Informática

Pesquisa e Ordenação de Dados

Gerardo Valdisio Rodrigues Viana
Glauber Ferreira Cintra
Ricardo Holanda Nobre

2ª edição
Fortaleza - Ceará

Ed UECE

2015

Editora Filiada à

ASSOCIAÇÃO BRASILEIRA DAS EDITORAS UNIVERSITÁRIAS

Editora da Universidade Estadual do Ceará – EdUECE
Av. Dr. Silas Munguba, 1700 – Campus do Itaperi – Reitoria – Fortaleza – Ceará
CEP: 60714-903 – Fone: (85) 3101-9893
Internet: www.uece.br – E-mail: eduece@uece.br

Secretaria de Apoio às Tecnologias Educacionais
Fone: (85) 3101-9962

Sumário

Apresentação 5

Capítulo 1 – Introdução 7

1. Introdução 9
2. Armazenamento de Informações 10
3. Procedimentos e Funções 12
   3.1 Blocos de Comandos 12
   3.2 Procedimentos 13
   3.3 Funções Simples 13
   3.4 Funções Recursivas 14
4. Estruturas Utilizadas nos Algoritmos 16
5. Exemplo de Uso de Procedimentos e Funções 18

Capítulo 2 – Ordenação Interna 19

1. Ordenação Interna 21
2. O Problema da Ordenação 22
3. Bolha 24
4. Inserção 27
5. Seleção 28
6. Shellsort 30
7. Mergesort 32
8. Quicksort 36
9. Heaps Binários 41
10. Heapsort 45
11. Countingsort 46
12. Bucketsort 48
13. Radixsort 50

Capítulo 3 – Ordenação Externa 55

1. Definição e mecanismos 57
2. Intercalação de dois caminhos 59
3. Intercalação de três caminhos 63
4. Intercalação de k caminhos 65
5. Comparação entre Distribuições Equilibradas 66
   5.1 Classificação Equilibrada de dois caminhos 66
   5.2 Classificação Equilibrada de três caminhos 67

5.3 Classificação Equilibrada de múltiplos caminhos....68
6. Intercalação Polifásica....69
Capítulo 4 – Técnicas de pesquisa....73
1. Técnicas de Pesquisa....75
2. Busca Sequencial....76
3. Busca Binária....77
4. Tabelas de Dispersão....80
4.1 Hashing Fechado....81
4.2 Hashing Aberto....83
5. Árvores de Busca Balanceadas....85
5.1 Árvores AVL....85
5.2 Árvores B....92
5.3 Árvores B+....98
Sobre os autores....103
Anexo....104
Relação de Algoritmos....104
Lista de Algoritmos....119

Apresentação

O problema da ordenação é um dos mais importantes e mais estudados na ciência da computação. Em diversas situações cotidianas é preciso colocar uma lista em ordem para facilitar a busca de informações nela contidas. Ordenar, ou classificar dados, significa armazená-los numa forma adequada de modo a facilitar sua pesquisa, ou busca, e assim tornar mais ágil a recuperação das informações.

Neste livro serão mostradas inicialmente as técnicas clássicas de classificação de dados, seguidas pelos métodos usuais de pesquisa. Em todos os procedimentos descritos utilizam-se estruturas de dados conhecidas, como vetores, matrizes, árvores, listas, pilhas e filas.

Adiantamos que, uma classificação é dita interna, quando os dados podem ser totalmente armazenados na memória principal, e externa, se parte deles estiverem, durante o processo de ordenação, num meio de armazenamento auxiliar. Os métodos de classificação externa são realizados através de uma técnica de projeto de algoritmos denominada "divisão e conquista", onde os dados são divididos em séries menores, de tamanho tal que seja possível ordená-los por algum método de classificação interna; ao final, as séries devem ser intercaladas a fim de gerar uma única saída, totalmente ordenada.

A pesquisa, outra tarefa importante na computação, corresponde à busca de informações contidas num arquivo ou, em geral, numa lista ou coleção de dados. Deseja-se que esse processo seja executado sem que haja a necessidade de inspecionar toda a coleção de dados, para isto são apresentadas diversas técnicas e estruturas de dados que permitem fazer pesquisa com bastante eficiência. Discutimos as operações de inserção, busca e remoção nessas estruturas, analisando sua complexidade temporal e espacial. Essa análise também é feita para os procedimentos de classificação interna.

Ao final de cada capítulo são propostos exercícios, cujo objetivo é avaliar os conhecimentos adquiridos e rever as definições abordadas no respectivo capítulo.

Os Autores

Capítulo 1

Introdução

Objetivos

- Introduzir os conceitos de pesquisa e ordenação de dados
- Mostrar as formas de armazenamento das informações
- Definir procedimentos e funções
- Apresentar as estruturas dos algoritmos

Apresentação

Nas considerações gerais contidas neste capítulo introdutório, destacamos inicialmente a importância da "Pesquisa e Ordenação" para a computação, e a necessidade do uso de técnicas eficientes para que seu processamento seja realizado no menor tempo possível. Mostramos, também, que a abordagem deste assunto deve ser feita de forma integrada entre os dois tópicos, pesquisa e ordenação, e que é requerida uma revisão do assunto Estrutura de Dados para que seja compreendido como as informações são armazenadas e de que forma os processos as utilizam. Por fim, é apresentada a forma padrão usada para codificação dos algoritmos, bem como de seus procedimentos e funções.

1. Introdução

A pesquisa e a ordenação são operações bastante utilizadas nos sistemas de programação, desde a origem da computação. Independente da geração ou do ambiente computacional, ordenar, ou classificar dados, significa armazená-los numa forma adequada de modo a facilitar sua pesquisa, ou busca, e assim tornar mais ágil a recuperação das informações.

O conceito de um conjunto ordenado de elementos tem importância tanto na informática como em nossa vida cotidiana. Imagine quão complicado seria localizar um número de telefone num catálogo se o mesmo não fosse classificado por ordem alfabética nem por endereço. O processo de busca somente é facilitado porque existe uma ordenação relacionada; observe no exemplo citado que, se temos o número de um telefone e desejamos identificar seu proprietário, ou seu endereço, a pesquisa seria complexa caso não existisse uma ordenação numérica, crescente ou decrescente, destes telefones. Em geral, a lista de telefones nesta ordem não existe porque esta é uma consulta incomum.

Aqui serão mostradas inicialmente as técnicas clássicas de classificação de dados, seguidas pelos métodos usuais de pesquisa. Em todos os procedimentos descritos utilizam-se estruturas de dados conhecidas, como vetores, matrizes, árvores, listas, pilhas e filas. Os algoritmos correspondentes a estes procedimentos são apresentados numa forma de programa estruturado, sendo ao final apresentados alguns algoritmos de ordenação, na linguaguem de programação "C". Nosso propósito é que o leitor entenda com facilidade a metodologia empregada.

Adiantamos também que uma classificação é dita interna se os dados puderem ser totalmente armazenados na memória principal e é dita externa se parte deles estiverem, durante o processo de ordenação, num meio de armazenamento auxiliar, ou seja, numa "área de trabalho" que geralmente está gravada em um disco magnético. Os métodos de classificação externa são realizados através de uma técnica de projeto de algoritmos denominada "divisão e conquista", onde os dados são divididos em séries menores, de tamanho tal que seja possível ordená-los por algum método de classificação interna; ao final, as séries devem ser intercaladas a fim de gerar uma única saída, totalmente ordenada.

O termo em inglês correspondente à ordenação é sort e à intercalação é merge, de modo que, é comum a literatura afim denominar um programa utilitário para classificação, de forma geral, como Sort-Merge.

Nas seções seguintes deste capítulo é apresentada a forma como os dados a serem classificados estão armazenados, seguido das definições de procedimentos e funções e da organização estruturada dos algoritmos. No capítulo 2 são mostrados os diversos métodos de classificação interna e no capítulo 3, os de classificação externa e, por fim, no capítulo 4, abordaremos as principais técnicas de pesquisa utilizadas.

2. Armazenamento de Informações

Na prática o processo de ordenação não se resume apenas a classificar dados isolados. Embora a maioria das abordagens a este tema sempre seja feita com valores numéricos, por motivos didáticos, devemos estar cientes que as aplicações representam informações em forma de registro, ou coleção de dados, que em geral contém um campo especial chamado chave que é o "valor" a ser ordenado que pode ser numérico ou não. No caso geral podemos chamá-lo de alfanumérico correspondendo a uma sequência de caracteres (letras, dígitos ou especiais) que são geralmente armazenados em oito bits usualmente codificados em ASCII (American Standard Code for Information Interchange).

Assim, quando um algoritmo de ordenação, após a comparação de

duas chaves, necessita trocá-las de posição é necessário trocar também todos os dados correspondentes acoplados a cada chave. Numa ordenação ascendente (do menor para o maior) do exemplo abaixo, se K1>K2 então, os campos "Nome" e "Endereço" do registro R1 devem ser permutados pelos respectivos campos de R2 , ou seja, os registros trocam de posição (R1↔R2).

	Chave	Nome	Endereço	
.	.	...	...	i -1
R_1 (posição i)	K_1	Guilherme Viana	Rua 5 de maio, 2608	i
.	.	...	...	i +1
.	.	...	...	.
.	.	...	...	.
.	.	...	...	j -1
R_2 (posição j)	K_2	Luan Lima	Rua 29 de julho, 1003	j
.	.	...	...	j +1

Figura 1.1: Modelo de registros de um arquivo

Para permutar dois registros há necessidade de se utilizar uma variável auxiliar que tem a mesma estrutura dos registros, conforme mostrado na Figura 1.2.

$R^{(aux)}$	Chave	Nome	Endereço

Figura 1.2: Formato de um registro auxiliar

Usando esta estrutura, escrever os seguintes comandos:

R(aux)←R1; R1←R2; R2←R(aux);

O conjunto de todos os registros considerados é chamado de arquivo que pode ser visualizado como uma tabela, onde cada linha representa um registro específico e cada coluna representa um campo dentro do registro. Uma classificação pode ocorrer sobre os próprios registros ou sobre uma tabela auxiliar de ponteiros. De forma abstrata, podemos imaginar sempre a tabela, ou arquivo, como no modelo Figura 1.2. No caso concreto do uso de ponteiros, ou apontadores, o processo é chamado "classificação por endereços" e de forma indireta há movimentação somente dos ponteiros em vez dos dados.

Além disso, também se deve considerar a ocorrência de mais de uma chave de classificação. Nesse caso, uma das chaves é chamada de "primária", ou principal, e as demais "secundárias". Para efeito, o processo de orde-

nação é aplicado tantas vezes quanto seja a quantidade de chaves. Há, no entanto, técnicas de concatenação de chaves com o objetivo de criar uma única chave alfanumérica que será aquela usada nas comparações.

É possível ainda observar casos em que dois registros tenham a mesma chave, ou seja, $K_1=K_2$. Quando isto ocorre não haverá necessidade de permutá-los, ou seja, os registros são mantidos nas mesmas posições originais.

Convencionamos que na apresentação dos métodos de classificação e de seus algoritmos, as comparações são feitas apenas com as chaves, ou valores isolados. Outra convenção útil é considerar a ordenação sempre ascendente, onde ao final as chaves estarão em ordem crescente, ou de forma geral em ordem "não decrescente", caso existam chaves repetidas.

Para uma ordenação na ordem inversa (não crescente) a única alteração a ser feita em todos os algoritmos é substituir a comparação $K_1>K_2$ por $K_1<K_2$.

3. Procedimentos e Funções

3.1 Blocos de Comandos

Um bloco é uma parte delimitada de um algoritmo que contém comandos. Um comando pode ser de atribuição, ou uma declaração de entrada ou saída, ou uma estrutura de controle, ou de repetição, ou ainda pode referenciar uma chamada a outros blocos. Os blocos, organizados de forma indentada, podem ser aninhados conforme mostrado na Figura 1.3.

Indentar: Neologismo derivado da palavra em inglês *indent* que significa "recuar" ou "abrir parágrafo".

```
ALGORITMO <nome>
// <comentários>
   ENTRADA: <lista de variáveis>
   SAÍDA: <lista de variáveis>

   BLOCO
   // <comentários>
      COMANDOS
  BLOCO
// <comentários>
   COMANDOS
FIM {nome}
```

Figura 1.3: Estrutura geral de um algoritmo

De modo geral, podemos dizer que um algoritmo é um bloco único correspondente à rotina ou módulo principal; de forma análoga podemos afirmar que um sub-algoritmo (procedimento ou função) é um bloco auxiliar que corresponde a uma sub-rotina.

3.2 Procedimentos

Um **procedimento** é um bloco criado através de uma declaração que o associa a um nome ou identificador para que seja feita sua chamada ou referência. O uso de procedimentos possibilita que um bloco usado repetidas vezes em vários pontos do algoritmo possa ser escrito uma única vez e chamado quando necessário.

A forma de um procedimento é mostrada na Figura 1.4.

```
PROCEDIMENTO <nome> (parâmetros)
// <comentários>
   ENTRADA: <lista de variáveis>
   SAÍDA: <lista de variáveis>

   BLOCO
   // <comentários>
      COMANDOS
  BLOCO
// <comentários>
   COMANDOS
RETORNE
FIM {nome}
```

Figura 1.4: Estrutura geral de um procedimento

O procedimento é um objeto estático, no sentido de que para ser executado é necessário que seja ativado por uma chamada pelo seu <nome> a partir do módulo principal do algoritmo ou através de outro procedimento num nível hierárquico superior.

Através dos parâmetros é possível a transferência de dados entre os módulos envolvidos. Cada parâmetro pode armazenar um único valor se a variável corresponde é simples ou vários valores se for uma variável estruturada de forma homogênea com uma ou mais dimensão.

3.3 Funções Simples

Uma função corresponde a um tipo especial de procedimento. Possuem características próprias como, por exemplo, a necessidade da existência de uma atribuição de uma expressão ou valor ao seu nome declarado em pelo menos um ponto da sequência de comandos.

Ao declarar uma função é necessário especificar seu tipo (inteiro, real, lógico ou literal) de forma análoga a uma função matemática.

A forma padrão de uma função simples é mostrada a seguir.

```
FUNÇÃO <nome> (parâmetros): <tipo básico>
// <comentários>
   ENTRADA: <lista de variáveis>

   BLOCO
   // <comentários>
      COMANDOS
  BLOCO
// <comentários>
   COMANDOS
   nome = <expressão>
   COMANDOS
  DEVOLVA <nome>
FIM {nome}
```

Figura 1.5: Estrutura geral de uma função

Na seção 5 é mostrado um exemplo para consolidar os conceitos de funções e procedimentos, bem como, para observar as semelhanças e as diferenças entre os mesmos.

3.4 Funções Recursivas

Uma função **recursiva** é aquela que faz uma chamada a si própria. A ideia consiste em utilizar a própria função no escopo de sua definição.

Em todas as funções recursivas devem existir dois passos bem definidos, um deles chamado **base** cujo resultado é imediatamente conhecido e o outro, **recursão** em que é possível resolver a função, ou seja, calcular o valor final da função utilizando a mesma sucessivas vezes com entradas menores.

A estrutura de uma função recursiva é semelhante a uma função simples, com a particularidade de que seu nome surge pelo menos uma vez no lado direito de um comando de atribuição.

Um exemplo clássico é o uso de uma função para calcular o valor do fatorial de um número inteiro não negativo. A seguir apresentamos duas formas

de implementação, uma através de uma função simples (FAT) e outra usando uma função recursiva (FATREC).

a) Função Fatorial Simples

```
FUNÇÃO FAT (N:inteiro): real

   ENTRADA: // um número inteiro positivo

   F = 1
   PARA i = 1 até N
      F = F * i
   // fim para
   FAT = F

   DEVOLVA FAT

FIM {FAT}
```

Algoritmo 1.1: Calcula o fatorial de N de forma direta

Mesmo sendo o fatorial um número inteiro, optamos aqui defini-lo como real para ampliar sua magnitude ou ordem de grandeza.

b) Função Fatorial Recursiva

```
FUNÇÃO FATREC (N:inteiro): real

ENTRADA: // um número inteiro positivo

SE N < 2
       FATREC = 1
SENÃO
       FATREC = N * FATREC(N-1)

DEVOLVA FATREC

FIM {FATREC}
```

Algoritmo 1.2: Calcula o fatorial de N de forma recursiva

4. Estruturas Utilizadas nos Algoritmos

Os algoritmos apresentados usam uma pseudo-linguagem simples, comumente chamada de Portugol, de fácil entendimento, que contém resumidamente os seguintes elementos:

Símbolos:

- = : atribuição;
- { } : delimitadores para descrever comentários; pode iniciar por //
- ( ) : para indicar ordem de execução das operações nas expressões ou agrupar parâmetros de sub-rotinas;

Termos Básicos:

- BLOCO - contém um ou mais comandos simples;
- lista de variáveis - variáveis separadas por vírgulas;
- condição - expressão lógica cujo resultado pode ser verdadeiro (V) ou falso (F);
- início–fim/fim-bloco - delimitadores de bloco (opcionais);
- RETORNE - comanda o encerramento de um procedimento ou função, devolvendo o controle de execução ao bloco de chamada;
- DEVOLVA v – mesma função anterior, indica que retorna para o bloco de chamada com a estrutura v.

- Procedimentos de Entrada e Saída:

```
ENTRADA: < lista de variáveis >

SAÍDA: < lista de variáveis >
```

Figura 1.6: Comandos de entrada e saída de dados

Estruturas de Controle de Decisão e Desvio:

```
// Desvio Simples
SE < condição >
     BLOCO
// Desvio Duplo
SE < condição >
     BLOCO-1
  SENÃO
     BLOCO-2
// Desvio Múltiplo
CASE < expressão-teste >
   <lista-1>: BLOCO-1
   <lista-2>: BLOCO-2
     . . .
   <lista-n>: BLOCO-n
```

Figura 1.7: Comandos de Decisão

Estruturas de Repetição:

```
// Com teste no fim do bloco

FAÇA
     BLOCO
ENQUANTO < condição >

// Com teste no início do bloco

ENQUANTO < condição >
     BLOCO

// Com limites da variável de controle

PARA v = <valor-inicial>  até  <valor-
-final>
    BLOCO
```

Figura 1.8: Comandos de Repetição

5. Exemplo de Uso de Procedimentos e Funções

Exemplo 1.1: Problema: Binômio de Newton

Dados dois valores inteiros não negativos, $m \geq n$, calcular o valor de

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

Utilizar a função **FAT** definida em 3.4.a (Algoritmo 1.1).

```
ALGORITMO BINÔMIO

ENTRADA: INTEIROS m E n
SAÍDA: REAL b COM O VALOR BINOMIAL

PROCEDIMENTO TROCA ( x , y )

ENTRADA: INTEIROS x E y
SAÍDA: x E y COM VALORES TROCADOS

w = x
x = y
y = w

RETORNA
FIM {TROCA}

FUNÇÃO FAT (n)
        // mesmo bloco da subseção
1.2.4.a
  FIM {FAT}

SE m < n
   TROCA ( m , n )

b = FAT (m) / ( FAT (n) * FAT (m-n) )

FIM {BINÔMIO}
```

Algoritmo 1.3: Calcula o Binômio de Newton

Capítulo 2

Ordenação Interna

Objetivos

- Definir o problema da ordenação
- Apresentar os métodos de classificação interna
- Analisar a complexidade dos algoritmos de ordenação

Apresentação

Neste capítulo iniciamos definindo formalmente o problema da ordenação e os principais critérios de classificação de dados. Em seguida, apresentamos em detalhes os métodos de ordenação interna, discutindo sua eficiência em termos de consumo de tempo e de memória e outros aspectos relevantes. Na ordem, abordaremos os métodos Bolha, Inserção e Seleção que são bastante simples e servem para introduzir ideias a serem utilizadas em outros métodos mais eficientes. Seguimos com os métodos Shellsort, Mergesort, Quicksort e Heapsort, considerados superiores aos anteriores, também, como eles, baseados na operação de comparação. Por fim, discutiremos os métodos Countingsort, Bucketsort e Radixsort que têm em comum o fato de não utilizarem a operação de comparação e, sob algumas hipóteses, eles ordenam em tempo linearmente proporcional ao tamanho da lista.

1. Ordenação Interna

Como dito no capítulo anterior, a ordenação de dados é um dos problemas mais importantes e mais estudados dentro da ciência da computação. Em diversas situações cotidianas é preciso colocar uma lista em ordem para facilitar a busca de informações nela contidas.

Neste capítulo discorremos sobre os principais critérios de ordenação, definindo antes, na seção 2, formalmente o problema da ordenação. Nas seções seguintes são apresentados em detalhes os diversos métodos de ordenação interna. Para cada um deles é mostrada sua eficiência em termos de consumo de tempo e de memória e outros aspectos considerados relevantes.

Inicialmente, nas seções 3, 4 e 5, abordaremos os métodos Bolha, Inserção e Seleção. Tais métodos, considerados inferiores, são bastante simples, mas introduzem ideias que servem de base para outros métodos mais eficientes. Esses métodos utilizam como uma de suas operações básicas a comparação de elementos da lista.

Em seguida, nas seções 6, 7, 8 e 10, apresentaremos os métodos Shellsort, Mergesort, Quicksort e Heapsort. Esses métodos, considerados superiores, utilizam estratégias mais sofisticadas como, por exemplo, divisão e conquista e o uso de estruturas de dados conhecidas como heaps binários, descritos na Seção 9. Esses métodos também são baseados na operação de comparação.

ASCII - American Standart Code for Information Interchange - Código padrão americano para o intercâmbio de informação

Por fim, nas seções 11, 12 e 13, discutiremos os métodos Countingsort, Bucketsort e Radixsort. Esses últimos métodos têm em comum o fato de não utilizarem a operação de comparação. Além disso, sob certas hipóteses, eles ordenam em tempo linearmente proporcional ao tamanho da lista.

2. O Problema da Ordenação

Para que uma lista possa ser ordenada os seus elementos devem ser comparáveis dois a dois. Isso significa que dados dois elementos da lista deve ser sempre possível determinar se eles são equivalentes ou se um deles é "menor" que o outro. A idéia associada ao termo "menor" depende do critério de comparação utilizado.

Diz-se que uma lista está em ordem crescente se cada um de seus elementos é menor ou igual ao seu sucessor segundo algum critério de comparação. A definição de ordem decrescente é análoga. Como dito, uma lista somente pode ser colocada em ordem se os seus elementos forem dois a dois comparáveis. O problema da ordenação consiste em, dada uma lista, colocá-la em ordem crescente ou decrescente. Os principais critérios são comparação numérica, comparação alfabética e comparação cronológica.

- Comparação numérica: um número x é menor do que um número y se a expressão x – y resulta em um número negativo. Esse é o tipo mais comum de comparação e, de certa forma, todos os demais critérios de comparação derivam dele.
- Comparação alfabética: um caractere é menor do que outro se precede esse outro na ordem do alfabeto. Por exemplo, o caractere "a" precede o "b", o "b" precede o "c" e assim por diante. No computador, os caracteres são representados através de números (segundo alguma codificação, por exemplo, ASCII, UNICODE etc). Dessa forma, a comparação entre caracteres acaba sendo uma comparação entre os seus códigos numéricos e, portanto, é uma comparação numérica.
- Comparação cronológica: Uma data é menor do que outra se ela antecede essa outra. Uma data pode ser representada através de três números, um para representar o ano, outro para o mês e outro

para o dia. Ao comparar uma data com outra, comparamos o ano. Em caso de empate, comparamos o mês. Em caso de novo empate, comparamos o dia. Sendo assim, a comparação cronológica também é feita comparando-se números.

É possível definir outros critérios de comparação. Por exemplo, podemos comparar caixas, considerando menor aquela que apresentar menor volume. Como o volume é expresso por um número, esse tipo de comparação também é uma comparação numérica.

Poderíamos, no entanto, considerar uma caixa menor do que outra se ela cabe nessa outra. Nesse caso, podemos ter caixas incomparáveis. Utilizando esse critério de comparação, podemos ter uma lista de caixas que não pode ser ordenada. Além disso, quando temos listas heterogêneas, pode não ser possível ordená-la. Por exemplo, a lista (4, "bola", 2, 15/02/2011) não pode ser ordenada, pois seus elementos não são todos dois a dois comparáveis.

Dizemos que uma lista está em ordem crescente se cada elemento da lista é menor ou igual (segundo o critério de comparação adotado) ao seu sucessor. Se cada elemento da lista é estritamente menor do que seu sucessor, dizemos que a lista está em ordem estritamente crescente. Analogamente, dizemos que uma lista está em ordem decrescente se cada elemento da lista é maior ou igual ao seu sucessor. Se cada elemento da lista é estritamente maior do que seu sucessor, dizemos que a lista está em ordem estritamente decrescente. Obviamente, ordenação estrita somente é possível se todos os elementos da lista forem distintos.

Aquilo que definimos como *ordem crescente* é chamada por alguns autores de *ordem não decrescente* enquanto que o que chamamos de *ordem estritamente crescente* é chamada por esses mesmos autores de *ordem crescente*. O que chamamos de *ordem decrescente* é chamada por alguns de *ordem não crescente* enquanto que a *ordem estritamente decrescente* é chamada de *ordem decrescente*.

O problema da ordenação consiste em, dada uma lista cujos elementos sejam todos dois a dois comparáveis, permutar a ordem de seus elementos de modo a deixar a lista em ordem crescente ou decrescente.

Se a lista não for muito grande ela pode ser armazenada na memória principal (interna) do computador e ser ordenada usando apenas a memória interna. Os algoritmos que ordenam utilizando apenas memória interna são chamados de métodos de ordenação de interna.

Em contrapartida, os algoritmos que fazem uso de memória externa são chamados de métodos de ordenação de externa. Alguns desses métodos serão discutidos no Capítulo 3.

No restante deste capítulo descreveremos diversos métodos de ordenação interna. Embora uma lista possa ser armazenada usando-se uma lista encadeada, descreveremos os algoritmos considerando que a lista está armazenada num vetor indexado a partir da posição zero. Além disso, consideraremos que o vetor armazena números e, portanto, o critério de comparação será numérico. Na descrição dos algoritmos de ordenação, vamos sempre considerar que o vetor é passado por referência.

Convém salientar que a lista pode ser constituída de registro e podemos ordená-la por um de seus campos, que chamamos de chave de ordenação. A comparação entre os elementos da lista é feita comparando-se os valores das chaves de ordenação dos dois elementos.

Obviamente, os métodos aqui descritos podem ser facilmente adaptados por ordenar listas encadeadas cujos elementos são registros. Em alguns exercícios tais adaptações são requeridas.

3. Bolha

O Método Bolha é bastante simples e intuitivo. Nesse método os elementos da lista são movidos para as posições adequadas de forma contínua, assim como uma bolha move-se num líquido. Se um elemento está inicialmente numa posição i e, para que a lista fique ordenada, ele deve ocupar a posição j, então ele terá que passar por todas as posições entre i e j.

Em cada iteração do método, percorremos a lista a partir de seu início comparando cada elemento com seu sucessor, trocando-os de posição se houver necessidade. É possível mostrar que, se a lista tiver n elementos, após no máximo (n–1) iterações a lista estará em ordem. A seguir fornecemos uma descrição em pseudocódigo do Bolha.

```
ALGORITMO BOLHA
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 1 até n - 1
    PARA j = 0 até n - 1 - i
      SE  L[j] >  L[j+1]
        AUX = L[j]          // SWAP
        L[j] = L[j+1]
        L[j+1] = AUX
FIM {BOLHA}
```

Algoritmo 2.1: Bolha

Na descrição do Algoritmo Bolha deixamos claro como realizar a troca (swap) de elementos da lista. No restante deste capítulo, faremos o swap usando o procedimento troca. Assim, a chamada troca(L[i], L[j]) serve para trocar os conteúdo das posições i e j do vetor L.

É fácil perceber que ao final da primeira iteração do laço mais externo do Algoritmo Bolha, o maior elemento da lista estará na última posição do vetor. Sendo assim, na iteração seguinte não precisamos mais nos preocupar com a última posição do vetor. Ao final da segunda iteração, os dois maiores valores

da lista estarão nas duas últimas posições do vetor L, em ordem crescente. Não precisamos então nos preocupar com as duas últimas posições do vetor.

A letra grega "Θ" é usada para denotar a ordem de complexidade media e "O" para o pior caso.

De fato, no Algoritmo Bolha é válido o seguinte invariante: ao final da iteração k do laço mais externo, os k maiores valores da lista estarão nas k últimas posições do vetor, em ordem crescente. A corretude do algoritmo decorre desse invariante. Além disso, o invariante explica porque no laço mais interno o contador j só precisa variar de 0 (primeira posição do vetor) até n – 1 – i.

Claramente, a instrução crítica do Algoritmo Bolha é a comparação (se). Em cada iteração do laço mais interno é feita uma comparação. Como a quantidade de iterações do laço interno diminui à medida que i aumenta, o número de comparações realizadas pelo algoritmo é:

i	Comparações
0	n – 1
1	n – 2
...	...
n – 1	1
Total	$(n^2 - n)/2$

Assim, a complexidade temporal do Algoritmo Bolha pertence a Θ(n2). Além dos parâmetros de entrada (L e n), o algoritmo utiliza apenas três variáveis escalares (i, j e aux). Dessa forma, a quantidade extra de memória utilizada pelo algoritmo é constante. Sendo assim, sua complexidade espacial pertence a O(1). Os métodos de ordenação cuja complexidade espacial pertence a O(1) são chamados métodos de ordenação in situ. Esse é o caso do Método Bolha.

In situ: Expressão latina, usada em vários contextos, que significa “no lugar”.

Outro aspecto importante dos métodos de ordenação é a estabilidade. Dizemos que um método de ordenação é estável se ele preserva a ordem relativa existente entre os elementos repetidos da lista. Por exemplo, se a ordenação da lista (4, 7, 6, 7, 2) resulta em (2, 4, 6, 7, 7), a ordenação foi estável. Observe que o sete sublinhado estava à direita do outro sete e ao final da ordenação ele continuou à direita. Se todas as ordenações produzidas por um método de ordenação são estáveis, dizemos que o método é estável.

Observe que no Algoritmo Bolha, quando comparamos dois elementos da lista, somente fazemos o swap se o antecessor for menor do que o sucessor, pois tivemos o cuidado de utilizar uma desigualdade estrita na comparação. Como consequência, o Algoritmo Bolha é estável.

A seguir exibimos o conteúdo de um vetor após cada iteração do laço mais externo do Algoritmo Bolha.

	0	1	2	3	4	5
Lista original	9	4	5	10	5	8
1ª iteração	4	5	9	5	8	10
2ª iteração	4	5	5	8	9	10
3ª iteração	4	5	5	8	9	10
4ª iteração	4	5	5	8	9	10
5ª iteração	4	5	5	8	9	10

Figura 2.1: Ordenação com o Algoritmo Bolha

Neste exemplo percebemos que no final da segunda iteração a lista já estava ordenada. Na terceira iteração não houve nenhuma alteração na lista. Essa fato nos permitiria concluir que a lista já estava em ordem, de modo que as duas últimas iterações foram inúteis.

Podemos introduzir uma pequena modificação no Método Bolha de modo a torná-lo um pouco mais "esperto" e evitar iterações inúteis. Tal modificação consiste em usar uma variável (flag) para sinalizar se foi realizada alguma troca de elementos numa iteração do método. Caso nenhuma troca tenha sido realizada, a lista já estará em ordem e, portanto, o método deve parar. No exemplo anterior, após a terceira iteração a ordenação seria finalizada.

Essa variante do Bolha é conhecida como Bolha com Flag e seu pseudocódigo é fornecido no Algoritmo 2.2.

```
ALGORITMO BOLHA_COM_FLAG
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  i = 0
  FAÇA
    FLAG = FALSO
    PARA j = 0 até n - 1 - i
      SE  L[j] >  L[j+1]
        TROCA(L[J], L[J+1])
        FLAG = VERDADEIRO
    i = i + 1
  ENQUANTO FLAG = VERDADEIRO
FIM {BOLHA_COM_FLAG}
```

Algoritmo 2.2: Bolha com Flag

Enquanto que o tempo requerido pelo Bolha é sempre quadrático em relação ao tamanho da lista, o comportamento do Bolha com Flag é sensível ao tipo de lista fornecida como entrada. No melhor caso, que ocorre quando a lista fornecida já esta ordenada, o Algoritmo Bolha_com_Flag requer tempo

Θ(n) pois a comparação será sempre falsa e a variável flag jamais receberá o valor verdadeiro. Dessa forma, após a primeira iteração do laço externo o algoritmo será finalizado.

Suponha agora que o menor elemento está inicialmente na última posição da lista. Observe que a cada iteração do laço externo o menor elemento será movido apenas uma posição para a esquerda. Serão necessárias n – 1 iterações para que o menor elemento alcance a primeira posição da lista. Nesse caso, o comportamento do Algoritmo Bolha_com_Flag será idêntico ao do Algoritmo Bolha. Concluímos que no pior caso o Bolha com Flag requer tempo Θ(n2). Finalmente, é possível mostrar que o desempenho do Bolha com Flag no caso médio também pertence a Θ(n2).

4. Inserção

O Método Inserção, também conhecido como Inserção Direta, é bastante simples e apresenta um desempenho significativamente melhor que o Método Bolha, em termos absolutos. Além disso, como veremos no final dessa seção, ele é extremamente eficiente para listas que já estejam substancialmente ordenadas.

Nesse método consideramos que a lista está dividida em parte esquerda, já ordenada, e parte direita, em possível desordem. Inicialmente, a parte esquerda contém apenas o primeiro elemento da lista. Cada iteração consiste em inserir o primeiro elemento da parte direita (pivô) na posição adequada da parte esquerda, de modo que a parte esquerda continue ordenada. É fácil perceber que se a lista possui n elementos, após n – 1 inserções ela estará ordenada.

Para inserir o pivô percorremos a parte esquerda, da direita para a esquerda, deslocando os elementos estritamente maiores que o pivô uma posição para direita. O pivô deve ser colocado imediatamente à esquerda do último elemento movido.

```
ALGORITMO INSERÇÃO
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 1 até n - 1
    PIVO = L[i]
    j = i - 1
    ENQUANTO j ≥ 0 e L[i] > PIVO
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = PIVO
FIM {INSERÇÃO}
```

Algoritmo 2.3: Inserção

No melhor caso, que ocorre quando a lista fornecida como entrada já está ordenada, cada inserção é feita em tempo constante, pois o pivô é maior ou igual a todos os elementos da parte esquerda e a condição do laço interno nunca é verdadeira. Nesse caso, a complexidade temporal do Algoritmo Inserção pertence a Θ(n).

No pior caso, que ocorre quando a lista está inversamente ordenada, cada inserção requer que todos os elementos da parte esquerda sejam movidos para a direita, pois todos eles serão maiores do que o pivô. Nesse caso, a quantidade total de iterações do laço interno será (n2 - n)/2 e a complexidade temporal do Algoritmo Inserção será Θ(n2).

A complexidade temporal do Algoritmo Inserção no caso médio também pertence a Θ(n2). Isso decorre do fato de que, em média, cada iteração requer que metade dos elementos da parte esquerda sejam movidos para a direita.

É possível mostrar ainda que se todos os elementos da lista estão inicialmente à distância no máximo c (onde c é uma constante) da posição em que deve ficar quando a lista estiver ordenada, o Método Inserção requer tempo linear.

O Algoritmo Inserção requer o uso de apenas três variáveis escalares. Sendo assim, sua complexidade espacial é O(1). Observe que um elemento da parte esquerda somente é movido para a direita se ele for estritamente maior do que o pivô. Por conta disso, o Algoritmo Inserção é estável.

Na Figura 2.2 exibimos o conteúdo de um vetor após cada iteração do laço mais externo do Algoritmo Inserção. A parte esquerda da lista aparece sombreada.

	0	1	2	3	4	5
Lista original	9	4	5	10	5	8
1ª iteração	4	9	5	10	5	8
2ª iteração	4	5	9	10	5	8
3ª iteração	4	5	9	10	5	8
4ª iteração	4	5	5	9	10	8
5ª iteração	4	5	5	8	9	10

Figura 2.2: Ordenação com o Algoritmo Inserção

5. Seleção

O Método Seleção tem como ponto forte o fato de que ele realiza poucas operações de swap. Seu desempenho, em termos absolutos, costuma ser superior ao do Método Bolha, mas inferior ao do Método Inserção.

Assim como no Método Inserção, nesse método consideramos que a lista está dividida em parte esquerda, já ordenada, e parte direita, em possível desordem. Além disso, os elementos da parte esquerda são todos menores ou iguais aos elementos da parte direita.

Cada iteração consiste em selecionar o menor elemento da parte direita (pivô) e trocá-lo com o primeiro elemento da parte direita. Com isso, a parte esquerda aumenta, pois passa a incluir o pivô, e a parte direita diminui. Note que o pivô é maior que todos os demais elementos da parte esquerda, portanto a parte esquerda continua ordenada. Além disso, o pivô era o menor elemento da parte direita e, portanto, continua sendo verdade que os elementos da parte esquerda são todos menores ou iguais aos elementos da parte direita. Inicialmente, a parte esquerda é vazia. Se a lista possui n elementos, após n – 1 iterações ela estará ordenada.

```
ALGORITMO SELEÇÃO
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 0 ate n - 2
    MIN = i
    PARA j = i + 1 até n - 1
      SE L[j] < L[MIN]
        MIN= j
    TROCA(L[i], L[MIN])
FIM {SELEÇÃO}
```

Algoritmo 2.4: Seleção

A análise do Algoritmo Seleção é semelhante à análise do Algoritmo Bolha. Os dois algoritmos são constituídos de dois laços contados encaixados que realizam as mesmas quantidades de iterações. Dessa forma, se torna evidente que o Algoritmo Seleção realiza (n2 - n)/2 comparações e sua complexidade temporal pertence a $\Theta(n2)$. Assim como no Bolha, não faz sentido falar em melhor nem pior caso. Ambos os algoritmos requerem tempo quadrático, qualquer que seja a lista.

O Algoritmo Seleção necessita de apenas quatro variáveis escalares para fazer o seu trabalho (uma variável auxiliar é usada no procedimento troca). Sendo assim, a complexidade espacial do Seleção é O(1).

Exibimos na Figura 2.3 o conteúdo de um vetor após cada iteração do laço mais externo do Algoritmo Seleção. A parte esquerda da lista aparece sombreada. Por esse exemplo percebemos que o Método Seleção é não-estável.

	0	1	2	3	4	5
Lista original	10	4	5	3	10	2
1ª iteração	2	4	5	3	10	10
2ª iteração	2	3	5	4	10	10
3ª iteração	2	3	4	5	10	10
4ª iteração	2	3	4	5	10	10
5ª iteração	2	3	4	5	10	10

Figura 2.3: Ordenação com o Algoritmo Seleção

6. Shellsort

O Método Shellsort foi proposto por Donald Shell em 1959 e é um exemplo de um algoritmo fácil de descrever e implementar, mas difícil de analisar. Uma explicação precisa para seu surpreendente desempenho é um dos enigmas que desafiam os pesquisadores há décadas.

Para descrever o Shellsort com mais facilidade é conveniente definir o termo h-lista. Dada uma lista L, uma h-Lista de L é uma sublista maximal de L na qual cada elemento está a distância h de seu sucessor. Por exemplo, para h = 3, a lista a seguir contém três h-listas. A primeira h-lista é constituída dos elementos nas posições 0, 3 e 6 do vetor. A segunda h-lista contém os elementos nas posições 1, 4 e 7. A terceira h-lista é constituída dos elementos nas posições 2 e 5.

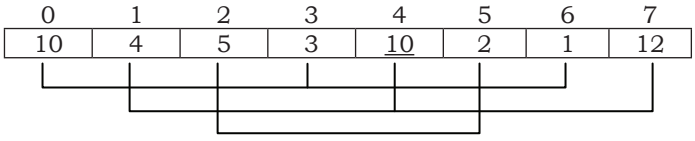


Figura 2.4: Uma lista dividida em h-listas (h = 3)

Em 2001, Marcin Ciura mostrou que utilizar a sequência 1, 4, 10, 23, 57, 132, 301, 701 e 1750 como valores de h produz resultados ainda melhores que utilizar a fórmula proposta por Knuth. A sequência de Ciura é a melhor conhecida atualmente.

No Shellsort, a lista é subdividida em h-listas, as quais são ordenadas com um método de ordenação qualquer. Esse procedimento é repetido para valores decrescentes de h, sendo que o último valor de h tem que ser 1. A lista estará então ordenada.

Observe que o Shellsort permite diversas implementações diferentes. Cabe ao programador decidir qual método de ordenação ele usará para ordenar as h-listas. Uma boa opção é utilizar o método Inserção.

Além disso, o programador também terá decidir quais valores de h ele utilizará. Donald Shell sugeriu usar potências de 2 como valores de h. Por exemplo, para ordenar uma lista de 100 elementos, seriam utilizados os números 64, 32, 16, 8, 4, 2 e 1 como valores de h.

Posteriormente, outro pesquisador, Donald Knuth, mostrou que o desempenho do Shellsort não é bom se os valores que h assume ao longo da execução do método são múltiplos uns dos outros. Ele então sugeriu calcular os valores de h através da seguinte fórmula de recorrência: h = 3 * h + 1. Como o último valor de h tem que ser 1, usando essa fórmula, o valor anterior de h seria 4. Antes de 4, o valor de h seria 13, e assim por diante. Por exemplo, para ordenar uma lista de 100 elementos, seriam utilizados os números 40, 13, 4 e 1 como valores de h. Donald Knuth mostrou empiricamente que o desempenho do Shellsort rivaliza com o desempenho dos melhores métodos de ordenação ao ordenar listas de pequeno e médio porte quando é usada a fórmula sugerida por ele.

```
ALGORITMO SHELLSORT
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  H = 1
  ENQUANTO H < n FAÇA H = 3 * H + 1
  FAÇA
    H = H / 3            // divisão in-
teira
    PARA i = H até n - 1 // Inserção
adaptado para h-listas
      PIVO = L[i]
      j = i - H
      ENQUANTO j ≥ 0 e L[j] > PIVO
        L[j + H] = L[j]
        j= j - H
      L[j + H] = PIVO
  ENQUANTO H > 1
FIM {SHELLSORT}
```

Algoritmo 2.5: Shellsort

Note que os valores de h decrescem praticamente numa progressão geométrica de razão 1/3. Isso implica que a quantidade de iterações do laço mais externo pertence a Θ(logn). No melhor caso, que ocorre quando a lista fornecida já está ordenada, a condição do laço interno nunca é verdadeira. Nesse caso, cada iteração do laço mais externo é feita em tempo linear e, portanto, a complexidade temporal do Algoritmo Shellsort pertence a Θ(nlogn). Não se conhece a exata complexidade do Shellsort no pior caso. Sabe-se, no entanto, que tal complexidade está entre Θ(nlogn) e Θ(n1,5).

O Shellsort é mais um método de ordenação in situ, pois sua complexidade espacial é constante. Sobre a estabilidade, o exemplo da Figura 2.5 deixa claro que ao ordenar as h-listas o Shellsort pode inverter a ordem existente entre os elementos repetidos, mesmo que seja usado um método de ordenação estável (como o Inserção) para ordenar as h-listas. Consequentemente, o Shellsort é não-estável.

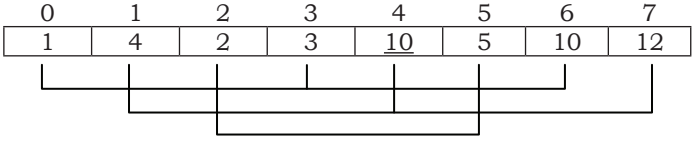


Figura 2.5: h-listas da Figura 2.4 em ordem crescente

7. Mergesort

Esse método, proposto por John Von Neumann em 1945, é baseado numa estratégia de resolução de problemas conhecida como divisão e conquista. Essa técnica consiste, basicamente, em decompor a instância a ser resolvida em instâncias menores do mesmo tipo de problema, resolver tais instâncias (em geral, recursivamente) e por fim utilizar as soluções parciais para obter uma solução da instância original.

Naturalmente, nem todo problema pode ser resolvido através de divisão e conquista. Para que seja viável aplicar essa técnica a um problema ele deve possuir duas propriedades estruturais. O problema deve ser decomponível, ou seja, deve ser possível decompor qualquer instância não trivial do problema em instâncias menores do mesmo tipo de problema.

Vale salientar que a técnica de divisão e conquista costuma apresentar desempenho melhor quando as instâncias obtidas com a decomposição têm aproximadamente o mesmo tamanho.

Uma instância trivial é uma instância que não pode ser decomposta, mas cuja solução é muito simples de ser calculada. Por exemplo, ordenar uma lista constituída de apenas um elemento é uma instância trivial do problema da ordenação.

Além disso, deve ser sempre possível utilizar as soluções obtidas com a resolução das instâncias menores para chegar a uma solução da instância original. Como veremos mais adiante, o problema da ordenação tem essas duas propriedades estruturais e, portanto, pode ser resolvido por divisão e conquista.

No Mergesort dividimos a lista em duas metades. Essas metades são ordenadas recursivamente (usando o próprio Mergesort) e depois são intercaladas. Embora seja possível descrever o Mergesort sem fazer uso de recursividade, a versão recursiva desse é algoritmo é elegante e sucinta, como vemos no Algoritmo 2.6.

```
ALGORITMO MERGESORT
  ENTRADA: UM VETOR L E AS POSIÇÕES INI-
CIO E FIM
  SAÍDA: O VETOR L EM ORDEM CRESCENTE DA
POSIÇÃO INICIO ATÉ
         A POSIÇÃO FIM

  SE inicio < fim
    meio = (inicio + fim)/2          //
divisão inteira
    SE inicio < meio
```

```
        MERGESORT(L, inicio, meio)
    SE meio + 1 < fim
        MERGESORT(L, meio + 1, fim)
    MERGE(L, inicio, meio, fim)
FIM {MERGESORT}
```

Algoritmo 2.6: Mergesort

Observe que a intercalação do intervalo a ser ordenado é feita com o Procedimento Merge descrito no Algoritmo 2.7. Tal procedimento é a parte principal do Mergesort e é nele onde os elementos são movimentados para as posições adequadas.

Nesse procedimento fazemos uso das variáveis i e j para percorrer a metade esquerda e a metade direita, respectivamente. Em cada iteração comparamos o elemento na posição i com o elemento na posição j. O menor deles é copiado para um vetor auxiliar. Esse procedimento é repetido até que uma das duas metades tenha sido totalmente copiada para o vetor auxiliar. Se alguns elementos da metade esquerda não tiverem sido copiados para o vetor auxiliar, eles devem ser copiados para o final do intervalo. Finalmente, copiamos os elementos do vetor auxiliar de volta para o intervalo.

```
PROCEDIMENTO MERGE
  ENTRADA: UM VETOR L E AS POSIÇÕES INI-
CIO, MEIO E FIM
           (L TEM QUE ESTAR EM ORDEM
CRESCENTE DA POSIÇÃO
           INICIO ATÉ MEIO E DA POSIÇÃO
MEIO + 1 ATÉ FIM)
  SAÍDA: O VETOR L EM ORDEM CRESCENTE DA
POSIÇÃO INICIO ATÉ
         A POSIÇÃO FIM

i = inicio, j = meio + 1, k = 0
ENQUANTO i ≤ meio e j ≤ fim
  SE L[i] ≤ L[j]
    AUX[k] = L[i], i = i + 1
  SENAO
    AUX[k] = L[j], j = j + 1
  k = k + 1
SE i ≤ meio
  PARA j = meio até i(passo -1)
```

```
                    L[fim-meio+j] = L[j]
                PARA i=0 ate k-1
                    L[i+inicio] = AUX[i]
                FIM {MERGE}
```

Algoritmo 2.7: Procedimento Merge

Uma análise cuidadosa desse procedimento nos leva a concluir que ele requer tempo linearmente proporcional ao tamanho do intervalo. Vamos chamar de n o tamanho do intervalo, ou seja, n = fim – inicio + 1. Observe que a condição do primeiro laço somente é verdadeira se i meio e j fim. Logo, para que o laço continue e ser executado é preciso termos i + j meio + fim.

Observe que inicialmente i + j = inicio + meio + 1. Como a cada iteração do primeiro laço ou i ou j é incrementado, após n – 1 iterações teremos:

i + j = inicio + meio + 1 + n – 1

i + j = inicio + meio + n

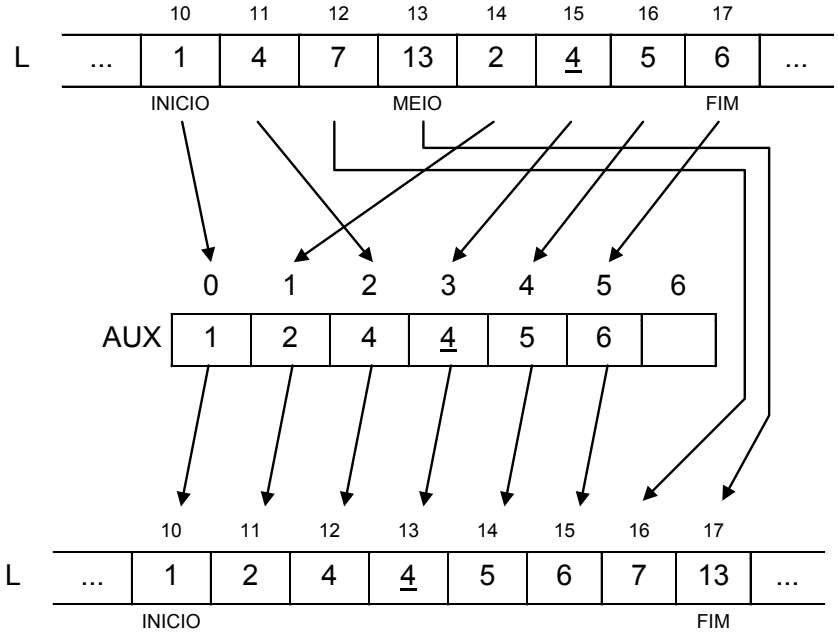
i + j = inicio + meio + fim – inicio + 1

i + j = meio + fim + 1

Dessa forma, após no máximo n – 1 iterações a condição do primeiro laço será violada. Claramente, o segundo laço, se executado, terá no máximo n/2 iterações. Note que a cada iteração do primeiro laço a variável k é incrementada. Dessa forma, a quantidade de iterações do primeiro laço será k. Como k é no máximo n – 1 e o último laço terá k iterações, concluímos que o Procedimento Merge requer tempo linear. Observe ainda que o Procedimento Merge faz uso do vetor AUX e, portanto, requer espaço linear.

Além disso, note que na comparação contida no primeiro laço, se L[i] e L[j] forem iguais, é copiado L[i] para o vetor auxiliar e isso remover preserva a ordem relativa entre os elementos repetidos. Esse cuidado na implementação faz com que o Procedimento Merge faça sempre intercalações estáveis. Como consequência, o Mergesort também é estável.

A Figura 2.6 ilustra o funcionamento do Procedimento Merge. Observe que o intervalo de L que vai da posição INICIO até MEIO e o intervalo que vai de MEIO + 1 até FIM já estão em ordem crescente.



Esse método é descrito e exemplificado no excelente livro Algoritmos – Teoria e Prática, de Cormen et al.

Figura 2.6: Exemplo de intercalação

Como o Algoritmo **Mergesort** foi descrito de maneira recursiva, podemos facilmente fazer sua análise utilizando fórmulas de recorrência. Para isso, vamos denotar por T(n) o tempo requerido para ordenar um intervalo de tamanho n. Sendo assim o tempo necessário para ordenar um intervalo de tamanho n/2 será T(n/2). Como o Procedimento Merge requer tempo linear, podemos supor que ele requer tempo cn (onde c é uma constante positiva). Temos então a seguinte fórmula de recorrência para T:

$$T(n) = 2T(n/2) + cn$$
$$T(1) = c$$

Podemos resolver essa fórmula facilmente utilizando o **método da iteração** supondo n = 2k. A partir da recorrência original, podemos obter as seguintes recorrências:

$$T(n) = 2T(n/2) + cn$$
$$2T(n/2) = 4T(n/4) + cn$$
$$4T(n/4) = 8T(n/8) + cn$$
...

$$2k\text{-}1T(n/2k\text{-}1) = 2kT(n/2k) + cn$$
$$2kT(n) = 2kc$$

Somando tais fórmulas chegamos a $T(n) = cnk + 2kc$. Lembre-se de que supomos $n = 2k$, logo, $k = \log 2n$. Sendo assim, $T(n) = cn\log 2n + cn$. Concluímos que a complexidade temporal do Algoritmo Mergesort pertence a $\Theta(n \log n)$.

Vamos agora denotar por E(n) o espaço extra de memória requerido para ordenar um intervalo de tamanho n. Como o Procedimento Merge requer espaço linear e a memória utilizada numa chamada recursiva pode ser reaproveitada na chamada recursiva seguinte, temos então a seguinte fórmula de recorrência para E:

$$E(n) = \max(E(n/2), cn)$$
$$E(1) = c$$

Novamente podemos resolver essa fórmula utilizando o método da iteração, obtendo $E(n) = cn$. Concluímos que a complexidade espacial do Algoritmo Mergesort pertence a $\Theta(n)$. Diferente de todos os métodos que estudamos anteriormente, o Mergesort não é in situ.

8. Quicksort

Esse método, desenvolvido em 1960 por C. A. R. Hoare e publicado em 1962, é talvez o mais utilizado de todos os métodos de ordenação. Isso ocorre porque quase sempre o Quicksort é significativamente mais rápido do que todos os demais métodos de ordenação baseados em comparação. Além disso, suas características fazem com que ele, assim como o Mergesort, possa ser facilmente paralelizado. Ele também pode ser adaptado para realizar ordenação externa (Quicksort Externo).

Um aspecto que torna o Quicksort muito interessante sob o ponto de visto teórico é o fato de que, embora em geral ele seja extremamente rápido, existem casos (muito raros, é verdade) em que seu desempenho é bem ruim, chegando a ser pior do que o de métodos de ordenação considerados inferiores como os métodos Bolha, Inserção e Seleção.

Assim como o Mergesort, o Quicksort também é baseado na estratégia de divisão e conquista. Ocorre que, enquanto no Mergesort a fase de divisão é trivial e a fase de conquista é trabalhosa, no Quicksort, com veremos a seguir, acontece o contrário, a fase de divisão é trabalhosa e a fase de conquista é trivial.

Nesse método, a lista é dividida em parte esquerda e parte direita, sendo que os elementos da parte esquerda são todos menores que os elementos da parte direita, conforme mostrado na Figura 2.7. Essa fase do método é chamada de partição.

Em seguida, as duas partes são ordenadas recursivamente (usando o próprio Quicksort). A lista estará então ordenada.

Uma estratégia para fazer a partição é escolher um valor como pivô e então colocar na parte esquerda os elementos menores ou iguais ao pivô e na parte direita os elementos maiores que o pivô.

Embora existam diversas maneiras de escolher o pivô, algumas das quais serão apresentadas nessa seção, adotaremos a princípio uma estratégia extremamente simples, mas que costuma apresentar resultados práticos muito bons. Além disso, tal estratégia facilita perceber os casos em que o procedimento de partição não divide a lista de maneira equilibrada. Dessa forma, torna-se mais fácil caracterizar e analisar o pior caso do Método Quicksort.

Utilizaremos como pivô o primeiro elemento da lista. Nesse caso, o pivô deve ser colocado entre as duas partes. O Procedimento Partição, que implementa tal estratégia, é descrito no Algoritmo 2.8.

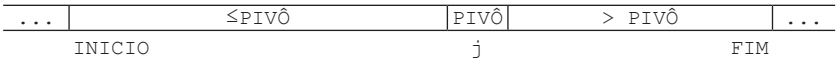


Figura 2.7: Partição

```
PROCEDIMENTO PARTIÇÃO
  ENTRADA: UM VETOR L E AS POSIÇÕES INI-
CIO E FIM
  SAÍDA:j, tal que L[INICIO]..L[j-1] ≤
L[j] e
        L[j+1]..L[FIM] > L[j]
     // j é a posição onde será colocado
o pivô, como
     // ilustrado na figura abaixo
  PIVO = L[INICIO], i = INICIO + 1, j =
FIM
  ENQUANTO i ≤ j
    ENQUANTO i ≤ j e L[i] ≤ PIVO
      i = i + 1
    ENQUANTO L[j] > PIVO
      j = j - 1
    SE i ≤ j
```

```
                    TROCA(L[i],L[j])
                    i = i + 1, j = j - 1
                TROCA(L[INICIO], L[j])
                DEVOLVA j
            FIM {PARTIÇÃO}
```

Algoritmo 2.8: Procedimento Partição

No Procedimento Partição a variável i avança até encontrar um elemento maior do que o pivô e a variável j recua até encontrar um elemento menor ou igual ao pivô. É feita então a troca do elemento que está na posição i com o elemento que está na posição j. Esse processo é repetido até termos i > j. Para finalizar, o pivô é colocado entre as duas partes fazendo troca(L[INICIO], L[j]). A posição j é devolvida ao chamador do procedimento.

A região crítica do Procedimento Partição é constituída dos dois laços mais internos, sendo n o tamanho do intervalo. Observe que inicialmente j – i = n – 2. Note ainda que a cada iteração de um dos laços internos ou i é incrementado ou j é decrementado. Isso significa que a cada iteração de um desses laços a expressão j – i é decrementada de uma unidade. Consequentemente, após no máximo n – 1 iterações desses laços teremos j – i = -1 o que implica que j será menor do que i. O laço será então finalizado. Concluímos que o Procedimento Partição requer tempo linear. Claramente, esse procedimento tem complexidade espacial constante.

Na Figura 2.8 é ilustrada a partição de uma lista. Observe que nesse exemplo que a partição ficou equilibrada, pois quatro elementos ficaram à esquerda do pivô e três elementos ficaram à direita do pivô.

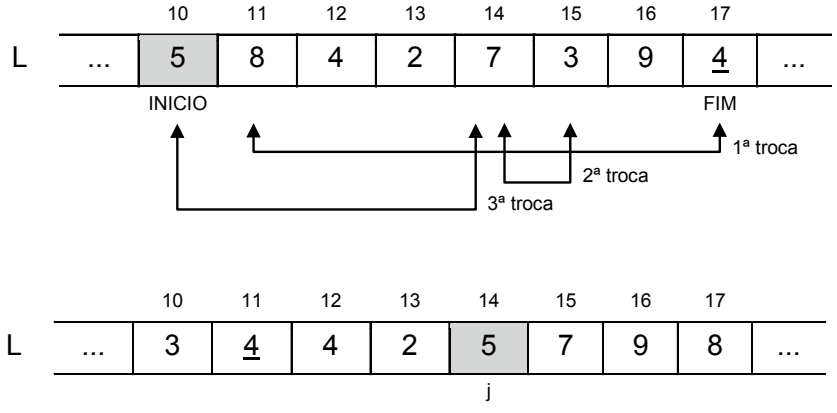


Figura 2.8: Exemplo de partição equilibrada

Nesse exemplo, percebemos que o Procedimento Partição pode inverter a ordem existente entre os elementos repetidos. Consequentemente, o Quicksort é não-estável.

Se o elemento escolhido como pivô for o maior elemento do intervalo, teremos uma partição desequilibrada, como ilustrado na Figura 2.9. Note que apenas uma troca será executada.

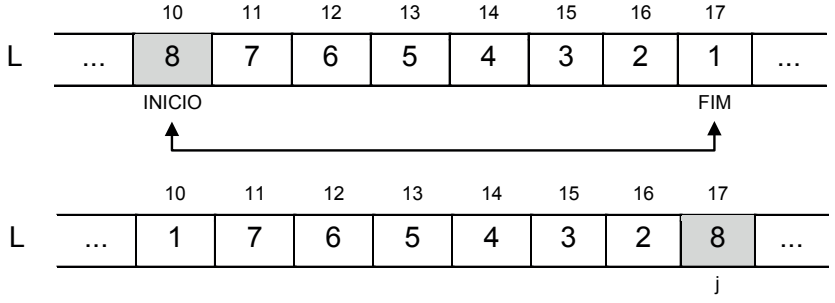


Figura 2.9: Exemplo de partição desequilibrada

É fácil perceber que se o menor elemento do intervalo for escolhido como pivô nenhuma troca será efetuada. Teremos outra partição desequilibrada. Em ambos os casos uma das metades produzidas com a partição ficará vazia e a outra terá todos os elementos do intervalo exceto o pivô. Esse fato terá um impacto importante sobre o desempenho do Quicksort como veremos mais adiante.

Descrevemos a seguir uma versão recursiva do Quicksort.

```
ALGORITMO QUICKSORT
  ENTRADA: UM VETOR L E AS POSIÇÕES INICIO E FIM
  SAÍDA: O VETOR L EM ORDEM CRESCENTE DA POSIÇÃO
INICIO ATÉ
          A POSIÇÃO FIM

  SE INICIO < FIM
    j = PARTICAO(L, INICIO, FIM)
    SE INICIO < j - 1
      QUICKSORT(L, INICIO, j - 1)
    SE j + 1 < FIM
      QUICKSORT(L, j + 1, FIM)
FIM {QUICKSORT}
```

Algoritmo 2.9: Quicksort

Para analisar o Algoritmo Quicksort vamos denotar por T(n) o tempo requerido para ordenar um intervalo de tamanho n e por E(n) o espaço extra requerido para ordenar um intervalo de tamanho n. No pior caso, que ocorre quando todas as escolhas do pivô recaem sempre sobre um elemento extremo (o maior ou o menor) do intervalo, a complexidade temporal do Quicksort é dada por:

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

Essa fórmula pode ser facilmente resolvida resultando em $T(n) = (n2 - n)/2$. A complexidade espacial do Quicksort no pior caso é dada por:

$$E(n) = E(n - 1) + c$$
$$E(1) = c$$

Resolvendo essa fórmula obtemos $E(n) = cn$. Concluímos que no pior caso o Algoritmo Quicksort requer tempo $\Theta(n2)$ e espaço $\Theta(n)$. Para nossa surpresa, essa complexidade temporal é tão ruim quanto as dos métodos Bolha, Inserção e Seleção e essa complexidade espacial é tão ruim quanto a do Mergesort. Some-se a isso o fato de que o Quicksort é não-estável e então concluímos que no pior caso o Quicksort é um dos piores métodos de ordenação.

É fácil perceber que com o critério de escolha do pivô que adotamos, se a lista estiver inversamente ordenada, todos os pivôs escolhidos serão elementos extremos de seus intervalos e, portanto, recairemos no pior caso do Quicksort. Para nossa surpresa, se a lista já estiver ordenada, todos os pivôs escolhidos também serão elementos extremos de seus intervalos e novamente recairemos no pior caso do Quicksort. Felizmente, a ocorrência do pior caso é extremamente rara.

No melhor caso, que ocorre quando as escolhas do pivô recaem sempre sobre a **mediana** do intervalo, a complexidade temporal do Quicksort é dada por:

$$T(n) = 2T(n/2) + cn$$
$$T(1) = c$$

Resolvemos essa fórmula ao discutir o Mergesort e sabemos que $T(n) \in \Theta(n\log n)$. A complexidade espacial do Quicksort é dada por:

$E(n) = E(n/2) + c$

$E(1) = c$

Resolvendo essa fórmula concluímos que $E(n) \in \Theta(\log n)$. Sendo assim, no melhor caso a complexidade temporal do Quicksort pertence a $\Theta(n \log n)$ e a complexidade espacial pertence a $\Theta(\log n)$. Essas também são as complexidades do Quicksort no caso médio.

Podemos utilizar estratégias mais elaboradas para escolha do pivô, com o objetivo de produzir partições mais equilibradas. Eis algumas ideias:

Utilizar a média aritmética do intervalo.

Utilizar a mediana do intervalo.

Escolher aleatoriamente um elemento do intervalo.

Nos dois primeiros casos precisaremos de tempo linear para escolher o **pivô**, mas isso não afetará a complexidade temporal assintótica da partição que é linear. No entanto, em termos absolutos, o tempo requerido pela partição aumentará. Além disso, o pivô poderá não ser um elemento do intervalo e isso exigirá algumas alterações adicionais do Procedimento Partição.

No livro Algoritmos – Teoria e Prática é descrito um método para calcular a mediana em tempo linear.

No último caso temos a versão conhecida como Quicksort probabilístico. A principal vantagem dessa versão é o fato de tornar impossível garantir que o Quicksort vai requerer tempo quadrático para uma determinada lista. Como vimos anteriormente, ao escolher deterministicamente o pivô, é possível que alguém disposto a degradar a performance Quicksort construa listas para as quais seu desempenho seja quadrático. Isso não é mais possível na versão probabilística do algoritmo.

9. Heaps Binários

Um heap é uma coleção de dados parcialmente ordenados. Num heap crescente é válida a seguinte propriedade: "o pai é menor ou igual aos seus filhos". Num heap decrescente, temos a propriedade análoga: "o pai é maior ou igual aos seus filhos". Existem diversos tipos de heaps: binários, binomiais, de Fibonacci etc. Abordaremos apenas heaps binários.

Um heap binário se caracteriza pelo fato de que cada elemento possui no máximo dois filhos. Podemos implementar heaps binários usando uma árvore binária, conforme mostra a Figura 2.10. Tal árvore terá todos os níveis completos, com exceção do último nível, que pode estar incompleto. Além disso, os níveis são preenchidos da esquerda para a direita.

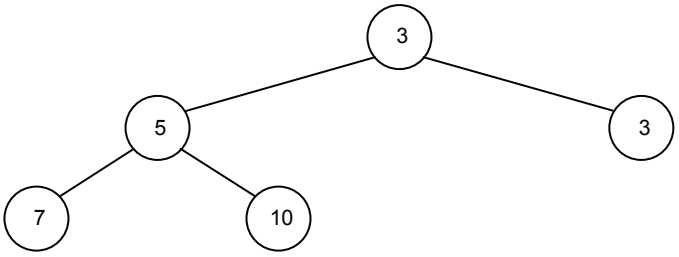


Figura 2.10: Exemplo de heap binário crescente armazenado numa árvore

Um heap binário também pode ser implementado através de um vetor. Nesse caso, os filhos da posição i são as posições 2 * i e 2 * i + 1 (A posição zero do vetor não é utilizada). Discutiremos em detalhes como fazer inserção e como remover o menor elemento de um heap binário crescente implementado num vetor. A Figura 2.11 mostra a implementação de um heap binário em um vetor.

1	2	3	4	5	6
3	5	3	7	10	

Figura 2.11: Exemplo de heap binário crescente armazenado num vetor

Observe que num heap binário crescente implementado numa árvore, o menor valor estará sempre na raiz da árvore. Se o heap binário crescente for implementado num vetor, o menor valor estará sempre na primeira posição do vetor. Dessa forma, encontrar o menor valor contido num heap binário crescente é uma operação que pode ser feita em tempo constante. Analogamente, o maior elemento num heap binário decrescente estará sempre no seu início.

Para implementar um heap binário usando um vetor, podemos usar uma estrutura com os seguintes campos:

- vetor: armazena os dados
- tamanho: indica o tamanho do vetor
- ultima: indica a última posição usada do vetor

Para inserir um valor x num heap binário devemos inseri-lo após o último elemento e depois comparar com seu pai, trocando-os de posição se houver necessidade. Tal processo é repetido até que x seja maior ou igual ao seu pai ou até x atingir a posição 1 do vetor. O Algoritmo 2.10 implementa a ideia descrita nesse parágrafo.

```
ALGORITMO INSERE_HBC
  ENTRADA: UM HEAP BINÁRIO CRESCENTE H E UM VALOR X
  SAÍDA: SE HOUVER ESPAÇO DISPONIVEL, INSERE X EM
```

```
H;
          CASO CONTRÁRIO, DEVOLVE UM ERRO.

SE H.ultima = H.tamanho                    //HEAP
OVERFLOW
  devolva ERRO
H.ultima = H.ultima + 1
i = H.ultima
ENQUANTO i > 1 e H.vetor[i/2] > x          // DIVI-
SÃO INTEIRA
        H.vetor[i] = H.vetor[i/2]
        i = i/2
H.vetor[i] = x
FIM {INSERE_HBC}
```

Algoritmo 2.10: Insere_HBC

$\lfloor x \rfloor$ (lê-se "chão de x") é o maior inteiro que é menor ou igual a x.

Claramente, a região crítica do Algoritmo Insere_HBC é o laço. Dessa forma, o tempo requerido pelo algoritmo é determinado pela quantidade de iterações do laço. Seja n = H.ultima. Note que inicialmente i = n e, portanto, i possui $\lfloor \log_2 n \rfloor + 1$ bits **significativos**. A cada iteração do laço, a variável i perde um bit significativo, pois o valor de i é atualizado fazendo-se i = i/2. Após $\lfloor \log_2 n \rfloor$ iterações, i terá apenas um bit significativo. Sendo assim, teremos i 1 e o laço será finalizado. Assim, a quantidade máxima de iterações do laço é $\lfloor \log_2 n \rfloor$. Concluímos que a complexidade temporal do Algoritmo Insere_HBC pertence a O(log n).

A Figura 2.12 ilustra a inserção do valor 4 num heap binário crescente. Observe que o valor 7 e depois o valor 5 precisam ser movidos para que o valor 4 possa ser colocado na posição 2 do vetor. Note ainda que a inserção pode inverter a ordem entre os elementos repetidos.

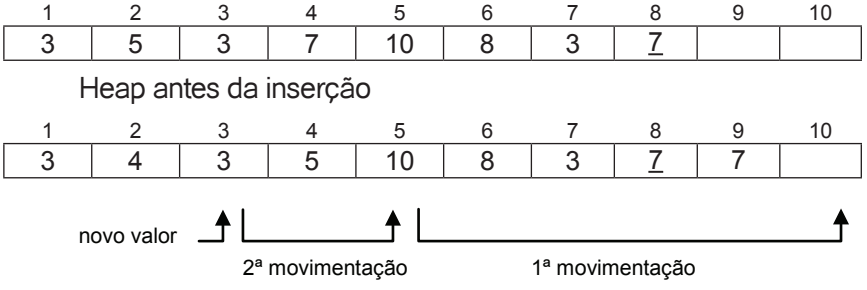


Figura 2.12: Exemplo de inserção num heap binário crescente

Embora seja possível remover um elemento qualquer do heap, vamos discutir apenas a remoção do menor elemento. Como vimos anteriormente, o menor elemento estará sempre na primeira posição do heap.

Para remover o menor elemento de um heap binário crescente implementado num vetor, movemos o último valor contido no heap (pivô) para a posição 1 do vetor. Em seguida, comparamos o pivô com seus filhos, trocando-o com o menor de seus filhos se houver necessidade. Esse procedimento é repetido até que o pivô seja menor ou igual aos seus filhos ou até o pivô atingir uma posição que não tenha filhos.

```
ALGORITMO REMOVE_MENOR
  ENTRADA: UM HEAP BINÁRIO CRESCENTE H
  SAÍDA: SE H NÃO ESTIVER VAZIO, REMOVE E DEVOLVE
O MENOR
          ELEMENTO DE H; CASO CONTRÁRIO; DEVOLVE
UM ERRO

  SE H.ultima = 0            //HEAP UNDERFLOW
    devolva ERRO e PARE
  valor= H.vetor[1]
  H.vetor[1] = H.vetor[H.ultima]
  H.ultima = H.ultima - 1
  i = 1
  ENQUANTO (2*i ≤ H.ultima e H.vetor[i] >
H.vetor[2*i]) ou
            (2*i < H.ultima e H.vetor[i] >
H.vetor[2*i+1])
    menor = 2*i
    SE 2*i < H.ultima e H.vetor[2*i+1] ≤
H.vetor[2*i]
      menor= menor+1
    TROCA(H.vetor[i], H.vetor[menor])
    i = menor
DEVOLVA valor
FIM {REMOVE_MENOR}
```

Algoritmo 2.11: Remove_Menor

A Figura 2.13 mostra a remoção do menor elemento de um heap binário crescente. Observe que após mover o valor 10 para a primeira posição do vetor, precisamos trocá-lo com o valor 3 e depois com o valor 6 para restaurar a ordenação parcial do heap. Note ainda que a remoção do menor elemento pode inverter a ordem entre elementos repetidos.

Figura 2.13: Remoção do menor elemento num heap binário crescente

Seja n = h.ultima. Observe que h.ultima / 2 possui $\lfloor \log_2 n \rfloor$ bits significativos. Inicialmente, a variável i possui apenas um bit significativo. A cada iteração do laço, i ganha um bit significativo. Após $\lfloor \log_2 n \rfloor$ iterações, i terá $\lfloor \log_2 n \rfloor$ + 1 bits significativos e, portanto, teremos i > h.ultima/2. Consequentemente, o laço será finalizado. Concluímos que o Algoritmo Remove_Menor requer tempo O(log n).

10. Heapsort

Podemos usar heaps binários para ordenar com bastante eficiência e elegância. O Método Heapsort é baseado no uso de heap binário. Inicialmente, inserimos os elementos da lista num heap binário crescente. Em seguida, fazemos sucessivas remoções do menor elemento do heap, colocando os elementos removidos do heap de volta na lista. A lista estará então em ordem crescente.

```
ALGORITMO HEAP SORT
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

inicialize um HBC H com n posições
PARA i = 0 até n - 1
  INSERE_HBC(H, L[i])
PARA i = 0 até n - 1
  L[i] = REMOVE_MENOR(H)
FIM {HEAP SORT}
```

Algoritmo 2.12: Heapsort

Na seção anterior mostramos que a inserção e a remoção do menor elemento requer tempo logarítmico no tamanho do heap. Sendo assim, a complexidade temporal do Heapsort pertence a $\Theta(n \log n)$.

Como vimos anteriormente, a inserção e a remoção do menor elemento do heap não preservam a ordem relativa existente entre os elementos repetidos. Como consequência desse fato, o Heapsort é não-estável.

Finalmente, a complexidade espacial do Algoritmo Heapsort é $\Theta(n)$. No entanto, podemos facilmente adaptar o Heapsort para fazer ordenação in situ. Para isso, devemos usar o próprio espaço do vetor como heap. Nesse caso sua complexidade espacial do Heapsort será O(1).

O sete métodos que descrevemos neste capítulo até agora utilizam como operação fundamental a comparação de elementos da lista. É possível mostrar que todo método de ordenação baseado na operação de comparação requer tempo $\Omega(n \log n)$. No entanto existem métodos de ordenação que, sob certas condições, ordenam em tempo linear. Naturalmente, tais métodos não são baseados em comparação. Discutiremos a seguir três desses métodos.

11. Countingsort

O leitor interessado poderá encontrar uma prova desse fato no livro Algoritmos – Teoria e Prática. (Comen et al., 2002).

Esse método foi proposto por Harold H. Seward em 1954 e faz a ordenação usando a ideia de contagem. Uma restrição desse método é que ele somente ordena listas de números naturais.

Essa não é uma restrição muito severa, pois podemos converter diversos tipos de listas em listas de números naturais. Por exemplo, uma lista de números inteiros onde o menor valor é -x pode ser convertida numa lista de números naturais somando x a cada elemento da lista. Uma lista de números racionais pode ser convertida numa lista de números inteiros multiplicando-se os elementos da lista por uma constante apropriada.

De fato, sempre que existir uma bijeção entre o tipo dos dados da lista e um subconjunto dos naturais é possível converter a lista em uma lista de números naturais. Infelizmente, algumas dessas conversões podem ter um impacto negativo sobre o desempenho do método.

Seja n o tamanho da lista e k o maior valor contido na lista. Fazemos uso de um vetor C com k + 1 posições. No primeiro laço do procedimento Countingsort, descrito no Algoritmo 2.13, o vetor C é inicializado com zeros.

Ω denota complexidade no melhor caso.

No laço seguinte, contamos quantas vezes cada um dos elementos de 0 a k aparece na lista. Para fazer isso, percorremos a lista e para cada valor i contido na lista incrementamos de uma unidade a posição i do vetor C. Dessa forma, ao final desse laço cada posição i de C indica quantas vezes o valor de i ocorre na lista.

Em seguida, fazemos a soma acumulada do vetor C. Percorremos novamente o vetor C, dessa vez a partir da segunda posição, somando cada posição i do vetor c com a posição anterior. O resultado da soma é guardado na própria posição i. Após realizar a soma acumulada, cada posição i do vetor C indicará quantos elementos da lista são menores ou iguais a i. Obviamente, se i ocorre na lista e C[i] = j, quando a lista estiver ordenada o valor i deverá ocupar a j-ésima posição da lista.

Percorremos mais uma vez a lista, dessa vez, da direita para a esquerda (para garantir que a ordenação seja estável) e colocamos cada valor i contido na lista na posição c[i] – 1 de um vetor auxiliar. Para evitar que múltiplas cópias de um valor contido na lista sejam colocadas numa mesma posição do vetor auxiliar, após colocar um valor i no vetor auxiliar, decrementamos de uma unidade a posição i de C. No último laço, copiamos o vetor auxiliar para a lista.

```
ALGORITMO COUNTING SORT
  ENTRADA: UM VETOR L CONTENDO N NÚMEROS NATURAIS
NO
           INTERVALO DE 0 A K
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 0 até k                  // INICIALIZA-
CAO DE C
    c[i]=0
  PARA i = 0 até n - 1              // CONTAGEM
    c[L[i]]= c[L[i]] +1
  PARA i = 1 até k                  // SOMA ACUMU-
LADA
    c[i]= c[i] + c[i-1]
  PARA i= n - 1 até 0 (passo -1)    // ORDENACAO
    c[L[i]] = c[L[i]] - 1
    aux[c[L[i]]]= L[i]
  PARA i = 0 até n - 1              // COPIANDO
AUX PARA L
    L[i]= aux[i]
FIM {COUNTING SORT}
```

Algoritmo 2.13: Countingsort

O Algoritmo Countingsort é constituído de cinco laços independentes. Três desses laços gastam tempo $\Theta(n)$ e os outros dois gastam tempo $\Theta(k)$. Além disso, vetor C tem k + 1 posições e o vetor aux tem n posições. Concluímos que a complexidade temporal e a complexidade espacial do Countingsort pertencem a $\Theta(n + k)$. Para a classe de listas onde k não é muito maior do que n, mais precisamente, se $k \in \Theta(n)$, então tais complexidades pertencem a $\Theta(n)$. O cuidado que tivemos ao percorrer a lista do final para o início no quarto laço garante a estabilidade de Countingsort .

Na Figura 2.14 exibimos o conteúdo do vetor C ao final de cada laço do Algoritmo Countingsort. O conteúdo do vetor aux corresponde à lista original em ordem crescente.

	0	1	2	3	4	5	6	7	8
Lista original	8	4	5	3	8	2			
Vetor C	0	1	2	3	4	5	6	7	8
1º laço	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8
2º laço	0	0	1	1	1	1	0	0	2
	0	1	2	3	4	5	6	7	8
3º laço	0	0	1	2	3	4	4	4	6
	0	1	2	3	4	5	6	7	8
4º laço	0	0	0	1	2	3	4	4	4
	0	1	2	3	4	5	6	7	8
5º laço	0	0	0	1	2	3	4	4	4
Vetor aux	0	1	2	3	4	5			
	2	3	4	5	8	8			

Figura 2.14: Ordenação com o Algoritmo Countingsort

12. Bucketsort

Embora esse método permita ordenar listas de números quaisquer, descreveremos uma versão desse método que ordena listas de números não negativos.

Se a lista a ser ordenada tem n elementos, será usado um vetor de ponteiros (bucket) com n posições. Se o maior elemento da lista é k, cada posição do bucket apontará para uma lista encadeada na qual serão inseridos os elementos da lista que pertencem ao intervalo [i * (k + 1) / n , (i + 1) * (k + 1) / n). Observe que o intervalo é fechado à esquerda e aberto à direita.

A ideia do método é dividir o intervalo que vai de 0 até k em n subintervalos de mesmo tamanho. Cada subintervalo estará associado a uma lista ligada que irá conter os elementos da lista que pertencem àquele subintervalo. Por exemplo, se a lista tem oito elementos é o maior deles é 71, teremos oito intervalos: [0, 9), [9, 18), ..., [63, 72). A posição zero do bucket apontará para

uma lista encadeada que irá conter os elementos da lista que são maiores ou iguais a zero e menores que 9, e assim por diante.

Chamaremos o bucket de vetor B. Para construir as listas encadeadas devemos inserir cada valor j contido na lista a ser ordenada na lista encadeada apontada por B[j * n / (k + 1)]. Em seguida, ordenamos as listas encadeadas com um método de ordenação qualquer (de preferência estável). Após isso, a concatenação das listas encadeadas produz a lista original ordenada.

```
ALGORITMO BUCKET SORT
  ENTRADA: UM VETOR L CONTENDO N NÚMEROS NO IN-
TERVALO DE
           0 ATÉ K
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

PARA i = 0 até n - 1                    //INICIA-
LIZACAO
        B[i] = NULO
//CONSTRUINDO AS LISTAS ENCADEADAS
PARA i = n - 1 até 0 (PASSO -1)
  insira L[i] no inicio da lista ligada apontada
por
  B[L[i] * n / (K + 1)]                 // DIVI-
SÃO INTEIRA

//ORDENANDO E CONCATENANDO AS LISTAS ENCADEADAS
j = 0
PARA i = 0 até n - 1
  coloque a lista apontada por B[i] em ordem
crescente
  p = B[i]
  ENQUANTO p ≠ NULO
    L[j] = p.info
    p = p.proximo
    j = j + 1
  remova a lista apontada por B[i]
FIM {BUCKET SORT}
```

Algoritmo 2.14: Bucketsort

Claramente os primeiros dois laços podem ser executados em tempo Θ(n). Note que o tamanho médio de cada lista encadeada será 1. De fato, se os elementos de L estiverem uniformemente distribuídos no intervalo [0, k), é possível mostrar que o tamanho esperado de cada lista encadeada será constante. Nesse caso, independente do método utilizado, a ordenação de cada lista encadeada será feita em tempo esperado constante.

A cópia dos elementos de uma lista encadeada requer tempo amortizado constante. A remoção de uma lista encadeada também é feita em tempo amortizado constante. Consequentemente, o terceiro laço irá requerer tempo esperado constante, e a complexidade temporal esperada do Algoritmo Bucketsort será Θ(n).

Note que no segundo laço percorremos a lista da direita para a esquerda. Com isso, ao construir as listas encadeadas preservamos a ordem relativa existente entre os elementos repetidos. Se o método utilizado para ordenar as listas encadeadas for estável, o Algoritmo Bucketsort também será estável.

O bucket terá n posições. A quantidade de nós nas listas encadeadas será n. Concluímos que a complexidade espacial do Bucketsort pertence a Θ(n). Finalmente, observe que Bucketsort não irá requerer comparações se o método usado como subrotina não for baseado em comparações.

A Figura 2.15 a seguir exemplifica as listas encadeadas construídas pelo Bucketsort após sua ordenação. Claramente, a concatenação das listas encadeadas produz a lista original em ordem.

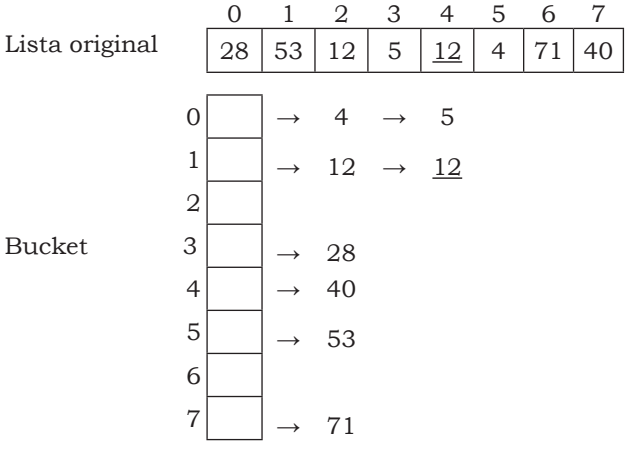


Figura 2.15: Ordenação com o Algoritmo Bucketsort

13. Radixsort

Esse método permite ordenar listas cujos elementos sejam comparáveis dois a dois. Uma versão desse método foi utilizada em 1887 por Herman Hollerith, fundador da Tabulating Machine Company que mais tarde deu origem à IBM.

Em cada iteração do Radixsort, ordenamos a lista por uma de suas posições, começando pela posição menos significante, até chegar à posição mais significante (os elementos da lista devem que ser representados com a mesma quantidade de posições). Cada ordenação tem que ser feita com um método estável.

```
ALGORITMO RADIX SORT
  ENTRADA: UM VETOR L COM N POSIÇÕES CUJOS ELE-
MENTOS
           POSSUEM D SÍMBOLOS
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 1 até d
    Coloque L em ordem crescente pelo i-ésimo
símbolo menos
    significante usando um método de ordenação es-
tável
FIM {RADIX SORT}
```

Algoritmo 2.15: Radixsort

Podemos adaptar o Countingsort e utilizá-lo como subrotina no Radixsort. Observe que os elementos da lista são constituídos de símbolos de algum alfabeto. Por exemplo, se a lista contém números decimais, o alfabeto é formado pelos dígitos do sistema numérico decimal. Note que o maior símbolo encontrado numa posição de um elemento não excede o maior símbolo do alfabeto. Como o tamanho do alfabeto é constante, podemos garantir que a complexidade temporal e a complexidade espacial dessa versão do Countingsort pertencem a $\Theta(n)$.

Outra possibilidade seria adaptar o Bucketsort e utilizá-lo como subrotina. Para isso, o bucket deverá ter uma posição para cada elemento do alfabeto. Não haverá necessidade de ordenar as listas encadeadas visto que cada uma delas contém apenas elementos com o mesmo símbolo na posição que está sendo ordenada. Podemos então garantir que a complexidade temporal dessa versão do Bucketsort será $\Theta(n)$.

Note que tanto o Countingsort quanto o Bucketsort são estáveis. Utilizando qualquer um deles como subrotina, o Radixsort terá complexidade temporal $\Theta(dn)$ e complexidade espacial $\Theta(n)$. Se d for constante a complexidade temporal será $\Theta(n)$. Note que o Radixsort é necessariamente estável, qualquer que seja o método usado como subrotina.

32		500		500		9
500		431		9		32
431		32		431		32
248	→	32	→	32	→	48
9	Ordenando pela unidade	63	Ordenando pela dezena	32	ordenando pela centena	63
32		248		248		248
63		48		48		431
48		9		63		500

Figura 2.16: Ordenação com o Algoritmo Radixsort

Atividades de avaliação

1. Complete a tabela abaixo.

Método	Complexidade temporal			Complexidade espacial	É estável?
	Melhor caso	Pior caso	Caso médio		
Bolha	S (n^2)	S (n^2)	S (n^2)	O(1)	Sim
Bolha com flag					
Inserção					
Seleção					
Shellsort					
Mergesort					
Quicksort					
Heapsort					

2. Caracterize o pior e o melhor caso, em termos assintóticos, de cada um dos métodos que aparecem na tabela acima.

3. Mostre como ficaria a lista (18, 23, 37, 22, 25, 16, 3, 31, 5, 28, 6, 34, 9, 45) após cada iteração do laço mais externo dos algoritmos bolha, inserção, seleção, shellsort e radixsort, descritos neste capítulo.

4. Mostre como ficaria a lista (18, 23, 37, 22, 25, 16, 3, 31, 5, 28, 6, 34, 9, 45) após ser particionada pelo Procedimento Partição apresentado neste capítulo.

5. Dentre os métodos bolha, bolha com flag, inserção, seleção, shellsort, mergesort e quicksort, quais seriam os mais eficientes, em termos de tempo, para colocar listas da forma (n, 1, 2, 3, ..., n – 1) em ordem crescente? Justifique sua resposta.

6. Escreva um algoritmo que implemente o método bolha para ordenar listas encadeadas contendo números. Considere que cada nó da lista possui os campos info, anterior e proximo.

7. Indique quais tipos de listas podem ser ordenadas pelos métodos Countingsort, Bucketsort e Radixsort e que condições precisam ser satisfeitas para que eles tenham complexidade de tempo linear no tamanho da lista.

8. Você deseja ordenar em tempo linear listas de n elementos contendo números inteiros entre 1 e 10n. Qual método você utilizaria ? Justifique sua resposta.

9. Mostre como ficaria o vetor C no final de cada um dos cinco laços do Algoritmo Countingsort ao ordenar a lista (18, 23, 17, 22, 25, 16, 3, 11, 5, 8, 6, 4, 9, 15).

10. Mostre como ficariam as listas encadeadas geradas no método bucket sort ao ordenar a lista (18, 23, 17, 22, 25, 16, 3, 41, 5, 8, 6, 34, 9, 15).

11. Após cada iteração do laço mais externo de um algoritmo de ordenação, a lista (382, 142, 474, 267) ficou como mostrado abaixo. Qual foi o algoritmo utilizado (bolha, inserção, seleção ou radixsort)? Justifique por que nenhum dos outros algoritmos estudados neste capítulo poderia ter produzido as listas abaixo.

(142, 382, 474, 267)

(142, 267, 474, 382)

(142, 267, 382, 474)

12. Escreva um procedimento que receba um heap binário crescente (passado por referência) e uma posição i e então remova o elemento que estiver na posição i do heap.

13. Escreva um procedimento que transforme um vetor num heap binário crescente em tempo linear.

14. Mostre como ficaria um heap binário crescente no qual foram inseridos os valores 18, 23, 17, 22, 25, 16, 3, 41, 5, 8, 6, 34, 9, 15, nesta ordem (seu heap deve ser implementado num vetor de 15 posições). Em seguida, remova o elemento que estiver na posição 3, depois o elemento que estiver na posição 2 e por fim o elemento que estiver na posição 1 e então mostre como ficaria o heap.

Capítulo 3

Ordenação Externa

Objetivos

- Identificar os mecanismos de ordenação externa
- Apresentar os processos de intercalação de arquivos
- Comparar a eficiência entre as diversas distribuições equilibradas
- Definir intercalação de um ou mais caminhos
- Definir intercalação polifásica

Apresentação

No início deste capítulo indicamos que quando os arquivos a serem classificados não podem ser armazenados integralmente na memória principal, é impossível realizar uma ordenação interna através de um dos métodos definidos no capítulo anterior. Mostramos também que, para resolver este problema, o arquivo original deve ser dividido em partes menores, de tamanho tal que a classificação interna seja suportada e que a seguir estas partes, ordenadas, devem ser intercaladas. Apresentamos então as formas eficientes de se processar uma intercalação destacando que os métodos abordados objetivam minimizar o número de leitura e gravação dos registros de cada parte armazenada em disco magnético, chamada área de trabalho. São, por fim, indicados os principais fatores que influem na eficiência de um algoritmo de classificação de forma geral, ou seja, usando a ordenação interna e se necessário a externa.

1. Definição e mecanismos

Quando os arquivos a serem classificados contém muitos registros, de modo que não podem ser contidos integralmente na memória principal, é impossível realizar uma ordenação interna através de um dos métodos definidos no capítulo anterior.

Para resolver este problema, o arquivo original deve ser dividido em partes menores de tamanho tal que a classificação interna seja suportada. Uma vez que as partes ou séries já estejam ordenadas elas são intercaladas sucessivamente até que todas estejam presentes num arquivo final ordenado.

Neste capítulo apresentaremos as formas eficientes de se processar uma intercalação. Os métodos abordados objetivam minimizar o número de leitura e gravação dos registros de cada parte armazenada em disco magnético, chamada área de trabalho. Os registros estão organizados nesta área de forma sequencial.

Os fatores que influem na eficiência de um algoritmo de classificação geral são os seguintes:

- Número de registros e tamanho de cada um deles;
- Uso ou não da memória auxiliar (necessita ou não da ordenação externa);
- Tipos e meios de armazenamento utilizados;
- Grau de classificação já existente e
- Periodicidade de atualização do arquivo.

A eficiência de uma classificação externa é medida em termos da quantidade total do tempo necessário para que os dados sejam lidos, ordenados e gravados na área de trabalho; isto corresponde à fase inicial do processo de ordenação.

Na fase seguinte as séries ordenadas são lidas e intercaladas duas a duas, três a três, etc. O número de séries intercaladas em cada passo corresponde ao número de caminhos. Quanto maior for o número de caminhos, mais complexo será o algoritmo de intercalação (merge). Como este processo é utilizado várias vezes para conseguir juntar todas as séries, em geral opta-se por trabalhar com 2 ou 3 caminhos.

Os fatores que influem na eficiência de um algoritmo de classificação externa são os seguintes:

- Número de séries produzidas;
- Tamanho de cada série e se são do mesmo tamanho;
- Forma de distribuição das séries pelos arquivos de trabalho e
- Características dos blocos e das memórias intermediárias utilizadas na fase de classificação

A estimativa do número de séries utilizadas, bem como seu comprimento, é função do número de registros do arquivo; isto determina o número de passadas durante a fase de classificação. É importante que esteja bem definida a sequência na qual as séries devem ser distribuídas pelos arquivos de entrada para a fase de intercalação.

Os tipos básicos de algoritmos para distribuir as séries são **equilibradas**, onde são colocadas um número igual de séries iniciais em cada arquivo e **desequilibradas** em que as séries são dispostas nos arquivos de modo a conseguir um melhor nível de intercalação.

Nas seções seguintes são apresentados os algoritmos de intercalação de dois caminhos (Merge2) e de três caminhos (Merge3).

2. Intercalação de dois caminhos

Para exemplificar como se processa uma intercalação, vamos supor dois arquivos contendo somente os valores das chaves de cada registro.

Considerando Ri como um registro com chave Ki e que o primeiro arquivo (Arq1) possui **4** registros (R1 a R4) com as respectivas chaves K1 a K4 iguais a {3, 7, 9, 11} e o segundo arquivo (Arq2) possui **6** registros (R5 a R10) com chaves K5 a K10 iguais a {1, 5, 8, 9, 12, 13}. Em cada passo a seleção é feita pelo teste das chaves Ki < Kj apontadas pelas setas. Arq3 é o arquivo de saída.

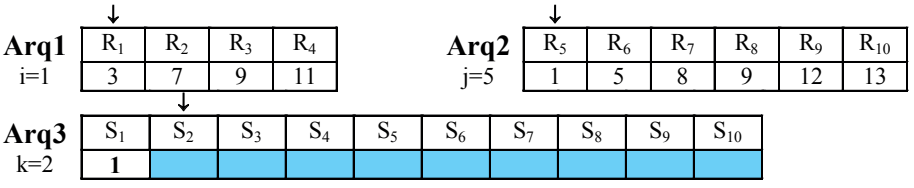


Figura 3.1: Exemplo da intercalação de dois caminhos (Passo-1 do Merge2)

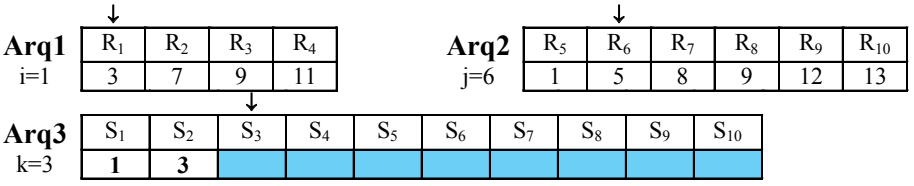


Figura 3.2: Exemplo da intercalação de dois caminhos (Passo-2 do Merge2)

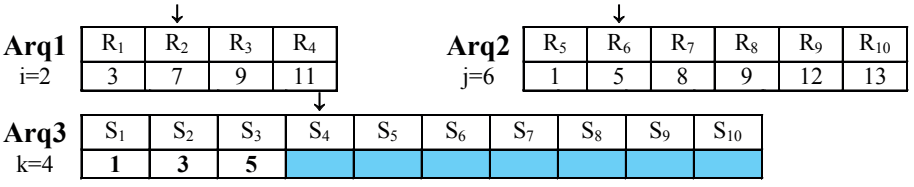


Figura 3.3: Exemplo da intercalação de dois caminhos (Passo-3 do Merge2)

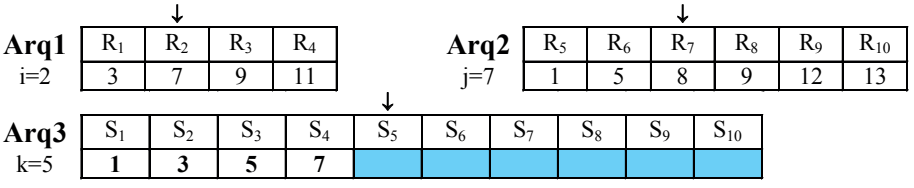


Figura 3.4: Exemplo da intercalação de dois caminhos (Passo-4 do Merge2)

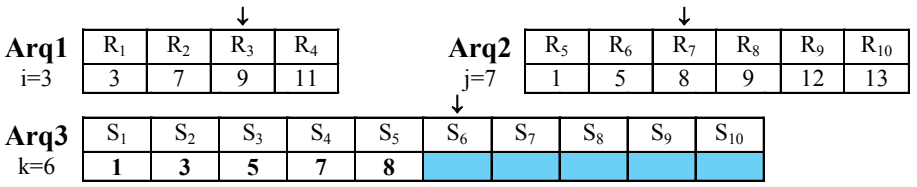


Figura 3.5: Exemplo da intercalação de dois caminhos (Passo-5 do Merge2)

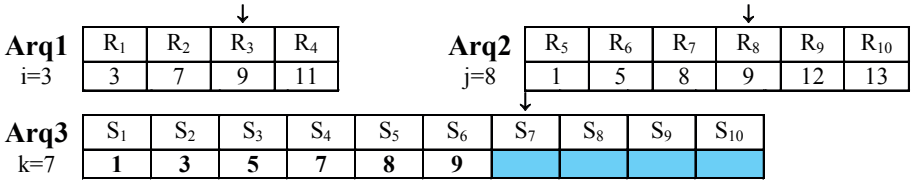


Figura 3.6: Exemplo da intercalação de dois caminhos (Passo-6 do Merge2)

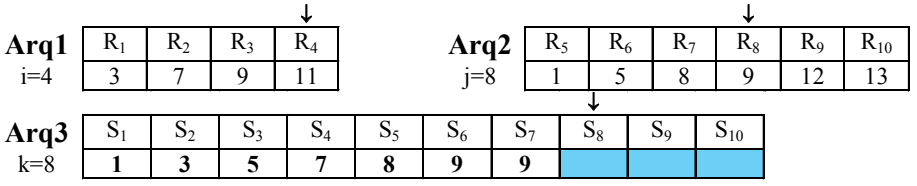


Figura 3.7: Exemplo da intercalação de dois caminhos (Passo-7 do Merge2)

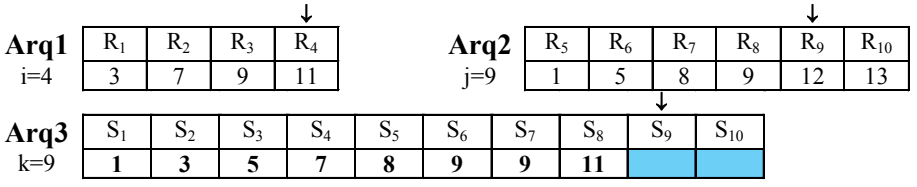


Figura 3.8: Exemplo da intercalação de dois caminhos (Passo-8 do Merge2)

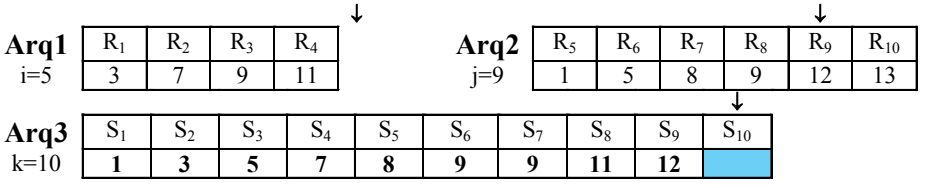


Figura 3.9: Exemplo da intercalação de dois caminhos (Passo-9 do Merge2)

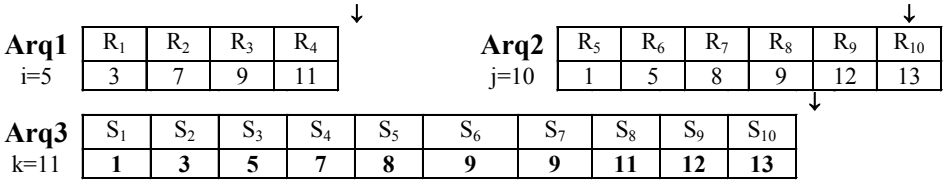


Figura 3.10: Exemplo da intercalação de dois caminhos (Passo-10 do Merge2)

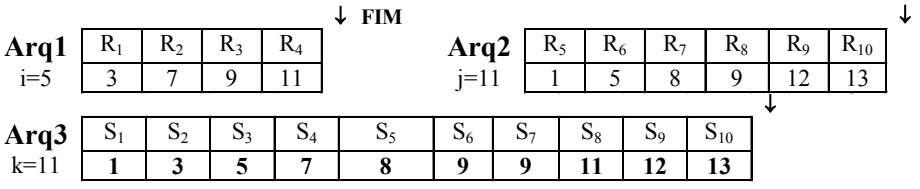


Figura 3.11: Exemplo da intercalação de dois caminhos (Passo-11 do Merge2)

Observe que a partir do passo 9 o ponteiro do arquivo **Arq1** é maior que o número de registros nele contidos, ou seja, significa que foi atingido o **"Fim de Arquivo"** ou "end_of_file". Quando há esta ocorrência em um dos arquivos, o processo final de intercalação consiste somente em copiar os demais registros do outro arquivo que ficou "aberto" para a saída.

No Algoritmo 3.1 que faz a intercalação de dois caminhos, identificado como o procedimento Merge2, utilizaremos as mesmas variáveis usadas no exemplo anterior. Sem perda de generalidade, consideramos que Ri, Rj e Sk representam as chaves dos seus respectivos registros.

Arq1: K1 < K2 < K3 < . . . < Km-1 < Km (ponteiro "i")
Arq2: Km+1 < Km+2 < . . . < Kn-1 < Kn (ponteiro "j")
Arq3: S1 < S2 < S3 < . . . . . .< Sn-1 < Sn (ponteiro "k")

```
PROCEDIMENTO MERGE2 (R, S, m, n)

ENTRADA: INTEIROS m, n
         Registro R com duas séries a intercalar

// R1:m  (primeira série ordenada)
// Rm+1:n (segunda série ordenada)
SAÍDA: Registro S com uma única série ordenada
```

```
// S1:n (série ordenada/intercalada)

// Variáveis auxiliares

// K: vetor [1:n] de inteiros
     // i, j, k, ini, fim: ponteiros/contadores
i = 1;     k = 1;      j = m + 1
ENQUANTO ( i ≤ m  E  j ≤ n )
     SE ( Ki ≤ Kj )
          Sk = Ri
          i = i + 1
     SENÃO
          Sk = Rj
          j = j + 1

SE ( i > m )
     ini = j
     fim = n
SENÃO
     ini = i
     fim = m

PARA i = ini  ATÉ  fim
      Sk = Ri
      k = k + 1
// fim para

// fim enquanto

RETORNA

FIM {MERGE2}
```

Algoritmo 3.1: Merge2 – Intercalação de dois caminhos

3. Intercalação de três caminhos

O algoritmo que faz a intercalação de três caminhos, identificado como o procedimento Merge3, utiliza Ri, Rj, Rl e Sk respectivamente representantes das chaves dos seus registros, como segue.

Arq1: K1 < K2 < K3 < . . . < Kp-1 < Kp (ponteiro "i")
Arq2: Kp+1 < Kp+2 < . . . < Kq-1 < Kq (ponteiro "j")
Arq3: Kq+1 < Kq+2 < . . . < KN-1 < KN (ponteiro "l")
Arq4: S1 < S2 < S3 < . . . . . .< SN-1 < SN (ponteiro "k")

Uma vez atingido o fim de um dos arquivos, será utilizada o procedimento Merge2, definido na seção 3.2, para intercalar os demais registros dos outros dois arquivos. É definida uma função MENOR que identifica o menor de três valores numéricos.

```
PROCEDIMENTO MERGE3 (R, S, p, q, N)

ENTRADA: INTEIROS p, q, N
         Registro R com três séries a intercalar

R1:p  (primeira série ordenada)
Rp+1:q (segunda série ordenada)
Rq+1:N (terceira série ordenada)

SAÍDA: Registro S com uma única série ordenada
// S1:N (série ordenada/intercalada)

// Variáveis auxiliares
     K: vetor [1:n] de inteiros
     i, j, k, l, m, n: ponteiros/contadores

FUNÇÃO MENOR ( X, Y, Z ): inteiro

     ENTRADA: X, Y, Z inteiros
     SAÍDA: MENOR
```

```
 // retorna 1 se o menor é X
 // retorna 2 se o menor é Y
 // retorna 3 se o menor é Z

// Variáveis auxiliares
          v: vetor [1:3] de inteiros
          w: inteiro

     v[1] = X;   v[2] = Y;   v[3] = Z;
     MENOR = 1;    w = v[1];

     PARA i = 2 ATÉ 3
          SE v[i] < w
             w = v[i]
             MENOR = i
     // fim para

     DEVOLVA MENOR

FIM {MENOR}

i = 1;          k = 1;          j = p + 1;    l
= q + 1
ENQUANTO ( i ≤ p  E  j ≤ q  E  l ≤ n )
     CASO  MENOR( Ki , Kj , Kl )
   1:  Sk = Ri
         i = i + 1
   2:  Sk = Rj
         j = j + 1
   3:  Sk = Rl
         l = l + 1
       // fim caso

   k = k + 1
```

```
// O trecho seguinte define os parâmetros dos dois
// últimos arquivos para o MERGE2.

SE ( i > p )
        m = q - j + 1
        n = N - l + 1
SENÃO
      SE ( i > p )
         m = p - i + 1
         n = N - l + 1
      SENÃO
           m = p - i + 1
         n = q - j + 1
MERGE2 ( R, S, m, n )

// fim enquanto
RETORNA
FIM {MERGE3}
```

Algoritmo 3.2: Merge3 – Intercalação de três caminhos

4. Intercalação de k caminhos

Num caso geral, o algoritmo para fazer a intercalação de k caminhos, identificado como o procedimento Mergek, pode ser implementado de forma similar ao algoritmo da seção 3.

Por exemplo, para o **Merge4** (k=4), a função MENOR selecionaria o menor valor entre quatro números e no momento em que fosse atingido o fim de um dos arquivos, deveria ser "chamado" o procedimento Merge3 para finalizar o processo de intercalação.

Porém, como já foi citado anteriormente, não é comum a utilização de uma intercalação com mais de três caminhos, devido a complexidade deste algoritmo, mesmo sabendo que o número de leituras e gravações poderia ser reduzido.

Os algoritmos de classificação por intercalação de k caminhos são conhecidos como de distribuição **equilibrada**. Na seção seguinte faremos uma comparação entre intercalações equilibradas para diferentes valores de k.

5. Comparação entre Distribuições Equilibradas

Esta comparação será feita através de um exemplo. Vamos supor que para classificar 7000 registros, a memória disponível para a ordenação interna, em função do tamanho dos registros, suporta processar no máximo 1000 registros por vez, ou seja, a cada 1000 registros lidos, é feita sua classificação e a série correspondente ordenada é gravada em um meio auxiliar, de forma sequencial.

Caso o número de registros do arquivo não seja múltiplo de 1000, somente a última série ficará incompleta, porém, como vimos nas seções anteriores, isto não será problema para o procedimento de intercalação do tipo Mergek.

O número de arquivos de trabalho, bem como a distribuição das séries nestes arquivos, depende do número **k** de caminhos a serem intercalados. Nas subseções seguintes fazemos uma simulação para k= 2, 3 e 4 caminhos.

5.1 Classificação Equilibrada de dois caminhos

Para k=2 serão necessários 4 arquivos de trabalho. A fase inicial de classificação cria séries de comprimento 1000 e as coloca alternadamente em dois destes arquivos (Arq1 e Arq2), assim:

Arq1	R_{1-1000}	$R_{2001-3000}$	$R_{4001-5000}$	$R_{6001-7000}$
Arq2	$R_{1001-2000}$	$R_{3001-4000}$	$R_{5001-6000}$	–

Figura 3.12: Distribuição equilibrada em dois arquivos (Passo-1)

A notação $\mathbf{R_{p-q}}$ corresponde à série ordenada a partir dos registros de entrada da posição **p** até a posição **q**.

As séries dispostas em cada coluna da tabela anterior são intercaladas via Merge2 produzindo outras séries até duas vezes maiores que as séries iniciais. Essas são gravadas em outros dois arquivos (Arq3 e Arq4), assim:

Arq3	R_{1-2000}	$R_{4001-6000}$
Arq4	$R_{2001-4000}$	$R_{6001-7000}$

Figura 3.13: Resultado da intercalação de dois caminhos (Passo-2)

Este processo é repetido de forma alternada entre os arquivos Arq1/Arq2 e Arq3/Arq4, até que apenas uma série (arquivo classificado) seja produzida.

Arq1	R_{1-4000}		**Arq3**	R_{1-7000}
Arq2	$R_{4001-7000}$		**Arq4**	–

Figura 3.14: Intercalação final de dois caminhos (Passos 3 e 4)

Observe que em todo o processo de classificação foram realizadas 28.000 leituras e 28.000 gravações, correspondendo a 7.000 delas em cada passo.

Para efeito, vamos considerar um arquivo com 20 registros, considerando somente suas chaves, e um limite de classificação interna de 3 registros.

R_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
K_i	5	27	14	2	29	22	33	9	18	7	1	23	11	17	4	12	9	8	5	23

Figura 3.15: Exemplo de classificação com 7 séries de até 3 registros cada

Arq1	5-14-27	9-18-33	4-11-17	5-23
Arq2	2-22-29	1-7-23	8-9-12	–

Figura 3.16: Fase-1 da intercalação de dois caminhos do exemplo da Fig. 3.15

Arq3	2-5-14-22-27-29	4-8-9-11-12-17
Arq4	1-7-9-18-23-33	5-23

Figura 3.17: Fase-2 da intercalação de dois caminhos a partir da Fig. 3.16

Arq1	1-2-5-7-9-14-18-22-23-27-29-33	**Arq3**	1-2-4-5-5-7-8-9-9-11-12-14-17-18-22-23-23-27-29-33
Arq2	4-5-8-9-11-12-17-23	**Arq4**	–

Figura 3.18: Fases 3 e 4 da intercalação de dois caminhos a partir da Fig. 3.17

5.2 Classificação Equilibrada de três caminhos

Para k=3 podem ser usados também 4 arquivos de trabalho. A fase inicial de classificação cria séries de comprimento 1000 e as distribui também em dois arquivos, a diferença é que em determinado momento da segunda fase, usando o Merge2, pelo menos um dos arquivos fica vazio, de modo que é possível a partir daí utilizar o Merge3.

Observe a seguir como seria a classificação.

Fase-1 (ordenação interna e gravação inicial)

Arq1	$R_{1–1000}$	$R_{2001–3000}$	$R_{4001–5000}$	$R_{6001–7000}$
Arq2	$R_{1001–2000}$	$R_{3001–4000}$	$R_{5001–6000}$	–
Arq3	–	–	–	–
Arq4	–	–	–	–

Figura 3.19: Distribuição equilibrada em três arquivos

Fase-2 (uso do Merge2) ↓ (ponteiro)

Arq1	–	–	–	$R_{6001-7000}$
Arq2	–	–	–	–
Arq3	R_{1-2000}	$R_{4001-6000}$	–	–
Arq4	$R_{2001-4000}$	–	–	–

Figura 3.20: Intercalação de dois caminhos do exemplo da Fig. 3.19

Fase-3 (uso do Merge3)

Arq1	–	–
Arq2	S_{1-5000}	–
Arq3	–	$R_{4001-6000}$
Arq4	–	–

Fase-4 (uso do Merge2)

Arq1	S_{1-7000}
Arq2	–
Arq3	–
Arq4	–

Figura 3.21: Intercalação de três e dois caminhos a partir da Fig. 3.20

A série S_{1-5000} é a saída do Merge3 com entradas obtidas a partir de $R_{6001-7000}$, R_{1-2000} e $R_{2001-4000}$, nesta ordem. A fase final corresponde ao Merge2 entre os registros com ponteiros em Arq2 e Arq3 com saída em Arq1.

Observe que para todas as fases deste tipo de classificação pode ser usado o Merge3, dado que, de acordo com sua descrição na seção 3, quando um dos arquivos está com o ponteiro no final (end_of_ file) o Merge2 é acionado.

O número de leituras e gravações é reduzido de 28.000 para 25.000, visto que para cada fase teríamos:

Fase-1	Fase-2	Fase-3	Fase-4	**Total (L/G)**
7.000	6.000	5.000	7.000	**25.000**

Figura 3.22: Número de Leituras e Gravações para o exemplo da Fig. 3.19

5.3 Classificação Equilibrada de múltiplos caminhos

Para k > 4 serão necessários (k+1) arquivos de trabalho. A fase inicial de classificação cria séries de comprimentos fixos e as coloca alternadamente em k arquivos (Arq1 a Arqk), deixando o de ordem (k+1) livre. As fases seguintes utilizam o procedimento de intercalação de k caminhos (Mergek).

Para o exemplo da subseção anterior, teríamos para k=4:

Fase-1 (classificação interna e distribuição alternada das séries)

Arq1	R_{1-1000}	$R_{4001-5000}$
Arq2	$R_{1001-2000}$	$R_{5001-6000}$
Arq3	$R_{2001-3000}$	$R_{6001-7000}$
Arq4	$R_{3001-4000}$	–
Arq5	–	–

Figura 3.23: Distribuição equilibrada em quatro arquivos

Fase-2 (Merge4)

Arq1	–	$R_{4001-5000}$
Arq2	–	$R_{5001-6000}$
Arq3	–	$R_{6001-7000}$
Arq4	–	–
Arq5	R_{1-4000}	–

Figura 3.24: Intercalação de quatro caminhos do exemplo da Fig. 3.23

Fase-3 (Merge4)

Arq4	R_{1-7000}

Figura 3.25: Intercalação de quatro caminhos a partir da Fig. 3.24

O número de leituras e gravações é reduzido mais ainda, no caso, de 25.000 para 18.000, assim:

Fase-1	Fase-2	Fase-3	**Total (L/G)**
7.000	4.000	7.000	**18.000**

Figura 3.26: Número de Leituras e Gravações para o exemplo da Fig. 3.23

6. Intercalação Polifásica

O objetivo deste tipo de intercalação é distribuir as séries iniciais de forma **desequilibrada**, de modo que menos passos sejam necessários para a classificação total.

Em função do número de arquivos **T** a serem utilizados é gerada uma tabela de distribuição com base na Série de Fibonacci. É necessário então que o algoritmo de classificação externa contenha uma tabela desta distribuição em função do valor de **T**. Sabemos que, de modo similar à intercalação de múltiplos caminhos, para **T** arquivos devemos ter k = (T –1) caminhos.

A seguir mostramos as tabelas para alguns casos. Caso o número de séries não esteja contido na tabela, considerar a distribuição imediatamente superior; isso acarreta que durante a fase de intercalação as séries complementares sejam vazias, ou seja, já foi atingido seu final de arquivo. Por exemplo, na Tabela 3.1, caso o número de séries seja igual a 10, a distribuição usada seria a de nível 5, ou 13 séries, sendo que nesse caso, 3 delas seriam vazias.

Nível	T_1	T_2	Séries
N	a	b	T
n+1	a+b	a	t+a
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21
7	21	13	34
8	34	21	55
9	55	34	89
10	89	55	144
11	144	89	233
. . .	. . .	. . .	. . .

Tabela 3.1: Distribuição polifásica para k=2 (T=3 arquivos, usar o Merge2)

Nível	T_1	T_2	T_3	Séries
N	a	b	c	t
n+1	a+b	a+c	a	t+2a
0	1	0	0	1
1	1	1	1	3
2	2	2	1	5
3	4	3	2	9
4	7	6	4	17
5	13	11	7	31
6	24	18	13	55
7	42	37	24	103
8	79	66	42	187
. . .	. . .	. . .		. . .

Tabela 3.2: Distribuição polifásica para k=3 (T=4 arquivos, usar o Merge3)

Nível	T_1	T_2	T_3	T_4	Séries
n	a	b	C	d	t
n+1	a+b	a+c	a+d	a	t+3a
0	1	0	0	0	1
1	1	1	1	1	4
2	2	2	2	1	7
3	4	4	3	2	13
4	8	7	6	4	25
5	15	14	12	8	49
6	29	27	23	15	94
7	56	52	44	29	181
8	108	100	85	56	349
. . .	. . .	. . .			. . .

Tabela 3.3: Distribuição polifásica para k=4 (T=5 arquivos, usar o Merge4)

Exemplo 3.1: Classificação por interpolação polifásica para 25 séries utilizando T=5 arquivos. Consultando a tabela acima observe que a distribuição inicial deve ser feita de acordo com a linha indicada pelo nível 5. Na Tabela 3.4 o arquivo de saída em cada fase está indicado pela símbolo '–'.

Fase	T_1	T_2	T_3	T_4	T_5
1	8 x 1	7 x 1	6 x 1	4 x 1	–
2	4 x 1	3 x 1	2 x 1	–	4 x 4
3	2 x 1	1 x 1	–	2 x 7	2 x 4
4	1 x 1	–	1 x 13	1 x 7	1 x 4
5	–	1 x 25	–	–	–

Tabela 3.4: Distribuição para k=4 caminhos e T=5 arquivos do Exemplo 3.1

A notação s x t indica s séries de tamanho t. Note que de uma fase para a seguinte o menor valor de s (número de séries) da fase "i" corresponde à quantidade a ser retirada de cada arquivo para que possa ser feita a intercalação Merge4 e gravadas no arquivo de saída da fase "i+1".

Exemplo 3.2: Classificação por interpolação polifásica para 25 séries utilizando T=4 arquivos. Consultando a Tabela 3.2 observamos que o número 31 constante é aquele que mais se aproxima de 25; no caso seriam distribuídas entre os arquivos 6 séries fictícias vazias.

Utilizando o Merge3, a sequência de intercalações seria esta:

Fase	T_1	T_2	T_3	T_4
1	13 x 1	11 x 1	7 x 1	–
2	6 x 1	4 x 1	–	7 x 3
3	2 x 1	–	4 x 5	3 x 3
4	–	2 x 9	2 x 5	1 x 3
5	1 x 17	1 x 9	1 x 5	–
6	–	–	–	1 x 31

Tabela 3.5: Distribuição para k=3 caminhos e T=4 arquivos do Exemplo 3.2

Atividades de avaliação

1. O algoritmo Merge2 da seção 2 não leva em consideração se os dois blocos a serem intercalados são tais que todas as chaves de um bloco sejam maiores que o outro. Faça então uma alteração neste procedimento para prever este caso específico. Indique qual o benefício desta ação.
2. Para a intercalação de 4 caminhos (Merge4) poderia ser usada uma das técnicas citadas a seguir. Indique qual a melhor delas e justifique.
   a) De forma semelhante ao Merge3 da seção 3, modificando a função MENOR para comparar quatro números e ampliando as comparações.
   b) Merge4 (A, B, C, D) = Merge2 (Merge3 (A,B,C), D)
   c) Merge4 (A, B, C, D) = Merge2 (Merge2 (A,B) , Merge2 (C,D))
3. Utilize algum procedimento apresentado neste capítulo para classificar um arquivo contendo 4.500 registros, sabendo que é possível classificar no

máximo 750 destes registros na memória interna. Indique o número de leituras e gravações ocorridas.

4. Defina uma estratégia diferente da utilizada no item anterior de modo a reduzir o número de leituras e gravações.

5. Considere as seguintes chaves de um arquivo que pode ser dividido em exatamente dois blocos para uso numa classificação externa:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
K_i	15	7	24	2	19	12	13	9	28	6	3	53	21	17	4	42	19	18	5	13

Suponha a existência de uma função MEDIANA(V,N), que calcula a mediana de um vetor V[1:N] e dos procedimentos QSORT(A,N) que ordena o vetor A[1:N] totalmente na memória principal e Merge2 (X, m,Y, n, S) que faz a intercalação de 2 caminhos, onde S[1:(m+n)] é o vetor ordenado resultante da intercalação dos vetores X[1:m] e Y[1:n].

Apresente os resultados após executar cada um dos seguintes passos:

a) W = MEDIANA (K, 20)

b) X = Ki, se Ki < W e Y = Ki, se Ki > W, $\forall$i = 1:20

c) QSORT (X, 10); QSORT (Y, 10);

d) Merge2 (X, m,Y, n, S); apresente S.

6. Escreva um algoritmo para resolver de forma geral o método descrito no exercício 5.

7. Generalize o método descrito no exercício 5 para proceder a classificação interna considerando qualquer número de séries.

8. Como foi apresentado, a ordenação de um arquivo através da classificação externa se faz necessário quando seus registros não cabem totalmente na memória principal, em função do tamanho de cada um deles, e não por causa do grande número de chaves. Indique uma alternativa para ordenar arquivos desta magnitude sem utilizar os procedimentos de intercalação.

9. Fazer uma comparação, em relação ao número de leitura e gravações necessárias para realizar uma classificação externa com 25 séries de tamanho 1000 cada, considerando as seguintes técnicas de intercalação equilibrada de:

a) dois caminhos

b) três caminhos

c) quatro caminhos

10. Repetir o exercício 9 considerando agora a intercalação polifásica de:

a) dois caminhos (T=3)

b) quatro caminhos (T=5)

Capítulo 4

Técnicas de pesquisa

Objetivos

- Definir as técnicas de pesquisa sequencial e binária
- Apresentar as tabelas de dispersão
- Apresentar as árvores de busca balanceadas
- Mostrar as técnicas de manutenção das árvores de pesquisa
- Analisar a complexidade dos algoritmos de pesquisa

Apresentação

Neste capítulo apresentamos diversas técnicas e estruturas de dados que permitem fazer pesquisa com bastante eficiência. Inicialmente, apresentamos o procedimento de busca sequencial que, apesar de ser ineficiente, serve para efeito de comparação com os demais métodos. A seguir abordamos a busca binária, que permite encontrar um valor contido num vetor ordenado com extrema eficiência. No entanto, as operações de inserção e remoção têm custo computacional elevado, por isso é comum utilizar outras estruturas de pesquisa em que os dados estejam implicitamente ordenados, como as tabelas de dispersão onde, sob certas hipóteses, essas operações podem ser feitas em tempo esperado constante. No final do capítulo abordamos três tipos de árvores de busca balanceadas: árvores AVL, árvores B e árvores B+. Descrevemos as operações de inserção, busca e remoção nessas estruturas e mostramos que tais operações são feitas em tempo logarítmico, no pior caso.

1. Técnicas de Pesquisa

Uma das tarefas de maior importância na computação é a pesquisa de informações contidas em coleções de dados. Em geral, desejamos que essa tarefa seja executada sem que haja a necessidade de inspecionar toda a coleção de dados.

Neste capítulo apresentamos diversas técnicas e estruturas de dados que permitem fazer pesquisa com bastante eficiência. Discutimos as operações de inserção, busca e remoção nessas estruturas, analisando sua complexidade temporal e espacial.

Inicialmente, nas seções 2 e 3, descrevemos respectivamente os procedimentos de busca sequencial que é trivial e ineficiente, e de busca binária, que permite encontrar um valor contido num vetor ordenado com extrema eficiência. Sabe-se, no entanto, que as operações de inserção e remoção num vetor ordenado têm custo computacional elevado.

Em seguida, na seção, apresentamos as tabelas de dispersão, discutindo as técnicas de hashing fechado e hashing aberto. Tais técnicas podem ser usadas para resolver o problema da colisão de chaves que é inerente ao uso dessas estruturas. Mostramos que, sob certas hipóteses, as operações de inserção, busca e remoção em tabelas de dispersão são feitas em tempo esperado constante.

Finalmente, na seção 5 abordamos três tipos de árvores de busca balanceadas: árvores AVL, árvores B e árvores B+. Nessas estruturas as operações de inserção, busca e remoção são feitas em tempo logarítmico.

2. Busca Sequencial

Podemos procurar, ou buscar, um valor x num vetor L inspecionando em sequência as posições de L a partir da primeira posição. Se encontrarmos x, a busca tem sucesso. Se alcançarmos a última posição de L sem encontrar x, concluímos que x não ocorre no vetor L. Esse tipo de busca é chamado de busca sequencial.

Considerando que o vetor L contém n elementos, ordenados ou não, é fácil verificar que a busca sequencial requer tempo linearmente proporcional ao tamanho do vetor, ou seja, O(n); por isso, é comum dizer que a busca sequencial é uma busca linear. Observa-se que, no melhor caso, o elemento x é encontrado logo na primeira tentativa da busca, através de uma comparação; no pior caso, x encontra-se na última posição e são feitas n comparações, logo, no caso médio, o elemento é encontrado após (n+1)/2 comparações. Para vetores de médio ou grande porte, esse tempo é considerado inaceitável, dado que existem técnicas mais eficientes descritas nas seções seguintes.

Antes apresentamos dois algoritmos para a busca sequencial.

```
PROCEDIMENTO BUSCA_SEQUENCIAL ( L , X , POS )

   ENTRADA: UM VETOR L e UM VALOR X
   SAÍDA: POS = i, SE X OCORRE NA POSIÇÃO i DE L
// SUCESSO
         POS = 0, CASO CONTRÁRIO.

   POS = 0

PARA i = 1 até N
     SE L[i] = X
        POS = i
```

```
            ESCAPE
        // fim para
        DEVOLVA POS
        FIM {BUSCA_SEQUENCIAL}
```

Algoritmo 4.1: Busca Sequencial Simples

```
        PROCEDIMENTO BUSCA_SEQUENCIAL_REC ( L , N, X )

           ENTRADA: UM VETOR L DE TAMANHO N e UM VALOR X
           SAÍDA: i, SE X OCORRE NA POSIÇÃO i DE L    //
        SUCESSO
                  0, CASO CONTRÁRIO.

         SE N = 1
         SE L[1] = X
             DEVOLVA 1
         SENÃO
             DEVOLVA 0
           SENÃO
         SE L[N] = X
            DEVOLVA N
         SENÃO
               BUSCA_SEQUENCIAL_REC ( L , N-1, X )

        FIM {BUSCA_SEQUENCIAL_REC}
```

Algoritmo 4.2: Busca Sequencial Recursiva

3. Busca Binária

Quando o vetor L estiver em ordem crescente, podemos determinar se x ocorre em L de forma mais rápida da seguinte maneira: inspecionamos a posição central de L; se ela contém x a busca para, tendo sido bem sucedida, caso contrário, se x for menor do que o elemento central passamos a procurar x, recursivamente, no intervalo de L que está à esquerda da posição central. Se x for maior do que o elemento central continuamos a procurar x, recursivamente, no intervalo de L que está à direita da posição central. Se o intervalo se tornar vazio, a busca para, tendo sido mal sucedida.

O procedimento que acabamos de descrever é chamado de busca binária. Facilmente, podemos adaptar a busca binária para procurar valores em vetores que estejam em ordem decrescente. O Algoritmo 4.3 fornecemos uma descrição recursiva em pseudocódigo desse procedimento.

```
PROCEDIMENTO BUSCA_BINARIA
  ENTRADA: UM VETOR L EM ORDEM CRESCENTE, UM VA-
LOR X, E AS
           POSIÇÕES INICIO E FIM.
  SAÍDA: SIM, SE X OCORRE ENTRE AS POSIÇÕES INI-
CIO E FIM DE
         L; NÃO, CASO CONTRÁRIO.

  SE INICIO > FIM
    DEVOLVA NÃO E PARE
  MEIO = (INICIO+FIM)/2                  // DIVISÃO
INTEIRA
  SE X = L[MEIO]
    DEVOLVA SIM E PARE
  SE X < L[MEIO]
    DEVOLVA BUSCA_BINARIA(L, X, INICIO, MEIO - 1)
  SENAO
    DEVOLVA BUSCA_BINARIA(L, X, MEIO + 1, FIM)
FIM {BUSCA_BINARIA}
```

Algoritmo 4.3: Busca Binária

A Figura 4.1 ilustra o funcionamento do Algoritmo Busca Binária ao procurar o valor 34 num vetor ordenado. Observe que apenas as posições 8, 13 e 10 do vetor (nessa ordem) são inspecionadas. A tabela que aparece após o vetor mostra os valores dos parâmetros de entrada (exceto o vetor L, que é passado por referência e é compartilhado por todas as chamadas), o conteúdo da variável MEIO e o valor devolvido por cada chamada.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L	2	5	5	8	10	13	16	17	29	31	34	36	43	49	51	56	65	69

			1ª	3ª	2ª	
Chamada	x	INICIO	FIM	MEIO	Valor devolvido
1	34	0	17	8	Sim
2	34	9	17	13	Sim
3	34	9	12	10	Sim

Figura 4.1: Exemplo de busca binária

Se estivéssemos procurando o valor 7 no vetor da Figura 4.1, iríamos inspecionar apenas as posições 8, 3, 1 e 2 do vetor (nessa ordem). A Tabela 4.1 mostra os valores dos parâmetros de entrada, o conteúdo da variável MEIO e o valor devolvido por cada chamada. Naturalmente, como 7 não ocorre no vetor, o valor devolvido é Não.

Chamada	x	INICIO	FIM	MEIO	Valor devolvido
1	7	0	17	8	Não
2	7	0	7	3	Não
3	7	0	3	1	Não
4	7	2	2	2	Não
5	7	3	2	-	Não

Tabela 4.1: Exemplo de busca binária

Na verdade, como o Algoritmo Busca Binária utiliza recursão de cauda, o compilador/interpretador pode gerar uma versão desse algoritmo em linguagem de máquina que tenha complexidade espacial constante.

Vamos denotar por T(n) o tempo requerido pelo Algoritmo Busca Binária para procurar um valor num intervalo com n posições. Note que se x não estiver na posição central do intervalo será feita uma chamada recursiva para procurar x num intervalo que terá aproximadamente n/2 posições. Dessa forma, é correto afirmar que:

T(n) T(n/2) + c

T(0) = c

Resolvendo essa fórmula de recorrência concluímos que $T(n) \in O(\log n)$. Sendo assim, a complexidade temporal do Algoritmo Busca Binária pertence a O(log n).

É fácil perceber que a complexidade espacial desse algoritmo também é dada pela mesma fórmula de recorrência. Concluímos que tal complexidade também pertence a O(log n). Vale salientar que se implementarmos o procedimento de busca binária ser usar recursividade, a complexidade espacial passa a ser constante.

É possível adaptar o procedimento de busca binária para encontrar um valor mais próximo de x num vetor ordenado. Por exemplo, no vetor da Figura 4.1, o valor mais próximo do 40 é 43. Esse tipo de busca é conhecida como busca binária aproximada. Tal adaptação é deixada como exercício para o leitor no final deste capítulo.

Apesar da busca binária ser extremamente eficiente, as operações de inserção e remoção são feitas em tempo linearmente proporcional ao tamanho do vetor e isso, em geral, é considerado inaceitável. Por esse motivo, nas situações em que as operações de inserção e remoção são frequentes, é preferível utilizar uma estrutura de dados onde os dados estejam implicitamente ordenados e que também permitam fazer inserções e remoções com bastante eficiência. Na Seção 4 descrevemos alguns tipos de árvores que possuem tais características.

4. Tabelas de Dispersão

As tabelas de dispersão, também conhecidas como tabelas de espalhamento ou tabelas de hashing, armazenam uma coleção de valores, sendo que cada valor está associado a uma chave. Tais chaves têm que ser todas distintas e são usadas para mapear os valores na tabela. Esse mapeamento é feito por uma função de hashing, que chamaremos de h.

Por exemplo, podemos implementar uma tabela de dispersão usando um vetor com oito posições e utilizar h(x) = x mod 8 como função de hashing. Dizemos que h(k) é a posição original da chave k e é nessa posição da tabela que a chave k deve ser inserida. A Figura 4.2 ilustra a inserção de uma coleção de valores com suas respectivas chaves numa tabela de dispersão.

Função de hashing:
h(x) = x mod 8

	Chave	Valor
0		
1	25	Lia
2	18	Ana
3		
4		
5	5	Rui
6		
7	31	Gil

Figura 4.2: Tabela de dispersão

O operador **mod** denota a operação de resto da divisão, inteira.

Observe que o valor Gil foi colocado na posição 7 da tabela, pois a chave associada a Gil é 31 e h(31) = 7. O que aconteceria se tentássemos inserir nessa tabela o valor Ivo com chave 33 ? Observe que h(33) = 1, mas a posição 1 da tabela já está ocupada. Dizemos que as chaves 25 e 33 colidiram. Mais precisamente, duas chaves x e y colidem se h(x) = h(y).

Note que o problema da colisão de chaves ocorre porque, na maioria dos casos, o domínio das chaves é maior que a quantidade de posições da tabela. Sendo assim, pelo princípio da casa de pombos, qualquer função do conjunto das chaves para o conjunto das posições da tabela não é injetora.

Não é aceitável recusar a inserção de uma chave que colida com outra já existente na tabela se ela ainda tiver posições livres. Precisamos, portanto, de alguma estratégia para lidar com as colisões de chaves. Existem diversas técnicas para lidar com as colisões. Nas próximas duas subseções apresentaremos duas dessas técnicas.

4.1 Hashing Fechado

Na técnica conhecida como hashing fechado, quando tentamos inserir uma chave y e ela colide com uma chave x já existente na tabela, colocamos y na primeira posição livre após a posição h(y). Uma tabela de dispersão que utilize essa estratégia para resolver as colisões de chaves costuma ser chamada de tabela de hashing fechado.

Na tabela da Figura 4.2, a chave 33 deve ser inserida na posição 3 visto que ela é a primeira posição livre após a posição h(33). Consideramos que após a última posição da tabela vem a posição 0. Sendo assim, na tabela da Figura 4.2, a chave 23 deve ser inserida na posição 0. A Figura 4.3 mostra a tabela da Figura 4.2 após a inserção das chaves 33 e 23. Dizemos que as chaves 23 e 33 estão deslocadas de suas posições originais e denotamos tal fato na figura usando o asterisco.

Função de hashing:
h(x) = x mod 8

	Chave	Valor
0	23*	Edu
1	25	Lia
2	18	Ana
3	33*	Ivo
4		
5	5	Rui
6		
7	31	Gil

Figura 4.3: Inserções em hashing fechado

Obviamente, se a tabela estiver cheia e tentarmos inserir mais uma chave, tal inserção não será possível. Nesse caso, dizemos que ocorreu um evento conhecido como estouro da tabela de dispersão (hash overflow).

Ao usarmos essa estratégia para resolver as colisões, a seguinte propriedade é mantida pelas operações de inserção: se uma chave k foi inserida numa posição j, então as posições da tabela desde h(k) até j têm que estar ocupadas. Como decorrência dessa propriedade, para buscar uma chave k numa tabela de hashing fechado devemos inspecionar em sequência as posições da tabela a partir da posição h(k). A busca termina quando ocorrer um dos seguintes eventos:

- A posição inspecionada contém k. Nesse caso, a busca foi bem sucedida.
- A posição inspecionada está livre. Nesse caso, podemos concluir que k não ocorre na tabela.

- Todas as posições foram inspecionadas. Se esse evento ocorrer antes dos dois anteriores significa a tabela não tem posições livres e que a chave k não ocorre na tabela.

Por exemplo, se procurarmos a chave 33 na tabela da Figura 4.3, teremos que inspecionar as posições 1, 2 e 3 e a busca será bem sucedida. Por outro lado, se procurarmos a chave 15, inspecionaremos as posições 7, 0, 1, 2 e 3 e a busca será mal sucedida.

A remoção numa tabela de hashing fechado é um pouco mais complicada que a inserção e a busca. Precisamos garantir que a propriedade decorrente das inserções, que enunciamos anteriormente, também seja preservada pelas remoções.

Para remover uma chave k, primeiramente precisamos encontrá-la. Obviamente, se k não ocorre na tabela a remoção é mal sucedida. Suponha que a chave k foi encontrada numa posição j. Tal posição deve ser liberada. Em seguida, inspecionamos as posições subsequentes em busca de uma chave que esteja deslocada. Se encontrarmos uma chave deslocada cuja posição original seja a posição que foi liberada ou uma posição anterior a ela, devemos movê-la para a posição que está livre. Note que a posição onde estava a chave deslocada agora está livre. O processo é então repetido até que uma posição livre seja alcançada.

Por exemplo, para remover a chave 18 da tabela da Figura 4.3 devemos liberar a posição 2 e em seguida mover a chave 33 para a posição 2. A remoção para quando a posição 4 é atingida. Se após essa remoção quisermos remover a chave 31, teremos que mover a chave 23 para a posição 7. Com isso, a posição 0 fica livre. Observe que chave 33, apesar de estar deslocada, não deve ser movida para a posição 0, pois sua posição original é a posição 1. A Figura 4.4 mostra como ficaria a tabela da Figura 4.3 após essas duas remoções.

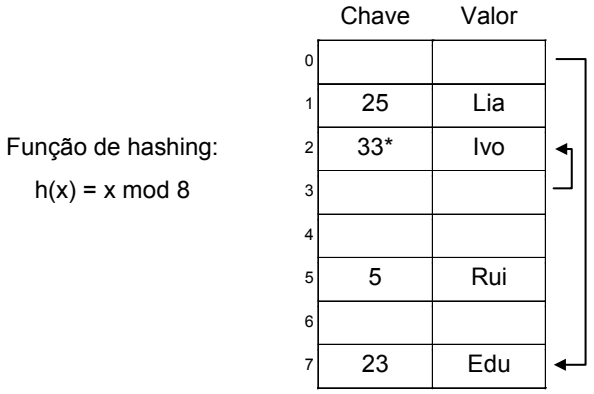


Figura 4.4: Remoções em hashing fechado

4.2 *Hashing* Aberto

A técnica de hashing aberto é bem mais simples que hashing fechado. Numa tabela de hashing aberto cada posição da tabela contém um ponteiro para uma lista encadeada que contém todas as chaves mapeadas pela função de hashing naquela posição. Note que usando essa técnica o espaço de armazenamento das chaves não fica restrito à tabela, daí o nome hashing aberto, e não há limitação quanto à quantidade de chaves que podem ser armazenadas na **tabela**. Dessa forma, ao utilizar essa estratégia para resolver as colisões, não precisamos nos preocupar com hash overflow.

Naturalmente existe um limite para a quantidade de chaves que podem ser armazenadas numa tabela de hashing aberto. No entanto, esse limite é determinado pela memória física do computador e não pelo tamanho da tabela.

Para inserir uma chave k numa tabela de hashing aberto, calculamos h(k) e então inserimos tal chave (e o valor associado a ela) no final da lista encadeada apontada pela posição h(k) da tabela, caso essa chave não ocorra nessa lista (lembre-se de que numa tabela de dispersão as chaves têm que ser todas distintas).

Ilustramos na Figura 4.5 a inserção das chaves 25, 18, 31, 5, 23 e 33, nessa ordem, numa tabela de hashing aberto com 5 posições e que utiliza a função de hashing h(x) = x mod 5.

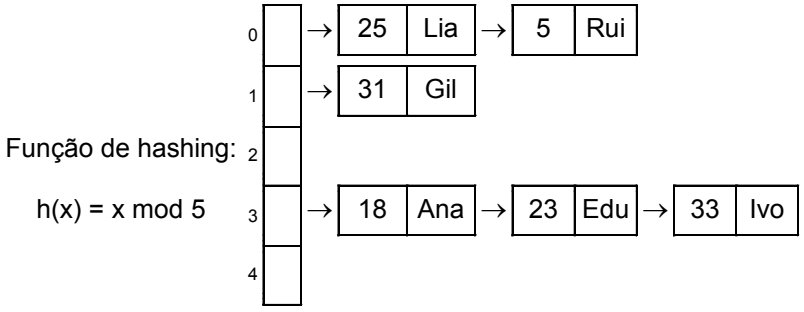


Figura 4.5: Inserções em Hashing aberto

Para encontrar uma chave k numa tabela de hashing aberto basta procurar tal chave na lista encadeada apontada pela posição h(k) da tabela. A remoção em hashing aberto também é bastante simples. Após encontrar a chave que se deseja remover, basta remover o nó que a contém. A Figura 4.6 mostra a tabela da Figura 4.5 após a remoção das chaves 18 e 31.

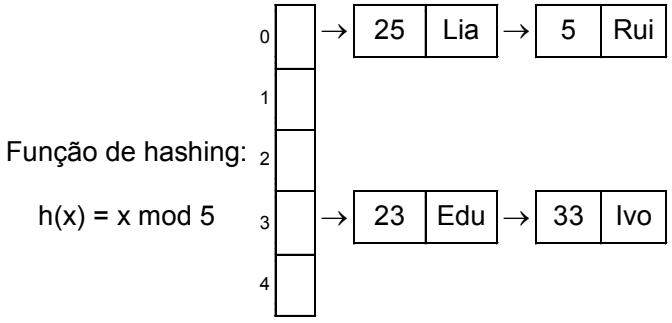


Figura 4.6: Remoções em Hashing aberto

Vamos agora discutir a eficiência das operações de inserção, busca e remoção em tabelas de dispersão. Antes, precisamos definir alguns termos.

Dizemos que uma função de hashing é boa se ela pode ser computada em tempo constante e se ela espalha as chaves na tabela de maneira uniforme. Por exemplo, dada uma tabela de dispersão com 37 posições, a função h(x) = 2x mod 37 não é boa, pois ela espalha as chaves somente nas posições ímpares da tabela. A função h(x) = x! mod 37 também não é boa, pois não pode ser computada em tempo constante. Já a função h(x) = x mod 37 é boa, se o domínio das chaves é o conjunto dos números naturais. Caso o domínio das chaves seja o conjunto dos naturais pares, h(x) = x mod 37 deixaria de ser uma boa função de hashing, pois mapearia as chaves somente nas posições ímpares da tabela. Nesse caso, a função h(x) = x/2 mod 37 seria considerada boa.

A carga de uma tabela de dispersão é a razão entre a quantidade de chaves e a quantidade de posições da tabela. Por exemplo, a carga da tabela da Figura 4.6 é 0,8. A carga de uma tabela de hashing fechado é considerada baixa se ela é menor ou igual a 0,5. No caso de uma tabela de hashing aberto, a carga é considerada baixa se ela é limitada por uma constante.

É possível mostrar que se a função de hashing é boa e a carga é baixa, as operações de inserção, busca e remoção são feitas em tempo esperado constante.

Observe que no caso de hashing fechado, as operações de inserção, busca e remoção sempre terminam quando uma posição livre é atingida. Se a carga é baixa, pelo menos a metade das posições da tabela estarão livres. Sendo assim, se a função de hashing espalha as chaves de maneira uniforme, a cada duas posições consecutivas é esperado que pelo menos uma esteja livre e consequentemente a quantidade esperada de posições que devem ser inspecionadas é constante.

No caso de um hashing aberto, se a carga é baixa e a função de hashing espalha as chaves de maneira uniforme, o tamanho esperado de cada lista encadeada será constante. Note que as operações de inserção, busca e remoção gastam tempo limitado pelo tamanho da maior lista encadeada. Isso implica que tais operações irão requerer tempo esperado constante.

Nas situações práticas nas quais a quantidade máxima de chaves que serão armazenadas na tabela pode ser estimada a priori, é possível determinar o tamanho que a tabela deve ter de modo a garantir que sua carga seja sempre baixa. Em tais situações, o uso de tabelas de dispersão pode ser bastante adequado.

5. Árvores de Busca Balanceadas

Nessa seção abordaremos as árvores de busca balanceadas. Nessas árvores, as chaves armazenadas são mantidas implicitamente ordenadas. Isso permite que a operação de busca seja feita percorrendo-se um ramo da árvore, desde a raiz até, no máximo, chegar a uma folha. Além disso, tais árvores possuem propriedades que garantem que sua altura seja muito pequena se comparada à quantidade de chaves contidas na árvore. Como veremos mais adiante, isso garante que as operações de inserção, busca e remoção sejam feitas com muita eficiência.

Discutiremos três tipos de árvores de busca balanceadas: árvores AVL, árvores B e árvores B+.

5.1 Árvores AVL

As árvores AVL são árvores binárias propostas por Adelson-Velski e Landis em 1962 que se caracterizam por duas propriedades:

- Se um nó da árvore contém uma chave x, as chaves contidas na subárvore à esquerda desse nó são todas menores do que x e as chaves contidas na subárvore à direita desse nó são todas maiores do que x.
- Para cada nó da árvore, a diferença de altura entre a subárvore à esquerda desse nó e a subárvore à direita desse nó é de no máximo 1.

A primeira propriedade garante que as chaves contidas na árvore estejam implicitamente ordenadas. A segunda propriedade garante o balanceamento da árvore.

A estrutura de um nó de uma árvore AVL pode ser constituída dos seguintes campos:

- chave: armazena uma chave.
- filhoesq: ponteiro para o filho esquerdo.
- filhodir: ponteiro para o filho direito.
- bal: a altura da subárvore à esquerda menos a altura da subárvore à direita do nó.

O campo bal indica como está o balanceamento das subárvores apontadas pelo nó e auxilia a manter o balanceamento da árvore nas operações de inserção e remoção. Observe que os únicos valores aceitáveis para o campo bal são –1, 0 e 1. A Figura 4.7 exibe um exemplo de árvore AVL. O valor do campo bal aparece acima de cada nó.

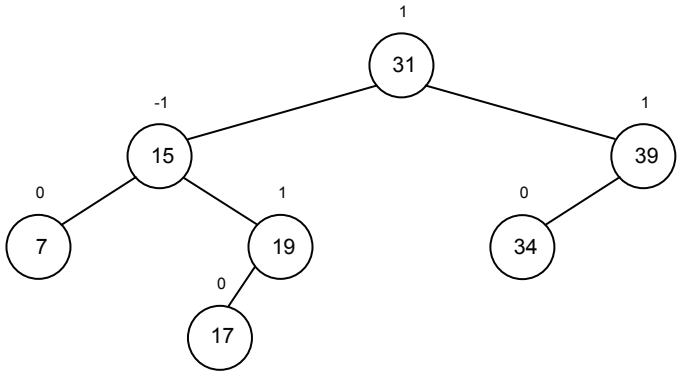


Figura 4.7: Exemplo de árvore AVL

A operação de busca numa árvore AVL é similar à busca numa árvore binária de busca qualquer. Para procurar uma chave k, primeiramente verificamos se a árvore é vazia. Em caso afirmativo, a busca para, tendo sido mal sucedida. Caso contrário, verificamos se a chave contida na raiz é k. Nesse caso, a busca tem sucesso. Caso contrário, se k for menor do que a chave contida na raiz, repetimos o processo recursivamente na subárvore à esquerda da raiz. Se k for maior do que a chave contida na raiz, repetimos o processo recursivamente na subárvore à direita da raiz. Na árvore da Figura 4.7, para chegar na chave 17 teremos que passar antes pelos nós que contém as chaves 31, 15 e 19. Se buscarmos a chave 42, passaremos pelos nós que contém as chaves 31 e 39 até atingir a subárvore à direita do 39, que é vazia. A busca será então mal sucedida.

Se permitirmos a ocorrência de chaves repetidas numa árvore AVL podemos chegar numa situação onde é impossível preservar a propriedade do balanceamento. Por exemplo, imagine como seria uma árvore AVL com as chaves 15, 15 e 15.

Já a operação de inserção é mais complicada. Para inserir uma chave k numa árvore AVL devemos percorrê-la, a partir da raiz, como se estivéssemos procurando k. Se a chave k for encontrada, a inserção deve ser abortada pois árvores AVL não podem ter **chaves repetidas**. Caso contrário, atingiremos um ponteiro nulo. Devemos então criar uma nova folha contendo a chave k e fazer o ponteiro que era nulo apontar para tal folha. Naturalmente, os campos filhoesq e filhodir dessa nova folha devem ser nulos e o campo bal deve receber o valor 0.

Precisamos ainda atualizar o bal dos ancestrais dessa nova folha. Isso deve ser feito da seguinte maneira. Se a folha foi inserida à esquerda de seu pai, o bal do pai deve ser incrementado. Se ela foi inserida à direita de seu pai, o bal do pai deve ser decrementado.

Durante uma inserção, se o bal de um nó tornar-se -1 ou 1 será preciso propagar a atualização do bal para o seu nó pai. Se esse nó estiver à esquerda de seu pai, o bal do pai deve ser incrementado, caso contrário, o bal do pai deve ser decrementado. Se o bal de um nó tornar-se 0, a inserção é finalizada.

Por exemplo, ao inserir a chave 3 na árvore da Figura 4.7 teremos que incrementar o bal do nó que contém a chave 7. Como o bal desse nó passará a ser 1, teremos que atualizar o bal do seu nó pai, que contém a chave 15. O bal do nó pai passará a ser 0 e a inserção será finalizada.

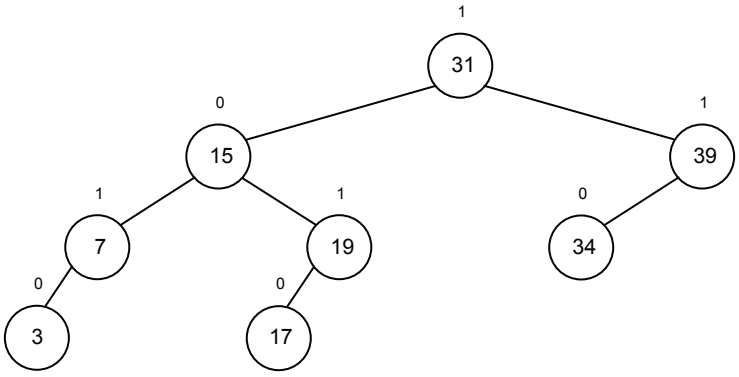


Figura 4.8: Árvore da Figura 4.7 após a inserção da chave 3

Existe ainda mais uma possibilidade a tratar. Se o bal de um nó tornar-se –2 ou 2, o que é inaceitável, será necessário realizar um procedimento nesse nó de modo a restaurar o balanceamento da árvore. Tal procedimento é chamado de rotação.

Em árvores AVL, existem essencialmente dois tipos de rotação: simples e dupla. Essas rotações podem ser à esquerda ou à direita. A figura 4.9 ilustra graficamente a rotação simples à esquerda, também chamada de rotação left-left ou simplesmente rotação LL. Nessa figura, B e C são subárvores de altura H e A é uma subárvore de altura H + 1. Observe que o nó n1 toma o lugar do nó n que então é movido para a direita. A subárvore B é posicionada à esquerda do nó n.

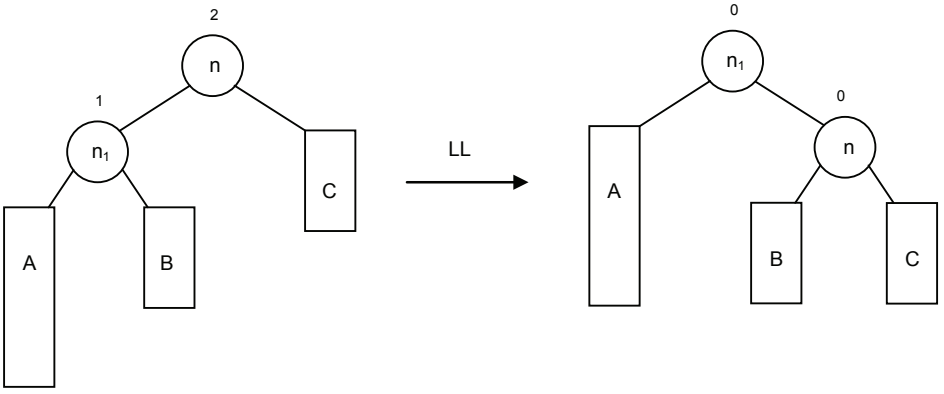


Figura 4.9: Rotação simples à esquerda (LL)

A rotação simples à direita (right-right ou RR) é análoga à rotação LL, conforme mostrado na Figura 4.10.

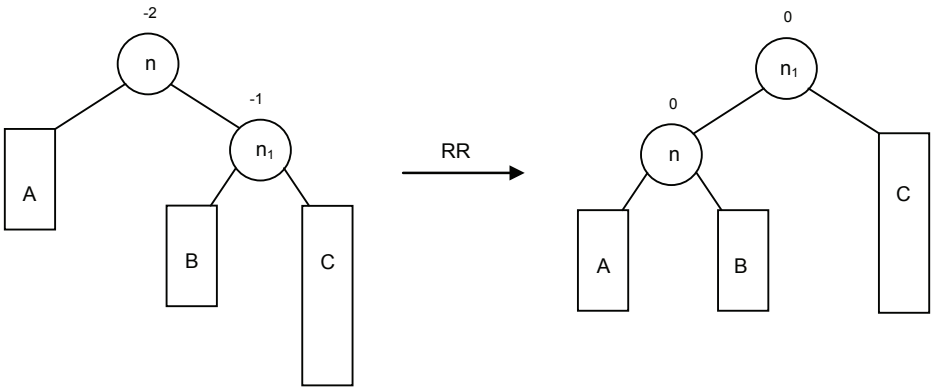


Figura 4.10: Rotação simples à direita (RR)

Temos ainda a rotação dupla à esquerda, também conhecida como rotação left-right ou LR. Na Figura 4.11 as subárvores A, B e D têm altura H + 1 e a subárvore C tem altura H. O nó n2 toma o lugar do nó n que então é movido para a direita. A subárvore B é colocada à direita do nó n1 e a subárvore C é colocada à esquerda do nó n. O nome dessa rotação deve-se ao fato de que ela equivale a fazer uma rotação simples à direita em torno de n2 e depois fazer uma rotação simples à esquerda em torno de n.

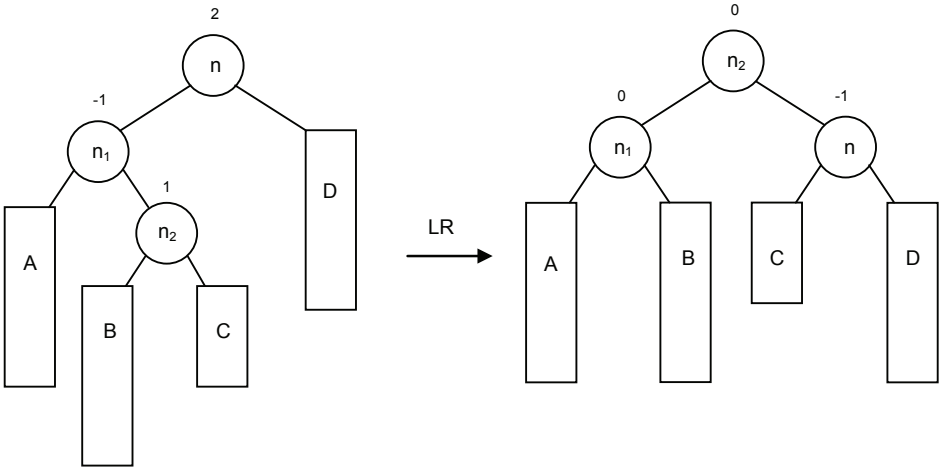


Figura 4.11: Rotação dupla à esquerda (LR)

Na rotação LR a subárvore B poderia ter altura H e a subárvore C poderia ter altura H + 1. A rotação seria feita da mesma forma que foi explicada no parágrafo anterior. Uma única diferença é que ao final da rotação o bal de n1 seria 1 e o bal de n seria 0. Temos ainda o caso especial em que as subárvores B e C são vazias (e portanto H é igual a 1). Mais uma vez, nada muda no procedimento de rotação, exceto pelo fato de que ao final da rotação os nós n1 e n terão bal 0.

A Figura 4.12 ilustra a rotação dupla à direita (right-left ou RL) que é análoga à rotação LR.

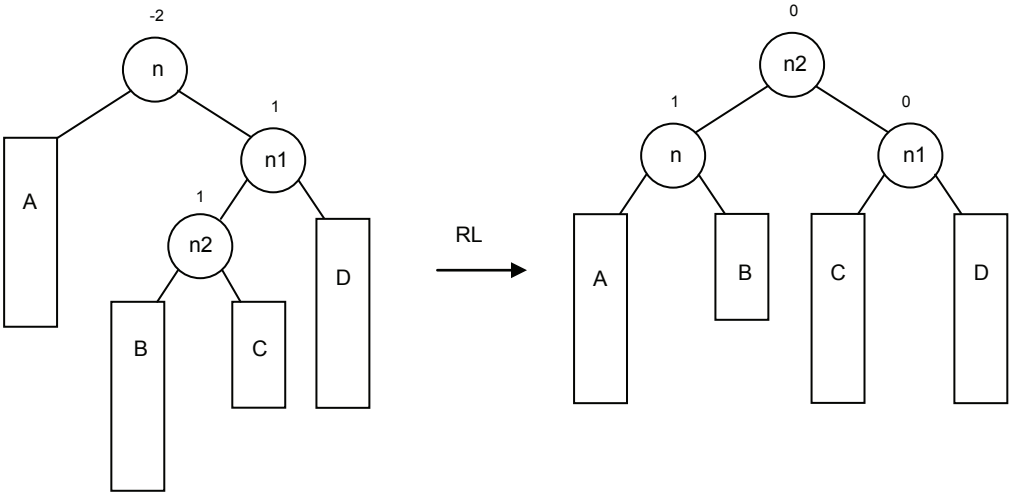


Figura 4.12: Rotação dupla à direita (RL)

Todas essas rotações podem ser feitas em tempo constante, pois exigem apenas o redirecionamento de uma quantidade limitada de ponteiros e a atualização do bal de no máximo três nós. Uma análise cuidadosa das rotações nos permite concluir que além de restaurar o balanceamento, as rotações mantém a ordenação existente na árvore.

Note que em todas as rotações a raiz da subárvore na qual foi feita a rotação passa a ter bal zero. Isso significa que numa operação de inserção será feita no máximo uma rotação.

Na Figura 4.13 mostramos como ficaria a árvore da Figura 4.8 após a inserção da chave 16. Note que o bal do nó que contém a chave 17 seria atualizado para 1 e o bal do nó que contém o 19 passaria a ser 2. Para restaurar o balanceamento foi necessária uma rotação LL em torno desse nó.

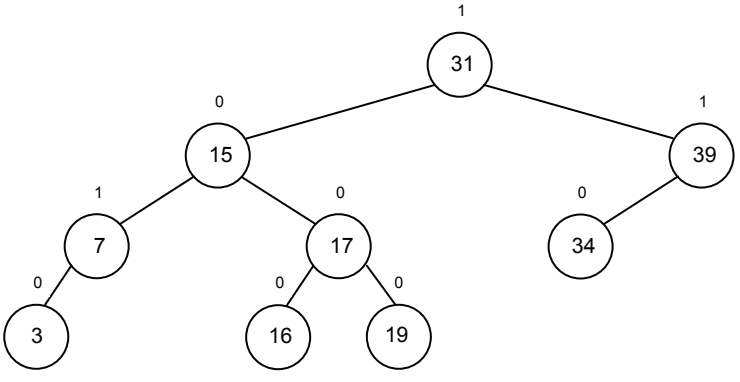


Figura 4.13: Árvore da Figura 4.8 após a inserção da chave 16

Resta-nos discutir a operação de remoção. Infelizmente essa operação é ainda mais complicada que a inserção, razão pela qual vamos tratá-la fazendo distinção entre dois casos.

Abordaremos primeiro a remoção de uma chave k contida numa folha. Naturalmente, tal folha deve ser removida. Em seguida, devemos atualizar o bal dos ancestrais dessa folha. Se a folha estava à esquerda de seu pai, o bal do pai deve ser decrementado. Se ela estava à direita, o bal do pai deve ser incrementado.

Durante uma remoção, se o bal de um nó tornar-se 0 será preciso propagar a atualização do bal para o seu nó pai. Se esse nó estiver à esquerda de seu pai, o bal do pai deve ser decrementado, caso contrário, o bal do pai deve ser incrementado. Se o bal de um nó tornar-se -1 ou 1, a remoção é finalizada. Se o bal de um nó tornar-se –2 ou 2, será necessário realizar uma das quatro rotações explicadas anteriormente.

Convém salientar que é possível que o bal de um nó seja atualizado para 2 e o bal de seu filho esquerdo seja 0. Nesse caso, podemos fazer uma rotação simples ou dupla à esquerda. Analogamente, é possível que o bal de um nó torne-se -2 e o bal de seu filho direito seja 0. Teremos a opção de fazer uma rotação simples ou dupla à direita. Em geral escolhemos fazer a rotação simples.

Note que em todas as rotações a raiz da subárvore na qual foi feita a rotação passa a ter bal zero. Isso significa que na remoção, após uma rotação será preciso propagar a atualização do bal para os seus ancestrais. Isso pode levar à necessidade de múltiplas rotações.

A Figura 4.14 mostra como ficaria a árvore da Figura 4.13 após a remoção da chave 34. Como essa chave estava à esquerda de seu pai, teremos que decrementar o bal de seu pai passará a ser 0. Em seguida, teremos que atualizar o bal da raiz, que passa a ser 2. Teremos então que fazer uma rotação à esquerda, que pode ser simples ou dupla, visto que o filho esquerdo da raiz tem bal 0. Optamos por fazer uma rotação LR para que possamos fazer uma remoção mais interessante em seguida.

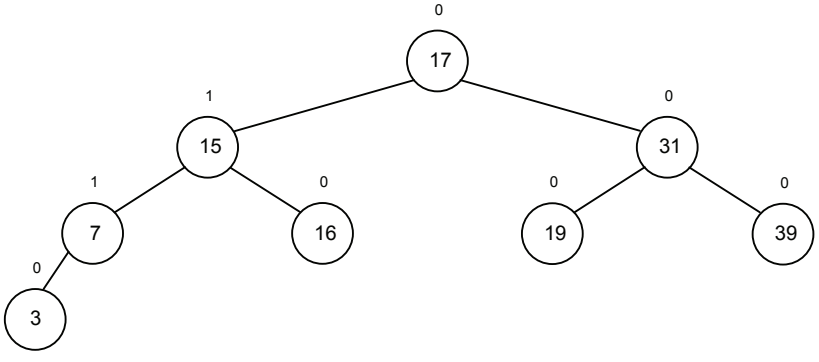


Figura 4.14: Árvore da Figura 4.13 após a remoção da chave 34

Vamos agora analisar a remoção de uma chave k contida em um nó interno. Nesse caso, devemos substituir k por sua **chave antecessora**, que é a maior chave da subárvore à esquerda de k. Encontramos tal chave percorrendo a subárvore à esquerda de k sempre para a direita até atingir um nó que não tenho filho direito.

Se o nó que contém a chave antecessora for uma folha, recaímos no caso que tratamos anteriormente, pois teremos que remover a chave antecessora. Se a chave antecessora não estiver numa folha, devemos substituí-la por seu filho esquerdo, que necessariamente será folha, visto que a chave antecessora não pode ter filho à direita. Como o filho esquerdo da chave antecessora é uma folha, recaímos novamente no caso de remover uma folha e devemos proceder como explicado anteriormente.

Se o nó que contém k não tiver filho esquerdo, devido à propriedade do balanceamento, seu filho direito será folha. Ele deve então tomar o lugar do nó que contém k.

Na Figura 4.15 mostramos como ficaria a árvore da Figura 4.14 após a remoção da chave 17. Observe que essa chave deve ser substituída por sua chave antecessora, que é o 16. Ao remover a folha que contém o 16, precisamos incrementar o bal de seu pai, que passará a ser 2. Precisaremos de uma rotação LL em torno do nó que contém o 15. Após essa rotação, teremos que decrementar o bal da raiz.

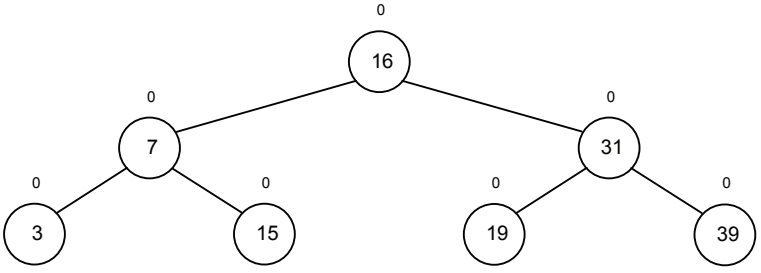


Figura 4.15: Árvore da Figura 4.14 após a remoção da chave 17

Vamos agora analisar a complexidade das operações de busca, inserção e remoção em árvores AVL. No pior caso da operação de busca, é preciso percorrer a árvore a partir da raiz até atingir uma folha que esteja no último nível. Na inserção e na remoção pode ser necessário percorrer a árvore desde a raiz até atingir o último nível e então subir na árvore atualizando o bal dos ancestrais do nó que foi inserido ou removido e, eventualmente, fazendo rotações. Torna-se claro, portanto, que essas operações gastam tempo limitado pela altura da árvore. O seguinte teorema relaciona a altura de uma árvore AVL com a quantidade de chaves contidas na árvore.

Teorema AVL: Seja H a altura de uma árvore AVL que contém n chaves. Então:

$$\log_2(n + 1) \quad H \quad 1{,}44 \log_2(n + 2) - 0{,}328.$$

Esse teorema implica que a complexidade das operações de busca, inserção e remoção em árvores AVL é O(logn). São, portanto, estruturas de pesquisa bastante eficientes.

5.2 Árvores B

Nessa e na próxima seção estudaremos as árvores B e as árvores B+. Tais estruturas são muito utilizadas na prática devido à extrema eficiência com que são feitas as operações de busca, inserção e remoção. A maioria dos sistemas gerenciadores de bancos de dados utiliza essas estruturas, especialmente as árvores B+, para criar arquivos de índices. Elas também são utilizadas por diversos sistemas de arquivos.

As árvores B foram propostas em 1972 por Rudolf Bayer e Edward McCreight, pesquisadores da Boeing Research Labs, e constituem uma generalização das árvores 2-3.

Ao contrário das árvores AVL, cada nó de uma árvore B pode armazenar diversas chaves. Uma característica importante de uma árvore B é a sua ordem. A ordem da árvore determina a quantidade de chaves que um nó pode armazenar e também a quantidade de filhos que um nó pode ter. Uma árvore B de ordem m possui as seguintes propriedades:

- Cada nó da árvore armazena de m a 2m chaves, exceto a raiz, que armazena de 1 a 2m chaves.
- Se um nó interno armazena k chaves então ele tem que ter k + 1 filhos.
- Todas as folhas estão contidas no último nível da árvore.
- A subárvore à esquerda de uma chave x contém apenas chaves menores do que x e a subárvore à direita de x contém apenas chaves maiores do que x.
- Em cada nó da árvore as chaves são mantidas em ordem estritamente crescente.

As três primeiras propriedades garantem o balanceamento da árvore. De fato, as árvores B são extremamente bem balanceadas. Surpreendentemente, não há necessidade de rotações para manter o balanceamento. Isso decorre do fato de que tais árvores crescem "para cima", na direção da raiz, fato incomum entre as árvores.

As outras duas propriedades garantem que as chaves contidas na árvore estejam implicitamente ordenadas. Elas também implicam que árvores B não podem armazenar chaves repetidas.

A estrutura de um nó de uma árvore B de ordem m pode ser constituída dos seguintes campos:

- c: um vetor de chaves com 2m posições.
- p: um vetor de ponteiros com 2m + 1 posições.
- numchaves: indica a quantidade de chaves contidas no nó.
- pai: ponteiro para o nó pai.

Os campos numchaves e pai, embora não sejam imprescindíveis, facilitam sobremaneira a implementação de diversas operações em árvores B. Em um nó da árvore, o ponteiro armazenado em p[i] aponta para a subárvore à esquerda da chave armazenada em c[i] e p[i + 1] aponta para a subárvore à direita de c[i]. A Figura 4.16 mostra como poderia ser a estrutura de um nó de uma árvore B de ordem 2.

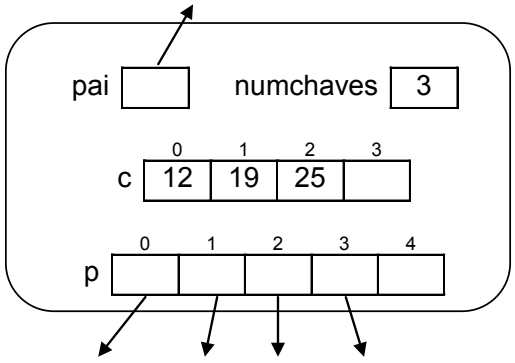


Figura 4.16: Estrutura de um nó de uma árvore B

A Figura 4.17 mostra um exemplo de árvore B de ordem 1.

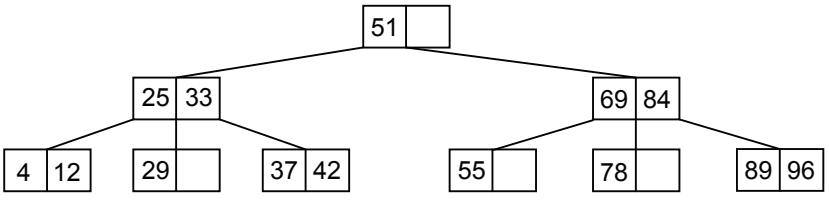


Figura 4.17: Exemplo de árvore B

A busca de uma chave em árvores B é simples. Para procurar uma chave k, primeiramente verificamos se a árvore é vazia. Em caso afirmativo, a busca para, tendo sido mal sucedida. Caso contrário, verificamos se k ocorre na raiz da árvore. Nesse caso, a busca tem sucesso. Caso contrário, determinamos a menor chave contida na raiz que é maior do que k. Se essa chave existir, repetimos o procedimento, recursivamente, na subárvore à esquerda dessa chave. Se k for maior do que todas as chaves contidas na raiz, repeti-

mos o procedimento, recursivamente, na subárvore à direita da maior chave contida na raiz.

Na árvore da Figura 4.17, para buscar a chave 37 teremos que inspecionar a raiz. Como a chave 37 é menor do que 51, devemos seguir o ponteiro para a subárvore à esquerda do 51, chegando ao nó que contém as chaves 25 e 33. Como 37 é maior do que essas chaves, devemos seguir o ponteiro para a subárvore à esquerda do 33, chegando ao nó que contém o 37. Se quisermos procurar a chave 30, começaremos inspecionando a raiz. Como a chave 30 é menor do que 51, seguiremos para o nó que contém as chaves 25 e 33. Como 33 é a menor chave desse nó que é maior do que 30, devemos seguir o ponteiro para a subárvore à esquerda do 33, chegando ao nó que contém o 29. Visto que 30 é maior do que 29, devemos seguir o ponteiro para a subárvore à direita do 29. Observe que tal subárvore é vazia e, portanto, a busca será mal sucedida.

Vamos agora explicar como fazer a inserção de chaves em árvores B. Para inserir uma chave k numa árvore B de ordem m devemos percorrê-la, a partir da raiz, como se estivéssemos procurando k. Se a chave k for encontrada, a inserção deve ser abortada, pois árvores B não podem ter chaves repetidas. Caso contrário, atingiremos uma folha. Se essa folha tiver menos do que 2m chaves, basta inserir k de modo que as chaves contidas nessa folha continuem em ordem crescente.

Suponha agora que a folha contenha 2m chaves. Nesse caso, a folha precisará ser subdividida, criando-se uma nova folha. Distribuiremos a chave k e as chaves contidas nessa folha da seguinte maneira. As m menores chaves continuarão na folha, as m maiores chaves serão movidas para a nova folha e a chave central (aquela que não está entre as m menores nem entre as m maiores chaves) deve "subir" para o nó pai. Note que se o nó pai tiver 2m chaves, ele também precisará ser subdividido.

Esse processo de subdivisão pode propagar-se até a raiz da árvore. Se a raiz for subdividida, a chave central será armazenada numa nova raiz e a altura da árvore aumentará. A Figura 4.18 mostra como seria a subdivisão da folha esquerda ao inserir a chave 12.

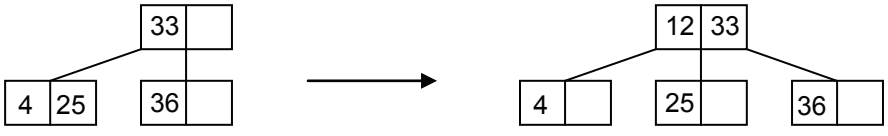


Figura 4.18: Subdivisão de uma folha numa árvore B

Na Figura 4.19 exibimos a árvore da Figura 4.17 após a inserção das chaves 73 e 45, nessa ordem. Note que a chave 73 deve ser inserida na folha que contém o 78. Já a chave 45 deve ser inserida na folha que contém as chaves 37 e 42. Como essa folha está cheia (armazena 2m chaves) ela precisará ser subdividida. A chave 42 (chave central) deverá ser movida para o nó pai. Como o nó pai também está cheio, ele também precisará ser subdividido. A chave 33 será movida para a raiz da árvore e a inserção será finalizada.

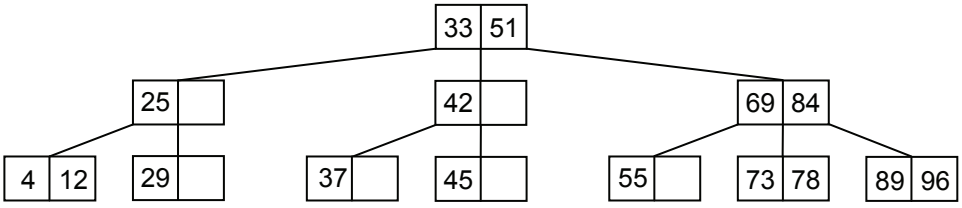


Figura 4.19: Árvore B da Figura 4.17 após a inserção das chaves 73 e 45

Observe que se inserirmos a chave 80 na árvore da Figura 4.19, teremos que subdividir a folha que contém o 73 e o 78, seu nó pai, que contém o 69 e o 84, e a raiz da árvore. Com isso, a árvore passará a ter altura 4.

Vamos agora a discutir operação de remoção. Para remover uma chave k de uma árvore B de ordem m precisamos encontrar o nó da árvore que contém k. Vamos tratar primeiramente o caso em que tal nó é uma folha. Se essa folha tiver mais do que m chaves ou for a raiz da árvore, basta remover k apropriadamente dessa folha.

Suponha agora que a folha que contém k não é a raiz da árvore e armazena exatamente m chaves. Após remover k dessa folha, devemos tentar obter mais uma chave para essa folha de modo que ele continue armazenando m chaves. Para isso, a folha irmã à esquerda (preferencialmente) ou a folha irmã à direita deve doar uma chave. Essa chave é movida para o nó pai e a chave do nó pai que está entre as folhas envolvidas na doação deve ser movida para a folha que continha k. A Figura 4.20 ilustra a doação de uma chave da folha irmã à esquerda.

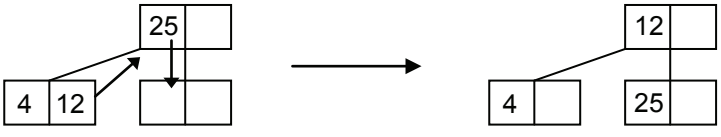


Figura 4.20: Doação de uma chave numa remoção em árvore B

Se nem a folha irmã à esquerda nem a folha irmã à direita tiver mais do que m chaves, será necessário fundir a folha que contém k com a folha irmã à sua

esquerda (preferencialmente) ou com a folha irmã à sua direita. A chave do nó pai que está entre as folhas que estão se fundindo deve ser movida para a nova folha que será obtida com a fusão. A Figura 4.21 ilustra a fusão de duas folhas.

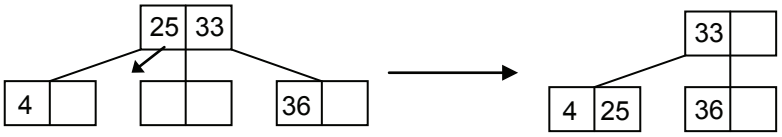


Figura 4.21: Fusão de duas folhas de uma árvore B

Observe que ao fundir duas folhas o nó pai dessas folhas perde uma chave. Se ele não for a raiz e tiver exatamente m chaves será necessário obter a doação de uma chave do seu nó irmão à esquerda (preferencialmente) ou do seu nó irmão à direita. Se nenhum desses dois nós irmãos puder doar uma chave, devemos fundir o nó pai com o seu nó irmão à esquerda (preferencialmente) ou com o seu nó irmão à direita.

Esse processo de subdivisão pode propagar-se até os filhos da raiz da árvore. Se for necessário fundir os únicos dois filhos da raiz, a raiz antiga deixará de existir, o nó obtido com a fusão passará a ser a nova raiz e a altura da árvore diminuirá.

Na Figura 4.22 exibimos a árvore da Figura 4.19 após a remoção das chaves 4 e 55, nessa ordem. Note que a folha que contém a chave 4 tem mais do que m chaves e, portanto, basta remover tal chave dessa folha. Já a remoção da chave 55 leva à necessidade de obter a doação de uma chave da folha irmã à sua direita.

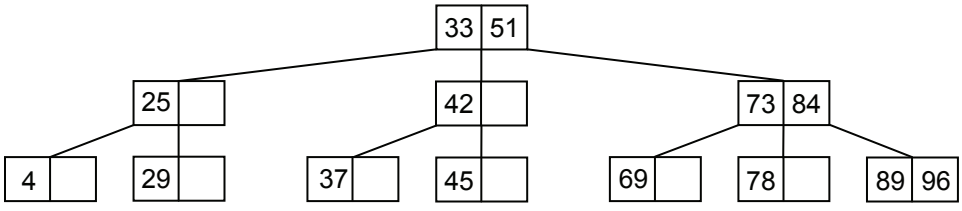


Figura 4.22: Árvore B da Figura 4.19 após a remoção das chaves 4 e 55

Ao removermos a chave 45 da árvore da Figura 4.22 teremos que fundir a folha que contém tal chave com à folha irmã à esquerda. O nó pai das folhas que serão fundidas precisará obter uma doação do nó irmão à direita. A árvore resultante é mostrada na Figura 4.23.

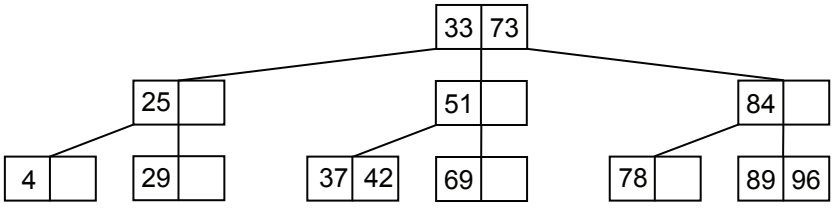


Figura 4.23: Árvore B da Figura 4.22 após a remoção da chave 45

Vamos agora tratar o caso em que a chave k, a ser removida, está em um nó interno. Nesse caso, devemos substituir k por sua chave antecessora, que é a maior chave da subárvore à esquerda de k. Observe que tal chave necessariamente estará numa folha e, portanto, recaímos no caso de remoção de uma chave. Devemos então proceder como explicado nos parágrafos anteriores.

Por exemplo, se quisermos remover a chave 33 da árvore mostrada na Figura 4.23, teremos que substituí-la pela chave 29, que é sua chave antecessora. A folha que continha o 29 ficará vazia e terá que ser fundida com sua folha irmã à esquerda e a chave 25 descerá para a folha obtida com a fusão. Precisaremos fazer mais uma fusão envolvendo o nó onde estava a chave 25 e o nó que contém o 51. Com isso, a chave 29 que foi para a raiz deverá descer para o nó obtido nessa última fusão.

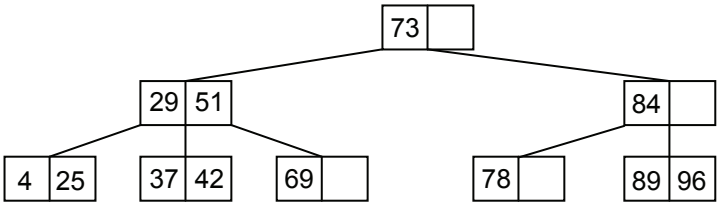


Figura 4.24: Árvore B da Figura 4.23 após a remoção da chave 33

Deve ter ficado claro que a operação de busca requer que a árvore seja percorrida a partir da raiz até, no máximo, atingir uma folha. Dessa forma, tal operação gasta tempo linearmente proporcional à altura da árvore, no pior caso. Na inserção e na remoção é necessário percorrer a árvore da raiz até atingir o último nível e, eventualmente, subir na árvore realizando subdivisões ou fusões. Fica claro, portanto, que essas operações gastam tempo limitado pela altura da árvore. O teorema a seguir relaciona a altura de uma árvore B com a quantidade de chaves contidas na árvore.

Teorema de Bayer-McCreight: Seja H a altura de uma árvore B de ordem m que contém n chaves. Então:

$\lfloor \log_{2m}(n + 1) \rfloor \quad H \quad \lceil \log_{m+1}(n + 1) \rceil$.

Esse teorema implica que a complexidade temporal das operações de busca, inserção e remoção em árvores B é O(log n). Vemos ainda que quanto maior a ordem da árvore, menor será sua altura. Na prática, em comum utilizar ordens bem maiores do que 1, fazendo com que a altura da árvore seja muito pequena. Por exemplo, se a ordem da árvore B for 50 e ela armazenar 1 bilhão de chaves, sua altura será no máximo 5. São, portanto, estruturas de pesquisa extremamente eficientes.

5.3 Árvores B+

As árvores B+, também propostas por Bayer e McCreight, são bem parecidas com as árvores B, tendo apenas duas diferenças mais significativas:

- Todas as chaves válidas contidas na árvore têm que aparecer em alguma folha da árvore.
- Cada folha possui um ponteiro que aponta para a folha imediatamente à sua direita.

Convém salientar que as chaves contidas nos nós internos servem apenas para orientar o caminhamento na árvore. A primeira diferença é mais importante e requer o relaxamento de uma das propriedades das árvores B, descrita no início da Subseção 5.2:

- A subárvore à esquerda de uma chave x contém apenas chaves menores ou iguais a x e a subárvore à direita de x contém apenas chaves maiores do que x.

A Figura 4.25 mostra uma árvore B+ de ordem 1 com as mesmas chaves contidas na árvore B da Figura 4.24.

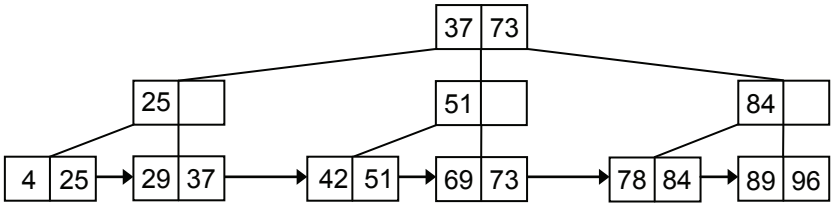


Figura 4.25: Exemplo de árvore B+

Como numa árvore B+ uma chave só é válida se aparece em alguma folha, em toda operação de busca devemos percorrer a árvore a partir da raiz até atingir uma folha. Para procurar uma chave k, primeiramente verificamos se a árvore é vazia. Em caso afirmativo, a busca para, tendo sido mal sucedida. Caso contrário, verificamos se a raiz da árvore é folha. Se esse for o caso, verificamos se x ocorre na raiz. Se for este o caso, a busca tem sucesso; caso contrário, a busca é mal sucedida. Se a raiz não for uma folha, determinamos a menor chave contida na raiz que é maior ou igual a k. Se essa chave existir, repetimos o procedimento, recursivamente, na subárvore à esquerda dessa chave. Se k for maior do que todas as chaves contidas na raiz, repetimos o procedimento, recursivamente, na subárvore à direita da maior chave contida na raiz.

Para buscar a chave 29 na árvore da Figura 4.25 teremos que inspecionar inicialmente a raiz. Como 29 é menor do que 37, devemos seguir o ponteiro para a subárvore à esquerda do 37, chegando ao nó que contém a

chave 25. Visto que 29 é maior do que 25, devemos seguir o ponteiro para a subárvore à direita do 25, chegando à folha contém o 29. Se quisermos procurar a chave 75, começaremos inspecionando a raiz. Como 75 é maior do que 73, seguiremos para o nó que contém a chave 84. Visto que 75 é menor do que 84, devemos seguir o ponteiro para a subárvore à esquerda do 84, chegando à folha que contém o 78 e o 84. Tal folha não contém o 75 e, portanto, a busca é mal sucedida.

A inserção em árvores B+ é similar à inserção em árvores B, com apenas uma diferença significativa: ao subdividir uma folha, a chave central deve ser copiada para o nó pai. Dessa maneira, a chave central continua a ser uma chave válida, pois ela permanece numa folha. Por exemplo, ao inserir a chave 75 na árvore da Figura 4.25, a folha que contém as chaves 78 e 84 precisará ser subdividida. A chave 78 será então copiada para o nó pai. A árvore resultante obtida com essa inserção é mostrada na Figura 4.26.

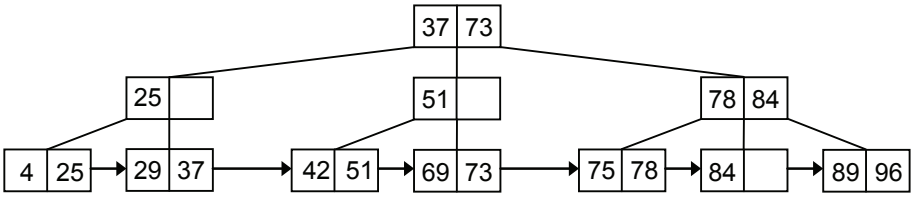


Figura 4.26: Árvore B+ da Figura 4.25 após a inserção da chave 75

A remoção em árvores B+ também é similar à remoção em árvores B, com apenas duas diferenças mais significativas. Uma delas está na forma com é feita a doação de uma chave contida numa folha.

Por exemplo, ao fazer a doação de uma chave da folha irmã à esquerda, a segunda maior chave da folha que está doando é copiada para o nó pai e ocupa o lugar da chave que estava entre as folhas envolvidas na doação. A maior chave da folha que está doando é então movida para a folha que está recebendo a doação. A Figura 4.27 ilustra essa diferença. Note que a chave 75 foi copiada para o nó pai e o 78 foi movido para a folha que antes continha o 84. Observe que o 84 ainda aparece na árvore, mas não numa folha. Dessa forma, o 84 não é mais uma chave válida da árvore.

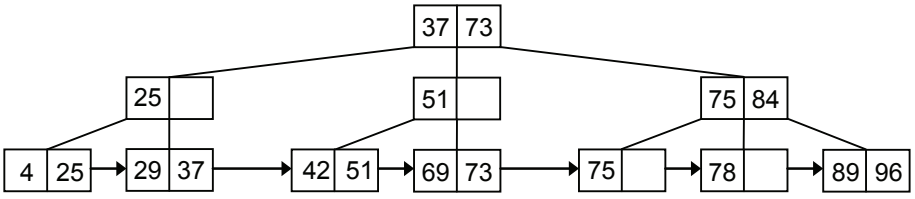


Figura 4.27: Árvore B+ da Figura 4.26 após a remoção da chave 84

A outra diferença ocorre na fusão de duas folhas. Nesse caso, a chave do nó pai que está entre elas simplesmente é eliminada. Observe na Figura 4.28 como fica a árvore da Figura 4.27 após a remoção da chave 75.

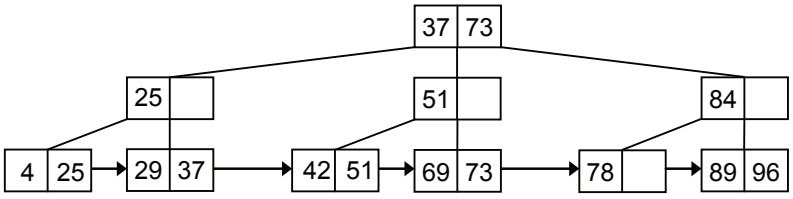


Figura 4.28: Árvore B+ da Figura 4.27 após a remoção da chave 75

É possível mostrar que uma árvore B+ que contenha as mesmas chaves válidas que uma árvore B terá no máximo um nível a mais do que a árvore B. Sendo assim, pelo Teorema de Bayer-McCreight, concluímos que a altura de uma árvore B+ é logarítmica na quantidade de chaves contidas na árvore.

Claramente, as operações de busca, inserção e remoção em árvores B+ gastam tempo linearmente proporcional à altura da árvore. Sendo assim, tais operações são feitas em tempo $\Theta(\log n)$. Assim como as árvores B, as árvores B+ também são estruturas de pesquisa extremamente eficientes.

Atividades de avaliação

1. Escreva uma versão não recursiva do algoritmo de busca binária.
2. Escreva uma função que receba um vetor em ordem crescente e um valor x e devolva o índice da posição do vetor cujo conteúdo é mais próximo de x. Sua função deverá requerer tempo logarítmico no tamanho do vetor (sugestão: adapte o algoritmo de busca binária).
3. Mostre como ficaria uma tabela de hashing fechado com 13 posições, após a inserção das chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nesta ordem (nessa e na próxima questão, os valores associados às chaves devem ser ignorados). Utilize a seguinte função de hashing: h(x) = x mod 13. Em seguida, remova as chaves 46, 8 e 74, nesta ordem, e mostre como ficaria a tabela.
4. Mostre como ficaria uma tabela de hashing aberto com 7 posições, após a inserção das chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nesta ordem. Utilize a seguinte função de hashing: h(x) = x mod 7. Em seguida, remova as chaves 15, 23 e 51, nesta ordem, e mostre como ficaria a tabela.

5. Explique o que é a carga de uma tabela de hashing e diga quando ela é considerada baixa. Explique também o que é uma boa função de hashing.
6. Escreva um algoritmo que receba uma tabela de hashing fechado (passada por referência) e uma chave k e então insira essa chave na tabela, se ela ainda não existir na tabela.
7. Insira as chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nessa ordem, numa árvore AVL. Em seguida, remova as chaves 83, 69 e 9, nessa ordem. Após cada inserção ou remoção desenhe como ficou a árvore, incluindo o bal de cada nó. Mencione também as rotações utilizadas nas inserções e remoções.
8. Escreva um algoritmo que receba um ponteiro para a raiz de uma árvore AVL e imprima o conteúdo dos nós da árvore em ordem decrescente.
9. Escreva uma função que receba um ponteiro para a raiz de uma árvore AVL e devolva a altura da árvore. Sua função deverá requerer tempo logarítmico no tamanho da árvore.
10. Escreva um algoritmo que receba um ponteiro para a um nó de uma árvore AVL e realize uma rotação simples à direita em torno desse nó.
11. Mostre como ficaria uma árvore B de ordem 1 após a inserção das chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nesta ordem. Em seguida, remova as chaves 24, 33, e 12, nessa ordem, e mostre como ficaria a árvore.
12. Mostre como ficaria uma árvore B+ de ordem 1 após a inserção das chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nesta ordem. Em seguida, remova as chaves 46, 8 e 74, nessa ordem, e mostre como ficaria a árvore.
13. Escreva uma função que receba um ponteiro para a raiz de uma árvore B+ e devolva a quantidade de chaves contidas na árvore.
14. Discorra sobre a eficiência das operações de inserção, remoção, busca, busca aproximada e listagem em ordem em árvores B e B+.

Leituras, filmes e sites

Sites

http://pt.wikipedia.org/wiki/Bubble_sort

http://pt.wikipedia.org/wiki/Insertion_sort

http://pt.wikipedia.org/wiki/Selection_sort

http://pt.wikipedia.org/wiki/Shell_sort

http://pt.wikipedia.org/wiki/Merge_sort

http://pt.wikipedia.org/wiki/Quick_sort

http://pt.wikipedia.org/wiki/Heapsort

http://pt.wikipedia.org/wiki/Count_sort

http://pt.wikipedia.org/wiki/Bucket_sort

http://pt.wikipedia.org/wiki/Radix_sort

http://pt.wikipedia.org/wiki/Busca_binária

http://pt.wikipedia.org/wiki/Tabela_de_dispersão

http://pt.wikipedia.org/wiki/Árvore_AVL

http://pt.wikipedia.org/wiki/Árvore_B

http://pt.wikipedia.org/wiki/Árvore_B+

http://pt.wikipedia.prg/wiki/Sort-merge_utility

http://www.cs.pitt.edu/~kirk/cs1501/animations/Sort1.html

http://slady.net/java/bt/view.php?w=750&h=500

Referências

Ascencio A.F.G., Aplicações de **Estrutura de Dados em Delphi.** São Paulo: Pearson Prentice Hall, 2005.

Cormen. T.H., C.E. Leiserson, R.L. Rivest, and C. Stein, **Algoritmos: Teoria e Prática.** Rio de Janeiro: Editora Campus, 2002.

Feofiloff, P. **Algoritmos em Linguagem C.** Elsevier, 2008.

Horowitz. E. and S. Sahni. **Fundamentos de Estrutura de Dados.** Rio de Janeiro: Editora Campus, 1987.

Tenenbaum, A.M., Y. Langsam and M.J. Augenstein. **Estrutura de Dados usando C.** São Paulo: Pearson Makron Books, 1996.

Ziviani, N. **Projeto de Algoritmos.** 2.ed., São Paulo: Thomson, 2004.

_____. **Projeto de Algoritmos com Implementações em Java e C++.** Cengage Learning, 2006.

Sobre os autores

Gerardo Valdisio Rodrigues Viana Possui Graduação em Engenharia Mecânica pela UFC - Universidade Federal do Ceará, Licenciatura em Matemática pela UECE - Universidade Estadual do Ceará, Especialização, Mestrado e Doutorado em Ciência da Computação pela UFC com período no IME/USP - Instituto de Matemática e Estatística da Universidade de São Paulo. Aposentou-se como Professor Associado da UFC em Agosto de 2010. Atualmente é Professor Adjunto da UECE. Nos últimos anos publicou 11 artigos em periódicos especializados e 6 trabalhos completos e 7 simples em anais de congressos. Possui 2 livros publicados e participou da elaboração de 1 capítulo de livro. Orienta trabalhos de conclusão de curso (monografias de graduação) e dissertações de mestrado na área de computação. Atua nas áreas de Matemática Computacional, Otimização Combinatória, Análise de Algoritmos e Bioinformática. Em suas atividades profissionais interagiu com diversos colaboradores em co-autorias de trabalhos científicos. Em seu currículo Lattes os termos mais freqüentes na contextualização da produção científica, tecnológica e artístico-cultural são: Otimização Combinatória, Programação Matemática, Metaheuristicas, Computação Paralela e Biologia Computacional.

Glauber Ferreira Cintra possui graduação em Ciência da Computação pela Universidade Estadual do Ceará, mestrado em Matemática Aplicada pela Universidade de São Paulo e doutorado em Ciência da Computação pela Universidade de São Paulo. Atualmente é professor do Instituto Federal de Educação, Ciência e Tecnológia do Ceará e professor da Faculdade 7 de Setembro. Tem experiência na área de ciência da computação, com ênfase em otimização combinatória, atuando principalmente nos seguintes temas: problemas de corte e empacotamento, geração de colunas, algoritmos de aproximação e programação matemática.

Ricardo Holanda Nobre possui Graduação em Ciências da Computação pela Universidade Estadual do Ceará e em Direito pela Universidade Federal do Ceará. É Especialista em Direito Empresarial, Mestre em Computação pela Universidade Estadual do Ceará e Doutorando do curso de Engenharia de Teleinformática da Universidade Federal do Ceará. Atua nas áreas de Otimização Combinatória, Análise de Algoritmos, Computação Paralela e Análise de Imagens. Foi Gerente de TIC da Caixa de Assistência dos Funcionários do Banco do Nordeste (CAMED) e Gerente de Desenvolvimento em TIC da Secretaria da Justiça e Cidadania do Estado do Ceará, desenvolvendo trabalhos de identificação biométrica, *Business Intelligence*, *Data Warehouse* dentre outros. Atualmente é Professor da Faculdade de Tecnologia do Nordeste, Professor Colaborador e Pesquisador da Universidade Estadual do Ceará e Analista de Sistemas do SERPRO, no qual desenvolve atividades ligadas a Banco de Dados e a *Data Warehouse*.

Anexo

Relação de Algoritmos

1 - Algoritmo da Bolha - bubble_sort.c
2 - Algoritimo da Inserção - insertion_sort.c
3 - Algortmo da Seleção - selection_sort.c
4 - Algoritmo Shellsort - shell_sort.c
5 - Algoritmo Mergesort - merge_sort.c
6 - Algoritmo Quicksort - quick_sort.c
7 - Algoritmo Heapsort - heap_sort.c
8 - Algoritmo Counting - counting_sort.c
9 - Algoritmo Bucketsort - bucket_sort.c
10 - Algoritmo Radixsort - radix_sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <stdbool.h>

int main() {
    int vetor[5], i, aux;

   bool parar = false;

   for(i=0;i<5;++i)
   {
       printf("Digite o valor: ");
       scanf("%d",&vetor[i]);
   }

   while(parar == false)
   {
       parar = true;

       for(i=0;i<4;++i)
       {
           if(vetor[i]>vetor[i+1])
           {
               parar = false;
               aux = vetor[i];
               vetor[i] = vetor[i+1];
               vetor[i+1] = aux;
           }
       }
   }

   for(i=0;i<5;++i)
   {
       printf("%d\n",vetor[i]);
   }

   getch();
   return 0;
}
```

1 - Algoritmo da Bolha - bubble_sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void insertionSort(int *V, int tam)
{
     int i, j, aux;

     for(i = 1; i < tam; i++){

         j = i;

         while((V[j] < V[j - 1])&&(j!=0)) {
             aux = V[j];
             V[j] = V[j - 1];
             V[j - 1] = aux;
             j--;
         }
     }
}

int main() {
   int vet[5], i, aux;

   for(i=0;i<5;++i)
   {
       printf("Digite o valor: ");
       scanf("%d",&vet[i]);
   }

   insertionSort(vet, 5);

   for(i=0;i<5;++i)
   {
       printf("%d\n",vet[i]);
   }

   getch();
}
```

2 - Algoritimo da Inserção - insertion_sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void selectionsort(int * vet, int tam)
{
  int i, j, min;
  for (i = 0; i < (tam-1); i++)
    {
     min = i;
     for (j = (i+1); j < tam; j++) {
       if(vet[j] < vet[min]) {
         min = j;
       }
     }
     if (i != min) {
       int swap = vet[i];
       vet[i] = vet[min];
       vet[min] = swap;
     }
  }
}

int main() {
   int vetor[5], i, aux;

   for(i=0;i<5;++i)
   {
       printf("Digite o valor: ");
       scanf("%d",&vetor[i]);
   }

   selectionsort(vetor, 5);

   for(i=0;i<5;++i)
   {
       printf("%d\n",vetor[i]);
   }
   getch();
}
```

3 - Algoritmo da Seleção - selection_sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void shellsort(int * vet, int size) {
    int i , j , value;
    int gap = 1;

    do {
        gap = 3*gap+1;
    } while(gap < size);

    do {
         gap /= 3;
         for(i = gap; i < size; i++) {
                 value =vet[i];
                 j = i - gap;
                 while (j >= 0 && value < vet[j]) {
                         vet [j + gap] =vet[j];
                         j -= gap;
                 }
                 vet [j + gap] = value;
         }
    } while ( gap > 1);
}


int main() {
   int vetor[5], i, aux;

   for(i=0;i<5;++i)
   {
       printf("Digite o valor: ");
       scanf("%d",&vetor[i]);
   }

   shellsort(vetor, 5);

   for(i=0;i<5;++i)
   {
       printf("%d\n",vetor[i]);
   }
   getch();
}
```

4 - Algoritmo Shellsort - shell_sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void merge(int vec[], int vecSize) {
  int mid;
  int i, j, k;
  int* tmp;

  tmp = (int*) malloc(vecSize * sizeof(int));
  if (tmp == NULL) {
    exit(1);
  }

  mid = vecSize / 2;

  i = 0;
  j = mid;
  k = 0;
  while (i < mid && j < vecSize) {
    if (vec[i] < vec[j]) {
      tmp[k] = vec[i];
      ++i;
    }
    else {
      tmp[k] = vec[j];
      ++j;
    }
    ++k;
  }

  if (i == mid) {
    while (j < vecSize) {
      tmp[k] = vec[j];
      ++j;
      ++k;
    }
  }
  else {
    while (i < mid) {
      tmp[k] = vec[i];
```

```
        ++i;
        ++k;
      }
    }

    for (i = 0; i < vecSize; ++i) {
      vec[i] = tmp[i];
    }

    free(tmp);
}

void mergeSort(int vec[], int vecSize) {
    int mid;

    if (vecSize > 1) {
      mid = vecSize / 2;
      mergeSort(vec, mid);
      mergeSort(vec + mid, vecSize - mid);
      merge(vec, vecSize);
    }
}

int main() {
   int vec[5], i, aux;

   for(i=0;i<5;++i)
   {
       printf("Digite o valor: ");
       scanf("%d",&vec[i]);
   }

   mergeSort(vec, 5);

   for(i=0;i<5;++i)
   {
       printf("%d\n",vec[i]);
   }
   getch();
}
```

5 - Algoritmo Mergesort - merge_sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void swap(int* a, int* b) {
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}

int partition(int vec[], int left, int right) {
  int i, j;

  i = left;
  for (j = left + 1; j <= right; ++j) {
    if (vec[j] < vec[left]) {
      ++i;
      swap(&vec[i], &vec[j]);
    }
  }
  swap(&vec[left], &vec[i]);

  return i;
}

void quickSort(int vec[], int left, int right) {
  int r;

  if (right > left) {
    r = partition(vec, left, right);
    quickSort(vec, left, r - 1);
    quickSort(vec, r + 1, right);
  }
}

int main() {
   int vec[5], i, aux;

   for(i=0;i<5;++i)
   {
       printf("Digite o valor: ");
       scanf("%d",&vec[i]);
   }

   quickSort(vec, 0, 5);

   for(i=0;i<5;++i)
   {
       printf("%d\n",vec[i]);
   }
   getch();
}
```

6 - Algoritmo Quicksort - quick_sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void heapsort(int vec[], int n)
{
   int i = n/2, pai, filho;
   int t;

   for (;;)
   {
      if (i > 0)
      {
          i--;
          t = vec[i];
      }
      else
      {
          n--;
          if (n == 0)
             return;
          t = vec[n];
          vec[n] = vec[0];
      }

      pai = i;
      filho = i*2 + 1;

      while (filho < n)
      {
          if ((filho + 1 < n)  &&  (vec[filho + 1] > vec[filho]))
              filho++;
          if (vec[filho] > t)
          {
             vec[pai] = vec[filho];
             pai = filho;
             filho = pai*2 + 1;
          }
```

```
                else
                    break;
            }
            vec[pai] = t;
        }
    }


    int main() {
        int vec[5], i, aux;

        for(i=0;i<5;++i)
        {
            printf("Digite o valor: ");
            scanf("%d",&vec[i]);
        }

        heapsort(vec, 5);

        for(i=0;i<5;++i)
        {
            printf("%d\n",vec[i]);
        }
        getch();
    }
```

7 - Algoritmo Heapsort - heap_sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void countingsort(int *vet, int n, int min, int max)
{
 int i, j, z;
 int range = max - min + 1;
 int *count = malloc(range * sizeof(*vet));

 for(i = 0; i < range; i++)
         count[i] = 0;

 for(i = 0; i < n; i++)
         count[ vet[i] - min ]++;

 for(i = min, z = 0; i <= max; i++)
 {
         for(j = 0; j < count[i - min]; j++)
         {
                 vet[z++] = i;
         }
 }
 free(count);
}


int main() {
 int vet[5], i, maior;

 for(i=0;i<5;++i)
 {
       printf("Digite o valor: ");
       scanf("%d",&vet[i]);
 }

    printf("Qual o maior elemento: ");
    scanf("%d",&maior);

    countingsort(vet, 5, 0, maior );

    for(i=0;i<5;++i)
    {
       printf("%d\n",vet[i]);
    }
    getch();
}
```

8 - Algoritmo Counting - counting_sort.c

```
#include <stdio.h>
 #include <stdlib.h>
 #include <conio.h>
 #define n 5
 #define num_bucket 3

 typedef struct {
     int topo;
     int balde[n];
  } bucket;

  void bubble(int *vet,int tam){
     int i,j,temp,flag;
     if(tam){
         for(j=0;j<tam-1;j++){
             flag=0;
             for(i=0;i<tam-1;i++){
                 if(vet[i+1]<vet[i]){
                     temp=vet[i];
                     vet[i]=vet[i+1];
                     vet[i+1]=temp;
                     flag=1;
                 }
             }
             if(!flag)
                 break;
         }
  }
  }

  void bucketsort(int *vet, int tam){
     bucket b[num_bucket];
     int i,j,k;
     for(i=0;i<num_bucket;i++)
         b[i].topo=0;

     for(i=0;i<tam;i++){
         j=(num_bucket)-1;
         while(1){
             if(j<0)
                 break;
             if(vet[i]>=j*10){
                 b[j].balde[b[j].topo]=vet[i];
                 (b[j].topo)++;
```

```
                    break;
                }
                j--;
            }
        }

        for(i=0;i<num_bucket;i++)
            if(b[i].topo)
                bubble(b[i].balde,b[i].topo);

        i=0;
        for(j=0;j<num_bucket;j++){
            for(k=0;k<b[j].topo;k++){
                vet[i]=b[j].balde[k];
                i++;
            }
        }
    }


int main() {
 int vet[n], i, maior;

 for(i=0;i<n;++i)
 {
          printf("Digite o valor: ");
          scanf("%d",&vet[i]);
 }

    bucketsort(vet, n);

    for(i=0;i<n;++i)
    {
         printf("%d\n",vet[i]);
    }
    getch();
}
```

9 - Algoritmo Bucketsort - bucket_sort.c

```
#include <limits.h>
#include <stdlib.h>
#include <conio.h>

typedef unsigned uint;
#define swap(a, b) { tmp = a; a = b; b = tmp; }

/* sort unsigned ints */
static void rad_sort_u(uint *from, uint *to, uint bit)
{
 if (!bit || to < from + 1) return;

 uint *ll = from, *rr = to - 1, tmp;
 while (1) {
          /* find left most with bit, and right most without bit, swap */
          while (ll < rr && !(*ll & bit)) ll++;
          while (ll < rr &&  (*rr & bit)) rr--;
          if (ll >= rr) break;
          swap(*ll, *rr);
 }

 if (!(bit & *ll) && ll < to) ll++;
 bit >>= 1;

 rad_sort_u(from, ll, bit);
 rad_sort_u(ll, to, bit);
}

/* sort signed ints: flip highest bit, sort as unsigned, flip back */
static void radix_sort(int *a, const size_t len)
{
 size_t i;
 uint *x = (uint*) a;

    for(i=0;i<len;++i)
         x[i] ^= INT_MIN;

 rad_sort_u(x, x + len, INT_MIN);
```

```
    for(i=0;i<5;++i)
        x[i] ^= INT_MIN;
}

static inline void radix_sort_unsigned(uint *a, const size_t len)
{
 rad_sort_u(a, a + len, (uint)INT_MIN);
}

int main(void)
{
    int vet[5], i;
    for(i=0;i<5;++i)
    {
       printf("Digite o valor: ");
       scanf("%d",&vet[i]);
    }

 radix_sort(vet, 5);

    for(i=0;i<5;++i)
    {
   printf("%d\n", vet[i]);
    }
    getch();

 return 0;
}
```

10 - Algoritmo Radixsort - radix_sort.c

Lista de Algoritmos

Capítulo 1

Algoritmo 1.1: Calcula o fatorial de N de forma direta 13
Algoritmo 1.2: Calcula o fatorial de N de forma recursiva 14
Algoritmo 1.3: Calcula o Binômio de Newton 16

Capítulo 2

Algoritmo 2.1: Bolha 21
Algoritmo 2.2: Bolha com Flag 23
Algoritmo 2.3: Inserção 24
Algoritmo 2.4: Seleção 25
Algoritmo 2.5: Shellsort 27
Algoritmo 2.6: Mergesort 28
Algoritmo 2.7: Procedimento Merge 30
Algoritmo 2.8: Procedimento Partição 32
Algoritmo 2.9: Quicksort 34
Algoritmo 2.10: Insere_HBC 36
Algoritmo 2.11: Remove_Menor 37
Algoritmo 2.12: Heapsort 38
Algoritmo 2.13: Countingsort 39
Algoritmo 2.14: Bucketsort 41
Algoritmo 2.15: Radixsort 42

Capítulo 3

Algoritmo 3.1: Merge2 – Intercalação de dois caminhos 51
Algoritmo 3.2: Merge3 – Intercalação de três caminhos 52

Capítulo 4

Algoritmo 4.1: Busca Sequencial Simples 64
Algoritmo 4.2: Busca Sequencial Recursiva 64
Algoritmo 4.3: Busca Binária 65