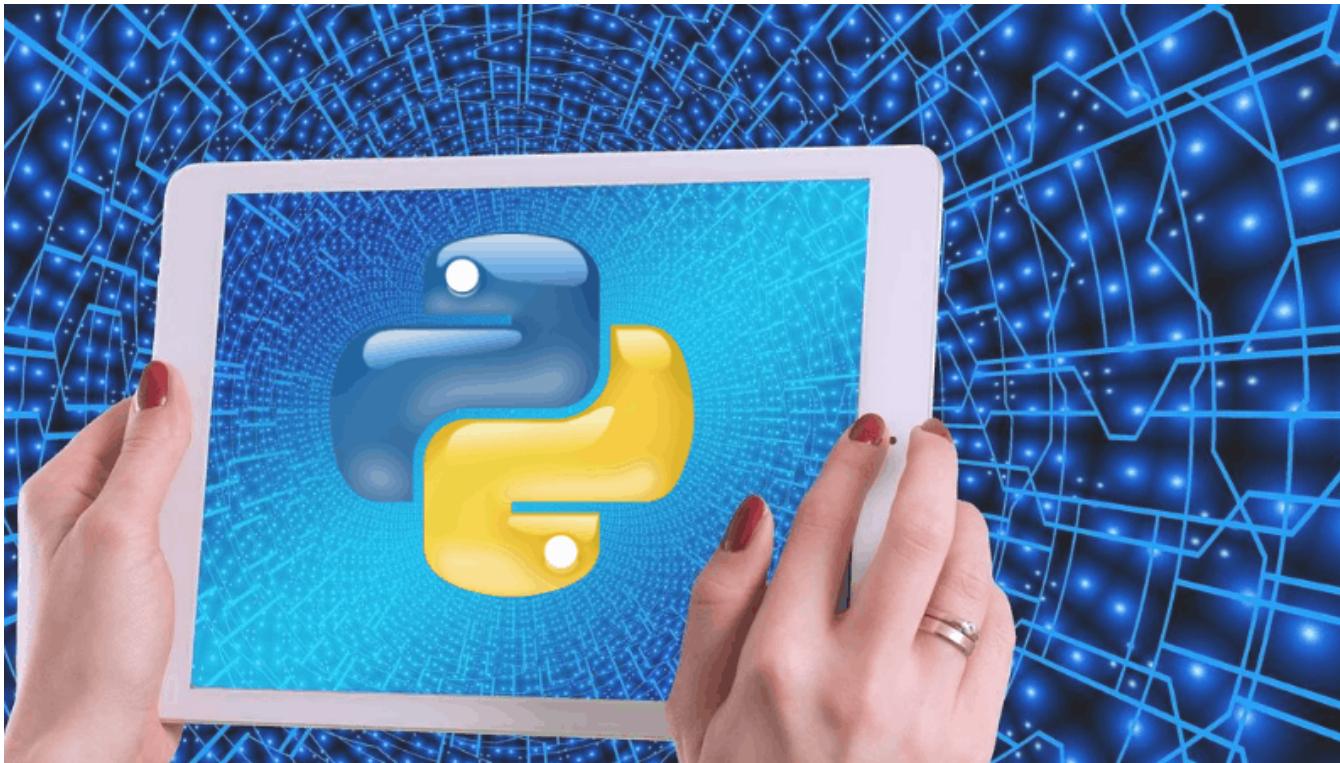


# ESTRUTURA DE DADOS COM PYTHON



# CONTEÚDO

- Programação básica em Python
- Notação Big-O
- Vetores não ordenados e ordenados
- Pilhas, filas e deques
- Listas encadeadas
- Recursão

# CONTEÚDO

- Algoritmos de ordenação
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Shell sort
  - Merge sort
  - Quick sort
- Árvores
- Grafos
  - Busca em largura e profundidade
  - Busca gulosa e A\*
  - Algoritmo de Dijkstra

## MAIS DETALHES

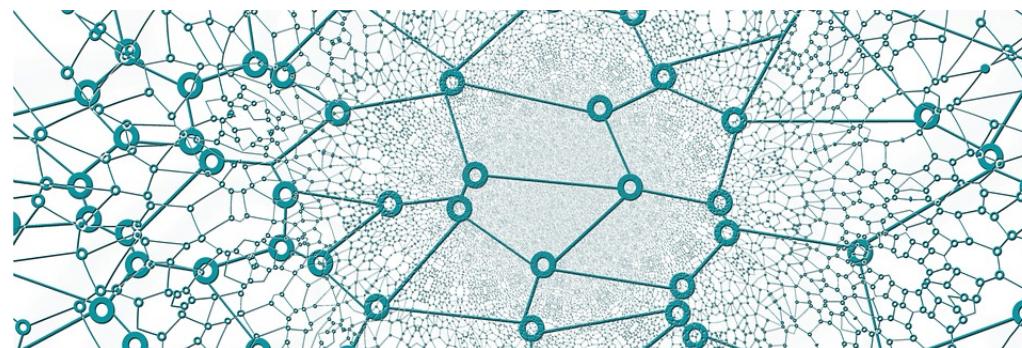
- 100% da codificação feita passo a passo
- Debug passo a passo do código fonte
- Questionários teóricos
- Exercícios práticos com soluções
- Não usaremos bibliotecas prontas!
- Disciplina de Estrutura de Dados
- Objetivo: entender a teoria e implementar/testar as estruturas de dados
  
- Pré-requisito: lógica de programação

# ESTRUTURA DE DADOS

- Disciplina que estuda as técnicas computacionais para a organização e manipulação eficiente de quaisquer quantidades de informações

# ESTRUTURA DE DADOS – PROBLEMA 1

- Temos uma rede de dispositivos eletrônicos. Tais dispositivos estão interligados por canais de comunicação. Cada canal tem um valor associado  $C$  (número real no intervalo  $0 \leq C \leq 1$ ) que representa sua confiabilidade. Interpreta-se como confiabilidade a probabilidade de que o canal não venha a falhar. Forneça um algoritmo para encontrar o caminho mais confiável entre dois dispositivos dados



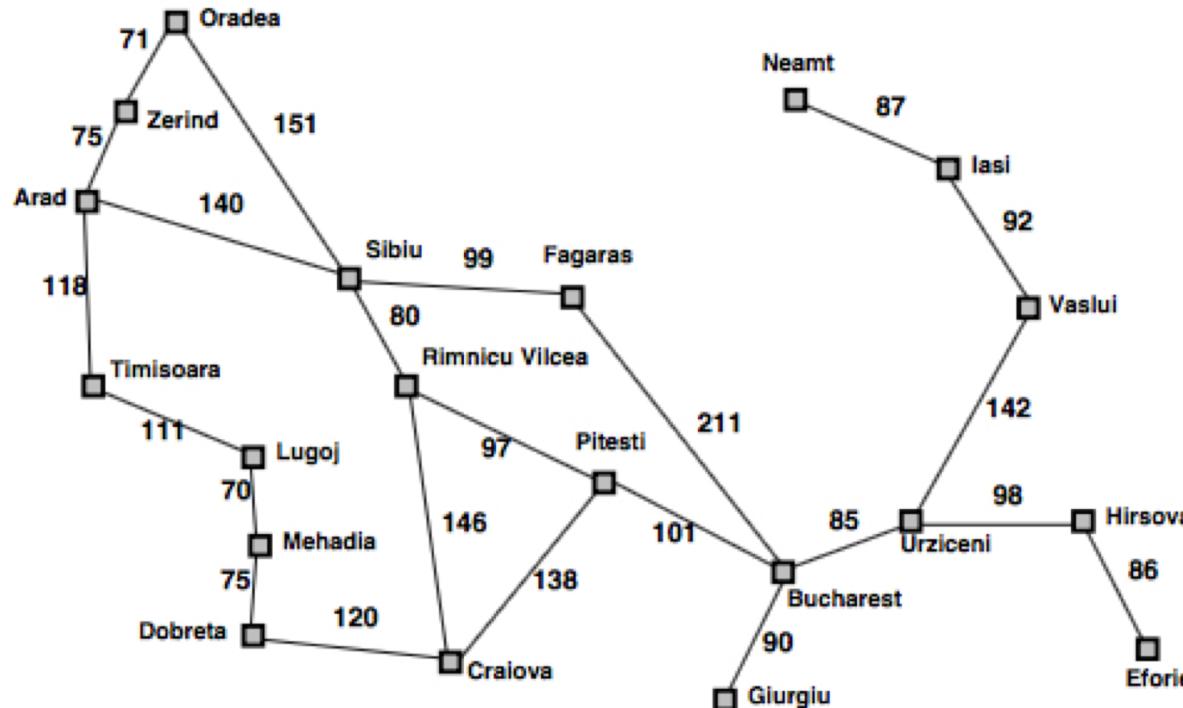
## ESTRUTURA DE DADOS – PROBLEMA 2

- Considere uma rede de ferrovias conectando duas cidades através de um número de cidades intermediarias, onde cada ferrovia tem uma capacidade de transporte específica. Como encontrar o fluxo máximo entre as duas cidades?



# ESTRUTURA DE DADOS – PROBLEMA 3

- Ir de Arad até Bucharest



# ESTRUTURA DE DADOS

- Como representar estes problemas em computadores?
  - Como construir os algoritmos necessários?
  - Que estrutura de dados utilizar?
  - De que forma os dados estarão organizados?
  - Que operações podem ser realizadas nestes dados?
- 
- Entender sobre estrutura de dados é essencial para garantir que os algoritmos sejam eficientes

# INTRODUÇÃO AO PYTHON

- Criada em 1991 por Guido van Rossum
- Simples e de fácil aprendizado
- Portátil
- Desenvolvimento web
- Inteligência Artificial
- Big Data
- Ciência de Dados
- Computação Gráfica
- Popularidade
- FAQ: <https://wiki.python.org.br/PerguntasFrequentes/SobrePython>



Fonte: [https://en.wikipedia.org/wiki/Python\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))

# EXPRESSÕES REGULARES

- Busca sofisticada por padrões
- Procurar todas as palavras que começam com letra maiúscula, ou que começam com letra maiúscula e terminam com letra minúscula
- Que começam com ‘a’ e terminam com ‘e’
- Procurar e-mails, número de telefone, CEPs, etc
- Padrões de busca são praticamente ilimitados

# EXPRESSÕES REGULARES

“O mundo está mudando. Tecnologias utilizadas na atualidade estão tornando as tarefas cada vez mais mecanizadas e práticas, onde a mente humana é deixada livre para exercer o papel criativo”

Extrair todas as palavras que começam com vogais e terminam com consoantes

Extrair todas as palavras que possuem de 5 a 9 letras

Extrair todos os trechos separados por ponto (‘.’)

Extrair todas as palavras que começam com ‘c’, possuem ‘r’, ‘e’ ou ‘u’ no meio e terminam com ‘a’, ‘l’ ou ‘x’

# EXPRESSÕES REGULARES

- Encontrar as posições de padrões dentro de uma string, se estes estiverem presentes (método **search**)
- Encontrar se o começo de uma string é igual a um determinado padrão (método **match**)
- Encontrar todas as substrings em uma string que correspondam a um padrão (método **findall**)

# EXPRESSÕES REGULARES - METACARACTERES

. – Qualquer caractere (exceto linha nova)

\w – Qualquer caractere alfanumérico

\W – qualquer caractere não-alfanumérico

\d – Qualquer caractere que seja um dígito (0-9)

\D - Qualquer caractere não dígito

\s – Espaço em branco

^ - começa com

\\$ - termina com

“\” – usado antes de metacaracteres para especificar seu significado literal

# EXPRESSÕES REGULARES - QUANTIFICADORES

Permitem determinar como e quantas vezes os metacaracteres aparecem

[ ] – opcional entre os que estão dentro dos colchetes

( ) – captura grupos de caracteres

\* - de zero a mais vezes

? – zero ou uma vez

+ - uma ou mais vezes

{m,n} – de m a n vezes

| - ou

# EXPRESSÕES REGULARES - EXEMPLO

- gabrielcosta@hotmail.com
- felipearruda@gmail.com
- joaosilva@yahoo.com.br
- Expressão regular que detecta e-mails



\w+@\w+\.\w+

# ORIENTAÇÃO A OBJETOS – CLASSE E OBJETO

- Agrupamento de objetos similares que apresentam os mesmos atributos e métodos
- Molde de bonecos de gesso
  - Define o formato e tamanho
  - O molde é sempre o mesmo, porém, os objetos gerados podem ter características variadas, respeitando a estrutura básica do molde



Cor = rosa



Cor = laranja



Cor = azul

# ORIENTAÇÃO A OBJETOS – CLASSE E OBJETO



# REPRESENTAÇÃO DE OBJETOS

Características

1. Cor
2. Tamanho
3. Nº de patas
4. Raça



Ações (métodos)

1. Latir
2. Correr
3. Morder
4. Brincar

# REPRESENTAÇÃO DE OBJETOS

Características

1. Cor
2. Marca
3. Tamanho
4. Data de fabricação



Ações

1. Somar
2. Subtrair
3. Dividir
4. Multiplicar

# OBJETOS

**Características**



Atributos

- Strings
- Listas
- Tuplas

**Ações**



Métodos

- Funções

Mundo	Programação
Molde de boneco de gesso	Classe
Bonecos de gesso	Objetos

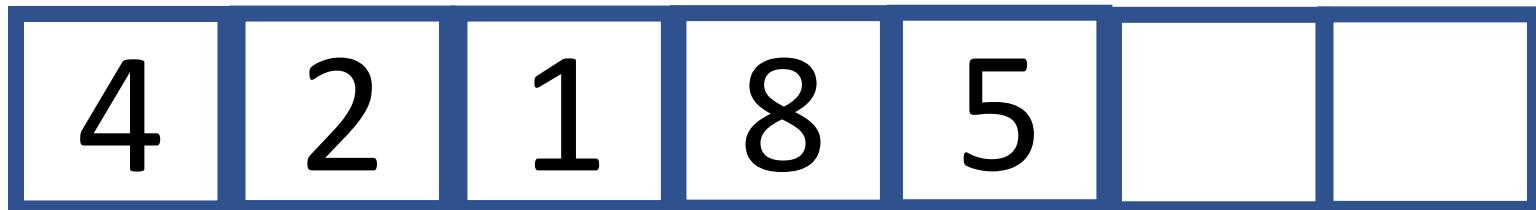
# VETORES NÃO ORDENADOS

- Controlar quais jogadores estão presentes no campo de treino (estrutura de dados)
- Várias ações poderiam ser executadas
  - Inserir um jogador na estrutura de dados quando ele chegar ao campo
  - Verificar se um determinado jogador está presente, pesquisando o número do jogador na estrutura
  - Remover um jogador da estrutura de dados quando ele for para casa



# VETORES NÃO ORDENADOS – INSERÇÃO

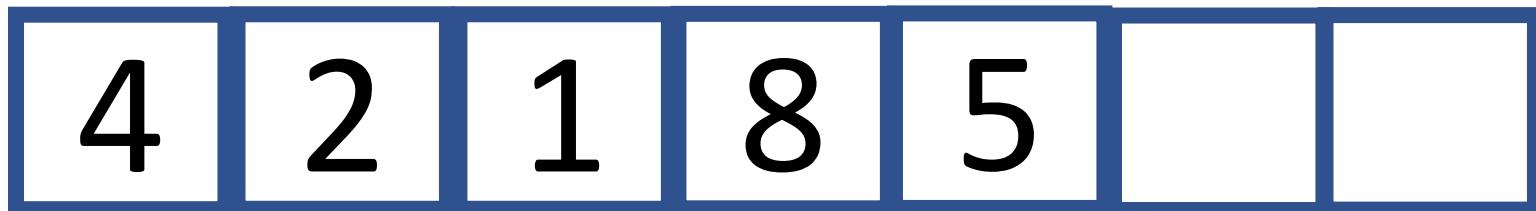
- Inserção de um novo jogador dentro do vetor



- Único passo (inserido na primeira célula vaga do vetor)
- O algoritmo já conhece essa localização porque ele já sabe quantos itens já estão no vetor
- O novo item é simplesmente inserido no próximo espaço disponível
- Big-O constante – O(1)

# VETORES NÃO ORDENADOS – PESQUISA LINEAR

- Percorrer cada posição do vetor
- Melhor caso: 4
- Pior caso: 5 ou número que não existe
- Em média, metade dos itens devem ser examinados ( $N/2$ )
- Big-O linear –  $O(n)$



## VETORES NÃO ORDENADOS – EXCLUSÃO

4	2	1	8	5		
4	2		8	5		
4	2	8		5		
4	2	8	5			

# VETORES NÃO ORDENADOS – EXCLUSÃO

- Pesquisar uma média de  $N/2$  elementos (pesquisa linear)
  - Pior caso: N
- Mover os elementos restantes ( $N/2$  passos)
  - Pior caso: N
- Big-O –  $O(2n) = O(n)$

# VETORES NÃO ORDENADOS – DUPLICATAS

- Deve-se decidir se itens com chaves duplicadas serão permitidos
- Exemplo de um arquivo de funcionários
  - Se a chave for o número de registro
  - Se a chave for o sobrenome
- Pesquisa: mesmo se encontrar o valor, o algoritmo terá que continuar procurando até a última célula ( $N$  passos)
- Inserção: verificar cada item antes de fazer uma inserção ( $N$  passos)
- Exclusão do primeiro item:  $N/2$  comparações e  $N/2$  movimentos
- Exclusão de mais itens: verificar  $N$  células e mais de  $N/2$  células



# VETORES ORDENADOS

- Os dados estão organizados na ordem ascendente de valores-chave, ou seja, com o menor valor no índice 0 e cada célula mantendo um valor maior que a célula abaixo
- Vantagem: agiliza os tempos de pesquisa



# VETORES ORDENADOS – INSERÇÃO

3	1	2	4	5			
1	2	4		5			
1	2		4	5			
1	2	3	4	5			

# VETORES ORDENADOS – INSERÇÃO

- Pesquisar uma média de  $N/2$  elementos (pesquisa linear)
  - Pior caso: N
- Mover os elementos restantes ( $N/2$  passos)
  - Pior caso: N
- Big-O –  $O(2n) = O(n)$

# VETORES ORDENADOS – PESQUISA LINEAR

- A pesquisa termina quando o primeiro item maior que o valor de pesquisa é atingido
- Como o vetor está ordenado, o algoritmo sabe que não há necessidade de procurar mais
- Pior caso: se o elemento não estiver no vetor ou na última posição
- Big-O – O(n)
- Visualização on-line
  - <https://www.cs.usfca.edu/~galles/visualization/Search.html>

## VETORES ORDENADOS – EXCLUSÃO

1	2	4	5	8		
1	2		5	8		
1	2	5		8		
1	2	5	8			

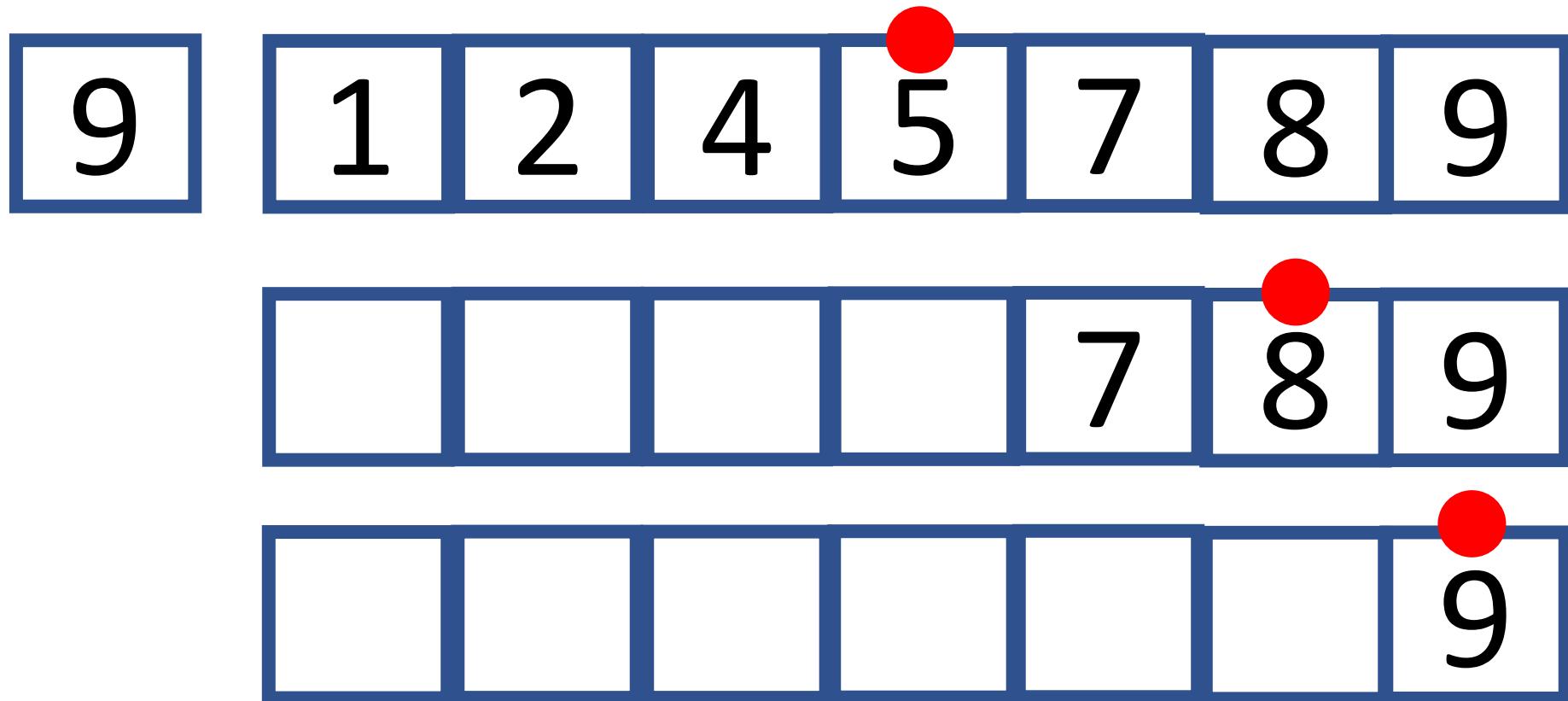
# VETORES ORDENADOS – EXCLUSÃO

- O algoritmo pode terminar na metade do caminho se não encontrar o item
- Pesquisar uma média de  $N/2$  elementos (pesquisa linear)
  - Pior caso: N
- Mover os elementos restantes ( $N/2$  passos)
  - Pior caso: N
- Big-O –  $O(2n) = O(n)$

# VETORES ORDENADOS – PESQUISA BINÁRIA

- Números de 1 até 100
- Pesquisar/adivinhar o número 47
- 1 até 100 / 2 = 50
  - 50 é o número pesquisado? Não
  - 47 é menor ou maior do que 50? Menor
- 1 até 49 / 2 = 25
  - 25 é o número pesquisado? Não
  - 47 é menor ou maior do que 25? Maior
- 26 até 49 / 2 = 38
  - 38 é o número pesquisado? Não
  - 47 é menor ou maior do que 38? Maior
- 39 até 49 / 2 = 44
  - 44 é o número pesquisado? Não
  - 47 é menor ou maior do que 44? Maior
- 45 até 49 / 2 = 47
  - 47 é o número pesquisado? Sim

# VETORES ORDENADOS – PESQUISA BINÁRIA



# VETORES ORDENADOS – PESQUISA BINÁRIA x PESQUISA LINEAR

Faixa	Comparações Binária	Comparações Linear (N/2)
10	4	5
100	7	50
1.000	10	500
10.000	14	5.000
100.000	17	50.000
1.000.000	20	500.000
10.000.000	24	5.000.000
100.000.000	27	50.000.000
1.000.000.000	30	500.000.000

# VETORES ORDENADOS – DISCUSSÕES

- A principal vantagem é que os tempos de pesquisa são muito mais rápidos do que em um vetor não ordenado
- A inserção leva mais tempo, pois os itens de dados devem ser movidos para criar espaço
- As remoções são lentas tanto nos vetores ordenados quanto nos não ordenados, pois os itens devem ser movidos para preencher o “buraco”
- Vetores ordenados são úteis quando pesquisas são frequentes, mas inserções e remoções não são
- Qual o tipo de vetor mais adequado para
  - Um cadastro de funcionários?
  - Um cadastro de produtos de uma loja de varejo?

# ORDENAÇÃO

- Após uma base de dados estar construída, pode ser necessário ordená-la, como:
  - Nomes em ordem alfabética
  - Alunos por nota
  - Clientes por CEP
  - Vendas por preço
  - Cidades em ordem crescente de população
- Ordenar dados pode ser um passo preliminar para pesquisá-los (pesquisa binária X pesquisa linear)

# BUBBLE SORT – BOLHA

- Notavelmente lento e é o mais simples dos algoritmos de ordenação
- Funcionamento
  - Comparação de dois números
  - Se o da esquerda for maior, os elementos devem ser trocados
  - Desloca-se uma posição à direita
- À medida que o algoritmo avança, os itens maiores “surgem como uma bolha” na extremidade superior do vetor
- Visualização on-line: <https://visualgo.net/en/sorting>

## BUBBLE SORT – BOLHA

- O algoritmo com 10 elementos faz 9 comparações na primeira passagem, 8 na segunda, 7 na terceira, etc ( $n - 1, n - 2, n - 3$ ). Para 10 itens:
  - $9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45$
- Big-O:  $O(n^2)$
- O algoritmo faz cerca de  $N^2/2$  comparações
- Há menos trocas do que há comparações, pois dois elementos serão trocados somente se precisarem
- Se os dados forem aleatórios, uma troca será necessária mais ou menos  $N^2/4$  (no pior caso, com os dados em modo inverso, uma troca será necessária a cada comparação)

# SELECTION SORT

- Melhora a ordenação pelo método da bolha reduzindo o número de trocas necessárias de  $N^2$  para N
- O número de comparações permanece  $N^2/2$
- Funcionamento
  - Percorrer todos os elementos e selecionar o menor
  - O menor elemento é trocado com o elemento da extremidade esquerda do vetor (posições iniciais)
  - Os elementos ordenados acumulam-se na esquerda
- Visualização on-line: <https://visualgo.net/en/sorting>

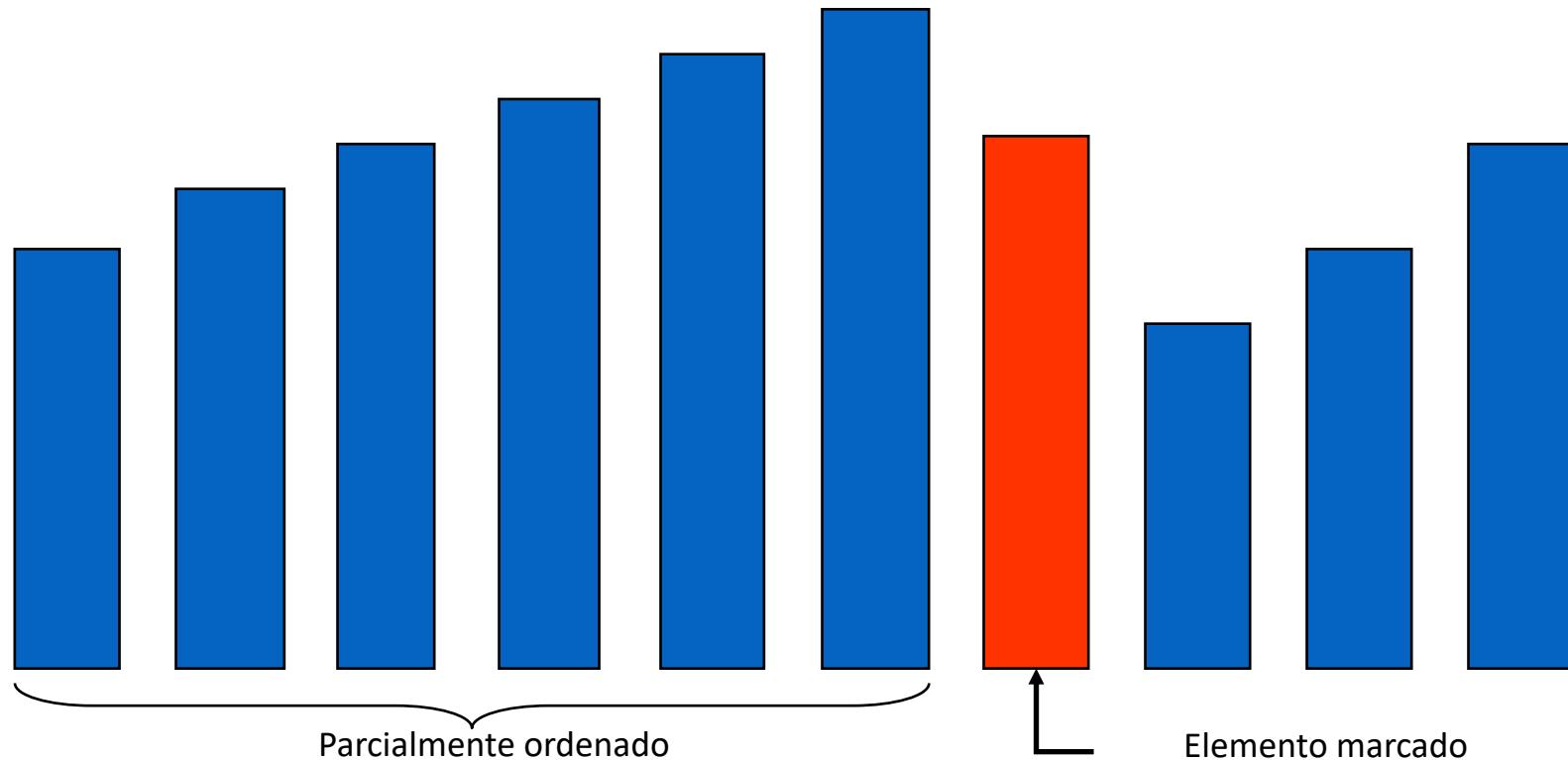
# SELECTION SORT

- O algoritmo com 10 elementos faz 9 comparações na primeira passagem, 8 na segunda, 7 na terceira, etc ( $n - 1, n - 2, n - 3$ ). Para 10 itens:
  - $9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45$
- Big-O:  $O(n^2)$
- O algoritmo faz cerca de  $N^2/2$  comparações (mesmo número de comparações que o método da bolha)
- Com 10 elementos, são requeridas menos de 10 trocas (geralmente é feita uma troca a cada passagem)
- Com 100 itens, são requeridas 4.950 comparações, mas menos de 100 trocas

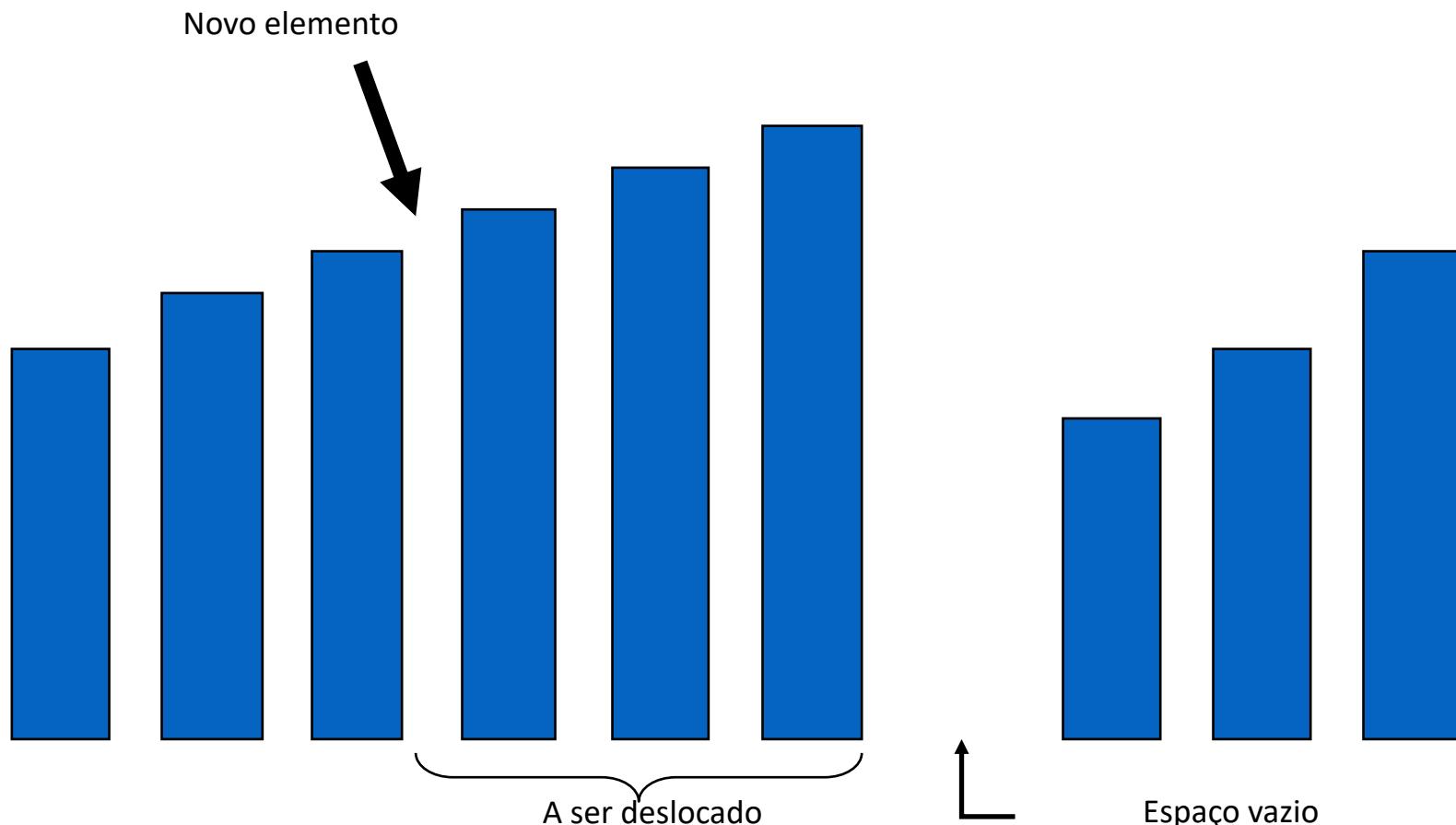
# INSERTION SORT

- É cerca de duas vezes mais rápido que a ordenação pelo método da bolha e um pouco mais rápido que a ordenação por seleção em situações normais
- Funcionamento
  - Há um marcador em algum lugar no meio do vetor
  - Os elementos à esquerda do marcador estão *parcialmente ordenados* (estão ordenados entre eles, porém não estão em suas posições finais)
  - Os elementos à direita do marcador não estão ordenados
- Visualização on-line: <https://visualgo.net/en/sorting>

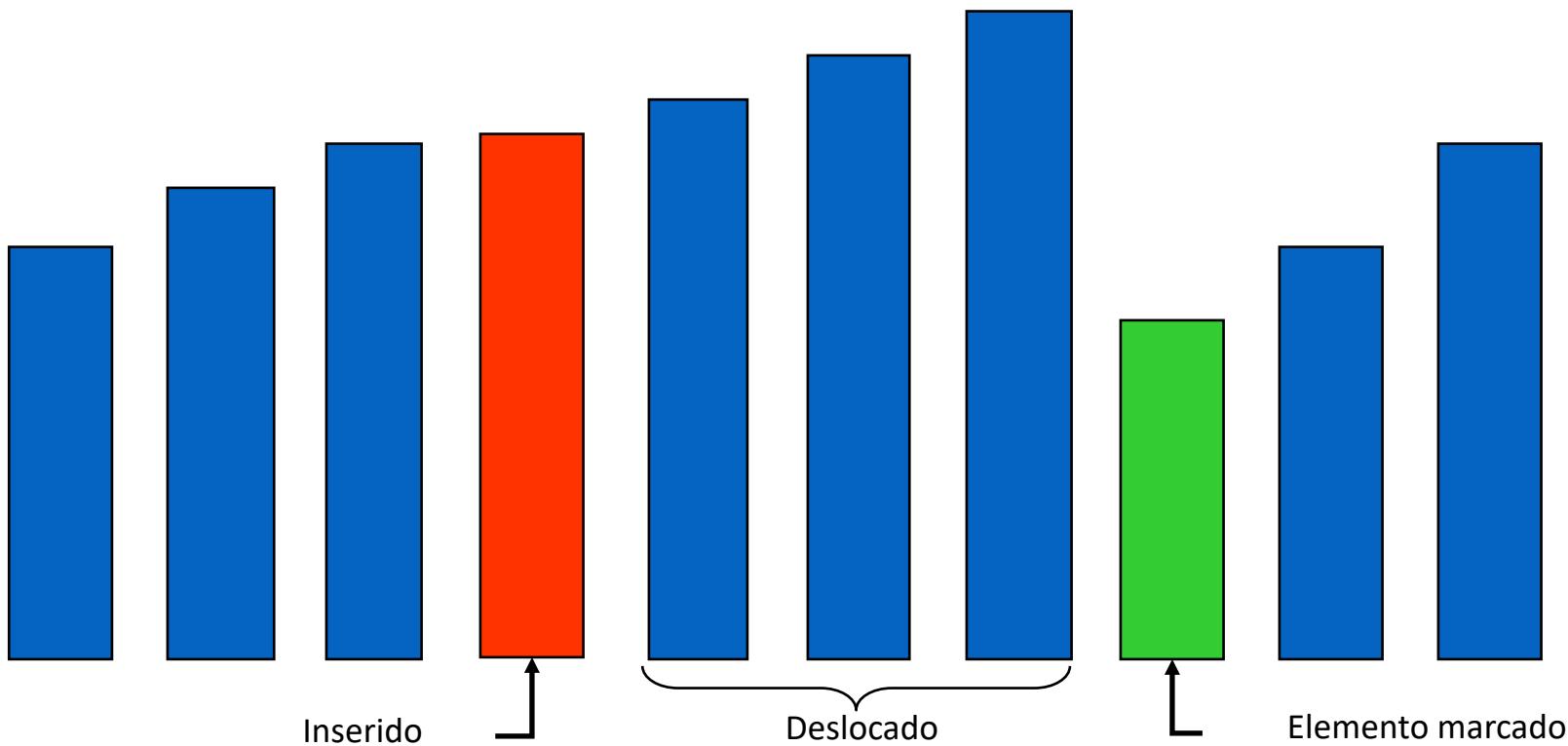
# INSERTION SORT



# INSERTION SORT



# INSERTION SORT



# INSERTION SORT

- Na primeira passagem, é comparado no máximo um item. Na segunda passagem, máximo de dois itens, etc
  - $1 + 2 + 3 + \dots + N - 1 = N*(N-1)/2$
- Como em cada passagem uma média de apenas metade do número máximo de itens é de fato comparada antes do ponto de inserção ser encontrado, então:
  - $N*(N-1)/4$
- O número de cópias é aproximadamente o mesmo que o número de comparações
- Para dados aleatórios esse algoritmo executa duas vezes mais rápido que o método da bolha e mais rápido que a ordenação por seleção
- Para dados que já estejam quase ordenados esse algoritmo é ainda mais eficiente
- Para dados em ordem inversa, todas as comparações e deslocamentos são executados, sendo mais lento que o método bolha

# SHELL SORT

- Melhora a ordenação por inserção, podendo ser considerado uma “versão” do insertion sort
- Funcionamento
  - O vetor original é quebrado em subvetores
  - Cada subvetor é ordenado comparando e trocando os valores
  - Ao final de uma rodada, o subvetor é quebrado em mais um subvetor
  - Vetor com 20 elementos
    - Primeira rodada: 10 elementos
    - Segunda rodada: 5 elementos
    - Terceira rodada: 2/3 elementos
- Visualizações
  - <https://www.programiz.com/dsa/shell-sort>
  - <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>
  - <https://www.w3resource.com/ODSA/AV/Sorting/shellsortAV.html>

# SHELL SORT

$N / 2$

8	5	1	4	2	3
8	5	1	4	2	3
4	5	1	8	2	3
4	5	1	8	2	3
4	2	1	8	5	3
4	2	1	8	5	3

# SHELL SORT

- A complexidade do algoritmo depende dos intervalos escolhidos
- Pior caso:  $O(n^2)$
- Melhor caso:  $O(n * \log n)$
- Melhor do que o selection sort  $O(n^2)$  e que o selection sort  $O(n^2)$
- Considerado um algoritmo “instável” porque não examina os elementos dentro de um intervalo

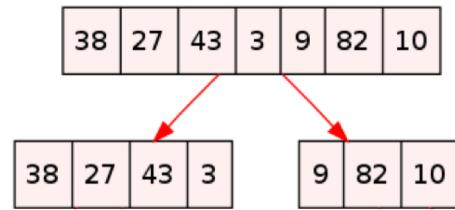
# MERGE SORT

- Um dos algoritmos de ordenação mais populares
- Funcionamento
  - Divisão do problema em subproblemas (dividir e conquistar)
  - Divide o vetor continuamente pela metade, ordena e combina (merge)
- Visualização on-line: <https://visualgo.net/en/sorting>

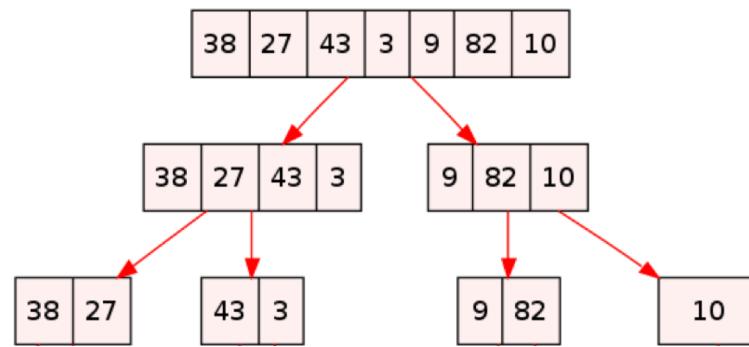
# MERGE SORT

38	27	43	3	9	82	10
----	----	----	---	---	----	----

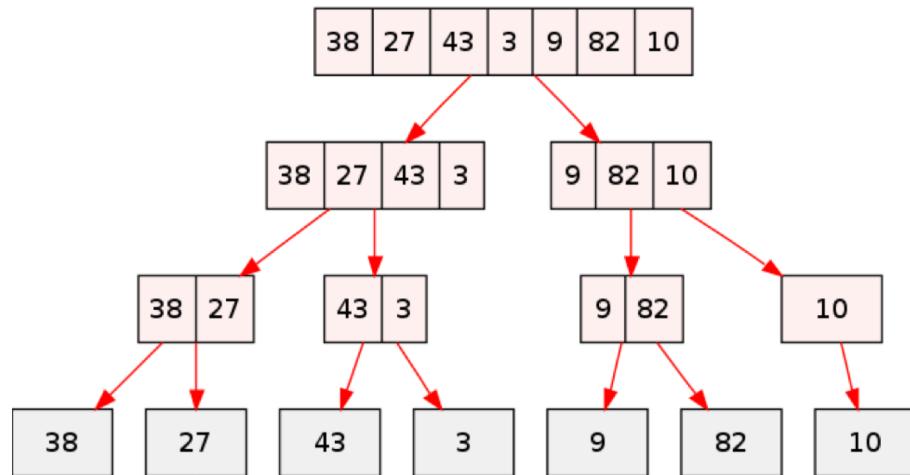
# MERGE SORT



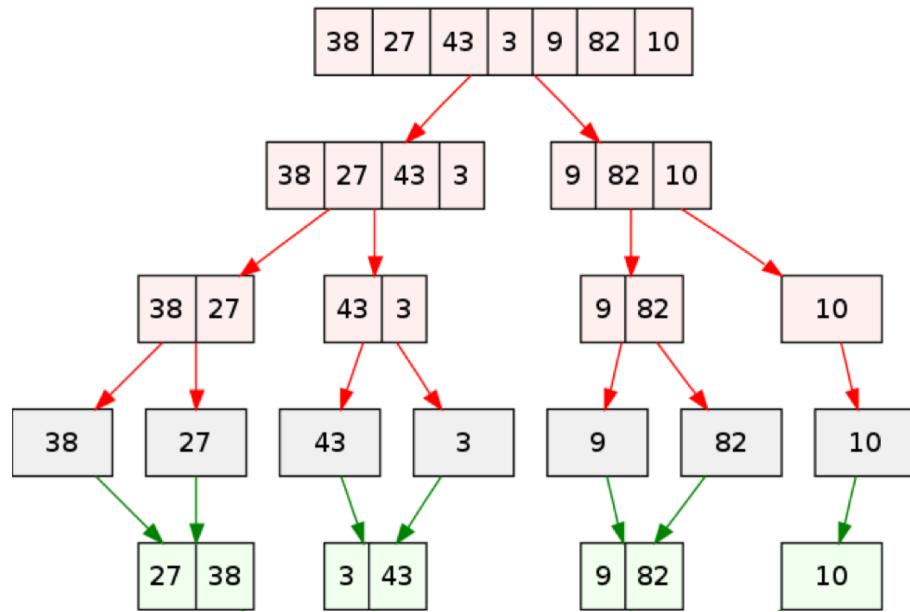
# MERGE SORT



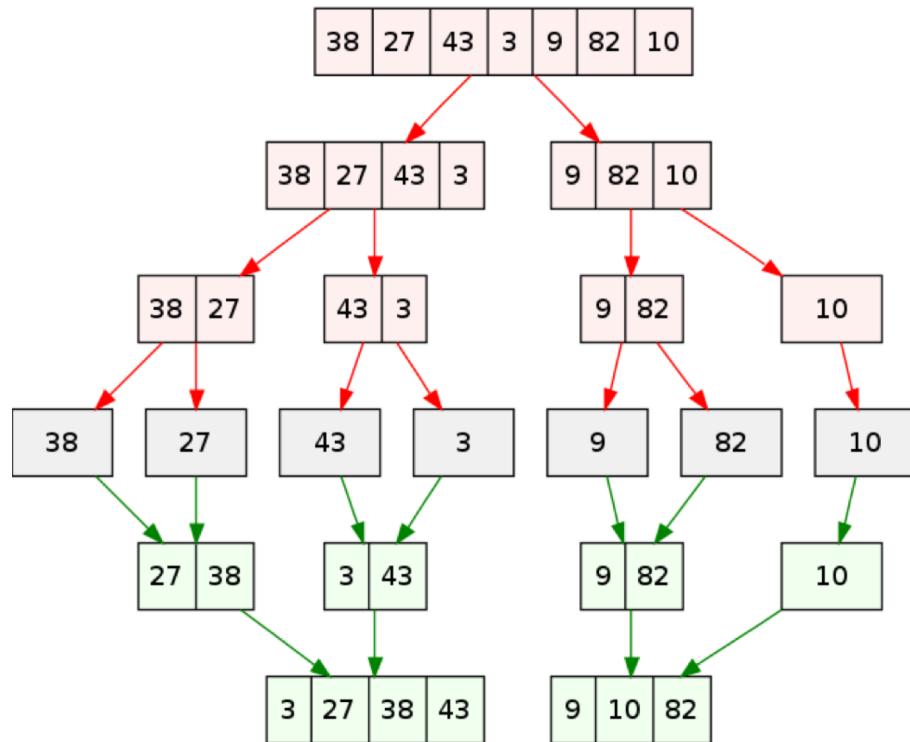
# MERGE SORT



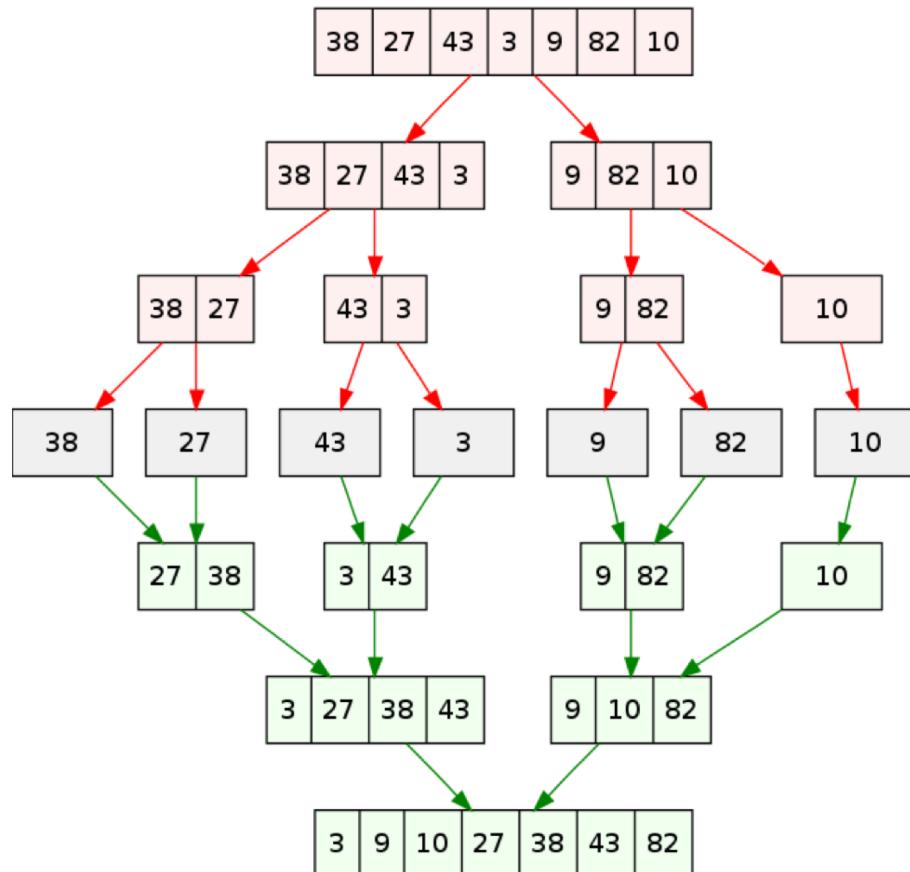
# MERGE SORT



# MERGE SORT



# MERGE SORT



# MERGE SORT

- Pior caso:  $O(n * \log n)$
- Melhor caso:  $O(n * \log n)$
- Melhor do que o bubble sort  $O(n^2)$ , que o selection sort  $O(n^2)$  e que o shell sort -  $O(n^2)$  no pior caso e  $O(n * \log n)$  em média

# QUICK SORT

- Algoritmo rápido e eficiente criado em 1960 para traduzir um dicionário de inglês para russo
- Funcionamento
  - O vetor é dividido em subvetores que são chamados recursivamente para ordenar os elementos
  - Estratégia da divisão e conquista
- Visualização on-line: <https://visualgo.net/en/sorting>

# QUICK SORT

- Pior caso:  $O(n^2)$  – quando o elemento pivot é o maior ou menor elemento
- Melhor caso:  $O(n * \log n)$

# PILHAS



# PILHAS

- A** – você está ocupado com um projeto de longo prazo
- B** – é interrompido por um colega solicitando ajuda em um outro projeto
- C** – enquanto estiver trabalhando em B, alguém da contabilidade aparece para uma reunião sobre despesas de viagem
- D** – durante a reunião, recebe um telefonema de emergência de alguém de vendas e passa alguns minutos resolvendo um problema relacionado a um novo produto
- Quando tiver terminado o telefonema **D**, voltará para a reunião **C**; quando tiver acabado com **C**, voltará para o projeto **B** e quando tiver terminado com **B**, poderá finalmente voltar para o projeto **A**

D  
C  
B  
A

# PILHAS

- Permite acesso a um item de dados: o último item inserido
- Se o último item for removido, o item anterior ao último inserido poderá ser acessado
- Aplicações
  - Correção de expressões aritméticas, tais como  $3 * (4 + 5)$
  - Percorrimento de uma árvore binária
  - Pesquisa do vértice de um grafo
  - Microprocessadores com arquitetura baseada em pilhas. Quando um método é chamado, seu endereço de retorno e seus parâmetros são empilhados em uma pilha e quando ele retorna, são desempilhados



# PILHAS – OPERAÇÕES

- Empilhar
  - Colocar um item de dados no topo da pilha
- Desempilhar
  - Remover um item do topo da pilha
- Ver o topo
  - Mostra o elemento que está no topo da pilha
- Último-A-Entrar-Primeiro-A-Sair (LIFO – Last-In-First-Out)

# PILHAS – VALIDADOR DE EXPRESSÕES

- Os delimitadores são as chaves { e }, os colchetes [ e ] e os parênteses ( e )
- Cada delimitador de abertura ou à esquerda deve ser casado com um delimitador de fechamento ou à direita
- Toda { deve ser seguida por uma } que case com ela
- Exemplos
  - c[d] (correto)
  - a{b[c]d}e (correto)
  - a{b(c)d}e (incorreto - ] não casa com (
  - a[b{c}d]e} (incorreto – nada casa com } no final
  - a{b(c) (incorreto – nada casa com { de abertura

# PILHAS – VALIDADOR DE EXPRESSÕES

Caractere lido	Conteúdo da pilha
a	
{	{
b	{
(	{(
c	{(
[	{([
d	{([
]	{(
e	{(
)	{
f	{
}	

# FILAS



# FILAS

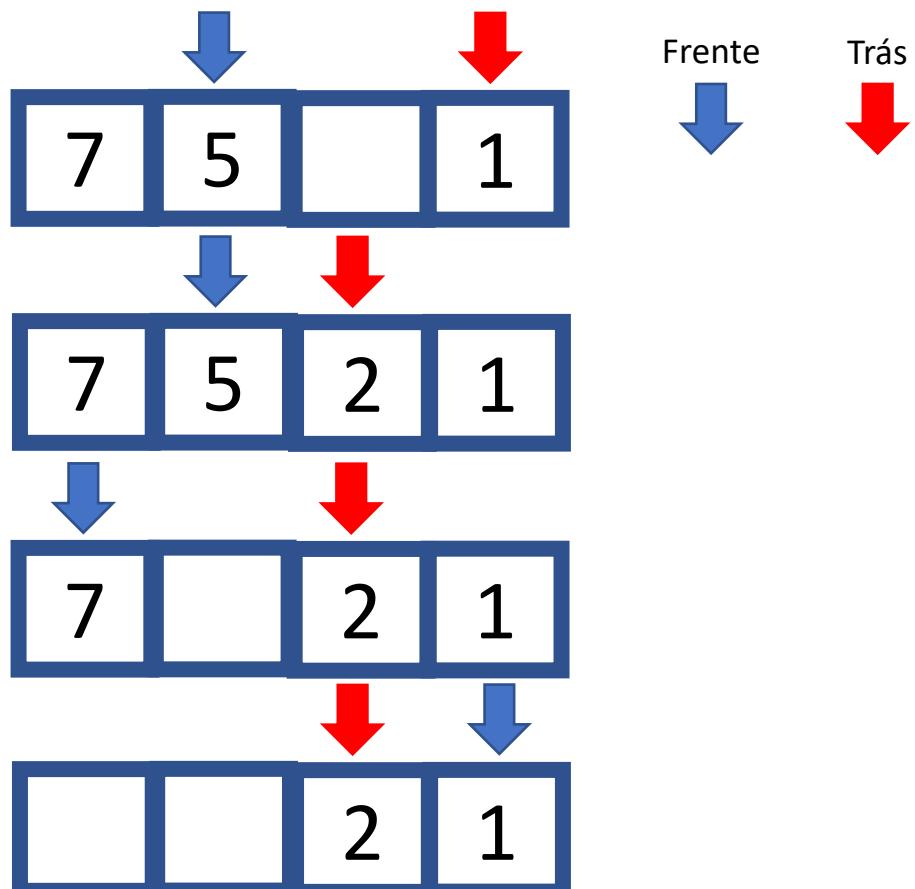
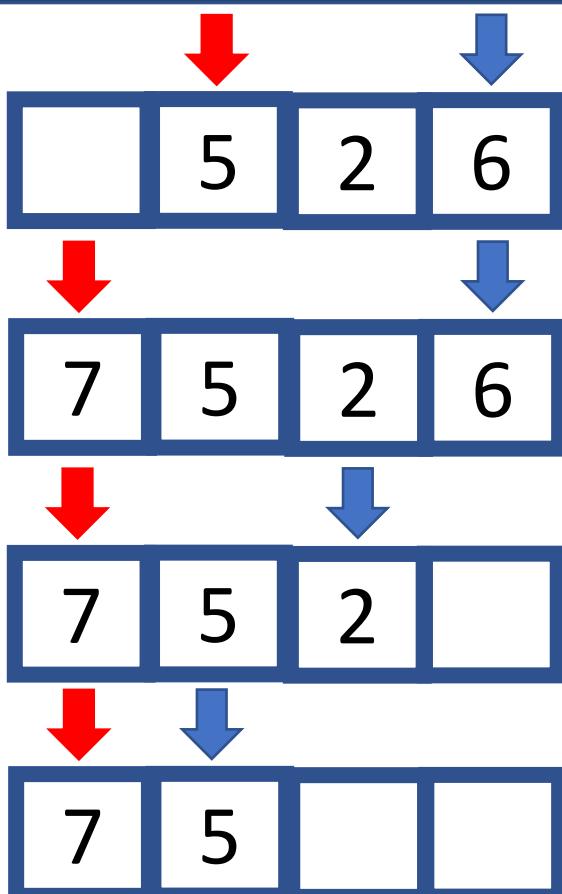
- A primeira pessoa a entrar no final da fila será a primeira pessoa a chegar na frente da fila
- É uma estrutura semelhante a uma pilha, exceto que em uma fila o primeiro elemento inserido é o primeiro a ser removido (First-In-First-Out, FIFO – Primeiro-A-Entrar-Primeiro-A-Sair)
- Aplicações
  - Modelar aviões aguardando para decolar
  - Pacotes de dados esperando para serem transmitidos pela rede
  - Fila da impressora, no qual serviços de impressão aguardam a impressora ficar disponível



# FILAS – OPERAÇÕES

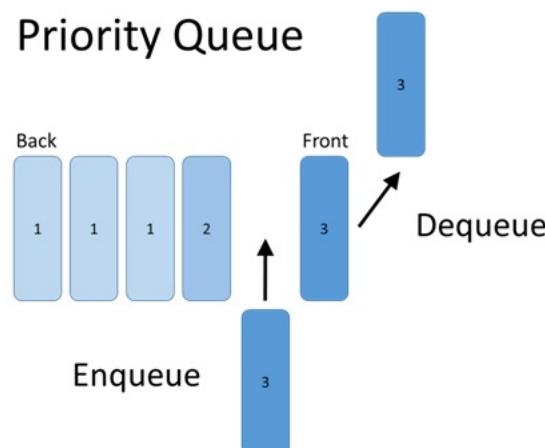
- Enfileirar
  - Colocar um item no final da fila
- Desenfileirar
  - Remover um item do início da fila
- Ver início da fila
  - Mostra o elemento que está no início da fila

# FILA CIRCULAR



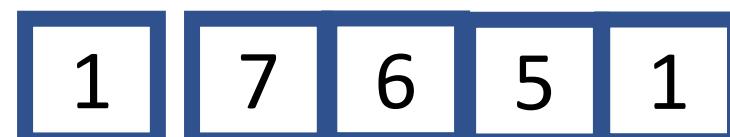
# FILA DE PRIORIDADE

- Os itens são ordenados por valor-chave, de modo que o item com a chave mais baixa/alta esteja sempre na frente
- Elementos de alta prioridade são colocados no início da fila, de média prioridade no meio da fila e elementos de baixa prioridade no final da fila

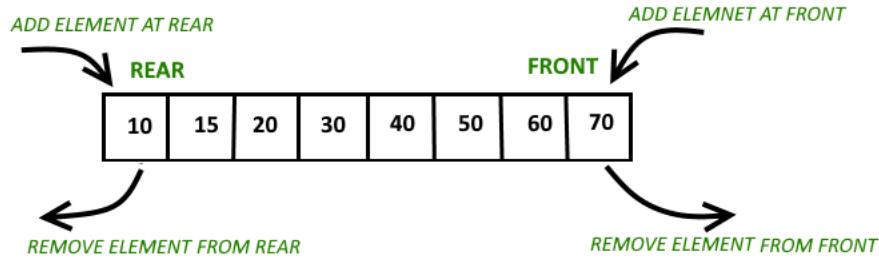


Fonte: <https://www.includehelp.com/ds/implementation-of-priority-queue-using-linked-list.aspx>

# FILA DE PRIORIDADE

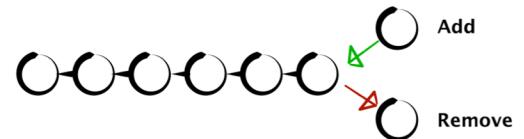


# DEQUES (DOUBLE ENDED QUEUE)

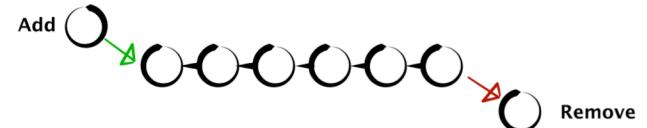


Fonte: <https://www.geeksforgeeks.org/implementation-deque-using-circular-array/>

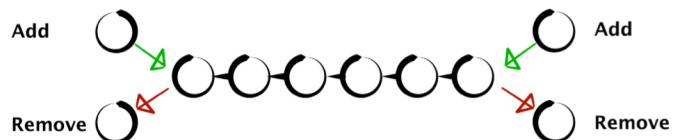
Stack



Queue



Deque



Fonte: <https://medium.com/@rasmussen.matias/fun-with-deques-in-python-31942bcb6321>

# DEQUES (DOUBLE ENDED QUEUE)

- Suporta operações de pilhas e filas
- Aplicações
  - Filas de prioridade
  - Agendamento de tarefas de vários processadores
  - O algoritmo de agendamento de trabalho furtivo é usado pela biblioteca Threading Building Blocks (TBB) da Intel para programação paralela (Fonte:  
[https://en.wikipedia.org/wiki/Double-ended\\_queue#Applications](https://en.wikipedia.org/wiki/Double-ended_queue#Applications))

# DEQUES – OPERAÇÕES

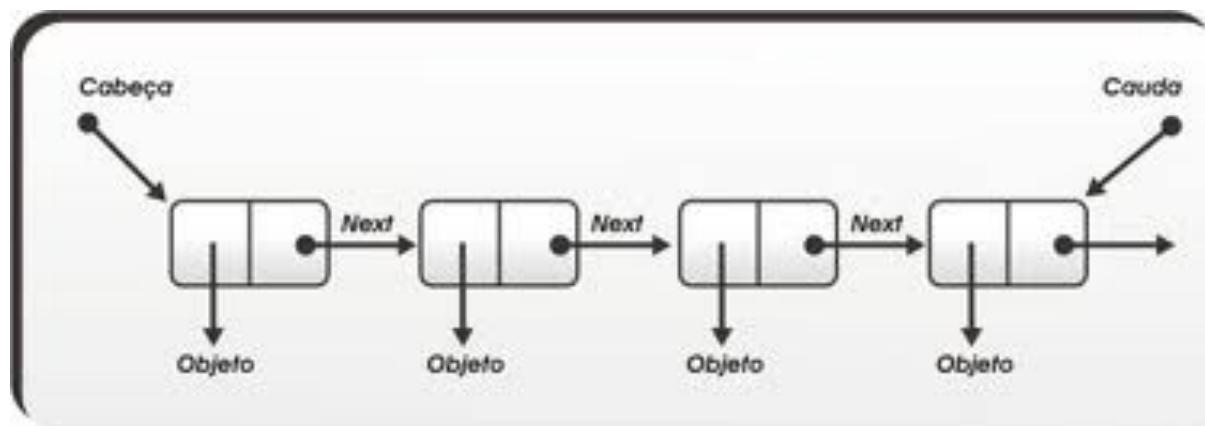
- Adicionar no início
  - Adicionar no final
  - Excluir do início
  - Excluir do final
- 
- Implementação estática X Circular

## DESVANTAGENS DE VETORES

- Em um vetor não ordenado, a busca é lenta
- Em um vetor ordenado, a inserção é lenta
- Em ambos, a remoção é lenta
- O tamanho do vetor “não pode” ser alterado depois de ter sido criado
- Mesmo vazios ocupam espaço na memória

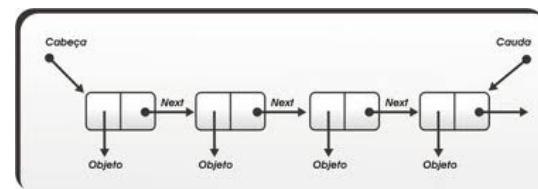
# LISTAS ENCADEADAS – NÓS

- Cada item de dados é incorporado em um nó
- Cada nó possui uma referência para o próximo nó da lista
- Um campo da própria lista contém uma referência para o primeiro nó



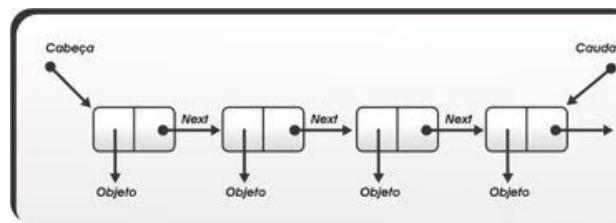
# LISTAS ENCADEADAS – POSIÇÃO X RELACIONAMENTO

- Vetor (posição)
  - Cada item ocupa uma certa posição
  - Cada posição pode ser acessada usando um número de índice
- Lista (relacionamento)
  - A única maneira de encontrar um elemento é seguir a sequência de elementos
  - Um item de dados não pode ser acessado diretamente, ou seja, o relacionamento entre eles deve ser utilizado
  - Inicia com o primeiro item, vai para o segundo, então o terceiro, até encontrar o item pesquisado

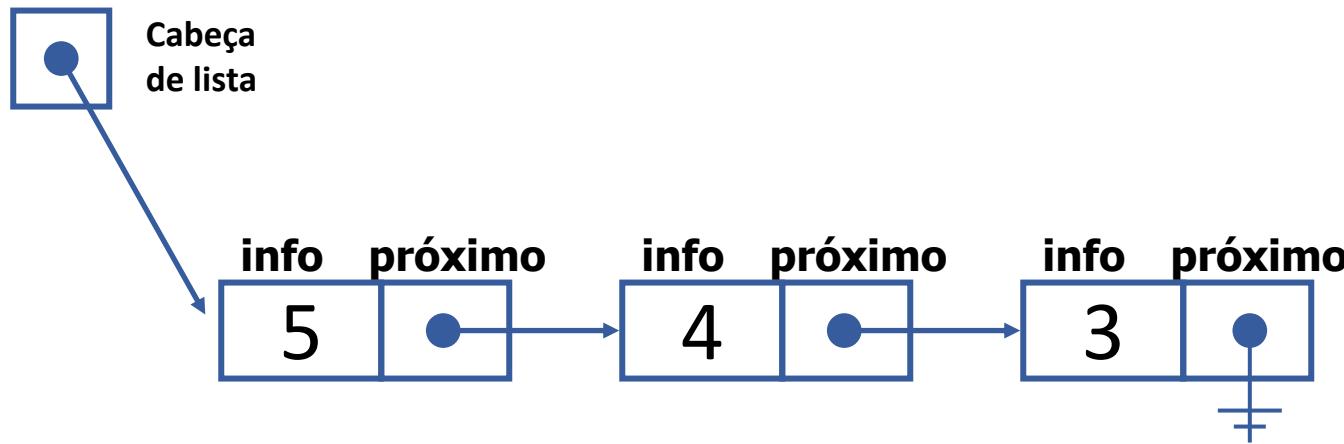


# LISTAS ENCADEADAS

- Cada elemento da lista é armazenado em um objeto
- Cada elemento da lista referencia o próximo e só é alocado dinamicamente quando é necessário
- Para referenciar o primeiro elemento é utilizado um elemento chamado cabeça da lista



# LISTAS ENCADEADAS SIMPLES



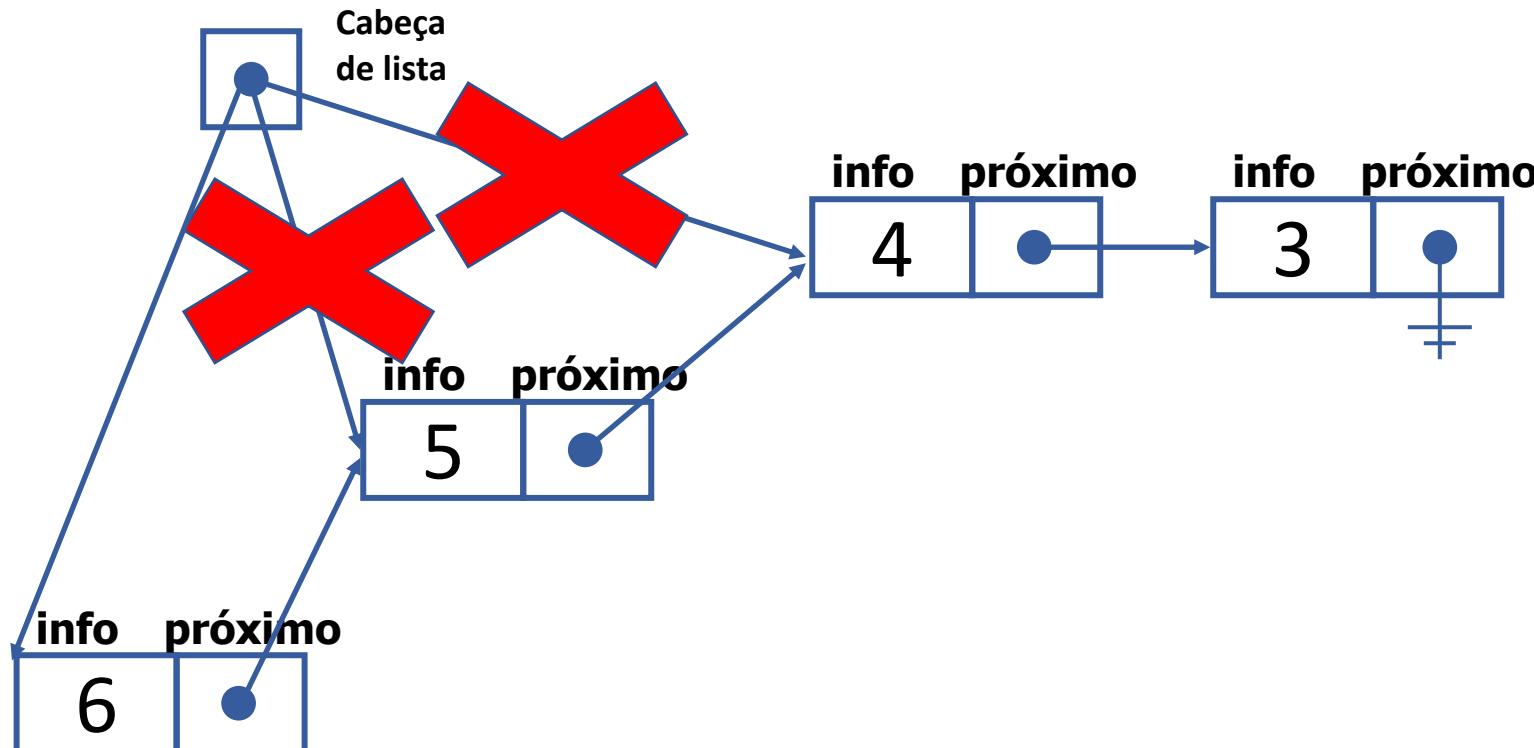
# LISTAS ENCADEADAS – OPERAÇÕES

- Insere no início
- Excluir do início
- Mostrar lista
- Pesquisar
- Excluir da posição

## LISTAS ENCADEADAS – INSERE NO INÍCIO

- Insere um novo nó no início da lista
- É o local mais fácil para inserir um nó
- O *próximo* deste novo elemento deve ser o primeiro da lista
- A cabeça da lista aponta para o novo elemento

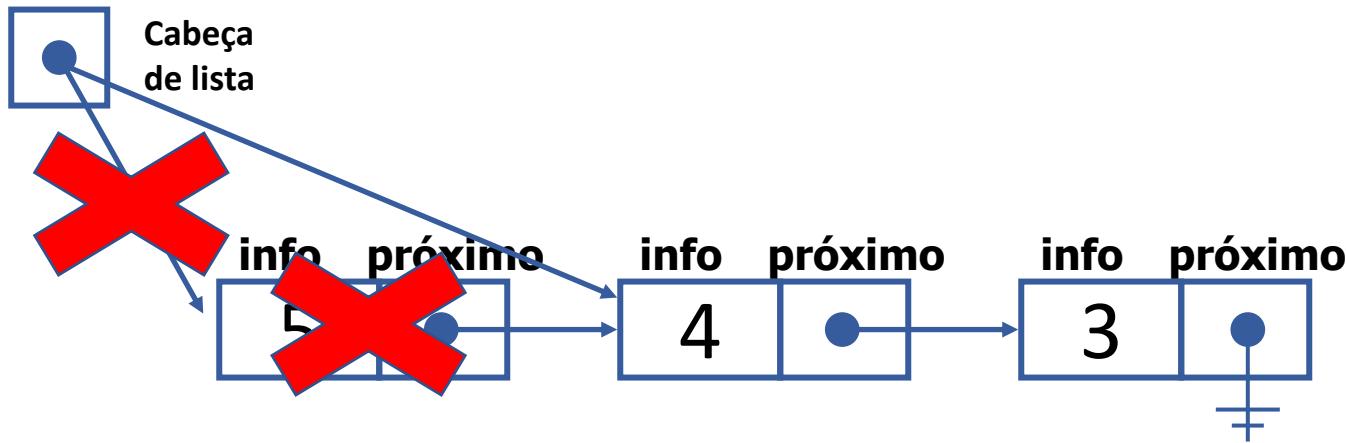
# LISTAS ENCADEADAS – INSERE NO INÍCIO



## LISTAS ENCADEADAS – EXCLUIR DO INÍCIO

- É o inverso do insere no início
- Desconecta o primeiro nó roteando de novo o primeiro para apontar para o segundo nó
- O segundo nó é encontrado por meio do campo *proximo* no primeiro nó

# LISTAS ENCADEADAS – EXCLUIR DO INÍCIO



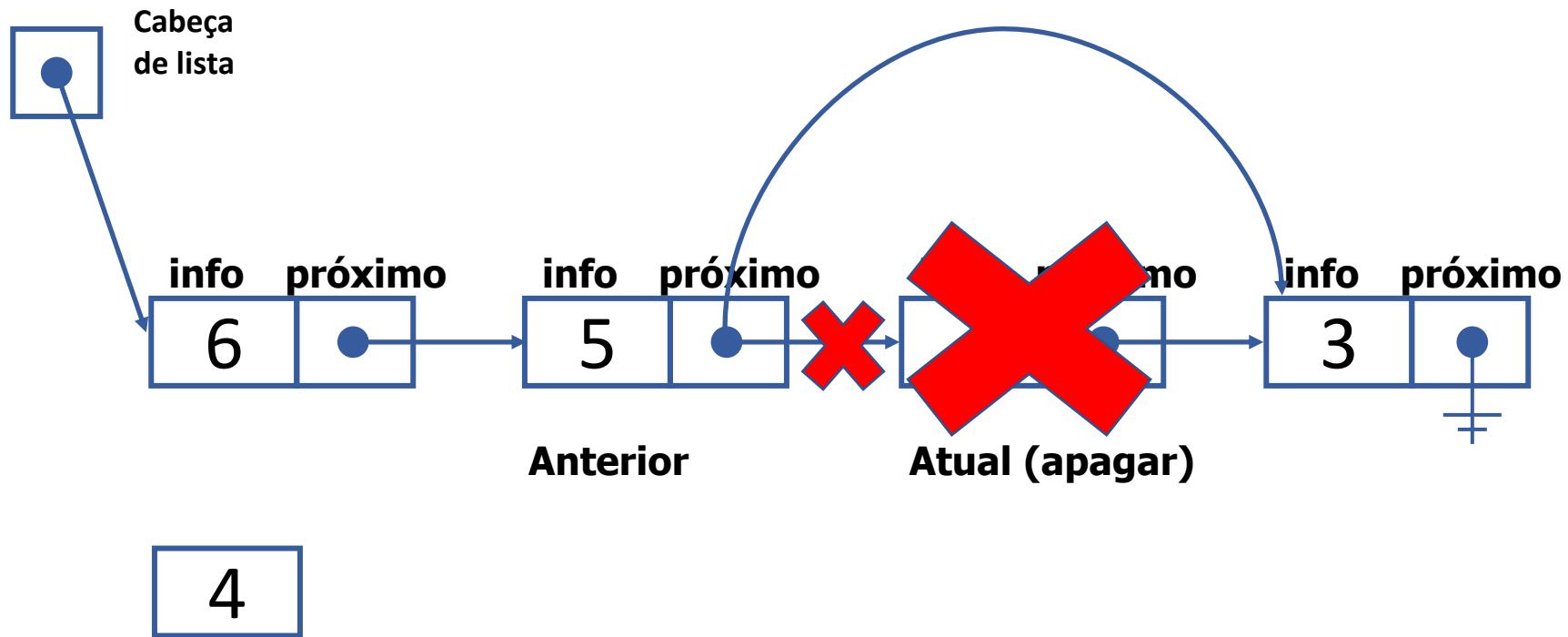
## LISTAS ENCADEADAS – MOSTRAR LISTA E PESQUISAR

- Para exibir a lista, deverá ser iniciado no primeiro elemento, seguindo a sequência de referências de nó em nó
- O final da lista é indicado pelo campo *próximo* no último nó apontando para *null/none* ao invés de outro nó
- Os nós são percorridos e é verificado se o valor do elemento é aquele que está sendo procurado
- Se atingir o final da lista sem encontrar o nó desejado, finaliza sem encontrar o elemento

## LISTAS ENCADEADAS – EXCLUIR DA POSIÇÃO

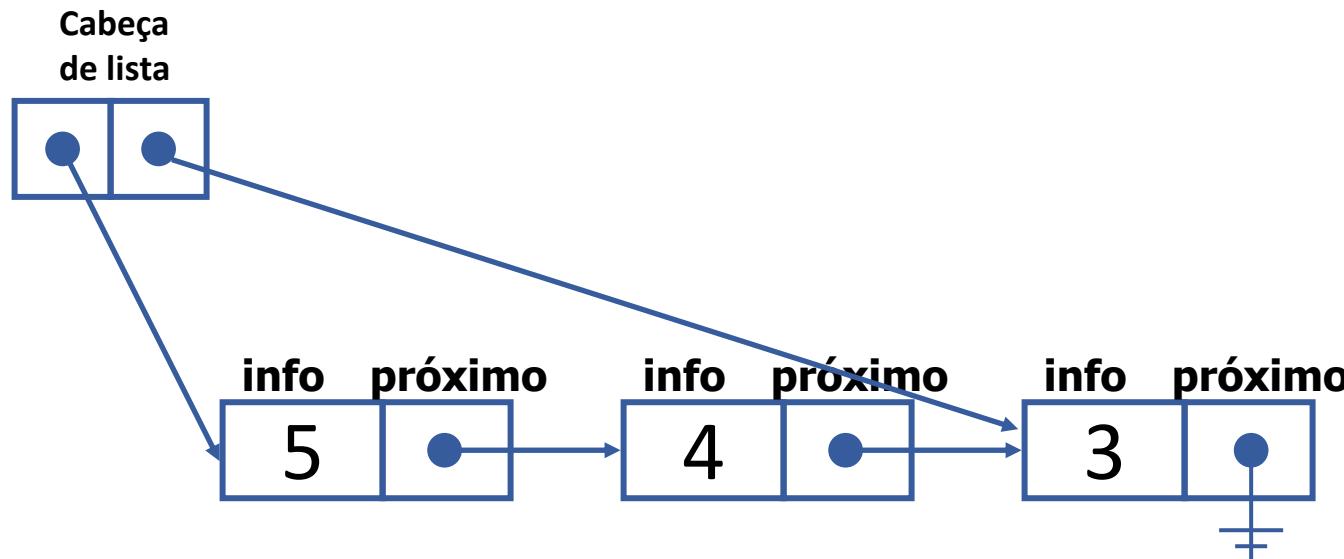
- O nó a ser eliminado deve ser localizado (pesquisa)
- Quando elimina o nó atual, terá que conectar o nó anterior ao nó seguinte

# LISTAS ENCADEADAS – EXCLUIR DA POSIÇÃO



# LISTAS ENCADEADAS COM EXTREMIDADES DUPLAS

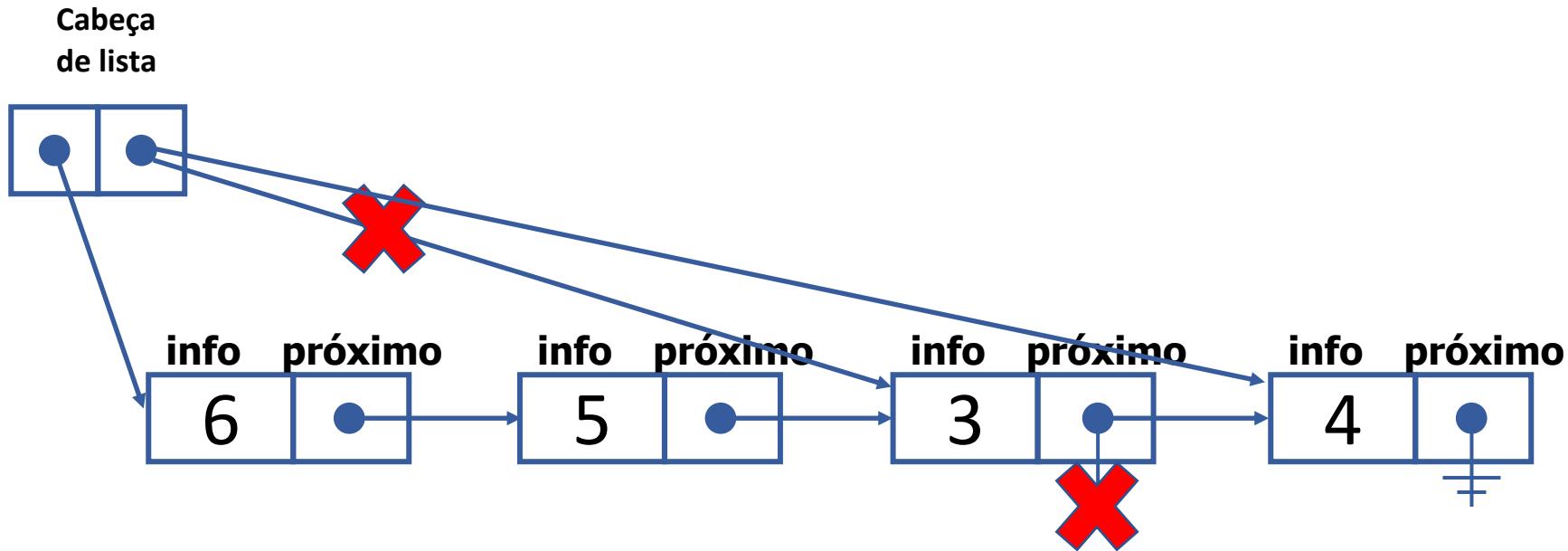
- Possui referência para o primeiro e para o último nó



# LISTAS ENCADEADAS COM EXTREMIDADES DUPLAS

- Para inserir um nó no final de uma lista com extremidade simples, é necessário percorrer todos os elementos
- A referência para o último nó permite inserir um novo nó diretamente no final da lista, assim como no início
- Operações
  - Inserir no início
  - **Inserir no final (adicional)**
  - Excluir do início

# LISTAS ENCADEADAS COM EXTREMIDADES DUPLAS – INSERIR NO FINAL



## LISTAS ENCADEADAS

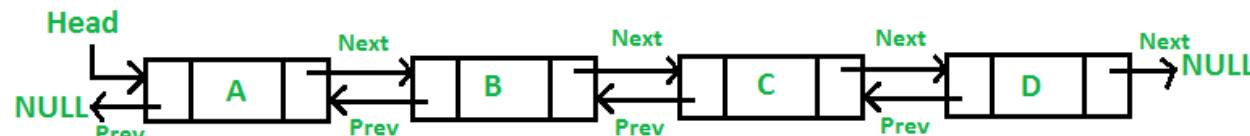
- Inserção e exclusão no início e no final (utilizando listas com extremidades duplas) são muito rápidas - tempo  $O(1)$
- Localizar, eliminar ou inserir próximo a um item específico requer buscar, em média, metade dos itens – requer  $O(N)$  comparações. Semelhante a um vetor, porém, na lista encadeada não é necessário mover itens
- Utiliza somente a memória que necessitar, podendo ser expandida de acordo com o aumento da lista

# LISTAS DUPLAMENTE ENCADEADAS

- Problema das listas encadeadas simples
  - Um comando *atual = atual.proximo* vai para o próximo nó, mas não há maneira correspondente de ir para o nó anterior
- Exemplo
  - Em um editor de texto implementado com uma lista encadeada simples, não seria possível voltar para o caractere anterior

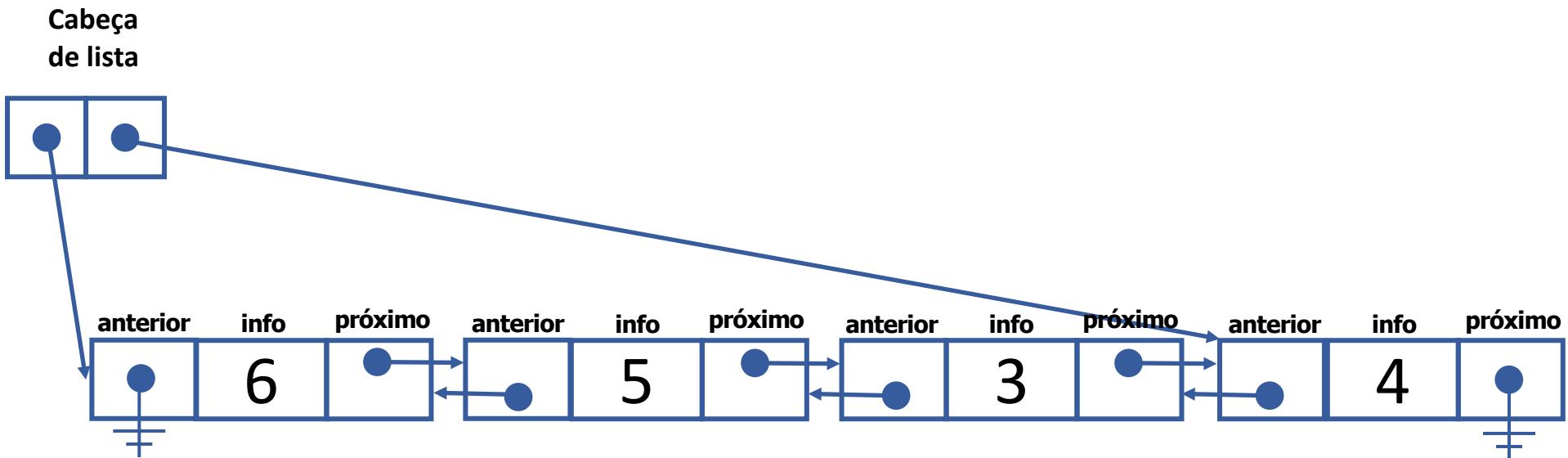
# LISTAS DUPLAMENTE ENCADEADAS

- Permite percorrer a lista para trás, assim como para frente
- Cada nó tem duas referências para outros nós, ao invés de uma
- A primeira referência é para o próximo nó e a segunda é para o nó anterior

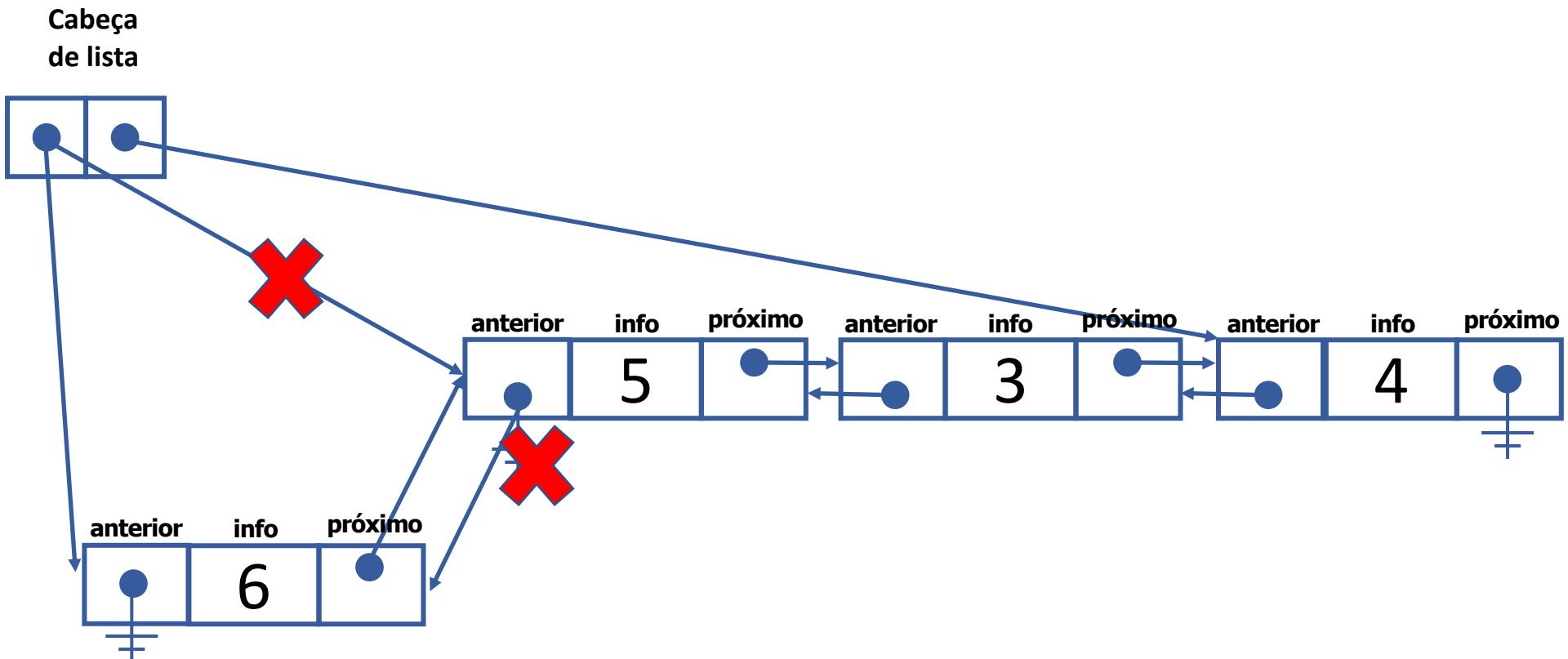


Fonte: <https://www.geeksforgeeks.org/doubly-linked-list/>

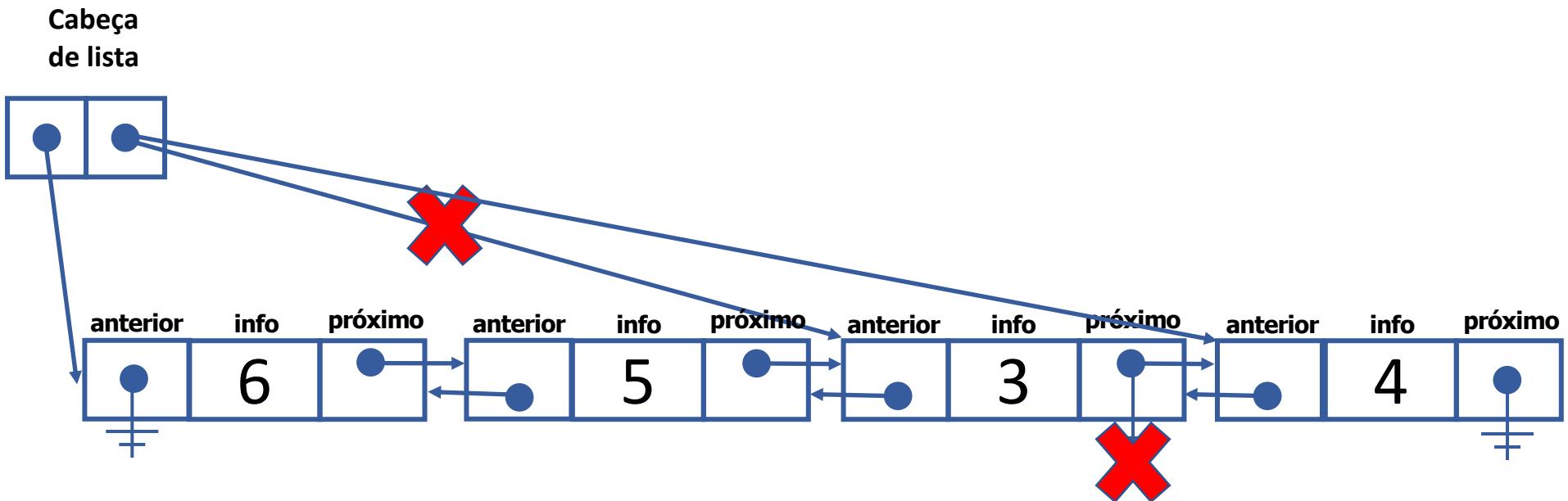
# LISTAS DUPLAMENTE ENCADEADAS



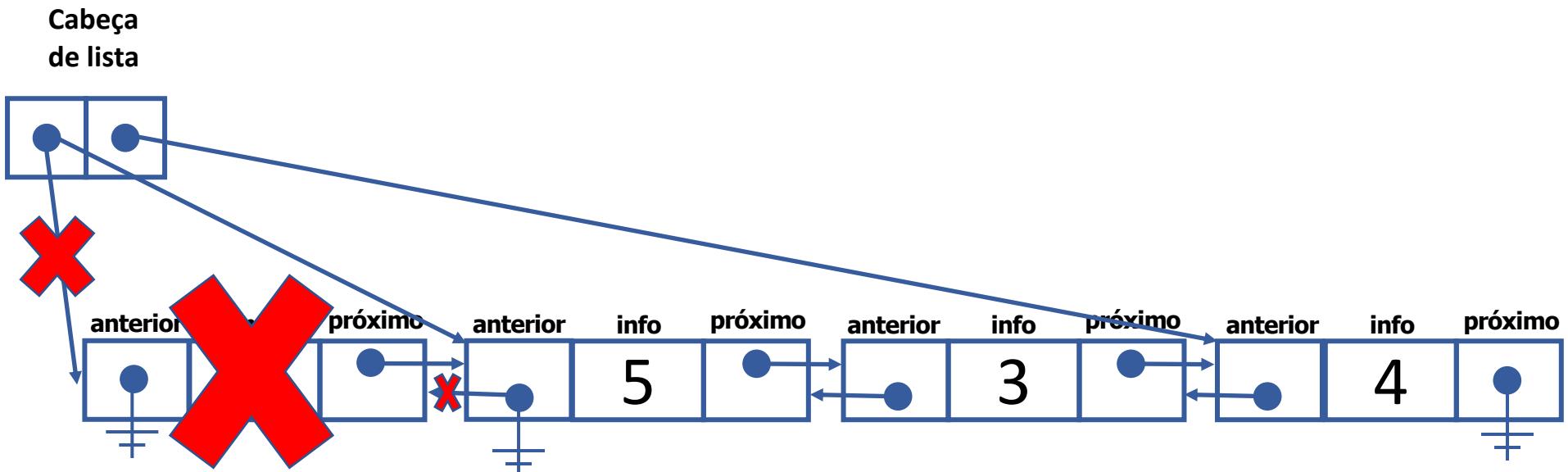
# LISTAS DUPLAMENTE ENCADEADAS – INSERE NO INÍCIO



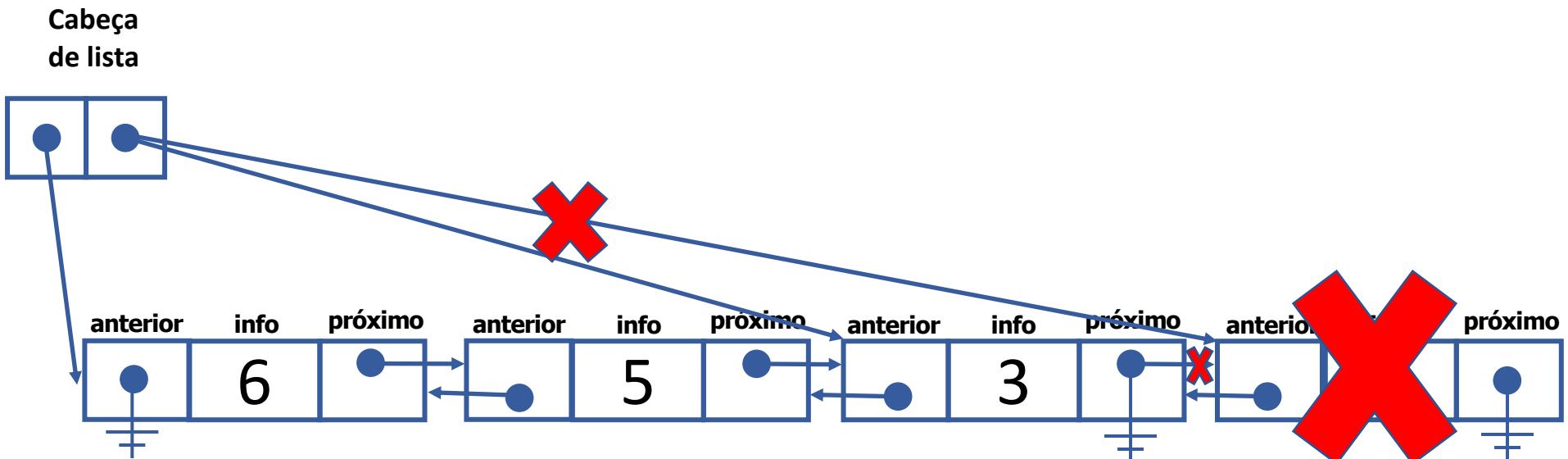
# LISTAS DUPLAMENTE ENCADEADAS – INSERE NO FINAL



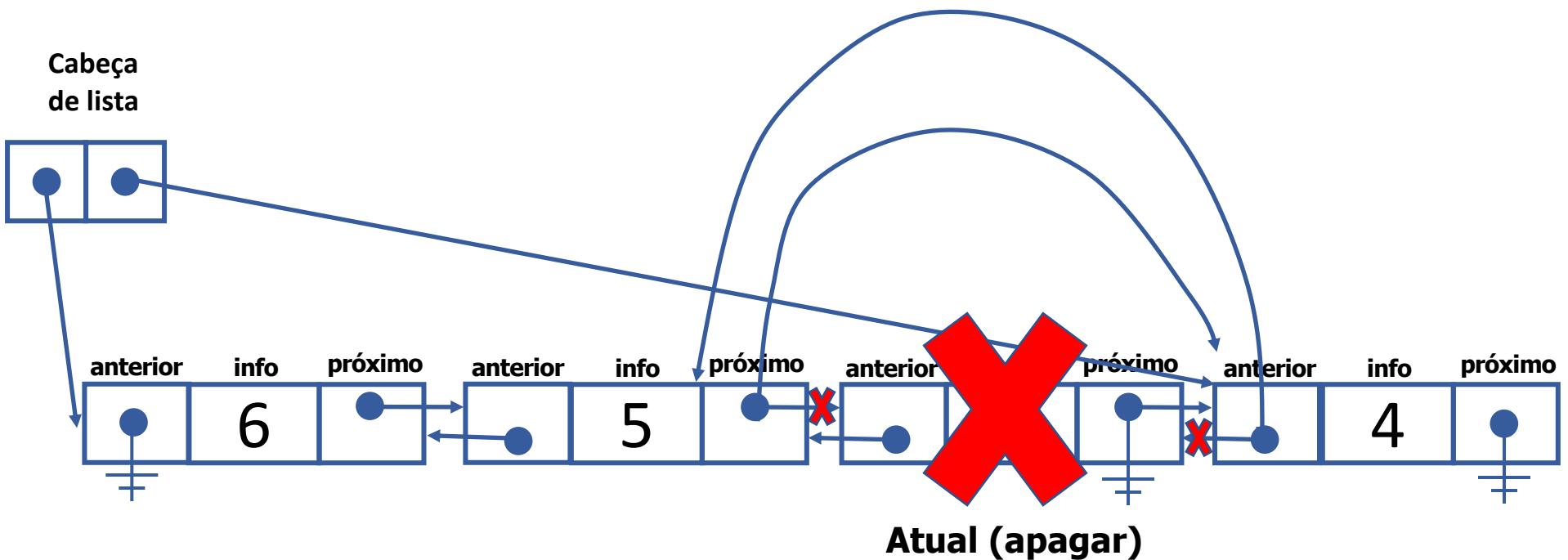
# LISTAS DUPLAMENTE ENCADEADAS – EXCLUIR DO INÍCIO



# LISTAS DUPLAMENTE ENCADEADAS – EXCLUIR DO FINAL



# LISTAS DUPLAMENTE – EXCLUIR DA POSIÇÃO

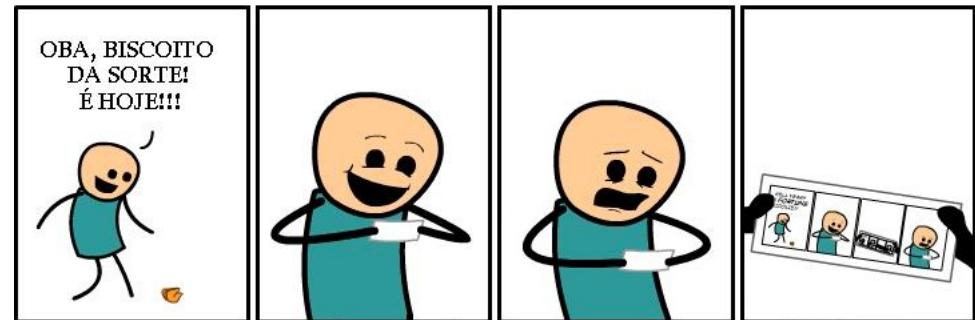


# RECURSÃO

- Quando uma função chama ela mesma
- Repetição de tarefas quando uma estrutura de repetição não é adequada
- Construção de estruturas de dados específicas



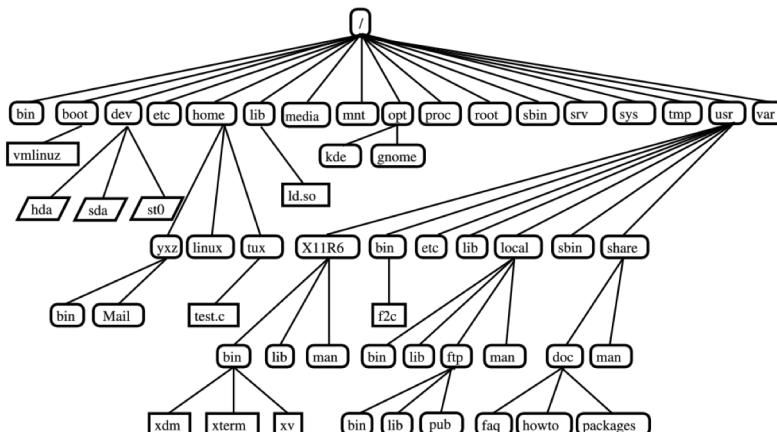
Fonte: <https://knowyourmeme.com/>



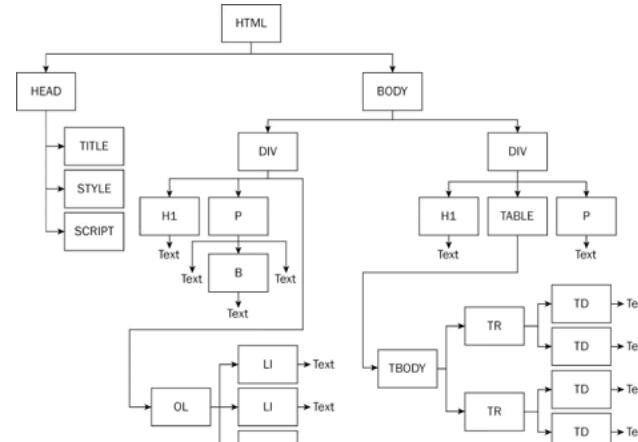
Fonte: <http://geraldoferraz.blogspot.com/2011/11/recursividade.html>

# ÁRVORES

- Uma árvore combina as vantagens de duas estruturas: um vetor ordenado e uma lista encadeada
- Busca rápida (como em um vetor ordenado)
- Inserção e eliminação rápida (como em uma lista encadeada)



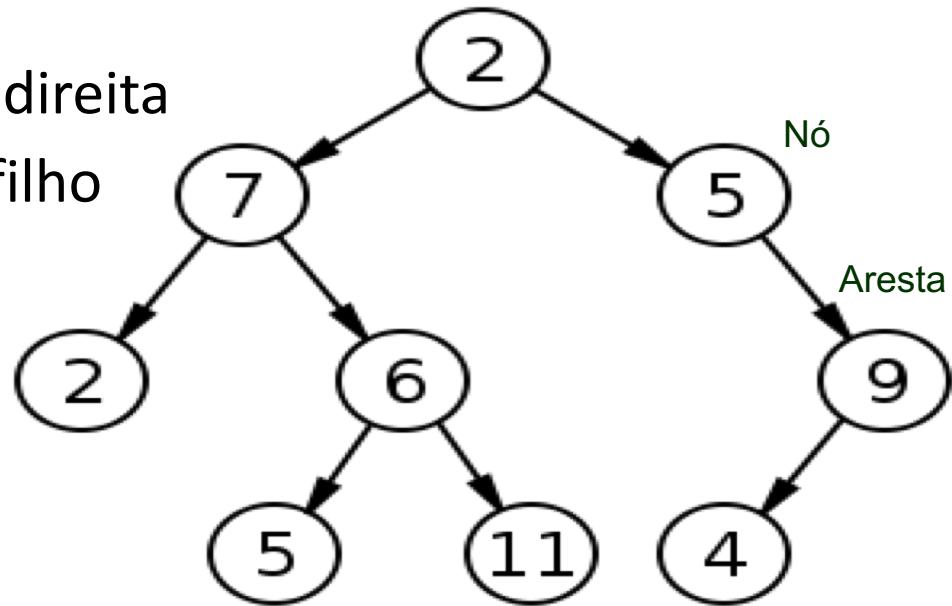
Fonte: <https://www.esli-nux.com/2016/02/aula-5-estrutura-da-arvore-do-sistema.html>



Fonte: <http://devfuria.com.br/javascript/dom/>

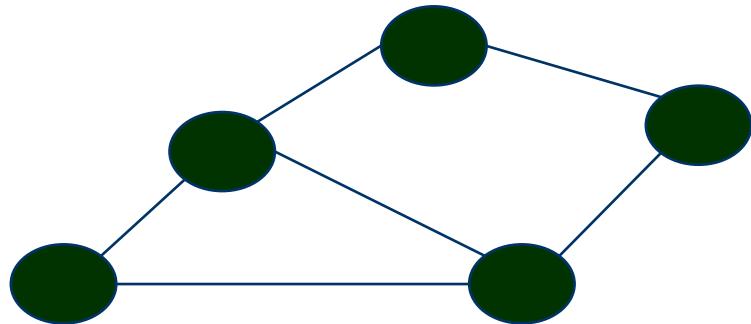
# ÁRVORE BINÁRIA

- Uma árvore consiste em nós (círculos) conectados por arestas (linhas)
- Máximo dois filhos
- Filho à esquerda e filho à direita
- Pode ter um ou nenhum filho

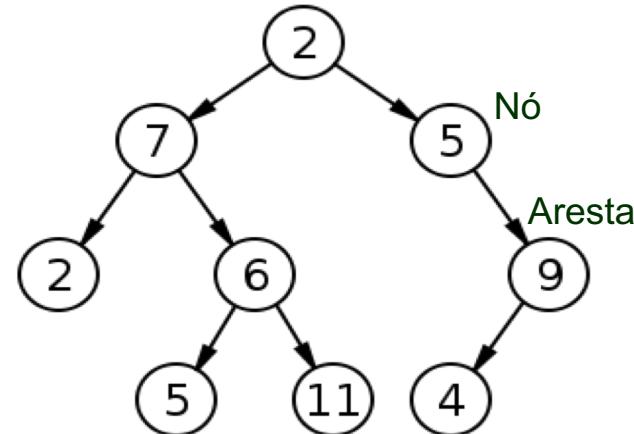


# TERMINOLOGIA DE ÁRVORES

- Caminho
  - Caminho que liga um nó até outro nó
- Raiz
  - É o nó na parte superior. Há apenas uma raiz em uma árvore e deve haver somente um caminho da raiz até qualquer outro nó

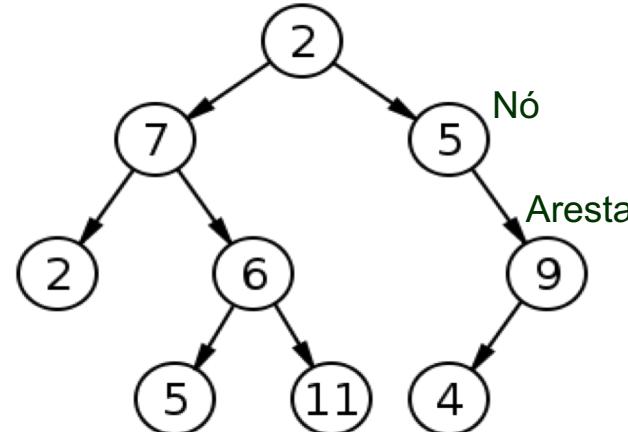


Não-árvore



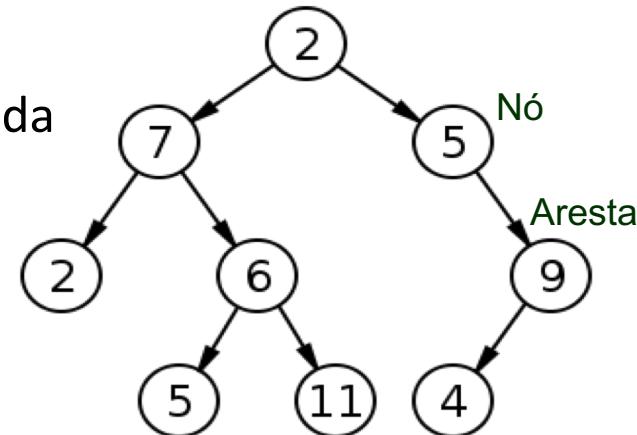
# TERMINOLOGIA DE ÁRVORES

- Pai
  - Qualquer nó (exceto a raiz) tem exatamente uma aresta que sobe para outro nó. O nó acima dele é chamado de pai do nó
- Filho
  - Qualquer nó pode ter uma ou mais linhas descendo para outros nós. Esses nós abaixo de um dado nó são chamados de seus filhos
- Folha
  - Um nó que não tem filhos



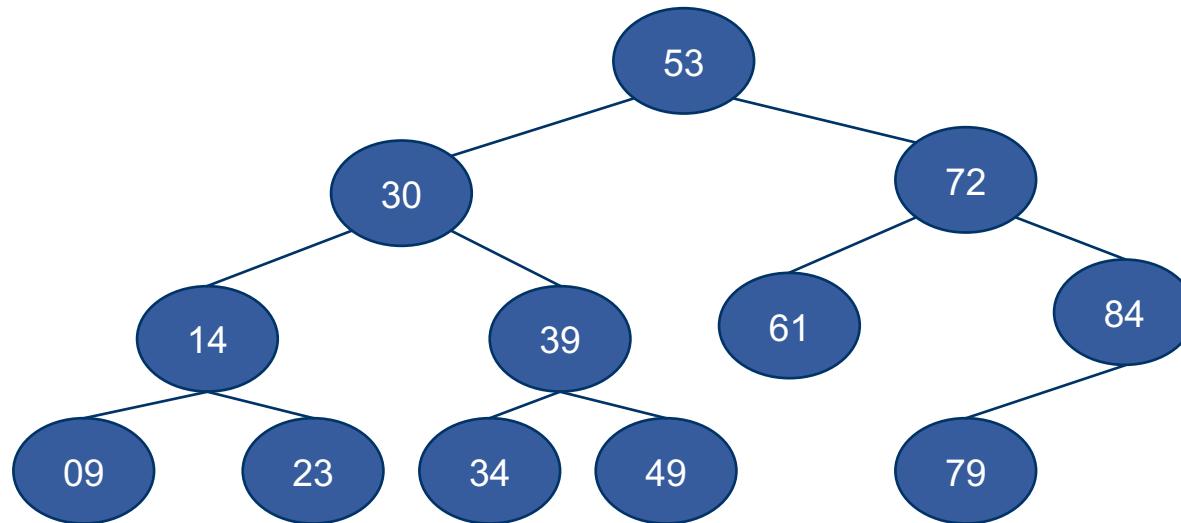
# TERMINOLOGIA DE ÁRVORES

- Subárvore
  - Qualquer nó pode ser considerado como sendo a raiz de uma subárvore, que consiste em seus filhos
- Visitando
  - Um nó é visitado quando o controle do programa chega ao nó, em geral para a finalidade de executar alguma operação do nó
- Percorrendo
  - Visitar todos os nós em alguma ordem especificada
- Níveis
  - O nível de um determinado nó refere-se a quantas gerações o nó está da raiz
- Chaves
  - Valor usado para buscar um item



# ÁRVORE BINÁRIA DE BUSCA

- O filho à esquerda de um nó tem que ter uma chave menor que seu pai e o filho à direita de um nó tem que ter uma chave maior ou igual ao seu pai



# ÁRVORE BINÁRIA DE BUSCA – INSERÇÃO

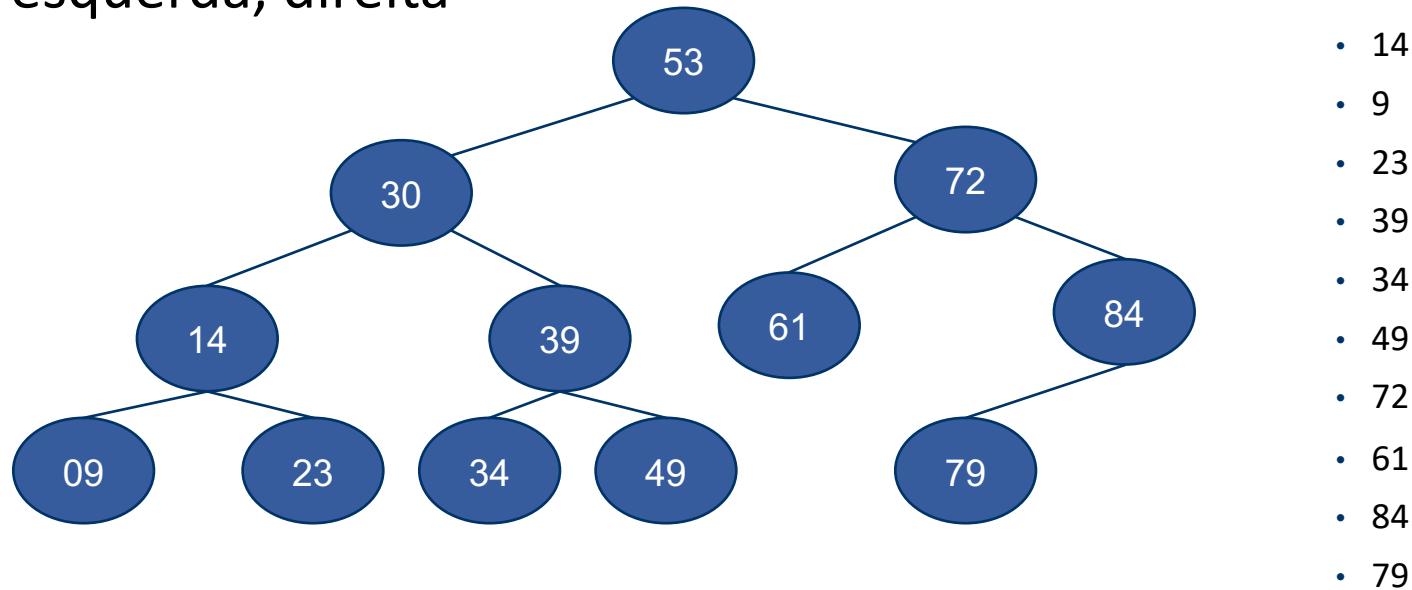
- Primeiro, o local para inserir deve ser encontrado
- Segue-se o caminho da raiz até o devido nó, que será pai do novo nó
- Quando esse pai for localizado, o novo nó será conectado como seu filho à esquerda ou a direita, dependendo da chave do novo nó ser menor ou maior que a do pai
- Visualização on-line: <https://visualgo.net/en/bst>
- Big-O:  $O(\log n)$  para o caso médio e  $O(n)$  no pior caso

# ÁRVORE BINÁRIA DE BUSCA – PESQUISA

- Procurar nas subárvore da esquerda ou direita
- Visualização on-line: <https://visualgo.net/en/bst>
- Big-O:  $O(\log n)$  para o caso médio e  $O(n)$  no pior caso

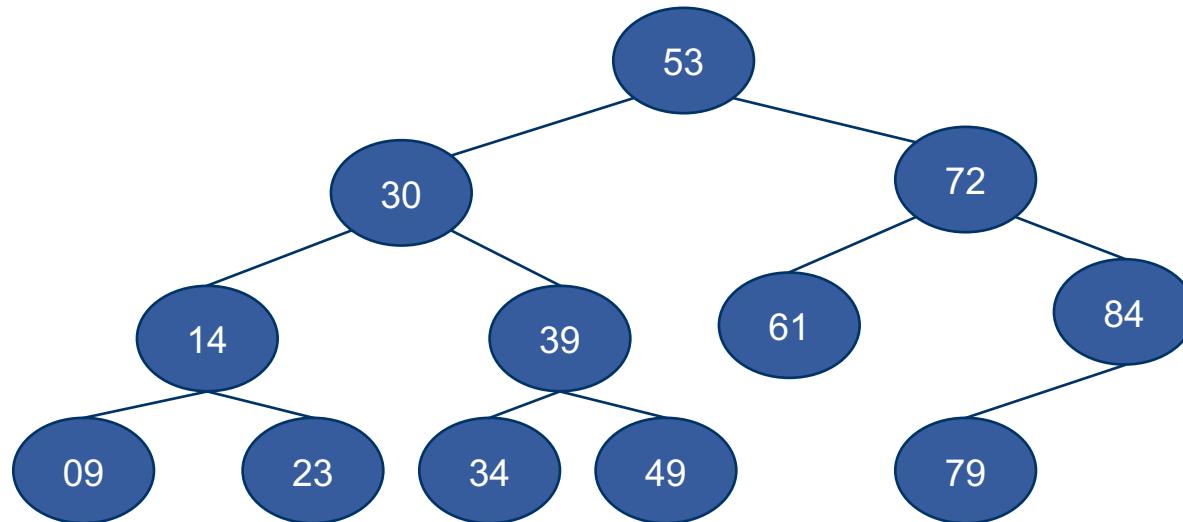
# ÁRVORE BINÁRIA – TRAVESSIA PRÉ-ORDEM

- Primeiro visita a raiz e depois recursivamente faz uma travessia na subárvore esquerda, seguido de uma travessia recursiva na subárvore direita
- Raiz, esquerda, direita



# ÁRVORE BINÁRIA – TRAVESSIA EM ORDEM

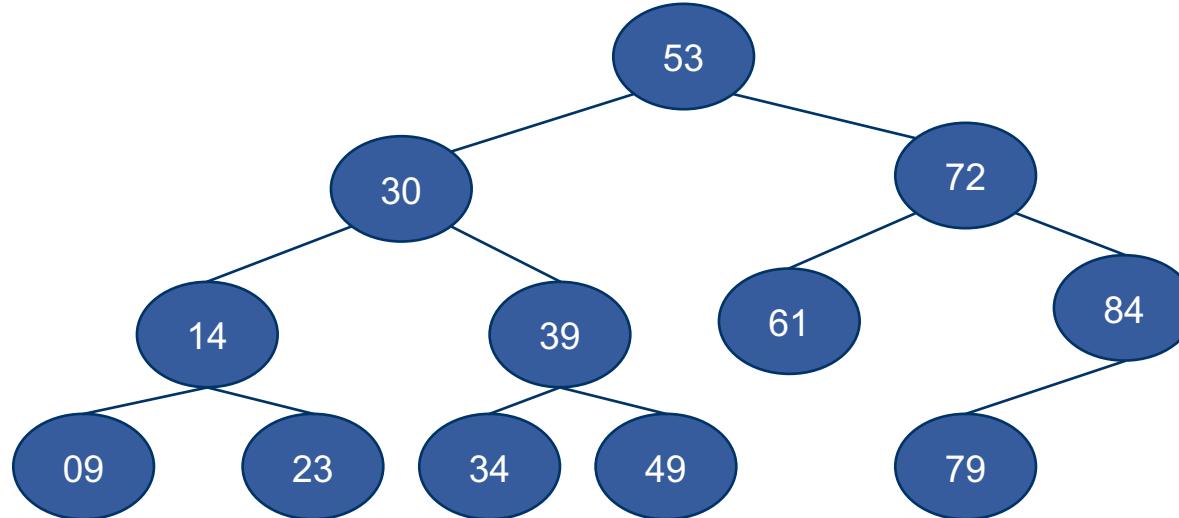
- Recursivamente faz a travessia na subárvore esquerda, visita a raiz e faz uma travessia recursiva na subárvore direita
- Esquerda, raiz, direita



- 09
- 14
- 23
- 30
- 34
- 39
- 49
- 53
- 61
- 72
- 79
- 84

# ÁRVORE BINÁRIA – TRAVESSIA PÓS-ORDEM

- Recursivamente faz a travessia na subárvore esquerda, faz uma travessia recursiva na subárvore direita e por fim visita a raiz
- Esquerda, direita, raiz



- 09
- 23
- 14
- 34
- 49
- 39
- 30
- 61
- 79
- 84
- 72
- 53

## ÁRVORE BINÁRIA DE BUSCA – EXCLUSÃO

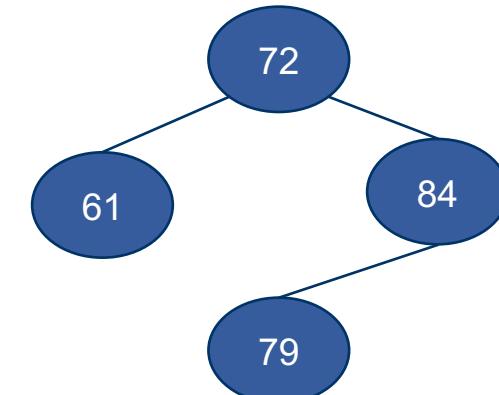
- Inicia-se localizando o nó que deseja eliminar
- Quando encontrar o nó
  - O nó a ser apagado é uma folha
  - O nó a ser apagado tem um filho
  - O nó a ser apagado tem dois filhos

# ÁRVORE BINÁRIA DE BUSCA – EXCLUSÃO (O NÓ A SER APAGADO É UMA FOLHA)

- A mais simples operação
- Altera o campo apropriado para o filho no nó pai para apontar para *null/none*, em vez do nó
- Visualização on-line:  
<https://visualgo.net/en/bst>

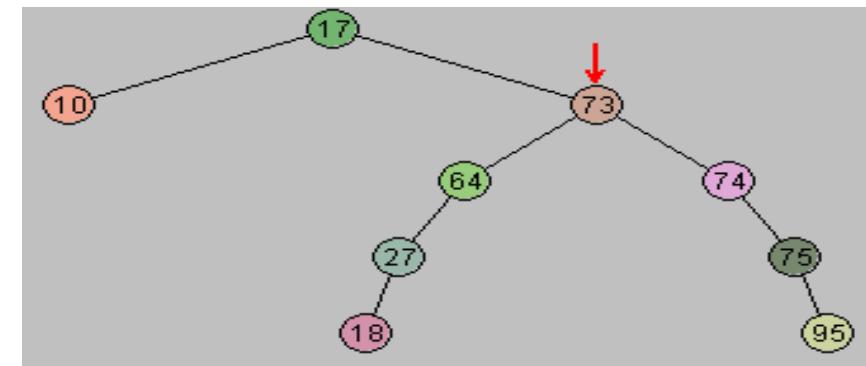
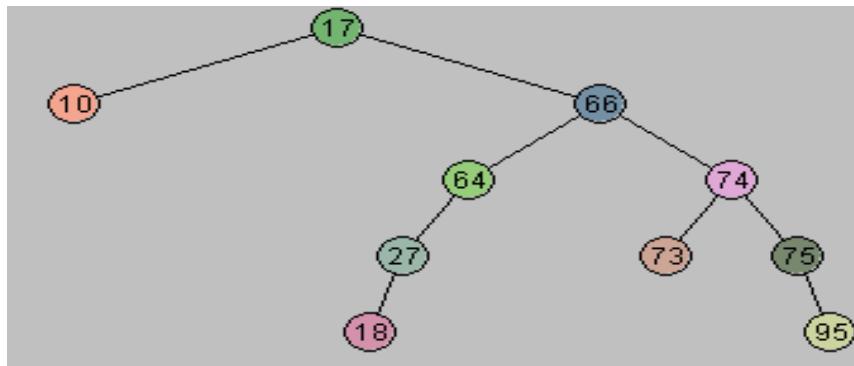
## ÁRVORE BINÁRIA DE BUSCA – EXCLUSÃO (O NÓ A SER APAGADO TEM UM FILHO)

- O nó tem apenas duas conexões: com seu pai e com seu único filho
- Deseja-se “cortar” o nó dessa sequência conectando o pai dele diretamente ao filho dele
- Visualização on-line:  
<https://visualgo.net/en/bst>



# ÁRVORE BINÁRIA DE BUSCA – EXCLUSÃO (O NÓ A SER APAGADO TEM DOIS FILHOS)

- O nó a ser apagado deve ser substituído por seu **sucessor em ordem**
- Visualização on-line:  
<https://visualgo.net/en/bst>



# ÁRVORE BINÁRIA DE BUSCA – EXCLUSÃO (O NÓ A SER APAGADO TEM UM FILHO)

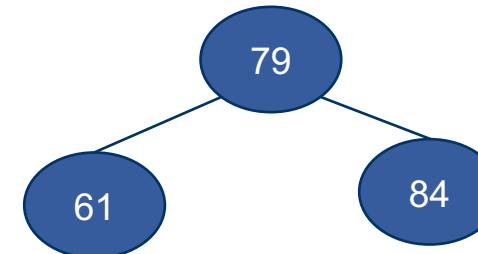
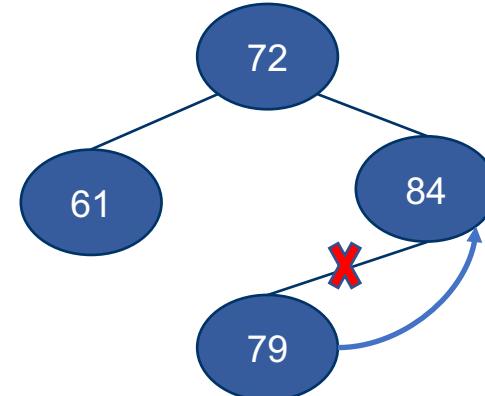
```
def get_sucessor(self, no):
    pai_sucessor = no
    sucessor = no
    atual = no.direita
    while atual != None:
        pai_sucessor = sucessor
        sucessor = atual
        atual = atual.esquerda
    if sucessor != no.direita:
        pai_sucessor.esquerda = sucessor.direita
        sucessor.direita = no.direita
    return sucessor
```

```
pai_sucessor = 72
sucessor = 72
atual = 84
Primeiro while:
    pai_sucessor = 72
    sucessor = 84
    atual = 79
```

```
Segundo while:
    pai_sucessor = 84
    sucessor = 79
    atual = None
```

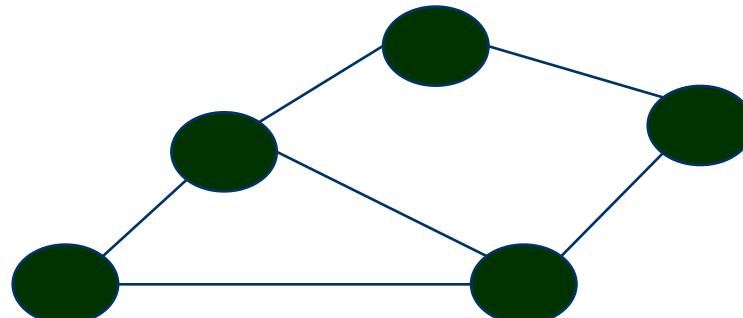
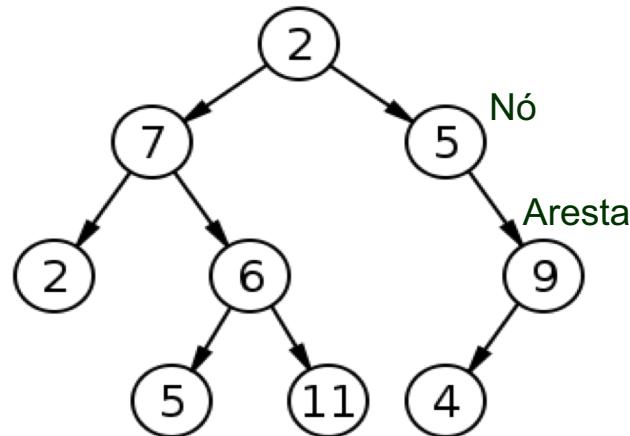
Se  $79 \neq 84$

```
pai_sucessor.esquerda (84) = None
sucessor.direita (79) = 84
```



# GRAFOS

- Árvores (uma raiz) x Grafos



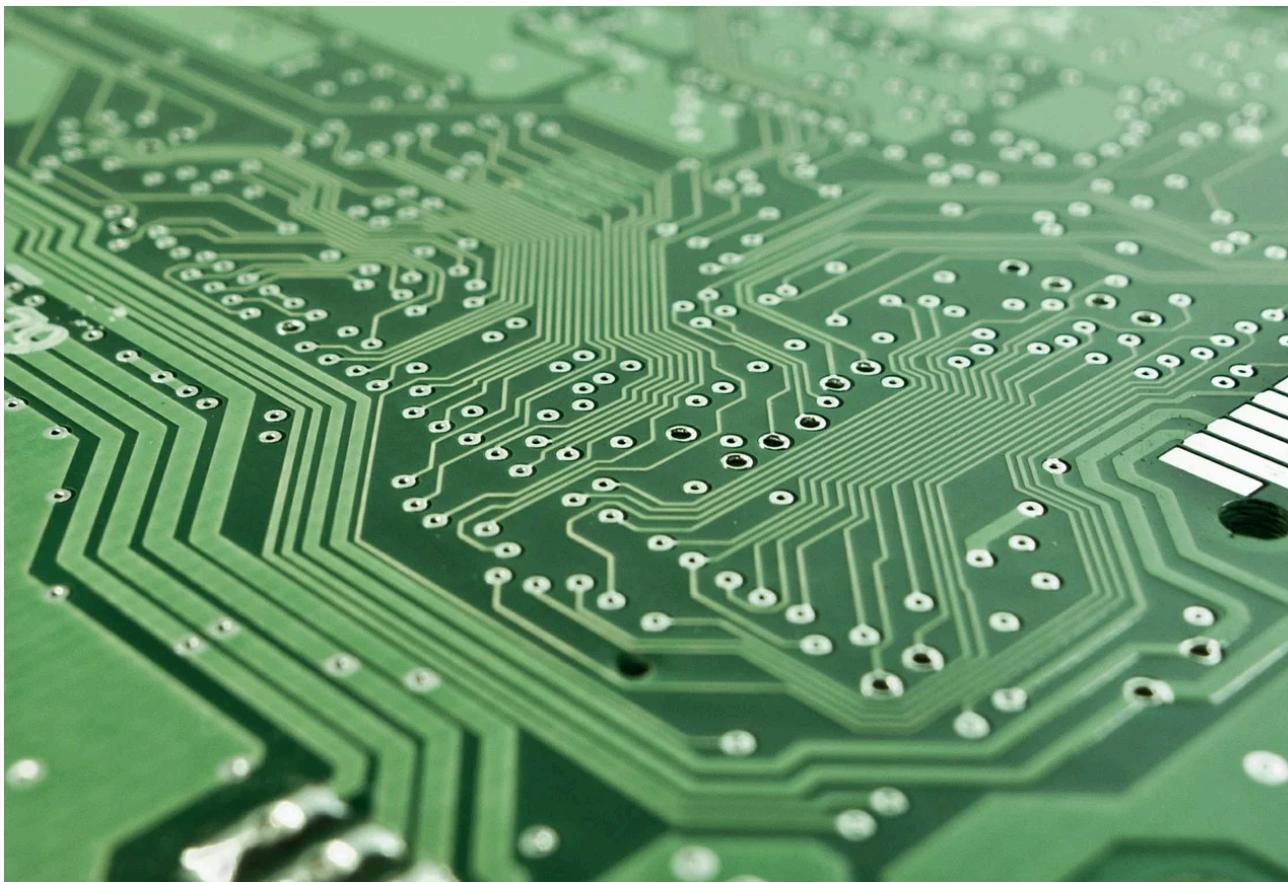
# GRAFOS



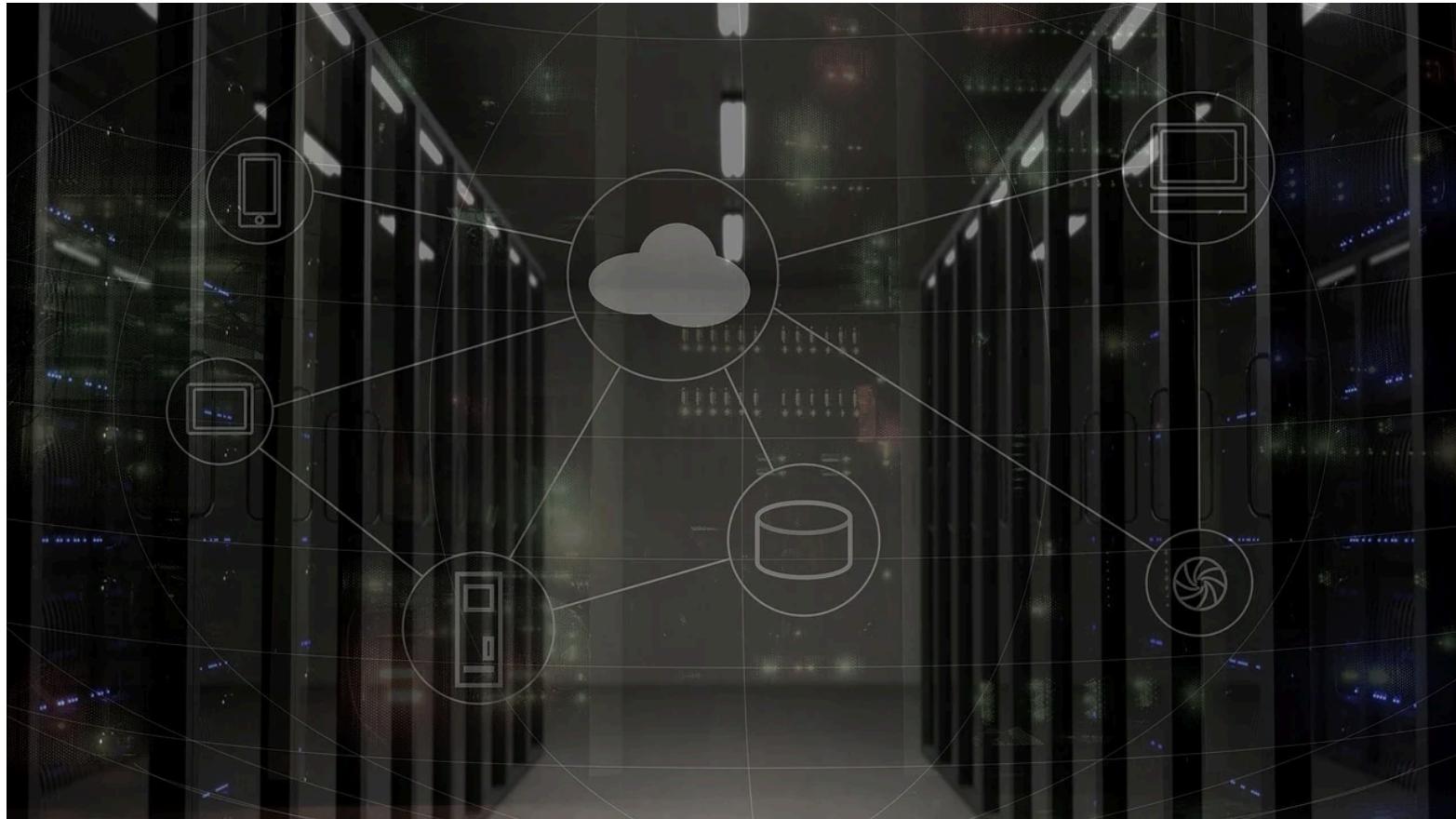
# GRAFOS



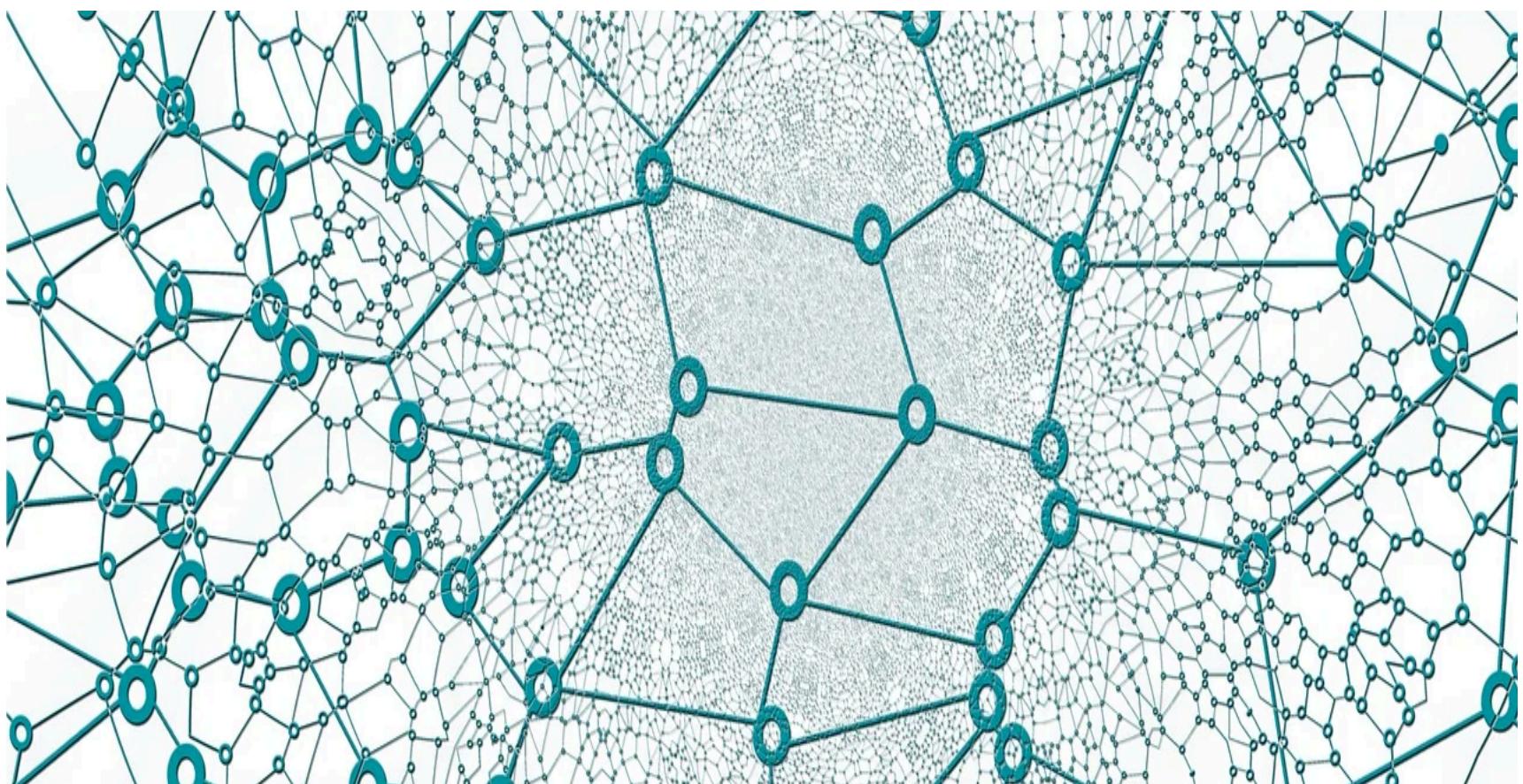
# GRAFOS



# GRAFOS



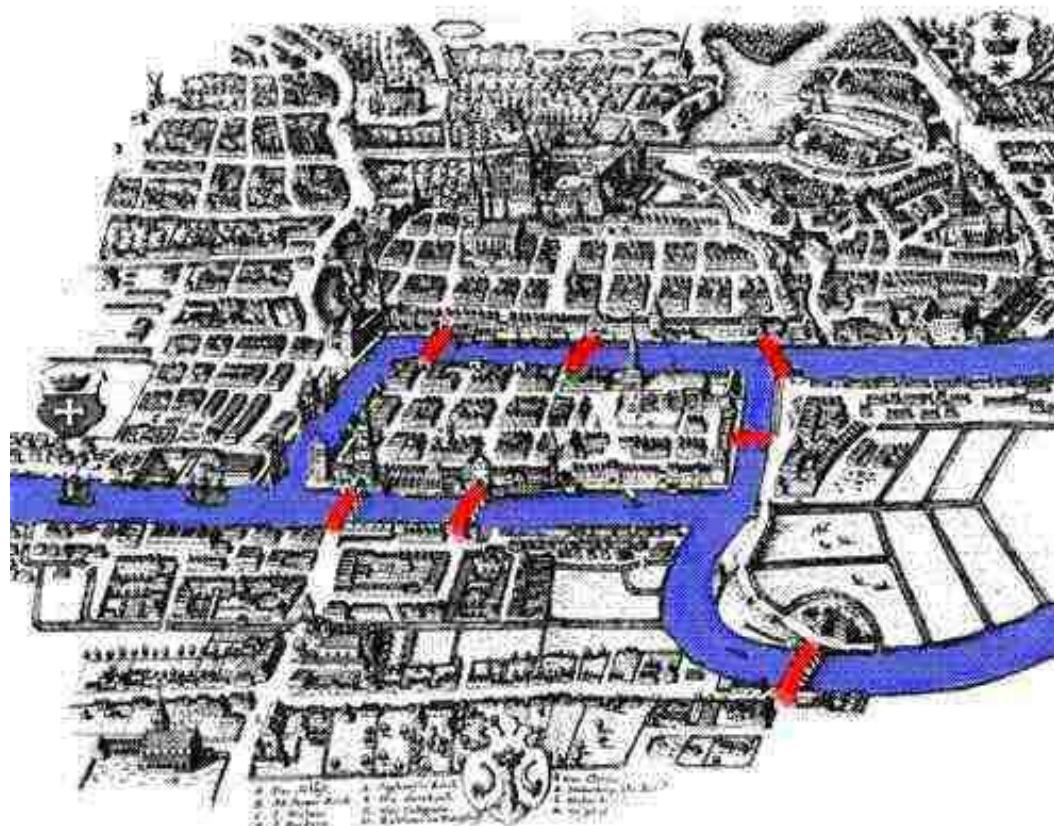
# GRAFOS



# GRAFOS

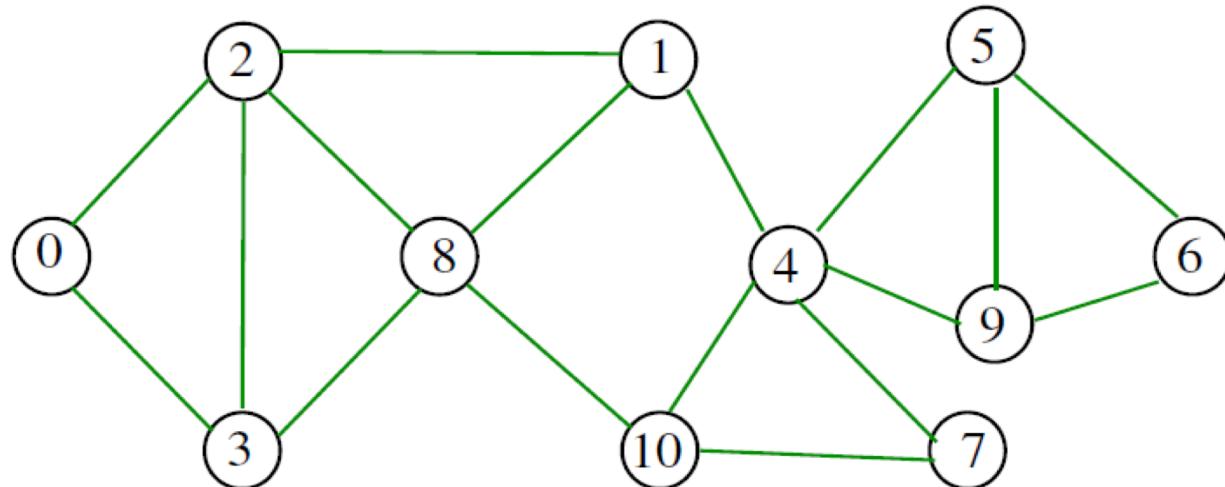


# GRAFOS (HISTÓRICO) – PONTES DE KÖNIGSBERG – LEONHARD EULER

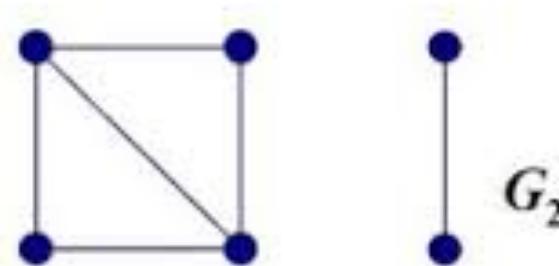
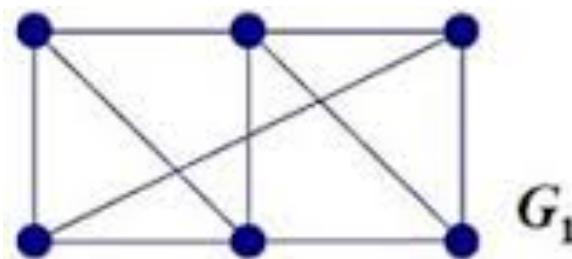


# GRAFOS

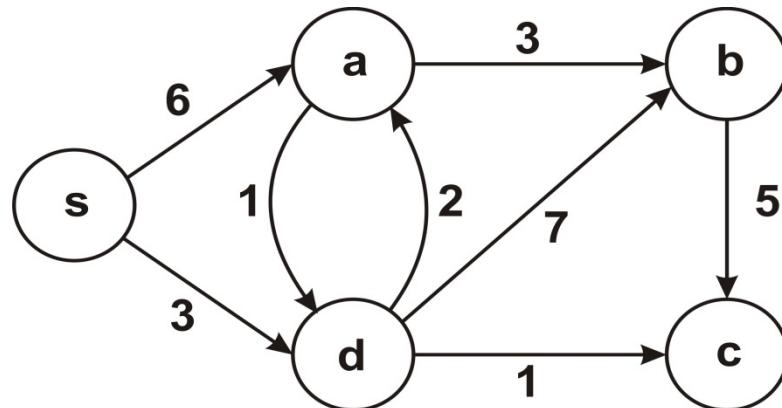
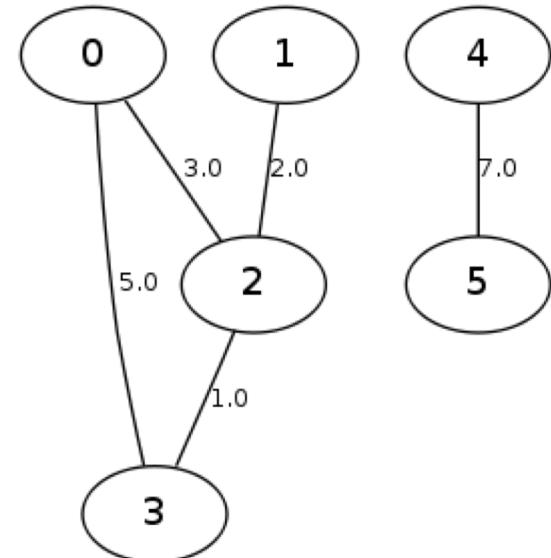
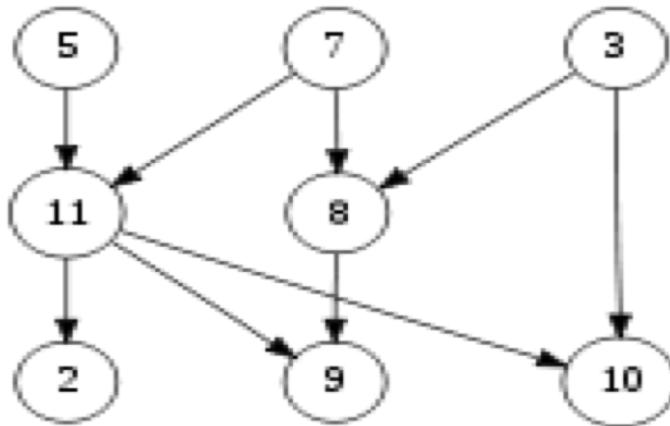
- Adjacência
  - Se um nó for conectado a outro por uma única aresta
- Caminhos
  - Sequência de arestas



# GRAFOS CONECTADOS E NÃO CONECTADOS

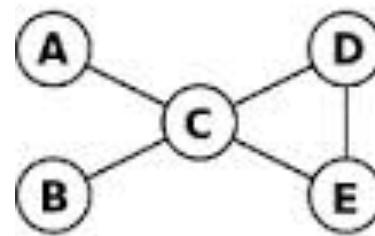


# GRAFOS ORIENTADOS E PONDERADOS



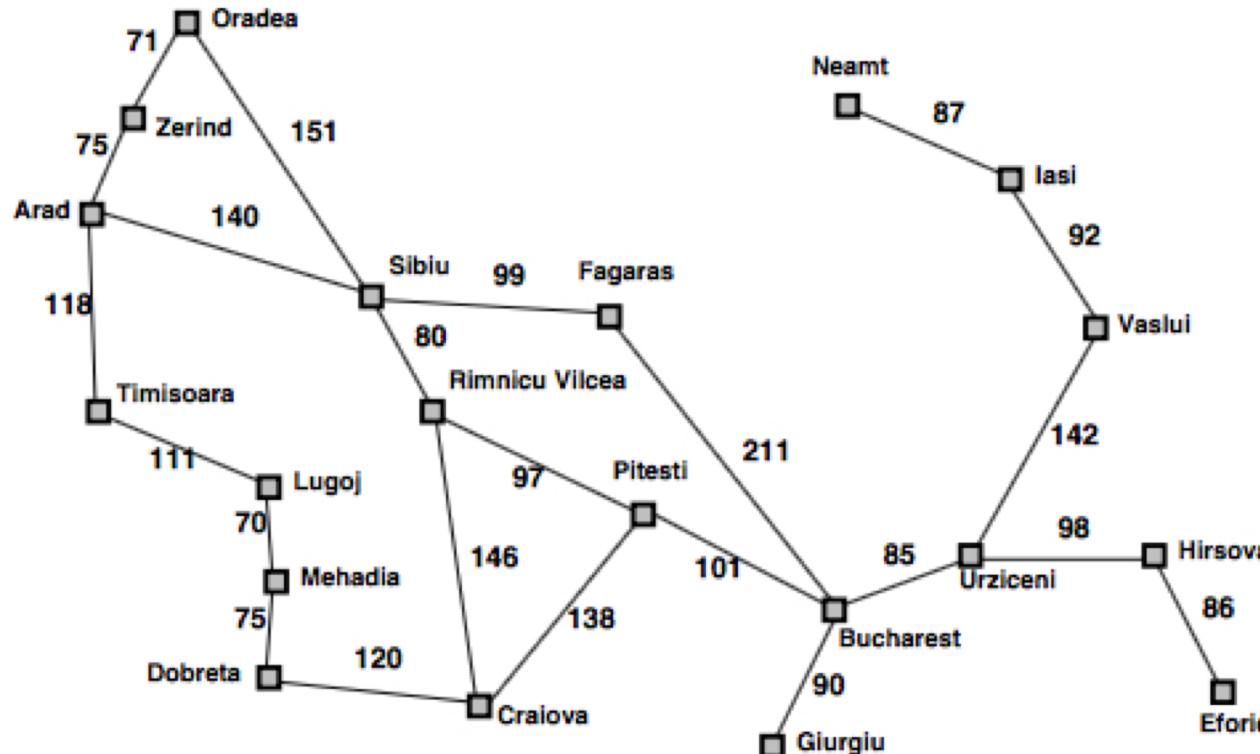
# NÓS E ARESTAS

- Um nó representa algum objeto do mundo real
- Uma aresta faz a ligação entre os nós
- Representação
  - Matriz de adjacência
  - Lista de adjacências



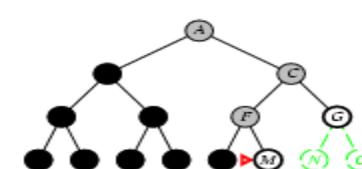
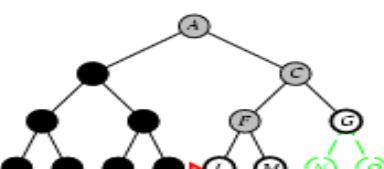
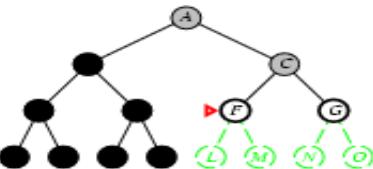
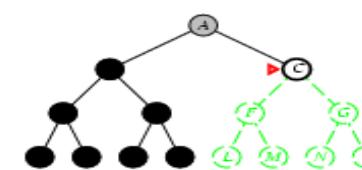
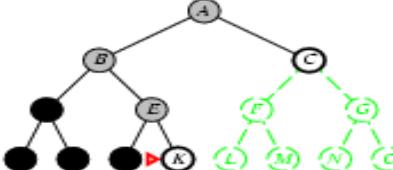
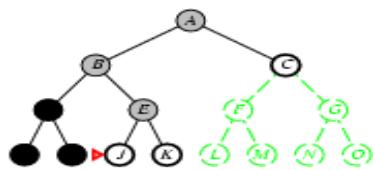
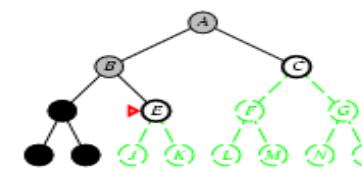
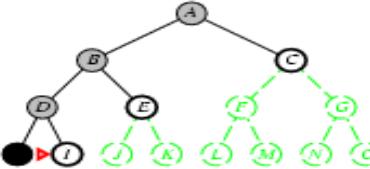
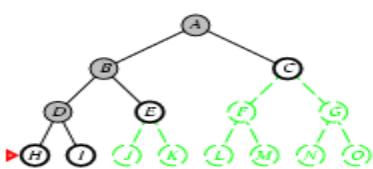
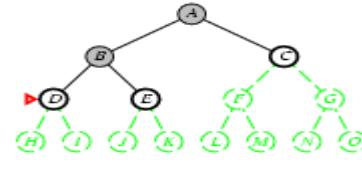
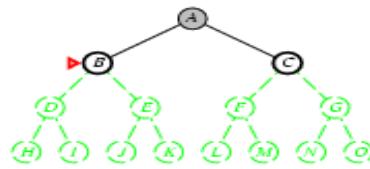
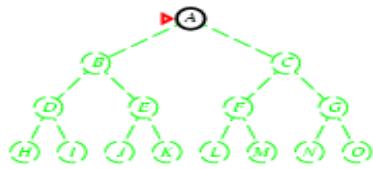
	A	B	C	D	E
A	0	0	1	0	0
B	0	0	1	0	0
C	1	1	0	1	1
D	0	0	1	0	1
E	0	0	1	1	0

# GRAFO – ARAD ATÉ BUCARESTE



Fonte: Livro Inteligência Artificial – Stuart Russell e Peter Norvig

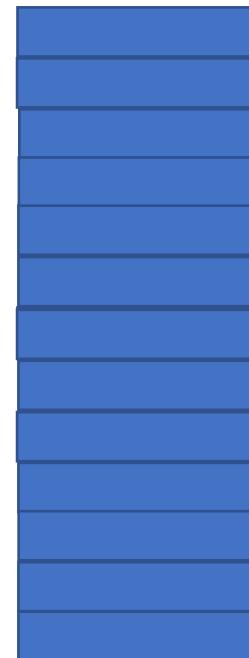
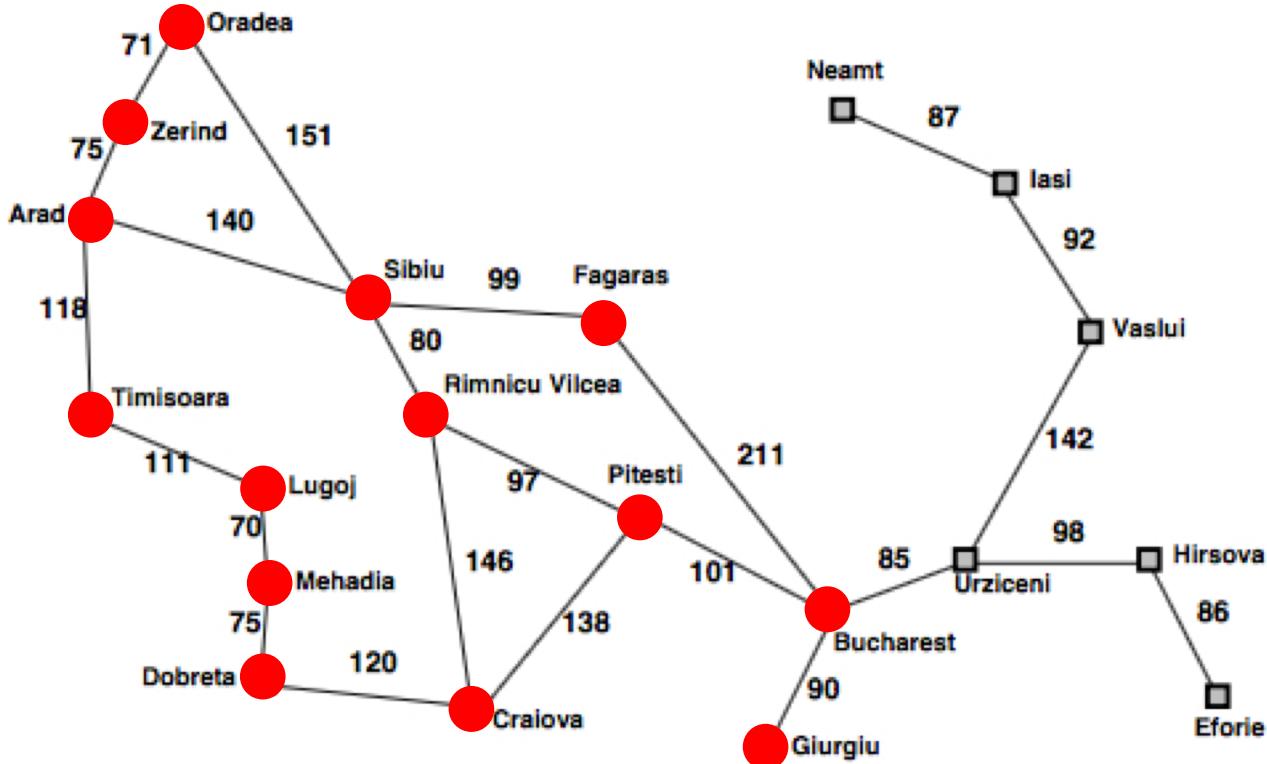
# GRAFO – BUSCA EM PROFUNDIDADE



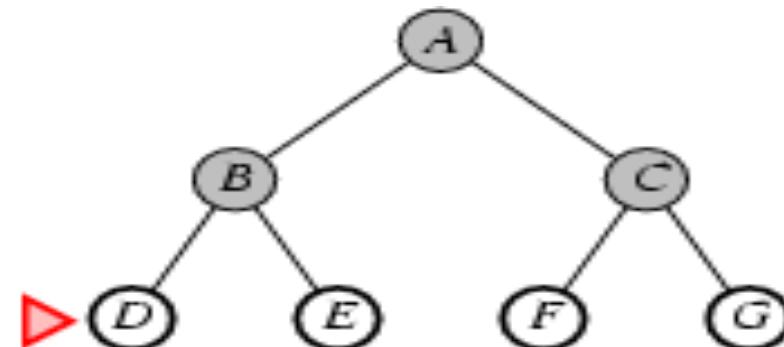
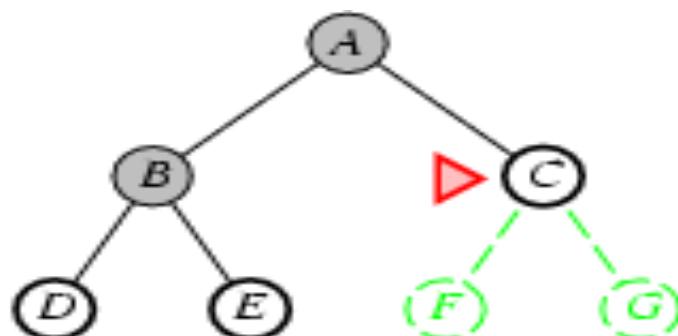
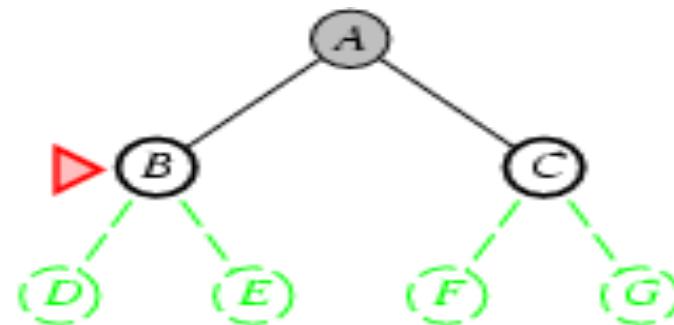
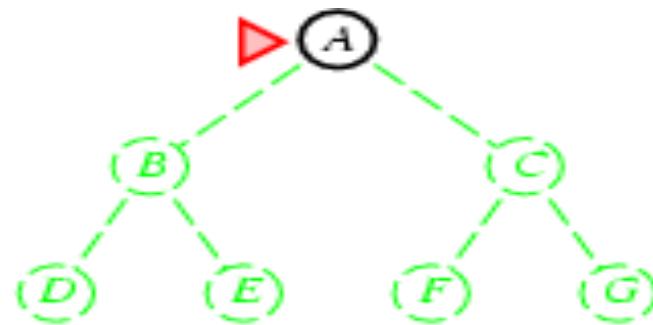
# GRAFO – BUSCA EM PROFUNDIDADE

- Regra 1
  - Visite um nó adjacente não visitado, marque-o e coloque-o na pilha
- Regra 2
  - Se não puder seguir a Regra 1 retire um nó da pilha
- Regra 3
  - Se não puder seguir a Regra 1 ou a Regra 2, terminou
- Analogia: labirinto utilizando uma corda

# GRAFO – BUSCA EM PROFUNDIDADE



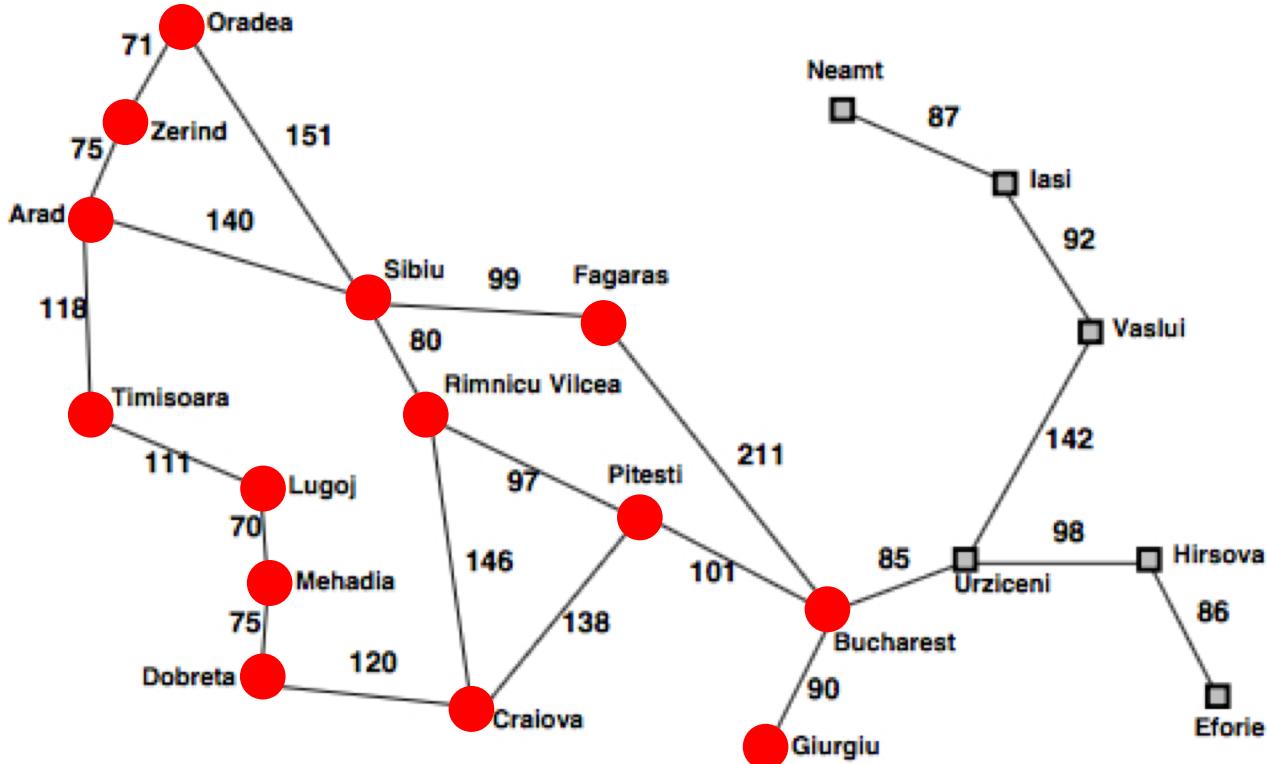
# GRAFO – BUSCA EM LARGURA



# GRAFO – BUSCA EM LARGURA

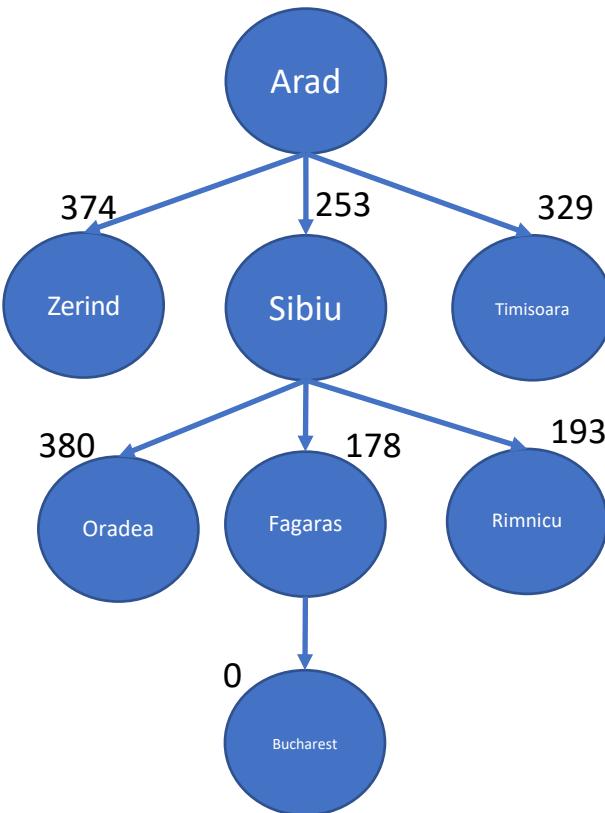
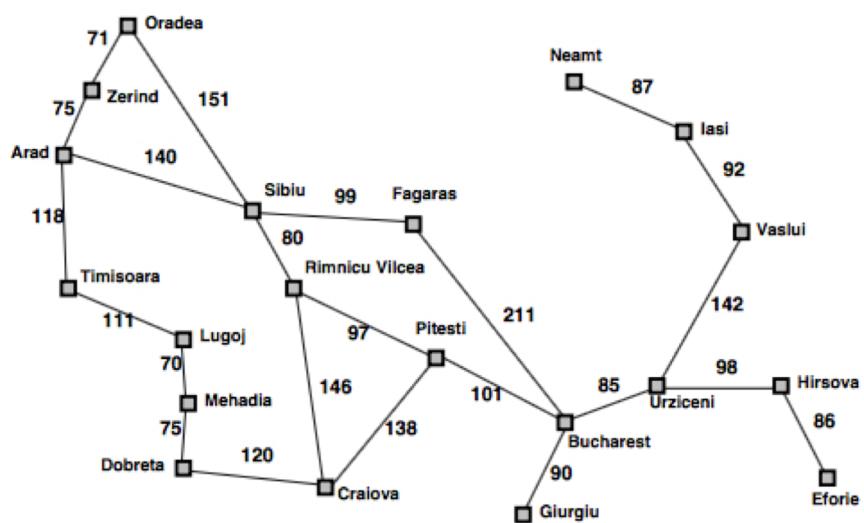
- Regra 1
  - Visite um nó adjacente não visitado, marque-o e coloque-o na fila
- Regra 2
  - Se não puder seguir a Regra 1 remova um nó da fila e torne-o o nó atual
- Regra 3
  - Se não puder executar a Regra 2 porque a fila está vazia, terminou

# GRAFO – BUSCA EM LARGURA



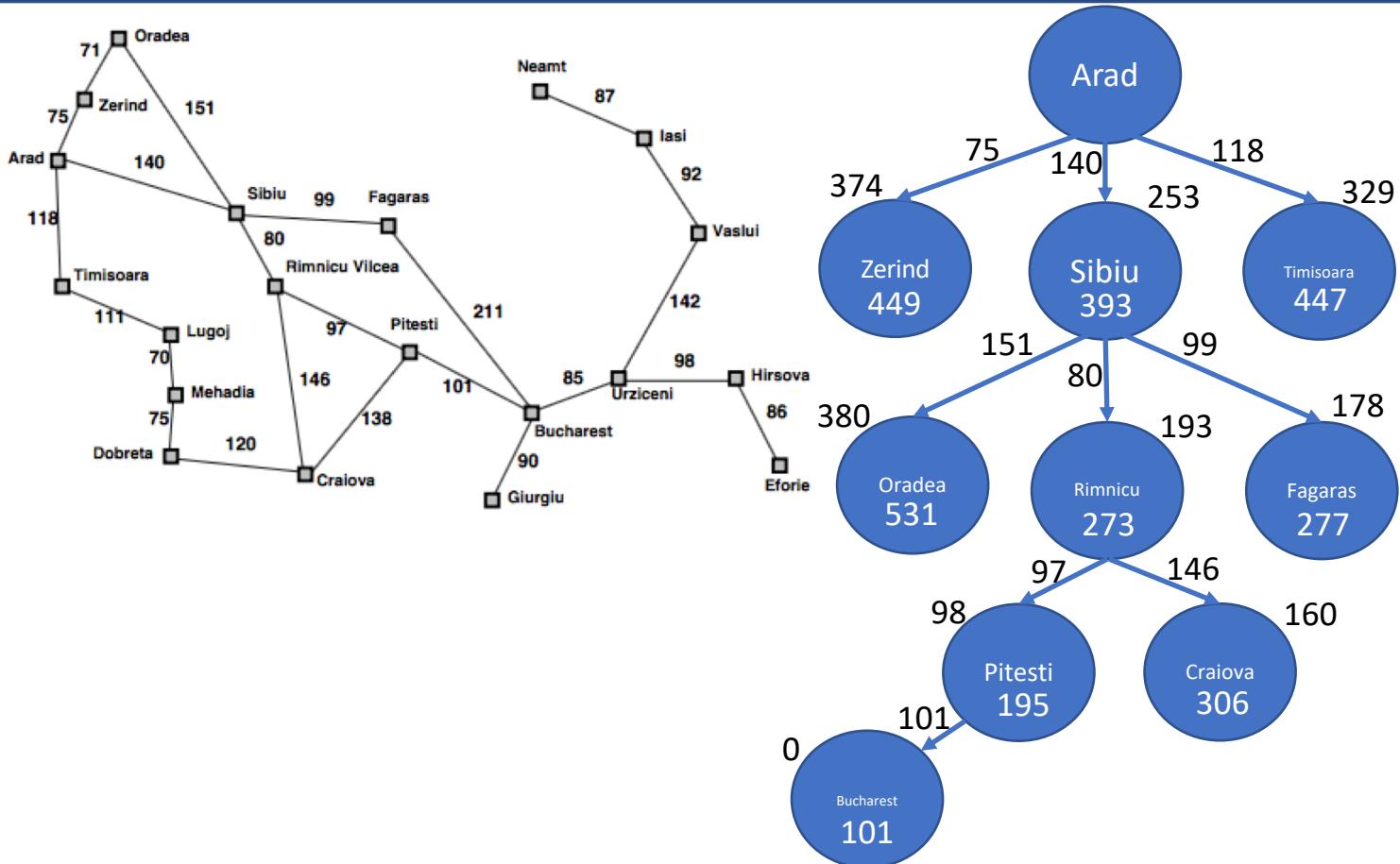
- Arad
  - Timisoara Sibiu Zerind
  - Oradea Timisoara Sibiu
- Rimnicu Fagaras Oradea Timisoara
  - Lugoj Rimnicu Fagaras Oradea
  - Lugoj Rimnicu Fagaras
  - Bucharest Lugoj Rimnicu
- Pitesti Craiova Bucharest Lugoj
- Mehadia Pitesti Craiova Bucharest
  - Giurgiu Mehadia Pitesti Craiova
  - Dobrete Giurgiu Mehadia Pitesti
  - Dobrete Giurgiu Mehadia
  - Dobrete Giurgiu
- Dobreta

# GRAFO – BUSCA GULOSA



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

# GRAFO – BUSCA AESTRELA (A\*)

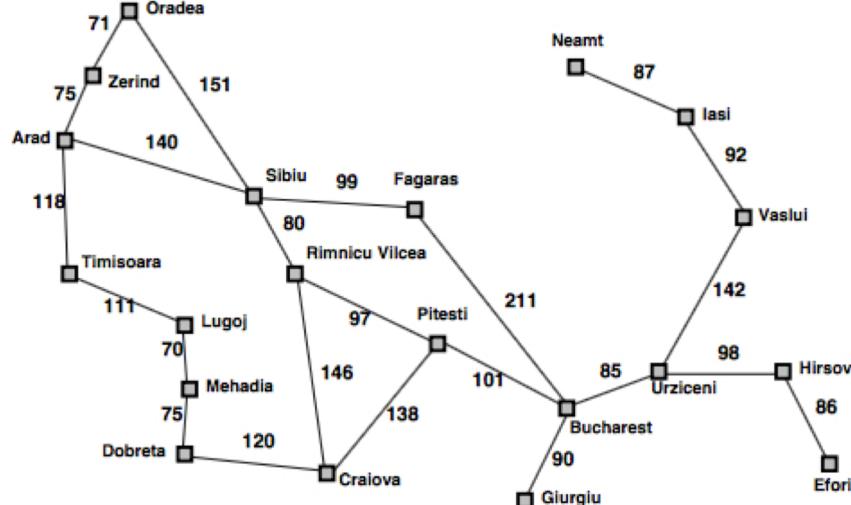


Straight-line distance  
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

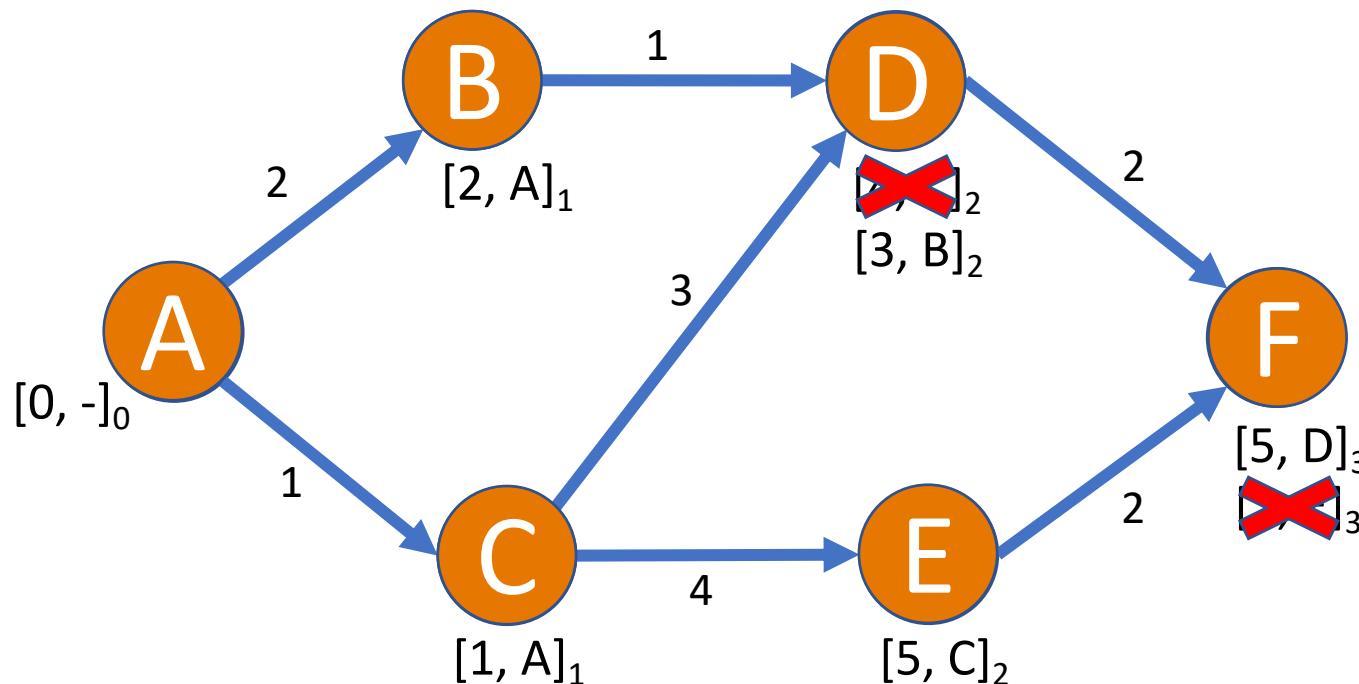
# GRAFO – ALGORITMO DE DIJKSTRA

- Edsger Dijkstra, 1959
- Encontra o caminho mais curto a partir de um nó especificado até todos os outros
- Descobrir a maneira mais barata de viajar de A até todas as outras cidades



# GRAFO – ALGORITMO DE DIJKSTRA

$[8, A]_1$  = acumulado, procedente, vértices



Rota de A até F = A  $\rightarrow$  B  $\rightarrow$  D  $\rightarrow$  F