Estruturas de Dados

Victor Machado da Silva, MSc
silva.victor@ibmec.edu.br

ibmec

Índice

Apresentação do curso

Apresentação do curso

- Contato: silva.victor@ibmec.edu.br
- Material: www.victor0machado.github.io
- Avaliação
  - Proporção
    - AC (20%): atividades em sala e exercícios semanais
    - AP1 (40%): projeto em grupo (30%) + prova subjetiva, sem consulta e individual (70%)
    - AP2 (40%): projeto em grupo
  - Detalhes das entregas
    - Atividades da AC individuais ou em dupla
    - Projetos da AP1 e AP2 em grupos, mínimo 2 e máximo 3 pessoas
  - A AS é uma única prova subjetiva, sem consulta

Algoritmos

Introdução

Um algoritmo é um processo sistemático para a resolução de um problema. O desenvolvimento de algoritmos é particularmente importante para problemas a serem solucionados em um computador, pela própria natureza do instrumento utilizado.

Um algoritmo computa uma saída, o resultado do problema, a partir de uma entrada, as informações inicialmente conhecidas e que permitem encontrar a solução do problema. Durante o processo de computação o algoritmo manipula dados, gerados a partir da sua entrada.

O estudo de estruturas de dados não pode ser desvinculado de seus aspectos algorítmicos. A escolha correta da estrutura adequada a cada caso depende diretamente do conhecimento de algoritmos para manipular a estrutura de maneira eficiente.

Apresentação dos algoritmos

As convenções seguintes serão utilizadas com respeito à linguagem:

- O início e o final de cada bloco são determinados por indentação, isto é, pela posição da margem esquerda. Se uma certa linha do algoritmo inicia um bloco, ele se estende até a última linha seguinte, cuja margem esquerda se localiza mais à direita do que a primeira do bloco;
- A declaração de atribuição é indicada pelo símbolo :=;
- As declarações seguintes são empregadas com significado semelhante ao usual:

```
se... então
se... então... senão
enquanto... faça
para... faça
pare
```

- Variáveis simples, vetores, matrizes e registro são considerados como tradicionalmente em linguagens de programação. Os elementos de vetores e matrizes são identificados por índices entre colchetes.

```
para i := 1, ..., |__n/2__|
    temp := S[i]
    S[i] := S[n - i + 1]
    S[n - i + 1] := temp
```

Aplicações

- Escreva os algoritmos, em pseudocódigo, para os problemas abaixo:
  - https://br.spoj.com/problems/TOMADA13/
  - https://br.spoj.com/problems/METEORO/
  - https://br.spoj.com/problems/JDESAF12/
  - https://br.spoj.com/problems/CARTAS14/
  - https://br.spoj.com/problems/ENCOTEL/

Recursividade

Um tipo especial de procedimento será utilizado, algumas vezes, ao longo do curso. É aquele que contém, em sua descrição, uma ou mais chamadas a si mesmo. Um procedimento dessa natureza é denominado **recursivo**.

Naturalmente, todo procedimento, recursivo ou não, deve possuir pelo menos uma chamada proveniente de um local exterior a ele. Essa chamada é denominada **externa**. Um procedimento não recursivo é, pois, aquele em que todas as chamadas são externas.

O exemplo clássico mais simples de recursividade é o cálculo do fatorial de um inteiro $n \geq 0$:

```
função fat(i)
    fat(i) := se i <= 1 então 1 senão i * fat(i - 1)
```

```
fat[0] := 1
para j := 1, ..., n faça
    fat[j] := j * fat[j - 1]
```

Aplicações

- Escreva os algoritmos, em pseudocódigo, para os problemas abaixo:
  - https://br.spoj.com/problems/F91/
  - https://br.spoj.com/problems/RUMO9S/
  - https://br.spoj.com/problems/PARIDADE/

Recursividade

Um exemplo conhecido, onde a solução recursiva é comum e extremamente mais simples que a solução não-recursiva, é o do Problema da Torre de Hanói.

Recursividade

A solução do problema é descrita a seguir. Para $n > 1$, o pino-trabalho deve ser utilizado como área de armazenamento temporário. O raciocínio utilizado para resolver o problema é semelhante ao de uma prova matemática por indução. Suponha que se saiba como resolver o problema até $n - 1$ discos, $n > 1$, de forma recursiva. A extensão para n discos pode ser obtida pela realização dos seguintes passos:

- Resolver o problema da Torre de Hanói para os $n - 1$ discos do topo do pino-origem A, supondo que o pino-destino seja C e o trabalho seja B;
- Mover o n-ésimo pino (maior de todos) de A para B;
- Resolver o problema da Torre de Hanói para os $n - 1$ discos localizados no pino C, suposto origem, considerando os pinos A e B como trabalho e destino, respectivamente.

```
procedimento hanoi(n, A, B, C)
se n > 0 então
    hanoi(n - 1, A, C, B)
    mover o disco do topo de A para B
    hanoi(n - 1, C, B, A)
```

Complexidade de algoritmos

Introdução

Conforme já mencionado, uma característica muito importante de qualquer algoritmo é o seu tempo de execução. Naturalmente, é possível determiná-lo através de métodos empíricos, isto é, obter o tempo de execução através da execução propriamente dita do algoritmo, considerando-se entradas diversas.

Ao contrário do método empírico, o método analítico visa aferir o tempo de execução de forma independente do computador utilizado, da linguagem e dos compiladores empregados e das condições locais de processamento.

Introdução

As seguintes simplificações serão introduzidas para o modelo proposto:

- Suponha que a quantidade de dados a serem manipulados pelo algoritmo seja suficientemente grande. Somente o comportamento assintótico será avaliado.
- Não serão consideradas constantes aditivas ou multiplicativas na expressão matemática obtida. Isto é, a expressão matemática obtida será válida, a menos de tais constantes.

O processo de execução de um algoritmo pode ser dividido em etapas elementares, denominadas *passos*. Cada passo consiste na execução de um número fixo de operações básicas cujos tempos de execução são considerados constantes.

Cálculo de passos

São considerados passos:

- Operações aritméticas, relacionais e lógicas
- Atribuições
- Acesso a elementos em vetores e matrizes
- Acesso a campos de uma estrutura (struct)
- Obtenção do endereço de uma variável (incluindo declarações de variáveis)
- Alteração/obtenção de conteúdo através de ponteiros
- Retorno de valores
- Instruções de alocação de memória
- Chamada de procedimento, função, método, etc.

Cálculo de passos

```
func main() int {
    x, y, media float64

    x = 1
    y = 2

    media = (x + y) / 2

    return 0
}
```

```
func main() int {
    x := 1
    y := 2

    media := (x + y) / 2

    return 0
}
```

```
func main() int {
    media float64

    media = (1 + 2) / 2

    return 0
}
```

Cálculo de passos

```
func main() int {
    x, y, media float64

    x = 1
    y = 2

    media = (x + y) / 2

    return 0
}
```

9

```
func main() int {
    x := 1
    y := 2

    media := (x + y) / 2

    return 0
}
```

9

```
func main() int {
    media float64

    media = (1 + 2) / 2

    return 0
}
```

5

Cálculo de passos

Nem sempre mudar ou melhorar o código vai causar diferenças de desempenho no algoritmo! Algumas podem simplesmente atrapalhar a legibilidade ou até provocar defeitos no software.

Prática: https://br.spoj.com/problems/JBUSCA12/

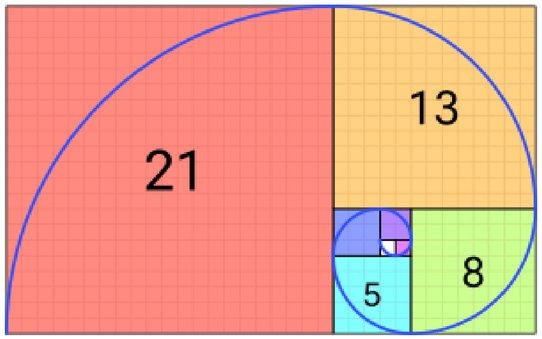
- Submeta um programa ao JBUSCA12 do SPOJ e avalie o tempo de execução
- Traga algumas melhorias para o programa e submeta novamente
- As melhorias reduziram o tempo de execução?

Cálculo de passos

Alguns algoritmos possuem um número variável de passos. Considere, por exemplo, um programa que calcule o n-ésimo número da série de Fibonacci:

```
func fibo(n int) int {
    penultimo, ultimo, proximo, i int

    i = 1
    penultimo = 1
    ultimo = 1

    for i < n {
        proximo = penultimo + ultimo
        penultimo = ultimo
        ultimo = proximo
        i++
    }

    return penultimo
}
```

```
4

3

1...n

6 ——————→ (n - 1) vezes!

1
```



$$8 + n + (n - 1) * 6$$
$$7n + 2$$

Cálculo de passos

Com base nesse exemplo, podemos dizer que o custo de execução de um algoritmo (em termos de números de passos) depende, principalmente, do tamanho da entrada dos dados.

Logo, é comum considerar o tempo de execução de um programa como uma **função do tamanho da entrada**.

```
func soma(v []int, n int) int {
    soma := 0

    for i := 0; i < n; i++ {
        soma += v[i]

    return soma
}
```

```
2

2 / 0...n / 2
            3

1
```

Esses passos são executados n vezes

$$5 + (n + 1) + n * 5$$
$$6n + 6$$

Cálculo de passos

Com base nesse exemplo, podemos dizer que o custo de execução de um algoritmo (em termos de números de passos) depende, principalmente, do tamanho da entrada dos dados.

Logo, é comum considerar o tempo de execução de um programa como uma **função do tamanho da entrada**.

```
func conta() {
    i := 0

    for i < 30 {
        i++
}
```

```
2

0...30
2
```

$$2 + (30 + 1) + 30 * 2$$
$$93$$

Cálculo de passos

Com base nesse exemplo, podemos dizer que o custo de execução de um algoritmo (em termos de números de passos) depende, principalmente, do tamanho da entrada dos dados.

Logo, é comum considerar o tempo de execução de um programa como uma **função do tamanho da entrada**.

```
func busca(v []int, n int, k int) int {
    i := 0

    for i < n {
        if v[i] == k {
            return i
        }
        i++
    }

    return -1
}
```

```
2

0...n
2
1

2

1
```

A quantidade de vezes que o loop vai executar depende:

- Do tamanho do vetor
- Se k existe no vetor
- Da posição de k no vetor

Melhor caso: chave em v[0] → **6**

Pior caso: k não está no vetor →
2 + (n + 1) + n * (2 + 2) + 1
5n + 4

Cálculo de passos

Noção de complexidade:

- Seja A um algoritmo, $\{E_1, \dots, E_m\}$, o conjunto de todas as entradas possíveis de A. Denote por t_i o número de passos efetuados por A, quando a entrada for E_i. Definem-se, com p_i sendo a probabilidade de ocorrência da entrada E_i:
  - Complexidade do pior caso: $\max_{E_{i \in E}}\{t_i\}$;
  - Complexidade do melhor caso: $min_{E_{i \in E}}\{t_i\}$;
  - Complexidade do caso médio: $\sum_{1 \leq i \leq m}(p_i \times t_i)$.
- As complexidades têm por objetivo avaliar a eficiência de tempo ou espaço. A complexidade de tempo de pior caso corresponde ao número de passos que o algoritmo efetua no seu pior caso de execução, isto é, para a entrada mais desfavorável. De certa forma, a complexidade de pior caso é a mais importante das três mencionadas.

Cálculo de passos

Exercício: Dada uma matriz n x n de valores inteiros, implemente uma função que localize um dado valor x. A função deve retornar VERDADEIRO se houver achado, e FALSO caso contrário.

Cálculo de passos

Exercício: Dada uma matriz n x n de valores inteiros, implemente uma função que localize um dado valor x. A função deve retornar VERDADEIRO se houver achado, e FALSO caso contrário.

```
func busca(matriz [][]int, n, x int) bool {
    i, j int
    i = 0

    for i < n {
        j = 0
        for j < n {
            if (matriz[i][j] == x) {
                return true // achou
            }
            j++
        }
        i++
    }
    return false // não achou
}
```

Qual o número de passos no melhor caso? E no pior caso?

Funções de tempo

As funções de tempo dos principais exemplos analisados anteriormente são:

Algoritmo	Função de tempo
Média	$f(n) = 9$
Fibonacci	$f(n) = 7n + 2$
Somatório de vetor	$f(n) = 6n + 6$
Busca em vetor	$f(n) = 5n + 4$
Busca em matriz	$f(n) = 5n^2 + 5n + 5$

Note que temos uma função **constante**, três funções **lineares** e uma função **quadrática**

Funções de tempo

Comparando o número de passos com base no valor n de entrada:

Valor de Entrada	Média de X e Y	N Fibonacci	Soma Vetor	Busca Vetor	Busca Matriz
Função	9	7n + 2	6n + 6	5n + 4	$5n^2 + 5n + 5$
1	9	9	12	9	15
2	9	16	18	14	35
4	9	30	30	24	105
8	9	58	54	44	365
16	9	114	102	84	1365
32	9	226	198	164	5285
64	9	450	390	324	20805
128	9	898	774	644	82565
256	9	1794	1542	1284	328965
512	9	3586	3078	2564	1313285
1024	9	7170	6150	5124	5248005
2048	9	14338	12294	10244	20981765
4096	9	28674	24582	20484	83906565
8192	9	57346	49158	40964	335585285
16384	9	114690	98310	81924	1342259205
32768	9	229378	196614	163844	5368872965
65536	9	458754	393222	327684	21475164165
131072	9	917506	786438	655364	85900001285
262144	9	1835010	1572870	1310724	3,43599E+11

A notação O

Quando se considera o número de passos efetuados por um algoritmo, podem-se desprezar constantes aditivas ou multiplicativas.

Por exemplo, um valor de número de passos igual a 3n será aproximado para n.

Além disso, como o interesse é restrito a valores assintóticos, termos de menor grau também podem ser desprezados. Assim, um valor de número de passos igual a $n^2 + n$ será aproximado para n^2. O valor $6n^3 + 4n - 9$ será transformado em n^3.

Torna-se útil, portanto, descrever operadores matemáticos que sejam capazes de representar situações como essas. A notação O será utilizada com essa finalidade.

A notação O

Sejam f, h funções reais positivas de variável inteira n. Diz-se que f é $O(h)$, escrevendo-se $f = O(h)$, quando existir uma constante $c > 0$ e um valor inteiro n_o, tal que:

$$n > n_o \Rightarrow f(n) \leq c \times h(n)$$

Ou seja, a função h atua como um limite superior para valores assintóticos da função f. Em seguida são apresentados alguns exemplos da notação O.

$$f = n^2 - 1 \Rightarrow f = O(n^2)$$
$$f = n^3 - 1 \Rightarrow f = O(n^3)$$
$$f = 403 \Rightarrow f = O(1)$$
$$f = 5 + 2\log n + 3\log^2 n \Rightarrow f = O(\log^2 n)$$

A notação O

Propriedades da notação O

- f(n) = O(f(n))
- c . O(f(n)) = O(c . f(n)) = O(f(n)) (c = constante)
- O(f(n)) + O(f(n)) = O(f(n))
- O(O(f(n))) = O(f(n))
- f1(n) . O(f2(n)) = O(f1(n) . f2(n))
- O(f1(n)) . O(f2(n)) = O(f1(n) . f2(n))
  - Isto significa que a complexidade de um algoritmo com dois trechos aninhados, em que o segundo é repetidamente executado pelo primeiro, é dada como o produto da complexidade do trecho mais interno pela complexidade do trecho mais externo.
- O(f1(n)) + O(f2(n)) = O(max(f1(n), f2(n)))
  - Isto significa que a complexidade de um algoritmo com dois trechos em sequência com tempos de execução diferentes é dada como a complexidade do trecho de maior complexidade.

Aplicações

Escreva um algoritmo em pseudocódigo e implemente em Go os problemas abaixo:

- Dado um array de números inteiros positivos e um valor alvo, encontre um par de números no array cuja soma seja igual ao valor alvo. Se nenhum par for encontrado, retorne um valor (-1, -1) indicando que nenhum par foi encontrado.
- https://br.spoj.com/problems/POPULAR/
- Dado um array de números inteiros positivos, encontre o comprimento da maior subsequência crescente contígua. Uma subsequência crescente é uma sequência de elementos em que cada elemento subsequente é estritamente maior do que o anterior.

Desafios

Escreva um algoritmo em pseudocódigo e implemente em Go o problema abaixo:

- Dado um array de números inteiros positivos, considerado ordenado crescentemente, e um valor alvo, encontre um par de números no array cuja soma seja igual ao valor alvo. Se nenhum par for encontrado, retorne um valor (-1, -1) indicando que nenhum par foi encontrado. Resolva esse problema com um algoritmo cuja complexidade é O(n).

Listas Lineares

Sobre listas

Dentre as estruturas de dados não primitivas, as listas lineares são as de manipulação mais simples. Iremos discutir seus algoritmos e estruturas de armazenamento.

Uma lista linear agrupa informações referentes a um conjunto de elementos que, de alguma forma, se relacionam entre si. Ela pode se constituir, por exemplo, de informações sobre os funcionários de uma empresa, sobre notas de compras, itens de estoque, notas de alunos, etc.

Uma *lista linear*, ou *tabela*, é então um conjunto de $n \geq 0$ nós L[1], L[2], ..., L[n] tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear.

Operações mais comuns

As operações mais frequentes em listas são a *busca*, a *inclusão* e a *remoção* de um determinado elemento, o que, aliás, ocorre na maioria das estruturas de dados. Tais operações podem ser consideradas como básicas e, por essa razão, é necessário que os algoritmos que as implementem sejam eficientes.

Outras operações também são relevantes, porém não serão estudadas a fundo neste curso:

- Alteração de um elemento da lista;
- Combinação de duas ou mais listas lineares.
- Ordenação dos nós segundo um determinado campo;
- Determinação do primeiro (ou do último) nó da lista;
- etc.

Casos particulares

Casos particulares de listas são de especial interesse:

- Se as inserções e remoções são permitidas apenas nas extremidades da lista, ela recebe o nome de **deque** (uma abreviatura do inglês *double ended queue*);
- Se as inserções e as remoções são realizadas somente em um extremo, a lista é denominada **pilha**;
- A lista é denominada **fila** no caso em que as inserções são realizadas em um extremo e remoções em outro.

Alocação em memória

O tipo de armazenamento de uma lista linear pode ser classificado de acordo com a posição relativa na memória de dois nós consecutivos na lista:

- Quando dois nós consecutivos na lista são alocados contiguamente na memória do computador, a lista é classificada como de **alocação sequencial de memória**;
- Quando dois nós consecutivos não são alocados contiguamente, a lista é classificada como de **alocação encadeada de memória**.

A escolha de um ou outro tipo depende essencialmente das operações que serão executadas sobre a lista, do número de listas envolvidas na operação, bem como de características particulares.

Em Go e na maioria das linguagens, as estruturas básicas de dados (como arrays) são tratadas com alocação sequencial de memória.

Alocação sequencial

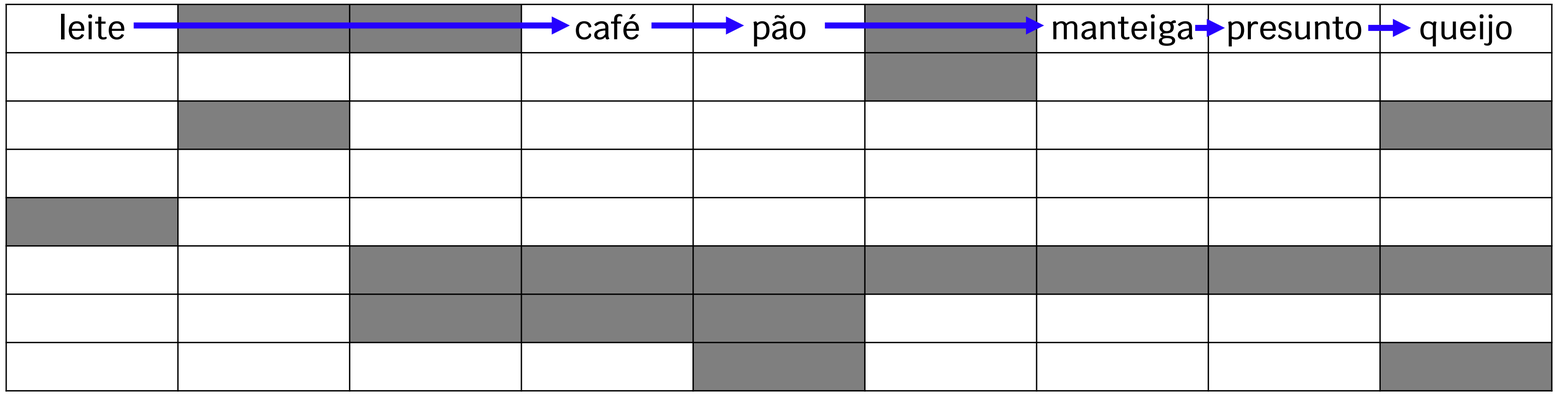
```
listaDeCompras [6]string{“leite”, “café”, “pão”, “manteiga”, “presunto”, “queijo”}
```

		leite	café	pão	manteiga	presunto	queijo	

Vantagens	Desvantagens
▪ Acesso rápido aos elementos ▪ Menos overhead de memória ▪ Operações de leitura e gravação sequenciais tendem a ser mais rápidas	▪ Difícil redimensionamento ▪ Inserir e remover elementos no meio do array podem ser atividades custosas

Alocação encadeada

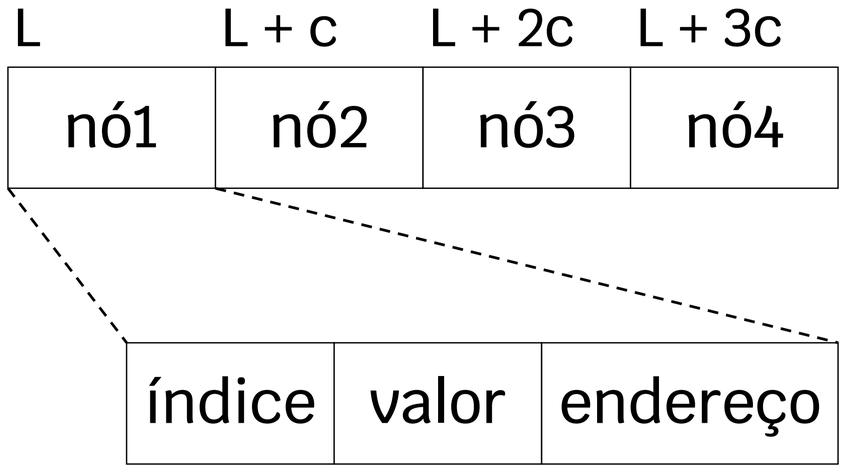
```
listaDeCompras → {“leite”, “café”, “pão”, “manteiga”, “presunto”, “queijo”}
```



Vantagens	Desvantagens
▪ Facilidade de redimensionamento ▪ Inserções e remoções são eficientes, pois envolvem apenas a atualização de ponteiros	▪ Acesso sequencial lento ▪ Maior overhead de memória devido aos ponteiros que conectam os elementos

Listas lineares em alocação sequencial

Seja uma lista linear. Cada nó é formado por campos, que armazenam as características distintas dos elementos da lista. Além disso, cada nó da lista possui, um identificador, denominado **índice**. Os nós podem se encontrar ordenados, ou não, segundo os valores acessados a partir de suas chaves. No primeiro caso a lista é denominada ordenada, e não ordenada no caso contrário.



Busca em uma lista linear

Observe que, para cada elemento da tabela referenciado na busca, o algoritmo realiza dois testes, a operação `i <= n` e a operação `L[i] = x`.

A complexidade de pior caso é $O(n)$.

```
PROGRAMA busca1(v, n, x)
    i := 0
    Enquanto i < n, faça:
        Se v[i] = x então:
            Retorna i
        Senão:
            i := i + 1
    Retorna -1
```

Busca em uma lista linear

É possível otimizar esse algoritmo reduzindo o número de passos necessários e removendo a estrutura condicional, bastando adicionar um elemento ao final da lista com o valor procurado. No entanto, o algoritmo continua com complexidade $O(n)$.

```
PROGRAMA busca2(v, n, x)
    i := 0
    v[n] := x
    Enquanto v[i] != x, faça:
        i := i + 1
    Se i != n, então:
        Retorna i
    Senão:
        Retorna -1
```

Busca em uma lista linear

Quando a lista está ordenada, pode-se tirar proveito desse fato. Se o número procurado não pertence à lista, não há necessidade de percorrê-la até o final.

Apesar da complexidade média reduzir por conta da interrupção da busca, a complexidade de pior caso ainda se mantém em $O(n)$.

```
PROGRAMA buscaOrd(v, n, x)
    i := 0
    v[n] := x
    Enquanto v[i] < x, faça:
        i := i + 1
    Se i = n ou v[i] != x, então:
        Retorna -1
    Senão:
        Retorna i
```

Busca em uma lista linear

Ainda no caso das listas ordenadas, um algoritmo diverso e bem mais eficiente pode ser apresentado: a *busca binária*. Em tabelas, o primeiro nó pesquisado é o que se encontra no meio; se a comparação não é positiva, metade da tabela pode ser abandonada na busca, uma vez que o valor procurado se encontra ou na metade inferior, ou na superior. Esse procedimento, aplicado recursivamente, esgota a tabela.

15?

3	6	10	11	12	19	25	26	31	32	38	39	40	52	64
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**

3	6	10	11	12	19	25	26	31	32	38	39	40	52	64
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**

3	6	10	11	12	19	25	26	31	32	38	39	40	52	64
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**

3	6	10	11	12	19	25	26	31	32	38	39	40	52	64
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**

Busca em uma lista linear

```
PROGRAMA buscaBin(v, n, x)
    inf := 0
    sup := n - 1
    Enquanto inf <= sup, faça:
        meio := parte inteira de (inf + sup) / 2
        Se v[meio] = x então:
            Retorna meio
        Senão se v[meio] < x então:
            inf := meio + 1
        Senão:
            sup := meio - 1
    Retorna -1
```

Busca em uma lista linear

Cálculo da complexidade de pior caso na busca binária:

- Pior caso acontece quando o elemento procurado é o último, ou mesmo quando não é encontrado;
- Primeira iteração: dimensão da tabela $\rightarrow n$
- Segunda iteração: dimensão da tabela $\rightarrow \lfloor n/2 \rfloor$
- Terceira iteração: dimensão da tabela $\rightarrow \lfloor \lfloor n/2 \rfloor /2 \rfloor$
- ...
- mª iteração: dimensão da tabela $\rightarrow 1$

$$\frac{n}{2^{m-1}} = 1 \rightarrow 2^{m-1} = n \rightarrow \log_2 2^{m-1} = \log_2 n \rightarrow (m-1) \times \log_2 2 = \log_2 n \rightarrow m = \log_2 n + 1$$

Portanto, o programa irá rodar o loop um total de $\log_2 n + 1$ vezes. Sendo assim, a complexidade de pior caso para a busca binária em uma lista linear ordenada é $O(\log n)$.

Inserção e remoção em uma lista linear

Ambas as operações de inserção e remoção utilizam o procedimento de busca. No primeiro caso, o objetivo é evitar valores repetidos e, no segundo, a necessidade de localizar o elemento a ser removido.

Os dois algoritmos a seguir consideram tabelas não ordenadas. A memória pressuposta disponível tem M posições. Devem-se levar em conta as hipóteses de se tentar fazer inserções numa lista que já ocupa M posições (situação conhecida como *overflow*), bem como a tentativa de remoção de um elemento de uma lista vazia (*underflow*). A atitude a ser tomada em cada um desses casos depende do problema tratado.

Ambos as operações possuem complexidade $O(n)$, apesar do algoritmo de remoção ser mais lento que o de inserção.

Caso seja assumido que é possível ter valores repetidos na tabela, é possível simplificar o algoritmo de inserção para que seja $O(1)$, já que não é necessário realizar a operação de busca.

Inserção e remoção em uma lista linear

```
PROGRAMA insere(v, n, M, novoValor)
    Se n < M então:
        Se busca(v, n, novoValor) = -1 então:
            v[n] := novoValor
            n++
        Senão:
            Escrever “Elemento já existe na tabela”
    Senão:
        Escrever “Overflow”
```

Inserção e remoção em uma lista linear

```
PROGRAMA remove(v, n, valor)
   Se n != 0 então:
      indice := busca(v, n, valor)
      Se índice != -1 então:
         Para i := índice até n - 2, faça:
            v[i] := v[i + 1]
         n := n -1
      Senão:
         Escrever “Elemento não se encontra na tabela”
   Senão:
      Escrever “Underflow”
```

Pilhas e Filas

Pilhas e Filas

Em geral, o armazenamento sequencial de listas é empregado quando as estruturas, ao longo do tempo, sofrem poucas remoções e inserções. Em casos particulares de listas, esse armazenamento também é empregado. Nesse caso, a situação favorável é aquela em que inserções e remoções não acarretam movimentação de nós, o que ocorre se os elementos a serem inseridos e removidos estão em posições especiais, como a primeira ou a última posição. Deques, pilhas e filas satisfazem tais condições.

Na alocação sequencial de listas genéricas, considera-se sempre a primeira posição da lista no endereço 1 da memória disponível. Uma alternativa a essa estratégia consiste na utilização de ponteiros para o acesso a posições selecionadas.

Pilhas

Pilha de Prato

Se quisermos colocar um item na pilha, colocamos no topo.

Se quisermos retirar um item da pilha, retiramos do topo.

Primeiro que entra é o último a sair.

FILO
(first in, last out)

Pilha de Livro

Pilhas

São estruturas de dados simples de entender e bastante utilizadas na computação

O primeiro a entrar é sempre o último a sair, seguindo a estratégia FILO (*first in, last out*)

É possível de ser implementada através de arrays, utilizando as funções:

- push: insere um item no topo da pilha;
- pop: remove o item do topo da pilha.

Como as inserções e remoções não exigem uma busca pela lista, é possível desenvolver algoritmos que sejam $O(1)$.

Pilhas

Um caso bem comum de uso de pilhas no desenvolvimento de software é no rastreamento do histórico de navegação em páginas web. Quando um *browser* é utilizado, o software mantém um registro das páginas que você visitou. Dessa forma, você consegue retornar à página anterior utilizando o botão “voltar” ou então atalhos no teclado, como Alt + ←.

Para gerenciar esse histórico de navegação, o *browser* utilizar uma estrutura de pilha. Cada vez que você visita uma nova página da web, uma entrada correspondente é colocada no topo da pilha. Isso significa que a página mais recente visitada fica no topo da pilha.

Quando você pressiona o botão “voltar” no navegador, o software desempilha a página superior da pilha, levando-o de volta à ´página que você visitou antes.

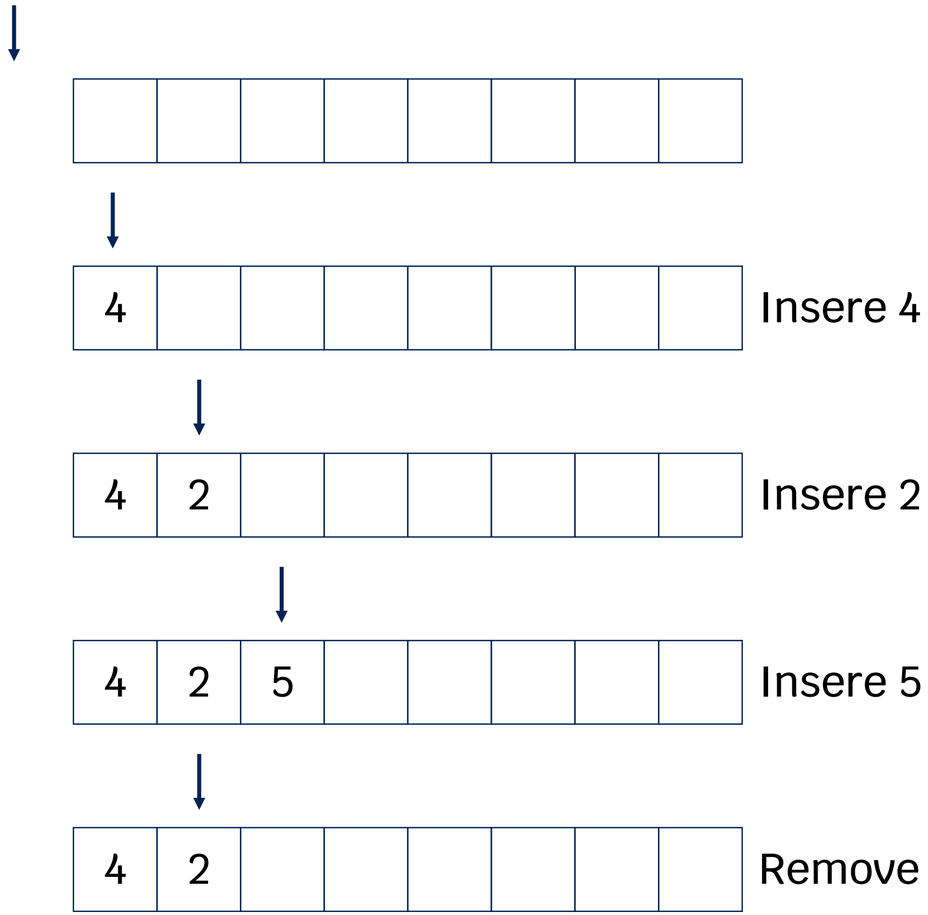
Pilhas

Um outro caso é na pilha de execução de um programa. Uma *stack* de chamadas é uma parte fundamental da memória usada em programas de computador para controlar a execução de funções ou métodos.

Passos de uma *stack* de chamadas:

- Quando um programa executa uma função ou método, ele cria um registro de ativação para essa função, que contém informações importantes, como parâmetros, variáveis e o endereço de retorno;
- À medida que novas funções são chamadas dentro de outras funções, os registros de ativação são empilhados uns sobre os outros na pilha de execução. A função mais recente é colocada no topo da pilha;
- O código da função no topo da pilha é executado (podendo incluir chamadas a outras funções);
- Quando uma função é concluída, seu registro de ativação é removido da parte superior da pilha, e a execução retorna para o ponto onde a função foi chamada.

Pilhas




```
PROGRAMA insere(p, topo, M, valor)
    Se topo != M então:
        topo++
        p[topo] := valor
    Senão:
        Escrever "Overflow"

PROGRAMA remove(p, topo)
    Se topo != -1 então:
        topo--
    Senão:
        Escrever "Underflow"
```

Filas

Se quisermos colocar um item na fila, colocamos no final.

Se quisermos retirar um item da fila, retiramos do início.

Primeiro que sai é o primeiro que entrou.

FIFO
(first in, first out)

Filas

São estruturas de dados também simples de entender

O primeiro a entrar é sempre o primeiro a sair, seguindo a estratégia FIFO (*first in, first out*)

É possível de ser implementada através de arrays, utilizando as funções:

- insert: insere um item no final da fila;
- remove: remove o item do início da fila.

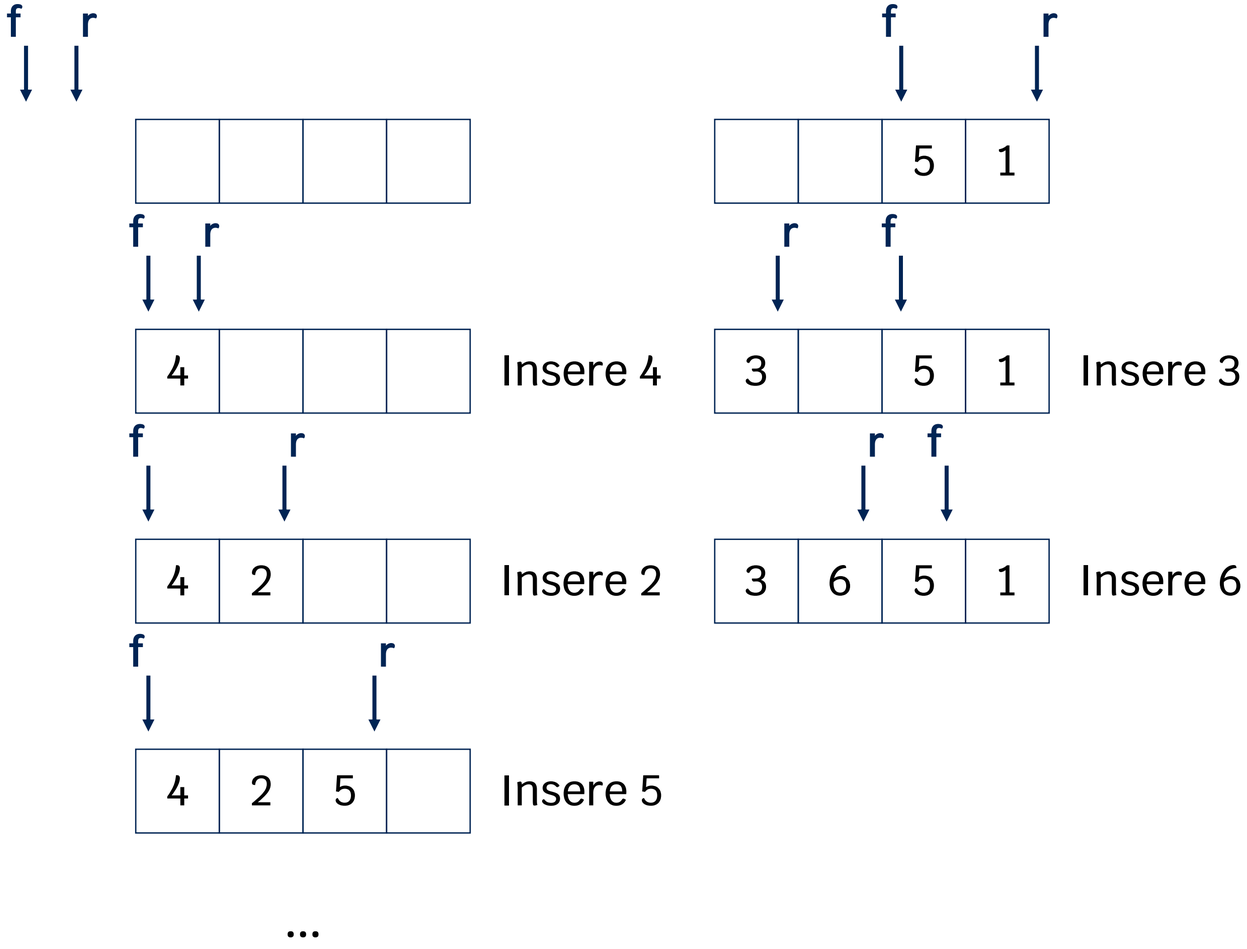
Como as inserções e remoções não exigem uma busca pela lista, é possível desenvolver algoritmos que sejam $O(1)$.

Filas

Dois casos comuns de uso de filas em software:

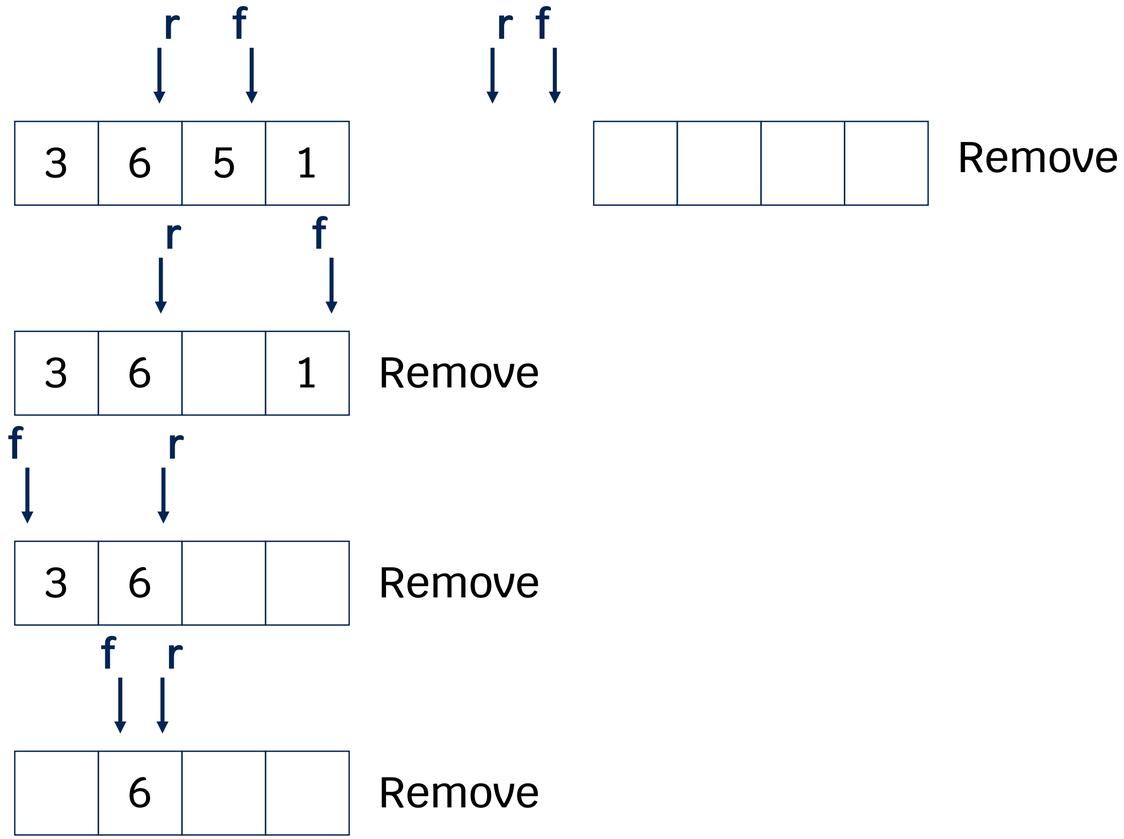
- Gerenciamento de tarefas em sistemas de filas de mensagens (*Message Queues*): As filas são amplamente usadas em sistemas de filas de mensagens para lidar com tarefas assíncronas. Por exemplo, em um sistema de processamento de pedidos online, os pedidos podem ser colocados em uma fila de mensagens para processamento posterior. Isso permite que o sistema processe pedidos de forma assíncrona e em paralelo, melhorando a eficiência e a escalabilidade.
- Fila de espera em aplicações de atendimento ao cliente: Em muitas aplicações de atendimento ao cliente, como centrais de atendimento e suporte online, as solicitações de atendimento ao cliente são colocadas em uma fila de espera. Os agentes de atendimento atendem às solicitações na ordem em que foram recebidas.

Filas




```
PROGRAMA insere(fila, f, r, M, valor)
    aux := (r + 1) mod M
    Se aux != f então:
        r := aux
        fila[r] := valor
        Se f = -1 então:
            f := 0
    Senão:
        Escrever "Overflow"
```

Filas




```
PROGRAMA remove(fila, f, r, M, valor)
    Se f != -1 então:
        Se f = r então:
            f := r := -1
        Senão:
            f := (f + 1) mod M
    Senão:
        Escrever “Underflow”
```

Listas Encadeadas

Alocação sequencial

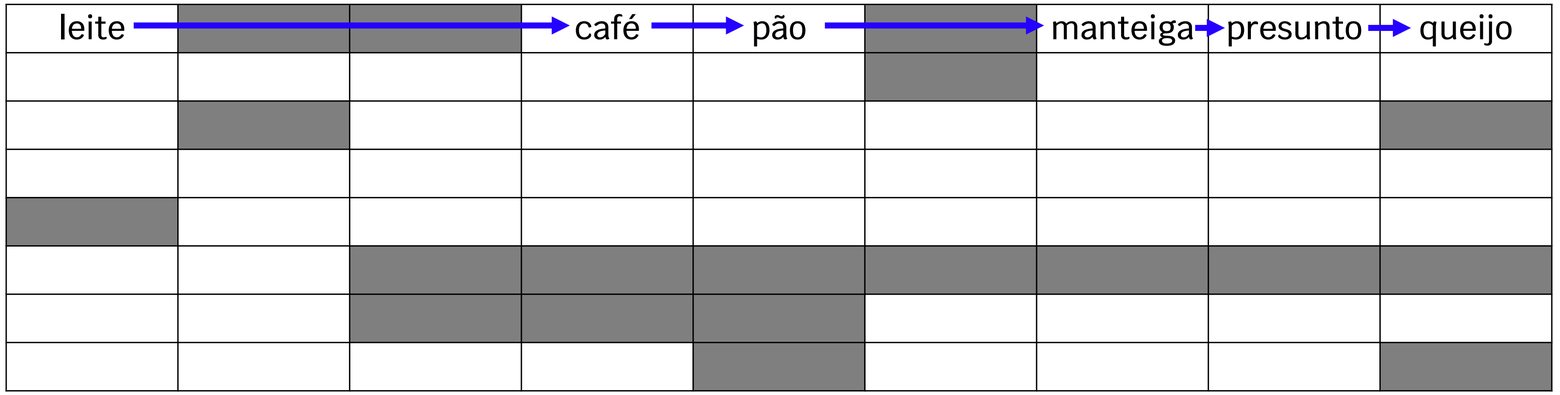
```
listaDeCompras [6]string{“leite”, “café”, “pão”, “manteiga”, “presunto”, “queijo”}
```

		leite	café	pão	manteiga	presunto	queijo	

Vantagens	Desvantagens
▪ Acesso rápido aos elementos ▪ Menos overhead de memória ▪ Operações de leitura e gravação sequenciais tendem a ser mais rápidas	▪ Difícil redimensionamento ▪ Inserir e remover elementos no meio do array podem ser atividades custosas

Alocação encadeada

```
listaDeCompras → {“leite”, “café”, “pão”, “manteiga”, “presunto”, “queijo”}
```



Vantagens	Desvantagens
▪ Facilidade de redimensionamento ▪ Inserções e remoções são eficientes, pois envolvem apenas a atualização de ponteiros	▪ Acesso sequencial lento ▪ Maior overhead de memória devido aos ponteiros que conectam os elementos

Alocação encadeada

O desempenho dos algoritmos que implementam operações realizadas em listas com alocação sequencial, mesmo sendo estes muito simples, pode ser bastante fraco. E mais, quando está prevista a utilização concomitante de mais de duas listas a gerência de memória se torna mais complexa.

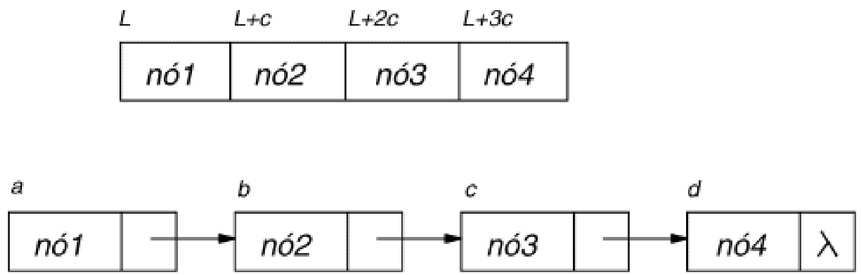
Nesses casos se justifica a utilização da alocação encadeada, também conhecida por alocação dinâmica, uma vez que posições de memórias são alocadas (ou desalocadas) na medida em que são necessárias (ou dispensadas).

Os nós de uma lista encontram-se então aleatoriamente dispostos na memória e são interligados por ponteiros, que indicam a posição do próximo elemento da tabela. É necessário o acréscimo de um campo a cada nó, justamente o que indica o endereço do próximo nó da lista.

Alocação encadeada

Há vantagens e desvantagens associadas a cada tipo de alocação. Estas, entretanto, só podem ser precisamente medidas ao se conhecerem as operações envolvidas na aplicação desejada.

De maneira geral pode-se afirmar que a alocação encadeada, a despeito de um gasto de memória maior em virtude da necessidade de um novo campo no nó (o campo do ponteiro), é mais conveniente quando o problema inclui o tratamento de mais uma lista.



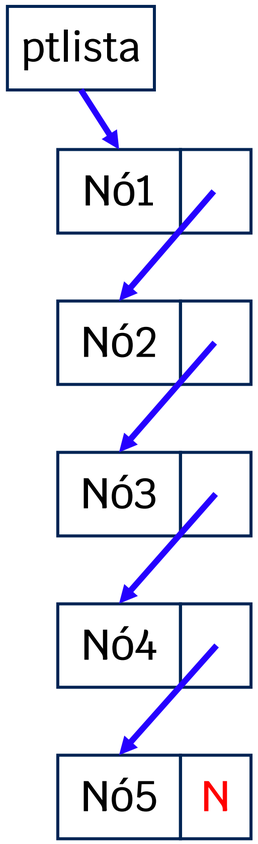
Propriedades

- Uma lista vazia aponta sempre para NULL;
- O último nó da lista sempre aponta para NULL;
- Qualquer nó da lista sempre aponta para o nó seguinte, permitindo um passeio do primeiro ao último;
- Todo nó criado deve sempre apontar para NULL, até que sua posição na lista seja definida.

Operações de listas encadeadas

- Imprimir as informações armazenadas na lista;
- Inserir um nó na lista;
- Procurar um nó na lista;
- Identificar o tamanho da lista;
- Remover um nó da lista;
- Liberar a lista (remover todos os nós da lista).

Operações de listas encadeadas



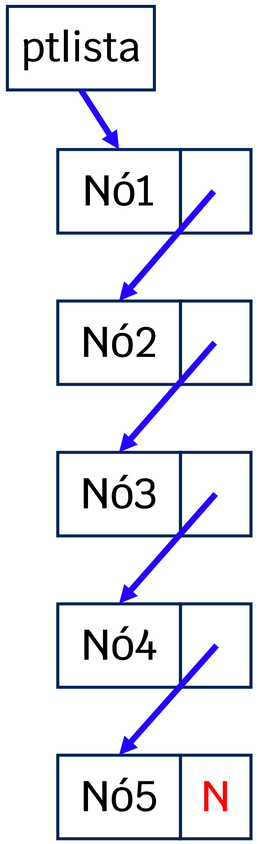
Exibe as informações dos nós

```
PROGRAMA exibe(ptlista)
   pont := ptlista
   Enquanto pont != NULL, faça:
      Escrever pont*.info  // Dados do nó
      pont := pont*.prox
```

Procura um nó com uma determinada operação

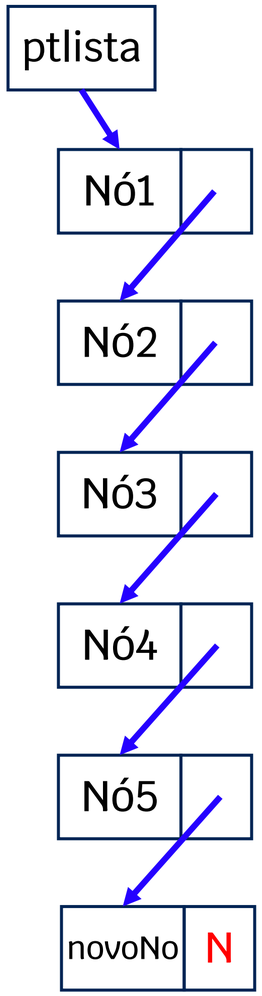
```
PROGRAMA buscaSimples(ptlista, chaveBuscada)
   pont := ptlista
   Enquanto pont != NULL, faça:
      Se pont*.chave = chaveBuscada, então:
         Retorna pont
      pont := pont*.prox
   Retorna NULL
```

Operações de listas encadeadas

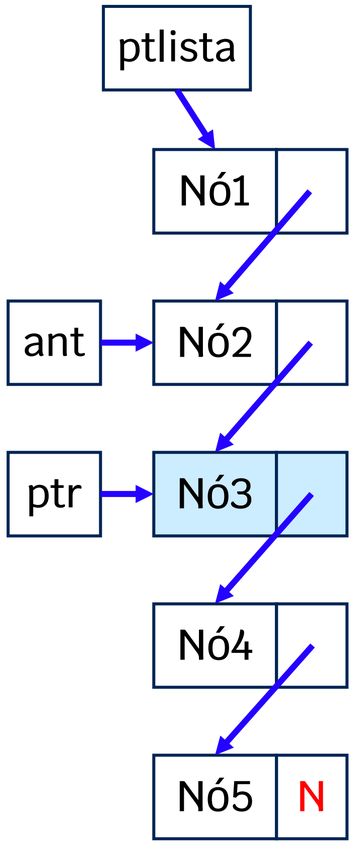


Insere um nó ao final da lista

```
PROGRAMA insere(ptlista, novaChave)
   Cria novoNo com novaChave

   Se ptlista = NULL, então:
      ptlista := novoNo
   Senão:
      pont := ptlista
      Enquanto pont*.prox != NULL, faça:
         pont := pont*.prox
      pont*.prox := novoNo
```



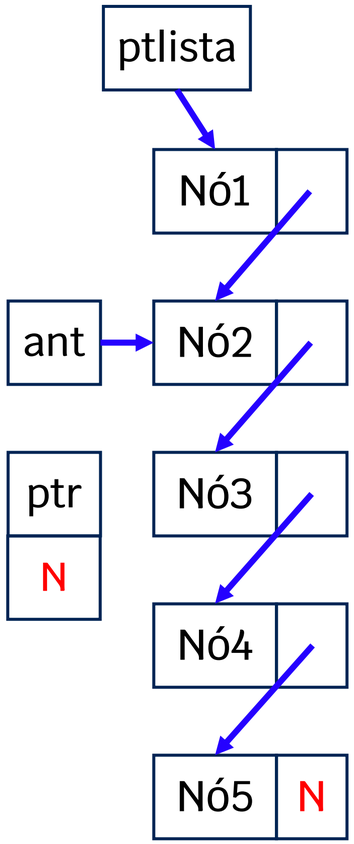
Operações de listas encadeadas



Busca um nó em uma lista ordenada pelo campo `chave`

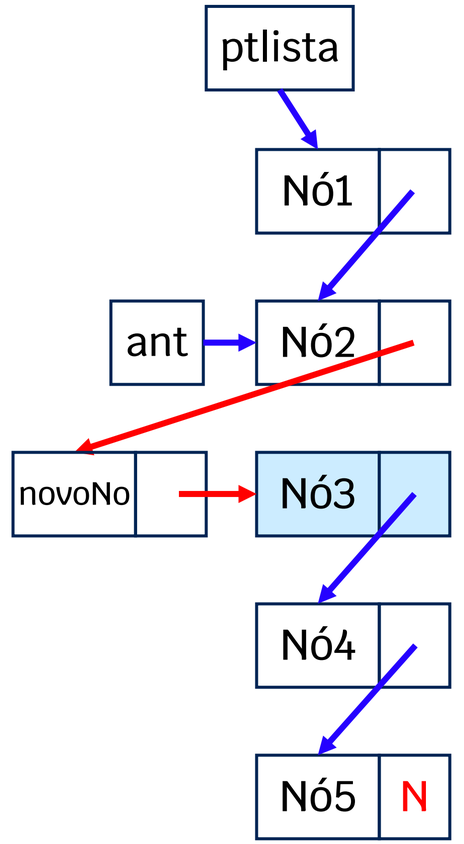
```
PROGRAMA buscaOrd(ptlista, chaveBuscada)
   ant := NULL
   ptr := ptlista

   Enquanto ptr != NULL, faça:
      Se ptr.chave = chaveBuscada, então:
         retorna ant, ptr
      Se ptr.chave > chaveBuscada, então:
         retorna ant, NULL

      ant := ptr
      ptr := ptr*.prox

   Retorna ant, NULL
```

Operações de listas encadeadas

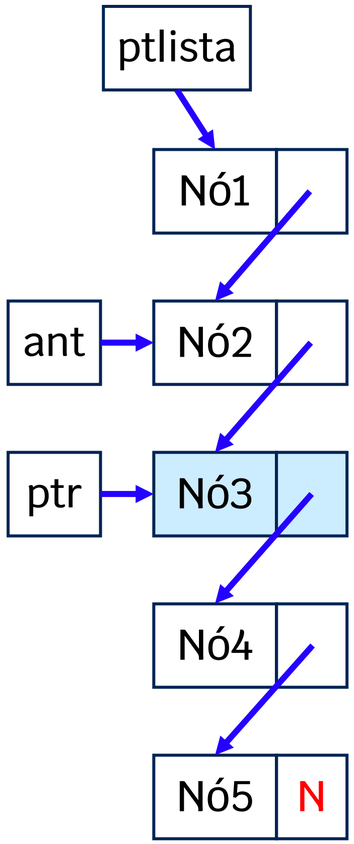


Insere um nó em uma lista ordenada pelo campo `chave`

```
PROGRAMA insereOrd(ptlista, novaChave)
   ant, ptr := buscaOrd(ptlista, novaChave)

   Se ptr != NULL, então encerra o programa

   Cria novoNo com novaChave

   Se ant = NULL, então:
      novoNo*.prox := ptlista
      ptlista := novoNo
   Senão:
      novoNo*.prox := ant*.prox
      ant*.prox := novoNo
```

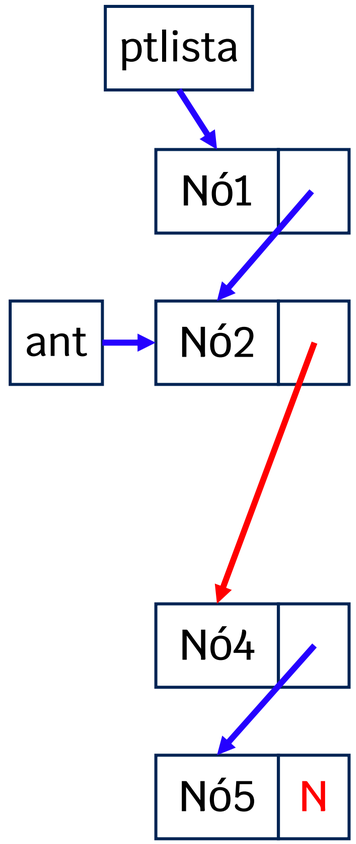


Operações de listas encadeadas



Remove um nó em uma lista ordenada pelo campo `chave`

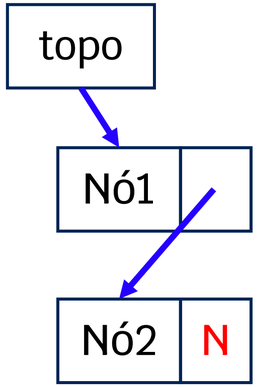
```
PROGRAMA removeOrd(ptlista, chaveBuscada)
   ant, ptr := buscaOrd(ptlista, chaveBuscada)

   Se ptr = NULL, então retorna NULL

   Se ant = NULL, então:
      ptlista := ptr*.prox
   Senão:
      ant*.prox := ptr*.prox

   Retorna ptr
```



Pilhas e Filas encadeadas

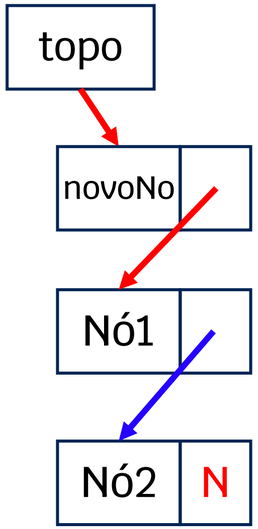
Como casos particulares, algumas modificações são necessárias para implementar operações eficientes em pilhas e filas. No caso de pilhas, as operações são muito simples. Considerando-se listas simplesmente encadeadas, o topo da pilha é o primeiro nó da lista, apontado por uma variável ponteiro $topo$. Se a pilha estiver vazia então $topo = \text{NULL}$. Filas exigem duas variáveis do tipo ponteiro: $inicio$, que aponta para o primeiro nó da lista, e fim, que aponta para o último. Na fila vazia, ambos apontam para NULL. Os algoritmos que se seguem implementam essas operações.

Operações de pilhas encadeadas



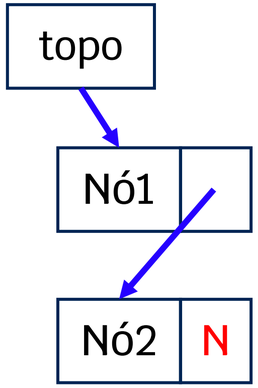
Insere um elemento no topo da pilha

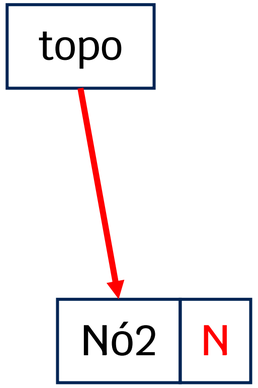
```
PROGRAMA insere(pilha, novaChave)
   Cria novoNo com novaChave

   novoNo*.prox := pilha*.topo
   pilha*.topo := novoNo
```



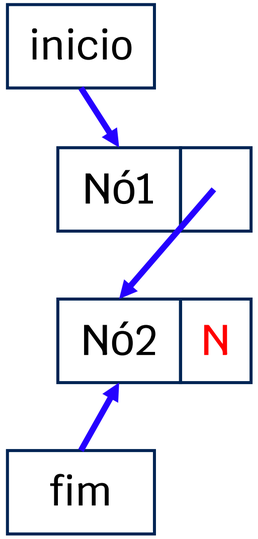
Remove um elemento no topo da pilha

```
PROGRAMA remove(pilha)
   Se pilha*.topo = NULL então retorna NULL

   noRemovido := pilha*.topo
   pilha*.topo := noRemovido*.prox

   Retorna noRemovido
```



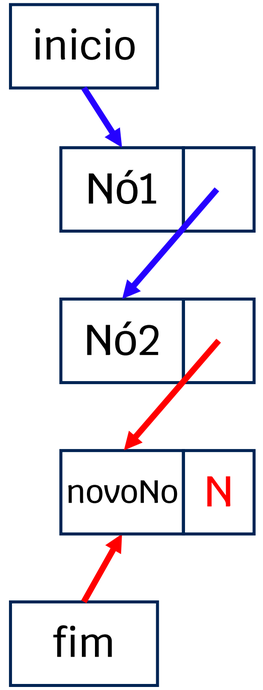


Operações de filas encadeadas

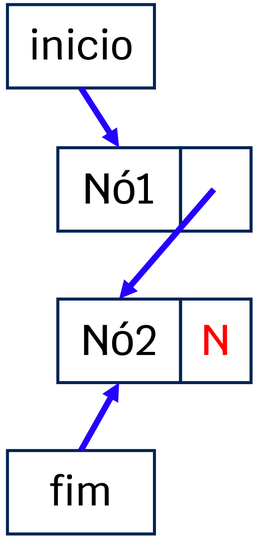


Insere um elemento no final da fila

```
PROGRAMA insere(fila, novaChave)
   Cria novoNo com novaChave

   Se fila*.fim = NULL, então:
      fila*.inicio = novoNo
      fila*.fim = novoNo
   Senão:
      fila*.fim*.prox = novoNo
      fila*.fim = novoNo
```

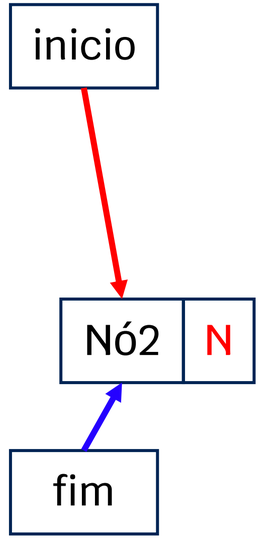


Operações de filas encadeadas

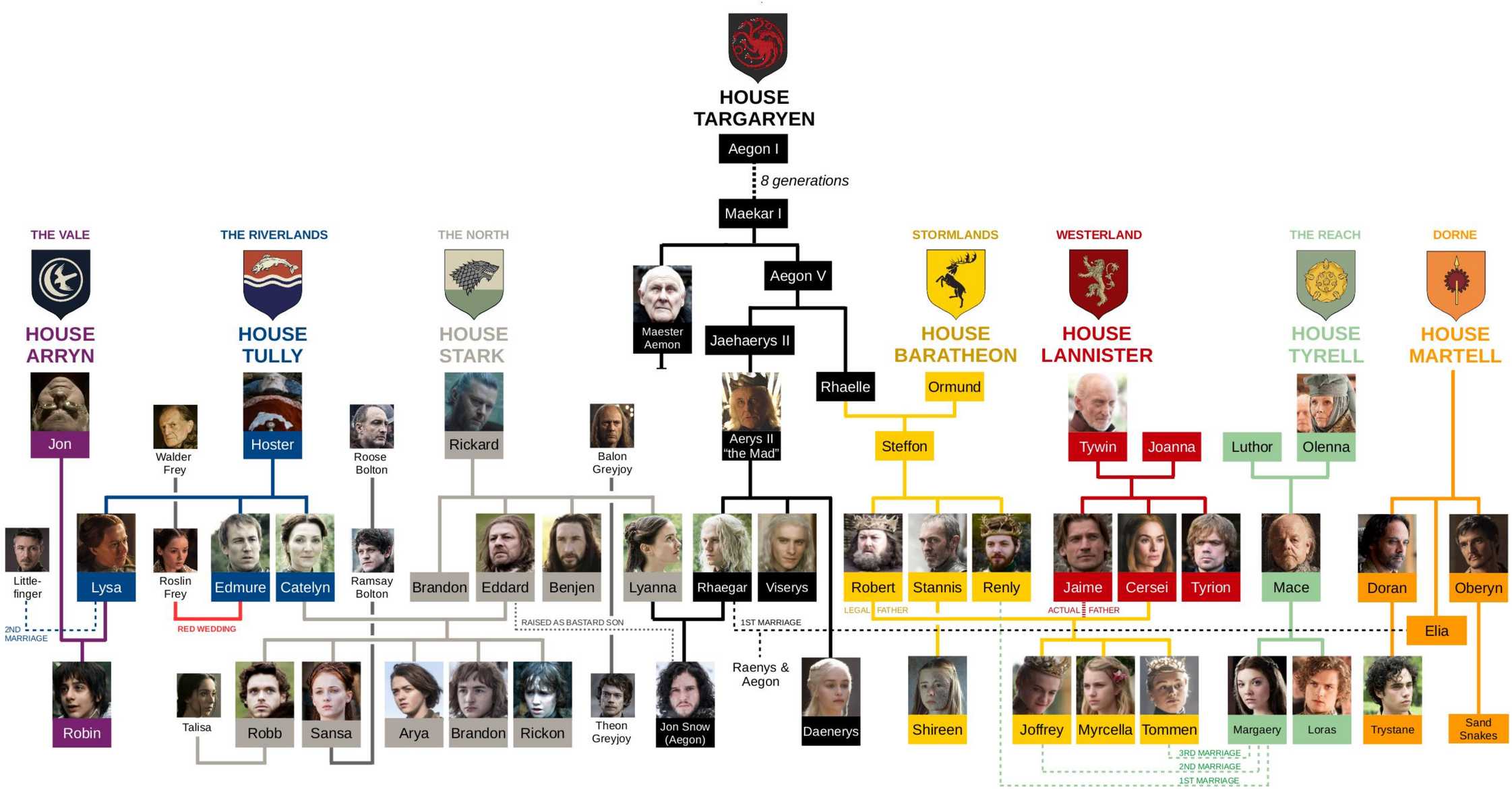


Remove um elemento no inicio da fila

```
PROGRAMA remove(fila)
   Se fila*.inicio = NULL, então retorna NULL

   noRemovido := fila*.inicio
   fila*.inicio := fila*.inicio*.prox

   Se fila*.inicio = NULL, então:
      fila*.fim = NULL

   Retorna noRemovido
```



Árvores


HOUSE TARGARYEN
Aegon I
8 generations
Maekar I
Maester Aemon
Aegon V
Jaehaerys II
Aerys II "the Mad"
Rhaelle
THE VALE
HOUSE ARRYN
Jon
Little-finger
2ND MARRIAGE
Robin
THE RIVERLANDS
HOUSE TULLY
Walder Frey
Hoster
Roose Bolton
Lysa
Roslin Frey
Edmure
Catelyn
Ramsay Bolton
RED WEDDING
Talisa
Robb
Sansa
THE NORTH
HOUSE STARK
Rickard
Balon Greyjoy
Brandon
Eddard
Benjen
Lyanna
RAISED AS BASTARD SON
Arya
Brandon
Rickon
Theon Greyjoy
Jon Snow (Aegon)
Rhaegar
Viserys
1ST MARRIAGE
Raenys & Aegon
Daenerys
STORMLANDS
HOUSE BARATHEON
Ormund
Steffon
Robert
Stannis
Renly
LEGAL FATHER
Shireen
WESTERLAND
HOUSE LANNISTER
Tywin
Joanna
Jaime
Cersei
Tyrion
ACTUAL FATHER
Joffrey
Myrcella
Tommen
3RD MARRIAGE
2ND MARRIAGE
1ST MARRIAGE
THE REACH
HOUSE TYRELL
Luthor
Olenna
Mace
Margaery
Loras
DORNE
HOUSE MARTELL
Doran
Oberyn
Elia
Trystane
Sand Snakes

Introdução

Em diversas aplicações necessita-se de estruturas mais complexas do que as puramente sequenciais, examinadas nas aulas anteriores. Entre essas, destacam-se as árvores, por existirem inúmeros problemas práticos que podem ser modelados através delas. Além disso, as árvores, em geral, admitem um tratamento computacional simples e eficiente. Isso não pode ser dito de estruturas mais gerais do que as árvores, como os grafos, por exemplo.

Nesta aula são apresentados os conceitos iniciais relativos às árvores, bem como os algoritmos para sua manipulação computacional básica.

Aplicações

Árvores possuem larga aplicação na área de computação:

- As estruturas de diretórios e arquivos em um sistema são organizadas em árvores, já que um diretório pode conter múltiplos arquivos e subdiretórios;
- As redes sociais trabalham com o conceito de árvores, ao representar as conexões de amizades e utilizar em sistemas de recomendação;
- Um dos principais modelos de aprendizado de máquina utiliza o conceito de árvores de decisão, em que uma determinada pergunta pode gerar múltiplos desdobramentos;
- Podem ser utilizadas para aumentar a eficiência de algoritmos em que normalmente listas são usadas, como algoritmos de ordenação (Heapsort);
- Motores de busca normalmente usam árvores de sufixos, para realizar pesquisa de padrões e processamento de texto;
- Estruturas complexas pré-implementadas nas linguagens como dicionários e mapas costumam ser desenvolvidas como mapas.

Definições e representações básicas

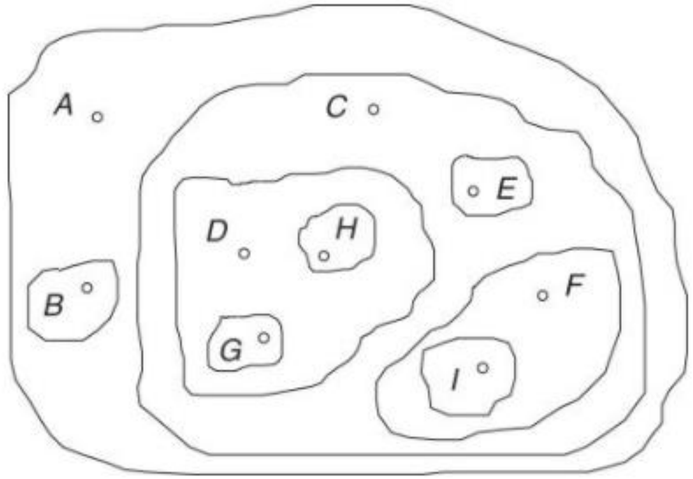
Uma **árvore enraizada T**, ou simplesmente **árvore**, é um conjunto finito de elementos denominados *nós* ou *vértices* tais que:

- $T = \emptyset$, e a árvore é dita *vazia*; ou
- Existe um nó especial chamado *raiz* de $T\big(r(T)\big)$; os restantes constituem um único conjunto vazio ou são divididos em $m \geq 1$ conjuntos disjuntos não vazios, as subárvores de $r(T)$, ou simplesmente **subárvores**, cada qual, por sua vez, uma árvore.

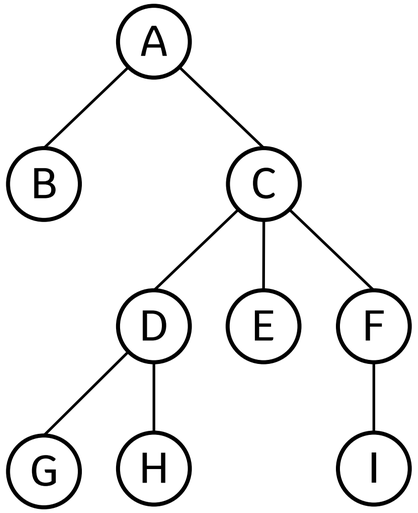
Uma floresta é um conjunto de árvores. Se v é um nó de T, a notação $T(v)$ indica a subárvore de T com raiz v.

Definições e representações básicas

Conjunto



Hierárquica



A árvore ao lado ainda pode ser representada na forma de parênteses aninhados, como abaixo:

(A (B) (C (D (G) (H)) (E) (F (I))))

Sabendo disso, quais seriam as representações de conjunto e hierárquica para as árvores abaixo?

(A (B (D) (F)) (C (G)) (E))

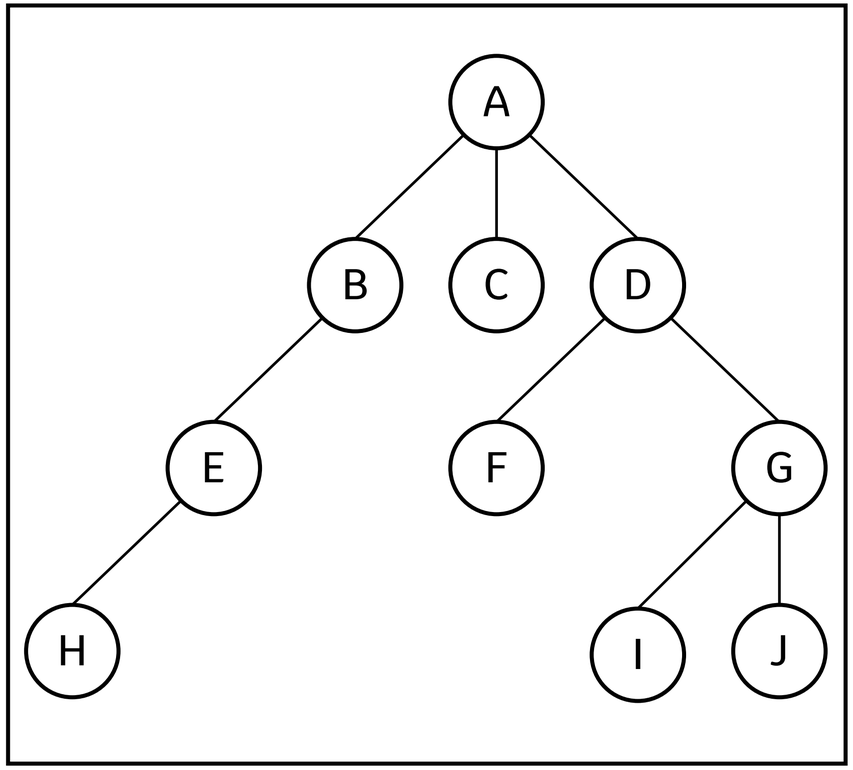
(A (B (C) (D) (E (F) (G)) (H) (I (J) (K (L)))))

Definições e representações básicas

Seja v o nó raiz da subárvore $T(v)$ de T. Os nós raízes $w_1, w_2, \ldots, w_j$ das subárvores de $T(v)$ são chamados **filhos** de v; v é chamado pai de $w_1, w_2, \ldots, w_j$. Os nós $w_1, w_2, \ldots, w_j$ são **irmãos**. Se z é filho de w_1, então w_2 é **tio** de z e v é **avô** de z. O número de filhos de um nó é chamado de **grau de saída** desse nó. Se x pertence à subárvore $T(v)$, x é **descendente** de v, e v é **ancestral** de x. Nesse caso, sendo x diferente de v, x é **descendente próprio** de v, e v é **ancestral próprio** de x.

Um nó que não possui descendentes próprios é chamado de **folha**. Toda árvore com $n > 1$ nós possui no mínimo 1 e no máximo $n - 1$ folhas. Um nó não folha é dito **interior**.

Definições e representações básicas



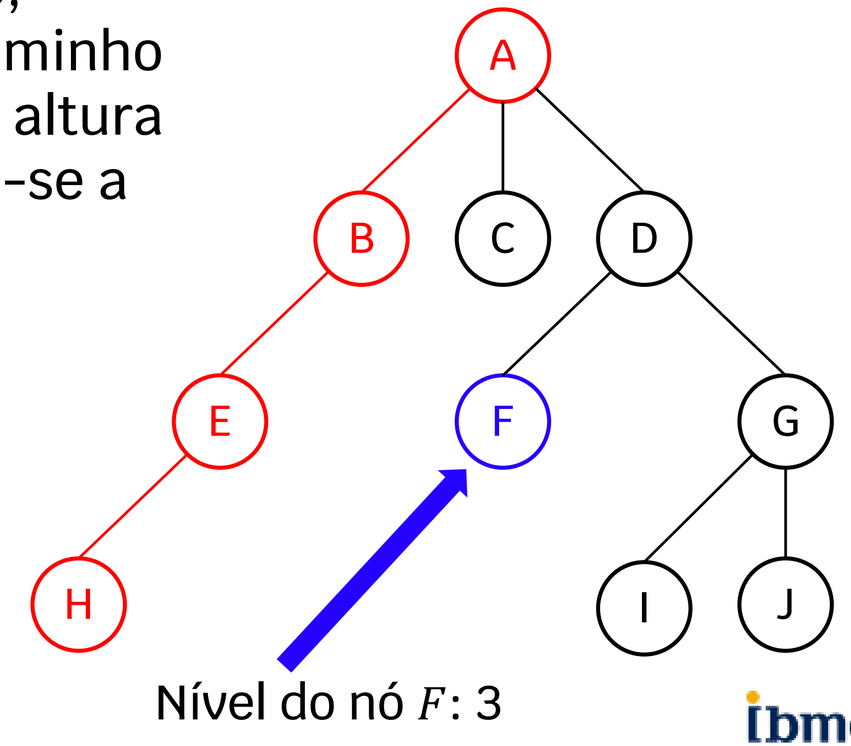
Árvore T

- A é o **nó raiz** da árvore T;
- B, C e D são **filhos** de A;
- B, C e D são **irmãos** entre si;
- A é **avô** de E, F e G;
- B é **tio** de F e G;
- A estrutura formada por $B - E - H$ é uma **subárvore** de T;
- A, B, C,..., I, J são **descendentes** de A, e A é **ancestral** de A, B, C,..., I, J;
- A é **ancestral próprio** de B, C,..., I, J, e estes são **descendentes próprios** de A;
- C, F, H, I e J são **folhas** da árvore T;
- A, B, D, E, G são **nós interiores** da árvore T.

Definições e representações básicas

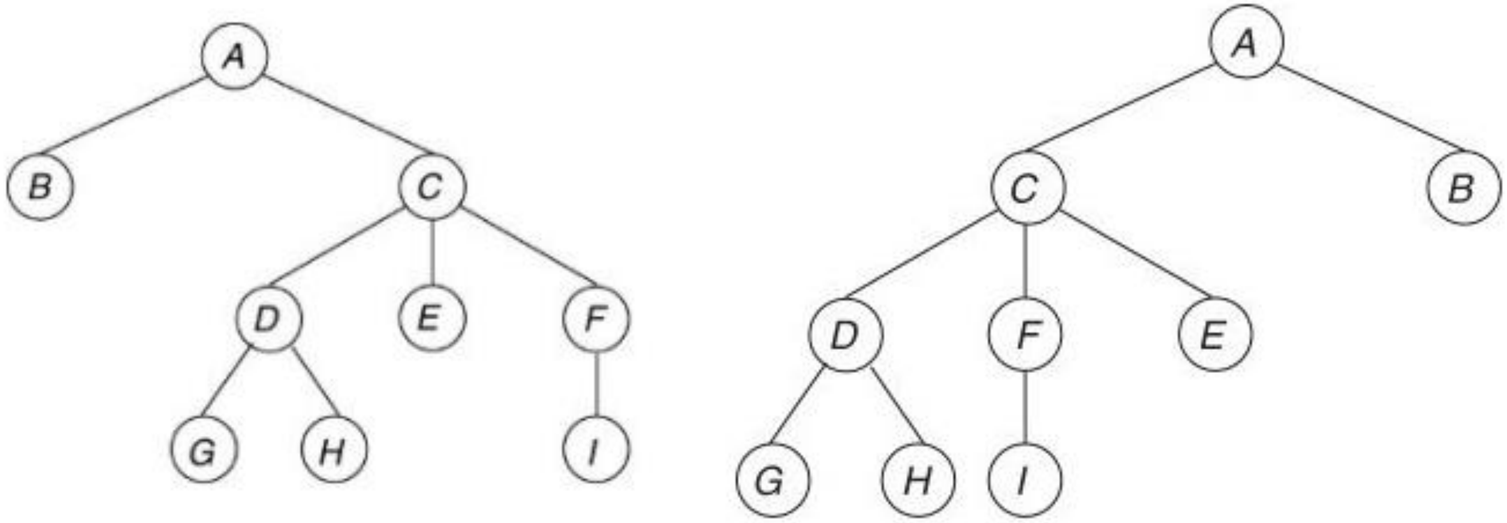
Uma sequência de nós distintos $v_1, v_2, \ldots, v_k$, tal que existe sempre entre nós consecutivos (v_1 e v_2, v_2 e v_3, ..., v_{k-1} e v_k) a relação "é filho de" ou "é pai de", é denominada **caminho da árvore**. Diz-se que v_1 alcança v_k e vice-versa. Um caminho de k nós é obtido pela sequência de $k-1$ pares da relação. O valor $k-1$ é o comprimento do caminho. **Nível de um nó** v é o número de nós do caminho da raiz até o nó v. O nível da raiz é, portanto, igual a 1. A **altura** de um nó v é o número de nós do maior caminho de v até um de seus descendentes. As folhas têm altura 1. A altura da árvore T é igual ao nível máximo de seus nós. Representa-se a altura de T por $h(T)$.

No exemplo ao lado, a altura da árvore é $h(T) = 4$.



Definições e representações básicas

Uma **árvore ordenada** é aquela na qual os filhos de cada nó estão ordenados. Assume-se que tal ordenação se desenvolva da esquerda para a direita. Assim, as árvores da figura abaixo são distintas se consideradas como ordenadas. Contudo, elas podem se tornar coincidentes mediante uma reordenação de nós irmãos.



Árvores binárias

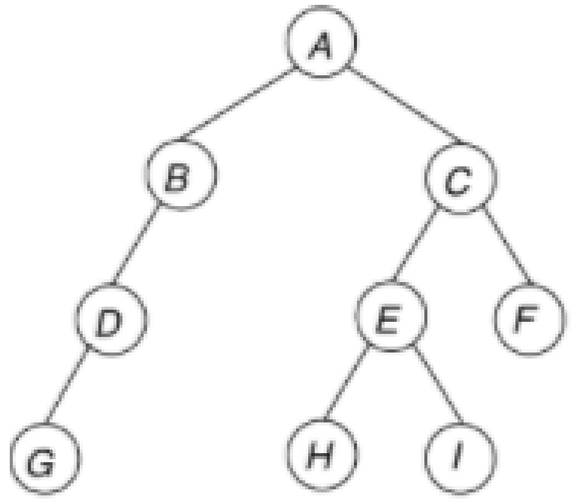
Conforme já mencionado, as árvores constituem as estruturas não sequenciais com maior aplicação em computação. Dentre as árvores, as binárias são, sem dúvida, as mais comuns.

Uma **árvore binária** T é um conjunto finito de elementos denominados nós ou vértices, tal que:

- $T = \emptyset$, e a árvore é dita *vazia*; ou
- Existe um nó especial chamado *raiz* de $T\big(r(T)\big)$; e os restantes podem ser divididos em dois subconjuntos disjuntos, $T_E\big(r(T)\big)$ e $T_D\big(r(T)\big)$, a subárvore esquerda e a direita da raiz, respectivamente, as quais são também árvores binárias.

Árvores binárias

A raiz da subárvore esquerda (direita) de um nó v, se existir, é denominada **filho esquerdo (direito)** de v. Naturalmente, o esquerdo pode existir sem o direito e vice-versa. Analogamente à seção anterior, a notação $T(v)$ indica a (sub) árvore binária, cuja raiz é v e cujas subárvores esquerda e direita de T são $T_E(v)$ e $T_D(v)$, respectivamente.



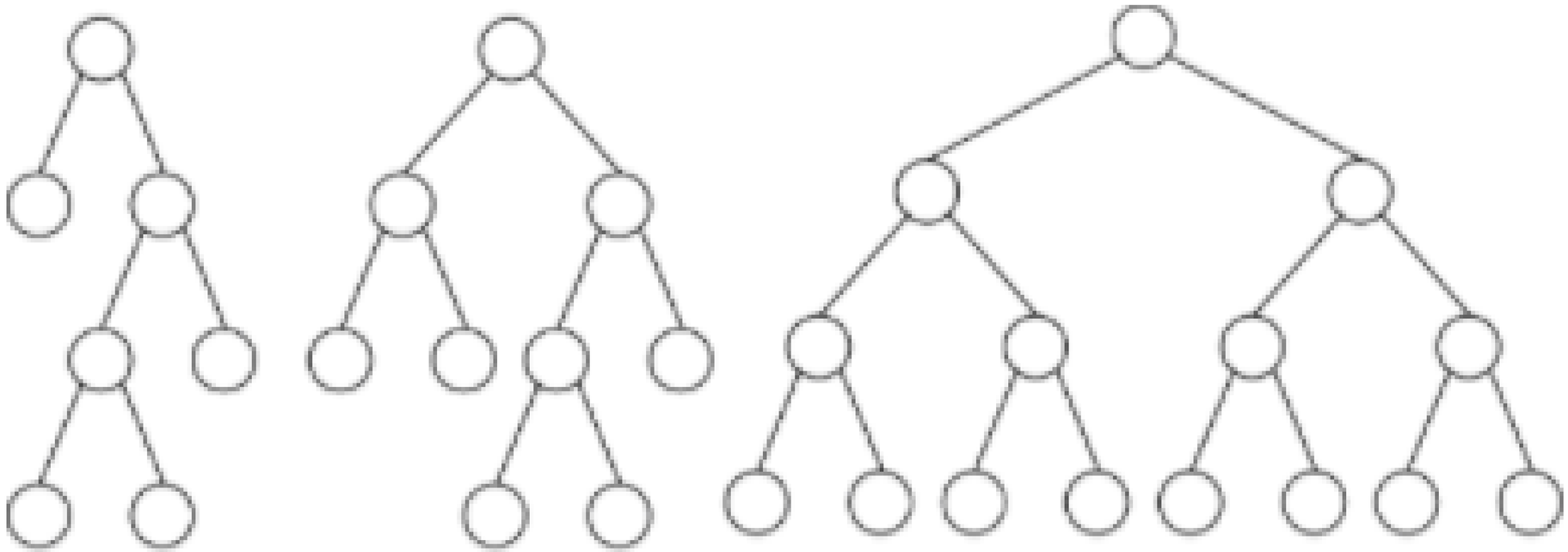
Árvores binárias

Toda árvore binária com n nós possui exatamente $n + 1$ subárvores vazias entre suas subárvores esquerdas e direitas. Por exemplo, a árvore da figura anterior possui 9 nós e 10 subárvores vazias: as subárvores esquerda e direita dos nós F, G, H, I e as subárvores direitas de B e D.

Em seguida, são introduzidos alguns tipos especiais de árvores binárias:

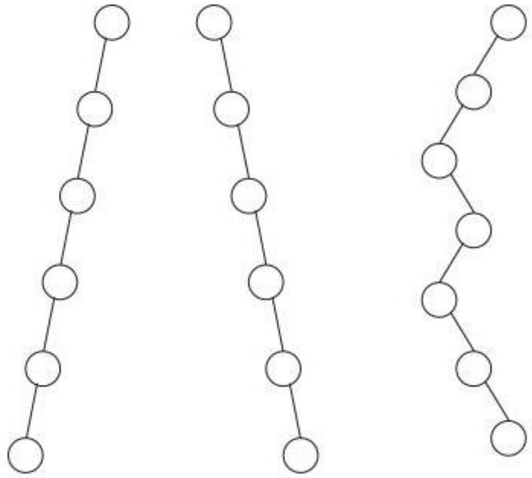
- Uma *árvore estritamente binária* é uma árvore binária em que cada nó possui 0 ou 2 filhos;
- Uma *árvore binária completa* é aquela que apresenta a seguinte propriedade: se v é um nó tal que alguma subárvore de v é vazia, então v se localiza ou no último (maior) ou no penúltimo nível da árvore;
- Uma *árvore binária cheia* é aquela em que, se v é um nó com alguma de suas subárvores vazias, então v se localiza no último nível. Segue-se que toda árvore binária cheia é completa e estritamente binária.

Árvores binárias



Árvores binárias

A relação entre a altura de uma árvore binária e o seu número de nós é um dado importante para várias aplicações. Para um valor fixo de n, indagar-se-ia quais são as árvore binárias que possuem altura h máxima e mínima. A resposta ao primeiro problema é imediata. A árvore binária que possui altura máxima é aquela cujos nós interiores possuem exatamente uma subárvore vazia. Essas árvores são denominadas **zigue-zague**. Naturalmente, a altura de uma árvore zigue-zague é igual a n.



Altura de árvores binárias completas

Seja T uma árvore binária completa com $n > 0$ nós. Então T possui altura h mínima.

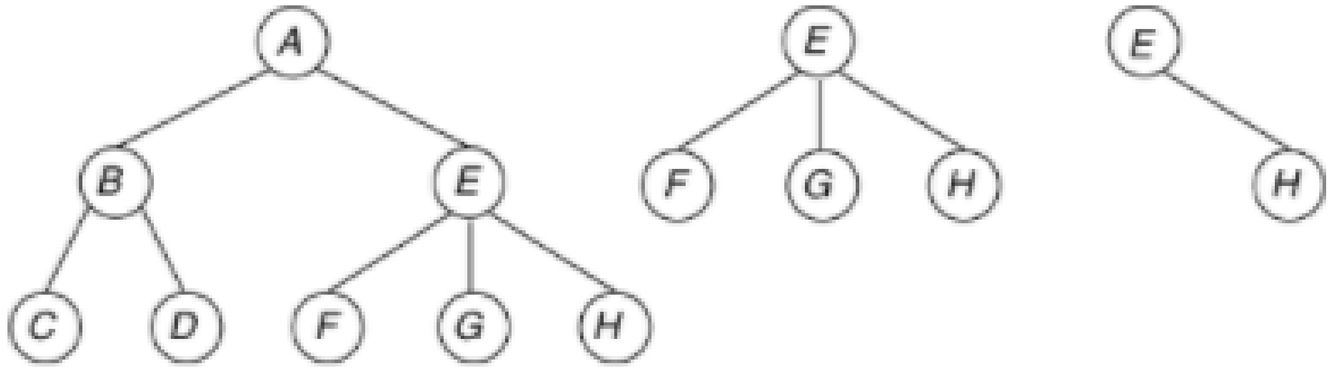
A altura h mínima é dada por $h = 1 + \lfloor \log n \rfloor$, cuja demonstração, por indução, é dada abaixo:

- Se $n = 1$, então $h = 1 + \lfloor \log n \rfloor = 1$, correto. Quando $n > 1$, suponha o resultado verdadeiro para todas as árvores binárias completas com até $n - 1$ nós. Seja T' a árvore obtida de T pela remoção de todos os nós, em número de k, do último nível. Logo, T' é uma árvore cheia com $n' = n - k$ nós. Pela hipótese de indução, $h(T') = 1 + \lfloor \log n' \rfloor$. Como T' é cheia, $n' = 2^m - 1$, para algum inteiro $m > 0$. Isto é, $h(T') = m$. Além disso, $1 \leq k \leq n' + 1$. Assim:

$$h(T) = 1 + h(T') = 1 + m = 1 + \log(n' + 1) = 1 + \lfloor \log(n' + k) \rfloor = 1 + \lfloor \log n \rfloor$$

Subárvore e subárvore parcial

Seja T uma árvore (ou uma árvore binária) e v um nó de T. Seja $T(v)$ a subárvore de T de raiz v, e S um conjunto de nós $T(v)$ tal que $T(v) - S$ é uma árvore. A árvore $T' = T(v) - S$ é chamada **subárvore parcial** de raiz v. Observe, por exemplo, a árvore T abaixo, à esquerda. A árvore ao centro é subárvore de T de raiz E, enquanto a árvore à direita é uma subárvore parcial de T de raiz E, porém não é subárvore de T. Observe que a diferença entre uma subárvore de raiz v e uma subárvore parcial de raiz v é que a primeira contém obrigatoriamente todos os descendentes de v, enquanto a segunda, não necessariamente.



Armazenamento de árvores

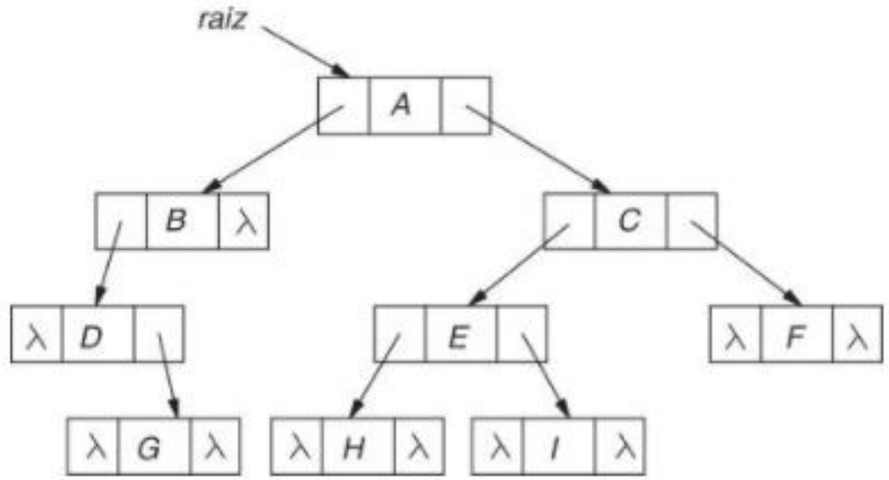
O armazenamento de árvores pode utilizar alocação sequencial ou encadeada. As vantagens e desvantagens de uma e outra já foram discutidas. Sendo a árvore uma estrutura mais complexa do que listas lineares, as vantagens na utilização da alocação encadeada prevalecem.

Não é difícil observar que a estrutura de armazenamento para árvores deve conter, em cada nó, ponteiros para seus filhos. A disposição mais econômica consiste em limitar o número de filhos a dois, exatamente o caso de árvores binárias. Note que o número de subárvores vazias cresce com o aumento do parâmetro m das árvores m-árias. Para um dado valor de n, a árvore binária é aquela que minimiza o número de ponteiros necessários.

Armazenamento de árvores

O armazenamento de uma árvore binária surge naturalmente de sua definição. Cada nó deve possuir dois campos de ponteiros, **esq** e **dir**, que apontam para as suas subárvores esquerda e direita, respectivamente. O ponteiro **ptraiz** indica a raiz da árvore. Da mesma forma que na alocação encadeada de listas lineares, a memória é inicialmente considerada uma lista de espaço disponível. Os campos do nó da árvore que contém as informações pertinentes ao problema serão aqui representados como um só campo de nome **info**.

A figura abaixo ilustra a estrutura de ponteiros usada no armazenamento de uma árvore binária.



Percurso em árvores binárias

Por **percurso** entende-se uma visita sistemática a cada um de seus nós; esta é uma das operações básicas relativas à manipulação de árvores. Uma árvore é, essencialmente, uma estrutura não sequencial. Por isso mesmo, ela pode ser utilizada em aplicações que demandem acesso direto. Contudo, mesmo nesse tipo de aplicação é imprescindível conhecer métodos eficientes para percorrer toda a estrutura. Por exemplo, para listar o conteúdo de um arquivo é necessário utilizar algoritmos para percurso.

Para percorrer uma árvore deve-se, então, visitar cada um de seus nós. Visitar um nó significa operar, de alguma forma, com a informação a ele relativa. Em geral, percorrer uma árvore significa visitar seus nós exatamente uma vez. Contudo, no processo de percorrer a árvore pode ser necessário passar várias vezes por alguns de seus nós sem visitá-los. A seguir são discutidas as ideias principais nas quais se baseiam alguns dos algoritmos de percurso em árvore.

Percurso em árvores binárias

Um dos passos de qualquer algoritmo de percurso é visitar a raiz v de cada subárvore da árvore T. Além disso, pode-se assumir que o algoritmo opere de tal forma que o percurso de T seja uma composição de percursos de suas subárvores.

Nesse caso, poder-se-iam se identificar, no percurso de T, os percursos de suas subárvores em forma contígua. Esses percursos correspondem, no algoritmo, às operações de **percorrer subárvores esquerda** e **direita** de v, para cada nó v de T. Essas três operações (visitar e percorrer subárvores esquerda e direita) compõem um algoritmo.

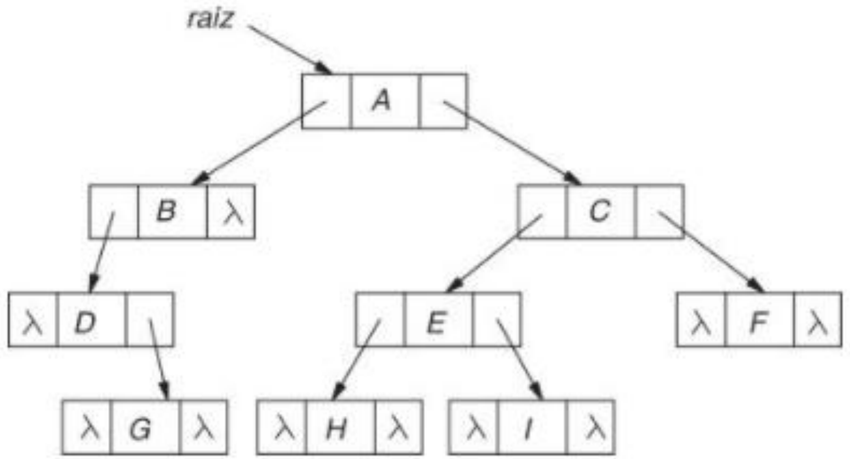
Cada um desses percursos pode ser mais ou menos adequado a um problema de aplicação dado. São apresentados a seguir três percursos diversos.

Percurso em árvores binárias

O **percurso em pré-ordem** segue recursivamente os seguintes passos, para cada subárvore da árvore:

- Visitar a raiz;
- Percorrer sua subárvore esquerda, em pré-ordem;
- Percorrer sua subárvore direita, em pré-ordem.

Para a árvore da figura ao lado, o percurso em pré-ordem para impressão de nós fornece a seguinte saída: A B D G C E H I F.

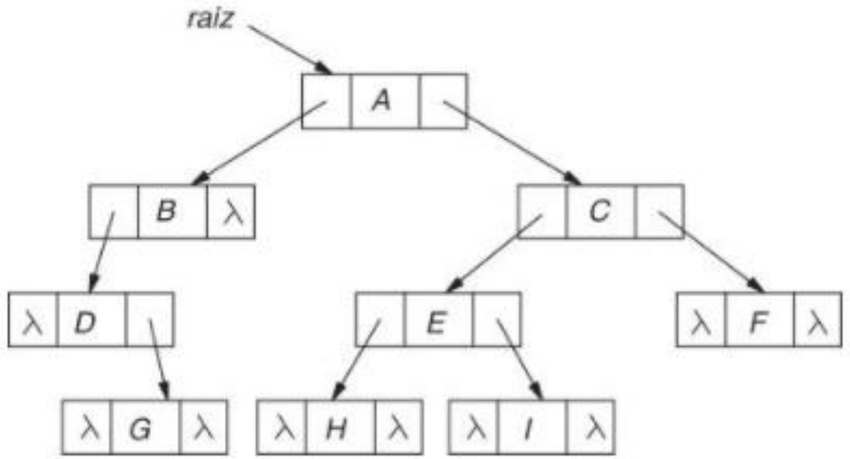


Percurso em árvores binárias

O **percurso em ordem simétrica** é composto dos passos seguintes, para cada uma de suas subárvores:

- Percorrer sua subárvore esquerda, em ordem simétrica;
- Visitar a raiz;
- Percorrer sua subárvore direita, em ordem simétrica.

Para a árvore da figura ao lado, o percurso em ordem simétrica para impressão de nós fornece a seguinte saída: D G B A H E I C F.

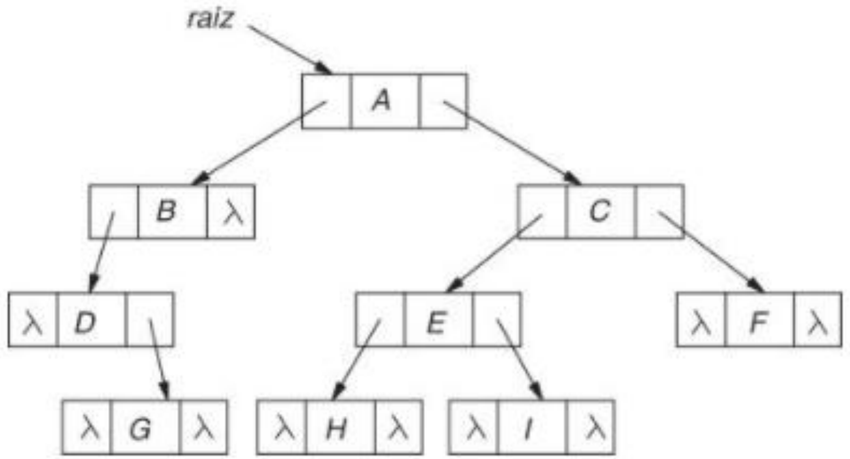


Percurso em árvores binárias

O **percurso em pós-ordem** é composto dos passos seguintes, para cada uma de suas subárvores:

- Percorrer sua subárvore esquerda, em pós-ordem;
- Percorrer sua subárvore direita, em pós-ordem.
- Visitar a raiz;

Para a árvore da figura ao lado, o percurso em ordem simétrica para impressão de nós fornece a seguinte saída: G D B H I E F C A.



Percurso em árvores binárias

```
PROGRAMA preOrdem(no)
   visita(no)
   Se no*.esq != NULL, então preOrdem(no*.esq)
   Se no*.dir != NULL, então preOrdem(no*.dir)

Se ptraiz != NULL, então preOrdem(ptraiz)
```

```
PROGRAMA simetrica(no)
   Se no*.esq != NULL, então simetrica(no*.esq)
   visita(no)
   Se no*.dir != NULL, então simetrica(no*.dir)

Se ptraiz != NULL, então simetrica(ptraiz)
```

```
PROGRAMA posOrdem(no)
   Se no*.esq != NULL, então posOrdem(no*.esq)
   visita(no)
   Se no*.dir != NULL, então posOrdem(no*.dir)

Se ptraiz != NULL, então posOrdem(ptraiz)
```

Cálculo da altura dos nós

O cálculo da altura de todos os nós de uma árvore binária é uma aplicação do percurso em pós-ordem.

A altura das folhas, pela própria definição, é 1. Para os outros nós, por exemplo v, é necessário conhecer o comprimento do maior caminho de v até um de seus descendentes. Isto equivale dizer que a altura de v deve ser calculada após a visita a seus descendentes.

O algoritmo a seguir mostra a implementação do procedimento **visita(pt)**, que executa a tarefa de determinar a altura do nó apontado por **pt**. Considera-se **altura** um campo do nó da árvore. As variáveis auxiliares **alt1** e **alt2** armazenam, respectivamente, as alturas das subárvores esquerda e direita do nó em questão. A altura desejada corresponderá à maior altura dentre as de suas duas subárvores incrementada de um.

Cálculo da altura dos nós

```
PROGRAMA visita(no)
    Se no*.esq != NULL, então:
        altE := no*.esq*.altura
    Senão:
        altE := 0

    Se no*.dir != NULL, então:
        altD := no*.dir*.altura
    Senão:
        altD := 0

    Se alt1 > alt2, então:
        no*.altura := altE + 1
    Senão:
        no*.altura := altD + 1
```

Exercícios

1. Elabore um algoritmo em pseudocódigo que, dado um determinado valor inteiro, retorne o nó que possua aquele nó. Assuma a busca em uma árvore binária, em que todos os nós são únicos;

2. Elabore um algoritmo em pseudocódigo que receba como parâmetros três valores: **valorInserir**, **valorPai** e **posicao**. O programa deve inserir, em uma árvore binária, como filho do nó cujo valor é **valorPai**, e na posição definida em **posicao** (esquerda ou direita), um novo nó cujo valor é **valorInserir**. Garanta que esse nó é único (**valorInserir** não aparece na árvore), e que seja possível inserir o nó, ou seja, o nó pai não possui nenhum outro nó na posição desejada;

3. Elabore um algoritmo em pseudocódigo que receba como parâmetro um valor inteiro a remover e remova esse nó e todos os seus descendentes da árvore;

4. Implemente os três algoritmos anteriores em Go.

Árvores Binárias de Busca

Introdução

Dado um conjunto de elementos, onde cada um é identificado por uma chave, o objetivo é localizar nesse conjunto o elemento correspondente a uma chave específica procurada.

Veremos um método de solução que emprega a árvore binária como estrutura na qual se processa a busca. Ou seja, os elementos do conjunto são previamente distribuídos pelos nós de uma árvore de forma conveniente. A localização da chave desejada é então obtida através de um caminhamento apropriado na árvore.

É importante ressaltar, mais uma vez, a relevância desse problema na área de computação, em especial nas aplicações não numéricas. Sem dúvida, a operação de busca é uma das mais frequentemente realizadas.

Introdução

Se não houver uma ordenação entre os elementos de uma árvore binária, a busca por um elemento da árvore pode ser muito custosa, já que todos os nós deverão ser percorridos.

As **árvores binárias de busca** surgem como uma forma de aprimorar o percurso e a manipulação dos dados em árvores binárias.

Além de possuir as características das árvores binárias (cada nó só pode ter no máximo dois filhos), para cada nó, os valores dos seus descendentes da esquerda são sempre inferiores ao valor do nó atual, que, por sua vez, é inferior aos nós descendentes da direita.

Conceitos básicos

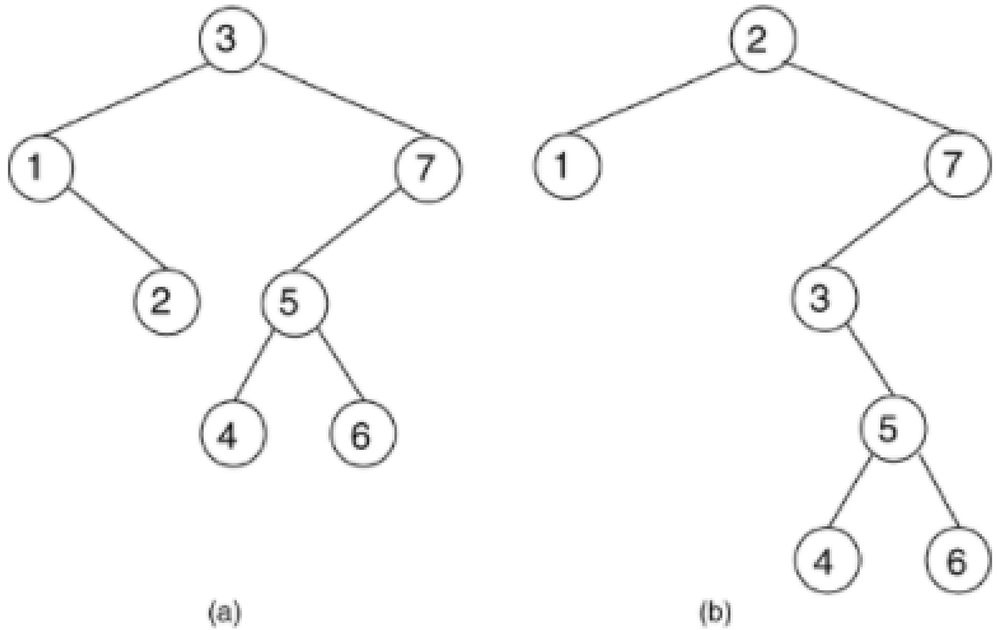
Seja $S = \{s_1, \ldots, s_n\}$ o conjunto de chaves satisfazendo $s_1 < \cdots < s_n$. Seja x um valor dado. O objetivo é verificar se $x \in S$ ou não. Em caso positivo, localizar x em S, isto é, determinar o índice j tal que $x = s_j$.

Para resolver esse problema, emprega-se uma árvore binária rotulada T, com as seguintes características:

- T possui n nós. Cada nó v corresponde a uma chave distinta $s_j \in S$ e possui como rótulo o valor $rt(v) = s_j$;
- Seja um nó v de T. Seja também v_1, pertencente à subárvore esquerda de v. Então $rt(v_1) < rt(v)$. Analogamente, se v_2 pertence à subárvore direita de v, $rt(v_2) > rt(v)$.

Conceitos básicos

A árvore T denomina-se **árvore binária de busca** para S. Naturalmente, se $|S| > 1$, existem várias árvores de busca para S. A figura abaixo ilustra duas dessas árvores para o conjunto $\{1, 2, 3, 4, 5, 6, 7\}$.



Busca

O algoritmo seguinte implementa a ideia. Suponha que a árvore esteja armazenada da forma habitual, isto é, para cada nó v, esq e dir designam os campos que armazenam ponteiros para os filhos esquerdo e direito de v, respectivamente. A raiz da árvore é apontada por $ptraiz$. O valor a retornar depende do resultado da busca:

- 0, se a árvore é vazia;
- 1, se $x \in S$. Nesse caso, pt aponta para o nó procurado;
- $2\ ou\ 3$, se $x \notin S$.

Busca

```
PROGRAMA buscaArvore(valor, no)
   Se no = NULL, então retorna 0
   Se valor = no*.chave, então retorna 1
   Se valor < no*.chave, então:
      Se no*.esq = NULL, então retorna 2

      no := no*.esq
      buscaArvore(valor, no)

   Senão:
      Se no*.dir = NULL, então retorna 3

      no := no*.dir
      buscaArvore(valor, no)
```

Inserção

Para resolver o problema de inserção de nós na árvore de busca T, utiliza-se também o procedimento *buscaArvore*. Seja x o valor da chave que se deseja inserir em T e $novo - valor$ a informação associada a x. A ideia inicial é verificar se $x \in S$. Em caso positivo, trata-se de uma chave duplicata e a inserção não pode ser realizada. Se $x \notin S$, a chave de valor x será o rótulo de algum novo nó w, situado à esquerda ou à direita de v, para $f = 2$ ou $f = 3$, respectivamente, de acordo com o procedimento *buscaArvore*. O algoritmo seguinte descreve o processo.

Inserção

```
PROGRAMA insere(valor, arvore)
    no := arvore.raiz
    f := buscaArvore(valor, no)
    Se f = 1, então Retorna “inserção inválida”

    Criar novoNo
    novoNo.chave := valor
    novoNo*.esq := NULL
    novoNo*.dir := NULL
    Se f = 0, então:
        arvore.raiz := novoNo
    Senão se f = 2, então:
        no*.esq := novoNo
    Senão:
        no*.dir := novoNo
```

Exercícios

1. Implemente em Go os algoritmos de busca e inserção em uma árvore binária de busca. Considere uma árvore com nós simples, que possuem um único valor inteiro, adotado também como chave do nó.

Algoritmos de Ordenação

Introdução

As aulas anteriores trataram, em geral, de estruturas genéricas, adequadas à representação de quaisquer massas de dados. Neste capítulo serão descritas técnicas distintas para solucionar um problema que aparece como pré-processamento em muitas aplicações que envolvam o uso de tabelas - a obtenção de uma tabela ordenada.

O problema da ordenação foi um dos primeiros a gerar discussões sobre implementações eficientes ou não. Métodos mais simples, como a ordenação bolha e a ordenação por inserção, apesar de possuírem complexidade de pior caso ruim, são bastante utilizados em razão de sua extrema simplicidade de implementação. Também não pode ser esquecido que esses métodos podem ser convenientes quando a tabela é pequena ou está quase ordenada.

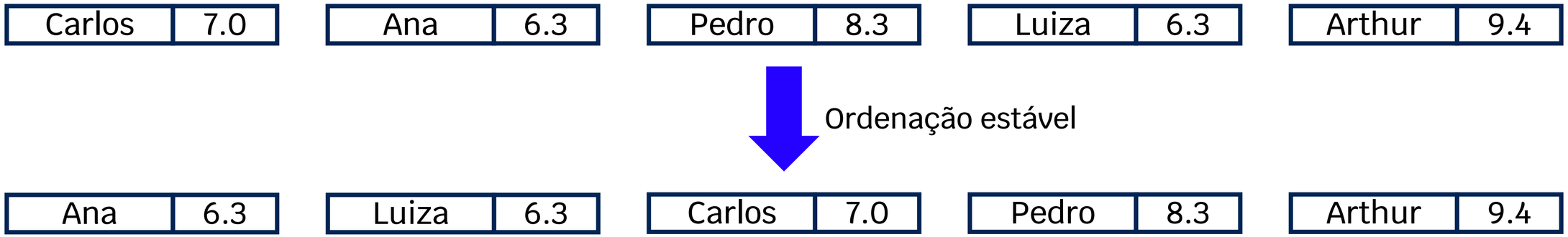
https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

https://www.youtube.com/watch?v=ZZuD6iUe3Pc

Introdução

Os algoritmos de ordenação podem ser ou não *inplace*, e ser ou não *estáveis*.

- Um algoritmo *inplace* não necessita de memória adicional, ou necessita de uma memória constante para executar a operação. Bubblesort e Insertion sort são exemplos de algoritmos *inplace*, enquanto que Mergesort não é considerado *inplace*;
- Um algoritmo *estável* é aquele em que elementos de chaves iguais no vetor não ordenado permanecem na mesma ordem relativa no vetor ordenado. Os algoritmos bubblesort e mergesort são estáveis (desde que se tenha alguns cuidados na implementação). Já o heapsort não é estável.



Bubblesort

O método de ordenação bolha é bastante simples, e talvez seja o método de ordenação mais difundido. Uma iteração do mesmo se limita a percorrer a tabela do início ao fim, sem interrupção, trocando de posição dois elementos consecutivos sempre que estes se apresentarem fora de ordem.

Intuitivamente percebe-se que a intenção do método é mover os elementos maiores em direção ao fim da tabela. Ao terminar a primeira iteração pode-se garantir que as trocas realizadas posicionam o maior elemento na última posição. Na segunda iteração, o segundo maior elemento é posicionado, e assim sucessivamente, até não haver mais mudanças.

O algoritmo que se segue implementa este método. A tabela se encontra armazenada na estrutura *L*. O algoritmo ordena *L* segundo valores não decrescentes do campo chave.

Bubblesort

```
PROGRAMA bubblesort(lista, n)
   trocou := verdadeiro
   limite := n
   Enquanto trocou = verdadeiro, faça:
      trocou := falso
      limite := limite - 1
      Para i := 0, 1, ..., limite - 1, faça:
         Se lista[i].chave > lista[i + 1].chave, então:
            Trocar lista[i] com lista[i + 1]
            trocou := verdadeiro
```

Pior caso (elementos em ordem decrescente): $O(n^2)$
Melhor caso (elementos ordenados): $O(n)$
Caso médio: $O(n^2)$

Bubblesort

Vantagens:

- Implementação simples
- Estável
- *Inplace*

Desvantagens

- Complexidade $O(n^2)$
- Tempo de execução degrada muito conforme a quantidade de elementos aumenta

Quando usar:

- A estrutura a ser trabalhada é pequena
- Não há tempo para implementar algo mais eficiente
- A operação de ordenação não impacta significativamente na complexidade geral do software

Selection sort

A ordenação por seleção (*selection sort*) é um algoritmo simples de ordenação que, inicialmente, passa o menor elemento para a primeira posição, depois o segundo menor elemento para a segunda posição, e assim sucessivamente.

```
PROGRAMA selectionsort(lista, n)
   Para i := 0, 1, ..., n - 2, faça:
      min := i
      Para j := i + 1, i + 2, ..., n - 1, faça:
         Se lista[j] < lista[min], então:
            min := j

      Se min != i, então:
         Troca lista[i] com lista[min]
```

Pior caso: $O(n^2)$
Melhor caso: $O(n^2)$
Caso médio: $O(n^2)$

Selection sort

Vantagens:

- Implementação simples
- Tempo de execução bem menor que o bubblesort (realiza menos trocas)
- *Inplace*

Desvantagens

- Complexidade $O(n^2)$
- Não estável
- Muito lento para grandes quantidades de elementos
- Faz sempre $(n^2 - n)/2$ comparações, independentemente do vetor estar ordenado ou não

Quando usar:

- Mesmos casos que o bubblesort

Insertion sort

O método de ordenação por inserção é também bastante simples, sendo sua complexidade equivalente à do bubblesort.

Imagine uma tabela já ordenada até o i-ésimo elemento. A ordenação da tabela pode ser estendida até o $(i+1)$-ésimo elemento por meio de comparações sucessivas deste com os elementos anteriores, isto é, com o i-ésimo elemento, com o $(i+1)$-ésimo elemento etc., procurando sua posição correta na parte da tabela que já está ordenada.

Pode-se então deduzir um algoritmo para implementar o método: considera-se sucessivamente todos os elementos, a partir do segundo deles, em relação à parte da tabela formada pelos elementos anteriores ao elemento considerado em cada iteração.

Insertion sort

```
PROGRAMA insertionsort(lista, n)
   Para j := 1, 2, ..., n - 1, faça:
      chave := lista[j]
      i := j - 1
      Enquanto i >= 0 e lista[i] > chave, faça:
         lista[i + 1] := lista[i]
         i := i - 1
      lista[i + 1] := chave
```

Pior caso: $O(n^2)$
Melhor caso: $O(n)$
Caso médio: $O(n^2)$

Insertion sort

Vantagens:

- Implementação simples
- Tempo de execução melhor que o Selection Sort
- *Inplace* e estável

Desvantagens

- Complexidade $O(n^2)$
- Lento para grandes quantidades de elementos

Quando usar:

- Mesmos casos que o bubblesort
- Particularmente eficiente para pequenas quantidades de elementos

Algoritmos de ordenação por divisão e conquista

Alguns dos algoritmos mais eficientes de ordenação utilizam a estratégia de divisão e conquista:

- Quebrar a entrada original em duas partes;
- Recursivamente, ordenar cada uma das partes;
- Combinar as duas partes ordenadas.

Existem duas categorias de soluções:

- Quebra simples e combinação difícil
- Quebra difícil e combinação simples

Quicksort

O nome *quicksort* (ordenação rápida) já indica o que se deve esperar do método, que é, na realidade, um dos mais eficientes dentre os conhecidos.

Dada uma tabela *L* com *n* elementos, o procedimento recursivo para ordenar *L* consiste nos seguintes passos:

- Se $n = 0$ ou $n = 1$ então a tabela está ordenada;
- Escolha qualquer elemento x em $L \rightarrow$ este elemento é chamado **pivô**;
- Separe $L - \{x\}$ em dois conjuntos de elementos disjuntos: $S_1 = \{w \in L - \{x\} \mid w < x\}$ e $S_2 = \{w \in L - \{x\} \mid w \geq x\}$;
- O procedimento de ordenação é chamado recursivamente para S_1 e S_2;
- *L* recebe a concatenação de S_1, seguido de x, seguido de S_2.

Quicksort

Dois pontos são decisivos para o bom desempenho do algoritmo:

- **Escolha do pivô:** Uma solução utilizada com bons resultados é a escolha da mediana dentre três elementos: o primeiro, o último e o central. Uma solução mais prática é sempre utilizar o elemento no início ou no final da tabela.
- **Particionamento da tabela:**
  - Dois ponteiros são utilizados:
    - i é inicializado apontando para o primeiro elemento da tabela, percorrendo-a enquanto os valores apontados são menores do que o pivô;
    - j é inicializado na penúltima posição, efetuando a tarefa inversa.
  - Os percursos são interrompidos quando i aponta para um elemento maior do que o pivô e j aponta para um elemento menor do que o pivô.

Quicksort

Dois pontos são decisivos para o bom desempenho do algoritmo:

- **Particionamento da tabela (cont.):**
  - Duas situações podem ocorrer:
    - Se $i \leq j$, os elementos da tabela devem ser trocados e o procedimento deve prosseguir;
    - Se $i > j$, a partição já está determinada.
  - O pivô, que se encontra na última posição da tabela, deve ser trocado com o elemento de índice i.
  - Após a troca, os elementos de índice menor do que i formam o conjunto de elementos maiores do que o pivô.

Quicksort

```
PROGRAMA quicksort(lista, inicio, fim)
   i, j := inicio, fim
   pivo := lista[i]

   Enquanto i <= j, faça:
      Enquanto lista[i] < pivo, faça:
         i := i + 1
      Enquanto pivo < lista[j], faça:
         j := j - 1
      Se i <= j, então:
         Troca lista[i] com lista[j]
         i := i + 1
         j := j - 1

   Se inicio < j, então:
      quicksort(lista, inicio, j)
   Se i < fim, então:
      quicksort(lista, i, fim)
```

Quicksort

```
PROGRAMA quicksort(lista, inicio, fim)
   i, j := inicio, fim
   pivo := lista[i]

   Enquanto i <= j, faça:
      Enquanto lista[i] < pivo, faça:
         i := i + 1
      Enquanto pivo < lista[j], faça:
         j := j - 1
      Se i <= j, então:
         Troca lista[i] com lista[j]
         i := i + 1
         j := j - 1

   Se inicio < j, então:
      quicksort(lista, inicio, j)
   Se i < fim, então:
      quicksort(lista, i, fim)
```

Trecho responsável por definir o ponto de particionamento.

Complexidade O(n)

Quicksort

Pior caso:

- Pivô escolhido é sempre menor ou maior que todos os elementos, ou seja, a lista já está ordenada
- A recursão é executada, então, outras $O(n)$ vezes
- Complexidade $O(n) \times O(n) = O(n^2)$

Melhor caso:

- Pivô escolhido é sempre exatamente na metade da tabela, ou seja, cada recursão considera uma tabela com tamanho igual à metade da tabela anterior
- A recursão é executada, então outras $O(\log n)$ vezes
- Complexidade $O(\log n) \times O(n) = O(n \log n)$

É possível demonstrar que a complexidade de caso médio também é $O(n \log n)$

Quicksort

Vantagens:

- Complexidade $O(n \log n)$
- *Inplace*

Desvantagens:

- Tradicionalmente baseia-se em chamadas recursivas, o que pode limitar o tamanho da tabela por conta de restrições da linguagem implementada (pilha de recursão ser muito profunda)
- Não é estável

Quando usar

- Praticamente todos os casos
- Quando se tem uma distribuição supostamente (ou próxima de) aleatória

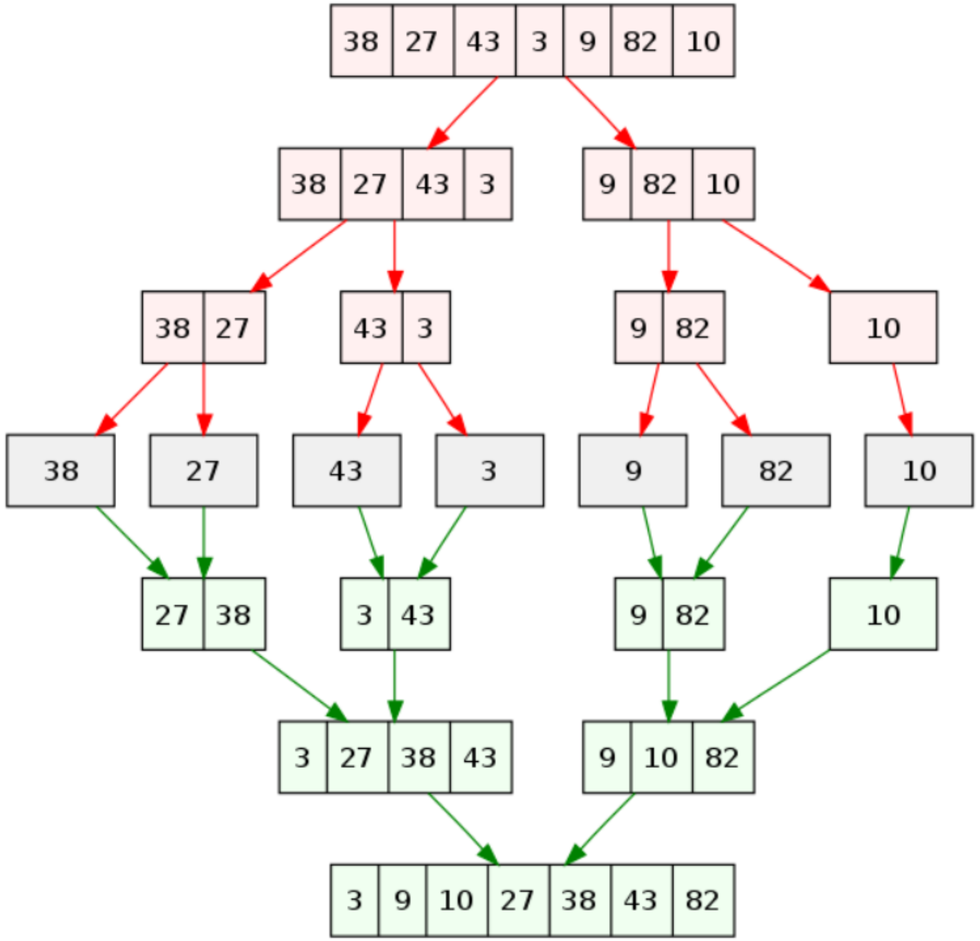
Mergesort

Este método tem como procedimento básico o de intercalação de listas.

A ideia básica do método é intercalar as duas metades da lista desejada quando estas já se encontram ordenadas. Deseja-se então ordenar primeiramente as duas metades, o que pode ser feito utilizando recursivamente o mesmo conceito.

Sejam duas listas *A* e *B*, ordenadas, com respectivamente *n* e *m* elementos. As duas listas são percorridas por ponteiros *ptA* e *ptB*, armazenando o resultado da intercalação na lista *C*, apontada pelo ponteiro *ptC*. O primeiro elemento de *A* é comparado com o primeiro elemento de *B*; o menor valor é colocado em *C*. O ponteiro da lista onde se encontra o menor valor é incrementado, assim como o ponteiro da lista resultado; o processo se repete até que uma das listas seja esgotada. Neste ponto, os elementos restantes da outra lista são copiados na lista resultado.

Mergesort


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

Mergesort

```
PROGRAMA mergesort(lista, aux, inicio, fim)
    Se inicio < fim, então:
        meio := parte inteira de (inicio + fim) / 2
        mergesort(lista, aux, inicio, meio)
        mergesort(lista, aux, meio + 1, fim)
        combina(lista, aux, inicio, meio, fim)
```

```
PROGRAMA combina(lista, aux, inicio, meio, fim)
    i, j := inicio, meio + 1
    Para k := inicio, ..., fim, faça:
        aux[k] := lista[k]

    k := i
    Enquanto i <= meio e j <= fim, faça:
        Se aux[i] <= aux[j], então:
            lista[k] := aux[i]
            i := i + 1
        Senão:
            lista[k] := aux[j]
            j := j + 1
        k := k + 1

    Enquanto i <= meio, faça:
        lista[k] := aux[i]
        i := i + 1
        k := k + 1

    Enquanto j <= fim, faça:
        lista[k] := aux[j]
        j := j + 1
        k := k + 1
```

Mergesort

```
PROGRAMA mergesort(lista, aux, inicio, fim)
    Se inicio < fim, então:
        meio := parte inteira de (inicio + fim) / 2
        mergesort(lista, aux, inicio, meio)
        mergesort(lista, aux, meio + 1, fim)
        combina(lista, aux, inicio, meio, fim)
```

Essa função é responsável por combinar as duas metades da lista, entre os índices `inicio` e `fim`

Quando o programa chega nesse ponto, já se sabe que as duas metades do vetor (entre `inicio` e `meio`, e entre `meio + 1` e `fim`) estão ordenadas

Veja que é necessário um vetor auxiliar para copiar os dados. Esse vetor “acompanha” o processo de recursão, para que não sejam criados vários vetores durante a execução

A complexidade desse trecho é **O(n)**

```
PROGRAMA combina(lista, aux, inicio, meio, fim)
    i, j := inicio, meio + 1
    Para k := inicio, ..., fim, faça:
        aux[k] := lista[k]

    k := i
    Enquanto i <= meio e j <= fim, faça:
        Se aux[i] <= aux[j], então:
            lista[k] := aux[i]
            i := i + 1
        Senão:
            lista[k] := aux[j]
            j := j + 1
        k := k + 1

    Enquanto i <= meio, faça:
        lista[k] := aux[i]
        i := i + 1
        k := k + 1

    Enquanto j <= fim, faça:
        lista[k] := aux[j]
        j := j + 1
        k := k + 1
```

Mergesort

Diferente do quicksort, no mergesort cada recursão considera exatamente a metade da lista anterior. Portanto, a complexidade do algoritmo, mesmo no pior caso, é igual a $O(\log n) \times O(n) = O(n \log n)$

Vantagens:

- Complexidade $O(n \log n)$
- Estável

Desvantagens:

- Tradicionalmente baseia-se em chamadas recursivas, o que pode limitar o tamanho da tabela por conta de restrições da linguagem implementada (pilha de recursão ser muito profunda)
- Não é *inplace*, precisando de uma área auxiliar do tamanho da sequência original

Quando usar

- Quando a frequência com que a tabela já está ordenada é relativamente alta

Resumindo

Algoritmos super eficientes assintoticamente (quicksort, mergesort, heapsort, etc.) tendem a fazer muitas trocas

Para entradas com poucos elementos (ordem de dezenas), o melhor algoritmo de ordenação costuma ser o insertion sort

Para uma sequência que está quase ordenada, o insertion sort também costuma ser a melhor escolha, pois ele tende a fazer menos trocas

O quicksort é considerado o melhor método de ordenação para grandes quantidades de elementos ($n \geq 10.000$)

O quicksort é, tipicamente, 2 vezes mais rápido que o mergesort

IBMEC.BR

/IBMEC

IBMEC

@IBMEC_OFICIAL

@IBMEC

Ibmec