z pelo seu 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 dir
            if y ≠ z.right {          # está mais embaixo da árvore
                # Substitui y por seu filho a direita
                Transplant(T, y, y.right)
                y.right = z.right      # aponta y.right para r
                y.right.p = y          # pai de r deve ser y
            }
            Transplant(T, z, y)      # substitui z por y
            y.left = z.left          # passa filho a esq de z para y
            y.left.p = y              # pai do filho a esq deve ser y
        }
    }
}
```

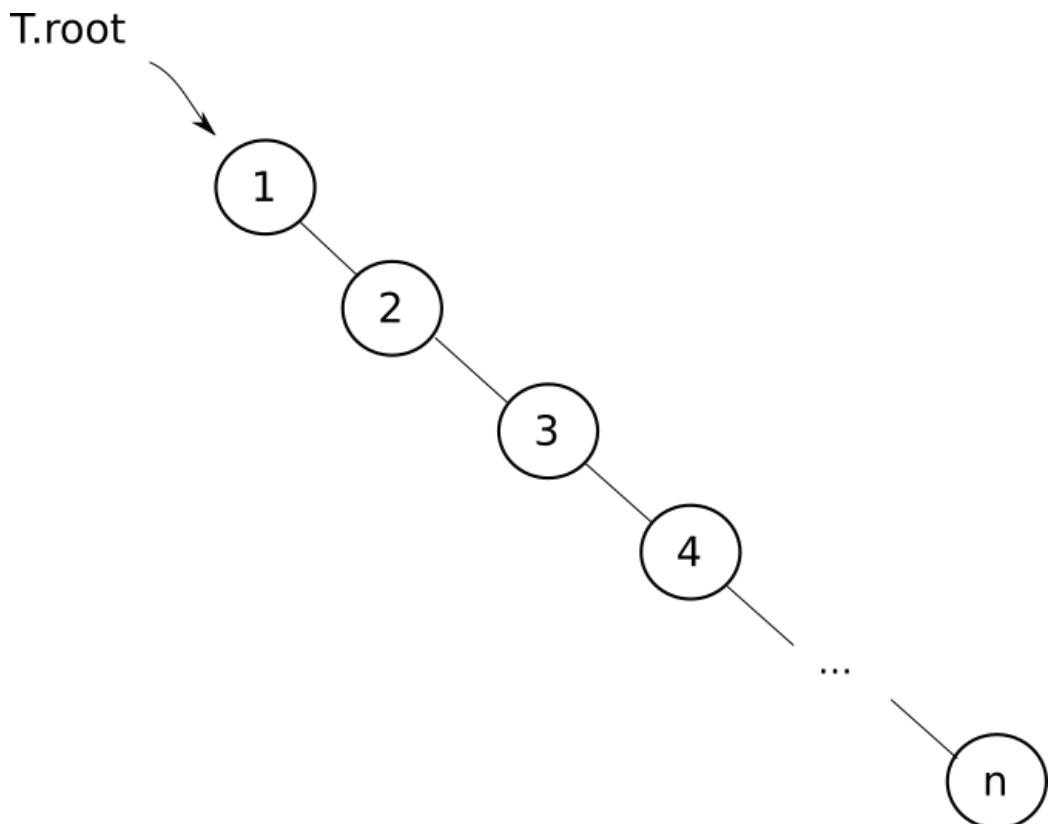
Como a primitiva Transplant tem complexidade $O(1)$, a complexidade da função Tree_Delete depende essencialmente da função Tree_minimum, que é $O(h)$, onde h denota a altura da árvore. Sendo assim, o custo computacional da remoção também é $O(h)$. Portanto, no melhor caso a remoção é $O(\log n)$ e no pior caso é $O(n)$.

"Lembre-se: a maioria das pessoas quer te ver bem, mas não melhor do que elas próprias."
-- 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 menores 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:

$$x.b = h_L - h_R$$

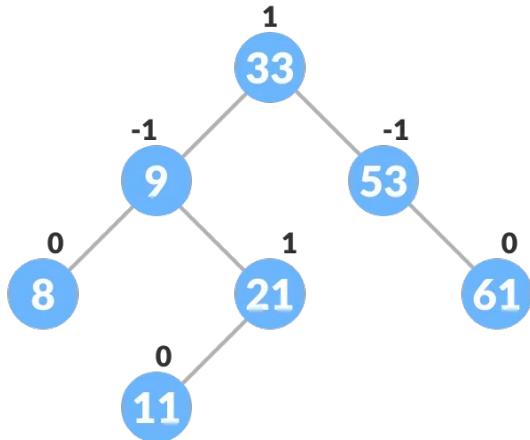
onde h_L é a altura da subárvore a esquerda e h_R é a altura da subárvore a direita.

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.



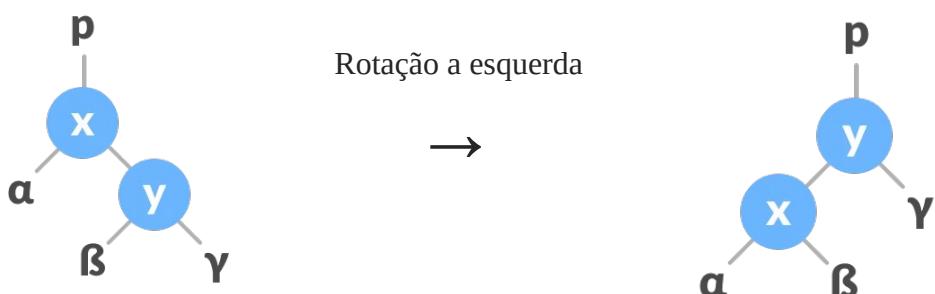
Para garantir que após a inserção ou remoção de um nó, a árvore em questão seja balanceada, devemos executar algumas operações de modo a garantir que o fator de balanço dos nós esteja entre -1 e +1. A seguir discutimos um pouco dessas operações.

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.

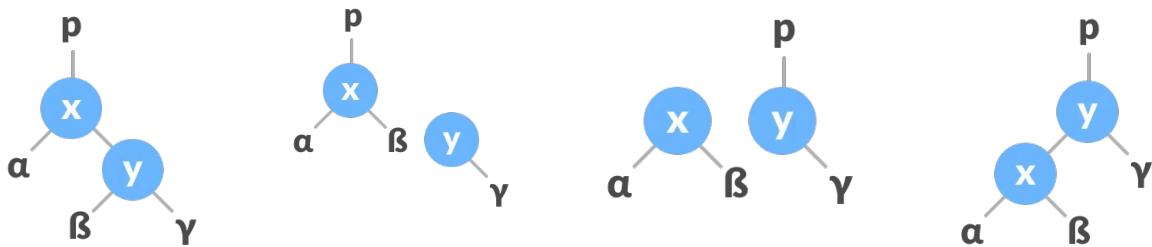


A função `leftRotate` a seguir mostra os passos lógicos dessa operação.

```
leftRotate(x) {
    y = x.right
    β = y.left
    x.right = β
    y.left = x
    return y
}
```

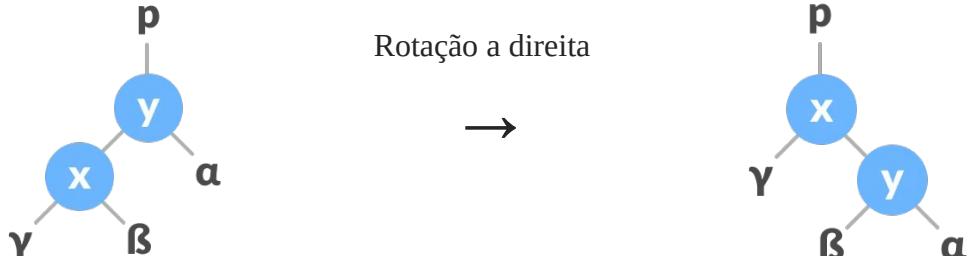
Note que em resumo temos os seguintes passos:

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 .



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.



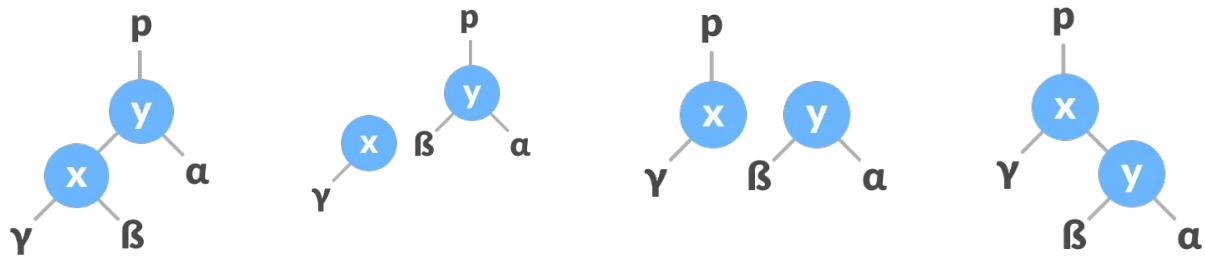
A função rightRotate a seguir mostra os passos lógicos dessa operação.

```
rightRotate(y) {
    x = y.left
    beta = x.right
    y.left = beta
    x.right = y
    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 .

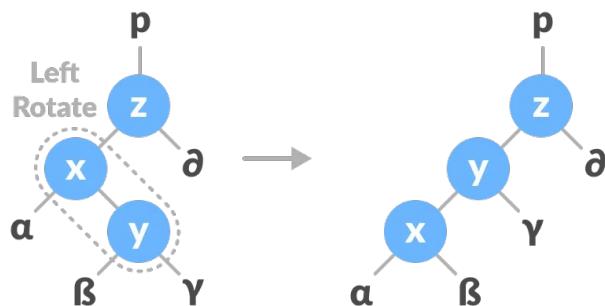
3. Faça x ser pai de y.



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.

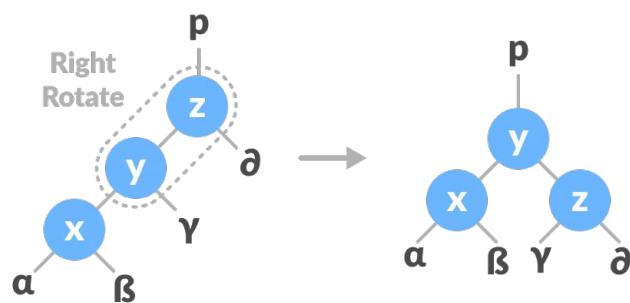
Rotação esquerda-direita

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



Puxando o y para cima de x

2. 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.

```

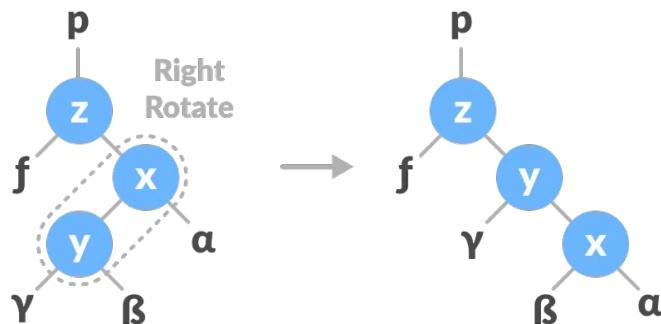
leftRightRotate(z) {
    x = z.left
    y = leftRotate(x)
    w = rightRotate(y)
    return w
}

```

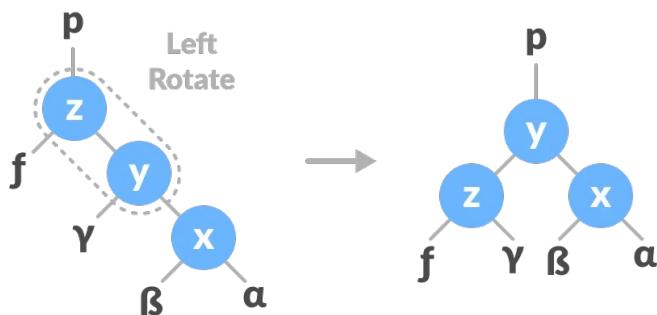
De forma análoga, podemos realizar uma rotação direita-esquerda.

Rotação direita-esquerda

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



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



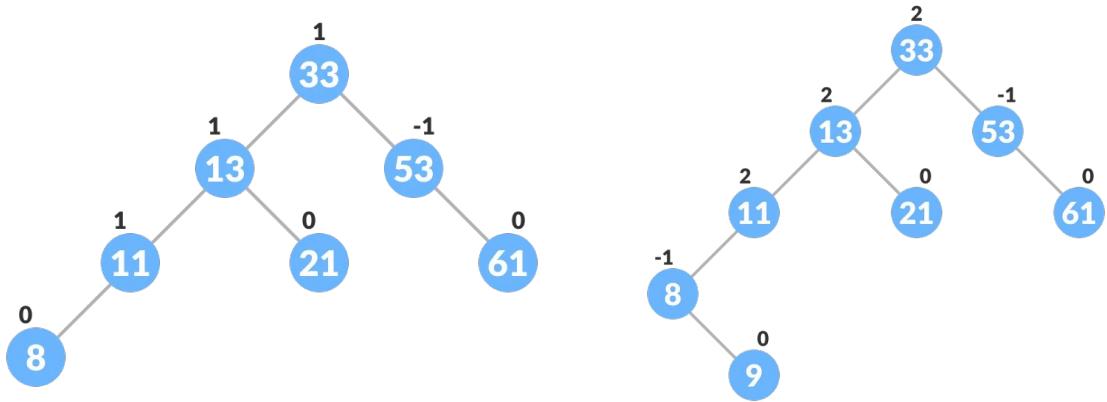
```

rightLeftRotate(z) {
    x = z.right
    y = rightRotate(x)
    w = leftRotate(y)
    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. Para o caso de fator de balanço positivo, deve-se aplicar a seguinte regra:

Se fator de balanço > 1

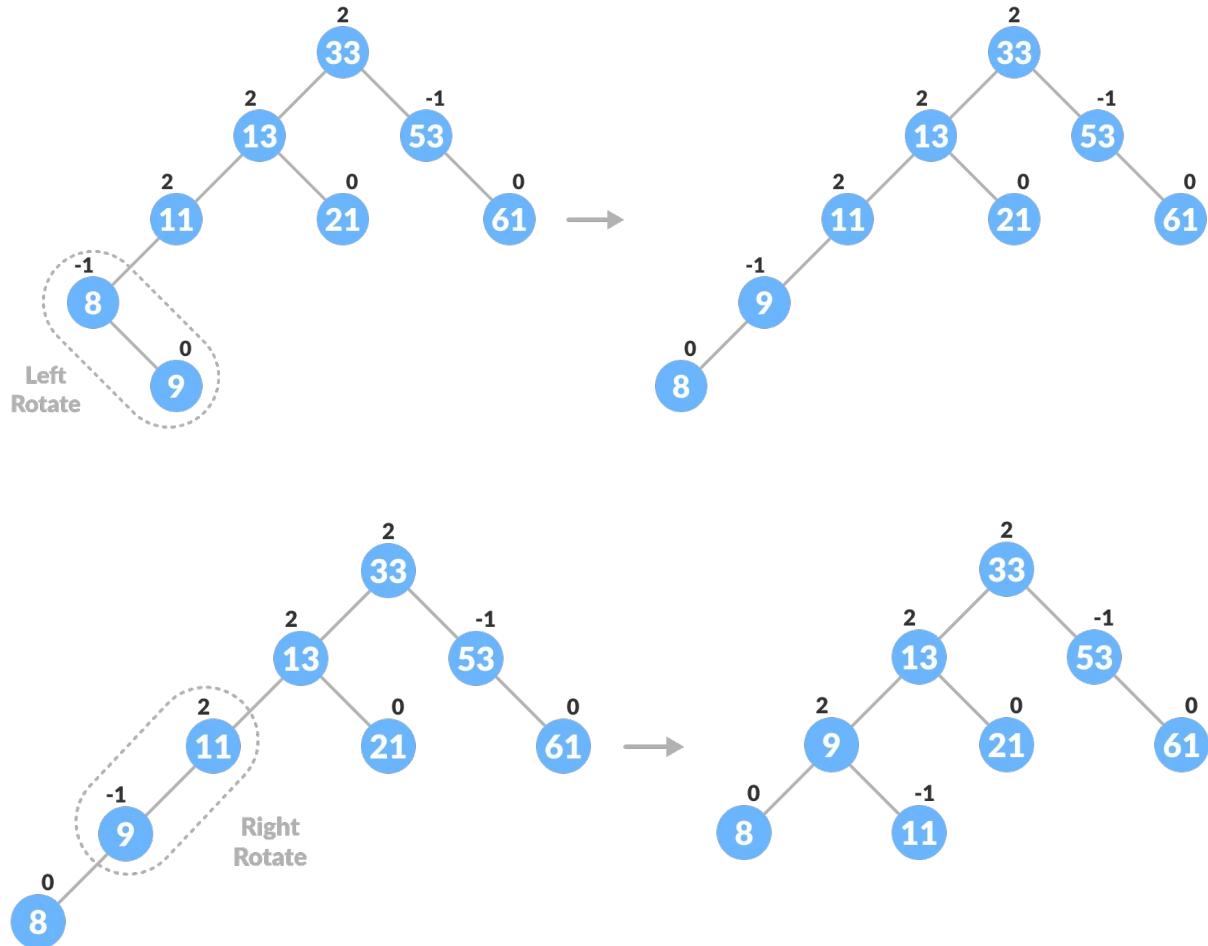
Se chave do novo nó (9) ≤ chave do filho a esquerda do nó desbalanceado (8)

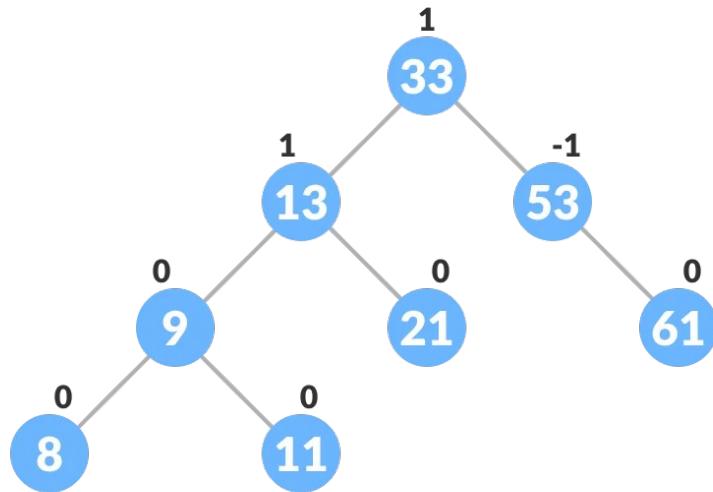
Execute uma rotação a direita

Senão

Execute uma rotação esquerda-direita

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.

Se fator de balanço < -1

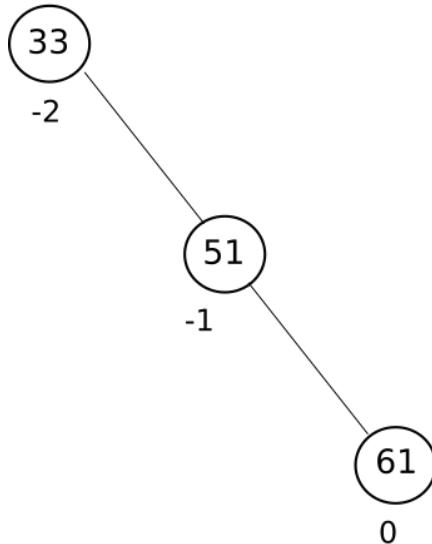
Se chave do novo nó > chave do filho a direita do nó desbalanceado

Execute uma rotação a esquerda

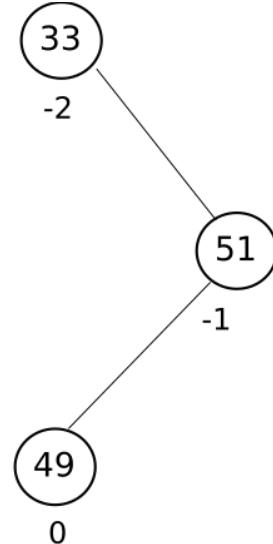
Senão

Execute uma rotação direita-esquerda

A figura a seguir ilustra um exemplo simples.



Rotação a esquerda



Rotação direita-esquerda

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

Caso 1: após a remoção, nada precisa ser feito, apenas checar o fator de balanço dos antecessores.

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

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.

A regra para o rebalanceamento da árvore consiste em rebalancear o ancestral mais próximo como:

Se fator de balanço > 1

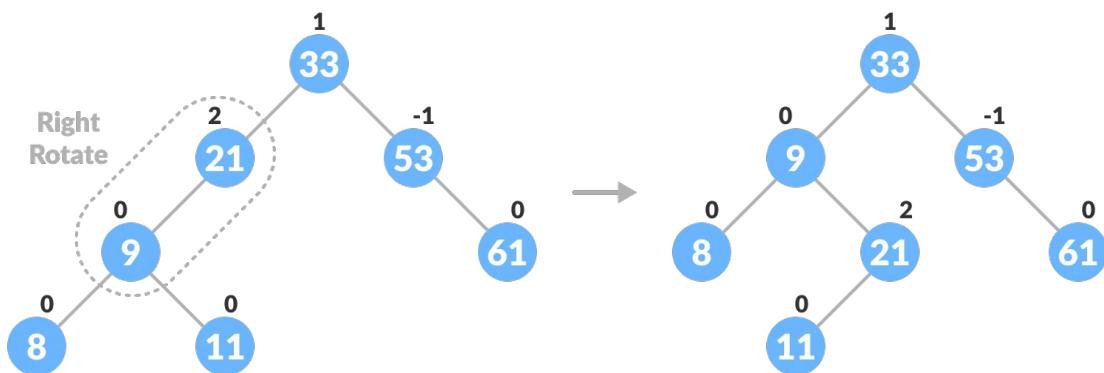
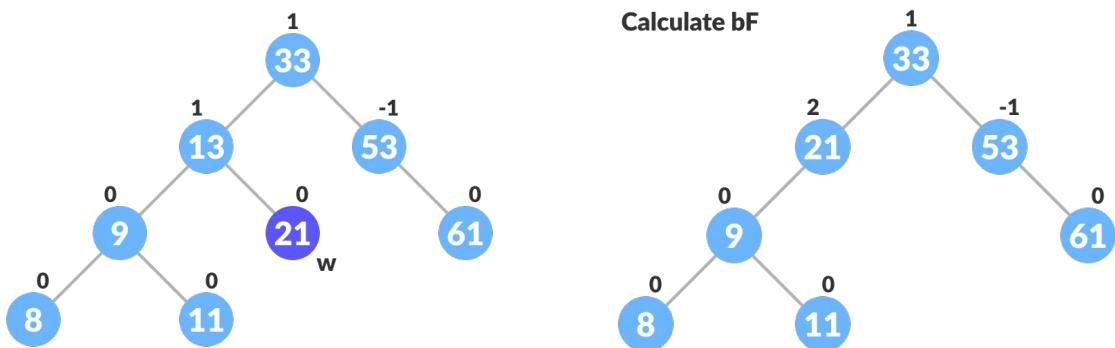
Se o fator de balanço do filho a esquerda ≥ 0

Execute uma rotação a direita

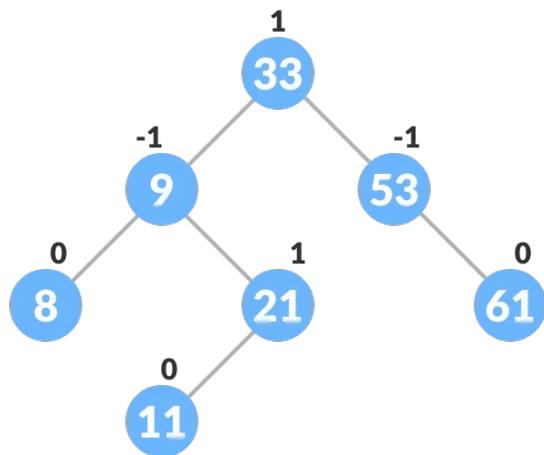
Senão

Execute uma rotação esquerda-direita

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



Após o rebalanceamento, os fatores de balanço são dados por:



Para o caso simétrico, temos uma regra similar.

Se fator de balanço ≤ -1

Se o fator de balanço do filho a esquerda ≤ 0

Execute uma rotação a esquerda

Senão

Execute uma rotação direita-esquerda

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 de busca digitais.

"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: 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:

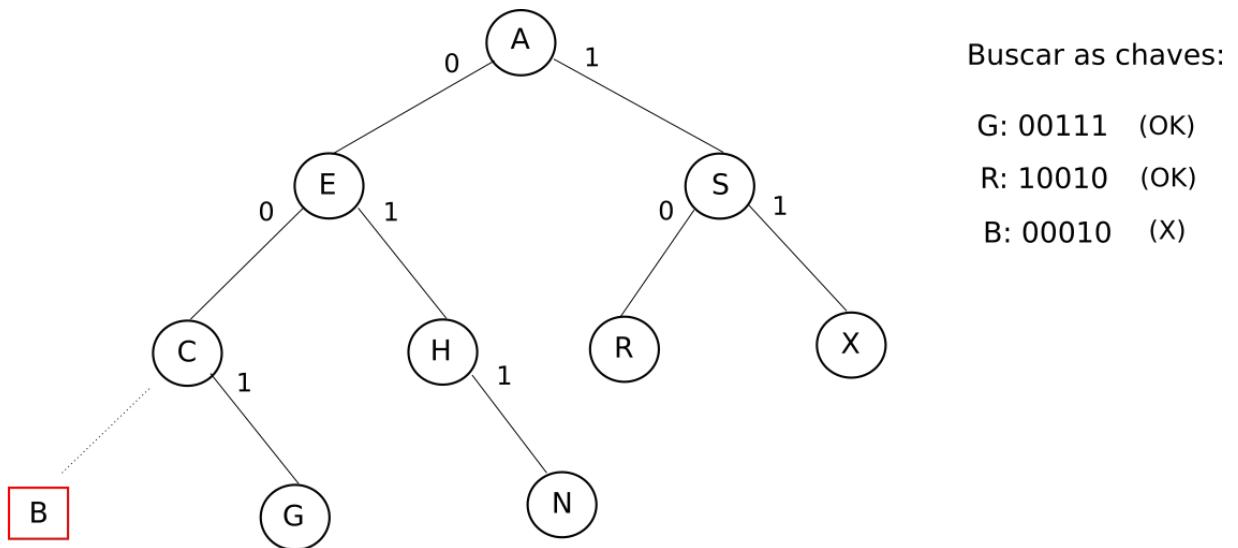
$$\Omega = \{A, B, C, D, \dots, W, X, Y, Z\}$$

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

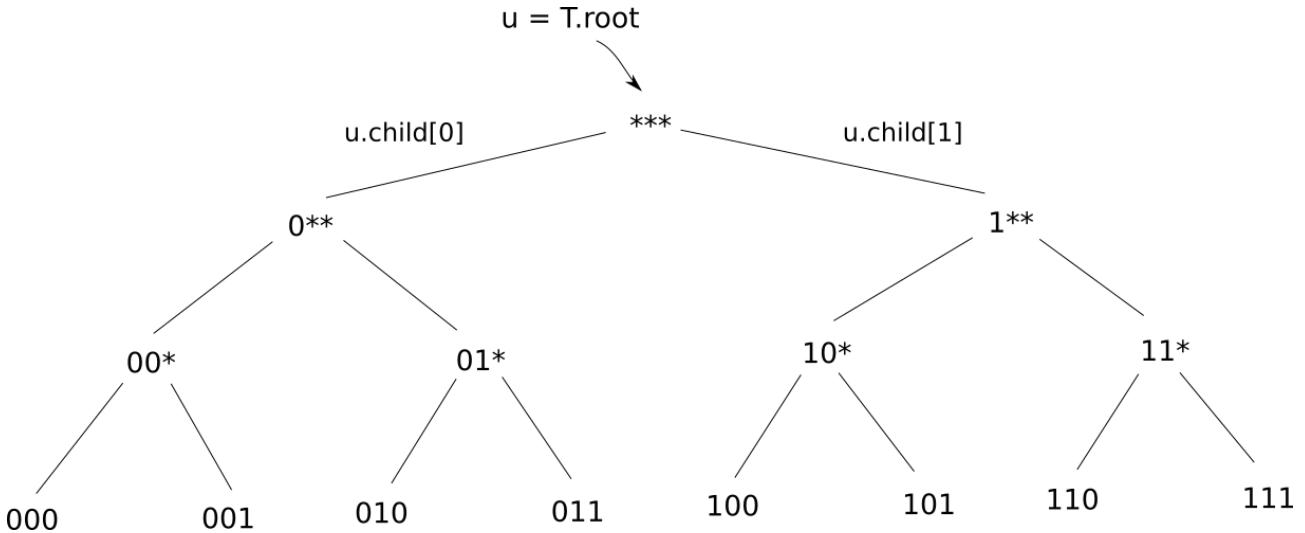
A estratégia para a busca em árvores binárias digitais consiste em comparar um bit da chave buscada com um bit do nó correto. Mas as chaves são compostas por vários bits, como saber quais deles devemos comparar? Depende da profundidade em que o nó correto se encontra! Se estamos na raiz da árvore, comparamos 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.



deveria estar aqui!

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.

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.child[c] == NIL
            return False # não encontrou
        if u.key != x.key
            u = u.child[c] # desce um nível
        else
            return True # encontrou
    }
}
  
```

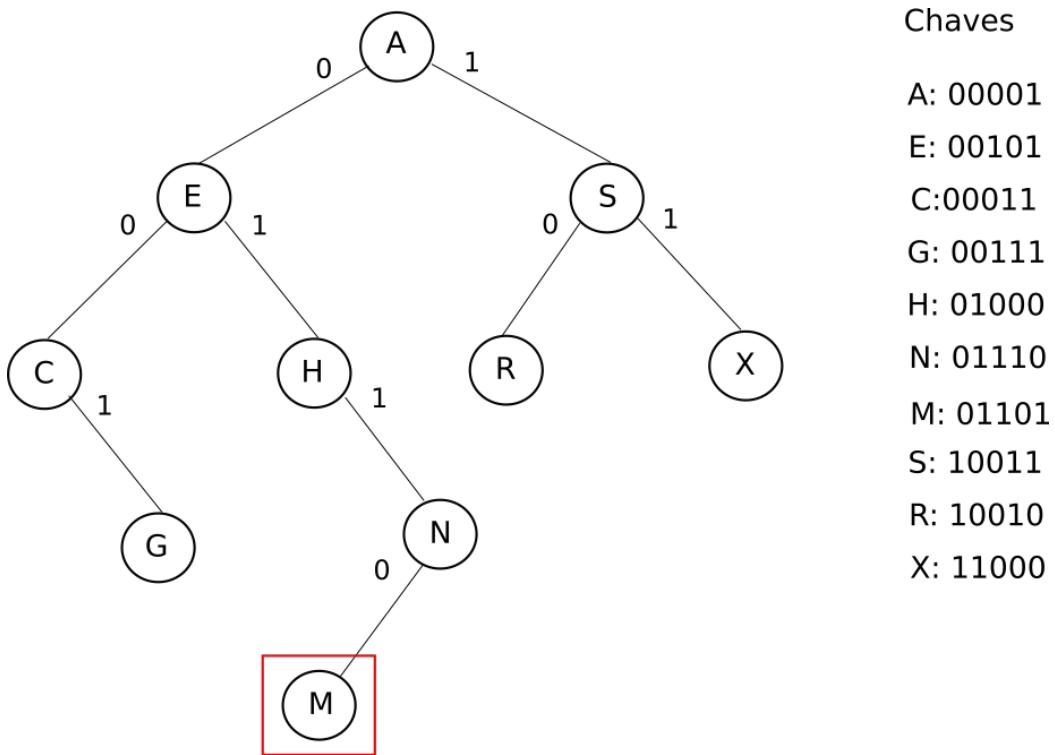
O comando para extrair o bit a ser analisado usa operadores binários. Suponha $w = 3$ bits e que a chave a ser buscada é 100. A ideia é que se estamos na raiz, o bit a ser analisado é o 1. Na raiz, o valor de $i = 0$ e assim, devemos deslocar os bits de $x.key$ duas vezes para direita gerando 001 e fazer 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.child[c] == NIL
            break                      # achou posição da chave
        if u.key == x.key
            return False              # chave já pertence a T
        u = u.child[c]
    }
    u.child[c] = x
    x.p = u
}

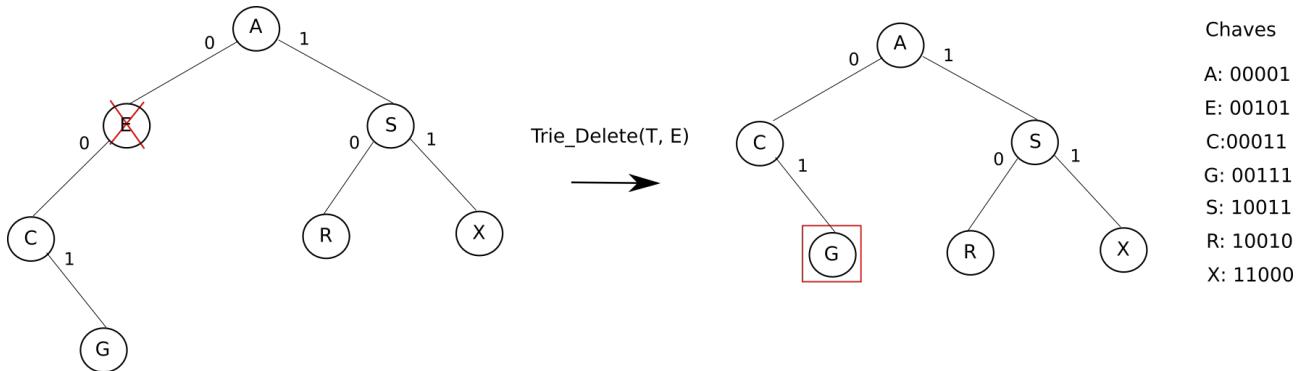
```

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.

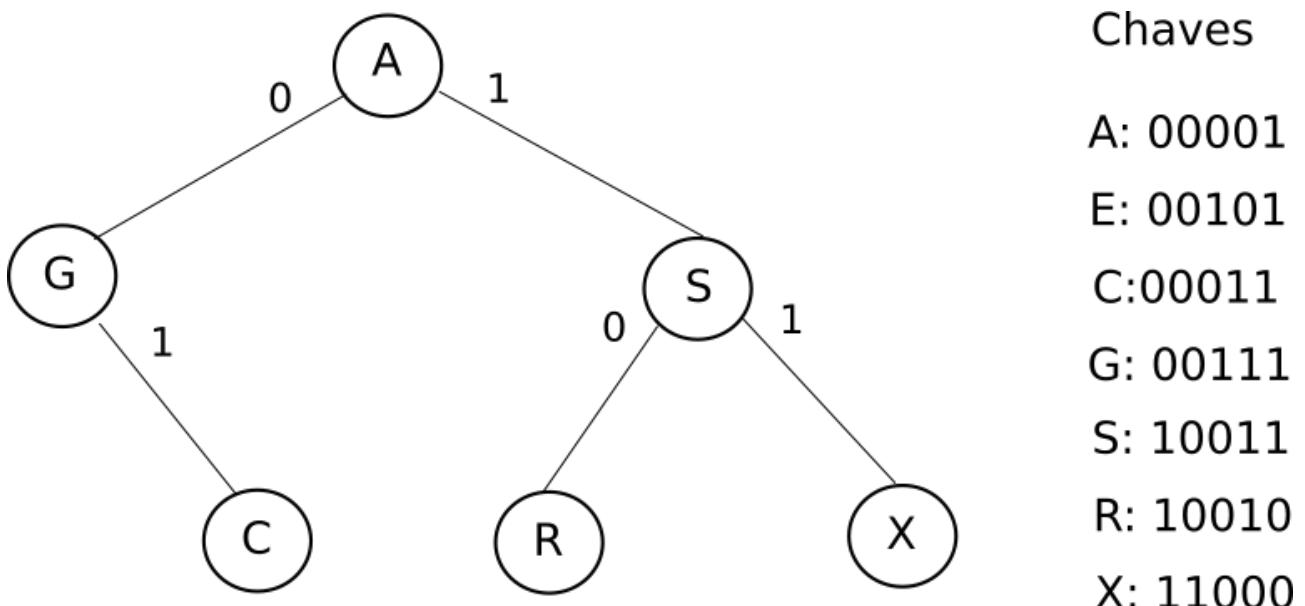
O caso mais simples ocorre quando o nó a ser removido é uma folha, ou seja, não possui filhos. Neste caso, basta remover o nó desligando as referências com o nó pai.

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.child[c] == NIL
            return False                  # chave não pertence a T
        if u.key == x.key
            break                         # encontrou chave a ser removida
        u = u.child[c]
    }
    if i == w-1                      # se nó é folha
        u.p.child[c] = NIL           # desaloca o nó
    else {
        z = u                        # se nó não é folha
        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: Skip Lists

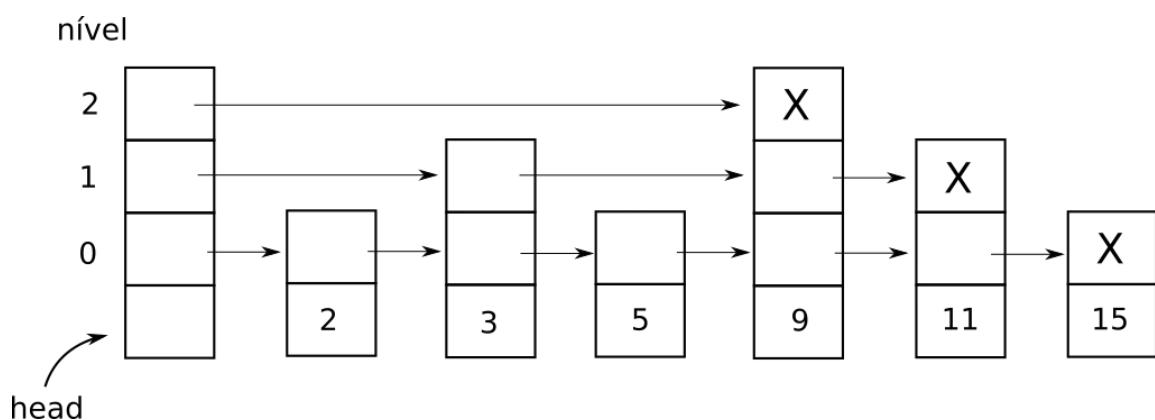
Skip Lists, ou listas com saltos, são estruturas de dados probabilísticas que permitem que as operações de busca, inserção e remoção de nós sejam realizadas com complexidade $O(\log n)$. Essas estruturas possuem as vantagens de um vetor ordenado (como na busca binária), mas com uma estrutura dinâmica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

* 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:

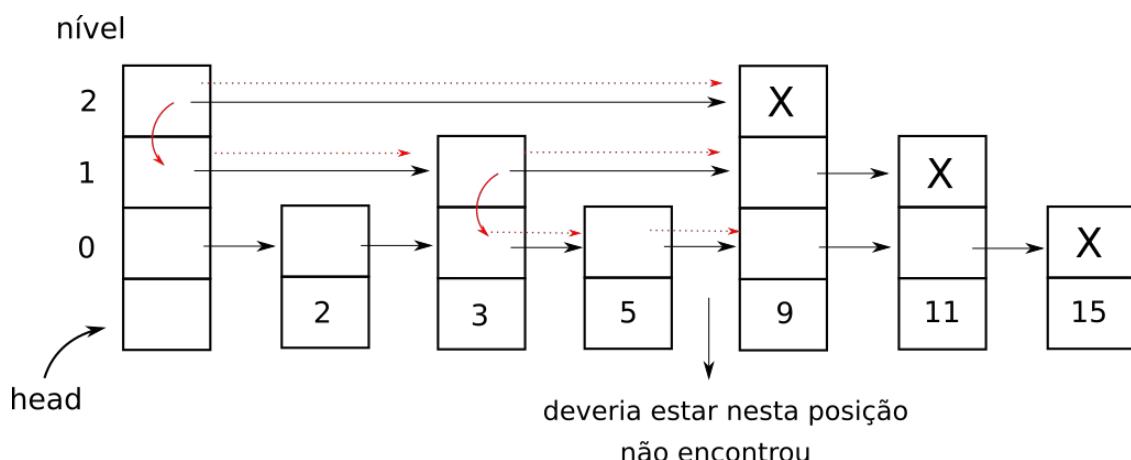
- a camada inferior é uma lista encadeada ordenada tradicional.
- cada camada superior é uma “via expressa”, onde um elemento da camada i aparece na camada $i+1$ com probabilidade $q = 1/d$. Note que se $d = 2$, temos 50% de chances de aparecer (é como se jogássemos uma moeda e: se o resultado for cara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

```
Skip_List_Insert(L, key) {
    x = L.head
    for i = L.level downto 0 {
        while x.forward[i].key < key
            x = x.forward[i]
    }
    # Aqui, temos que x.key < key ≤ x.forward[0].key (posição)
    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 $d'' = 10$

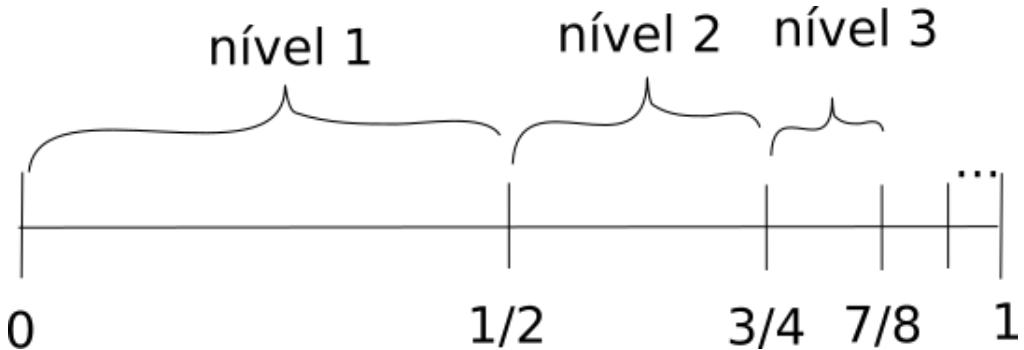
$$\text{Tempo: } \frac{10}{2} \log_{10} n = 5 \left(\frac{\log_2 n}{\log_2 10} \right) = \frac{5}{3.32} \log_2 n \approx 1.5 \log_2 n$$

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

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 é gerar um número 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	
Nível	0	1	2	3	4	...

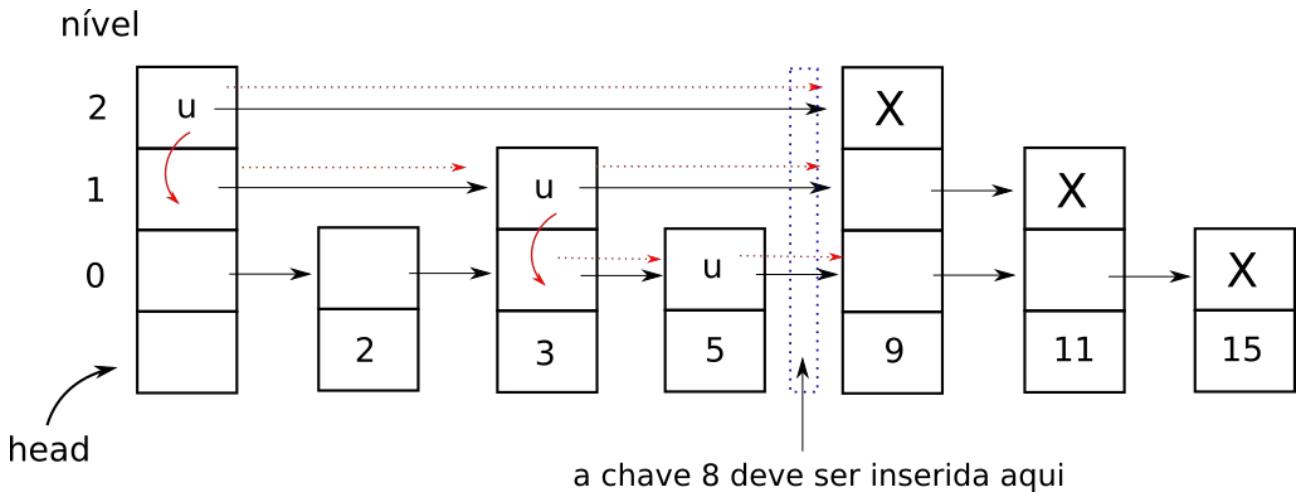
Inserção em Skip Lists

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() < p 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.



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

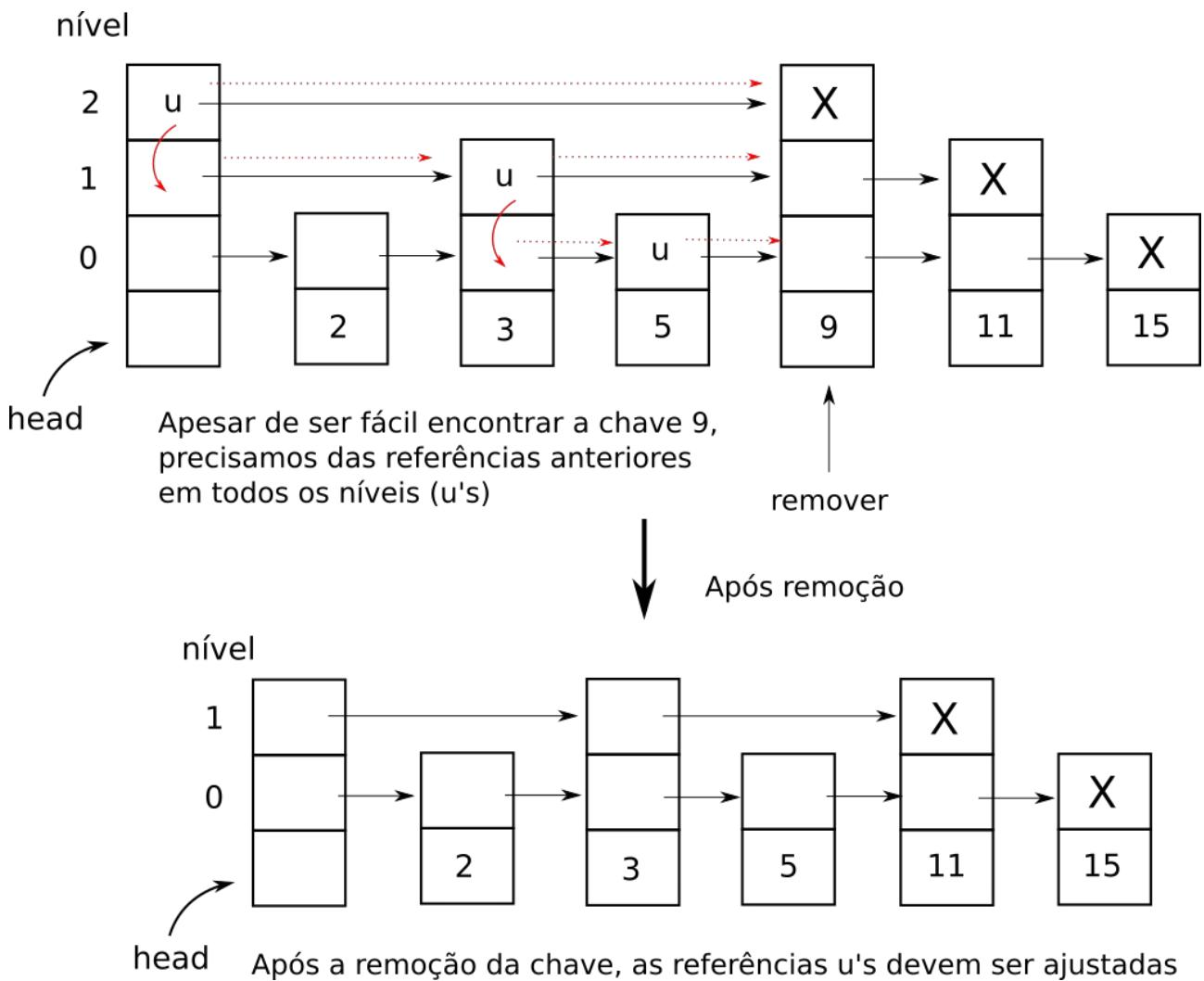
```

Skip_List_Insert(L, searchKey, newKey) {
    Let update[0..maxlevel] be an array of references (pointers)
    x = L.header
    for i = L.level downto 0 {
        while x.forward[i].key < searchKey
            x = x.forward[i]
        update[i] = x
    }
    x = x.forward[0]
    if x.key == searchKey
        x.key = newKey
    else {
        level = Random_Level(q)
        if level > L.level {
            for i = L.level+1 to level
                update[i] = L.header
            L.level = level
        }
        x = Make_Node(level, searchKey)
        for i = 0 to level {
            x.forward[i] = update[i].forward
            update[i].forward = x
        }
    }
}

```

Remoção em Skip Lists

Ao remover um nó da Skip List, além de acertar as referências (ponteiros), 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.

```

Skip_List_Delete(L, searchKey) {
    Let update[0..maxlevel] be an array of references (pointers)
    x = L.header
    for i = L.level downto 0 {
        while x.forward[i].key < searchKey
            x = x.forward[i]
        update[i] = x
    }
    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)
    while L.level > 0 and L.header.forward[L.level] == NIL
        L.level = L.level - 1
    }
}

```

Análise da complexidade

Como 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 custo computacional de tais operações é dado por:

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

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

"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 $O(1)$.

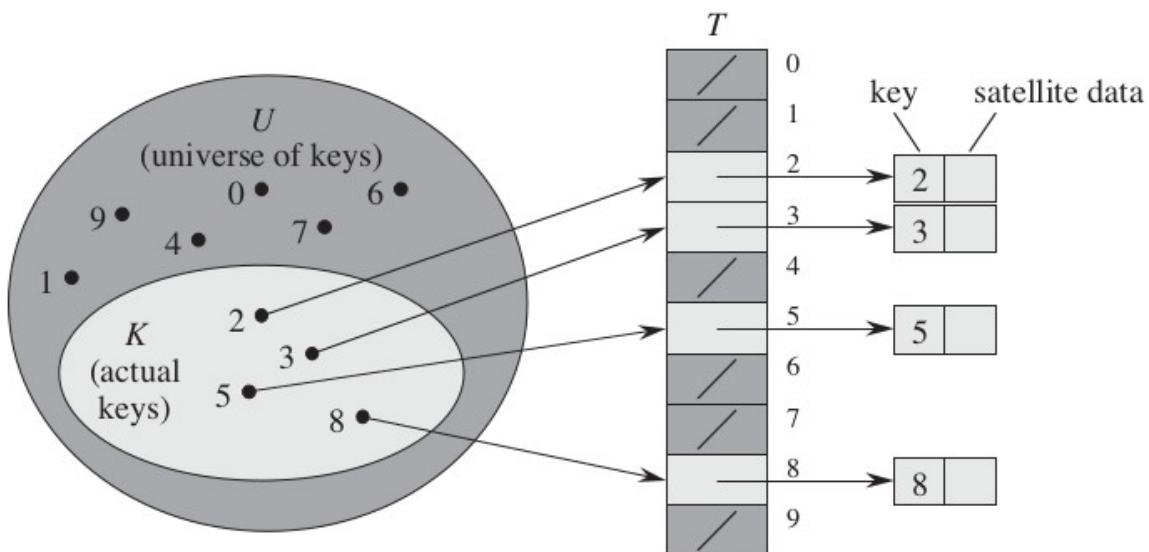
A vantagem das Hash Tables é que quando o número de chaves utilizado é relativamente pequeno em relação ao número total de possíveis chaves, essas estruturas de dados não utilizam a chave propriamente dita como índice para o acesso direto, mas sim uma função hashing.

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, 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 $O(1)$.

```
Direct-Address-Search( $T$ ,  $k$ )
    return  $T[k]$ 
```

```
Direct-Address-Insert(T, x)
```

```
    T[x.key] = x
```

```
Direct-Address-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!

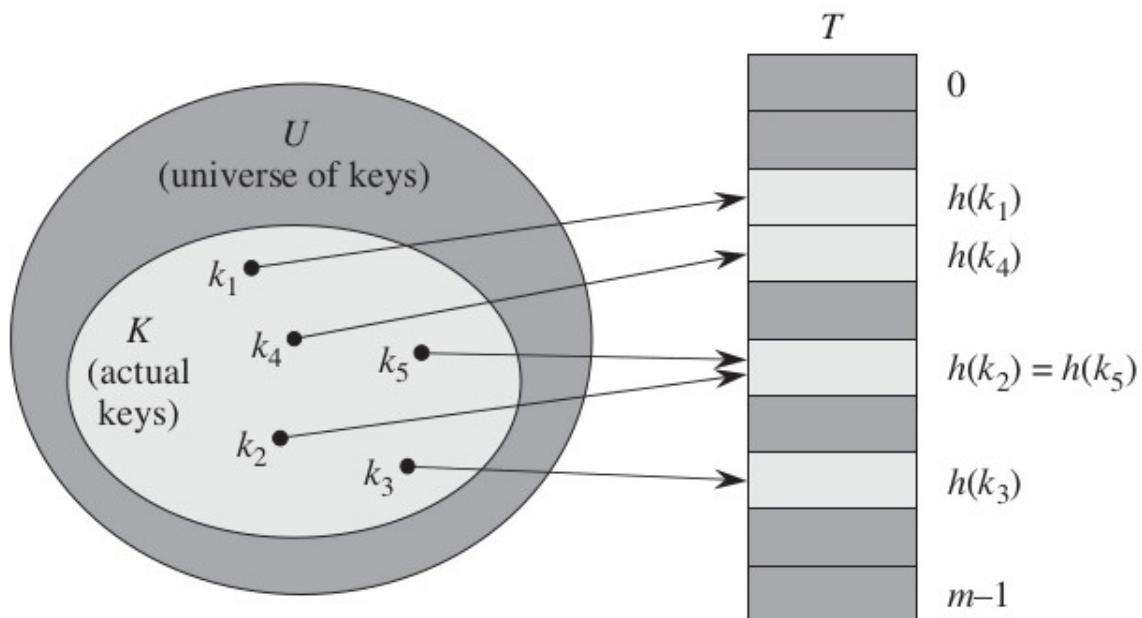
Além disso, nesse caso, o conjunto K de chaves ativas, ou seja, utilizadas de fato para o endereçamento, pode ser tão pequeno em relação ao conjunto U que a grande maioria do espaço de memória alocado para T seria um enorme desperdício.

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 $|U|$.



Porém, podem ocorrer colisões! Duas chaves distintas mapeadas para o mesmo slot!

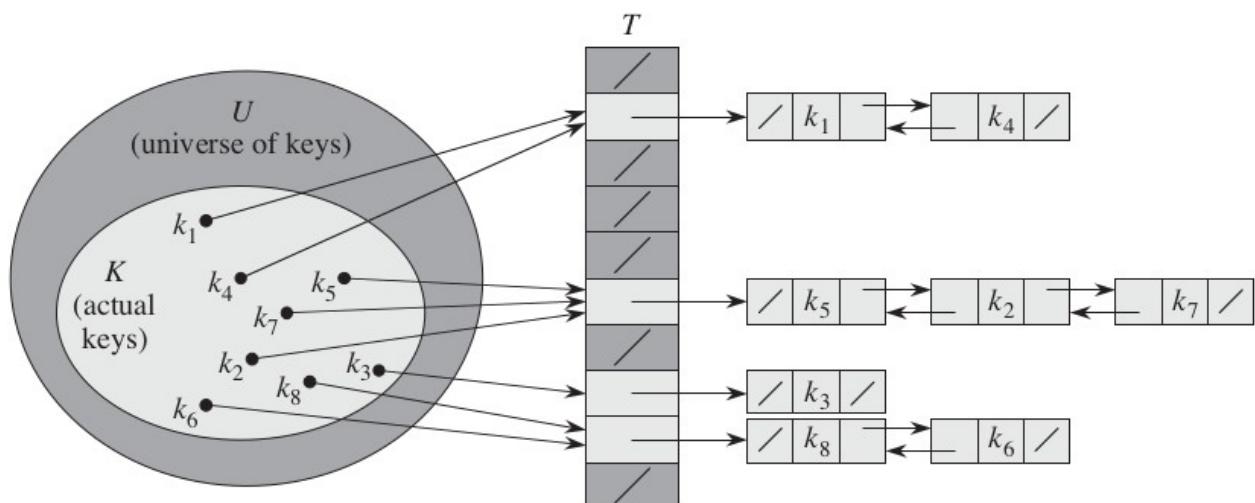
A ideia é fazer a função hash h parecer o mais “aleatória” possível, de modo que o espalhamento das chaves nos slots da tabela seja quase uniforme.

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 um ponteiro 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:

Chained-Hash-Insert(T, x)

Insira x na cabeça da lista encadeada em $T[h(x.key)]$

Chained-Hash-Search(T, k)

Busque por um elemento com chave k na lista encadeada $T[h(k)]$

Chained-Hash-Delete(T, x)

Remova x da lista encadeada $T[h(x.key)]$

Note que em termos de complexidade temos os seguintes fatos:

- A inserção de uma nova chave tem complexidade $O(1)$.
- 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 dos

ponteiros não necessite de um ponteiro auxiliar (a primitiva Chained-Hash-Delete assume que já temos uma referência para o nó a ser removido).

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:

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

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 cálculos 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 Tabela de Espalhamento em que as colisões são tratadas por encadeamentos, uma busca a um elemento que não pertence a estrutura tem complexidade $O(1+\alpha)$, 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 conteém (quanto maior, mais provável)

Teorema: Em uma Tabela de Espalhamento em que as colisões são tratadas por encadeamentos, uma busca a um elemento que pertence a estrutura tem complexidade $O(1+\alpha)$, 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:

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

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

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

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:

$$\begin{array}{r} 100 \\ | \\ 12 \\ | \\ \hline 4 & 8 \end{array}$$

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 é 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 :

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

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 has neste caso é definida como:

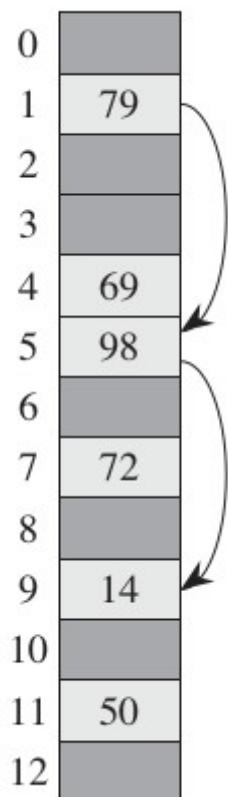
$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$$

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)
- c) Duplo espalhamento (double hashing)

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:

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

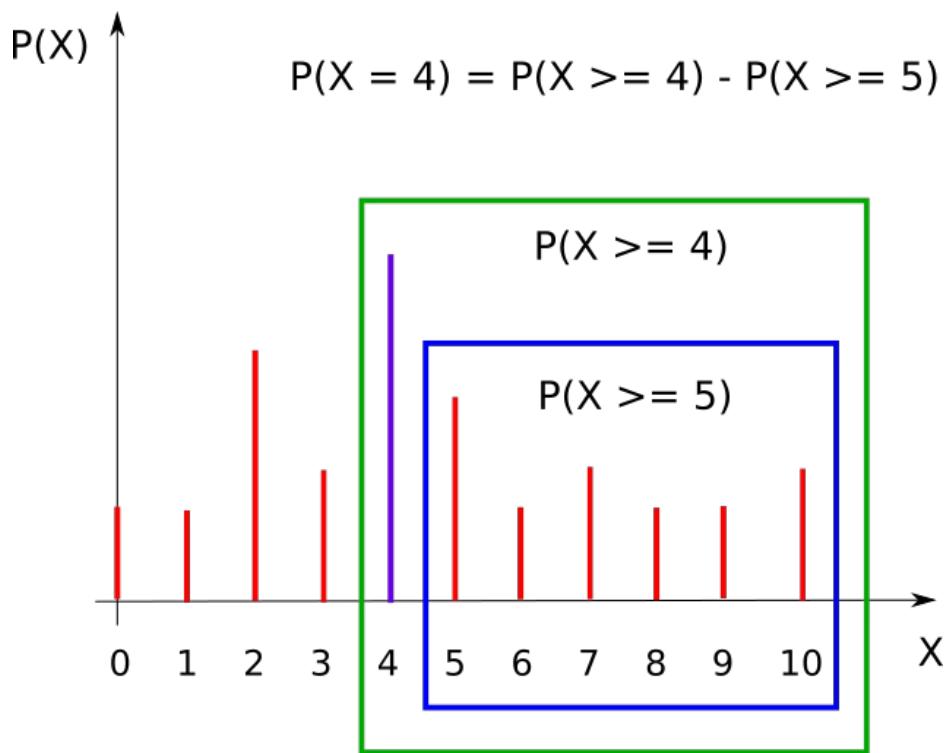
pois como $n < m$, temos $\frac{(n-j)}{(m-j)} \leq \frac{n}{m}$ para $0 \leq j \leq m$.

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 4 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

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

assumindo hashing uniforme.

Prova:

Note que a busca ela 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:

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

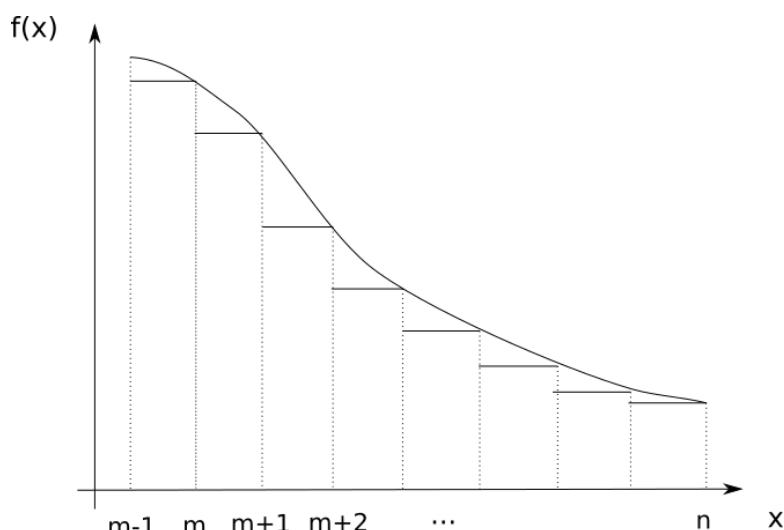
Desa forma, o somatório pode ser expresso como:

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

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 1.387
- Se a Tabela Hash está 90% cheia, número esperado de sondagens é aproximadamente 2.559

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 calcular essa probabilidade, primeiramente, é necessário calcular o número total de possíveis pares de chaves. Da análise combinatória, sabemos que o número total de maneiras possíveis de selecionar pares de elementos a partir de um conjunto de n elementos é dado por:

$$\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)(n-2)!}{(n-2)!2} = \frac{n(n-1)}{2}$$

Assumindo a hipótese de hashing uniforme, cada par tem a mesma probabilidade de colidir, igual a $1/m$. A probabilidade de ocorrer pelo menos uma colisão é dada por:

$$P(\# c \geq 1) = 1 - P(\# c = 0)$$

Note que a probabilidade de um par arbitrário não colidir é:

$$1 - \frac{1}{m}$$

o que implica que a probabilidade de não haver colisão em nenhum dos possíveis pares é:

$$P(\# c = 0) = \left(1 - \frac{1}{m}\right) \times \left(1 - \frac{1}{m}\right) \times \left(1 - \frac{1}{m}\right) \times \dots \times \left(1 - \frac{1}{m}\right) = \left(1 - \frac{1}{m}\right)^{\frac{n(n-1)}{2}}$$

Portanto, a probabilidade de ocorrer ao menos uma colisão é:

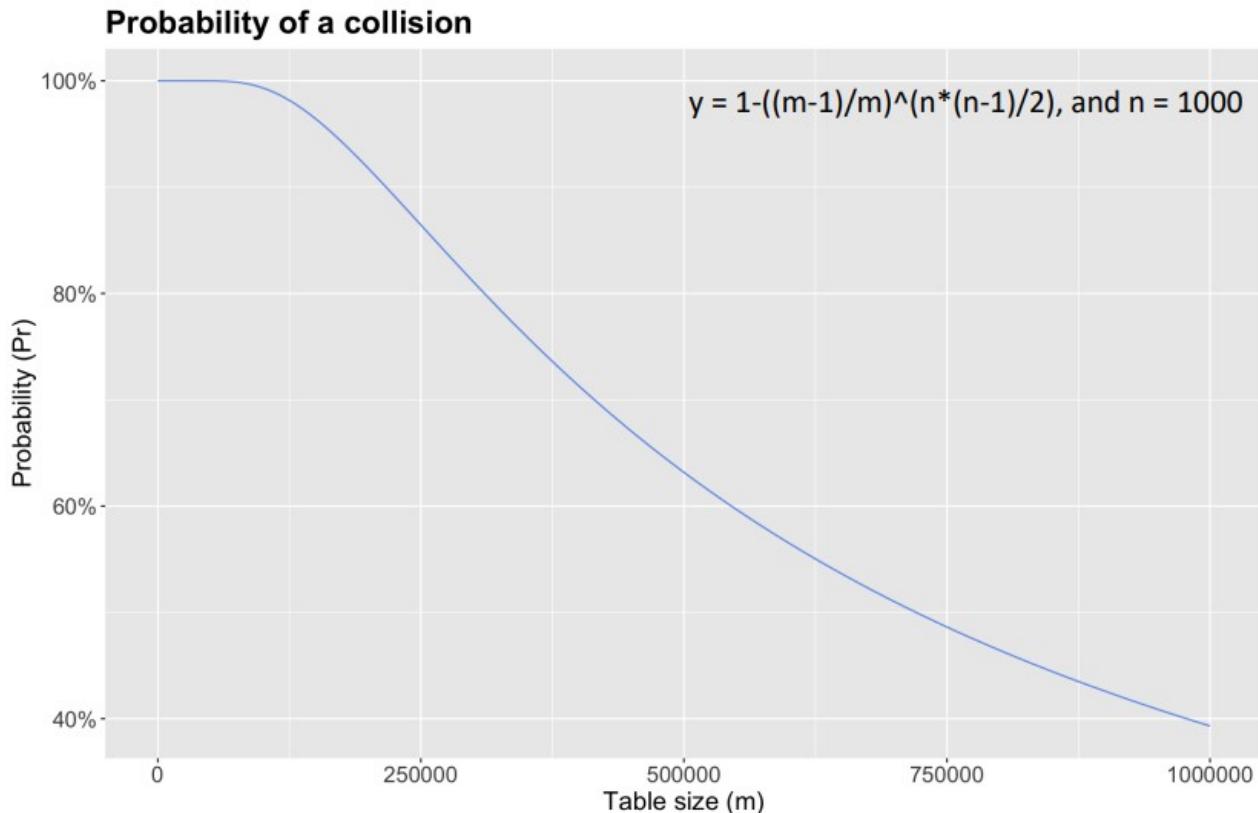
$$P(\# c \geq 1) = 1 - \left(1 - \frac{1}{m}\right)^{\frac{n(n-1)}{2}}$$

Outra pergunta interessante é: quanto grande a Tabela de Espalhamento precisa ser para termos uma probabilidade de colisão pequena?

Suponha $n = 1000$ e $m = 100000$, ou seja, temos um conjunto de mil chaves e um total de 100 mil slots. Substituindo os valores na equação anterior, chega-se a:

$$P(\# c \geq 1) = 0.9932$$

o que significa que a chance de termos colisões é praticamente 100%. O gráfico a seguir ilustra o comportamento da probabilidade de colisões para um conjunto de 1000 chaves:



Note que para a probabilidade de colisão ser menor que 50% precisamos ter cerca de 800 mil slots!

Se adotarmos a estratégia de hashing universal, ou seja, escolher funções hash aleatoriamente a partir de uma família de 100 funções, a probabilidade de que em todas elas ocorram colisões é:

$$0.9932^{100} = 0.507$$

Isso significa que há 50% de chances de uma delas não cause colisões. Seria necessário considerar uma família de 340 funções hash aleatórias para chegar próximo a 90% de chances de uma delas não causar colisões. Portanto, essa estratégia não é a mais eficiente para atingir um hashing perfeito (sem colisões).

Portanto, precisamos pensar em formas mais eficientes de garantir de não haja colisões.

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 .

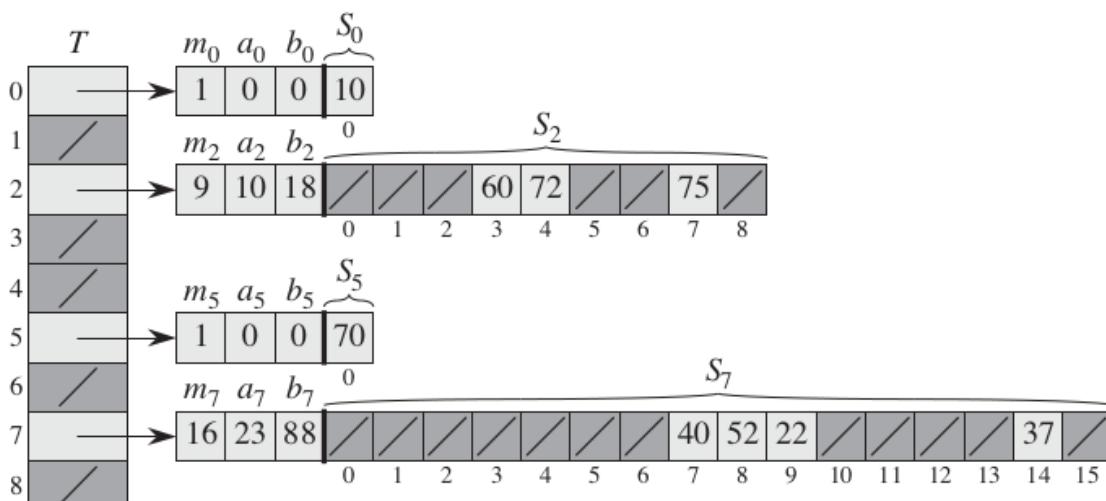
Considere o seguinte exemplo: desejamos utilizar o perfect hashing para armazenar o conjunto estático de chaves a seguir: $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. A função de hash externa (global) é dada por:

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

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:

$$h_j(k) = ((a_{jk} + b_j) \bmod p) \bmod m_j$$

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.



Uma 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 $h_j(k)$

$$h_j(k) = ((a_{jk} + b_j) \bmod p) \bmod m_j$$

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 a seguir.

$$K = \{8, 22, 36, 75, 61, 13, 84, 58\}$$

$$p = 87, a = 64, b = 5$$

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

$$h_j(k) = ((a_j k + b_j) \% p) \% m$$

0	
1	
2	
3	
4	
5	
6	
7	

Step 1: Randomly generate values for a and b , select $p > K$

Step 2: For each key k , work out home cell $h(k)$. Keep a count.

Step 3: Check if $\sum_{j=0}^{m-1} n_j^2 < 2n$

Step 4: Populate sub-tables, calculating new hash functions

$$K = \{8, 22, 36, 75, 61, 13, 84, 58\}$$

$$p = 87, a = 64, b = 5$$

		Count
0	/	0
1	/	61
2	/	8, 84
3	/	0
4	/	75
5	/	22
6	/	13
7	/	36, 58

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

$$h_j(k) = ((a_jk + b_j)\%p)\%m$$

Step 1: Randomly generate values for a and b , select $p > K$

Step 2: For each key k , work out home cell $h(k)$. Keep a count.

Step 3: Check if $\sum_{j=0}^{m-1} n_j^2 < 2n$

Step 4: Populate sub-tables, calculating new hash functions

$$K = \{8, 22, 36, 75, 61, 13, 84, 58\}$$

$$p = 87, a = 64, b = 5$$

		Count
0	/	0
1	/	61
2	/	8, 84
3	/	0
4	/	75
5	/	22
6	/	13
7	/	36, 58

Step 1: Randomly generate values for a and b , select $p > K$

Step 2: For each key k , work out home cell $h(k)$. Keep a count.

Step 3: Check if $\sum_{j=0}^{m-1} n_j^2 < 2n$

Step 4: Populate sub-tables, calculating new hash functions

$$\sum_{j=0}^{m-1} n_j^2 = 0 + 1 + 4 + 0 + 1 + 1 + 1 + 4 = 12$$

$$12 < 2 * n = 16 \quad \checkmark$$

$$K = \{8, 22, 36, 75, 61, 13, 84, 58\}$$

$$p = 87, a = 64, b = 5$$

		m_j	a_j	b_j	
0	/				
1		1	0	0	61
2		4	82	53	/ 84 / 8
3	/				
4		1	0	0	75
5		1	0	0	22
6		1	0	0	13
7		4	10	54	/ 58 / 36 /

Step 4: Populate sub-tables, calculating new hash functions

"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

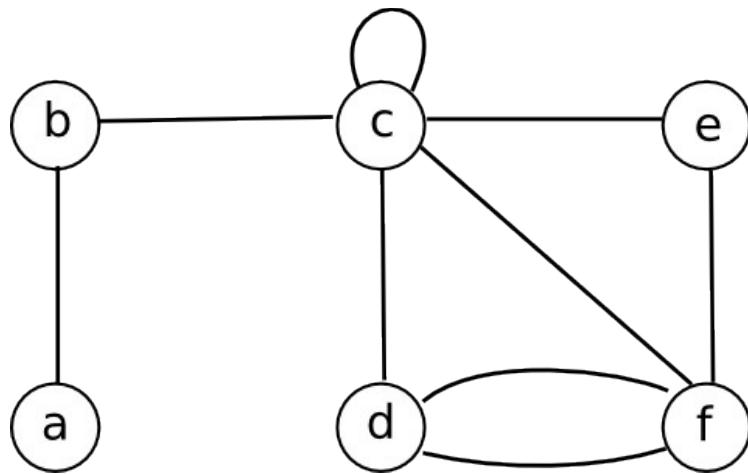
Grafos: Fundamentos básicos

Grafos são estruturas matemáticas que representam relações binárias entre elementos de um conjunto finito. 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 sistemas complexos.

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 (não precisa ser apenas equivalência ou ordem parcial, pode ser qualquer coisa): conjunto de **arestas**

A figura a seguir ilustra um grafo $G = (V, E)$ arbitrário.



Grafo ou Multigrafo

- Existem loops
- Existem arestas paralelas

Grafo básico simples

- Não existem loops
- Não existem arestas paralelas

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: $d(a) = 1$, $d(b) = 2$, $d(c) = 6$, $d(d) = 3$, ...

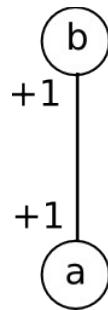
Def: A lista de graus de $G = (V, E)$ é a lista que armazena os graus dos vértices em ordem crescente.

$$L_G = (1, 2, 2, 3, 4, 6)$$

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.



Cada aresta contribui com +1 para o grau de cada vértice

Prova: (por indução)

Seja o predicado $P(n)$ 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,

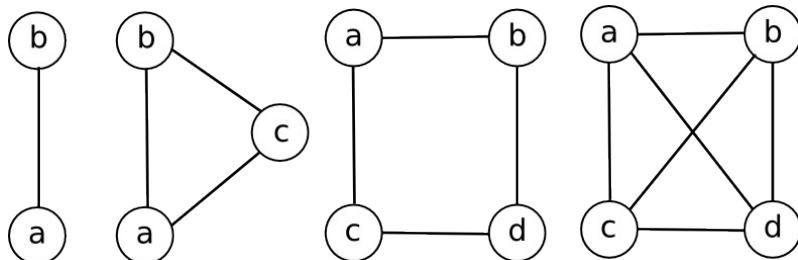
$$\sum_{i=1}^n d(v_i) = \sum_{v \in P} d(v) + \sum_{u \in I} d(u) = 2m$$

Isso implica em

$$\sum_{u \in I} d(u) = 2m - \sum_{v \in P} d(v)$$

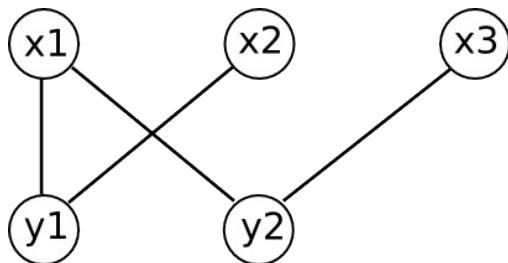
Como $2m$ é par e a soma de números pares é sempre par, resulta que a soma dos números ímpares também é par. Para que isso ocorra temos que ter $|I|$ par (número de elementos do conjunto I é par)

Def: G é k -regular $\Leftrightarrow \forall v \in V (d(v)=k)$, ou seja, $L_G = (k, k, k, k, \dots, k)$

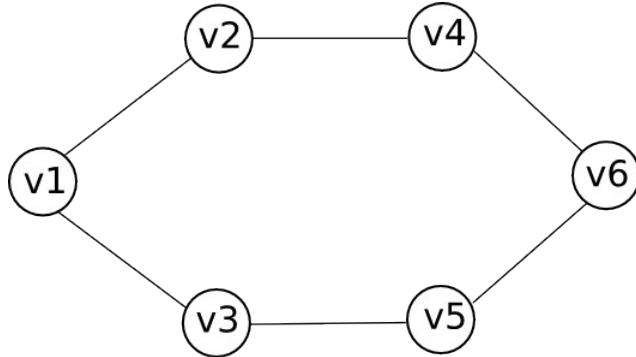


Def: Grafo Bipartido

$G = (V, E)$ é bipartido $\Leftrightarrow V = X \cup Y$ com $X \cap Y = \emptyset$ tal que
 $\forall e \in E (e = (a, b) / a \in X \wedge b \in Y)$



Considere o exemplo a seguir: o grafo a abaixo é bipartido ou não? Justifique sua resposta



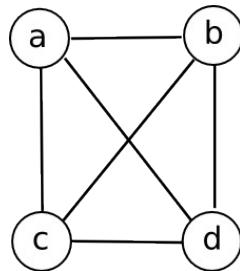
SIM, é bipartido!
Mesmo parecendo que não.

Como decidir se grafo G é bipartido? Seja $R = 0,1$ o conjunto de rótulos e seja $r \in R$ um rótulo arbitrário e $\bar{r} \in R$ o seu complementar, ou seja, $\bar{r} = 1 - r$.

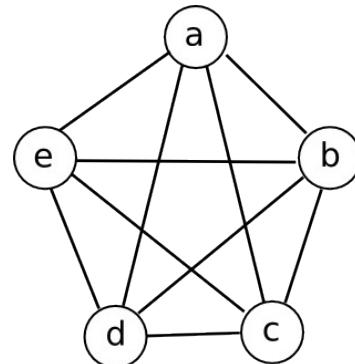
- 1) Escolha um vértice inicial v e rotule-o como r .
- 2) Para todos os vértices u vizinhos de v ainda não rotulados, rotule-os como \bar{r} . Ou seja, se um vértice recebe rótulo r , todos seus vizinhos não rotulados devem receber rótulo \bar{r} e vice-versa.
- 3) Condição de parada: Pare quando todos os vértices do grafo estiverem rotulados.
- 4) Se ao fim do processo toda aresta do grafo for do tipo (r, \bar{r}) então o grafo G é bipartido. Caso contrário, o grafo não é bipartido.

Def: Grafo completo

G é um grafo completo de n vértices, denotado por K_n , se cada vértice é ligado a todos os demais, ou seja, se $L_G = (n-1, n-1, n-1, \dots, n-1)$



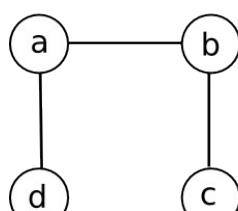
K_4



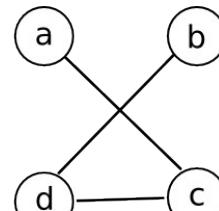
K_5

O número de arestas do grafo K_n é dado por $\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)}{2}$

Def: Complementar de um grafo G : $\bar{G} = K_n - G$



G



\bar{G}

Note que, pela definição de grafo complementar, temos $G + \bar{G} = K_n$.

Def: Subgrafo

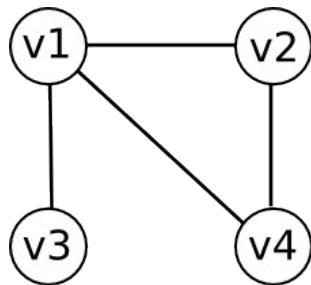
Seja $G = (V, E)$ um grafo. Dizemos que $H = (V', E')$ é um subgrafo de G se $V' \subseteq V$ e $E' \subseteq E$. Em outras palavras, é todo grafo que pode ser obtido a partir de G através de remoção de vértices e/ou arestas.

Representações computacionais de grafos

1) Matriz de adjacências A : matriz quadrada $n \times n$ definida como:

a) Grafos básicos simples

$$A_{i,j} = \begin{cases} 1, & i \leftrightarrow j \\ 0, & c.c \end{cases}$$

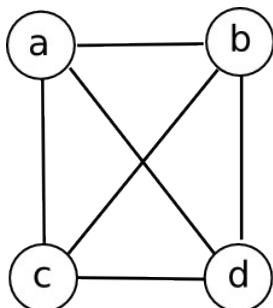


$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Propriedades básicas

- i) $\text{diag}(A) = 0$
- ii) matriz binária
- iii) $A = A^T$ (com exceção de dígrafos)
- iv) $\sum_j A_{i,j} = d(v_i)$
- v) Esparsa
- vi) $O(n^2)$ em espaço

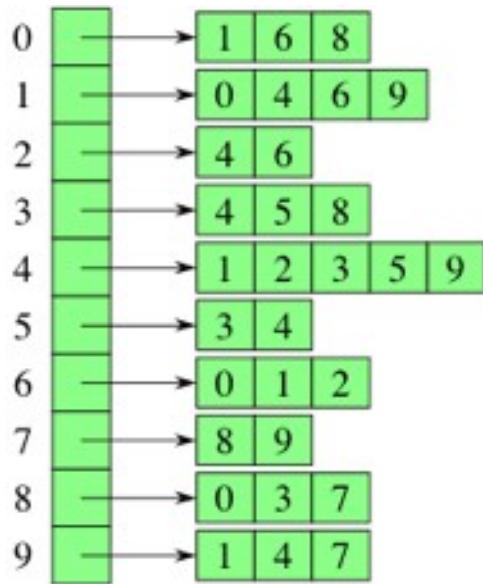
2) Matriz de Incidência M : matriz $n \times m$ em que as linhas referem-se aos vértices e as colunas referem-se as arestas



$$M = \begin{bmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \quad \begin{array}{l} a \\ b \\ c \\ d \end{array}$$

Note que loops são devem ser indicados pelo número 2 no vértice em questão

3) Lista de adjacências: estrutura do tipo hash-table (mais eficiente em termos de memória)



Em Python, na biblioteca NetworkX a classe Graph implementa grafos básicos simples e a classe MultiGraph implementa multigrafos (permitem arestas paralelas).

A estrutura de dados básica utilizada é um dicionário de dicionários de dicionários, que simula uma tabela hash na forma de uma lista de adjacências. As chaves do dicionário são os nós de modo que $G[u]$ retorna um dicionário cuja chave é o extremo da respectiva aresta e o campo valor é um dicionário para os atributos da aresta. A expressão $G[u][v]$ retorna o dicionário que armazena os atributos da aresta (u,v) . O código em Python a seguir mostra como podemos criar e manipular um grafo utilizando a biblioteca NetworkX.

```
# Classes e funções para manipulação de grafos
import networkx as nx
# Classes e funções para plotagem de gráficos
import matplotlib.pyplot as plt

# Cria grafo vazio
G = nx.Graph()

# Adiciona vértices
G.add_node('v1')
G.add_node('v2')
G.add_node('v3')
G.add_node('v4')
G.add_node('v5')

# Adiciona arestas
G.add_edge('v1', 'v2')
G.add_edge('v2', 'v3')
G.add_edge('v3', 'v4')
G.add_edge('v4', 'v5')
G.add_edge('v5', 'v1')
G.add_edge('v2', 'v4')
```

```

# Lista os vértices
print('Lista de vértices')
print(G.nodes())
input()

# Percorre o conjunto de vértices
print('Percorrendo os vértices')
for v in G.nodes():
    print(v)
input()

# Lista as arestas
print('Lista de arestas')
print(G.edges())
input()

# Percorre o conjunto de arestas
print('Percorrendo as arestas')
for e in G.edges():
    print(e)
input()

# Mostra a lista de graus
print('Lista de graus de G')
print(G.degree())
input()
# Acessa o grau do vértice v2
print('O grau do vértice v2 é %d' %G.degree()['v2'])
print()

# Grafo como lista de adjacências
print('Grafo como lista de adjacências')
print(G['v1'])
print(G['v2'])
print(G['v3'])
print(G['v4'])
print(G['v5'])
input()

# Obtém a matriz de adjacências do grafo G
print('Matriz de adjacências de G')
A = nx.adjacency_matrix(G)          # retorna uma matriz esparsa para
economizar memória
print(A.todense())                 # converte para matriz densa (padrão)

# Adiciona um campo peso em cada aresta do grafo
G['v1']['v2']['peso'] = 5
G['v2']['v3']['peso'] = 10
G['v3']['v4']['peso'] = 2
G['v4']['v5']['peso'] = 7
G['v5']['v1']['peso'] = 4
G['v2']['v4']['peso'] = 8

```

```
# Lista cada aresta e seus respectivos pesos
print('Adicionando pesos as arestas')
for edge in G.edges():
    u = edge[0]
    v = edge[1]
    print('O peso da aresta', edge, 'vale ', G[u][v]['peso'])
input()
print()

print('Plotando o grafo como imagem...')

plt.figure(1)
# Há vários layouts, mas spring é um dos mais bonitos
nx.draw_networkx(G, pos=nx.spring_layout(G), with_labels=True)
plt.show()
```

Um excelente guia de referência oficial para a biblioteca NetworkX pode ser encontrada em:
https://networkx.github.io/documentation/stable/_downloads/networkx_reference.pdf

O leitor poderá encontrar diversos exemplos práticos de inúmeras funções da biblioteca.

"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.

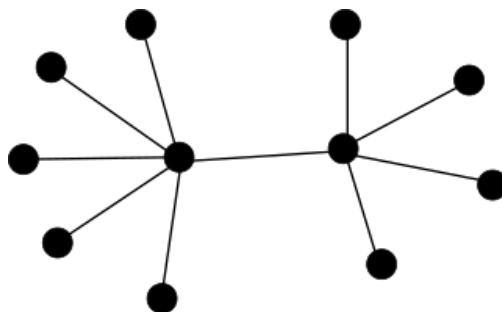
Objetivo: acessar/recuperar todos os elementos do conjunto V

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.



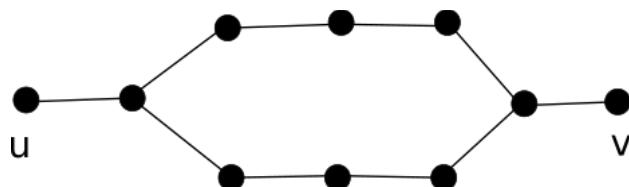
Teorema: G é uma árvore $\Leftrightarrow \exists$ um único caminho entre quaisquer 2 vértices $u, v \in V$

(ida) $p \rightarrow q = !q \rightarrow !p$

\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.

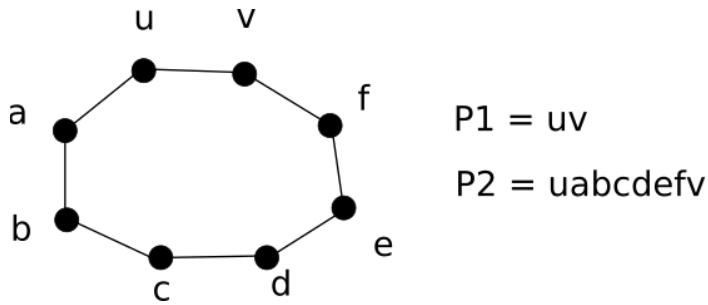


Porém neste caso temos a formação de um ciclo e portanto G não pode ser árvore.

(volta) $q \rightarrow p = !p \rightarrow !q$

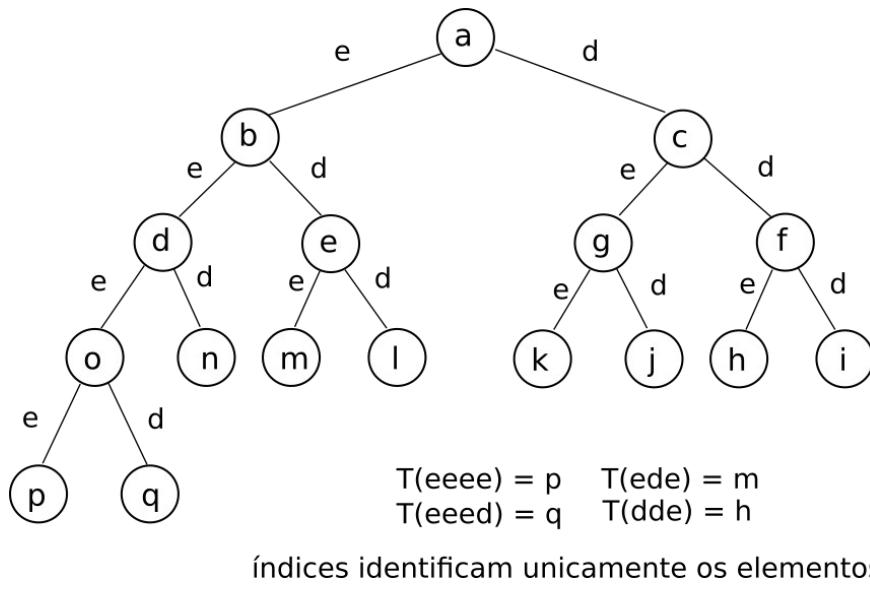
G não é árvore $\rightarrow \nexists$ único caminho entre quaisquer u, v

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

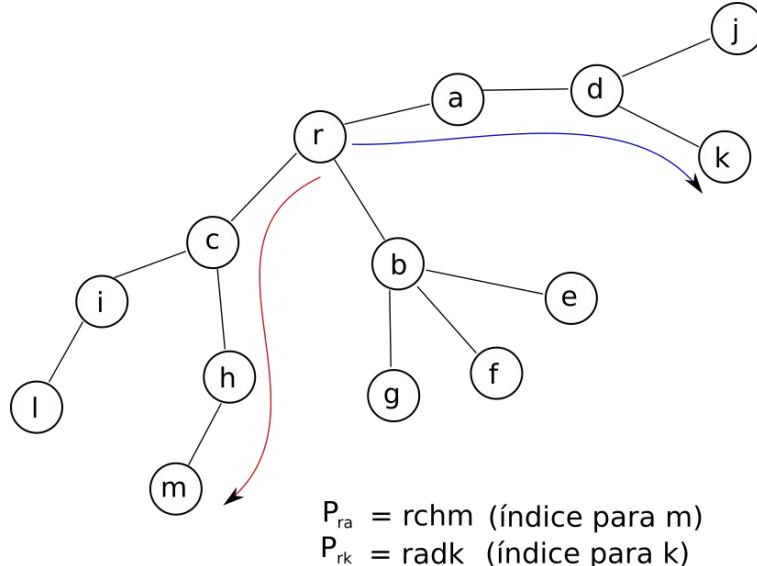


Relação entre busca em grafos e árvores

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 árvore existe um único caminho entre 2 vértices u, v . Considere uma árvore binária



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.

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

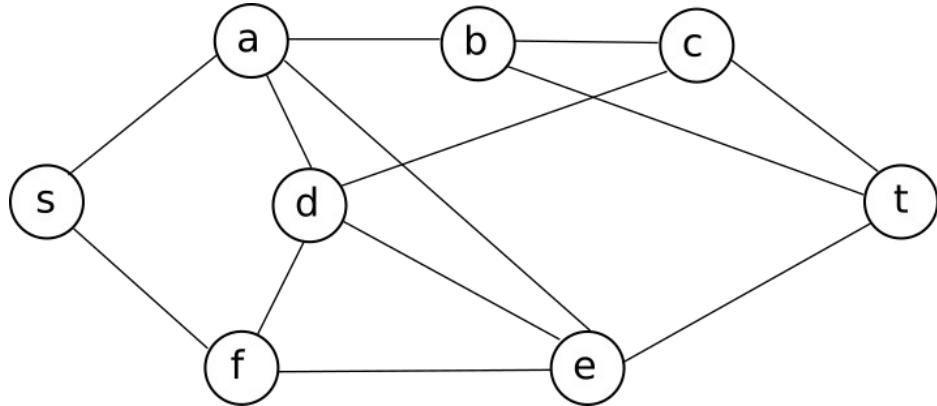
Definição das variáveis usadas no algoritmo

- 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.



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	---	---

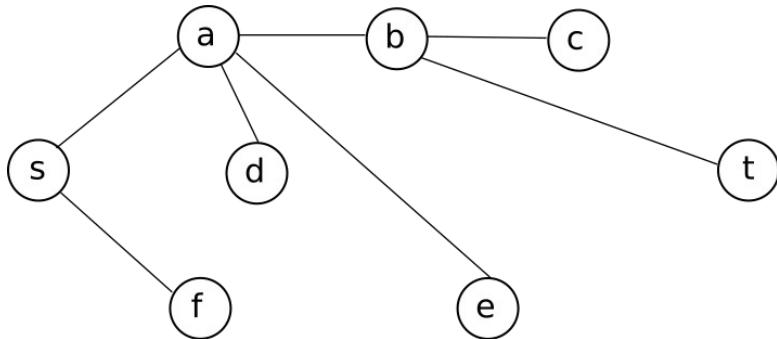
FILA

$$\begin{aligned}
 Q^{(0)} &= [s] \\
 Q^{(1)} &= [a, f] \\
 Q^{(2)} &= [f, b, d, e] \\
 Q^{(3)} &= [b, d, e] \\
 Q^{(4)} &= [d, e, c, t] \\
 Q^{(5)} &= [e, c, t] \\
 Q^{(6)} &= [c, t] \\
 Q^{(7)} &= [t] \\
 Q^{(8)} &= \emptyset
 \end{aligned}$$

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	



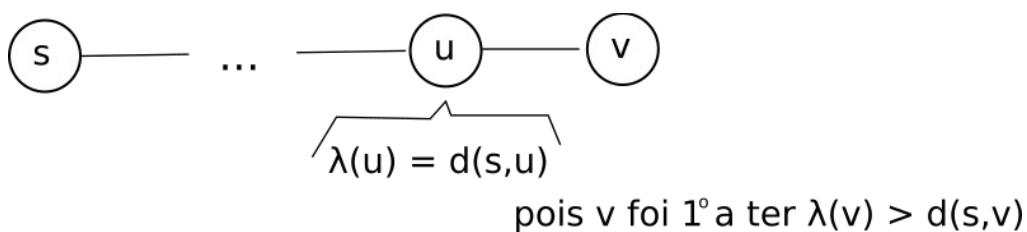
Note que a árvore nada mais é que a união dos caminhos mínimos de s (origem) a qualquer um dos vértices do grafo (destinos). A BFS-tree geralmente não é única, porém todas possuem a mesma profundidade (mínima distância da raiz ao mais distante)

Pergunta: Como podemos implementar um script para computar o diâmetro de um grafo G usando a BFS? Pense em termos do cálculo da excentricidade de cada vértice.

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

$$\lambda(v) > d(s, v) = d(s, u) + 1 = \lambda(u) + 1$$

$$(2) \quad (6) \quad (5)$$

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, $u = \text{pop}(Q)$

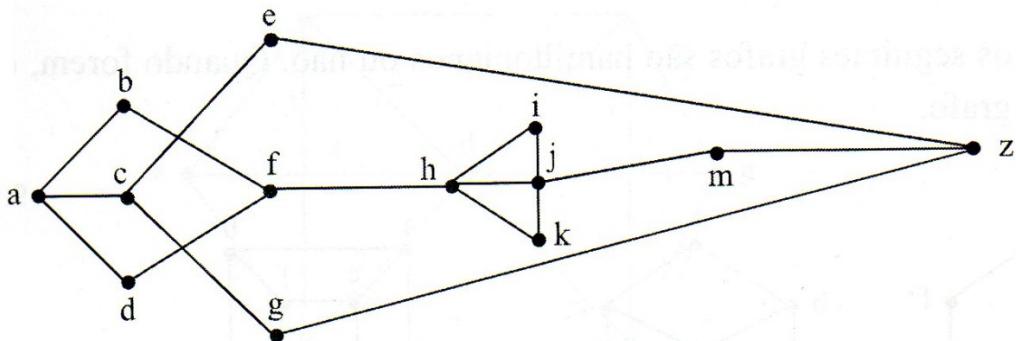
i) v é WHITE: $\lambda(v) = \lambda(u) + 1$ (contradição)

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 $\lambda(v) > d(s, v)$.

Exercício: Obtenha a BFS-Tree do grafo a seguir. Qual a profundidade da árvore. Obtenha o número de caminhos mínimos de a até m .



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)

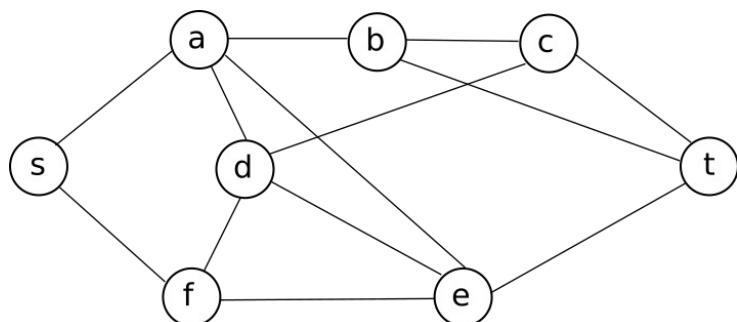
```
DFS(G, s) {
    for each u ∈ V {    # inicializa as variáveis
        u.color = WHITE
        π(v) = NIL
    }
    time = 0             # variável global para contar o tempo
    for each u ∈ V {
        if u.color == WHITE
            DFS_visit(G, u)
    }
}

# 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	$V' = \{ v \in N(u) / v.\text{color} = \text{WHITE}\}$	$\pi(u)$	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)

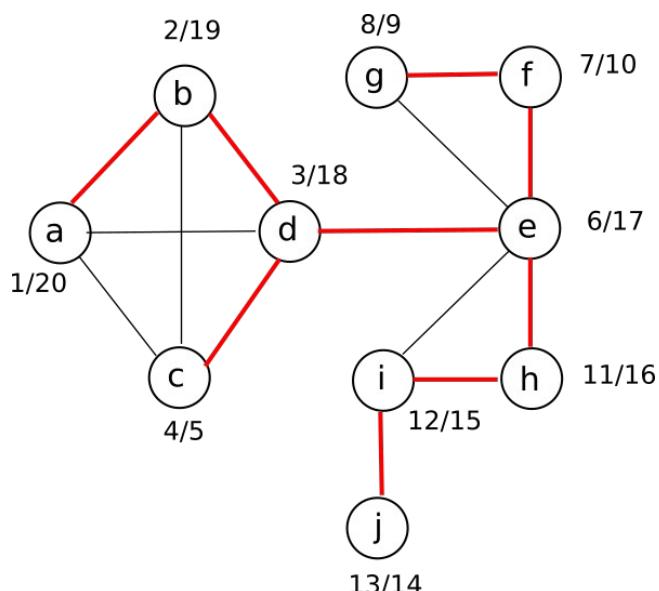
1) A rotulação tem o seguinte significado:

- 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.



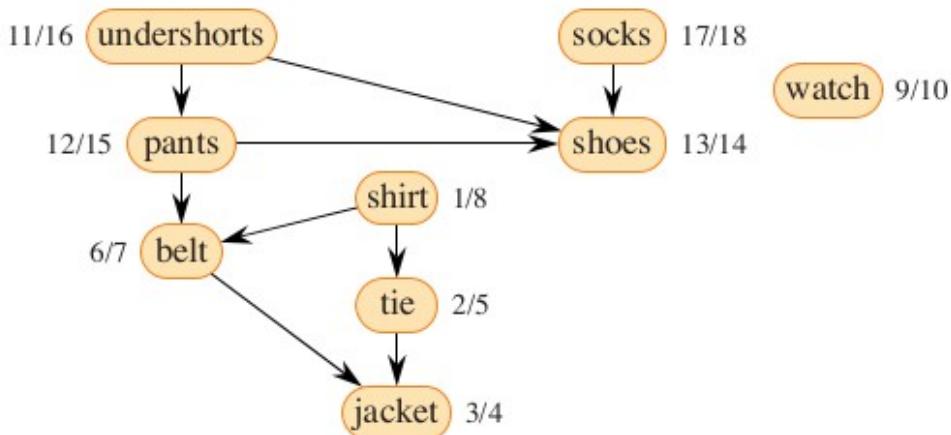
Perguntas:

- 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.

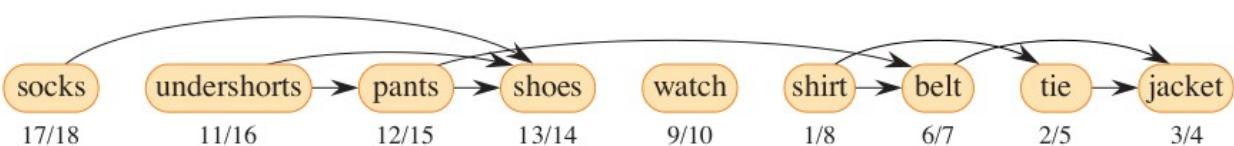
Uma aplicação da ordenação topológica seria a seguinte: imagine que você queira “ensinar” um robô humanoide a se vestir. Claramente, existe uma ordem natural que precisa ser seguida durante o processo. Por exemplo, não podemos calçar o sapato e depois colocar as meias. Suponha que o grafo acíclico direcionado 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.

(ida) $p \rightarrow q \equiv \neg q \rightarrow \neg p$

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 .

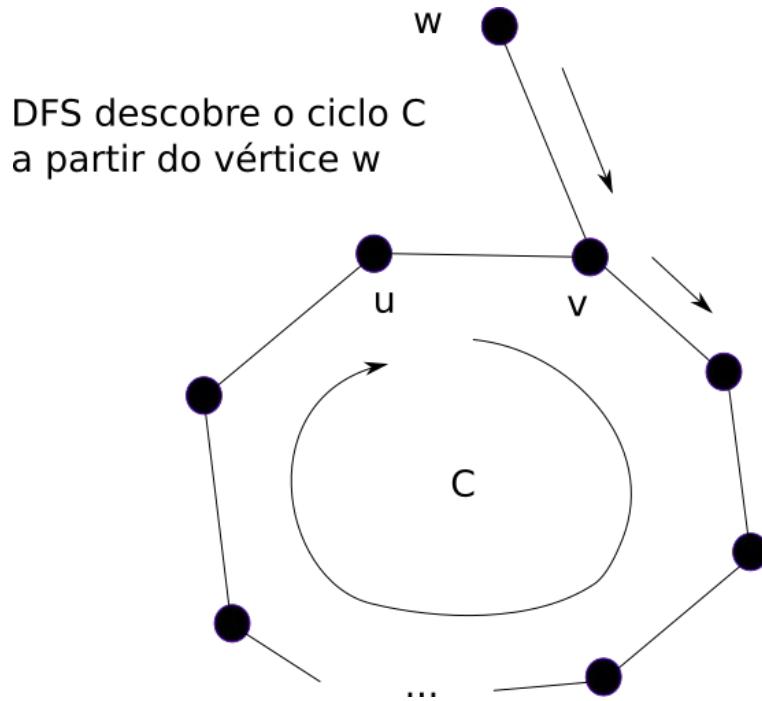
3. Mas, na DFS_tree deve existir um caminho de v até u (pois é uma árvore), o que implica na geração de um ciclo.

(volta) $q \rightarrow p = \neg p \rightarrow \neg q$

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.

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

$$w(P^*) = \sum_{i=0}^{n-1} w(v_i, v_{i+1}) = w(v_0, v_1) + w(v_1, v_2) + \dots + w(v_{n-1}, v_n) \quad (\text{soma dos pesos das arestas})$$

é o menor possível.

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

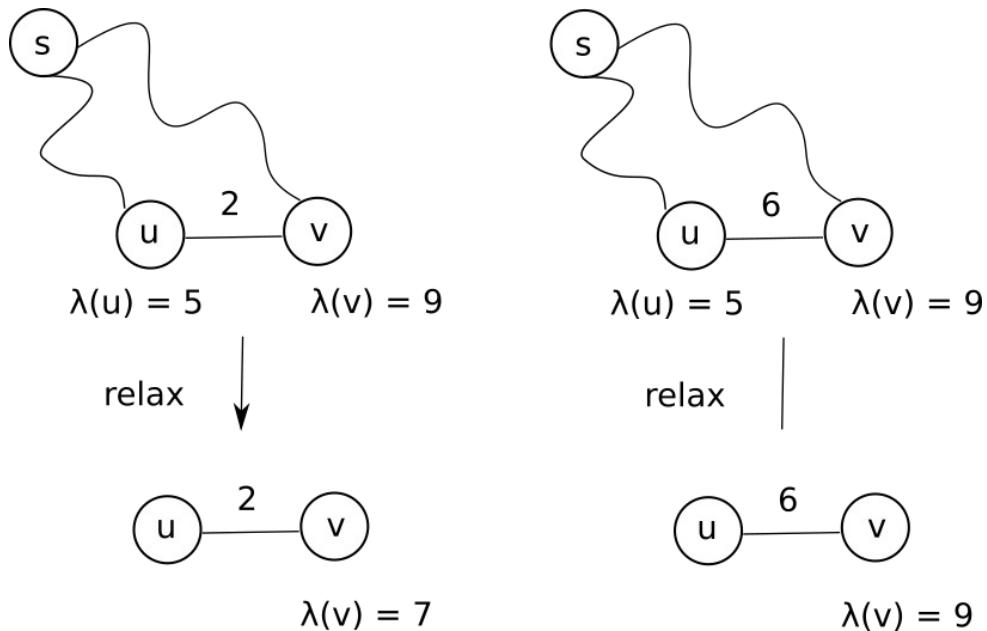
$\text{relax}(u, v, w)$: relaxar a aresta (u, v) de peso w

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 $\lambda(v)$)

ALGORITMO

```

Dijkstra(G, w, s)
{
    for each v  $\in V$  {
         $\lambda(v) = \infty$ 
         $\pi(v) = NIL$ 
    }
     $\lambda(s) = 0$ 
    Q = V # fila de prioridades
    while Q  $\neq \emptyset$  {
        u = ExtractMin(Q) # extrai vértice de menor  $\lambda(v)$ 
        S = S  $\cup \{u\}$ 
        for each v  $\in N(u)$ 
            relax(u, v, w) # relaxa toda aresta incidente a u
    }
}

```

Algoritmos em grafos e suas estruturas de dados

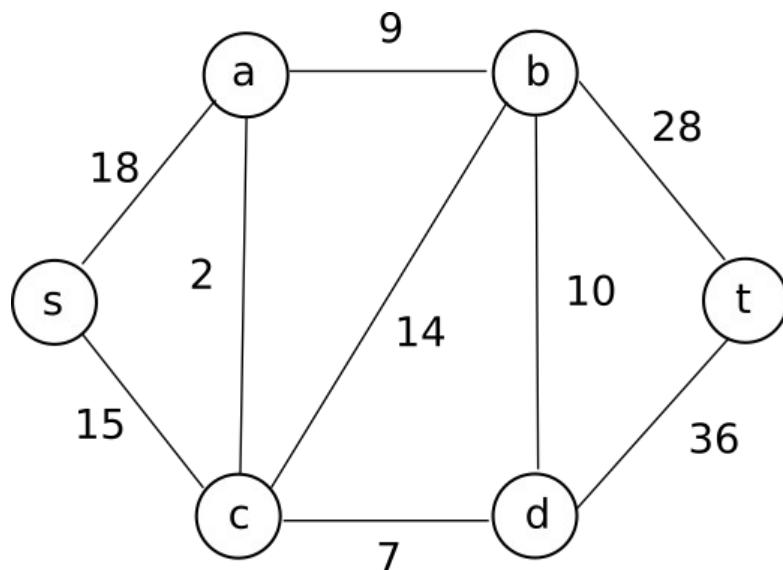
Busca em Largura (BFS) – Fila

Busca em Profundidade (DFS) – Pilha

Algoritmo de Dijkstra – Fila de prioridades

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 tamanho, já no algoritmo de Dijkstra esse tamanho 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	

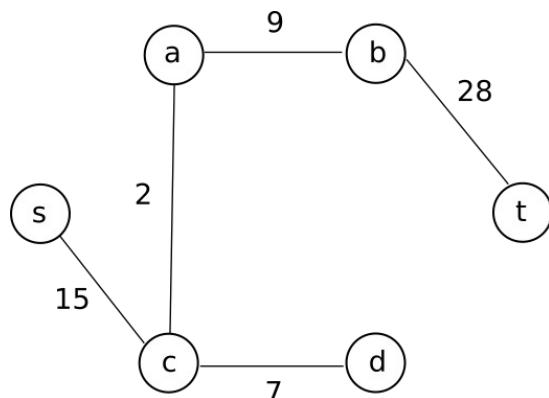
Ordem de acesso aos vértices

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}	$\lambda(c) = \min\{\lambda(c), \lambda(s) + w(s, c)\} = \min\{\infty, 15\} = 15$	$\pi(c) = s$
a	{b}	$\lambda(a) = \min\{\lambda(a), \lambda(c) + w(c, a)\} = \min\{18, 17\} = 17$	$\pi(a) = c$
b	{t}	$\lambda(b) = \min\{\lambda(b), \lambda(c) + w(c, b)\} = \min\{\infty, 29\} = 29$	$\pi(b) = c$
d	{b, t}	$\lambda(d) = \min\{\lambda(d), \lambda(c) + w(c, d)\} = \min\{\infty, 22\} = 22$	$\pi(d) = c$
t	\emptyset	$\lambda(b) = \min\{\lambda(b), \lambda(a) + w(a, b)\} = \min\{29, 26\} = 26$	$\pi(b) = a$
		$\lambda(b) = \min\{\lambda(b), \lambda(d) + w(d, b)\} = \min\{26, 32\} = 26$	---
		$\lambda(t) = \min\{\lambda(t), \lambda(d) + w(d, t)\} = \min\{\infty, 58\} = 58$	$\pi(t) = d$
		$\lambda(t) = \min\{\lambda(t), \lambda(b) + w(b, t)\} = \min\{58, 54\} = 54$	$\pi(t) = b$
		---	---

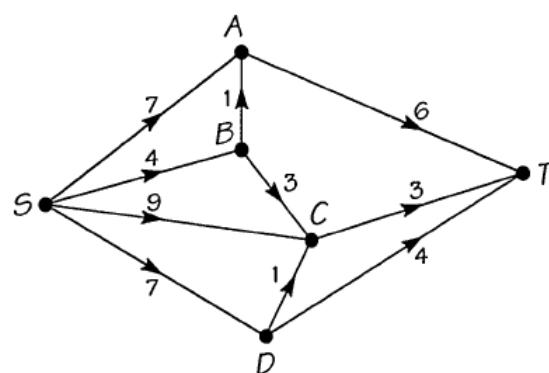
Mapa de predecessores (árvore final)

v		s		a		b		c		d		t	
$\pi(v)$		---		c		a		s		c		b	

Árvore de caminhos mínimos (armazena os menores caminhos de s a todos os demais vértices)



Exercício: Utilizando o algoritmo de Dijkstra, construa a árvore de caminhos mínimos, mostrando o trace completo.

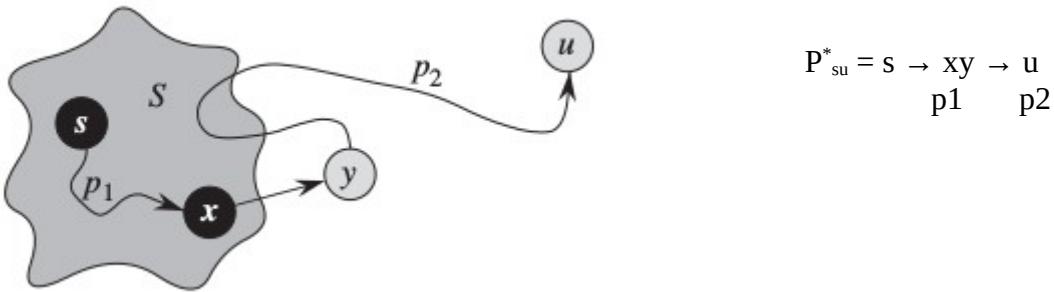


A seguir iremos demonstrar a otimalidade do algoritmo de Dijkstra.

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 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 a 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 se utiliza-se estruturas de dados estáticas ou dinâmicas.

* Caso 1: $G = (V, E)$ representado por uma matriz de adjacências e fila de prioridades Q representada por um array estático de n elementos (acesso direto)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$
- 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 vetor)
- 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 e fila de prioridades Q representada por um heap binário (estruturas dinâmicas)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(\log n)$ (árvore binária)
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = \text{ExtractMin}(Q)$ é $O(\log n)$ (busca em árvore binária)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(\log n)$ (árvore binária)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(\log n) + O(n)*O(\log n) + (O(1) + O(\log n))*(d(v_1) + d(v_2) + \dots + d(v_n))$$

De modo similar ao caso anterior, podemos escrever:

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

Como os dois últimos termos dominam 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!

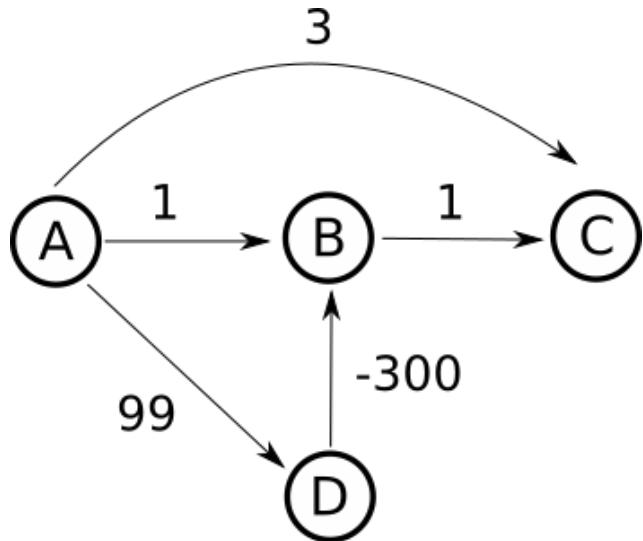
Em grafos mais densos (m muito grande), o caso 1 é mais eficiente.

Em grafos menos densos (poucas arestas), o caso 2 é mais eficiente.

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. Para essa situação, o algoritmo Bellman-Ford é a melhor alternativa.

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.

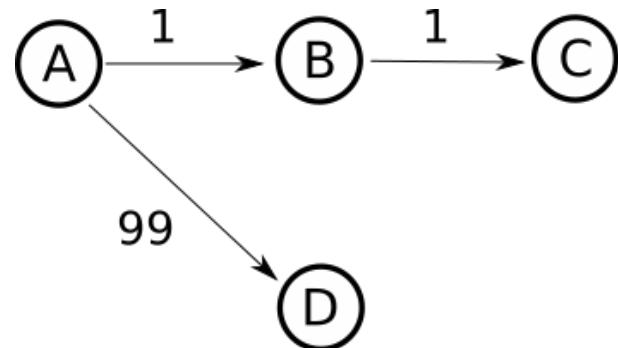
Fila

	a	b	c	d
$\lambda^{(0)}(v)$	0	∞	∞	∞
$\lambda^{(1)}(v)$		1	3	99
$\lambda^{(2)}(v)$			1	99
$\lambda^{(3)}(v)$				99

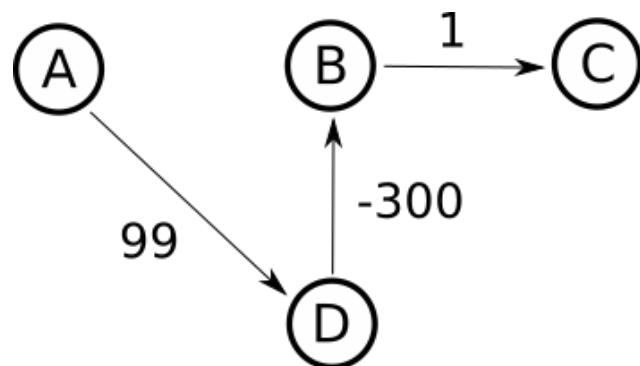
Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
a	{b, c, d}	$\lambda(b) = \min\{\infty, 1\} = 1$ $\lambda(c) = \min\{\infty, 3\} = 3$ $\lambda(d) = \min\{\infty, 99\} = 99$	$\pi(b) = a$ $\pi(c) = a$ $\pi(d) = a$
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 geral: 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

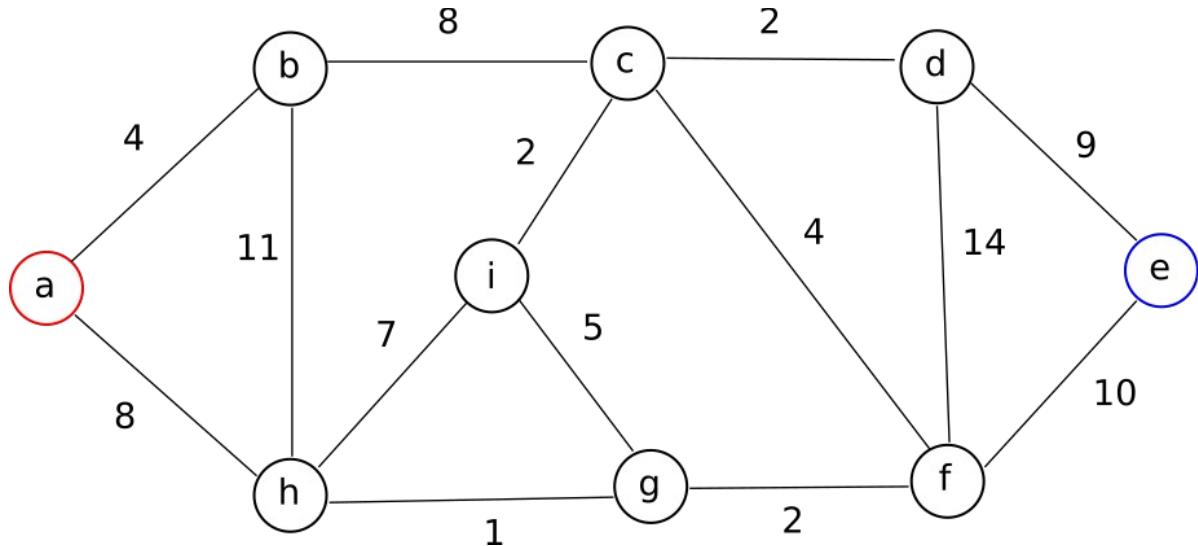
Ideia: utilizar múltiplas sementes/raízes.

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

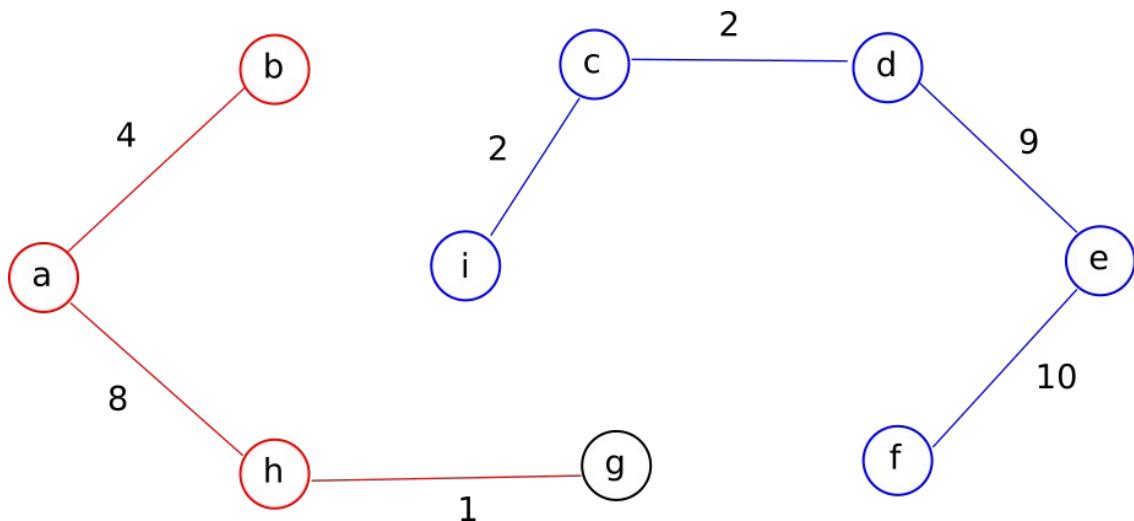
	a	b	c	d	e	f	g	h	i
$\lambda^{(0)}(v)$	0	∞	∞	∞	0	∞	∞	∞	∞
$\lambda^{(1)}(v)$		4	∞	∞	0	∞	∞	8	∞
$\lambda^{(2)}(v)$		4	∞	9		10	∞	8	∞
$\lambda^{(3)}(v)$			12	9		10	∞	8	∞
$\lambda^{(4)}(v)$			12	9		10	9		15
$\lambda^{(5)}(v)$			11			10	9		15
$\lambda^{(6)}(v)$			11			10			14
$\lambda^{(7)}(v)$			11						14
$\lambda^{(8)}(v)$									13

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}	$\lambda(b) = \min\{\infty, 4\} = 4$ $\lambda(h) = \min\{\infty, 8\} = 8$	$\pi(b) = a$ $\pi(h) = a$
e	{d, f}	$\lambda(d) = \min\{\infty, 9\} = 9$ $\lambda(f) = \min\{\infty, 10\} = 10$	$\pi(d) = e$ $\pi(f) = e$
b	{c, h}	$\lambda(c) = \min\{\infty, 4+8\} = 12$ $\lambda(h) = \min\{8, 4+11\} = 8$	$\pi(c) = b$ -----

h	$\{i, g\}$	$\lambda(i) = \min\{\infty, 8+7\} = 15$	$\pi(i) = h$
d	$\{c, f\}$	$\lambda(g) = \min\{\infty, 8+1\} = 9$	$\pi(g) = h$
g	$\{f, i\}$	$\lambda(c) = \min\{12, 9+2\} = 11$	$\pi(c) = d$
f	$\{c\}$	$\lambda(f) = \min\{10, 9+14\} = 10$	-----
c	$\{i\}$	$\lambda(i) = \min\{10, 9+14\} = 10$	-----
i	\emptyset	$\lambda(i) = \min\{15, 9+5\} = 14$	$\pi(i) = g$
		$\lambda(c) = \min\{11, 10+4\} = 11$	-----
		$\lambda(i) = \min\{14, 11+2\} = 13$	$\pi(i) = c$
		-----	-----

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

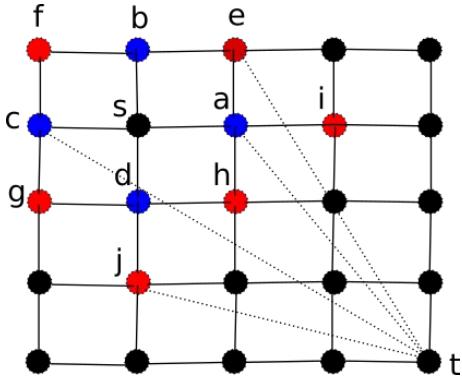
$$\alpha(v) = \lambda(v) + \gamma(v) \quad \text{onde}$$

$\lambda(v)$: 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:

$$\begin{aligned}\gamma(a) &= \gamma(d) = \sqrt{4+9} = \sqrt{13} \\ \gamma(b) &= \gamma(c) = \sqrt{9+16} = \sqrt{25} = 5\end{aligned}$$

Ou seja, no A*, devemos priorizar a e d em detrimento de b e c uma vez que

$$1 + \sqrt{13} < 1 + 5$$

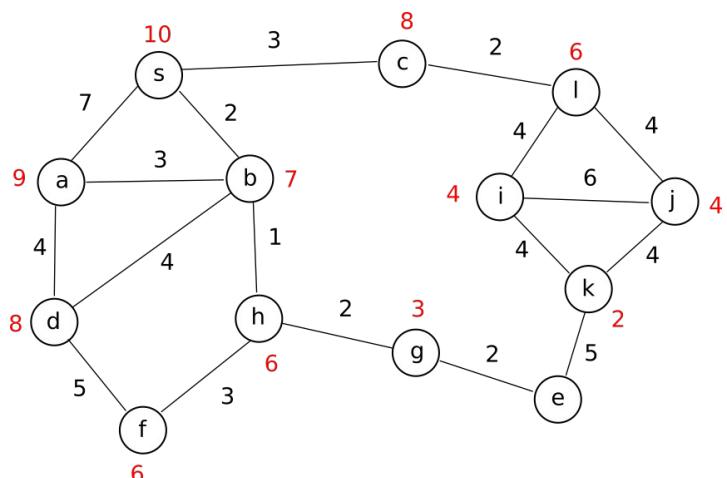
e portanto a e d saem da fila de prioridades antes. Isso ocorre pois no A* eles são considerados mais importantes. O mesmo ocorre nos demais níveis

$$\lambda(e) = \lambda(f) = \lambda(g) = \lambda(h) = \lambda(i) = \lambda(j) = 2$$

$$\gamma(e) = \sqrt{18} \quad \gamma(j) = \sqrt{10} \quad \gamma(h) = \sqrt{8}$$

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 . O 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}	$\lambda(a) = \min\{\infty, 7\} = 7$ $\lambda(b) = \min\{\infty, 2\} = 2$ $\lambda(c) = \min\{\infty, 3\} = 3$	$\pi(a) = s$ $\pi(b) = s$ $\pi(c) = s$
b	{a, d, h}	$\lambda(a) = \min\{7, 2+3\} = 5$ $\lambda(d) = \min\{\infty, 2+4\} = 6$ $\lambda(h) = \min\{\infty, 2+1\} = 3$	$\pi(a) = b$ $\pi(d) = b$ $\pi(h) = b$
h	{f, g}	$\lambda(f) = \min\{\infty, 3+3\} = 6$ $\lambda(g) = \min\{\infty, 3+2\} = 5$	$\pi(f) = h$ $\pi(g) = h$
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 $O(n^3)$. Em termos de custo computacional é equivalente a executar N vezes o algoritmo de Dijkstra.

"Solutions are not found by pointing fingers; they are reached by extending hands."
-- Aysha Taryam

Bibliografia

CORMEN, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 3^a edição, The MIT Press, 2009.

KLEINBERG, J.; TARDOS, E. Algorithmic Design, Addison Wesley, 2005.

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>

PUGH, W. Skip Lists: a probabilistic alternative to balanced trees, Communications of the ACM, vol. 33, n. 6, pp. 668-676, 1990.

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.

SZWARCFITER, J. L. Teoria Computacional de Grafos: Os algoritmos, Elsevier, 2018.

ZIVIANI N. Projeto de algoritmos: com implementações em Java e C++. 2. ed. São Paulo: Cengage Learning, 2011.

NICOLETTI, M. C.; HRUSCHKA, E. R. Fundamentos da Teoria dos Grafos para Computação, 2^a ed., Série Apontamentos, EdUFSCar, 2009.

CLARK, J., DEREK, A. H. A First Look at Graph Theory, World Scientific, 1998.

Geeks for Geeks – A computer science portal for geeks. <https://www.geeksforgeeks.org/>

Sobre o autor

Alexandre L. M. Levada é bacharel em Ciências da Computação pela Universidade Estadual Paulista “Júlio de Mesquita Filho” (UNESP), mestre em Ciências da Computação pela Universidade Federal de São Carlos (UFSCar) e doutor em Física Computacional pela Universidade de São Paulo (USP). Atualmente é professor adjunto no Departamento de Computação da Universidade Federal de São Carlos e seus interesses em pesquisa são reconhecimento de padrões, processamento de sinais e imagens e matemática aplicada.