

estruturas de dados

algoritmos, análise da
complexidade e
implementações em
JAVA e C/C++

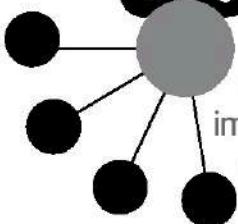
Ana Fernanda
Gomes Ascencio
&
Graziela Santos
de Araújo

PEARSON

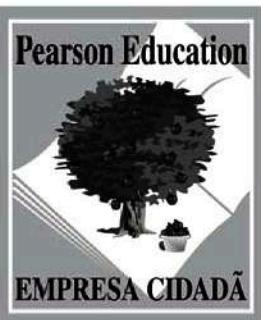


Companion
Website

estruturas de dados

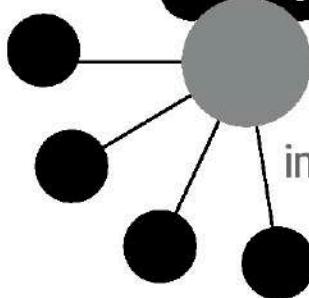


algoritmos, análise da
complexidade e
implementações em
JAVA e C/C++



**Ana Fernanda Gomes Ascencio
& Graziela Santos de Araújo**

estruturas de dados



algoritmos, análise da
complexidade e
implementações em
JAVA e C/C++



São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha
Guatemala México Peru Porto Rico Venezuela

“... plante seu próprio jardim e embeleze a sua própria alma, em vez de esperar que alguém lhe traga flores...”

William Shakespeare

“Aos meus filhos, Eduardo e Pedro, amores eternos e verdadeiros.”

Ana Fernanda Gomes Ascencio

“Ao meu pai (*in memoriam*), que em todos os dias de sua vida semeou amor, sabedoria, compreensão, paciência, admiração e apoio incondicional.”

Graziela Santos de Araújo

© 2010 by Pearson Education do Brasil

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Pearson Education do Brasil.

Diretor editorial: Roger Trimer

Gerente editorial: Sabrina Cairo

Supervisor de produção editorial: Marcelo Françozo

Editor de desenvolvimento: Yuri Bileski

Editora plena: Arlete Sousa

Preparação: Renata Nakano

Revisão: Vanessa de Paula

Capa: Alexandre Mieda

Editoração eletrônica e diagramação: Globaltec

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Ascencio, Ana Fernanda Gomes

Estruturas de dados : algoritmos, análise da complexidade e implementações em JAVA e C/C++ / Ana Fernanda Gomes Ascencio, Graziela Santos de Araújo. – São Paulo : Pearson Prentice Hall, 2010.

ISBN 978-85-7605-881-6

1. Algoritmos 2. C/C++ (Linguagens de programação para computadores) 3. Dados - Estruturas (Ciência da computação) 4. Java (Linguagem de programação para computador) 5. Software - Desenvolvimento
I. Araújo, Graziela Santos de. II. Título.

10-10686

CDD-005.1

Índices para catálogo sistemático:

1. Algoritmos : Computadores : Programação : Processamento de dados 005.1
2. Estrutura de dados : Computadores : Processamento de dados 005.1

2011

Direitos exclusivos para a língua portuguesa cedidos à

Pearson Education do Brasil,

uma empresa do grupo Pearson Education

Rua Nelson Francisco, 26, Limão

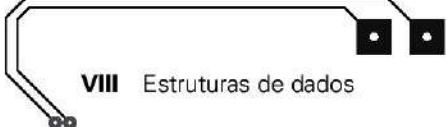
CEP: 02712-100 – São Paulo – SP

Tel: (11) 2178-8686 – Fax: (11) 2178-8688

e-mail: vendas@pearson.com

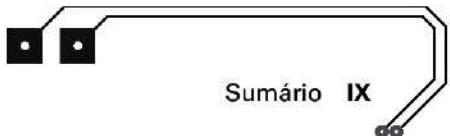
Sumário

Prefácio	11
Introdução	13
Capítulo 1 Análise de algoritmo	1
Algoritmo	1
Estrutura de dados	1
O que é análise de algoritmos.....	2
Elementos da análise assintótica	5
Notação O	7
Notação Ω	8
Notação Θ	8
Notação δ	9
Notação ω	9
Recorrências	10
Notações, funções e somatórios.....	16
Capítulo 2 Algoritmos de ordenação e busca.....	21
Algoritmo de ordenação por troca (<i>BUBBLE SORT</i>)	21
Análise da complexidade.....	26
<i>BUBBLE SORT</i> melhorado (1 ^a versão).....	27
Análise da complexidade.....	32
<i>BUBBLE SORT</i> melhorado (2 ^a versão).....	33
Análise da complexidade.....	37
Algoritmo de ordenação por inserção (<i>INSERTION SORT</i>)	38
Análise da complexidade.....	43
Algoritmo de ordenação por seleção (<i>SELECTION SORT</i>).....	44
Análise da complexidade.....	51

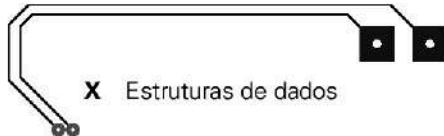


VIII Estruturas de dados

Algoritmo de ordenação por intercalação (<i>MERGE SORT</i>)	53
Análise da complexidade.....	59
Algoritmos de ordenação rápida (<i>QUICK SORT</i>)	61
Análise da complexidade.....	72
Algoritmo de ordenação (<i>heap sort</i>)	74
Análise da complexidade.....	89
Algoritmo de busca sequencial	90
Análise da complexidade.....	97
Algoritmo de busca binária	98
Análise da complexidade.....	102
Exercícios	103
Capítulo 3 Estrutura de dados do tipo listas.....	105
Lista simplesmente encadeada e não ordenada.....	106
Lista simplesmente encadeada e ordenada	117
Lista duplamente encadeada e não ordenada	130
Lista duplamente encadeada e ordenada.....	142
Listas circulares	155
Análise da complexidade.....	180
Inserção no início da lista	180
Inserção no fim da lista	180
Inserção ordenada	180
Consultar toda a lista.....	180
Remoção.....	181
Esvaziar a lista	181
Exercícios	181
Capítulo 4 Estruturas de dados do tipo pilha e fila	183
Pilha	184
Análise da complexidade.....	191
Fila.....	191
Análise da complexidade.....	197
Exercícios	198
Capítulo 5 Estrutura de dados do tipo lista de prioridades ...	201
<i>Heap</i> máximo e <i>heap</i> mínimo	202
Análise da complexidade das estruturas <i>heap</i> máximo e <i>heap</i> mínimo	215
<i>Heap max-min</i> e <i>heap min-max</i>	216
Análise da complexidade da estrutura <i>heap max-min</i> e <i>heap min-max</i>	242
Exercícios	243



Capítulo 6 Estrutura de dados do tipo tabela <i>hashing</i>.....	244
Tabela <i>hashing</i> implementada com endereçamento aberto	245
Tentativa linear.....	245
Análise da complexidade da tabela <i>hashing</i> com tentativa linear.....	256
Tentativa quadrática	257
Análise da complexidade da tabela <i>hashing</i> com tentativa quadrática	267
Tabela <i>hashing</i> implementada com lista	267
Análise da complexidade da tabela <i>hashing</i> implementada com lista.....	276
Exercícios	276
Capítulo 7 Estruturas de dados do tipo árvore.....	278
Árvore binária	278
Análise da complexidade.....	328
Árvore AVL	329
Análise da complexidade.....	366
Exercícios	366
Capítulo 8 Algoritmos em grafos.....	368
Conceitos de grafos.....	368
Laços, arestas múltiplas e multigrafo.....	369
Grafo trivial e grafo simples	369
Grafo completo.....	369
Grafo regular	369
Subgrafo	370
Grafo bipartido	370
Caminho simples, trajeto e ciclos	371
Grafo conexo e desconexo	371
Grafos isomorfos.....	371
Grafo ponderado	372
Dígrafo.....	372
Árvores	373
Corte de vértice e aresta	374
Representação de grafos	375
Matriz de adjacências	375
Matriz de incidências.....	376
Lista de adjacências	377
Algoritmos de busca	378
Algoritmo de busca em profundidade	378
Análise da busca em profundidade	390



X Estruturas de dados

Algoritmo de busca em largura	390
Análise da busca de profundidade.....	408
Algoritmo do caminho mínimo	408
Análise do algoritmo do caminho mínimo.....	425
Exercícios	425
Referências.....	427
Índice remissivo.....	429
Sobre as autoras.....	433

Prefácio

Os seres humanos, além de todas as características biológicas necessárias à sua sobrevivência, necessitam desenvolver extrema habilidade de adaptação a mudanças. As informações chegam até nós de forma assustadoramente rápida. Diariamente, incontáveis novas fontes de informação são disponibilizadas pela Internet, novos equipamentos são colocados no mercado e novos meios de produção são definidos. Aos olhos da maioria das pessoas, este intenso movimento pode causar fascinação, deslumbramento e, às vezes, desconforto e insegurança, uma vez que essas mudanças se refletem em todos os contextos econômicos e sociais.

Muitas vezes lemos e ouvimos que os profissionais da área de computação devem estar em constante atualização para, cada vez mais, oferecerem ao mundo tais inovações. Isso é uma grande verdade! Contudo, aos olhos da ciência da computação, é imprescindível que esses profissionais (e, principalmente, seus professores) também não se esqueçam de um dos pilares que dão sustentação à sua formação: a matemática. É ela que torna possível a criação e a avaliação de novos modelos, ponto inicial do processo de desenvolvimento de todas as inovações tecnológicas.

A proposta desta obra é o estudo de conteúdos clássicos de cursos da área de computação (também chamada de informática, em algumas regiões). Para as autoras, não basta apresentar a solução para um problema, o importante é apresentar a melhor solução para ele, ou, em muitos casos, a solução mais viável para o momento e com os recursos disponíveis, analisando sempre seus pontos positivos e negativos. Esta abordagem crítica não é facilmente aplicada em sala de aula. Fortemente influenciados pelo momento histórico em que vivemos, no qual o imediatismo tem presença marcante, nossos alunos não veem a necessidade da análise de soluções, da busca por melhores respostas. Muitas vezes, quando questionamos uma solução obtida, ouvimos: “Para que pensar em melhorias? Minha resposta resolveu o problema, não resolveu?”.

Neste cenário desafiador, as autoras, que possuem larga experiência como professoras em cursos de graduação e pós-graduação, sabem como poucos justificar a importância e a necessidade de profissionais críticos na área de computação. Este livro enfoca estruturas de dados e análise de algoritmos, e também ilustrações dos funcionamentos dos algoritmos e da análise da complexidade, que constituem o ponto alto da obra, trazendo à tona todo o formalismo necessário para a comprovação ou não da viabilidade das soluções desenvolvidas.

O capítulo 1 conceitua algoritmos, estrutura de dados e análise de algoritmos, explanando, também, os elementos da análise assintótica. No capítulo 2 são apresentados diferentes algoritmos de busca e ordenação, juntamente com ilustração, implementação em Java e C/C++ e respectiva análise de complexidade. Os capítulos 3, 4, 5, 6 e 7 apresentam diferentes estruturas de dados, acompanhadas de ilustrações, da análise da complexidade e de suas implementações desenvolvidas em Java e C/C++. Por fim, o capítulo 8 conceitua grafos, apresenta alguns algoritmos e analisa as implementações desenvolvidas em Java e em C/C++.

Considerando que este livro é indicado, principalmente, para alunos de cursos de graduação, a proposta de levá-los à análise de diferentes soluções para um mesmo problema contribui para o desenvolvimento de profissionais críticos, que adotarão no desempenho de suas atividades profissionais padrões elevados de qualidade, tanto nos produtos desenvolvidos como nos processos de desenvolvimento. Já para os professores, esta obra representará uma importante fonte bibliográfica, capaz de nortear melhor questões pedagógicas e cognitivas, dando nova dinâmica à prática em sala de aula.

Profa. MSc. Edilene A. Veneruchi de Campos

Introdução

Este livro destina-se a acadêmicos dos cursos de graduação e pós-graduação em ciência da computação, engenharia da computação, processamento de dados, sistemas de informação, análise de sistemas e outros que tenham relação com a área de informática, mais especificamente com conceitos da computação.

Os objetivos do livro são: conceituar algoritmos, estruturas de dados e análise de algoritmos; descrever, implementar e analisar algoritmos de busca (sequencial e binária) e ordenação de dados (Bubble sort, Quick sort, Merge sort, Heap sort, Selection sort, Insertion sort); descrever, implementar e analisar estruturas de dados lineares (pilha, fila e listas); descrever, implementar e analisar estruturas de dados não lineares (lista de prioridades – *Heap*, Tabela *Hashing*, Árvore Binária e Árvore AVL) e descrever, implementar e analisar algoritmos em grafos.

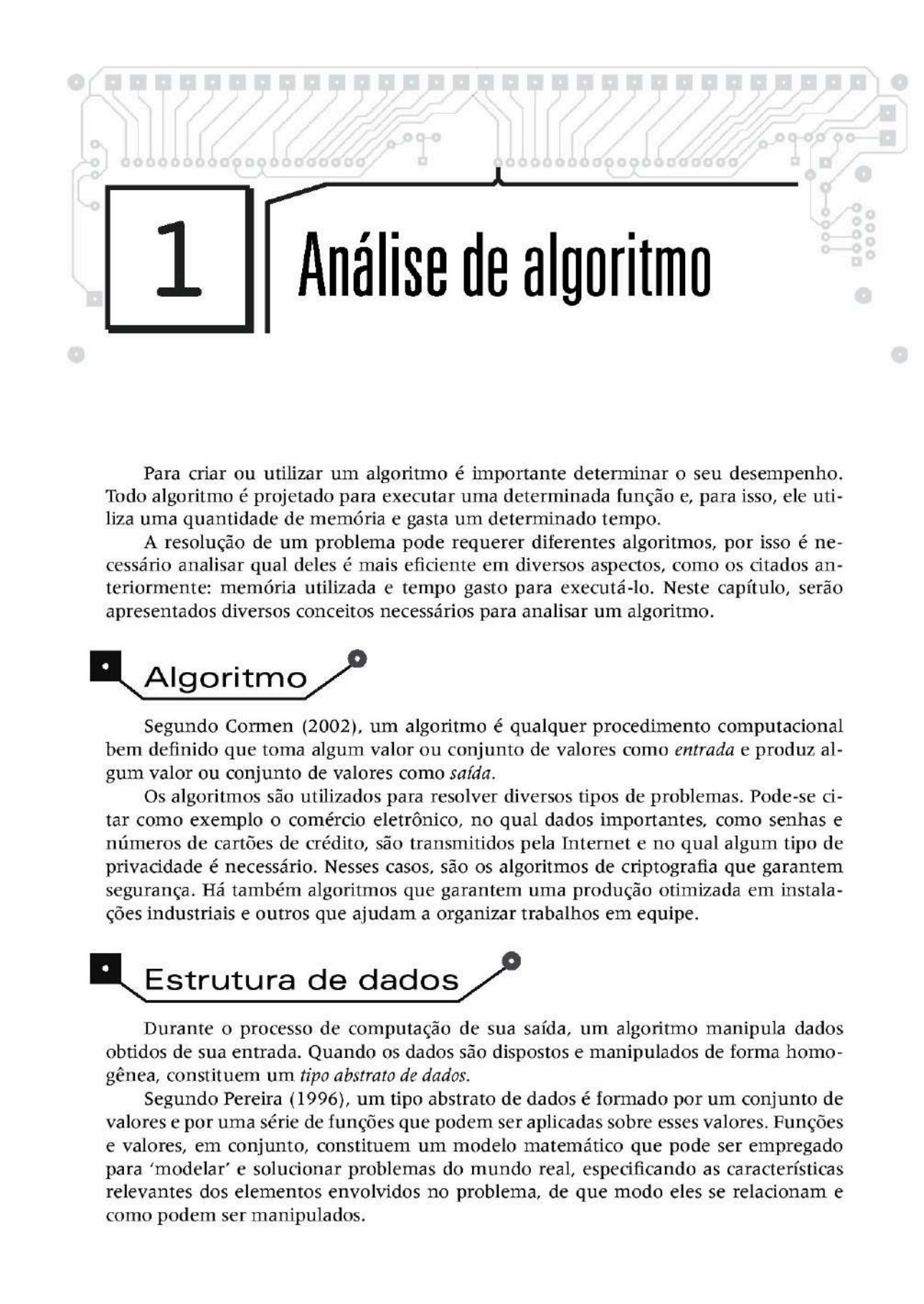
O maior diferencial do livro está na junção do conteúdo de estrutura de dados com o conteúdo de análise de algoritmos, em que para cada estrutura são apresentados os conceitos, ilustrações das operações, implementações em JAVA e em C/C++, bem como o cálculo associado ao tempo de execução das operações.

Assim, é possível utilizar o livro juntamente com o computador e fazer na prática o desenvolvimento de programas que usam as estruturas de dados abordadas no texto, bem como utilizá-lo isoladamente, visualizando o que acontece por meio das representações gráficas.

Companion Website



O Companion Website desta obra (www.prenhall.com/ascencio_araujo_br) oferece, para professores, apresentações em Power Point e, para estudantes, exercícios resolvidos utilizando códigos-fonte.

A decorative background featuring a light gray circuit board with various electronic components like resistors, capacitors, and inductors. The board has a complex network of silver-colored tracks and small blue and grey components.

1

Análise de algoritmo

Para criar ou utilizar um algoritmo é importante determinar o seu desempenho. Todo algoritmo é projetado para executar uma determinada função e, para isso, ele utiliza uma quantidade de memória e gasta um determinado tempo.

A resolução de um problema pode requerer diferentes algoritmos, por isso é necessário analisar qual deles é mais eficiente em diversos aspectos, como os citados anteriormente: memória utilizada e tempo gasto para executá-lo. Neste capítulo, serão apresentados diversos conceitos necessários para analisar um algoritmo.

Algoritmo

Segundo Cormen (2002), um algoritmo é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como *entrada* e produz algum valor ou conjunto de valores como *saída*.

Os algoritmos são utilizados para resolver diversos tipos de problemas. Pode-se citar como exemplo o comércio eletrônico, no qual dados importantes, como senhas e números de cartões de crédito, são transmitidos pela Internet e no qual algum tipo de privacidade é necessário. Nesses casos, são os algoritmos de criptografia que garantem segurança. Há também algoritmos que garantem uma produção otimizada em instalações industriais e outros que ajudam a organizar trabalhos em equipe.

Estrutura de dados

Durante o processo de computação de sua saída, um algoritmo manipula dados obtidos de sua entrada. Quando os dados são dispostos e manipulados de forma homogênea, constituem um *tipo abstrato de dados*.

Segundo Pereira (1996), um tipo abstrato de dados é formado por um conjunto de valores e por uma série de funções que podem ser aplicadas sobre esses valores. Funções e valores, em conjunto, constituem um modelo matemático que pode ser empregado para ‘modelar’ e solucionar problemas do mundo real, especificando as características relevantes dos elementos envolvidos no problema, de que modo eles se relacionam e como podem ser manipulados.

2 Estruturas de dados

Um algoritmo é projetado em função de tipos abstratos de dados. Para implementá-los em uma linguagem de programação é necessário representá-los de alguma maneira nessa linguagem, utilizando tipos e operações suportadas pelo computador. Na representação do tipo abstrato de dados, emprega-se uma *estrutura de dados*.

É durante o processo de implementação que a estrutura de armazenamento dos valores é especificada e os algoritmos que desempenharão o papel das funções (operações) são projetados.

Segundo Cormen (2002), uma estrutura de dados é um meio para armazenar e organizar dados com o objetivo de facilitar o acesso e as modificações.

As estruturas diferem umas das outras pela disposição ou manipulação de seus dados, de modo que não se pode separar as estruturas de dados e os algoritmos associados a elas. Todos os problemas a serem resolvidos por algoritmos possuem dados que precisam ser armazenados, e o são em estruturas, de acordo com o contexto do problema. A escolha de um algoritmo para resolver um problema depende também do tipo de estrutura utilizada para armazenamento dos dados. Assim, uma estrutura é escolhida de acordo com as operações que podem ser realizadas sobre ela e com o custo de cada uma dessas operações.

O que é análise de algoritmos

O projeto de um algoritmo deve considerar o desempenho que este terá após sua implementação. Quando um problema é estudado e um projeto de algoritmo deve ser feito, várias soluções podem surgir, e aspectos de tempo de execução e espaço ocupado são pontos muito relevantes na escolha.

Segundo Cormen (2002), analisar um algoritmo significa prever os recursos de que ele necessitará. Em geral, memória, largura de banda de comunicação ou *hardware* de computação são a preocupação primordial, mas frequentemente é o tempo de computação que se deseja medir.

Para Knuth(*apud* Ziviani, 1971), a área da análise de algoritmos possui dois tipos de problemas distintos:

- Análise de um algoritmo particular: calcular o custo de um determinado algoritmo na resolução de um problema específico. Geralmente, algumas características devem ser investigadas, como o número de vezes que cada parte do algoritmo será executada e a quantidade de memória necessária.
- Análise de uma classe de algoritmos: considerando um problema em particular, determinar o algoritmo de menor custo possível para resolvê-lo. Logo, um conjunto de algoritmos para resolver um problema específico é estudado com o propósito de determinar qual o melhor. Para isso, é necessário colocar limites para o tempo de execução dos algoritmos pertencentes a esse conjunto.

O custo de utilização ou tempo de execução de um algoritmo pode ser medido por meio de sua execução em um computador real, anotando-se o tempo. Os resultados obtidos podem ser considerados inadequados levando-se em conta que (1) dependem do compilador, que pode privilegiar algumas construções e outras não, (2) dependem

do *hardware*; e (3) grandes quantidades de memória também interferem no tempo de execução do algoritmo.

Outra forma de medir o custo é por meio do uso de um modelo matemático ou modelo de computação genérico com um único processador, a RAM (*Random Access Machine* – Máquina de Acesso Aleatório). Esse modelo será usado para realizar a análise dos algoritmos deste livro. Nele, as instruções são executadas uma após a outra, sem operações concorrentes. Para Cormen (2002), deve-se definir as instruções do modelo e os custos associados a elas, isso, no entanto, proporciona pouca percepção ao projeto e à análise de algoritmos.

O modelo de RAM contém instruções encontradas em computadores reais, tais como instruções aritméticas (soma, subtração, multiplicação, divisão, piso, teto, resto), de movimentação de dados (carregar, armazenar, copiar), de controle (desvio condicional, chamada e retorno de funções). Como o custo de cada operação não é algo relevante neste momento, considera-se que cada uma das instruções demora um tempo constante. As operações mais significativas dentro de um algoritmo é que contribuem para o seu tempo de execução. Por exemplo, nos algoritmos de ordenação, considera-se o número de comparações entre os elementos do conjunto a ser ordenado e desconsideram-se as operações aritméticas, de atribuição e manipulação de índices, quando existem.

O tempo de execução de um algoritmo será representado por uma função de custo T , onde $T(n)$ é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n . Logo, T é chamada *função complexidade de tempo* do algoritmo. Se $T(n)$ é a medida de memória necessária para a execução do algoritmo, então T é chamada *função de complexidade de espaço*. Conforme Ziviani (2004), é importante enfatizar que $T(n)$ não representa diretamente o tempo de execução, mas o número de vezes que certa operação relevante é executada.

Para aplicar os conceitos vistos até agora, considere o programa representado no box, implementado na linguagem JAVA, para encontrar o menor elemento de um vetor A de inteiros, com n elementos.



```

int calculamenor (int A[], int n)
{
    int i, menor;

    menor = A[0];
    for (i = 1; i < n; i++)
    {
        if (A[i] < menor)
            menor = A[i];
    }
    return menor;
}

```

4 Estruturas de dados

Na função `calculamenor` mostrada, a função de complexidade de tempo $T(n)$ é o número de comparações entre os elementos de A, visto que, para encontrar o menor elemento do vetor, é preciso mostrar que cada um dos $n - 1$ elementos é menor do que algum outro, e para isso gasta-se pelo menos $n - 1$ comparações. Logo, $T(n) = n - 1$.

O tempo de execução de um algoritmo depende principalmente do tamanho da entrada de dados. No exemplo anterior, o tempo de execução é uniforme para qualquer tamanho de entrada n . Há algoritmos, porém, que dependem de outros fatores, como os de ordenação, em que o algoritmo gastará menos tempo se a entrada estiver quase ordenada.

Considere um problema bastante conhecido em computação: realizar uma busca. É comum a ocorrência de problemas em que é preciso buscar um dado a partir de um valor denominado chave, que é único entre todos os dados armazenados. Para procurá-lo, compara-se a chave buscada com a chave armazenada. Isso pode ser feito em um arquivo contendo registros de dados ou até mesmo em um vetor.

Considere um arquivo contendo registros de dados. A busca mais conhecida é a sequencial. Nesse caso, o algoritmo não se comporta de maneira uniforme como no exemplo do cálculo do menor mostrado anteriormente. Em situações como esta, identificam-se três casos: pior caso, melhor caso e caso médio. O *pior caso* corresponde ao maior tempo de execução sobre todas as entradas de tamanho n . O *melhor caso* corresponde ao menor tempo de execução sobre todas as entradas de tamanho n . O *caso médio* corresponde à média dos tempos de execução do algoritmo sobre todas as entradas de tamanho n .

Dessa forma, no caso médio leva-se em consideração uma distribuição de probabilidades, que é uma suposição feita sobre a entrada. Comumente supõe-se que todas as entradas são igualmente prováveis, contudo, como essa suposição nem sempre é verdadeira, a análise do caso médio é mais difícil de se calcular do que as análises de pior e melhor caso. Para ilustrar os três casos, será utilizado o problema da busca sequencial. No entanto, neste livro será apresentado apenas o pior e melhor caso dos tempos de execução dos demais algoritmos.

No problema da busca sequencial, a função de complexidade de tempo T é calculada em função do número de registros consultados no arquivo. Logo, o melhor caso da busca sequencial acontece quando o registro a ser procurado é o primeiro a ser consultado. Já o pior caso acontece quando o registro a ser procurado é o último a ser consultado ou mesmo quando o registro não é encontrado no arquivo.

Portanto, o tempo de execução para a busca sequencial considerando os três casos é:

$$\text{Pior caso: } T(n) = n$$

$$\text{Melhor caso: } T(n) = 1$$

$$\text{Caso médio: } T(n) = (n + 1)/2$$

Para encontrar o tempo do caso médio, considere p_i a probabilidade de procurar o i -ésimo registro. Como para encontrar o i -ésimo registro são necessárias i comparações, temos que:

$$T(n) = 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + \dots + n \cdot p_n$$

Considerando que a probabilidade de cada um dos registros ser encontrado é $1/n$, temos que:

$$\begin{aligned} T(n) &= 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + 3 \cdot \frac{1}{n} + \dots + n \cdot \frac{1}{n} \\ T(n) &= \frac{1}{n} \cdot (1 + 2 + 3 + \dots + n) \quad (\text{PA}) \\ T(n) &= \frac{1}{n} \cdot \left(\frac{n \cdot (n+1)}{2} \right) \\ T(n) &= \frac{n+1}{2}. \end{aligned}$$

Na segunda linha do cálculo anterior, aparece um somatório de termos ($1 + 2 + 3 \dots + n$), que é uma progressão aritmética (PA) de razão 1, cujo conteúdo é apresentado na seção Somatórios, mais adiante. Como se pode observar, o tempo de execução de um algoritmo aumenta à medida que o tamanho da entrada n do problema aumenta. No caso do problema para calcular o menor elemento de um vetor, o número de comparações realizadas aumentará à medida que o tamanho n do vetor aumentar.

Para valores pequenos de n , qualquer algoritmo, mesmo que ineficiente, gastará pouco tempo. O que faz o programador escolher um algoritmo não é o seu desempenho sobre tamanhos de entrada pequenos, mas sim sobre tamanhos de entrada grandes. Logo, sua análise é feita sobre tamanhos de entrada grandes. Estuda-se, então, o comportamento assintótico da função de complexidade de tempo dos algoritmos, sua eficiência *assintótica*. Ou seja, a preocupação é a maneira como o tempo de execução de um algoritmo aumenta, como ele se comporta à medida que o tamanho da entrada aumenta indefinidamente.

Elementos da análise assintótica

Nesta seção são mostrados alguns tipos de notação assintótica. Tais notações são utilizadas para representar o comportamento assintótico das funções de complexidade de tempo dos algoritmos, bem como relacionar o comportamento das funções de complexidade de dois algoritmos.

As notações vistas nesta seção para descrever o tempo de execução assintótico de um algoritmo são definidas em termos de funções cujos domínios são o conjunto dos números naturais $\mathbb{N} = \{0, 1, 2, \dots\}$.

Antes da apresentação das notações, veja a definição a seguir.

Definição: uma função $g(n)$ *domina assintoticamente* outra função $f(n)$ se existem duas constantes positivas c e n_0 tais que, para $n \geq n_0$, temos que $|f(n)| \leq c \cdot |g(n)|$.

O gráfico da Figura 1.1 demonstra o significado da definição.

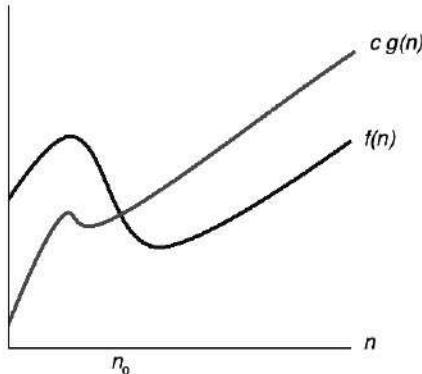
Os exemplos a seguir demonstram a aplicação da definição.

Exemplo 1: Considere $f(n) = n$ e $g(n) = -n^2$. É fácil ver que $|n| \leq c \cdot |-n^2|$ para todo $n \in \mathbb{N}$. Considerando $c = 1$ e $n_0 = 1$, temos que $|n| \leq 1 \cdot |-n^2|$ (conforme se pode observar no

6 Estruturas de dados

Quadro 1.1), e a definição anterior é satisfeita. Portanto, $g(n)$ domina assintoticamente $f(n)$. No entanto, $f(n)$ não domina assintoticamente $g(n)$, pois $|n^2| > c \cdot |n|$, para todo $n > c$ e $n > 1$ e para qualquer valor de c .

Figura 1.1 Notação O



Quadro 1.1 Verificação da equação $|n| \leq c \cdot |-n^2|$

n	$ n \leq c \cdot -n^2 (c = 1)$
1	$1 \leq 1$
2	$2 \leq 4$
3	$3 \leq 9$
4	$4 \leq 16$
...	...

Exemplo 2: Considere $f(n) = (n+1)^3$ e $g(n) = n^3$. Pode-se provar que $f(n)$ domina assintoticamente $g(n)$. Considerando $c = 3$ e $n_0 = 3$, tem-se que $|(n+1)^3| \leq 3 \cdot |n^3|$, para todo $n \geq 3$, como se pode ver no Quadro 1.2. Nesse exemplo, o inverso também é verdadeiro, ou seja, $|n^3| \leq |(n+1)^3|$, para $c = 1$ e $n_0 = 1$.

Quadro 1.2 Verificação da equação $|(n+1)^3| \leq c \cdot |n^3|$

n	$ (n+1)^3 \leq c \cdot n^3 , (c = 3)$
3	$64 \leq 81$
4	$125 \leq 192$
5	$216 \leq 375$
...	...

Notação O

Uma função $f(n)$ é $O(g(n))$ se existem duas constantes positivas c e n_0 tais que $f(n) \leq c \cdot g(n)$, para todo $n \geq n_0$. Pode-se representar também que $f(n) = O(n^2)$ ou $f(n) \in O(n^2)$. Essa representação se aplica às outras duas notações.

A notação O é utilizada para dar um *limite assintótico superior* sobre uma função, dentro de um fator constante. A Figura 1.1 dá uma noção da notação O . Para todos os valores n à direita de n_0 , o valor da função $f(n)$ está em ou abaixo de $g(n)$.

Exemplo 1: Seja $f(n) = \frac{1}{3}n^2 - 3n$, $f(n)$ é $O(n^2)$, quando $c = \frac{1}{3}$, $n_0 = 1$. Outras constantes podem existir, mas o importante é que exista uma escolha para estas duas.

Exemplo 2: Seja $f(n) = (n+1)^2$, $f(n)$ é $O(n^2)$, quando $c = 3$, $n_0 = 2$.

Exemplo 3: Seja $f(n) = 2n^3 + 3n^2 + n$, $f(n)$ é $O(n^3)$, quando $c = 4$, $n_0 = 8$.

Exemplo 4: Seja $f(n) = n$ e $g(n) = n^2$. Pode-se provar que $f(n)$ é $O(g(n))$, ou seja, $n = O(n^2)$, para $c = 1$, $n_0 = 0$. O contrário, no entanto, não é verdadeiro, $g(n)$ não é $O(f(n))$, pois para $n^2 \leq cn$ ser verdadeiro, a condição $c \geq n$ deveria ser verdadeira, para qualquer $n \geq n_0$, mas não existe uma constante c tal que a condição seja satisfeita para qualquer valor de n .

Quando se afirma que o tempo de execução de um algoritmo é $O(n^2)$, significa que existe uma função $f(n)$ que é $O(n^2)$ tal que, para qualquer valor de n , não importando que entrada específica de tamanho n seja escolhida, o tempo de execução sobre essa entrada tem um limite superior determinado pelo valor $c \cdot n^2$.

Na Quadro 1.3 (ZIVIANI, 2004) são mostradas algumas operações que podem ser realizadas com a notação O .

Quadro 1.3 □ Operações com a notação O

$f(n)$	=	$O(f(n))$
$c \times O(f(n))$	=	$O(f(n))$ $c = \text{constante}$
$O(f(n)) + O(f(n))$	=	$O(f(n))$
$O(O(f(n)))$	=	$O(f(n))$
$O(f(n)) + O(g(n))$	=	$O(\max(f(n), g(n)))$
$O(f(n)) \cdot O(g(n))$	=	$O(f(n) \cdot g(n))$
$f(n) \cdot O(g(n))$	=	$O(f(n) \cdot g(n))$

Exemplo: A operação da soma $O(f(n)) + O(g(n))$ pode ser usada para calcular o tempo de execução de uma sequência de trechos de programas. Considere três trechos de programas com tempos de execução $O(n)$, $O(n^2)$ e $O(n \cdot \log n)$. Logo, o tempo de execução dos dois primeiros trechos de programa é $O(\max(n, n^2))$, que é $O(n^2)$. Portanto, o tempo de execução final dos três trechos é $O(\max(n^2, n \cdot \log n))$, que é $O(n^2)$.

Notação Ω

Uma função $f(n)$ é $\Omega(g(n))$ se existem duas constantes positivas c e n_0 tais que $c \cdot g(n) \leq f(n)$, para todo $n \geq n_0$.

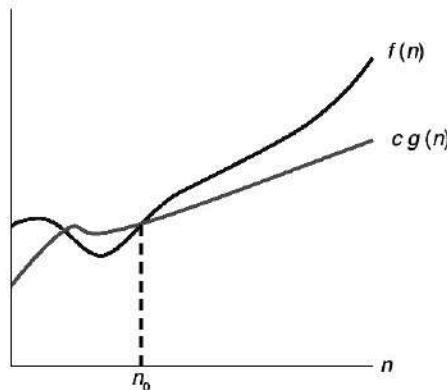
A notação Ω é utilizada para dar um *limite assintótico inferior* sobre uma função, dentro de um fator constante. A Figura 1.2 dá uma noção da notação Ω . Para todos os valores n à direita de n_0 , o valor da função $f(n)$ está em ou acima de $g(n)$.

Exemplo 1: Seja $f(n) = \frac{1}{2}n^2 - 3n$, $f(n)$ é $\Omega(n^2)$, quando $c = \frac{1}{8}$, $n_0 = 8$. Outras constantes podem existir, mas o importante é que existe uma escolha para essas duas.

Exemplo 2: Seja $f(n) = 2n^3 + 3n^2 + n$, $f(n)$ é $\Omega(n^3)$, quando $c = 1$, $n_0 = 1$.

Quando se afirma que o tempo de execução de um algoritmo é $\Omega(n^2)$, significa que independentemente da entrada específica de tamanho n escolhida para cada valor de n , o tempo de execução sobre ela é pelo menos uma constante vezes n^2 , para um valor de n suficientemente grande.

Figura 1.2 Notação Ω

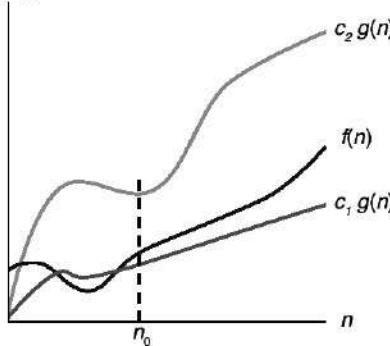


Notação Θ

Uma função $f(n)$ é $\Theta(g(n))$ se existem constantes positivas c_1 , c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$, para todo $n \geq n_0$.

A notação Θ é utilizada para dar um *limite assintótico firme* sobre uma função. A Figura 1.3 proporciona uma noção da notação Θ . Para todos os valores n à direita de n_0 , o valor da função $f(n)$ está sobre ou acima de $c_1 \cdot g(n)$ e sobre ou abaixo de $c_2 \cdot g(n)$. Ou seja, a função $f(n)$ é igual a $g(n)$ a menos de uma constante.

Exemplo: Seja $f(n) = \frac{1}{2}n^2 - 3n$, $f(n)$ é $\Theta(n^2)$, ou seja, $c_1 \cdot n^2 \leq f(n) \leq c_2 \cdot n^2$, quando $c_1 = \frac{1}{8}$, $c_2 = \frac{1}{2}$ e $n_0 = 8$. Outras constantes podem existir, mas o importante é que exista uma escolha para estas três.

Figura 1.3 Notação Θ 

As notações mostradas até aqui são de uso mais frequente, ao passo que as notações a seguir, embora não sejam tão utilizadas, são úteis na análise de alguns algoritmos.

Notação o

A notação o fornece um limite assintótico superior que pode ou não ser assintoticamente restrito. A notação o , também chamada de 'o minúsculo', é utilizada para fornecer um limite superior que não é assintoticamente restrito. Por exemplo, o limite $2n^2 = O(n^2)$ é assintoticamente restrito, mas o limite $2n = O(n^2)$ não. A notação o é definida formalmente como o conjunto

$o(g(n)) = \{f(n) : \text{para qualquer constante positiva } c > 0, \text{ existe uma constante } n_0 > 0 \text{ tal que } 0 \leq f(n) < c \cdot g(n)\} \text{ para todo } n \geq n_0\}$.

Considere as funções $f(n) = 2n$ e $g(n) = n^2$. Os seguintes resultados podem ser verificados: $2n = o(n^2)$, mas $2n^2 \neq o(n^2)$.

As definições das notações O e o , apesar de serem semelhantes, possuem uma principal diferença. Em $f(n) = O(g(n))$, o limite $0 \leq f(n) \leq c \cdot g(n)$ se mantém válido para *alguma* constante $c > 0$. Já em $f(n) = o(g(n))$, o limite $0 \leq f(n) < g(n)$ é válido para *todas* as constantes $c > 0$. O que significa intuitivamente que, na notação o , a função $f(n)$ se torna insignificante em relação a $g(n)$ à medida que n se aproxima do infinito, isto é, $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Notação ω

A notação ω , também chamada de 'ômega minúsculo', é utilizada para fornecer um limite inferior que não é assintoticamente restrito. A notação ω é definida formalmente como o conjunto

$\omega(g(n)) = \{f(n) : \text{para qualquer constante positiva } c > 0, \text{ existe uma constante } n_0 > 0 \text{ tal que } 0 \leq c \cdot g(n) < f(n) \text{ para todo } n \geq n_0\}$.

Por exemplo, $n^2/2 = \omega(n)$, mas $n^2/2 \neq \omega(n^2)$. Segundo Cormen (2002), a relação $f(n) = \omega(g(n))$ significa que

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

se o limite existir. Ou seja, a função $f(n)$ torna-se arbitrariamente grande em relação à função $g(n)$ à medida que n se aproxima do infinito.

Uma vez identificadas quais são as notações geralmente utilizadas para denotar o tempo de execução de um algoritmo, pode-se seguir alguns princípios, listados a seguir, para analisar programas, embora não exista um conjunto completo de regras para fazer tal análise. Segundo Ziviani (*apud* AHO, HOPCROFT e ULLMAN, 1983):

- O tempo de execução de um comando de atribuição, de leitura ou de escrita pode ser considerado como $O(1)$.
- O tempo de execução de uma sequência de comandos é determinado pelo maior tempo de execução de qualquer comando da sequência.
- O tempo de execução de um comando de decisão é composto pelo tempo de execução dos comandos realizados dentro do comando condicional, mais o tempo de avaliar a condição, e equivale a $O(1)$.
- O tempo para executar uma estrutura de repetição é a soma do tempo de execução do corpo da estrutura mais o tempo de avaliar a condição para terminação, multiplicado pelo número de iterações da estrutura. Geralmente, o tempo para avaliar a condição para terminação é $O(1)$.
- Quando o programa possui procedimentos não recursivos, o tempo de execução de cada procedimento deve ser computado separadamente, um a um, iniciando com os procedimentos que não chamam outros procedimentos.
- Quando o programa possui procedimentos recursivos, para cada procedimento é associada uma função de complexidade $T(n)$ desconhecida, onde n mede o tamanho dos argumentos para o procedimento, conforme será mostrado mais adiante.

Recorrências

Uma função que chama a si mesma é dita *recursiva*. Para exemplificar o uso da recursividade será apresentado o problema do fatorial de um número. O fatorial de um número n , para todo $n \geq 0$, é definido por:

$$n! = \begin{cases} 1, & \text{se } n = 0 \text{ ou } n = 1 \\ n * (n-1)!, & \text{se } n > 1. \end{cases}$$

O cálculo de $4!$ é exemplificado no Quadro 1.4.

Quadro 1.4 Exemplo do cálculo recursivo de $4!$

$4! = 4 * 3!$	Para se obter o resultado de $4!$, chama-se a função fatorial novamente para calcular o valor de $3!$.
$3! = 3 * 2!$	Para se obter o resultado de $3!$, chama-se a função fatorial novamente para calcular o valor de $2!$.
$2! = 2 * 1!$	Para se obter o resultado de $2!$, chama-se a função fatorial novamente para calcular o valor de $1!$.
$1! = 1$	O valor de $1!$ é 1 e este resultado será retornado para a chamada da função anterior que completará a sua execução, e esta também retornará até voltar à primeira chamada da função que calculava o valor de $4!$.
$2! = 2 * 1! = 2 * 1 = 2$	
$3! = 3 * 2! = 3 * 2 = 6$	
$4! = 4 * 3! = 4 * 6 = 24$	

Considere a função no box a seguir, implementada em JAVA, que calcula o fatorial de um número. Na função abaixo, verifica-se o valor de n . Caso seja 0 ou 1, já se sabe que o valor do fatorial do número é 1; caso contrário, se o valor de n for maior que 1, calcula-se o fatorial do número recursivamente, como mostrado na linha 6.



```

int factorial (int n)
1{
2 int res;
3 if(n == 0 || n == 1)
4     res = 1;
5 else
6     res = n * factorial(n-1);
7 return res;
8}

```

A análise do tempo de execução de um algoritmo recursivo é um pouco diferente dos algoritmos que não possuem recursividade.

Como visto no começo desta seção, a definição do cálculo do fatorial de um número pode ser representada através do cálculo do fatorial de um número menor. Tal definição é um exemplo de recorrência. Segundo Cormen (2002), uma *relação de recorrência*, ou simplesmente *recorrência*, é uma equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores.

Como a solução do problema de tamanho n depende da solução de problemas de tamanhos menores, o tempo de execução também é representado através do tempo de execução do problema para tamanhos menores. No exemplo do fatorial, o tempo de execução seria representado pela seguinte expressão de recorrência:

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

Analizando a função, verifique que não há estrutura de repetição, há apenas uma condição, atribuições e instrução de retorno de função. Todas essas instruções gastam tempo constante, representados por tempo $O(1)$. A única linha com bastante relevância é a linha de número 6, que possui uma chamada recursiva para a função, com um tamanho de entrada menor. Todas as vezes que a linha 6 é chamada, o tamanho de entrada passado como parâmetro é enviado com o valor decrementado de 1. Portanto, representa-se no tempo de execução por $T(n - 1)$, que indicará o tempo gasto para realizar o cálculo da função para a entrada de tamanho $n - 1$. É importante lembrar que todo algoritmo recursivo tem um caso base, caso este que permite finalizar a recursão. No exemplo do fatorial é quando o valor de $n = 1$ ou $n = 0$. Nesse caso, o custo de realizar o cálculo é um, representado por $T(1) = 1$, visto que se realiza apenas uma comparação, a da linha 3.

Assim como o problema do fatorial, existem outros inúmeros problemas que podem ser resolvidos através de uma solução recursiva. Quando se compara a eficiência de algoritmos, normalmente existe o interesse em comparar as funções que descrevem as suas complexidades, analisando-as de maneira assintótica e utilizando, para isso, as

12 Estruturas de dados

notações O , Ω e Θ . Se uma função $T(n)$ se encontra na forma de uma relação de recorrência, não é possível compará-la com outras funções conhecidas.

A seguir, métodos de solução de recorrências são apresentados.

Expansão telescópica

Segundo Rezende (2002), a ideia básica da expansão telescópica é expandir a relação de recorrência até que possa ser detectado o seu comportamento no caso geral.

Considere a seguinte relação de recorrência:

$$T(n) = T\left(\frac{n}{2}\right) + 1.$$

Para resolvê-la, aplica-se $T\left(\frac{n}{2}\right)$ sobre a fórmula $T(n)$ acima e obtém-se que $T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$. Da mesma maneira, aplica-se $T\left(\frac{n}{4}\right)$ sobre a fórmula $T(n)$ e obtém-se que $T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + 1$. E assim continua-se até encontrar uma fórmula geral. Logo, a solução desta recorrência pela técnica da expansão é mostrada abaixo:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 \\ T(n) &= \left(T\left(\frac{n}{4}\right) + 1\right) + 1 \\ T(n) &= \left(\left(T\left(\frac{n}{8}\right) + 1\right) + 1\right) + 1 \\ T(n) &= \left(\left(\left(T\left(\frac{n}{16}\right) + 1\right) + 1\right) + 1\right) + 1 \\ &\dots \\ T(n) &= \underbrace{\left(\dots\left(\left(T\left(\frac{n}{2^k}\right) + 1 + 1\right) + 1\right) + \dots + 1\right)}_{k-1 \text{ paradas}} \end{aligned}$$

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

$$T(n) = k$$

$$T(n) = \log_2 n$$

$$\therefore T(n) = O(\log n)$$

Uma relação de recorrência sempre é obtida de um algoritmo recursivo, que por sua vez possui um caso base, um caso de parada para a recursão. Suponha, no exemplo acima, que o caso base acontece quando $n = 1$, e então neste caso, $T(1) = 1$. Logo, para que a solução fosse concluída, na 5^a linha de baixo para cima, assume-se que $T\left(\frac{n}{2^k}\right) = T(1) = 1$, então

$$\begin{aligned}\frac{n}{2^k} &= 1 \\ n &= 2^k \\ \log_2 n &= \log_2 2^k \\ \log_2 n &= k,\end{aligned}$$

e a expansão da relação de recorrência se resolve como mostrado anteriormente.

Árvore de recursão

Este método consiste em desenhar uma árvore cujos nós representam os tamanhos dos problemas correspondentes. Cada nível i contém todos os subproblemas de profundidade i .

Dois aspectos importantes devem ser considerados: a altura da árvore e o número de passos executados em cada nível.

A solução da recorrência, que é o tempo de execução do algoritmo, é a soma de todos os passos de todos os níveis.

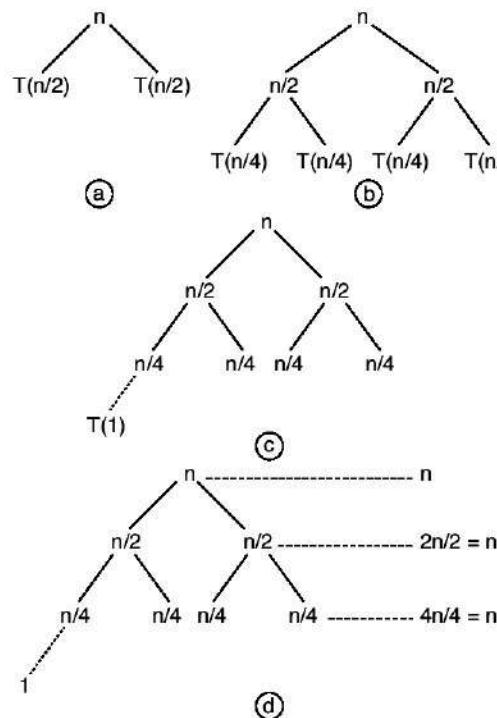
Considere a recorrência $T(n) = 2T\left(\frac{n}{2}\right) + n$ resolvida pelo método da árvore de recursão. Começa-se na primeira iteração representando a expressão de recorrência inicial sob a forma de árvore. Coloca-se o termo n na raiz e as duas chamadas recursivas como filhos da raiz, conforme será visto na Figura 1.4 (a).

Na próxima iteração, cada chamada recursiva será expandida novamente, ou seja, substituindo na expressão de recorrência inicial, pode-se encontrar $T(n/2) = 2T(n/4) + n/2$, conforme veremos na Figura 1.4 (b).

O processo continua até que a recorrência atinja o caso base da recursão, que no caso acima será considerado quando se atinge uma recursão de tamanho 1, ou seja, $T(1)$, conforme Figura 1.4 (c).

Suponha que $T(1) = 1$. Esta árvore de recursão terá vários níveis e o número de níveis dependerá da altura da árvore. Como a solução da recorrência é a soma de todos os passos de todos os níveis, observa-se que cada nível tem um custo n , conforme Figura 1.4 (d). O tempo de execução do algoritmo correspondente à recorrência de entrada é a soma dos passos de todos os níveis, que é $\sum_{i=0}^h n$. É necessário saber o valor de h , ou seja, o valor que corresponde à altura da árvore. Para calcular o valor de h , verifique que o termo $T(n)$ a cada chamada recursiva vai reduzindo, pois vai sendo dividido por 2, até chegar ao valor $T(1)$. Logo, na primeira iteração aparece o termo $T(n/2^1)$, na segunda aparece o termo $T(n/2^2)$, na terceira apareceria o termo $T(n/2^3)$ e assim por diante. Esse termo vai aparecendo até que atinja $T(1)$, portanto, quando $T(n/2^h) = T(1)$, o processo se encerra. Dessa equação, pode-se obter o valor de h , como mostrado a seguir.

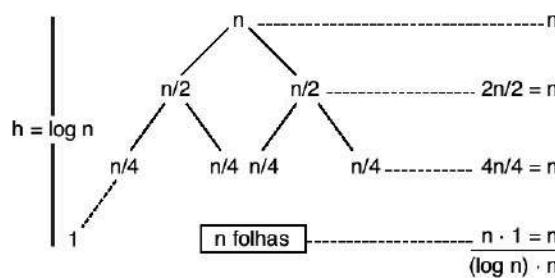
$$\begin{aligned}T\left(\frac{n}{2^h}\right) &= T(1) \\ \frac{n}{2^h} &= 1 \\ n &= 2^h \\ \log_2 n &= \log_2 2^h \\ h &= \log_2 n.\end{aligned}$$

Figura 1.4 Iterações da árvore de recursão

Portanto, o tempo de execução correspondente à recorrência é

$$T(n) = \sum_{i=0}^h n = n \cdot \sum_{i=0}^h 1 = n \cdot (\log_2 n + 1) = O(n \cdot \log_2 n).$$

A Figura 1.5 mostra uma descrição detalhada das iterações e tempo final dessa recorrência pelo método da árvore de recursão.

Figura 1.5 Árvore de recursão

Método master

Este método apresenta um teorema para resolver quase todas as recorrências $T(n)$ que possuam a forma $a \cdot T\left(\frac{n}{b}\right) + f(n)$, sendo $a, b > 1$. Este método é utilizado como uma 'receita' para resolver recorrências.

O método possui três casos:

1. Se $f(n) = O(n^{\log_b a - \varepsilon})$, para alguma constante $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$.

2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.

3. Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$, para algum $\varepsilon > 0$ e se $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ para alguma constante $c < 1$ e para todo n suficientemente grande, então $T(n) = \Theta(f(n))$.

A prova dos três casos pode ser encontrada em *Algoritmos: teoria e prática* (COR-MEN, 2002). A seguir, são apresentados três exemplos de recorrências resolvidas pelo Teorema Master.

$$\text{Exemplo 1: } T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n$$

Analizando a recorrência, pode-se identificar que $a = 4$, $b = 2$ e $f(n) = n$. Em seguida, calculando-se $n^{\log_b a}$ obtém-se que $n^{\log_b 4} = n^{\log_2 4} = n^2$.

Como $f(n) = n = O(n^{\log_b a - \varepsilon}) = O(n^{2-\varepsilon})$ para $\varepsilon = 1$, pode-se aplicar o caso 1 do teorema master. Conclui-se então que a solução da recorrência é $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

$$\text{Exemplo 2: } T(n) = T\left(\frac{2n}{3}\right) + 1$$

Analizando a recorrência, pode-se identificar que $a = 1$, $b = \frac{3}{2}$ e $f(n) = 1$. Em seguida, calculando-se $n^{\log_b a}$ obtém-se que $n^{\log_b 1} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$.

Como $f(n) = 1 = \Theta(n^{\log_b a}) = \Theta(1)$, pode-se aplicar o caso 2 do teorema master. Conclui-se então que a solução da recorrência é $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(1 \cdot \log n) = \Theta(\log n)$.

$$\text{Exemplo 3: } T(n) = 9 \cdot T\left(\frac{n}{3}\right) + n^3$$

Analizando a recorrência, pode-se identificar que $a = 9$, $b = 3$ e $f(n) = n^3$. Em seguida, calculando-se $n^{\log_b a}$ obtém-se que $n^{\log_3 9} = n^2$.

Como $f(n) = n^3 = \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{2+\varepsilon})$, para $\varepsilon = 1$, pode-se aplicar o caso 3 do teorema master desde que se prove também a condição $a \cdot f(n/b) \leq c \cdot f(n)$ para alguma constante $c < 1$ e para todo n suficientemente grande. A prova é mostrada a seguir.

$$a \cdot f(n/b) \leq c \cdot f(n)$$

$$9 \cdot f(n/3) \leq c \cdot f(n), \rightarrow f(n) = n^3$$

$$9 \cdot \frac{n^3}{27} \leq c \cdot n^3$$

$$\frac{1}{3}n^3 \leq c \cdot n^3, c = \frac{1}{3}, n \geq 1.$$

Portanto, de acordo com o caso 3, a solução da recorrência é $T(n) = \Theta(f(n)) = \Theta(n^3)$.

Notações, funções e somatórios

Esta seção revisará notações e funções matemáticas e somatórios que podem ser úteis para a análise de algoritmos.

Monotonicidade

Segundo Cormen (2002), uma função $f(n)$ é *monotonicamente crescente* se $a \leq b$ produz $f(a) \leq f(b)$. Do mesmo modo, ela é monotonicamente decrescente se $a \leq b$ produz $f(a) \geq f(b)$. Uma função $f(n)$ é *estritamente crescente* se $a < b$ produz $f(a) < f(b)$ e *estritamente decrescente* se $a < b$ produz $f(a) > f(b)$.

Pisos e tetos

Seja x um número pertencente ao conjunto dos números reais.

Para qualquer número x , denota-se por $\lfloor x \rfloor$ (piso de x) o maior inteiro menor ou igual a x .

Para qualquer número x , denota-se por $\lceil x \rceil$ (teto de x) o menor inteiro maior ou igual a x .

Para todo número x ,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1. \quad (\text{Equação 1.1})$$

Polinômios

Segundo Cormen (2002), dado um inteiro não negativo d , um polinômio em n de grau d é uma função $p(n)$ da forma

$$p(n) = \sum_{i=0}^d a_i n^i, \quad (\text{Equação 1.2})$$

onde as constantes a_0, a_1, \dots, a_d são os coeficientes do polinômio e $a_d \neq 0$. Um polinômio é assintoticamente positivo somente se $a_d > 0$. No caso de um polinômio assintoticamente positivo $p(n)$ de grau d , tem-se que $p(n) = \Theta(n^d)$.

Seja c um número pertencente ao conjunto dos números reais. Para qualquer constante $c \geq 0$, a função n^c é monotonicamente crescente, e para qualquer constante $c \leq 0$, a função n^c é monotonicamente decrescente. Pode-se dizer que uma função $f(n)$ é *polinomialmente limitada* se $f(n) = O(n^k)$ para alguma constante k .

Exponenciais

Sejam a, m e n números pertencentes ao conjunto dos números reais, para todos os valores $a \neq 0$, m e n , as seguintes identidades são válidas:

Quadro 1.5 Exponenciais

a^0	=	1
a^1	=	a
a^{-1}	=	$1/a$
$(a^m)^n$	=	a^{mn}
$(a^m)^n$	=	$(a^n)^m$
$a^m a^n$	=	a^{m+n}

Para todo n e $a \geq 1$, a função a^n é monotonicamente crescente em n .

As taxas de crescimento de polinômios e exponenciais podem ser relacionadas pelo seguinte fato: para todas as constantes reais a e b tais que $a > 1$,

$$\lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0, \quad (\text{Equação 1.3})$$

da qual conclui-se que

$$n^b = o(a^n)$$

Portanto, qualquer função exponencial com uma base estritamente maior que 1 cresce mais rapidamente que qualquer função polinomial.

Logaritmos

Em todo o livro, as seguintes notações serão utilizadas:

Quadro 1.6 ■ Notações logarítmicas

$\lg n = \log_2 n$	(logaritmo binário)
$\log n = \log_2 n$	(logaritmo binário)
$\log^k n = (\log n)^k$	(exponenciação)
$\log \log n = \log(\log n)$	(composição)

Outra notação importante é que a função logarítmica se aplica apenas ao próximo termo na fórmula. Dessa maneira, $\log n - k$ significa $(\log n) - k$ e não $\log(n - k)$. A função $\log_b n$ será estritamente crescente para $n > 0$ se o valor de $b > 1$ se mantiver constante.

Algumas propriedades de logaritmos são lembradas no Quadro 1.7. Considere para todo $a > 0$, $b > 0$, $c > 0$ e n real:

Quadro 1.7 ■ Propriedades de logaritmos

$\log_b 1$	=	0
$\log_b b$	=	1
a	=	$b^{\log_b a}$
$\log_c(ab)$	=	$\log_c a + \log_c b$
$\log_b a^n$	=	$n \log_b a$

$$\log_c a = \frac{\log_b a}{\log_b c} \quad (\text{Equação 1.4})$$

$$\log_b(1/a) = -\log_b a \quad (\text{Equação 1.5})$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Segundo Cormen (2002), conforme Equação 1.4, a mudança de base de um logaritmo de uma constante para outra só altera o valor do logaritmo por um fator constante, e assim o uso da notação “ $\lg n$ ” será frequente quando fatores constantes não forem importantes, como na notação O . Ainda segundo esse autor, os cientistas da computação consideram 2 a base mais natural para logaritmos, pois muitos algoritmos e estruturas de dados envolvem a divisão de um problema em duas partes.

Dizemos que uma função $f(n)$ é *polilogaritmicamente limitada* se $f(n) = O(\lg^k n)$ para alguma constante k . Para relacionar o crescimento de polinômios e polilogaritmos, substitua n por $\lg n$ e a por 2^a na Equação 1.3, resultando em:

$$\lim_{n \rightarrow \infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n \rightarrow \infty} \frac{\lg^b n}{n^a} = 0$$

A partir desse resultado, conclui-se que

$$\lg^b n = o(n^a)$$

para qualquer constante $a > 0$. Portanto, qualquer função polinomial positiva cresce mais rapidamente que qualquer função polilogarítmica.

Somatários

Os somatórios aparecem em todas as partes da matemática. Na análise de algoritmos, quando o tempo de execução de um algoritmo é calculado, é comum aparecer um somatório de termos durante a determinação do número de vezes que o conjunto de instruções mais importante do algoritmo é executado. Portanto, a seguir são apresentados alguns dos principais somatórios que aparecem e podem ser utilizados para encontrar o tempo de execução dos algoritmos. O termo “soma” também será utilizado para indicar um somatório.

Dada a sequência de números a_1, a_2, \dots, a_n , a soma finita $a_1 + a_2 + \dots + a_n$ pode ser escrita como:

$$\sum_{k=1}^n a_k.$$

Se $n = 0$, o valor do somatório é definido ser 0. Se n não é um inteiro, assume-se que o limite superior da soma é $|n|$. Da mesma forma, se a soma começa com $k = x$, onde x não é um inteiro, considera-se que o valor inicial, ou limite inferior, para a soma é $|x|$. O valor de uma soma finita é sempre bem definido e seus termos podem ser adicionados em qualquer ordem.

A seguinte soma de termos $1 + 2 + 4 + \dots + 2^n$ pode ser escrita como $\sum_{i=0}^n 2^i$, onde o valor inferior indica que o primeiro valor começa com o expoente $i = 0$, ou seja, $2^0 = 1$ e o último termo da soma é o termo com expoente n , ou seja, 2^n .

Outro exemplo de soma é $1 + \frac{1}{2} + \dots + \frac{1}{n}$, que pode ser reescrito como $\sum_{k=1}^n \frac{1}{k}$.

Um item-chave na manipulação de somas está na habilidade de transformar somas em outras mais simples ou deixá-las mais próximas de uma expressão objetivo. Para

isso, podem ser utilizadas três regras. Considere I qualquer conjunto finito de inteiros. Os somatórios indexados por I podem ser transformados por meio das três regras indicadas no Quadro 1.8.

Quadro 1.8 ■ Transformação de somatórios indexados por orações com somatórios

$\sum_{i \in I} ca_i = c \sum_{i \in I} a_i$	(distributividade)
$\sum_{i \in I} (a_i + b_i) = \sum_{i \in I} a_i + \sum_{i \in I} b_i$	(associatividade)
$\sum_{i \in I} a_i = \sum_{p(i) \in I} a_{p(i)}$	(comutatividade)

A regra de distributividade permite mover constantes para dentro e para fora de um Σ . A regra da associatividade permite quebrar um Σ em duas partes ou combinar dois somatórios em um. A comutatividade permite colocar os termos da soma em qualquer ordem, ou seja, o somatório $1 + 2 + \dots + n$ é igual ao somatório $n + \dots + 2 + 1$.

Dada uma sequência de números a_1, a_2, \dots , a soma infinita $a_1 + a_2 + \dots$ pode ser escrita como:

$$\sum_{k=1}^{\infty} a_k,$$

que é interpretada como

$$\lim_{n \rightarrow \infty} \sum_{k=1}^n a_k.$$

Se o limite não existe, a soma diverge; caso contrário, ela converge. Os termos de uma soma convergente nem sempre podem ser adicionados em qualquer ordem.

A seguir são mostrados alguns somatórios que aparecem com frequência na análise de algoritmos.

a) Série aritmética

Segundo Oliveira (2008), uma sequência de números $a_1 + a_2 + a_3 + \dots + a_n$ tal que $a_i - a_{i-1} = q$ para $1 < k \leq n$ e q constante é chamada de *progressão aritmética* (PA).

Por exemplo, a sequência $(5, 7, 9, 11, 13, 15, 17)$ é uma progressão aritmética, pois os seus elementos são formados pela soma de seu antecessor com o valor constante $q = 2$. A sequência $1 + 2 + 3 + \dots + n = \sum_{k=1}^n k$ é outra progressão aritmética com valor constante $q = 1$.

Dada uma PA finita qualquer, a soma de seus termos é chamada *série aritmética* e o seu cálculo é dado pela fórmula:

$$S_n = \sum_{k=1}^n a_k = \frac{(a_1 + a_n) \cdot n}{2},$$

onde a_1 é o primeiro termo da sequência, a_n é o último termo da sequência e n é o número de elementos da soma.

20 Estruturas de dados

Exemplo: O valor da soma $1 + 2 + 3 + \dots + n$ é $\frac{(1+n) \cdot n}{2} = \frac{(n+n^2)}{2}$.

b) Série geométrica

Segundo Oliveira (2008), uma sequência de números $a_1 + a_2 + a_3 + \dots + a_n$ tal que $a_i/a_{i-1} = q$ para $1 < k \leq n$ e q constante é chamada de *progressão geométrica* (PG).

Por exemplo, a sequência $(1, 2, 4, 8, 16, 32, 64)$ é uma PG com valor constante $q = 2$. A sequência $(5, 15, 45, 135, 405)$ é uma PG com valor constante $q = 3$. A sequência $(2, 1, 1/2, 1/4, 1/8, 1/16)$ é uma PG com valor constante $q = 1/2$.

Dada uma PG finita qualquer, a soma de seus termos é chamada *série geométrica* e seu cálculo é dado pela fórmula:

$$S_n = \sum_{k=1}^n a_k = \frac{a_1 \cdot (q^n - 1)}{q - 1},$$

onde a_1 é o primeiro termo da sequência, q é a razão da progressão e n é o número de termos da PG.

Exemplo: O valor da soma $1 + 2 + 4 + 8 + 16 + 32 + 64$ é

$$\frac{1 \cdot (2^7 - 1)}{2 - 1} = \frac{1 \cdot (128 - 1)}{1} = \frac{127}{1} = 127$$

c) Soma geométrica infinita

Quando o somatório de termos é infinito e $|x| < 1$, a série infinita, decrescente e geométrica é calculada como:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad (\text{Equação 1.6})$$

d) Série harmônica

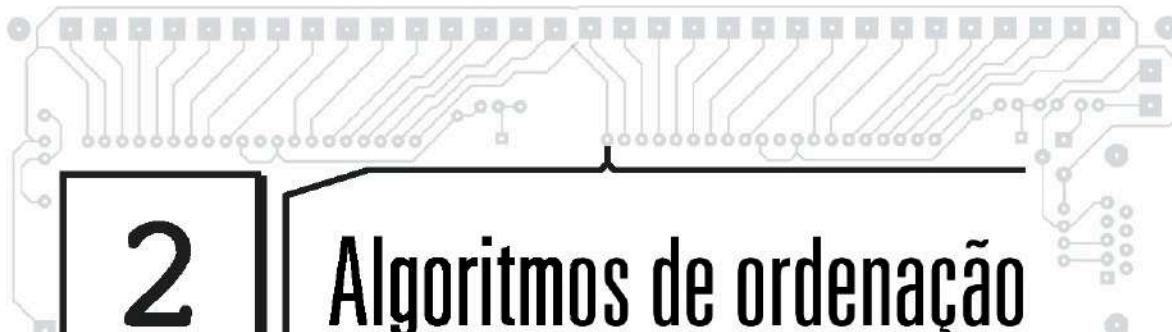
Para inteiros positivos n , o n -ésimo número harmônico é

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} \\ &= \sum_{k=1}^n \frac{1}{k} \\ &= \ln n + O(1) \end{aligned}$$

e) Somas integrais e diferenciais

Fórmulas adicionais podem ser obtidas integrando ou derivando as fórmulas apresentadas. Por exemplo, derivando ambos os lados da soma infinita e geométrica da Equação 1.6 e multiplicando por x , obtém-se

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$$



2

Algoritmos de ordenação e busca

Há situações em que é necessário ordenar dados. Para esse procedimento existem algoritmos de ordenação. As seções a seguir abordam os seguintes algoritmos: *BUBBLE SORT*, *INSERTION SORT*, *SELECTION SORT*, *MERGE SORT*, *QUICK SORT* e *HEAP SORT*. Em outras situações, ocorre a necessidade de encontrar um dado em um conjunto ordenado ou desordenado. Para esse fim, existem os algoritmos de busca descritos nas duas últimas seções deste capítulo.

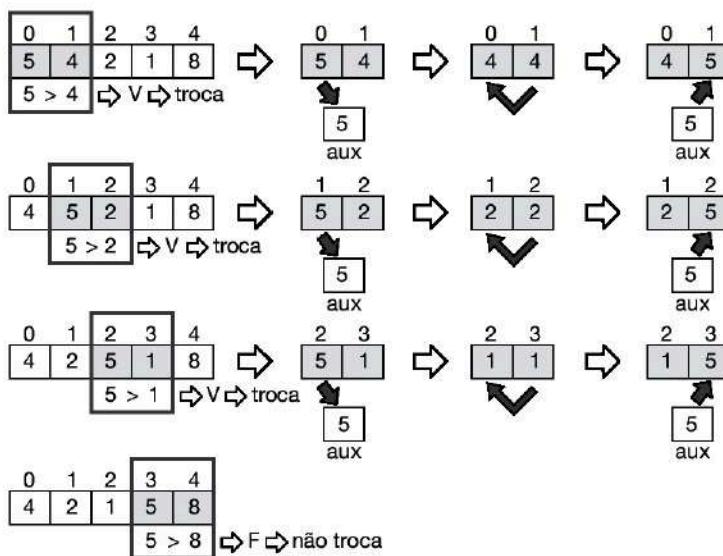
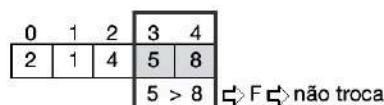
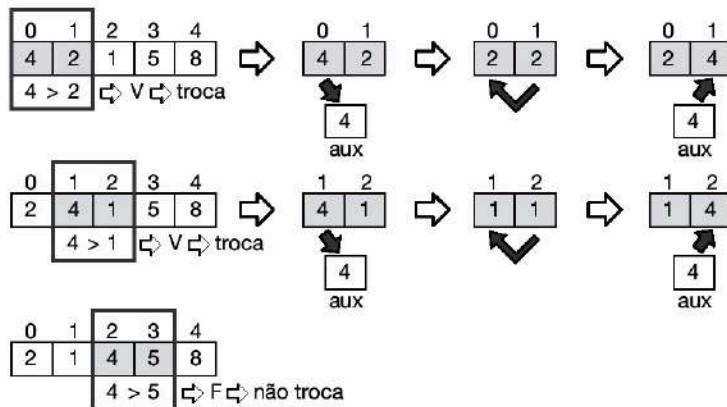
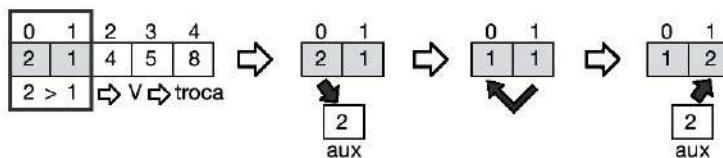
Algoritmo de ordenação por troca (*BUBBLE SORT*)

Neste algoritmo de ordenação serão efetuadas comparações entre os dados armazenados em um vetor de tamanho n . Cada elemento de posição i será comparado com o elemento de posição $i+1$, e quando a ordenação procurada (crescente ou decrescente) é encontrada, uma troca de posições entre os elementos é feita. Assim, um laço com a quantidade de elementos do vetor será executado (`for (j=1; j<=n; j++)`), e dentro deste, um outro laço que percorre da primeira à penúltima posição do vetor (`for (i=0; i<n-1; i++)`). Na próxima página, o algoritmo de ordenação por troca, o *BUBBLE SORT*, será ilustrado, implementado e analisado com o objetivo de ordenar os dados de forma crescente.

A ilustração a seguir mostra a execução do algoritmo *BUBBLE SORT* na ordenação crescente de um vetor com 5 elementos.



ILUSTRAÇÃO

1^a execução do laço2^a execução do laço3^a execução do laço

0	1	2	3	4
1	2	4	5	8
2 > 4	F	não troca		

0	1	2	3	4
1	2	4	5	8
4 > 5	F	não troca		

0	1	2	3	4
1	2	4	5	8
5 > 8	F	não troca		

4^a execução do laço

0	1	2	3	4
1	2	4	5	8
1 > 2	F	não troca		

0	1	2	3	4
1	2	4	5	8
2 > 4	F	não troca		

0	1	2	3	4
1	2	4	5	8
4 > 5	F	não troca		

0	1	2	3	4
1	2	4	5	8
5 > 8	F	não troca		

5^a execução do laço

Apesar de o vetor já estar ordenado, mais uma execução do laço será realizada.

0	1	2	3	4
1	2	4	5	8
1 > 2	F	não troca		

0	1	2	3	4
1	2	4	5	8
2 > 4	F	não troca		

24 Estruturas de dados

0	1	2	3	4
1	2	4	5	8
		4 > 5	F	não troca

0	1	2	3	4
1	2	4	5	8
		5 > 8	F	não troca



ALGORITMO

```
algoritmo
declare X[5], n, i, aux numérico
// carregando os números no vetor
para i ← 0 até 4 faça
    início
        escreva "Digite o ",i+1,"º número: "
        leia X[i]
    fim
// ordenando de forma crescente
// laço com a quantidade de elementos do vetor
para n ← 1 até 5 faça
início
// laço que percorre da primeira à
// penúltima posição do vetor
    para i ← 0 até 3 faça
        início
            se (X[i] > X[i+1])
                então início
                    aux ← X[i]
                    X[i] ← X[i+1]
                    X[i+1] ← aux
                fim
            fim
        fim
// mostrando o vetor ordenado
para i ← 0 até 4 faça
    início
        escreva i+1,"º número: ",X[i]
    fim
fim_algoritmo.
```



```

import java.util.*;
public class bubble_sort
{
    public static void main (String args[])
    {
        int X[] = new int[5];
        int n, i, aux;
        Scanner entrada = new Scanner(System.in);
        // carregando os números no vetor
        for(i=0;i<=4;i++)
        {
            System.out.println("Digite o "+(i+1)+"º número: ");
            X[i] = entrada.nextInt();
        }
        // ordenando de forma crescente
        // laço com a quantidade de elementos do vetor
        for(n=1;n<=5;n++)
        {
            // laço que percorre da primeira à
            // penúltima posição do vetor
            for(i=0;i<=3;i++)
            {
                if (X[i] > X[i+1])
                {
                    aux = X[i];
                    X[i] = X[i+1];
                    X[i+1] = aux;
                }
            }
            // mostrando o vetor ordenado
            for(i=0;i<=4;i++)
            {
                System.out.println((i+1)+"º número: "+X[i]);
            }
        }
    }
}

```



```

#include <iostream.h>
#include <conio.h>
void main()
{
    int X[5], n, i, aux;
    clrscr();
    // carregando os números no vetor

```

```

for(i=0;i<=4;i++)
{
    cout<<"Digite o "<<i+1<<"º número: ";
    cin>>X[i];
}
// ordenando de forma crescente
// laço com a quantidade de elementos do vetor
for(n=1;n<=5;n++)
{
    // laço que percorre da primeira à
    // penúltima posição do vetor
    for(i=0;i<=3;i++)
    {
        if (X[i] > X[i+1])
        {
            aux = X[i];
            X[i] = X[i+1];
            X[i+1] = aux;
        }
    }
    // mostrando o vetor ordenado
    for(i=0;i<=4;i++)
    {
        cout<<i+1<<"º número: "<<X[i]<<"\n";
    }
    getch();
}

```

Análise da complexidade

O principal trecho de código do algoritmo *BUBBLE SORT* é aquele em que a ordenação realmente é realizada. O trecho é mostrado a seguir.



1. para $n \leftarrow 1$ até 5 faça
2. início
3. para $i \leftarrow 0$ até 3 faça
4. início
5. se ($X[i] > X[i+1]$)
6. então início
7. aux $\leftarrow X[i]$
8. $X[i] \leftarrow X[i+1]$
9. $X[i+1] \leftarrow aux$
10. fim

```

11.      fim
12.      fim

```

Em alguns algoritmos de ordenação o fator relevante que determina seu tempo de execução é o número de comparações realizadas. Considerando que o algoritmo foi implementado para um vetor com 5 posições, verifica-se que o número de iterações do primeiro laço é 5. O segundo laço possui 4 iterações, mas como está interno ao primeiro, este será executado 20 vezes (5×4). Logo, o número de comparações (condição se) realizadas será 20.

Aplicando a mesma ideia sobre um algoritmo com um vetor de tamanho n , ele realizará $n(n - 1) = n^2 - n$ comparações (linha 5). Utilizando uma das notações vistas no capítulo anterior, pode-se dizer que o tempo de execução do algoritmo *BUBBLE SORT* é $O(n^2)$, pois

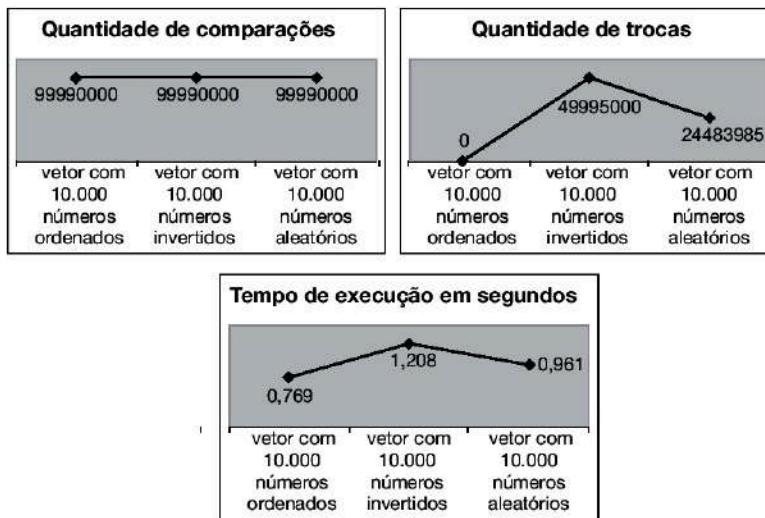
$$n^2 - n \leq cn^2, c = 1, n \geq 1.$$

Nesse algoritmo não há situações melhores ou piores. Qualquer que seja o vetor de entrada, o algoritmo se comportará da mesma maneira, realizando todas as comparações, mesmo que desnecessárias. Veja os gráficos de desempenho na Figura 2.1.

BUBBLE SORT melhorado (1^a versão)

Neste algoritmo de ordenação serão efetuadas comparações entre os dados armazenados em um vetor de tamanho n . Cada elemento de posição i será comparado com o de posição $i-1$, e quando a ordenação procurada (crescente ou decrescente) é encon-

Figura 2.1 Gráficos de desempenho



trada, uma troca de posições entre os elementos é feita. Assim, um laço com a quantidade de elementos do vetor menos um será executado (`for(j=1; j<n; j++)`) e, dentro dele, um outro laço que percorre da última posição à posição j , fazendo com que as posições já ordenadas não sejam mais percorridas (`for(i=n-1; i>=j; i--)`). Abaixo, a 1^a versão melhorada do algoritmo de ordenação por troca *BUBBLE SORT* será ilustrada, implementada e analisada com o objetivo de ordenar os dados de forma crescente de um vetor com 5 elementos.

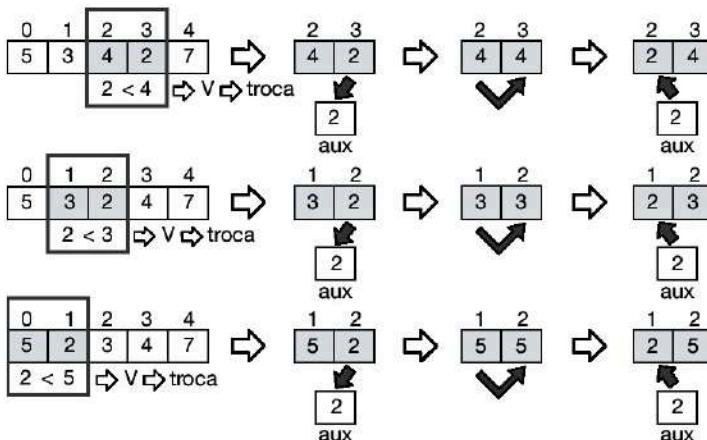


ILUSTRAÇÃO

1^a execução do laço

0	1	2	3	4
5	3	4	2	7

7 < 2 \Rightarrow F \Rightarrow não troca

2^a execução do laço

0	1	2	3	4
2	5	3	4	7

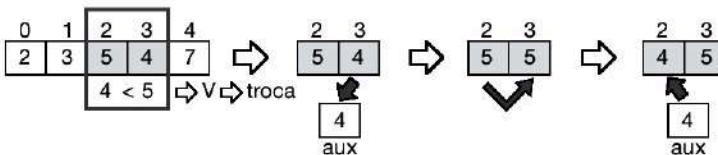
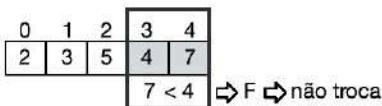
7 < 4 \Rightarrow F \Rightarrow não troca

0	1	2	3	4
2	5	3	4	7

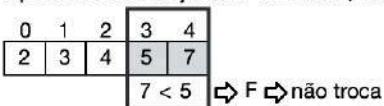
4 < 3 \Rightarrow F \Rightarrow não troca

0	1	2	3	4
2	5	3	4	7

3 < 5 \Rightarrow V \Rightarrow troca
aux

3^a execução do laço4^a execução do laço

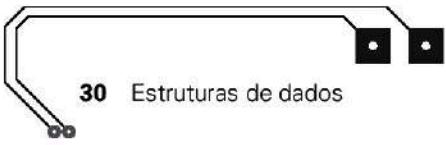
Apesar de o vetor já estar ordenado, mais uma execução do laço será realizada.



```

algoritmo
declare X[5], j, i, aux numérico
// carregando os números no vetor
para i ← 0 até 4 faça
    início
        escreva "Digite o ", i+1, "º número: "
        leia X[i]
    fim
// ordenando de forma crescente
// laço com a quantidade de elemento do vetor - 1
para j ← 1 até 4 faça
início
// laço que percorre da última posição à
// posição j do vetor
    para i ← 4 até j faça passo -1
    início
        se (X[i] < X[i-1])
        então
            aux ← X[i]
            X[i] ← X[i-1]
            X[i-1] ← aux
        fim
    fim
fim

```



30 Estruturas de dados

```
fim
// mostrando o vetor ordenado
para i ← 0 até 4 faça
    início
        escreva i+1,"º número: ",x[i]
    fim
fim_algoritmo.
```



```
import java.util.*;
public class bubble_sort_melhorado1
{
    public static void main (String args[])
    {
        int X[] = new int[5];
        int j, i, aux;
        Scanner entrada = new Scanner(System.in);
        // carregando os números no vetor
        for(i=0;i<=4;i++)
        {
            System.out.println("Digite o "+(i+1)+"º número: ");
            X[i] = entrada.nextInt();
        }
        // ordenando de forma crescente
        // laço com a quantidade de elementos do vetor - 1
        for(j=1;j<=4;j++)
        {
            // laço que percorre da última posição à posição j do vetor
            for(i=4;i>=j;i--)
            {
                if (X[i] < X[i-1])
                {
                    aux = X[i];
                    X[i] = X[i-1];
                    X[i-1] = aux;
                }
            }
        }
        // mostrando o vetor ordenado
        for(i=0;i<=4;i++)
        {
```

```

        System.out.println((i+1)+"º número: "+X[i]);
    }
}
}
}

```



C/C++

```

#include <iostream.h>
#include <conio.h>
void main()
{
    int X[5], j, i, aux;
    clrscr();
    // carregando os números no vetor
    for(i=0;i<=4;i++)
    {
        cout<<"Digite o "<<i+1<<"º número: ";
        cin>>X[i];
    }
    // ordenando de forma crescente
    // laço com a quantidade de elementos do vetor - 1
    for(j=1;j<=4;j++)
    {
        // laço que percorre da última posição à
        // posição j do vetor
        for(i=4;i>=j;i--)
        {
            if (X[i] < X[i-1])
            {
                aux = X[i];
                X[i] = X[i-1];
                X[i-1] = aux;
            }
        }
    }
    // mostrando o vetor ordenado
    for(i=0;i<=4;i++)
    {
        cout<<i+1<<"º número: "<<X[i]<<"\n";
    }
    getch();
}

```

Análise da complexidade

O trecho desta versão do algoritmo *BUBBLE SORT* em que ocorre a ordenação é mostrado a seguir.



ALGORITMO

```

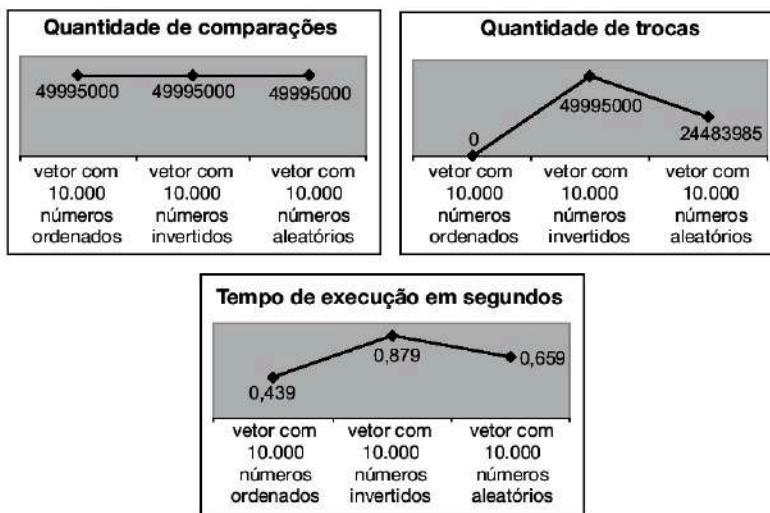
1. para j ← 1 até 4 faça
2.   início
3.     para i ← 4 até j faça passo -1
4.       início
5.         se (X[i] < X[i-1])
6.           então início
7.             aux ← X[i]
8.             X[i] ← X[i-1]
9.             X[i-1] ← aux
10.          fim
11.      fim
12.  fim

```

Nessa versão, o *para* da linha 1 será executado $n - 1$ vezes (4). O segundo *para*, da linha 3, será executado de acordo com o valor de j do primeiro *para*. Logo, o *para* da linha 3 será executado 4 vezes para $j = 1$, 3 vezes para $j = 2$, 2 vezes para $j = 3$ e uma vez para $j = 4$. Ou seja, para o vetor de tamanho 5, o número de comparações realizadas (linha 5), que é interna ao *para* da linha 3, será $(4 + 3 + 2 + 1) = 10$. Considerando um vetor de tamanho n , serão realizadas $((n-1) + \dots + 3 + 2 + 1)$ comparações. Portanto, o tempo de execução do algoritmo é:

$$\begin{aligned}
T(n) &= \left(\sum_{i=1}^{n-1} i \right) \\
&= \frac{(1 + n - 1)(n - 1)}{2} \\
&= \frac{n(n - 1)}{2} \\
&= \frac{n^2 - n}{2} \\
&= O(n^2), \text{ para } c = \frac{1}{2}, n \geq 1.
\end{aligned}$$

Para qualquer vetor de entrada, o algoritmo se comporta da mesma maneira. Veja os gráficos de desempenho na Figura 2.2.

Figura 2.2 Gráficos de desempenho

BUBBLE SORT melhorado (2^a versão)

Neste algoritmo de ordenação serão efetuadas comparações entre os dados armazenados em um vetor de tamanho n . Cada elemento de posição i será comparado com o de posição $i + 1$ e, quando a ordenação que se busca (crescente ou decrescente) é encontrada, uma troca de posições entre os dados é feita. Assim, um laço com a quantidade de elementos do vetor, enquanto houver trocas, será executado ($j = 1$) e ($\text{while } j \leq n \& \text{ troca} == 1$), e dentro dele, outro laço que percorre da primeira à penúltima posição do vetor ($\text{for } (i=0; i < n-1; i++)$). Abaixo, a 2^a versão melhorada do algoritmo de ordenação por troca *BUBBLE SORT* será ilustrada, implementada e analisada com o objetivo de ordenar os dados de forma decrescente.



ILUSTRAÇÃO

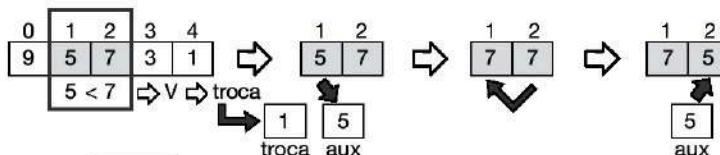
1^a execução do laço
 $n = 1$ e troca = 1

Início \Rightarrow

0

 troca

0	1	2	3	4
9	5	7	3	1
9 < 5	F	não troca		



34 Estruturas de dados

0	1	2	3	4
9	7	5	3	1
5 < 3	F	não troca		

0	1	2	3	4
9	7	5	3	1
3 < 1	F	não troca		

Fim = 1 troca Deve continuar.

2ª execução do laço

n = 2 e troca = 1

Início 0
troca

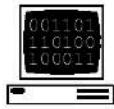
0	1	2	3	4
9	7	5	3	1
9 < 7	F	não troca		

0	1	2	3	4
9	7	5	3	1
7 < 5	F	não troca		

0	1	2	3	4
9	7	5	3	1
5 < 3	F	não troca		

0	1	2	3	4
9	7	5	3	1
3 < 1	F	não troca		

Fim = 0 troca Vetor ordenado, sai do laço.



ALGORITMO

```

algoritmo
declare X[5], n, i, aux, troca numérico
// carregando os números no vetor
para i ← 0 até 4 faça
    início
        escreva "Digite o ",i+1,"º número: "

```

```

leia X[i]
fim
// ordenando de forma decrescente
// laço com a quantidade de elementos do vetor
// e enquanto houver troca
n ← 1
troca ← 1
enquanto (n ≤ 5 E troca = 1) faça
    início
    troca ← 0
    para i ← 0 até 3 faça
        início
        se (X[i] < X[i+1])
        então início
            troca ← 1
            aux ← X[i]
            X[i] ← X[i+1]
            X[i+1] ← aux
        fim
    fim
    n ← n + 1
fim
// mostrando o vetor ordenado
para i ← 0 até 4 faça
    início
    escreva i+1,"º número: ",X[i]
    fim
fim algoritmo.

```



```

import java.util.*;
public class bubble_sort_melhorado2
{
    public static void main (String args[])
    {
        int X[] = new int[5];
        int n, i, aux, troca;
        Scanner entrada = new Scanner(System.in);
        // carregando os números no vetor
        for(i=0;i<=4;i++)
        {
            System.out.println("Digite o "+(i+1)+"º número: ");

```

36 Estruturas de dados

```
        X[i] = entrada.nextInt();
    }
    // ordenando de forma decrescente
    // laço com a quantidade de elementos do vetor
    // e enquanto houver troca
    n = 1;
    troca = 1;
    while (n <= 5 && troca == 1)
    {
        troca = 0;
        for(i=0;i<=3;i++)
        {
            if (X[i] < X[i+1])
            {
                troca = 1;
                aux = X[i];
                X[i] = X[i+1];
                X[i+1] = aux;
            }
        }
        n = n + 1;
    }
    // mostrando o vetor ordenado
    for(i=0;i<=4;i++)
    {
        System.out.println((i+1)+"º número: "+X[i]);
    }
}
```



```
#include <iostream.h>
#include <conio.h>
void main()
{
    int X[5], n, i, aux, troca;
    clrscr();
    // carregando os números no vetor
    for(i=0;i<=4;i++)
    {
        cout<<"Digite o "<<i+1<<"º número: ";
        cin>>X[i];
    }
```

```

// ordenando de forma decrescente
// laço com a quantidade de elementos do vetor
// e enquanto houver troca
n = 1;
troca = 1;
while (n <= 5 && troca == 1)
{
    troca = 0;
    for(i=0;i<=3;i++)
    {
        if (X[i] < X[i+1])
        {
            troca = 1;
            aux = X[i];
            X[i] = X[i+1];
            X[i+1] = aux;
        }
    }
    n = n + 1;
}
// mostrando o vetor ordenado
for(i=0;i<=4;i++)
{
    cout<<i+1<<"º número: "<<X[i]<<"\n";
}
getch();
}

```

Análise da complexidade

O trecho da 2^a versão do algoritmo *BUBBLE SORT* em que ocorre a ordenação é mostrado a seguir.



1. $n \leftarrow 1$
2. $troca \leftarrow 1$
3. enquanto ($n \leq 5$ E $troca = 1$) faça
4. início
5. $troca \leftarrow 0$
6. para $i \leftarrow 0$ até 3 faça
7. início
8. se ($X[i] < X[i+1]$)
9. então início

```

10.          troca ← 1
11.          aux ← X[i]
12.          X[i] ← X[i+1]
13.          X[i+1] ← aux
14.          fim
15.          fim
16.          n ← n + 1
17.          fim

```

Considerando que o algoritmo acima foi implementado para um vetor com 5 posições, verifica-se que o número de iterações da estrutura `enquanto` depende também do número de trocas efetuadas. O segundo para possui 4 iterações (um valor a menos que o tamanho do vetor), mas como está interno ao `enquanto`, este será executado x vezes, onde $x = y * 4$ (onde y é o número de vezes que o `enquanto` é executado). Logo, o número de comparações (condição se, na linha 8) realizadas também será x .

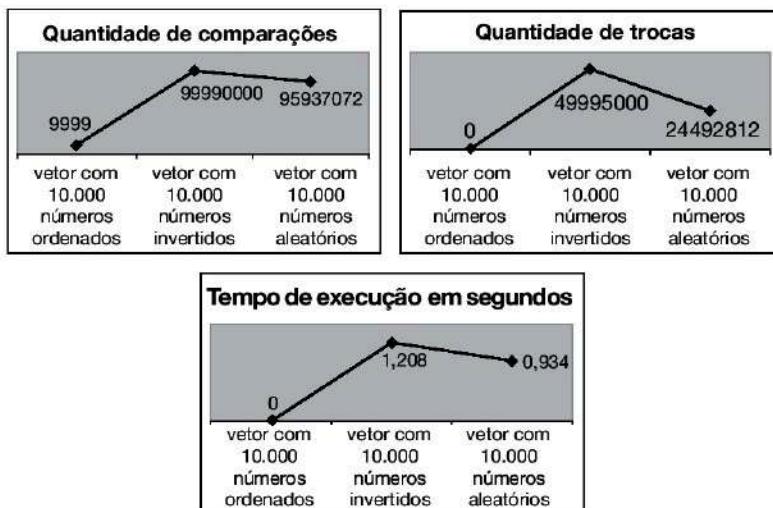
Nessa versão melhorada pode-se verificar a presença do melhor e pior caso. O melhor acontece quando o vetor submetido para ordenação já está ordenado. Logo, nenhuma troca será efetuada e a estrutura `enquanto` será executada uma única vez. Com isso, a estrutura para será executada e o número de comparações realizadas será 4. Considerando que o tamanho do vetor é n , o número de comparações realizadas é $n - 1$. No melhor caso, esse algoritmo gasta $O(n)$.

O pior caso ocorre quando o vetor a ser ordenado está na ordem inversa. Dessa maneira, a estrutura `enquanto` será executada n vezes (onde n é o tamanho do vetor, ($n = 5$)). A estrutura para da linha 6 que possui 4 ($n - 1$) iterações será executada n vezes também. Logo, o número de comparações realizadas na linha 8 será $n(n - 1) = n^2 - n$. Portanto, no pior caso, o tempo de execução desse algoritmo é $O(n^2)$.

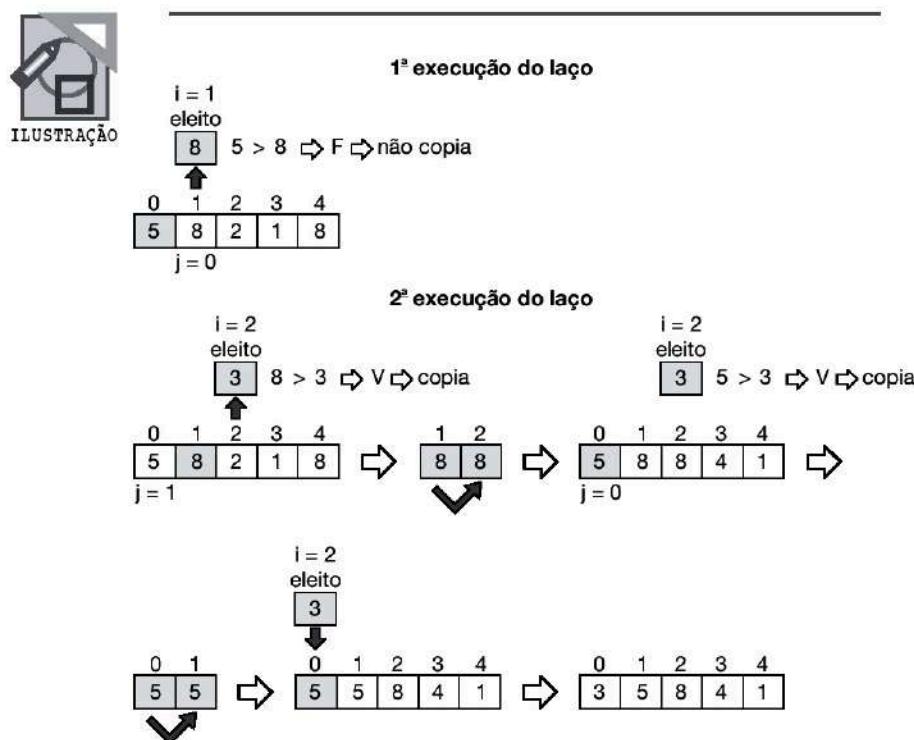
Veja os gráficos de desempenho na Figura 2.3.

Algoritmo de ordenação por inserção (*INSERTION SORT*)

Neste algoritmo de ordenação será eleito o segundo número do vetor para iniciar as comparações. Assim, os elementos à esquerda do número eleito estão sempre ordenados de forma crescente ou decrescente. Logo, um laço com as comparações será executado do segundo elemento ao último, ou seja, na quantidade de vezes igual ao número de elementos do vetor menos um (`for (i=1; i<n; i++)`). Enquanto existirem elementos à esquerda do número eleito para comparações e a posição que atende a ordenação que se busca não for encontrada, o laço será executado. O número eleito está na posição i . Os números à esquerda do eleito estão nas posições de $i-1$ à 0, logo, o laço a ser executado será ($j=i-1$) e (`while (j>=0 && elemento[j] > eleito)`).

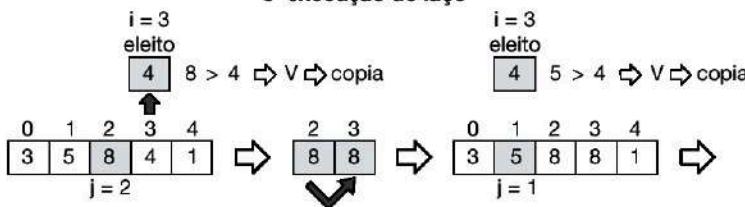
Figura 2.3 Gráficos de desempenho

Abaixo, o algoritmo de ordenação por inserção, *INSERTION SORT*, será ilustrado, implementado e analisado com o objetivo de ordenar os dados de forma crescente.

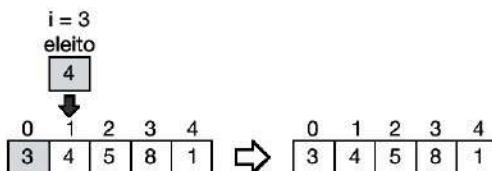
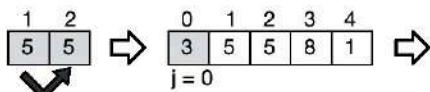


40 Estruturas de dados

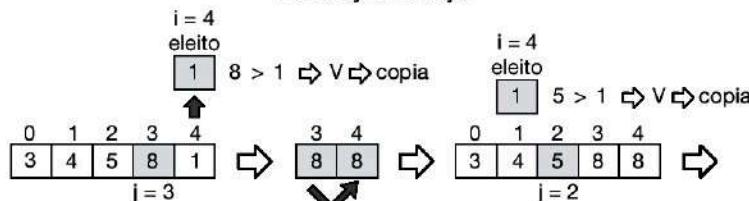
3^a execução do laço



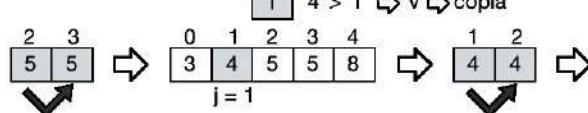
$i = 3$
eleito
4 3 > 4 $\Rightarrow F \Rightarrow$ copia



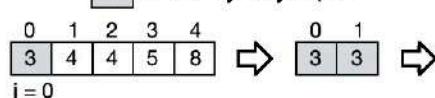
4^a execução do laço



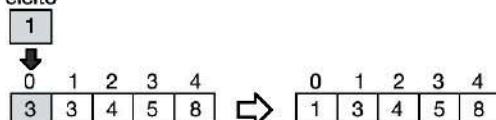
$i = 4$
eleito
1 4 > 1 $\Rightarrow V \Rightarrow$ copia



$i = 4$
eleito
1 3 > 1 $\Rightarrow V \Rightarrow$ copia



$i = 4$
eleito
1





ALGORITMO

```

algoritmo
declare X[5], i, j, eleito numérico
// carregando os números no vetor
para i ← 0 até 4 faça
    início
        escreva "Digite o ",i+1,"º número: "
        leia X[i]
    fim
// ordenando de forma crescente
// laço com a quantidade de elementos do vetor – 1
para i ← 1 até 4 faça
    início
        eleito ← X[i]
        j ← i - 1
        // laço que percorre os elementos
        // à esquerda do número eleito
        // ou até encontrar a posição para
        // recolocação do número eleito
        // respeitando a ordenação procurada
        enquanto (j ≥ 0 E X[j] > eleito)
            início
                X[j+1] ← X[j]
                j ← j - 1
            fim
            X[j+1] ← eleito
        fim
    // mostrando o vetor ordenado
    para i ← 0 até 4 faça
        início
            escreva i+1,"º número: ",X[i]
        fim
fim_algoritmo.

```



J A V A

```

import java.util.*;
public class insertion_sort
{
    public static void main (String args[])
    {
        int X[] = new int[5];
        int i, j, eleito;

```

```

Scanner entrada = new Scanner(System.in);
// carregando os números no vetor
for(i=0;i<=4;i++)
{
    System.out.println("Digite o "+(i+1)+"º número: ");
    X[i] = entrada.nextInt();
}
// ordenando de forma crescente
// laço com a quantidade de elementos do vetor - 1
for(i=1;i<=4;i++)
{
    eleito = X[i];
    j = i - 1;
    // laço que percorre os elementos à
    // esquerda do número eleito
    // ou até encontrar a posição para
    // recolocação do número eleito
    // respeitando a ordenação procurada
    while (j >= 0 && X[j] > eleito)
    {
        X[j+1] = X[j];
        j = j - 1;
    }
    X[j+1] = eleito;
}
// mostrando o vetor ordenado
for(i=0;i<=4;i++)
{
    System.out.println((i+1)+"º número: "+X[i]);
}
}

```



```

#include <iostream.h>
#include <conio.h>

void main()
{
    int X[5];
    int i, j, eleito;
    clrscr();
    // carregando os números no vetor

```

```

for(i=0;i<=4;i++)
{
    cout<<"Digite o "<<i+1<<"º número: ";
    cin>>X[i];
}
// ordenando de forma crescente
// laço com a quantidade de elementos do vetor - 1
for(i=1;i<=4;i++)
{
    eleito = X[i];
    j = i - 1;
    // laço que percorre os elementos à
    // esquerda do número eleito
    // ou até encontrar a posição para
    // recolocação do número eleito
    // respeitando a ordenação procurada
    while (j >= 0 && X[j] > eleito)
    {
        X[j+1] = X[j];
        j = j - 1;
    }
    X[j+1] = eleito;
}
// mostrando o vetor ordenado
for(i=0;i<=4;i++)
{
    cout<<"\n"<<i+1<<"º número: "<<X[i];
}
getch();
}

```

• Análise da complexidade

O trecho do algoritmo em que ocorre a ordenação é mostrado a seguir.



ALGORITMO

1. para $i \leftarrow 1$ até 4 faça
2. início
3. $eleito \leftarrow X[i]$
4. $j \leftarrow i - 1$
5. enquanto ($j \geq 0$ E $X[j] > eleito$)
6. início
7. $X[j+1] \leftarrow X[j]$

```

8.           j ← j - 1
9.       fim
10.      X[j+1] ← eleito
11.     fim

```

No algoritmo anterior existem duas estruturas de repetição. A estrutura para executa $n - 1$ vezes, onde n é o tamanho do vetor. A estrutura de repetição enquanto será executada a princípio também $n - 1$ vezes, porém dependendo do vetor de entrada, as linhas 5, 7 e 8 podem ser executadas menos vezes.

O pior caso do algoritmo ocorre quando o vetor de entrada possui os elementos na ordem inversa, ou seja, se o vetor está em ordem decrescente e busca-se a ordem crescente. Nesse caso, cada elemento $X[j]$ é comparado com cada elemento no subvetor ordenado $X[0..j-1]$, e então executa-se a linha das comparações (linha 5) $j + 2$ vezes para cada valor de $j = 0, 1, \dots, n-2$. Logo, o tempo de execução, que é o número de comparações, é dado pela fórmula:

$$T(n) = 2 + 3 + 4 + \dots + n$$

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

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

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

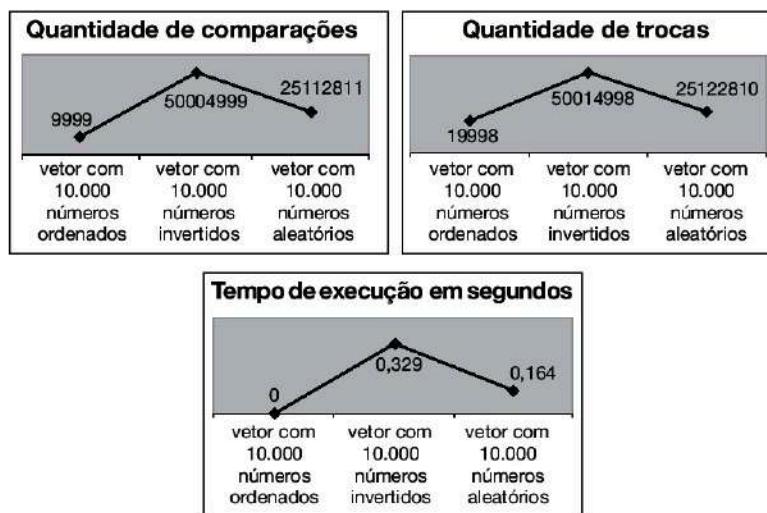
$$T(n) = O(n^2), \text{ para } c = 2, n \geq 1.$$

O melhor caso do algoritmo ocorre quando o vetor de entrada fornecido possui os elementos já ordenados. Para cada $j = 0, 1, \dots, n-2$, tem-se que a condição $X[j] > \text{eleito}$ na linha 5 é falsa. Então, o custo de executar a linha 5 é 1 para cada valor de j e o tempo de execução do melhor caso é $T(n) = O(n - 1)$, já que se tem $n - 1$ valores para j .

Veja os gráficos de desempenho na Figura 2.4.

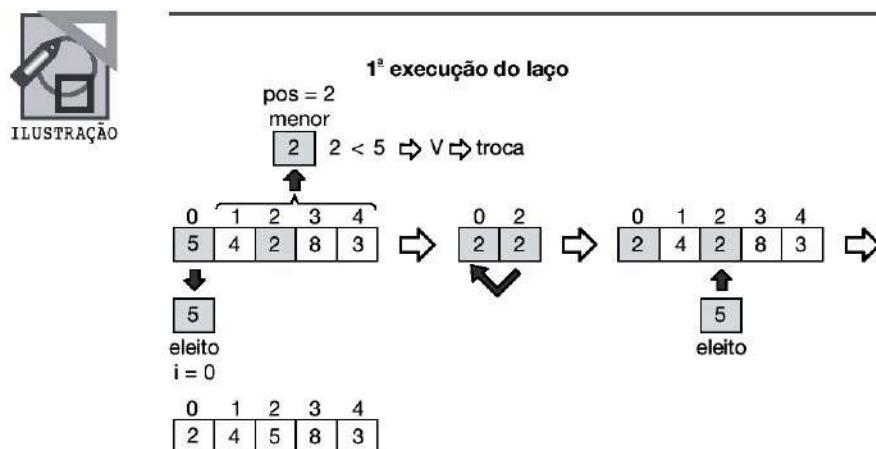
Algoritmo de ordenação por seleção (SELECTION SORT)

Neste algoritmo de ordenação cada número do vetor, a partir do primeiro, será eleito e comparado com o menor ou maior, dependendo da ordenação desejada, número dentre aqueles que estão à direita do eleito. Nessas comparações procura-se um número menor que o eleito (quando a ordenação for crescente) ou um maior que o eleito (quando a ordenação for decrescente). Quando um número satisfaz as condições da ordenação desejada (menor ou maior que o eleito), este trocará de posição com o eleito, assim, todos os números à esquerda do eleito ficam sempre ordenados. Nesse algoritmo, um laço com as comparações será executado do primeiro ao penúltimo ele-

Figura 2.4 Gráficos de desempenho

mento, ou seja, na quantidade de vezes igual ao número de elementos do vetor menos um (`for (i=0; i<n-1; i++)`), pois as comparações são realizadas com os elementos à direita do número eleito, e o número da última posição não tem elementos à direita.

O número eleito está na posição i . Os números à direita do eleito estão nas posições de $i+1$ à $n-1$, sendo n o número de elementos do vetor. Logo, o laço a ser executado para encontrar o menor elemento à direita do eleito será (`for (j=i+2; j<=n-1; j++)`). Lembrando que o primeiro elemento à direita do número eleito começa sendo considerado o menor número. Abaixo, o algoritmo de ordenação por seleção, *SELECTION SORT*, será ilustrado, implementado e analisado com o objetivo ordenar os dados de forma crescente.



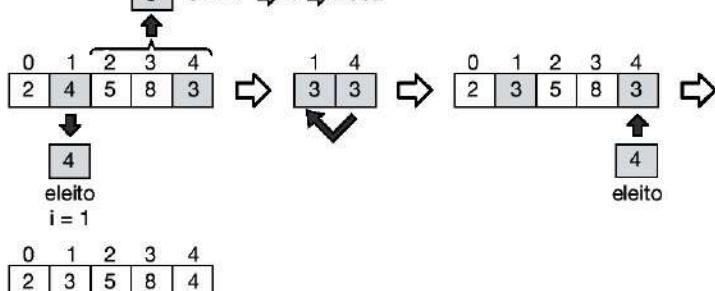
46 Estruturas de dados

2^a execução do laço

pos = 4

menor

3 $3 < 4 \Rightarrow V \Rightarrow$ troca

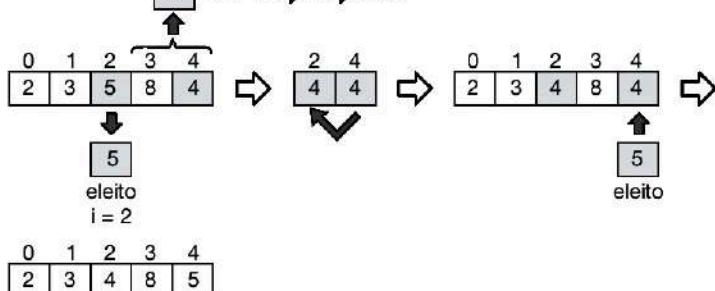


3^a execução do laço

pos = 4

menor

4 $4 < 5 \Rightarrow V \Rightarrow$ troca

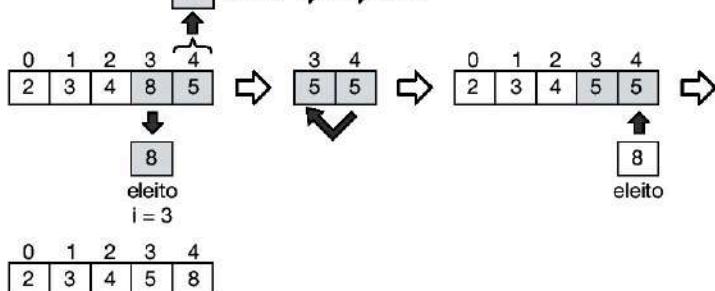


4^a execução do laço

pos = 4

menor

5 $5 < 8 \Rightarrow V \Rightarrow$ troca

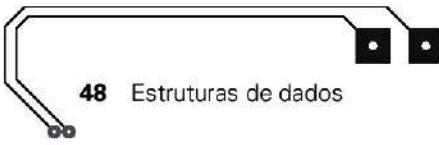




```

algoritmo
declare X[5], i, j, eleito, menor, pos numérico
// carregando os números no vetor
para i ← 0 até 4 faça
    início
        escreva "Digite o ",i+1,"º número: "
        leia X[i]
    fim
// ordenando de forma crescente
// laço que percorre da 1ª posição à
// penúltima posição do vetor
// elegendo um número para ser comparado
para i ← 0 até 3 faça
    início
        eleito ← X[i]
        // encontrando o menor número à direita do eleito
        // com sua respectiva posição
        // posição do eleito = i
        // primeiro número à direita do eleito
        // na posição = i + 1
        menor ← X[i+1]
        pos ← i+1
        // laço que percorre os elementos que estão à direita do
        // número eleito, retornando o menor número à direita e
        // sua posição
        para j ← i+2 até 4 faça
        início
            se (X[j] < menor)
            então início
                menor ← X[j]
                pos ← j
            fim
        fim
        // troca do número eleito com o número da posição pos
        // o número da posição pos é o menor número à direita
        // do número eleito
        se (menor < eleito)
        então início
            X[i] ← X[pos]
            X[pos] ← eleito
        fim
    fim

```



48 Estruturas de dados

```
// mostrando o vetor ordenado
para i ← 0 até 4 faça
    início
        escreva i+1,"º número: ",X[i]
        fim
    fim_algoritmo.
```



```
import java.util.*;
public class selection_sort
{
    public static void main (String args[])
    {
        int X[] = new int[5];
        int i, j, eleito, menor, pos;
        Scanner entrada = new Scanner(System.in);
        // carregando os números no vetor
        for(i=0;i<=4;i++)
        {
            System.out.println("Digite o "+(i+1)+"º número: ");
            X[i] = entrada.nextInt();
        }
        // ordenando de forma crescente
        // laço que percorre da 1ª posição à
        // penúltima posição do vetor
        // elegendo um número para ser comparado
        for(i=0;i<=3;i++)
        {
            eleito = X[i];
            // encontrando o menor número à direita do eleito
            // com sua respectiva posição
            // posição do eleito = i
            // primeiro número à direita do eleito
            // na posição = i + 1
            menor = X[i+1];
            pos = i + 1;
            // laço que percorre os elementos
            // que estão à direita do
            // número eleito, retornando o menor número à
            // direita e sua posição
            for (j=i+2;j<=4;j++)
            {
```

```

        if (X[j] < menor)
        {
            menor = X[j];
            pos = j;
        }
    }
    // troca do número eleito com o número da posição pos
    // o número da posição pos é o menor número à direita
    // do número eleito
    if (menor < eleito)
    {
        X[i] = X[pos];
        X[pos] = eleito;
    }
}
// mostrando o vetor ordenado
for(i=0;i<=4;i++)
{
    System.out.println((i+1)+"º número: "+X[i]);
}
}
}

```



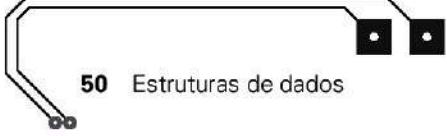
c/c++

```

#include <iostream.h>
#include <conio.h>

void main ()
{
    int X[5];
    int i, j, eleito, menor, pos;
    clrscr();
    // carregando os números no vetor
    for(i=0;i<=4;i++)
    {
        cout<<"Digite o "<<i+1<<"º número: ";
        cin>>X[i];
    }
    // ordenando de forma crescente
    // laço que percorre da 1ª posição
    // à penúltima posição do vetor

```



50 Estruturas de dados

```
// elegendo um número para ser comparado
for(i=0;i<=3;i++)
{
    eleito = X[i];
    // encontrando o menor número à direita do eleito
    // com sua respectiva posição
    // posição do eleito = i
    // primeiro número à direita do
    // eleito na posição = i + 1
    menor = X[i+1];
    pos = i + 1;
    // laço que percorre os elementos
    // que estão à direita do
    // número eleito, retornando o
    // menor número à direita
    // e sua posição
    for (j=i+1;j<=4;j++)
    {
        if (X[j] < menor)
        {
            menor = X[j];
            pos = j;
        }
    }
    // troca do número eleito com o número da posição pos
    // o número da posição pos é o menor número à direita
    // do número eleito
    if (menor < eleito)
    {
        X[i] = X[pos];
        X[pos] = eleito;
    }
}
// mostrando o vetor ordenado
for(i=0;i<=4;i++)
{
    cout<<"\n"<<i+1<<"º número: "<<X[i];
}
getch();
}
```

Análise da complexidade

O trecho de código relevante do algoritmo *SELECTION SORT* é mostrado a seguir.



ALGORITMO

```

1. para i ← 0 até 3 faça
2.   início
3.     eleito ← X[i]
4.     menor ← X[i+1]
5.     pos ← i + 1
6.     para j ← i+1 até 4 faça
7.       início
8.         se (X[j] < menor)
9.           então início
10.             menor ← X[j]
11.             pos ← j
12.         fim
13.     fim
14. se (menor < eleito)
15. então início
16.   X[i] ← X[pos]
17.   X[pos] ← eleito
18.   fim
19. fim

```

Neste algoritmo o vetor utilizado possui n posições, onde n , no exemplo acima, é 5. Cada elemento i da primeira até a penúltima posição é trocado de lugar com o menor elemento que se encontra entre as posições $(i + 1)$ e $(n - 1)$. Para encontrar o primeiro menor elemento e trocá-lo com o da posição 0, realizam-se 5 comparações (considerando a da linha 14). Já para encontrar o segundo menor elemento e trocá-lo com o da posição 1, realizam-se 4 comparações. Para encontrar o terceiro menor elemento e trocá-lo com o da posição 2, realizam-se 3 comparações. E para encontrar o penúltimo menor elemento e colocá-lo na posição 3, realizam-se duas comparações.

Logo, para $i = 0$ na estrutura para da linha 1, a estrutura para da linha 6 (por consequência a comparação da linha 8 também) será executada $n - 1$ vezes (4); para $i = 1$ na estrutura para da linha 1, a estrutura para da linha 6 será executada $n - 2$ vezes (2); para $i = 2$ na estrutura para da linha 1, a estrutura para da linha 6 será executada $n - 3$ vezes (1), e assim por diante até que no último valor de i a estrutura para da linha 8 executará uma única vez. Representa-se o tempo de execução como um somatório de termos:

$$T(n) = 1 + 2 + 3 + \dots + n - 1.$$

52 Estruturas de dados

A fórmula mostrada contém uma progressão aritmética de razão 1 e pode ser calculada aplicando a fórmula do somatório de progressão aritmética, obtendo-se:

$$T(n) = \frac{(a_1 + a_n).n}{2}$$

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

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

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

$$T(n) = \frac{n^2}{2} - \frac{n}{2}.$$

Logo, o tempo de execução do *SELECTION SORT* é $\Theta(n^2)$. Independentemente do vetor de entrada, o algoritmo se comportará da mesma maneira. Lembrando que a notação $\Theta(n^2)$ indica que o tempo de execução do algoritmo é limitado superiormente e inferiormente pela função n^2 , ou seja:

$$c_1 \cdot n^2 \leq \frac{n^2}{2} - \frac{n}{2} \leq c_2 \cdot n^2, c_1 = \frac{1}{10}, c_2 = \frac{1}{2}, n \geq 2.$$

Veja os gráficos de desempenho na Figura 2.5.

Figura 2.5 Gráficos de desempenho

Quantidade de comparações

49995000 49995000 49995000

vetor com
10.000
números
ordenados

vetor com
10.000
números
invertidos

vetor com
10.000
números
aleatórios

Quantidade de trocas

9999 14999 19988

vetor com
10.000
números
ordenados

vetor com
10.000
números
invertidos

vetor com
10.000
números
aleatórios

Tempo de execução em segundos

0,21978 0,21978 0,21978

vetor com
10.000
números
ordenados

vetor com
10.000
números
invertidos

vetor com
10.000
números
aleatórios

Algoritmo de ordenação por intercalação (*MERGE SORT*)

Neste algoritmo de ordenação, o vetor é dividido em vetores com a metade do tamanho do original por meio de um procedimento recursivo. Essa divisão ocorre até que o vetor fique com apenas um elemento e estes sejam ordenados e intercalados.

Neste algoritmo, será aplicada a técnica da divisão e conquista, uma técnica recursiva que envolve três passos em cada nível da recursão:

- Dividir o problema em um certo número de subproblemas.
- Conquistar os subproblemas solucionando-os recursivamente. Se os tamanhos dos subproblemas são suficientemente pequenos, então, solucionar os subproblemas de forma simples.
- Combinar as soluções dos subproblemas na solução de problema original.

Assim, no algoritmo de ordenação por intercalação, *MERGE SORT*, tem-se a técnica da divisão e conquista da seguinte forma:

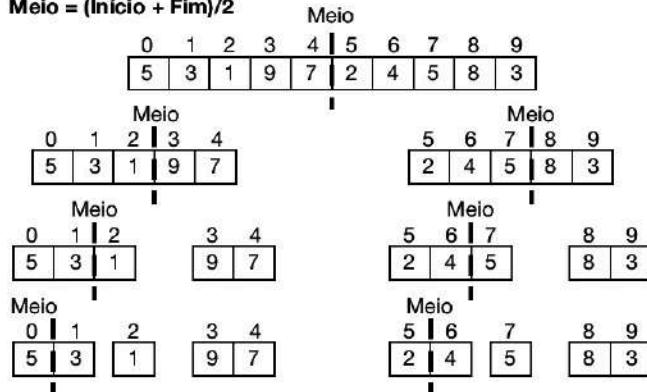
- Dividir: dividir a sequência de n elementos a serem ordenados em duas subsequências de $n/2$ elementos cada.
- Conquistar: ordenar as duas subsequências recursivamente utilizando a ordenação por intercalação;
- Combinar: intercalar as duas subsequências ordenadas para produzir a solução.

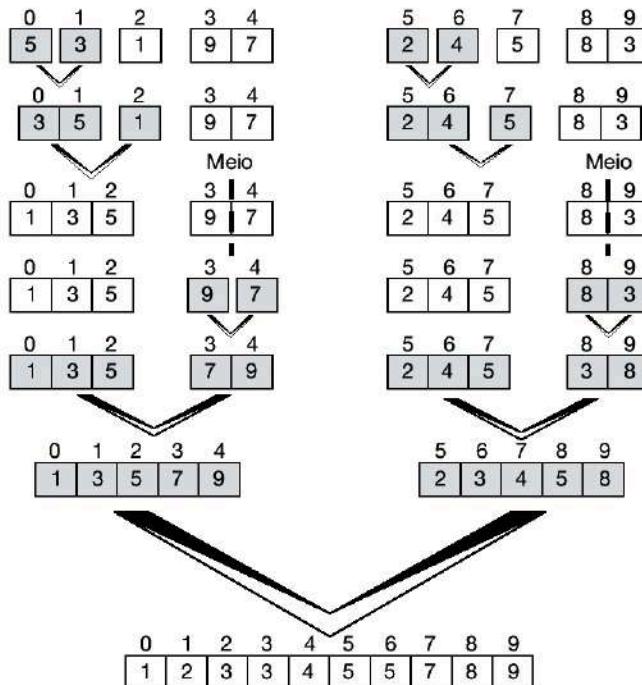
Abaixo, o algoritmo de ordenação por intercalação, *MERGE SORT*, será ilustrado, implementado e analisado com o objetivo ordenar os dados de forma crescente.



ILUSTRAÇÃO

Meio = (Início + Fim)/2





```

algoritmo
declare X[10], i numérico
// carregando os números no vetor
para i ← 0 até 9 faça
    início
        escreva "Digite o ",i+1,"º número: "
        leia X[i]
    fim
// ordenando de forma crescente
merge(X,0,9);
// mostrando o vetor ordenado
para i ← 0 até 9 faça
    início
        escreva i+1,"º número: ",X[i]
    fim
fim_algoritmo.

Função intercala (X, início, fim, meio)
    início
        declare poslivre, inicio_vetor1 numérico
        inicio_vetor2, i, aux[10] numérico

```

```

início_vetor1 ← início
início_vetor2 ← meio+1
poslivre ← início
enquanto (início_vetor1 ≤ meio e início_vetor2 ≤ fim)
    início
        se (X[início_vetor1] ≤ X[início_vetor2])
        então início
            aux[poslivre] ← X[início_vetor1]
            início_vetor1 ← início_vetor1+1
            fim
        senão início
            aux[poslivre] ← X[início_vetor2]
            início_vetor2 ← início_vetor2+1
            fim
        poslivre ← poslivre+1
        fim
    // se ainda existem números no primeiro vetor
    // que não foram intercalados
    para i ← início_vetor1 até meio faça
        início
            aux[poslivre] ← X[i]
            poslivre ← poslivre+1
        fim
    // se ainda existem números no segundo vetor
    // que não foram intercalados
    para i ← início_vetor2 até fim faça
        início
            aux[poslivre] ← X[i]
            poslivre ← poslivre+1
        fim
    // retorna os valores do vetor aux para o vetor X
    para i ← início até fim faça
        início
            X[i] ← aux[i]
        fim
    fim_função_intercala.

Função merge (X, início, fim)
    início
        declare meio numérico
        se (início < fim)
        então início
            meio ← parteinteira((início + fim)/2)
            merge(X,início,meio)
            merge(X,meio+1,fim)
            intercala(X,início, fim, meio)
            fim
    fim_função_merge.

```



```
import java.util.*;
public class merge_sort
{
    public static void main (String args[])
    {
        int X[] = new int[10];
        int i;
        Scanner entrada = new Scanner(System.in);
        // carregando os números no vetor
        for(i=0;i<=9;i++)
        {
            System.out.println("Digite o "+(i+1)+"º número: ");
            X[i] = entrada.nextInt();
        }
        // ordenando de forma crescente
        merge(X,0,9);
        // mostrando o vetor ordenado
        for(i=0;i<=9;i++)
        {
            System.out.println((i+1)+"º número: "+X[i]);
        }
    }

    public static void merge (int X[], int inicio, int fim)
    {
        int meio;
        if (inicio < fim)
        {
            meio = (inicio + fim)/2;
            merge(X,inicio,meio);
            merge(X,meio+1,fim);
            intercala(X,inicio, fim, meio);
        }
    }

    public static void intercala (int X[],int inicio, int fim,
        int meio)
    {
        int poslivre, inicio_vetor1, inicio_vetor2, i;
        int aux[] = new int[10];
        inicio_vetor1 = inicio;
        inicio_vetor2 = meio + 1;
        poslivre = inicio;
        while (inicio_vetor1 <= meio && inicio_vetor2 <= fim)
        {
            if (X[inicio_vetor1] <= X[inicio_vetor2])
```

```

    {
        aux[poslivre] = X[início_vetor1];
        início_vetor1 = início_vetor1 + 1;
    }
    else
    {
        aux[poslivre] = X[início_vetor2];
        início_vetor2 = início_vetor2 + 1;
    }
    poslivre = poslivre + 1;
}
// se ainda existem números no primeiro vetor
// que não foram intercalados
for (i=início_vetor1;i<=meio;i++)
{
    aux[poslivre] = X[i];
    poslivre = poslivre + 1;
}
// se ainda existem números no segundo vetor
// que não foram intercalados
for (i=início_vetor2;i<=fim;i++)
{
    aux[poslivre] = X[i];
    poslivre = poslivre + 1;
}
// retorna os valores do vetor aux para o vetor X
for (i=início;i<=fim;i++)
{
    X[i] = aux[i];
}
}
}

```



c/c++

```

#include <iostream.h>
#include <conio.h>

void intercala (int X[],int inicio, int fim, int meio)
{
    int poslivre, início_vetor1, início_vetor2, i;
    int aux[10];
    início_vetor1 = inicio;
    início_vetor2 = meio + 1;
    poslivre = inicio;

```

58 Estruturas de dados

```
while (inicio_vetor1 <= meio && inicio_vetor2 <= fim)
{
    if (X[inicio_vetor1] <= X[inicio_vetor2])
    {
        aux[poslivre] = X[inicio_vetor1];
        inicio_vetor1 = inicio_vetor1 + 1;
    }
    else
    {
        aux[poslivre] = X[inicio_vetor2];
        inicio_vetor2 = inicio_vetor2 + 1;
    }
    poslivre = poslivre + 1;
}
// se ainda existem números no primeiro vetor
// que não foram intercalados
for (i=inicio_vetor1;i<=meio;i++)
{
    aux[poslivre] = X[i];
    poslivre = poslivre + 1;
}
// se ainda existem números no segundo vetor
// que não foram intercalados
for (i=inicio_vetor2;i<=fim;i++)
{
    aux[poslivre] = X[i];
    poslivre = poslivre + 1;
}
// retorna os valores do vetor aux para o vetor X
for (i=inicio;i<=fim;i++)
{
    X[i] = aux[i];
}
}

void merge (int X[], int inicio, int fim)
{
    int meio;
    if (inicio < fim)
    {
        meio = (inicio + fim)/2;
        merge(X,inicio,meio);
        merge(X,meio+1,fim);
        intercala(X,inicio, fim, meio);
    }
}
```

```

void main()
{
    int X[10];
    int i;
    clrscr();
    // carregando os números no vetor
    for(i=0;i<=9;i++)
    {
        cout<<"Digite o "<<i+1<<"º número: ";
        cin>>X[i];
    }
    // ordenando de forma crescente
    merge(X, 0, 9);
    // mostrando o vetor ordenado
    for(i=0;i<=9;i++)
    {
        cout<<"\n"<<i+1<<"º número: "<<X[i];
    }
    getch();
}

```

● Análise da complexidade

O trecho de código relevante do algoritmo *MERGE SORT* é mostrado a seguir.



1. Função `merge (X, início, fim)`
2. `início`
3. declare meio numérico
4. se (`início < fim`)
5. então `início`
6. `meio ← parteinteira((início + fim)/2)`
7. `merge(X,início,meio)`
8. `merge(X,meio+1,fim)`
9. `intercala(X,início, fim, meio)`
10. `fim`
11. `fim_função_merge.`

Para calcular o tempo de execução do *MERGE SORT*, é preciso calcular inicialmente o tempo de execução da função `intercala`. O algoritmo da função `intercala`, que realiza a intercalação de dois vetores cujos tamanhos suponha que sejam m_1 e m_2 , respectivamente, faz a varredura de todas as posições dos dois vetores, gastando com isso tempo $n = m_1 + m_2$.

Analisando o trecho de código anterior, verifica-se que o algoritmo recursivo `merge` possui três chamadas de função, sendo que as duas primeiras são chamadas recursivas e recebem metade ($\frac{n}{2}$) e ($\frac{n}{2}$) dos elementos do vetor passado, e a outra é a chamada para a função que realiza a intercalação das duas metades.

A linha da comparação (`if`) e a linha da atribuição gastam tempo constante, $O(1)$. Para calcular o tempo de execução de um algoritmo recursivo inicialmente deve-se obter a expressão de recorrência.

Então, desconsiderando a princípio as funções piso e teto, a expressão de recorrência do algoritmo *MERGE SORT* é dada por $T(n) = 2T\left(\frac{n}{2}\right) + n$, sendo que $2T\left(\frac{n}{2}\right)$ correspondem às duas chamadas recursivas e n ao tempo gasto com a intercalação das duas metades do vetor. O tempo gasto nas demais linhas do algoritmo ($O(1)$ tempo constante) é menor que o gasto pelo `intercala`, por isso soma-se apenas n à expressão de recorrência.

A recorrência obtida acima será solucionada pelo método master, como mostrado a seguir. Os valores necessários para resolução por esse método são: $a = 2$, $b = 2$, $f(n) = n$. Como

$$f(n) = \Theta(n^{\log_2 2})$$

$$n = \Theta(n^{\log_2 2})$$

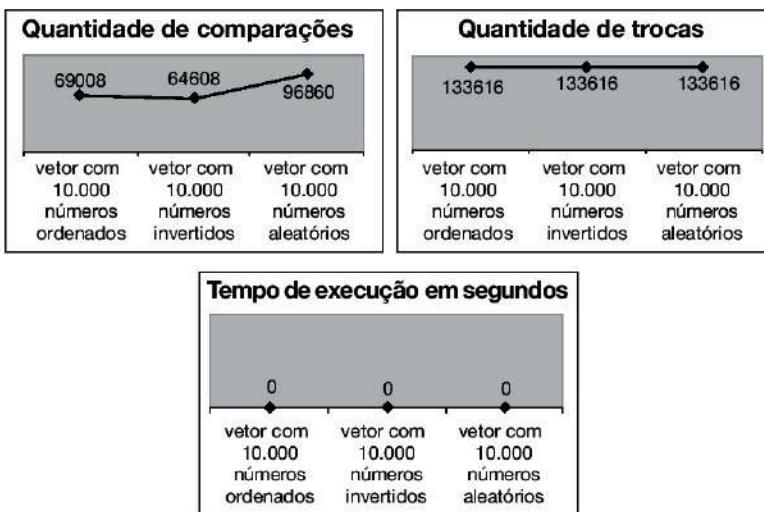
$$n = \Theta(n^1),$$

então

$$T(n) = \Theta(n \log n).$$

Nesse algoritmo, qualquer que seja o vetor de entrada, o algoritmo trabalhará da mesma maneira, dividindo o vetor ao meio, ordenando cada metade recursivamente e intercalando as duas metades ordenadas. Veja os gráficos de desempenho na Figura 2.6.

Figura 2.6 Gráficos de desempenho



Algoritmos de ordenação rápida (*QUICK SORT*)

Neste algoritmo de ordenação o vetor é particionado em dois por meio de um procedimento recursivo. Essa divisão ocorre até que o vetor fique com apenas um elemento, enquanto os demais ficam ordenados à medida que ocorre o particionamento.

Esse algoritmo também é baseado na técnica da divisão e conquista mencionada na seção do algoritmo *MERGE SORT*.

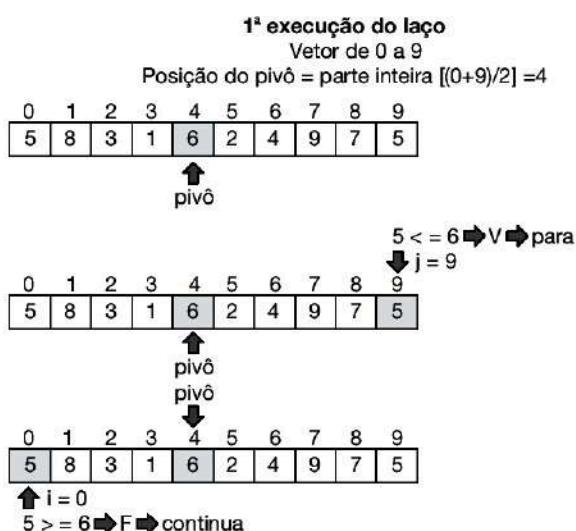
Assim, no algoritmo de ordenação rápida, *QUICK SORT*, tem-se a técnica da divisão e conquista da seguinte forma:

- Dividir: o vetor $X[p..r]$ é particionado (rearranjado) em dois subvetores não vazios $X[p..q]$ e $X[q+1..r]$, tais que cada elemento de $X[p..q]$ é menor ou igual a cada elemento de $X[q+1..r]$. O índice q é calculado como parte do processo de particionamento. Para determinar o índice q , escolhe-se o elemento que se encontra na metade do vetor original, chamado de *pivô*, e rearranjam-se os elementos do vetor de forma que os que ficarem à esquerda de q são menores (ou iguais) ao pivô e os que ficarem à direita de q são maiores (ou iguais) ao pivô.
- Conquistar: os dois subvetores são ordenados $X[p..q]$ e $X[q+1..r]$ por chamadas recursivas ao *QUICK SORT*.
- Combinar: durante o processo recursivo, os elementos vão sendo ordenados no próprio vetor, não exigindo nenhum processamento nesta etapa.

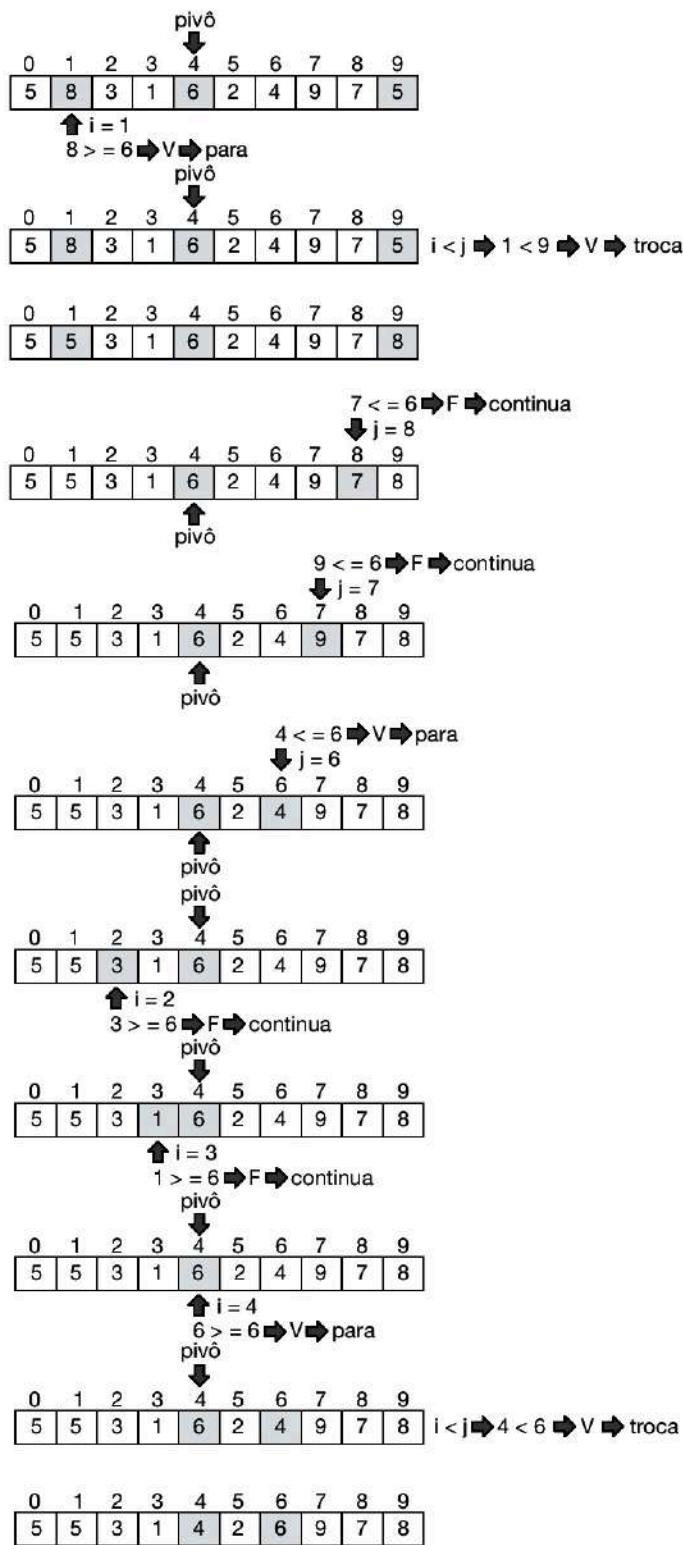
Abaixo, o algoritmo de ordenação rápida, *QUICK SORT*, será ilustrado, implementado e analisado com o objetivo ordenar os dados de forma crescente.



ILUSTRAÇÃO

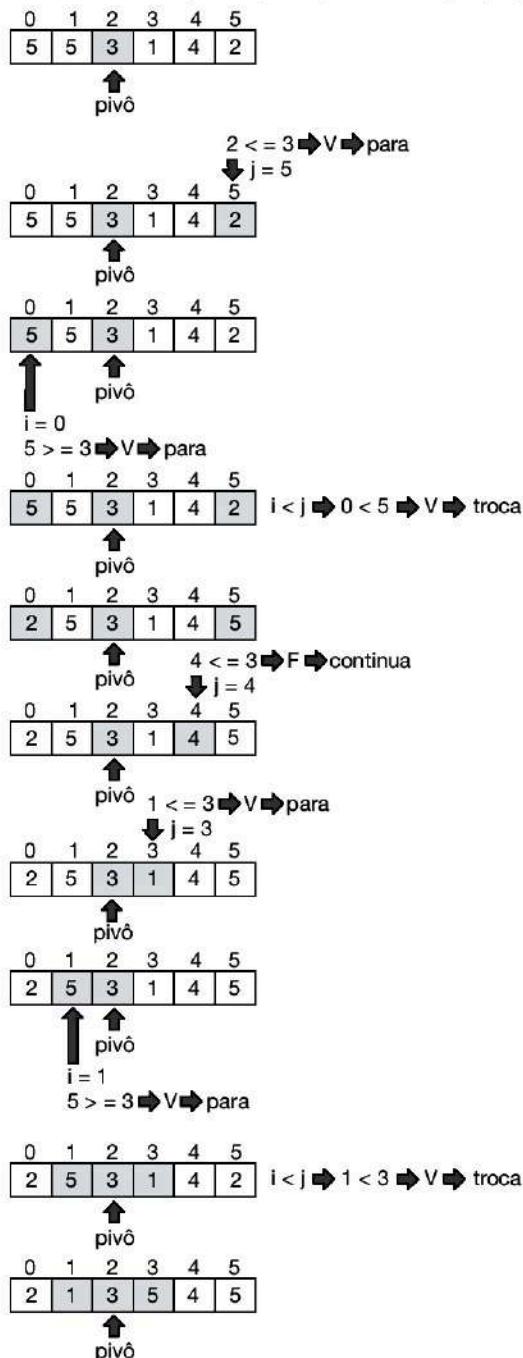


62 Estruturas de dados



2ª execução do laço

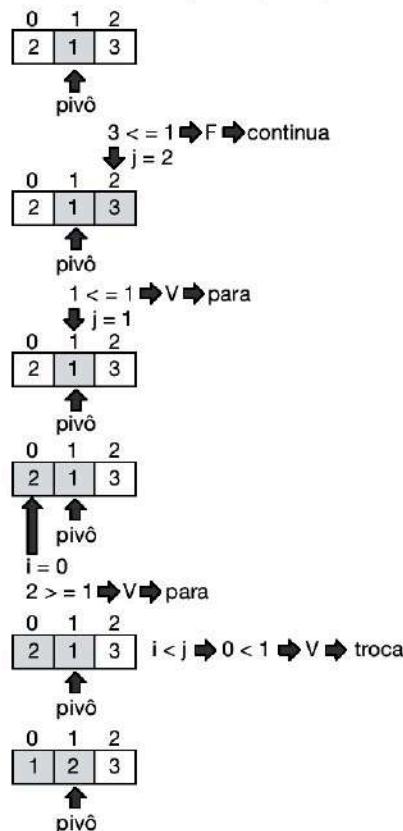
Vetor de 0 a 5

Posição do pivô = parte inteira $[(0+5)/2] = 2$ 

3^a execução do laço

Vetor de 0 a 2

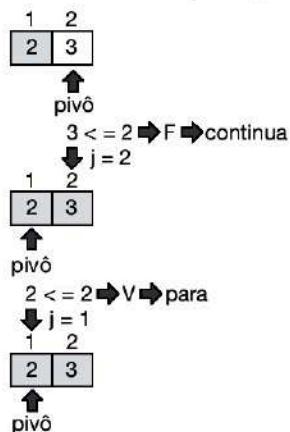
Posição do pivô = parte inteira $[(0+2)/2] = 1$

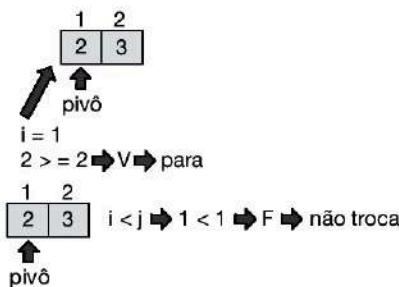
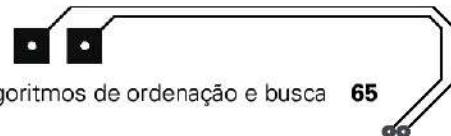


4^a execução do laço

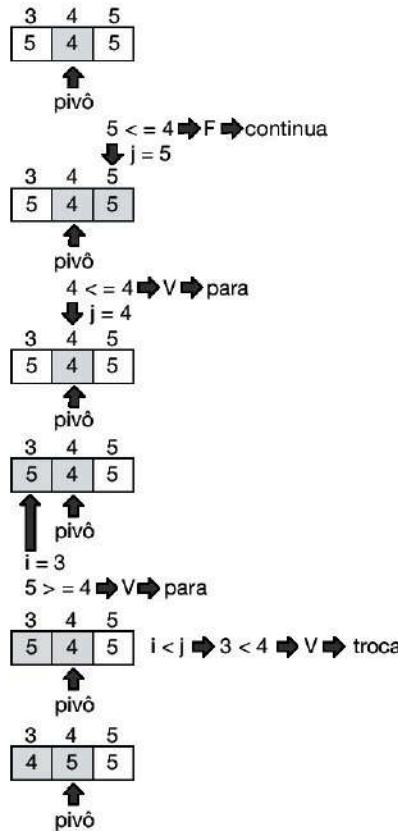
Vetor de 1 a 2

Posição do pivô = parte inteira $[(1+2)/2] = 1$

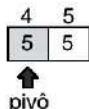


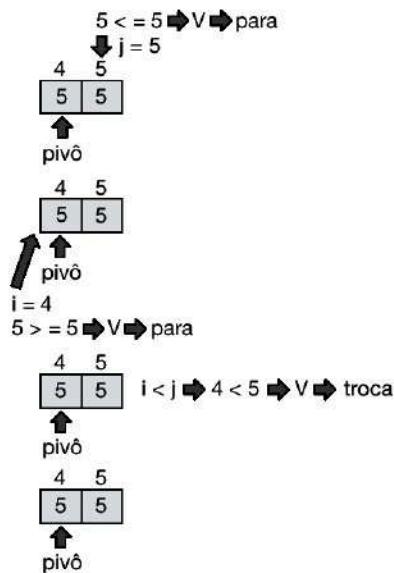

5^a execução do laço

Vetor de 3 a 5

 Posição do pivô = parte inteira $[(3+5)/2] = 4$

6^a execução do laço

Vetor de 4 a 5

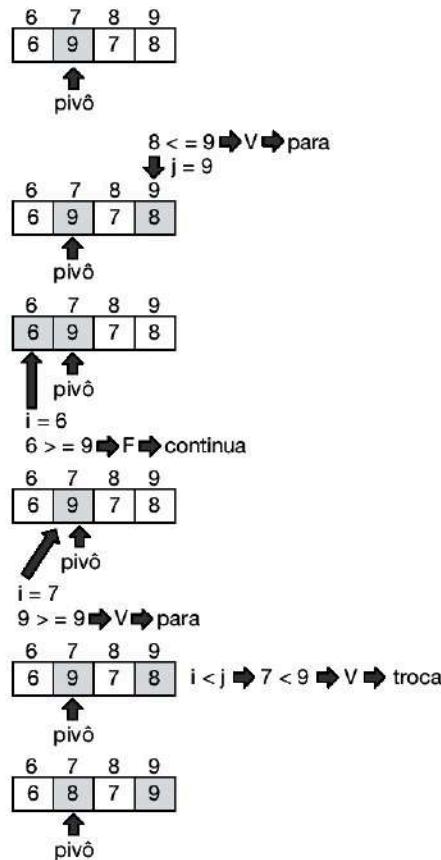
 Posição do pivô = parte inteira $[(4+5)/2] = 4$




7^a execução do laço

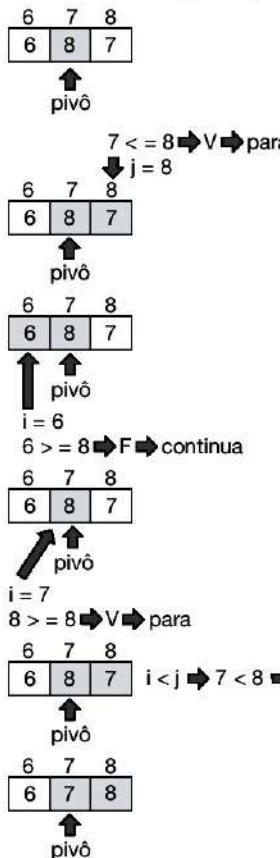
Vetor de 6 a 9

Posição do pivô = parte inteira $[(6+9)/2] = 7$



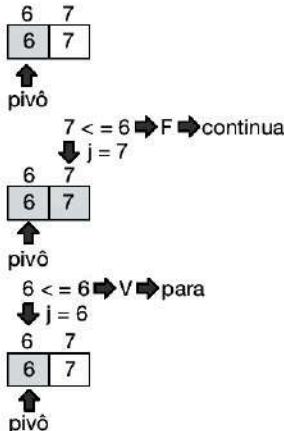
8ª execução do laço

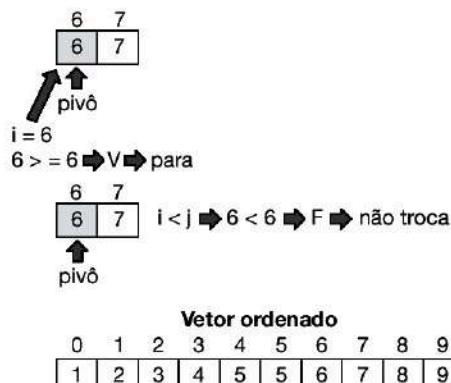
Vetor de 6 a 8

Posição do pivô = parte inteira $[(6+8)/2] = 7$ 

9ª execução do laço

Vetor de 6 a 7

Posição do pivô = parte inteira $[(6+7)/2] = 7$ 



ALGORITMO

```

algoritmo
declare X[10], i numérico
// carregando os números no vetor
para i ← 1 até 10 faça
    início
        escreva "Digite o ",i,"º número: "
        leia X[i]
    fim
// ordenando de forma crescente
quicksort(X,0,9)
// mostrando o vetor ordenado
para i ← 1 até 10 faça
    início
        escreva i,"º número: ",X[i]
    fim
fim_algoritmo.

```

Função troca(X, i, j)

```

início
    declare aux numérico
    aux ← X[i]
    X[i] ← X[j]
    X[j] ← aux
fim_função_troca.

```

Função particao(X, p, r)

```

início
    declare pivo, i, j numéricico

```

```

pivo ← X[(p+r)/2]
i ← p-1
j ← r+1
enquanto (i < j) faça
    início
        repita
            j ← j - 1
            até (X[j] <= pivo)
        repita
            i ← i + 1
            até (X[i] >= pivo)
            se (i < j)
                então troca(X,i,j)
    fim
    retorno j
fim_função_particao.

```

```

Função quicksort(X, p, r)
início
    declare q numérico
    se (p < r)
        então início
            q ← particao(X,p,r)
            quicksort(X,p,q)
            quicksort(X,q+1,r)
        fim
    fim_função_quicksort.

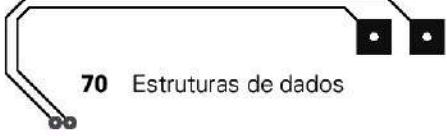
```



```

import java.util.*;
public class quick_sort
{
    public static void main(String args[])
    {
        int X[] = new int[10];
        int i;
        Scanner entrada = new Scanner(System.in);
        // carregando os números no vetor
        for(i=0;i<=9;i++)
        {
            System.out.println("Digite o "+(i+1)+"º número: ");
            X[i] = entrada.nextInt();
        }
    }
}

```



70 Estruturas de dados

```
// ordenando de forma crescente
quicksort(X,0,9);
// mostrando o vetor ordenado
System.out.println("Vetor Ordenado");
for (i=0;i<=9;i++)
{
    System.out.println(" " + X[i]);
}
}

public static void troca(int X[], int i, int j)
{
    int aux;
    aux = X[i];
    X[i] = X[j];
    X[j] = aux;
}

public static int particao(int X[],int p,int r)
{
    int pivo, i, j;
    pivo = X[(p+r)/2];
    i = p-1;
    j = r+1;
    while (i < j)
    {
        do
        {
            j = j - 1;
        }
        while (X[j] > pivo);
        do
        {
            i = i + 1;
        }
        while (X[i] < pivo);
        if (i < j) troca(X,i,j);
    }
    return j;
}

public static void quicksort(int X[], int p, int r)
{
    int q;
    if (p < r)
```



```
{  
    q = particao(X,p,r);  
    quicksort(X,p,q);  
    quicksort(X,q+1,r);  
}  
}  
}
```



```
#include <iostream.h>  
#include <conio.h>  
  
void troca(int X[], int i, int j)  
{  
    int aux;  
    aux = X[i];  
    X[i] = X[j];  
    X[j] = aux;  
}  
  
int particao(int X[], int p, int r)  
{  
    int pivo, i, j;  
    pivo = X[(p+r)/2];  
    i = p-1;  
    j = r+1;  
    while (i < j)  
    {  
        do  
        {  
            j = j - 1;  
        }  
        while (X[j] > pivo);  
        do  
        {  
            i = i + 1;  
        }  
        while (X[i] < pivo);  
        if (i < j) troca(X,i,j);  
    }  
    return j;  
}  
void quicksort(int X[], int p, int r)
```

```

    {
        int q;
        if (p < r)
        {
            q = particao(X,p,r);
            quicksort(X,p,q);
            quicksort(X,q+1,r);
        }
    }

void main()
{
    int X[10];
    int i;
    clrscr();
    // carregando os números no vetor
    for(i=0;i<=9;i++)
    {
        cout<<"Digite o "<<i+1<<"º número: ";
        cin>>X[i];
    }
    // ordenando de forma crescente
    quicksort(X,0,9);
    // mostrando o vetor ordenado
    cout<<"Vetor Ordenado";
    for (i=0;i<=9;i++)
    {
        cout<<" "<<X[i];
    }
    getch();
}

```

Análise da complexidade



ALGORITMO

1. Função particao(X, p, r)
2. início
3. declare pivo, i, j numérico
4. pivo \leftarrow X[(p+r)/2]
5. i \leftarrow p-1
6. j \leftarrow r+1
7. enquanto (i < j) faça

```

8.    inicio
9.        repita
10.           j ← j - 1
11.           até (X[j] <= pivo)
12.           repita
13.               i ← i + 1
14.               até (X[i] >= pivo)
15.               se (i < j)
16.                   então troca(X,i,j)
17.           fim
18.       retorno j
19.   fim_função_particao.
20. Função quicksort(X, p, r)
21. inicio
22. declare q numérico
23. se (p < r)
24. então inicio
25.     q ← particao(X,p,r)
26.     quicksort(X,p,q)
27.     quicksort(X,q+1,r)
28.   fim
29. fim_função_quicksort.

```

A ideia principal do algoritmo de ordenação *QUICK SORT* é fazer o processo de partitionar o vetor em duas partes, realizado pela função *particao*. Nesse procedimento, o vetor é partitionado na posição *j*, de forma que todos os elementos do lado esquerdo de *j* são menores ou iguais ao elemento denominado pivô, e todos os do lado direito são maiores que o pivô. Nesse procedimento, o tempo de execução é limitado pelo tamanho do vetor, no caso *n*. Isso acontece porque para realizar esse partitionamento, ele compara os elementos de posição *i* (cujo valor inicia na primeira posição e vai aumentando) e os da posição *j* (cujo valor inicia com a última posição e decresce) com o valor pivô. Ou seja, ele compara todos os elementos do vetor com o pivô enquanto os índices atenderem a condição *i < j*. Logo, o procedimento *particao* realizará $O(n)$ comparações.

O tempo de execução do *QUICK SORT* depende se o partitionamento é ou não balanceado, e isto depende de quais elementos são utilizados para o partitionamento. Se é balanceado, o algoritmo executa tão rapidamente quanto o *MERGE SORT*. Se não é balanceado, o algoritmo é tão lento quanto o *INSERTION SORT*.

O pior caso ocorre quando o procedimento de partitionamento produz uma região com $n-1$ elementos e outra com somente um elemento. Considere que esse partitionamento ocorra em todo passo do algoritmo. Como o custo do partitionamento é $\Theta(n)$, a recorrência para o tempo de execução do *QUICK SORT* é

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

A recorrência obtida no pior caso pode ser resolvida utilizando o método da expansão telescópica, conforme mostrado a seguir.

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

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

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

...

$$T(n) = T(n - i) + in$$

Deve-se agora calcular o valor de i para obter-se o tempo final. Considerando que a recursão termina quando se atinge um vetor de tamanho 1 ($T(1)$) e o custo para se ordenar tal vetor é $\Theta(1)$, ocorre no algoritmo do *QUICK SORT* quando $T(n - i) = T(1)$, ou seja,

$$n - i = 1$$

$$i = n - 1.$$

Agora, substituindo o valor de i na expressão anterior, obtém-se que

$$T(n) = T(n - i) + in$$

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

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

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

Portanto, o tempo de execução do *QUICK SORT* no pior caso é $\Theta(n^2)$ para $c = 2$, $n \geq 1$.

O melhor caso ocorre quando o procedimento de particionamento produz duas regiões de tamanho $n/2$. A recorrência então é

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

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

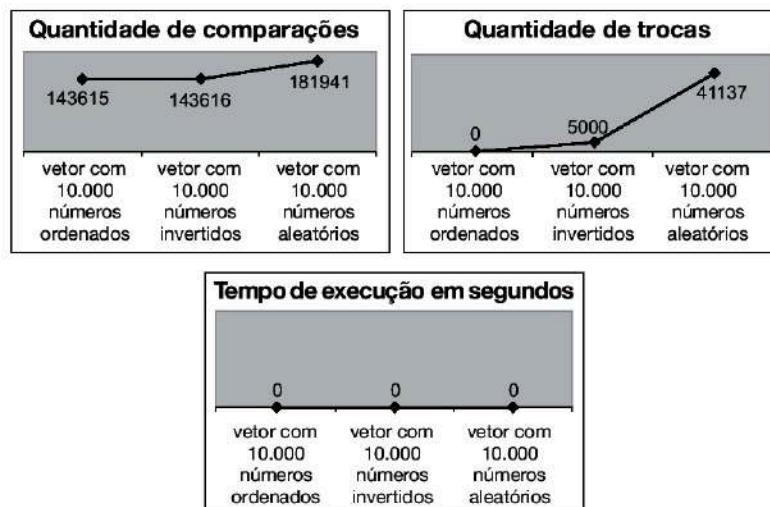
Essa é a mesma recorrência já foi calculada na sessão do algoritmo *MERGE SORT*. Então, pelo teorema master, sabe-se que essa recorrência solucionada fornece o tempo de execução $T(n) = \Theta(n \log n)$.

Veja os gráficos de desempenho na Figura 2.7.

Algoritmo de ordenação (*HEAP SORT*)

Este algoritmo de ordenação é baseado na estrutura de dados *HEAP*, que nada mais é do que um vetor (X) que pode ser visto como uma árvore binária completa, onde cada nó possui no máximo 2 filhos. Cada vértice da árvore corresponde a um elemento do vetor. A árvore é completamente preenchida exceto no último nível. Cada nível é sempre preenchido da direita para a esquerda. Além disso, num *HEAP*, para todo vértice i diferente da raiz, a seguinte propriedade deve ser válida: $X[\text{Pai}(i)] \geq X[i]$.

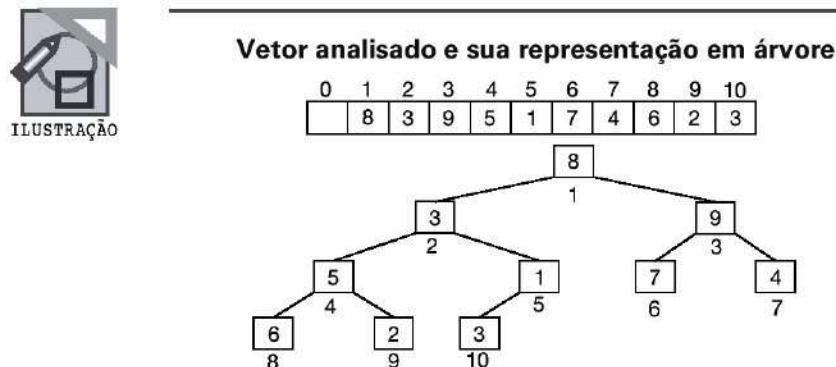
Dado um índice i do vetor, para se descobrir as posições em que se encontram o seu pai, o filho esquerdo e o direito, realizam-se os cálculos: $\text{Pai}(i) = i/2$, $\text{Filho Esquerdo}(i) = 2*i$ e $\text{Filho Direito}(i) = 2*i + 1$.

Figura 2.7 Gráficos de desempenho

Para ordenar o vetor de entrada X , inicialmente transforma-se o vetor num *HEAP*, utilizando o procedimento `heap_fica`. O procedimento `heap_fica` analisa se um determinado elemento da posição j atende a propriedade. Caso não atenda, o maior de seus filhos é trocado com o pai j que não atende a propriedade *HEAP*, e o processo continua até levar o elemento j a uma folha ou até que a propriedade seja satisfeita.

Após ter transformado o vetor em um *HEAP*, passa-se para a etapa em que a ordenação é realizada de fato. Com o *HEAP* sendo um vetor, o maior elemento ficará na raiz (ou primeira posição do vetor), então o primeiro elemento pode ser trocado com o último de maneira que ocupará a posição correta dentro do vetor que está sendo ordenado, visto que quem está na raiz é o maior elemento e deve ficar na última posição após a ordenação. Após feita a troca, desconsidera-se esse último elemento e aplica-se o método `heap_fica` novamente sobre a raiz da árvore, que agora deixou de atender à propriedade *HEAP*. Dessa forma, após a aplicação do método, é novamente obtido o maior elemento na raiz da árvore, e pode ser feito o mesmo processo de troca com o penúltimo elemento. O processo continua até a árvore permanecer com um único elemento.

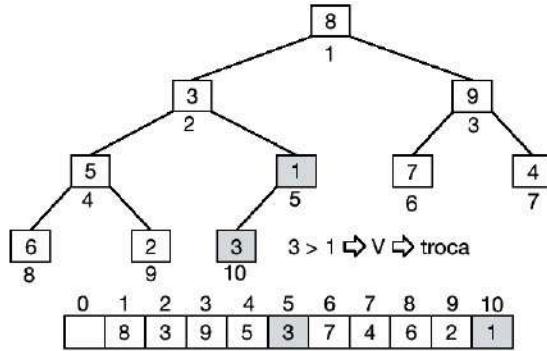
A ilustração a seguir mostra a ordenação de um vetor de forma crescente.



1^a execução do procedimento HEAP_FICA

$i = \text{qtde}/2 = 10/2 = 5$ até 1 passo -1
 $i = 5$

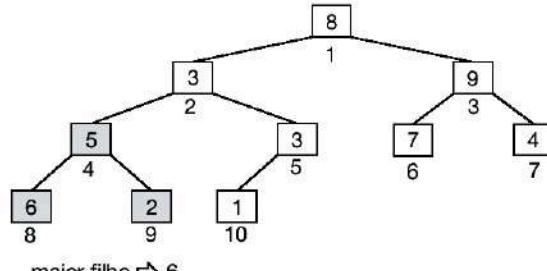
0	1	2	3	4	5	6	7	8	9	10
8	3	9	5	1	7	4	6	2	3	



2^a execução do procedimento HEAP_FICA

$i = 4$

0	1	2	3	4	5	6	7	8	9	10
8	3	9	5	3	7	4	6	2	1	



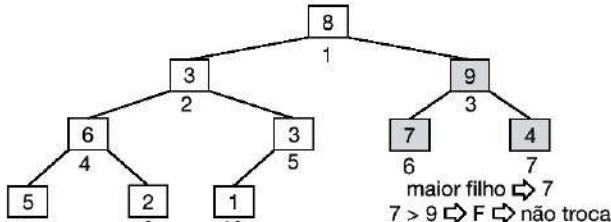
maior filho \Rightarrow 6
6 > 5 \Rightarrow V \Rightarrow troca

0	1	2	3	4	5	6	7	8	9	10
8	3	9	6	3	7	4	5	2	1	

3^a execução do procedimento HEAP_FICA

$i = 3$

0	1	2	3	4	5	6	7	8	9	10
8	3	9	6	3	7	4	5	2	1	



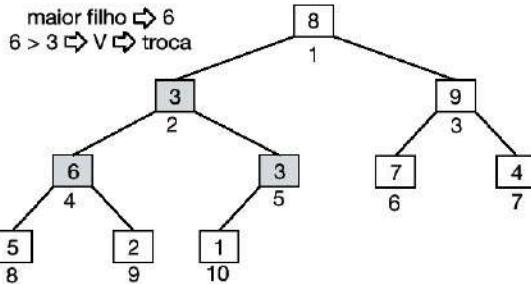
maior filho \Rightarrow 7
7 > 9 \Rightarrow F \Rightarrow não troca

0	1	2	3	4	5	6	7	8	9	10
8	3	9	6	3	7	4	5	2	1	


 4^a execução do procedimento **HEAP_FICA**

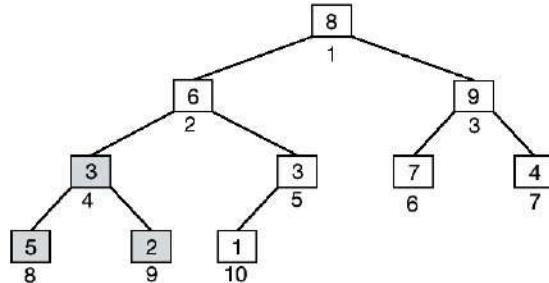
i = 2

0	1	2	3	4	5	6	7	8	9	10
8	3	9	6	3	7	4	5	2	1	



0	1	2	3	4	5	6	7	8	9	10
8	6	9	3	3	7	4	5	2	1	

0	1	2	3	4	5	6	7	8	9	10
8	6	9	3	3	7	4	5	2	1	



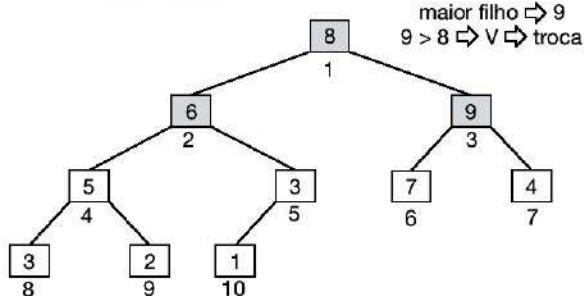
maior filho \Rightarrow 5
 $5 > 3 \Rightarrow V \Rightarrow$ troca

0	1	2	3	4	5	6	7	8	9	10
8	6	9	5	3	7	4	3	2	1	

 5^a execução do procedimento **HEAP_FICA**

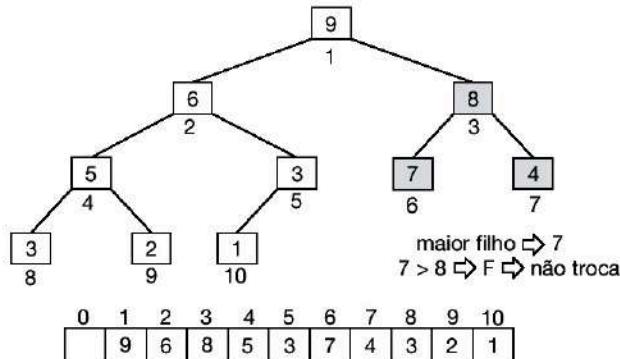
i = 1

0	1	2	3	4	5	6	7	8	9	10
8	6	9	5	3	7	4	3	2	1	



0	1	2	3	4	5	6	7	8	9	10
9	6	8	5	3	7	4	3	2	1	

0	1	2	3	4	5	6	7	8	9	10
9	6	8	5	3	7	4	3	2	1	



0	1	2	3	4	5	6	7	8	9	10
9	6	8	5	3	7	4	3	2	1	

1º execução do procedimento **ORDENA**

i = qtde = 10 até 2 passo - 1

i = 10

0	1	2	3	4	5	6	7	8	9	10
9	6	8	5	3	7	4	3	2	1	

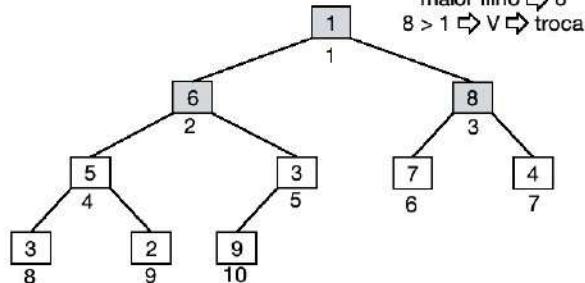
troca

0	1	2	3	4	5	6	7	8	9	10
1	6	8	5	3	7	4	3	2	9	

heap_fica

0	1	2	3	4	5	6	7	8	9	10
1	6	8	5	3	7	4	3	2	9	

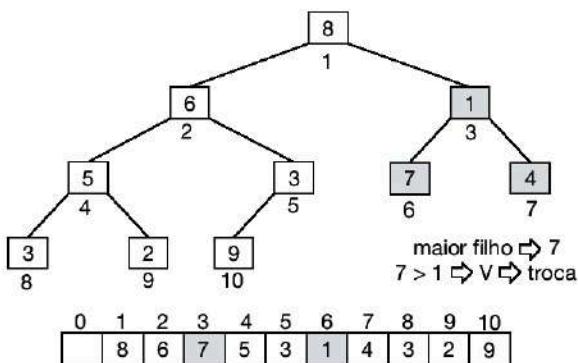
maior filho → 8
8 > 1 → V → troca



0	1	2	3	4	5	6	7	8	9	10
8	6	1	5	3	7	4	3	2	9	

heap_fica

0	1	2	3	4	5	6	7	8	9	10
8	6	1	5	3	7	4	3	2	9	



2ª execução do procedimento **ORDENA**
 $i = 9$

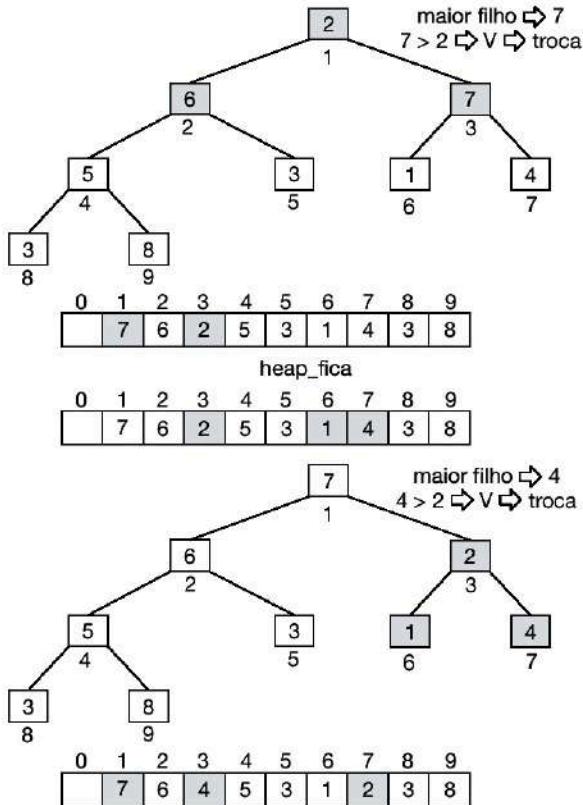
0	1	2	3	4	5	6	7	8	9
8	6	7	5	3	1	4	3	2	9

troca

0	1	2	3	4	5	6	7	8	9
2	6	7	5	3	1	4	3	8	9

heap_fica

0	1	2	3	4	5	6	7	8	9
2	6	7	5	3	1	4	3	8	9



3^a execução do procedimento **ORDENA**

i = 8

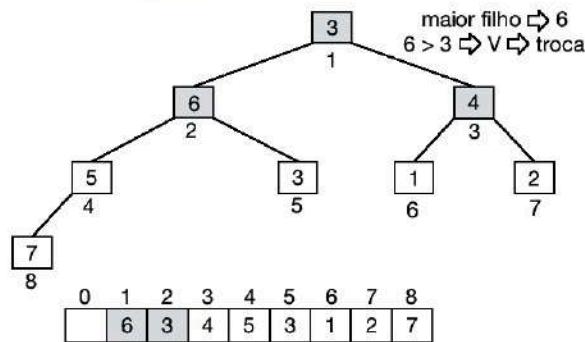
0	1	2	3	4	5	6	7	8
7	6	4	5	3	1	2	3	

troca

0	1	2	3	4	5	6	7	8
3	6	4	5	3	1	2	7	

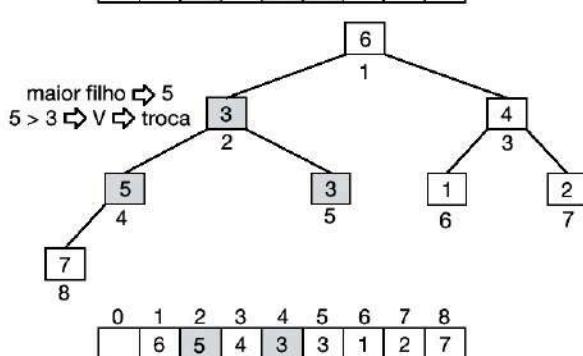
heap_fica

0	1	2	3	4	5	6	7	8
3	6	4	5	3	1	2	7	



0	1	2	3	4	5	6	7	8
6	3	4	5	3	1	2	7	

heap_fica



4^a execução do procedimento **ORDENA**

i = 7

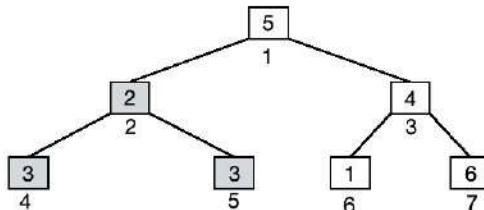
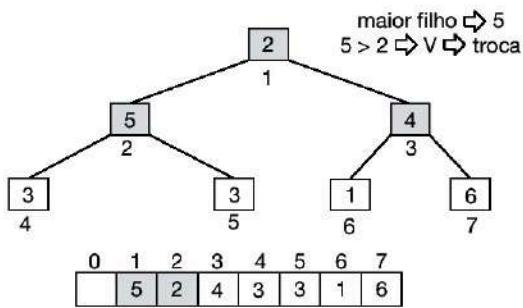
0	1	2	3	4	5	6	7
6	5	4	3	3	1	2	

troca

0	1	2	3	4	5	6	7
2	5	4	3	3	1	6	

heap_fica

0	1	2	3	4	5	6	7
2	5	4	3	3	1	6	



0	1	2	3	4	5	6	7
5	3	4	2	3	1	6	

5º execução do procedimento **ORDENA**
i = 6

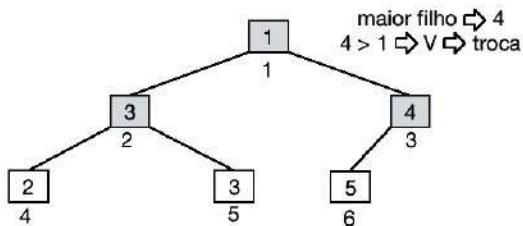
0	1	2	3	4	5	6
5	3	4	2	3	1	

troca

0	1	2	3	4	5	6
1	3	4	2	3	5	

heap_fica

0	1	2	3	4	5	6
1	3	4	2	3	5	



0	1	2	3	4	5	6
4	3	1	2	3	5	

6^a execução do procedimento **ORDENA**

$i = 5$

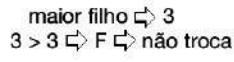
0	1	2	3	4	5
4	3	1	2	3	

troca

0	1	2	3	4	5
3	3	1	2	4	

heap_fica

0	1	2	3	4	5
3	3	1	2	4	



maior filho \Leftrightarrow 3

$3 > 3 \Leftrightarrow F \Leftrightarrow$ não troca

0	1	2	3	4	5
3	3	1	2	4	

7^a execução do procedimento **ORDENA**

$i = 4$

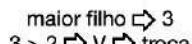
0	1	2	3	4
3	3	1	2	

troca

0	1	2	3	4
---	---	---	---	---

heap_fica

0	1	2	3	4
---	---	---	---	---



maior filho \Leftrightarrow 3

$3 > 2 \Leftrightarrow V \Leftrightarrow$ troca

0	1	2	3	4
---	---	---	---	---

8^a execução do procedimento **ORDENA***i = 3*

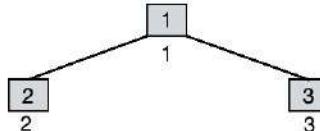
0	1	2	3
	3	2	1

troca

0	1	2	3
	1	2	3

heap_fica

0	1	2	3
	1	2	3



2 > 1 ↗ V ↘ troca

0	1	2	3
	2	1	3

9^a execução do procedimento **ORDENA***i = 2*

0	1	2
	2	1

troca

0	1	2
	1	2

Vetor ordenado

0	1	2	3	3	4	5	6	7	8	9
	1	2	3	3	4	5	6	7	8	9



ALGORITMO
 algoritmo
 declare X[11], i, qtde numérico
 // carregando os números no vetor
 para i ← 1 até 10 faça
 início

 escreva "Digite o ",i,"º número: "
 leia X[i]

fim

// transforma o vetor digitado em um heap
// cada nó pai é maior que seus filhos

```

qtde ← 10
transforma_heap(qtde);
// ordenando de forma crescente
ordena(qtde);
// mostrando o vetor ordenado
para i ← 1 até 10 faça
    início
        escreva i,"º número: ",X[i]
    fim
fim_algoritmo.

Função transforma_heap(qtde numérico)
    início
    declare i, pai, aux numérico
    para i ← qtde/2 até 1 faça passo -1
        início
            heap_fica(i,qtde)
        fim
fim_função_transforma_heap.

Função heap_fica(i, qtde numérico)
    início
        declare f_esq, f_dir, maior, aux numérico
        maior ← i
        se (2*i+1 ≤ qtde)
        então início
            // o nó que está sendo analisado
            // tem filhos para esquerda e direita
            f_esq ← 2*i+1
            f_dir ← 2*i
            se (X[f_esq] ≥ X[f_dir] E X[f_esq] > X[i])
            então maior ← 2*i+1
            senão se (X[f_dir] > X[f_esq] E X[f_dir] >
            X[i])
                então maior ← 2*i
            fim
        senão se (2*i ≤ qtde)
        então início
            // o nó que está sendo analisado
            // tem filho apenas para a direita
            f_dir ← 2*i
            se (X[f_dir] > X[i])
            então maior ← 2*i
            fim
        se (maior ≠ i)

```

```

        então inicio
            aux ← X[i]
            X[i] ← X[maior]
            X[maior] ← aux
            heap_fica(maior,qtde)
        fim
    fim_função_heap_fica.

Função ordena(qtde numérico)
    início
    declare i, aux, ultima_posi numérico
    para i ← qtde até 2 faça passo -1
        início
            aux ← X[1]
            X[1] ← X[i]
            X[i] ← aux
            ultima_posi ← i-1
            heap_fica(1,ultima_posi)
        fim
    fim_função_ordenar.

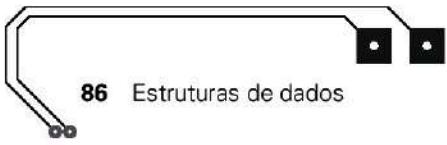
```



```

import java.util.*;
public class heap_sort
{
    static int X[] = new int[11];
    static Scanner entrada = new Scanner(System.in);
    public static void main (String args[])
    {
        int i,qtde;
        // carregando os números no vetor
        for(i=1;i<=10;i++)
        {
            System.out.println("Digite o "+i+"º número: ");
            X[i] = entrada.nextInt();
        }
        // transforma o vetor digitado em um heap
        // cada nó pai é maior que seus filhos
        qtde=10;
        transforma_heap(qtde);
        // ordenando de forma crescente
        ordena(qtde);
        // mostrando o vetor ordenado
    }
}

```



86 Estruturas de dados

```
        for(i=1;i<=10;i++)
    {
        System.out.println(i+"º número: "+X[i]);
    }
}

public static void transforma_heap(int qtde)
{
    int i, pai, aux;
    for (i=qtde/2;i>=1;i--)
    {
        heap_fica(i,qtde);
    }
}

public static void heap_fica(int i, int qtde)
{
    int f_esq, f_dir, maior, aux;
    maior = i;
    if (2*i+1 <= qtde)
    {
        // o nó que está sendo analisado
        // tem filhos para esquerda e direita
        f_esq = 2*i+1;
        f_dir = 2*i;
        if (X[f_esq] >= X[f_dir] && X[f_esq] > X[i])
            maior = 2*i+1;
        else if (X[f_dir] > X[f_esq] && X[f_dir] > X[i])
            maior = 2*i;
    }
    else if (2*i <= qtde)
    {
        // o nó que está sendo analisado
        // tem filho apenas para a direita
        f_dir = 2*i;
        if (X[f_dir] > X[i])
            maior = 2*i;
    }
    if (maior != i)
    {
        aux = X[i];
        X[i] = X[maior];
        X[maior] = aux;
        heap_fica(maior,qtde);
    }
}
```

```

    }

    public static void ordena(int qtde)
    {
        int i, aux, ultima_posi;
        for (i=qtde;i>=2;i--)
        {
            aux = X[1];
            X[1] = X[i];
            X[i] = aux;
            ultima_posi = i - 1;
            heap_fica(1,ultima_posi);
        }
    }
}

```



C/C++

```

#include <iostream.h>
#include <conio.h>

int X[11];

void heap_fica(int i, int qtde)
{
    int f_esq, f_dir, maior, aux;
    maior = i;
    if (2*i+1 <= qtde)
    {
        // o nó que está sendo analisado
        // tem filhos para esquerda e direita
        f_esq = 2*i+1;
        f_dir = 2*i;
        if (X[f_esq] >= X[f_dir] && X[f_esq] > X[i])
            maior = 2*i+1;
        else if (X[f_dir] > X[f_esq] && X[f_dir] > X[i])
            maior = 2*i;
    }
    else if (2*i <= qtde)
    {
        // o nó que está sendo analisado
        // tem filho apenas para a direita
        f_dir = 2*i;
        if (X[f_dir] > X[i])

```

```
        maior = 2*i;
    }
    if (maior != i)
    {
        aux = X[i];
        X[i] = X[maior];
        X[maior] = aux;
        heap_fica(maior,qtde);
    }
}

void transforma_heap(int qtde)
{
    int i, pai, aux;
    for (i=qtde/2;i>=1;i--)
    {
        heap_fica(i,qtde);
    }
}

void ordena(int qtde)
{
    int i, aux, ultima_posi;
    for (i=qtde;i>=2;i--)
    {
        aux = X[1];
        X[1] = X[i];
        X[i] = aux;
        ultima_posi = i - 1;
        heap_fica(1,ultima_posi);
    }
}

void main()
{
    int i,qtde;
    clrscr();
    // carregando os números no vetor
    for(i=1;i<=10;i++)
    {
        cout<<"Digite o "<<i<<"º número: ";
        cin>>X[i];
    }
    // transforma o vetor digitado em um heap
    // cada nó pai é maior que seus filhos
```

```

qtde=10;
transforma_heap(qtde);
// ordenando de forma crescente
ordena(qtde);
// mostrando o vetor ordenado
for(i=1;i<=10;i++)
{
    cout<<"\n"<<i<<"º número: "<<x[i];
}
getch();
}

```

Análise da complexidade

Para realizar a análise do algoritmo de ordenação *HEAP SORT*, é necessário antes analisar a complexidade de dois procedimentos: *heap_fica* e *transforma_heap*.

O procedimento *heap_fica* é aplicado sempre a um nó da árvore, que representa na verdade um elemento do vetor, e “afunda” esse nó até que a propriedade *HEAP* seja válida. O pior caso ocorre quando o procedimento é aplicado sobre o primeiro elemento (raiz da árvore) e este deve afundar até alcançar uma folha. Logo, o número de trocas realizadas corresponderá à altura da árvore, que é $\log n$. Portanto, o procedimento *heap_fica* gasta $O(\log n)$.

O procedimento *transforma_heap* utiliza o procedimento *heap_fica*, conforme mostrado a seguir. Considerando que *qtde* é o número de elementos do vetor que formam o *HEAP*, e a estrutura de repetição para será executada $qtde/2$ vezes e ainda que o tempo de execução do *heap_fica* é $\log n$, portanto, o tempo de execução do procedimento *transforma_heap* é $T(n) = \frac{n}{2} \cdot \log n = O(n \cdot \log n)$.



1. para $i \leftarrow qtde/2$ até 1 faça passo -1
 2. início
 3. *heap_fica*($i, qtde$)
 4. fim
-

Após a transformação do vetor de entrada em um *HEAP*, através do procedimento *transforma_heap*, realiza-se de fato a ordenação por meio do procedimento *ordena*, como mostrado a seguir.



- ```

Função ordena(qtde numérico)
1. início
2. declare i, aux, ultima_posi numérico

```
-

```

3. para i ← qtde até 2 faça passo -1
4. início
5. aux ← X[1]
6. X[1] ← X[i]
7. X[i] ← aux
8. ultima_posi ← i-1
9. heap_fica(1,ultima_posi)
10. fim
11. fim_função_ordena.

```

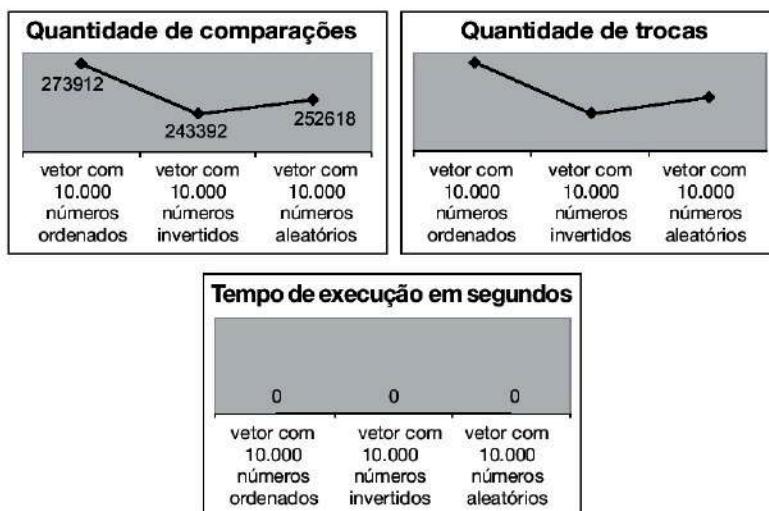
O procedimento possui um laço que é executado  $n - 1$  vezes. Internamente ao laço, existem linhas de atribuição que têm custo constante,  $O(1)$ , e a linha 9, que é uma chamada para o já analisado procedimento `heap_fica`, que custa  $\log n$ . Portanto, o tempo de execução do procedimento `ordena` é  $T(n) = (n - 1) \cdot \log n = O(n \cdot \log n)$ . Mesmo sendo necessário realizar a transformação do vetor em um *HEAP* antes da ordenação de fato, o que custa como visto ainda nesta seção,  $n \cdot \log n$ , o tempo do algoritmo *HEAP SORT* não ultrapassa o limitante  $n \cdot \log n$ .

Veja os gráficos de desempenho na Figura 2.8.

## Algoritmo de busca sequencial

O algoritmo de busca sequencial pode ser executado em um vetor não ordenado e em um vetor ordenado. Em um vetor não ordenado, será buscado o número até que ele

**Figura 2.8** Gráficos de desempenho



seja encontrado ou até se chegar ao final do vetor. Em um vetor ordenado, será buscado o número até que ele seja encontrado e enquanto for maior que o número do vetor.

A primeira ilustração é feita para um vetor não ordenado.



ILUSTRAÇÃO

**1<sup>a</sup> execução do laço**

nº procurado

8 | 8 = 5  $\Rightarrow$  Falso

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 3 | 1 | 8 | 2 |

**2<sup>a</sup> execução do laço**

nº procurado

8 | 8 = 3  $\Rightarrow$  Falso

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 3 | 1 | 8 | 2 |

**3<sup>a</sup> execução do laço**

nº procurado

8 | 8 = 1  $\Rightarrow$  Falso

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 3 | 1 | 8 | 2 |

**4<sup>a</sup> execução do laço**

nº procurado

8 | 8 = 8  $\Rightarrow$  Verdadeiro, nº encontrado

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 3 | 1 | 8 | 2 |

A segunda ilustração é feita para um vetor ordenado.



ILUSTRAÇÃO

**1<sup>a</sup> execução do laço**

nº procurado

4 | 4 = 1  $\Rightarrow$  Falso, 4 > 1  $\Rightarrow$  Verdadeiro, continua.

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 3 | 5 | 7 | 9 |

**2<sup>a</sup> execução do laço**

nº procurado

4

4 = 3 → Falso, 4 &gt; 3 → Verdadeiro, continua.

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 3 | 5 | 7 | 9 |

**3<sup>a</sup> execução do laço**

nº procurado

4

4 = 5 → Falso, 4 &gt; 5 → Falso, nº Não encontrado.

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 3 | 5 | 7 | 9 |



ALGORITMO

```

algoritmo
declare X[10], i, n, achou numérico
// carregando os números no vetor
// vetor com números NÃO ORDENADOS
para i ← 1 até 10 faça
 início
 escreva "Digite o ",i,"º número: "
 leia X[i]
 fim
// digitando o número a ser buscado no vetor
escreva "Digite o número a ser buscado no vetor: "
leia n
// buscando o número digitado no vetor
achou ← 0
i ← 0
enquanto (i <= 9 e achou = 0) faça
 início
 se (X[i] = n)
 então achou ← 1
 senão i ← i + 1
 fim
 se (achou = 0)
 então escreva "Número não encontrado no vetor"
 senão escreva "Número encontrado na posição ",i+1
fim_algoritmo.

```



```

algoritmo
declare X[10], i, n, achou numérico
// carregando os números no vetor
// vetor com números ORDENADOS
para i ← 1 até 10 faça
 início
 escreva "Digite o ",i,"º número: "
 leia X[i]
 fim
// digitando o número a ser buscado no vetor
escreva "Digite o número a ser buscado no vetor: "
leia n
// buscando o número digitado no vetor
achou ← 0
i ← 0
enquanto (i <= 9 e achou = 0 e n >= X[i]) faça
 início
 se (X[i] = n)
 então achou ← 1
 senão i ← i + 1
 fim
se (achou = 0)
então escreva "Número não encontrado no vetor"
senão escreva "Número encontrado na posição ",i+1
fim_algoritmo.

```



```

import java.util.*;
public class busca_sequencial_desordenado
{
 public static void main (String args[])
 {
 int X[] = new int[10];
 int n, i, achou;
 Scanner entrada = new Scanner(System.in);
 // carregando os números no vetor
 // vetor com números ORDENADOS
 for(i=0;i<=9;i++)
 {
 System.out.println("Digite o "+(i+1)+"º número: ");
 X[i] = entrada.nextInt();
 }
 // digitando o número a ser buscado no vetor

```

```

 System.out.println("Digite o número a ser buscado no
 ↵ vetor: ");
 n = entrada.nextInt();
 // buscando o número digitado no vetor
 achou = 0;
 i = 0;
 while (i <= 9 && achou == 0)
 {
 if (X[i] == n)
 achou = 1;
 else
 i++;
 }
 if (achou == 0)
 System.out.println("Número não encontrado no
 ↵ vetor");
 else
 System.out.println("Número encontrado na posição
 ↵ "+(i+1));
 }
}

```




---

```

import java.util.*;
public class busca_sequencial_ordenado
{
 public static void main (String args[])
 {
 int X[] = new int[10];
 int n, i, achou;
 Scanner entrada = new Scanner(System.in);
 // carregando os números no vetor
 // vetor com números ORDENADOS
 for(i=0;i<=9;i++)
 {
 System.out.println("Digite o "+(i+1)+"º número: ");
 X[i] = entrada.nextInt();
 }
 // digitando o número a ser buscado no vetor
 System.out.println("Digite o número a ser buscado no
 ↵ vetor: ");
 n = entrada.nextInt();
 // buscando o número digitado no vetor
 achou = 0;
 }
}

```

```

 i = 0;
 while (i <= 9 && achou == 0 && n >= X[i])
 {
 if (X[i] == n)
 achou = 1;
 else
 i++;
 }
 if (achou == 0)
 System.out.println("Número não encontrado no
 ↵ vetor");
 else
 System.out.println("Número encontrado na posição
 ↵ "+(i+1));
 }
}

```




---

```

#include <iostream.h>
#include <conio.h>
void main()
{
 int X[10], n, i, achou;
 clrscr();
 // carregando os números no vetor
 // vetor com números NAO ORDENADOS
 for(i=0;i<=9;i++)
 {
 cout<<"Digite o "<<i+1<<"º número: ";
 cin>>X[i];
 }
 // digitando o número a ser buscado no vetor
 cout<<"Digite o número a ser buscado no vetor: ";
 cin>>n;
 // buscando o número digitado no vetor
 achou = 0;
 i = 0;
 while (i <= 9 && achou == 0)
 {
 if (X[i] == n)
 achou = 1;
 }
}

```

```

 else
 i++;
 }
 if (achou == 0)
 cout<<"Número não encontrado no vetor";
 else
 cout<<"Número encontrado na posição "<<i+1;
 getch();
}

```



c/c++

---

```

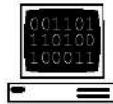
#include <iostream.h>
#include <conio.h>
void main()
{
 int X[10], n, i, achou;
 clrscr();
 // carregando os números no vetor
 // vetor com números ORDENADOS
 for(i=0;i<=9;i++)
 {
 cout<<"Digite o "<<i+1<<"º número: ";
 cin>>X[i];
 }
 // digitando o número a ser buscado no vetor
 cout<<"Digite o número a ser buscado no vetor: ";
 cin>>n;
 // buscando o número digitado no vetor
 achou = 0;
 i = 0;
 while (i <= 9 && achou == 0 && n >= X[i])
 {
 if (X[i] == n)
 achou = 1;
 else
 i++;
 }
 if (achou == 0)
 cout<<"Número não encontrado no vetor";
 else
 cout<<"Número encontrado na posição "<<i+1;
 getch();
}

```

---

## Análise da complexidade

O trecho de algoritmo abaixo mostra a busca sequencial em um vetor não ordenado.



ALGORITMO

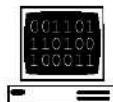
```

1. achou ← 0
2. i ← 0
3. enquanto (i <= 9 e achou = 0) faça
4. início
5. se (X[i] = n)
6. então achou ← 1
7. senão i ← i + 1
8. fim

```

Como possui apenas uma estrutura de repetição, o pior caso da busca acontece quando o número procurado é o último elemento do vetor ou quando o número procurado não se encontra no vetor, realizando-se então  $n$  comparações. Logo, o tempo de execução é  $T(n) = O(n)$ . Já o melhor caso ocorre quando o número procurado é o elemento que se encontra na primeira posição, sendo comparado uma única vez e cujo tempo de execução é constante, ou seja,  $T(n) = O(1)$ .

O trecho de algoritmo abaixo mostra a busca sequencial em um vetor ordenado. Neste caso, as comparações são realizadas até o número ser encontrado ou até encontrar-se um número maior que aquele procurado.



ALGORITMO

```

1. achou ← 0
2. i ← 0
3. enquanto (i <= 9 e achou = 0 e n >= X[i]) faça
4. início
5. se (X[i] = n)
6. então achou ← 1
7. senão i ← i + 1
8. fim

```

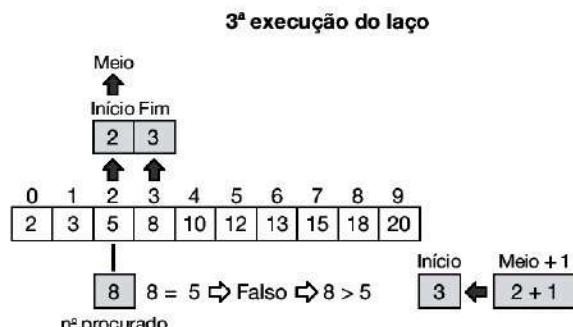
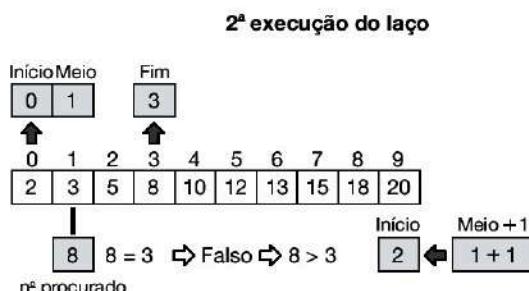
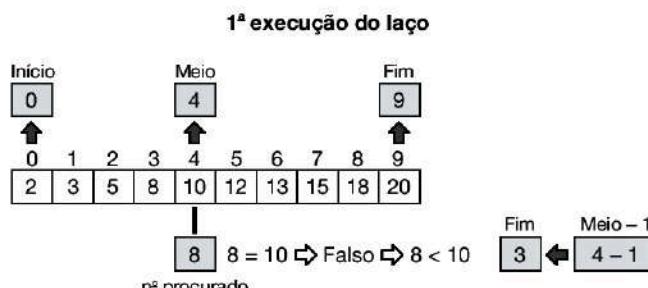
O pior caso da busca acontece quando o número procurado é o último elemento do vetor ou quando o número procurado não se encontra no vetor, realizando-se então  $n$  comparações. Logo, o tempo de execução é  $T(n) = O(n)$ . Já o melhor caso ocorre quando o número procurado é o elemento que se encontra na primeira posição, realizando uma única comparação e cujo tempo de execução é constante, ou seja,  $T(n) = O(1)$ .

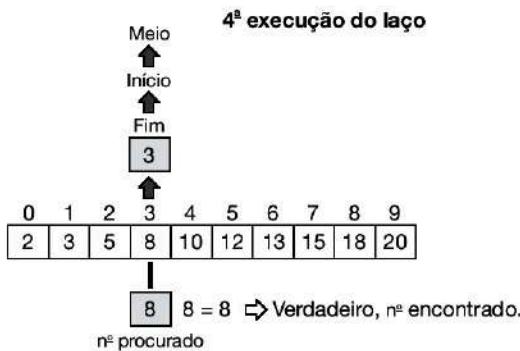
## Algoritmo de busca binária

O algoritmo de busca binária é executado somente em vetores ordenados. Nesse algoritmo o vetor com os dados é dividido ao meio e o número do meio é comparado com o número procurado. Se estes forem iguais, a busca termina, caso contrário, se o número procurado é menor que o do meio, a busca será realizada no vetor à esquerda ao do meio. Se o número procurado é maior que o do meio, a busca será realizada no vetor à direita ao vetor do meio. Esse procedimento de divisão e comparação acontece até que o vetor de dados fique com apenas um elemento ou até o número procurado ser encontrado.



ILUSTRAÇÃO

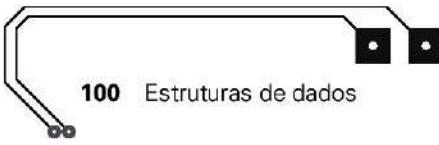




```

algoritmo
declare X[10], i, np, achou, inicio, fim, meio numérico
// carregando os números no vetor
para i ← 1 até 10 faça
 início
 escreva "Digite o ",i,"º número: "
 leia X[i]
 fim
// digitando o número a ser buscado no vetor
escreva "Digite o número a ser buscado no vetor: "
leia np
// buscando o número digitado no vetor
achou ← 0
início ← 0
fim ← 9
meio ← parteinteira((início+fim)/2)
enquanto (início <= fim e achou = 0) faça
 início
 se (X[meio] = np)
 então achou ← 1
 senão início
 se (np < X[meio])
 então fim ← meio-1
 senão início ← meio+1
 meio ← parteinteira((início+fim)/2)
 fim

```



## 100 Estruturas de dados

```
 fim
 se (achou = 0)
então escreva "Número não encontrado no vetor"
senão escreva "Número encontrado na posição ",meio
fim_algoritmo.
```

---



J A V A

```
import java.util.*;
public class busca_binaria
{
 public static void main (String args[])
 {
 int X[] = new int[10];
 int np, i, inicio, fim, meio, achou;
 Scanner entrada = new Scanner(System.in);
 // carregando os números no vetor - ORDENADOS
 for(i=0;i<=9;i++)
 {
 System.out.println("Digite o "+(i+1)+"º número: ");
 X[i] = entrada.nextInt();
 }
 // digitando o número a ser buscado no vetor
 System.out.println("Digite o número a ser buscado no
 → vetor: ");
 np = entrada.nextInt();
 // buscando o número digitado no vetor
 achou = 0;
 inicio = 0;
 fim = 9;
 meio = (inicio+fim)/2;
 while (inicio <= fim && achou == 0)
 {
 if (X[meio] == np)
 achou = 1;
 else {
 if (np < X[meio])
 fim = meio-1;
 else
 inicio = meio+1;
 meio = (inicio+fim)/2;
 }
 }
 }
}
```

```

 }
 }
 if (achou == 0)
 System.out.println("Número não encontrado no
 vetor");
 else
 System.out.println("Número encontrado na posição
 "+meio);
 }
}

```

---



C/C++

```

#include <iostream.h>
#include <conio.h>
void main()
{
 int X[10], np, i, inicio, fim, meio, achou;
 clrscr();
 // carregando os números no vetor - ORDENADOS
 for(i=0;i<=9;i++)
 {
 cout<<"Digite o "<<i+1<<"º número: ";
 cin>>X[i];
 }
 // digitando o número a ser buscado no vetor
 cout<<"Digite o número a ser buscado no vetor: ";
 cin>>np;
 // buscando o número digitado no vetor
 achou = 0;
 inicio = 0;
 fim = 9;
 meio = (inicio+fim)/2;
 while (inicio <= fim && achou == 0)
 {
 if (X[meio] == np)
 achou = 1;
 else {
 if (np < X[meio])
 fim = meio-1;
 else

```

```

 inicio = meio+1;
 meio = (inicio+fim)/2;
 }
}
if (achou == 0)
 cout<<"Número não encontrado no vetor";
else
 cout<<"Número encontrado na posição "<<meio+1;
getch();
}

```

---

## Análise da complexidade



ALGORITMO

1. achou  $\leftarrow$  0
2. início  $\leftarrow$  0
3. fim  $\leftarrow$  9
4. meio  $\leftarrow$  parteinteira((início+fim)/2)
5. enquanto (início  $\leq$  fim e achou = 0) faça
6.     início
7.         se (X[meio] = np)
8.             então achou  $\leftarrow$  1
9.         senão início
10.             se (np < X[meio])
11.                 então fim  $\leftarrow$  meio-1
12.                 senão início  $\leftarrow$  meio+1
13.             meio  $\leftarrow$  parteinteira((início+fim)/2)
14.         fim
15. fim

---

O algoritmo da busca binária, ao procurar um elemento  $np$ , compara-o com o elemento do meio do vetor cujo tamanho de entrada é  $n$ . Como o vetor está ordenado, caso  $np$  não seja encontrado na posição meio, e sendo  $np$  menor que  $X[meio]$ , passa-se a procurar o número  $np$  no lado esquerdo da posição meio (metade esquerda  $\hat{e}n/2\hat{u}$  do vetor) ou no lado direito da posição meio (metade direita  $\hat{e}n/2\hat{u}$  do vetor) se o número  $np$  for maior que o elemento  $X[meio]$ . O novo vetor a ser buscado agora passa a ter tamanho próximo a  $n/2$ , visto que o elemento da posição meio não é mais considerado. O mesmo processo é refeito e, se o número não for encontrado, o vetor que antes tinha tamanho  $n/2$ , passa a ter tamanho  $n/4$ . Na próxima iteração terá

tamanho  $n/8$  e assim por diante, até alcançarmos um vetor de tamanho 1. Assim, na última iteração a condição `início <= fim` torna-se falsa, o que representa que o vetor fica com tamanho menor que 1.

Portanto, o número de iterações realizadas depende do tamanho do vetor. No início da primeira iteração, o vetor tem tamanho  $n$ . No início da segunda, vale aproximadamente  $n/2$ . No início da terceira,  $n/4$ . No início da  $(k+1)$ -ésima,  $n/2^k$ .

Quando  $k$  passar de  $\log_2 n$ , o valor da expressão  $n/2^k$  fica menor que 1 e o algoritmo para. Logo, o número de iterações é aproximadamente  $\lg n$  no pior caso, sendo  $\lg n$  o piso de  $\log_2 n$ .

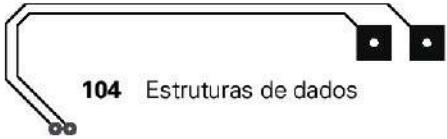
Para se entender o valor encontrado para  $k$  no parágrafo anterior, considere que o vetor atingirá em algum momento o tamanho 1, que ocorre quando  $n/2^k = 1$ , ou seja,

$$\begin{aligned}\frac{n}{2^k} &= 1 \\ n &= 2^k \\ \log_2 n &= \log_2 2^k \\ \log_2 n &= k.\end{aligned}$$

O consumo de tempo da busca binária é, portanto, proporcional a  $\lg n$ .

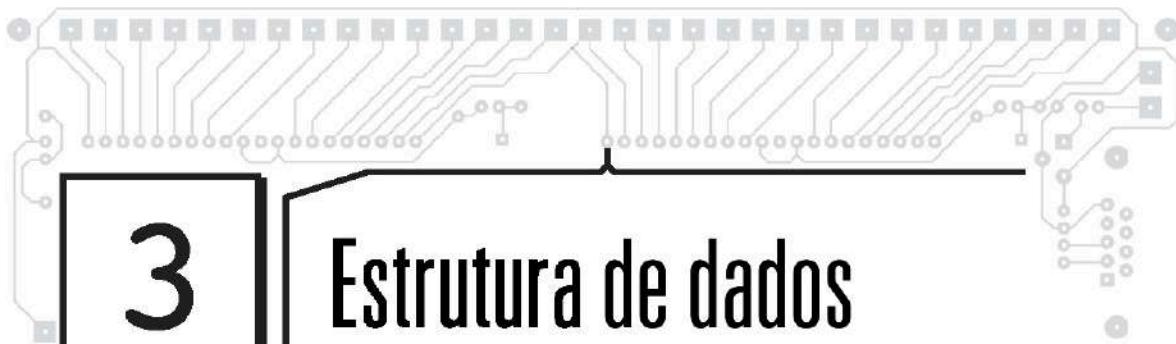
## Exercícios

1. Faça um programa que cadastre o nome e o salário de 5 funcionários. Usando três métodos de ordenação diferentes, liste todos os dados dos funcionários das seguintes formas:
  - a) em ordem crescente de salário;
  - b) em ordem decrescente de salário;
  - c) em ordem alfabética.
2. Faça um programa que cadastre 10 números, ordene-os e em seguida encontre e mostre:
  - a) o menor número e quantas vezes ele aparece no vetor;
  - b) o maior número e quantas vezes ele aparece no vetor.
3. Faça um programa que cadastre 12 produtos. Para cada produto devem ser cadastrados os seguintes dados: código, descrição e preço. Use um método de ordenação e em seguida calcule e mostre quantas comparações devem ser feitas para encontrar um funcionário pelo código:
  - a) usando busca sequencial;
  - b) usando busca binária.
4. Faça um programa que cadastre 8 alunos. Para cada aluno devem ser cadastrados: nome, nota 1 e nota 2. Primeiro, liste todos os alunos cadastrados ordenando-os pela média ponderada das notas, tendo a primeira nota peso 2 e a segunda peso 3. Em seguida, ordene os alunos, de forma crescente, pela nota 1, e liste-os. Finalmente, considerando que para ser aprovado o aluno dever ter no mínimo média 7, liste, em ordem alfabética, os alunos reprovados.



## **104** Estruturas de dados

- 5.** Faça um programa que cadastre 15 números, não permitindo números repetidos. Orde-ne-os e, em seguida, verifique se um número digitado pelo usuário está no vetor. Caso encontre, verifique se está em uma posição par ou ímpar do vetor:
- a)** usando busca sequencial;
  - b)** usando busca binária.



# 3

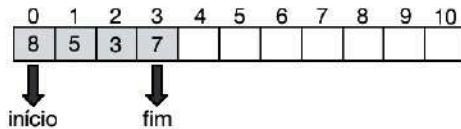
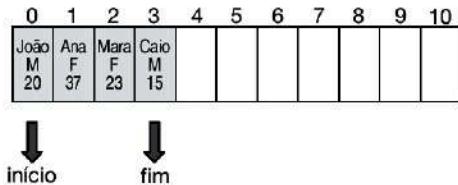
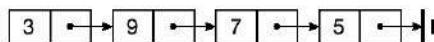
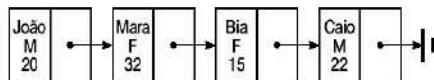
# Estrutura de dados do tipo listas

Em Ciência da Computação é fundamental o trabalho com conjuntos de dados, que podem representar coleções de: números, dados de um funcionário, dados de um produto, entre outros. Esses conjuntos são aqui chamados de *conjuntos dinâmicos*, pois os algoritmos que os manipulam fazem com que eles cresçam, encolham ou sofram alterações ao longo do tempo. Assim, as principais operações sobre os conjuntos dinâmicos são: inserir um elemento, excluir outro, buscá-lo, encontrar o maior, o menor, contar os elementos, alterá-los, buscar o elemento sucessor e o predecessor.

Uma estrutura de dados do tipo lista representa um conjunto de dados organizados em ordem linear. Quando a estrutura lista é representada por um arranjo, ou seja, é feita a utilização de vetores na representação, tem-se o uso de endereços contíguos de memória do computador e a ordem linear é determinada pelos índices do vetor, o que em algumas situações exige um maior esforço computacional. Tal representação denomina-se lista estática. Quando a estrutura lista é representada por elementos que, além de conter o dado, possuem também um ponteiro para o próximo elemento, ou seja, elementos encadeados, tem-se a representação denominada lista dinâmica. Toda lista dinâmica tem pelo menos um ponteiro para o início.

Quando um elemento de uma lista contém apenas um dado primitivo, como um número, chama-se lista homogênea, e quando um elemento de uma lista contém um dado composto, como o nome e o salário de um funcionário, chama-se lista heterogênea. As figuras de 3.1 a 3.4 ilustram as listas estáticas e dinâmicas, homogêneas e heterogêneas.

A estrutura de dados do tipo lista pode ser representada em cinco tipos diferentes, e cada um pode ser implementado de forma estática ou dinâmica: lista simplesmente encadeada e não ordenada, lista simplesmente encadeada e ordenada, lista duplamente encadeada e não ordenada, lista duplamente encadeada e ordenada e listas circulares. As próximas seções mostram os tipos acima citados com ilustração, implementação dinâmica e análise.

**Figura 3.1** Lista estática homogênea**Figura 3.2** Lista estática heterogênea**Figura 3.3** Lista dinâmica homogênea**Figura 3.4** Lista dinâmica heterogênea

## Lista simplesmente encadeada e não ordenada

Nesta estrutura, cada elemento armazena um ou vários dados (estrutura homogênea ou heterogênea, respectivamente) e um ponteiro para o próximo elemento, que permite o encadeamento e mantém a estrutura linear. Nesse tipo de estrutura serão abordadas as seguintes operações: inserir no início da lista, inserir no fim, consultar toda a lista, remover um elemento qualquer dela e esvaziá-la.

A seguir, uma ilustração com endereços de memória meramente ilustrativos.

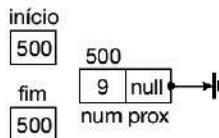


ILUSTRAÇÃO

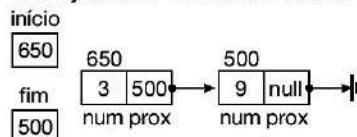
**1<sup>a</sup> operação**  
A lista está vazia



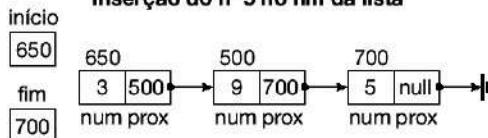
**2<sup>a</sup> operação**  
Inserção do nº 9 no início da lista



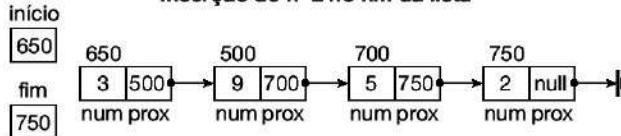
**3<sup>a</sup> operação**  
Inserção do nº 3 no início da lista



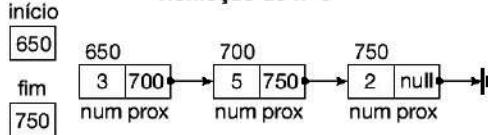
**4<sup>a</sup> operação**  
Inserção do nº 5 no fim da lista



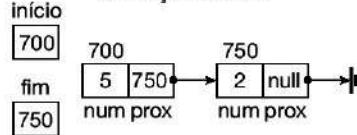
**5<sup>a</sup> operação**  
Inserção do nº 2 no fim da lista



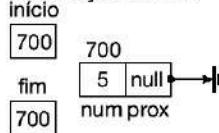
**6<sup>a</sup> operação**  
Remoção do nº 9



**7<sup>a</sup> operação**  
Remoção do nº 3



**8<sup>a</sup> operação**  
Remoção do nº 2





```
import java.util.*;
public class LS_Nao_Ordenada
{
 //Definindo a classe que representará
 //cada elemento da lista
 private static class LISTA
 {
 public int num;
 public LISTA prox;
 }

 public static void main(String args[])
 {
 Scanner entrada = new Scanner(System.in);
 // a lista está vazia, logo,
 // o objeto inicio tem o valor null
 // o objeto inicio conterá o endereço
 // do primeiro elemento da lista
 LISTA inicio = null;
 // o objeto fim conterá o endereço
 // do último elemento da lista
 LISTA fim = null;
 // o objeto aux é um objeto auxiliar
 LISTA aux;
 // o objeto anterior é um objeto auxiliar
 LISTA anterior;
 // apresentando o menu de opções
 int op, numero, achou;
 do
 {
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir no início
 ↪ da lista");
 System.out.println("2 - Inserir no fim da
 ↪ lista");
 System.out.println("3 - Consultar toda a
 ↪ lista");
 System.out.println("4 - Remover da lista");
 System.out.println("5 - Esvaziar a lista");
 System.out.println("6 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();
 if (op < 1 || op > 6)
 achou = 1;
 else
 switch (op)
 {
 case 1:
 // inserir no início
 // cria novo nó
 // aponta para o inicio
 // aponta para o proximo
 // ...
 break;
 case 2:
 // inserir no fim
 // cria novo nó
 // aponta para o fim
 // aponta para o proximo
 // ...
 break;
 case 3:
 // consultar toda a lista
 // ...
 break;
 case 4:
 // remover da lista
 // ...
 break;
 case 5:
 // esvaziar a lista
 // ...
 break;
 case 6:
 // sair
 // ...
 break;
 }
 } while (achou == 0);
 }
}
```

```
System.out.println("Opção inválida!!");
if (op == 1)
{
 System.out.println("Digite o número a
 ser inserido no início da lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo.prox = null;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no início da lista
 novo.prox = inicio;
 inicio = novo;
 }
 System.out.println("Número inserido no
 início da lista!!");
}
if (op == 2)
{
 System.out.println("Digite o número a
 ser inserido no fim da lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo.prox = null;
 }
 else
 {
 // a lista já contém elementos
```

```
// é o novo elemento
// será inserido no fim da lista
fim.prox = novo;
fim = novo;
fim.prox = null;
}
System.out.println("Número inserido no fim da
➥ lista!!!");
}
if (op == 3)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!!");
 }
 else
 {
 // a lista contém elementos e estes
 // serão mostrados do início ao fim
 System.out.println("\nConsultando toda
➥ a lista\n");
 aux = inicio;
 while (aux != null)
 {
 System.out.print(aux.num+" ");
 aux = aux.prox;
 }
 }
}
if (op == 4)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!!");
 }
 else
 {
 // a lista contém elementos
 // e o elemento a ser
 // removido deve ser digitado
 System.out.print("\nDigite o elemento a ser
➥ removido: ");
 numero = entrada.nextInt();
 // todas as ocorrências da lista,
```

```
// iguais ao número digitado,
// serão removidas
aux = inicio;
anterior = null;
achou = 0;
while (aux != null)
{
 if (aux.num == numero)
 {
 // o número digitado
 // foi encontrado na lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a ser removido
 // é o primeiro da lista
 inicio = aux.prox;
 aux = inicio;
 }
 else if (aux == fim)
 {
 // o número a ser
 // removido
 // é o último da lista
 anterior.prox = null;
 fim = anterior;
 aux = null;
 }
 else
 {
 // o número a ser removido
 // está no meio da lista
 anterior.prox = aux. prox;
 aux = aux.prox;
 }
 }
 else
 {
 anterior = aux;
 aux = aux.prox;
 }
}
if (achou == 0)
 System.out.println("Número não
 ↵ encontrado");
```

```

 else if (achou == 1)
 System.out.println("Número
 ↪ removido 1 vez");
 else
 System.out.println("Número
 ↪ removido "+achou+" vezes");
 }
}
if (op == 5)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!!");
 }
 else
 {
 // a lista será esvaziada
 inicio = null;
 System.out.println("Lista
 ↪ esvaziada");
 }
}
while (op != 6);
}
}

```

---



```
#include <iostream.h>
#include <conio.h>
```

```

void main()
{
//Definindo o registro que representará
//cada elemento da lista
struct LISTA
{
 int num;
 LISTA *prox;
};

```

```
// a lista está vazia, logo,
// o ponteiro inicio tem o valor NULL
// o ponteiro inicio conterá o endereço
// do primeiro elemento da lista
LISTA *inicio = NULL;
// o ponteiro fim conterá o endereço
// do último elemento da lista
LISTA *fim = NULL;
// o ponteiro aux é um ponteiro auxiliar
LISTA *aux;
// o ponteiro anterior é um ponteiro auxiliar
LISTA *anterior;
// apresentando o menu de opções
int op, numero, achou;
do
{
 clrscr();
 cout<<"\nMENU DE OPÇÕES\n";
 cout<<"\n1 - Inserir no início da
 ↵ lista";
 cout<<"\n2 - Inserir no fim da lista";
 cout<<"\n3 - Consultar toda a lista";
 cout<<"\n4 - Remover da lista";
 cout<<"\n5 - Esvaziar a lista";
 cout<<"\n6 - Sair";
 cout<<"\nDigite sua opção: ";
 cin>>op;
 if (op < 1 || op > 6)
 cout<<"Opção inválida!!";
 if (op == 1)
 {
 cout<<"Digite o número a ser
 ↵ inserido no início da lista:";
 LISTA *novo = new LISTA();
 cin>>novo->num;
 if (inicio == NULL)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 fim->prox = NULL;
 }
 else
 {
```

```
// a lista já contém elementos
// e o novo elemento
// será inserido no início da
// lista
novo->prox = inicio;
inicio = novo;
}
cout<<"Número inserido no
➥ início da lista!!";
}
if (op == 2)
{
 cout<<"Digite o número a ser
➥ inserido no fim da lista: ";
LISTA *novo = new LISTA();
cin>>novo->num;
if (inicio == NULL)
{
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 fim->prox = NULL;
}
else
{
 // a lista já contém elementos e
 // o novo elemento
 // será inserido no fim da lista
 fim->prox = novo;
 fim = novo;
 fim->prox=NULL;
}
cout<<"Número inserido no fim da
➥ lista!!";
}
if (op == 3)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
```

```
// a lista contém elementos e estes serão
// mostrados do inicio ao fim
cout<<"\nConsultando toda a lista\n";
aux = inicio;
while (aux != NULL)
{
 cout<<aux->num<<" ";
 aux = aux->prox;
}
}

if (op == 4)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista contém elementos e o elemento
 // a ser removido deve ser digitado
 cout<<"\nDigite o elemento a ser removido:";
 cin>>numero;
 // todas as ocorrências da lista, iguais ao
 // número digitado, serão removidas
 aux = inicio;
 anterior = NULL;
 achou = 0;
 while (aux != NULL)
 {
 if (aux->num == numero)
 {
 // o número digitado
 // foi encontrado na lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a ser removido
 // é o primeiro da lista
 inicio = aux->prox;
 delete(aux);
 aux = inicio;
 }
 }
 }
 }
}
```

```
 else if (aux == fim)
 {
 // o número a ser removido
 // é o último da lista
 anterior->prox = NULL;
 fim = anterior;
 delete(aux);
 aux = NULL;
 }
 else
 {
 // o número a ser
 // removido
 // está no meio da
 // lista
 anterior->prox = aux-
 ↵ >prox;
 delete(aux);
 aux = anterior->prox;
 }
else
{
 anterior = aux;
 aux = aux->prox;
}
}
if (achou == 0)
 cout<<"Número não encontrado";
else if (achou == 1)
 cout<<"Número removido 1 vez";
else
 cout<<"Número removido "<<achou<<
 ↵ vezes";
}
}
if (op == 5)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!";
 }
else
{
```

```

 // a lista será esvaziada
 aux = inicio;
 while (aux != NULL)
 {
 inicio = inicio->prox;
 delete(aux);
 aux=inicio;
 }
 cout<<"Lista esvaziada";
}
getch();
}
while (op != 6);
}

```

## Lista simplesmente encadeada e ordenada

Nesta estrutura, cada elemento armazena um ou vários dados (estrutura homogênea ou heterogênea, respectivamente) e um ponteiro para o próximo elemento, que permite o encadeamento e mantém a estrutura linear. Tem-se também um campo denominado chave através do qual uma determinada ordenação é mantida. Nesse tipo de estrutura serão abordadas as seguintes operações: inserir na lista, consultar toda a lista, remover um elemento qualquer dela e esvaziá-la.

A seguir, uma ilustração com endereços de memória meramente ilustrativos, cujos números da lista estão ordenados de forma crescente.

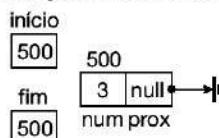


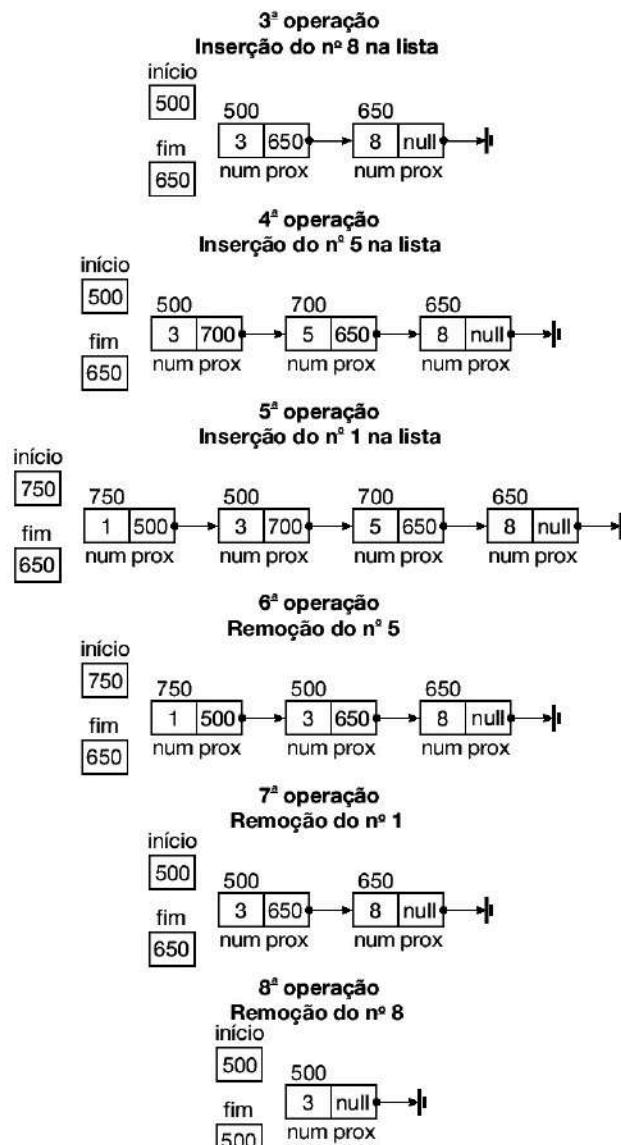
ILUSTRAÇÃO

**1<sup>a</sup> operação**  
A lista está vazia

início  
[null]

**2<sup>a</sup> operação**  
Inserção do nº 3 na lista






---

```

import java.util.*;
public class LS_Ordenada
{
 //Definindo a classe que representará
 //cada elemento da lista
 private static class LISTA

```

```
{
 public int num;
 public LISTA prox;
}

public static void main(String args [])
{
 Scanner entrada = new Scanner(System.in);
 // a lista está vazia, logo,
 // o objeto inicio tem o valor null
 // o objeto inicio conterá o endereço
 // do primeiro elemento da lista
 LISTA inicio = null;
 // o objeto fim conterá o endereço
 // do último elemento da lista
 LISTA fim = null;
 // o objeto aux é um objeto auxiliar
 LISTA aux;
 // o objeto anterior é um objeto auxiliar
 LISTA anterior;
 // apresentando o menu de opções
 int op, numero, achou;
 do
 {
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir na lista");
 System.out.println("2 - Consultar toda a
 ↪ lista");
 System.out.println("3 - Remover da lista");
 System.out.println("4 - Esvaziar a lista");
 System.out.println("5 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();
 if (op < 1 || op > 5)
 System.out.println("Opção inválida!!");
 if (op == 1)
 {
 System.out.println("Digite o número a
 ↪ ser inserido na lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
```

```
// o elemento inserido será
// o primeiro e o último
inicio = novo;
fim = novo;
novo.prox = null;
}
else
{
 // a lista já contém elementos
 // e o novo elemento
 // será inserido na lista
 // respeitando a ordenação
 // crescente
 anterior = null;
 aux = inicio;
 while (aux != null &&
 novo.num > aux.num)
 {
 anterior = aux;
 aux = aux.prox;
 }
 if (anterior == null)
 {
 // o novo número a ser
 // inserido
 // é menor que todos os
 // números da lista,
 // logo, será inserido no
 // início
 novo.prox = inicio;
 inicio = novo;
 }
 else if (aux == null)
 {
 // o novo número a
 // ser inserido
 // é maior que todos os
 // números
 // da lista, logo,
 // será inserido no fim
 fim.prox = novo;
 fim = novo;
 fim.prox = null;
 }
}
```

```
 else
 {
 // o novo número a ser
 // inserido
 // será inserido entre
 // dois
 // números que já
 // estão na lista
 anterior.prox = novo;
 novo.prox = aux;
 }
 }
 System.out.println("Número
 → inserido na lista!!");
}
if (op == 2)
{
if (inicio == null)
{
 // a lista está vazia
 System.out.println("Lista
 → vazia!!");
}
else
{
 // a lista contém elementos e
 // estes serão
 // mostrados do início ao fim
 System.out.println("\
 → nConsultando toda a lista\n");
 aux = inicio;
 while (aux != null)
 {
 System.out.print(aux.
 → num+" ");
 aux = aux.prox;
 }
}
if (op == 3)
{
 if (inicio == null)
 {
```

```
// a lista está vazia
System.out.println("Lista
↳ vazia!!");

}

else
{
 // a lista contém
 // elementos
 // e o elemento a ser
 // removido deve ser
 // digitado
 System.out.print("\nDigite o
↳ elemento a ser
↳ removido: ");
 numero = entrada.
↳ nextInt();
 // todas as ocorrências
 // da lista,
 // iguais ao número
 // digitado,
 // serão removidas
 aux = inicio;
 anterior = null;
 achou = 0;
 while (aux != null)
 {
 if (aux.num == numero)
 {
 // o número digitado
 // foi encontrado na
 // lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a ser
 // removido
 // é o primeiro
 // da lista
 inicio = aux.prox;
 aux = inicio;
 }
 else if (aux == fim)
 {

```

```
// o número a
// ser
// removido
// é o último
// da lista
anterior.prox =
↳ null;
fim = anterior;
aux = null;
}
else {
 // o número a ser
 // removido
 // está no meio
 // da lista
 anterior.prox =
 ↳ aux.prox;
 aux = aux.prox;
}
}
else
{
 anterior = aux;
 aux = aux.prox;
}
}
if (achou == 0)
 System.out.println("Número não
 ↳ encontrado");
else if (achou == 1)
 System.out.println("Número
 ↳ removido 1 vez");
else
 System.out.println("Número
 ↳ removido +achou+ vezes");
}
}
if (op == 4)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!");
 }
}
```

```

 else
 {
 // a lista será esvaziada
 inicio = null;
 System.out.println("Lista esvaziada");
 }
 }
}
while (op != 5);
}
}

```

---



c/c++

```

#include <iostream.h>
#include <conio.h>

void main()
{
 //Definindo o registro que
 //representará cada elemento da lista
 struct LISTA
 {
 int num;
 LISTA *prox;
 };
 // a lista está vazia, logo,
 // o ponteiro inicio têm o valor NULL
 // o ponteiro inicio conterá o endereço
 // do primeiro elemento da lista
 LISTA *inicio = NULL;
 // o ponteiro fim conterá o endereço
 // do último elemento da lista
 LISTA *fim = NULL;
 // o ponteiro aux é um ponteiro auxiliar
 LISTA *aux;
 // o ponteiro anterior é um ponteiro auxiliar
 LISTA *anterior;
 // apresentando o menu de opções
 int op, numero, achou;
 do

```

```
{
 clrscr();
 cout<<"\nMENU DE OPÇÕES\n";
 cout<<"\n1 - Inserir na lista";
 cout<<"\n2 - Consultar toda a lista";
 cout<<"\n3 - Remover da lista";
 cout<<"\n4 - Esvaziar a lista";
 cout<<"\n5 - Sair";
 cout<<"\nDigite sua opção: ";
 cin>>op;
 if (op < 1 || op > 5)
 cout<<"Opção inválida!";
 if (op == 1)
 {
 cout<<"Digite o número a ser
 inserido na lista: ";
 LISTA *novo = new LISTA();
 cin>>novo->num;
 if (inicio == NULL)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo->prox = NULL;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido na lista
 // respeitando a ordenação
 // crescente
 anterior = NULL;
 aux = inicio;
 while (aux != NULL
 && novo->num > aux->num)
 {
 anterior = aux;
 aux = aux->prox;
 }
 if (anterior == NULL)
```

```
{
 // o novo número a ser inserido
 // é menor que todos os
 // números da lista,
 // logo, será inserido no
 // início
 novo->prox = inicio;
 inicio = novo;
}
else if (aux == NULL)
{
 // o novo número a ser
 // inserido
 // é maior que todos os
 // números da
 // lista, logo, será
 // inserido no fim
 fim->prox = novo;
 fim = novo;
 fim->prox=NULL;
}
else
{
 // o novo número a ser
 // inserido
 // será inserido entre
 // dois
 // números que já estão na
 // lista
 anterior->prox =
 ↪ novo;
 novo->prox = aux;
}

}
cout<<"Número inserido na lista!!";
}
if (op == 2)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
```

```
 }
 else
 {
 // a lista contém elementos e
 // estes serão
 // mostrados do inicio ao fim
 cout<<"\nConsultando toda a
 ↵ lista\n";
 aux = inicio;
 while (aux != NULL)
 {
 cout<<aux->num<<" ";
 aux = aux->prox;
 }
 }
}
if (op == 3)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista contém elementos
 // e o elemento a ser
 // removido deve ser digitado
 cout<<"\nDigite o elemento a
 ↵ ser removido:";
 cin>>numero;
 // todas as ocorrências da
 // lista,
 // iguais ao número digitado,
 // serão removidas
 aux = inicio;
 anterior = NULL;
 achou = 0;
 while (aux != NULL)
 {
 if (aux->num == numero)
 {
 // o número digitado
```

```
// foi encontrado na lista
// e será removido
achou = achou + 1;
if (aux == inicio)
{
 // o número a ser
 // removido
 // é o primeiro da
 // lista
 inicio = aux->prox;
 delete(aux);
 aux = inicio;
}
else if (aux == fim)
{
 // o número a ser
 // removido
 // é o último da
 // lista
 anterior->prox =
 NULL;
 fim = anterior;
 delete(aux);
 aux = NULL;
}
else
{
 // o número a ser
 // removido
 // está no meio
 // da lista
 anterior->prox =
 =aux->prox;
 delete(aux);
 aux = anterior-
 >prox;
}
else
{
 anterior = aux;
 aux = aux->prox;
```

```
 }
 }
 if (achou == 0)
 cout<<"Número não encontrado";
 else if (achou == 1)
 cout<<"Número removido 1 vez";
 else
 cout<<"Número removido "<<achou<<
 vezes";
 }
}
if (op == 4)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!";
 }
 else
 {
 // a lista será esvaziada
 aux=inicio;
 while (aux != NULL)
 {
 inicio = inicio->prox;
 delete(aux);
 aux = inicio;
 }
 cout<<"Lista esvaziada";
 }
 getch();
}
while (op != 5);
```

## Listas duplamente encadeadas e não ordenadas

Nesta estrutura, cada elemento armazena um ou vários dados (estrutura homogênea ou heterogênea, respectivamente) e dois ponteiros; o primeiro para o próximo elemento, e o segundo para o elemento anterior. Esses ponteiros permitem o duplo encadeamento e mantêm a estrutura linear. Nesse tipo de estrutura serão abordadas as seguintes operações: inserir no início da lista, inserir no fim, consultar toda a lista do início ao fim ou do fim ao início, remover um elemento qualquer e esvaziá-la.

A seguir, uma ilustração com endereços de memória meramente ilustrativos.

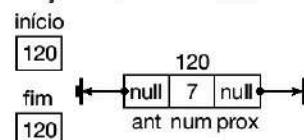


ILUSTRAÇÃO

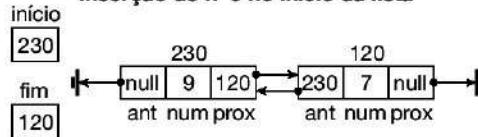
### 1<sup>a</sup> operação A lista está vazia



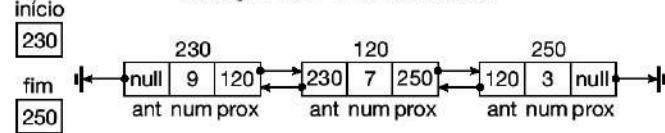
### 2<sup>a</sup> operação Inserção do nº 7 no início da lista



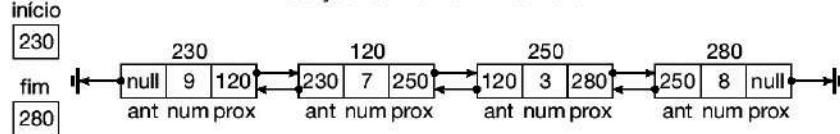
### 3<sup>a</sup> operação Inserção do nº 9 no início da lista

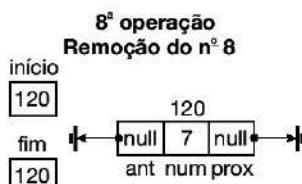
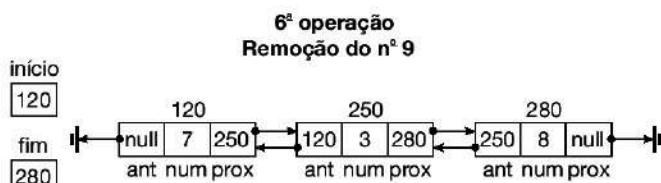


### 4<sup>a</sup> operação Inserção do nº 3 no fim da lista



### 5<sup>a</sup> operação Inserção do nº 8 no fim da lista





```

import java.util.*;
public class LD_Nao_Ordenada
{
 //Definindo a classe que representará
 // cada elemento da lista
 private static class LISTA
 {
 public int num;
 public LISTA prox;
 public LISTA ant;
 }

 public static void main(String args[])
 {
 Scanner entrada = new Scanner(System.in);
 // a lista está vazia, logo,
 // o objeto inicio tem o valor null
 // o objeto inicio conterá o endereço
 // do primeiro elemento da lista
 LISTA inicio = null;
 // o objeto fim conterá o endereço
 // do último elemento da lista
 LISTA fim = null;
 }
}

```

```
// o objeto aux é um objeto auxiliar
LISTA aux;
// apresentando o menu de opções
int op, numero, achou;
do
{
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir no início da
 lista");
 System.out.println("2 - Inserir no fim da
 lista");
 System.out.println("3 - Consultar a lista do
 início ao fim");
 System.out.println("4 - Consultar a lista do fim
 ao início");
 System.out.println("5 - Remover da lista");
 System.out.println("6 - Esvaziar a lista");
 System.out.println("7 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();
 if (op < 1 || op > 7)
 System.out.println("Opção inválida!!!");
 if (op == 1)
 {
 System.out.println("Digite o número a ser
 inserido no início da lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo.prox = null;
 novo.ant = null;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no início da lista
 novo.prox = inicio;
 inicio.ant = novo;
 novo.ant = null;
 inicio = novo;
 }
 }
```

```
System.out.println("Número inserido no
 ↵ inicio da lista!!");
}
if (op == 2)
{
 System.out.println("Digite o número a ser
 ↵ inserido no fim da lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo.prox = null;
 novo.ant = null;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no fim da lista
 fim.prox = novo;
 novo.ant = fim;
 novo.prox = null;
 fim = novo;
 }
 System.out.println("Número inserido no fim da
 ↵ lista!!");
}
if (op == 3)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!");
 }
 else
 {
 // a lista contém elementos e estes
 // serão
 // mostrados do início ao fim
 System.out.println("\nConsultando a
 ↵ lista do início ao fim\n");
 aux = inicio;
 while (aux != null)
 {
```

```
System.out.print(aux.num+" ");
aux = aux.prox;
}
}
}

if (op == 4)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!");
 }
 else
 {
 // a lista contém elementos e estes
 // serão
 // mostrados do fim ao início
 System.out.println("\nConsultando a
 ➔ lista do fim ao
 inicio\n");
 aux = fim;
 while (aux != null)
 {
 System.out.print(aux.num+" ");
 aux = aux.ant;
 }
 }
}
if (op == 5)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!");
 }
 else
 {
 // a lista contém elementos
 // e o elemento a ser
 // removido deve ser digitado
 System.out.print("\nDigite o elemento a ser
 ➔ removido: ");
 numero = entrada.nextInt();
 // todas as ocorrências da lista,
 // iguais ao número digitado,
 // serão removidas
 aux = inicio;
 achou = 0;
 while (aux != null)
```

```
{
 if (aux.num == numero)
 {
 // o número digitado
 // foi encontrado na lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a ser
 // removido
 // é o primeiro da
 // lista
 inicio = aux.prox;
 if (inicio != null)
 {
 inicio.ant = null;
 }
 aux = inicio;
 }
 else if (aux == fim)
 {
 // o número a ser
 // removido
 // é o último da lista
 fim = fim.ant;
 fim.prox = null;
 aux = null;
 }
 else
 {
 // o número a ser
 // removido
 // está no meio da
 // lista
 aux.ant.prox = aux.
 ↲ prox;
 aux.prox.ant = aux.ant;
 aux = aux.prox;
 }
 }
 else
 {
 aux = aux.prox;
 }
}
if (achou == 0)
System.out.println("Número não encontrado");
```

```

 else if (achou == 1)
 System.out.println("Número removido
 → 1 vez");
 else
 System.out.println("Número removido
 → "+achou+" vezes");
 }
}
if (op == 6)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!!");
 }
 else
 {
 // a lista será esvaziada
 inicio = null;
 System.out.println("Lista esvaziada");
 }
}
while (op != 7);
}
}

```

---



```

#include <iostream.h>
#include <conio.h>

void main()
{
 //Definindo o registro representará
 //cada elemento da lista
 struct LISTA
 {
 int num;
 LISTA *prox;
 LISTA *ant;
 };
 // a lista está vazia, logo,

```

```
// o ponteiro inicio tem o valor NULL
// o ponteiro inicio conterá o endereço
// do primeiro elemento da lista
LISTA *inicio = NULL;
// o ponteiro fim conterá o endereço
// do último elemento da lista
LISTA *fim = NULL;
// o ponteiro aux é um ponteiro auxiliar
LISTA *aux;
// apresentando o menu de opções
int op, numero, achou;
do
{
 clrscr();
 cout<<"\nMENU DE OPÇÕES\n";
 cout<<"\n1 - Inserir no início da
 ↵ lista";
 cout<<"\n2 - Inserir no fim da lista";
 cout<<"\n3 - Consultar a lista do
 ↵ início ao fim";
 cout<<"\n4 - Consultar a lista do fim
 ↵ ao início";
 cout<<"\n5 - Remover da lista";
 cout<<"\n6 - Esvaziar a lista";
 cout<<"\n7 - Sair";
 cout<<"\nDigite sua opção: ";
 cin>>op;
 if (op < 1 || op > 7)
 cout<<"Opção inválida!!";
 if (op == 1)
 {
 cout<<"Digite o número a ser inserido
 ↵ no início da lista: ";
 LISTA *novo = new LISTA();
 cin>>novo->num;
 if (inicio == NULL)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo->prox = NULL;
 novo->ant = NULL;
```

```
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no início da lista
 novo->prox = inicio;
 inicio->ant = novo;
 novo->ant = NULL;
 inicio = novo;
 }
 cout<<"Número inserido no início da lista!!";
}
if (op == 2)
{
 cout<<"Digite o número a ser
 → inserido no fim da lista: ";
 LISTA *novo = new LISTA();
 cin>>novo->num;
 if (inicio == NULL)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo->prox = NULL;
 novo->ant = NULL;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no fim da lista
 fim->prox = novo;
 novo->ant = fim;
 novo->prox = NULL;
 fim = novo;
 }
 cout<<"Número inserido no fim da
 → lista!!";
}
if (op == 3)
{
```

```
if (inicio == NULL)
{
 // a lista está vazia
 cout<<"Lista vazia!!";
}
else
{
 // a lista contém elementos e
 // estes serão
 // mostrados do início ao fim
 cout<<"\nConsultando a lista
 ↵ do início ao fim\n";
 aux = inicio;
 while (aux != NULL)
 {
 cout<<aux->num<<" ";
 aux = aux->prox;
 }
}
if (op == 4)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista contém elementos e
 // estes serão
 // mostrados do fim ao início
 cout<<"\nConsultando a lista
 ↵ do fim ao início\n";
 aux = fim;
 while (aux != NULL)
 {
 cout<<aux->num<<" ";
 aux = aux->ant;
 }
 }
}
if (op == 5)
{
```

```
if (inicio == NULL)
{
 // a lista está vazia
 cout<<"Lista vazia!!";
}
else
{
 // a lista contém elementos
 // e o elemento a ser
 // removido deve ser digitado
 cout<<"\nDigite o elemento a
 ↪ ser removido:";
 cin>>numero;
 // todas as ocorrências da
 // lista,
 // iguais ao número digitado,
 // serão removidas
 aux = inicio;
 achou = 0;
 while (aux != NULL)
 {
 if (aux->num == numero)
 {
 // o número digitado
 // foi encontrado na lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a
 // ser removido
 // é o primeiro da
 // lista
 inicio = aux->prox;
 if (inicio != NULL)
 {
 inicio->ant = NULL;
 }
 delete (aux);
 aux = inicio;
 }
 else if (aux == fim)
 {
```

```

 // o número a ser
 // removido
 // é o último da
 // lista
 fim = fim->ant;
 fim->prox = NULL;
 delete(aux);
 aux = NULL;
 }
 else {
 // o número a ser
 // removido
 // está no meio
 // da lista
 aux->ant->prox =
 ↵ aux->prox;
 aux->prox->ant =
 ↵ aux->ant;
 LISTA *aux2;
 aux2 = aux->prox;
 delete(aux);
 aux = aux2;
 }
}
else
{
 aux = aux->prox;
}
}
if (achou == 0)
 cout<<"Número não encontrado";
else if (achou == 1)
 cout<<"Número removido 1 vez";
else
 cout<<"Número removido
 ↵ <<achou<<" vezes";
}
}
if (op == 6)
{
 if (inicio == NULL)
 {
 // a lista está vazia

```

```

 cout<<"Lista vazia!!";
 }
else
{
 // a lista será esvaziada
 aux = inicio;
 while (aux != NULL)
 {
 inicio = inicio->prox;
 delete(aux);
 aux = inicio;
 }
 cout<<"Lista esvaziada";
}
getch();
}
while (op != 7);
}

```

---

## Lista duplamente encadeada e ordenada

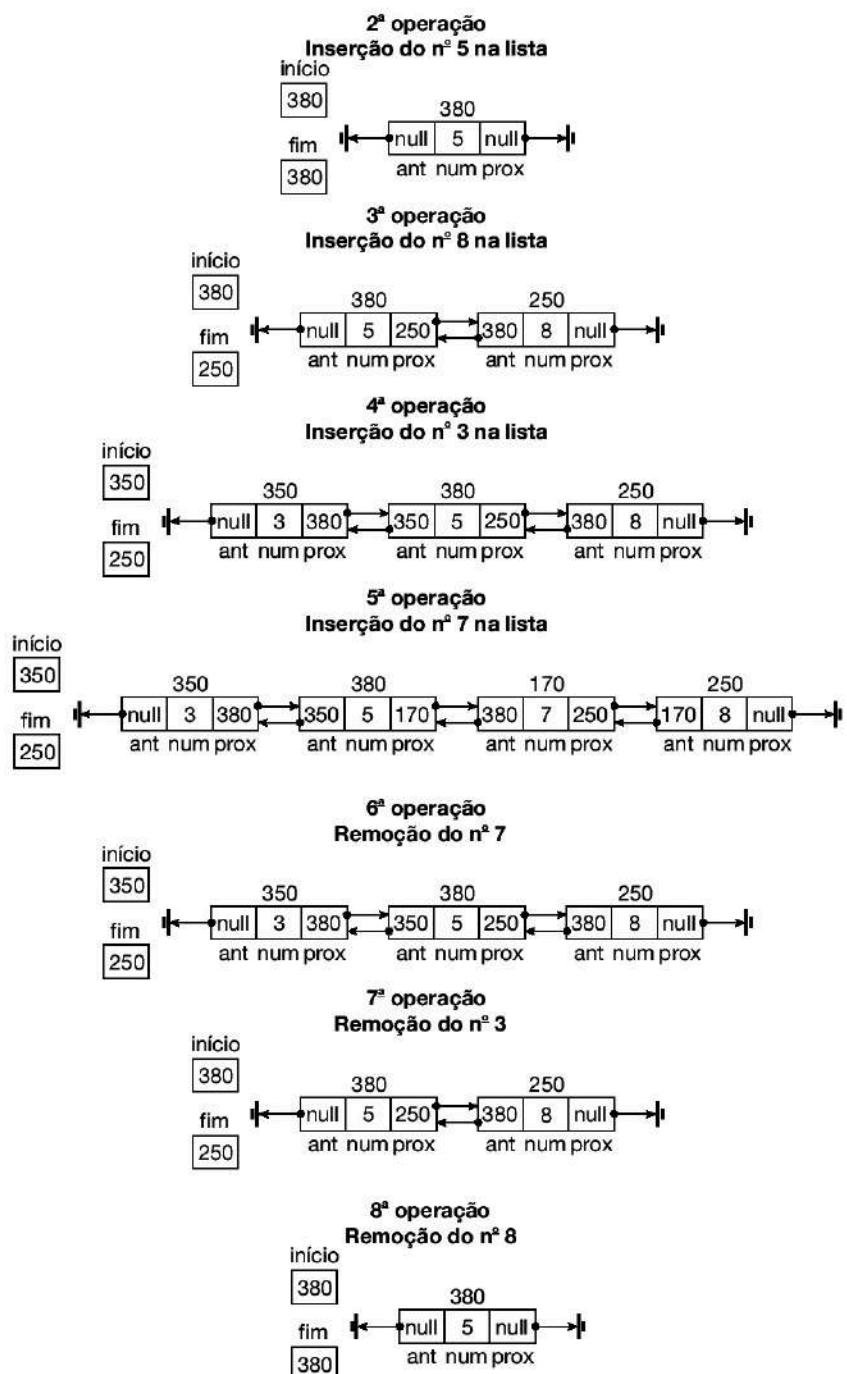
Nesta estrutura, cada elemento armazena um ou vários dados (estrutura homogênea ou heterogênea, respectivamente) e dois ponteiros; o primeiro para o próximo elemento, e o segundo para o elemento anterior, permitindo o duplo encadeamento e mantendo a estrutura linear. Tem-se também um campo denominado chave através do qual uma determinada ordenação é mantida. Nesse tipo de estrutura as operações possíveis são: inserir na lista, consultar toda a lista do início ao fim ou do fim ao início, remover um elemento qualquer e esvaziá-la.

A seguir, uma ilustração com endereços de memória meramente ilustrativos, cujos números da lista estão ordenados de forma crescente.



ILUSTRAÇÃO

**1<sup>a</sup> operação**  
A lista está vazia  
início  
[null]





```
import java.util.*;
public class LD_Ordenada
{
 //Definindo a classe que representará
 //cada elemento da lista
 private static class LISTA
 {
 public int num;
 public LISTA prox;
 public LISTA ant;
 }

 public static void main(String args[])
 {
 Scanner entrada = new Scanner(System.in);
 // a lista está vazia, logo,
 // o objeto inicio tem o valor null
 // o objeto inicio conterá o endereço
 // do primeiro elemento da lista
 LISTA inicio = null;
 // o objeto fim conterá o endereço
 // do último elemento da lista
 LISTA fim = null;
 // o objeto aux é um objeto auxiliar
 LISTA aux;
 // apresentando o menu de opções
 int op, numero, achou;
 do
 {
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir na lista");
 System.out.println("2 - Consultar a lista do
 → início ao fim");
 System.out.println("3 - Consultar a lista do fim
 → ao início");
 System.out.println("4 - Remover da lista");
 System.out.println("5 - Esvaziar a lista");
 System.out.println("6 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();
 if (op < 1 || op > 6)
 System.out.println("Opção inválida!!!");
 if (op == 1)
```

```
{
 System.out.println("Digite o número a ser
 inserido na lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo.prox = null;
 novo.ant = null;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido na lista
 // respeitando a ordenação crescente
 aux = inicio;
 while (aux != null &&
 novo.num > aux.num)
 {
 aux = aux.prox;
 }
 if (aux == inicio)
 {
 // o novo número a ser inserido
 // é menor que todos os números
 // da lista,
 // logo, será inserido no
 // início
 novo.prox = inicio;
 novo.ant = null;
 inicio.ant = novo;
 inicio = novo;
 }
 else if (aux == null)
 {
 // o novo número a ser
 // inserido
 // é maior que todos os
 // números
```

```
// da lista, logo,
// será inserido no fim
fim.prox = novo;
novo.ant = fim;
fim = novo;
fim.prox = null;
}
else
{
 // o novo número a ser
 // inserido
 // será inserido entre dois
 // números que já estão na
 // lista
 novo.prox = aux;
 aux.ant.prox = novo;
 novo.ant = aux.ant;
 aux.ant = novo;
}
}
System.out.println("Número inserido na
↳ lista!!");
}
if (op == 2)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!");
 }
 else
 {
 // a lista contém elementos e estes serão
 // mostrados do início ao fim
 System.out.println("\nConsultando a lista
↳ do início ao fim\n");
 aux = inicio;
 while (aux != null)
 {
 System.out.print(aux.num+" ");
 aux = aux.prox;
 }
 }
}
if (op == 3)
{
 if (inicio == null)
```

```
{
 // a lista está vazia
 System.out.println("Lista vazia!!");
}
else
{
 // a lista contém elementos e estes serão
 // mostrados do fim ao início
 System.out.println("\nConsultando a lista
 // do fim ao início\n");
 aux = fim;
 while (aux != null)
 {
 System.out.print(aux.num+" ");
 aux = aux.ant;
 }
}
}
if (op == 4)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!");
 }
 else
 {
 // a lista contém elementos
 // e o elemento a ser
 // removido deve ser digitado
 System.out.print("\nDigite o
 ➔ elemento a ser removido: ");
 numero = entrada.nextInt();
 // todas as ocorrências da lista,
 // iguais ao número digitado,
 // serão removidas
 aux = inicio;
 achou = 0;
 while (aux != null)
 {
 if (aux.num == numero)
 {
 // o número digitado
 // foi encontrado na lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
```

```
{
 // o número a ser
 // removido
 // é o primeiro da
 // lista
 inicio = aux.prox;
 if (inicio != null)
 {
 inicio.ant = null;
 }
 aux = inicio;
}
else if (aux == fim)
{
 // o número a ser
 // removido
 // é o último da lista
 fim = fim.ant;
 fim.prox = null;
 aux = null;
}
else
{
 // o número a ser
 // removido
 // está no meio da
 // lista
 aux.ant.prox =
 ↪ aux. prox;
 aux.prox.ant = aux.
 ↪ ant;
 aux = aux.prox;
}
}
else
{
 aux = aux.
 ↪ prox;
}
}
}
if (achou == 0)
System.out.
 ↪ println("Número não ↪ encontrado");
else if (achou == 1)
 System.out.
 ↪ println("Número ↪ removido 1
 ↪ vez");
}
```

```

 else
 System.out.println("Número
 ↵ removido "+achou+" vezes");
 }
}
if (op == 5)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!!");
 }
 else
 {
 // a lista será esvaziada
 inicio = null;
 System.out.println("Lista esvaziada");
 }
}
while (op != 6);
}
}

```



c/c++

---

```

#include <iostream.h>
#include <conio.h>

void main()
{
 //Definindo o registro que representará
 //cada elemento da lista
 struct LISTA
 {
 int num;
 LISTA *prox;
 LISTA *ant;
 };
 // a lista está vazia, logo,

```

```
// o ponteiro inicio tem o valor null
// o ponteiro inicio conterá o endereço
// do primeiro elemento da lista
LISTA *inicio = NULL;
// o ponteiro fim conterá o endereço
// do último elemento da lista
LISTA *fim = NULL;
// o ponteiro aux é um ponteiro auxiliar
LISTA *aux;
// apresentando o menu de opções
int op, numero, achou;
do
{
 clrscr();
 cout<<"\nMENU DE OPÇÕES\n";
 cout<<"\n1 - Inserir na lista";
 cout<<"\n2 - Consultar a lista do início ao fim";
 cout<<"\n3 - Consultar a lista do fim ao início";
 cout<<"\n4 - Remover da lista";
 cout<<"\n5 - Esvaziar a lista";
 cout<<"\n6 - Sair";
 cout<<"\nDigite sua opção: ";
 cin>>op;
 if (op < 1 || op > 6)
 cout<<"Opção inválida!!";
 if (op == 1)
 {
 cout<<"Digite o número a ser inserido na
 ↵ lista: ";
 LISTA *novo = new LISTA();
 cin>>novo->num;
 if (inicio == NULL)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 novo->prox = NULL;
 novo->ant = NULL;
 inicio = novo;
 fim = novo;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido na lista
 // respeitando a ordenação crescente
```

```
aux = inicio;
while (aux != NULL && novo->num > aux->num)
{
 aux = aux->prox;
}
if (aux == inicio)
{
 // o novo número a ser inserido
 // é menor que todos os números da lista,
 // logo, será inserido no início
 novo->prox = inicio;
 novo->ant = NULL;
 inicio->ant = novo;
 inicio = novo;
}
else if (aux == NULL)
{
 // o novo número a ser inserido
 // é maior que todos os números
 // da lista, logo,
 // será inserido no fim
 fim->prox = novo;
 novo->ant = fim;
 fim = novo;
 fim->prox = NULL;
}
else
{
 // o novo número a ser inserido
 // será inserido entre dois
 // números que já estão na lista
 novo->prox = aux;
 aux->ant->prox = novo;
 novo->ant = aux->ant;
 aux->ant = novo;
}
cout<<"Número inserido na lista!!";
}
if (op == 2)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
```

```
{
 // a lista contém elementos e estes serão
 // mostrados do início ao fim
 cout<<"\nConsultando a lista do início ao
 ↵ fim\n";
 aux = inicio;
 while (aux != NULL)
 {
 cout<<aux->num<<" ";
 aux = aux->prox;
 }
}
}
if (op == 3)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista contém elementos e estes serão
 // mostrados do fim ao início
 cout<<"\nConsultando a lista do fim ao
 ↵ início\n";
 aux = fim;
 while (aux != NULL)
 {
 cout<<aux->num<<" ";
 aux = aux->ant;
 }
 }
}
if (op == 4)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista contém elementos
 // e o elemento a ser
 // removido deve ser digitado
 cout<<"\nDigite o elemento a ser
 ↵ removido: ";
 }
}
```

```
cin>>numero;
// todas as ocorrências da lista,
// iguais ao número digitado,
// serão removidas
aux = inicio;
achou = 0;
while (aux != NULL)
{
 if (aux->num == numero)
 {
 // o número digitado
 // foi encontrado na lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a ser
 // removido
 // é o primeiro da
 // lista
 inicio = aux->prox;
 if (inicio != NULL)
 {
 inicio->ant = NULL;
 }
 delete(aux);
 aux = inicio;
 }
 else if (aux == fim)
 {
 // o número a ser
 // removido
 // é o último da lista
 fim = fim->ant;
 fim->prox = NULL;
 delete(aux);
 aux = NULL;
 }
 else
 {
 // o número a ser
 // removido
 // está no meio da
 // lista
 aux->ant->prox = aux-
 ↵ >prox;
 aux->prox->ant = aux-
 ↵ >ant;
```

```
LISTA *aux2;
aux2 = aux->prox;
delete(aux);
aux=aux2;
}
}
else
{
 aux = aux->prox;
}
}

if (achou == 0)
cout<<"Número não encontrado";
else if (achou == 1)
 cout<<"Número removido 1 vez";
else
 cout<<"Número removido "<<achou<<
 vezes";
}
}

if (op == 5)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista será esvaziada
 aux = inicio;
 while (aux != NULL)
 {
 inicio = inicio->prox;
 delete(aux);
 aux = inicio;
 }
 cout<<"Lista esvaziada";
 }
}
getch();
}
while (op != 6);
}
```

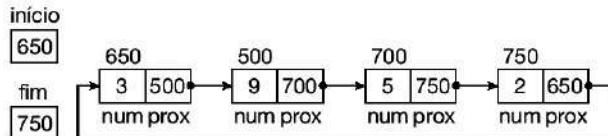
## Listas circulares

As listas citadas nas seções anteriores podem também ser implementadas de forma circular. Assim, quando simplesmente encadeadas, o último elemento delas terá o ponteiro próximo apontando para o primeiro, enquanto as listas duplamente encadeadas terão o último com o ponteiro próximo apontando para o primeiro elemento, e o primeiro com o ponteiro anterior apontando para o último.

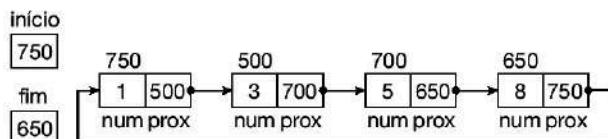


ILUSTRAÇÃO

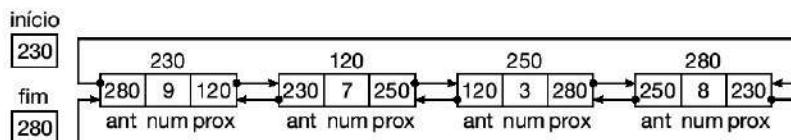
### Lista circular simplesmente encadeada e não ordenada



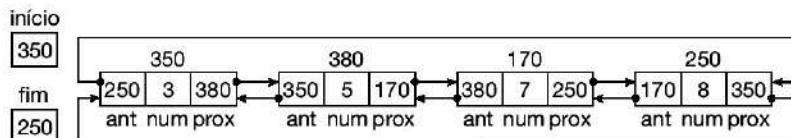
### Lista circular simplesmente encadeada e ordenada



### Lista circular duplamente encadeada e não ordenada



### Lista circular duplamente encadeada e ordenada



A seguir, serão mostradas implementações de lista circular simplesmente encadeada e não ordenada e de lista circular duplamente encadeada e não ordenada, respectivamente.



```
import java.util.*;
public class L_Circular_S_Nao_Ordenada
{
 //Definindo a classe que representará
 // cada elemento da lista
 private static class LISTA
 {
 public int num;
 public LISTA prox;
 }

 public static void main(String args[])
 {
 Scanner entrada = new Scanner(System.in);
 // a lista está vazia, logo,
 // o objeto inicio tem o valor null
 // o objeto inicio conterá o endereço
 // do primeiro elemento da lista
 LISTA inicio = null;
 // o objeto fim conterá o endereço
 // do último elemento da lista
 LISTA fim = null;
 // o objeto aux é um objeto auxiliar
 LISTA aux;
 // o objeto anterior é um objeto auxiliar
 LISTA anterior;
 // apresentando o menu de opções
 int op, numero, achou;
 do
 {
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir no
 ↵ início da lista");
 System.out.println("2 - Inserir no fim
 ↵ da lista");
 System.out.println("3 - Consultar toda
 ↵ a lista");
 System.out.println("4 - Remover da lista");
 System.out.println("5 - Esvaziar a lista");
 System.out.println("6 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();
 if (op < 1 || op > 6)
```

```
System.out.println("Opção inválida!!");
if (op == 1)
{
 System.out.println("Digite o número a
 → ser inserido no início da lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 fim.prox = inicio;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no início da lista
 novo.prox = inicio;
 inicio = novo;
 fim.prox = inicio;
 }
 System.out.println("Número inserido no
 → início da lista!!");
}
if (op == 2)
{
 System.out.println("Digite o número a
 → ser inserido no fim da lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 fim.prox = inicio;
 }
 else
```

```
{
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no fim da lista
 fim.prox = novo;
 fim = novo;
 fim.prox = inicio;
}
System.out.println("Número inserido no fim da
↳ lista!!");
}
if (op == 3)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!");
 }
 else
 {
 // a lista contém elementos e estes serão
 // mostrados do início ao fim
 System.out.println("\nConsultando toda a
↳ lista\n");
 aux = inicio;
 do
 {
 System.out.print(aux.num+" ");
 aux = aux.prox;
 }
 while (aux != inicio);
 }
}
if (op == 4)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!");
 }
 else
 {
 // a lista contém elementos
 // e o elemento a ser
```

```
// removido deve ser digitado
System.out.print("\nDigite o
➥ elemento a ser removido: ");
numero = entrada.nextInt();
// todas as ocorrências da lista,
// iguais ao número digitado,
// serão removidas
aux = inicio;
anterior = null;
achou = 0;
// descobrindo a quantidade de
// elementos da lista
int quantidade = 0;
aux = inicio;
do
{
 quantidade = quantidade + 1;
 aux = aux.prox;
}
while (aux != inicio);
int elemento = 1;
do
{
 // se a lista possui
 // apenas um elemento
 if (inicio == fim
 && inicio.num == numero)
 {
 inicio = null;
 achou = achou + 1;
 }
 else
 {
 if (aux.num == numero)
 {
 // o número digitado
 // foi encontrado na
 // lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a ser
 // removido
```

```
// é o primeiro
// da lista
inicio = aux;
→ prox;
fim.prox = inicio;
aux = inicio;
}
else if (aux == fim)
{
 // o número a ser
 // removido
 // é o último
 // da lista
fim =
anterior;
fim.prox = inicio;
aux = aux. prox;
}
else
{
 // o número a ser
 // removido
 // está no meio
 // da lista
anterior.prox =
aux. prox;
aux = aux. prox;
}
}
else
{
 anterior = aux;
 aux = aux. prox;
}
}
}
elemento = elemento + 1;
}
while (elemento <= quantidade);

if (achou == 0)
 System.out.println("Número não
 → encontrado");
else if (achou == 1)
 System.out.println("Número removido
 → 1 vez");
```

```

 else
 System.out.println("Número
 ↵ removido " +achou+ " vezes");
 }
}
if (op == 5)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!!");
 }
 else
 {
 // a lista será esvaziada
 inicio = null;
 System.out.println("Lista esvaziada");
 }
}
while (op != 6);
}
}

```

---



```

#include<iostream.h>
#include<conio.h>

void main()
{
 //Definindo o registro que representará
 //cada elemento da lista
 struct LISTA
 {
 int num;
 LISTA *prox;
 };
 // a lista está vazia, logo,
 // o ponteiro inicio têm o valor null
 // o ponteiro inicio conterá o endereço
 // do primeiro elemento da lista

```

```
LISTA *inicio = NULL;
// o ponteiro fim conterá o endereço
// do último elemento da lista
LISTA *fim = NULL;
// o ponteiro aux é um ponteiro auxiliar
LISTA *aux;
// o ponteiro anterior é um ponteiro auxiliar
LISTA *anterior;
// apresentando o menu de opções
int op, numero, achou;
do
{
 clrscr();
 cout<<"\nMENU DE OPÇÕES\n";
 cout<<"\n1 - Inserir no início da lista";
 cout<<"\n2 - Inserir no fim da lista";
 cout<<"\n3 - Consultar toda a lista";
 cout<<"\n4 - Remover da lista";
 cout<<"\n5 - Esvaziar a lista";
 cout<<"\n6 - Sair";
 cout<<"\nDigite sua opção: ";
 cin>>op;
 if (op < 1 || op > 6)
 cout<<"Opção inválida!!";
 if (op == 1)
 {
 cout<<"Digite o número a ser inserido no
 ↵ início da lista: ";
 LISTA *novo = new LISTA();
 cin>>novo->num;
 if (inicio == NULL)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 fim->prox = inicio;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no início da lista
```

```
 novo->prox = inicio;
 inicio = novo;
 fim->prox = inicio;
}
cout<<"Número inserido no início da
→ lista!!";
}
if (op == 2)
{
 cout<<"Digite o número a ser inserido
→ no fim da lista: ";
 LISTA *novo = new LISTA();
 cin>>novo->num;
 if (inicio == NULL)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 fim->prox = inicio;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no fim da lista
 fim->prox = novo;
 fim = novo;
 fim->prox = inicio;
 }
 cout<<"Número inserido no fim da
→ lista!!";
}
if (op == 3)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista contém elementos
```

```
// e estes serão mostrados
// do início ao fim
cout<<"\nConsultando toda a
➥ lista\n";
aux = inicio;
do
{
 cout<<aux->num<<" ";
 aux = aux->prox;
}
while (aux != inicio);
}

if (op == 4)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista contém elementos
 // e o elemento a ser
 // removido deve ser digitado
 cout<<"\nDigite o elemento a ser
➥ removido:";
 cin>>numero;
 // todas as ocorrências da lista,
 // iguais ao número digitado,
 // serão removidas
 aux = inicio;
 anterior = NULL;
 achou = 0;
 // descobrindo a quantidade de
 // elementos da lista
 int quantidade = 0;
 aux = inicio;
 do
 {
 quantidade = quantidade + 1;
 aux = aux->prox;
 }
 while (aux != inicio);
```

```
int elemento = 1;
do
{
 // se a lista possui apenas
 // um elemento
 if (inicio == fim &&
 inicio->num == numero)
 {
 delete(inicio);
 inicio = NULL;
 achou = achou + 1;
 }
 else
 {
 if (aux->num == numero)
 {
 // o número digitado
 // foi encontrado na lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a ser
 // removido
 // é o primeiro da lista
 inicio = aux->prox;
 fim->prox = inicio;
 delete(aux);
 aux = inicio;
 }
 else if (aux == fim)
 {
 // o número a ser
 // removido
 // é o último da
 // lista
 fim = anterior;
 fim->prox = inicio;
 delete(aux);
 aux = NULL;
 }
 else
 {
 // o número a ser
```

```
// removido
// está no meio da
// lista
anterior->prox =
 aux->prox;
delete(aux);
aux = anterior-
 >prox;
}
}
else
{
 anterior = aux;
 aux = aux->prox;
}
}
elemento = elemento + 1;
}
while (elemento <= quantidade);
if (achou == 0)
 cout<<"Número não encontrado";
else if (achou == 1)
 cout<<"Número removido 1 vez";
else
 cout<<"Número removido "<<achou<<
 " vezes";
}
}
if (op == 5)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista será esvaziada
 aux = inicio;
 do
 {
 inicio = inicio ->prox;
 delete(aux);
 }
 while (inicio != NULL);
 }
}
```

```

 aux = inicio;
 }
 while (aux != fim);
 delete(fim);
 inicio=NULL;
 cout<<"Lista esvaziada";
}
getch();
}
while (op != 6);
}

```



```

import java.util.*;
public class L_Circular_D_Nao_Ordenada
{
 //Definindo a classe que representará
 //cada elemento da lista
 private static class LISTA
 {
 public int num;
 public LISTA prox;
 public LISTA ant;
 }

 public static void main(String args[])
 {
 Scanner entrada = new Scanner(System.in);
 // a lista está vazia, logo,
 // o objeto inicio tem o valor null
 // o objeto inicio conterá o endereço
 // do primeiro elemento da lista
 LISTA inicio = null;
 // o objeto fim conterá o endereço
 // do último elemento da lista
 LISTA fim = null;
 // o objeto aux é um objeto auxiliar
 }
}

```

```
LISTA aux;
// apresentando o menu de opções
int op, numero, achou;
do
{
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir no início da
 lista");
 System.out.println("2 - Inserir no fim da
 lista");
 System.out.println("3 - Consultar a lista do
 início ao fim");
 System.out.println("4 - Consultar a lista do
 fim ao início");
 System.out.println("5 - Remover da lista");
 System.out.println("6 - Esvaziar a lista");
 System.out.println("7 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();
 if (op < 1 || op > 7)
 System.out.println("Opção inválida!!!");
 if (op == 1)
 {
 System.out.println("Digite o número a
 ser inserido no início da lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo.prox = inicio;
 novo.ant = inicio;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no início da lista
 novo.prox = inicio;
```

```
 inicio.ant = novo;
 novo.ant = fim;
 fim.prox = novo;
 inicio = novo;
 }
 System.out.println("Número inserido no
 → início da lista!!");
}
if (op == 2)
{
 System.out.println("Digite o número a
 → ser inserido no fim da lista: ");
 LISTA novo = new LISTA();
 novo.num = entrada.nextInt();
 if (inicio == null)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo.prox = inicio;
 novo.ant = inicio;
 }
 else
 {
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no fim da lista
 fim.prox = novo;
 novo.ant = fim;
 fim = novo;
 fim.prox = inicio;
 inicio.ant = fim;
 }
 System.out.println("Número inserido no
 → fim da lista!!");
}
if (op == 3)
{
 if (inicio == null)
 {
 // a lista está vazia
```

```
 System.out.println("Lista vazia!!");
 }
else
{
 // a lista contém elementos e estes
 // serão mostrados do início ao fim
 System.out.println("\nConsultando a
 ↵ lista do início ao fim\n");
 aux = inicio;
 do
 {
 System.out.print(aux.num+" ");
 aux = aux.prox;
 }
 while (aux != inicio);
}
}
if (op == 4)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!");
 }
 else
 {
 // a lista contém elementos e estes
 // serão mostrados do fim ao início
 System.out.println("\nConsultando a
 ↵ lista do fim ao início\n");
 aux = fim;
 do
 {
 System.out.print(aux.num+" ");
 aux = aux.ant;
 }
 while (aux != fim);
 }
}
if (op == 5)
{
 if (inicio == null)
```

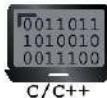
```
{
 // a lista está vazia
 System.out.println("Lista
 ↵ vazia!!");
}
else
{
 // a lista contém elementos
 // e o elemento a ser
 // removido deve ser digitado
 System.out.print("\nDigite o
 ↵ elemento a ser removido: ");
 numero = entrada.nextInt();
 // todas as ocorrências da
 // lista,
 // iguais ao número digitado,
 // serão removidas
 aux = inicio;
 achou = 0;
 // descobrindo a quantidade de
 // elementos da lista
 int quantidade = 0;
 aux = inicio;
 do
 {
 quantidade = quantidade + 1;
 aux = aux.prox;
 }
 while (aux != inicio);
 int elemento = 1;
 do
 {
 // se a lista possui
 // apenas um elemento
 if (inicio == fim
 && inicio.num == numero)
 {
 inicio = null;
 achou = achou + 1;
 }
 else
 {
 if (aux.num == numero)
```

```
{
 // o número digitado
 // foi encontrado
 // na lista e será
 // removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a ser
 // removido
 // é o primeiro
 // da lista
 inicio = aux. prox;
 inicio.ant = fim;
 fim.prox = inicio;
 aux = inicio;
 }
 else if (aux == fim)
 {
 // o número a
 // ser
 // removido
 // é o último
 // da
 // lista
 fim = fim. ant;
 fim.prox =
 inicio;
 inicio.ant =
 // fim;
 aux = null;
 }
 else
 {
 // o número a ser
 // removido
 // está no meio
 // da lista aux.
 // ant.prox =
 aux.prox;
 aux.prox. ant =
 aux.ant;
 aux = aux.prox;
 }
}
else
{
 aux = aux.prox;
```

```

 }
 }
 elemento = elemento + 1;
}
while (elemento <= quantidade);
if (achou == 0)
 System.out.println("Número não
 ↪ encontrado");
else if (achou == 1)
 System.out.println("Número
 ↪ removido 1 vez");
else
 System.out.println("Número
 ↪ removido " + achou + " vezes");
}
}
if (op == 6)
{
 if (inicio == null)
 {
 // a lista está vazia
 System.out.println("Lista vazia!!!");
 }
 else
 {
 // a lista será esvaziada
 inicio = null;
 System.out.println("Lista esvaziada");
 }
}
while (op != 7);
}
}

```




---

```

#include <iostream.h>
#include <conio.h>

void main()
{
 //Definindo o registro

```

```
// que representará cada elemento da lista
struct LISTA
{
 int num;
 LISTA *prox;
 LISTA *ant;
};

// a lista está vazia, logo,
// o ponteiro inicio têm o valor NULL
// o ponteiro inicio conterá o endereço
// do primeiro elemento da lista
LISTA *inicio = NULL;
// o ponteiro fim conterá o endereço
// do último elemento da lista
LISTA *fim = NULL;
// o ponteiro aux é um ponteiro auxiliar
LISTA *aux;
// apresentando o menu de opções
int op, numero, achou;
do
{
 clrscr();
 cout<<"\nMENU DE OPÇÕES\n";
 cout<<"\n1 - Inserir no início da lista";
 cout<<"\n2 - Inserir no fim da lista";
 cout<<"\n3 - Consultar a lista do início
 ao fim";
 cout<<"\n4 - Consultar a lista do fim ao
 início";
 cout<<"\n5 - Remover da lista";
 cout<<"\n6 - Esvaziar a lista";
 cout<<"\n7 - Sair";
 cout<<"\nDigite sua opção: ";
 cin>>op;
 if (op < 1 || op > 7)
 cout<<"Opção inválida!";
 if (op == 1)
 {
 cout<<"Digite o número a ser inserido no
 início da lista: ";
 LISTA *novo = new LISTA();
 cin>>novo->num;
 if (inicio == NULL)
 {
```

```
// a lista estava vazia
// e o elemento inserido será
// o primeiro e o último
inicio = novo;
fim = novo;
novo->prox = inicio;
novo->ant = inicio;
}
else
{
 // a lista já contém elementos
 // e o novo elemento
 // será inserido no início da lista
 novo->prox = inicio;
 inicio->ant = novo;
 novo->ant = fim;
 fim->prox = novo;
 inicio = novo;
}
cout<<"Número inserido no início da
➥ lista!!";
}
if (op == 2)
{
 cout<<"Digite o número a ser inserido
➥ no fim da lista: ";
 LISTA *novo = new LISTA();
 cin>>novo->num;
 if (inicio == NULL)
 {
 // a lista estava vazia
 // e o elemento inserido será
 // o primeiro e o último
 inicio = novo;
 fim = novo;
 novo->prox = inicio;
 novo->ant = inicio;
 }
 else
 {
 // a lista já contém elementos e o
 // novo elemento
 // será inserido no fim da lista
 fim->prox = novo;
```

```
 novo->ant = fim;
 fim = novo;
 fim->prox = inicio;
 inicio->ant = fim;
}
cout<<"Número inserido no fim da
→ lista!!";
}
if (op == 3)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista contém elementos e estes
 // serão
 // mostrados do início ao fim
 cout<<"\nConsultando a lista do
→ início ao fim\n";
 aux = inicio;
 do
 {
 cout<<aux->num<<" ";
 aux = aux->prox;
 }
 while (aux != inicio);
 }
}
if (op == 4)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!!";
 }
 else
 {
 // a lista contém elementos
 // e estes serão
 // mostrados do fim ao início
 cout<<"\nConsultando a lista do fim ao
→ início\n";
 aux = fim;
```

```
 do
 {
 cout<<aux->num<<" ";
 aux = aux->ant;
 }
 while (aux != fim);
}

if (op == 5)
{
 if (inicio == NULL)
 {
 // a lista está vazia
 cout<<"Lista vazia!";
 }
 else
 {
 // a lista contém elementos
 // e o elemento a ser
 // removido deve ser digitado
 cout<<"\nDigite o elemento a ser
 removido:";
 cin>>numero;
 // todas as ocorrências da lista,
 // iguais ao número digitado,
 // serão removidas
 aux = inicio;
 achou = 0;
 // descobrindo a quantidade de
 // elementos da lista
 int quantidade = 0;
 aux = inicio;
 do
 {
 quantidade = quantidade + 1;
 aux = aux->prox;
 }
 while (aux != inicio);
 int elemento = 1;
 do
 {
 // se a lista possui apenas um elemento
 if (inicio == fim &&
 inicio->num == numero)
 {
 inicio = NULL;
```

```
 delete(inicio);
 achou = achou + 1;
 }
else
{
 if (aux->num == numero)
 {
 // o número digitado
 // foi encontrado na lista
 // e será removido
 achou = achou + 1;
 if (aux == inicio)
 {
 // o número a ser removido
 // é o primeiro da lista
 inicio = aux->prox;
 inicio->ant = fim;
 fim->prox = inicio;
 delete(aux);
 aux = inicio;
 }
 else if (aux == fim)
 {
 // o número a ser
 // removido
 // é o último da lista
 fim = fim->ant;
 fim->prox = inicio;
 inicio->ant = fim;
 delete(aux);
 aux = NULL;
 }
 else
 {
 // o número a ser
 // removido
 // está no meio da lista
 aux->ant->prox = aux-
 >prox;
 aux->prox->ant = aux->ant;
 LISTA *aux2;
 aux2 = aux;
 aux = aux->prox;
 delete(aux2);
 }
 }
}
```

```
 else
 {
 aux = aux->prox;
 }
 }
 elemento = elemento + 1;
}
while (elemento <= quantidade);
if (achou == 0)
 cout<<"Número não encontrado";
else if (achou == 1)
 cout<<"Número removido 1 vez";
else
 cout<<"Número removido "<<achou<<
 ↵ vezes";
}
}
if (op == 6)
{
if (inicio == NULL)
{
 // a lista está vazia
 cout<<"Lista vazia!";
}
else
{
 // a lista será esvaziada
 aux = inicio;
 do
 {
 inicio = inicio->prox;
 delete(aux);
 aux = inicio;
 }
 while (aux != fim);
 delete(fim);
 inicio=NULL;
 cout<<"Lista esvaziada";
}
}
getch();
}
while (op != 7);
```

## Análise da complexidade

Como visto nas seções anteriores, as principais operações realizadas em listas são: inserção no início da lista, inserção no fim, inserção ordenada, consulta e remoção. A seguir, serão mostradas as análises para essas operações.

### Inserção no início da lista

Para fazer a operação de inserção no início de uma lista encadeada, é necessário apenas realizar operações de atribuições atualizando o ponteiro do início (e fim da lista, quando necessário). Com isso, o tempo gasto na operação é constante,  $O(1)$ .

Ela é realizada nas seguintes listas: lista simplesmente encadeada e não ordenada, lista duplamente encadeada e não ordenada, lista circular simplesmente encadeada e não ordenada e lista circular duplamente encadeada e não ordenada.

### Inserção no fim da lista

Para fazer a operação de inserção no fim de uma lista encadeada, é necessário apenas realizar operações de atribuições atualizando o ponteiro do fim (e início da lista, quando necessário). Com isso, o tempo gasto na operação é constante,  $O(1)$ .

Ela é realizada nas seguintes listas: lista simplesmente encadeada e não ordenada, lista duplamente encadeada e não ordenada, lista circular simplesmente encadeada e não ordenada e lista circular duplamente encadeada e não ordenada.

### Inserção ordenada

Esta operação é realizada quando a lista armazena seus dados ordenadamente, como é o caso da lista simplesmente encadeada e ordenada, da lista duplamente encadeada e ordenada, da lista circular simplesmente encadeada e ordenada e da lista duplamente encadeada e ordenada.

Nela, o elemento a ser inserido pode ser, no pior caso, o maior de todos os já existentes na lista. Com isso, percorre-se toda a lista, comparando o novo elemento com os demais, para encontrar a posição correta e inseri-lo. Logo, considerando que uma lista possui  $n$  elementos, o tempo necessário para inserir um novo é  $O(n)$ .

### Consultar toda a lista

Em todos os tipos de listas abordadas aqui, para realizar a operação de consultar toda a lista é necessário acessar cada elemento. Com isso, considerando que uma lista possui  $n$  elementos, o tempo necessário para tal operação é  $O(n)$ .

No caso da lista duplamente encadeada, ordenada ou não, é possível percorrer a lista do início ao fim ou vice-versa, mas o tempo necessário ainda continua sendo proporcional ao tamanho da lista.

## Remoção

A remoção de um elemento em uma lista, de qualquer tipo já mencionado, consiste em procurá-lo e depois acertar os ponteiros para que a lista continue sendo acessada após a remoção. Na busca pelo elemento a ser removido, percorre-se, no pior caso, todos os elementos da lista, gastando com isso tempo proporcional ao tamanho dela, ou seja,  $O(n)$ .

## Esvaziar a lista

A operação de esvaziamento da lista consiste em remover todos os elementos dela. O tempo gasto nessa operação depende da linguagem de programação que está sendo utilizada. No caso da linguagem JAVA, não é necessário realizar a remoção de cada um dos elementos, apenas atualiza-se o ponteiro para o início da lista em nulo e as memórias alocadas serão desalocadas por um procedimento JAVA. Já na linguagem C/C++, é necessário desalocar cada um dos elementos da lista, gastando tempo proporcional ao tamanho dela, ou seja,  $O(n)$ .

## Exercícios

1. Faça um programa que cadastre 5 produtos. Para cada produto devem ser cadastrados código do produto, preço e quantidade estocada. Os dados devem ser armazenados em uma lista simplesmente encadeada e não ordenada. Posteriormente, receber do usuário a taxa de desconto (ex.: digitar 10 para taxa de desconto de 10%). Aplicar a taxa digitada ao preço de todos os produtos cadastrados e finalmente mostrar um relatório com o código e o novo preço. O final desse relatório deve apresentar também a quantidade de produtos com quantidade estocada superior a 500.
2. Faça um programa que cadastre 8 funcionários. Para cada funcionário devem ser cadastrados nome e salário. Os dados devem ser armazenados em uma lista simplesmente encadeada e ordenada, de forma decrescente, pelo salário do funcionário. Posteriormente, o programa deve mostrar:
  - a) o nome do funcionário que tem o maior salário (em caso de empate mostrar todos);
  - b) a média salarial de todos os funcionários juntos;
  - c) a quantidade de funcionários com salário superior a um valor fornecido pelo usuário.
 Caso nenhum funcionário satisfaça essa condição, mostrar mensagem.
3. Faça um programa que cadastre 5 alunos. Para cada aluno devem ser cadastrados nome e nota final. Os dados devem ser armazenados em uma lista duplamente encadeada e não ordenada. Em seguida, o programa deve mostrar apenas o nome dos alunos aprovados, ou seja, alunos com nota final de no mínimo 7. Se nenhum aluno estiver aprovado, mostrar mensagem.
4. Faça um programa que cadastre o nome e o salário de 6 funcionários em uma lista duplamente encadeada e ordenada pelo salário de forma crescente. A seguir, o programa deve mostrar o nome, o valor do imposto e o valor a receber, ou seja, o salário menos o imposto de todos os funcionários cadastrados. Posteriormente, o programa deve mostrar os nomes e os salários dos funcionários cujos nomes começem por uma letra digitada

pelo usuário (considerar a possibilidade de letras maiúsculas e minúsculas). Se nenhum funcionário tem o nome começado com a letra digitada pelo usuário, mostrar mensagem. Finalmente, o programa deve apresentar duas listagens:

- a)** dos nomes e salários dos funcionários por ordem crescente de salário;
- b)** dos nomes e salários dos funcionários por ordem decrescente de salário.

Observação: os percentuais de imposto seguem a tabela abaixo:

| Valor do salário  | Percentual de imposto |
|-------------------|-----------------------|
| até 850           | Isento                |
| entre 850 e 1200  | 10% do salário        |
| de 1200 para cima | 20% do salário        |

5. Faça um programa que receba 20 números e armazene os pares em uma lista simplesmente encadeada e não ordenada e os ímpares em uma segunda lista simplesmente encadeada e não ordenada. Posteriormente, o programa deve montar uma terceira lista, duplamente encadeada e ordenada de forma crescente, com os números das duas listas anteriores. Para finalizar, o programa deve mostrar todos os números da terceira lista das seguintes formas:
  - a)** crescente;
  - b)** decrescente.



# 4

# Estruturas de dados do tipo pilha e fila

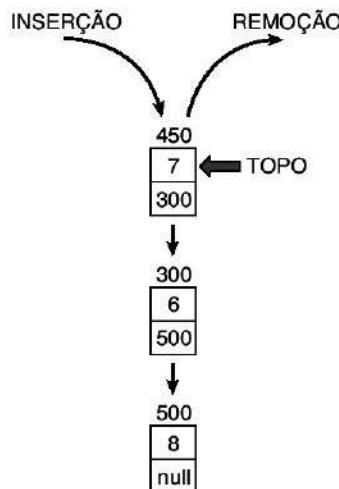
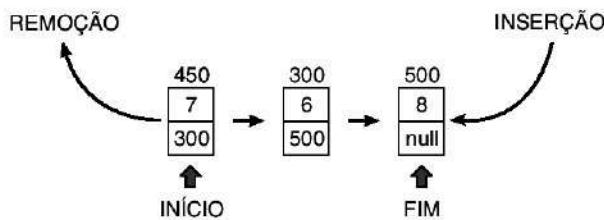
As estruturas de dados do tipo pilha e fila são consideradas listas especializadas por possuírem características próprias. Mas, apesar dessas características, ambas possuem operações como inserir um elemento, excluir um elemento, encontrar o maior, encontrar o menor, contar os elementos, alterar um elemento, buscá-lo, buscar o sucessor e buscar o predecessor.

Essas duas estruturas de dados representam conjuntos de dados que estão organizados em ordem linear. Quando essas estruturas são representadas por arranjos, ou seja, quando é feita a utilização de vetores nas representações, tem-se o uso de endereços contíguos de memória do computador e a ordem linear é determinada pelos índices dos vetores, o que em algumas situações exige maior esforço computacional. Tais representações denominam-se pilha e fila estáticas. Quando as estruturas pilha e fila são representadas por elementos que, além de conter o dado, possuem também um ponteiro para o próximo elemento, ou seja, elementos encadeados, têm-se representações denominadas pilha e fila dinâmicas.

Quando um elemento de uma pilha ou de uma fila contém apenas um dado primitivo, como um número, chama-se pilha ou fila homogênea, e quando um elemento de uma pilha ou fila contém um dado composto, como o nome e o salário de um funcionário, chama-se pilha ou fila heterogênea.

As próximas seções mostram as estruturas pilha e fila com ilustração, implementação dinâmica e análise.

A Figura 4.1 ilustra uma estrutura de dados do tipo pilha, e a Figura 4.2 ilustra uma estrutura de dados do tipo fila.

**Figura 4.1** Estrutura de dados do tipo pilha**Figura 4.2** Estrutura de dados do tipo fila

## Pilha

A estrutura denominada pilha é considerada do tipo FILO (*first in last out*), ou seja, o primeiro elemento inserido será o último a ser removido. Nessa estrutura, cada elemento armazena um ou vários dados (estrutura homogênea ou heterogênea, respectivamente) e um ponteiro para o próximo elemento, permitindo o encadeamento e mantendo a estrutura linear. Nesse tipo de estrutura serão abordadas as seguintes operações: inserir na pilha, consultar toda a pilha, remover e esvaziá-la. Qualquer estrutura desse tipo possui um ponteiro denominado TOPO, no qual todas as operações de inserção e remoção acontecem. Assim, as operações ocorrem sempre na mesma extremidade da estrutura.

Veja a ilustração a seguir, com endereços de memória meramente ilustrativos.



ILUSTRAÇÃO

**1<sup>a</sup> operação****A pilha está vazia**topo  
null**2<sup>a</sup> operação****Inserção do nº 9 na pilha**topo 800  
800 9  
null**3<sup>a</sup> operação****Inserção do nº 3 na pilha**topo 650  
650 3  
800  
↓  
800 9  
null**4<sup>a</sup> operação****Inserção do nº 5 na pilha**topo 750  
750 5  
650  
↓  
650 3  
800  
↓  
800 9  
null**5<sup>a</sup> operação****Remoção da pilha**topo 650  
650 3  
800  
↓  
800 9  
null**6<sup>a</sup> operação****Remoção da pilha**topo 800  
800 9  
null




---

```

import java.util.*;
public class Pilha
{
 //Definindo a classe que representará
 //cada elemento da pilha
 private static class PILHA
 {
 public int num;
 public PILHA prox;
 }

 public static void main(String args[])
 {
 Scanner entrada = new Scanner(System.in);
 // a pilha está vazia, logo,
 // o objeto topo tem o valor null
 // as operações de inserção e remoção
 // acontecem no TOPO
 PILHA topo = null;
 // o objeto aux é um objeto auxiliar
 PILHA aux;
 // apresentando o menu de opções
 int op;
 do
 {
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir na pilha");
 System.out.println("2 - Consultar toda a
 pilha");
 System.out.println("3 - Remover da pilha");
 System.out.println("4 - Esvaziar a pilha");
 System.out.println("5 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();
 if (op < 1 || op > 5)
 System.out.println("Opção inválida!!!");
 if (op == 1)
 {
 System.out.println("Digite o número a ser
 inserido na pilha: ");
 PILHA novo = new PILHA();
 novo.num = entrada.nextInt();
 novo.prox = topo;
 topo = novo;
 }
 } while (op != 5);
 }
}

```

```
System.out.println("Número inserido na
→ pilha!!");
}
if (op == 2)
{
 if (topo == null)
 {
 // a pilha está vazia
 System.out.println("Pilha vazia!!");
 }
 else
 {
 // a pilha contém elementos e
 // estes serão mostrados
 // do último inserido ao primeiro
 System.out.println("\nConsultando a
→ pilha\n");
 aux = topo;
 while (aux != null)
 {
 System.out.print(aux.num+" ");
 aux = aux.prox;
 }
 }
}
if (op == 3)
{
 if (topo == null)
 {
 // a pilha está vazia
 System.out.println("Pilha vazia!!");
 }
 else
 {
 // a pilha contém elementos
 // e o último elemento inserido
 // será removido
 System.out.println("Número "+topo.num+"
→ removido");
 topo = topo.prox;
 }
}
if (op == 4)
{
```

```

 if (topo == null)
 {
 // a pilha está vazia
 System.out.println("Pilha vazia!!!");
 }
 else
 {
 // a pilha será esvaziada
 topo = null;
 System.out.println("Pilha esvaziada");
 }
 }
 while (op != 5);
}
}

```

---



```

#include <iostream.h>
#include <conio.h>

//Definindo o registro que representará cada elemento
//cada elemento da pilha

struct PILHA
{
 int num;
 PILHA *prox;
};

void main()
{
 // a pilha está vazia, logo,
 // o ponteiro topo tem o valor null
 // as operações de inserção e remoção
 // acontecem no TOPO
 PILHA *topo = NULL;
 // o ponteiro aux é um ponteiro auxiliar
 PILHA *aux;
 // apresentando o menu de opções
 int op;
 do

```

```
{
clrscr();
cout<<"\nMENU DE OPÇÕES\n";
cout<<"\n1 - Inserir na pilha";
cout<<"\n2 - Consultar toda a pilha";
cout<<"\n3 - Remover da pilha";
cout<<"\n4 - Esvaziar a pilha";
cout<<"\n5 - Sair";
cout<<"\nDigite sua opção: ";
cin>>op;
if (op < 1 || op > 5)
 cout<<"Opção inválida!!";
if (op == 1)
{
 cout<<"Digite o número a ser inserido na
 ↪ pilha: ";
 PILHA *novo = new PILHA();
 cin>>novo->num;
 novo->prox = topo;
 topo = novo;
 cout<<"Número inserido na pilha!!";
}
if (op == 2)
{
 if (topo == NULL)
 {
 // a pilha está vazia
 cout<<"Pilha vazia!!";
 }
 else
 {
 // a pilha contém elementos e
 // estes serão mostrados
 // do último inserido ao primeiro
 cout<<"\nConsultando toda a pilha\n";
 aux = topo;
 while (aux != NULL)
 {
 cout<<aux->num<<" ";
 aux = aux->prox;
 }
 }
}
if (op == 3)
{
```

```
 if (topo == NULL)
 {
 // a pilha está vazia
 cout<<"Pilha vazia!!";
 }
 else
 {
 // a pilha contém elementos
 // e o último elemento inserido
 // será removido
 aux = topo;
 cout<<"Número "<<topo->num<< " → removido";
 topo = topo->prox;
 delete(aux);
 }
 }
 if (op == 4)
 {
 if (topo == NULL)
 {
 // a pilha está vazia
 cout<<"Pilha vazia!!";
 }
 else
 {
 // a pilha será esvaziada
 aux=topo;
 while(aux!=NULL)
 {
 topo = topo->prox;
 delete(aux);
 aux=topo;
 }
 cout<<"Pilha esvaziada";
 }
 }
 getch();
}
while (op != 5);
}
```

## Análise da complexidade

A operação de inserção e remoção na pilha sempre realiza operações básicas, como a de atribuição, para atualizar o topo da pilha. Logo, são operações de tempo constante e gastam tempo  $O(1)$ .

Já a operação de consultar toda a pilha percorre todos os elementos armazenados nela. Considerando que uma pilha contém  $n$  elementos, o tempo de execução será  $O(n)$ .

A operação de esvaziamento da pilha consiste em remover todos os elementos dela. O tempo gasto nessa operação depende da linguagem de programação que está sendo utilizada. No caso da JAVA, não é necessário remover cada um dos elementos, apenas atualiza-se o ponteiro topo da pilha para nulo e as memórias alocadas serão desalocadas por um procedimento JAVA. Já na linguagem C/C++, é necessário desalocar cada um dos elementos da pilha, gastando tempo proporcional ao tamanho dela, ou seja,  $O(n)$ .

## Fila

A estrutura denominada fila é considerada do tipo FIFO (First In First Out), ou seja, o primeiro elemento inserido será o primeiro a ser removido. Nessa estrutura, cada elemento armazena um ou vários dados (estrutura homogênea ou heterogênea, respectivamente) e um ponteiro para o próximo elemento, permitindo o encadeamento e mantendo a estrutura linear. Nesse tipo de estrutura serão abordadas as seguintes operações: inserir na fila, consultar toda a fila, remover e esvaziá-la. A estrutura do tipo fila possui um ponteiro denominado INICIO, onde as remoções acontecem, e um denominado FIM, onde as inserções acontecem. Assim, as operações ocorrem nas duas extremidades da estrutura.

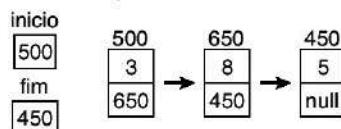
Veja a ilustração a seguir, com endereços de memória meramente ilustrativos.



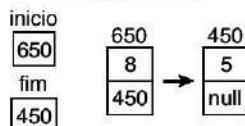
ILUSTRAÇÃO



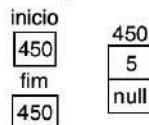
**4<sup>a</sup> operação  
Inserção do nº 5 na fila**



**5<sup>a</sup> operação  
Remoção da fila**



**6<sup>a</sup> operação  
Remoção da fila**



```

import java.util.*;
public class Fila
{
 //Definindo a classe que representará
 //cada elemento da fila
 private static class FILA
 {
 public int num;
 public FILA prox;
 }

 public static void main(String args[])
 {
 Scanner entrada = new Scanner(System.in);
 // a fila está vazia, logo,
 // o objeto inicio tem o valor null
 // a operação de remoção acontece no INÍCIO
 // e a operação de inserção acontece no FIM
 FILA inicio = null;
 FILA fim = null;
 // o objeto aux é um objeto auxiliar
 FILA aux;
 // apresentando o menu de opções
 }
}

```

```
int op;
do
{
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir na fila");
 System.out.println("2 - Consultar toda a fila");
 System.out.println("3 - Remover da fila");
 System.out.println("4 - Esvaziar a fila");
 System.out.println("5 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();
 if (op < 1 || op > 5)
 System.out.println("Opção inválida!!!");
 if (op == 1)
 {
 System.out.println("Digite o número a
➥ ser inserido na fila: ");
 FILA novo = new FILA();
 novo.num = entrada.nextInt();
 novo.prox = null;
 if (inicio == null)
 {
 // a fila está vazia
 // é o número inserido
 // será o primeiro e o último
 inicio = novo;
 fim = novo;
 }
 else
 {
 fim.prox = novo;
 fim = novo;
 }
 System.out.println("Número inserido na
➥ fila!!!");
 }
 if (op == 2)
 {
 if (inicio == null)
 {
 // a fila está vazia
 System.out.println("Fila vazia!!!");
 }
 else
 {
```

```
// a fila contém elementos e
// estes serão mostrados
// do primeiro inserido ao último
System.out.println("\nConsultando toda a
➥ fila\n");
aux = inicio;
while (aux != null)
{
 System.out.print(aux.num+" ");
 aux = aux.prox;
}
}
if (op == 3)
{
 if (inicio == null)
 {
 // a fila está vazia
 System.out.println("Fila vazia!!!");
 }
 else
 {
 // a fila contém elementos
 // e o primeiro elemento
 // inserido será removido
 System.out.println("Número "+inicio.num+
➥ removido");
 inicio = inicio.prox;
 }
}
if (op == 4)
{
 if (inicio == null)
 {
 // a fila está vazia
 System.out.println("Fila vazia!!!");
 }
 else
 {
 // a fila será esvaziada
 inicio = null;
 System.out.println("Fila esvaziada");
 }
}
```

```

 while (op != 5);
 }
}

```

---



```

#include <iostream.h>
#include <conio.h>

//Definindo o registro que representará
//cada elemento da fila

struct FILA
{
 int num;
 FILA *prox;
};

void main()
{
 // a fila está vazia, logo,
 // o ponteiro inicio tem o valor null
 // a operação de remoção acontece no INICIO
 // e a operação de inserção acontece no FIM
 FILA *inicio = NULL;
 FILA *fim = NULL;
 // o ponteiro aux é um ponteiro auxiliar
 FILA *aux;
 // apresentando o menu de opções
 int op;
 do
 {
 clrscr();
 cout<<"\nMENU DE OPÇÕES\n";
 cout<<"\n1 - Inserir na fila";
 cout<<"\n2 - Consultar toda a fila";
 cout<<"\n3 - Remover da fila";
 cout<<"\n4 - Esvaziar a fila";
 cout<<"\n5 - Sair";
 cout<<"\nDigite sua opção: ";
 cin>>op;
 if (op < 1 || op > 5)
 cout<<"Opção inválida!";

```

```
if (op == 1)
{
 cout<<"Digite o número a ser inserido na
 fila: ";
 FILA *novo = new FILA();
 cin>>novo->num;
 novo->prox = NULL;
 if (inicio == NULL)
 {
 // a fila está vazia e o número inserido
 // será o primeiro e o último
 inicio = novo;
 fim = novo;
 }
 else
 {
 fim->prox = novo;
 fim = novo;
 }
 cout<<"Número inserido na fila!!";
}
if (op == 2)
{
 if (inicio == NULL)
 {
 // a fila está vazia
 cout<<"Fila vazia!!";
 }
 else
 {
 // a fila contém elementos
 // e estes serão mostrados
 // do primeiro inserido ao último
 cout<<"\nConsultando toda a fila\n";
 aux = inicio;
 while (aux != NULL)
 {
 cout<<aux->num<<" ";
 aux = aux->prox;
 }
 }
}
if (op == 3)
{
 if (inicio == NULL)
 {
```

```

 // a fila está vazia
 cout<<"Fila vazia!";
 }
else
{
 // a fila contém elementos
 // e o primeiro elemento inserido
 // será removido
 aux = inicio;
 cout<<"Número "<<inicio->num<<"removido";
 inicio = inicio->prox;
 delete(aux);
}
if (op == 4)
{
 if (inicio == NULL)
 {
 // a fila está vazia
 cout<<"Fila vazia!";
 }
 else
 {
 // a fila será esvaziada
 aux = inicio;
 while (aux!= NULL)
 {
 inicio = inicio->prox;
 delete(aux);
 aux=inicio;
 }
 cout<<"Fila esvaziada";
 }
 getch();
}
while (op != 5);
}

```

## Análise da complexidade

A operação de inserção na fila sempre realiza operações básicas, como a de atribuição, para atualizar o INICIO e FIM da fila. O mesmo ocorre no caso da remoção para atualizar o INICIO da fila. Logo, são operações de tempo constante e gastam tempo  $O(1)$ .

Já a operação de consultar toda a fila percorre todos os elementos armazenados nela. Considerando que uma fila contém  $n$  elementos, o tempo de execução será  $O(n)$ .

A operação de esvaziamento da fila consiste em remover todos os seus elementos. O tempo gasto nessa operação depende da linguagem de programação que está sendo utilizada. No caso da JAVA, não é necessário remover cada um dos elementos, apenas atualiza-se o ponteiro início da fila para nulo e as memórias alocadas serão desalocadas por um procedimento JAVA. Já na linguagem C/C++, é necessário desalocar cada um dos elementos da fila, gastando tempo proporcional ao tamanho dela, ou seja,  $O(n)$ .

## Exercícios

- 1)** Faça um programa que cadastre 5 números em uma fila dinâmica e mais 5 em uma pilha dinâmica. Em seguida, o programa deve mostrar três relatórios. O primeiro terá os números que estão nas duas estruturas. O segundo terá os que estão apenas na fila e o terceiro terá os que estão apenas na pilha.

- 2)** Faça um programa que apresente o menu de opções abaixo:

MENU

**1-** Cadastrar número

**2-** Mostrar números pares entre o primeiro e o último número cadastrado

**3-** Excluir número

**4-** Sair

Observações:

**a)** O programa deve ser implementado usando uma estrutura do tipo pilha.

**b)** A opção 1 do menu cobra um número de cada vez.

**c)** Mostrar mensagem para opção inválida do menu.

**d)** Cuidado com o intervalo de números formado pelo primeiro e pelo último número da pilha, pois este pode ser crescente, decrescente ou ainda ser o mesmo número.

**e)** Quando a opção do menu não puder ser realizada, mostrar mensagem.

- 3)** Faça um programa que apresente o menu de opções abaixo:

MENU

**1-** Cadastrar aluno

**2-** Cadastrar nota

**3-** Calcular média de um aluno

**4-** Listar os nomes dos alunos sem notas

**5-** Excluir aluno

**6-** Excluir nota

**7-** Sair

Observações:

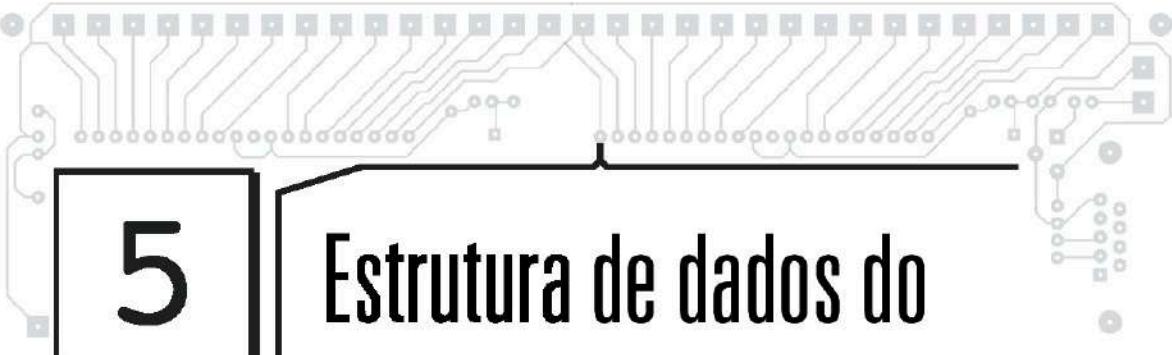
**a)** Na opção 1 deve ser cadastrado um aluno (número e nome) de cada vez em uma pilha. A mensagem disponível nesta opção é: Aluno cadastrado. Os números dos alunos devem ser gerados automaticamente, partindo do nº 1.

**b)** Na opção 2 deve ser cadastrada uma nota (número do aluno e nota) de cada vez em uma fila. Uma nota só pode ser cadastrada se pertencer a um aluno cadastrado na pilha de alunos. As mensagens disponíveis nesta opção são: Nota cadastrada e Aluno não cadastrado. Não é necessário validar a nota digitada e cada aluno pode ter várias notas cadastradas.

- c) Na opção 3 o usuário deve digitar o número de um aluno e o programa deve mostrar o nome dele e a média aritmética das notas desse aluno. As mensagens disponíveis nesta opção são: Aluno não cadastrado, Aluno sem notas e Média do aluno = valor calculado.
- d) Na opção 4 os nomes dos alunos que não possuem notas devem ser listados. As mensagens disponíveis nesta opção são: A listagem dos nomes e Todos os alunos possuem notas.
- e) Na opção 5 um aluno da pilha de alunos deve ser excluído, respeitando duas regras: a) um aluno só pode ser excluído se não possuir notas; b) o usuário não deve escolher o aluno a ser excluído, pois a exclusão deve obedecer as regras de funcionamento de uma estrutura de dados do tipo pilha. As mensagens disponíveis nesta opção são: Aluno excluído, Pilha vazia e Este aluno possui notas, logo, não poderá ser excluído.
- f) Na opção 6 uma nota deve ser excluída, respeitando as regras de funcionamento de uma estrutura do tipo fila. As mensagens disponíveis nesta opção são: Nota excluída e Fila vazia.
- g) A opção 7 é a única que sai do programa. Uma mensagem deve ser mostrada para opções inválidas.
- 4) Faça um programa que cadastre em uma pilha vários números. A entrada deles será finalizada com a digitação de um número menor ou igual a zero. Posteriormente, o programa deve gerar duas filas, a primeira com os números pares e a segunda com os números ímpares. A saída do programa deve apresentar a pilha digitada e as filas geradas. Caso alguma das filas seja vazia, deve-se mostrar a mensagem.
- 5) Faça um programa que apresente o menu de opções abaixo:
- MENU
- 1-** Cadastrar tipo
  - 2-** Cadastrar produto
  - 3-** Consultar o preço de um produto
  - 4-** Excluir tipo
  - 5-** Sair
- Observações:
- a) Mostrar mensagem de opção inválida no menu. A opção 5 é a única que sai do programa.
- b) Para a implementação do programa acima é necessário utilizar duas estruturas de dados do tipo fila.
- c) Na primeira estrutura serão armazenados os tipos dos produtos com seus respectivos percentuais de impostos. Lembrando que não é necessário validar a repetição de tipos, ou seja, suponha que todos os tipos cadastrados são diferentes. Cada tipo é apenas uma letra.
- d) Na segunda estrutura serão armazenados os produtos cujo número deve ser gerado automaticamente. O preço e o tipo devem ser digitados. Lembrando que um produto só pode ser cadastrado se for de um tipo também já cadastrado, faça essa verificação.
- e) Na primeira opção do menu serão cadastrados os tipos, um de cada vez: cada vez que o usuário escolhe a opção 1 do menu, ele tem a possibilidade de cadastrar um novo tipo (letra que representa o tipo e percentual de imposto). Nesta opção, a única mensagem disponível é: Tipo cadastrado.
- f) Na segunda opção do menu serão cadastrados os produtos, um de cada vez: cada vez que o usuário escolhe a opção 2 do menu, ele tem a possibilidade de cadastrar um

novo produto (número gerado automaticamente, preço e tipo digitados). Lembrando que um produto só pode ser cadastrado se o tipo ao qual ele pertence já existe na fila de tipos. Nesta opção as mensagens disponíveis são: Produto cadastrado e Tipo de produto inexistente.

- g)** Na terceira opção do menu o usuário digita o número do produto que deseja consultar o preço e, se este existir na fila de produtos, o programa deve procurar por seu percentual de imposto, de acordo com o tipo do produto na fila de tipos, calcular e mostrar seu preço, ou seja, preço cadastrado menos o percentual de imposto. Nesta opção, as mensagens disponíveis são: Preço = valor calculado, Produto não cadastrado e Fila vazia.
- h)** Na quarta opção o programa deve excluir um tipo da fila de tipos, respeitando a forma de organização de uma fila. Lembrando que um tipo só pode ser excluído se não existir nenhum produto cadastrado para ele.



# 5

# Estrutura de dados do tipo lista de prioridades

A estrutura de dados do tipo lista de prioridades, também denominada *heap*, é composta por um conjunto finito de dados, cada qual com uma chave que determinará sua prioridade dentro da lista. Logo, essa estrutura é um vetor e pode ser vista como uma árvore binária completa onde cada nó possui no máximo dois filhos, e os nós que não possuem dois filhos estão no último ou penúltimo nível. Assim, a árvore que representa a estrutura *heap* é sempre preenchida da esquerda para a direita.

Dado um índice  $i$  do vetor, para se descobrir as posições em que se encontram o elemento pai, o filho esquerdo e o filho direito, realizam-se os cálculos:  $Pai(i) = \lfloor i / 2 \rfloor$ ,  $Filho\ Esquerdo(i) = 2.i$  e  $Filho\ Direito(i) = 2.i + 1$ .

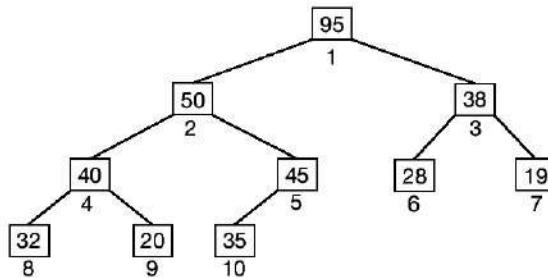
Segundo Cormen (2002), existem dois tipos de *heaps* binários: *heaps máximos* e *heaps mínimos*. Todos os nós em um *heap* atendem a uma determinada propriedade. Se o *heap* é máximo, então a propriedade a ser atendida é  $X[Pai(i)] \geq X[i]$ . Se o *heap* é mínimo, então a propriedade a ser atendida é  $X[Pai(i)] \leq X[i]$ . Logo, em um *heap máximo*, o maior elemento sempre se encontra na raiz, enquanto que em um *heap mínimo* é o menor que sempre está na raiz.

Um exemplo de lista de prioridades bastante próximo à computação é a de processos pendentes usados por um sistema operacional, onde cada processo possui um valor associado que indica a prioridade do trabalho. Os processos são mantidos em uma lista de prioridades, onde os com prioridade urgente (ou mais alta) serão executados antes de quaisquer trabalhos com necessidades menos urgentes.

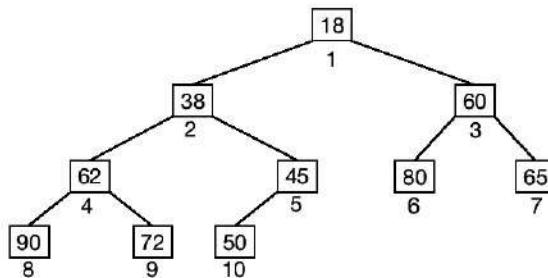
A Figura 5.1 ilustra a estrutura de dados do tipo lista de prioridades como um *heap* máximo, e a Figura 5.2 ilustra um *heap* mínimo.

**Figura 5.1** • Heap máximo

|             |    |    |    |    |    |    |    |    |    |    |    |
|-------------|----|----|----|----|----|----|----|----|----|----|----|
| <i>heap</i> | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
|             | 95 | 50 | 38 | 40 | 45 | 28 | 19 | 32 | 20 | 35 |    |

**Figura 5.2** • Heap mínimo

|             |    |    |    |    |    |    |    |    |    |    |    |
|-------------|----|----|----|----|----|----|----|----|----|----|----|
| <i>heap</i> | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
|             | 18 | 38 | 60 | 62 | 45 | 80 | 65 | 90 | 72 | 50 |    |



## Heap máximo e heap mínimo

Nestas estruturas, as seguintes propriedades devem ser obedecidas:

- o elemento raiz deve ocupar a posição 1 do vetor, o que facilita o cálculo dos endereços dos elementos filhos da direita e da esquerda;
- o vetor utilizado para armazenar  $n$  elementos tem a posição inicial zero e como esta não será utilizada o vetor deve ser declarado com  $n+1$  posições;
- se o *heap* for máximo, a prioridade de um elemento pai deve ser maior que a de seus filhos. Esta propriedade deve ser aplicada a cada elemento da lista.
- se o *heap* for mínimo, a prioridade de um elemento pai deve ser menor que a de seus filhos. Esta propriedade deve ser aplicada a cada elemento da lista.

As operações comumente aplicadas em um *heap* máximo ou mínimo são: inserir um elemento na estrutura *heap*; remover um da estrutura *heap* (o de maior prioridade em um *heap* máximo ou o de menor em um *heap* mínimo); consultar um elemento (de maior prioridade em um *heap* máximo ou de menor em um *heap* mínimo).

Na operação de inserção o elemento deveria ser inserido na próxima posição livre do vetor, porém, antes de realizar a inserção de fato, é verificado se o pai atende à propriedade *heap*. Caso não atenda, o pai é copiado para a posição de inserção e a posição antiga do pai é avaliada para inserir o novo elemento aplicando a mesma regra, sucessivamente, até encontrar a posição correta.

A operação de remoção sempre retira o elemento de maior prioridade (*heap* máximo) ou o elemento de menor prioridade (*heap* mínimo), que é o encontrado na raiz. A remoção é feita colocando o último elemento no lugar do primeiro. Como após esta movimentação a raiz passa a não atender mais a propriedade *heap*, um procedimento é aplicado para que o conjunto de elementos volte a ser um *heap*. O procedimento é o *Heap\_Fica*, já apresentado no Capítulo 2. Ele é aplicado sempre a um nó da árvore, que representa na verdade um elemento do vetor, e "afunda" esse nó até que a propriedade do *heap* seja válida.

A operação de consultar um elemento acessa o valor da raiz da árvore, que é o primeiro elemento do vetor.

A ilustração a seguir mostra um *heap* máximo com capacidade de até sete números, e "ind" representa a posição a ser ocupada pelo número a ser inserido.



ILUSTRAÇÃO

### 1<sup>a</sup> operação A lista de prioridades (*heap* máximo) está vazia

|             |   |   |   |   |   |   |   |
|-------------|---|---|---|---|---|---|---|
| 0           | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| <i>heap</i> |   |   |   |   |   |   |   |

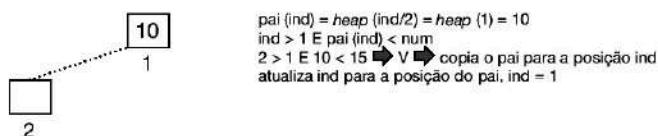
### 2<sup>a</sup> operação Inserção do nº 10 na lista de prioridades (*heap* máximo)

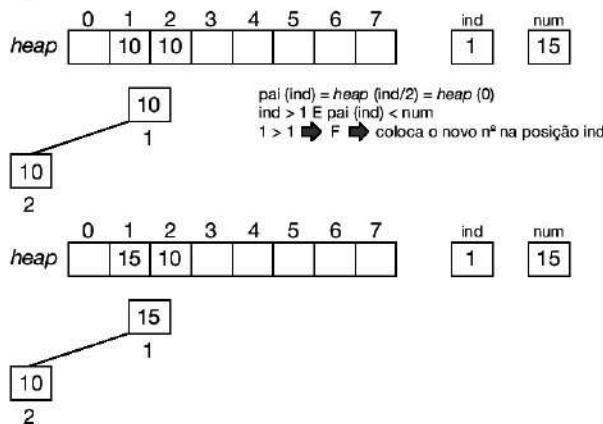
|             |    |   |   |   |   |   |   |
|-------------|----|---|---|---|---|---|---|
| 0           | 1  | 2 | 3 | 4 | 5 | 6 | 7 |
| <i>heap</i> | 10 |   |   |   |   |   |   |

|    |
|----|
| 10 |
| 1  |

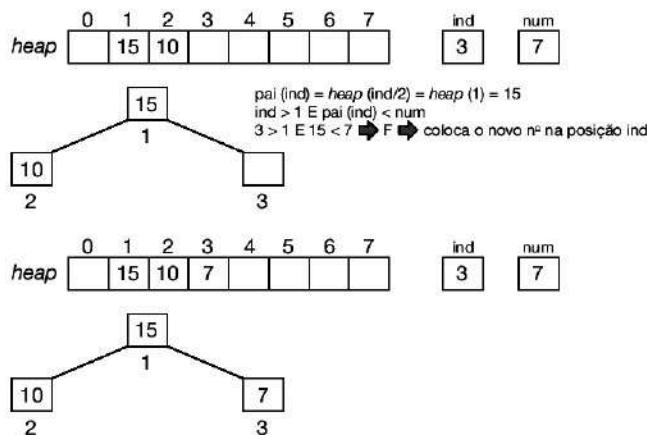
### 3<sup>a</sup> operação Inserção do nº 15 na lista de prioridades (*heap* máximo)

|             |    |   |   |   |   |   |   |
|-------------|----|---|---|---|---|---|---|
| 0           | 1  | 2 | 3 | 4 | 5 | 6 | 7 |
| <i>heap</i> | 10 |   |   |   |   |   |   |

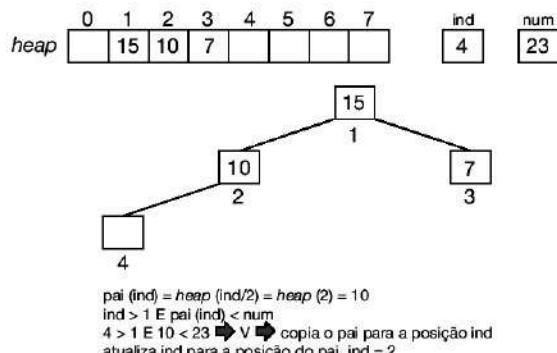




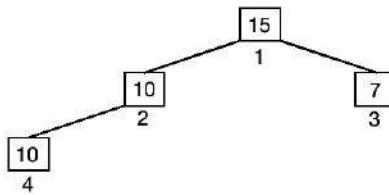
#### 4<sup>a</sup> operação Inserção do nº 7 na lista de prioridades (heap máximo)



#### 5<sup>a</sup> operação Inserção do nº 23 na lista de prioridades (heap máximo)

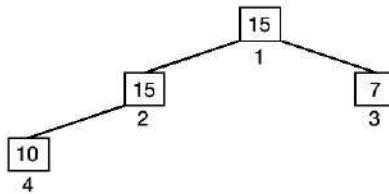


| heap | 0  | 1  | 2 | 3  | 4 | 5 | 6 | 7 | ind | num |
|------|----|----|---|----|---|---|---|---|-----|-----|
|      | 15 | 10 | 7 | 10 |   |   |   |   | 2   | 23  |



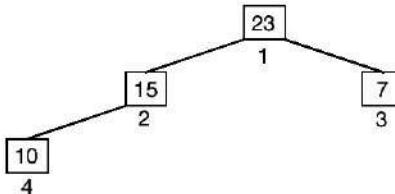
pai(ind) = heap(ind/2) = heap(1) = 15  
 ind > 1 E pai(ind) < num  
 2 > 1 E 15 < 23  $\Rightarrow$  V  $\Rightarrow$  copia o pai para a posição ind  
 atualiza ind para a posição do pai, ind = 1

| heap | 0  | 1  | 2 | 3  | 4 | 5 | 6 | 7 | ind | num |
|------|----|----|---|----|---|---|---|---|-----|-----|
|      | 15 | 15 | 7 | 10 |   |   |   |   | 1   | 23  |



pai(ind) = heap(ind/2) = heap(0)  
 ind > 1 E pai(ind) < num  
 1 > 1  $\Rightarrow$  F  $\Rightarrow$  coloca o novo n° na posição ind

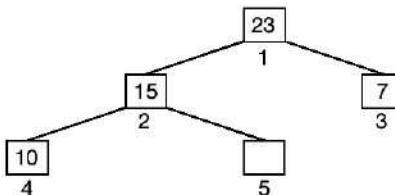
| heap | 0  | 1  | 2 | 3  | 4 | 5 | 6 | 7 | ind | num |
|------|----|----|---|----|---|---|---|---|-----|-----|
|      | 23 | 15 | 7 | 10 |   |   |   |   | 1   | 23  |



#### 6ª operação

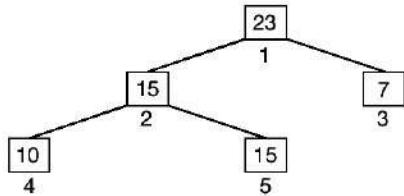
Inserção do nº 18 na lista de prioridades (heap máximo)

| heap | 0  | 1  | 2 | 3  | 4 | 5 | 6 | 7 | ind | num |
|------|----|----|---|----|---|---|---|---|-----|-----|
|      | 23 | 15 | 7 | 10 |   |   |   |   | 5   | 18  |



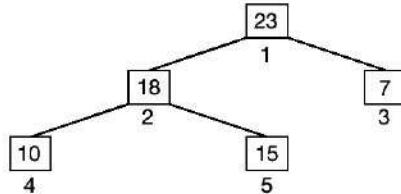
pai(ind) = heap(ind/2) = heap(2) = 15  
 ind > 1 E pai(ind) < num  
 5 > 1 E 15 < 18  $\Rightarrow$  V  $\Rightarrow$  copia o pai para a posição ind  
 atualiza ind para a posição do pai, ind = 2

| <i>heap</i> | 0  | 1  | 2 | 3  | 4  | 5 | 6 | 7 | <i>ind</i> | <i>num</i> |
|-------------|----|----|---|----|----|---|---|---|------------|------------|
|             | 23 | 15 | 7 | 10 | 15 |   |   |   | 2          | 18         |



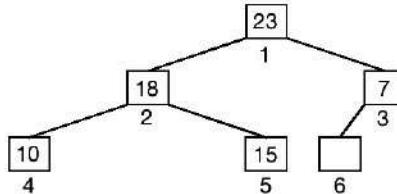
$\text{pai(ind)} = \text{heap}(\text{ind}/2) = \text{heap}(1) = 23$   
 $\text{ind} > 1 \text{ E } \text{pai(ind)} < \text{num}$   
 $2 > 1 \text{ E } 23 < 18 \Rightarrow F \Rightarrow \text{coloca o novo n\º na pos\º ind}$

| <i>heap</i> | 0  | 1  | 2 | 3  | 4  | 5 | 6 | 7 | <i>ind</i> | <i>num</i> |
|-------------|----|----|---|----|----|---|---|---|------------|------------|
|             | 23 | 18 | 7 | 10 | 15 |   |   |   | 2          | 18         |



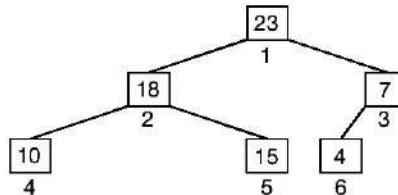
#### 7ª operação Inserção do nº 4 na lista de prioridades (*heap* máximo)

| <i>heap</i> | 0  | 1  | 2 | 3  | 4  | 5 | 6 | 7 | <i>ind</i> | <i>num</i> |
|-------------|----|----|---|----|----|---|---|---|------------|------------|
|             | 23 | 18 | 7 | 10 | 15 |   |   |   | 6          | 4          |



$\text{pai(ind)} = \text{heap}(\text{ind}/2) = \text{heap}(3) = 7$   
 $\text{ind} > 1 \text{ E } \text{pai(ind)} < \text{num}$   
 $6 > 1 \text{ E } 7 < 4 \Rightarrow F \Rightarrow \text{coloca o novo n\º na pos\º ind}$

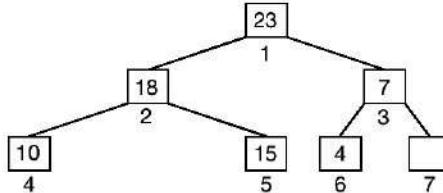
| <i>heap</i> | 0  | 1  | 2 | 3  | 4  | 5 | 6 | 7 | <i>ind</i> | <i>num</i> |
|-------------|----|----|---|----|----|---|---|---|------------|------------|
|             | 23 | 18 | 7 | 10 | 15 | 4 |   |   | 6          | 4          |



8<sup>a</sup> operação

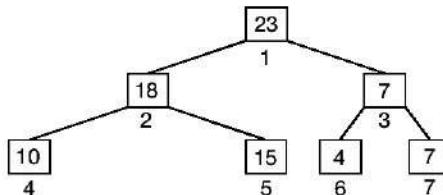
Inserção do nº 36 na lista de prioridades (heap máximo)

| heap | 0  | 1  | 2 | 3  | 4  | 5 | 6 | 7 | ind | num |
|------|----|----|---|----|----|---|---|---|-----|-----|
|      | 23 | 18 | 7 | 10 | 15 | 4 |   |   | 7   | 36  |



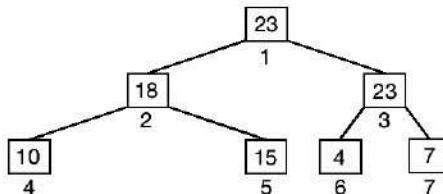
pai (ind) = heap (ind/2) = heap (3) = 7  
 ind > 1 E pai (ind) < num  
 7 > 1 E 7 < 36 → V copia o pai para a posição ind  
 atualiza ind para a posição do pai, ind = 3

| heap | 0  | 1  | 2 | 3  | 4  | 5 | 6 | 7 | ind | num |
|------|----|----|---|----|----|---|---|---|-----|-----|
|      | 23 | 18 | 7 | 10 | 15 | 4 | 7 |   | 3   | 36  |



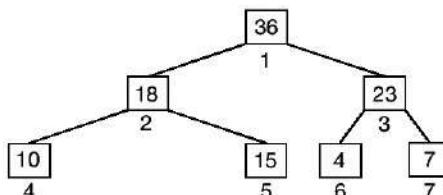
pai (ind) = heap (ind/2) = heap (1) = 23  
 ind > 1 E pai (ind) < num  
 3 > 1 E 23 < 36 → V copia o pai para a posição ind  
 atualiza ind para a posição do pai, ind = 1

| heap | 0  | 1  | 2  | 3  | 4  | 5 | 6 | 7 | ind | num |
|------|----|----|----|----|----|---|---|---|-----|-----|
|      | 23 | 18 | 23 | 10 | 15 | 4 | 7 |   | 1   | 36  |



pai (ind) = heap (ind/2) = heap (0)  
 ind > 1 E pai (ind) < num  
 1 > 1 → F coloca o novo nº na posição ind

| heap | 0  | 1  | 2  | 3  | 4  | 5 | 6 | 7 | ind | num |
|------|----|----|----|----|----|---|---|---|-----|-----|
|      | 36 | 18 | 23 | 10 | 15 | 4 | 7 |   | 1   | 36  |



9<sup>a</sup> operação

## Remoção da lista de prioridades (heap máximo)

o elemento da posição 1 será removido = *heap* (1) = 36

| 0           | 1  | 2  | 3  | 4  | 5  | 6 | 7 |
|-------------|----|----|----|----|----|---|---|
| <i>heap</i> | 36 | 18 | 23 | 10 | 15 | 4 | 7 |

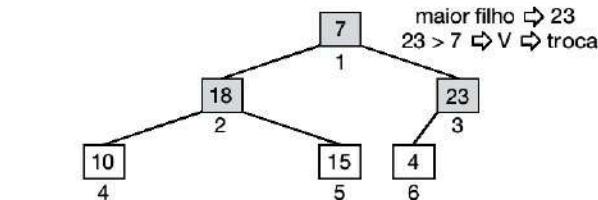
o elemento da última posição ocupada será copiado para a posição 1

| 0           | 1 | 2  | 3  | 4  | 5  | 6 | 7 |
|-------------|---|----|----|----|----|---|---|
| <i>heap</i> | 7 | 18 | 23 | 10 | 15 | 4 | 7 |

o procedimento *heap\_fica* será executado para o elemento da posição 1

o elemento da última posição foi descartado

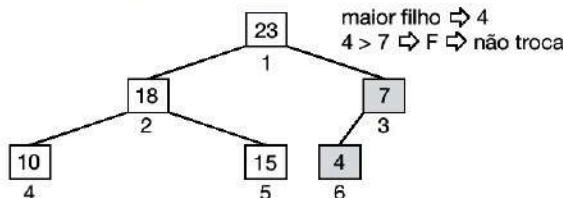
| 0           | 1 | 2  | 3  | 4  | 5  | 6 | 7 |
|-------------|---|----|----|----|----|---|---|
| <i>heap</i> | 7 | 18 | 23 | 10 | 15 | 4 |   |



| 0           | 1  | 2  | 3 | 4  | 5  | 6 | 7 |
|-------------|----|----|---|----|----|---|---|
| <i>heap</i> | 23 | 18 | 7 | 10 | 15 | 4 |   |

o procedimento *heap\_fica* será executado para o elemento da posição 3

| 0           | 1  | 2  | 3 | 4  | 5  | 6 | 7 |
|-------------|----|----|---|----|----|---|---|
| <i>heap</i> | 23 | 18 | 7 | 10 | 15 | 4 |   |

10<sup>a</sup> operação

## Remoção da lista de prioridades (heap máximo)

o elemento da posição 1 será removido = *heap* (1) = 23

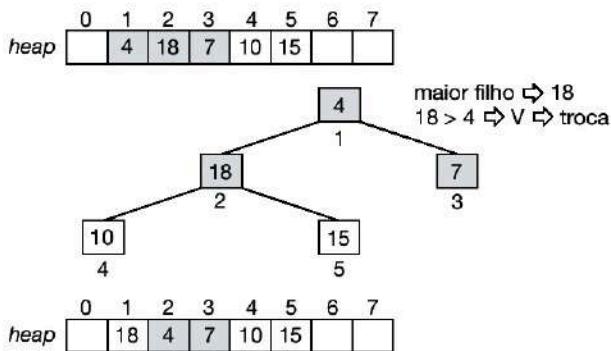
| 0           | 1  | 2  | 3 | 4  | 5  | 6 | 7 |
|-------------|----|----|---|----|----|---|---|
| <i>heap</i> | 23 | 18 | 7 | 10 | 15 | 4 |   |

o elemento da última posição ocupada será copiado para a posição 1

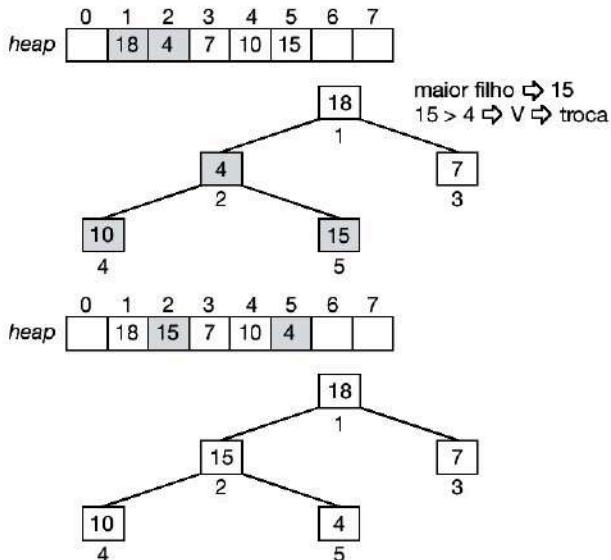
| 0           | 1 | 2  | 3 | 4  | 5  | 6 | 7 |
|-------------|---|----|---|----|----|---|---|
| <i>heap</i> | 4 | 18 | 7 | 10 | 15 | 4 |   |

o procedimento *heap\_fica* será executado para o elemento da posição 1

o elemento da última posição foi descartado



o procedimento *heap\_fica* será executado para o elemento da posição 2



A ilustração de um *heap* mínimo funciona da mesma forma, alterando apenas a condição para que só haja troca quando o pai possui prioridade maior que a do filho. Logo, a condição para troca será:  $ind > 1$  E  $pai(ind) < num$ .




---

```

import java.util.*;
public class Heap_Maximo
{
 // declarando o vetor com capacidade de no máximo 10
 // números
 static int vet[] = new int[11];

 public static void main(String[] args)
 {
 Scanner entrada = new Scanner(System.in);
 int op, tam=0, ind, num;
 }
}

```

```
do
{
 System.out.println("\nMENU DE OPÇÕES - HEAP MÁXIMO\n");
 System.out.println("1 - Inserir elemento na lista de
 prioridades");
 System.out.println("2 - Consultar o elemento de maior
 prioridade");
 System.out.println("3 - Remover o elemento de maior
 prioridade");
 System.out.println("4 - Consultar toda a lista");
 System.out.println("5 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();

 if (op < 1 || op > 5)
 System.out.println("Opção inválida!!");

 else if (op==1)
 {
 // verifica se ainda existe espaço disponível no vetor
 // para inserção do novo número
 if(tam < vet.length-1)
 {
 tam++;
 System.out.print("Digite um número: ");
 // leitura do número a ser inserido
 num = entrada.nextInt();
 ind=tam;
 while(ind>1 && vet[Pai(ind)]< num)
 {
 vet[ind]=vet[Pai(ind)];
 ind=Pai(ind);
 }
 vet[ind]=num;
 System.out.println("Número inserido");
 }
 else System.out.println("Lista de prioridades
 ↪ Lotada!");
 }

 else if (op==2)
 {
 if (tam==0)
 System.out.println("Lista de prioridades
 ↪ vazia!");
 }
}
```

```
 else
 System.out.println("Elemento de maior
 ↪ prioridade: " +vet[1]);
 }

 else if (op==3)
 {
 if (tam==0)
 System.out.println("Lista de prioridades vazia!");
 else
 {
 int maior_prior=vet[1];
 vet[1]=vet[tam];
 tam--;
 heap_fica(1, tam);
 System.out.println("O elemento removido:
 " +maior_prior);
 }
 }

 else if (op==4)
 {
 if (tam==0)
 System.out.println("Lista de prioridades vazia!");
 else
 {
 System.out.println("\nTodos os elementos
 ↪ da lista de prioridades\n");
 for(int j=1; j<=tam; j++)
 System.out.print(vet[j]+ " ");
 System.out.println();
 }
 }
}

while(op!=5);
}

public static int Pai (int x)
{
 return x/2;
}

public static void heap_fica(int i, int qtde)
{
 int f_esq, f_dir, maior, aux;
```

```

 maior = i;
 if (2*i+1 <= qtde)
 {
 // o nó que está sendo analisado
 // tem filhos p/ esquerda e direita
 f_esq = 2*i+1;
 f_dir = 2*i;
 if (vet[f_esq] >= vet[f_dir] &&
 vet[f_esq] > vet[i])
 maior = 2*i+1;
 else if(vet[f_dir] > vet[f_esq] &&
 vet[f_dir] > vet[i])
 maior = 2*i;
 }
 else if (2*i <= qtde)
 {
 // o nó que está sendo analisado
 // tem filho apenas p/ a esquerda
 f_dir = 2*i;
 if (vet[f_dir] > vet[i])
 maior = 2*i;
 }
 if (maior != i)
 {
 aux = vet[i];
 vet[i] = vet[maior];
 vet[maior] = aux;
 heap_fica(maior,qtde);
 }
 }
}

```

---



```

#include<iostream.h>
#include<conio.h>

int Pai (int x)
{
 return x/2;
}

void heap_fica(int vet[], int i, int qtde)
{
 int f_esq, f_dir, maior, aux;
 maior = i;

```

```

if (2*i+1 <= qtde)
{
 // o nó que está sendo analisado tem filhos p/
 // esquerda e direita
 f_esq = 2*i+1;
 f_dir = 2*i;
 if (vet[f_esq] >= vet[f_dir] &&
 vet[f_esq] > vet[i])
 maior = 2*i+1;
 else if (vet[f_dir] > vet[f_esq] &&
 vet[f_dir] > vet[i])
 maior = 2*i;
}
else if (2*i <= qtde)
{
 // o nó que está sendo analisado tem filho apenas
 // p/ a
 // esquerda
 f_dir = 2*i;
 if (vet[f_dir] > vet[i])
 maior = 2*i;
}
if (maior != i)
{
 aux = vet[i];
 vet[i] = vet[maior];
 vet[maior] = aux;
 heap_fica(vet,maior,qtde);
}
}

void main()
{
 // declarando o vetor com capacidade de no máximo 10
 // números
 int vet[11];

 int op, tam=0, ind, num;
 do
 {
 clrscr();
 cout << "\nMENU DE OPÇÕES - HEAP MÁXIMO\n";
 cout << "\n1 - Inserir elemento na lista de
 ↪ prioridades";
 cout << "\n2 - Consultar o elemento de maior

```

```
 ↵ prioridade";
 cout << "\n3 - Remover o elemento de maior
 ↵ prioridade";
 cout << "\n4 - Consultar toda a lista";
 cout << "\n5 - Sair";
 cout << "\nDigite sua opção: ";
 cin >> op;

 if (op < 1 || op > 5)
 cout << "\nOpção inválida!!";
 else if (op==1)
 {
 // verifica se ainda existe espaço disponível
 // no vetor para inserção do novo número
 if(tam < 10)
 {
 tam++;
 cout << "Digite um número: ";
 // leitura do número a ser inserido
 cin >> num;
 ind=tam;
 while(ind>1 && vet[Pai(ind)]< num)
 {
 vet[ind]=vet[Pai(ind)];
 ind=Pai(ind);
 }
 vet[ind]=num;
 cout << "Número inserido";
 }
 else cout << "Lista de prioridades Lotada!";
 }
 else if (op==2)
 {
 if (tam==0)
 cout << "Lista de prioridades vazia!";
 else
 cout << "Elemento de maior prioridade: "
 << vet[1];
 }
 else if (op==3)
 {
 if (tam==0)
 cout << "Lista de prioridades vazia!";
 else
 {
```

```

 int maior_prior=vet[1];
 vet[1]=vet[tam];
 tam--;
 heap_fica(vet,1, tam);
 cout << "O elemento removido: "
 <<maior_ prior;
 }
}
else if (op==4)
{
 if (tam==0)
 cout << "Lista de prioridades vazia!";
 else
 {
 cout << "\nTodos os elementos da lista de "
 << "prioridades\n";
 for(int j=1; j<=tam; j++)
 cout << vet[j] << " ";
 cout << "\n";
 }
}
getch();
}
while(op!=5);
}

```

---

## Análise da complexidade das estruturas

### *heap* máximo e *heap* mínimo

A operação de consulta do elemento de maior ou menor prioridade (*heap* máximo ou *heap* mínimo, respectivamente) em uma lista de prioridades gasta tempo constante,  $O(1)$ , uma vez que a operação acessa apenas o primeiro elemento da lista.

A operação de remoção do elemento de maior ou menor prioridade (*heap* máximo ou *heap* mínimo), ou seja, o elemento raiz, gasta tempo  $O(\log n)$ , visto que o último elemento da lista é colocado no lugar do removido, e o procedimento *Heap\_Fica* é aplicado sobre o mesmo. Como no pior caso o elemento raiz deverá ser trocado até chegar em uma folha, o número de trocas realizadas será correspondente à altura da árvore, cujo valor é  $O(\log n)$ .

Já a operação de inserção na lista sempre tenta incluir o novo elemento no fim, porém verificando se na posição a ser inserida a propriedade *heap* será mantida. Quando isso não acontece, o novo elemento vai sendo levado um nível acima até encontrar a posição correta em que a propriedade *heap* é satisfeita. No pior caso, o elemento será colocado na raiz, ou seja, percorreu o caminho desde a folha até à raiz, gastando tempo proporcional à altura da árvore,  $O(\log n)$ .

## Heap max-min e heap min-max

Em algumas aplicações utilizando *heap*, é necessário acessar o elemento que possui a maior prioridade e também o elemento que possui a menor prioridade. Nesses casos, a estrutura a ser implementada é chamada *heap max-min* ou *heap min-max* e estas, assim como as estruturas *heap* máximo e mínimo, são vetores e podem ser vistas como árvores binárias completas.

As operações comumente aplicadas em um *heap max-min* e em um *heap min-max* são: consulta do elemento de maior ou o de menor prioridade, inserção de um elemento e remoção do elemento de maior ou de menor prioridade.

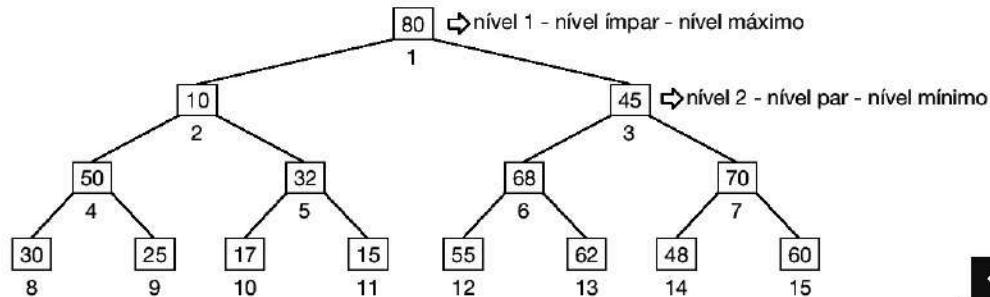
Nessas estruturas, as seguintes propriedades devem ser obedecidas:

- o elemento raiz deve ocupar a posição 1 do vetor, o que facilita o cálculo dos endereços dos elementos filhos da direita e da esquerda;
- o vetor utilizado para armazenar  $n$  elementos tem a posição inicial zero e, como esta não será utilizada, ele deve ser declarado com  $n+1$  posições;
- em um *heap max-min*, o elemento de maior prioridade ocupa sempre a posição 1 do vetor, e o elemento de menor prioridade ocupa sempre a posição 2 ou 3 do vetor;
- em um *heap min-max*, o elemento de menor prioridade ocupa sempre a posição 1 do vetor, e o elemento de maior prioridade ocupa sempre a posição 2 ou 3 do vetor;
- cada elemento do vetor, denotado por  $v_i$ , corresponde a um nó da árvore binária e encontra-se em um nível. O nível de um nó  $i$  é determinado pela seguinte fórmula:  $\log_2 i + 1$ , para  $1 \leq i \leq n$ ;
- em um *heap max-min*, os níveis ímpares são chamados de níveis máximos e os pares de níveis mínimos;
- em um *heap min-max*, os níveis ímpares são chamados de níveis mínimos e os pares de níveis máximos;
- em um *heap max-min*, se o nível de um nó  $i$  é máximo, ou seja, ímpar, a sua chave é superior à chave do seu pai ( $v_i > v_{[i/2]}$  quando  $i \geq 4$ ), e inferior à chave do seu avô ( $v_i < v_{[i/4]}$  quando  $i \geq 4$ );
- em um *heap min-max*, se o nível de um nó  $i$  é mínimo, ou seja, ímpar, a sua chave é inferior à chave do seu pai ( $v_i < v_{[i/2]}$  quando  $i \geq 4$ ), e superior à chave do seu avô ( $v_i > v_{[i/4]}$  quando  $i \geq 4$ ).

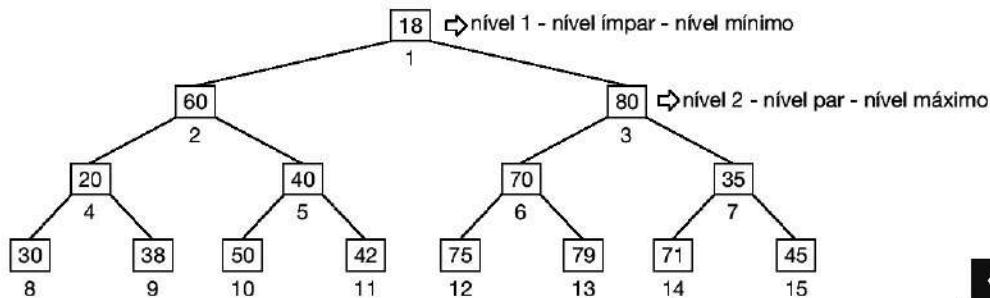
De acordo com Szwarcfiter (1994), em um *heap max-min* e em um *heap min-max*, a chave de um nó  $i$  é superior às chaves de seus descendentes quando  $i$  está situado em um nível máximo, e inferior às chaves de seus descendentes quando  $i$  é situado em um nível mínimo. Na Figura 5.3, é apresentado um exemplo de *heap max-min* e, na Figura 5.4, um exemplo de *heap min-max*.

**Figura 5.3** • Heap max-min

|      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| heap | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
|      | 80 | 10 | 45 | 50 | 32 | 68 | 70 | 30 | 25 | 17 | 15 | 55 | 62 | 48 | 60 |    |

**Figura 5.4** • Heap min-max

|      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| heap | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
|      | 18 | 60 | 80 | 20 | 40 | 70 | 35 | 30 | 38 | 50 | 42 | 75 | 79 | 71 | 45 |    |



ILUSTRAÇÃO

1<sup>a</sup> operação  
A lista de prioridades (heap min-max) está vazia

|      |   |   |   |   |   |   |   |   |   |   |    |
|------|---|---|---|---|---|---|---|---|---|---|----|
| heap | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|      |   |   |   |   |   |   |   |   |   |   |    |

2<sup>a</sup> operação  
Inserção do nº 25 na lista de prioridades (heap min-max)

|      |    |   |   |   |   |   |   |   |   |   |    |   |     |
|------|----|---|---|---|---|---|---|---|---|---|----|---|-----|
| heap | 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | i | num |
|      | 25 |   |   |   |   |   |   |   |   |   |    | 1 | 25  |

25

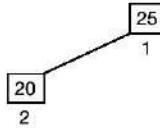
1

nível =  $\log_2 1 + 1 = 1 \Rightarrow$  nível ímpar  $\Rightarrow$  nível mínimo  
 pai =  $1/2 = 1/2 = 0$   
 verifica se a posição do novo nº possui pai  $\Rightarrow$  pai  $>= 1$   
 $0 >= 1 \Rightarrow F \Rightarrow$  novo nº permanece na posição onde foi inserido, posição 1

**3<sup>a</sup> operação**

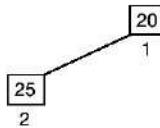
Inserção do nº 20 na lista de prioridades (heap min-max)

| heap | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | i | num |
|------|----|----|---|---|---|---|---|---|---|---|----|---|-----|
|      | 25 | 20 |   |   |   |   |   |   |   |   |    | 2 | 20  |



nível =  $\log_2 i + 1 = 2 \Rightarrow$  nível par  $\Rightarrow$  nível máximo  
 pai =  $i/2 = 2/2 = 1$   
 verifica se a posição do novo nº possui pai  $\Rightarrow$  pai  $>= 1$   
 $1 >= 1$  verifica se o novo nº é menor que seu pai  
 $heap[i] < heap[pai] \Rightarrow 20 < 25 \Rightarrow V \Rightarrow$  troca e atualiza i para a posição do pai, i = 1

| heap | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | i | num |
|------|----|----|---|---|---|---|---|---|---|---|----|---|-----|
|      | 20 | 25 |   |   |   |   |   |   |   |   |    | 1 | 20  |

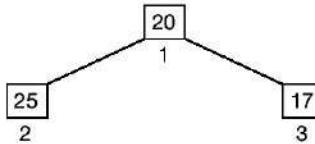


avo =  $i/4 = 1/4 = 0$   
 verifica se a posição i possui avô  $\Rightarrow$  avo  $>= 1$   
 $0 >= 1 \Rightarrow F \Rightarrow$  novo nº permanece na posição onde está

**4<sup>a</sup> operação**

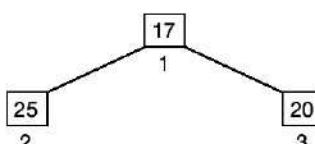
Inserção do nº 17 na lista de prioridades (heap min-max)

| heap | 0  | 1  | 2  | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | i | num |
|------|----|----|----|---|---|---|---|---|---|---|----|---|-----|
|      | 20 | 25 | 17 |   |   |   |   |   |   |   |    | 3 | 17  |



nível =  $\log_2 i + 1 = 2 \Rightarrow$  nível par  $\Rightarrow$  nível máximo  
 pai =  $i/2 = 3/2 = 1$   
 verifica se a posição do novo nº possui pai  $\Rightarrow$  pai  $>= 1$   
 $1 >= 1 \Rightarrow V \Rightarrow$  verifica se o novo nº é menor que seu pai  
 $heap[i] < heap[pai] \Rightarrow 17 < 20 \Rightarrow V \Rightarrow$  troca e atualiza i para a posição do pai, i = 1

| heap | 0  | 1  | 2  | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | i | num |
|------|----|----|----|---|---|---|---|---|---|---|----|---|-----|
|      | 17 | 25 | 20 |   |   |   |   |   |   |   |    | 1 | 17  |

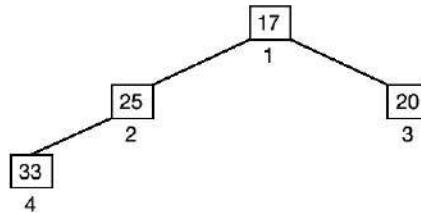


avo =  $i/4 = 1/4 = 0$   
 verifica se a posição i possui avô  $\Rightarrow$  avo  $>= 1$   
 $0 >= 1 \Rightarrow F \Rightarrow$  novo nº permanece na posição onde está

**5<sup>a</sup> operação**  
**Inserção do nº 33 na lista de prioridades (heap min-max)**

|      |    |    |    |    |   |   |   |   |   |   |    |  |
|------|----|----|----|----|---|---|---|---|---|---|----|--|
| heap | 0  | 1  | 2  | 3  | 4 | 5 | 6 | 7 | 8 | 9 | 10 |  |
|      | 17 | 25 | 20 | 33 |   |   |   |   |   |   |    |  |

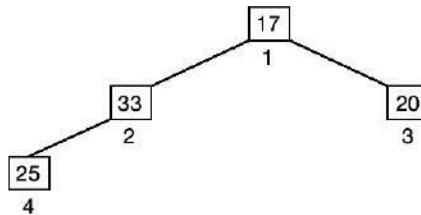
|   |    |
|---|----|
| i | 4  |
|   | 33 |



nível =  $\log_2 i + 1 = 3 \Rightarrow$  nível ímpar  $\Rightarrow$  nível mínimo  
 pai =  $i/2 = 4/2 = 2$   
 verifica se a posição do novo nº possui pai  $\Rightarrow$  pai  $>= 1$   
 $2 >= 1 \Rightarrow V \Rightarrow$  verifica se o novo nº é maior que seu pai  
 $heap(i) > heap(pai) \Rightarrow 33 > 25 \Rightarrow V \Rightarrow$  troca e atualiza i para a posição do pai, i = 2

|      |    |    |    |    |   |   |   |   |   |   |    |  |
|------|----|----|----|----|---|---|---|---|---|---|----|--|
| heap | 0  | 1  | 2  | 3  | 4 | 5 | 6 | 7 | 8 | 9 | 10 |  |
|      | 17 | 33 | 20 | 25 |   |   |   |   |   |   |    |  |

|   |    |
|---|----|
| i | 2  |
|   | 33 |

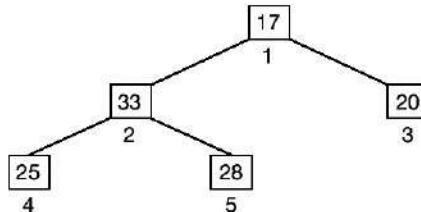


avô =  $i/4 = 2/4 = 0$   
 verifica se a posição i possui avô  $\Rightarrow$  avô  $>= 1$   
 $0 >= 1 \Rightarrow F \Rightarrow$  novo nº permanece na posição onde está

**6<sup>a</sup> operação**  
**Inserção do nº 28 na lista de prioridades (heap min-max)**

|      |    |    |    |    |    |   |   |   |   |   |    |  |
|------|----|----|----|----|----|---|---|---|---|---|----|--|
| heap | 0  | 1  | 2  | 3  | 4  | 5 | 6 | 7 | 8 | 9 | 10 |  |
|      | 17 | 33 | 20 | 25 | 28 |   |   |   |   |   |    |  |

|   |    |
|---|----|
| i | 5  |
|   | 28 |

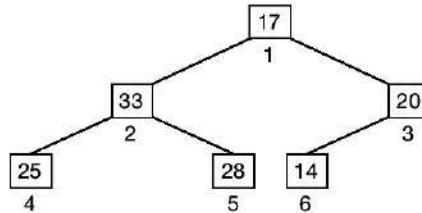


nível =  $\log_2 i + 1 = 3 \Rightarrow$  nível ímpar  $\Rightarrow$  nível mínimo  
 pai =  $i/2 = 5/2 = 2$   
 verifica se a posição do novo nº possui pai  $\Rightarrow$  pai  $>= 1$   
 $2 >= 1 \Rightarrow V \Rightarrow$  verifica se o novo nº é maior que seu pai  
 $heap(i) > heap(pai) \Rightarrow 28 > 33 \Rightarrow F \Rightarrow$  verifica se o novo nº possui avô  
 avô =  $i/4 = 5/4 = 1$   
 verifica se a posição i possui avô  $\Rightarrow$  avô  $= >= 1$   
 $1 >= 1 \Rightarrow V \Rightarrow$  verifica se o novo nº é menor que seu avô  
 $heap(i) > heap(avô) \Rightarrow 28 < 17 \Rightarrow F \Rightarrow$  novo nº permanece na posição onde está

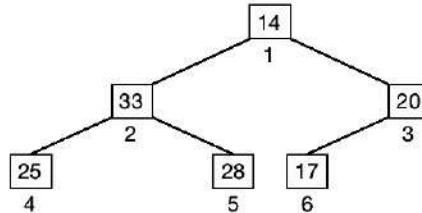
7<sup>a</sup> operação

Inserção do nº 14 na lista de prioridades (heap min-max)

| heap | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | 10 | i | 6 | num | 14 |
|------|----|----|----|----|----|----|---|---|---|---|----|---|---|-----|----|
|      | 17 | 33 | 20 | 25 | 28 | 14 |   |   |   |   |    |   |   |     |    |

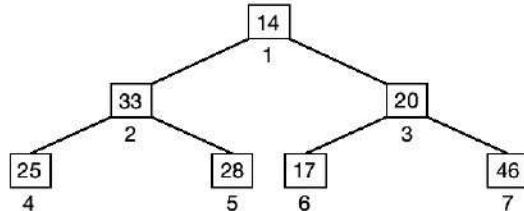
nível =  $\log_2 i + 1 = 3 \rightarrow$  nível ímpar  $\rightarrow$  nível mínimopai =  $i/2 = 6/2 = 3$ verifica se a posição do novo nº possui pai  $\rightarrow$  pai  $>= 1$  $3 >= 1 \rightarrow V \rightarrow$  verifica se o novo nº é maior que seu pai $heap(i) > heap(pai) \rightarrow 14 > 20 \rightarrow F \rightarrow$  verifica se o novo nº possui avôavo =  $i/4 = 6/4 = 1$ verifica se a posição i possui avô  $\rightarrow$  avo  $>= 1$  $1 >= 1 \rightarrow V \rightarrow$  verifica se o novo nº é menor que seu avô $heap(i) > heap(avô) \rightarrow 14 < 17 \rightarrow V \rightarrow$  troca e atualiza i para a posição do avô, i = 1

| heap | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | 10 | i | 1 | num | 14 |
|------|----|----|----|----|----|----|---|---|---|---|----|---|---|-----|----|
|      | 14 | 33 | 20 | 25 | 28 | 17 |   |   |   |   |    |   |   |     |    |

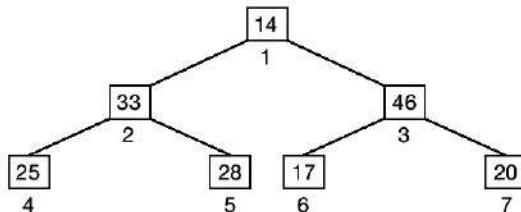
avo =  $i/4 = 1/4 = 0$ verifica se a posição i possui avô  $\rightarrow$  avo  $>= 1$  $0 >= 1 \rightarrow F \rightarrow$  novo nº permanece na posição onde está8<sup>a</sup> operação

Inserção do nº 46 na lista de prioridades (heap min-max)

| heap | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | 10 | i | 7 | num | 46 |
|------|----|----|----|----|----|----|----|---|---|---|----|---|---|-----|----|
|      | 14 | 33 | 20 | 25 | 28 | 17 | 46 |   |   |   |    |   |   |     |    |

nível =  $\log_2 i + 1 = 3 \rightarrow$  nível ímpar  $\rightarrow$  nível mínimopai =  $i/2 = 7/2 = 3$ verifica se a posição do novo nº possui pai  $\rightarrow$  pai  $>= 1$  $3 >= 1 \rightarrow V \rightarrow$  verifica se o novo nº é maior que seu pai $heap(i) > heap(pai) \rightarrow 46 > 20 \rightarrow V \rightarrow$  troca e atualiza i para a posição do pai, i = 3

|      |   |   |   |   |   |   |   |   |   |   |    |   |   |     |    |
|------|---|---|---|---|---|---|---|---|---|---|----|---|---|-----|----|
| heap | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | i | 3 | num | 46 |
|------|---|---|---|---|---|---|---|---|---|---|----|---|---|-----|----|

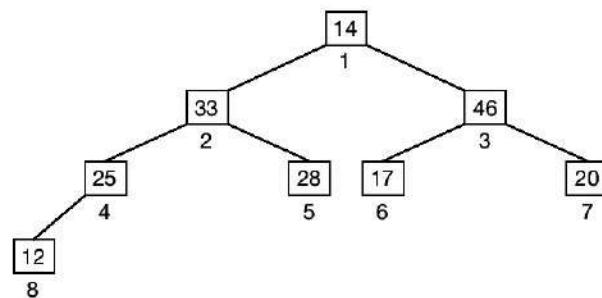


avo =  $i/4 = 3/4 = 0$   
verifica se a posição i possui avô  $\Rightarrow$  avo >= 1  
 $0 >= 1 \Rightarrow F \Rightarrow$  novo nº permanece na posição onde está

#### 9ª operação

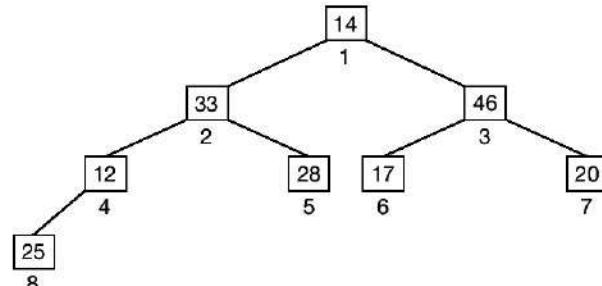
Inserção do nº 12 na lista de prioridades (heap min-max)

|      |   |   |   |   |   |   |   |   |   |   |    |   |   |     |    |
|------|---|---|---|---|---|---|---|---|---|---|----|---|---|-----|----|
| heap | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | i | 8 | num | 12 |
|------|---|---|---|---|---|---|---|---|---|---|----|---|---|-----|----|



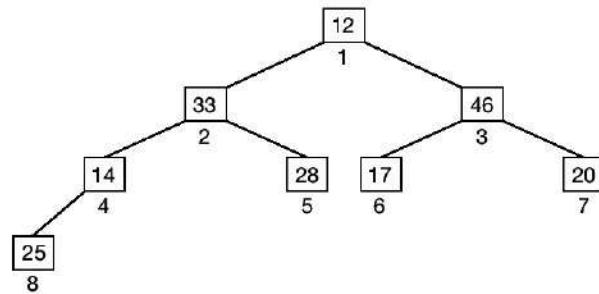
nível =  $\log_2 i + 1 = 4 \Rightarrow$  nível par  $\Rightarrow$  nível máximo  
pai =  $i/2 = 8/2 = 4$   
verifica se a posição do novo nº possui pai  $\Rightarrow$  pai >= 1  
 $4 >= 1 \Rightarrow V \Rightarrow$  verifica se o novo nº é menor que seu pai  
 $heap(i) < heap(pai) \Rightarrow 12 < 25 \Rightarrow V \Rightarrow$  troca e atualiza i para a posição do pai, i = 4

|      |   |   |   |   |   |   |   |   |   |   |    |   |   |     |    |
|------|---|---|---|---|---|---|---|---|---|---|----|---|---|-----|----|
| heap | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | i | 4 | num | 12 |
|------|---|---|---|---|---|---|---|---|---|---|----|---|---|-----|----|



avo =  $i/4 = 4/4 = 1$   
verifica se a posição i possui avô  $\Rightarrow$  avo >= 1  
 $1 >= 1 \Rightarrow V \Rightarrow$  verifica se o novo nº é menor que seu avô  
 $heap(i) < heap(avo) \Rightarrow 12 < 14 \Rightarrow V \Rightarrow$  troca e atualiza i para a posição do avô, i = 1

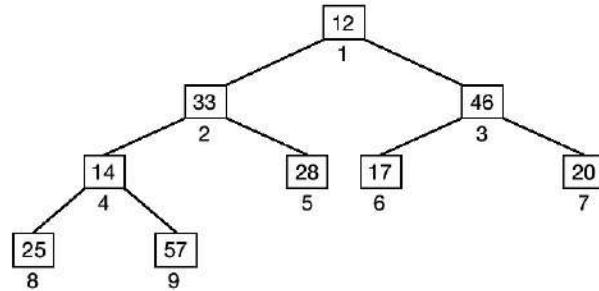
|      |   |   |   |   |   |   |   |   |   |   |    |   |   |     |    |
|------|---|---|---|---|---|---|---|---|---|---|----|---|---|-----|----|
| heap | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | i | 1 | num | 12 |
|------|---|---|---|---|---|---|---|---|---|---|----|---|---|-----|----|



$avo = i/4 = 1/4 = 0$   
 verifica se a posição i possui avô  $\Rightarrow avo \geq 1$   
 $0 \geq 1 \Rightarrow F \Rightarrow$  novo nº permanece na posição onde está

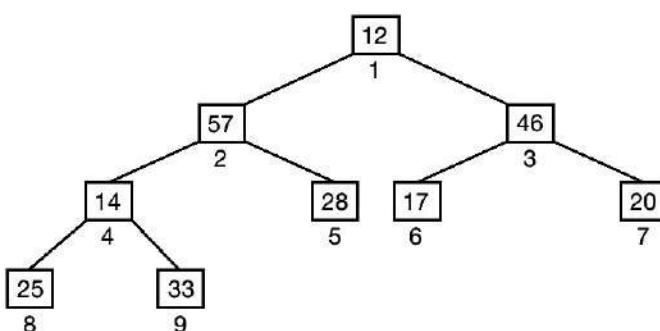
**10ª operação**  
**Inserção do nº 57 na lista de prioridades (heap min-max)**

| heap | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | i | num |
|------|---|----|----|----|----|----|----|----|----|----|----|---|-----|
|      |   | 12 | 33 | 46 | 14 | 28 | 17 | 20 | 25 | 57 |    | 9 | 57  |



$nível = \log_2 i + 1 = 4 \Rightarrow nível par \Rightarrow nível máximo$   
 $pai = i/2 = 9/2 = 4$   
 verifica se a posição do novo nº possui pai  $\Rightarrow pai \geq 1$   
 $4 \geq 1 \Rightarrow V \Rightarrow$  verifica se o novo nº é menor que seu pai  
 $heap(i) > heap(pai) \Rightarrow 57 < 14 \Rightarrow F \Rightarrow$  verifica se o novo nº possui avô  
 $avo = i/4 = 9/4 = 2$   
 verifica se a posição i possui avô  $\Rightarrow avo \geq 1$   
 $2 \geq 1 \Rightarrow V \Rightarrow$  verifica se o novo nº é maior que seu avô  
 $heap(i) > heap(avo) \Rightarrow 57 > 33 \Rightarrow V \Rightarrow$  troca e atualiza i para a posição do avô,  $i = 2$

| heap | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | i | num |
|------|---|----|----|----|----|----|----|----|----|----|----|---|-----|
|      |   | 12 | 57 | 46 | 14 | 28 | 17 | 20 | 25 | 33 |    | 2 | 57  |

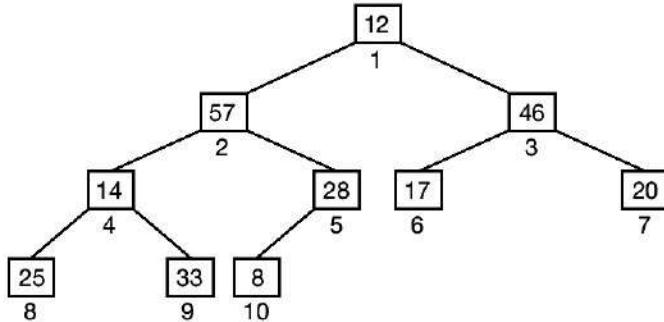


$avo = i/4 = 2/4 = 0$   
 verifica se a posição  $i$  possui avô  $\Rightarrow avo >= 1$   
 $0 >= 1 \Rightarrow F \Rightarrow$  novo nº permanece na posição onde está

### 11ª operação

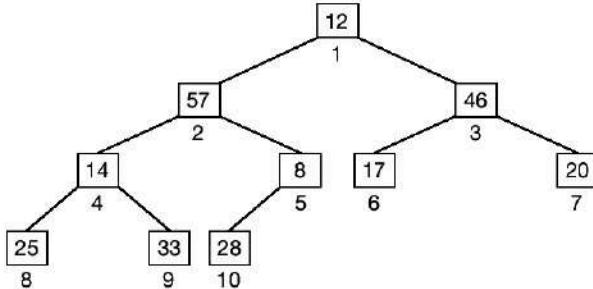
#### Inserção do nº 8 na lista de prioridades (heap min-max)

| heap | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | i | num |
|------|---|----|----|----|----|----|----|----|----|----|----|---|-----|
|      |   | 12 | 57 | 46 | 14 | 28 | 17 | 20 | 25 | 33 | 8  |   |     |



nível =  $\log_2 i + 1 = 4 \Rightarrow$  nível par  $\Rightarrow$  nível máximo  
 pai =  $i/2 = 10/2 = 5$   
 verifica se a posição do novo nº possui pai  $\Rightarrow$  pai  $>= 1$   
 $5 >= 1 \Rightarrow V \Rightarrow$  verifica se o novo nº é menor que seu pai  
 $heap(i) < heap(pai) \Rightarrow 8 < 28 \Rightarrow V \Rightarrow$  troca e atualiza i para a posição do pai,  $i = 5$

| heap | 0 | 1  | 2  | 3  | 4  | 5 | 6  | 7  | 8  | 9  | 10 | i | num |
|------|---|----|----|----|----|---|----|----|----|----|----|---|-----|
|      |   | 12 | 57 | 46 | 14 | 8 | 17 | 20 | 25 | 33 | 28 |   |     |



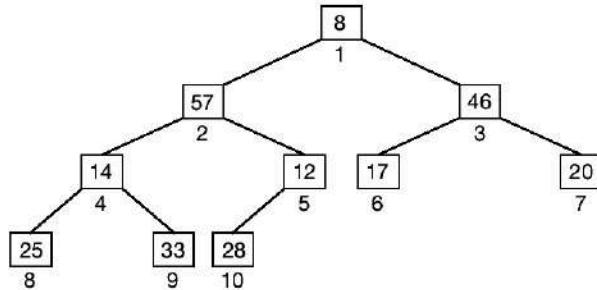
$$\text{avo} = i/4 = 5/4 = 1$$

verifica se a posição  $i$  possui avô  $\rightarrow \text{avo} \geq 1$

$1 \geq 1 \rightarrow V \rightarrow$  verifica se o novo nº é menor que seu avô

$\text{heap}(i) < \text{heap}(\text{avo}) \rightarrow 8 < 12 \rightarrow V \rightarrow$  troca e atualiza  $i$  para a posição do avô,  $i=1$

| 0    | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | i | num |
|------|---|----|----|----|----|----|----|----|----|----|---|-----|
| heap | 8 | 57 | 46 | 14 | 12 | 17 | 20 | 25 | 33 | 28 | 1 | 8   |



$$\text{avo} = i/4 = 1/4 = 0$$

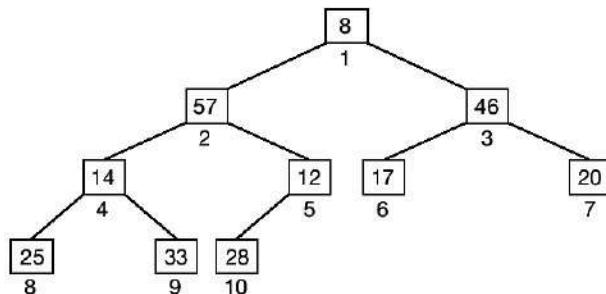
verifica se a posição  $i$  possui avô  $\rightarrow \text{avo} \geq 1$

$0 \geq 1 \rightarrow F \rightarrow$  novo nº permanece onde está

## 12ª operação

Remoção do elemento de maior prioridade da lista de prioridades (heap min-max)

| 0    | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | tam |
|------|---|----|----|----|----|----|----|----|----|----|-----|
| heap | 8 | 57 | 46 | 14 | 12 | 17 | 20 | 25 | 33 | 28 | 10  |



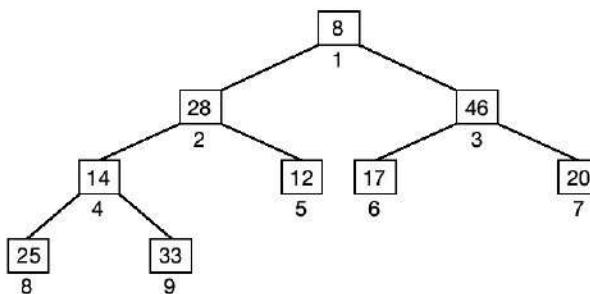
estrutura *heap min-max*  $\rightarrow$  o elemento de maior prioridade está na posição 2 ou 3  
verifica qual o maior elemento  $\rightarrow \text{heap}(2) > \text{heap}(3) \rightarrow 57 > 46 \rightarrow V$

elemento da posição 2 será removido

elemento da última posição ocupada será copiado para a posição 2 e será descartado

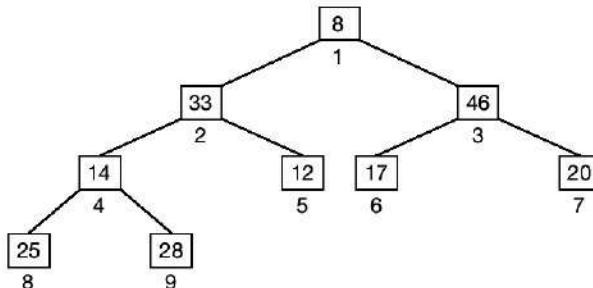
analisar as prioridades *heap* para o novo valor da posição 2

| 0    | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | tam | i |
|------|---|----|----|----|----|----|----|----|----|----|-----|---|
| heap | 8 | 28 | 46 | 14 | 12 | 17 | 20 | 25 | 33 |    | 9   | 2 |



nível =  $\log_2 i + 1 = 2 \Rightarrow$  nível par  $\Rightarrow$  nível máximo  
 verifica se o elemento da posição  $i$  tem descendentes  $\Rightarrow 2 * i <= \text{tam}$   
 $2 * 2 <= 9 \Rightarrow V \Rightarrow$  o elemento da posição  $i$  tem descendentes  
 encontrar a posição do maior elemento entre os descendentes (filhos e netos)  
 $m = \text{posição do maior descendente do elemento da posição } i$   
 $m = 9 \Rightarrow$  compara o elemento da posição  $i$  com o elemento da posição  $m$   
 $\text{heap}(m) > \text{heap}(i) \Rightarrow \text{heap}(9) > \text{heap}(2) \Rightarrow 33 > 28 \Rightarrow V \Rightarrow$  troca

|      |   |   |    |    |    |    |    |    |    |    |    |     |   |
|------|---|---|----|----|----|----|----|----|----|----|----|-----|---|
| heap | 0 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | tam | i |
|      |   | 8 | 33 | 46 | 14 | 12 | 17 | 20 | 25 | 28 |    | 9   | 2 |

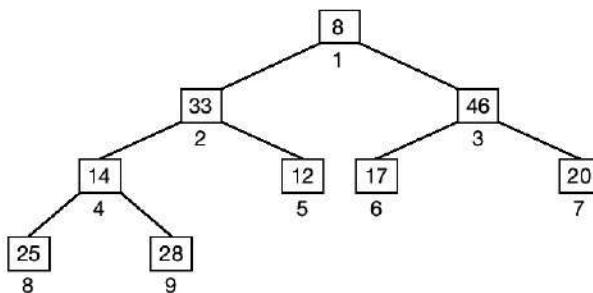


verifica se elemento da posição  $m$  é neto  $\Rightarrow m >= 4 * i$   
 $9 >= 4 * 2 \Rightarrow V \Rightarrow$  é neto, verificar as propriedades  $\text{heap}$  com o pai  
 pai =  $m/2 = 9/2 = 4$   
 $\text{heap}(\text{pai}) > \text{heap}(m) \Rightarrow \text{heap}(4) > \text{heap}(9)$   
 $14 > 28 \Rightarrow F \Rightarrow$  elementos permanecem onde estão

### 13ª operação

Remoção do elemento de menor prioridade da lista de prioridades ( $\text{heap min-max}$ )

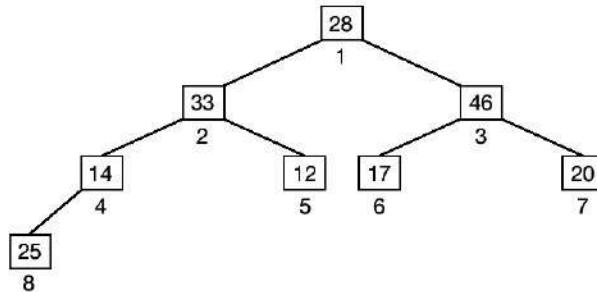
|      |   |   |    |    |    |    |    |    |    |    |    |     |
|------|---|---|----|----|----|----|----|----|----|----|----|-----|
| heap | 0 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | tam |
|      |   | 8 | 33 | 46 | 14 | 12 | 17 | 20 | 25 | 28 |    | 9   |



estrutura  $\text{heap min\_max}$   $\Rightarrow$  o elemento de menor prioridade está na posição 1  
 elemento da posição 1 será removido  
 elemento da última posição ocupada será copiado para a posição 1 e será descartado  
 analisar as propriedades  $\text{heap}$  para o novo valor da posição 1

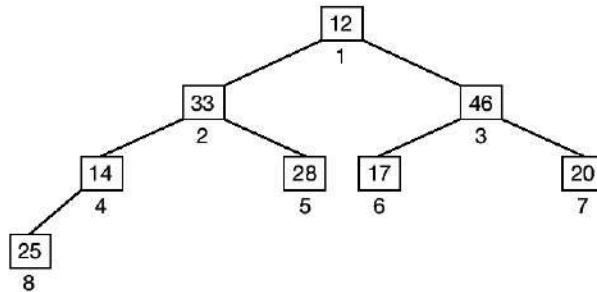
## 226 Estruturas de dados

|             |    |    |    |    |    |    |    |    |   |   |    |            |          |
|-------------|----|----|----|----|----|----|----|----|---|---|----|------------|----------|
| <i>heap</i> | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 | 10 | <i>tam</i> | <i>i</i> |
|             | 28 | 33 | 46 | 14 | 12 | 17 | 20 | 25 |   |   |    | 8          | 1        |



nível =  $\log_2 i + 1 = 1 \Rightarrow$  nível ímpar  $\Rightarrow$  nível mínimo  
 verifica se o elemento da posição *i* tem descendentes  $\Rightarrow 2 * i <= \text{tam}$   
 $2 * 1 <= 8 \Rightarrow V \Rightarrow$  o elemento da posição *i* tem descendentes  
 encontrar a posição do menor elemento entre os descendentes (filhos e netos)  
 $m = \text{posição do menor descendente do elemento da posição } i$   
 $m = 5 \Rightarrow$  compara o elemento da posição *i* com o elemento da posição *m*  
 $\text{heap}(m) < \text{heap}(i) \Rightarrow \text{heap}(5) < \text{heap}(1) \Rightarrow 12 < 28 \Rightarrow V \Rightarrow$  troca

|             |    |    |    |    |    |    |    |    |   |   |    |            |          |
|-------------|----|----|----|----|----|----|----|----|---|---|----|------------|----------|
| <i>heap</i> | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 | 10 | <i>tam</i> | <i>i</i> |
|             | 12 | 33 | 46 | 14 | 28 | 17 | 20 | 25 |   |   |    | 8          | 1        |

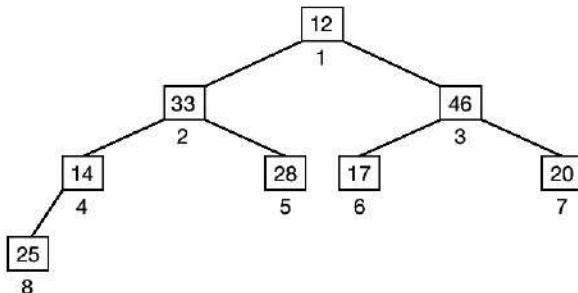


verifica se o elemento da posição *m* é neto  $\Rightarrow m >= 4 * i$   
 $5 >= 4 * 1 \Rightarrow V \Rightarrow$  é neto, verificar as propriedades *heap* com o pai  
 pai =  $m/2 = 5/2 = 2$   
 $\text{heap}(\text{pai}) < \text{heap}(m) \Rightarrow \text{heap}(2) < \text{heap}(5)$   
 $33 < 28 \Rightarrow F \Rightarrow$  elementos permanecem onde estão

### 14ª operação

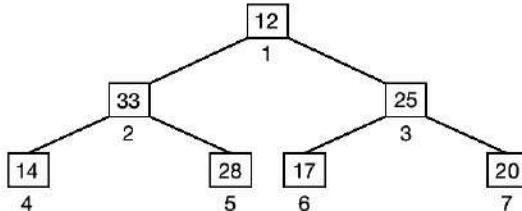
#### Remoção do elemento de maior prioridade da lista de prioridades (*heap min-max*)

|             |    |    |    |    |    |    |    |    |   |   |    |            |
|-------------|----|----|----|----|----|----|----|----|---|---|----|------------|
| <i>heap</i> | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 | 10 | <i>tam</i> |
|             | 12 | 33 | 46 | 14 | 28 | 17 | 20 | 25 |   |   |    | 8          |



estrutura heap min\_max → o elemento de maior prioridade está na posição 2 ou 3  
 verifica qual o maior elemento →  $\text{heap}(2) > \text{heap}(3)$  →  $33 > 25 \Rightarrow F$   
 elemento da posição 3 será removido  
 elemento da última posição ocupada será copiado para a posição 3 e será descartado  
 analisar as propriedades heap para o novo valor da posição 3

|      |   |    |    |    |    |    |    |    |   |   |    |     |   |   |   |
|------|---|----|----|----|----|----|----|----|---|---|----|-----|---|---|---|
| heap | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 | 10 | tam | 7 | i | 3 |
|      |   | 12 | 33 | 25 | 14 | 28 | 17 | 20 |   |   |    |     |   |   |   |

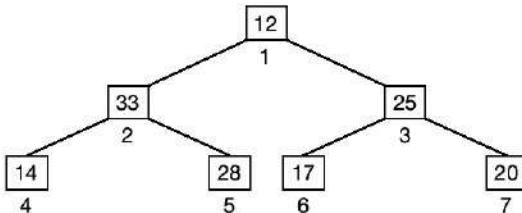


nivel =  $\log_2 i + 1 = 2 \Rightarrow$  nível par → nível máximo  
 verifica se o elemento da posição i tem descendentes →  $2 * i <= \text{tam}$   
 $2 * 3 <= 7 \Rightarrow V$  → o elemento da posição i tem descendentes  
 encontrar a posição do maior elemento entre os descendentes (filhos e netos)  
 m = posição do maior descendente do elemento da posição i  
 m = 7 → compara o elemento da posição i com o elemento da posição m  
 $\text{heap}(m) > \text{heap}(i) \Rightarrow \text{heap}(7) > \text{heap}(3)$   
 $20 > 25 \Rightarrow F$  → elementos permanecem onde estão

### 15ª operação

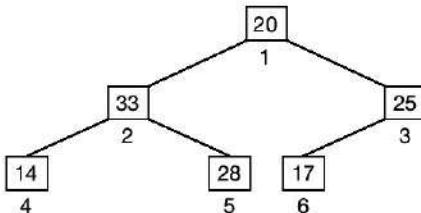
#### Remoção do elemento de menor prioridade da lista de prioridades (heap min-max)

|      |   |    |    |    |    |    |    |    |   |   |    |     |   |  |
|------|---|----|----|----|----|----|----|----|---|---|----|-----|---|--|
| heap | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 | 10 | tam | 7 |  |
|      |   | 12 | 33 | 25 | 14 | 28 | 17 | 20 |   |   |    |     |   |  |



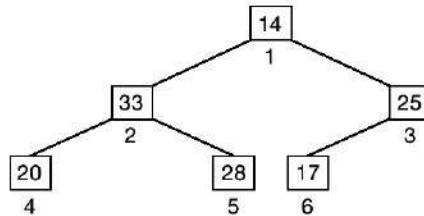
estrutura heap min\_max → o elemento de menor prioridade está na posição 1  
 elemento da posição 1 será removido  
 elemento da última posição ocupada será copiado para a posição 1 e será descartado  
 analisar as propriedades heap para o novo valor da posição 1

|      |   |    |    |    |    |    |    |   |   |   |    |     |   |   |   |
|------|---|----|----|----|----|----|----|---|---|---|----|-----|---|---|---|
| heap | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | 10 | tam | 6 | i | 1 |
|      |   | 20 | 33 | 25 | 14 | 28 | 17 |   |   |   |    |     |   |   |   |



nivel =  $\log_2 i + 1 = 1 \Rightarrow$  nível ímpar → nível mínimo  
 verifica se o elemento da posição i tem descendentes →  $2 * i <= \text{tam}$   
 $2 * 1 <= 6 \Rightarrow V$  → o elemento da posição i tem descendentes  
 encontrar a posição do menor elemento entre os descendentes (filhos e netos)  
 m = posição do menor descendente do elemento da posição i  
 m = 4 → compara o elemento da posição i com o elemento da posição m  
 $\text{heap}(m) < \text{heap}(i) \Rightarrow \text{heap}(4) < \text{heap}(1) \Rightarrow 14 < 20 \Rightarrow V$  → troca

|             |   |    |    |    |    |    |    |   |   |   |    |            |          |
|-------------|---|----|----|----|----|----|----|---|---|---|----|------------|----------|
| <i>heap</i> | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | 10 | <i>tam</i> | <i>i</i> |
|             |   | 14 | 33 | 25 | 20 | 28 | 17 |   |   |   |    | 6          | 1        |



verifica se o elemento da posição m é neto  $\rightarrow m >= 4 * i$   
 $4 >= 4 * 1 \rightarrow V \rightarrow$  é neto, verificar as propriedades heap com o pai  
 $pai = m/2 = 4/2 = 2$   
 $heap(pai) < heap(m) \rightarrow heap(2) < heap(4)$   
 $33 < 20 \rightarrow F \rightarrow$  elementos permanecem onde estão



```

import java.util.Scanner;
public class Heap_Min_Max
{
 // declarando o vetor com capacidade de no máximo 10
 // números
 static int vet[] = new int[11];
 static int tam;

 public static void main(String[] args)
 {
 Scanner entrada = new Scanner(System.in);
 int op, mp, num;
 tam = 0;
 do
 {
 System.out.println("\nMENU DE OPÇÕES - HEAP Min_
 → Max\n");
 System.out.println("1 - Inserir elemento na lista de
 prioridades");
 System.out.println("2 - Consultar o elemento de
 → menor prioridade");
 System.out.println("3 - Consultar o elemento de
 → maior prioridade");
 System.out.println("4 - Remover o elemento de
 → menor prioridade");
 System.out.println("5 - Remover o elemento de
 → maior prioridade");
 System.out.println("6 - Consultar toda a lista");
 }
 }
}

```

```
System.out.println("7 - Sair");
System.out.print("Digite sua opção: ");
op = entrada.nextInt();

if (op < 1 || op > 7)
 System.out.println("Opção inválida!!");

else if (op==1)
{
 // verifica se ainda existe espaço disponível
 ↪ no vetor
 // para inserção do novo número
 if (tam <= vet.length)
 {
 System.out.print("Digite um número: ");
 // leitura do número a ser inserido
 num = entrada.nextInt();
 tam++;
 inserir_mm(num, tam);
 System.out.println("Número inserido");
 }
 else System.out.println("Lista de prioridades
 ↪ Lotada!");
}

else if(op==2)
{
 if(tam==0)
 System.out.println("Lista de prioridades
 ↪ vazia!");
 else
 System.out.println("Elemento de menor
 ↪ prioridade:" +vet[1]);
}

else if(op==3)
{
 if(tam==0)
 System.out.println("Lista de prioridades
 ↪ vazia!");
 else
 {
 mp = maior_prior();
 System.out.println("Elemento de maior
```

```
 ↪ prioridade: " +vet[mp]);
 }
}

else if (op==4)
{
if(tam==0)
 System.out.println("Lista de prioridades
 ↪ vazia!");
else
{
 System.out.println("O elemento removido:
 ↪ "+vet[1]);
 vet[1] = vet[tam];
 tam--;
 descer(1);
}
}

else if (op==5)
{
if(tam==0)
 System.out.println("Lista de prioridades
 ↪ vazia!");
else if(tam==2)
 tam = 1;
else
{
 int max = 2;
 if(tam >= 3)
 {
 if(vet[3]>vet[2])
 max = 3;
 }
 System.out.println("O elemento removido: "
 "+vet[max]);
 vet[max] = vet[tam];
 tam--;
 descer(max);
}
}

else if(op==6)
impressao();

}
```

```
 while(op!=7);
}

static void inserir_mm(int num, int i)
{
 vet[i] = num;
 subir(i);
}

static void subir(int i)
{
 int pai = i/2; // é o índice do possível pai

 // função 'mínimo'
// verifica se o nó i encontra-se em um nível mínimo
if(minimo(i))
{
 // verifica se o elemento de nível mínimo
 // é maior que o pai
 if(pai>=1)
 {
 if(vet[i] > vet[pai])
 {
 trocar(i, pai);
 subir_max(pai);
 }
 else
 subir_min(i);
 }
}
else
{
 // verifica se o elemento de nível máximo
 // é menor que o pai
 if(pai>=1)
 {
 if(vet[i] < vet[pai])
 {
 trocar(i, pai);
 subir_min(pai);
 }
 else
 subir_max(i);
 }
}
```

```
}
```

```
static void subir_min(int i)
{
 int avo = i/4; // índice do possível avô

 // verifica se o elemento é menor que o avô
 if(avo >= 1 && vet[i] < vet[avo])
 {
 trocar(i, avo);
 subir_min(avo);
 }
}
```

```
static void subir_max(int i)
{
 int avo = i/4; // índice do possível avô
 // verifica se o elemento é maior que o avô
 if(avo >= 1 && vet[i] > vet[avo])
 {
 trocar(i, avo);
 subir_max(avo);
 }
}
```

```
static int maior_prior()
{
 if(tam==1)
 return 1;
 else if(tam > 2 && vet[3]> vet[2])
 return 3;
 else
 return 2;
}
```

```
static void descer(int i)
{
 if(minimo(i))
 descer_min(i);
 else
 descer_max(i);
}
```

```
static boolean minimo(int i)
```

```
{
 int nivel = ((int)(Math.log(i)/Math.log(2))) + 1;
 if(nivel % 2 == 0)
 return false;
 else
 return true;
}

static void descer_min(int i)
{
 if(2*i <= tam) // i tem filhos
 {
 // menor dos descendentes entre filhos e netos
 int m = min_descendente(i);
 if(vet[i] > vet[m])
 {
 trocar(i, m);

 if(m >= 4*i)
 {
 int p = m/2; // p é o pai
 if(vet[p] < vet[m])
 trocar(p,m);

 descer_min(m);
 }
 }
 }
}

static int min_descendente(int i)
{
 // retorna índice do menor dos descendentes
 // entre filhos e netos do elemento i

 int m=0; // índice do menor elemento
 if(2*i <= tam)
 {
 // índice do menor (primeiro filho)
 m = 2*i;
 // verifica o menor filho
 if(vet[m+1] < vet[m])
 m = m+1;
 }
}
```

```
// verifica o menor neto
for(int k=4*i; (k<=4*i+3)&& k <= tam; k++)
 if(vet[k] < vet[m])
 m = k;
}
return m;
}

static void descer_max(int i)
{
 // i tem filhos
 if(2*i <= tam)
 {
 int m = max_descendente(i);
 if(vet[i] < vet[m])
 {
 trocar(i, m);

 if(m >= 4*i)
 {
 int p = m/2; // p é o pai
 if(vet[p] > vet[m])
 trocar(p,m);

 descer_max(m);
 }
 }
 }
}

static int max_descendente(int i)
{
 // retorna índice do maior dos descendentes
 // entre filhos e netos do elemento i

 int m=0; // índice do maior elemento
 if(2*i <= tam)
 {
 m = 2*i; // índice do maior (primeiro filho)
 if(vet[m+1] > vet[m])
 m = m+1;
 // índice do neto
 for(int k=4*i; (k<=4*i+3)&& k <= tam; k++)
 if(vet[k] > vet[m])
```

```

 m = k;
 }
 return m;
}

static void impressao()
{
 if(tam == 0)
 System.out.println("Lista de prioridades vazia");
 else
 {
 System.out.println("\nTodos os elementos da
 lista de prioridades\n");
 for(int j=1; j <= tam; j++)
 System.out.print(vet[j]+ " ");
 System.out.println();
 }
}

static void trocar(int x, int y)
{
 // função que troca o valor
 // entre duas posições do vetor (heap)
 int temp;
 temp = vet[x];
 vet[x] = vet[y];
 vet[y] = temp;
}
}

```




---

```

#include<iostream.h>
#include<conio.h>
#include<math.h>

int minimo(int i)
{
 int nivel = ((int)(log(i)/log(2))) + 1;
 if(nivel % 2 == 0)
 return 0;
 else
 return 1;

```

```
}
```

```
void trocar(int vet[], int x, int y)
{
 // função que troca o valor
 // entre duas posições do vetor (heap)
 int temp;
 temp = vet[x];
 vet[x] = vet[y];
 vet[y] = temp;
}
void subir_max(int vet[], int i)
{
 int avo = i/4; // índice do possível avô

 // verifica se o elemento é maior que o avô
 if(avo >= 1 && vet[i] > vet[avo])
 {
 trocar(vet, i, avo);
 subir_max(vet, avo);
 }
}

void subir_min(int vet[], int i)
{
 int avo = i/4; // índice do possível avô

 // verifica se o elemento é menor que o avô
 if(avo >= 1 && vet[i] < vet[avo])
 {
 trocar(vet, i, avo);
 subir_min(vet, avo);
 }
}

void subir(int vet[], int i)
{
 int pai = i/2; // é o índice do possível pai

 // função 'minimo'
 // verifica se o nó i encontra-se em um nível mínimo
 if(minimo(i))
 {
 // verifica se o elemento de nível mínimo
 // é maior que o pai
 if(pai>=1)
```

```
{
 if(vet[i] > vet[pai])
 {
 trocar(vet, i, pai);
 subir_max(vet, pai);
 }
 else
 subir_min(vet, i);
}
}
else
{
 // verifica se o elemento de nível máximo
 // é menor que o pai
 if(pai>=1)
 {
 if(vet[i] < vet[pai])
 {
 trocar(vet, i, pai);
 subir_min(vet, pai);
 }
 else
 subir_max(vet, i);
 }
}
}

void inserir_mm(int vet[], int num, int i)
{
 vet[i] = num;
 subir(vet, i);
}

int maior_prior(int vet[], int tam)
{
 if(tam==1)
 return 1;
 else if(tam > 2 && vet[3]> vet[2])
 return 3;
 else
 return 2;
}
```

```
int min_descendente(int vet[], int i, int tam)
{
 // retorna índice do menor dos descendentes
 // entre filhos e netos do elemento i

 int m=0; // índice do menor elemento
 if(2*i <= tam)
 {
 // índice do menor (primeiro filho)
 m = 2*i;

 // verifica o menor filho
 if(vet[m+1] < vet[m])
 m = m+1;

 // verifica o menor neto
 for(int k=4*i; (k<=4*i+3)&& k <= tam; k++)
 if(vet[k] < vet[m])
 m = k;
 }
 return m;
}

void descer_min(int vet[], int i, int tam)
{
 if(2*i <= tam) // i tem filhos
 {
 // menor dos descendentes entre filhos e netos
 int m = min_descendente(vet, i, tam);
 if(vet[i] > vet[m])
 {
 trocar(vet, i, m);

 if(m >= 4*i)
 {
 int p = m/2; // p é o pai
 if(vet[p] < vet[m])
 trocar(vet,p,m);

 descer_min(vet, m, tam);
 }
 }
 }
}
```

```
int max_descendente(int vet[], int i, int tam)
{
 // retorna índice do maior dos descendentes
 // entre filhos e netos do elemento i

 int m=0; // índice do maior elemento
 if(2*i <= tam)
 {
 m = 2*i; // índice do maior (primeiro filho)
 if(vet[m+1] > vet[m])
 m = m+1;

 // índice do neto
 for(int k=4*i; (k<=4*i+3)&& k <= tam; k++)
 if(vet[k] > vet[m])
 m = k;
 }
 return m;
}

void descer_max(int vet[], int i, int tam)
{
 // i tem filhos
 if(2*i <= tam)
 {
 int m = max_descendente(vet, i, tam);
 if(vet[i] < vet[m])
 {
 trocar(vet, i, m);

 if(m >= 4*i)
 {
 int p = m/2; // p é o pai
 if(vet[p] > vet[m])
 trocar(vet, p, m);

 descer_max(vet, m, tam);
 }
 }
 }
}

void descer(int vet[], int i, int tam)
{
 if(minimo(i))
```

```
 descer_min(vet, i, tam);
 else
 descer_max(vet, i, tam);
}

void impressao(int vet[], int tam)
{
 if(tam == 0)
 cout << "\nLista de prioridades vazia";
 else
 {
 cout << "\nTodos os elementos da lista de
 prioridades\n";
 for(int j=1; j <= tam; j++)
 cout << vet[j] << " ";
 cout << "\n";
 }
}

void main()
{
 // declarando o vetor com capacidade de no máximo
 // 10 números
 int vet[11];
 int tam;
 int op, mp, num;

 tam = 0;

 do
 { clrscr();
 cout << "\nMENU DE OPÇÕES - HEAP Min_Max\n";
 cout << "\n1 - Inserir elemento na lista de
 prioridades";
 cout << "\n2 - Consultar o elemento de menor
 prioridade";
 cout << "\n3 - Consultar o elemento de maior
 prioridade";
 cout << "\n4 - Remover o elemento de menor prioridade";
 cout << "\n5 - Remover o elemento de maior prioridade";
 cout << "\n6 - Consultar toda a lista";
```

```
cout << "\n7 - Sair";
cout << "\nDigite sua opção: ";
cin >> op;

if (op < 1 || op > 7)
 cout << "\nOpção inválida!!";
else if (op==1)
{
 // verifica se ainda existe espaço disponível
 // no vetor
 // para inserção do novo número
 if (tam <= 9)
 {
 cout << "Digite um número: ";
 // leitura do número a ser inserido
 cin >> num;
 tam++;
 inserir_mm(vet, num, tam);
 cout << "\nNúmero inserido";
 }
 else cout << "\nLista de prioridades Lotada!";
}
else if (op==2)
{
 if (tam==0)
 cout << "\nLista de prioridades vazia!";
 else
 cout << "Elemento de menor prioridade:" << vet[1];
}
else if (op==3)
{
 if (tam==0)
 cout << "Lista de prioridades vazia!";
 else
 {
 mp = maior_prior(vet, tam);
 cout << "Elemento de maior prioridade:
 << vet[mp];
 }
}
else if (op==4)
{
 if (tam==0)
 cout << "\nLista de prioridades vazia!";
}
else
```

```

 cout << "\nO elemento removido: "<<vet[1];
 vet[1] = vet[tam];
 tam--;
 descer(vet, 1, tam);
 }
}
else if (op==5)
{
 if(tam==0)
 cout << "\nLista de prioridades vazia!";
 else if(tam==2)
 tam = 1;
 else
 {
 int max = 2;
 if(tam >= 3)
 {
 if(vet[3]>vet[2])
 max = 3;
 }
 cout << "\nO elemento removido: "<<vet[max];
 vet[max] = vet[tam];
 tam--;
 descer(vet, max, tam);
 }
}
else if(op==6)
 impressao(vet, tam);

getch();
}
while(op!=7);
getch();
}

```

---

## Análise da complexidade da estrutura

### heap max-min e heap min-max

As operações de consulta realizadas no *heap max-min* gastam tempo constante,  $O(1)$ , pois para acessar o elemento de maior prioridade é necessário apenas acessar o primeiro elemento do vetor. Já o de menor prioridade se encontra na segunda ou terceira posição do vetor. Logo, é necessário apenas verificar qual das duas posições possui o menor elemento.

As operações de consulta realizadas no *heap min-max* também gastam tempo constante,  $O(1)$ , pois para acessar o elemento de menor prioridade, é necessário apenas acessar o primeiro elemento do vetor. Já o de maior prioridade se encontra na segunda ou terceira posição do vetor. Logo, é necessário apenas verificar qual das duas posições possui o maior elemento.

Na operação de inserção em um *heap min-max*, o novo elemento é incluído sempre na primeira posição vazia. Em seguida, um processo de "subida" é aplicado sobre o mesmo, para que possa ser colocado na posição correta, respeitando sempre o fato de que, se estiver em um nível mínimo (ímpar), deve ser menor que seus filhos; por sua vez, os seus filhos devem ser maiores que os seus netos e, se o novo elemento estiver em um nível máximo (par), deve ser maior que seu pai, e seu pai deve ser menor que seus filhos.

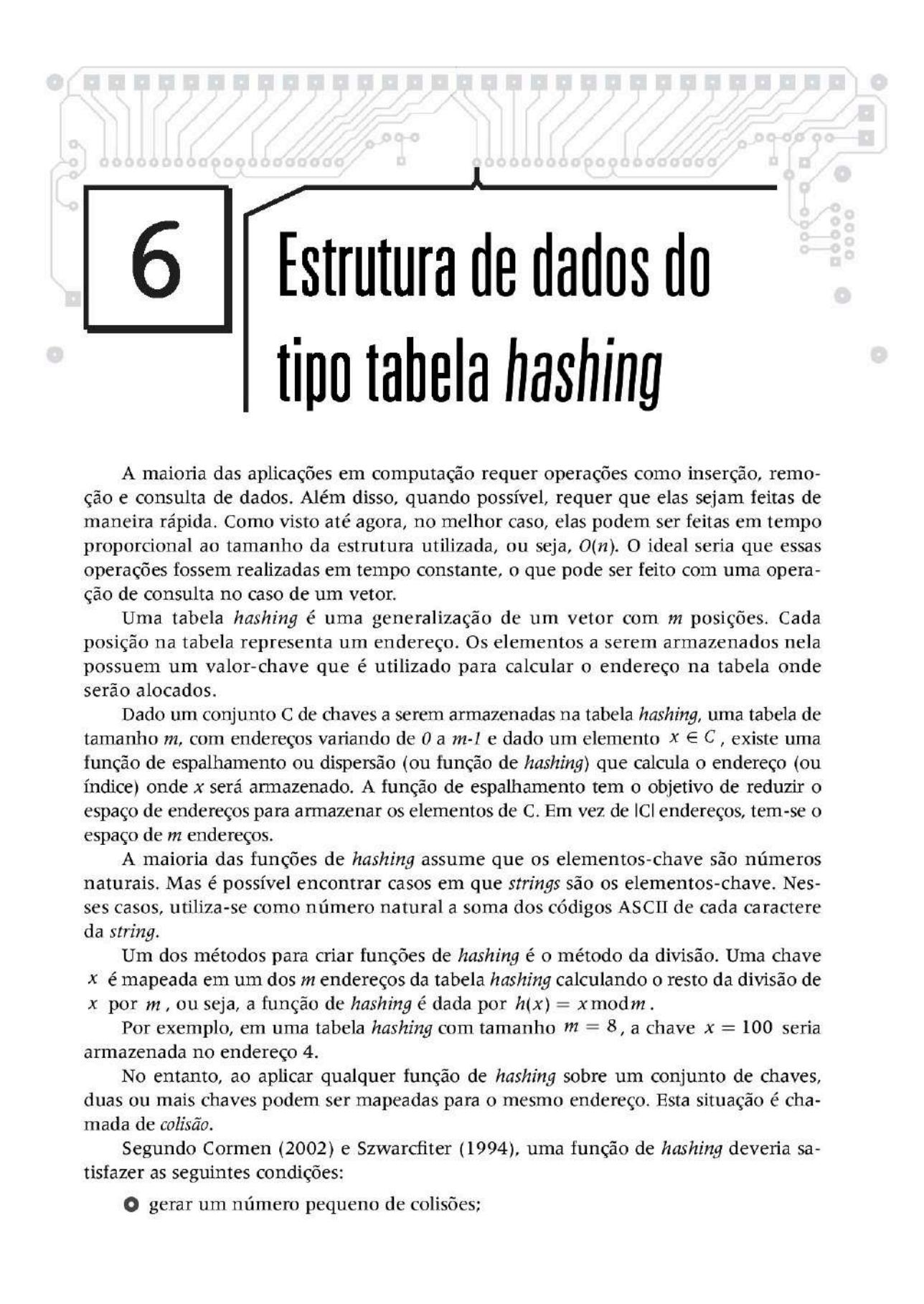
Na operação de inserção, em um *heap max-min*, o novo elemento é incluído sempre na primeira posição vazia. Em seguida, um processo de "subida" é aplicado sobre o mesmo, para que possa ser colocado na posição correta, respeitando sempre o fato de que, se estiver em um nível máximo (ímpar), deve ser maior que seus filhos; por sua vez, os seus filhos devem ser menores que os seus netos e, se o novo elemento estiver em um nível mínimo (par), deve ser menor que seu pai, e seu pai deve ser maior que seus filhos.

Assim, tanto na estrutura *heap min-max* quanto na *heap max-min*, o tempo gasto na inserção é, no pior caso,  $O(\log n)$ , pois levará o novo elemento inserido até a raiz, gastando tempo proporcional à altura da árvore.

No caso da remoção, seja do elemento de menor ou maior prioridade, o procedimento é substituir o que será removido pelo último do *heap*, e aplicar agora um processo de "descida" do elemento para que as regras que mantêm um *heap max-min* ou um *heap min-max* continuem sendo respeitadas. No pior caso, o elemento será levado até uma folha, gastando com isso a altura da árvore, que é  $O(\log n)$ .

## Exercícios

- 1) Mostre, passo a passo, as inserções, no vetor e na árvore de representação de uma lista de prioridades, dos seguintes números: 15, 21, 23, 48, 35, 70 e 67. Considere que a lista representará um *heap* máximo.
- 2) Mostre, passo a passo, a configuração do vetor e da árvore de representação da lista de prioridades do exercício 1, após realizar duas remoções.
- 3) Mostre, passo a passo, as inserções, no vetor e na árvore de representação de uma lista de prioridades, dos seguintes números: 70, 67, 35, 48, 21, 23 e 15. Considere que a lista representará um *heap* mínimo.
- 4) Mostre, passo a passo, a configuração do vetor e da árvore de representação da lista de prioridades do exercício 3, após realizar duas remoções.
- 5) Mostre, passo a passo, as inserções, no vetor e na árvore de representação de uma lista de prioridades, dos seguintes números: 70, 67, 35, 48, 21, 23, 15 e 8. Considere que a lista representará um *heap* min-max.

A decorative background image of a printed circuit board (PCB) with various electronic components like resistors, capacitors, and integrated circuits. A horizontal line extends from the top of the chapter title down to the left margin.

# 6

# Estrutura de dados do tipo tabela *hashing*

A maioria das aplicações em computação requer operações como inserção, remoção e consulta de dados. Além disso, quando possível, requer que elas sejam feitas de maneira rápida. Como visto até agora, no melhor caso, elas podem ser feitas em tempo proporcional ao tamanho da estrutura utilizada, ou seja,  $O(n)$ . O ideal seria que essas operações fossem realizadas em tempo constante, o que pode ser feito com uma operação de consulta no caso de um vetor.

Uma tabela *hashing* é uma generalização de um vetor com  $m$  posições. Cada posição na tabela representa um endereço. Os elementos a serem armazenados nela possuem um valor-chave que é utilizado para calcular o endereço na tabela onde serão alocados.

Dado um conjunto  $C$  de chaves a serem armazenadas na tabela *hashing*, uma tabela de tamanho  $m$ , com endereços variando de 0 a  $m-1$  e dado um elemento  $x \in C$ , existe uma função de espalhamento ou dispersão (ou função de *hashing*) que calcula o endereço (ou índice) onde  $x$  será armazenado. A função de espalhamento tem o objetivo de reduzir o espaço de endereços para armazenar os elementos de  $C$ . Em vez de  $|C|$  endereços, tem-se o espaço de  $m$  endereços.

A maioria das funções de *hashing* assume que os elementos-chave são números naturais. Mas é possível encontrar casos em que *strings* são os elementos-chave. Nesses casos, utiliza-se como número natural a soma dos códigos ASCII de cada caractere da *string*.

Um dos métodos para criar funções de *hashing* é o método da divisão. Uma chave  $x$  é mapeada em um dos  $m$  endereços da tabela *hashing* calculando o resto da divisão de  $x$  por  $m$ , ou seja, a função de *hashing* é dada por  $h(x) = x \bmod m$ .

Por exemplo, em uma tabela *hashing* com tamanho  $m = 8$ , a chave  $x = 100$  seria armazenada no endereço 4.

No entanto, ao aplicar qualquer função de *hashing* sobre um conjunto de chaves, duas ou mais chaves podem ser mapeadas para o mesmo endereço. Esta situação é chamada de *colisão*.

Segundo Cormen (2002) e Szwarcfiter (1994), uma função de *hashing* deveria satisfazer as seguintes condições:

- gerar um número pequeno de colisões;

- ser facilmente computável, o que significa que o tempo gasto para realizar o cálculo do endereço pela função de *hashing* deve ser o menor possível, assim como o número de acessos feitos à memória ou ao disco;
- ser *uniforme*, ou seja, a probabilidade de que o endereço calculado pela função de *hashing* seja igual a  $k$  deve ser igual para todas as chaves e todos os endereços  $k \in [0, m - 1]$ .

Como nem sempre é possível gerar um número pequeno de colisões, tenta-se, ao menos, minimizá-las, utilizando métodos de tratamento de colisão.

A colisão em uma tabela *hashing* pode ser tratada utilizando o encadeamento, criando-se para cada endereço da tabela uma lista encadeada das chaves que são mapeadas nele. Outra alternativa para tratamento de colisões é o endereçamento aberto, em que as colisões são tratadas dentro da própria tabela.

## Tabela *hashing* implementada com endereçamento aberto

Neste tipo de implementação, a tabela *hashing* é um vetor com  $m$  posições. Todas as chaves são armazenadas na própria tabela sem a necessidade de espaços extras ou ponteiros. Este método é aplicado quando o número de chaves a serem armazenadas na tabela *hashing* é reduzido e as posições vazias da tabela são utilizadas para o tratamento de colisões.

Quando uma chave  $x$  é endereçada na posição  $h(x)$  e essa já está ocupada, outras posições vazias na tabela são procuradas para armazenar  $x$ . Caso nenhuma seja encontrada, a tabela está totalmente preenchida e  $x$  não pode ser armazenada.

A busca por uma posição livre para armazenar  $x$  pode ser feita de duas maneiras: *tentativa linear* ou *tentativa quadrática*.

### Tentativa linear

Na tentativa linear, quando uma chave  $x$  deve ser inserida e ocorre uma colisão, a seguinte função é utilizada para obter um novo endereço:  $h'(x) = (h(x) + j) \bmod m$ , para  $1 \leq j \leq m - 1$ , sendo que  $h(x) = x \bmod m$ . O objetivo é armazenar a chave no endereço consecutivo  $h(x) + 1, h(x) + 2, \dots$ , até encontrar uma posição vazia.

A operação de remoção é delicada, não se pode remover de fato uma chave do endereço, pois haveria perda da sequência de tentativas. Com isso, cada endereço da tabela é marcado como livre (L), ocupado (O) ou removido (R). Livre quando a posição ainda não foi utilizada, ocupado quando uma chave está armazenada, e removido quando armazena uma chave que já foi removida – sendo que uma nova chave poderá ocupar a posição marcada como removido.

A ilustração a seguir mostra uma tabela *hashing* implementada com tentativa linear. Neste capítulo, todas as ilustrações utilizam uma tabela *hashing* com 8 posições ( $\text{tam} = 8$ ).



ILUSTRAÇÃO

**1<sup>a</sup> operação**  
A tabela hashing está vazia

|   | livre chave |
|---|-------------|
| 0 | L           |
| 1 | L           |
| 2 | L           |
| 3 | L           |
| 4 | L           |
| 5 | L           |
| 6 | L           |
| 7 | L           |

**2<sup>a</sup> operação**  
Inserção do nº 16 na tabela hashing

|   | livre chave |
|---|-------------|
| 0 | L           |
| 1 | L           |
| 2 | L           |
| 3 | L           |
| 4 | L           |
| 5 | L           |
| 6 | L           |
| 7 | L           |

num      tam      i      pos  
**16**      **8**      **0**      **0**      pos = num%tam = 16%8 = 0

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].livre ≠ 'R'  
 0 < 8 E tabela[0%8].livre ≠ 'L' E tabela[0%8].livre ≠ 'R'  
 0 < 8 E tabela[0].livre ≠ 'L' E tabela[0].livre ≠ 'R'  
 0 < 8 E 'L' ≠ 'L' E 'L' ≠ 'R'  
 V E F E V = F → verifica se a posição encontrada, ou seja, i, é válida para inserção

|   | livre chave |
|---|-------------|
| 0 | O   16      |
| 1 | L           |
| 2 | L           |
| 3 | L           |
| 4 | L           |
| 5 | L           |
| 6 | L           |
| 7 | L           |

i < tam  
 0 < 8  
 V → insere o nº 16 na posição 0 e altera tabela[0].livre para 'O'

**3<sup>a</sup> operação**  
Inserção do nº 23 na tabela hashing

|   | livre chave |
|---|-------------|
| 0 | O   16      |
| 1 | L           |
| 2 | L           |
| 3 | L           |
| 4 | L           |
| 5 | L           |
| 6 | L           |
| 7 | L           |

num      tam      i      pos  
**23**      **8**      **0**      **7**      pos = num%tam = 23%8 = 7

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].livre ≠ 'R'  
 0 < 8 E tabela[7%8].livre ≠ 'L' E tabela[7%8].livre ≠ 'R'  
 0 < 8 E tabela[7].livre ≠ 'L' E tabela[7].livre ≠ 'R'  
 0 < 8 E 'L' ≠ 'L' E 'L' ≠ 'R'  
 V E F E V = F → verifica se a posição encontrada, ou seja, i, é válida para inserção

|   | livre chave |
|---|-------------|
| 0 | O   16      |
| 1 | L           |
| 2 | L           |
| 3 | L           |
| 4 | L           |
| 5 | L           |
| 6 | L           |
| 7 | O   23      |

i < tam  
 0 < 8  
 V → insere o nº 23 na posição 7 e altera tabela[7].livre para 'O'

## 4ª operação

## Inserção do nº 41 na tabela hashing

livre chave

|   |   |    |
|---|---|----|
| 0 | O | 16 |
| 1 | L |    |
| 2 | L |    |
| 3 | L |    |
| 4 | L |    |
| 5 | L |    |
| 6 | L |    |
| 7 | O | 23 |



livre chave

|   |   |    |
|---|---|----|
| 0 | O | 16 |
| 1 | O | 41 |
| 2 | L |    |
| 3 | L |    |
| 4 | L |    |
| 5 | L |    |
| 6 | L |    |
| 7 | O | 23 |

| num | tam | i | pos |
|-----|-----|---|-----|
| 41  | 8   | 0 | 1   |

$$\text{pos} = \text{num \% tam} = 41 \% 8 = 1$$

$i < \text{tam}$  E  $\text{tabela}[(\text{pos}+i)\% \text{tam}].\text{livre} \neq 'L'$  E  $\text{tabela}[(\text{pos}+i)\% \text{tam}].\text{livre} \neq 'R'$

$0 < 8$  E  $\text{tabela}[1 \% 8].\text{livre} \neq 'L'$  E  $\text{tabela}[1 \% 8].\text{livre} \neq 'R'$

$0 < 8$  E  $\text{tabela}[1].\text{livre} \neq 'L'$  E  $\text{tabela}[1].\text{livre} \neq 'R'$

$0 < 8$  E  $'L' \neq 'L'$  E  $'L' \neq 'R'$

V E F E V = F → verifica se a posição encontrada, ou seja, i, é válida para inserção

$i < \text{tam}$

$0 < 8$

V → insere o nº 41 na posição 1 e altera  $\text{tabela}[1].\text{livre}$  para 'O'

livre chave

|   |   |    |
|---|---|----|
| 0 | O | 16 |
| 1 | O | 41 |
| 2 | L |    |
| 3 | L |    |
| 4 | L |    |
| 5 | L |    |
| 6 | L |    |
| 7 | O | 23 |



livre chave

|   |   |    |
|---|---|----|
| 0 | O | 16 |
| 1 | O | 41 |
| 2 | O | 25 |
| 3 | L |    |
| 4 | L |    |
| 5 | L |    |
| 6 | L |    |
| 7 | O | 23 |

| num | tam | i | pos |
|-----|-----|---|-----|
| 25  | 8   | 0 | 1   |

$$\text{pos} = \text{num \% tam} = 25 \% 8 = 1$$

$i < \text{tam}$  E  $\text{tabela}[(\text{pos}+i)\% \text{tam}].\text{livre} \neq 'L'$  E  $\text{tabela}[(\text{pos}+i)\% \text{tam}].\text{livre} \neq 'R'$

$0 < 8$  E  $\text{tabela}[1 \% 8].\text{livre} \neq 'L'$  E  $\text{tabela}[1 \% 8].\text{livre} \neq 'R'$

$0 < 8$  E  $\text{tabela}[1].\text{livre} \neq 'L'$  E  $\text{tabela}[1].\text{livre} \neq 'R'$

$0 < 8$  E  $'O' \neq 'L'$  E  $'O' \neq 'R'$

V E V E V = V → incrementa i

| num | tam | i | pos |
|-----|-----|---|-----|
| 25  | 8   | 1 | 1   |

$$\text{pos} = \text{num \% tam} = 25 \% 8 = 1$$

$i < \text{tam}$  E  $\text{tabela}[(\text{pos}+i)\% \text{tam}].\text{livre} \neq 'L'$  E  $\text{tabela}[(\text{pos}+i)\% \text{tam}].\text{livre} \neq 'R'$

$1 < 8$  E  $\text{tabela}[2 \% 8].\text{livre} \neq 'L'$  E  $\text{tabela}[2 \% 8].\text{livre} \neq 'R'$

$1 < 8$  E  $\text{tabela}[2].\text{livre} \neq 'L'$  E  $\text{tabela}[2].\text{livre} \neq 'R'$

$1 < 8$  E  $'L' \neq 'L'$  E  $'L' \neq 'R'$

V E F E V = F → verifica se a posição encontrada, ou seja, i, é válida para inserção

$i < \text{tam}$

$1 < 8$

V → insere o nº 25 na posição 2 e altera  $\text{tabela}[2].\text{livre}$  para 'O'

livre chave

|   |   |    |
|---|---|----|
| 0 | O | 16 |
| 1 | O | 41 |
| 2 | O | 25 |
| 3 | L |    |
| 4 | L |    |
| 5 | L |    |
| 6 | L |    |
| 7 | O | 23 |

| num | tam | i | pos |
|-----|-----|---|-----|
| 39  | 8   | 0 | 7   |

$$\text{pos} = \text{num \% tam} = 39 \% 8 = 7$$

$i < \text{tam}$  E  $\text{tabela}[(\text{pos}+i)\% \text{tam}].\text{livre} \neq 'L'$  E  $\text{tabela}[(\text{pos}+i)\% \text{tam}].\text{livre} \neq 'R'$

$0 < 8$  E  $\text{tabela}[7 \% 8].\text{livre} \neq 'L'$  E  $\text{tabela}[7 \% 8].\text{livre} \neq 'R'$

$0 < 8$  E  $\text{tabela}[7].\text{livre} \neq 'L'$  E  $\text{tabela}[7].\text{livre} \neq 'R'$

$0 < 8$  E  $'O' \neq 'L'$  E  $'O' \neq 'R'$

V E V E V = V → incrementa i

| num | tam | i | pos |
|-----|-----|---|-----|
| 39  | 8   | 1 | 7   |

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].livre ≠ 'R'  
 1 < 8 E tabela[8%8].livre ≠ 'L' E tabela[8%8].livre ≠ 'R'  
 1 < 8 E tabela[0].livre ≠ 'L' E tabela[0].livre ≠ 'R'  
 1 < 8 E 'O' ≠ 'L' E 'O' ≠ 'R'  
 V E V E V = V → incrementa i

| livre chave |
|-------------|
| O 16        |
| O 41        |
| O 25        |
| L           |
| L           |
| L           |
| L           |
| O 23        |



| livre chave |
|-------------|
| O 16        |
| O 41        |
| O 25        |
| O 39        |
| L           |
| L           |
| O 23        |

| num | tam | i | pos |
|-----|-----|---|-----|
| 39  | 8   | 2 | 7   |

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].livre ≠ 'R'  
 2 < 8 E tabela[9%8].livre ≠ 'L' E tabela[9%8].livre ≠ 'R'  
 2 < 8 E tabela[1].livre ≠ 'L' E tabela[1].livre ≠ 'R'  
 2 < 8 E 'O' ≠ 'L' E 'O' ≠ 'R'  
 V E V E V = V → incrementa i

| num | tam | i | pos |
|-----|-----|---|-----|
| 39  | 8   | 3 | 7   |

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].livre ≠ 'R'  
 3 < 8 E tabela[10%8].livre ≠ 'L' E tabela[10%8].livre ≠ 'R'  
 3 < 8 E tabela[2].livre ≠ 'L' E tabela[2].livre ≠ 'R'  
 3 < 8 E 'O' ≠ 'L' E 'O' ≠ 'R'  
 V E V E V = V → incrementa i

| num | tam | i | pos |
|-----|-----|---|-----|
| 39  | 8   | 4 | 7   |

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].livre ≠ 'R'  
 4 < 8 E tabela[11%8].livre ≠ 'L' E tabela[11%8].livre ≠ 'R'  
 4 < 8 E tabela[3].livre ≠ 'L' E tabela[3].livre ≠ 'R'  
 4 < 8 E 'L' ≠ 'L' E 'L' ≠ 'R'  
 V E F E V = F → verifica se a posição encontrada, ou seja, i, é válida para inserção

i < tam  
 4 < 8  
 V → insere o nº 39 na posição 3 e altera tabela[3].livre para 'O'

### 7ª operação Remoção do nº 41 da tabela hashing

| livre chave |
|-------------|
| O 16        |
| O 41        |
| O 25        |
| O 39        |
| L           |
| L           |
| L           |
| O 23        |



| num | tam | i | pos |
|-----|-----|---|-----|
| 41  | 8   | 0 | 1   |

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].chave ≠ num  
 0 < 8 E tabela[1%8].livre ≠ 'L' E tabela[1%8].chave ≠ 41  
 0 < 8 E tabela[1].livre ≠ 'L' E tabela[1].chave ≠ 41

0 < 8 E 'O' ≠ 'L' E 41 ≠ 41  
 V E V E F = F → verifica se na posição (pos+i)%tam está a chave e se esta já foi removida

| livre chave |
|-------------|
| O 16        |
| R 41        |
| O 25        |
| O 39        |
| L           |
| L           |
| L           |
| O 23        |

tabela[(pos+i)%tam].chave = num E tabela[(pos+i)%tam].livre ≠ 'R'  
 tabela[1%8].chave = num E tabela[1%8].livre ≠ 'R'  
 tabela[1].chave = num E tabela[1].livre ≠ 'R'  
 41 = 41 E 'O' ≠ 'R'  
 V E V = V → remove o nº 41 da posição 1 alterando tabela[1].livre para 'R'

**8<sup>a</sup> operação**  
**Remoção do nº 23 da tabela hashing**

livre chave

|   |        |
|---|--------|
| 0 | O   16 |
| 1 | R   41 |
| 2 | O   25 |
| 3 | O   39 |
| 4 | L      |
| 5 | L      |
| 6 | L      |
| 7 | O   23 |

num      tam

|    |
|----|
| 23 |
| 8  |

i      pos

|   |
|---|
| 0 |
| 7 |

$$\text{pos} = \text{num} \% \text{tam} = 23 \% 8 = 7$$

i &lt; tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].chave ≠ num

0 &lt; 8 E tabela[7%8].livre ≠ 'L' E tabela[7%8].chave ≠ 23

0 &lt; 8 E tabela[7].livre ≠ 'L' E tabela[7].chave ≠ 23

0 &lt; 8 E 'O' ≠ 'L' E 23 ≠ 23

V E V E F = F → verifica se na posição (pos+i)%tam está a chave e se esta já foi removida



livre chave

|   |        |
|---|--------|
| 0 | O   16 |
| 1 | R   41 |
| 2 | O   25 |
| 3 | O   39 |
| 4 | L      |
| 5 | L      |
| 6 | L      |
| 7 | R   23 |

num      tam

|    |
|----|
| 25 |
| 8  |

i      pos

|   |
|---|
| 0 |
| 1 |

$$\text{pos} = \text{num} \% \text{tam} = 25 \% 8 = 1$$

i &lt; tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].chave ≠ num

0 &lt; 8 E tabela[1%8].livre ≠ 'L' E tabela[1%8].chave ≠ 25

0 &lt; 8 E tabela[1].livre ≠ 'L' E tabela[1].chave ≠ 25

0 &lt; 8 E 'R' ≠ 'L' E 41 ≠ 25

V E V E V = V → incrementa i



livre chave

|   |        |
|---|--------|
| 0 | O   16 |
| 1 | R   41 |
| 2 | R   25 |
| 3 | O   39 |
| 4 | L      |
| 5 | L      |
| 6 | L      |
| 7 | R   23 |

num      tam

|    |
|----|
| 25 |
| 8  |

i      pos

|   |
|---|
| 1 |
| 1 |

$$\text{pos} = \text{num} \% \text{tam} = 25 \% 8 = 1$$

i &lt; tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].chave ≠ num

1 &lt; 8 E tabela[2%8].livre ≠ 'L' E tabela[2%8].chave ≠ 25

1 &lt; 8 E tabela[2].livre ≠ 'L' E tabela[2].chave ≠ 25

1 &lt; 8 E 'O' ≠ 'L' E 25 ≠ 25

V E V E F = F → verifica se na posição (pos+i)%tam está a chave e se esta já foi removida

tabela[(pos+i)%tam].chave = num E tabela[(pos+i)%tam].livre ≠ 'R'

tabela[2%8].chave = num E tabela[2%8].livre ≠ 'R'

tabela[2].chave = num E tabela[2].livre ≠ 'R'

25 = 25 E 'O' ≠ 'R'

V E V = V → remove o nº 25 da posição 2 alterando tabela[2].livre para 'R'

## 10ª operação

Inserção do nº 34 na tabela *hashing*

livre chave

|   |   |    |
|---|---|----|
| 0 | O | 16 |
| 1 | R | 41 |
| 2 | R | 25 |
| 3 | O | 39 |
| 4 | L |    |
| 5 | L |    |
| 6 | L |    |
| 7 | R | 23 |

num  
34tam  
8i  
0pos  
2

$$\text{pos} = \text{num} \% \text{tam} = 34 \% 8 = 2$$

$i < \text{tam}$  E  $\text{tabela}[(\text{pos}+i)\%8].\text{livre} \neq 'L'$  E  $\text{tabela}[(\text{pos}+i)\%8].\text{livre} \neq 'R'$   
 $0 < 8$  E  $\text{tabela}[2\%8].\text{livre} \neq 'L'$  E  $\text{tabela}[2\%8].\text{livre} \neq 'R'$   
 $0 < 8$  E  $\text{tabela}[2].\text{livre} \neq 'L'$  E  $\text{tabela}[2].\text{livre} \neq 'R'$   
 $0 < 8$  E ' $R$ '  $\neq 'L'$  E ' $R$ '  $\neq 'R'$

V E V E F = F → verifica se a posição encontrada, ou seja, i, é válida para inserção

livre chave

|   |   |    |
|---|---|----|
| 0 | O | 16 |
| 1 | R | 41 |
| 2 | O | 34 |
| 3 | O | 39 |
| 4 | L |    |
| 5 | L |    |
| 6 | L |    |
| 7 | R | 23 |



O box a seguir apresenta uma implementação em JAVA de uma tabela *hashing* com tentativa linear.




---

```

import java.util.Scanner;
public class Hash_Abertol
{
 public static class hash
 {
 int chave;
 char livre; // L = livre, O = ocupado, R =
 // removido
 };

 static int tam=8; // tamanho da função hashing
 static hash tabela[]=new hash[tam];
 static Scanner entrada = new Scanner(System.in);

 public static void inserir(int pos, int n)
 {
 int i=0;
 while(i < tam
 && tabela[(post+i)%tam].livre != 'L'
 && tabela[(post+i)%tam].livre != 'R')

```

```
i = i+1;

if (i < tam)
{
 tabela[(pos+i)%tam].chave = n;
 tabela[(pos+i)%tam].livre = '0';
}
else
 System.out.println("Tabela cheia!");
}

public static void remover(int n)
{
 int posicao = buscar(n);

 if (posicao < tam)
 tabela[posicao].livre = 'R';
 else
 System.out.println("Elemento não
 → está presente.");
}

public static int buscar(int n)
{
 int i=0;
 int pos=funcao_hashing(n);

 while(i < tam
 && tabela[(pos+i)%tam].livre != 'L'
 && tabela[(pos+i)%tam].chave != n)
 i = i+1;

 if (tabela[(pos+i)%tam].chave == n
 && tabela[(pos+i)%tam].livre != 'R')
 return (pos+i)%tam;
 else
 return tam; // não encontrado
}

static int funcao_hashing(int num)
{
 return num % tam;
}

static void mostrar_hash()
```

```
{
 for(int i=0; i < tam; i++)
 if (tabela[i].livre == 'O')
 System.out.println("Entrada "+i+": "
 +tabela[i].chave+ " " + tabela[i].
 livre);
}

public static void main(String[] args)
{
 int op, pos;
 int num, i;

 // inicialização da tabela
 for(i = 0; i < tam; i++)
 {
 tabela[i] = new hash();
 tabela[i].livre='L';
 }
 do
 {
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir elemento");
 System.out.println("2 - Mostrar tabela
 hashing");
 System.out.println("3 - Excluir elemento");
 System.out.println("4 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();

 if (op < 1 || op > 4)
 System.out.println("Opção Inválida!");
 else
 {
 switch(op)
 {
 case 1:
 System.out.
 println("Digite um
 número:");
 }
 }
 }
}
```

```

 num = entrada.nextInt();
 pos=funcao_hashing(num);
 inserir(pos, num);
 break;
 case 2:
 mostrar_hash();
 break;
 case 3:
 System.out.println("Digite um número: ");
 num = entrada.nextInt();
 remover(num);
 break;
 }
}
while (op!=4);
}
}
}

```



c/c++

```

#include<iostream.h>
#include<conio.h>

const int tam=8;

struct hash
{
 int chave;
 char livre; // L = livre, O = ocupado, R = removido
};

int funcao_hashing(int num)
{
 return num % tam;
}

void mostrar_hash(hash tabela[])
{
 for(int i=0; i < tam; i++)
 if (tabela[i].livre == 'O')

```

```
 cout << "\nEntrada " << i << ": " <<
 tabela[i].chave;
 }

void inserir(hash tabela[], int pos, int n)
{
 int i=0;
 while(i < tam && tabela[(pos+i)%tam].livre != 'L'
 && tabela[(pos+i)%tam].livre != 'R')
 i = i+1;

 if (i < tam)
 {
 tabela[(pos+i)%tam].chave = n;
 tabela[(pos+i)%tam].livre = 'O';
 }
 else
 cout << "\nTabela cheia!";
}

int buscar(hash tabela[], int n)
{
 int i=0;
 int pos=funcao_hashing(n);

 while(i < tam && tabela[(pos+i)%tam].livre != 'L' &&
 tabela[(pos+i)%tam].chave != n)
 i = i+1;

 if (tabela[(pos+i)%tam].chave == n &&
 tabela[(pos+i)%tam].livre != 'R')
 return (pos+i)%tam;
 else
 return tam; // não encontrado
}

void remover(hash tabela[], int n)
{
 int posicao = buscar(tabela, n);

 if (posicao < tam)
 tabela[posicao].livre = 'R';
```

```
else
 cout << "\nElemento não está presente.";
}

void main()
{
 hash tabela[tam];

 int op, pos;
 int num, i;

 // inicialização da tabela
 for(i = 0; i < tam; i++)
 tabela[i].livre='L';

 do
 {
 clrscr();
 cout << "\nMENU DE OPÇÕES\n";
 cout << "\n1 - Inserir elemento";
 cout << "\n2 - Mostrar tabela hashing";
 cout << "\n3 - Excluir elemento";
 cout << "\n4 - Sair";
 cout << "\nDigite sua opção: ";
 cin >> op;

 if (op < 1 || op > 4)
 cout << "\nOpção Inválida! ";
 else
 {
 switch(op)
 {
 case 1:
 cout << "\nDigite um número:";
 cin >> num;
 pos=funcao_hashing(num);
 inserir(tabela, pos, num);
 break;
 case 2:
 mostrar_hash(tabela);
 break;
 }
 }
 } while (op != 4);
}
```

```

 case 3:
 cout << "\nDigite um
 → número: ";
 cin >> num;
 remover(tabela, num);
 break;
 }
 }
 getch();
}
while (op!=4);
}

```

---

## Análise da complexidade da tabela *hashing* com tentativa linear

Para uma tabela *hashing* com  $m$  posições que armazena  $n$  elementos, define-se o fator de carga  $\alpha$  para a tabela como  $n/m$ , ou seja, é o número médio de elementos armazenados em uma entrada. A análise em geral dos métodos aplicados sobre uma tabela *hashing* são realizados sobre o valor  $\alpha$ , que segundo Cormen (2002), pode ser menor, igual ou maior que 1.

A análise é feita supondo-se que a função de *hashing* é uma *hashing* uniforme, ou seja, supondo que cada chave tem a mesma probabilidade de ter qualquer das  $m!$  permutações  $(0, 1, \dots, m-1)$  em sua sequência de tentativas. Como o *hashing* uniforme é difícil de implementar na prática, aproximações dela são usadas. Para calcular as sequências de tentativas para o endereçamento aberto, são usadas a tentativa linear e a tentativa quadrática.

Segundo Cormen (2002), as duas tentativas mencionadas acima calculam endereços que são permutações de  $(0, 1, \dots, m-1)$ , porém, nenhuma delas utiliza a hipótese de *hashing* uniforme, nenhuma calcula mais que  $m^2$  sequências de tentativas diferentes em vez de  $m!$ .

No caso da tentativa linear, ao tentar inserir, remover ou buscar um elemento, realiza-se uma operação aritmética para descobrir o endereço onde o elemento deverá ser inserido, removido ou buscado. O risco da colisão, porém, é sempre presente. Com isso, quando as tentativas por novas posições são realizadas, nas três operações gasta-se no pior caso  $O(n)$ , visto que são gerados agrupamentos primários, que são longos trechos de endereços ocupados.

Porém, o número de tentativas realizadas, segundo Cormen (2002), deve ser expresso em termos do fator de carga  $\alpha = n / m$  da tabela *hashing*. No caso de endereçamento aberto, tem-se no máximo um elemento por posição. Logo,  $n \leq m$  e  $\alpha \leq 1$ .

Segundo Cormen (2002), supondo-se uma função de *hashing* uniforme, para uma tabela *hashing* com fator de carga  $\alpha < 1$ , os seguintes resultados são verificados:

- O número de tentativas esperado em uma pesquisa malsucedida é no máximo  $1 / (1 - \alpha)$ .
- Em uma tabela com endereçamento aberto, a inserção gasta no máximo  $1 / (1 - \alpha)$  tentativas em média.
- O número esperado de tentativas em uma pesquisa bem-sucedida é no máximo  $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$ .

## Tentativa quadrática

Na tentativa linear, as chaves que colidem geram os chamados agrupamentos primários, o que aumenta o tempo de busca.

Nesta tentativa, outro tipo de agrupamento também é gerado, o agrupamento secundário, mas as degradações são reduzidas se comparadas com a tentativa linear. No entanto, se duas chaves possuem o mesmo endereço inicial, então todas as tentativas seguintes serão iguais nos dois métodos.

Segundo Szwarcfiter (1994), para a aplicação deste método os endereços calculados pela função de *hashing*  $h'(x, k)$  devem corresponder à varredura de toda a tabela, para  $k = 0, \dots, m - 1$ . As equações recorrentes a seguir fornecem uma maneira de calcular os endereços diretamente.

$$h'(x, 0) = h(x)$$

$$h'(x, k) = (h(x, k - 1) + k) \bmod m, 1 \leq k \leq m - 1$$

A próxima ilustração mostra uma tabela *hashing* implementada com tentativa quadrática.



ILUSTRAÇÃO

**1ª operação**  
A tabela *hashing* está vazia

|   | livre chave |
|---|-------------|
| 0 | L           |
| 1 | L           |
| 2 | L           |
| 3 | L           |
| 4 | L           |
| 5 | L           |
| 6 | L           |
| 7 | L           |

## 2ª operação

## Inserção do nº 12 na tabela hashing

livre chave

|   |   |
|---|---|
| 0 | L |
| 1 | L |
| 2 | L |
| 3 | L |
| 4 | L |
| 5 | L |
| 6 | L |
| 7 | L |

num

tam

k

pos

12

8

1

4

$$\text{pos} = \text{num} \% \text{tam} = 12 \% 8 = 4$$

$k \leq \text{tam}$  E  $\text{tabela}[\text{pos}].\text{livre} \neq 'L'$  E  $\text{tabela}[\text{pos}].\text{livre} \neq 'R'$

$1 \leq 8$  E  $\text{tabela}[4].\text{livre} \neq 'L'$  E  $\text{tabela}[4].\text{livre} \neq 'R'$

$1 \leq 8$  E ' $L$ '  $\neq 'L'$  E ' $L$ '  $\neq 'R'$

V E F E V = F



livre chave

|   |      |
|---|------|
| 0 | L    |
| 1 | L    |
| 2 | L    |
| 3 | L    |
| 4 | O 12 |
| 5 | L    |
| 6 | L    |
| 7 | L    |

verifica se existe posição livre ou removida para inserção, ou seja, se  $k \leq \text{tam}$ , sendo a posição de inserção  $\text{pos} = \text{num} \% \text{tam}$

$k \leq \text{tam}$

$1 \leq 8$

V  $\rightarrow$  insere o nº 12 na posição 4 e altera  $\text{tabela}[4].\text{livre}$  para 'O'

## 3ª operação

## Inserção do nº 20 na tabela hashing

num

tam

k

pos

20

8

1

4

$$\text{pos} = \text{num} \% \text{tam} = 20 \% 8 = 4$$

livre chave

|   |      |
|---|------|
| 0 | L    |
| 1 | L    |
| 2 | L    |
| 3 | L    |
| 4 | O 12 |
| 5 | L    |
| 6 | L    |
| 7 | L    |

$k \leq \text{tam}$  E  $\text{tabela}[\text{pos}].\text{livre} \neq 'L'$  E  $\text{tabela}[\text{pos}].\text{livre} \neq 'R'$

$1 \leq 8$  E  $\text{tabela}[4].\text{livre} \neq 'L'$  E  $\text{tabela}[4].\text{livre} \neq 'R'$

$1 \leq 8$  E ' $O$ '  $\neq 'L'$  E ' $O$ '  $\neq 'R'$

V E V E V = V  $\rightarrow$  atualiza pos e k

$$\text{pos} = (\text{pos} + k) \% \text{tam} = (4 + 1) \% 8 = 5 \% 8 = 5$$

$$k = k + 1 = 1 + 1 = 2$$

num

tam

k

pos

20

8

2

5

livre chave

|   |      |
|---|------|
| 0 | O    |
| 1 | L    |
| 2 | L    |
| 3 | L    |
| 4 | O 12 |
| 5 | O 20 |
| 6 | L    |
| 7 | L    |

$k \leq \text{tam}$  E  $\text{tabela}[\text{pos}].\text{livre} \neq 'L'$  E  $\text{tabela}[\text{pos}].\text{livre} \neq 'R'$

$2 \leq 8$  E  $\text{tabela}[5].\text{livre} \neq 'L'$  E  $\text{tabela}[5].\text{livre} \neq 'R'$

$2 \leq 8$  E ' $L$ '  $\neq 'L'$  E ' $L$ '  $\neq 'R'$

V E F E V = F

verifica se existe posição livre ou removida para inserção, ou seja, se  $k \leq \text{tam}$ , sendo a posição de inserção  $\text{pos} = \text{num} \% \text{tam}$

$k \leq \text{tam}$

$2 \leq 8$

V  $\rightarrow$  insere o nº 20 na posição 5 e altera  $\text{tabela}[5].\text{livre}$  para 'O'

**4<sup>a</sup> operação**  
**Inserção do nº 28 na tabela hashing**

| livre chave | num | tam | k | pos |                                  |
|-------------|-----|-----|---|-----|----------------------------------|
| 0 L         | 28  | 8   | 1 | 4   | $pos = num \% tam = 28 \% 8 = 4$ |
| 1 L         |     |     |   |     |                                  |
| 2 L         |     |     |   |     |                                  |
| 3 L         |     |     |   |     |                                  |
| 4 O 12      |     |     |   |     |                                  |
| 5 O 20      |     |     |   |     |                                  |
| 6 L         |     |     |   |     |                                  |
| 7 L         |     |     |   |     |                                  |



| livre chave | num | tam | k | pos |  |
|-------------|-----|-----|---|-----|--|
| 0 L         | 28  | 8   | 2 | 5   |  |
| 1 L         |     |     |   |     |  |
| 2 L         |     |     |   |     |  |
| 3 L         |     |     |   |     |  |
| 4 O 12      |     |     |   |     |  |
| 5 O 20      |     |     |   |     |  |
| 6 L         |     |     |   |     |  |
| 7 O 28      |     |     |   |     |  |

$$pos = (pos+k)\%tam = (4+1)\%8 = 5\%8 = 5$$

$$k = k + 1 = 1 + 1 = 2$$

| num | tam | k | pos |
|-----|-----|---|-----|
| 28  | 8   | 3 | 7   |

$k \leq tam$  E  $tabela[pos].livre \neq 'L'$  E  $tabela[pos].livre \neq 'R'$

$3 \leq 8$  E  $tabela[7].livre \neq 'L'$  E  $tabela[7].livre \neq 'R'$

$3 \leq 8$  E ' $L$ '  $\neq 'L'$  E ' $O$ '  $\neq 'R'$

V E F E V = F

verifica se existe posição livre ou removida para inserção, ou seja,  
 se  $k \leq tam$ , sendo a posição de inserção  $pos = num \% tam$

$k \leq tam$

$3 \leq 8$

V → insere o nº 28 na posição 7 e altera tabela[7].livre para 'O'

**5<sup>a</sup> operação**  
**Inserção do nº 36 na tabela hashing**

| livre chave | num | tam | k | pos |                                  |
|-------------|-----|-----|---|-----|----------------------------------|
| 0 L         | 36  | 8   | 1 | 4   | $pos = num \% tam = 36 \% 8 = 4$ |
| 1 L         |     |     |   |     |                                  |
| 2 L         |     |     |   |     |                                  |
| 3 L         |     |     |   |     |                                  |
| 4 O 12      |     |     |   |     |                                  |
| 5 O 20      |     |     |   |     |                                  |
| 6 L         |     |     |   |     |                                  |
| 7 L         |     |     |   |     |                                  |

$k \leq tam$  E  $tabela[pos].livre \neq 'L'$  E  $tabela[pos].livre \neq 'R'$

$1 \leq 8$  E  $tabela[4].livre \neq 'L'$  E  $tabela[4].livre \neq 'R'$

$1 \leq 8$  E ' $O$ '  $\neq 'L'$  E ' $O$ '  $\neq 'R'$

V E V E V = V → atualiza pos e k

$$pos = (pos+k)\%tam = (4+1)\%8 = 5\%8 = 5$$

$$k = k + 1 = 1 + 1 = 2$$

## 260 Estruturas de dados

livre chave

|   | num | tam | k | pos |
|---|-----|-----|---|-----|
| 0 | 36  | 8   | 2 | 5   |
| 1 |     |     |   |     |
| 2 |     |     |   |     |
| 3 |     |     |   |     |
| 4 |     |     |   |     |
| 5 |     |     |   |     |
| 6 |     |     |   |     |
| 7 |     |     |   |     |

k <= tam E tabela[pos].livre ≠ 'L' E tabela[pos].livre ≠ 'R'  
 2 <= 8 E tabela[5].livre ≠ 'L' E tabela[5].livre ≠ 'R'  
 2 <= 8 E 'O' ≠ 'L' E 'O' ≠ 'R'  
 V E V E V = V → atualiza pos e k

pos = (pos+k)%tam = (5+2)%8 = 7%8 = 7  
 k = k + 1 = 2 + 1 = 3

↓

livre chave

|   | num | tam | k | pos |
|---|-----|-----|---|-----|
| 0 | 36  | 8   | 3 | 7   |
| 1 |     |     |   |     |
| 2 |     |     |   |     |
| 3 |     |     |   |     |
| 4 |     |     |   |     |
| 5 |     |     |   |     |
| 6 |     |     |   |     |
| 7 |     |     |   |     |

k <= tam E tabela[pos].livre ≠ 'L' E tabela[pos].livre ≠ 'R'  
 3 <= 8 E tabela[7].livre ≠ 'L' E tabela[5].livre ≠ 'R'  
 3 <= 8 E 'O' ≠ 'L' E 'O' ≠ 'R'  
 V E V E V = V → atualiza pos e k

pos = (pos+k)%tam = (7+3)%8 = 10%8 = 2  
 k = k + 1 = 3 + 1 = 4

|   | num | tam | k | pos |
|---|-----|-----|---|-----|
| 0 | 36  | 8   | 4 | 2   |
| 1 |     |     |   |     |
| 2 |     |     |   |     |
| 3 |     |     |   |     |
| 4 |     |     |   |     |
| 5 |     |     |   |     |
| 6 |     |     |   |     |
| 7 |     |     |   |     |

k <= tam E tabela[pos].livre ≠ 'L' E tabela[pos].livre ≠ 'R'  
 4 <= 8 E tabela[2].livre ≠ 'L' E tabela[2].livre ≠ 'R'  
 4 <= 8 E 'L' ≠ 'L' E 'L' ≠ 'R'  
 V E F E V = F

verifica se existe posição livre ou removida para inserção, ou seja,  
 se k <= tam, sendo a posição de inserção pos = num%tam

k <= tam  
 4 <= 8  
 V → insere o nº 36 na posição 2 e altera tabela[2].livre para 'O'

### 6ª operação Remoção do nº 12 da tabela hashing

livre chave

|   | num | tam | k | pos |
|---|-----|-----|---|-----|
| 0 | 12  | 8   | 1 | 4   |
| 1 |     |     |   |     |
| 2 |     |     |   |     |
| 3 |     |     |   |     |
| 4 |     |     |   |     |
| 5 |     |     |   |     |
| 6 |     |     |   |     |
| 7 |     |     |   |     |

pos = num%tam = 12%8 = 4

k <= tam E tabela[pos].livre ≠ 'L' E tabela[pos].chave ≠ num  
 1 <= 8 E tabela[4].livre ≠ 'L' E tabela[4].chave ≠ 12  
 1 <= 8 E 'O' ≠ 'L' E 12 ≠ 12  
 V E V E F = F → verifica se na posição pos está a chave e se esta já foi removida

↓

livre chave

|   | num | tam | k | pos |
|---|-----|-----|---|-----|
| 0 |     |     |   |     |
| 1 |     |     |   |     |
| 2 |     |     |   |     |
| 3 |     |     |   |     |
| 4 |     |     |   |     |
| 5 |     |     |   |     |
| 6 |     |     |   |     |
| 7 |     |     |   |     |

tabela[pos].chave = num E tabela [pos].livre ≠ 'R'  
 tabela[4].chave = num E tabela [4].livre ≠ 'R'  
 12 = 12 E 'O' ≠ 'R'  
 V E V = V → remove o nº 12 da posição 4 alterando tabela[4].livre para 'R'

## 7ª operação

Remoção do nº 20 da tabela *hashing*

| num | tam | k | pos |
|-----|-----|---|-----|
| 20  | 8   | 1 | 4   |

$$\text{pos} = \text{num} \% \text{tam} = 20 \% 8 = 4$$

livre chave

|   |   |    |
|---|---|----|
| 0 | L |    |
| 1 | L |    |
| 2 | O | 36 |
| 3 | L |    |
| 4 | R | 12 |
| 5 | O | 20 |
| 6 | L |    |
| 7 | O | 28 |

$k \leq tam$  E  $\text{tabela}[pos].\text{livre} \neq 'L'$  E  $\text{tabela}[pos].\text{chave} \neq \text{num}$

$1 \leq 8$  E  $\text{tabela}[4].\text{livre} \neq 'L'$  E  $\text{tabela}[4].\text{chave} \neq 20$

$1 \leq 8$  E ' $R$ '  $\neq 'L'$  E  $12 \neq 20$

V E V E V = V  $\rightarrow$  atualiza pos e k

$$\text{pos} = (\text{pos} + k) \% \text{tam} = (4 + 1) \% 8 = 5 \% 8 = 5$$

$$k = k + 1 = 1 + 1 = 2$$



livre chave

|   |   |    |
|---|---|----|
| 0 | L |    |
| 1 | L |    |
| 2 | O | 36 |
| 3 | L |    |
| 4 | R | 12 |
| 5 | R | 20 |
| 6 | L |    |
| 7 | O | 28 |

$k \leq tam$  E  $\text{tabela}[pos].\text{livre} \neq 'L'$  E  $\text{tabela}[pos].\text{chave} \neq \text{num}$

$2 \leq 8$  E  $\text{tabela}[5].\text{livre} \neq 'L'$  E  $\text{tabela}[5].\text{chave} \neq 20$

$2 \leq 8$  E ' $O$ '  $\neq 'L'$  E  $20 \neq 20$

V E V E F = F  $\rightarrow$  verifica se na posição pos está a chave e se esta já foi removida

$\text{tabela}[pos].\text{chave} = \text{num}$  E  $\text{tabela}[pos].\text{livre} \neq 'R'$

$\text{tabela}[5].\text{chave} = \text{num}$  E  $\text{tabela}[4].\text{livre} \neq 'R'$

$20 = 20$  E ' $O$ '  $\neq 'R'$

V E V = V  $\rightarrow$  remove o nº 20 da posição 5 alterando  $\text{tabela}[5].\text{livre}$  para ' $R$ '

## 8ª operação

Inserção do nº 44 na tabela *hashing*

livre chave

|   |   |    |
|---|---|----|
| 0 | L |    |
| 1 | L |    |
| 2 | O | 36 |
| 3 | L |    |
| 4 | R | 12 |
| 5 | R | 20 |
| 6 | L |    |
| 7 | O | 28 |

| num | tam | k | pos |
|-----|-----|---|-----|
| 44  | 8   | 1 | 4   |

$$\text{pos} = \text{num} \% \text{tam} = 44 \% 8 = 4$$

$k \leq tam$  E  $\text{tabela}[pos].\text{livre} \neq 'L'$  E  $\text{tabela}[pos].\text{livre} \neq 'R'$

$1 \leq 8$  E  $\text{tabela}[4].\text{livre} \neq 'L'$  E  $\text{tabela}[4].\text{livre} \neq 'R'$

$1 \leq 8$  E ' $R$ '  $\neq 'L'$  E ' $R$ '  $\neq 'R'$

V E V E F = F

livre chave

|   |   |    |
|---|---|----|
| 0 | L |    |
| 1 | L |    |
| 2 | O | 36 |
| 3 | L |    |
| 4 | O | 44 |
| 5 | R | 20 |
| 6 | L |    |
| 7 | O | 28 |

verifica se existe posição livre ou removida para inserção, ou seja, se  $k \leq tam$ , sendo a posição de inserção  $\text{pos} = \text{num} \% \text{tam}$

$k \leq tam$

$1 \leq 8$

V  $\rightarrow$  insere o nº 44 na posição 4 e altera  $\text{tabela}[4].\text{livre}$  para ' $O$ '




---

```

import java.util.Scanner;

public class Hash_Aberto2
{
 public static class hash
 {
 int chave;
 char livre; // L = livre, O = ocupado, R = removido
 };

 static int tam=8; // tamanho da funcao hashing
 static hash tabela[] = new hash[tam];
 static Scanner entrada = new Scanner(System.in);

 public static void inserir(int n)
 {
 int pos = funcao_hashing(n);
 int k=1;
 while(k <= tam
 && tabela[pos].livre != 'L'
 && tabela[pos].livre != 'R')
 {
 pos = (pos+k)%tam;
 k = k+1;
 }
 if(k <= tam)
 {
 tabela[pos].chave = n;
 tabela[pos].livre = 'O';
 }
 else
 System.out.println("Tabela cheia ou em
 ↪ loop!");
 }

 public static void remover(int n)
 {
 int posicao = buscar(n);
 if (posicao < tam)
 tabela[posicao].livre = 'R';
 else
 System.out.println("Elemento não
 ↪ está presente.");
 }

 public static int buscar(int n)
 {

```

```
int pos = funcao_hashing(n);
int k=1;
while(k <= tam
 && tabela[pos].livre != 'L'
 && tabela[pos].chave != n)
{
 pos = (pos+k)%tam;
 k = k+1;
}
if(tabela[pos].chave == n && tabela[pos].livre != 'R')
 return pos;
else
 return tam;
}

static int funcao_hashing(int num)
{
 return num % tam;
}
static void mostrar_hash()
{
 for(int i=0; i < tam; i++)
 if (tabela[i].livre == 'O')
 System.out.println("Entrada
 "+i+": "+tabela[i].chave);
}

public static void main(String[] args)
{
 int op;
 int num, i;

 // inicialização da tabela
 for(i = 0; i < tam; i++)
 {
 tabela[i] = new hash();
 tabela[i].livre='L';
 }
 do
 {
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir elemento");
 System.out.println("2 - Mostrar tabela hashing");
 System.out.println("3 - Excluir elemento");
 }
```

```

 System.out.println("4 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();

 if (op < 1 || op > 4)
 System.out.println("Opção Inválida! ");
 else
 {
 switch(op)
 {
 case 1:
 System.out.println("Digite um
➥ número:");
 num = entrada.nextInt();
 inserir(num);
 break;
 case 2:
 mostrar_hash();
 break;
 case 3:
 System.out.println("Digite um número: ");
 num = entrada.nextInt();
 remover(num);
 break;
 }
 }
 }
}
}
while (op!=4);
}
}

```

---



```

#include<iostream.h>
#include<conio.h>

const int tam=8;

struct hash
{
 int chave;
 char livre; // L = livre, O = ocupado, R = removido
};

int funcao_hashing(int num)

```

```
{
 return num % tam;
}

void mostrar_hash(hash tabela[])
{
 for(int i=0; i < tam; i++)
 if (tabela[i].livre == 'O')
 cout << "\nEntrada "<< i << ":" <<
 tabela[i].chave;
}
void inserir(hash tabela[], int n)
{
 int pos = funcao_hashing(n);
 int k=1;
 while(k <= tam && tabela[pos].livre != 'L' &&
 tabela[pos].livre != 'R')
 {
 pos = (pos+k)%tam;
 k = k+1;
 }
 if(k <= tam)
 {
 tabela[pos].chave = n;
 tabela[pos].livre = 'O';
 }
 else
 cout << "\nTabela cheia ou em loop!";
}

int buscar(hash tabela[], int n)
{
 int pos = funcao_hashing(n);
 int k=1;
 while(k <= tam && tabela[pos].livre != 'L' &&
 tabela[pos].chave != n)
 {
 pos = (pos+k)%tam;
 k = k+1;
 }
 if(tabela[pos].chave == n && tabela[pos].livre != 'R')
 return pos;
 else
 return tam;
}

void remover(hash tabela[], int n)
{
```

```
int posicao = buscar(tabela, n);
if (posicao < tam)
 tabela[posicao].livre = 'R';
else
 cout << "\nElemento não está presente.";
}

void main()
{
 hash tabela[tam];
 int op;
 int num, i;

 // inicialização da tabela
 for(i = 0; i < tam; i++)
 tabela[i].livre='L';

 do
 {
 clrscr();
 cout << "\nMENU DE OPÇÕES\n";
 cout << "\n1 - Inserir elemento";
 cout << "\n2 - Mostrar tabela hashing";
 cout << "\n3 - Excluir elemento";
 cout << "\n4 - Sair";
 cout << "\nDigite sua opção: ";
 cin >> op;

 if (op < 1 || op > 4)
 cout << "\nOpção Inválida! ";
 else
 {
 switch(op)
 {
 case 1:
 cout << "\nDigite um
 → número:";
 cin >> num;
 inserir(tabela, num);
 break;
 case 2:
 mostrar_hash(tabela);
 break;
 case 3:
 cout << "\nDigite um
 → número: ";
 cin >> num;
 remover(tabela, num);
 break;
 }
 }
 } while (op != 4);
}
```

```

 }
 getch();
}
while (op!=4);
}

```

## Análise da complexidade da tabela *hashing* com tentativa quadrática

Para uma tabela *hashing* com  $m$  posições que armazena  $n$  elementos, define-se o fator de carga  $\alpha$  para a tabela como  $n/m$ , ou seja, é o número médio de elementos armazenados em uma entrada.

Em geral, a análise dos métodos aplicados sobre uma tabela *hashing* é realizada sobre o valor  $\alpha = n / m$ , onde  $m$  é o número de entradas da tabela e  $n$  a quantidade de elementos em cada entrada. De acordo com Cormen (2002), a análise é feita supondo-se que a função de *hashing* é uma *hashing* uniforme, ou seja, supondo que cada chave tem a mesma probabilidade de ser armazenada em qualquer das  $m!$  entradas. Os resultados para uma função de *hashing* uniforme podem ser encontrados na seção “Análise da complexidade da tabela *hashing* com tentativa linear”.

## Tabela *hashing* implementada com lista

Neste tipo de implementação, a tabela *hashing* é um vetor com  $m$  posições, sendo que cada posição possui um ponteiro para uma lista encadeada onde estarão contidos todos os elementos que possuem o mesmo endereço mapeado. Os endereços da tabela *hashing* nesta seção são calculados utilizando o método da divisão.

A ilustração a seguir mostra uma tabela *hashing* implementada com lista.



ILUSTRAÇÃO

**1<sup>a</sup> operação**  
A tabela *hashing* está vazia

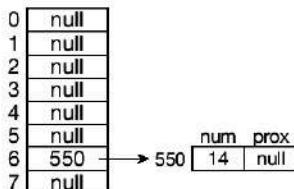
|   |      |
|---|------|
| 0 | null |
| 1 | null |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |

**2<sup>a</sup> operação**  
**Inserção do nº 14 na tabela hashing**

|     |
|-----|
| num |
| 14  |

|     |
|-----|
| tam |
| 8   |

$pos = num \% tam = 14 \% 8 = 6$

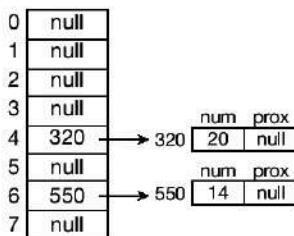


**3<sup>a</sup> operação**  
**Inserção do nº 20 na tabela hashing**

|     |
|-----|
| num |
| 20  |

|     |
|-----|
| tam |
| 8   |

$pos = num \% tam = 20 \% 8 = 4$

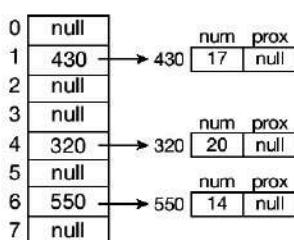


**4<sup>a</sup> operação**  
**Inserção do nº 17 na tabela hashing**

|     |
|-----|
| num |
| 17  |

|     |
|-----|
| tam |
| 8   |

$pos = num \% tam = 17 \% 8 = 1$

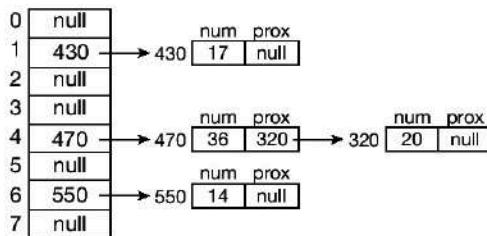


**5<sup>a</sup> operação**  
**Inserção do nº 36 na tabela hashing**

|     |
|-----|
| num |
| 36  |

|     |
|-----|
| tam |
| 8   |

$pos = num \% tam = 36 \% 8 = 4$

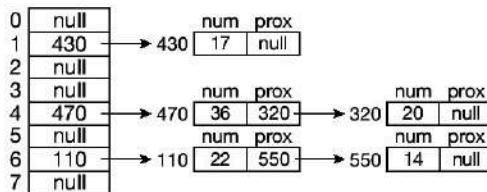


**6<sup>a</sup> operação**  
**Inserção do nº 22 na tabela hashing**

|     |
|-----|
| num |
| 22  |

|     |
|-----|
| tam |
| 8   |

$pos = num \% tam = 22 \% 8 = 6$

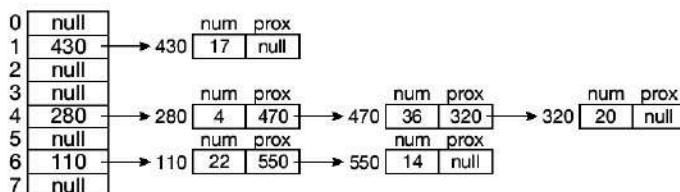


**7<sup>a</sup> operação**  
**Inserção do nº 4 na tabela hashing**

|     |
|-----|
| num |
| 4   |

|     |
|-----|
| tam |
| 8   |

$pos = num \% tam = 4 \% 8 = 4$

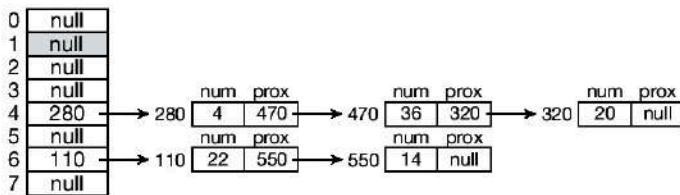
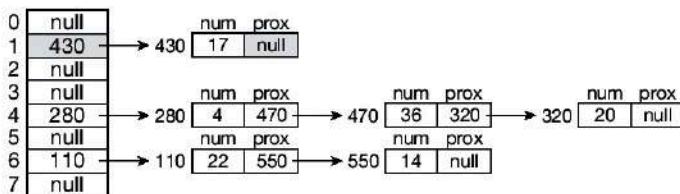


**8<sup>a</sup> operação**  
**Remoção do nº 17 da tabela hashing**

|     |     |
|-----|-----|
| num | tam |
| 17  | 8   |

pos = num%tam = 17%8 = 1

busca pelo nº 17 na posição 1  
caso encontre, quem apontava para o nº 17  
passará a apontar para o prox do nº 17

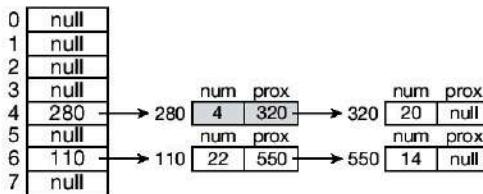
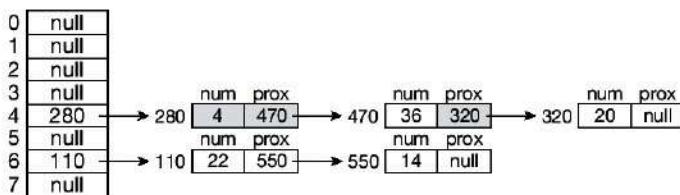


**9<sup>a</sup> operação**  
**Remoção do nº 36 da tabela hashing**

|     |     |
|-----|-----|
| num | tam |
| 36  | 8   |

pos = num%tam = 36%8 = 4

busca pelo nº 36 na posição 4  
caso encontre, quem apontava para o nº 36  
passará a apontar para o prox do nº 36



A seguir, o box apresenta uma implementação da tabela *hashing* implementada com lista.



```

import java.util.Scanner;

public class Hash_Lista
{
 public static class hash
 {
 int chave;
 hash prox;
 };

 static int tam=10; // tamanho da função hashing
 static hash tabela[] = new hash[tam];
 static Scanner entrada = new Scanner(System.in);

 public static void inserir(int pos, int n)
 {
 hash novo;
 novo = new hash();
 novo.chave = n;
 novo.prox = tabela[pos];
 tabela[pos] = novo;
 }

 static int funcao_hashing(int num)
 {
 return num % tam;
 }

 static void mostrar_hash()
 {
 hash aux;
 for(int i=0; i < tam; i++)
 {
 aux=tabela[i];
 while(aux != null)
 {
 System.out.println("Entrada "+i+
 " : "+aux.chave);
 aux = aux.prox;
 }
 }
 }

 static void excluir_hash(int num)
 {
 int pos = funcao_hashing(num);
 hash aux;
 }
}

```

```
 if(tabela[pos] != null)
 {
 if(tabela[pos].chave==num)
 tabela[pos] = tabela[pos].prox;
 else
 {
 aux=tabela[pos].prox;
 hash_ant=tabela[pos];
 while(aux != null &&
 aux.chave != num)
 {
 ant = aux;
 aux = aux.prox;
 }
 if (aux != null)
 ant.prox = aux.prox;
 else
 System.out.println("Número não
 → encontrado");
 }
 }
 else
 System.out.println("Número não
 encontrado");
 }
 public static void main(String[] args)
 {
 int op, pos;
 int num;

 do
 {
 System.out.println("\nMENU DE
 → OPÇÕES\n");
 System.out.println("1 - Inserir
 → elemento");
 System.out.println("2 - Mostrar
 → tabela hashing");
 System.out.println("3 - Excluir
 → elemento");
 System.out.println("4 - Sair");
 System.out.print("Digite sua
 → opção: ");
 op = entrada.nextInt();

 if (op < 1 || op > 4)
 System.out.println("Opção Inválida! ");
 else
 {
 switch(op)
 {
 case 1:
```

```

 System.out.println("Digite um
 → número:");
 num = entrada.nextInt();
 pos=funcao_hashing(num);
 inserir(pos, num);
 break;
 case 2:
 mostrar_hash();
 break;
 case 3:
 System.out.println("Digite um
 → número:");
 num = entrada.nextInt();
 excluir_hash(num);
 break;
 }
}
while (op!=4);
}
}

```

---



C/C++

```

#include<iostream.h>
#include<conio.h>

const int tam=8; // tamanho da tabela hashing
struct hash
{
 int chave;
 hash* prox;
};

void inserir(hash* tabela[],int pos, int n)
{
 hash* novo;
 novo = new hash();
 novo->chave = n;
 novo->prox = tabela[pos];
 tabela[pos] = novo;
}
int funcao_hashing(int num)
{
 return num % tam;
}

void mostrar_hash(hash* tabela[])

```

```

 {
 hash* aux;
 for(int i=0; i < tam; i++)
 {
 aux=tbla[i];
 while(aux != NULL)
 {
 cout << "\nEntrada " << i << ": "
 << aux->chave;
 aux = aux->prox;
 }
 }
 }

void excluir_hash(hash* tabela[], int num)
{
 int pos = funcao_hashing(num);
 hash* aux;

 if(tabela[pos] != NULL)
 {
 if(tabela[pos]->chave==num)
 {
 aux = tabela[pos];
 tabela[pos] = tabela[pos]->prox;
 delete aux;
 }
 else
 {
 aux=tabela[pos]->prox;
 hash* ant=tabela[pos];
 while(aux != NULL &&
 aux->chave != num)
 {
 ant = aux;
 aux = aux->prox;
 }
 if (aux != NULL)
 {
 ant->prox = aux->prox;
 delete aux;
 }
 else
 cout << "\nNúmero não
 encontrado";
 }
 }
 else
 cout << "\nNúmero não encontrado";
}

void main()

```

```
{
 hash* tabela[tam];
 hash* aux;
 int op, pos;
 int num, i;

 // inicialização da tabela
 for(i = 0; i < tam; i++)
 tabela[i] = NULL;
 do
 {
 clrscr();
 cout << "\nMENU DE OPÇÕES\n";
 cout << "\n1 - Inserir elemento";
 cout << "\n2 - Mostrar tabela hashing";
 cout << "\n3 - Excluir elemento";
 cout << "\n4 - Sair";
 cout << "\nDigite sua opção: ";
 cin >> op;

 if (op < 1 || op > 4)
 cout << "\nOpção Inválida! ";
 else
 {
 switch(op)
 {
 case 1:
 cout << "\nDigite um
 número:";
 cin >> num;
 pos=funcao_
 hashing(num);
 inserir(tabela, pos, num);
 break;
 case 2:
 mostrar_hash(tabela);
 getch();
 break;
 case 3:
 cout << "\nDigite um número: ";
 cin >> num;
 excluir_hash(tabela, num);
 break;
 }
 }
 }while (op!=4);
 // desalocando memoria
 for(i=0; i < tam; i++)
 {
 while(tabela[i] != NULL)
 {
 aux = tabela[i];
 tabela[i] = tabela[i->prox];
 delete aux;
 }
 }
}
```

```

 tabela[i] = tabela[i]->prox;
 delete aux;
 }
 tabela[i] = NULL;
}
}

```

## Análise da complexidade da tabela *hashing*

### implementada com lista

Na tabela *hashing* implementada com lista encadeada para tratamento de colisão, a operação de inserção gasta tempo constante,  $O(1)$ , pois para descobrir o endereço onde um elemento deve ser armazenado, a função de espalhamento realiza apenas uma operação aritmética e armazena o elemento naquele endereço. Se ocorrer uma colisão, o novo elemento também será armazenado na posição, porém, uma lista encadeada será formada e, no caso, o novo elemento será inserido no começo da lista, gastando também tempo constante, uma vez que é necessário apenas atualizar os ponteiros.

Já na operação de remoção de um elemento  $x$  qualquer, a função de espalhamento calcula o endereço onde  $x$  deveria estar armazenado, e uma busca por  $x$  é feita na lista encadeada dessa posição. Caso o número seja encontrado, ele é removido da lista, atualizando os ponteiros dela adequadamente. No pior caso, o número a ser removido se encontra no último nó da lista, gastando tempo proporcional ao número  $k$  de elementos armazenados nessa posição da tabela,  $O(k)$ .

## Exercícios

- Faça um programa que apresente o menu de opções abaixo. As operações devem ser implementadas em uma tabela *hashing*. Utilize endereçamento aberto e, como chave da tabela, o tipo do produto. Suponha que existam apenas os tipos informados abaixo (em número de 4, use a ordem alfabética para organizar a ordem da tabela *hashing*) e que os dados de cada entrada da tabela sejam armazenados na forma de uma lista encadeada.

Menu

- Inserir produto
- Consultar todos os produtos cadastrados de um tipo
- Contar quantos produtos estão cadastrados em cada tipo
- Sair

Observações:

- Na opção 1, deve ser inserido um produto de cada vez, sendo que para cada um deles o usuário deve fornecer a descrição e o tipo (A – Alimentação, H – Higiene, L – Limpeza e V – Vestuário).
- Na opção 2, o usuário deve digitar a letra que corresponde ao tipo a ser consultado, e todos os produtos do tipo fornecido devem ser listados (listar apenas a descrição). Caso não tenha nenhum produto do tipo fornecido, mostrar mensagem.
- Na opção 3, o programa deve mostrar quantos produtos estão cadastrados em cada tipo.

Exemplo:

Alimentação – 3 produtos

Higiene – 0 produto

Limpeza – 1 produto

Vestuário – 2 produtos

2. Faça um programa que apresente o menu de opções abaixo. As operações devem ser implementadas em uma tabela *hashing*. Use o método da divisão. Como chave da tabela, utilize o salário do funcionário. Suponha que a tabela possua tamanho 20 e que os dados de cada entrada da tabela sejam armazenados na forma de uma lista encadeada.

Menu

**1** – Cadastrar funcionário

**2** – Conceder aumento percentual para todos os funcionários

**3** – Consultar a soma salarial dos funcionários com salário superior a 500

**4** – Consultar todos os funcionários

**5** – Excluir por nome

**6** – Sair

3. Faça um programa que apresente o menu de opções abaixo. As operações devem ser implementadas em uma tabela *hashing*. Use o método da divisão. Como chave da tabela, utilize o código do aluno. Suponha que a tabela possua tamanho 15 e que os dados de cada entrada sejam armazenados na forma de uma lista encadeada.

Menu

**1** – Cadastrar aluno (código, nome e nota final)

**2** – Consultar aprovados (nota final de no mínimo 7)

**3** – Consultar todos os alunos

**4** – Sair

4. Faça um programa que apresente o menu de opções abaixo. As operações devem ser implementadas em uma tabela *hashing*. Utilize endereçamento aberto e, como chave da tabela, a primeira letra do nome. Suponha que a tabela possua tamanho 26 e que os dados de cada entrada sejam armazenados na forma de uma lista encadeada.

Menu

**1** – Inserir (apenas o nome)

**2** – Consultar todas as pessoas

**3** – Consultar uma pessoa

**4** – Consultar as pessoas com uma inicial digitada

**5** – Excluir uma pessoa

**6** – Sair

5. Faça um programa que apresente o menu de opções abaixo. As operações devem ser implementadas em uma tabela *hashing*. Utilize como chave da tabela o mês de aniversário. Suponha que a tabela possua tamanho 20 e que seja implementada como uma de endereçamento aberto (tentativa linear).

Menu

**1** – Cadastrar amigo (nome, dia e mês do aniversário)

**2** – Consultar aniversariantes de um mês

**3** – Contar as pessoas com idade superior a 18

**4** – Excluir uma pessoa pelo nome

**5** – Excluir as pessoas de um mês

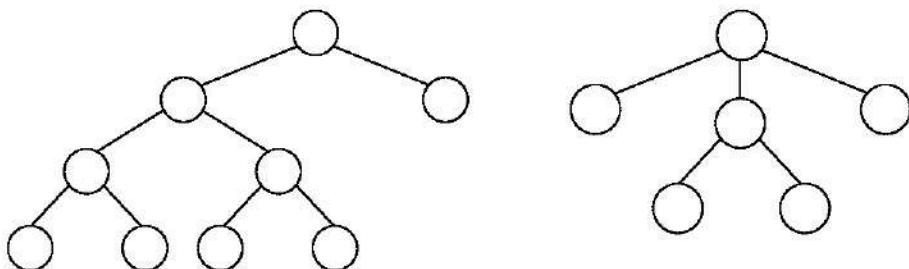
**6** – Sair

# 7

# Estruturas de dados do tipo árvore

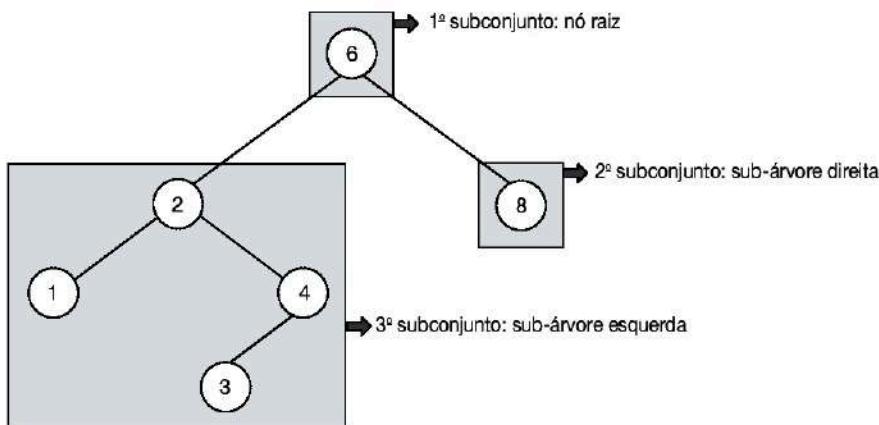
As estruturas de dados do tipo árvore são não lineares, ou seja, os elementos que as compõem não estão armazenados de forma sequencial e também não estão todos encadeados. A Figura 7.1 ilustra duas árvores.

Figura 7.1 Árvores

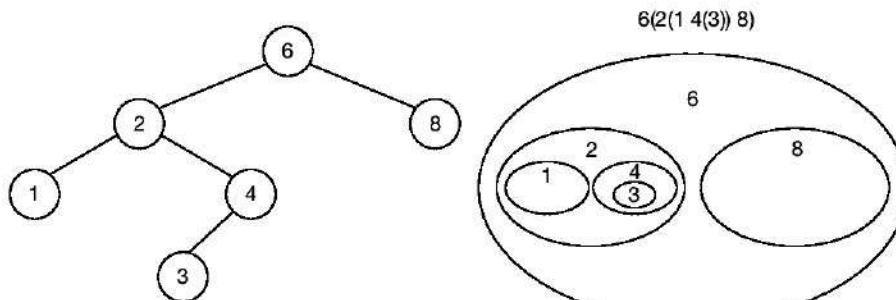


## Árvore binária

Uma árvore binária é um conjunto finito de elementos, onde cada elemento é denominado nó e o primeiro é conhecido como raiz da árvore. Esse conjunto pode estar vazio ou ser particionado em três subconjuntos distintos, sendo eles: 1<sup>a</sup> subconjunto (nó raiz), 2<sup>a</sup> subconjunto (sub-árvore direita) e 3<sup>a</sup> subconjunto (sub-árvore esquerda), como mostra a Figura 7.2.

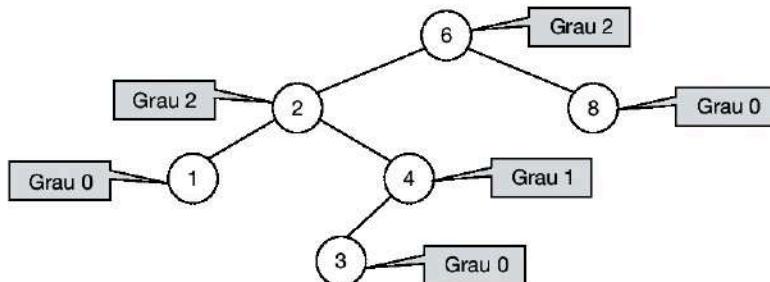
**Figura 7.2** Árvore binária

As árvores binárias podem ser ilustradas de três formas distintas, conforme a Figura 7.3.

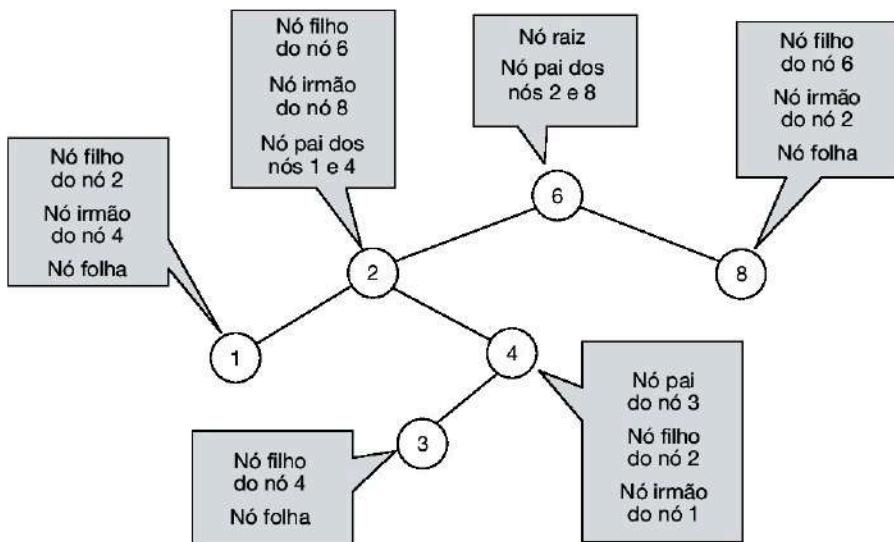
**Figura 7.3** Ilustrações distintas de uma árvore binária

Toda árvore binária possui as seguintes propriedades:

- Todos os nós de uma sub-árvore direita são maiores que o nó raiz.
- Todos os nós de uma sub-árvore esquerda são menores que o nó raiz.
- Cada sub-árvore é também uma árvore binária.
- O grau de um nó representa o seu número de sub-árvore. A Figura 7.4 mostra os graus dos nós de uma árvore binária.

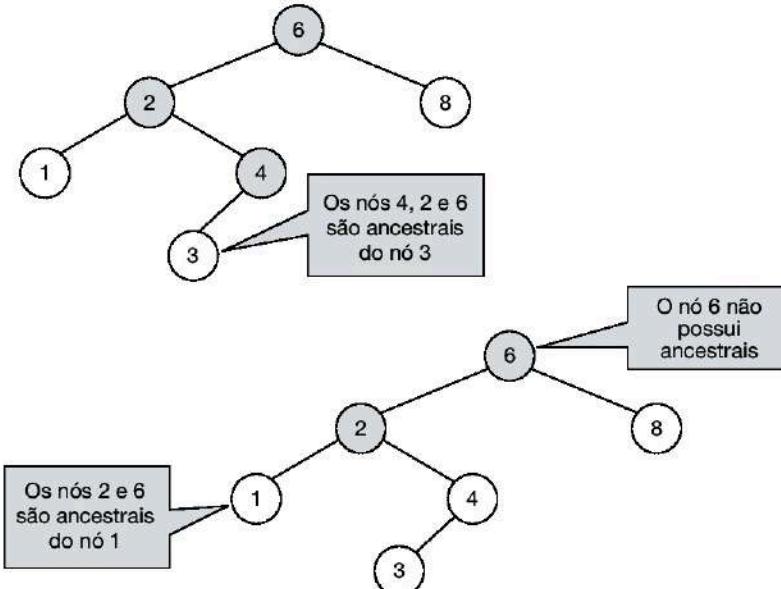
**Figura 7.4** Graus dos nós de uma árvore binária

- e) Em uma árvore binária, o grau máximo de um nó é 2.
- f) O grau de uma árvore é igual ao máximo dos graus de todos os seus nós.
- g) Uma árvore binária tem grau máximo igual a 2.
- h) Nô pai, Figura 7.5: nó acima e com ligação direta a outro nó.
- i) Nô filho, Figura 7.5: nó abaixo e com ligação direta a outro nó. São os nós raízes das sub-árvore.
- j) Nô irmãos, Figura 7.5: são os nós que possuem o mesmo nô pai.
- k) Nô folha ou terminal, Figura 7.5: nô que nô possui filhos.

**Figura 7.5** Nós raiz, pâi, filho, irmãos e folha

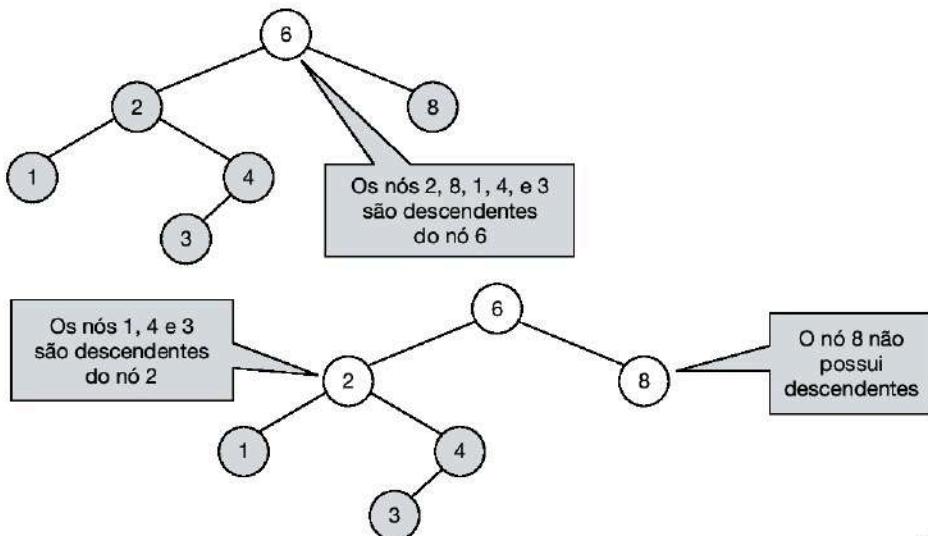
- I) Nó ancestral, Figura 7.6: são os nós que estão acima de um nó e possuem ligação direta ou indireta.

**Figura 7.6** • Nós ancestrais



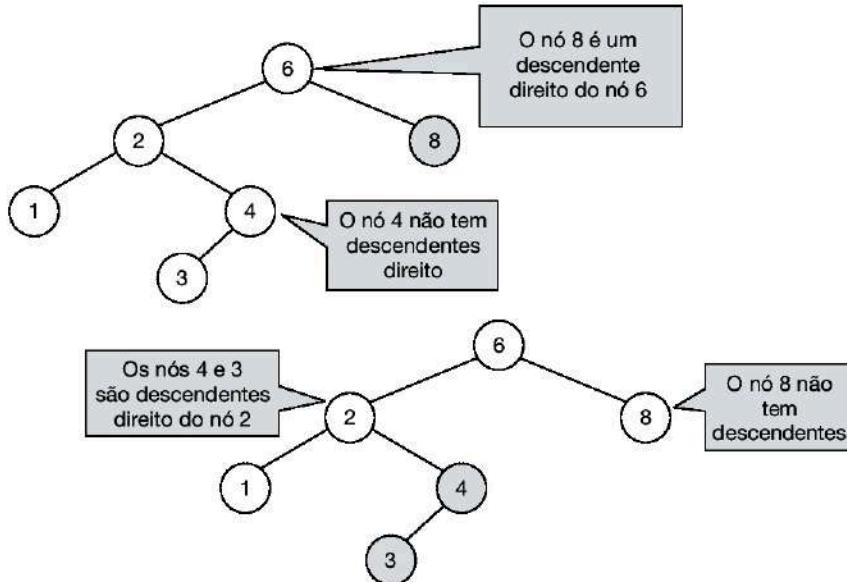
- m) Nó descendente, Figura 7.7: são os nós que estão abaixo de um nó e possuem ligação direta ou indireta.

**Figura 7.7** • Nós descendentes



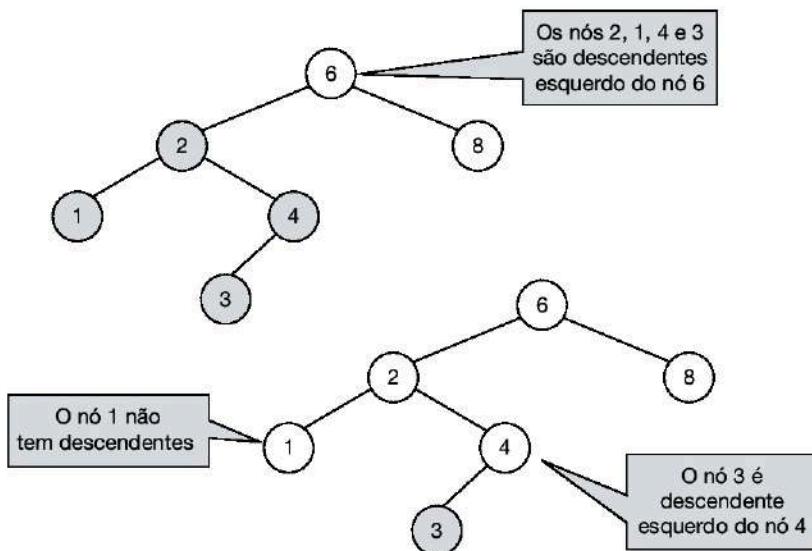
- n) Nós descendentes direito, Figura 7.8: são os nós que estão abaixo de um nó, possuem ligação direta ou indireta e fazem parte da sub-árvore direita.

**Figura 7.8** Nós descendentes direito

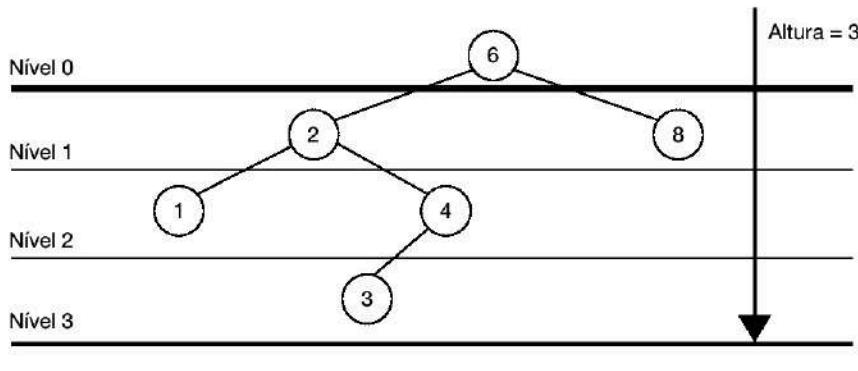


- o) Nós descendentes esquerdo, Figura 7.9: são os nós que estão abaixo de um nó, possuem ligação direta ou indireta e fazem parte da sub-árvore esquerda.

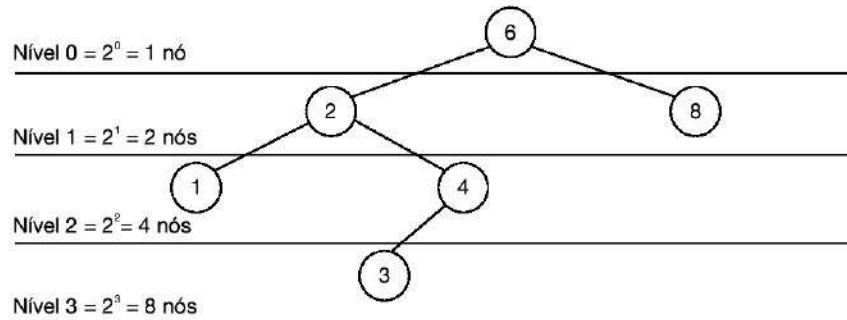
**Figura 7.9** Nós descendentes esquerdo



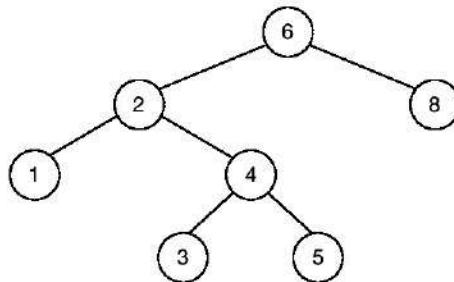
- p) Nível de um nó, Figura 7.10: é a sua distância do nó raiz. Logo, o nível do nó raiz é sempre zero.
- q) Altura ou profundidade de uma árvore, Figura 7.10: é o nível do nó mais distante da raiz.

**Figura 7.10** Nível e altura

- r) Expressão que representa o número máximo de nós em um nível da árvore binária =  $2^n$ , onde  $n$  é o nível em questão, Figura 7.11.

**Figura 7.11** Número máximo de nós em um nível

- s) Árvore estritamente binária, Figura 7.12: árvore em que todos os nós têm 0 ou 2 filhos.
- t) Expressão que representa o número de nós de uma árvore estritamente binária =  $2n - 1$ , onde  $n$  é o número de nós folha, Figura 7.12.

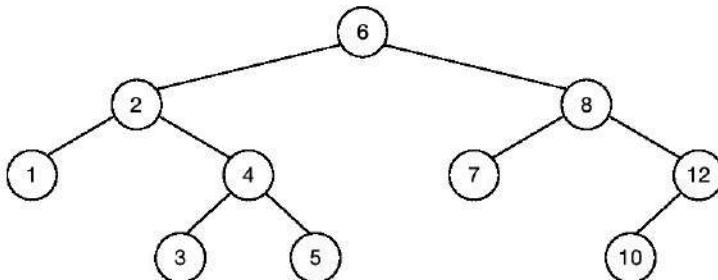
**Figura 7.12** Árvore estritamente binária

Quantidade de nós

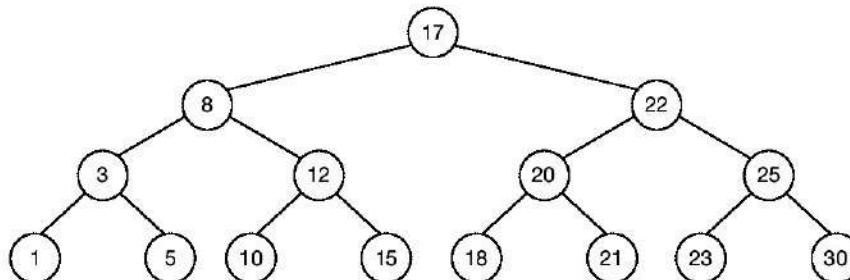
folha = 4.

Os nós folha são:  
1, 3, 5 e 8.Número de nós desta  
árvore estritamente  
binária =  $2 \cdot n - 1$ , onde  
 $n$  é o número de folhas  
 $2 \cdot 4 - 1 = 7$  nós

- u) Árvore completa, Figura 7.13: árvore em que todos os nós com menos de dois filhos ficam no último e no penúltimo nível.

**Figura 7.13** Árvore completa

- v) Árvore cheia, Figura 7.14: árvore estritamente binária e completa.

**Figura 7.14** Árvore cheia

Neste tipo de estrutura serão abordadas as seguintes operações: inserir um nó na árvore, removê-lo, consultar os nós da árvore em ordem, consultar em pré-ordem, consultar em pós-ordem e esvaziar a árvore.

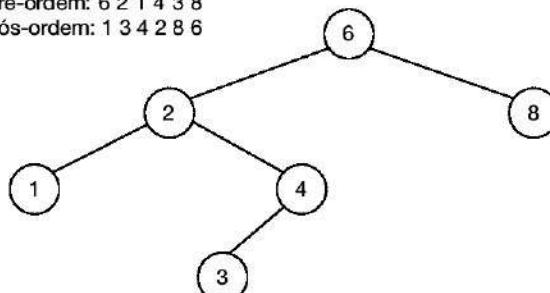
Na operação de inserção, as propriedades de uma árvore devem ser obedecidas e todo novo nó é sempre uma folha. Na operação de remoção, o filho da direita, que é o mais velho, assume o lugar do nó pai.

Nas operações de consulta, em ordem, pré-ordem e pós-ordem, todos os nós da árvore são listados, alterando-se apenas sua ordem. Na consulta em ordem, cada árvore é mostrada com o ramo da esquerda, a raiz e posteriormente o ramo da direita. Na consulta pré-ordem, cada árvore é mostrada com a raiz, o ramo da esquerda e posteriormente o ramo da direita. Na consulta pós-ordem, cada árvore é mostrada com o ramo da esquerda, o ramo da direita e posteriormente a raiz.

A Figura 7.15 ilustra as consultas dos nós de uma árvore binária em ordem, em pré-ordem e em pós-ordem.

**Figura 7.15** Consultas em um árvore binária

Em ordem: 1 2 3 4 6 8  
Pré-ordem: 6 2 1 4 3 8  
Pós-ordem: 1 3 4 2 8 6

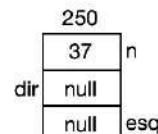
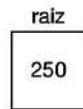


ILUSTRAÇÃO

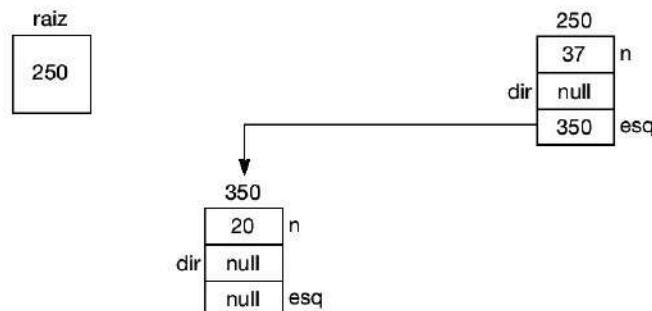
**1<sup>a</sup> operação**  
A árvore binária está vazia



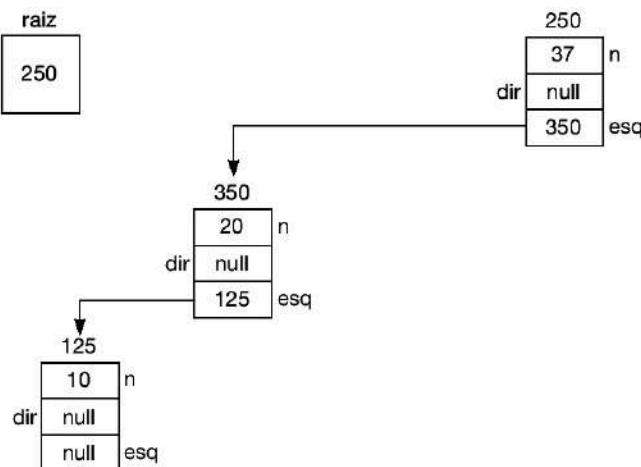
**2<sup>a</sup> operação**  
Inserção do nº 37



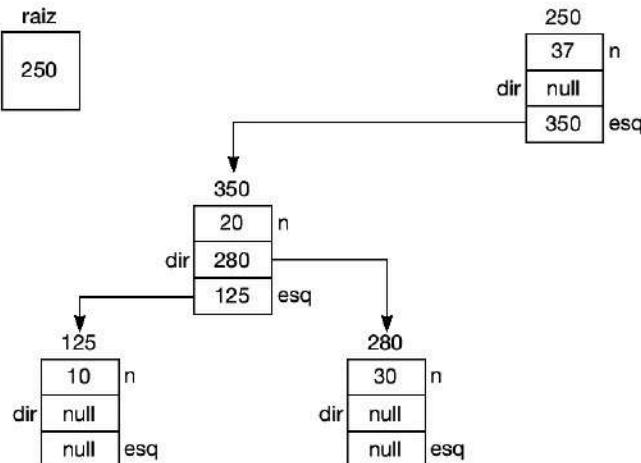
**3<sup>a</sup> operação  
Inserção do nº 20**

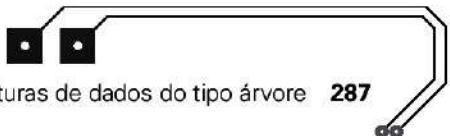


**4<sup>a</sup> operação  
Inserção do nº 10**

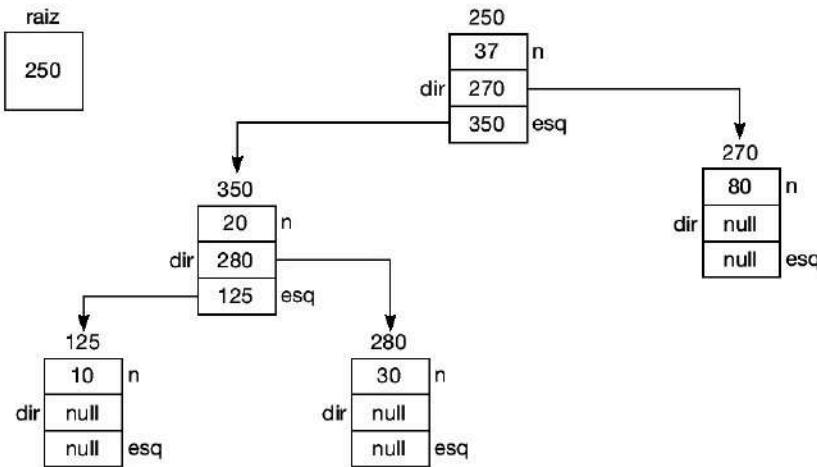


**5<sup>a</sup> operação  
Inserção do nº 30**

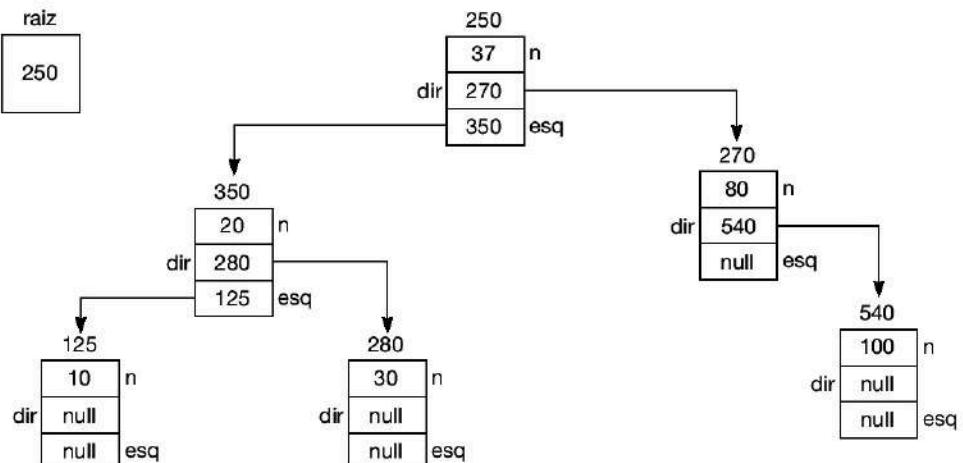




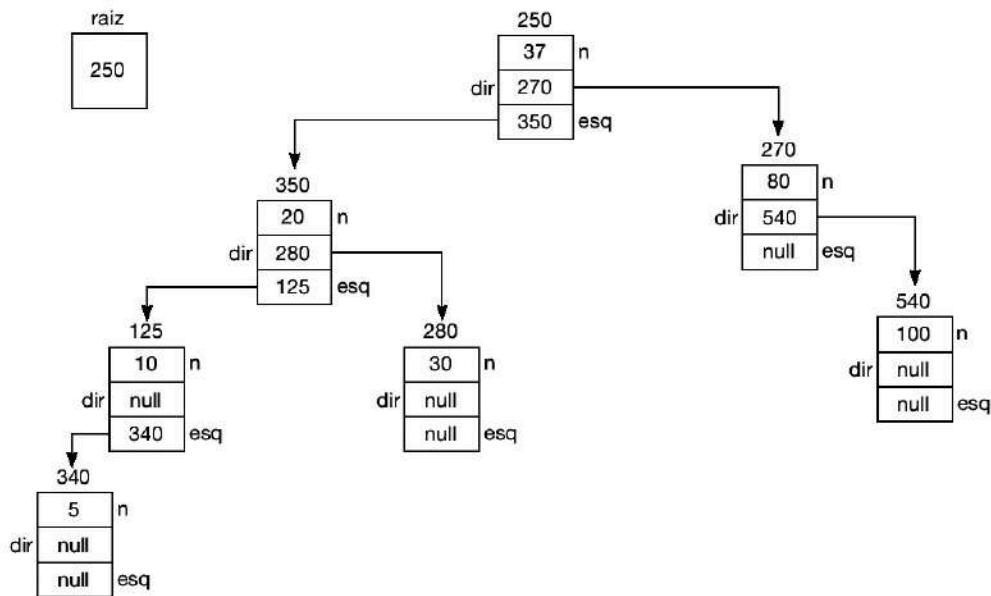
**6<sup>a</sup> operação**  
**Inserção do nº 80**



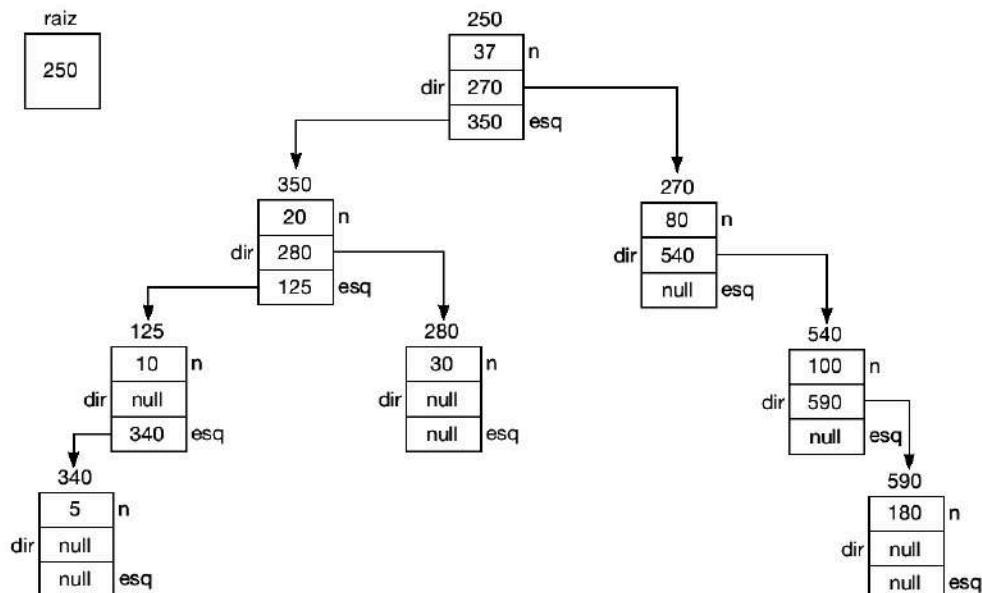
**7<sup>a</sup> operação**  
**Inserção do nº 100**

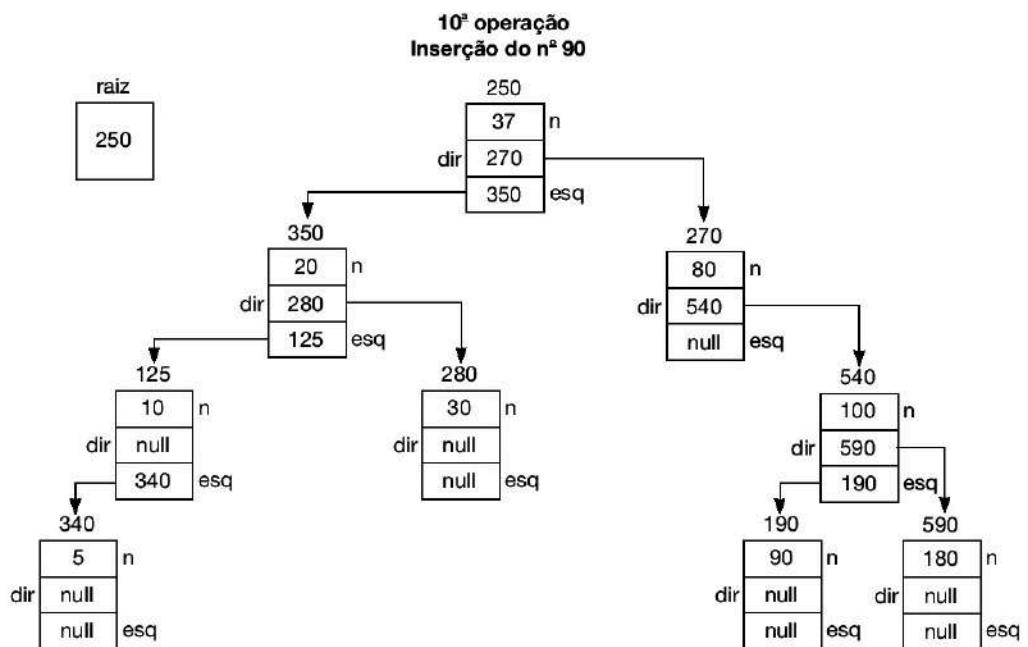


**8<sup>a</sup> operação  
Inserção do nº 5**



**9<sup>a</sup> operação  
Inserção do nº 180**





**A árvore acima é estritamente binária?**

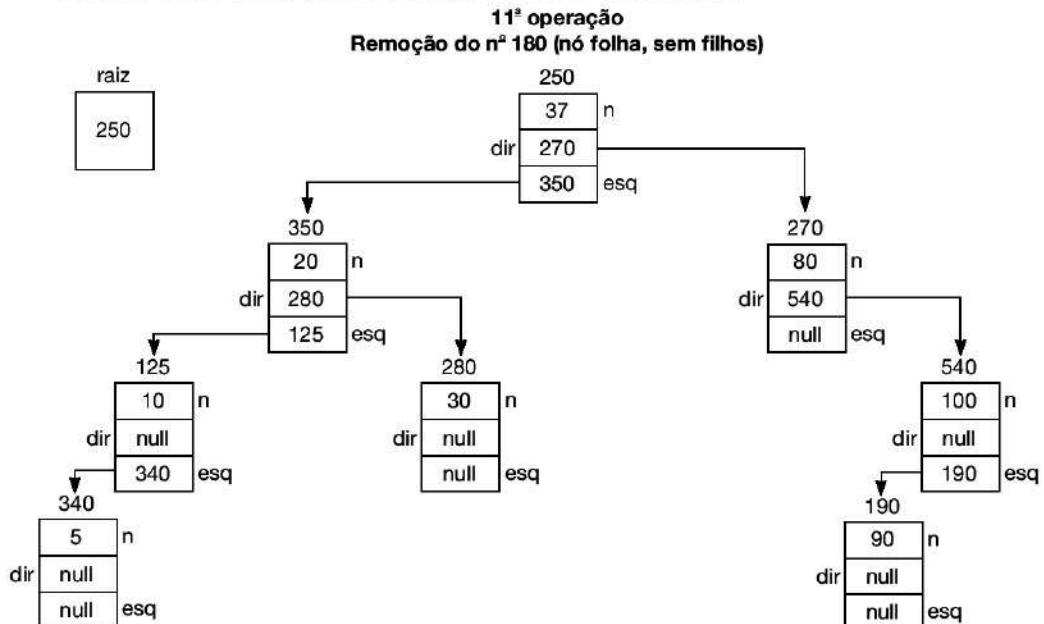
Não. Para ser estritamente binária, todos os nós devem ter 0 ou 2 filhos. Os nós 10 e 80 não obedecem essa propriedade.

**A árvore acima é completa?**

Não. Para ser completa, todos os nós com menos de dois filhos devem estar no último e no penúltimo nível. O nó 80 não obedece essa propriedade.

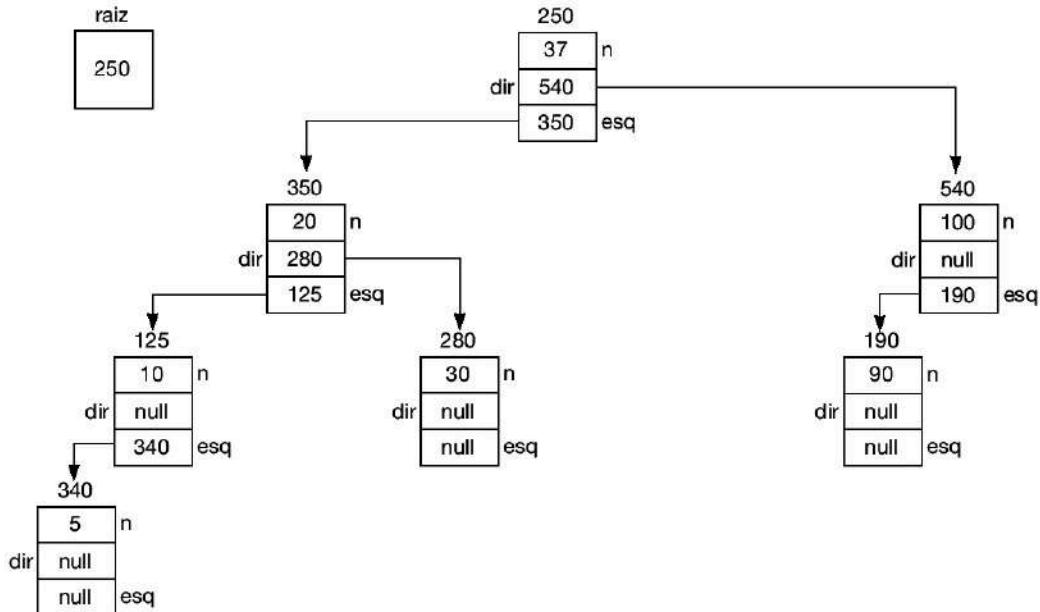
**A árvore acima é cheia?**

Não. Para ser cheia, a árvore deve ser binária e completa.



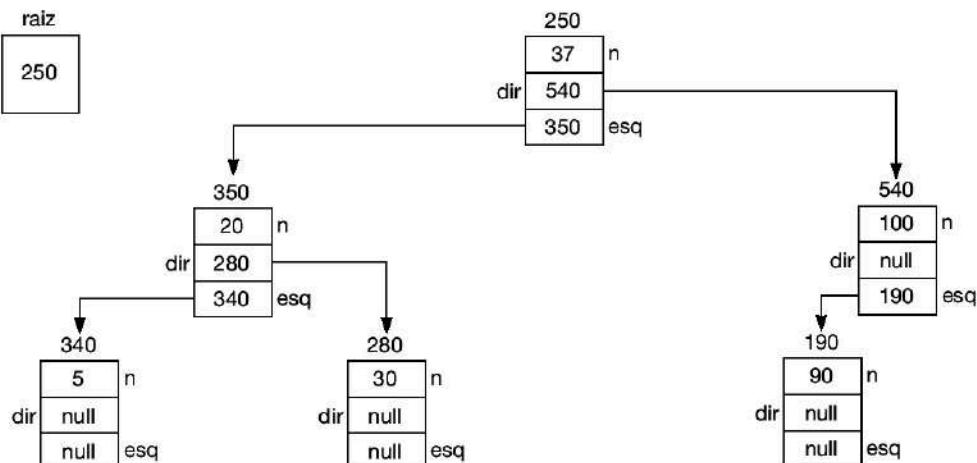
12<sup>a</sup> operação

Remoção do nº 80 (nó com filho apenas para a direita)



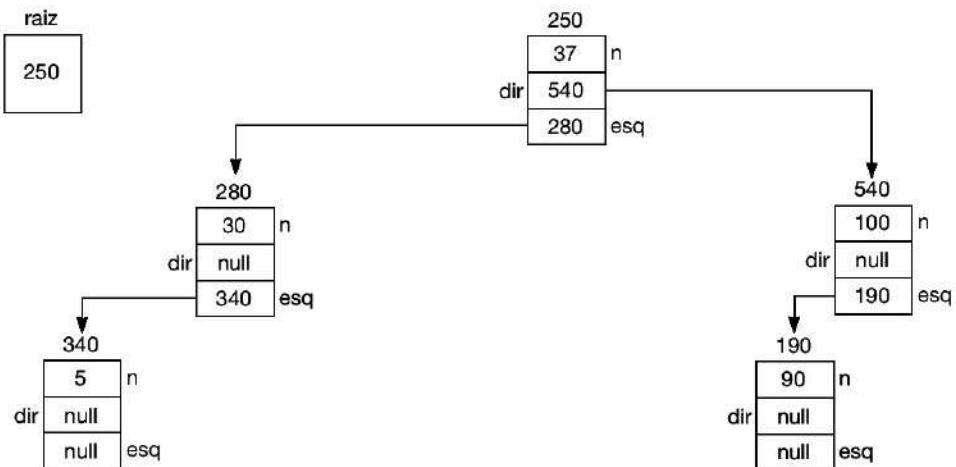
13<sup>a</sup> operação

Remoção do nº 10 (nó com filho apenas para a esquerda)

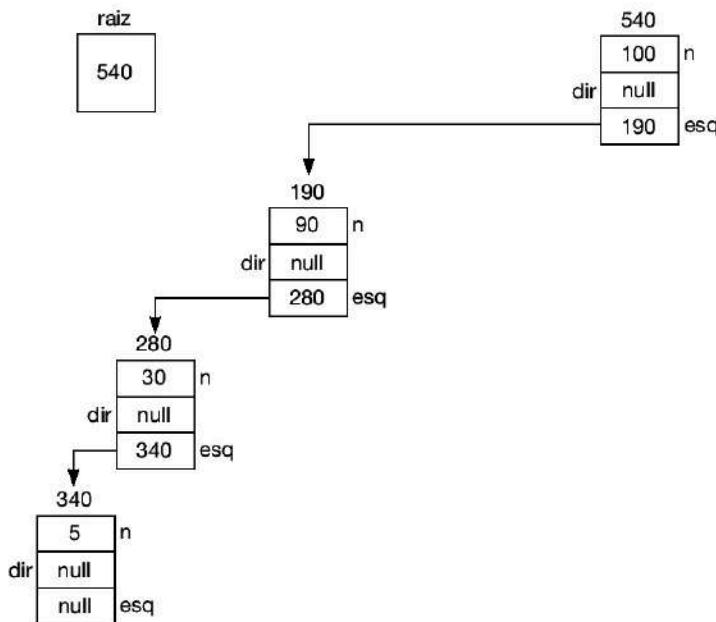


14<sup>a</sup> operação

Remoção do nº 20 (nó com filhos para a direita e para a esqueda)

15<sup>a</sup> operação

Remoção do nº 37 (nó raiz)



A seguir, apresentamos uma implementação em JAVA e em C/C++ das operações em uma árvore binária sem recursividade.



```
import java.util.*;
public class Arvore_Binaria_NRecursiva
{
 //Definindo a classe que representará
 // cada elemento da árvore binária
 private static class ARVORE
 {
 public int num;
 public ARVORE dir, esq;
 }

 //Definindo a classe que representará
 // cada elemento da árvore binária na pilha
 private static class PILHA
 {
 public ARVORE num;
 public PILHA prox;
 }

 public static void main(String args[])
 {
 Scanner entrada = new Scanner(System.in);
 // a árvore está vazia, logo,
 // o objeto raiz tem o valor null
 ARVORE raiz = null;
 // o objeto aux é um objeto auxiliar
 ARVORE aux;
 // o objeto aux1 é um objeto auxiliar
 ARVORE aux1;
 // o objeto novo é um objeto auxiliar
 ARVORE novo;
 // o objeto anterior é um objeto auxiliar
 ARVORE anterior;
 // o objeto topo representa a topo da pilha
 PILHA topo;
 // o objeto aux_pilha é um objeto auxiliar da pilha
 PILHA aux_pilha;
 // apresentando o menu de opções
 int op, achou, numero;
 do
 {
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir na árvore");
```

```
System.out.println("2 - Consultar um nó da árvore");
System.out.println("3 - Consultar toda a árvore em
 ↵ordem");
System.out.println("4 - Consultar toda a árvore em
 ↵pré-ordem");
System.out.println("5 - Consultar toda a árvore em
 ↵pós-ordem");
System.out.println("6 - Excluir um nó da árvore");
System.out.println("7 - Esvaziar a árvore");
System.out.println("8 - Sair");
System.out.print("Digite sua opção: ");
op = entrada.nextInt();
if (op < 1 || op > 8)
 System.out.println("Opção inválida!!!");
if (op == 1)
{
 System.out.println("Digite o número a ser
 ↵inserido na árvore: ");
 novo = new ARVORE();
 novo.num = entrada.nextInt();
 novo.dir = null;
 novo.esq = null;
 if (raiz == null)
 raiz = novo;
 else
 {
 aux = raiz;
 achou = 0;
 while (achou == 0)
 {
 if (novo.num < aux.num)
 {
 if (aux.esq == null)
 {
 aux.esq = novo;
 achou = 1;
 }
 else aux = aux.esq;
 }
 else if (novo.num >= aux.num)
 {
 if (aux.dir == null)
 {
 aux.dir = novo;
 achou = 1;
 }
 }
 }
 }
}
```

```
 }
 else aux = aux.dir;
 }
}
}
}
System.out.println("Número inserido na
 árvore!!");
}
if (op == 2)
{
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
 }
 else
 {
 // a árvore contém elementos
 System.out.println("Digite o elemento a
 ser consultado");
 numero = entrada.nextInt();
 achou = 0;
 aux = raiz;
 while (aux != null && achou == 0)
 {
 if (aux.num == numero)
 {
 System.out.print("Número
 encontrado na árvore!");
 achou = 1;
 }
 else if (numero < aux.num)
 aux = aux.esq;
 else
 aux = aux.dir;
 }
 if (achou == 0)
 System.out.println("Número não encontrado
 na árvore!");
 }
}
if (op == 3)
{
 if (raiz == null)
 {
```

```
// a árvore está vazia
System.out.println("Árvore vazia!!");

}

else
{
 // a árvore contém elementos
 // e estes serão mostrados em ordem
 System.out.println("Listando todos os elementos
 da árvore em ordem");
 aux = raiz;
 // a pilha, uma estrutura auxiliar, está vazia
 topo = null;
 do
 { // caminha na árvore pelo
 // ramo da esquerda até null
 // colocando cada elemento
 // visitado em uma PILHA
 while (aux != null)
 {
 aux_pilha = new PILHA();
 aux_pilha.num = aux;
 aux_pilha.prox = topo;
 topo = aux_pilha;
 aux=aux.esq;
 }
 if (topo != null)
 {
 aux_pilha = topo;
 System.out.print(aux_pilha.num.num+" ");
 aux = topo.num.dir;
 topo = topo.prox;
 }
 } while(topo != null || aux != null);
}
}

if (op == 4)
{
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
 }
 else
 {
 // a árvore contém elementos
 // e estes serão mostrados em pré-ordem
```

```
System.out.println("Listando todos os elementos
 da árvore em pré-ordem");
aux = raiz;
// a pilha, uma estrutura auxiliar, está vazia
topo = null;
do
{ // caminha na árvore mostrando
 // cada nó visitado
 // colocando cada elemento
 // visitado em uma PILHA
 while (aux != null)
 {
 aux_pilha = new PILHA();
 System.out.print(aux.num+" ");
 aux_pilha.num = aux;
 aux_pilha.prox = topo;
 topo = aux_pilha;
 aux=aux.esq;
 }
 if (topo != null)
 {
 aux_pilha = topo;
 topo = topo.prox;
 aux = aux_pilha.num.dir;
 }
} while(topo != null || aux != null);
}
}
if (op == 5)
{
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
 }
 else
 {
 // a árvore contém elementos
 // e estes serão mostrados em pós-ordem
 System.out.println("Listando todos os
 elementos da árvore em
 pós-ordem");
 aux = raiz;
 // a pilha, uma estrutura auxiliar,
 // está vazia
 topo = null;
```

```
do
{
 // caminha na árvore
 // pelo ramo da esquerda até null
 // colocando cada elemento
 // visitado em uma PILHA
 // antes de colocar a raiz
 // de cada sub-árvore na pilha
 // caminha também no ramo da direita
 do
 {
 while (aux != null)
 {
 aux_pilha = new PILHA();
 aux_pilha.num = aux;
 aux_pilha.prox = topo;
 topo = aux_pilha;
 aux=aux.esq;
 }
 if (topo.num.dir != null)
 {
 aux = topo.num.dir;
 }
 }
 while (aux != null);
 if (topo != null)
 {
 System.out.print(topo.num.num+" ");
 if (topo.prox != null)
 {
 // existe mais de um
 // elemento empilhado
 if (topo.prox.num.dir != null
 &&
 topo.prox.num.dir != topo.num)
 {
 aux = topo.prox.num.dir;
 topo = topo.prox;
 }
 else
 {
 while (topo.prox != null
 &&
 topo.prox.num.dir ==
 topo.num)
 {
 topo = topo.prox;
 System.out.print (topo.
```

```
 num.num += " ");
 }
 topo = topo.prox;
 if (topo != null)
 aux = topo.num.dir;
 else
 aux = null;
}
}
else
{
 topo = topo.prox;
 aux = null;
}
}
}
while(topo != null || aux != null);
}
}
if (op == 6)
{
 if (raiz == null)
 System.out.println("Árvore vazia!!!");
 else
 {
 System.out.println("Digite o número que deseja "
 "excluir");
 numero = entrada.nextInt();
 aux = raiz;
 achou = 0;
 while (achou == 0 && aux != null)
 {
 if (aux.num == numero)
 achou = 1;
 else if (aux.num > numero)
 // o número está à esquerda da árvore
 aux = aux.esq;
 else
 // o número está à
 // direita da árvore
 aux = aux.dir;
 }
 if (achou == 0)
 System.out.println("Número não encontrado");
 else
 {
 if (aux != raiz)
```

```
{
 // o número foi encontrado,
 // será excluído e não é a raiz
 // é necessário encontrar o anterior
 // para acertar os ponteiros
 // anterior = elemento que aponta
 // para o número a ser excluído
 anterior = raiz;
 while (anterior.dir != aux
 && anterior.esq != aux)
 {
 if (anterior.num > numero)
 // o número está à
 // esquerda da árvore
 anterior = anterior.esq;
 else // o número está à
 // direita da árvore
 anterior = anterior.dir;
 }
 if (aux.dir == null && aux.esq == null)
 { // um número folha será excluído
 if (anterior.dir == aux)
 anterior.dir = null;
 else
 anterior.esq = null;
 }
 else
 {
 // um número não folha
 // será excluído
 if (aux.dir != null
 && aux.esq == null)
 { // um número que possui
 // filhos apenas para a direita
 if (anterior.esq == aux)
 anterior.esq = aux.dir;
 else
 anterior.dir = aux.dir;
 }
 else if (aux.esq != null
 && aux.dir == null)
 { // um número que possui
 // filhos apenas para
 // à esquerda
 if (anterior.esq == aux)
 anterior.esq = aux.esq;
 else
```

```
anterior.dir = aux.esq;
}
else if (aux.esq != null &&
 aux.dir != null)
{
 // um número que possui
 // filhos para a esquerda
 // e para a direita
 if (anterior.dir == aux)
 {
 anterior.dir = aux.dir;
 aux1 = aux.esq;
 }
 else
 {
 anterior.esq = aux.esq;
 aux1 = aux.dir;
 }
 // recolocando o pedaço
 // da árvore
 aux = anterior;
 while (aux != null)
 {
 if (aux.num < aux1.num)
 {
 if (aux.dir == null)
 {
 aux.dir = aux1;
 aux = null;
 }
 else aux = aux.dir;
 }
 else if (aux.num > aux1.num)
 {
 if (aux.esq == null)
 {
 aux.esq = aux1;
 aux = null;
 }
 else aux = aux.esq;
 }
 }
}
else
{
```

```
// o número a ser excluído é a raiz
if (aux.dir == null &&
 aux.esq == null)
{
 // a raiz não tem filhos
 // e será excluída
 raiz = null;
}
else
{
 if (aux.dir != null
 && aux.esq == null)
 {
 // a raiz será excluída
 // e possui filhos
 // apenas para à direita
 raiz = aux.dir;
 }
 else if (aux.esq != null
 && aux.dir == null)
 {
 // a raiz será excluída
 // e possui filhos
 // apenas para à esquerda
 raiz = aux.esq;
 }
 else if (aux.esq != null
 && aux.dir != null)
 {
 // a raiz será
 // excluída
 // e possui filhos
 // para à direita e
 // para à esquerda
 raiz = aux.dir;
 aux1 = aux.esq;
 // recolocando
 // o pedaço
 // da árvore
 aux = raiz;
 while (aux != null)
 {
 if (aux.num < aux1.num)
 {
 if (aux.dir == null)
 {
 aux.dir = aux1;
 aux = null;
 }
 }
 else aux = aux.dir;
```

```
 }
 else if(aux.num>aux1.num)
 {
 if (aux.esq == null)
 {
 aux.esq = aux1;
 aux = null;
 }
 else aux = aux.esq;
 }
 }
}
System.out.println("Número excluído da árvore!");
}
}
while (op != 8);
}
}
```

---



```
#include<iostream.h>
#include<conio.h>

// Definindo o registro que representará
// cada elemento da árvore binária
struct ARVORE
{
 int num;
 ARVORE *esq, *dir;
};

//Definindo o registro que representará
// cada elemento da árvore binária na pilha
struct PILHA
{
 ARVORE *num;
 PILHA *prox;
};

void main()
{
 // a árvore está vazia, logo,
 // o ponteiro raiz tem o valor null
 ARVORE *raiz = NULL;
 // o ponteiro aux é um ponteiro auxiliar
 ARVORE *aux;
 // o ponteiro aux1 é um ponteiro auxiliar
 ARVORE *aux1;
 // o ponteiro novo é um ponteiro auxiliar
 ARVORE *novo;
 // o ponteiro anterior é um ponteiro auxiliar
 ARVORE *anterior;
 // o ponteiro topo representa a topo da pilha
 PILHA *topo;
 // o ponteiro aux_pilha é um ponteiro
 // auxiliar da pilha
 PILHA *aux_pilha;
 // apresentando o menu de opções
 int op, achou, numero;
 do
 {
 clrscr();
 cout << "\nMENU DE OPÇÕES\n";
 cout << "\n1 - Inserir na árvore";
```

```
cout << "\n2 - Consultar um nó da árvore";
cout << "\n3 - Consultar toda a árvore em
 ↵ordem";
cout << "\n4 - Consultar toda a árvore em pré-
 ↵ordem";
cout << "\n5 - Consultar toda a árvore em pós-
 ↵ordem";
cout << "\n6 - Excluir um nó da árvore";
cout << "\n7 - Esvaziar a árvore";
cout << "\n8 - Sair";
cout << "\nDigite sua opção: ";
cin >> op;
if (op < 1 || op > 8)
 cout << "\nOpção inválida!!";
else if (op == 1)
{
 cout << "\nDigite o número a ser
 ↵inserido na árvore:";
 novo = new ARVORE();
 cin >> novo->num;
 novo->dir = NULL;
 novo->esq = NULL;
 if (raiz == NULL)
 raiz = novo;
 else
 {
 aux = raiz;
 achou = 0;
 while (achou == 0)
 {
 if (novo->num < aux->num)
 {
 if (aux->esq == NULL)
 {
 aux->esq = novo;
 achou = 1;
 }
 else aux = aux->esq;
 }
 else if (novo->num >= aux->num)
 {
 if (aux->dir == NULL)
 {
```

```
 aux->dir = novo;
 achou = 1;
}
else aux = aux->dir;
}

}

cout << "\nNúmero inserido na árvore!!";
}

else if (op == 2)
{
 if (raiz == NULL)
 {
 // a árvore está vazia
 cout << "\nÁrvore vazia!!";
 }
 else
 {
 // a árvore contém elementos
 cout << "\nDigite o elemento a ser
 ↪consultado";
 cin >> numero;
 achou = 0;
 aux = raiz;
 while (aux != NULL && achou == 0)
 {
 if (aux->num == numero)
 {
 cout << "\nNúmero encontrado na
 ↪árvore!";
 achou = 1;
 }
 else if (numero < aux->num)
 aux = aux->esq;
 else
 aux = aux->dir;
 }
 if (achou == 0)
 cout << "\nNúmero não encontrado
 ↪na árvore!";
 }
}

else if (op == 3)
{
 if (raiz == NULL)
 {
```

```
// a árvore está vazia
cout << "\nÁrvore vazia!!";
}
else
{
 // a árvore contém elementos
 // e estes serão mostrados em ordem
 cout << "\nListando todos os elementos
 da árvore em ordem\n";
 aux = raiz;
 // a pilha, uma estrutura auxiliar,
 // está vazia
 topo = NULL;
 do
 {
 // caminha na árvore pelo ramo
 // da esquerda até NULL
 // colocando cada elemento
 // visitado em uma PILHA
 while (aux != NULL)
 {
 aux_pilha = new PILHA();
 aux_pilha->num = aux;
 aux_pilha->prox = topo;
 topo = aux_pilha;
 aux=aux->esq;
 }
 if (topo != NULL)
 {
 aux_pilha = topo;
 cout << aux_pilha->num->num << " ";
 aux = topo->num->dir;
 topo = topo->prox;
 }
 }while(topo != NULL || aux != NULL);
}
else if (op == 4)
{
 if (raiz == NULL)
 {
 // a árvore está vazia
 }
}
```

```
cout << "\nÁrvore vazia!!";
```

```
}
```

```
else
```

```
{
```

```
// a árvore contém elementos e
// estes serão mostrados em pré-ordem
cout << "\nListando todos os elementos
da árvore em pré-ordem\n";
```

```
aux = raiz;
```

```
// a pilha, uma estrutura auxiliar,
// está vazia
```

```
topo = NULL;
```

```
do
```

```
{
```

```
// caminha na árvore mostrando
```

```
// cada nó visitado
```

```
// colocando cada elemento
visitado
```

```
// em uma PILHA
```

```
while (aux != NULL)
```

```
{
```

```
aux_pilha = new PILHA();
```

```
cout << aux->num << " ";
```

```
aux_pilha->num = aux;
```

```
aux_pilha->prox = topo;
```

```
topo = aux_pilha;
```

```
aux=aux->esq;
```

```
}
```

```
if (topo != NULL)
```

```
{
```

```
aux_pilha = topo;
```

```
topo = topo->prox;
```

```
aux = aux_pilha->num->dir;
```

```
}
```

```
}while(topo != NULL ||
aux != NULL);
```

```
}
```

```
}
```

```
else if (op == 5)
```

```
{
```

```
if (raiz == NULL)
```

```
{
```

```
// a árvore está vazia
cout << "\nÁrvore vazia!!";
}

else
{
 // a árvore contém elementos
 // e estes serão mostrados em pós-ordem
 cout << "\nListando todos os
 //elementos da árvore em pós-ordem\n";
 aux = raiz;
 // a pilha, uma estrutura auxiliar,
 // está vazia
 topo = NULL;
 do
 {
 // caminha na árvore
 // pelo ramo da esquerda até NULL
 // colocando cada elemento visitado
 // em uma PILHA
 // antes de colocar a raiz
 // de cada sub-árvore na pilha
 // caminha também no ramo da direita
 do
 {
 while (aux != NULL)
 {
 aux_pilha = new PILHA();
 aux_pilha->num = aux;
 aux_pilha->prox = topo;
 topo = aux_pilha;
 aux=aux->esq;
 }
 if (topo->num->dir != NULL)
 {
 aux = topo->num->dir;
 }
 }while (aux != NULL);
 if (topo != NULL)
```

```
{
 cout << topo->num->num << " ";
 if (topo->prox != NULL)
 {
 // existe mais de um
 // elemento empilhado
 if (topo->prox->num->dir != NULL
&& topo->prox->num->dir != topo->num)
 {
 aux = topo->prox->num->dir;
 topo = topo->prox;
 }
 else
 {
 while (topo->prox != NULL &&
topo->prox->num->dir == topo->num)
 {
 topo = topo->prox;
 cout<<topo->num->num<<" ";
 }
 topo = topo->prox;
 if (topo != NULL)
 aux = topo->num->dir;
 else
 aux = NULL;
 }
 }
 else
 {
 topo = topo->prox;
 aux = NULL;
 }
}
}
}
}
}
}
}
}
}
}
}
```

```
if (raiz == NULL)
 cout << "\nÁrvore vazia!";
else
{
 cout << "\nDigite o número que
 deseja excluir";
 cin >> numero;
 aux = raiz;
 achou = 0;
 while (achou == 0 && aux != NULL)
 {
 if (aux->num == numero)
 achou = 1;
 else if (aux->num > numero)
 // o número está à
 // esquerda da árvore
 aux = aux->esq;
 else
 // o número está à
 // direita da árvore
 aux = aux->dir;
 }
 if (achou == 0)
 cout << "\nNúmero não encontrado";
else
{
 if (aux != raiz)
 {
 // o número foi encontrado,
 // será excluído e não é a
 // raiz
 // é necessário encontrar o
 // anterior
 // para acertar os ponteiros
 // anterior = elemento que aponta
 // para o número a ser excluído
 anterior = raiz;
 while (anterior->dir != aux
 && anterior->esq != aux)
 {
 if (anterior->num > numero)
 // o número está à
 // esquerda da árvore
 anterior = anterior->esq;
```

```
 else
 // o número está à
 // direita da árvore
 anterior = anterior->dir;
 }

 if (aux->dir == NULL
 && aux->esq == NULL)
 {
 // um número folha será
 // excluído

 if (anterior->dir == aux)
 anterior->dir = NULL;
 else
 anterior->esq = NULL;

 delete (aux);
 }
 else
 {
 // um número não folha será //
 // excluído
 if (aux->dir != NULL
 && aux->esq == NULL)
 {
 // um número que possui
 // filhos apenas
 // para a direita
 if (anterior->esq == aux)
 anterior->esq = aux->dir;
 else
 anterior->dir = aux->dir;
 delete (aux);
 }
 else if (aux->esq != NULL
 && aux->dir == NULL)
 {
 // um número que possui
 // filhos apenas
 // para a esquerda
```

```
if (anterior->esq == aux)
 anterior->esq = aux->esq;
else
 anterior->dir = aux->esq;
delete (aux);
}
else if (aux->esq != NULL
&& aux->dir != NULL)
{
 // um número que
 // possui filhos
 // para a esquerda
 // e para a direita
 if (anterior->dir == aux)
 {
 anterior->dir = aux->dir;
 aux1 = aux->esq;
 }
 else
 {
 anterior->esq = aux->dir;
 aux1 = aux->esq;
 }
 delete (aux);
 // recolocando o pedaço
 // da árvore
 aux = anterior;
 while (aux != NULL)
 {
 if (aux->num < aux1->num)
 {
 if (aux->dir == NULL)
 {
 aux->dir = aux1;
 aux = NULL;
 }
 else aux = aux->dir;
 }
 else if (aux->num>aux1->num)
 {
 if (aux->esq == NULL)
```

```
 {
 aux->esq = aux1;
 aux = NULL;
 }
 else aux = aux->esq;
 }
}
}
}
}
else
{
 // o número a ser excluído é a raiz
 if (aux->dir == NULL && aux->esq == NULL)
 {
 // a raiz não tem filhos e será excluída
 delete aux;
 raiz = NULL;
 }
 else
 {
 if (aux->dir != NULL &&
 aux->esq == NULL)
 {
 // a raiz será excluída
 // e possui filhos
 // apenas para à direita
 raiz = aux->dir;
 delete aux;
 }
 else if (aux->esq != NULL
 && aux->dir == NULL)
 {
 // a raiz será excluída
 // e possui filhos
 // apenas para à esquerda
 raiz = aux->esq;
 delete aux;
 }
 else if (aux->esq != NULL
 && aux->dir != NULL)
 {
 // a raiz será excluída
 // e possui filhos
 // para à direita e
 // para à esquerda
 }
 }
}
```

```

 raiz = aux->dir;
 aux1 = aux->esq;
 // recolocando o pedaço
 // da árvore
 delete aux;
 aux = raiz;
 while (aux != NULL)
 {
 if (aux->num < aux1->num)
 {
 if (aux->dir == NULL)
 {
 aux->dir = aux1;
 aux = NULL;
 }
 else aux = aux->dir;
 }
 else if(aux->num>aux1->num)
 {
 if (aux->esq ==
 NULL)
 {
 aux->esq =
 aux1;
 aux = NULL;
 }
 else aux = aux->esq;
 }
 }
 }
 cout << "\nNúmero excluído da árvore!";
}
}
else if(op==7)
{
 if (raiz == NULL)
 cout<<"\nÁrvore vazia";
 else
 {
 aux = raiz;
 // a pilha, uma estrutura auxiliar, está vazia
 }
}

```

```

topo = NULL;
do
{ // caminha na árvore mostrando cada nó visitado
 // colocando cada elemento visitado em uma PILHA
 while (aux != NULL)
 {
 aux_pilha = new PILHA();
 cout << aux->num << " ";
 aux_pilha->num = aux;
 aux_pilha->prox = topo;
 topo = aux_pilha;
 aux=aux->esq;
 }
 if (topo != NULL)
 {
 aux_pilha = topo;
 topo = topo->prox;
 aux = aux_pilha->num->dir;
 }
}while(topo != NULL || aux != NULL);
// passa por todos os elementos da pilha
// removendo-os
aux_pilha=topo;
while (aux_pilha != NULL)
{
 topo=topo->prox;
 delete(topo->num);
 delete(aux_pilha);
 aux_pilha=topo;
}
raiz=NULL;
cout<<"\nÁrvore esvaziada!!";
}
}
getch();
}while (op != 8);
}

```

A seguir, é demonstrada uma implementação em JAVA e em C/C++ das operações de uma árvore binária com recursividade.



```
 numero = entrada.nextInt();
 raiz = inserir(raiz,numero);
 System.out.println("Número inserido na
 árvore!!");
}
if (op == 2)
{
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
 }
 else
 {
 // a árvore contém elementos
 System.out.println("Digite o elemento a ser
 consultado");
 numero = entrada.nextInt();
 achou = 0;
 achou = consultar(raiz,numero,achou);
 if (achou == 0)
 System.out.println("Número não encontrado
 na árvore!");
 else
 System.out.println("Número encontrado na
 árvore!");
 }
}
if (op == 3)
{
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
 }
 else
 {
 // a árvore contém elementos
 // e estes serão mostrados em ordem
 System.out.println("Listando todos os
 elementos da árvore em ordem");
 mostrarEmOrdem(raiz);
 }
}
```

```
if (op == 4)
{
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
 }
 else
 {
 // a árvore contém elementos
 // e estes serão mostrados em pré-ordem
 System.out.println("Listando todos os elementos
 da árvore em pré-ordem");
 mostrarpreordem(raiz);
 }
}
if (op == 5)
{
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
 }
 else
 {
 // a árvore contém elementos
 // e estes serão mostrados em pós-ordem
 System.out.println("Listando todos os elementos
 da árvore em pós-ordem");
 aux = raiz;
 mostrarpesordem(aux);
 }
}
if (op == 6)
{
 if (raiz == null)
 System.out.println("Árvore vazia!!!");
 else
 {
 System.out.println("Digite o número que deseja
 excluir");
 numero = entrada.nextInt();
 achou = 0;
 achou = consultar(raiz,numero,achou);
 if (achou == 0)
```

```
System.out.println("Número não encontrado na
árvore!");
else
{
 raiz = remover(raiz, numero);
 System.out.println("Número excluído da
árvore!");
}
}
}
if (op == 7)
{
 if (raiz == null)
 System.out.println("Árvore vazia!!");
 else
 {
 raiz = null;
 System.out.println("Árvore esvaziada!!");
 }
}
}
while (op != 8);

public static ARVORE inserir(ARVORE aux, int num)
{
 if (aux == null)
 {
 aux = new ARVORE();
 aux.num = num;
 aux.esq = null;
 aux.dir = null;
 }
 else if (num < aux.num)
 aux.esq = inserir(aux.esq, num);
 else
 aux.dir = inserir(aux.dir, num);
 return aux;
}

public static int consultar(ARVORE aux, int num, int
achou)
{
 if (aux != null && achou == 0)
 {
```

```
 if (aux.num == num)
 {
 achou = 1;
 }
 else if (num < aux.num)
 achou = consultar(aux.esq, num, achou);
 else
 achou = consultar(aux.dir, num, achou);
 }
 return achou;
}

public static void mostraremordem(ARVORE aux)
{
 if (aux != null)
 {
 mostraremordem(aux.esq);
 System.out.print(aux.num+" ");
 mostraremordem(aux.dir);
 }
}

public static void mostrarpreadem(ARVORE aux)
{
 if (aux != null)
 {
 System.out.print(aux.num+" ");
 mostrarpreadem(aux.esq);
 mostrarpreadem(aux.dir);
 }
}

public static void mostraposordem(ARVORE aux)
{
 if (aux != null)
 {
 mostraposordem(aux.esq);
 mostraposordem(aux.dir);
 System.out.print(aux.num+" ");
 }
}

public static ARVORE remover(ARVORE aux, int num)
{
 ARVORE p, p2;
```

```
if (aux.num == num)
{
 if (aux.esq == aux.dir)
 { // o elemento a ser removido não
 // tem filhos
 return null;
 }
 else if (aux.esq == null)
 { // o elemento a ser removido
 // não tem filho para à esquerda
 return aux.dir;
 }
 else if (aux.dir == null)
 { // o elemento a ser removido
 // não tem filho para à direita
 return aux.esq;
 }
 else
 { // o elemento a ser removido
 // tem filho para ambos os lados
 p2= aux.dir;
 p= aux.dir;
 while (p.esq != null)
 {
 p=p.esq;
 }
 p.esq = aux.esq;
 return p2;
 }
}
else if (aux.num < num)
 aux.dir = remover(aux.dir, num);
else
 aux.esq = remover(aux.esq, num);
return aux;
}
```



```
#include<iostream.h>
#include<conio.h>

// Definindo o registro que representará
// cada elemento da árvore binária
struct ARVORE
{
 int num;
 ARVORE *esq, *dir;
};

ARVORE* inserir(ARVORE* aux, int num)
{
 if (aux == NULL)
 {
 aux = new ARVORE();
 aux->num = num;
 aux->esq = NULL;
 aux->dir = NULL;
 }
 else if (num < aux->num)
 aux->esq = inserir(aux->esq, num);
 else
 aux->dir = inserir(aux->dir, num);
 return aux;
}

int consultar(ARVORE *aux, int num, int achou)
{
 if (aux != NULL && achou == 0)
 {
 if (aux->num == num)
 {
 achou = 1;
 }
 else if (num < aux->num)
 achou = consultar(aux->esq, num, achou);
 else
 achou = consultar(aux->dir, num, achou);
 }
 return achou;
}
```

```
void mostraremordem(ARVORE *aux)
{
 if (aux != NULL)
 {
 mostraremordem(aux->esq);
 cout << aux->num << " ";
 mostraremordem(aux->dir);
 }
}

void mostrarpreadem(ARVORE *aux)
{
 if (aux != NULL)
 {
 cout << aux->num << " ";
 mostrarpreadem(aux->esq);
 mostrarpreadem(aux->dir);
 }
}

void mostraposordem(ARVORE *aux)
{
 if (aux != NULL)
 {
 mostraposordem(aux->esq);
 mostraposordem(aux->dir);
 cout << aux->num << " ";
 }
}

ARVORE* remover(ARVORE *aux, int num)
{
 ARVORE *p, *p2;
 if (aux->num == num)
 {
 if (aux->esq == aux->dir)
 { // o elemento a ser removido não tem filhos
 delete aux;
 return NULL;
 }
 else if (aux->esq == NULL)
 { // o elemento a ser removido
 // não tem filho para a esquerda
```

```
 p = aux->dir;
 delete aux;
 return p;
}
else if (aux->dir == NULL)
{ // o elemento a ser removido
 // não tem filho para à direita
 p = aux->esq;
 delete aux;
 return p;
}
else
{ // o elemento a ser removido
 // tem filho para ambos os lados
 p2= aux->dir;
 p= aux->dir;
 while (p->esq != NULL)
 {
 p=p->esq;
 }
 p->esq = aux->esq;
 delete aux;
 return p2;
}
else if (aux->num < num)
 aux->dir = remover(aux->dir, num);
else
 aux->esq = remover(aux->esq, num);
return aux;
}

ARVORE* desalocar(ARVORE* aux)
{
 if(aux!=NULL)
 {
 aux->esq=desalocar(aux->esq);
 aux->dir=desalocar(aux->dir);
 delete aux;
 }
 return NULL;
}

void main()
```

```
{
 ARVORE *raiz = NULL;
 // o ponteiro aux é um ponteiro auxiliar
 ARVORE *aux;
 // o ponteiro aux1 é um ponteiro auxiliar
 int op, achou, numero;
 do
 {
 clrscr();
 cout << "\nMENU DE OPÇÕES\n";
 cout << "\n1 - Inserir na árvore";
 cout << "\n2 - Consultar um nó da árvore";
 cout << "\n3 - Consultar toda a árvore em
 ↵ordem";
 cout << "\n4 - Consultar toda a árvore em
 ↵pré-ordem";
 cout << "\n5 - Consultar toda a árvore em
 ↵pós-ordem";
 cout << "\n6 - Excluir um nó da árvore";
 cout << "\n7 - Esvaziar a árvore";
 cout << "\n8 - Sair";
 cout << "\nDigite sua opção: ";
 cin >> op;
 if (op < 1 || op > 8)
 cout << "\nOpção inválida!!";
 else if (op == 1)
 {
 cout << "\nDigite o número a ser
 ↵inserido na árvore: ";
 cin >> numero;
 raiz = inserir(raiz, numero);
 cout << "\nNúmero inserido na
 ↵árvore!!";
 }
 else if (op == 2)
 {
 if (raiz == NULL)
 {
 // a árvore está vazia
 cout << "\nÁrvore vazia!!";
 }
 else
 {
```

```
// a árvore contém elementos
cout <<"\nDigite o elemento a ser
 ↪consultado";
cin >> numero;
achou = 0;
achou =
 ↪consultar(raiz,numero,achou);
if (achou == 0)
 cout <<"\nNúmero não
 ↪encontrado na árvore!";
else
 cout <<"\nNúmero encontrado
 ↪na árvore!";
}
}
else if (op == 3)
{
 if (raiz == NULL)
 {
 // a árvore está vazia
 cout <<"\nÁrvore vazia!!";
 }
 else
 {
 // a árvore contém elementos
 // e estes serão mostrados em ordem
 cout <<"\nListando todos os
 ↪elementos da árvore em ordem";
 mostraremordem(raiz);
 }
}
else if (op == 4)
{
 if (raiz == NULL)
 {
 // a árvore está vazia
 cout <<"\nÁrvore vazia!!";
 }
 else
 {
 // a árvore contém elementos
 // e estes serão mostrados
 // em pré-ordem
```

```
 cout <<"\nListando todos os
 ↵ elementos
 ↵ da árvore
 ↵ em pré-ordem";
 mostrarpreordem(raiz);
}

}
else if (op == 5)
{
 if (raiz == NULL)
 {
 // a árvore está vazia
 cout <<"\nÁrvore vazia!!";
 }
 else
 {
 // a árvore contém elementos
 // e estes serão mostrados em
 // pós-ordem
 cout <<"\nListando todos os
 ↵ elementos
 ↵ da árvore
 ↵ em pós-ordem";
 aux = raiz;
 mostrarposordem(aux);
 }
}
else if (op == 6)
{
 if (raiz == NULL)
 cout <<"\nÁrvore vazia!!";
 else
 {
 cout <<"\nDigite o número que
 ↵ deseja excluir:";
 cin >> numero;
 achou = 0;
 achou =
 ↵ consultar(raiz,numero,achou);
 if (achou == 0)
 cout <<"\nNúmero não encontrado na
 ↵ árvore!";
 else
 {
 raiz = remover(raiz,numero);
 cout <<"\nNúmero excluído da
 ↵ árvore!";
```

```

 }
 }
}
else if (op == 7)
{
 if (raiz == NULL)
 cout << "\nÁrvore vazia!";
 else
 {
 raiz=desalocar(raiz);
 cout << "\nÁrvore
 esvaziada!";
 }
}
getch();
}while (op != 8);
raiz = desalocar(raiz);
}

```

---

## Análise da complexidade

A relação existente entre a altura da árvore ( $h$ ) e o número de nós ( $n$ ) de uma árvore binária é uma informação muito importante em muitas aplicações. É comum a pergunta pela altura máxima e mínima de árvores binárias. Possuem altura máxima aquelas em que cada nó possui apenas um único filho. A altura de tais árvores é igual a  $n$ . Já uma árvore completa possui altura mínima.

Segundo Markenzon (1994), uma árvore binária completa com  $n > 0$  nós possui altura mínima  $h = 1 + \lfloor \log n \rfloor$ .

A operação de busca em uma árvore binária é igual ao número de nós existentes no caminho desde a raiz da árvore até o nó procurado. Em uma árvore binária genérica, no pior caso, esse nó se encontra a uma distância  $O(n)$  da raiz da árvore, logo, a complexidade da busca é  $O(n)$ . Conclui-se então que a complexidade de busca corresponde à altura da árvore. No melhor caso, em que uma árvore pode possuir altura mínima, que é o caso de uma árvore binária completa, o tempo de busca é  $O(\log n)$ .

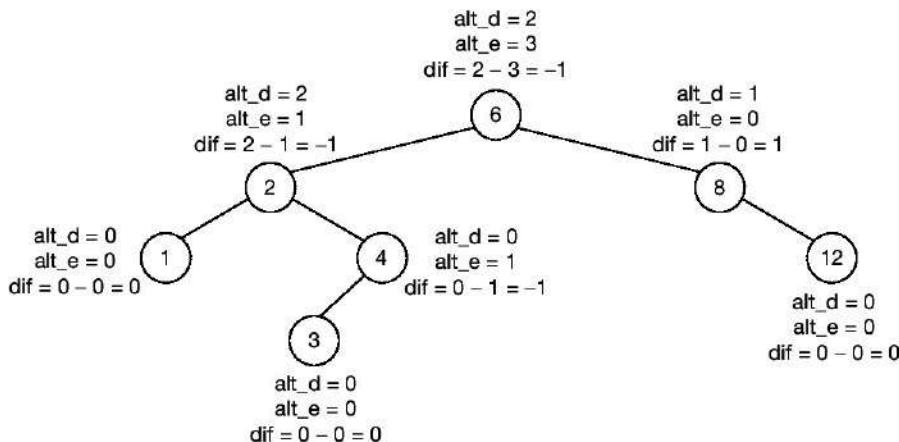
Considerando ainda uma árvore de altura mínima, na operação de inserção, o nó sempre é inserido em uma folha, tendo que percorrer todos os nós desde a raiz, até chegar em uma folha e acrescentar um filho a ela, gastando com isso a altura da árvore, ou seja,  $O(\log n)$ .

Na operação de remoção, o pior caso acontece quando o nó a ser removido encontra-se em uma folha no nível mais baixo. Gasta-se a altura da árvore para encontrá-lo, no caso de uma árvore de altura mínima, e algumas operações constantes de atualização de ponteiros, gerando uma complexidade  $O(\log n)$ .

## Árvore AVL

A árvore AVL, criada em 1962 por Adelson-Velsky e Landis, é uma árvore binária balanceada, ou seja, é uma árvore que obedece a todas as propriedades da árvore binária e em que cada nó apresenta diferença de altura entre as sub-árvore direita e esquerda de 1, 0 ou -1, como ilustra a Figura 7.16.

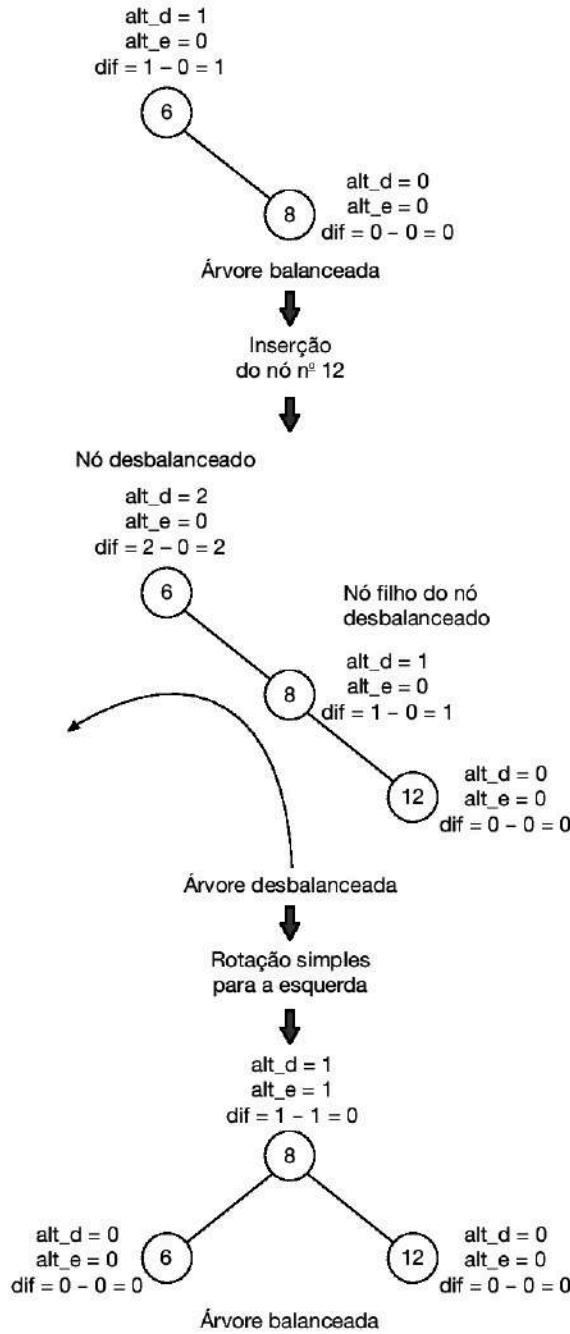
**Figura 7.16** Árvore AVL

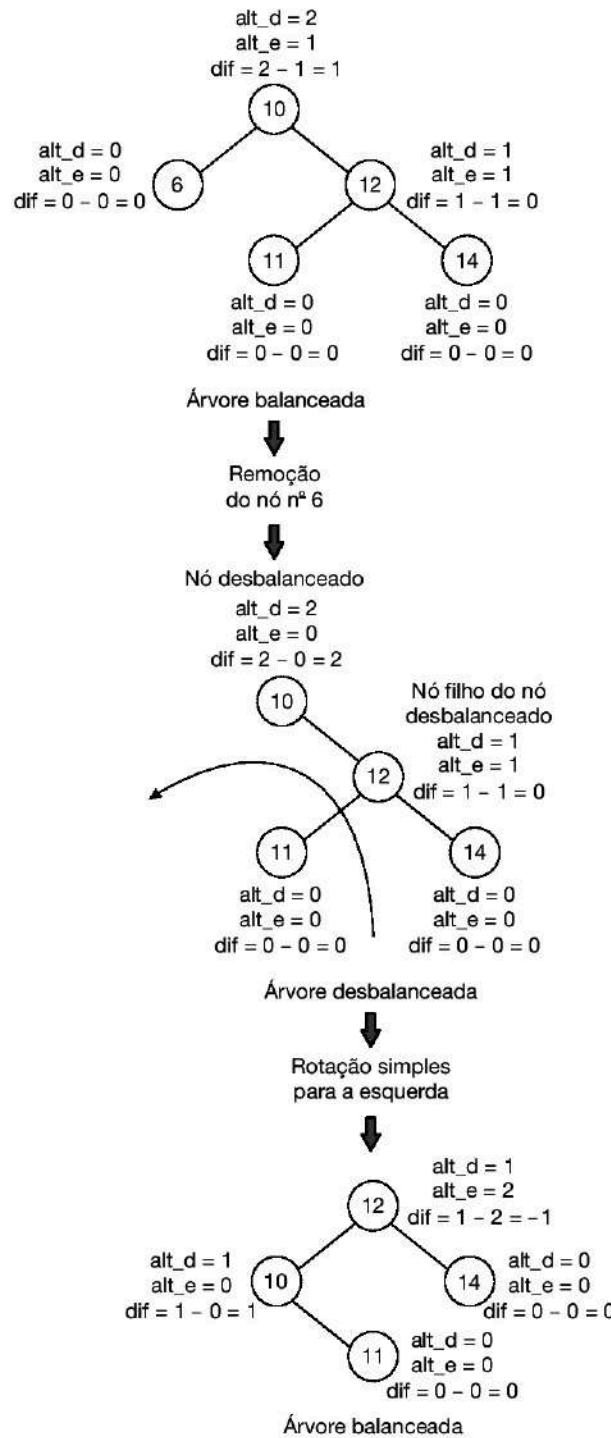


Se a diferença de altura entre as sub-árvore de um nó é maior que 1 ou menor que -1, a árvore está desbalanceada e haverá uma rotação. As possíveis rotações estão descritas e ilustradas no Quadro 7.1 e nas figuras 7.17 a 7.22.

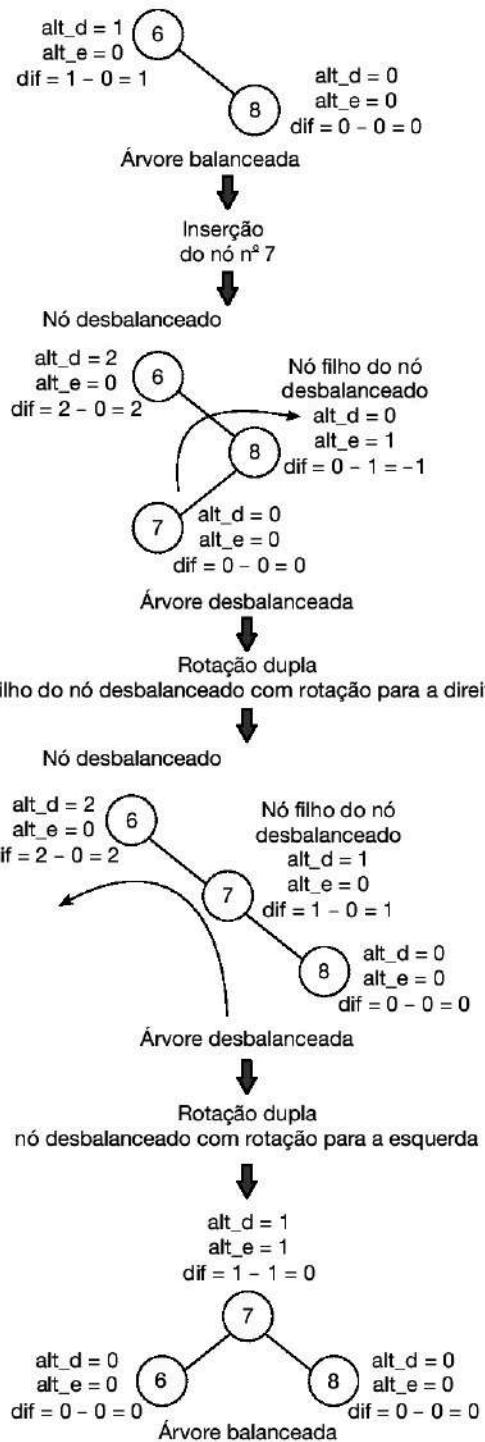
**Quadro 7.1** Descrição de rotações

| Diferença de altura de um nó | Diferença de altura do nó filho do nó desbalanceado | Tipo de rotação                                      | Figura |
|------------------------------|-----------------------------------------------------|------------------------------------------------------|--------|
|                              | 1                                                   | Simples à esquerda                                   | 7.17   |
| 2                            | 0                                                   | Simples à esquerda                                   | 7.18   |
|                              | -1                                                  | Dupla com filho para a direita e pai para a esquerda | 7.19   |
|                              | 1                                                   | Dupla com filho para a esquerda e pai para a direita | 7.20   |
| -2                           | 0                                                   | Simples à direita                                    | 7.21   |
|                              | -1                                                  | Simples à direita                                    | 7.22   |

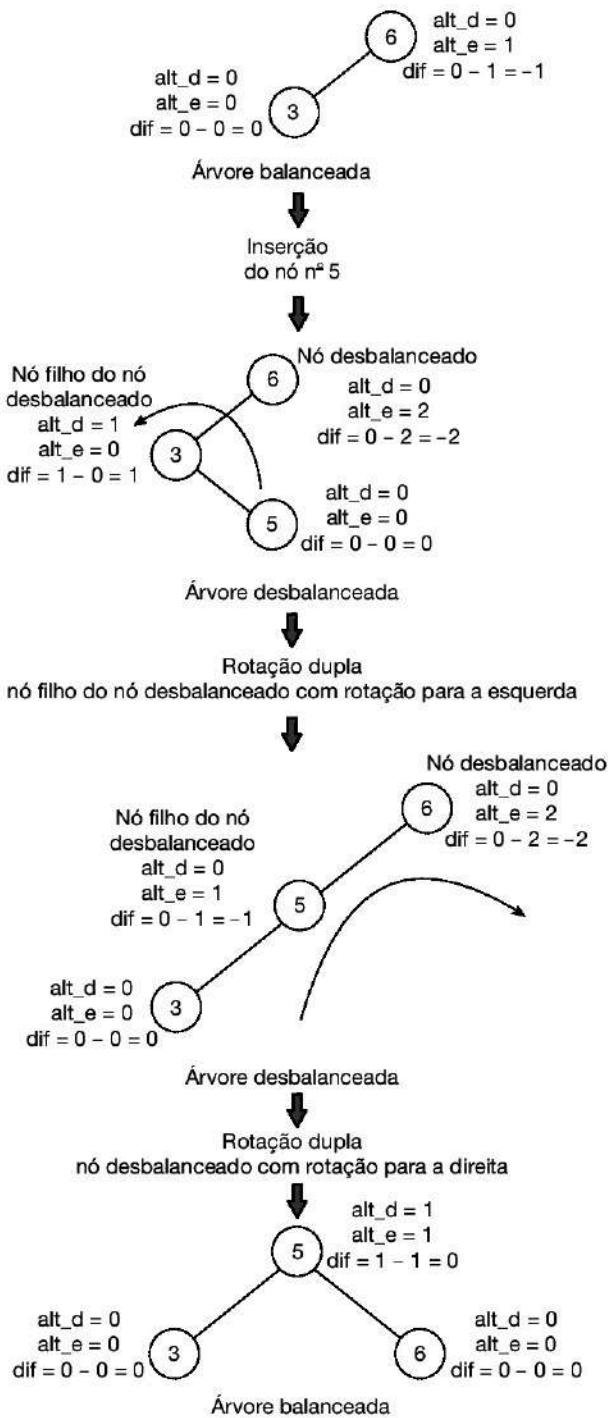
**Figura 7.17** Rotação simples para a esquerda

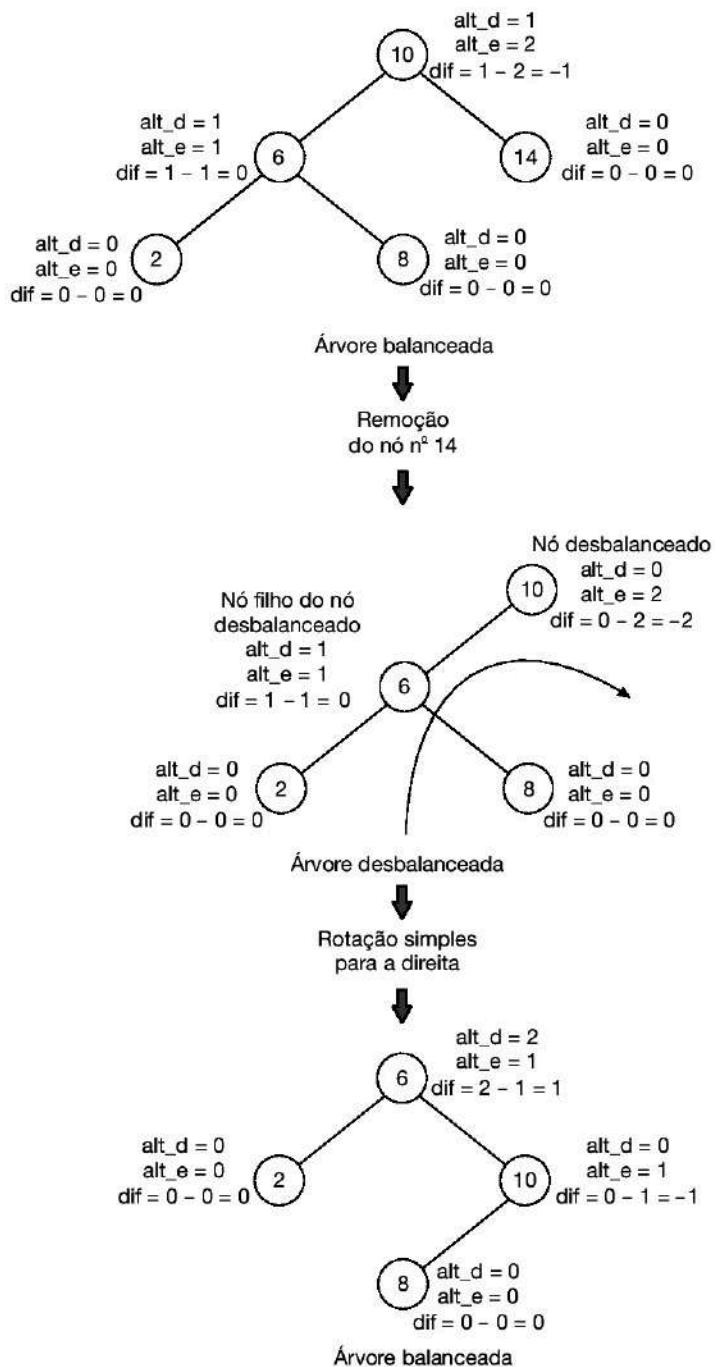
**Figura 7.18** • Rotação simples para a esquerda

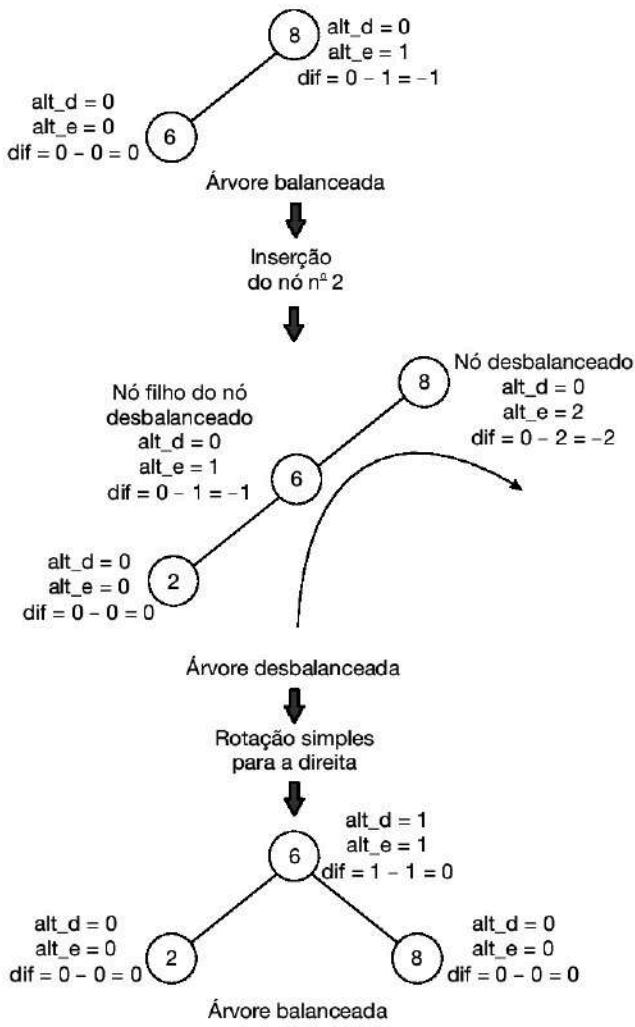
**Figura 7.19** • Rotação dupla com filho para a direita e pai para a esquerda



**Figura 7.20** Rotação dupla com filho para a esquerda e pai para a direita



**Figura 7.21** Rotação simples para a direita

**Figura 7.22** • Rotação simples para a direita



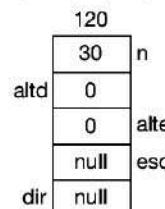
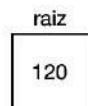
ILUSTRAÇÃO

**1<sup>a</sup> operação**  
A árvore AVL está vazia



**2<sup>a</sup> operação**  
Inserção do nº 30

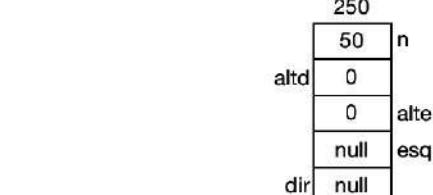
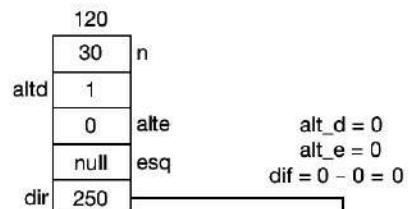
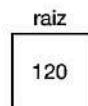
alt\_d = 0  
alt\_e = 0  
dif = 0 - 0 = 0



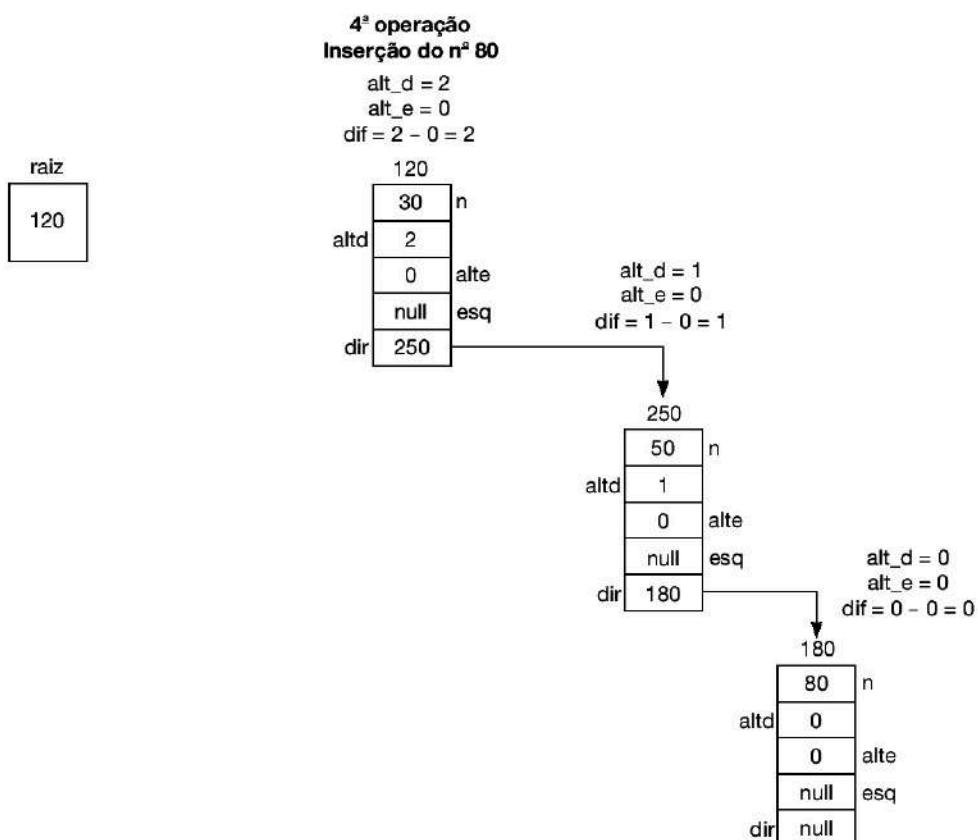
Árvore balanceada

**3<sup>a</sup> operação**  
Inserção do nº 50

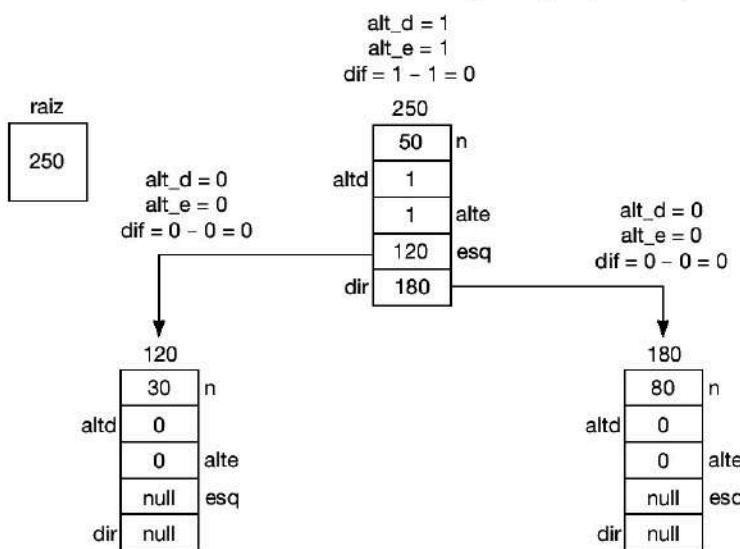
alt\_d = 1  
alt\_e = 0  
dif = 1 - 0 = 1



Árvore balanceada



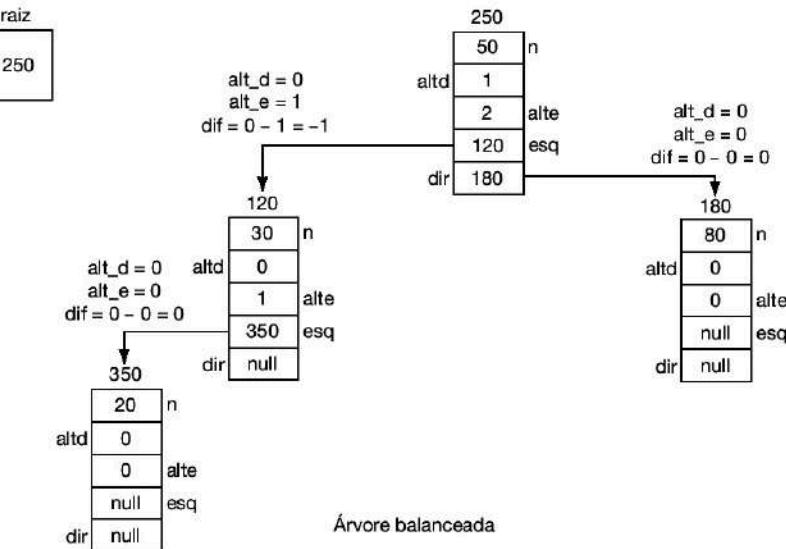
Árvore desbalanceada  
Rotação simples para a esquerda



Árvore balanceada

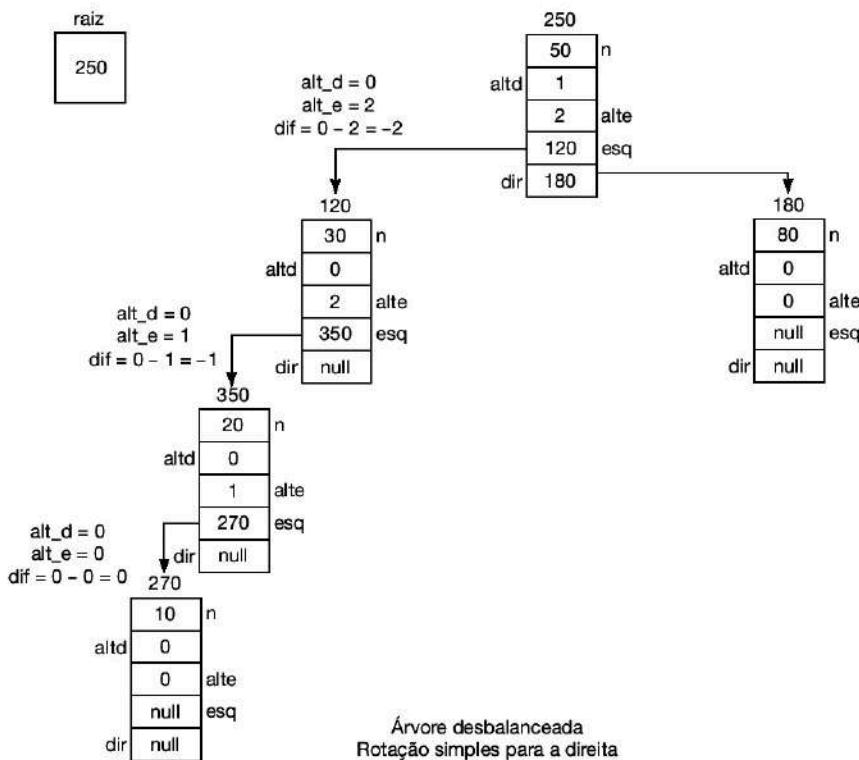
**5<sup>a</sup> operação**  
**Inserção do nº 20**

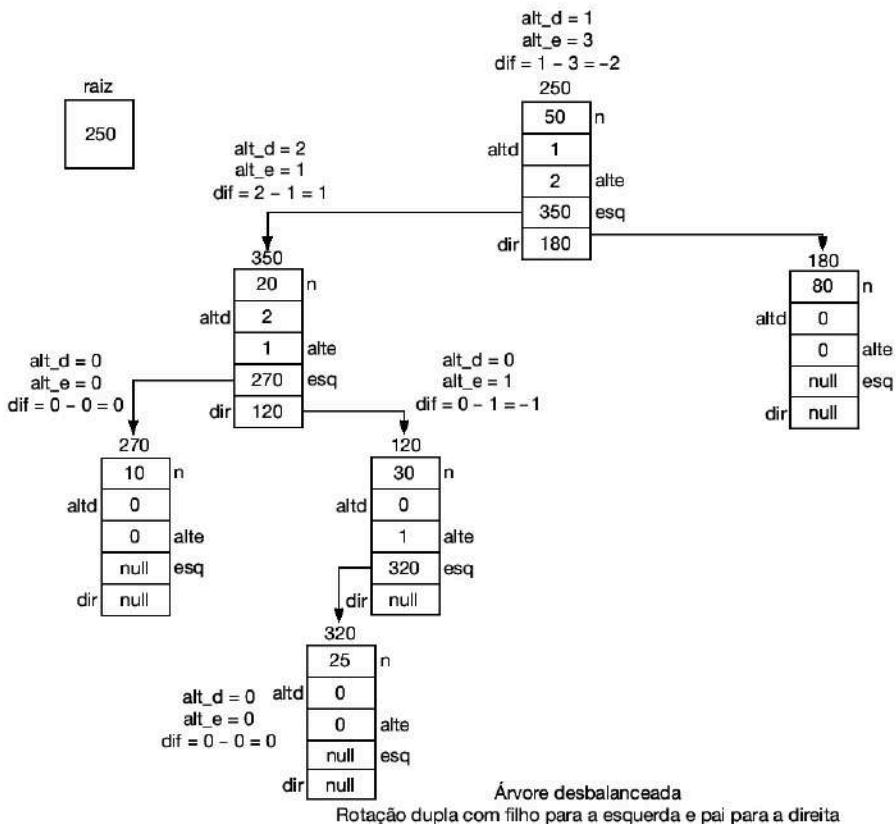
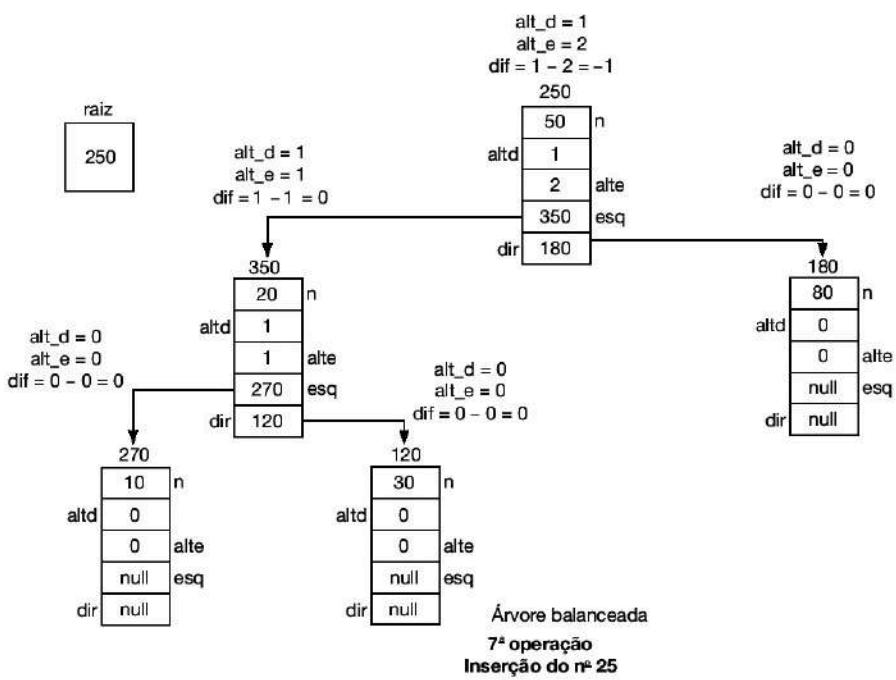
alt\_d = 1  
alt\_e = 2  
dif = 1 - 2 = -1

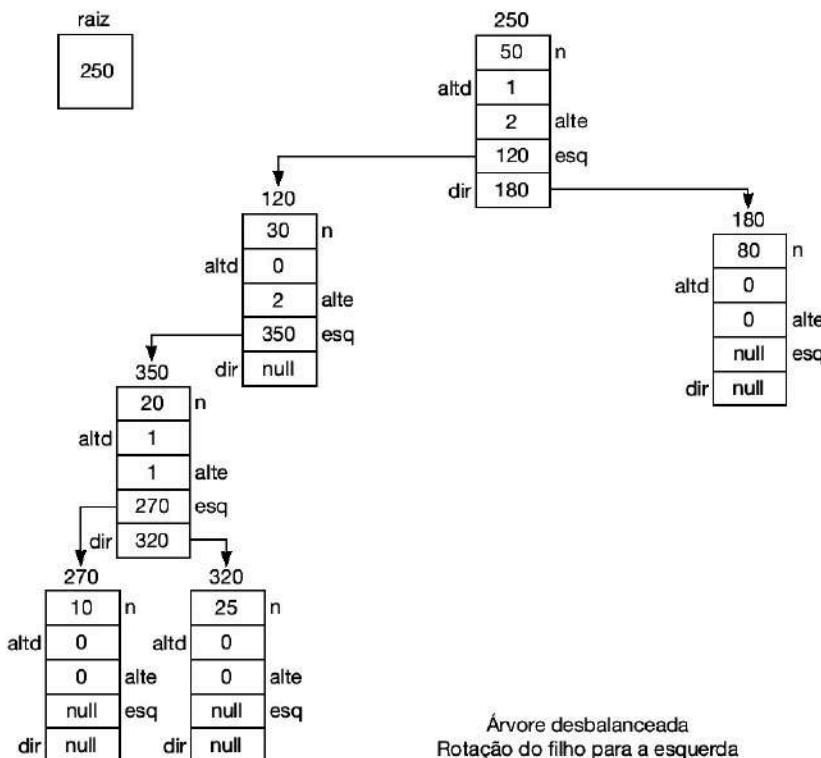


Árvore balanceada

**6<sup>a</sup> operação**  
**Inserção do nº 10**

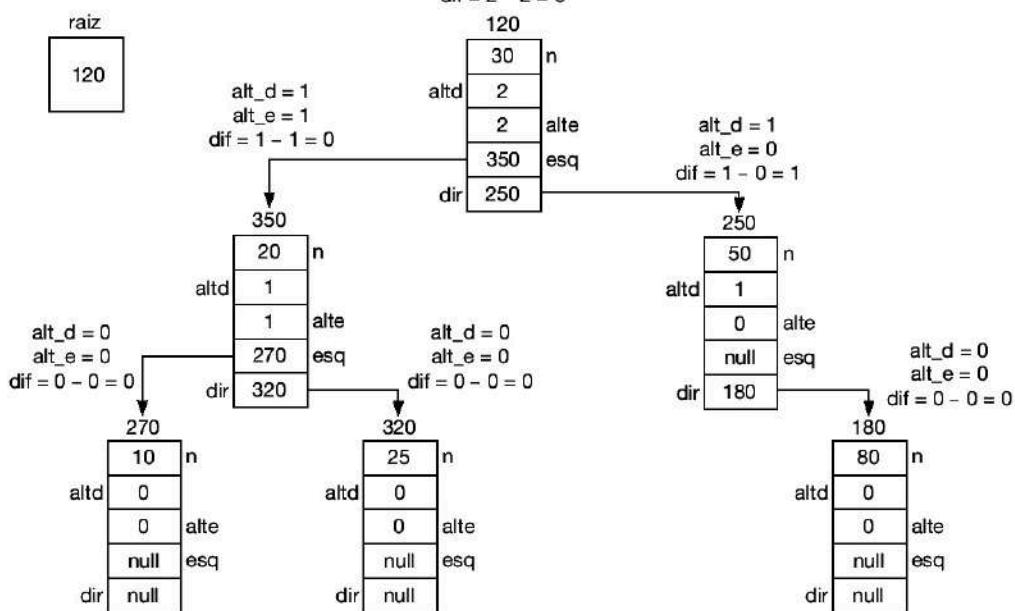




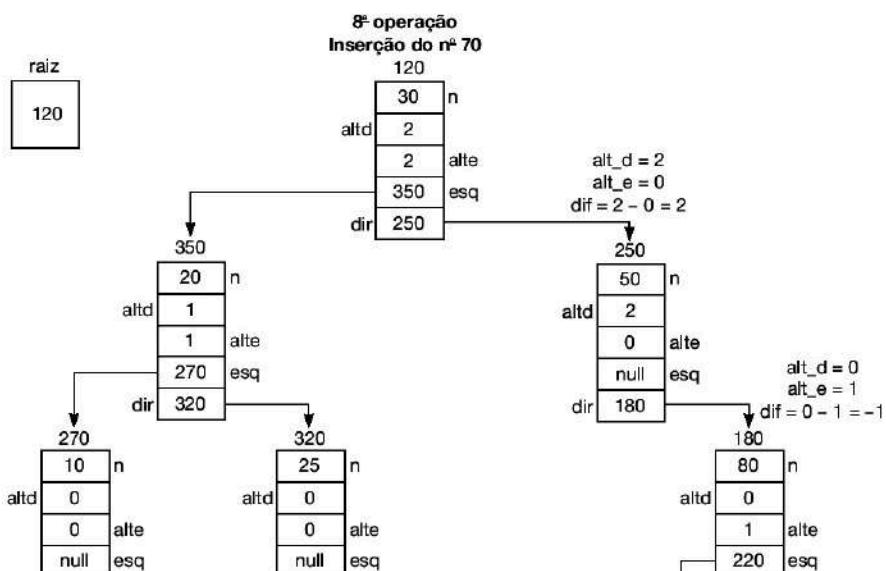


Árvore desbalanceada  
Rotação do filho para a esquerda

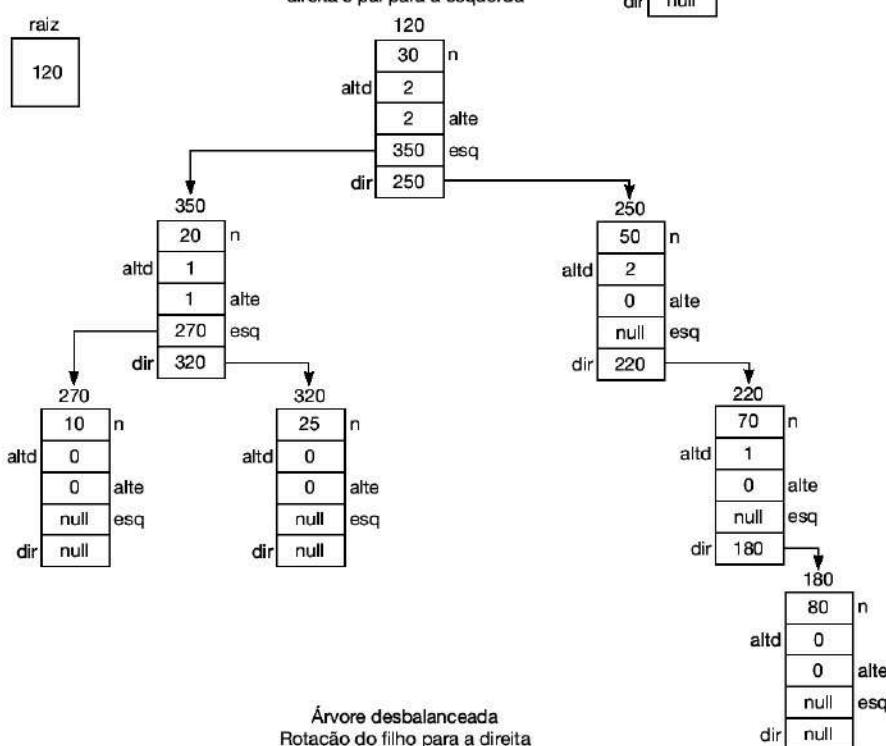
alt\_d = 2  
alt\_e = 2  
dif = 2 - 2 = 0



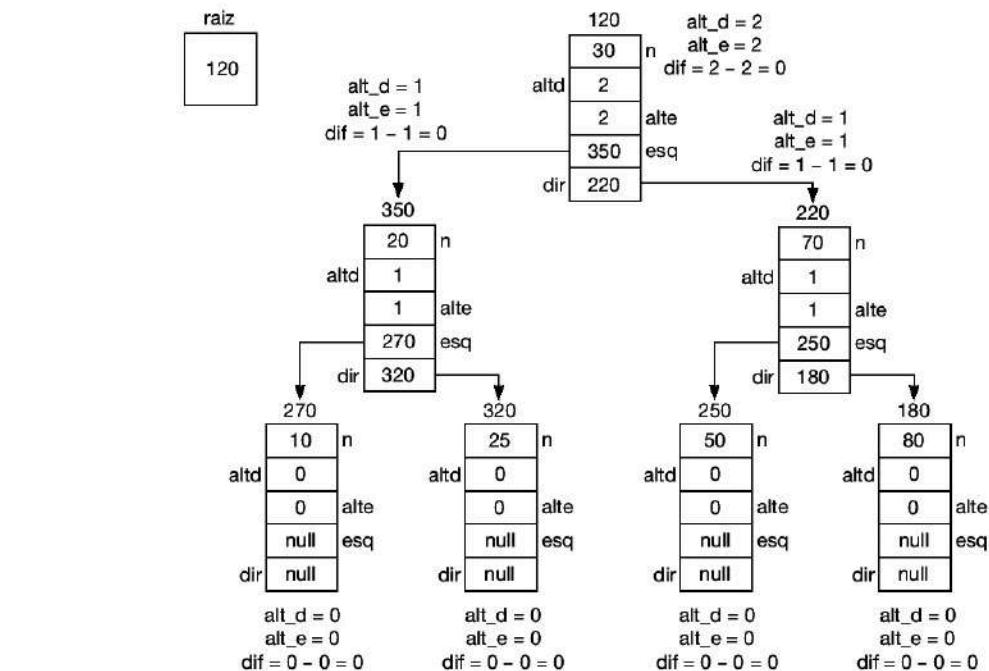
Rotação do pai para a direita  
Árvore balanceada



Árvore desbalanceada  
Rotação dupla com filho para a direita e pai para a esquerda

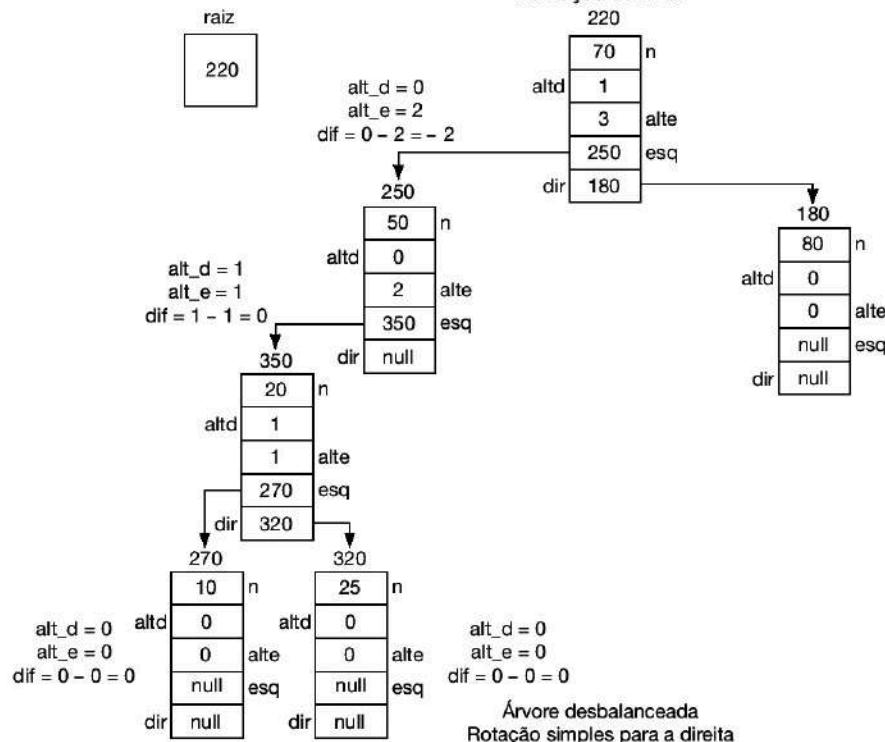


Árvore desbalanceada  
Rotação do filho para a direita

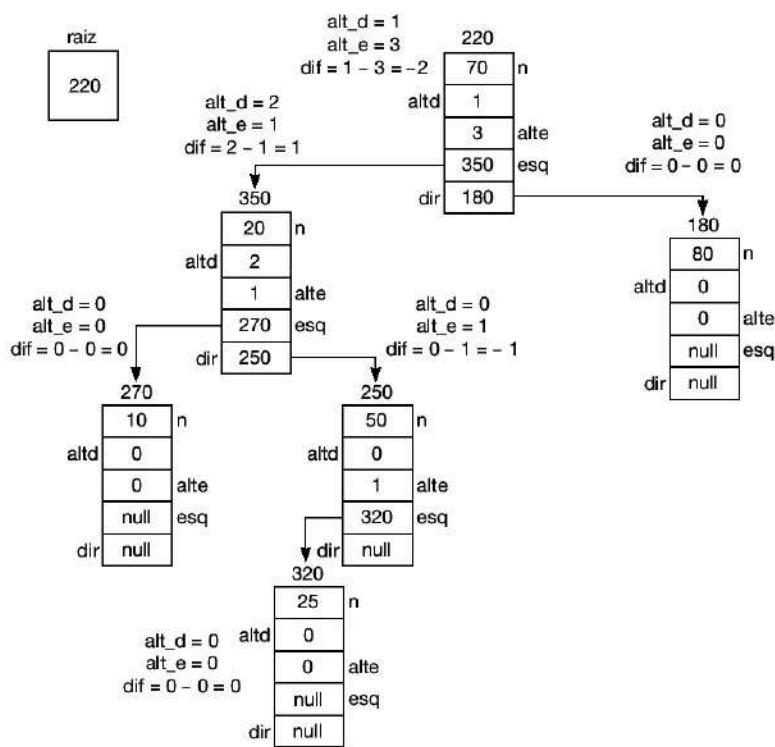


Rotação do pai para a esquerda  
Árvore balanceada

### 9<sup>a</sup> operação Remoção do nº 30

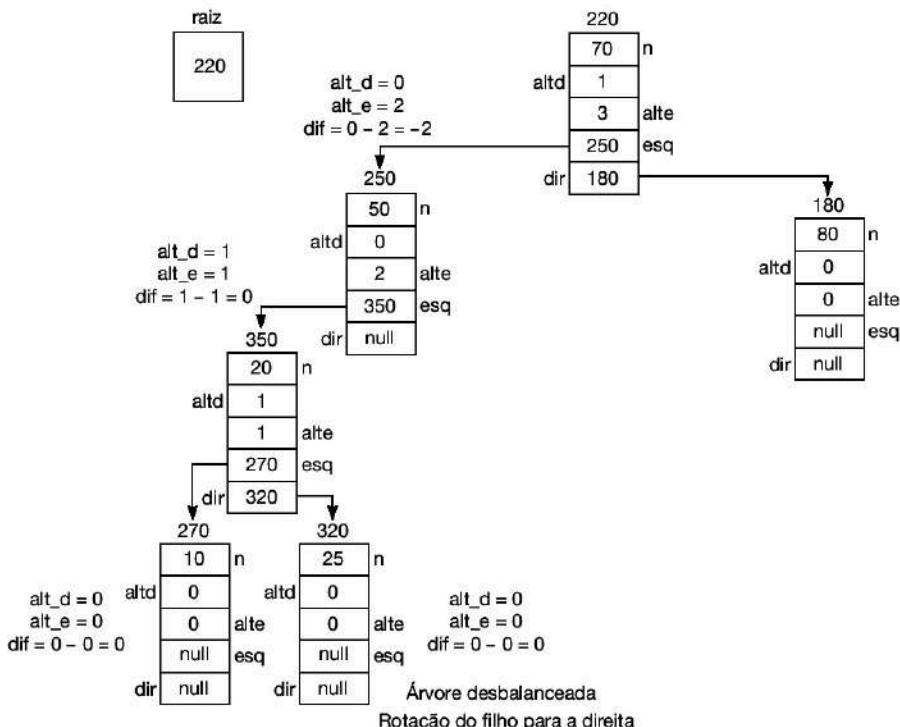


Árvore desbalanceada  
Rotação simples para a direita



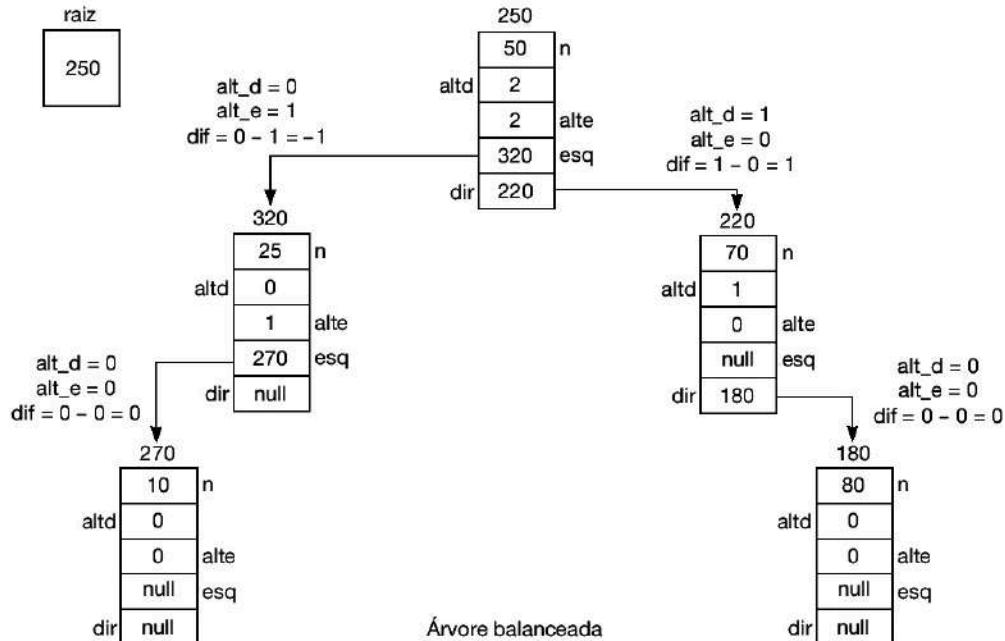
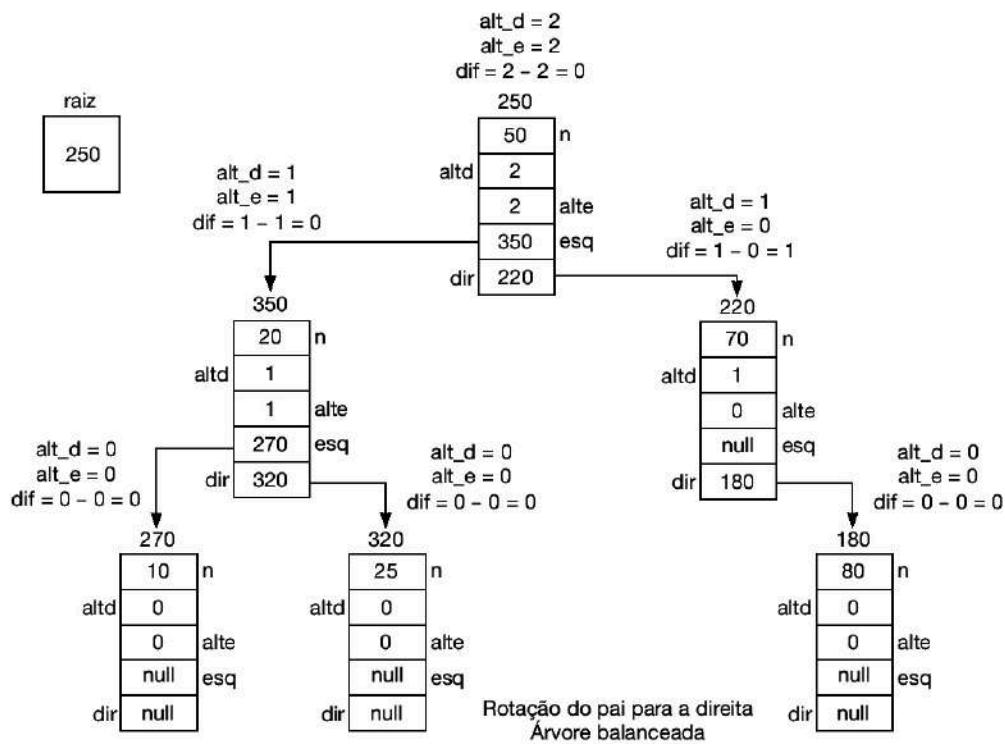
Árvore desbalanceada

Rotação dupla com filho para a esquerda e pai para a direita

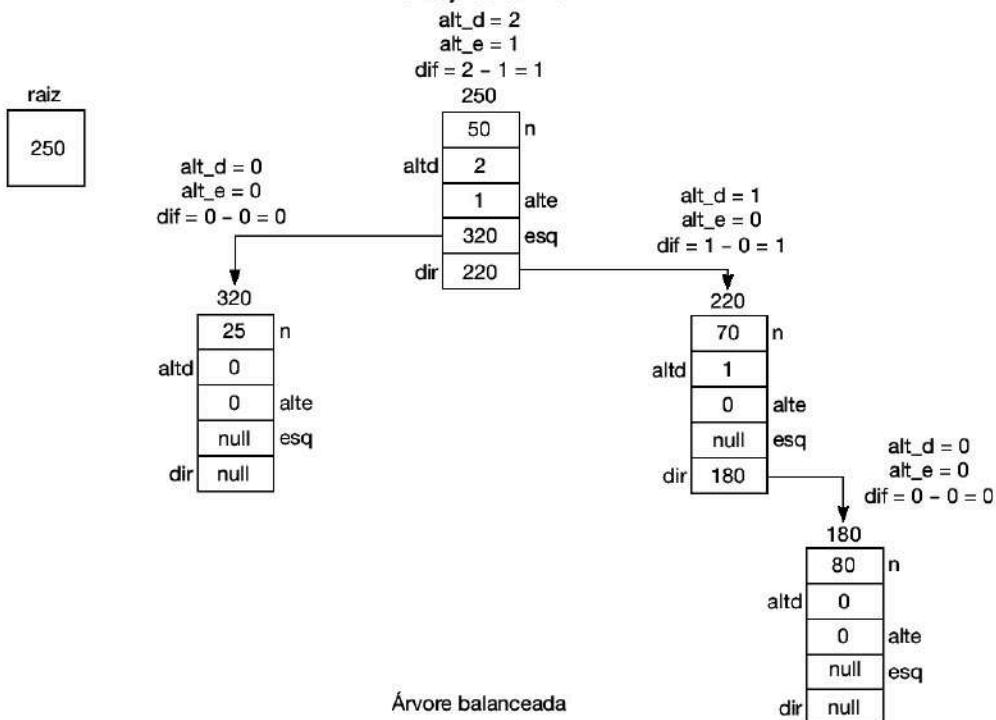


Árvore desbalanceada

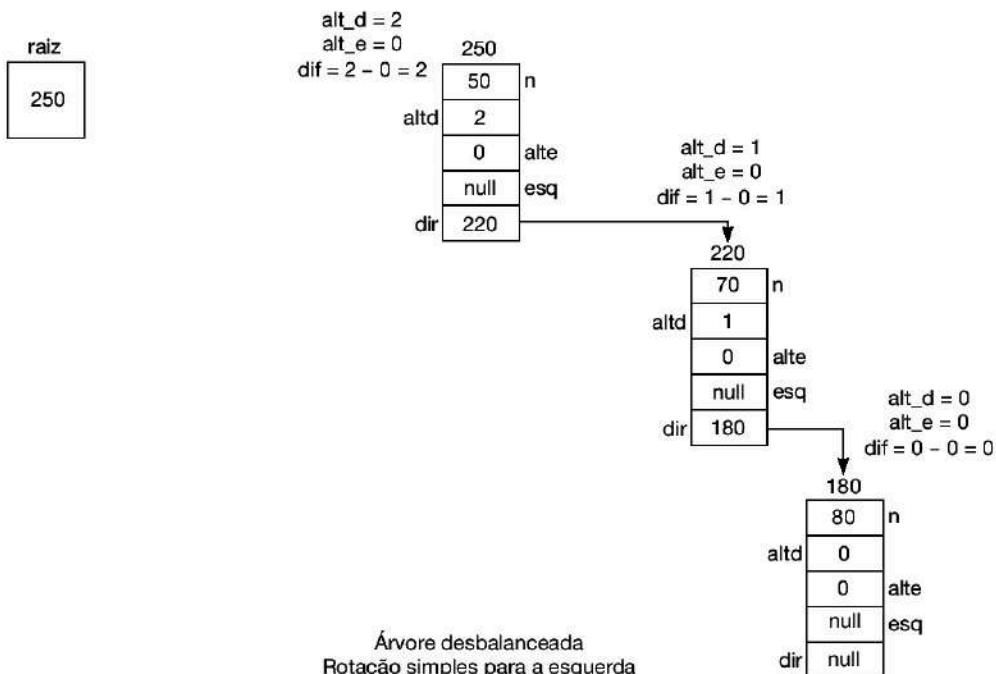
Rotação do filho para a direita

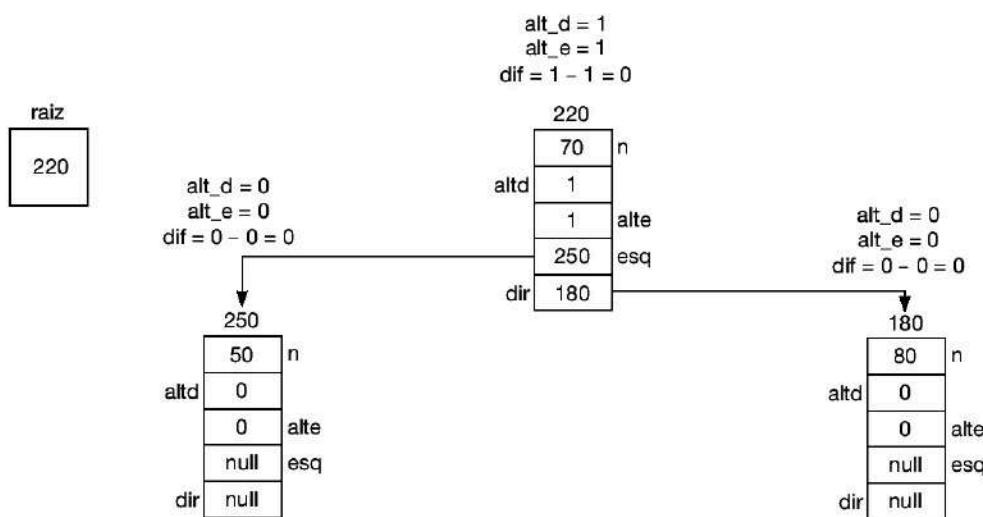


**11ª operação**  
**Remoção do nº 10**



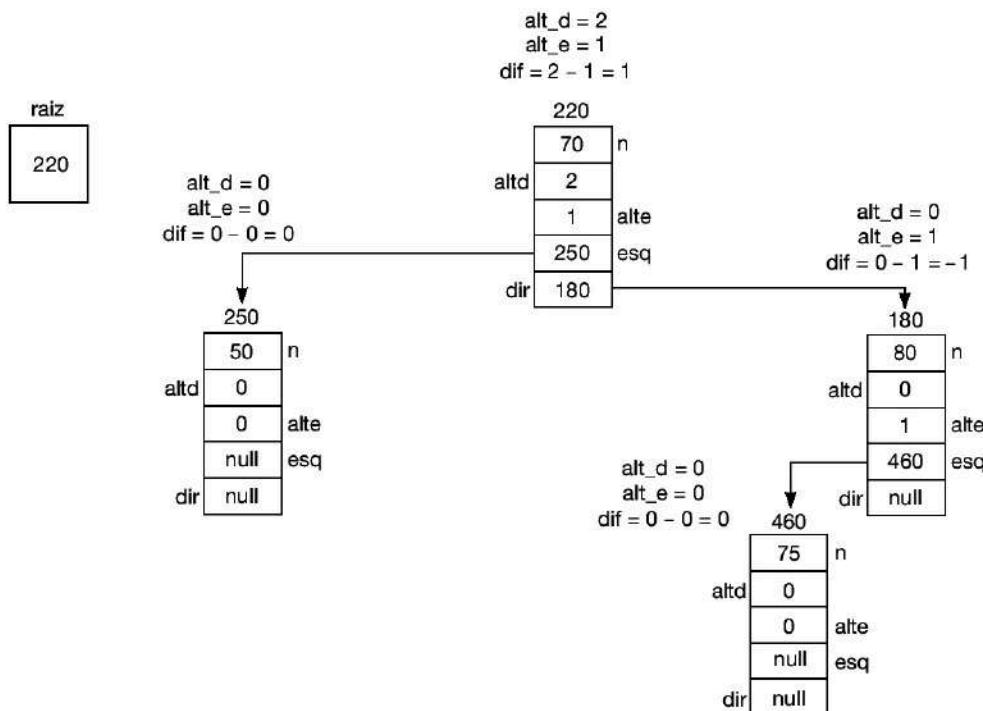
**12ª operação**  
**Remoção do nº 25**



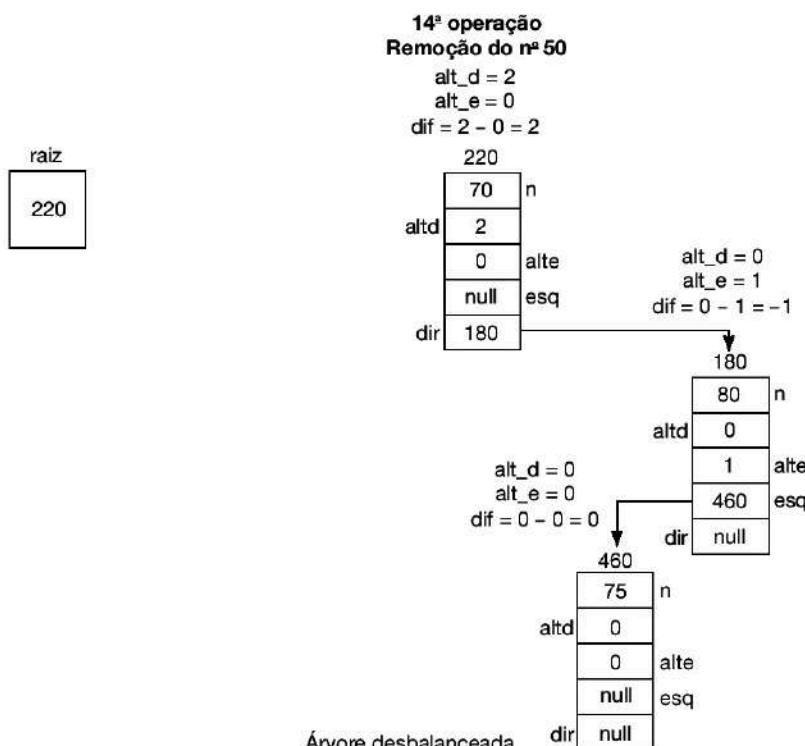


Árvore balanceada

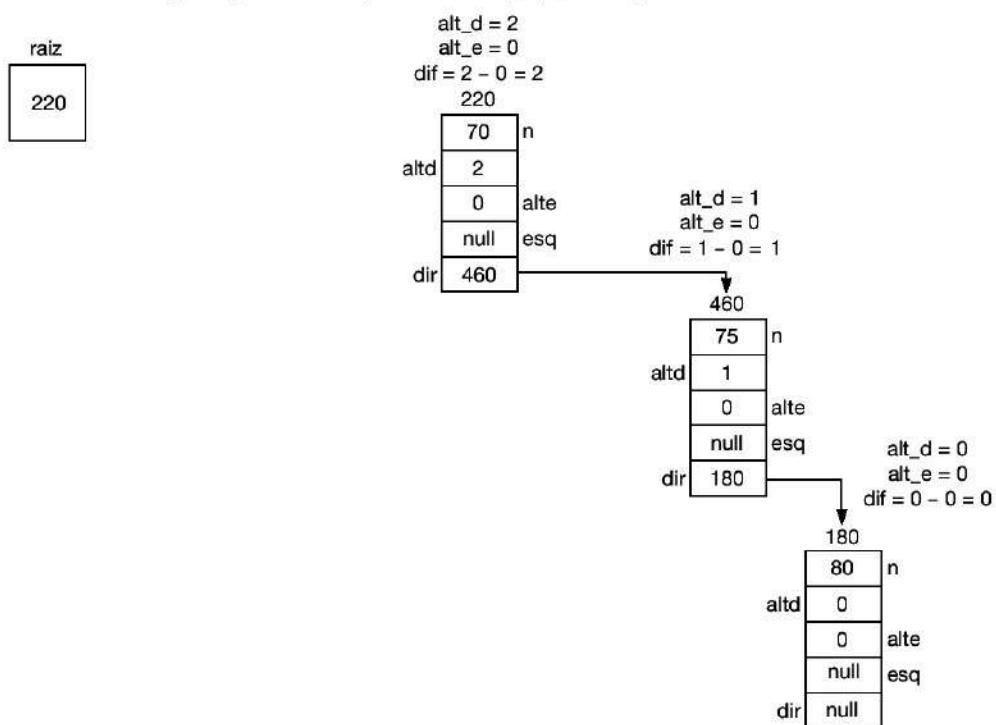
**13ª operação**  
**Inserção do nº 75**



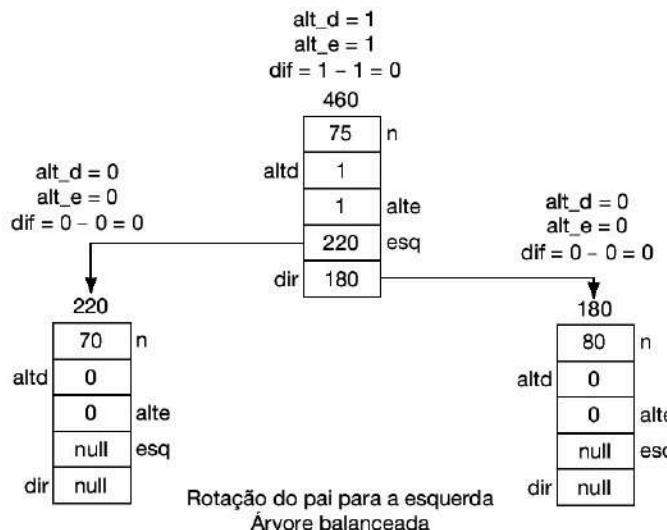
Árvore balanceada



Rotação dupla com filho para a direita e pai para a esquerda



Rotação do filho para a direita  
Árvore desbalanceada



A seguir, implementações em JAVA e em C/C++ das operações de uma árvore AVL.




---

```

import java.util.*;
public class Arvore_AVL
{
 //Definindo a classe que representará
 // cada elemento da árvore AVL
 private static class ARVORE
 {
 public int num, altd, alte;
 public ARVORE dir, esq;
 }

 public static void main(String args[])
 {
 Scanner entrada = new Scanner(System.in);
 // a árvore está vazia, logo,
 // o objeto raiz têm o valor null
 ARVORE raiz = null;
 }
}

```

```
// o objeto aux é um objeto auxiliar
ARVORE aux;
// o objeto aux1 é um objeto auxiliar
int op, achou, numero;
do
{
 System.out.println("\nMENU DE OPÇÕES\n");
 System.out.println("1 - Inserir na árvore");
 System.out.println("2 - Consultar um nó da árvore");
 System.out.println("3 - Consultar toda a árvore em
 ↪ordem");
 System.out.println("4 - Consultar toda a árvore em
 ↪pré-ordem");
 System.out.println("5 - Consultar toda a árvore em
 ↪pós-ordem");
 System.out.println("6 - Excluir um nó da árvore");
 System.out.println("7 - Esvaziar a árvore");
 System.out.println("8 - Sair");
 System.out.print("Digite sua opção: ");
 op = entrada.nextInt();
 if (op < 1 || op > 8)
 System.out.println("Opção inválida!!!");
 if (op == 1)
 {
 System.out.println("Digite o número a ser inseri
 ↪do na árvore: ");
 numero = entrada.nextInt();
 raiz = inserir(raiz,numero);
 System.out.println ("Número inserido na árvore!!!");
 }
 if (op == 2)
 {
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!!");
 }
 else
 {
 // a árvore contém elementos
 System.out.println("Digite o elemento a ser
 ↪consultado");
 numero = entrada.nextInt();
 achou = 0;
 achou = consultar(raiz,numero,achou);
```

```
 if (achou == 0)
 System.out.println("Número não encontrado na
 árvore!");
 else
 System.out.println("Número encontrado na
 árvore!");
 }
}
if (op == 3)
{
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
 }
 else
 {
 // a árvore contém elementos e
 // estes serão mostrados em ordem
 System.out.println("Listando todos os
 elementos da árvore em
 ordem");
 mostraremordem(raiz);
 }
}
if (op == 4)
{
 if (raiz == null)
 {
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
 }
 else
 {
 // a árvore contém elementos e
 // estes serão mostrados em pré-ordem
 System.out.println("Listando todos os
 elementos da árvore em
 pré-ordem");
 mostrarpreordem(raiz);
 }
}
if (op == 5)
{
 if (raiz == null)
```

```

{
 // a árvore está vazia
 System.out.println("Árvore vazia!!");
}
else
{
 // a árvore contém elementos e
 // estes serão mostrados em pós-ordem
 System.out.println("Listando todos os
 ↪elementos da árvore em
 ↪pós-ordem");
 aux = raiz;
 mostrarpesordenm(aux);
}
}

if (op == 6)
{
 if (raiz == null)
 System.out.println("Árvore vazia!!");
 else
 {
 System.out.println("Digite o número que
 ↪deseja excluir");
 numero = entrada.nextInt();
 achou = 0;
 achou = consultar(raiz, numero, achou);
 if (achou == 0)
 System.out.println("Número não encontrado
 ↪na árvore!");
 else
 {
 raiz = remover(raiz, numero);
 raiz = atualiza(raiz);
 System.out.println("Número excluído da
 ↪árvore!");
 }
 }
}
if (op == 7)
{
 if (raiz == null)
 System.out.println("Árvore vazia!!");
 else
 {
 raiz = null;
 System.out.println("Árvore esvaziada!!");
 }
}

```

```
 }
 }
}

while (op != 8);
}

public static ARVORE inserir(ARVORE aux, int num)
{
 // o objeto novo é um objeto auxiliar
 ARVORE novo;
 if (aux == null)
 {
 novo = new ARVORE();
 novo.num = num;
 novo.altd = 0;
 novo.alte = 0;
 novo.esq = null;
 novo.dir = null;
 aux = novo;
 }
 else if (num < aux.num)
 {
 aux.esq = inserir(aux.esq, num);
 if (aux.esq.altd > aux.esq.alte)
 aux.alte = aux.esq.altd + 1;
 else
 aux.alte = aux.esq.alte + 1;
 aux = balanceamento(aux);
 }
 else
 {
 aux.dir = inserir(aux.dir, num);
 if (aux.dir.altd > aux.dir.alte)
 aux.altd = aux.dir.altd + 1;
 else
 aux.altd = aux.dir.alte + 1;
 aux = balanceamento(aux);
 }
}
return aux;
}

public static ARVORE balanceamento(ARVORE aux)
{
 int d, df;
 d = aux.altd - aux.alte;
 if (d == 2)
```

```
{
 df = aux.dir.altd - aux.dir.alte;
 if (df >= 0)
 {
 aux = rotacao_esquerda(aux);
 }
 else
 {
 aux.dir = rotacao_direita(aux.dir);
 aux = rotacao_esquerda(aux);
 }
}
else if (d == -2)
{
 df = aux.esq.altd - aux.esq.alte;
 if (df <= 0)
 {
 aux = rotacao_direita(aux);
 }
 else
 {
 aux.esq = rotacao_esquerda(aux.esq);
 aux = rotacao_direita(aux);
 }
}
return aux;
}

public static ARVORE rotacao_esquerda(ARVORE aux)
{
 ARVORE aux1, aux2;
 aux1 = aux.dir;
 aux2 = aux1.esq;
 aux.dir = aux2;
 aux1.esq = aux;
 if (aux.dir == null)
 aux.altd = 0;
 else if (aux.dir.alte > aux.dir.altd)
 aux.altd = aux.dir.alte+1;
 else
 aux.altd = aux.dir.altd+1;

 if (aux1.esq.alte > aux1.esq.altd)
 aux1.alte = aux1.esq.alte + 1;
 else
 aux1.alte = aux1.esq.altd + 1;
}
```

```
 return aux1;
}

public static ARVORE rotacao_direita(ARVORE aux)
{
 ARVORE aux1, aux2;
 aux1 = aux.esq;
 aux2 = aux1.dir;
 aux.esq = aux2;
 aux1.dir = aux;
 if (aux.esq == null)
 aux.alte = 0;
 else if (aux.esq.alte > aux.esq.altd)
 aux.alte = aux.esq.alte+1;
 else
 aux.alte = aux.esq.altd+1;

 if (aux1.dir.alte > aux1.dir.altd)
 aux1.altd = aux1.dir.alte + 1;
 else
 aux1.altd = aux1.dir.altd + 1;
 return aux1;
}

public static int consultar(ARVORE aux, int num, int
achou)
{
 if (aux != null && achou == 0)
 {
 if (aux.num == num)
 {
 achou = 1;
 }
 else if (num < aux.num)
 achou = consultar(aux.esq, num, achou);
 else
 achou = consultar(aux.dir, num, achou);
 }
 return achou;
}

public static void mostraremordem(ARVORE aux)
{
 if (aux != null)
 {
 mostraremordem(aux.esq);
 System.out.print(aux.num + " ");
 mostraremordem(aux.dir);
 }
}
```

```
 System.out.print(aux.num+" ");
 mostraremordem(aux.dir);
 }
}

public static void mostrarpreadem(ARVORE aux)
{
 if (aux != null)
 {
 System.out.print(aux.num+" ");
 mostrarpreadem(aux.esq);
 mostrarpreadem(aux.dir);
 }
}

public static void mostrarpesordem(ARVORE aux)
{
 if (aux != null)
 {
 mostrarpesordem(aux.esq);
 mostrarpesordem(aux.dir);
 System.out.print(aux.num+" ");
 }
}

public static ARVORE remover(ARVORE aux, int num)
{
 ARVORE p, p2;
 if (aux.num == num)
 {
 if (aux.esq == aux.dir)
 {
 // o elemento a ser removido não tem filhos
 return null;
 }
 else if (aux.esq == null)
 {
 // o elemento a ser removido
 // não tem filho para à esquerda
 return aux.dir;
 }
 else if (aux.dir == null)
 {
 // o elemento a ser removido
 // não tem filho para à direita
 return aux.esq;
 }
 else
 { // o elemento a ser removido
```

```
// tem filho para ambos os lados
p2= aux.dir;
p= aux.dir;
while (p.esq != null)
{
 p=p.esq;
}
p.esq = aux.esq;
return p2;
}

}
else if (aux.num < num)
 aux.dir = remover(aux.dir, num);
else
 aux.esq = remover(aux.esq, num);
return aux;
}

public static ARVORE atualiza(ARVORE aux)
{
 if (aux!=null)
 {
 aux.esq = atualiza(aux.esq);
 if (aux.esq==null)
 aux.alte = 0;
 else if (aux.esq.alte > aux.esq.altd)
 aux.alte = aux.esq.alte + 1;
 else
 aux.alte = aux.esq.altd + 1;

 aux.dir = atualiza(aux.dir);
 if (aux.dir==null)
 aux.altd = 0;
 else if (aux.dir.alte > aux.dir.altd)
 aux.altd = aux.dir.alte + 1;
 else
 aux.altd = aux.dir.altd + 1;
 aux = balanceamento(aux);
 }
 return aux;
}
}
```



```

#include <iostream.h>
#include <conio.h>

//Definindo o registro que representará
// cada elemento da árvore AVL

struct ARVORE
{
 int num, altd, alte;
 ARVORE *dir, *esq;
};

ARVORE* rotacao_esquerda(ARVORE* aux)
{
 ARVORE *aux1, *aux2;
 aux1 = aux->dir;
 aux2 = aux1->esq;
 aux->dir = aux2;
 aux1->esq = aux;
 if (aux->dir == NULL)
 aux->altd = 0;
 else if (aux->dir->alte > aux->dir->altd)
 aux->altd = aux->dir->alte+1;
 else
 aux->altd = aux->dir->altd+1;

 if (aux1->esq->alte > aux1->esq->altd)
 aux1->alte = aux1->esq->alte + 1;
 else
 aux1->alte = aux1->esq->altd + 1;
 return aux1;
}

ARVORE* rotacao_direita(ARVORE* aux)
{
 ARVORE *aux1, *aux2;
 aux1 = aux->esq;
 aux2 = aux1->dir;
 aux->esq = aux2;
 aux1->dir = aux;
 if (aux->esq == NULL)
 aux->alte = 0;
 else if (aux->esq->alte > aux->esq->altd)

```

```
aux->alte = aux->esq->alte+1;
else
 aux->alte = aux->esq->altd+1;

if (aux1->dir->alte > aux1->dir->altd)
 aux1->altd = aux1->dir->alte + 1;
else
 aux1->altd = aux1->dir->altd + 1;
return aux1;
}

ARVORE* balanceamento(ARVORE *aux)
{
 int d, df;
 d = aux->altd - aux->alte;
 if (d == 2)
 {
 df = aux->dir->altd - aux->dir->alte;
 if (df >= 0)
 {
 aux = rotacao_esquerda(aux);
 }
 else
 {
 aux->dir = rotacao_direita(aux->dir);
 aux = rotacao_esquerda(aux);
 }
 }
 else if (d == -2)
 {
 df = aux->esq->altd - aux->esq->alte;
 if (df <= 0)
 {
 aux = rotacao_direita(aux);
 }
 else
 {
 aux->esq = rotacao_esquerda(aux->esq);
 aux = rotacao_direita(aux);
 }
 }
 return aux;
}
```

```

ARVORE* inserir(ARVORE *aux, int num)
{
 // o objeto novo é um objeto auxiliar
 ARVORE *novo;
 if (aux == NULL)
 {
 novo = new ARVORE();
 novo->num = num;
 novo->altd = 0;
 novo->alte = 0;
 novo->esq = NULL;
 novo->dir = NULL;
 aux = novo;
 }
 else if (num < aux->num)
 {
 aux->esq = inserir(aux->esq, num);
 if (aux->esq->altd > aux->esq->alte)
 aux->alte = aux->esq->altd + 1;
 else
 aux->alte = aux->esq->alte + 1;
 aux = balanceamento(aux);
 }
 else
 {
 aux->dir = inserir(aux->dir, num);
 if (aux->dir->altd > aux->dir->alte)
 aux->altd = aux->dir->altd + 1;
 else
 aux->altd = aux->dir->alte + 1;
 aux = balanceamento(aux);
 }
 return aux;
}

int consultar(ARVORE* aux, int num, int achou)
{
 if (aux != NULL && achou == 0)
 {
 if (aux->num == num)
 achou = 1;
 else if (num < aux->num)
 achou = consultar(aux->esq, num, achou);
 }
}

```

```
 else
 achou = consultar(aux->dir, num, achou);
 }
 return achou;
}

void mostraremordem(ARVORE* aux)
{
 if (aux != NULL)
 {
 mostraremordem(aux->esq);
 cout << aux->num << " ";
 mostraremordem(aux->dir);
 }
}

void mostrarpreordem(ARVORE* aux)
{
 if (aux != NULL)
 {
 cout << aux->num << " ";
 mostrarpreordem(aux->esq);
 mostrarpreordem(aux->dir);
 }
}

void mostrarposordem(ARVORE* aux)
{
 if (aux != NULL)
 {
 mostrarposordem(aux->esq);
 mostrarposordem(aux->dir);
 cout << aux->num << " ";
 }
}

ARVORE* remover(ARVORE* aux, int num)
{
 ARVORE *p, *p2;
 if (aux->num == num)
 {
 if (aux->esq == aux->dir)
 {
 // o elemento a ser removido não tem filhos
```

```

 delete aux;
 return NULL;
}
else if (aux->esq == NULL)
{
// o elemento a ser removido
// não tem filho para a esquerda
p = aux->dir;
delete aux;
return p;
}
else if (aux->dir == NULL)
{
// o elemento a ser removido
// não tem filho para
// a direita
p = aux->esq;
delete aux;
return p;
}
else
{
// o elemento a ser removido
// tem filho para ambos os lados
p2 = aux->dir;
p = aux->dir;
while (p->esq != NULL)
{
 p = p->esq;
}
p->esq = aux->esq;
delete aux;
return p2;
}
}
else if (aux->num < num)
 aux->dir = remover(aux->dir, num);
else
 aux->esq = remover(aux->esq, num);
return aux;
}

ARVORE* atualiza(ARVORE *aux)
{
if (aux!=NULL)

```

```

 {
 aux->esq = atualiza(aux->esq);
 if (aux->esq==NULL)
 aux->alte = 0;
 else if (aux->esq->alte > aux->esq->altd)
 aux->alte = aux->esq->alte + 1;
 else
 aux->alte = aux->esq->altd + 1;

 aux->dir = atualiza(aux->dir);
 if (aux->dir==NULL)
 aux->altd = 0;
 else if (aux->dir->alte > aux->dir->altd)
 aux->altd = aux->dir->alte + 1;
 else
 aux->altd = aux->dir->altd + 1;
 aux = balanceamento(aux);
 }
 return aux;
}

ARVORE* desalocar(ARVORE* aux)
{
 if (aux!=NULL)
 {
 aux->esq=desalocar(aux->esq);
 aux->dir=desalocar(aux->dir);
 delete aux;
 }
 return NULL;
}

void main()
{
 // a árvore está vazia, logo,
 // o ponteiro raiz tem o valor null
 ARVORE *raiz = NULL;
 // o ponteiro aux é um ponteiro auxiliar
 ARVORE *aux;
 // o ponteiro aux1 é um ponteiro auxiliar
 int op, achou, numero;
 do
 {
 clrscr();
 cout << "\nMENU DE OPCÕES\n";

```

```
cout << "\n1 - Inserir na árvore";
cout << "\n2 - Consultar um nó da árvore";
cout << "\n3 - Consultar toda a árvore em
 ↪ordem";
cout << "\n4 - Consultar toda a árvore em pré-
 ↪ordem";
cout << "\n5 - Consultar toda a árvore em pós-
 ↪ordem";
cout << "\n6 - Excluir um nó da árvore";
cout << "\n7 - Esvaziar a árvore";
cout << "\n8 - Sair";
cout << "\nDigite sua opção: ";
cin >> op;
if (op < 1 || op > 8)
 cout << "\nOpção inválida!!";
else if (op == 1)
{
 cout << "\nDigite o número a ser inserido na
 ↪árvore: ";
 cin >> numero;
 raiz = inserir(raiz,numero);
 cout << "\nNúmero inserido na árvore!!";
}
else if (op == 2)
{
 if (raiz == NULL)
 {
 // a árvore está vazia
 cout << "\nÁrvore vazia!!";
 }
 else
 {
 // a árvore contém elementos
 cout << "\nDigite o elemento a ser
 ↪consultado: ";
 cin >> numero;
 achou = 0;
 achou = consultar(raiz,numero,achou);
 if (achou == 0)
 cout << "\nNúmero não encontrado na
 ↪árvore!";
 else
 cout << "\nNúmero encontrado na
 ↪árvore!";
 }
}
```

```
 else if (op == 3)
 {
 if (raiz == NULL)
 {
 // a árvore está vazia
 cout << "\nÁrvore vazia!!";
 }
 else
 {
 // a árvore contém elementos e
 // estes serão mostrados em ordem
 cout << "\nListando todos os elementos
 da árvore em ordem: ";
 mostraremordem(raiz);
 }
 }
 else if (op == 4)
 {
 if (raiz == NULL)
 {
 // a árvore está vazia
 cout << "\nÁrvore vazia!!";
 }
 else
 {
 // a árvore contém elementos e
 // estes serão mostrados em pré-ordem
 cout << "\nListando todos os elementos
 da árvore em pré-ordem: ";
 mostrarpreadordem(raiz);
 }
 }
 else if (op == 5)
 {
 if (raiz == NULL)
 {
 // a árvore está vazia
 cout << "\nÁrvore vazia!!";
 }
 else
 {
 // a árvore contém elementos e
 // estes serão mostrados em pós-ordem
 cout << "\nListando todos os elementos
 da árvore em pós-ordem: ";
```

```
 aux = raiz;
 mostrarpesordem(aux);
}
}
else if (op == 6)
{
 if (raiz == NULL)
 cout << "\nÁrvore vazia!";
 else
 {
 cout << "\nDigite o número que deseja
 excluir: ";
 cin >> numero;
 achou = 0;
 achou = consultar(raiz,numero,achou);
 if (achou == 0)
 cout << "\nNúmero não encontrado na
 árvore!";
 else
 {
 raiz = remover(raiz,numero);
 raiz = atualiza(raiz);
 cout << "\nNúmero excluído da
 árvore!";
 }
 }
}
else if (op == 7)
{
 if (raiz == NULL)
 cout << "\nÁrvore vazia!";
 else
 {
 raiz=desalocar(raiz);
 cout << "\nÁrvore
 esvaziada!";
 }
}
getch();
}
while (op != 8);
// desalocando a arvore
raiz = desalocar(raiz);
}
```

## Análise da complexidade

Para árvores平衡adas, como é o caso da árvore AVL, o custo das operações se mantém na mesma ordem de grandeza das árvores binárias.

Cada nó da árvore AVL armazena a altura da sua sub-árvore esquerda e direita, o que facilita as operações de inserção e remoção e torna o custo dessas operações na ordem de  $O(\log n)$ .

Ao se inserir um novo nó  $u$  numa árvore, o pai desse nó (chamado genericamente de  $v$ ) terá a altura de uma de suas sub-árvores alterada. Com isso, é necessário verificar se essa sub-árvore de raiz  $v$  está desbalanceada ou não. A verificação é feita apenas subtraindo-se as alturas das duas sub-árvores de  $v$ , cujos valores estão armazenados no próprio nó  $v$ . Caso ocorra o desbalanceamento, será necessário realizar uma rotação simples ou dupla, que consiste em atualizar alguns ponteiros de nós, gerando um custo constante.

No entanto, estando o nó  $v$  balanceado ou não, outros nós no caminho de  $v$  até a raiz podem também ficar desbalanceados e a verificação deverá ser feita. Com isso, o percurso do nó até a raiz da árvore é feito em  $O(\log n)$  passos.

A operação de exclusão de algum nó também pode ser feita em  $O(\log n)$  passos. Para isso, após a exclusão do nó, deve-se verificar se a árvore ficou desbalanceada e examinar os nós no caminho da raiz até alguma folha. Nesse caso, pode ser que mais de uma rotação seja necessária para balancear a árvore fazendo com que o número de rotações necessárias possa alcançar a ordem  $O(\log n)$ .

## Exercícios

- Faça um programa para executar as operações abaixo em uma árvore binária.

Menu

- 1** – Inserir número
- 2** – Mostrar todos os números
- 3** – Mostrar o maior número
- 4** – Mostrar o menor número
- 5** – Sair

- Faça um programa para executar as operações abaixo em uma árvore binária.

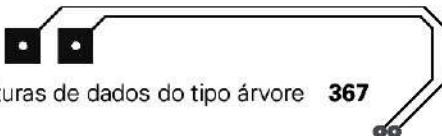
Menu

- 1** – Inserir número
- 2** – Mostrar os nós folha
- 3** – Mostrar os nós ancestrais de um nó
- 4** – Mostrar os nós descendentes de um nó
- 5** – Mostrar o nó pai e os nós filhos de um nó
- 6** – Sair

- Faça um programa para executar as operações abaixo em uma árvore binária.

Menu

- 1** – Inserir número



- 2 – Mostrar todos**
  - 3 – Mostrar a sub-árvore direita de um nó**
  - 4 – Mostrar a sub-árvore esquerda de um nó**
  - 5 – Sair**
- 4.** Redesenhando a árvore a cada remoção, desenhe uma árvore AVL com os números a seguir: 50, 30, 55, 10, 15, 20, 80, 90, 68. A seguir, remova os números 50 e 90.
  - 5.** Redesenhando a árvore a cada remoção, desenhe uma árvore AVL com os números a seguir: 50, 180, 200, 190, 198. A seguir, remova os números 50 e 200.

# 8

# Algoritmos em grafos

Dentre os vários problemas já existentes em computação, além dos que surgem a cada dia, é comum deparar-se com problemas de conexão entre elementos. Os seguintes problemas ou situações podem ser representados pela teoria dos grafos: a malha de estradas que conectam cidades; o conjunto de links de um *website*; os circuitos que compõem a placa-mãe de um computador; os cômodos de uma casa, as portas que interligam esses cômodos e a distribuição entre disciplinas e professores de uma escola.

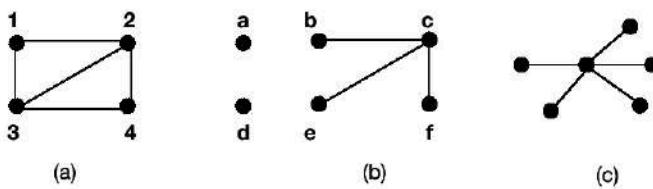
## Conceitos de grafos

Um grafo  $G$  é formado pelo par de conjuntos  $V$  e  $E$ , sendo  $V$  o conjunto de vértices de  $G$ , e  $E$  o conjunto de arestas de  $G$ . Uma aresta  $e \in E(G)$  é representada por  $e = (u, v)$ , e sempre interliga dois vértices quaisquer  $u$  e  $v$  de  $V$ . Dois vértices ligados por uma aresta são denominados *adjacentes*. As seguintes notações serão utilizadas para denotar um grafo:  $G = (V, E)$  ou  $G = (V(G), E(G))$  ou  $G(V, E)$ .

A representação geométrica dos grafos é feita marcando pontos distintos no plano para representar os vértices, e uma linha ligando dois pontos para representar a aresta. Três exemplos de grafos são apresentados na Figura 8.1.

Os vértices podem ou não possuir nomes. Quando recebem nomes, estes podem ser números ou letras. Quando são nomeados por letras, é necessário fazer um mapeamento de cada nome de vértice para um número correspondente  $i$ , sendo  $1 \leq i \leq n$ ,  $n \in N$ .

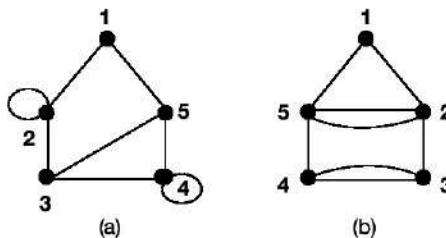
**Figura 8.1** Exemplos de grafos



## ● Laços, arestas múltiplas e multigrafo

É possível encontrar em um grafo arestas do tipo  $e = (u, u)$ , ou seja, com as extremidades da aresta iguais. Quando isso ocorre, a aresta é denominada *laço*. É também possível a existência de mais de uma aresta entre o mesmo par de vértices, chamadas *arestas paralelas* ou *arestas múltiplas*. Um grafo que possui arestas paralelas é denominado *multigrafo*. Na Figura 8.2(a) é apresentado um grafo com laços, e na Figura 8.2(b) é apresentado um multigrafo.

**Figura 8.2** (a) Grafo com laços; (b) Grafo com arestas múltiplas, ou seja, multigrafo



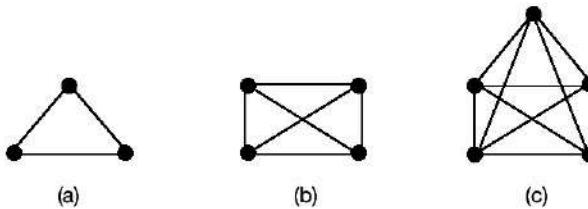
## ● Grafo trivial e grafo simples

Um grafo com apenas um vértice é chamado de grafo *trivial*. Um grafo  $G$  é chamado grafo *simples* se não possui laço ou aresta múltipla. Os grafos da Figura 8.2 não são grafos simples, já os grafos da Figura 8.1 são.

## ● Grafo completo

Um grafo  $G$  é chamado grafo *completo* quando existe uma aresta para cada par de vértices distintos de  $G$ . Na Figura 8.3 são apresentados exemplos de grafos completos com 3, 4 e 5 vértices, respectivamente.

**Figura 8.3** Grafos completos



## ● Grafo regular

Dado um grafo  $G(V, E)$ , o grau de um vértice  $v \in V$ , denotado por  $g(v)$ , é o número de arestas que incidem nele. No grafo da Figura 8.3(b), o grau de cada vértice é 3. No

grafo da Figura 8.1(a), o grau do vértice 1 é 2, o grau do vértice 4 é 2, e o grau dos vértices 2 e 3 é 3. No grafo da Figura 8.2(a), o vértice 2 possui grau 4, uma vez que a aresta incide no mesmo vértice; e o vértice 4 na Figura 8.2(b) possui grau 3.

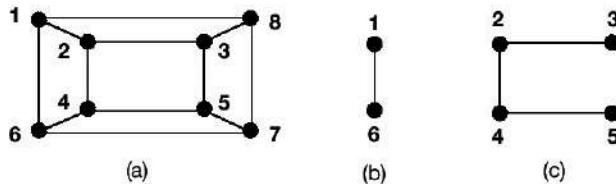
Considerando que em cada vértice  $v \in V$  incidem  $g(v)$  arestas e que cada aresta incide em 2 vértices, logo,  $\sum_{v \in V} g(v) = 2 \cdot |E|$ , onde  $|E|$  representa o número de arestas do conjunto  $E$ .

Quando todos os vértices de um grafo  $G$  possuem o mesmo grau, ele é chamado grafo *regular* de grau  $r$ . Todos os grafos da Figura 8.3 são grafos regulares, onde os vértices do grafo(a) possuem grau  $r = 2$ , os vértices do grafo(b) possuem grau  $r = 3$ , e no grafo(c) os vértices possuem grau  $r = 4$ .

## Subgrafo

Dado um grafo  $G(V, E)$ ,  $H(V', E')$ , é dito um *subgrafo* de  $G$  se  $V' \subseteq V$  e  $E' \subseteq E$ . A Figura 8.4 apresenta um exemplo de grafo  $G$  e dois subgrafos de  $G$ .

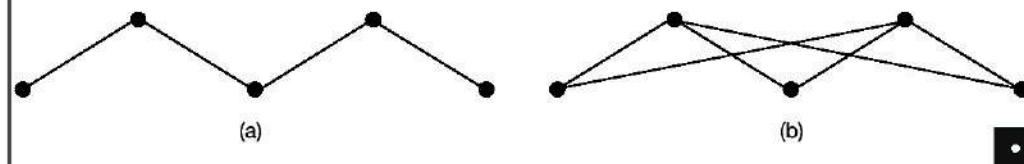
**Figura 8.4** (a) Grafo G; (b) e (c) Subgrafos de G



## Grafo bipartido

Um grafo  $G(V, E)$  é chamado *bipartido* quando seu conjunto de vértices  $V$  pode ser particionado em dois subconjuntos  $V_1$  e  $V_2$ , tal que toda aresta de  $G$  faz a ligação de um vértice de  $V_1$  a um vértice de  $V_2$ . Logo, um grafo bipartido  $G$  pode ser escrito como  $G(V_1 \cup V_2, E)$ . Um grafo bipartido completo  $G(V_1 \cup V_2, E)$  é um grafo bipartido tal que existe uma aresta para todo par de vértices  $u, v$ , sendo  $u \in V_1$  e  $v \in V_2$ . Um grafo bipartido completo é denotado por  $K_{m,n}$ , sendo  $m = |V_1|$  e  $n = |V_2|$ . A Figura 8.5 mostra dois exemplos de grafos bipartidos.

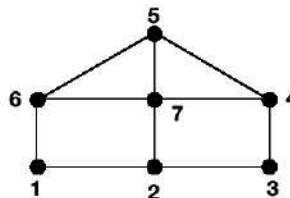
**Figura 8.5** (a) Grafo bipartido; (b) Grafo bipartido completo



## Caminho simples, trajeto e ciclos

Segundo Szwarcfiter et al. (1986), dado um grafo  $G(V, E)$ , uma sequência de vértices  $v_1, v_2, \dots, v_j$  tal que  $(v_i, v_{i+1}) \in E(G), 1 \leq i \leq j - 1$ , recebe o nome de *caminho* de  $v_1$  a  $v_j$ . O número de arestas do caminho é denominado *comprimento* do caminho. Quando todos os vértices do caminho são distintos, sua sequência recebe o nome de *caminho simples* ou *elementar*. Se as arestas forem distintas, a sequência recebe o nome de *trajeto*. No grafo da Figura 8.6, a sequência 1,6,7,5 é um caminho simples, enquanto 4,7,2,3,4,5 é um trajeto. Um *ciclo* é um caminho  $v_1, v_2, \dots, v_j, v_{j+1}$ , sendo  $v_1 = v_{j+1}$  e  $j \geq 2$ . A sequência 6,7,2,1,6 é um ciclo na Figura 8.6. Já um grafo que não possui ciclos é chamado de *acíclico*.

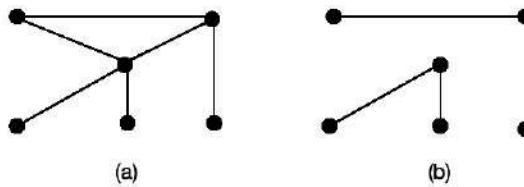
**Figura 8.6** Caminho e ciclos



## Grafo conexo e desconexo

Um grafo  $G$  é chamado *conexo* se existe um caminho para cada par de vértices de  $G$ . Caso contrário, é chamado *desconexo*. Um grafo desconexo possui partes conexas que são chamadas de *componentes*. Um grafo conexo possui uma única componente conexa enquanto um grafo desconexo possui várias componentes conexas. A Figura 8.7(a) mostra um grafo conexo com número de componentes igual a 1, enquanto que a Figura 8.7(b) mostra um grafo desconexo com número de componentes igual a 3.

**Figura 8.7** (a) Grafo conexo; (b) Grafo desconexo

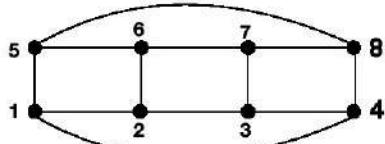


## Grafos isomorfos

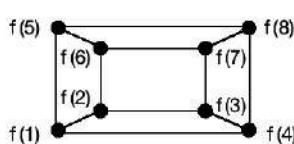
Segundo Szwarcfiter et al. (1986), dois grafos  $G_1(V_1, E_1)$  e  $G_2(V_2, E_2)$ , com  $|V_1| = |V_2| = n$ , são ditos isomorfos entre si, quando existe uma função unívoca  $f : V_1 \rightarrow V_2$ , tal que  $(u, v)$

$\in E_1$  se e somente se  $(f(u), f(v)) \in E_2$ , para todo  $u, v \in E_1$ . Por exemplo, as representações geométricas dos grafos da Figura 8.8 se tornam coincidentes ao se aplicar a função  $f$  indicada na figura. Logo,  $G_1$  e  $G_2$  são isomorfos entre si, já os grafos  $G_1$  e  $G_3$  não o são, pois não existe uma função  $f$  que realize esse mapeamento entre vértices e que faça com que as representações tornem-se coincidentes.

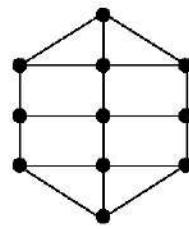
**Figura 8.8** Grafos isomorfos



(a)  $G_1$



(a)  $G_2$



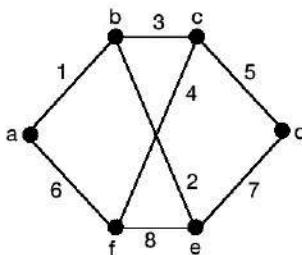
(a)  $G_3$

Fonte: Szwarcfiter et al. (1986).

## Grafo ponderado

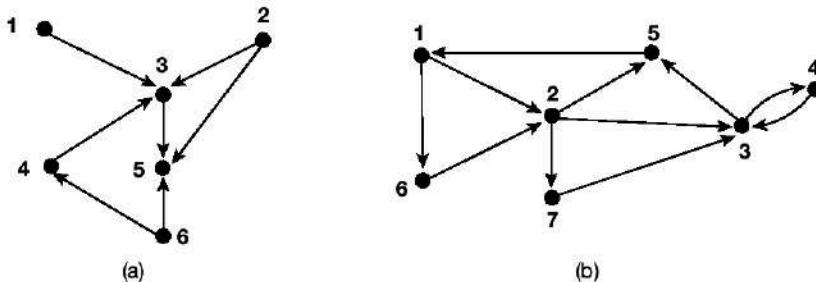
Um grafo ponderado é aquele com peso nas arestas. Esses pesos podem representar custos ou distâncias, por exemplo. Um grafo ponderado é mostrado na Figura 8.9.

**Figura 8.9** Grafo ponderado



## Dígrafo

Os grafos vistos até agora são chamados grafos não direcionados, pois suas arestas não possuem orientação. Um grafo direcionado  $D(V,E)$ , também chamado de *dígrafo*, possui um conjunto não vazio de vértices  $V$  e um conjunto de arestas  $E$ , tal que para toda aresta  $(u,v) \in E$  existe uma única direção de  $u$  para  $v$ . Exemplos de dígrafos podem ser vistos na Figura 8.10.

**Figura 8.10** Dígrafos

Os conceitos de trajeto, caminho e ciclo são análogos aos conceitos do grafo orientado, devendo apenas respeitar a orientação das arestas. A diferença é que nos grafos orientados encontram-se ciclos de comprimento 2, que ocorre quando o grafo possui as duas arestas  $(u,v)$  e  $(v,u)$ . A Figura 8.10(b) mostra um exemplo de dígrafo com ciclo entre os vértices 3 e 4 de comprimento 2. Na mesma figura, a sequência de vértices 2, 3, 5, 1, 6 é um caminho entre os vértices 2 e 6.

Seja  $D(V,E)$  um dígrafo e um vértice  $v \in V$ . O *grau de entrada* de  $v$  é o número de arestas que incidem em  $v$ . O *grau de saída* é o número de arestas que partem de  $v$ . Na Figura 8.10(b), o vértice 3 possui grau de entrada 3 e grau de saída 2. Um vértice com grau de entrada nulo é chamado vértice *fonte* e um vértice com grau de saída zero é chamado vértice *sumidouro*. Na Figura 8.10(a), o vértice 1 é chamado fonte e o 5 é chamado sumidouro.

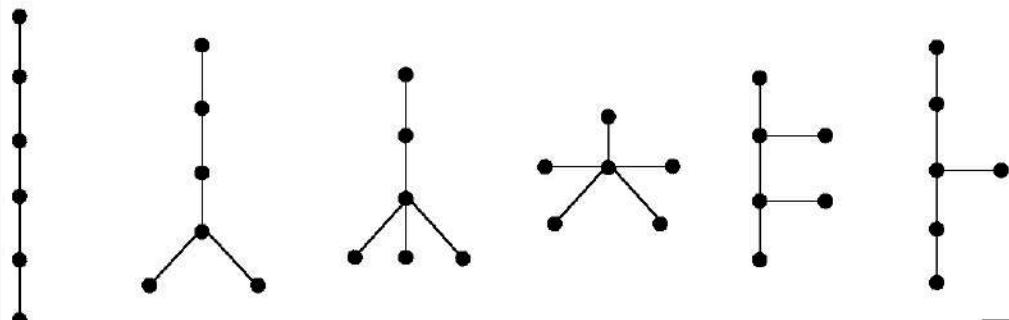
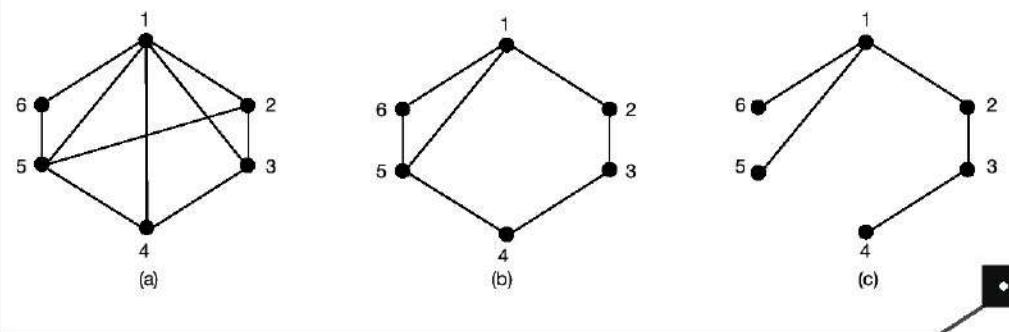
## Árvores

Um grafo  $T(V, E)$  que não possui ciclos e é conexo é chamado *árvore*. Toda árvore possui as seguintes características:

- Seja  $v \in V$ , se  $v$  possui grau menor ou igual a 1, então  $v$  é uma *folha*. Caso o grau seja maior que 1,  $v$  é um vértice *interno*.
- Uma árvore  $T$  com  $n$  vértices possui  $n-1$  arestas.
- Um grafo  $G$  é uma árvore somente se existir um único caminho entre cada par de vértices de  $G$ . A prova desse resultado pode ser encontrada em *Estruturas de dados e seus algoritmos* (SZWARCFITER, 1986).
- Um conjunto de árvores é chamado de *floresta*.

Na Figura 8.11 são apresentadas todas as possíveis árvores com 6 vértices.

Dado um grafo  $G(V(G), E(G))$ , denomina-se *subgrafo gerador* o grafo  $H(V(H), E(H))$  que é subgrafo de  $G$ , tal que  $V(H) = V(G)$ . Se o subgrafo  $H$  é uma árvore, então é chamado de *árvore geradora*. Na Figura 8.12 é apresentado um grafo  $G$ , um subgrafo gerador de  $G$  e uma árvore geradora de  $G$ , respectivamente.

**Figura 8.11** Árvores com 6 vértices**Figura 8.12** (a) Grafo  $G$ ; (b) Subgrafo gerador; (c) Árvore geradora

## Corte de vértice e aresta

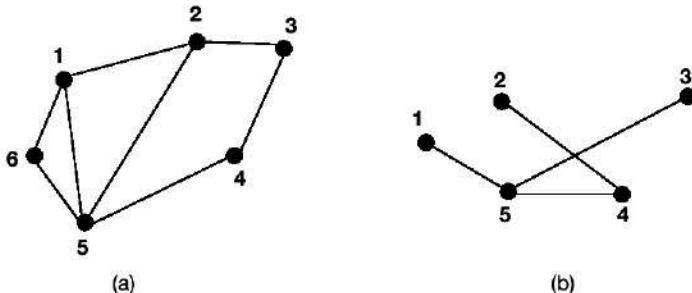
Um *corte de vértices* em um grafo  $G(V, E)$  é um subconjunto mínimo de vértices  $V_1$ , sendo  $V_1 \subseteq V$  tal que se os vértices de  $V_1$  forem removidos do grafo  $G$ , o grafo fica desconexo ou se transforma no grafo trivial. Do mesmo modo, um *corte de arestas* em um grafo  $G(V, E)$  é um subconjunto mínimo de arestas  $E_1$ , sendo  $E_1 \subseteq E$  tal que se as arestas de  $E_1$  forem removidas do grafo  $G$ , o grafo fica desconexo. Considerando o grafo da Figura 8.13(a), o conjunto de vértices  $\{1, 4\}$  constitui um corte de vértices. Já o conjunto de arestas  $\{(2, 3), (5, 4)\}$ , do grafo (a) da Figura 8.13, constitui um corte de arestas.

Um vértice em um grafo  $G$  é chamado *vértice de corte* se, ao ser retirado do grafo, e consequentemente todas as arestas que incidiam nele, o grafo torna-se desconexo.

Uma aresta em um grafo  $G$  é chamada *aresta de corte* se, ao ser retirada do grafo, ele torna-se desconexo.

Em uma árvore, todas as arestas são de corte. Além disso, todo vértice com grau maior que 1 é um vértice de corte.

No grafo da Figura 8.13(b), todas as arestas são de corte; no mesmo grafo, os vértices 4 e 5 são de corte.

**Figura 8.13** Corte de vértices e arestas

## Representação de grafos

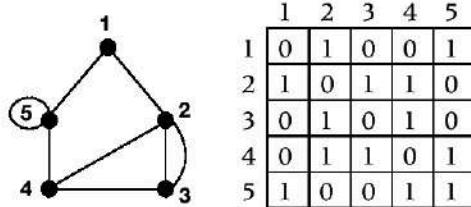
Um grafo  $G(V, E)$  com  $|V| = n$  pode ser representado adequadamente por meio de matrizes ou listas.

### Matriz de adjacências

Dado um grafo  $G(V, E)$ , uma matriz de adjacências  $M$  é formada por  $n$  linhas e  $n$  colunas, sendo  $n$  o número de vértices do grafo. A matriz é preenchida da seguinte forma:

$$M_{ij} = \begin{cases} 1, & \text{se } (i, j) \in E(G) \\ 0, & \text{se } (i, j) \notin E(G) \end{cases}$$

A Figura 8.14 apresenta um grafo não direcionado e sua matriz de adjacências.

**Figura 8.14** Matriz de adjacências para um grafo não direcionado

Uma matriz de adjacências é simétrica para grafos não direcionados.

Além disso, o número de 1s é igual a  $2 * |E|$ , pois cada aresta  $(v_j, v_i)$  contribui com dois 1s em  $M$ ,  $M_{ij}$  e  $M_{ji}$ , exceto no caso em que o grafo possui laços ou arestas múltiplas.

A matriz de adjacências também pode ser definida para um dígrafo. Para cada aresta  $(v_j, v_i)$ , partindo de  $v_i$  e chegando em  $v_j$ , somente a posição  $M_{ij}$  será preenchida com 1. Se não existir a aresta  $(v_j, v_i)$ , a posição  $M_{ji}$  receberá o valor 0. Nesse caso, a matriz pode não ser simétrica e o número de 1s é igual ao número de arestas do grafo.

O espaço reservado para o armazenamento das informações nessa matriz é da ordem  $O(|V|^2)$ , logo, é indicado para utilizar quando se tratar de grafos densos, em que o número de arestas é próximo a  $|V|^2$ . Ela se torna desvantajosa para grafos esparsos em que o número de arestas é bem menor que  $|V|^2$ .

## Matriz de incidências

Segundo Rabuske (1992), dado um grafo  $G(V, E)$  de  $n$  vértices e  $m$  arestas, a matriz de incidência de  $G$ , denotada por  $M_{n \times m}$ , é definida por:

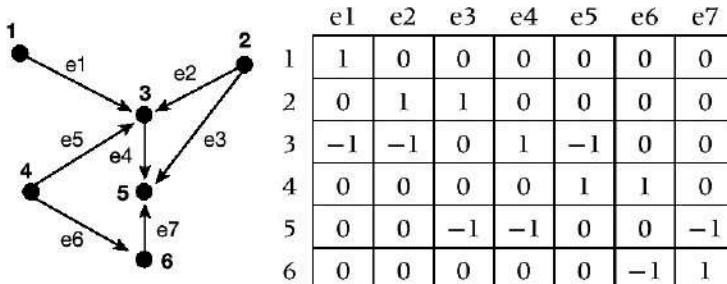
$$M_{ij} = \begin{cases} 1, & \forall v_i, se(v_i, v_j) \in E(G) \\ 0, & se(v_i, v_j) \notin E(G) \\ 0, & se(v_i, v_i) \in E(G) \end{cases}$$

No caso do grafo ser orientado, então a matriz é definida como:

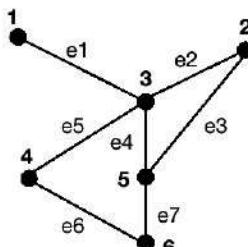
$$M_{ij} = \begin{cases} 1, & \forall v_i, se(v_i, v_j) \in E(G) \\ -1, & \forall v_j, se(v_i, v_j) \in E(G) \\ 0, & se(v_i, v_j) \notin E(G) \\ 0, & se(v_i, v_i) \in E(G) \end{cases}$$

Na Figura 8.15 são mostrados um grafo orientado e sua matriz de incidência.

**Figura 8.15** Grafo orientado e sua matriz de incidência



Na Figura 8.16 são mostrados um grafo não orientado e sua matriz de incidência.

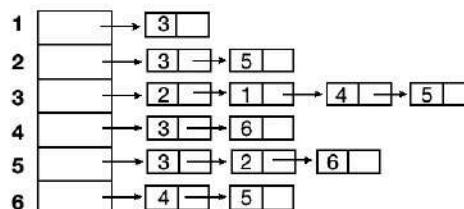
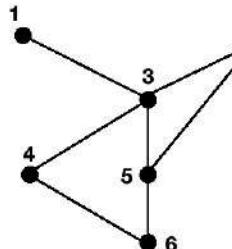
**Figura 8.16** Grafo não orientado e sua matriz de incidência

|   | e1 | e2 | e3 | e4 | e5 | e6 | e7 |
|---|----|----|----|----|----|----|----|
| 1 | 1  | 0  | 0  | 0  | 0  | 0  | 0  |
| 2 | 0  | 1  | 1  | 0  | 0  | 0  | 0  |
| 3 | 1  | 1  | 0  | 1  | 1  | 0  | 0  |
| 4 | 0  | 0  | 0  | 0  | 1  | 1  | 0  |
| 5 | 0  | 0  | 1  | 1  | 0  | 0  | 1  |
| 6 | 0  | 0  | 0  | 0  | 0  | 1  | 1  |

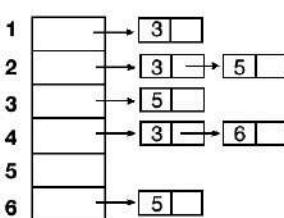
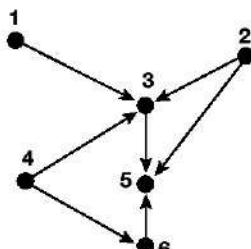
## Lista de adjacências

Uma lista de adjacências para um grafo  $G(V, E)$  consiste em um vetor  $Adj$  com  $n = |V|$  entradas, uma para cada vértice do grafo. Cada entrada  $Adj[v]$  possui uma lista encadeada de vértices adjacentes a  $v$  em  $G$ . Não existe uma ordem dos vértices nessa lista.

A Figura 8.17(a) mostra um exemplo de grafo não direcionado e sua representação por lista de adjacências. A Figura 8.17(b) mostra um exemplo de dígrafo e sua representação por lista de adjacências.

**Figura 8.17** (a) Lista de adjacências para grafo não orientado; (b) Lista de adjacências para grafo orientado

(a)



(b)

A lista de adjacências consiste de  $n$  listas contendo  $2m$  elementos, onde  $m$  é o número de arestas do grafo. Logo, o espaço utilizado pela lista é da ordem de  $O(n + m)$ .

## Algoritmos de busca

A busca é um dos procedimentos mais utilizados em algoritmos na teoria dos grafos. Ela tem o objetivo de mostrar uma sistemática de como passear pelos vértices e arestas de um grafo. A seguir são apresentadas duas formas de busca bastante conhecidas na teoria dos grafos.

Para passear ou caminhar pelos vértices e arestas, utiliza-se *marcar* um vértice quando este já foi visitado, alcançado ou verificado no algoritmo. Isso é feito para que não haja repetição de vértices e eles possam ser distinguidos daqueles que ainda não foram processados por determinado algoritmo.

Dado um grafo conexo  $G(V, E)$  qualquer, considere que todos os seus vértices estão na condição de desmarcados. Considere agora que um vértice  $u$  qualquer seja marcado e que a aresta  $(u, v) \in E(G)$  ainda não foi selecionada. Então, a aresta  $(u, v)$  torna-se selecionada e o vértice  $v$  marcado. A ideia geral de uma busca em um grafo  $G$  pode ser descrita dessa maneira se o mesmo processo continuar sendo aplicado a partir do vértice  $v$ , até que todas as outras arestas do grafo sejam selecionadas.

O termo *visita* será aplicado a vértices e arestas toda vez que forem alcançados.

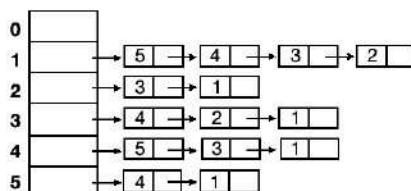
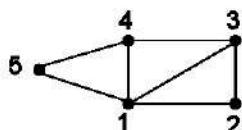
Nos algoritmos de busca deste capítulo os grafos serão representados através da lista de adjacências.

### Algoritmo de busca em profundidade

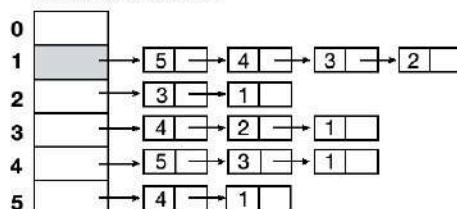
Segundo Szwarcfiter *et al.* (1986), uma busca em profundidade é aquela que atende ao seguinte critério para explorar um vértice marcado: dentre os vários marcados e incidentes a alguma aresta ainda não explorada, escolher aquele vértice mais recentemente alcançado na busca.

A ordem de visita das arestas incidentes a um vértice depende da ordem em que seus vizinhos foram inseridos na lista de adjacências, que no caso é arbitrária. Considerando o grafo da ilustração a seguir, suponha que uma busca em profundidade será realizada a partir do vértice 1. Inicialmente, visita-se o vértice 1, em seguida o primeiro vizinho de 1, que é 5, em seguida, um vizinho do vértice 5, que é 4, em seguida um vizinho de 4, no caso 3, mas já foi visitado. Então, busca-se o próximo vizinho de 4, que é 3. Visita-se então o vizinho de 3, o vértice 4, que já foi visitado. Passa-se para o próximo vizinho de 3, no caso 2. Visita-se agora o vizinho de 2, o primeiro que é 3 já foi visitado, e o próximo vizinho de 2 é 1, também já foi visitado. Dessa maneira, visita-se todos os vértices em profundidade possíveis a partir do vértice 1.

Considerando o grafo a seguir e sua lista de adjacências, ilustraremos a busca em profundidade a partir do vértice 1.

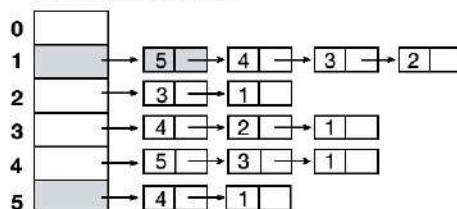


vértice origem: 1  
marca-se o vértice 1



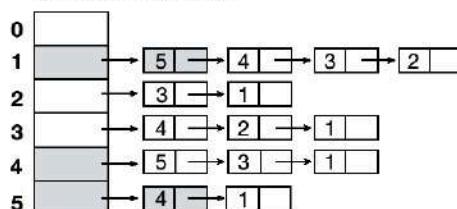
próximo vértice: 1<sup>a</sup> vizinho do 1 (vértice 5)

marca-se o vértice 5

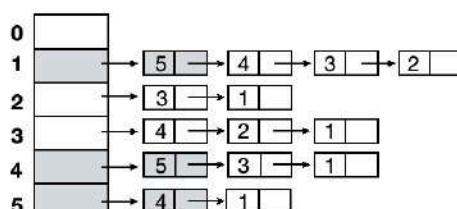


próximo vértice: 1<sup>a</sup> vizinho do 5 (vértice 4)

marca-se o vértice 4

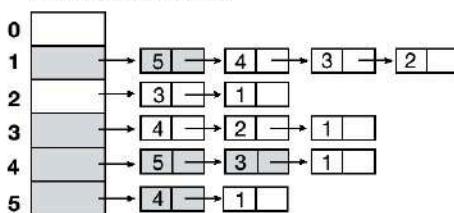


próximo vértice: 1<sup>a</sup> vizinho do 4 (vértice 5)

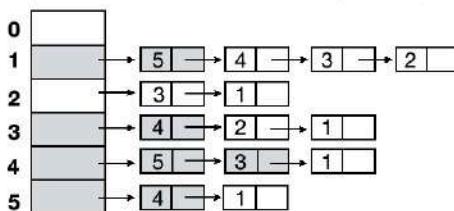


como o vértice 5 já foi marcado, busca-se o próximo vértice  
próximo vértice: 2<sup>a</sup> vizinho do 4 (vértice 3)

marca-se o vértice 3



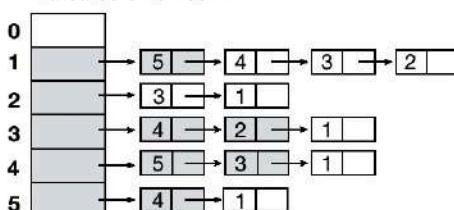
próximo vértice: 1º vizinho do 3 (vértice 4)



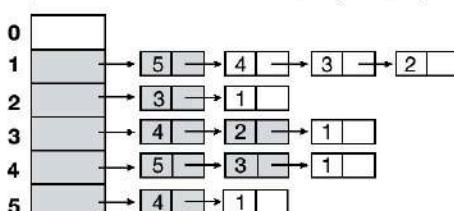
como o vértice 4 já foi marcado, busca-se o próximo vértice

próximo vértice: 2º vizinho do 3 (vértice 2)

marca-se o vértice 2

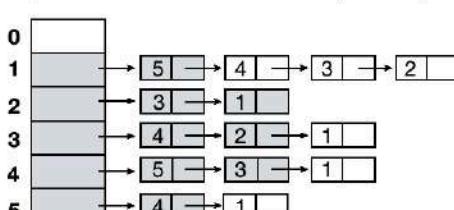


próximo vértice: 1º vizinho do 2 (vértice 3)

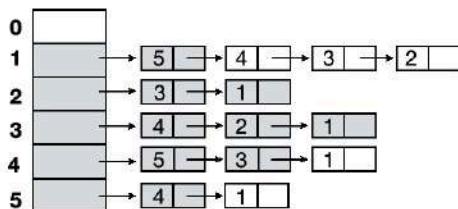


como o vértice 3 já foi marcado, busca-se o próximo vértice

próximo vértice: 2º vizinho do 2 (vértice 1)

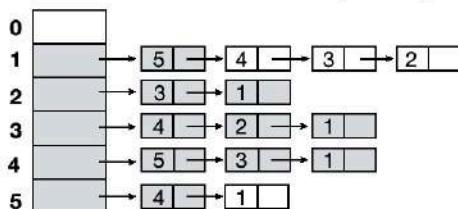
como o vértice 1 já foi marcado e todos os vizinhos do 2 já foram verificados  
na recursividade, volta-se para o vértice 3

próximo vértice: 3º vizinho do 3 (vértice 1)



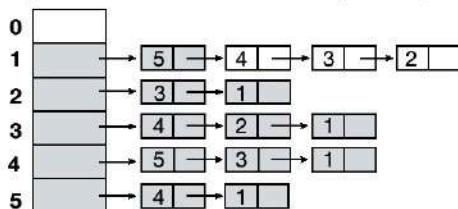
como o vértice 1 já foi marcado e todos os vizinhos do 3 já foram verificados na recursividade, volta-se para o vértice 4

próximo vértice: 3º vizinho do 4 (vértice 1)



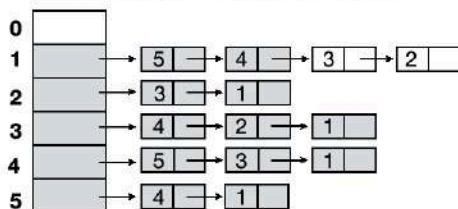
como o vértice 1 já foi marcado e todos os vizinhos do 4 já foram verificados na recursividade, volta-se para o vértice 5

próximo vértice: 2º vizinho do 5 (vértice 1)

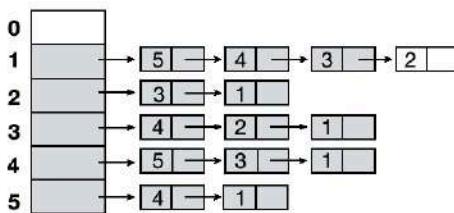


como o vértice 1 já foi marcado e todos os vizinhos do 5 já foram verificados na recursividade, volta-se para o vértice 1

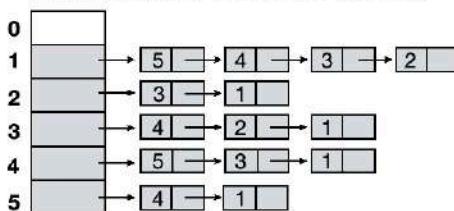
próximo vértice: 2º vizinho do 1 (vértice 4)



como o vértice 4 já foi marcado, busca-se o próximo vértice  
próximo vértice: 3º vizinho do 1 (vértice 3)



como o vértice 3 já foi marcado, busca-se o próximo vértice  
próximo vértice: 4º vizinho do 1 (vértice 2)



como o vértice 2 já foi marcado e todos os vizinhos do 1 já foram verificados,  
a busca termina



```
import java.util.Scanner;

public class buscap
{
 public static class vertice
 {
 int num;
 vertice prox;
 }
 public static class listaadj
 {
 public vertice listav;
 }
 public static class queue
 {
 int numv;
 queue prox;
 }
 public static void empilhar(int n)
 {
 queue novo = new queue();
 novo.numv = n;
 novo.prox = pilha;
```

```
 pilha = novo;
 }
 public static void desempilhar(int v)
 {
 if(pilha.numv==v)
 {
 pilha = pilha.prox;
 }
 }
 static queue pilha = null; // pilha p/armazenar
 // vertices
 // visitados
 static Scanner entrada = new Scanner(System.in);
 static int marcado[]; // vetor que armazena
 // se vértice for
 // marcado
 static listaadj Adj[]; // lista de adjacencias
 // entre
 // vértices

 public static void main(String args[])
 {
 vertice novo;

 int tam, org, dest, op, num, tipo;
 String menu;

 System.out.println("\n Tipo do grafo (1- não
 ↪ orientado, 2 - orientado:");
 tipo = entrada.nextInt();

 System.out.println("\n Digite número de vértices
 ↪ do grafo:");
 tam = entrada.nextInt();
 // alocação de memória
 Adj = new listaadj[tam+1];
 marcado = new int [tam+1];
 // inicialização de variáveis
 for(int i = 1; i <= tam; i++)
 {
 Adj[i]=new listaadj(); //alocação de memória
 marcado[i] = 0;
 }
 System.out.println("\n Arestas do grafo:
 ↪ VérticeOrigem (-1 para parar):");
```

```
org = entrada.nextInt();
System.out.println("\n Arestas do grafo:
 ↵ VérticeDestino(-1 para parar):");
dest = entrada.nextInt();
while(org != -1 && dest != -1)
{
 // alocando um nó com o valor do vértice
 // destino para
 // colocar na entrada
 // do vertice de origem da lista de
 // adjacencias --> (u,v)
 novo = new vertice();
 novo.num = dest;
 // inserindo vértice adjacente a vértice "org"
 // na lista de adjacencias
 novo.prox = Adj[org].listav;
 Adj[org].listav = novo;

 if(tipo==1){
 // inserindo (v,u)
 novo = new vertice();
 novo.num = org;
 // inserindo vértice adjacente a
 // vértice "org" na
 // lista de adjacencias
 novo.prox = Adj[dest].listav;
 Adj[dest].listav = novo;
 }
 // proxima entrada
 System.out.print("\n Arestas do grafo:
 ↵ VérticeOrigem(-1 para parar):");
 org = entrada.nextInt();
 System.out.print("\n Arestas do grafo:
 ↵ VérticeDestino(-1 para parar):");
 dest = entrada.nextInt();
}
do
{
 menu = "\n1-Busca em profundidade"+
 "\n2-Mostrar lista de adjacencias"+
 "\n4-Sair "+
 "\nDigite sua opcao: ";
 System.out.print(menu);
 op = entrada.nextInt();
```

```
switch(op)
{
 case 1: System.out.print("Digite um
 ↪ vértice de
 partida da
 busca:");
 num = entrada.nextInt();
 System.out.print(" "+num);
 buscaprof(Adj, tam, num);

 for(int i = 1; i <= tam; i++)
 marcado[i] = 0;
 break;
 case 2: mostrar_Adj(Adj, tam);
 break;
} //fim switch
} while (op!=4);
} //fim main

static void buscaprof(listaadj Adj[], int tam, int v)
{
 vertice vert;
 int w;

 marcado[v] = 1;
 // inserindo "v" numa pilha
 empilhar(v);
 for(int i = 1; i <= tam; i++)
 {
 // primeiro vizinho de "v"
 vert = Adj[v].listav;
 while(vert != null)
 {
 w = vert.num;
 if(marcado[w] != 1)
 {
 System.out.println(" "+w);
 buscaprof(Adj, tam, w);
 }
 // proximo vizinho de v
 vert = vert.prox;
 }
 }
 desempilhar(v);
}
static void mostrar_Adj(listaadj Adj[], int tam)
```

```

 {
 vertice v;
 for(int i=1; i < tam; i++)
 {
 v = Adj[i].listav;
 System.out.println("Entrada "+i+" ");
 while(v != null)
 {
 System.out.println("(+"+i+","+v. num
 +") " + " ");
 v=v.prox;
 }
 }
 }
}

```

---



```

#include<iostream.h>
#include<conio.h>

struct vertice
{
 int num;
 vertice *prox;
};

struct listaadj
{
 vertice *listav;
};

struct queue
{
 int numv;
 queue *prox;
};

void empilhar(queue* &pilha, int n)
{
 queue *novo = new queue();
 novo->numv = n;
 novo->prox = pilha;
}

```

```
pilha = novo;
}

void desempilhar(queue *&pilha, int v)
{
 if(pilha->numv==v)
 {
 queue* aux;

 aux=pilha;
 pilha = pilha->prox;
 delete aux;
 }
}

void buscaprof(listaadj Adj[], int tam, int v, int
 → marcado[], queue* &pilha)
{
 vertice *vert;
 int w;

 marcado[v] = 1;
 // inserindo "v" numa pilha
 empilhar(pilha, v);
 for(int i = 1; i <= tam; i++)
 {
 // primeiro vizinho de "v"
 vert = Adj[v].listav;
 while(vert != NULL)
 {
 w = vert->num;
 if(marcado[w] != 1)
 {
 cout << " " << w;
 buscaprof(Adj, tam, w, marcado,
 → pilha);
 }
 // proximo vizinho de v
 vert = vert->prox;
 }
 }
 desempilhar(pilha, v);
}

void mostrar_Adj(listaadj Adj[], int tam)
{
 vertice *v;
 for(int i=1; i <= tam; i++)
```

```

 {
 v = Adj[i].listav;
 cout << "\n Entrada " << i << " ";
 while(v != NULL)
 {
 cout << "(" << i << "," << v->num
 << ")" << " ";
 v=v->prox;
 }
 }

void main()
{
 queue *pilha = NULL; // pilha p/armazenar vertices
 // visitados
 int *marcado = NULL; // vetor que armazena se
 // vértice for
 // marcado
 listaadj *Adj = NULL; // lista de adjacencias
 // entre vértices
 vertice *novo, *aux;

 int tam, org, dest, op, num, tipo;

 cout << "\n Tipo do grafo (1- não orientado, 2 -
 orientado): ";
 cin >> tipo;

 cout << "\n Digite número de vértices do grafo:";
 cin >> tam;
 // alocação de memória
 Adj = new listaadj[tam+1];
 marcado = new int [tam+1];
 // inicialização de variáveis
 for(int i = 0; i <= tam; i++)
 {
 marcado[i] = 0;
 Adj[i].listav = NULL;
 }

 cout << "\n Arestas do grafo: VérticeOrigem (-1 para
 parar):";
 cin >> org;
 cout << "\n Arestas do grafo: VérticeDestino(-1 para
 parar):";
}

```

```
cin >> dest;
while(org != -1 && dest != -1)
{
 // alocando um nó com o valor do vértice
 // destino para
 // colocar na entrada
 // do vertice de origem da lista de adjacencias
 // --> (u,v)
 novo = new vertice();
 novo->num = dest;

 // inserindo vértice adjacente a vértice
 // "org" na lista de
 // adjacencias
 novo->prox = Adj[org].listav;
 Adj[org].listav = novo;

 if(tipo==1){
 // inserindo (v,u)
 novo = new vertice();
 novo->num = org;
 // inserindo vértice adjacente a vértice
 // "org" na
 // lista de adjacencias
 novo->prox = Adj[dest].listav;
 Adj[dest].listav = novo;
 }
 // proxima entrada
 cout << "\n Arestas do grafo - VérticeOrigem(-1
 → para parar):";
 cin >> org;
 cout << "\n Arestas do grafo - VérticeDestino
 → (-1 para parar):";
 cin >> dest;
}
do
{
 cout << "\n1-Busca em profundidade"
 << "\n2-Mostrar lista de adjacencias"
 << "\n3-Sair "
 << "\nDigite sua opção: ";
 cin >> op;
 if(op==1)
 {
 cout << "\nDigite um vértice de
 → partida da busca:";
```

```

 cin >> num;
 cout << " " << num;
 buscaprof(Adj, tam, num, marcado,
 & pilha);

 for(int i = 1; i <= tam; i++)
 marcado[i] = 0;
 }
 else if(op==2)
 mostrar_Adj(Adj, tam);
}while (op!=3);

// Desalocando memória
delete marcado;
for(i=1; i <= tam;i++)
{
 while(Adj[i].listav!=NULL)
 {
 aux = Adj[i].listav;
 Adj[i].listav = Adj[i].listav->prox;
 delete aux;
 }
}
delete Adj;
} //fim main

```

---

## Análise da busca em profundidade

A busca em profundidade procura acessar todos os vértices em um grafo  $G = (V, E)$ , onde  $|V| = n$  e  $|E| = m$ . Para acessar todos os possíveis vértices, varre a lista de arestas de cada vértice do grafo e com isso gasta tempo  $O(n + m)$ .

## Algoritmo de busca em largura

Segundo Szwarcfiter *et al.* (1986), uma busca em largura é aquela que atende ao seguinte critério para explorar um vértice marcado: dentre os vários marcados e incidentes a alguma aresta ainda não explorada, escolher aquele vértice alcançado por último na busca.

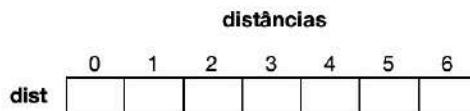
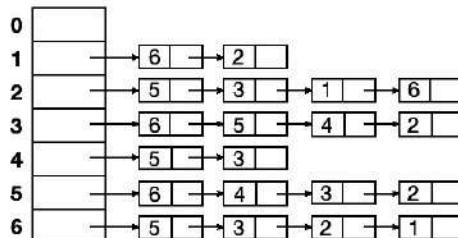
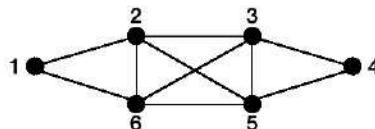
Segundo Cormen (2002), dado um grafo conexo  $G (V, E)$  e um vértice de origem  $s$ , a busca em largura explora as arestas de  $G$  até explorar todos os vértices alcançáveis a partir de  $s$ . Além disso, o algoritmo também calcula a menor distância em número de arestas de  $s$  até todos os vértices acessíveis a ele. A busca em largura recebe esse nome porque descobre todos os vértices que se encontram a uma distância  $k$  de  $s$ , antes de descobrir os vértices que se encontram à distância  $k+1$ .

Nesse algoritmo, todos os vértices são marcados para indicar se já foram alcançados na busca ou não, além de possuírem um valor de distância em relação à origem representado por  $dist[v]$ , onde  $v \in V$ .

Considerando o grafo a seguir e sua lista de adjacências, ilustraremos a busca em largura a partir do vértice 1.

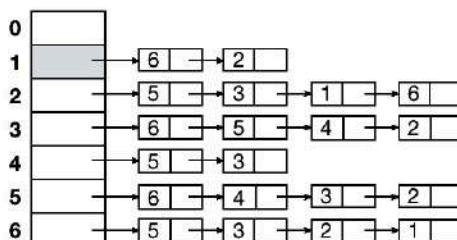


ILUSTRAÇÃO



vértice origem: 1  
marca-se o vértice 1 colocando-o na fila  
 $dist[1] = 0$

fila 1



distâncias

|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|------|---|---|---|---|---|---|---|
| dist |   | 0 |   |   |   |   |   |

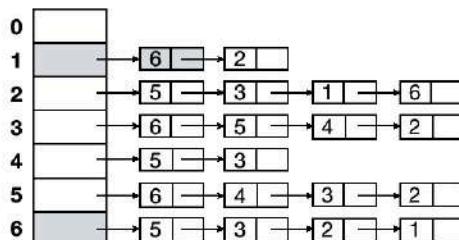
fila vazia? Não  
remove o nº 1

fila

vizinhos de 1: 6, 2

6 está marcado? Não, então marca-se o vértice 6 colocando-o na fila  
 $dist[6] = dist[1] + 1 = 0 + 1 = 1$

fila  6

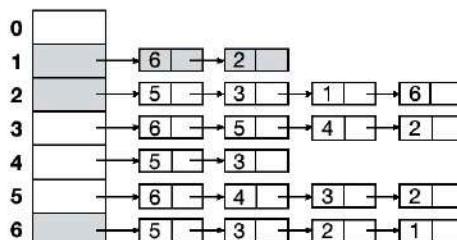


distâncias

|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|------|---|---|---|---|---|---|---|
| dist |   | 0 |   |   |   |   | 1 |

2 está marcado? Não, então marca-se o vértice 2 colocando-o na fila  
 $dist[2] = dist[1] + 1 = 0 + 1 = 1$

fila  6  2



**distâncias**

| dist | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|------|---|---|---|---|---|---|---|
|      |   | 0 | 1 |   |   |   | 1 |

fila vazia? Não  
remove o nº 6

fila 

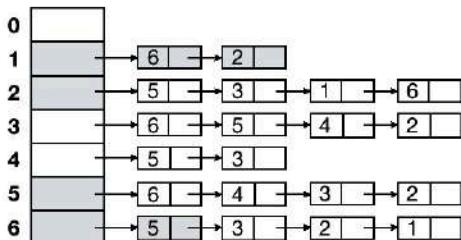
|   |
|---|
| 2 |
|---|

vizinhos de 6: 5, 3, 2, 1

5 está marcado? Não, então marca-se o vértice 5 colocando-o na fila  
 $\text{dist}[5] = \text{dist}[6] + 1 = 1 + 1 = 2$

fila 

|   |   |
|---|---|
| 2 | 5 |
|---|---|

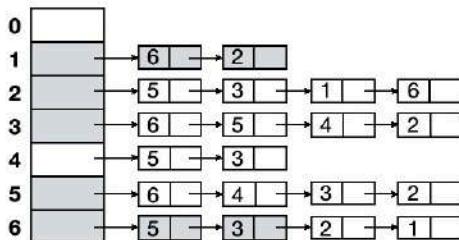
**distâncias**

| dist | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|------|---|---|---|---|---|---|---|
|      |   | 0 | 1 |   |   | 2 | 1 |

3 está marcado? Não, então marca-se o vértice 3 colocando-o na fila  
 $\text{dist}[3] = \text{dist}[6] + 1 = 1 + 1 = 2$

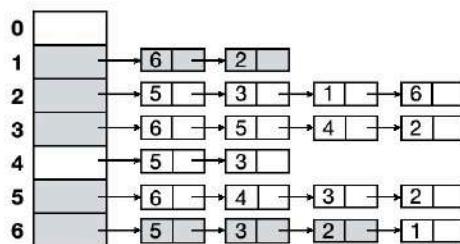
fila 

|   |   |   |
|---|---|---|
| 2 | 5 | 3 |
|---|---|---|

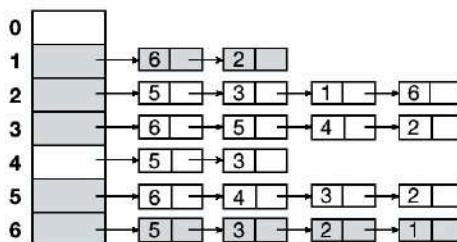
**distâncias**

| dist | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|------|---|---|---|---|---|---|---|
|      |   | 0 | 1 | 2 |   | 2 | 1 |

2 está marcado? Sim, busca-se o próximo vizinho do 6



1 está marcado? Sim, acabaram os vizinhos do 6

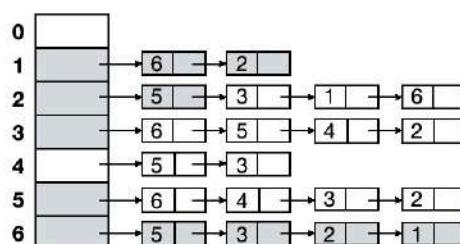


fila vazia? Não  
remove o nº 2

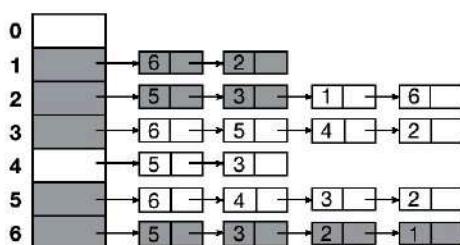
fila [5 | 3]

vizinhos de 2: 5, 3, 1, 6

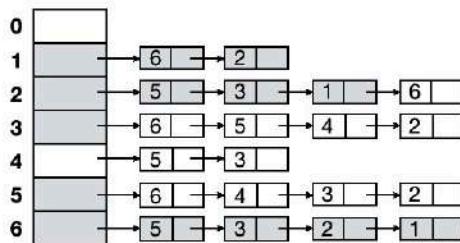
5 está marcado? Sim, busca-se o próximo vizinho do 2



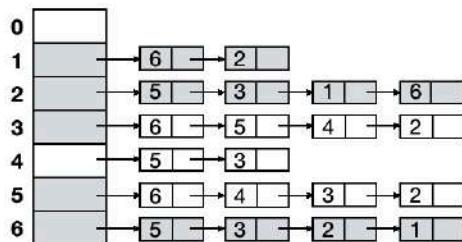
3 está marcado? Sim, busca-se o próximo vizinho do 2



1 está marcado? Sim, busca-se o próximo vizinho do 2



6 está marcado? Sim, acabaram os vizinhos do 2

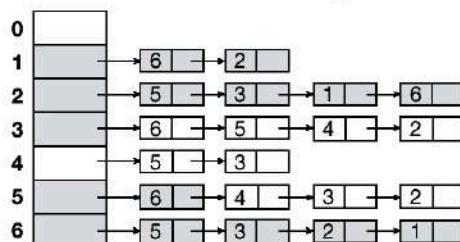


fila vazia? Não  
remove o nº 5

fila [ 3 ]

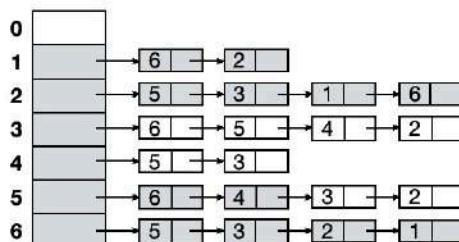
vizinhos de 5: 6, 4, 3, 2

6 está marcado? Sim, busca-se o próximo vizinho do 5



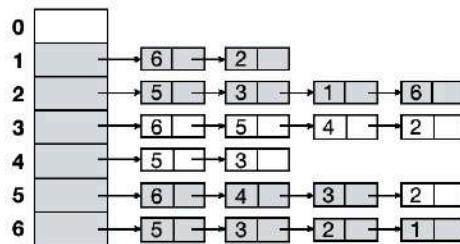
4 está marcado? Não, então marca-se o vértice 4  
colocando-o na fila  $\text{dist}[4] = \text{dist}[5] + 1 = 2 + 1 = 3$

fila [ 3 | 4 ]

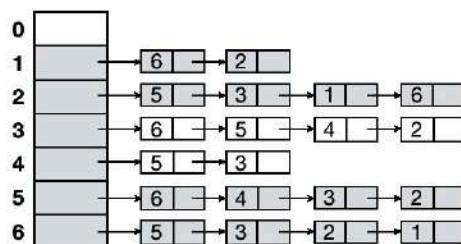


|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|------|---|---|---|---|---|---|---|
| dist |   | 0 | 1 | 2 | 3 | 2 | 1 |

3 está marcado? Sim, busca-se o próximo vizinho do 5



2 está marcado? Sim, acabaram os vizinhos do 5

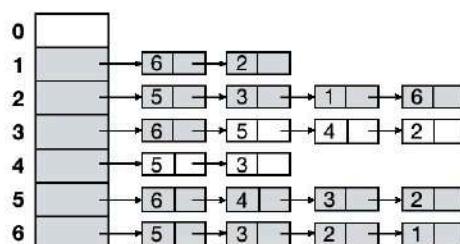


fila vazia? Não  
remove o nº 3

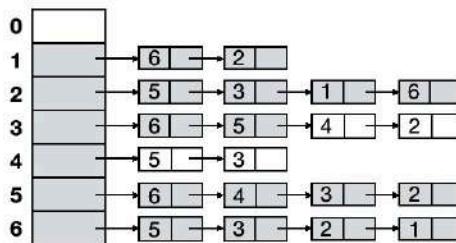
fila 4

vizinhos de 3: 6, 5, 4, 2

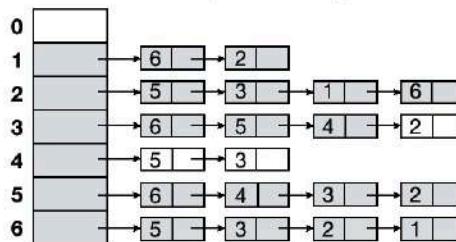
6 está marcado? Sim, busca-se o próximo vizinho do 3



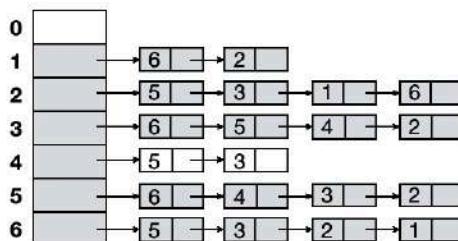
5 está marcado? Sim, busca-se o próximo vizinho do 3



4 está marcado? Sim, busca-se o próximo vizinho do 3



2 está marcado? Sim, acabaram os vizinhos do 3

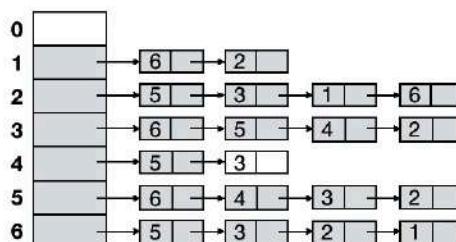


fila vazia? Não  
remove o nº 4

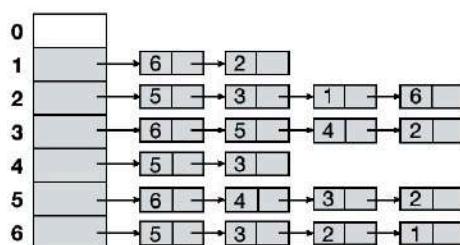
fila

vizinhos de 4: 5, 3

5 está marcado? Sim, busca-se o próximo vizinho do 4



3 está marcado? Sim, acabaram os vizinhos do 4



fila vazia? Sim

**distâncias**

| dist | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|------|---|---|---|---|---|---|---|
|      | 0 | 1 | 2 | 3 | 2 | 2 | 1 |



```

import java.util.Scanner;

public class buscal
{
 public static class vertice
 {
 int num;
 vertice prox;
 }
 public static class listaadj
 {
 public vertice listav;
 }
 public static class fila
 {
 int numv;
 fila prox;
 }
 public static void inserir(int n)
 {
 fila novo = new fila();
 novo.numv = n;
 novo.prox = null;

 if(inicio==null)

```

```
 inicio = fim = novo;
 else
 {
 fim.prox=novo;
 fim = novo;
 }
}
public static int remover()
{
 if(inicio!=null)
 {
 int vert;
 if(inicio==fim)
 fim = null;
 vert = inicio.numv;
 inicio=inicio.prox;
 return vert;
 }
 System.out.println("lista vazia");
 return 0;
}
static fila inicio = null, fim = null;
static Scanner entrada = new Scanner(System.in);
static int marcado[];
static int dist[];
static listaadj Adj[];

public static void main(String args[])
{
 vertice novo;

 int tam, org, dest, op, num=0, tipo, flag=0;
 String menu;

 System.out.println("\n Tipo do grafo (1- não
 → orientado, 2 - orientado:");
 tipo = entrada.nextInt();

 System.out.println("\n Digite número de vértices
 → do grafo:");
 tam = entrada.nextInt();

 Adj = new listaadj[tam+1];
 marcado = new int [tam+1];
 dist = new int [tam+1];

 for(int i = 1; i <= tam; i++)
 {
```

```

 Adj[i]=new listaadj();
 marcado[i] = 0;
 }
 System.out.println("\n Arestas do grafo:
 ↵ VérticeOrigem (-1 para parar):");
 org = entrada.nextInt();
 System.out.println("\n Arestas do grafo:
 ↵ VérticeDestino (-1 para parar):");
 dest = entrada.nextInt();
 while(org != -1 && dest != -1)
 {
 // alocando um nó com o valor do vértice
 // destino para
 // colocar na entrada
 // do vertice de origem da lista de adjacencias
 // --> (u,v)
 novo = new vertice();
 novo.num = dest;
 // inserindo vértice adjacente a vértice org
 // na lista de adjacencias
 novo.prox = Adj[org].listav;
 Adj[org].listav = novo;

 if(tipo==1){
 // inserindo (v,u)
 novo = new vertice();
 novo.num = org;
 // inserindo vértice adjacente a vértice
 // org na lista de adjacencias
 novo.prox = Adj[dest].listav;
 Adj[dest].listav = novo;
 }
 // proxima entrada
 System.out.print("\n Arestas do grafo:
 ↵ VérticeOrigem(-1 para parar):");
 org = entrada.nextInt();
 System.out.print("\n Arestas do grafo:
 ↵ VérticeDestino(-1 para parar):");
 dest = entrada.nextInt();
 }
 do
 {
 menu = "\n1-Busca em largura"+
 "\n2-Mostrar lista de adjacencias"+
 "\n3-Menor distância a partir do
 ↵ vértice de origem"+
 "\n4-Sair "+
 "\nDigite sua opcao: ";

```

```

System.out.print(menu);
op = entrada.nextInt();

switch(op)
{
 case 1: System.out.print("Digite um
 → vértice de origem da busca:");
 num = entrada.nextInt();

 for(int i = 1; i <= tam; i++)
 {
 marcado[i] = 0;
 dist[i] = 0;
 }
 buscalargura(Adj, tam, num);
 flag=1;
 break;
 case 2: mostrar_Adj(Adj, tam);
 break;
 case 3:
 if(flag==0)
 System.out.println("É
 → necessário realizar a
 → busca primeiro");
 else mostrar_dist(tam, num);
 break;
 } //fim switch
} while (op!=4);
} //fim main

static void buscalargura(listaadj Adj[], int tam, int v)
{
 vertice listavert;
 int w;
 int vertice;
 // alcançou vertice v
 marcado[v] = 1;
 // dist de v a origem (ele mesmo) é zero
 dist[v] = 0;
 // inserir v em uma fila
 inserir(v);
 while(inicio!=null)
 {
 // removendo um vértice da fila
 vertice = remover();
 for(int i = 1; i <= tam; i++)
 {
 // varrendo a lista de vizinhos de "vertice"
 listavert = Adj[vertice].listav;

```

```
 while(listavert != null)
 {
 w = listavert.num;
 // se vértice não marcado, calcular
 // a distância
 // em relação a origem e inseri-lo
 // na fila para
 // visitar seus vizinhos depois
 if(marcado[w] == 0)
 {
 marcado[w] = 1;
 dist[w] = dist[vertice]+1;
 inserir(w);
 }
 // próximo vértice adjacente a v
 listavert = listavert.prox;
 }
 }
}

static void mostrar_Adj(listaadj Adj[], int tam)
{
 vertice v;
 for(int i=1; i <= tam; i++)
 {
 v = Adj[i].listav;
 System.out.println("Entrada "+i+" ");
 while(v != null)
 {
 System.out.println("("+i+", "
 + v.num+") " + " ");
 v=v.prox;
 }
 }
}

static void mostrar_dist(int tam, int or)
{
 System.out.println("Distância da origem "+ or
 + " para os demais vértices\n");
 for(int i=1; i <= tam; i++)
 {
 System.out.println(""+i+" - "+dist[i]);
 }
}
```



c/c++

```

#include<iostream.h>
#include<conio.h>

struct vertice
{
 int num;
 vertice *prox;
};

struct listaadj
{
 vertice *listav;
};

struct fila
{
 int numv;
 fila *prox;
};

void inserir(fila* &inicio, fila* &fim, int n)
{
 fila *novo = new fila();
 novo->numv = n;
 novo->prox = NULL;

 if(inicio==NULL)
 inicio = fim = novo;
 else
 {
 fim->prox=novo;
 fim = novo;
 }
}

int remover(fila* &inicio, fila* &fim)
{
 if(inicio!=NULL)
 {
 int vert;
 if(inicio==fim)
 fim = NULL;
 vert = inicio->numv;
 inicio=inicio->prox;
 return vert;
 }
 cout << "\nLista vazia";
 return 0;
}

```

```

void buscalargura(listaadj Adj[], fila* &inicio, fila* &fim,
 int tam,
 int v, int marcado[], int dist[])
{
 vertice *listavert;
 int w;
 int vertice;
 // alcançou vértice v
 marcado[v] = 1;
 // dist de v a origem (ele mesmo) é zero
 dist[v] = 0;
 // inserir v em uma fila
 inserir(inicio, fim, v);
 while(inicio!=NULL)
 {
 // removendo um vértice da fila
 vertice = remover(inicio, fim);
 for(int i = 1; i <= tam; i++)
 {
 // varrendo a lista de vizinhos de "vertice"
 listavert = Adj[vertice].listav;
 while(listavert != NULL)
 {
 w = listavert->num;
 // se vértice não marcado,
 // calcular a distância
 // em relação a origem e
 // inseri-lo na fila para
 // visitar seus vizinhos depois
 if(marcado[w] == 0)
 {
 marcado[w] = 1;
 dist[w] = dist[vertice]+1;
 inserir(inicio, fim, w);
 }
 // próximo vértice adjacente a v
 listavert = listavert->prox;
 }
 }
 }
}

void mostrar_Adj(listaadj Adj[], int tam)
{
 vertice *v;
 for(int i=1; i <= tam; i++)
 {
 v = Adj[i].listav;
 cout << "\nEntrada " << i << " ";
 while(v != NULL)

```

```

 {
 cout << "(" << i << "," << v->num
 << ")" << " ";
 v=v->prox;
 }
}

void mostrar_dist(int dist[], int tam, int or)
{
 cout << "\nDistância da origem " << or << " para
 << os demais " << "vértices\n";
 for(int i=1; i <= tam; i++)
 cout << "\nVertice "<<i<< " - "<<dist[i];
}

void main()
{
 fila *inicio = NULL,
 *fim = NULL,
 *temp;
 int *marcado;
 int *dist;
 listaadj *Adj;

 vertice *novo, *aux;

 int tam, org, dest, op, num=0, tipo, flag=0;

 cout << "\n Tipo do grafo (1 - não orientado, 2 -
 << orientado):";
 cin >> tipo;

 cout << "\n Digite número de vértices do grafo:";
 cin >> tam;

 Adj = new listaadj[tam+1];
 marcado = new int [tam+1];
 dist = new int [tam+1];

 for(int i = 1; i <= tam; i++)
 {
 Adj[i].listav = NULL;
 marcado[i] = 0;
 }
 cout << "\n Arestas do grafo: VérticeOrigem (-1
 << para parar):";
 cin >> org;
 cout << "\n Arestas do grafo: VérticeDestino(-1
 << para parar):";
}

```

```
 cin >> dest;

 while(org != -1 && dest != -1)
 {
 // alocando um nó com o valor do vértice
 // destino para colocar na entrada
 // do vertice de origem da lista de
 // adjacencias --> (u,v)
 novo = new vertice();
 novo->num = dest;
 // inserindo vértice adjacente a vértice org
 // na lista de adjacencias
 novo->prox = Adj[org].listav;
 Adj[org].listav = novo;

 if(tipo==1){
 // inserindo (v,u)
 novo = new vertice();
 novo->num = org;
 // inserindo vértice adjacente a
 // vértice org na lista
 // de adjacencias
 novo->prox = Adj[dest].listav;
 Adj[dest].listav = novo;
 }
 // proxima entrada
 cout << "\n Areistas do grafo:
 ↵ VérticeOrigem(-1 para"
 << " parar):";
 cin >> org;
 cout << "\n Areistas do grafo:
 ↵ VérticeDestino(-1 para "
 << "parar):";
 cin >> dest;
 }
 do
 {
 clrscr();
 cout << "\n1-Busca em largura"
 << "\n2-Mostrar lista de adjacencias"
 << "\n3-Menor distância a partir do
 ↵ vértice de origem"
 << "\n4-Sair "
 << "\nDigite sua opcao: ";
 cin >> op;

 if(op==1)
 {
 cout << "\nDigite um vértice de origem
 ↵ da busca:",
```

```
 cin >> num;

 for(int i = 1; i <= tam; i++)
 {
 marcado[i] = 0;
 dist[i] = 0;
 }
 buscalargura(Adj, inicio, fim, tam,
 num, marcado, dist);
 flag=1;
}
else if (op==2)
 mostrar_Adj(Adj, tam);
else if (op==3)
{
 if(flag==0)
 cout << "\nÉ necessário "
 "realizar a busca "
 << "primeiro";
 else mostrar_dist(dist, tam, num);
}
getch();
} while (op!=4);

// desalocando memoria
delete marcado;
delete dist;
for(i=1; i <= tam;i++)
{
 while(Adj[i].listav!=NULL)
 {
 aux = Adj[i].listav;
 Adj[i].listav = Adj[i].listav->prox;
 delete aux;
 }
}
delete Adj;

// fila
while(inicio!=NULL)
{
 temp = inicio;
 inicio=inicio->prox;
 delete aux;
}

} //fim main
```

## Análise da busca de profundidade

Primeiro, o algoritmo realiza a inicialização de todos os vértices, marcando-os como não visitados. A distância de todos em relação à origem é “infinita”, o que indica que não foi calculada ainda ou não é possível alcançar tal vértice a partir do de origem. Essa inicialização gasta tempo  $O(V)$ .

A medida que os vértices vão sendo descobertos, eles são colocados em uma fila. As operações de inserção e remoção na fila gastam tempo constante,  $O(1)$ , e como todos os vértices são enfileirados e retirados da mesma, o tempo total gasto com estas operações é  $O(V)$ . Quando os vértices são retirados da fila, sua lista de vizinhos é analisada. Como cada vértice só é colocado na fila uma vez, logo sua lista de vizinhos é analisada apenas uma vez e, com isso, todas as listas de todos os vértices são analisadas, gastando tempo proporcional a  $O(E)$ . Portanto, o tempo de execução da busca em largura é  $O(V + E)$ .

## Algoritmo do caminho mínimo

O problema do caminho mínimo a ser considerado nesta seção consiste em determinar o mais curto (com menor peso) em um dígrafo com pesos nas arestas a partir de um vértice de origem informado, sendo os pesos das arestas maiores ou iguais a zero.

O algoritmo para calcular o caminho mínimo a ser apresentado nesta seção é o Algoritmo de Dijkstra. Ele mantém um conjunto  $C$  com vértices cujo valor do caminho mínimo em relação ao de origem  $v$  já foi determinado. Além disso, cada vértice possui uma distância em relação à origem, que é armazenada em um vetor chamado  $dist$ , com tamanho  $n=|V|$ . No vetor  $dist$ , a posição do vértice de origem é inicializada com zero, e as demais posições são inicializadas com “infinito” (na linguagem de programação, um valor bastante grande), para indicar que tal distância ainda não foi calculada ou que não é possível alcançar tal vértice a partir do de origem. O algoritmo também utiliza uma lista de prioridades mínima chamada  $lista$ , cujos valores das chaves são as distâncias (valores de  $dist$ ). A lista é inicializada com todos os vértices e suas distâncias.

O algoritmo trabalha de maneira a escolher um vértice  $i$  pertencente à lista de prioridades mínima  $lista = V - C$ , tal que a estimativa de distância de  $i$  à origem é mínima. Em seguida, os vértices que são vizinhos a  $i$  têm sua distância recalculada, se for o caso. Para recalcular essas distâncias, as arestas passam por um processo de relaxamento.

Segundo Cormen (2002), relaxar uma aresta  $(i,j)$  significa testar se é possível melhorar o menor caminho encontrado até agora para  $j$  pela passagem através do vértice  $i$  e, com isso, atualizar a distância mínima que  $j$  possui. O trecho abaixo realiza o relaxamento de uma aresta  $(i,j)$ , considerando que  $peso$  retorna o valor do peso da aresta  $(i,j)$ .




---

```

if (dist[j] > dist[i] + peso(i,j))
{
 dist[j] ← dist[i] + peso(i,j)
}

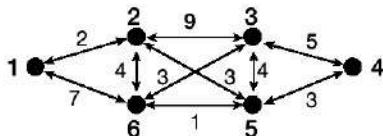
```

---

Considerando o grafo a seguir e sua lista de adjacências, ilustraremos a busca de um caminho mínimo partindo do vértice 1. Neste exemplo o peso da aresta é mostrado entre parênteses.



ILUSTRAÇÃO



|   |  |      |      |      |      |  |  |
|---|--|------|------|------|------|--|--|
| 0 |  |      |      |      |      |  |  |
| 1 |  | 6(7) | 2(2) |      |      |  |  |
| 2 |  | 6(4) | 5(3) | 3(9) | 1(2) |  |  |
| 3 |  | 6(3) | 5(4) | 4(5) | 2(9) |  |  |
| 4 |  | 5(3) | 3(5) |      |      |  |  |
| 5 |  | 6(1) | 4(3) | 3(4) | 2(3) |  |  |
| 6 |  | 5(1) | 3(3) | 2(4) | 1(7) |  |  |

vértice origem: 1

$\text{dist}[1] = 0$

Lista de prioridades: 1 2 3 4 5 6

O tamanho da lista é diferente de 0? Sim

Então remove da lista o vértice com menor distância

$w = 1$

vizinhos de 1: 6, 2

|      | distâncias |   |          |          |          |          |          |
|------|------------|---|----------|----------|----------|----------|----------|
| dist | 0          | 1 | 2        | 3        | 4        | 5        | 6        |
|      |            | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |

Lista de prioridades: 2 3 4 5 6

$x = 6$

$\text{dist}[x] > \text{dist}[w] + \text{peso}(w, x)$

$\text{dist}[6] > \text{dist}[1] + \text{peso}(1, 6)$

$\infty > 0 + 7$

V

$\text{dist}[x] = \text{dist}[w] + \text{peso}(w, x)$

$\text{dist}[6] = \text{dist}[1] + \text{peso}(1, 6)$

$\text{dist}[6] = 0 + 7 = 7$

|      | distâncias |   |          |          |          |          |   |
|------|------------|---|----------|----------|----------|----------|---|
| dist | 0          | 1 | 2        | 3        | 4        | 5        | 6 |
|      |            | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 7 |

$x = 2$

$\text{dist}[x] > \text{dist}[w] + \text{peso}(w, x)$

$\text{dist}[2] > \text{dist}[1] + \text{peso}(1, 2)$

$\infty > 0 + 2$

V

$\text{dist}[x] = \text{dist}[w] + \text{peso}(w, x)$

$\text{dist}[2] = \text{dist}[1] + \text{peso}(1, 2)$

$\text{dist}[2] = 0 + 2 = 2$

|      | distâncias |   |   |          |          |          |   |
|------|------------|---|---|----------|----------|----------|---|
| dist | 0          | 1 | 2 | 3        | 4        | 5        | 6 |
|      |            | 0 | 2 | $\infty$ | $\infty$ | $\infty$ | 7 |

O tamanho da lista é diferente de 0? Sim

Então é removido da lista o vértice com menor distância

$w = 2$

vizinhos de 2: 6, 5, 3, 1



ILUSTRAÇÃO

Lista de prioridades: 3 4 5 6

 $x = 6$  $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[6] > \text{dist}[2] + \text{peso } (2, 6)$ 

$$7 > 2 + 4$$

V

 $\text{dist}[x] = \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[6] = \text{dist}[2] + \text{peso } (2, 6)$   
 $\text{dist}[6] = 2 + 4 = 6$ 

|      | distâncias |   |   |   |   |   |   |
|------|------------|---|---|---|---|---|---|
| dist | 0          | 1 | 2 | 3 | 4 | 5 | 6 |
|      | 0          |   | 2 | ∞ | ∞ | ∞ | 6 |

 $x = 5$  $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[5] > \text{dist}[2] + \text{peso } (2, 5)$ 

$$\infty > 2 + 3$$

V

 $\text{dist}[x] = \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[5] = \text{dist}[2] + \text{peso } (2, 5)$   
 $\text{dist}[5] = 2 + 3 = 5$ 

|      | distâncias |   |   |   |   |   |   |
|------|------------|---|---|---|---|---|---|
| dist | 0          | 1 | 2 | 3 | 4 | 5 | 6 |
|      | 0          |   | 2 | ∞ | ∞ | 5 | 6 |

 $x = 3$  $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[3] > \text{dist}[2] + \text{peso } (2, 3)$ 

$$\infty > 2 + 9$$

V

 $\text{dist}[x] = \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[3] = \text{dist}[2] + \text{peso } (2, 3)$   
 $\text{dist}[3] = 2 + 9 = 11$ 

|      | distâncias |   |   |    |   |   |   |
|------|------------|---|---|----|---|---|---|
| dist | 0          | 1 | 2 | 3  | 4 | 5 | 6 |
|      | 0          |   | 2 | 11 | ∞ | 5 | 6 |

 $x = 1$  $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[1] > \text{dist}[2] + \text{peso } (2, 1)$ 

$$0 > 2 + 2$$

F

|      | distâncias |   |   |    |   |   |   |
|------|------------|---|---|----|---|---|---|
| dist | 0          | 1 | 2 | 3  | 4 | 5 | 6 |
|      | 0          |   | 2 | 11 | ∞ | 5 | 6 |

O tamanho da lista é diferente de 0? Sim

Então é removido da lista o vértice com menor distância

 $w = 5$ 

vizinhos de 5: 6, 4, 3, 2



ILUSTRAÇÃO

Lista de prioridades: 3 4 6

 $x = 6$  $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[6] > \text{dist}[5] + \text{peso } (5, 6)$ 

$$6 > 5 + 1$$

F

|      | distâncias |   |   |    |   |   |   |
|------|------------|---|---|----|---|---|---|
| dist | 0          | 1 | 2 | 3  | 4 | 5 | 6 |
|      | 0          |   | 2 | 11 | ∞ | 5 | 6 |

$x = 4$   
 $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[4] > \text{dist}[5] + \text{peso } (5, 4)$   
 $\infty > 5 + 3$   
 $V$

$\text{dist}[x] = \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[4] = \text{dist}[5] + \text{peso } (5, 4)$   
 $\text{dist}[4] = 5 + 3 = 8$

$x = 3$   
 $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[3] > \text{dist}[5] + \text{peso } (5, 3)$   
 $11 > 5 + 4$   
 $V$

$\text{dist}[x] = \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[3] = \text{dist}[5] + \text{peso } (5, 3)$   
 $\text{dist}[3] = 5 + 4 = 9$

$x = 2$   
 $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[2] > \text{dist}[5] + \text{peso } (5, 2)$   
 $2 > 5 + 3$   
 $F$

| distâncias |   |   |   |    |   |   |   |
|------------|---|---|---|----|---|---|---|
| dist       | 0 | 1 | 2 | 3  | 4 | 5 | 6 |
|            |   | 0 | 2 | 11 | 8 | 5 | 6 |

| distâncias |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|
| dist       | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|            |   | 0 | 2 | 9 | 8 | 5 | 6 |

| distâncias |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|
| dist       | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|            |   | 0 | 2 | 9 | 8 | 5 | 6 |

O tamanho da lista é diferente de 0? Sim

Então é removido da lista o vértice com menor distância

$w = 6$

vizinhos de 6: 5, 3, 2, 1



ILUSTRAÇÃO

Lista de prioridades: 3 4

$x = 5$   
 $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[5] > \text{dist}[6] + \text{peso } (6, 5)$   
 $5 > 6 + 1$   
 $F$

$x = 3$   
 $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[3] > \text{dist}[6] + \text{peso } (6, 3)$   
 $9 > 6 + 3$   
 $F$

$x = 2$   
 $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[2] > \text{dist}[6] + \text{peso } (6, 2)$   
 $2 > 6 + 4$   
 $F$

$x = 1$   
 $\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$   
 $\text{dist}[1] > \text{dist}[6] + \text{peso } (6, 1)$   
 $0 > 6 + 7$   
 $F$

| distâncias |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|
| dist       | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|            |   | 0 | 2 | 9 | 8 | 5 | 6 |

| distâncias |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|
| dist       | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|            |   | 0 | 2 | 9 | 8 | 5 | 6 |

| distâncias |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|
| dist       | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|            |   | 0 | 2 | 9 | 8 | 5 | 6 |

| distâncias |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|
| dist       | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|            |   | 0 | 2 | 9 | 8 | 5 | 6 |

## 412 Estruturas de dados

O tamanho da lista é diferente de 0? Sim

Então é removido da lista o vértice com menor distância

$w = 4$

vizinhos de 4: 5, 3



ILUSTRAÇÃO

Lista de prioridades: 3

$x = 5$

$\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$

$\text{dist}[5] > \text{dist}[4] + \text{peso } (4, 5)$

$$5 > 8 + 3$$

F

| dist | distâncias |   |   |   |   |   |   |
|------|------------|---|---|---|---|---|---|
|      | 0          | 1 | 2 | 3 | 4 | 5 | 6 |
|      | 0          | 2 | 9 | 8 | 5 | 6 |   |

$x = 3$

$\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$

$\text{dist}[3] > \text{dist}[4] + \text{peso } (4, 3)$

$$9 > 8 + 5$$

F

| dist | distâncias |   |   |   |   |   |   |
|------|------------|---|---|---|---|---|---|
|      | 0          | 1 | 2 | 3 | 4 | 5 | 6 |
|      | 0          | 2 | 9 | 8 | 5 | 6 |   |

O tamanho da lista é diferente de 0? Sim

Então é removido da lista o vértice com menor distância

$w = 3$

vizinhos de 3: 6, 5, 4, 2



ILUSTRAÇÃO

Lista de prioridades: vazia

$x = 6$

$\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$

$\text{dist}[6] > \text{dist}[3] + \text{peso } (3, 6)$

$$6 > 9 + 3$$

F

| dist | distâncias |   |   |   |   |   |   |
|------|------------|---|---|---|---|---|---|
|      | 0          | 1 | 2 | 3 | 4 | 5 | 6 |
|      | 0          | 2 | 9 | 8 | 5 | 6 |   |

$x = 5$

$\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$

$\text{dist}[5] > \text{dist}[3] + \text{peso } (3, 5)$

$$5 > 9 + 4$$

F

| dist | distâncias |   |   |   |   |   |   |
|------|------------|---|---|---|---|---|---|
|      | 0          | 1 | 2 | 3 | 4 | 5 | 6 |
|      | 0          | 2 | 9 | 8 | 5 | 6 |   |

$x = 4$

$\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$

$\text{dist}[4] > \text{dist}[3] + \text{peso } (3, 4)$

$$8 > 9 + 5$$

F

| dist | distâncias |   |   |   |   |   |   |
|------|------------|---|---|---|---|---|---|
|      | 0          | 1 | 2 | 3 | 4 | 5 | 6 |
|      | 0          | 2 | 9 | 8 | 5 | 6 |   |

$x = 2$

$\text{dist}[x] > \text{dist}[w] + \text{peso } (w, x)$

$\text{dist}[2] > \text{dist}[3] + \text{peso } (3, 2)$

$$2 > 9 + 9$$

F

| dist | distâncias |   |   |   |   |   |   |
|------|------------|---|---|---|---|---|---|
|      | 0          | 1 | 2 | 3 | 4 | 5 | 6 |
|      | 0          | 2 | 9 | 8 | 5 | 6 |   |

O tamanho da lista é diferente de 0? Não

Na implementação desse algoritmo, foi necessária a criação de uma lista de prioridades, cujo código também é apresentado nesta seção. Em seguida, o código do programa implementando o algoritmo de Dijkstra é apresentado.



```

import java.util.Scanner;

public class ListaPrior
{
 static int vet[];
 static Scanner entrada = new Scanner(System.in);
 static int tam;
 public ListaPrior(int n)
 {
 vet = new int [n+1];
 tam = 0;
 }
 public static void inserir(int num, int dist[])
 {
 int ind;
 if(tam < vet.length-1)
 {
 tam++;
 ind=tam;
 while(ind>1 && dist[vet[Pai(ind)]] >
 dist[num])
 {
 vet[ind]=vet[Pai(ind)];
 ind=Pai(ind);
 }
 vet[ind]=num;
 }
 }
 public static int Pai (int x)
 {
 return x/2;
 }
 public static void heap_fica(int i, int qtde, int dist[])
 {
 int f_esq, f_dir, menor, aux;
 menor = i;
 if (2*i+1 <= qtde)
 {
 // o nó que está sendo analisado tem
 // filhos p/ esquerda e direita
 f_esq = 2*i;
 f_dir = 2*i+1;
 if (dist[vet[f_esq]] < dist[vet[f_dir]] &&

```

```

 dist[vet[f_esq]] < dist[vet[i]])

 menor = 2*i;

 else if (dist[vet[f_dir]] <

 - dist[vet[f_esq]] &&

 dist[vet[f_dir]] < dist[vet[i]])

 menor = 2*i+1;

 }

else if (2*i <= qtde)

{

 // o nó que está sendo analisado tem

 // filho apenas p/ a esquerda

 f_esq = 2*i;

 if (dist[vet[f_esq]] < dist[vet[i]])

 menor = 2*i;

}

if (menor != i)

{

 aux = vet[i];

 vet[i] = vet[menor];

 vet[menor] = aux;

 heap_fica(menor,qtde, dist);

}

}

public static void constroiheap(int dist[])
{
 for(int i = tam/2; i >=1; i--)
 heap_fica(i,tam, dist);
}
public static int remover(int dist[])
{
 if(tam==0) System.out.println("Lista vazia!");
 else
 {
 int menor_prior=vet[1];
 vet[1]=vet[tam];
 tam--;
 heap_fica(1, tam, dist);

 return menor_prior;
 }
 return 0;
}
public static void imprimir()
{
 for(int i=1; i <= tam; i++)
 System.out.println(" "+vet[i]);
}
public class CaminhoMinimo
}

```

```
{
 public static class vertice
 {
 int num;
 int peso;
 vertice prox;
 }
 public static class listaadj
 {
 public vertice listav;
 }
 static ListaPrior lista; // lista de prioridades
 static Scanner entrada = new Scanner(System.in);
 static int marcado[]; // vetor p/ marcar
 // visitação dos vértices
 static int pai[]; // antecessor dos vértices no
 // caminho mínimo
 static int dist[]; // distância mínima em
 // relação à origem
 static listaadj Adj[]; // lista de adjacencias dos
 // vértices

 public static void main(String args[])
 {
 vertice novo;

 int tam, // número de vértices do grafo
 org, // vertice de origem na aresta
 dest, // vertice destino na aresta
 op, // opção do menu
 num=0, // vértice de origem do caminho minimo
 flag=0, // variavel para validação do menu
 peso=0; // peso da aresta do grafo
 String menu; // texto do menu

 System.out.println("\n Digite número de vértices
 do grafo orientado:");
 tam = entrada.nextInt();

 Adj = new listaadj[tam+1];
 marcado = new int [tam+1];
 pai = new int[tam+1];
 dist = new int [tam+1];

 for(int i = 1; i <= tam; i++)
 {
 Adj[i]=new listaadj();
 marcado[i] = 0;
```

```
 }
 System.out.println("\n Arestas do grafo:
 ↪ VérticeOrigem (-1 para
 parar):");
 org = entrada.nextInt();
 System.out.println("\n Arestas do grafo:
 ↪ VérticeDestino(-1 para
 parar):");
 dest = entrada.nextInt();

 while(org != -1 && dest != -1)
 {
 System.out.println("\n Peso da aresta:");
 peso = entrada.nextInt();
 // alocando um nó com o valor do vértice
 // destino para
 // colocar na entrada
 // do vertice de origem da lista de
 // adjacencias --> (u,v)
 novo = new vertice();
 novo.num = dest;
 novo.peso=peso;
 // inserindo vértice adjacente a vértice org
 // na lista de adjacencias
 novo.prox = Adj[org].listav;
 Adj[org].listav = novo;

 // proxima entrada
 System.out.print("\n Arestas do grafo:
 ↪ VérticeOrigem(-1
 para parar):");
 org = entrada.nextInt();
 System.out.print("\n Arestas do grafo:
 ↪ VérticeDestino(-1
 para parar):");
 dest = entrada.nextInt();
 }
 do
 {
 menu = "\n1-Caminho Mínimo"+
 "\n2-Mostrar lista de adjacencias"+
 "\n3-Mostrar distâncias"+
 "\n4-Sair "+
 "\nDigite sua opção: ";
 System.out.print(menu);
 op = entrada.nextInt();
 switch(op)
 {
```

```

case 1: System.out.print
 → ("Digite um vértice de
 origem:");
 num = entrada.nextInt();

 for(int i = 1; i <= tam; i++)
 {
 marcado[i] = 0;
 dist[i] = 0;
 }
 dijkstra(Adj, tam, num);
 flag=1;
 break;
case 2: mostrar_Adj(Adj, tam);
 break;
case 3:
 if(flag==0)
 System.out.println
 → ("É necessário
 → realizar
 a busca primeiro");
 else mostrar_dist(tam, num);
 break;
} //fim switch
} while (op!=4);
} //fim main

static void dijkstra(listaadj Adj[], int tam, int v)
{
 int i, w;
 int C[] = new int[tam];
 int tamC = 0;
 lista = new ListaPrior(tam);

 dist[v]=0;
 lista.inserir(v, dist);
 for(i=1; i <= tam; i++)
 {
 if(i!=v)
 {
 dist[i]=Integer.MAX_VALUE;
 pai[i]=0;
 lista.inserir(i, dist);
 }
 }
 while(lista.tam != 0)
 {
 w = lista.remover(dist);
 C[tamC] = w;

```

```
 tamC++;

 vertice x = Adj[w].listav;
 while(x!=null)
 {
 // relax (w,x, peso_wx)
 relax(w,x.num,x.peso);
 // proximo vizinho de w
 x = x.prox;
 }
 lista.constroiheap(dist);
}

static void relax(int u, int v, int peso)
{
 if(dist[v] > dist[u]+peso)
 {
 dist[v] = dist[u] + peso;
 pai[v] = u;
 }
}
static void mostrar_Adj(listaadj Adj[], int tam)
{
 vertice v;
 for(int i=1; i <= tam; i++)
 {
 v = Adj[i].listav;
 System.out.println("Entrada "+i+" ");
 while(v != null)
 {
 System.out.println("(" + i + ", "
 + v.num + ")" + " ");
 v=v.prox;
 }
 }
}
static void mostrar_dist(int tam, int or)
{
 System.out.println("Distância da origem "+ or +
 " para os demais vértices\n");
 for(int i=1; i <= tam; i++)
 {
 System.out.println(" "+i+" - "+dist[i]);
 }
}
```



```

#include<iostream.h>
#include<conio.h>
#include<values.h>

struct ListaPrior {
 int *vet;
 int tam;
};

struct vertice
{
 int num;
 int peso;
 vertice *prox;
};

struct listaadj
{
 vertice *listav;
};

int Pai (int x)
{ return x/2; }

void inserirListaPrior(ListaPrior &lista, int num, int
 *dist[], int tam)
{
 int ind;
 if(lista.tam < tam-1)
 {
 lista.tam++;
 ind=lista.tam;
 while(ind>1 && dist[lista.vet[Pai(ind)]]>
 * dist[num])
 {
 lista.vet[ind]=lista.vet[Pai(ind)];
 ind=Pai(ind);
 }
 lista.vet[ind]=num;
 }
}

void heap_fica(ListaPrior &lista, int i, int qtde, int dist[])
{
 int f_esq, f_dir, menor, aux;
 menor = i;
 if (2*i+1 <= qtde)

```

```

{
 // o nó que está sendo analisado tem filhos p/
 // esquerda e direita
 f_esq = 2*i;
 f_dir = 2*i+1;
 if [dist[lista.vet[f_esq]] <
 dist[lista.vet[f_dir]] &&
 dist[lista.vet[f_esq]] <
 dist[lista.vet[i]])]
 menor = 2*i;
 else if (dist[lista.vet[f_dir]] &&
 < dist[lista.vet[f_esq]] &&
 dist[lista.vet[f_dir]] &&
 < dist[lista.vet[i]])]
 menor = 2*i+1;
 }
 else if (2*i <= qtde)
 {
 // o nó que está sendo analisado tem filho
 // apenas p/ a esquerda
 f_esq = 2*i;
 if (dist[lista.vet[f_esq]] < dist[lista.vet[i]])
 menor = 2*i;
 }
 if (menor != i)
 {
 aux = lista.vet[i];
 lista.vet[i] = lista.vet[menor];
 lista.vet[menor] = aux;
 heap_fica(lista, menor, qtde, dist);
 }
}

void constroiheap(ListaPrior &lista, int dist[])
{
 for(int i = lista.tam/2; i >=1; i--)
 heap_fica(lista, i, lista.tam, dist);
}

int remover(ListaPrior &lista, int dist[])
{
 if(lista.tam==0) cout << "\nLista vazia!";
 else
 {
 int menor_prior=lista.vet[1];
 lista.vet[1]=lista.vet[lista.tam];
 lista.tam--;
 heap_fica(lista, 1, lista.tam, dist);
 }
}

```

```
 return menor_prior;
 }
 return 0;
}

void imprimir(ListaPrior lista)
{
 for(int i=1; i <= lista.tam; i++)
 cout << " " << lista.vet[i];
}

void relax(int u, int v, int peso, int pai[], int dist[])
{
 if(dist[v] > dist[u]+peso)
 {
 dist[v] = dist[u] + peso;
 pai[v] = u;
 }
}

void dijkstra(listadj Adj[], int tam, int v, int pai[],
 int dist[])
{
 int i, w;
 int *C = new int[tam];
 int tamC = 0;
 ListaPrior lista; // lista de prioridades

 lista.vet = new int [tam+1];
 lista.tam = 0;

 dist[v]=0;
 inserirListaPrior(lista, v, dist, tam);
 for(i=1; i <= tam; i++)
 {
 if(i!=v)
 {
 dist[i]=MAXINT;
 pai[i]=0;
 inserirListaPrior(lista, i, dist, tam);
 }
 }
 while(lista.tam != 0)
 {
 w = remover(lista, dist);
 C[tamC] = w;
```

```

 tamC++;

 vertice *x = Adj[w].listav;
 while(x!=NULL)
 {
 // relax (w,x, peso_wx)
 relax(w,x->num,x->peso, pai, dist);
 // proximo vizinho de w
 x = x->prox;
 }
 constroiheap(lista, dist);
 }
}

void mostrar_Adj(listaadj Adj[], int tam)
{
 vertice *v;
 for(int i=1; i <= tam; i++)
 {
 v = Adj[i].listav;
 cout << "\nEntrada " << i << " ";
 while(v != NULL)
 {
 cout << "(" << i << ", " << v->num
 << ")" << " ";
 v=v->prox;
 }
 }
}

void mostrar_dist(int tam, int or, int dist[])
{
 cout << "\nDistância da origem " << or
 << " para os demais vértices\n";
 for(int i=1; i <= tam; i++)
 cout << "\nVertice " << i << " - " << dist[i];
}

void main()
{
 int *marcado; // vetor p/ marcar visitação dos
 // vértices
 int *pai; // antecessor dos vértices no
 // caminho mínimo
 int *dist; // distância mínima em relação à
 // origem
 listaadj *Adj; // lista de adjacencias dos vértices
}

```

```
vertice *novo;

int tam, // número de vértices do grafo
 org, // vertice de origem na aresta
 dest, // vertice destino na aresta
 op, // opção do menu
 num=0, // vértice de origem do caminho minimo
 flag=0, // variavel para validação do menu
 peso=0; // peso da aresta do grafo

cout << "\n Digite número de vértices do grafo
 ↪ orientado:";
cin >> tam;

Adj = new listaadj[tam+1];
marcado = new int [tam+1];
pai = new int[tam+1];
dist = new int [tam+1];

for(int i = 1; i <= tam; i++)
{
 Adj[i].listav = NULL;
 marcado[i] = 0;
}
cout << "\n Arestas do grafo: VérticeOrigem
 ↪ (-1 para parar):";
cin >> org;
cout << "\n Arestas do grafo: VérticeDestino
 ↪ (-1 para parar):";
cin >> dest;

while(org != -1 && dest != -1)
{
 cout << "\n Peso da aresta:";
 cin >> peso;
 // alocando um nó com o valor do vértice
 // destino para
 // colocar na entrada
 // do vertice de origem da lista de
 // adjacencias --> (u,v)
 novo = new vertice();
 novo->num = dest;
 novo->peso=peso;
 // inserindo vértice adjacente a vértice org
 // na lista de adjacencias
 novo->prox = Adj[org].listav;
 Adj[org].listav = novo;

 // proxima entrada
 cout << "\n Arestas do grafo: VérticeOrigem
```

```

 ↵ (-1 para " << "parar):";
 cin >> org;
 cout << "\n Arestas do grafo: VérticeDestino
 ↵ (-1 para " << "parar):";
 cin >> dest;
}
do
{
 clrscr();
 cout << "\n1-Caminho Mínimo"
 << "\n2-Mostrar lista de adjacencias"
 << "\n3-Mostrar distâncias"
 << "\n4-Sair "
 << "\nDigite sua opção: ";
 cin >> op;

 if(op==1)
 {
 cout << "\nDigite um vértice de origem:";
 cin >> num;

 for(int i = 1; i <= tam; i++)
 {
 marcado[i] = 0;
 dist[i] = 0;
 }
 dijkstra(Adj, tam, num, pai, dist);
 flag=1;
 }
 else if(op==2)
 {
 mostrar_Adj(Adj, tam);
 }
 else if(op==3)
 {
 if(flag==0)
 cout << "\nÉ necessário
 ↵ realizar a busca"
 << " primeiro";
 else mostrar_dist(tam, num, dist);
 } //fim
 getch();
} while (op!=4);
} //fim main

```

---

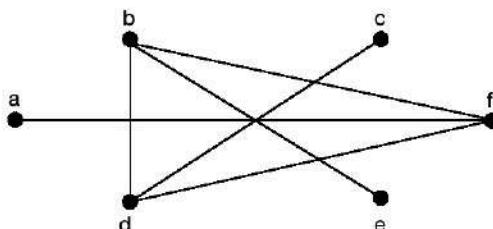
## Análise do algoritmo de caminho mínimo

O tempo de execução do algoritmo Dijkstra será determinado pela forma como a lista de prioridades mínima é implementada. Neste caso, a lista foi implementada como um *heap* binário mínimo, logo, cada operação de inserção e remoção custa no pior caso  $O(\log V)$ , onde  $V$  representa o número de vértices do grafo.

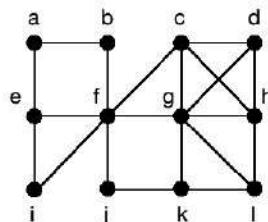
O algoritmo é executado enquanto a lista de prioridade não for vazia. Como a lista foi inicializada com as distâncias de todos os vértices, o algoritmo executará  $|V|$  vezes. Em cada uma dessas operações, ele remove da lista ( $O(\log V)$ ) o vértice que possui a menor distância em relação à origem, analisando as arestas adjacentes a esse vértice ( $O(|V|)$ ) e recalculando os valores das distâncias. Em seguida, com a alteração das distâncias, a lista de prioridades mínima é reconstruída para remoção do próximo vértice com menor distância. Considerando que o custo para se construir um *heap* binário é da ordem  $O(V)$ , o tempo de execução do algoritmo é proporcional a  $V \cdot (\log V + E + V) = V \cdot \log V + V \cdot E + V \cdot V = O(V^2)$ .

## Exercícios

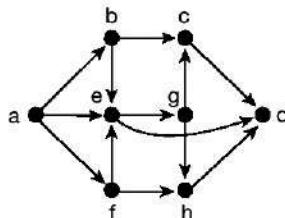
- $\bar{G}$  é o grafo complemento de um grafo  $G(V, E)$ , que possui o mesmo conjunto de vértices  $V$  de  $G$ . O conjunto de arestas é constituído por todo par de vértices  $v, w$  distintos pertencentes a  $V$ , que não é aresta em  $G$ .
  - Determine o grafo complemento do grafo a seguir.
  - A partir dos dados de um grafo  $G$  lido em uma Matriz de Adjacências, escreva um algoritmo para listar as arestas de um grafo complementar de  $G$ .



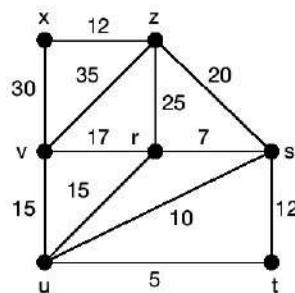
- Considerando o grafo a seguir, execute a busca em largura partindo do vértice b.



3. Considerando o grafo a seguir, execute a busca em profundidade partindo do vértice  $a$ .



4. Execute o algoritmo do caminho mínimo para o grafo abaixo e determine o custo do menor caminho entre os vértices  $x$  e  $t$ .



5. Utilizando o grafo do exercício 2, construa uma árvore geradora para cada item abaixo, considerando:
- Partida da construção da árvore pelo vértice  $i$  e realizando a escolha dos demais vértices como se executasse a busca em largura. Ao visitar os vizinhos de um nó, siga a ordem alfabética.
  - Partida da construção da árvore pelo vértice  $a$  e realizando a escolha dos demais vértices como se executasse a busca em profundidade. Ao visitar os vizinhos de um nó, siga a ordem alfabética.

# Referências

- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Campus, 2002.
- LOUDON, K. *Dominando algoritmos com C*. [il]. Rio de Janeiro: Ciência Moderna, 2000.
- OLIVEIRA, J. B. *Somatórios e recorrências*. Disponível em:  
<http://www.inf.pucrs.br/~oliveira/complex/>. Acesso em: 15 set. 2008.
- PEREIRA, S. L. *Estrutura de dados fundamentais: conceitos e aplicações*. São Paulo: Érica, 1996.
- REZENDE, P. J. "Relações de recorrência". Disponível em:  
<http://www.dct.ufms.br/~edson/complexidade/recorrencias.pdf>. Acesso em: 31 jul. 2010.
- SZWARCFITER, J. L.; MARKENZON, L. *Estruturas de dados e seus algoritmos*. Rio de Janeiro: LTC, 1996.
- ZIVIANI, N. *Projeto de algoritmos: com implementações em Pascal e C*. 2. ed. São Paulo: Pioneira Thomson Learning, 2004.



# Índice remissivo

## A

algoritmo, 1-20

análise de, 2-5

conceito, 1

de busca, 378

de busca binária, 98-102

de busca em largura, 390

de busca em profundidade, 378

de busca sequencial, 90-97

de ordenação (*heap sort*), 74-89

de ordenação e busca, 21-104

de ordenação por inserção (*INSERTION SORT*), 38-43

de ordenação por intercalação (*MERGE SORT*), 53-59

de ordenação por seleção (*SELECTION SORT*), 44-50

de ordenação por troca (*BUBBLE SORT*), 21-26

de ordenação rápida (*QUICK SORT*), 61-72

do caminho mínimo, 408

em grafos, 368

tempo de execução de um, 2

Análise

de um algoritmo particular, 2

de uma classe de algoritmos, 2

Análise da complexidade, 26-27, 32, 37, 43, 51, 59, 72, 89, 97, 102, 180, 191, 197, 328, 366

Análise da complexidade da tabela *hashing*

com tentativa linear, 256

com tentativa quadrática, 267

implementada com lista, 276

Análise da complexidade das estruturas

*heap max-min* e *heap min-max*, 242

*heap* máximo e *heap* mínimo, 215

aresta de corte, 374

arestas múltiplas, 369

árvore(s)

cheia, 284

completa, 284

estritamente binária, 283

geradora, 373

árvore AVL (Adelson-Velsky e Landis), 329

árvore binária, 278-328

com recursividade, 315-329

sem recursividade, 291-315

## B

*BUBBLE SORT* melhorado (1<sup>a</sup> versão), 27-33

*BUBBLE SORT* melhorado (2<sup>a</sup> versão), 33

busca sequencial, 4, 20, 90, 97

## C

caminho

elementar, 371

simples, 371

caso médio, 4

ciclos, 371

componentes, 371  
 conjuntos dinâmicos, 105  
 consultar toda a lista, 180  
 consultas em uma árvore binária, 285  
 corte de arestas, 374  
 corte de vértices, 374

**D**

descrição de rotações, 329  
 dígrafos, 372-373

**E**

elementos da análise assintótica, 5-6  
 estrutura de dados, 1-2  
 do tipo árvore, 278-367  
   do tipo fila, 184  
   do tipo lista de prioridades, 201  
   do tipo listas, 105-182  
   do tipo pilha, 184  
   do tipo tabela *hashing*, 244  
 esvaziar a lista, 181  
 exemplo do cálculo recursivo de  $4!$ , 10  
 exemplos de grafos, 368  
 expansão telescópica, 12-13  
 exponenciais, 16-17

**F**

fila, 191, 192  
   estática, 183  
   heterogênea, 183  
   homogênea, 183  
 FILO (*first in last out*), 184  
 folha, 373  
 função  
   complexidade de tempo, 3  
   de complexidade de espaço, 3  
   de espalhamento, 276  
   de *hashing*, 244  
   logarítmica, 17  
   recursiva, 10

função monotonicamente  
 crescente, 16  
 decrescente, 16

**G**

grafo direcionado, 372  
 grafo não orientado e sua matriz de  
 incidência, 377  
 grafo orientado e sua matriz de incidência, 376  
 grafos  
   conceito, 368  
   acíclico, 371  
   bipartido, 360  
   completo, 369  
   conexo, 371  
   desconexo, 371  
   isomorfos, 371-372  
   ponderado, 372  
   regular, 369-370  
   simples, 369  
   trivial, 369  
 grau de entrada, 373  
 grau de saída, 373

**H**

*hashing* uniforme, 267  
*heap*, 201  
   binário, 425  
   máximo, 201, 202-215  
   *max-min*, 216  
   mínimo, 201, 202-215  
   *min-max*, 216  
*heaps* binários  
   máximos, 201, 202  
   mínimo, 201, 202

**I**

ilustrações distintas de uma árvore binária, 279  
 implementação em JAVA de uma tabela  
   *hashing*, 261  
 índices dos vetores, 183

**I**  
Inserção  
no fim da lista, 180  
no início da lista, 180  
ordenada, 180

**L**  
laços, 369  
limite  
assintótico firme, 8  
assintótico inferior, 8  
assintótico superior, 7  
linguagem  
de programação, 2, 181, 191, 198, 408  
JAVA, 3, 181  
lista  
de adjacências, 377-378  
dinâmica, 105  
dinâmica heterogênea, 106  
dinâmica homogênea, 106  
duplamente encadeada e não ordenada, 105, 130  
duplamente encadeada e ordenada, 105, 142  
estática heterogênea, 106  
homogênea, 105  
simplesmente encadeada e não ordenada, 105, 106  
simplesmente encadeada e ordenada, 105, 107  
listas circulares, 105, 155  
logaritmos, 17-18

**M**

matriz de adjacências, 375-376  
para um grafo não direcionado, 375  
matriz de incidências, 376-377  
melhor caso, 4  
memória, 1-3, 5, 105, 106, 117, 143, 183, 185, 191, 244  
método  
da divisão, 244  
da expansão telescópica, 12, 74  
master, 15  
modelo

de RAM, 3  
matemático, 1  
monotonicidade, 16  
multigrafo, 369

**N**

Nível de um nó, 216, 283  
Nó  
ancestral, 281  
descendente, 281  
filho, 280  
folha ou terminal, 280  
pai, 280  
descendentes direito, 282  
descendentes esquerdo, 282  
irmãos, 280  
Notação  $o$ , 9  
Notação  $O$ , 7  
notação  $\Omega$ , 8  
notação  $v$ , 9-10  
notação  $Q$ , 8  
notações logarítmicas, 17  
número  
harmônico, 20  
máximo de nós em um nível, 283  
pequeno de colisões, 244

**O**

ômega minúsculo, 8. Veja também notação  $v$   
Operações com a notação  $O$ , 7

**P**

pilha, 184-191  
estática, 183  
heterogênea, 183  
homogênea, 183  
pior caso, 4  
pisos, 16  
polilogaritmicamente limitada, 18  
polinomialmente limitada, 16  
polinômios, 16-18

procedimento

processo

progressão

aritmética (PA), 5, 19

geométrica (PG), 20

propriedades de logaritmos, 17

## R

raiz da árvore, 73, 89, 203, 278, 329, 366

RAM (*Random Access Machine* — Máquina de Acesso Aleatório), 3

recorrências, 10-12, 15

regra

da associatividade, 19

de distributividade, 19

relação de recorrência, 11, 12, 13

remoção, 180, 181

representação de grafos, 375

rotação

dupla com filho para a direita e pai para a esquerda, 332

dupla com filho para a esquerda e pai para a direita, 333

simples para a direita, 334, 335

simples para a esquerda, 330, 331

## S

série

aritmética, 19-20

geométrica, 20

harmônica, 20

soma geométrica infinita, 20

somas integrais e diferenciais, 20

somatório de termos, 5, 18, 20, 51

somatórios, 18-20

subgrafo gerador, 360, 373, 374

sumidouro, 373

## T

tabela com endereçamento aberto, 256

tabela *hashing*, 244

implementada com endereçamento aberto, 245

implementada com lista, 267, 270-276

técnica da divisão e conquista, 53

combinar, 53, 61

conquistar, 53, 61

dividir, 53, 61

tentativa

linear, 245, 250, 256, 267

quadrática, 245, 256, 257, 261, 267

Teorema Master, 15, 74

tetos, 3, 16

tipo abstrato de dados, 1, 2

tipos de notação assintótica, 5

trajeto, 371, 373

## V

vértice de corte, 374

vértice *fonte*, 373

vértice *sumidouro*, 373

# Sobre as autoras

## Ana Fernanda Gomes Ascencio

Graduada em ciência da computação pela Pontifícia Universidade Católica de São Paulo (PUC – SP, 1992). Especialista em sistemas de informação pela Universidade Federal de São Carlos (UFSCAR, 1994). Especialista em educação pela Universidade para o Desenvolvimento do Estado e da Região do Pantanal (UNIDERP, 2004). Mestre em ciência da computação pela Universidade Federal do Rio Grande do Sul (UFRGS, 2000). Professora da Universidade Anhanguera-Uniderp, ministrando disciplinas na área de programação de computadores desde 1994. Autora dos livros *Lógica de programação com Pascal* (Makron Books, 1999), *Fundamentos da programação de computadores, 1<sup>a</sup> ed.* (Prentice Hall, 2002), *Introdução ao desenvolvimento de aplicações em Delphi* (Uniderp, 2004), *Aplicações das estruturas de dados em Delphi* (Prentice Hall, 2005), *Desenvolvimento de um sistema com Delphi, Postgresql e Sql* (Visual Books, 2007) e *Fundamentos da programação de computadores, 2<sup>a</sup> ed.* (Prentice Hall, 2007).

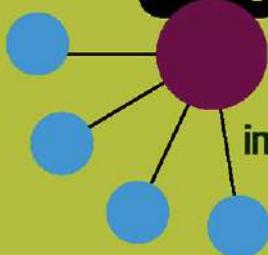
## Graziela Santos de Araújo

Graduada em ciência da Computação pela Universidade Federal de Mato Grosso do Sul (UFMS, 1999), mestre em ciência da computação pela Universidade Federal de Mato Grosso do Sul (UFMS, 2003), professora da Universidade Anhanguera-Uniderp, ministrando disciplinas na área de programação de computadores e teoria da computação de 2003 a 2010. Atualmente é professora da Universidade Federal de Mato Grosso do Sul.

**[Ciência da computação]**

Ana Fernanda Gomes Ascencio  
& Graziela Santos de Araújo

# estruturas de dados



algoritmos, análise da  
complexidade e  
implementações em  
JAVA e C/C++

Este livro tem dois objetivos claros: ser um forte aliado dos professores da área de computação, ajudando-os a levar os estudantes a analisar diversas soluções para um mesmo problema, e contribuir para o desenvolvimento de profissionais que querem adotar padrões elevados de qualidade no desempenho de suas atividades.

Para atingir esses objetivos, ele aborda estruturas de dados e análise da complexidade, bem como análise de algoritmos e sua representação gráfica (os grafos), de maneira bastante didática, altamente explicativa. Além disso, traz as implementações e os códigos tanto em Java como em C e C++, permitindo ao leitor entender não apenas a aplicação prática dessas linguagens, mas também suas semelhanças e diferenças.



[www.prenhall.com/ascencio\\_araujo\\_br](http://www.prenhall.com/ascencio_araujo_br)

O site de apoio oferece, para professores, apresentações em Power Point  
e, para estudantes, exercícios resolvidos utilizando códigos-fonte.

Prentice Hall  
é um selo da

**PEARSON**

[www.pearson.com.br](http://www.pearson.com.br)

ISBN 978-85-7605-881-6

