# Estruturas de Dados

Versão em elaboração

Prof.: Andreu Carminati, Dr.

Fevereiro de 2021

# SUMÁRIO

# Capítulo 1: Introdução

Este roteiro de estudos tem por objetivo guiar o aluno no processo de ensino e aprendizagem para Atividades Não Presenciais durante o período da pandemia relacionada a COVID-19. O texto aqui apresentado foi pensado para ter cobertura suficiente sobre os conteúdos da Unidade, entretanto, é recomendável que o aluno não desconsidere a utilização dos livros constantes nas Bibliografias Básica e Complementar do plano de ensino. Este roteiro foi baseado didaticamente no livro texto base da unidade (Estruturas de Dados e Seus Algoritmos - Jayme Luiz Szwarcfiter e Lilian Markenzon)[1] e materiais do próprio docente.

## Como Utilizar este Material

Como este guia cobre todo o conteúdo que deve ser visto na Unidade, ele também apresenta exercícios/questões e problemas práticos em cada capítulo que deverão ser submetidos ao professor para contabilização da carga horária, conforme planejamento. Algumas das questões foram retiradas de concursos relevantes que incluem assuntos da área de Estruturas de Dados. Para todas as estruturas discutidas são fornecidas implementações em linguagem C, entretanto algumas operações podem estar faltando de maneira proposital. Estas operações deverão ser implementadas como parte dos problemas práticos que serão apresentados.

## Breve Contextualização Do Que Será estudado

O primeiro assunto que será estudado será a recursividade, técnica esta que permite a solução mais facilitada e elegante de problemas que envolvem estratégias de divisão e conquista. Conceitos básicos sobre complexidade de algoritmos também compõem o escopo deste material. Também serão estudados algoritmos de ordenação de dados em suas mais diversas formas, como ordenação bolha, por inserção, intercalação e ordenação rápida. Em relação aos algoritmos de ordenação será visto que a optimalidade de algoritmos de ordenação está em soluções cujo pior caso é $O(n\log n)$, mas que existem soluções com caso médio $O(n\log n)$ muito boas, como as variantes da ordenação rápida. A seguir, serão apresentadas as estruturas em alocação sequencial, listas, pilhas e filas, as quais fornecem uma maneira conveniente para o armazenamento de dados para uma vasta gama de problemas. Em seguida, serão apresentadas as estruturas em alocação encadeada, que possuem a vantagem de possibilitar o crescimento do conjunto de dados sem a necessidade de alocação contínua de memória. Outro assunto a ser estudado são as árvores, as quais permitem a implementação de soluções para problemas que envolvem desde a representação de operações matemáticas bem como a indexação de dados de uma maneira eficiente para um volume gigantesco de informação. Ressalta-se aqui a importância da Unidade Curricular para a construção de sistemas de software, pois a escolha correta das estruturas habilita eficiência e escalabilidade no trato dos dados.

# Organização do Restante do Texto

O restante deste texto está organizado da seguinte forma: o Capítulo 2 apresenta conceito da recursividade. O Capítulo 3 apresenta noções de complexidade de algoritmos. O Capítulo 4 discorre sobre as principais técnicas para ordenação de dados. Já o Capítulo 5 possui como foco as estruturas para armazenamento utilizando alocação sequencial (listas, pilhas e filas). o Capítulo 6 apresenta as estruturas em alocação encadeada. Serão estudadas neste capítulo as listas, pilhas e filas encadeadas. Já o Capítulo 7 apresenta conceitos de árvores computacionais. As árvores agrupam uma família grande estruturas com variadas formas e diferentes propósitos, como as adequadas para representação construções estruturadas e as que são aplicadas ao armazenamento e indexação de conjuntos.

# Capítulo 2: Recursividade

Neste capítulo será apresentada a recursão como técnica para a resolução de problemas de computação, mas que também pode ser utilizada na representação de diversos problemas matemáticos.

## Conceitos Básicos

A recursão é um tipo especial de procedimento/algoritmo que pode chamar a si mesmo. É muito utilizada na implementação de analisadores sintáticos em compiladores (analisador descendente recursivo), na implementação de algoritmos de grafos e árvores e qualquer outro problema programação que envolva repetição. Algumas linguagens fornecem apenas recursão (paradigma lógico e funcional), não possuindo, por exemplo, estruturas de repetição para resolução de problemas. Na matemática, as definições recursivas aparecem em problemas cujas definições dependem do próprio problema, como no caso do fatorial, onde têm-se que $fatrial(n) = n \times fatorial(n-1)$, logo, a definição do fatorial depende do próprio fatorial. Na Figura 2.1 é apresentado um exemplo alegórico de recursão, onde um problema é definido utilizando sua própria definição em um caso reduzido.

Na computação, a recursão costuma ser muito utilizada em técnicas de *divisão e conquista*, onde um problema é dividido em sub-problemas iguais, porém menores, cujas soluções podem ser combinadas para se ter a solução do problema original. A programação recursiva se contrapõe a programação iterativa, onde estruturas de repetição são empregadas. Todo procedimento ou algoritmo recursivo pode ser reescrito de maneira não recursiva, ou seja, de maneira iterativa e o contrário também é verdadeiro. A pergunta que fica é, se são abordagens equivalentes, por que usar? A resposta é simples: pela simplicidade e elegância e pela facilidade para se provar a complexidade computacional, conforme será visto mais adiante.

Figura 2.1: Recursão e uma imagem
(Fonte: https://commons.wikimedia.org/wiki/File:Parse_Tree_Derivations.svg)

# Resolvendo um Problema Recursivo

Primeiramente, todo problema recursivo é dividido em duas partes:

- **Caso base**, que representa a menor instância do problema.
- **Passo de recursão ou caso geral**, o qual busca reduzir o problema em instâncias menores.

Por exemplo, voltando ao fatorial mencionado anteriormente, têm-se:

- Caso base: $fat(0) = 1$, que é a menor instância do problema.
- Caso geral: $fat(n) = n \times fat(n-1)$, onde $fat(n-1)$ representa a redução para uma instância menor do problema.

No Algoritmo 1 é apresentada uma solução de implementação recursiva para o problema do cálculo do fatorial, enquanto no Algoritmo 2 a mesma solução é fornecida, mas através de uma abordagem iterativa. Perceba que na solução recursiva não existe a utilização de estruturas de repetição. No Algoritmo 3 é apresentado ainda uma solução alternativa para o problema, evidenciando que sempre existem muitas possibilidades.

**Algoritmo 1** Algoritmo para o cálculo do fatorial recursivo

```
1: function FATORIAL(X)
2:     if X = 0 then
3:         return 1
4:     else
5:         return X × fatorial(X − 1)
6:     end if
7: end function
```

**Algoritmo 2** Algoritmo para o cálculo do fatorial iterativo

```
1: function FATORIAL(X)
2:     fat ← 1
3:     for i de 1 até X do
4:         fat ← fat × i
5:     end for
6:     return fat
7: end function
```

**Algoritmo 3** Algoritmo para o cálculo do fatorial iterativo (outra versão)

```
1: function FATORIAL(X)
2:     fat[0] ← 1
3:     for i de 1 até X do
4:         fat[i] ← i × fat[i − 1]
5:     end for
6:     return fat
7: end function
```

Considerando ainda a solução recursiva para o problema do fatorial mostrada no Algoritmo 1, uma implementação possível em C é apresentada na Listagem de código 2.1.

```
int fatorial(int x){
  if(x == 0){
    return 1;
  }
  return x*fatorial(x-1);
}
```

Listagem de código 2.1: Implementação do fatorial

Ainda considerando a execução da Listagem de código 2.1 para um valor de $x = 5$, a seguinte dinâmica de execução é obtida, onde cada instância de chamada da função fatorial deve aguardar o retorno da chamada seguinte para terminar sua computação, onde $|->$ representa que uma função chamou outra:

```
Função main chama n = fatorial(5):
|->fatorial(x=5)
.       |->fatorial(x=4)
.       .       |->fatorial(x=3)
.       .       .       |->fatorial(x=2)
.       .       .       .       |->fatorial(x=1)
.       .       .       .       .       |->fatorial(x=0)
.       .       .       .       .       return 1; // caso base, linha 3
.       .       .       .       return 1*1; (1)
.       .       .       return 2*1; (2)
.       .       return 3*2; (6)
.       return 4*6; (24)
return 5*24; (120)
Função main obtem n = 120.
```

Outros problemas são facilmente implementados com recursão, como a Sequência de Fibonacci, representada por: 0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987... Tal sequência é importante em diversas aplicações como finanças, computação e biologia. A definição recursiva (caso geral) é:

$$fib(n) = fib(n-1) + fib(n-2) \tag{2.1}$$

onde (caso base):

$$fib(0) = 0 \quad e \quad fib(1) = 1 \tag{2.2}$$

Como outro exemplo, existe o problema do cálculo da multiplicação de dois números naturais, através de incrementos sucessivos. Solução:

- Considera-se $x \times y$:
- Caso base: $1 \times y = y$
- Caso geral $(x > 1)$: $y + ((x-1) \times y)$ – pode-se chamar $((x-1) \times y)$ de passo de recursão.

É importante salientar que, para alguns problemas, a definição dos casos nem sempre é trivial. Alguns problemas interessantes que admitem soluções por

recursividade como as Torres de Hanoi e diferentes estratégias de busca em vetores e matrizes.

# Questões

***Questão 1:.*** Implemente em C um procedimento que imprima uma string invertida na tela, sem usar laço de repetição.

***Questão 2:.*** Escreva uma função recursiva que calcule a multiplicação de dois números naturais, através de incrementos sucessivos.

***Questão 3:.*** Escreva uma função recursiva que ache o maior elemento em um vetor de inteiros.

***Questão 4:.*** Escreva uma função recursiva que some os elementos em um vetor de inteiros.

***Questão 5:.*** Os números tetranacci representam uma generalização da sequência fibonacci. Eles iniciam com quatro termos pré-determinados e a partir daí todos os demais números são obtidos pela soma dos quatro números anteriores. Os primeiros números são: 0, 1, 1, 2, 4, 8, 15, 29, 56, 108, 208, ... Faça uma função recursiva que receba um número N e retorne o N-ésimo termo da sequência.

***Questão 6:.*** Dado um número n na base decimal, escreva uma função recursiva em C que converte (basta imprimir) este número para binário.

***Questão 7:.*** Fazer o mesmo para a base hexadecimal.

***Questão 8:.*** Implemente uma função que calcula o MDC (máximo divisor comum):

- $mdc(x, y) = y$, se $x \geq y$ e $x \mod y = 0$
- $mdc(x, y) = mdc(y, x)$, se $x < y$
- $mdc(x, y) = mdc(y, x \mod y)$, caso contrário.

***Questão 9:.*** Considerando um array de inteiros, faça uma função que inverta a posição dos seus elementos.

# Capítulo 3: Noções de Complexidade de Algoritmos

O objetivo deste capítulo é fornecer um ferramental teórico para auxiliar na compreensão do comportamento de diferentes algoritmos, comportamento este relacionado ao tempo de resposta considerando os dados de entrada utilizados. Tal ferramental é importante para decisões relativas a escolha das melhores soluções (ou mais adequadas) para resolução de um determinado problema. O conhecimento aqui apresentado, por exemplo, pode auxiliar no entendimento do conceito de optimalidade, o qual, por exemplo, é evidenciado no capítulo referente a ordenação.

## Conceitos Básicos

Algoritmos concebidos para resolver o mesmo problema muitas vezes diferem dramaticamente em sua eficiência, sendo que estas diferenças podem ser muito mais significativas do que as diferenças induzidas pelo hardware ou pelo softwate que implementa a solução. Tal percepção de diferença aparece na variável tempo, ou seja, no tempo de resposta dos algoritmos.

Não existe um única maneira de avaliar o desemplenho de um algoritmo. Pode-se, por exemplo, avaliar o desempenho de um algoritmo (em relação ao tempo) usando métodos empíricos, ou seja através de medições de tempo de resposta variando o conjunto de dados de entrada. Por outro lado, também pode-se utilizar métodos analíticos (matemáticos) para estimar a ordem de grandeza deste mesmo tempo, o que é preferível, pois não depende do computador utilizado.

## Avaliação Analítica

Considerando técnicas analíticas para avaliação de desempenho e comportamento temporal, obter uma expressão matemática para avaliar o tempo não é simples, e pode exigir certas simplificações, como considerar a **quantidade de dados manipulado extremamente grande** e desconsiderar **constantes aditivas ou multiplicativas nas expressões** obtidas. Considerando uma expressão matemática para avaliação de tempo, será utilizado *número de entradas n* como sendo o parâmetro variável. Em outras palavras, pretende-se descobrir o comportamento temporal de um algoritmo em função da quantidade de dados processados, tudo isso de maneira analítica.

A execução de um algoritmo pode ser dividida em etapas elementares, que são denominadas de *passos*. Um passo é um número fixo de **operações básicas** com tempos considerados constantes, do ponto de vista do processador. Em um algoritmo, a operação básica de maior frequência é denominada **operação dominante**. Então, o objetivo real é obter uma expressão matemática de avaliação de tempo que fornece o número de passos efetuados (execução da operação dominante) pelo algoritmo a partir

de uma certa entrada. Por exemplo, considera-se a soma de matrizes:

$$c_{ij} = a_{ij} + b_{ij} \tag{3.1}$$

Considerando a equação matricial 3.1, pode-se resolver ela através do Algoritmo 4. Este algoritmo efetua $n^2$ passos para somar duas matrizes de $n \times n$, ou seja, a linha 4 é executada $n^2$, pois está dentro de duas estruturas de repetição aninhadas as quais estão condicionadas a $n$.

**Algoritmo 4** Algoritmo para soma de matriz

```
1: procedure SOMA_MATRIZ(A_n, B_n, C_n)
2:     for i de 1 até n do
3:         for j de 1 até n do
4:             c_ij ← a_ij + b_ij
5:         end for
6:     end for
7: end procedure
```

Agora, considera-se a multiplicação de matrizes:

$$d_{ij} = \sum_{1 \leq k \leq n} a_{ik} \times b_{kj} \tag{3.2}$$

Considerando a equação matricial de multiplicação 3.2, pode-se resolver ela através do Algoritmo 5. Este algoritmo efetua $n^3$ passos para multiplicar duas matrizes de $n \times n$, ou seja, a linha 6 é executada $n^3$, pois está dentro de 3 estruturas de repetição aninhadas as quais estão todas condicionadas a $n$.

**Algoritmo 5** Algoritmo para multiplicação de matriz

```
1: procedure MULTIPLICA_MATRIZ(A_n, B_n, C_n)
2:     for i de 1 até n do
3:         for j de 1 até n do
4:             c_ij ← 1
5:             for k de 1 até n do
6:                 c_ij ← c_ij + a_ik × b_kj
7:             end for
8:         end for
9:     end for
10: end procedure
```

Mais adiante, serão estudados os algoritmos de ordenação. Por exemplo, o algoritmo *insertsort* (ordenação por inserção) leva aproximadamente $c_1 n^2$ unidades de tempo para ordenar um conjunto de dados, onde $c_1$ é uma constante que não depende de $n$. Por outro lado, o *mergesort* (ordenação por inserção) leva tempo aproximadamente igual a $c_2 nlgn$, onde $lgn$ representa $log_2 n$ e $c_2$ é outra constante que também não depende de $n$, sendo que, no geral, $c_1 < c_2$.

Agora, pode-se considerar o seguinte cenário:

- *Insertsort* implementado em um computador rápido e em assembly por um programador experiente (computador A), resultando em $c_1 = 2$. A executa um bilhão de instruções por segundo.
- *Mergesort* implementado por um programador mediano em C usando um compilador ineficiente (computador B), resultando em $c_2 = 50$. A é 100 vezes mais rápido que B.

Isto posto, quanto tempo levaria para ordenar um milhão de números? Resposta:

- Computador A: $\frac{2.(10^6)^2 instrucoes}{10^9 instrucoes/segundo} = 20000 \quad segundos$
- Computador B: $\frac{50.10^6.lg10^6}{10^7 instrucoes/segundo} = 100 \quad segundos$

Percebe-se pelo exemplo anterior que a diferença entre as duas abordagens está relacionada muito mais com a técnica utilizada do que com a tecnologia utilizada para resolver o problema.

# Noções de Complexidade de Tempo

Seja um algoritmo $A$ e $\{E_1, ..., E_m\}$ o conjunto de todas as entradas possíveis, e $t_i$ o número de passos quando a entrada fornecida é $E_i$. Desta maneira, definem-se:

- **Complexidade de tempo de pior caso** $= max_{E_i \in E}\{t_i\}$, ou seja, o pior tempo possível considerando uma entrada específica entre todas as possíveis.
- **Complexidade de tempo de melhor caso** $= min_{E_i \in E}\{t_i\}$, ou seja, o melhor tempo possível considerando uma entrada específica.
- **Complexidade de tempo de caso médio** $= \sum_{1 \leq i \leq m} pi \times ti$ onde $pi$ é a probabilidade de ocorrência da entrada $E_i$, ou seja, o tempo médio, considerando todas as entradas possíveis.

Pode-se definir também a complexidade de espaço, a qual está relacionada com a demanda de memória para uma determinada entrada. Neste caso, utiliza-se de modo mais usual a complexidade de pior caso (entrada mais desfavorável). A partir daqui, somente será levada em consideração a complexidade de tempo, pois sua obtenção é mais relevante e menos intuitiva que a de espaço.

# Notação Assintótica

As definições anteriores implicam ao atendimento das simplificações sugeridas anteriormente, onde pode-se desprezar constantes aditivas ou multiplicativas. Por exemplo, $3n$ será aproximado para $n$ pelo desprezo da constante multiplicativa. Além disso, como o interesse é restrito aos valores assintóticos, termos de menor grau podem ser desprezados, como, por exemplo, $n^2 + n$ será aproximado para $n^2$ pela eliminação do termo de menor grau $n$.

Neste contexto, a anotação assintótica descreve o comportamento limitante de funções considerando as simplificações propostas. Entre as notações assintóticas, a mais utilizada é a do **Big *O***, a qual fornece um limite superior para o crescimento das funções utilizadas para representação da complexidade dos algoritmos. Segundo a notação $O$:

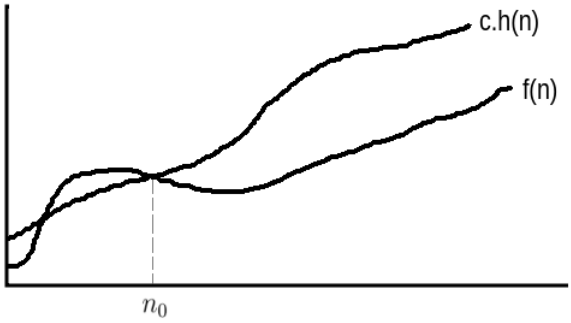
- Seja $f$ e $h$ duas funções reais positivas da variável n.

- Diz-se que $f$ é $O(h)$ quando existir uma constante $c > 0$ e um valor $n_0$ tal que:

$$n > n_0 \rightarrow f(n) \leq c.h(n) \tag{3.3}$$

Na Figura 3.1 está representado graficamente a relação anterior sobre $h(n)$ e $f(n)$ relativa a notação Big *O*.

Figura 3.1: Notação Big O



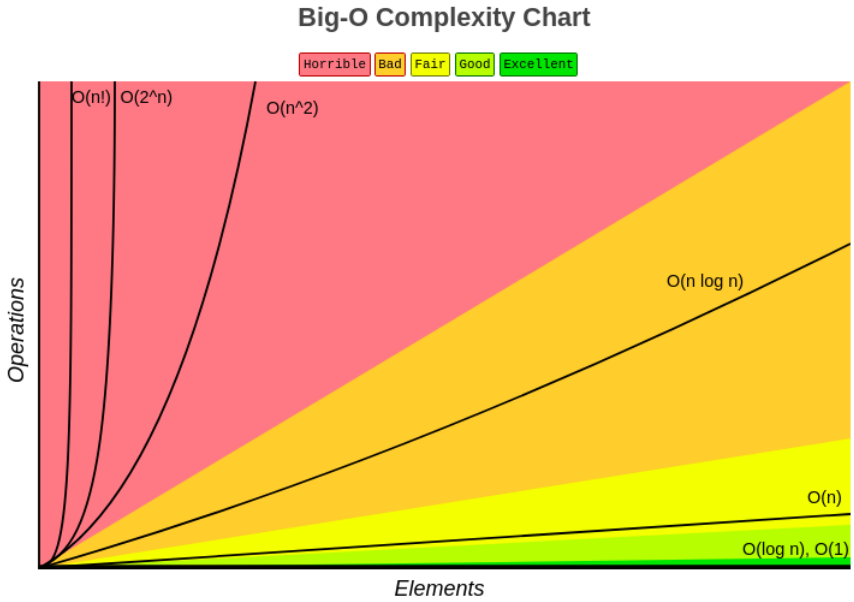

Ou seja, a função $h$ atua como um limite superior para valores assintóticos de $f$. Exemplos:

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

É apresentado na Figura 3.2 um gráfico contendo diferentes complexidades de tempo considerando uma entrada $n$, utilizando uma escala de cores. Veja nesta figura que a menor complexidade de tempo é a constante $O(1)$ e a pior é a combinatória $O(n!)$. Nesta figura também pode-se perceber que, por exemplo, um algoritmo com complexidade $O(nlogn)$ resolveria um problema com muito menos operações que um algoritmo com complexidade $O(n^2)$, para uma entrada de mesmo tamanho, o que impacta diretamente no tempo de resposta. Desta maneira, pode-se dizer que $O(nlogn)$ possui complexidade menor que $O(n^2)$.

Existem outras notações igualmente importantes mas menos utilizadas que são a notação $\Theta$ utilizada para limites apertados assintóticos e a notação $\Omega$ para limites inferiores assintóticos.

Figura 3.2: Gráfico da notação Big O




Big-O Complexity Chart: http://bigocheatsheet.com

# QUESTÕES

***Questão 1 (IFPI - IF-PI - 2016):.*** Considere o código-fonte que segue:

```
int f1(int n) {
if (n == 0 || n == 1) return n;
else return (2 * f1(n-1) + 3 * f1(n-2));
    }

int f2(int n) {
int a; int[] X = new int [n];
int[] X = new int [n]; int[] Z = new int
    [n];
X [0] = Y [0] = Z [0] = 0;
X [1] = 1; Y [1] = 2; Z [1] = 3;
for (a = 2; a <= n; a ++) {
X [a] = Y [a-1] + Z [a-2];
Y [a] = 2 * X [a]; Z [a] = 3 * X [a]; }
return X [n]; }
```

Qual é o tempo de execução de f1(n) e f2(n), respectivamente?
A) O(2n) e O(2n).
B) O(n) e O(2n).
C) O(2n) e O(n).
D) O(n) e O(n).
E) O(n) e O(log n).

***Questão 2 (CETRO - AMAZUL - 2015):.*** É correto afirmar que a complexidade assintótica de algoritmos é usada
A) quando são desprezados determinados tempos da função ou quando somente aproximações da função são possíveis de se obter.
B) quando é possível determinar todos os tempos de uma função.
C) quando uma função é extremamente complexa, mas é possível calcular os seus tempos.
D) somente quando uma função é logarítmica.
E) somente quando a linguagem de programação utilizada for C++.

***Questão 3 (INSTITUTO AOCP - UFFS - 2016):.*** Na análise de complexidade de algoritmos, em que o interesse é restrito a valores assintóticos e se desconsidera as constantes multiplicativas e aditivas, qual é o número de passos a ser considerado na expressão $2(n^2 - 1) + 10n^3$?
A) $2(n^2 - 1)$
B) $10$
C) $2n$
D) $n^3$
E) $n^2 - 1$

***Questão 4 (INSTITUTO AOCP - UFOB - 2018):.*** Um algoritmo de complexidade $nlogn$ é mais complexo que um algoritmo de complexidade $n^2$.
A) Certo
B) Errado

***Questão 5 (COMPERVE - UFRN - 2018):.*** Considere o código representado na figura abaixo.

```
void funcao(int* vet, int n) {
   int i, j, temp;
   for(i = 0; i < n - 1; i++){
      for(j = (i+1); j < n; j++){
         if(vet[j] < vet[i]){
            temp = vet[i];
            vet[i] = vet[j];
            vet[j] = temp;
         }
      }
   }
}
```

O algoritmo apresenta complexidade no pior caso de
A) $O(n)$
B) $O(n^2)$
C) $O(nlogn)$
D) $O(n^4)$

# Capítulo 4: Ordenação de dados

O objetivo deste capítulo é apresentar conceitos e algoritmos de ordenação de dados, ferramental este de muita importância para praticamente todos os ramos da computação. É importante o conhecimento das técnicas de ordenação para que seja possível a escolha das melhores alternativas, sempre considerando o tamanho aproximado do conjunto alvo do processo. Antes de mais nada, é necessária a apresentação dos conceitos básicos relacionados ao assunto.

## Conceitos Básicos

Em diversas aplicações, dados devem ser armazenados de acordo com uma determinada ordem, sendo que alguns algoritmos podem explorar a ordenação dos dados para operar de maneira mais eficiente. Desta maneira, têm-se basicamente duas alternativas: ou (1) se insere os elementos na estrutura de dados respeitando a ordenação ou (2), a partir de um conjunto de dados já criado, aplica-se um algoritmo para ordenar seus elementos. Devido ao seu uso muito frequente, é importante ter à disposição diferentes algoritmos de ordenação que sejam eficientes em termos de tempo, ou seja, devem ser rápidos e também eficientes em termos de espaço, ocupando pouca memória durante a execução.

Os algoritmos de ordenação podem ser aplicados a qualquer informação desde que se possa estabelecer uma ordem definida entre os elementos. Pode-se, por exemplo, ordenar um vetor de inteiros que adote uma ordem crescente ou decrescente como critério. Também é possível aplicar algoritmos de ordenação em vetores responsáveis por guardar informações que sejam mais complexas, como um vetor que armazena os dados pertencentes aos alunos de uma turma, como nome, número de matrícula etc. Considerando este último cenário, por exemplo, a ordem entre os elementos precisa ser definida usando uma das informações do aluno com chave da ordenação: alunos ordenados pelo nome, ou alunos ordenados pelo número de matrícula, etc.

Quando se descreve ou compara algoritmos de ordenação, alguns conceitos podem ser utilizados:

**Estabilidade:** Um algoritmo de ordenação estável mantém a mesma ordem nos vetores não ordenado e ordenado, considerando elementos cujas chaves sejam comparativamente iguais. Por exemplo, se um vetor com dados de pessoas for ordenado pela idade, pessoas com a mesma idade permanecerão na mesma ordem que estavam no vetor original.

**Optimalidade:** Optimalidade está relacionada com a complexidade dos algoritmos. Um algoritmo ótimo possui a menor complexidade possível para uma determinada classe de problemas. Genericamente, algoritmos ótimos para ordenação possuem

complexidade $O(nlogn)$, ou seja, não se pode ordenar dados com uma complexidade de pior caso menor que isto.

**Ordenação *in loco*:** Algoritmos de ordenação *in loco* operam do próprio vetor original de dados. Algoritmos que não são *in loco* precisam de memória adicional para processar os dados.

A seguir, serão apresentados os principais algoritmos de ordenação existentes.

# Ordenação Bolha

Também conhecido como *Bubble sort*, a ideia deste algoritmo é fazer uma série de comparações entre os elementos do vetor, sempre utilizando algum critério. Quando dois elementos estão fora de ordem, então existe uma inversão, e esses dois elementos são trocados de posição, ficando em ordem correta. Assim, o primeiro elemento é comparado com o segundo. Se uma inversão for encontrada, a troca é feita. Em seguida, independente se houve ou não troca após a primeira comparação, o segundo elemento é comparado como terceiro, e, caso uma inversão seja encontrada, a troca é feita. O processo continua até que o penúltimo elemento seja comparado com o último, então com esse processo, garante-se que o elemento de maior valor do vetor seja movido para a última posição. A ordenação continua com o posicionamento do segundo maior elemento. do terceiro e etc. A ideia fundamental é que a cada rodada completa, um elemento é movido (flutua com uma bolha) para sua posição final. Se for utilizando um critério crescente de ordenação, na primeira rodada o maior elemento irá para a última posição, na segunda rodada, o segundo maior irá para a penúltima posição, e assim sucessivamente.

Na Figura 4.1 é apresentado um exemplo de uma primeira rodada do algoritmo de ordenação bolha sobre um conjunto. Neste exemplo o elemento 93 "flutua"para a posição final. Em uma rodada posterior, o algoritmo poderia ignorar a última posição, pois o elemento que está ali não precisaria mais ser alterado.

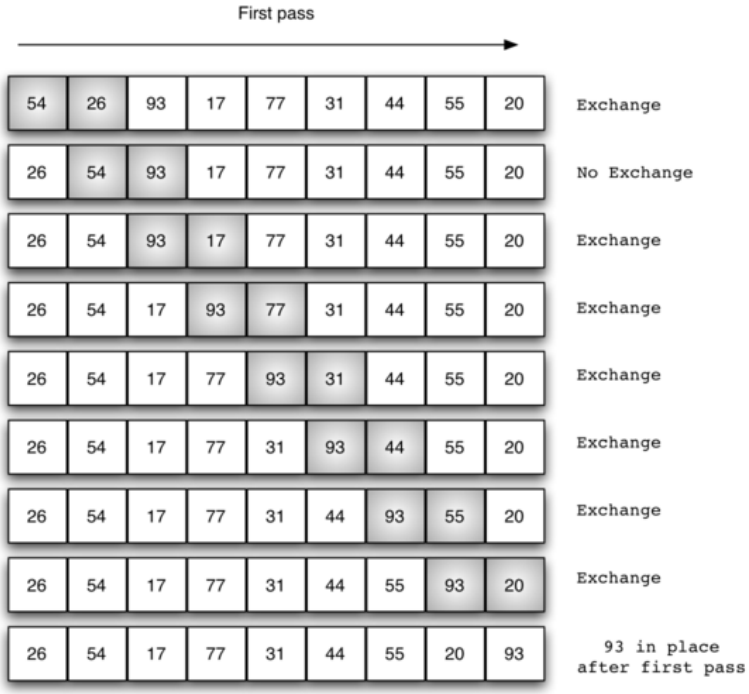
A ordenação bolha é apresentada no Algoritmo 6. Neste algoritmo, o *for* mais externo (linha 2) serve para definir qual elemento irá para a sua posição final, sendo as comparações feitas utilizando o *for* mais interno (linha 3). Neste algoritmo, sempre que dois elementos estão fora de ordem (linha 4), eles devem ser trocados (linha 5, 6 e 7).

A complexidade de pior caso deste algoritmo é $O(n^2)$, sendo também a de caso médio $O(n^2)$. A complexidade de melhor caso também é $O(n^2)$, mas otimizando o algoritmo consegue-se chegar a $O(n)$. As vantagens dele são a simplicidade, é estável e faz ordenação *in loco*. Como desvantagem, pode-se dizer que é um dos mais lentos.

# Ordenação por Inserção

A ordenação por inserção (também conhecido como *Insertion sort*) segue uma ideia semelhante a organização de cartas de baralho na mão de um jogador. No início, parte-se de um sub-vetor ordenado, que no momento inicial possui somente um elemento (o primeiro), o que equivaleria a primeira carta na analogia do jogador. Em

Figura 4.1: Exemplo de Ordenação pelo método Bolha



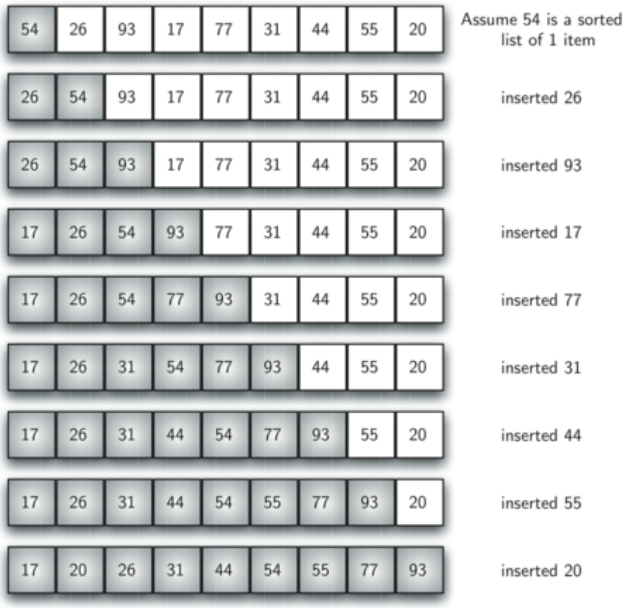

Fonte: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html

**Algoritmo 6** Algoritmo de ordenação bolha

```
1:  procedure ORDENACAO_BOLHA(A[], tamanho)
2:      for i de 0 e i < tamanho − 1 do
3:          for j de 0 e j < tamanho − i − 1 do
4:              if A[j] > A[j + 1] then
5:                  temp ← A[j]
6:                  A[j] ← A[j + 1]
7:                  A[j + 1] ← temp
8:              end if
9:          end for
10:     end for
11: end procedure
```

seguida, deve-se adicionar o segundo elemento ao sub-vetor ordenado, podendo haver deslocamento para abertura de espaço, o que equivaleria a inserção da segunda carta na mão do jogador, na analogia considerada. Neste momento tem-se um sub-vetor ordenado com 2 elementos. Em seguida, efetua-se a inserção do terceiro, quarto, até que todos os elementos tenham sido inseridos no sub-vetor. Na Figura 4.2 é possível visualizar um exemplo de aplicação do algoritmo de ordenação sobre um conjunto de dados numéricos. Neste exemplo, como caso inicial, apenas o sub-vetor com o elemento 54 é considerado, sendo que o primeiro elemento a ser inserido neste é o 26, o que exige o deslocamento do 54 para frente. O próximo elemento a ser inserido é o 93, o qual permanece em sua posição porque é maior que o 54. Seguindo este processo, todos os elementos são inseridos até que se tenha um vetor final ordenado.

Figura 4.2: Exemplo de Ordenação por Inserção




Fonte: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html

O algoritmo de ordenação por inserção é apresentado no Algoritmo 7. Neste algoritmo, o *for* mais externo seleciona um elemento por vez para ser inserido no sub-vetor ordenado (elemento chamado de eleito, começando do segundo elemento do vetor). A seguir, o *while* da linha 5 desloca os elementos do sub-vetor, enquanto estes forem maiores que o elemento eleito ou chegue-se ao início do vetor em questão. Este processo "abre"espaço no sub-vetor para a inserção do elemento eleito, o que ocorre na linha 9.

**Algoritmo 7** Algoritmo de ordenação por inserção

```
1:  procedure ORDENACAO_POR_INSERCAO(A[], tamanho)
2:      for i de 1 e i < tamanho do
3:          eleito ← A[i]
4:          j ← i − 1
5:          while j ≥ 0 e eleito < A[j] do
6:              A[j + 1] ← A[j]
7:              j ← j − 1
8:          end while
9:          A[j + 1] ← eleito
10:     end for
11: end procedure
```

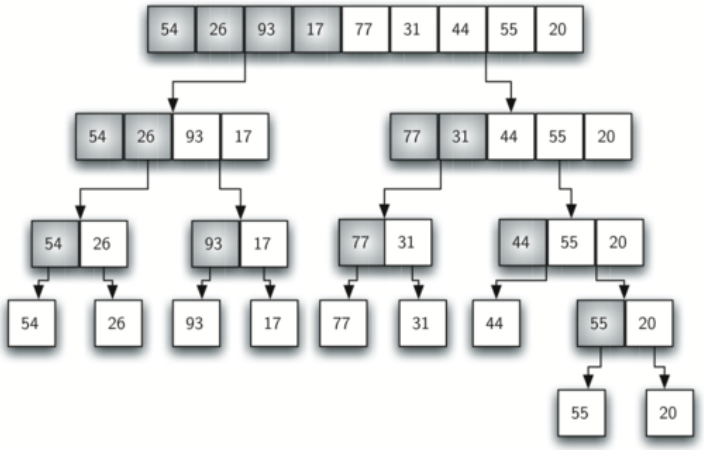
A complexidade de pior caso deste algoritmo é $O(n^2)$, sendo também a de caso médio $O(n^2)$. A complexidade de melhor caso é $O(n)$. As vantagens dele são a simplicidade, a estabilidade e o fato de fazer ordenação *in loco*. Outra vantagem é que se o vetor está encontra-se ordenado, o algoritmo termina em $O(n)$. Como desvantagem, pode-se dizer que é lento, embora muito mais rápido que o bolha, por apenas deslocar elementos e

não fazer trocas.

# Ordenação por Intercalação

Este algoritmo, também chamado de *Merge sort* opera pelo método de *divisão e conquista*, onde divide-se o problema em vários sub-problemas (sub-vetores) e resolve-se esses sub-problemas de maneira recursiva. Após a resolução dos sub-problemas ocorre a conquista, que é a união das resoluções dos sub-problemas (intercalação). Para entender melhor este processo, considera-se o exemplo da Figura 4.3. Neste exemplo, o vetor é inicialmente dividido ao meio, sendo que cada metade é recursivamente dividida também, até se chegar a vetores unitários. Note que, a partir de um vetor desordenado de $n$ elementos chega-se a $n$ vetores unitários (ordenados, pois contém um único elemento). Após a fase de divisão, ocorre a conquista ou combinação dos resultados obtidos, conforme pode-se observar na Figura 4.4. Nesta Figura, o mesmo exemplo é apresentado sob a perspectiva da combinação (ou conquista) dos resultados. Neste exemplo, vetores unitários são combinados (intercalados) para formar, em um primeiro momento, sub-vetores de até dois elementos (ordenados). Em seguida, sub-vetores de até dois elementos são intercalados para se obter sub-vetores de até quatro elementos, e assim sucessivamente.

Figura 4.3: Exemplo de ordenação pelo Método da Intercalação (fase da divisão)

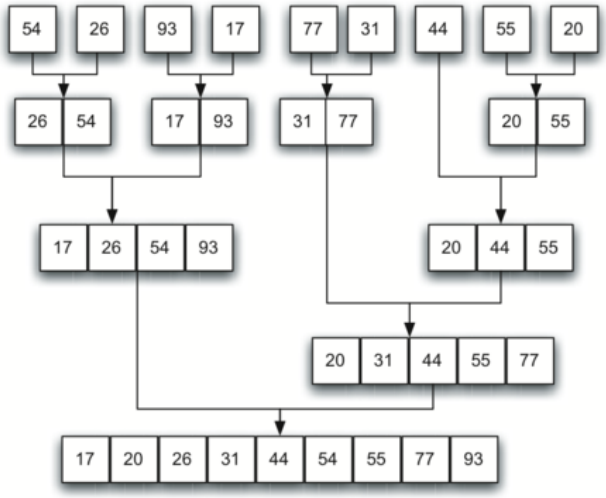



http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

O algoritmo de ordenação por intercalação é apresentado no Algoritmo 8. No algoritmo, os parâmetros são o vetor original, um vetor auxiliar e os índices inicial e final. Neste algoritmo, o vetor é dividido ao meio (linha 3), sendo que cada metade é ordenada recursivamente (linhas 4 e 5), sendo os resultados intercalados pela linha 6. De fato, o algoritmo por si só é simples, sendo a parte da intercalação a mais trabalhosa, conforme mostrado no Algoritmo 9.

A ideia deste segundo algoritmo (Algoritmo 9) é, a partir de dois sub-vetores ordenados, um que começa em esquerda e termina em meio e outro que inicia após o meio e termina em direita, produzir um vetor ordenado da esquerda até a direita. Como

Figura 4.4: Exemplo de ordenação pelo Método da Intercalação (fase da conquista)




http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

**Algoritmo 8** Algoritmo de ordenação por intercalacao

```
1: procedure ORDENACAO_POR_INTERCALACAO(A[], Aux[], inicial, final)
2:     if inicial < final then
3:         meio ← (inicial + final)/2
4:         ordenacao_por_intercalacao(A, Aux, inicial, meio)
5:         ordenacao_por_intercalacao(A, Aux, meio + 1, final)
6:         intercala(A, Aux, inicial, meio, final)
7:     end if
8: end procedure
```

não se pode intercalar dois sub-vetores em um vetor *in-loco*, sem perda de elementos armazenados, é utilizado o vetor auxiliar *Aux*. No algoritmo em questão, é copiado (a cada iteração) para o vetor auxiliar o menor elemento considerando os elementos dos sub-vetores, conforme as linhas de 5 a 13. Se algum sub-vetor, seja o da esquerda ou o da direita, possuir ainda elementos ao final deste processo, então estes deverão ser integralmente copiados para o vetor auxiliar, pela execução das linhas de 15 a 19 OU da 20 a 24. Ao final, basta transferir os elementos do vetor auxiliar novamente para o vetor principal, conforme linhas de 25 a 28.

Este algoritmo possuir complexidade de pior caso de $O(n\log n)$, sendo a complexidade caso médio $O(n\log n)$ e a de melhor caso: $O(n\log n)$. Como vantagem, pode-se dizer que este algoritmo é estável e ***ótimo***. Dizer que este algoritmo é ótimo implica dizer que nenhum outro algoritmo de ordenação conseguirá ordenar de maneira assintóticamente melhor, ou seja, $O(n\log n)$ é o limite de pior caso para ordenação. Como desvantagem, este algoritmo precisa de armazenamento auxiliar, sendo desaconselhável para vetores pequenos, pois mostra o seu valor para conjuntos grandes de dados.

**Algoritmo 9** Algoritmo para intercalação

```
1:  procedure INTERCALA(A[], Aux[], inicial, meio, final)
2:      esquerda ← inicial
3:      direita ← meio + 1
4:      posicao_final ← inicial
5:      while (esquerda ≤ meio) e (direita ≤ final) do
6:          if A[esquerda] ≤ A[direita] then
7:              Aux[posicao_final] ← A[esquerda]
8:              esquerda ← esquerda + 1
9:          else
10:             Aux[posicao_final] ← vetor[direita]
11:             direita ← direita + 1
12:         end if
13:         posicao_final ← posicao_final + 1
14:     end while
15:     while esquerda ≤ meio do
16:         aux[posicao_final] ← vetor[esquerda]
17:         posicao_final ← posicao_final + 1
18:         esquerda ← esquerda + 1
19:     end while
20:     while direita ≤ final do
21:         Aux[posicao_final] ← A[direita]
22:         posicao_final ← posicao_final + 1
23:         direita ← direita + 1
24:     end while
25:     while inicial ≤ final do
26:         A[inicial] ← Aux[inicial]
27:         inicial ← inicial + 1
28:     end while
29: end procedure
```

# Ordenação Rápida

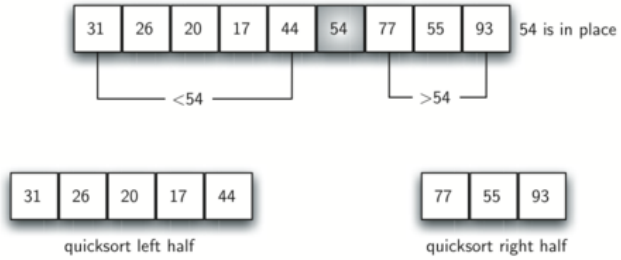
Também conhecido como *Quick sort*, é um algoritmo muito rápido para o caso médio e que foi proposto por Charles Antony Richard Hoare em 1960. Este algoritmo provavelmente é o mais utilizado hoje e possui uma ideia simples: escolher um elemento pivô (arbitrário) e mover todos os elementos menores para antes do pivô e os maiores para depois. Feito isto, o pivô estará em seu lugar final e não precisará ser movido novamente, sendo que antes dele haverá um sub-vetor de elementos menores e depois dele haverá um sub-vetor de elementos maiores. Como último passo, basta repetir o processo recursivamente para as duas partes restantes (sub-vetores).

Como exemplo de aplicação do algoritmo de ordenação rápida, considere o vetor apresentado na Figura 4.5, com a escolha do primeiro elemento como pivô. Na Figura 4.6 é apresentado o resultado do processo de movimentação dos elemento maiores para a direita e dos menores para a esquerda do pivô. Este processo movimentação em relação ao pivô recebe o nome de **particionamento**.

Figura 4.5: Exemplo da ordenação rápida com escolha do pivô


http://interactivepython.org/runestone/static/pythonds/SortSearch/TheQuickSort.html

Figura 4.6: Exemplo da ordenação rápida particionamento entre elementos maiores e menores



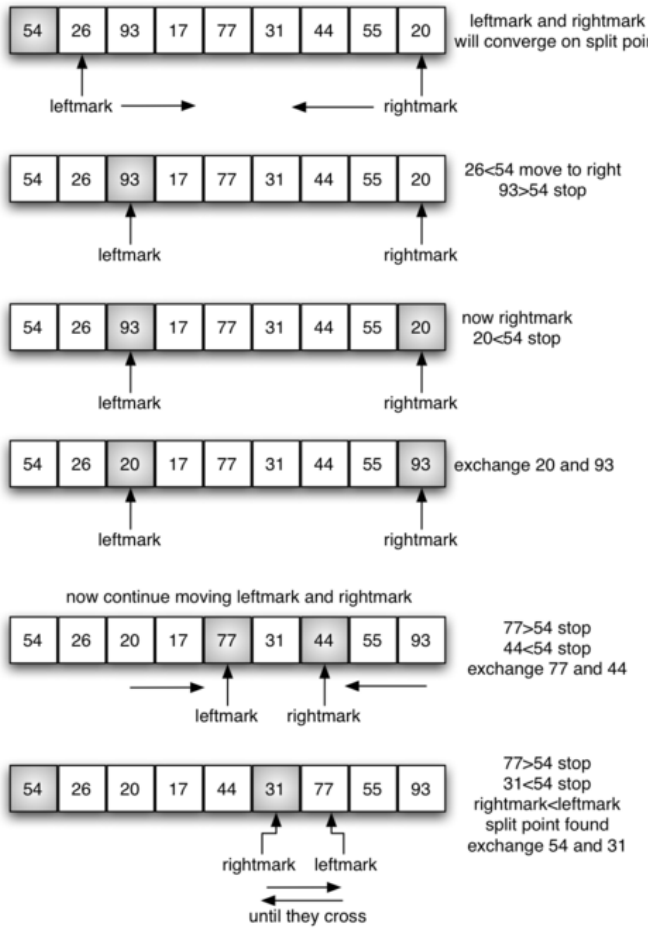

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheQuickSort.html

A processo de particionamento que deu origem ao cenário da Figura 4.6 pode ser visto na Figura 4.7. Este processo é iniciado pela escola do primeiro elemento como pivô (poderia ser outro, mas o processo se torna mais simples com o primeiro). A seguir, são definidos os marcadores da esquerda (uma posição a frente do pivô) e o marcador da direita (última posição do vetor). Em seguida, o marcador da esquerda é deslocado para direita enquanto ele apontar para elementos menores que o pivô. Após isto, o marcador da direita é deslocado para a esquerda enquanto apontar para elementos maiores que o pivô. Terminado este processo, os marcadores apontarão para elementos que estão fora de ordem em relação ao pivô, então eles devem ser trocados como ocorre com o 20 e o 93 na Figura do exemplo. O processo todo continua até os marcadores se cruzarem, sendo o pivô trocado pelo elemento apontado pelo marcador da direita. A partir deste ponto o dois sub-vetores podem ser ordenados recursivamente.

O algoritmo de ordenação rápida pode ser visto no Algoritmo 10. Neste algoritmo, os parâmetros são o vetor e os índices do primeiro e do último elemento. Obviamente, o algoritmo deve ordenar os elemento enquanto o sub-vetor possuir ao menos dois elementos, o que é indicado pela linha 2. A chamada para a função particiona (linha 3) possui como função a escolha de um pivô e a movimentação dos elementos conforme explanado anteriormente. Após o particionamento, os problema é resolvido recursivamente dos dois sub-vetores demarcados no particionamento (linhas 4 e 5).

O algoritmo de particionamento, conforme apresentado no Algoritmo 11 opera da seguinte maneira: primeiramente, o pivô é escolhido como sendo o primeiro elemento (linha 4), sendo os marcadores da direita e esquerda definidos como i e j (linha 2 e 3). A seguir, enquanto os marcadores não se cruzarem (linha 5) o marcador da esquerda avança (linha 7) e o da direita retrocede (linha 17), considerando a relação com o pivô. Quando os marcadores terminarem de se mover, os elemento apontados deverão ser trocados (linhas 12, 13 e 14), sendo estes atualizados para a continuação do algoritmo

Figura 4.7: Exemplo da ordenação rápida e o processo de particionamento




http://interactivepython.org/runestone/static/pythonds/SortSearch/TheQuickSort.html

**Algoritmo 10** Algoritmo de ordenação rápida

```
1: procedure ORDENACAO_RAPIDA(A[], inicial, final)
2:     if inicial < final then
3:         p ← particiona(A, inicial, final)
4:         ordenacao_rapida(A, inicial, p − 1)
5:         ordenacao_rapida(A, p + 1, final)
6:     end if
7: end procedure
```

(linhas 15 e 16). Quando os marcadores se cruzarem, o elemento apontado pelo marcador da direita deverá ser trocado com o pivô (linha 18 e 19). A algoritmo por fim retorna a posição final do pivô representada por j (linha 20).

A complexidade pior caso da ordenação rápida é $O(n^2)$ e ocorre quando o pivô é o maior ou o menor elemento. Já a complexidade de caso médio é de $O(n\log n)$, sendo a complexidade melhor caso $O(n\log n)$. A vantagens deste algoritmo é que ele rápido e eficiente e a desvantagem é que ele não é estável.

**Algoritmo 11** Algoritmo de particionamento da ordenação rápida

```
1:  function PARTICIONA(A[], inicial, final)
2:      i ← inicial + 1
3:      j ← final
4:      pivo ← A[inicial]
5:      while i ≤ j do
6:          while A[i] < pivo do
7:              i ← i + 1
8:          end while
9:          while A[j] > pivo do
10:             j ← j − 1
11:         end while
12:         tmp ← A[i]
13:         A[i] ← A[j]
14:         A[j] ← tmp
15:         i ← i + 1
16:         j ← j − 1
17:     end while
18:     A[inicial] ← A[j]
19:     A[j] ← pivo
20:     return j
21: end function
```

# Resumo dos Algoritmos de Ordenação

Na Tabela 4.1 um resumo dos algoritmos de ordenação apresentados.

Tabela 4.1: Comparativo dos algoritmos de ordenação

| **Algoritmo** | **Pior caso** | **Caso médio** | **Melhor caso** | **Ótimo** | **Estabilidade** | ***In loco*** |
|---|---|---|---|---|---|---|
| **Bolha (*bubble sort*)** | $O(n^2)$ | $O(n^2)$ | $O(n^2)$/$O(n)$[1] | Não | Sim | Sim |
| **Inserção (*inserion sort*)** | $O(n^2)$ | $O(n^2)$ | $O(n)$ | Não | Sim | Sim |
| **Intercalação (*merge sort*)** | $O(n\log n)$ | $O(n\log n)$ | $O(n\log n)$ | Sim | Sim | Não |
| **Rápida (*quick sort*)** | $O(n^2)$ | $O(n\log n)$ | $O(n\log n)$ | Não | Não | Sim |

# Questões

***Questão 1 (UFCG - UFCG - Analista de Tecnologia da Informação - 2019):.*** Sobre o algoritmo de ordenação Quick Sort, escolha a assertiva correta.
A) Foi inventado após 1970.
B) Tem O(n log n) no pior caso.
C) Quick Sort, assim como Bubble Sort, é um algoritmo recomendado apenas para finalidades didáticas, não sendo bem aplicável em ambientes de produção. D) Não é um algoritmo de ordenação estável.
E) Não é um algoritmo que permite paralelismo.

***Questão 2 (FEPESE - Prefeitura de Ituporanga - SC - 2019):.*** Assinale a alternativa a respeito dos algoritmos de ordenação Bubble Sort e Quicksort.
A) Quicksort tem um tempo de execução logarítmico no pior caso.
B) Bubble Sort tem um tempo de execução logarítmico em média.
C) Bubble Sort e Quicksort têm um tempo de execução quadrático no pior caso.
D) O algoritmo Quicksort efetua a ordenação da lista, realizando trocas de ordem sucessivas de elementos subsequentes.
E) Bubble Sort é um algoritmo recursivo que efetua, a cada passo, o particionamento da lista que será ordenada em duas sublistas: uma com os elementos maiores que um elemento escolhido como pivô, e a outra com os elementos maiores que este.

***Questão 3 (SUGEP - UFRPE - 2019):.*** Os algoritmos de ordenação são utilizados para os mais diversos cenários de dados. Apesar de terem o mesmo objetivo (ordenação), possuem diferentes complexidades em relação ao número (n) de elementos a serem ordenados. O "quiksort" se destaca como um dos algoritmos mais rápidos para ordenação. No pior caso, a complexidade "quicksort" será:
A) $O(logn)$
B $O(nlogn)$
C) $O(n)$
D) $O(n^2)$
E) $O(n^3)$

***Questão 4 (CS-UFG - IF Goiano - 2019):.*** Seja o pseudocódigo que segue a definição de um algoritmo para a ordenação de um vetor V de números inteiros, em que o primeiro elemento do vetor está na posição 1 e o último na posição n (n>1).

```
for i=1 to n
   for j=n downto i+1
       if A[j] < A[j-1]
          exchange A[j] with A[j-1]
       end-if
   end-for
end-for
```

Sobre a notação do algoritmo, for-to indica o incremento da variável de controle do laço, for-downto indica o decremento da variável de controle do laço e exchangewith denota a permuta de valores entre duas posições de memória. O pseudocódigo refere-se ao seguinte algoritmo:
A) quicksort.
B) bubblesort.
C) ordenação por inserção.
D) ordenação por seleção

***Questão 5 (FUNRIO - IF-PA - 2016):.*** Quantas comparações e trocas de posição ocorrerão se utilizarmos o algoritmo Bubble Sort para ordenar do menor para o maior valor o vetor [60,32,45,5,6,2], respectivamente:
A) 10 e 4.
B) 15 e 13.
C) 25 e 15.
D) 22 e 12.
E) 12 e 9.

# Capítulo 5: Estruturas em Alocação Sequencial

Nos capítulos anteriores foram apresentados conceitos de recursão, complexidade e algoritmos de ordenação. Neste capítulo serão apresentadas as primeiras estruturas de dados que podem ser utilizadas para o armazenamento de informações. As estruturas que serão estudadas neste capítulo são as listas, filas e pilhas, todas implementadas usando alocação sequencial, ou seja, dados dispostos de maneira contínua na memória.

## Conceitos Básicos

As listas (como genericamente serão mencionadas) são as estruturas de dados primitivas de manipulação mais simples, que representam também uma TAD, ou tipo abstrato de dados. Uma lista linear agrupa informações de elementos que se relacionam entre si, ou seja, do mesmo tipo, como por exemplo informações sobre funcionários de uma empresa, notas de compra, itens de estoque etc.

### Definição de Lista

Uma lista é um conjunto $n \geq 0$ de nós (ou elementos) $x_i, x_2, ..., x_n$, tais que, se $n > 0$, $x_1$ é o primeiro nó, para $1 < k \leq n$, o nó $x_k$ é precedido pelo nó $x_{k-1}$. As operações mais frequentes neste tipo de estrutura são a busca, inclusão e remoção.

Existem alguns casos particulares de lista, que são:

- **Deque** (*double ended queue*): inserções ocorrem somente na extremidade;
- **Pilha**: inserções e remoções em apenas uma extremidade;
- **Fila**: inserção em um estremo e remoção em outro.

Quanto a implementação deste tipo de estrutura, pode-se utilizar para armazenamento de seus nós a *Alocação sequencial de memória*, que é o alvo deste capítulo e a *Alocação encadeada* que será vista o capítulo seguinte, sendo que a escolha entre um tipo de alocação ou outro depende do tipo de operação que será realizada, do número de listas envolvidas e etc. A seguir serão apresentados em detalhes as listas, filas e pilhas usando o tipo de alocação supracitada.

## Listas em alocação sequencial

Seja uma lista linear, onde cada nó é formado por campos, que armazenam características dos elementos (ou simplesmente os dados armazenados). Cada nó possui, em geral, um identificador chamado chave, sendo que os nós podem estar ordenados pelos valores de suas chaves ou não. Considerando o critério de ordenação, pode-se ter *Listas ordenadas* e *Listas não ordenadas*. Como exemplo, considera-se a seguinte lista:

| nó 0 | nó 1 | nó 2 | nó 3 |
|---|---|---|---|

Perceba na lista anterior que cada nó está posicionado imediatamente ao lado no nó anterior na memória, daí a denominação de alocação sequencial. Nesta lista exemplo, cada nó possui os seguintes campos:

| chave | nome | endereço |
|---|---|---|

As listas admitem um conjunto de operações, as quais serão apresentadas nas próximas seções.

## BUSCA

A operação mais básica que pode ser efetuada em uma lista é a busca. A busca consiste em localizar um determinado elemento usando a sua chave. No Algoritmo 12 é apresentada uma ideia para a implementação da referida busca (cabendo outras abordagens). O algoritmo de busca (usando chave x) percorre todas as posições (nós) da lista (0 até L.tamanho-1) verificando de algum nó possui uma chave igual a x. Se o elemento com chave igual a x for encontrado, então a busca é terminada (i recebe tamanho) e esse nó é retornado. Caso nenhum elemento seja encontrado, um valor nulo é retornado. Veja que "tamanho" é um campo ou atributo da lista. A complexidade deste algoritmo é $O(n)$.

**Algoritmo 12** Algoritmo para busca em lista

```
1:  function BUSCA(L[], x)
2:      i ← 0
3:      buscado ← null
4:      while i < L.tamanho do
5:          if L[i].chave = x then
6:              buscado ← L[i]
7:              i ← L.tamanho
8:          else
9:              i ← i + 1
10:         end if
11:     end while
12:     return buscado
13: end function
```

Quando a lista está ordenada pela chave, pode-se efetuar a **busca binária**. Neste tipo de busca, divide-se o espaço de busca ao meio, de maneira iterativa ou recursiva. Isto significa que, a cada passo, metade dos elementos são descartados do espaço de busca. Para entender melhor a ideia, considera-se o seguinte vetor como exemplo (números são ilustrativos, poderia ser qualquer informação):

| 4 | 5 | 70 | 51 | 1 | 25 | 100 | 3 | 6 | 9 |
|---|---|---|---|---|---|---|---|---|---|

Supondo que o número a ser buscado é exatamente o 9, o algoritmo tradicional deverá operar da seguinte maneira:

| it\indice | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **4(não)** | 5 | 70 | 51 | 1 | 25 | 100 | 3 | 6 | 9 |
| **2** | 4 | **5(não)** | 70 | 51 | 1 | 25 | 100 | 3 | 6 | 9 |
| **3** | 4 | 5 | **70(não)** | 51 | 1 | 25 | 100 | 3 | 6 | 9 |
| **4** | 4 | 5 | 70 | **51(não)** | 1 | 25 | 100 | 3 | 6 | 9 |
| **5** | 4 | 5 | 70 | 51 | **1(não)** | 25 | 100 | 3 | 6 | 9 |
| **6** | 4 | 5 | 70 | 51 | 1 | **25(não)** | 100 | 3 | 6 | 9 |
| **7** | 4 | 5 | 70 | 51 | 1 | 25 | **100(não)** | 3 | 6 | 9 |
| **8** | 4 | 5 | 70 | 51 | 1 | 25 | 100 | **3(não)** | 6 | 9 |
| **9** | 4 | 5 | 70 | 51 | 1 | 25 | 100 | 3 | **6(não)** | 9 |
| **10** | 4 | 5 | 70 | 51 | 1 | 25 | 100 | 3 | 6 | **9(achei!)** |

Ou seja, o algoritmo inspeciona[1] cada um dos elementos do vetor para verificar se é o elemento que se procura. Percebe-se também que a complexidade deste algoritmo é $O(n)$, pois no pior caso, o elemento buscado ou está no final, ou nem está presente no vetor acessado.

Uma estratégia interessante que poderia ser aplicada neste exemplo seria a chamada busca binária, cujo objetivo é reduzir a complexidade de pior caso. A premissa de utilização deste algoritmo é quedados estejam ordenados. O algoritmo funciona da seguinte maneira para buscar um elemento x no vetor/lista, onde as posições iniciais e finais são demarcadas por i e j:

1. Verifica-se o elemento central (ou c na posição m = (i+j)/2) do vetor, e dependendo se:
   a) É igual a x, para-se a busca, elemento encontrado.
   b) x < c, então volta-se para o passo 1 considerando agora o intervalo [i, m-1].
   c) x > c, então volta-se para o passo 1 considerando agora o intervalo [m+1, j].

Percebe-se que o algoritmo anterior reduz, a cada iteração, o espaço de busca para metade do que era no passo anterior, daí a sua superioridade. Mas ainda existe uma questão omitida no algoritmo anterior: caso o elemento não seja encontrado, como o algoritmo para? A resposta é simples, quando o tamanho do vetor chegar zero (0), então o passo 1 do algoritmo só deve ser enquanto i <= j ou o elemento encontrado.

Como exemplo, novamente, considera-se os dados anteriores, porém desta vez ordenados:

| 1 | 3 | 4 | 5 | 6 | 9 | 25 | 51 | 70 | 100 |
|---|---|---|---|---|---|---|---|---|---|

[1] Cada linha representa uma iteração e cada coluna um índice de elemento.

Para achar o elemento 9, considerando a seguinte execução:

| [i,j]\indice | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| [0,9] | 1 | 3 | 4 | 5 | 6(não)->[3] | 9 | 25 | 51 | 70 | 100 |
| [5,9] | ~~1~~ | ~~3~~ | ~~4~~ | ~~5~~ | ~~6~~ | 9 | 25 | <-51(não) | ~~70~~ | ~~100~~ |
| [5,6] | ~~1~~ | ~~3~~ | ~~4~~ | ~~5~~ | ~~6~~ | 9(achei!) | 25 | ~~51~~ | ~~70~~ | ~~100~~ |

Na primeira iteração[2], a posição central do vetor é a 4, cujo elemento é o 6, e, como 9 > 6 pode-se descartar todos o elemento entre os índices 0 e 4. Desta maneira, a seta indica que o elemento pode estar na parte direita do vetor (índices entre 5 e 9). O próximo elemento central é o 7, cujo valor é 51, descartando tudo o que está entre 7 e 9. O que resta é o intervalo [5, 6], cujo elemento central (central considerando aritmética inteira) está no índice 5, tendo valor 9, o qual representa o valor buscado.

No exemplo anterior, o algoritmo conseguiu encontrar o elemento em apenas 3 iterações. Agora pode-se tentar procurar um elemento que não está no vetor, por exemplo o 10:

| [i,j]\indice | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| [0,9] | 1 | 3 | 4 | 5 | 6(não)->[5] | 9 | 25 | 51 | 70 | 100 |
| [5,9] | ~~1~~ | ~~3~~ | ~~4~~ | ~~5~~ | ~~6~~ | 9 | 25 | <-51(não) | ~~70~~ | ~~100~~ |
| [5,6] | ~~1~~ | ~~3~~ | ~~4~~ | ~~5~~ | ~~6~~ | 9(não)-> | 25 | ~~51~~ | ~~70~~ | ~~100~~ |
| [6,6] | ~~1~~ | ~~3~~ | ~~4~~ | ~~5~~ | ~~6~~ | ~~9~~ | <-25(não) | ~~51~~ | ~~70~~ | ~~100~~ |
| [6,5] (fim) | ~~1~~ | ~~3~~ | ~~4~~ | ~~5~~ | ~~6~~ | ~~9~~ | ~~25~~ | ~~51~~ | ~~70~~ | ~~100~~ |

Na execução do algoritmo anterior, o elemento não foi encontrado e o mesmo parou no intervalo [6,5] (i > j).

Nos Algoritmos 13 e 14 são apresentados os algoritmos para a busca binária, de maneira recursiva e iterativa, respetivamente.

A grande vantagem da busca binária está em sua complexidade, que é $O(log_2 n)$. Ou seja, a cada iteração eliminamos de fato metade do que havia na iteração anterior. A seguinte tabela exemplifica o número de comparações efetuadas pela busca binária e busca sequencial:

| Número de elementos | Comparações na Binária | Comparações na Seq. |
|---|---|---|
| 100 | 7 | 100 |
| 1000 | 10 | 1000 |
| 10000 | 14 | 10000 |
| 100000 | 17 | 100000 |

[2]Cada linha representa uma iteração e cada coluna um índice de elemento. A seta indica onde o elemento vai ser procurado na próxima iteração, considerando que este elemento está no meio.

**Algoritmo 13** Algoritmo para busca binária em lista iterativo

```
1: function BUSCA_BIN_IT(L[], x)
2:     i ← 0
3:     j ← L.tamanho − 1
4:     while i ≤ j do
5:         meio ← (i + j)/2
6:         if L[meio].chave = x then
7:             return L[meio]
8:         else
9:             if x > L[meio].chave then
10:                i ← meio + 1
11:            else
12:                j ← meio − 1
13:            end if
14:        end if
15:    end while
16:    return NULL
17: end function
```

**Algoritmo 14** Algoritmo para busca binária em lista recursivo

```
1: function BUSCA_BIN_REC(L[], x, i, j)
2:     meio ← (i + j)/2
3:     if L[meio].chave = x then
4:         return L[meio]
5:     else
6:         if i ≥ j then
7:             return NULL
8:         else
9:             if x < L[meio].chave then
10:                return busca_bin_rec(L, x, i, meio − 1)
11:            else
12:                return busca_bin_rec(L, x, meio + 1, j)
13:            end if
14:        end if
15:    end if
16:    return NULL
17: end function
```

É mostrado na tabela anterior que existe uma grande vantagem da binária sobre a sequencial. Entretanto, ela só deve ser utilizada em situações onde os dados já estiverem ordenados, ou precisarem ser ordenados uma única vez para diversas buscas posteriores, de outra maneira, o custo de ordenação ($O(nlog2n)$ no melhor caso) irá majorar sobre o custo de busca.

# Problema Prático 1

Considere o seguinte vetor:

```
int vetor[] = {300, 2270, 9350, 5860, 3220, 650, 6720,
8620, 990, 4520,  8780, 420, 1000, 2400, 9930, 540,
5160, 6300, 140, 6890, 8240, 2050, 70, 9900,  9630,
2280, 6000, 8840, 4130, 1350, 7440, 150, 3340, 3670,
4550, 70, 7730,  9820, 4110, 5590, 4030, 5420, 4960,
7130, 2620, 5950, 5020, 4680, 7460, 3290,  1010, 9560,
2670, 4460, 1210, 8140, 2790, 1030, 650, 1330, 5890,
8120, 3380,  5080, 2750, 50, 3440, 9750, 6510, 280,
4980, 3180, 9050,  800, 2450, 9720, 4780, 2510, 8320,
6800, 2040, 1800, 7330, 8300, 6790, 1190, 3520, 780,
3560,  5710, 3150, 3950, 9600, 910, 6460, 1030, 8200,
9000, 660, 7820};
```

Faça o seguinte:

1. Escreva um programa em C que declare o vetor anterior como variável global.
2. Usando quick sort, ordene o vetor.
3. Implemente a busca binária (iterativa ou recursiva - sua escolha).
4. Adapte o algoritmo anterior para contar o número de comparações efetuadas.
5. Solicite números para o usuário usuário, faça a busca e apresente as informações obtidas (se foi encontrado e em quantas comparações).

## Inserção

O algoritmo de inserção na lista (Algoritmo 15) opera da seguinte maneira: um elemento é inserido na lista desde que não esteja ainda lá (sem duplicatas). Perceba isto pela busca que é efetuada antes da inserção propriamente dita. Caso um elemento puder ser inserido, ele é posicionado ao final da lista e seu nó de armazenamento retornado. Caso contrario, nulo é retornado Complexidade: $O(n)$: pense na justificativa.

**Algoritmo 15** Algoritmo para inserção em lista

```
1: function INSERE(L[], x)
2:     if busca(L, x) = NULL then
3:         posicao ← L.tamanho
4:         L[posicao].chave ← x
5:         L.tamanho ← L.tamanho + 1
6:         return L[posicao]
7:     end if
8:     return NULL
9: end function
```

## Remoção

A remoção funciona da seguinte maneira: primeiro, o índice do elemento a ser removido descoberto (função procurar_indice, semelhante a busca, mas que retorna um índice – omitido pela simplicidade). Se existir um índice válido (elemento existe), todos os elementos existentes após este elemento são deslocados uma posição para frente (lembra do ordenação por inserção? É o inverso). Ou seja. O "remove" sobrescreve o elemento a ser removido pelo deslocamento dos sucessores. Caso o elemento puder ser removido, o tamanho também precisará ser ajustado (linha 7). Complexidade: $O(n)$: pense na justificativa.

**Algoritmo 16** Algoritmo para remoção em lista

```
1: function REMOVE(L[], x)
2:     indice ← procurar_indice(L, x)        ▷ Auxiliar para obter a posicao do elemento
3:     if indice ≥ 0 then
4:         for i de indice ate L.tamanho − 2 do
5:             L[i] ← L[i + 1]
6:         end for
7:         L.tamanho ← L.tamanho − 1
8:     end if
9: end function
```

# Problema Prático 2

Considere o exemplo de lista em alocação sequencial fornecido com este material (Lista). Este exemplo apresenta parte da implementação de uma lista simples para o armazenamento de números inteiros. Os tipos definidos para esta lista são (lista.h):

```c
typedef struct No{
  // exemplo usando chave inteira
  int chave;

} No;

typedef struct Lista{
  No* elementos; // elemento que esta na lista
  int tamanho; // tamanho atual da lista
  int limite; // limite da lista
} Lista;
```

Todas as operações da lista que já estão implementadas estão comentadas com **//Pronto**, enquanto as que precisam de implementação estão comentadas com **//Implementar**. Desta maneira, você deverá fornecer uma implementação funcional para estas operações, sem alterar outras operações e as structs da lista. A implementação deverá ser enviada ao professor.

Nas listas genéricas, como as que foram estudadas anteriormente, considera-se apenas que o primeiro elementos esta na posicao "0" e qualquer elemento $x_k$ e precedido pelo elemento $x_{k1}$ (elementos também chamados de nós). Em listas especiais, pode-se usar indicadores para acessar posições definidas e é justamente estes indicadores e sua unicidade que definem as estruturas de dados que serão estudadas.

Embora a existência destas restrições pareça dificultar a implementação, tais estruturas são, na verdade, muito mais simples que as listas genéricas. Todas as estruturas discutidas aqui apresentam implementação em bibliotecas de linguagens de alto nível, dada a sua recorrente utilização. A seguir, tais estruturas serão apresentadas.

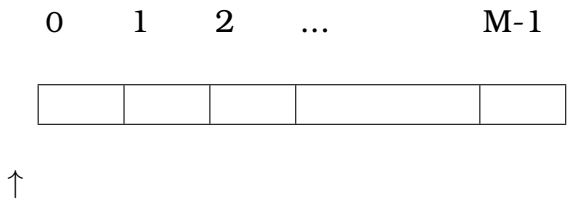
# PILHA

Pilhas ou *stacks* são estruturas para armazenamento de dados cujas únicas operações obrigatórias são empilhar (mais conhecida como *push*) e desempilhar (mais conhecida como *pop*). Esta estrutura não fornece ao programador a flexibilidade de escolher qual elemento remover e onde inserir um determinado dado. Isto acontece pois a pilha é um tipo de lista com apenas uma posição (ou indicador/índice) visível: o topo. Todas a inserções e remoções sao efetuadas na extremidade da pilha denominada topo. Uma pilha pode parecer uma estrutura simples, mas sua aplicabilidade é imensa na computação:

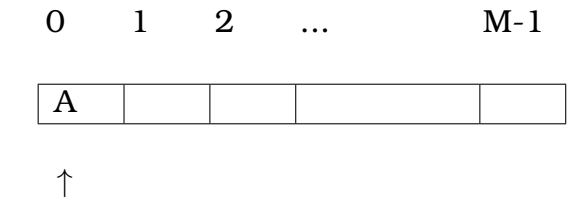
- Em compiladores (tabelas de símbolos com escopo, reconhecedores gramaticais/autômatos de pilha).
- Maquinas virtuais de pilha, como a máquina virtual Java (exceto a do Android).
- Pilha de chamada de procedimentos de qualquer linguagem.
- Notação Polonesa: - * AB/CD (equivalente a: A * B - C/D).
- Notação Polonesa reversa (ou inversa ou RPN[3]): AB * CD/- (também equivalente a: A * B - D / C ) etc.

Pilhas também são conhecidas como estruturas LIFO ou *Last In First Out*, pois o último elemento a entrar será o primeiro a sair. A seguir, um exemplo de uma pilha implementada como um vetor (similar a lista vista anteriormente). Neste exemplo, o índice topo aponta para o topo da pilha e quando esta encontra-se vazia, topo aponta para -1, sendo M o número máximo de elementos que podem ser armazenados:

- Situação inicial: pilha vazia (topo indicado por ↑):

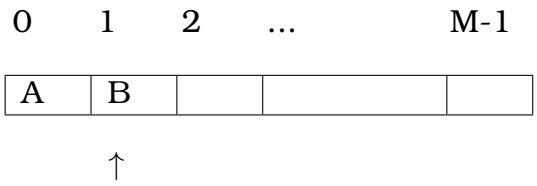



- Inserindo informação A:

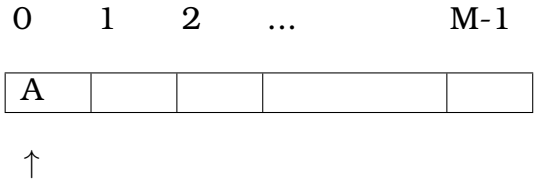



- Inserindo informação B:

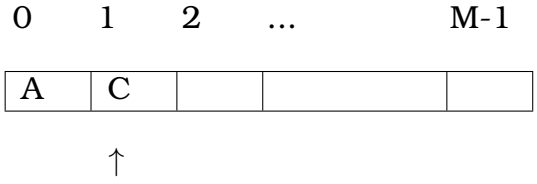
[3]Esta notação é utilizada em algumas calculadoras financeiras, como HP-12C.



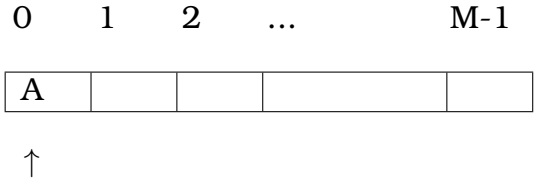

- Retirando informação B:




- Inserindo informação C:




- Retirando informação C:




Veja que no exemplo anterior todas as operações são efetuadas no topo da pilha. Salienta-se novamente que quando a pilha tem topo -1, então ela está vazia e quanto topo armazena o valor de M-1 ela está cheia.

A seguir, são apresentados os algoritmos para as operações de *push* e *pop* da pilha. Para os algoritmos, considere que a pilha tem um atributo topo que representa a posição (índice) do topo da pilha e M que representa a capacidade da mesma. Lembrando que estes algoritmos são apenas sugestões.

## Push

O algoritmo de inserção (Algoritmo 17) insere um elemento no topo, caso exista espaço disponível (P.topo $\neq$ M-1). Veja que o topo sempre indica o último elemento que foi inserido. Complexidade: O(1): pense na justificativa.

## Pop

O algoritmo de remoção na pilha (Algoritmo 18) opera da seguinte maneira: um elemento é removido na pilha desde que ela não esteja vazia. Complexidade: O(1): pense na justificativa.

**Algoritmo 17** Algoritmo para a operação de push

```
1: function PUSH(P[], x)
2:     if P.topo ≠ M − 1 then
3:         P.topo ← P.topo + 1
4:         P[P.topo] ← x
5:     else
6:         Overflow
7:     end if
8: end function
```

**Algoritmo 18** Algoritmo para a operação de push

```
1: function POP(P[])
2:     valor ← NULL
3:     if P.topo ≠ −1 then
4:         valor ← P[P.topo]
5:         P.topo ← P.topo − 1
6:     else
7:         Underflow
8:     end if
9:     return valor
10: end function
```

# Problema Prático 3

Considere o exemplo de pilha em alocação sequencial fornecido com este material (Pilha). Este exemplo apresenta parte da implementação de uma pilha para o armazenamento de números inteiros. O tipo definido para esta pilha é (pilha.h):

```
typedef struct Pilha{
    int M;
    int topo;
    int* dados;
}Pilha;
```

Todas as operações da pilha que já estão implementadas estão comentadas com **//Pronto**, enquanto as que precisam de implementação estão comentadas com **//Implementar**. Desta maneira, você deverá fornecer uma implementação funcional para estas operações, sem alterar outras operações e a struct da pilha. A implementação deverá ser enviada ao professor.

# Fila

Filas são estruturas de dados tambem conhecidas como estruturas FIFO - *First in, First out*, pois o primeiro elemento a entrar será o primeiro a sair. Estas estruturas podem ser utilizadas sempre que houver a necessidade de tratamento de dados ou eventos considerando a ordem em que foram gerados/criados. Filas são muito importantes também para alguns algoritmo que envolvem manipulação de árvores e grafos.

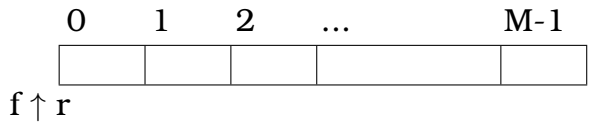
As filas possuem implementação ligeiramente mais complexa que pilha, exigindo dois

indicadores especiais: $f$ (início da fila) e $r$ (fim da fila):

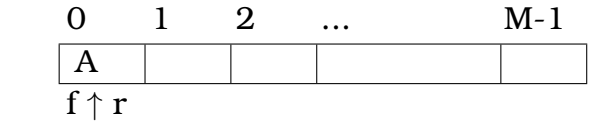
- Para adição, move-se o indicador r. Para remoção, move-se o indicador f.

Os nomes de $f$ (*front*) e $r$ (*reverse*) são apenas convenções. Em muitas implementações estes indicadores podem apresentar nomes como *head* e *tail*, *front* e *end*/*back*, etc. O importante que adições/inserções ocorrem no fim da fila e remoções no começo da fila, ou seja, em extremidades opostas. Considerando que os indicadores $f$ e r apontam ambos para -1 quando a estrutura de dados está vazia, a seguir, um exemplo de uma fila implementada com armazenamento sequencial (vetor):

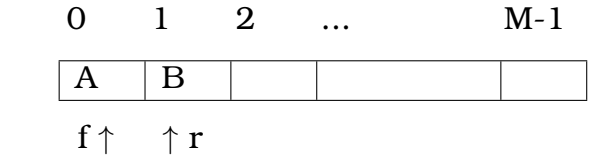
- Situação inicial: fila vazia:



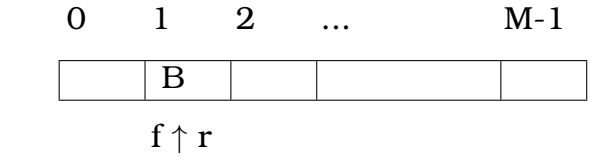

- Inserindo a informação A:



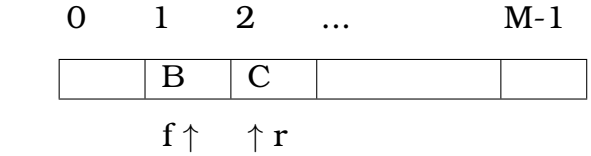

- Inserindo a informação B:



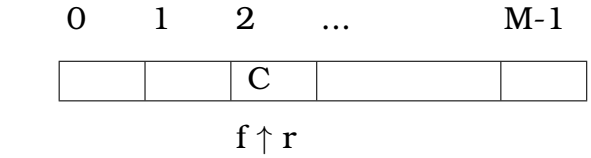

- Removendo a informação A:




- Inserindo a informação C:




- Retirando a informação B:




A seguir são apresentados os algoritmos para as operações de adicionar e remover da fila. Considere, por convenção que a fila tem um atributo **tam** que armazena o total de inseridos na fila e **M** representa a capacidade de fila. Lembrando que estes algoritmos são apenas sugestões.

## Inserção

O algoritmo de inserção (Algoritmo 19) insere um elemento no fim da fila, caso exista espaço disponível. Na linha 4 existe o incremento da posição de inserção (fim da fila), onde o resto da divisão garante que o valor volte para 0 quando chegar em M. Complexidade: $O(1)$, pense na justificativa.

**Algoritmo 19** Operação para adicionar na fila

```
1: function ADICIONAR(F[], x)
2:     if F.tam < M then
3:         F.tam ← F.tam + 1
4:         F.r ← (F.r + 1)Mod   M
5:         F[F.r] ← x
6:     else
7:         Overflow
8:     end if
9: end function
```

## Remoção

O algoritmo de remoção na fila (Algoritmo 20) opera da seguinte maneira: um elemento é removido na fila desde que ela não esteja vazia (F.tam $>$ 0). Além do mais, o elemento sempre é removido da frente da fila. Na linha 5 existe o incremento da posição de remoção (início da fila), onde o resto da divisão garante que o índice volte para 0 quando chegar em M. Complexidade: $O(1)$, pense na justificativa.

**Algoritmo 20** Operação para remover da fila

```
1: function REMOVER(F[])
2:     valor ← NULL
3:     if F.tam > 0 then
4:         F.tam ← F.tam − 1
5:         F.f ← (F.f + 1)Mod   M
6:         valor ← F[F.f]
7:     else
8:         Underflow
9:     end if
10:    return valor
11: end function
```

# Problema Prático 4

Considere o exemplo de fila em alocação sequencial fornecido com este material (Fila). Este exemplo apresenta parte da implementação de uma fila para o armazenamento de números inteiros. O tipo definido para esta fila é (fila.h):

```
1 typedef struct Fila{
2     int M;
3     int f;
```

```
4       int r;
5       int tamanho;
6       int* dados;
7   }Fila;
```

Todas as operações da fila que já estão implementadas estão comentadas com **//Pronto**, enquanto as que precisam de implementação estão comentadas com **//Implementar**. Desta maneira, você deverá fornecer uma implementação funcional para estas operações, sem alterar outras operações e a struct da fila. A implementação deverá ser enviada ao professor.

# Questões

***Questão 1 (IDECAN - IF-RR - 2020):.*** Em Ciência da Computação, as Estruturas de Dados definem como os dados podem ser organizados, bem como quais operações podem ser realizadas para manipular esses dados. Existe uma estrutura de dados que representa um conjunto ordenado de elementos e cujas operações se baseiam no princípio FIFO (First-In, First-Out), ou seja, o primeiro elemento que entra é o primeiro a sair. Marque a sentença referente à estrutura descrita:
A) Pilha
B) Árvore B
C) Hash
D) Grafo
E) Fila

***Questão 2 (CESPE/CEBRASPE - DETRAN-PA - 2006):.*** No que se refere a programação e estruturas de dados, assinale a opção incorreta.
A) Uma fila é caracterizada pelo conceito de que qualquer elemento pode ser atendido independentemente da ordem de chegada.
B) Uma pilha é caracterizada pelo conceito de que o último elemento que chega é o primeiro a ser atendido.
C) Um ponteiro é um tipo de estrutura de dado que aponta para uma posição de memória em que está o valor do dado em si.
D) Uma fila pode ser implementada utilizando-se o conceito de ponteiro em um algoritmo computacional.

***Questão 3 ( INSTITUTO AOCP - IBGE - 2019):.*** Existem algumas estruturas elementares de dados que implementam diferentes políticas de remoção de elementos. Sabendo disso, assinale a alternativa que apresenta corretamente o nome das estruturas que implementam FIFO (PEPS) e LIFO (UEPS), respectivamente.
A) Vetor e matriz.
B) Lista e pilha.
C) Fila e pilha.
D) Matriz e fila.
E) Lista e fila.

***Questão 4 (CS-UFG - UFG - 2019):.*** A pilha P e a fila F possuem originalmente n elementos cada (n>5), e suas operações são:
**empilha(P, elemento)**: inserir elemento na pilha P;
**desempilha(P)**: remover da pilha P e retornar o elemento removido;
**enfileira(F, elemento)**: inserir elemento na fila F;
**desenfileira(F)**: remover da fila F e retornar o elemento removido.
Seja o pseudocódigo abaixo:

```
para i = 1 até n, faça
    empilha(P, desempilha(P))
    enfileira(F, desenfileira(F))
fim-para
```

Ao final da execução do pseudocódigo, os estados finais de P e F são, respectivamente:
A) elementos em ordem original e elementos em ordem original.
B) elementos em ordem inversa e elementos em ordem inversa.
C) elementos em ordem original e elementos em ordem inversa.
D) elementos em ordem inversa e elementos em ordem original.

***Questão 5 (FAURGS - UFRGS - 2018):.*** A maioria dos softwares de aplicação possui comandos de "Desfazer"e "Refazer". O primeiro desfaz a última operação ou texto digitado, enquanto que, o segundo refaz uma operação ou texto desfeito, conforme sugerem os nomes dos comandos.

Internamente, nos softwares, podem ser usadas duas estruturas de dados que armazenam as sucessivas operações de "Desfazer"e "Refazer", de modo que o próximo "Refazer"sempre recupera o último "Desfazer". Os tipos de estrutura de dados que podem ser usados para "Desfazer"e "Refazer"são, respectivamente:
A) Fila e Fila
B) Fila e Pilha
C) Pilha e Fila
D) Pilha e Pilha
E) Pilha e Fila duplamente encadeada

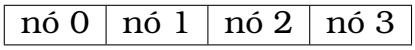
# Capítulo 6: Estruturas Encadeadas

Nas unidades anteriores foram estudadas estruturas com alocação sequencial: listas, pilhas e filas. Todavia, o desempenho dos algoritmos para manipulação destas estruturas em alocação sequencial pode ser bastante fraco, especialmente quando são necessárias movimentações de elementos para inserção e remoção. O objetivo desta parte do estudo é cobrir uma outra estratégia para construção e manipulação de listas e derivadas, que é a alocação e manutenção encadeada de elementos, a qual possui ainda a vantagem de facilmente permitir o crescimento arbitrário dos conjuntos de dados armazenados. A seguir, serão apresentados conceitos básicos necessários para o restante do estudo.

## Conceitos Básicos

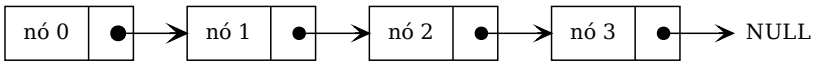
Conforme estudado, as listas em alocação sequencial possuem a característica de, dado um elemento (nó) $j$, ter-se o elemento $j+1$ posicionado de forma adjacente a este na memória. Além disso, o elemento $j=0$ possui posição conhecida. A Figura 6.1 apresenta um exemplo de lista com alocação sequencial. Perceba nesta lista a necessidade a alocação de um único bloco de memória para comportar todos os elementos.

Figura 6.1: Estrutura em alocação sequencial




A Figura 6.2, por outro lado, apresenta como seria a mesma lista da Figura 6.1 mas agora implementada de maneira encadeada. Há vantagens e desvantagens associadas a cada tipo de alocação: alocação encadeada é mais conveniente para tratamento de mais de uma lista (juntar, separar em sub-listas...) e listas grandes onde ocorrem muitas inserções e remoções, pois, como será visto, estas não acarretam deslocamento de elementos. Todavia, a alocação sequencial torna o acesso ao *k*-ésimo elemento é imediato, pois é implementada sobre vetores acessados diretamente pelo índice.

Figura 6.2: Estrutura encadeada

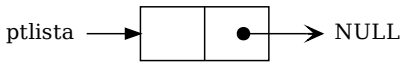



## Listas Encadeadas

Em sua forma mais simples, uma lista encadeada necessita de um ponteiro para indicar a localização do primeiro nó, sendo que qualquer nó possui também um ponteiro para indicar qual é o nó seguinte. Qualquer percurso nesta lista sempre será feito partindo deste ponteiro, então, a ideia é sempre seguir "o próximo nó"(análogo a somar 1 ao índice utilizado para percorrer uma lista alocação sequencial).

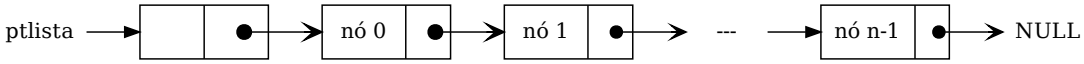
A Figura 6.3 apresenta um exemplo de uma lista encadeada inicialmente vazia. Veja que o nome do ponteiro que armazena o nó de inicio da lista é denominado de **ptlista** e este nome será utilizado sempre para esta finalidade ao longo do texto. Também note neste exemplo que, mesmo a lista estando vazia, existe um nó. Este é um nó especial que não armazena dados da lista e é denominado **nó cabeça**. A utilização de nó cabeça não é obrigatória para implementação de uma lista, mas facilita algumas operações. Pode-se perceber que este nó também é o último da lista, pois o ponteiro que deveria apontar para o próximo nó aponta para **NULL**.

Figura 6.3: Lista vazia




A Figura 6.4 apresenta uma lista completa, composta por $n$ nós e com um nó cabeça, o qual aponta para o primeiro nó útil da lista. Em cada nó poderia estar armazenado qualquer tipo de informação, similar a lista em alocação sequencial.

Figura 6.4: Lista completa com nó cabeça




Supondo que cada nó tenha um campo *info* (poderia ser qualquer outro nome), pode-se fazer um percurso para impressão, conforme apresentado pelo Algoritmo 21. Veja que este algoritmo passa de um nó a outro utilizando o ponteiro **prox**. O processo termina quando **NULL** é encontrado como próximo nó. A complexidade de pior caso deste algoritmo é $O(n)$, pois todos os elementos serão visitados.

**Algoritmo 21** Algoritmo para percorrer lista

```
1: function IMPRIMIR_LISTA(lista)
2:     pont ← lista ↑ .ptlista
3:     while pont ≠ NULL do
4:         imprimir(pont ↑ .info)
5:         pont ← pont ↑ .prox
6:     end while
7: end function
```

Utilizando a ideia de sempre passar para o próximo nó através de ponteiros, agora serão discutidas as principais operações que podem ser feitas em uma lista encadeada. Algumas convenções que serão utilizadas pelos algoritmos que serão apresentados:

- A lista sempre armazenará elementos de maneira ordenada (utilizando a chave), do menor para o maior. Isto significa que tanto a busca como a inserção levarão isto em consideração. Esta medida não remove a generalidade das técnicas, apenas coloca uma restrição que deixa a implementação didaticamente mais interessante.

- A lista não comporta elementos repetidos. Como elementos repetidos considera-se elementos com mesmo valor de chave.

## Busca

A busca consiste em achar o elemento com o valor de chave igual ao valor fornecido. Para este processo, pode-se utilizar o Algoritmo 22. Este algoritmo recebe como parâmetros a *lista*, o valor de chave *x* a ser buscado, uma referência de ponteiro *ant*, e uma referência de ponteiro *pont*. A ideia deste algoritmo é percorrer a lista enquanto não se chegar ao elemento **NULL**. Neste processo, se a chave do elemento atual for menor que a chave fornecida (parâmetro $x$, comparado na linha 6), então o elemento ainda não foi encontrado (lista está ordenada), mas pode estar presente. Entretanto, se o elemento possuir chave igual a $x$ (linha 10), o elemento foi encontrado e a busca pode ser encerrada (linha 12). Se nenhum destes casos acontecer, as chaves dos elementos restantes na lista são maiores que $x$ (linha 13), então a busca terminou sem sucesso (linha 14).

**Algoritmo 22** Algoritmo para busca em lista

```
1:  function BUSCA_ENC(lista, x, ant, pont)
2:      ant ← lista.ptlista
3:      pont ← NULL
4:      ptr ← lista.ptlista ↑ .prox
5:      while ptr ≠ NULL do
6:          if ptr ↑ .chave < x then                    ▷ Pode continuar
7:              ant ← ptr
8:              ptr ← ptr ↑ .prox
9:          else
10:             if ptr ↑ .chave = x then                ▷ Elemento encontrado, pode parar
11:                 pont ← ptr
12:                 ptr ← NULL
13:             else   ▷ Elemento não encontrado, pois a chave é maior que x, pode parar
14:                 ptr ← NULL
15:             end if
16:         end if
17:     end while
18: end function
```

Ao final da execução do algoritmo 22, existem duas possibilidades:

- Item encontrado: *pont* com o nó encontrado, *ant* com o ponteiro do nó anterior ao nó encontrado.

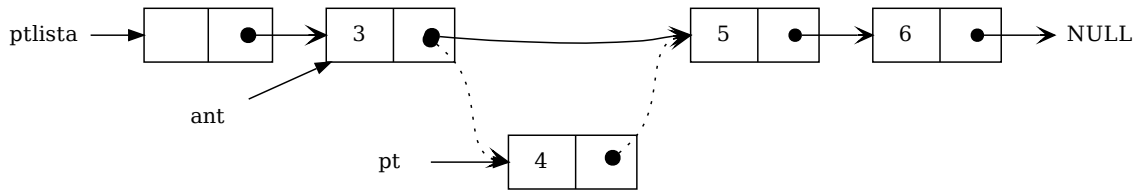
- Item não encontrado: *pont* com **NULL**, *ant* com nó anterior a posição onde o nó deveria estar, caso existisse. Novamente, lembre-se que a lista está ordenada.

Este algoritmo pode parecer complicado a primeira vista, mas ele implementa também detalhes que são importantes tanto para a inserção quanto para a remoção, tornando estas muito mais simples. A complexidade de pior caso deste algoritmo também é $O(n)$, pois no pior caso, o elemento que se procura é o último.

## Inserção

Para inserção de um elemento, considere o exemplo da Figura 6.5. Suponha que neste exemplo existam 3 elementos: um com chave 3, outro com chave 5 e outro com chave 6. Ainda neste exemplo, precisa-se inserir o elemento cuja chave é 4. Se o algoritmo utilizado para a busca for o Algoritmo 22, então, pelo resultado dele têm-se que $pont = NULL$ (elemento não encontrado) e $ant = ponteiro\_no3$. Para inserir o nó com chave 4, basta posicioná-lo como sendo o próximo do $ant$ e fazer com que ele tenha como próximo o $ant \uparrow .prox$, conforme a Figura.

Figura 6.5: Inserção em lista




O Algoritmo 23 sintetiza a inserção: partindo de uma busca (linha 4), caso o elemento não seja encontrado (linha 5), um novo nó é alocado (linha 6), inicializado (linhas 7 e 8) e posicionado entre o $ant$ e o próximo do $ant$ (linhas 9 e 10). A complexidade de pior caso deste algoritmo é $O(n)$, pois ele depende da busca.

## Remoção

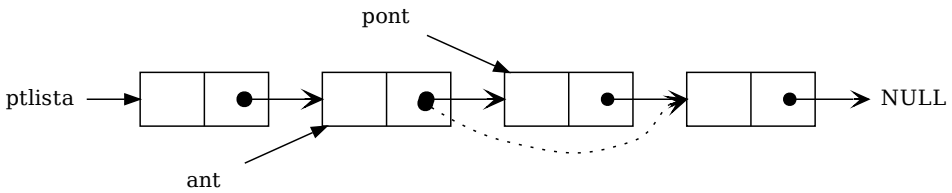
A remoção segue a linha de raciocínio parecida com a da inserção. Considere o exemplo da Figura 6.6. Neste exemplo, o nó a ser removido é o apontado por $pont$, então, basta fazer o próximo do $ant$ apontar para o próximo do $pont$. Feito isso, o nó estará fora da lista.

O Algoritmo 24 sistematiza a ideia de remoção ilustrada no exemplo da Figura 6.6. Neste algoritmo, partindo da busca (linha 4), se o elemento for encontrado (linha 5), este é retirado da sua posição (linha 6) e desalocado (linha 7). A complexidade de pior caso deste algoritmo também é $O(n)$, pois ele depende da busca, que no pior caso não encontrará o elemento.

**Algoritmo 23** Algoritmo para inserção em lista

```
1:  function INSERIR(lista, x, novo_valor)
2:      ant ← NULL
3:      pont ← NULL
4:      busca_enc(lista.ptlista, x, ant, pont)          ▷ Buscar antes de inserir
5:      if pont = NULL then          ▷ Se busca não encontrar, inserir depois do ant
6:          pt ← novo_no
7:          pt↑.info ← novo_valor
8:          pt↑.chave ← x
9:          pt↑.prox ← ant↑.prox
10:         ant↑.prox ← pt
11:     else
12:         Elemento já está na lista
13:     end if
14: end function
```

Figura 6.6: Remoção em lista




**Algoritmo 24** Algoritmo para remoção em lista

```
1:  function REMOVER(lista, x, novo_valor)
2:      ant ← NULL
3:      pont ← NULL
4:      busca_enc(liata.ptlista, x, ant, pont)          ▷ Buscar antes de remover
5:      if pont ≠ NULL then          ▷ Se elemento for encontrado, isolar ele e desalocar
6:          ant↑.prox ← pont↑.prox
7:          desalocar(pont)
8:      else
9:          Elemento não está na lista
10:     end if
11: end function
```

# Problema Prático 1

Considere o exemplo de lista encadeada fornecido com este material (Lista). Este exemplo apresenta a implementação de uma lista encadeada simples para o armazenamento de números inteiros. Os tipos definidos para esta lista são (lista.h):

```
1 typedef struct No{
2   struct No* prox;
```

```
  int chave;

} No;

typedef struct Lista{
  No* ptlista; // noh cabeca
} Lista;
```

Todas as operações da lista já estão implementadas, exceto uma, a que retorna o tamanho da lista, ou seja, o número de elementos inseridos:

```
// retorna o tamanho da lista (numero de elementos inseridos)
int tamanho(Lista* l);
```

Desta maneira, você deverá fornecer uma implementação funcional para esta operação, sem alterar outras operações e as structs da lista. A implementação deverá ser enviada ao professor.

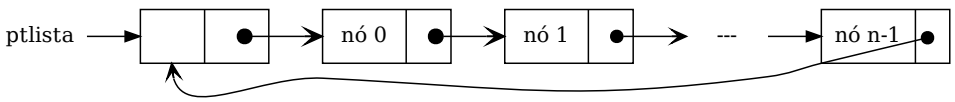
## Outras listas

A título informativo, esta seção apresenta dois outros tipos de lista: a **circular** e a **duplamente encadeada**, as quais podem apresentar vantagens para algumas das operações. Entretanto, não serão fornecidos algoritmos, ficando estes a cargo do aluno caso se interesse. Todavia, a estratégia básica destas listas deve ser entendida.

## Lista Circular

Listas circulares, conforme mostrado no exemplo da Figura 6.7, podem permitir uma busca mais eficiente. Nestas listas, pode-se colocar a chave do elemento procurado no nó cabeça, podendo assim eliminar o teste de fim de lista.

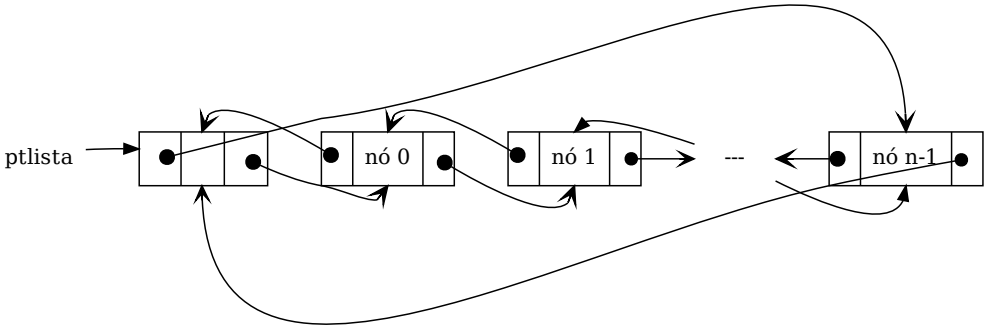
Figura 6.7: Lista circular




## Lista com Encadeamento Duplo

Existe um outra maneira de implementar uma lista encadeada, mas considerando que cada elemento aponta para o seu próximo e seu anterior. Esta alternativa recebe o nome de encadeamento duplo (ou lista duplamente encadeada), cujo exemplo pode ser visto na Figura 6.8. Neste exemplo, o nó cabeça é utilizado tanto para marcar o início quanto o fim da lista, pois esta é implementada de maneira circular (similar a lista anterior). Entretanto, assim como qualquer lista, a utilização de nó cabeça é opcional bem como o fato de ser circular ou não.

Figura 6.8: Lista duplamente encadeada




Este tipo de lista é interessante pois pode-se precisar percorrer a lista nos dois sentidos, então, a presença de um campo adicional (anterior) pode ser justificada pela economia ao não precisar percorrer a lista inteira. Além do mais, algumas operações (especialmente a remoção) podem ser implementadas de maneira mais fácil na presença de encadeamento duplo.

# Pilhas Encadeadas

Estruturas encadeadas podem ser utilizadas também para a implementação de pilhas e filas. No caso de pilhas, operações são simples, primeiro elemento da lista é o topo (sem nó cabeça). Quando a pilha está vazia, então $\rightarrow topo = NULL$. Os Algoritmos 25 e 26 apresentam os algoritmos para inserção e remoção, respectivamente. Note que pilha não usa o conceito de chave previamente utilizado nas listas, então, elementos duplicados são tolerados. A complexidade de pior caso dos algoritmos é $O(1)$, ou seja, executam em tempo constante.

**Algoritmo 25** Algoritmo para adicionar na pilha (*push*)

```
1: function PUSH(pilha, x)
2:     pt ← novo_no
3:     pt ↑ .info ← x
4:     pt ↑ .prox ← pilha.topo
5:     pilha.topo ← pt
6: end function
```

# Filas Encadeadas

Para a implementação de filas encadeadas, são necessários dois ponteiros: *início* e *fim*, então se a fila está vazia têm-se que $inicio = fim = NULL$. Os Algoritmos 27 e 28 apresentam as operações de inserção e remoção, respectivamente. Veja no algoritmo de inserção que, se a fila estiver vazia, o elemento inserido será o primeiro elemento e o último elemento desta (Algoritmo 27, linha 8). Observe algo similar na remoção: se o elemento for o único elemento fila, então o fim da fila deverá receber **NULL** após a remoção (Algoritmo 28, linha 7). A complexidade de pior caso dos algoritmos é $O(1)$, ou seja, também executam em tempo constante.

**Algoritmo 26** Algoritmo para remover da pilha (*pop*)

```
1: function POP(pilha)
2:     valor ← NULL
3:     pt ← pilha.topo
4:     pilha.topo ← pt↑.prox
5:     valor ← pt↑.info
6:     desalocar(pt)
7:     return valor
8: end function
```

**Algoritmo 27** Algoritmo para adicionar na fila

```
1: function ADICIONAR(fila, x)
2:     pt ← novo_no
3:     pt↑.info ← x
4:     pt↑.prox ← NULL
5:     if fila.fim ≠ NULL then
6:         fila.fim↑.prox ← pt
7:     else
8:         fila.inicio ← pt
9:     end if
10:    fila.fim ← pt
11: end function
```

**Algoritmo 28** Algoritmo para remover da fila

```
1: function REMOVER(fila)
2:     valor ← NULL
3:     if fila.inicio ≠ NULL then
4:         pt ← fila.inicio
5:         fila.inicio ← fila.inicio↑.prox
6:         if fila.inicio = NULL then
7:             fila.fim ← NULL
8:         end if
9:         valor ← pt↑.info
10:        desalocar(pt)
11:    else
12:        Underflow
13:    end if
14:    return valor
15: end function
```

# Problema Prático 2

Considere o exemplo de fila encadeada fornecido com este material (Fila). Este exemplo apresenta a implementação de uma fila encadeada simples para o armazenamento de números inteiros. Os tipos definidos para esta fila são (fila.h):

```
1 typedef struct No{
2     struct No* prox;
```

```
    int dado;
} No;

typedef struct Fila{
    No* inicio;
    No* fim;
}Fila;
```

Todas as operações da fila já estão implementadas, exceto uma, remover:

```
// remove cabeca da fila
int remover(Fila* f);
```

Desta maneira, você deverá fornecer uma implementação funcional para esta operação, sem alterar outras operações e as structs da fila. Leve em consideração que esta fila não utiliza nó cabeça. A implementação deverá ser enviada ao professor.

# QUESTÕES

***Questão 1:.*** Considere a seguinte afirmação sobre estruturas encadeadas: *"As pilhas são listas encadeadas cujos elementos são retirados e acrescentados sempre ao final, enquanto as filas são listas encadeadas cujos elementos são retirados e acrescentados sempre no início."* Esta afirmação é correta ou errada? Justifique.

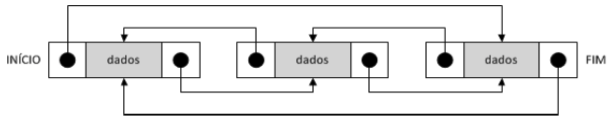
***Questão 2 (FGV - 2015):.*** A tabela a seguir ilustra uma lista duplamente encadeada de cores, sem a presença de nó cabeça.

| Elemento | Anterior | Seguinte | Cor |
|---|---|---|---|
| 1 | 4 | 2 | ... |
| 2 | 1 | 3 | ... |
| 3 | 2 | **NULL** | ... |
| 4 | 5 | 1 | ... |
| 5 | **NULL** | 4 | ... |

Dado que a ordem correta das cores é Marrom-Verde-Azul-Vermelho-Amarelo, a coluna Cor, na tabela acima, deveria apresentar, de cima para baixo, os seguintes valores:

A) Marrom-Vermelho-Amarelo-Azul-Verde;
B) Azul-Marrom-Verde-Vermelho-Amarelo;
C) Amarelo-Azul-Marrom-Vermelho-Verde;
D) Azul-Vermelho-Amarelo-Verde-Marrom;
E) Verde-Azul-Vermelho-Marrom-Amarelo.

***Questão 3 (FUMARC - 2014):.*** Considere a figura a seguir representando uma estrutura de dados:




São características da estrutura de dados representada, EXCETO:

A) Os elementos da estrutura estão duplamente encadeados.
B) O último elemento inserido é sempre o primeiro a ser retirado da estrutura.
C) A estrutura representada é circular, ou seja, o último elemento aponta para o primeiro e este, para o último.
D) Nesse tipo de estrutura, cada elemento possui um ponteiro usado para apontar para o elemento anterior e outro usado para apontar para o próximo elemento da estrutura.

***Questão 4 (CESGRANRIO - 2006):.*** Os registros em uma lista, duplamente encadeada com 20 elementos possuem cada um três campos: **próximo**: um ponteiro para o próximo elemento da lista; **valor**: informação armazenada pelo elemento; **anterior**: um ponteiro para o elemento anterior da lista. Sendo "Z"o décimo elemento desta lista e "X"e "Y"dois outros elementos que não pertencem à lista, com seus respectivos ponteiros "pZ", "pX"e "pY", considere o trecho de código abaixo.

1: $pY \uparrow .proximo = pX$
2: $pX \uparrow .anterior = pY$
3: $pX \uparrow .proximo = pZ \uparrow .proximo$
4: $pZ \uparrow .proximo \uparrow .anterior = pX$
5: $pZ \uparrow .proximo = pY$
6: $pY \uparrow .anterior = pZ$

Este trecho de código é usado para inserir na lista os elementos:

A) Y, logo após o Z, e X, logo após o Y.
B) Y, antes do Z, e X, logo após o Z.
C) Y, antes do Z, e X, antes do Y.
D) X, logo após o Z, e Y, logo após o X.
E) X, antes do Z, e Y, logo após o Z.

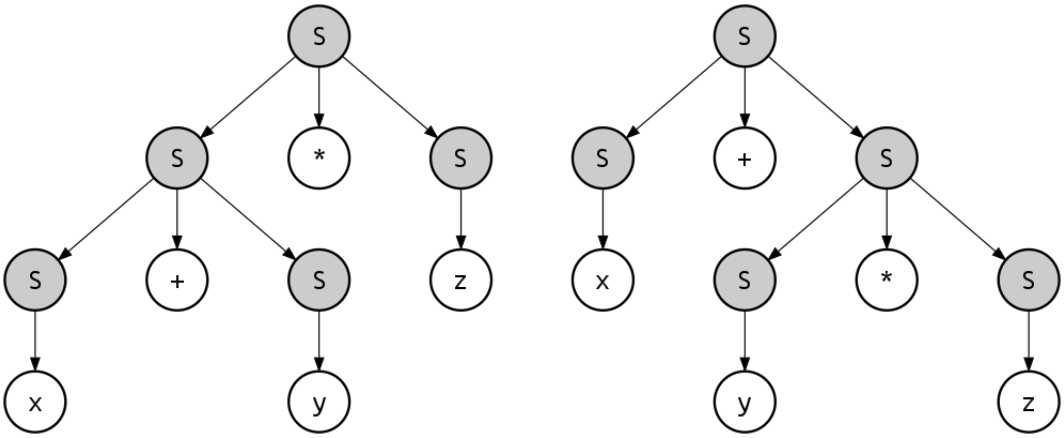
# Capítulo 7: Árvores

Nas unidades anteriores foram estudadas estruturas lineares ou sequenciais para o armazenamento de dados. Mas, em diversas aplicações, são necessárias estruturas mais complexas que as sequenciais, sendo a mais utilizada a **Árvore**. Árvores representam estruturas com aplicabilidade em diversos problemas, pois são fáceis de trabalhar de maneira computacional, ao contrário de grafos. Árvores são aplicadas em diversos domínios, como construção de dicionários, indexação e busca em bancos de dados, implementação de representações sintáticas em compiladores, representação de conjuntos de dados em linguagens (*sets*), mapeamentos chave-valor (*maps*) entre muitos outros. Na próxima seção serão apresentados conceitos básicos e fundamentais para o estudo e entendimento das Árvores em suas diversas formas.

## Conceitos Básicos

Como contato inicial, a Figura 7.1 apresenta um exemplo com duas árvores que representam expressões matemáticas. A primeira expressão é relativa a $(x + y) * z$ e a segunda a $x + y * x$.

Figura 7.1: Exemplo de Árvore para um expressão




Fonte: https://commons.wikimedia.org/wiki/File:Parse_Tree_Derivations.svg

Considerando a árvore apresentada como exemplo, as seguintes definições são válidas: uma *árvore enraizada T*, ou *árvore* é um conjunto de elementos chamados *nós* ou *vértices* tais que $T = \varnothing$ é a *árvore* vazia ou existe um nó especial chamado de *raiz* de $T(r(T))$. Os outros nós constituem um conjunto vazio ou são divididos em $m \geq 1$ conjuntos disjuntos não vazios (subárvores de $r(T)$), cada um sendo uma árvore. Ainda considerando a definição, se $v$ é um nó de $T$, a notação $T(v)$ indica a subárvore de $T$ com raiz $v$. Nós que não possuem filhos são chamados de *folhas* da árvore.

### Representações

Uma das maneiras de se representar uma árvore é através de um *diagrama de inclusão*, conforme o exemplo da Figura 7.2. Outra representação para a árvore (maneira mais

tradicional) da Figura 7.2 pode ser vista na Figura 7.3, a qual também pode ser ainda representada por $(A(B)(C(D(G)(H))(E)(F(I))))$ (notação parentizada).

Figura 7.2: Árvore como diagrama de inclusão

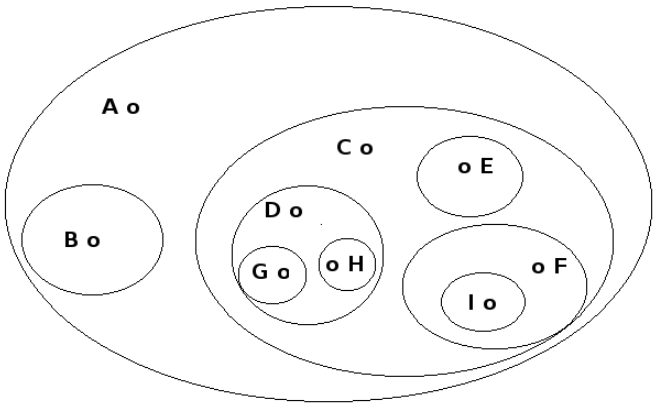
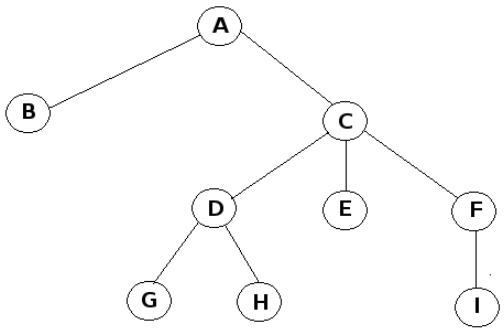



Figura 7.3: Árvore com representação tradicional



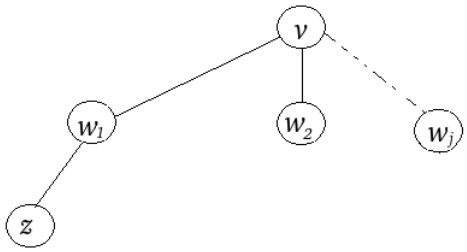

Um exemplo de aplicação da notação parentizada é que qualquer expressão aritmética pode ser colocada sob a forma de uma sequência de parênteses aninhados, por exemplo: $a + b * (c/d - e)$ é equivalente a $(a + (b * ((c/d) - e)))$.

## Relacionamentos e métricas

Considere a árvore da Figura 7.4. Seja $v$ o nó raiz da subárvore $T(v)$ de $T$, então os nós raízes $w_1, w_2, ..., w_j$ das subárvores de $T(v)$ são chamados de *filhos* de $v$, sendo $v$ é chamado *pai* de $w_1, w_2, ..., w_j$. Os nós $w_1, w_2, ..., w_j$ 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* deste nó. Por exemplo, o nó $v$ possui grau de saída $j$ enquanto $w_1$ possui grau 1 e $z$ 0.

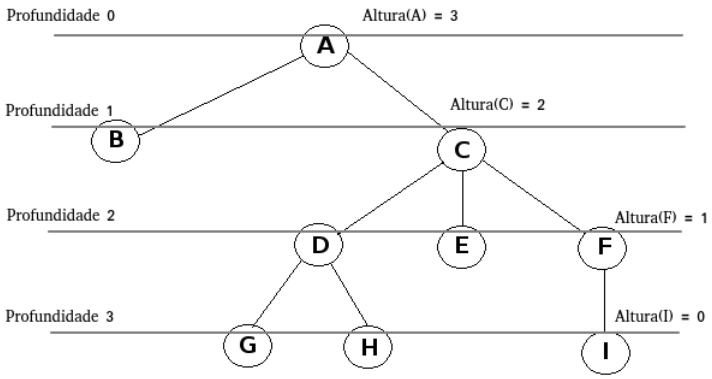
Uma sequência $v_1, v_2, ..., v_k$ em que para cada par de nó existe a relação *"é filho de"* ou *"é pai de"* é denominada **caminho** da árvore, então, diz-se que $v_1$ alcança $v_k$ e o valor $k - 1$ é o comprimento do caminho. A **profundidade** de um nó é o comprimento do caminho entre a raiz e este nó (raiz tem profundidade 0). O conjunto de nós com a mesma profundidade é denominado **nível**, sendo que a raiz tem nível 0. A **altura** de um nó $v$ é o número de nós do maior caminho de $v$ até um de seus descendentes (folhas

Figura 7.4: Árvore e relacionamentos




tem altura 0). A altura de árvore é a altura da raiz desta. A Figura 7.5 sumariza a noção de altura e profundidade de uma árvore.

Figura 7.5: Altura e profundidade de uma árvore




# Árvores Binárias

Uma **árvore binaria T** é um conjunto finito de elementos denominados nós, tal que $T = \varnothing$, é a *árvore* vazia, ou existe um nó especial chamado de *raiz* de $T(r(T))$, os outros nós constituem um conjunto vazio ou são divididos em dois conjuntos disjuntos, $T_E(r(T))$ e $T_D(r(T))$, a subárvores *direita* e *esquerda* da raiz (também árvores binárias). A Figura 7.6 apresenta dos exemplos de árvores binárias.

A Figura 7.7 apresenta tipos particulares de árvores binárias: **(a)** estritamente binária, onde todos os nós que não são folhas possuem dois filhos, **(b)** binária completa, que é uma estrita com todos os níveis completos, exceto o último e **(c)** cheia, que é uma completa até o último nível. A altura $h$ é uma árvore binária *cheia* é $h = 1 + \lfloor log_n \rfloor$

## Percurso em Árvores Binárias

Existem diversas maneira de percorrer uma árvore, ou seja, visitar todos os nós para algum devido propósito, são elas:

**Percurso em pré-ordem:** deve-se visitar a raiz, depois o nó esquerdo e finalmente o direito, recursivamente. Para a árvore da Figura 7.8 obtém-se a seguinte ordem de visitação: A, B, D, G, C, E, H, I, F.

Figura 7.6: Árvore binárias

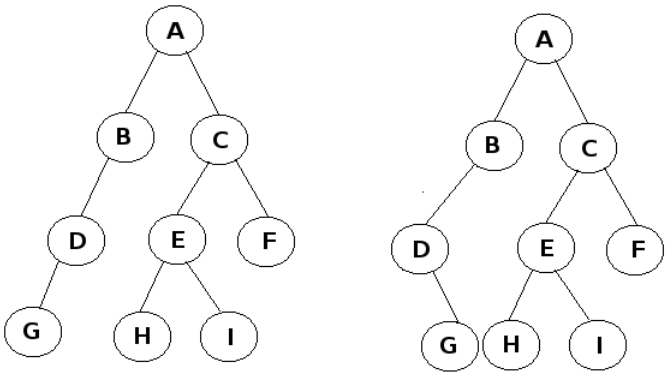
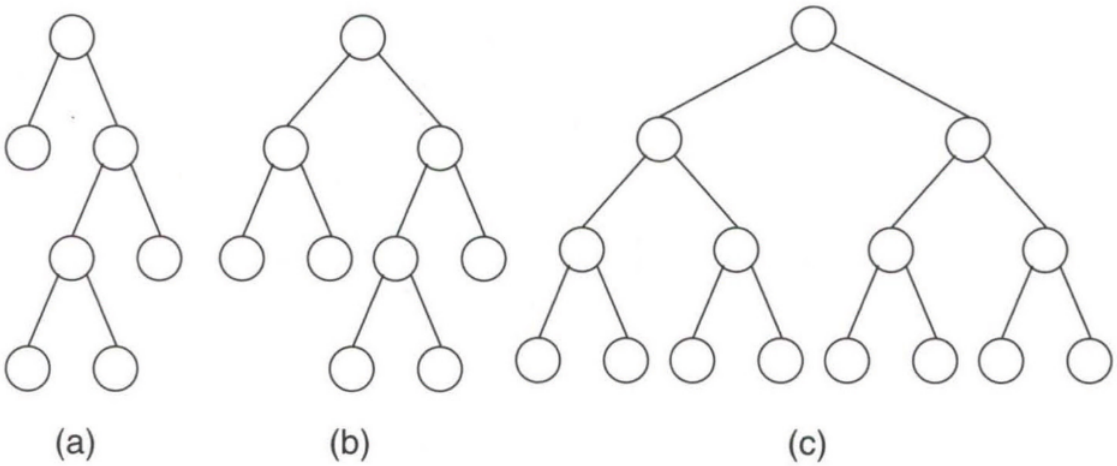



Figura 7.7: Tipos particulares de árvore binária




**Percurso em ordem simétrica (em ordem):** deve-se visitar o nó esquerdo, depois a raiz e finalmente o direito, recursivamente. Para a árvore da Figura 7.8 obtém-se a seguinte ordem de visitação: G, D, B, A, H, E, I, C, F.

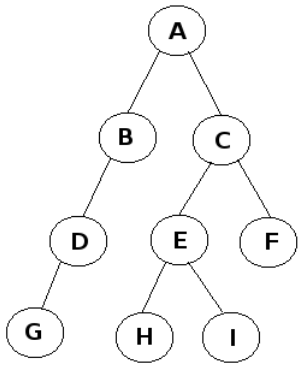
**Percurso em pós-ordem:** deve-se visitar o nó esquerdo, depois o direito e finalmente a raiz, recursivamente. Para a árvore da Figura 7.8 obtém-se a seguinte ordem de visitação: G, D, B, H, I, E, F, C, A.

**Percurso em largura:** deve-se visitar todos os nós de um determinado nível antes de passar para o próximo nível. Para a árvore da Figura 7.8 obtém-se a seguinte ordem de visitação: A, B, C, D, E, F, G, H, I. Esta é a única estratégia onde a abordagem recursiva não é a mais recomendada. Um algoritmo interessante para este problema consiste em utilizar uma fila para o armazenamento de nós. Primeiramente deve se adicionar a raiz na fila e, enquanto esta não estiver vazia, retirar um elemento, processar ele e adicionar seus filhos na mesma fila (caso existam).

# Problema Prático 3

Para este problema, você deverá utilizar o código do diretório **ArvoreBinaria**. Este código implementa no arquivo **main.c** a árvore descrita na Figura 7.8. Em **arvore.c** estão implementados os percursos para impressão em pré-

Figura 7.8: Árvore e percursos




ordem (recursivo) e largura (usando uma fila). O seu trabalho é implementar a impressão em ordem **simétrica** e **pós-ordem**. Para tanto, observe que cada nó da árvore (**arvore.h**) representa recursivamente uma arvore contendo os filhos à esquerda e à direita, conforme abaixo.

```
typedef struct No{
    char chave;
    struct No* esquerda;
    struct No* direita;
} No;
```

Neste contexto, as seguintes funções em **arvore.c** deverão ser implementadas:

```
void imprimir_simetrica(No* no){

}
void imprimir_posordem(No* no){

}
```

# Árvores Binárias de Busca

As árvores de busca são estruturas que armazenam dados e chaves e que permitem a localização eficiente de elementos nela inseridos. Para este objetivo, os elementos estão dispostos de maneira ordenada, considerando as relações de comparabilidade entre as chaves.

Primeiramente, considera-se que uma árvore de busca armazena um conjunto de chaves $S = \{s_1, ..., s_n\}$, as quais satisfazem a condição $s_i < ... < s_n$, ou seja, são comparáveis entre si, estabelecendo uma relação de ordem. Então, o objetivo é, dado uma valor $x$, verificar se $x \in S$. Caso positivo, pode-se localizar $x$ em $S$ e ainda determinar o índice $j$ tal que $x = s_j$. Para que isto seja possível, usa-se uma árvore binária rotulada $T$, com as seguintes características:

(i) $T$ possui $n$ nós. Cada nó $v$ corresponde a uma chave distinta $s_j \in S$ e possui como

rótulo o valor $rt(v) = s_j$

(ii) Seja um nó $v$ de T. Seja também $v_1$, pertencente à subárvore esquerda de $v$. então

$$rt(v_1) < rt(v) \tag{7.1}$$

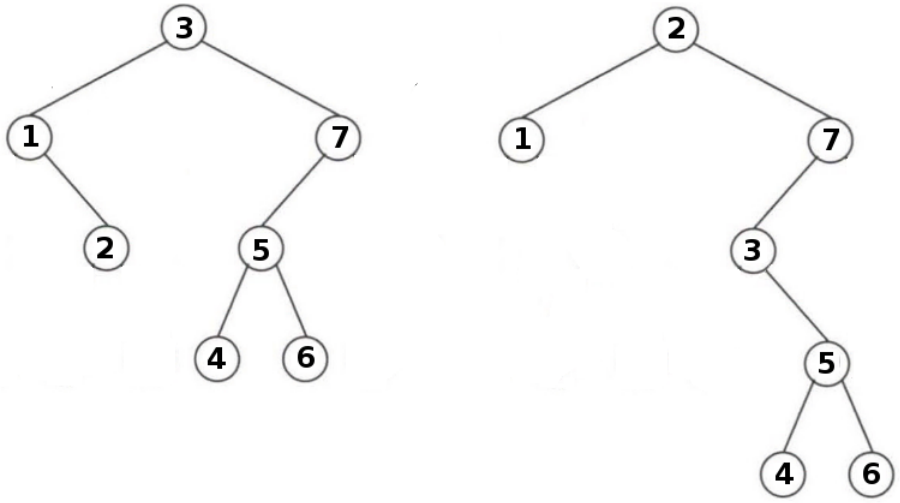
Analogamente, se $v_2$ pertence à subárvore direita de $v$,

$$rt(v_2) > rt(v) \tag{7.2}$$

Desta maneira, a árvore $T$ é denominada *árvore binária de busca* para $S$. De outra maneira, nesta árvore, todas a chaves menores que a chave da raiz ($rt(v)$) estarão na subárvore à esquerda, enquanto os valores maiores estarão na subárvore à direita.

Se $|S| > 1$, ou seja, sempre que existir mais de uma chave no conjunto, podem existir várias árvores binárias de busca para $S$. A Figura 7.9 apresenta duas árvores para a o conjunto $\{1, 2, 3, 4, 5, 6, 7\}$.

Figura 7.9: Árvores de busca




A seguir serão apresentados algoritmos para as operações mais comuns em árvores binárias de busca. É importante frisar que tais algoritmos são apenas sugestões, sendo que diversas estratégias existem para cada problema. Por exemplo, a operação de busca aqui descrita foi pensada para auxiliar também a inserção, de maneira similar a lista. Todavia, também será apresentado um algoritmo de inserção que não depende de uma busca.

## Busca

A operação de busca é em uma árvore binaria de busca é apresentada no Algoritmo 29. Este algoritmo recebe um nó da árvore e um valor $x$ como parâmetro. Se o valor for igual a chave do nó (linha 2), termina-se a busca (linha 3) pois o o elemento foi encontrado. Caso contrário, se o valor for menor que a chave (linha 5) existem duas opções: encerrar a busca caso não se tenha filhos à esquerda (linha 7), retornando a última folha da árvore analisada (importante para a inserção) ou continuar a busca pela subárvore esquerda (linha 9). De maneira correlata, caso o valor seja maior que a chave

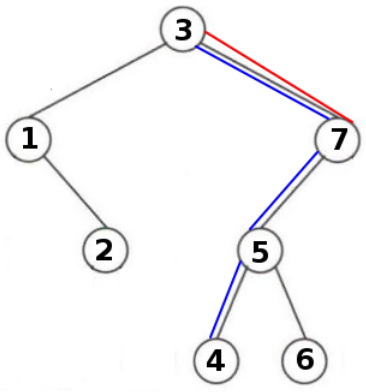
faz-se o mesmo com a subárvore à direita (linhas 12 a 16). Embora este algoritmo seja recursivo, abordagens iterativas podem ser aplicadas também. Perceba também que este algoritmo opera a partir de um nó, retornando ou o elemento procurado ou um próximo a este. Qualquer semelhança com a busca binária vista anteriormente não é coincidência, pois a estratégia é a mesma.

**Algoritmo 29** Algoritmo para busca em árvore

```
1:  function BUSCA_NO_ARVORE(pt, x)
2:      if pt↑.chave = x then
3:          return pt
4:      else
5:          if x < pt↑.chave then
6:              if pt↑.esq = NULL then
7:                  return pt
8:              else
9:                  return BUSCA_NO_ARVORE(pt↑.esq, x)
10:             end if
11:         else
12:             if pt↑.dir = NULL then
13:                 return pt
14:             else
15:                 return busca − no − arvore(pt↑.dir, x)
16:             end if
17:         end if
18:     end if
19: end function
```

Um exemplo da aplicação do algoritmo pode ser visto na Figura 7.10. Neste exemplo, uma busca pela chave 4 será bem sucedida, retornando o nó em questão (caminho azul), enquanto uma busca pelo valor 8 resultará no retorno do nó cuja chave é 7 (caminho vermelho).

Figura 7.10: Exemplo de busca




## INSERÇÃO

O objetivo da inserção é colocar um novo nó na árvore com a chave fornecida. Por simplicidade, será considerado aqui que somente a chave fara parte do nó, mas qualquer

tipo de informação poderia ser agregada também, sem perda de generalidade. O Algoritmo 30 apresenta a inserção de uma chave com valor $x$ em uma árvore. O algoritmo segue uma ideia muito parecida com a lista encadeada, que é a inserção a partir de uma busca. Primeiramente, o algoritmo verifica se a árvore está vazia, ou seja, se o nó raiz é nulo (linha 3), se sim, ele cria um novo nó a atribui como raiz. Se a árvore não estiver vazia, o primeiro passo é fazer uma busca e se o elemento já estiver na árvore o processo deve parar (linhas 9 e 10). Caso não exista nó com a chave em questão, o algoritmo de busca deve retornar um nó folha. Então, um novo nó é alocado e adicionado como filho do nó previamente retornado pela busca, sendo a posição determinada pela comparação dos valores de chave: à esquerda, caso $x$ seja menor que a chave do nó, ou à direita, caso contrário.

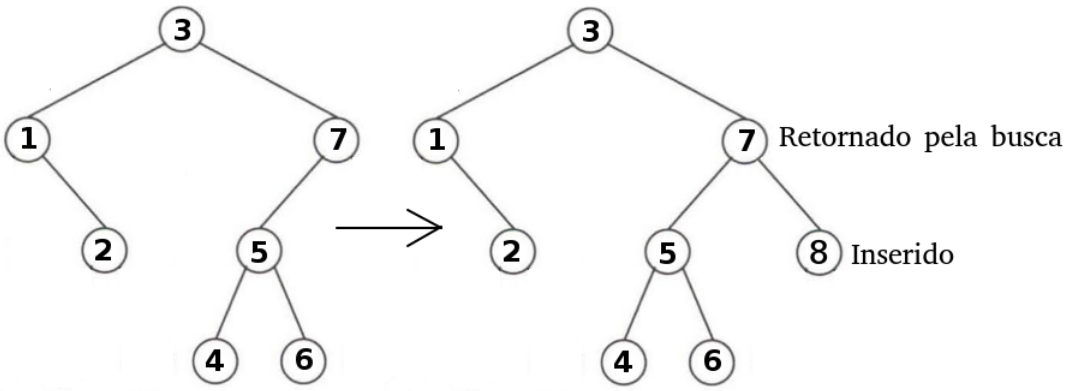
**Algoritmo 30** Algoritmo para inserção em árvore

```
1:  function INSERIR_ARVORE(arvore, x)
2:      pt ← arvore ↑ .raiz
3:      if pt = NULL then
4:          ptnovo ← novo_no
5:          ptnovo ↑ .chave ← x
6:          arvore ↑ .raiz ← ptnovo
7:      else
8:          ptbuscado ← BUSCA_NO_ARVORE(pt, x)
9:          if ptbuscado ↑ .chave = x then
10:             Elemento já inserido
11:         else
12:             ptnovo ← novo_no
13:             ptnovo ↑ .chave ← x
14:             if x < ptbuscado ↑ .chave then
15:                 ptbuscado ↑ .esq ← ptnovo
16:             else
17:                 ptbuscado ↑ .dir ← ptnovo
18:             end if
19:         end if
20:     end if
21: end function
```

A Figura 7.11 apresenta um exemplo de inserção. Neste exemplo, pretende-se inserir a chave 8 na árvore, sendo a busca a primeira operação a ser efetuada. A busca não obtém sucesso para a chave 8, retornando o nó com a chave 7. O nó com a chave 8 é inserido à direita do nó 7, por ser maior.

Observe que ainda outra abordagem possível que a inserção direta de nós, de maneira concomitante a busca, conforme é mostrado no Algoritmo 31. Neste algoritmo, que é substancialmente menor que as abordagens anteriores, a ideia descer através da árvore até chegar a uma folha, onde o nó será inserido. Perceba que a função $INSERIR\_NO$ somente retornará um nó novo (linhas 2-4) quando o primeiro argumento for NULL, caso contrário retornará o próprio nó, não alterando a árvore. Quando a chave já estiver na árvore, o algoritmo interromperá a busca, não entrando nem no **if** da linha 6, nem no da linha 9.

Figura 7.11: Exemplo de inserção em árvore




**Algoritmo 31** Algoritmo para inserção direta de nó

```
1:  function INSERIR_NO(pt, chave)
2:      if pt = NULL then
3:          pt ← novo_no
4:          pt ↑ .chave ← chave
5:      else
6:          if chave < pt ↑ .chave then
7:              pt ↑ .esq = INSERIR_NO(pt ↑ .esq, chave)
8:          else
9:              if chave > pt ↑ .chave then
10:                 pt ↑ .dir = INSERIR_NO(pt ↑ .dir, chave)
11:             else
12:                 Elemento já inserido
13:             end if
14:         end if
15:     end if
16:     return pt
17: end function
```

## Remoção

A Remoção de nós em uma árvore binária de busca é uma operação um pouco mais complicada que a inserção. A remoção considera 3 casos:

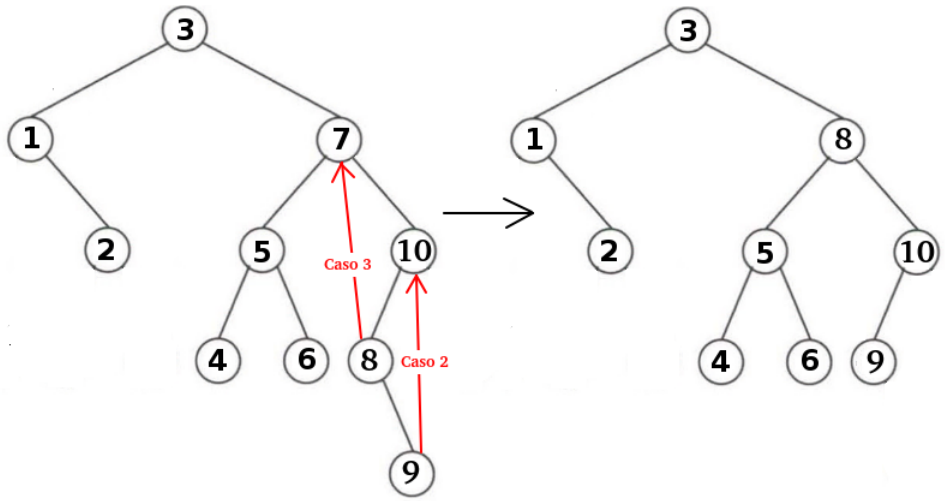
1. O Nó não possui filhos (folha): situação simples, basta fazer com que o pai aponte para NULL.

2. O Nó possui apenas um filho: situação também simples, basta fazer o pai apontar para neto.

3. O Nó possui os dois filhos: esta situação é um pouco mais complexa, deve-se substituir o Nó pelo (1)elemento mais à direita da subárvore esquerda (imediatamente menor) ou pelo (2)elemento mais à esquerda da subárvore direita (imediatamente maior).

Para um melhor entendimento, considere o exemplo da Figura 7.12. Neste exemplo, pretende-se remover a nó com chave 7. Para remover este nó, optou-se pela opção (2) do caso **3** anterior, ou seja, substituição pelo elemento mais à esquerda da subárvore da direita. Neste caso, tal elemento é o com chave 8, e este possui um filho. Antes de mais nada, aplica-se o caso **2** anterior para remoção do elemento 8, fazendo o nó 10 apontar para o nó 9 como novo filho. Feito isto, o nó 8 pode substituir o nó 7 como mostrado na figura. Perceba que se o caminho escolhido fosse pelo elemento mais à direita da subárvore esquerda, o nó 6 poderia facilmente substituir o 7 sem a aplicação de outros passos.

Figura 7.12: Exemplo de remoção




Em termos de algoritmos, pode-se adotar a ideia de remover o nó em um mesmo algoritmo de busca, utilizando uma ideia similar a apresentada no algoritmo de inserção em 31. A ideia em questão é apresentada no Algoritmo 32, que tenta localizar a chave pela esquerda (se ela for menor, linha 4) ou pela direita (se for maior, linha 7). Se a chave for encontrada (else da linha 8), a remoção efetiva do nó é processada, considerando ele como raiz das subárvores que estão abaixo dele (linha 8). O algoritmo na linha 9 simplesmente substitui a raiz pelo nó mais à direita da subárvore à esquerda, sendo apresentado no Algoritmo 33. A estratégia deste algoritmo é aplicar precisamente as regras anteriores: se a raiz não possui filho à esquerda, filho a direita assume a raiz (linhas 4-8). Entretanto, se este não for o caso descobre-se o filho mas à direita da subárvore esquerda (linhas 9-14) para finalmente este substituir a raiz (linhas 15-19).

A complexidade dos algoritmos de busca, inserção e remoção são todos, no pior caso, $O(n)$. Isto se dá pois toda árvore binária não balanceada pode assumir uma forma patológica similar a uma lista encadeada, pois a forma final da mesma depende exatamente da ordem de inserção das chaves na mesma. Por exemplo, uma árvore pode ter somente filhos a direita, sendo nesta situação equivalente a uma lista. Na próxima parte dos estudos, serão apresentados conceitos relacionados ao balanceamento de árvores cujo objetivo é melhorar o desempenho das mesmas de maneira substancial.

**Algoritmo 32** Algoritmo para remoção de um nó

```
1: function REMOVER_NO(pt, chave)
2:     if pt ≠ NULL then
3:         if chave < pt↑.chave then
4:             pt↑.esq = REMOVER_NO(pt↑.esq, chave)
5:         else
6:             if chave > pt↑.chave then
7:                 pt↑.dir = REMOVER_NO(pt↑.dir, chave)
8:             else                          ▷ Achou a chave, remover raiz da subárvore
9:                 pt = REMOVER_RAIZ(pt)
10:            end if
11:        end if
12:    end if
13:    return pt
14: end function
```

**Algoritmo 33** Algoritmo para remoção de uma raiz de subárvore

```
1: function REMOVER_RAIZ(raiz)
2:     pai = NULL
3:     nova_raiz = NULL
4:     if raiz↑.esq = NULL then
5:         nova_raiz = raiz↑.dir
6:         desalocar(raiz);
7:         return nova_raiz
8:     end if
9:     pai = raiz
10:    nova_raiz = raiz↑.esq
11:    while nova_raiz↑.dir ≠ NULL do
12:        pai = nova_raiz
13:        nova_raiz = nova_raiz↑.dir
14:    end while
15:    if pai ≠ raiz then   ▷ Se verdadeiro, resolver filho da esq., caso contrário filho da
   esq. estará correto
16:        pai↑.dir = nova_raiz↑.esq
17:        nova_raiz↑.esq = raiz↑.esq
18:    end if
19:    nova_raiz↑.dir = raiz↑.dir
20:    desalocar(raiz)
21:    return nova_raiz
22: end function
```

# Árvores Balanceadas

Antes de mais nada, cabe uma reflexão: Quanto cabe de informação em uma árvore binária? Em outras palavras, quantos nós podem existir em uma árvore para uma determinada altura ou número de níveis? A Tabela 7.1 apresenta a resposta para determinadas quantidades de níveis, considerando **árvores completas** para uma determinada altura. É fácil verificar que com 1 nível só existe a raiz, com 2 a raiz e dois

filhos e assim por diante, com um crescimento de $2^N - 1$ para N níveis. Nesta tabela, é possível verificar que, dentre um conjunto de 1 bilhão de chaves (ou registros quaisquer), serão necessárias no máximo 30 comparações para se achar qualquer uma delas, pois no pior caso a chave estará na folha de uma árvore de altura 30. Estas características permitem hoje que bancos de dados processem consultas muito rapidamente, por exemplo.

Tabela 7.1: Capacidade das árvores binárias

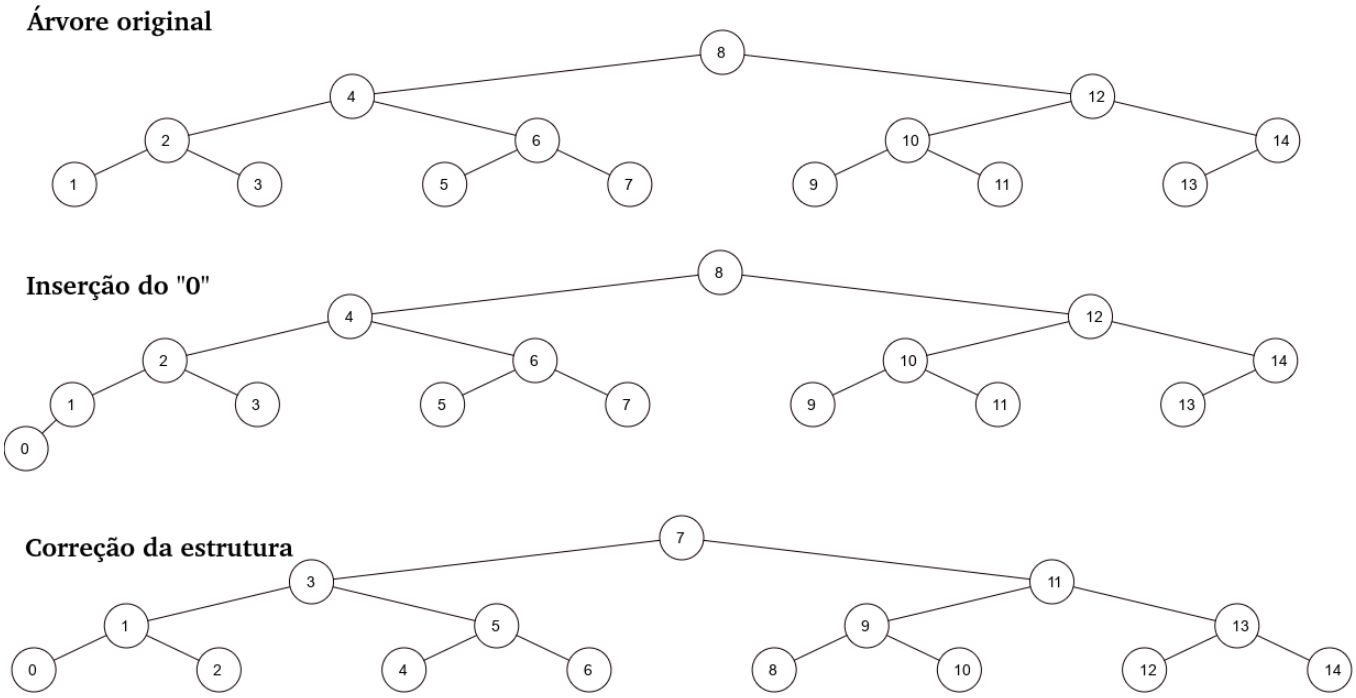
| **Nível** | **Número de nós** |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 7 |
| ... | ... |
| N | $2^N - 1$ |
| ... | ... |
| 16 | 65.535 |
| 20 | 1 milhão |
| 30 | 1 bilhão |
| ... | ... |

Um aspecto fundamental das árvores de busca é a manutenção custo de acesso a uma chave. Considerando que uma árvore completa irá possuir altura $O(\log\, n)$ para $n$ nós, a ideia é tentar manter a altura das árvores de busca na mesma ordem desta, pois a altura é quem irá ditar o custo de acesso as chaves. Para manter a altura e por consequência o custo de acesso na ordem $O(\log\, n)$ tanto para inclusões quanto para remoções é necessário que a árvore sofra ajustes periódicos para regulagem da estrutura/altura.

Conforme exemplificado, arvores completas minimizam comparações, e árvores não completas podem assumir formas ruins para busca. A primeira tentativa de correção da estrutura de uma árvore seria aplicar uma algoritmo para deixar a árvore completa quando esta característica é perdida. Considere o exemplo da Figura 7.13 representando a inserção de um elementos em uma árvore. Neste exemplo, a chave com valor 0 precisa ser inserida na árvore de cima e só existe uma posição para isso, a esquerda do nó com chave 1. Acontece que, como mostrado na figura do meio, a inserção de um novo elemento retirou a característica completa da árvore. Aplicando um algoritmo qualquer para a correção da completude obteve-se a árvore mais abaixo. Embora a árvore tenha se tornado completa, este procedimento envolveu a movimentação de praticamente todos os nós da árvore, ou seja, o custo de inserção no pior caso foi $O(n)$. A conclusão deste exemplo é que árvores completas são inadequadas para estruturas dinâmicas, pois o reestabelecimento da condição pode movimentar todos os nós.

Uma alternativa consiste em permitir que, no pior caso, a busca não seja necessariamente tão pequena quanto o mínimo $1 + \lfloor \log n \rfloor$. A ideia é permitir que a altura da árvore seja da ordem de grandeza de uma completa com mesmo número de

Figura 7.13: Inserção e reestabelecimento de completude




nós, ou seja $O(\log n)$ para $n$ nós, em outras palavras maior que $1 + \lfloor \log n \rfloor$ mas que não ultrapasse assintoticamente $O(\log n)$. Estruturas com esta característica são denominadas árvores *balanceadas*. Árvore com estas características são denominadas *balanceadas* e são massivamente utilizadas em C++ (std::map/std::set), Java (TreeMap/TreeSet) e em sistemas que necessitem de operações de conjunto e indexação de dados, por exemplo. Exemplos de árvores balanceadas são as **Árvores B**, **Árvores Rubro-Negras** (*Red-Black Trees*) e **Árvores AVL**. A próxima parte deste estudo apresenta as Árvores AVL como exemplo representativo.

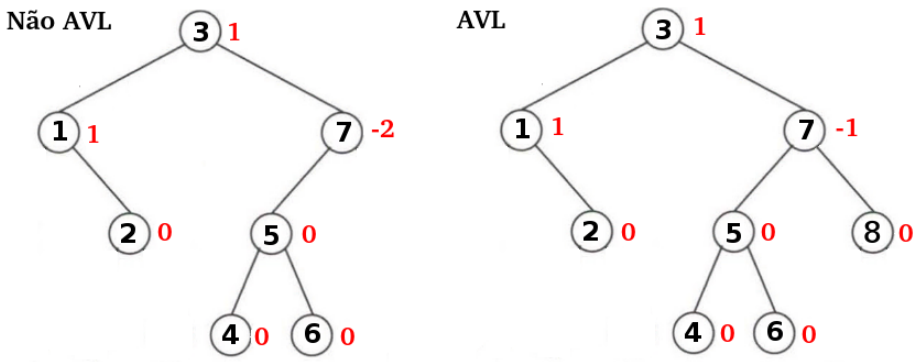
## Árvore AVL

As Árvores AVL representam um tipo estrutura proposta por Georgy **A**delson-**V**elsky e Yevgeniy **L**andis em 1962. Uma árvore binária $T$ é denominada AVL quando para qualquer nó $T$, para cada subárvore, a altura difere em módulo de até uma unidade. Nós *regulados/balanceados* satisfazem esta condição e nós *desregulados/desbalanceados* não satisfazem. A Figura 7.14 apresenta exemplos de árvores que atendem ou não a condição AVL. A árvore da esquerda não atende pois o nó com chave 7 está desbalanceado, ou seja, a diferença de altura entre duas suas subárvores é em modulo 2. Os valores anotados em vermelho são denominados fatores de balanceamento e representam a altura da subárvore direita menos altura da esquerda.

### Balanceamento

O fator de balanceamento de cada nó (conforme anotação em vermelho no exemplo) ao longo do caminho da raiz até o nó inserido ou excluído deve ser recalculado a cada

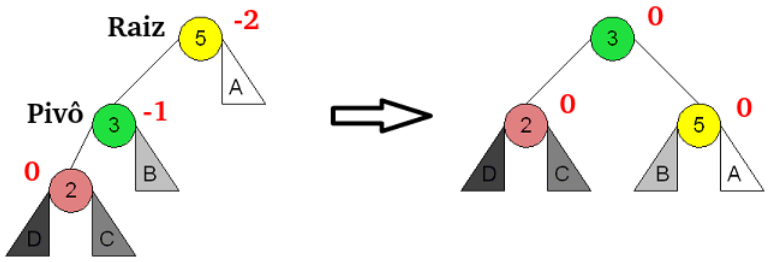
Figura 7.14: Exemplos de condição AVL




operação, que obviamente são mesmas operações vistas anteriormente para inserção e remoção. Se o fator de balanceamento para cada nó for -1, 0 ou +1, a árvore está balanceada e nada precisa ser feito. Todavia, se um fator de balanceamento não estiver nesta faixa, um dos quatro tipos de rotação deverá ser executado:

**Rotação simples à direita:** Usado quando o nó está fora de balanceamento para a esquerda, ou seja, fator de balanceamento $\leq$ -2 com subárvore esquerda com fator de balanceamento = -1. A Figura 7.15 apresenta um exemplo de rotação simples à direita. Neste exemplo, o pivô é o elemento que se tornará a nova raiz, ocorrendo a transferência de filhos entre ele e a atual raiz.

Figura 7.15: Rotação simples à direita

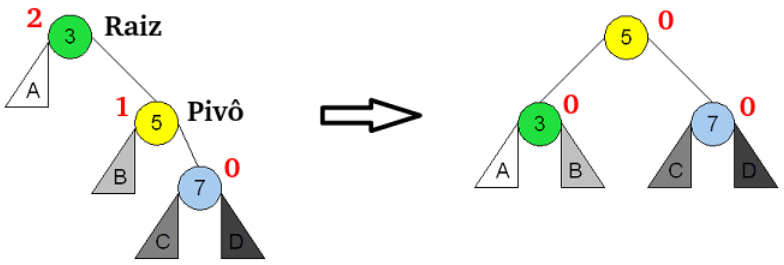



Fonte: http://occcwiki.org/index.php/AVL_Trees (adaptado)

**Rotação simples à esquerda:** Usado quando o nó está fora de balanceamento para a direita (fator de balanceamento $\geq$ 2) com subárvore direita com fator de balanceamento = 1. Figura 7.16 apresenta um exemplo de rotação simples à esquerda. Neste exemplo, o pivô também é o elemento que se tornará a nova raiz, ocorrendo a transferência de filhos entre ele e a atual raiz em uma situação muito parecida com a anterior.

**Rotação dupla à esquerda e à direita (também chamada dupla à direita):** Usado quando o nó está fora de balanceamento para a esquerda (fator de balanceamento $\leq$ -2) com subárvore esquerda com fator de balanceamento = 1. Figura 7.17 apresenta

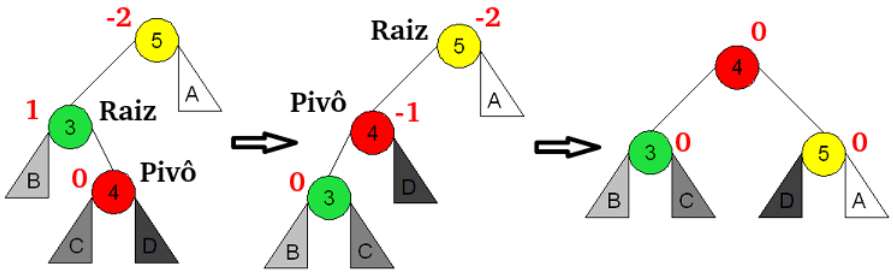
Figura 7.16: Rotação simples à esquerda




Fonte: http://occcwiki.org/index.php/AVL_Trees (adaptado)

um exemplo de rotação dupla à direita. Neste exemplo, perceba que, para o pivô chegar a raiz, ele precisará primeiro sofrer uma rotação à esquerda e depois à direita.

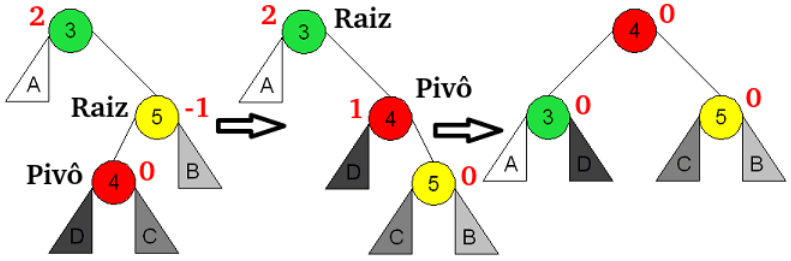
Figura 7.17: Rotação dupla à direita




Fonte: http://occcwiki.org/index.php/AVL_Trees (adaptado)

**Rotação dupla à direita e à esquerda (também chamada dupla à esquerda):** Usado quando o nó está fora de balanceamento para a direita (fator de balanceamento $\geq 2$) com subárvore direita com fator de balanceamento = -1. Figura 7.18 apresenta um exemplo de rotação dupla à esquerda. Neste exemplo, perceba que, para o pivô chegar a raiz, ele precisará primeiro sofrer uma rotação à direita e depois a esquerda, de maneira contrária a situação anterior.

Figura 7.18: Rotação dupla à esquerda




Fonte: http://occcwiki.org/index.php/AVL_Trees (adaptado)

A grande vantagem das Árvores AVL é a complexidade de pior caso para as operações. Neste caso, todas as operações (busca, inserção e remoção) possuem complexidade

$O(\log n)$ e a complexidade de espaço é $O(n)$. Todas as operações possuem complexidade da ordem de $O(\log n)$ por conta da altura da árvore.

# Problema Prático 4

Considere o exemplo de implementação de Árvore Binária de Busca AVL fornecido com este material (ArvoreAVL). Este exemplo apresenta a implementação das operações apresentadas anteriormente bem como algumas outras, para tamanho, por exemplo. A árvore é representada por duas **structs**:

```
typedef struct No{
    int chave;
    struct No* esquerdo;
    struct No* direito;
    int peso;
} No;

typedef struct ArvoreAVL{
    No* raiz;
} ArvoreAVL;
```

Sendo as operações permitidas:

```
ArvoreAVL* criar_arvore();
void destruir_arvore(ArvoreAVL* arvore);
int buscar(ArvoreAVL* arvore, int chave);
void inserir(ArvoreAVL* arvore, int chave);
void remover(ArvoreAVL* arvore, int chave);
int altura(ArvoreAVL* arvore);
```

O seu trabalho consiste em utilizar a implementação de árvore para alguns testes específicos:

1. Crie um projeto com os 3 arquivos fornecidos.
2. Execute o programa compilado e veja o resultado. Neste exemplo, são inseridas 30000 chaves em uma árvore, a qual não é balanceada. Anote os resultados.
3. Agora, procure a linha:

```
  #define BALANCEAR 0

```

e altere o valor de 0 para 1. Compile e execute. O resultado será diferente pois agora a árvore será balanceada. Compare o valor que está na raiz e a altura da árvore com o caso anterior. Anote os resultados e as suas observações.

4. Faça a seguinte alteração: ao invés de medir o tempo total, faça o programa medir e imprimir na tela o tempo de cada inserção de chave. Qual o comportamento do tempo para a estrutura balanceada? E para a não balanceada?

Faça o envio da alteração no código e também das suas respostas.

# Questões

***Questão 1 (CESGRANRIO - 2018):.*** A sequência de chaves 20 – 30 – 25 – 31 – 12 – 15 – 8 – 6 – 9 – 14 – 18 é organizada em uma árvore binária de busca. Em seguida, a árvore é percorrida em pré-ordem. Qual é a sequência de nós visitados?
A) 6 – 9 – 8 – 14 – 18 – 15 – 12 – 25 – 31 – 30 – 20
B) 20 – 12 – 8 – 6 – 9 – 15 – 14 – 18 – 30 – 25 – 31
C) 6 – 8 – 9 – 12 – 14 – 15 – 18 – 20 – 25 – 30 – 31
D) 20 – 30 – 31 – 25 – 12 – 15 – 18 – 14 – 8 – 9 – 6
E) 6 – 8 – 9 – 14 – 15 – 18 – 12 – 25 – 30 – 31 – 20

***Questão 2 (IBFC - 2014):.*** Quanto ao Algoritmo e estrutura de dados no caso de árvore AVL (ou árvore balanceada pela altura), analise as afirmativas abaixo, dê valores Verdadeiro (V) ou Falso (F) e assinale a alternativa que apresenta a sequencia correta de cima para baixo:
( ) Uma árvore AVL é dita balanceada quando, para cada nó da árvore, a diferença entre as alturas das suas sub- árvores (direita e esquerda) não é maior do que um.
( ) Caso a árvore não esteja balanceada é necessário seu balanceamento através da rotação simples ou rotação dupla.

Assinale a alternativa correta:
A) F-F
B) F-V
C) V-F
D) V-V

***Questão 3 (CESGRANRIO - 2018):.*** Em uma árvore AVL com grande quantidade de nós, o custo para inclusão de um nó no meio da árvore é proporcional a
A) $\log(n)$
B) $n$
C) $n \log(n)$
D) $n^2$
D) $n^2 \log(n)$

***Questão 4 (FGV - 2009):.*** Acerca das estruturas de dados Árvores, analise as afirmativas a seguir.

I. A árvore AVL é uma árvore binária com uma condição de balanço, porém não completamente balanceada.

II. Árvores admitem tratamento computacional eficiente quando comparadas às estruturas mais genéricas como os grafos.

III. Em uma Árvore Binária de Busca, todas as chaves da subárvore esquerda são maiores que a chave da raiz.

Assinale:
A) se somente a afirmativa I estiver correta.
B) se somente as afirmativas I e II estiverem corretas.
C)se somente as afirmativas I e III estiverem corretas.
D) se somente as afirmativas II e III estiverem corretas.
E) se todas as afirmativas estiverem corretas.

# Referências Bibliográficas

[1] Jayme Luiz Szwarcfiter and Lilian Markenzon. Estruturas de dados e seus algoritmos (3a. ediçao). *Editora LTC*, 2010.