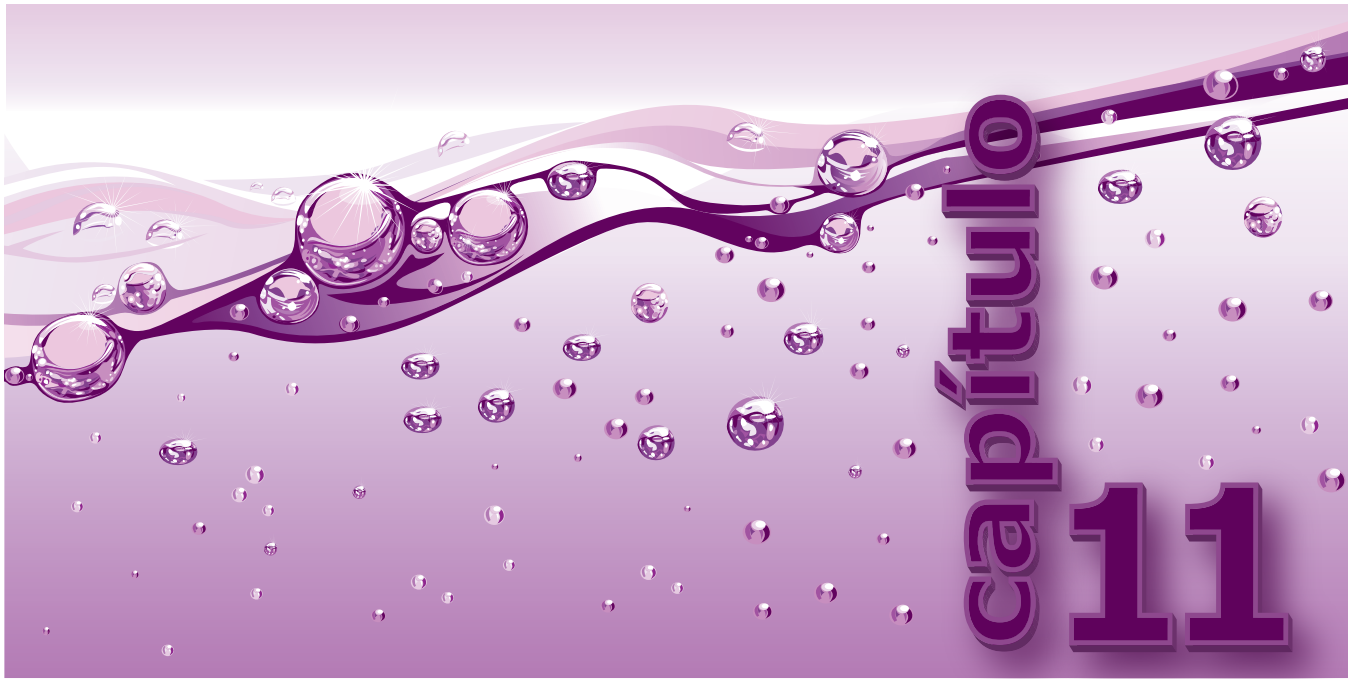

Ordenação em Memória Principal

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Chave de ordenação
  - ❒ Ordenação in loco
  - ❒ Estado de ordenação
  - ❒ Ordenação por troca
  - ❒ Pivô de QuickSort
  - ❒ Partição de QuickSort
  - ❒ SelectionSort
  - ❒ InsertionSort
  - ❒ BubbleSort
  - ❒ QuickSort
  - ❒ MergeSort
  - ❒ HeapSort
  - ❒ CountingSort
  - ❒ BucketSort
  - ❒ RadixSort
  - ❒ Ordenações estável e instável
  - ❒ Ordenações interna e externa
  - ❒ Ordenação por comparação
  - ❒ Estado de ordenação
  - ❒ Inversão
- Descobrir o melhor o pior casos de um algoritmo de ordenação e analisá-lo usando notação θ
- Classificar métodos de ordenação de acordo com seus custos temporal e espacial
- Descrever três métodos de ordenação por troca
- Implementar os algoritmos: SelectionSort, InsertionSort, BubbleSort, QuickSort, MergeSort, HeapSort, CountingSort e BucketSort
- Explicar o que é um algoritmo de divisão e conquista e por que o algoritmo de busca binária é erroneamente assim denominado
- Implementar um algoritmo de ordenação de lista simplesmente encadeada
- Escolher o algoritmo mais adequado para ordenação de uma dada tabela específica
- Expressar o limite inferior para algoritmos baseados em comparações usando notação Ω
- Definir e saber usar ordenação de ponteiros

objetivos

ESTE CAPÍTULO LIDA com **ordenação de dados em memória principal**, um dos mais antigos e bem estudados problemas em computação. Organizações governamentais, instituições financeiras e comerciais organizam suas informações ordenando-as. Manter dados ordenados torna possível efetuar buscas eficientemente.

Embora alguns algoritmos de ordenação apresentados neste capítulo sejam fáceis de entender e implementar, outros algoritmos levam um pouco mais de tempo para entender e um pouco mais de prática para implementar. Tipicamente, os algoritmos de ordenação mais fáceis de entender e implementar têm aplicações limitadas a pequenas quantidades de dados e vice-versa. Deve-se notar ainda que os algoritmos mais eficientes (i.e., aqueles com custo temporal linear) apresentam aplicabilidade limitada.

Os algoritmos de ordenação discutidos neste livro possuem denominações derivadas da língua inglesa. Todas elas terminam com *sort*, que significa ordenação em inglês. Essas denominações são tão comuns que se decidiu mantê-las. Este capítulo não tenta descrever todos os algoritmos de ordenação conhecidos. Em vez disso, serão apresentados aqueles que são mais populares, dos quais muitas variações existem. Deve ficar claro nesta discussão que nenhuma ordenação funciona otimamente em todas as situações.

11.1 Fundamentos de Ordenação

11.1.1 Conceitos Básicos

Ordenação (ou **classificação**) é um dos processos mais comuns de programação e consiste em ordenar uma coleção de dados segundo algum critério (**chave de ordenação**).

Define-se uma **tabela** de tamanho n como uma sequência de n **registros** (ou, simplesmente, **itens**) r_1, r_2, ..., r_n. Uma **chave** c_1 é associada com cada registro r_i. A chave é usualmente um campo do registro. Uma tabela ordenada é uma permutação da tabela original com as chaves ordenadas em alguma ordem. Ou seja, uma tabela é considerada **ordenada** por uma chave c se $i < j$ implica em $c_i < c_j$, para alguma ordem definida sobre as chaves (p. ex., a ordem natural dos números inteiros, se as chaves forem números inteiros). Considerando uma lista telefônica como exemplo, a tabela consiste da lista inteira e cada entrada na lista é um registro. Cada registro possui campos para o nome do assinante, seu endereço e seu número do telefone. A chave sobre a qual a tabela é ordenada é o nome do assinante.

Normalmente, as tabelas usadas em ordenação são indexadas; i.e., representadas por meio de arrays. Por isso, frequentemente, se usa *ordenação de array* ou *ordenação de lista* (indexada) em vez de *ordenação de tabela*.

Estabilidade

É possível que dois (ou mais) registros numa mesma tabela possuam uma mesma chave. Uma técnica de ordenação é denominada **estável** se, para todos registros r_i e r_j com chaves de ordenação c_i e c_j tais que $c_i = c_j$, se r_i precede r_j na tabela original, então r_i precede r_j na tabela ordenada. Ou seja, informalmente, numa ordenação estável, a ordem dos registros que possuem a mesma chave de ordenação é mantida.

Algumas vezes, é importante usar um método de ordenação estável quando a ordenação é feita sobre uma chave de registro que contém vários campos ou sobre parte de uma chave. Por exemplo, uma ordenação estável é indicada quando se deseja ordenar uma lista de pessoas pelo nome e depois pelo sobrenome.

A **Figura 11–1** e a **Figura 11–2** mostram exemplos de ordenações estável e instável.

A estabilidade de um algoritmo de ordenação só deve ser levada em consideração quando os registros cujas chaves estão sendo ordenadas são distinguíveis por um campo que não seja a chave de ordenação. Mas, mesmo em tal situação, nem sempre é necessário dar atenção a essa propriedade. Além disso, algoritmos de ordenação instáveis podem ser tornados estáveis. Um modo de fazer isso é estender a comparação de chaves de modo a

incluir a ordem em que os respectivos registros aparecem na tabela original como critério de desempate quando as chaves forem iguais. Evidentemente, isso tornaria o algoritmo menos eficiente do que ele seria sem essa comparação adicional.

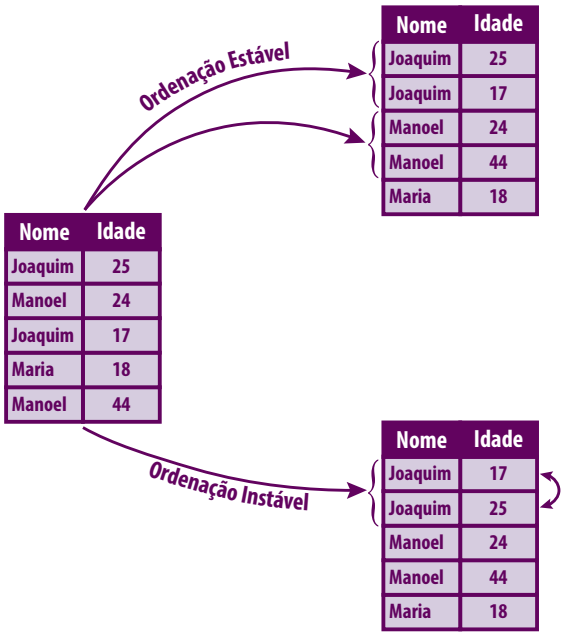

Figura 11–1: Ordenação Estável e Ordenação Instável de Registros

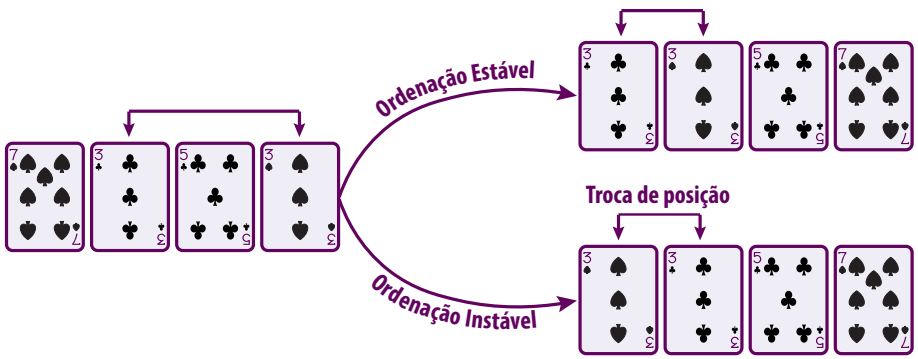

Figura 11–2: Ordenação Estável e Ordenação Instável de Cartas de Baralho

Alguns algoritmos, como, por exemplo, InsertionSort (v. **Seção 11.2.3**), são **inerentemente estáveis**, enquanto outros (p. ex., HeapSort — v. **Seção 11.3.3**) são **inerentemente instáveis**. A estabilidade da maioria dos algoritmos discutidos neste capítulo depende de implementação. Aqui, algoritmos que são implementados *naturalmente* de modo estável, são classificados como estáveis.

Uso de Espaço Adicional

Alguns poucos algoritmos de ordenação utilizam espaço adicional proporcional ao número de itens a ordenar. Algoritmos que não usam espaço adicional são classificados como **in loco**.

Ordenações Interna e Externa

Uma ordenação pode ser classificada como **interna**, se os registros que estão sendo ordenados estão na memória principal, ou **externa**, se alguns dos registros que estão sendo ordenados estiverem em memória auxiliar.

Algoritmos de Propósito Geral e Específico

Alguns algoritmos podem ser usados para ordenação de tabelas cujas chaves de ordenação podem ser de qualquer natureza. Tais algoritmos são considerados de **propósito geral**. Por outro lado, alguns algoritmos impõem certas restrições sobre as chaves de ordenação. Esses algoritmos são considerados de **propósito específico**. Os algoritmos CountingSort, BucketSort e RadixSort (v. **Seção 11.4**) têm propósitos específicos, enquanto todos os demais algoritmos discutidos neste capítulo são de propósito geral. Todos os algoritmos de propósito geral são baseados em comparações (v. adiante).

Ordenações Baseadas em Comparações e Distribuições

Um **algoritmo de ordenação baseado em comparação** recebe como entrada uma tabela $r_1, r_2, ..., r_n$ com n itens e obtém informação sobre os itens comparando pares deles. Cada comparação verifica se um registro é considerado maior, menor ou igual a outro. O algoritmo pode reordenar itens com base no resultado de tal comparação.

Em resumo, esses são algoritmos de ordenação que apenas comparam pares de elementos e movem elementos com base no resultado dessas comparações. Por exemplo, QuickSort e MergeSort (v. **Seção 11.3**) são algoritmos de ordenação baseados em comparação.

As operações básicas efetuadas por um algoritmo de ordenação baseado em comparações são: comparação de chaves e troca ou cópia de registros. Em alguns casos, cópia é mais onerosa do que comparação; em outros casos, comparação é mais onerosa do que cópia. Por exemplo, se as chaves de ordenação forem números inteiros e os registros são bem grandes, copiar registros tem custo mais elevado do que comparar chaves. Além disso, troca é a mais onerosa dessas operações, pois, usualmente, envolve três operações de cópia e uma variável auxiliar.

Algoritmos de ordenação baseados em distribuições (p. ex., BucketSort — v. **Seção 11.4.2**) contam com o conhecimento prévio sobre o conjunto de chaves possíveis e não são considerados algoritmos de propósito geral.

Ordenação Adaptativa

Algoritmos que levam em consideração ou se beneficiam do fato de a tabela a ser ordenada já estar previamente ordenada são chamados **adaptativos**. O algoritmo InsertionSort (v. **Seção 11.2.3**) é um exemplo de algoritmo adaptativo.

Ordenações Online e Offline

Um algoritmo é **online** quando ele pode ordenar chaves à medida que as recebe num fluxo contínuo; caso contrário, ele é classificado como **offline**. Um exemplo de algoritmo online é InsertionSort (v. **Seção 11.2.3**).

Métodos de Ordenação

Não há nenhum método de ordenação que seja universalmente superior a todos os demais, de maneira que o programador deve examinar cuidadosamente o problema em mãos antes de decidir sobre qual método escolher. Existe um grande número de métodos de ordenação que podem ser utilizados para ordenar uma tabela. O programador deve estar ciente das várias considerações de eficiência para escolher adequadamente o método de ordenação que é mais conveniente para determinado problema particular. Considerações importantes são:

- ❒ Tempo a ser despendido para codificar o algoritmo de ordenação
- ❒ Tempo de execução do algoritmo
- ❒ Quantidade de memória adicional gasta pelo algoritmo
- ❒ Informações específicas sobre a tabela a ser ordenada (p. ex., se ela é grande ou pequena, se ela está quase ordenada ou não, tipo de chave de ordenação)

Se a tabela for pequena, técnicas sofisticadas de ordenação elaboradas para minimizar o uso de tempo e espaço são às vezes piores ou apenas ligeiramente melhores do que métodos mais simples e geralmente menos eficientes. Da mesma forma, se um algoritmo de ordenação deve ser executado apenas uma vez, não faz sentido que um programador leve dias tentando encontrar o melhor método para obter a eficiência máxima.

É importante que o programador tenha conhecimento de várias técnicas de ordenação e reconheça as vantagens e desvantagens de cada uma, de forma que, quando for necessária a utilização de ordenação ele seja capaz de escolher o método mais conveniente para uma situação particular.

Neste capítulo, serão analisados vários métodos de ordenação agrupados de acordo com seus custos temporais.

Estado de Ordenação e Inversões

Um algoritmo de ordenação pode receber uma tabela de entrada com seus registros arranjados de três maneiras, denominadas **estados de ordenação**, que são:

- **Tabela ordenada**. Nesse caso os registros já se encontram na ordem desejada. Esse tipo de arranjo de registros corresponde ao melhor caso para a maioria dos algoritmos de ordenação.
- **Tabela inversamente ordenada**. Nesse caso os registros se encontram em ordem inversa da ordem desejada e corresponde ao pior caso para a maioria dos algoritmos de ordenação.
- **Tabela aleatória** (ou **desordenada**). Nesse caso, os registros não obedecem a nenhuma ordem. O fato de os registros estarem ordenados aleatoriamente corresponde ao caso médio de qualquer algoritmo de ordenação.

Uma **inversão** é um par de registros que se encontram em ordem trocada.

11.1.2 Aplicações de Ordenação

O conceito de ordenação de itens tem uma importância considerável no cotidiano. Considere, por exemplo, o processo de encontrar um número de telefone numa lista telefônica. Esse processo é consideravelmente simplificado se os nomes na lista telefônica estiverem em ordem alfabética. Imagine a dificuldade que se teria se os números dos telefones fossem colocados na lista segundo a ordem com que foram adquiridos na companhia telefônica. Os livros numa biblioteca também mantêm uma dada ordem (sistema de catalogação), de forma que cada livro é mantido numa posição específica relativa aos outros, o que permite encontrá-los facilmente. Em geral, um conjunto de itens é mantido ordenado para produzir um relatório (para simplificar a recuperação manual de informação, como no caso da lista telefônica) ou para tornar mais eficiente o acesso aos dados. Mas existem outras situações nas quais a ordenação de dados simplifica o processamento de informação, como, por exemplo:

- **Teste de unicidade**, que verifica se todos os elementos de uma coleção são distintos.
- **Remoção de duplicatas**, que remove itens duplicados de uma coleção de dados. Existem chaves duplicadas numa coleção de itens comparáveis? Quantas chaves distintas existem numa tabela? Qual valor aparece mais frequentemente? Com ordenação, pode-se responder essas questões com custo temporal linear logarítmico: primeiro ordena-se a tabela, então faz-se uma passagem pela tabela ordenada, anotando-se os valores duplicados que aparecem consecutivamente na tabela ordenada.
- **Operações sobre tabelas de busca**, que conforme foi visto em capítulos anteriores (v. **Capítulo 3**, por exemplo), são bastante facilitadas quando os dados são ordenados.
- **Problemas de seleção**, nos quais se tenta encontrar o i-ésimo maior (ou menor) item de uma coleção.
- **Contagem de frequência** (**moda**) que encontra o elemento que ocorre com mais frequência numa coleção.

- Algumas **operações sobre conjuntos**, tais como encontrar a união ou a interseção de dois conjuntos.

11.2 Ordenação com Custo Temporal Quadrático

Os algoritmos de ordenação apresentados nesta seção são os mais fáceis de entender e todo programador deve saber implementar sem hesitar pelo menos um desses métodos de ordenação em sua linguagem favorita. Contudo esses são os algoritmos de ordenação que apresentam o pior custo temporal, de maneira que eles só são convenientes para tabelas relativamente pequenas.

Por simplicidade, em todas as implementações de algoritmos de ordenação apresentadas neste capítulo, a tabela a ser ordenada é um array de elementos do tipo **int**. Além disso, nessas implementações, são usados dois parâmetros:

- `tabela` (entrada/saída) — tabela que será ordenada
- `nElem` (entrada) — número de elementos na tabela

Algumas funções de ordenação chamam a função `TrocaGenerica()`, que troca os valores de duas variáveis e foi definida na **Seção 10.2.4**.

11.2.1 Ordenação por Borbulhamento (BubbleSort)

Descrição

O algoritmo **BubbleSort** é provavelmente o mais conhecido e elementar algoritmo de ordenação. Esse algoritmo é apresentado na **Figura 11–3**.

Algoritmo BubbleSort

Entrada/Saída: Uma tabela indexada com *n* elementos

1. Use uma variável (*emOrdem*) para indicar quando a tabela estiver ordenada e inicie-a com um valor que indique que a tabela não está ordenada
2. Enquanto a tabela não estiver ordenada, faça o seguinte:
   - **2.1** Atribua à variável *emOrdem* um valor que indique que a tabela está ordenada
   - **2.2** Para cada par de elementos adjacentes da tabela, faça o seguinte:
     - **2.2.1** Se os elementos estiverem fora de ordem, troque-os de posição
     - **2.2.2** Atribua à variável *emOrdem* um valor que indique que a tabela não está ordenada
   - **2.3** Decremente o número de elementos da tabela que precisam ser ordenados

Figura 11–3: Algoritmo BubbleSort

Cada execução do laço interno (**Passo 2.2**) do algoritmo da **Figura 11–3** é denominada **passagem** e, em resumo, consiste em comparar cada par de elementos da tabela que podem estar fora de ordem.

Depois da primeira passagem, o maior elemento da tabela terá sido movido para sua última posição. Desse modo, não faz sentido levá-lo novamente em consideração na próxima passagem, posto que ele já se encontra em sua posição definitiva. Aplica-se o mesmo raciocínio aos demais elementos da tabela. Quer dizer, na segunda passagem, o segundo maior elemento é colocado em sua devida posição, na terceira passagem, o terceiro maior elemento é colocado em seu devido lugar e assim por diante. Logo, após cada passagem, o número de elementos que serão comparados é um a menos do que na última passagem. Se você ainda não entendeu: na segunda passagem, o último elemento do array não precisa ser levando em consideração; na terceira passagem os dois últimos elementos não precisam mais ser considerados; e assim por diante. Esses elementos deixam de ser levados em conta porque eles já se encontram em suas posições definitivas. Por isso, o número de elementos que precisam ser ordenados é decrementado após cada passagem (**Passo 2.3** do algoritmo).

A denominação do algoritmo **BubbleSort** é derivada do fato de os elementos maiores irem *subindo* aos poucos para o final da tabela, como se fossem *bolhas*. A **Figura 11–4** ilustra a ordenação de um array de inteiros usando o método da bolha.

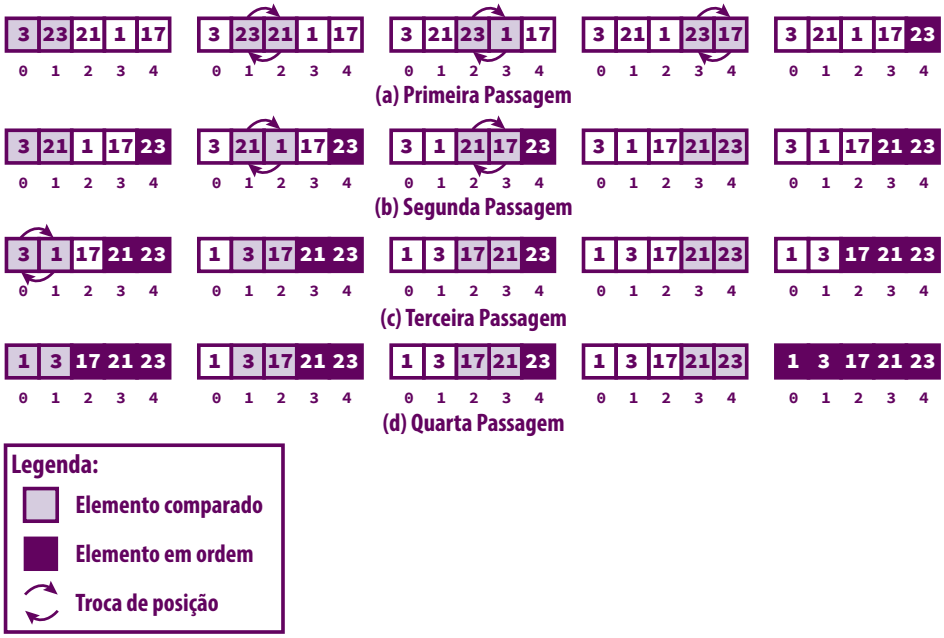

Figura 11–4: Ordenação pelo Método BubbleSort

Implementação

A função `BubbleSort()` apresentada a seguir ordena uma tabela usando **BubbleSort**.

```
void BubbleSort(int tabela[], int nElem)
{
   int i, emOrdem = 0;

   while (!emOrdem){
      emOrdem = 1; /* Supõe que a tabela está ordenada */

      for (i = 0; i < nElem - 1; i++)
            /* Compara elementos adjacentes */
         if (tabela[i] > tabela[i + 1]){
            emOrdem = 0; /* Pelo menos um par de elementos está fora de ordem */

               /* Troca elementos que estão fora de ordem */
            TrocaGenerica(tabela + i, tabela + i + 1, sizeof(tabela[0]));
         }

      --nElem; /* Mais um elemento já está em seu devido lugar */
   }
}
```

Análise

Teorema 11.1: No pior caso, o custo temporal do algoritmo **BubbleSort** é $\theta(n^2)$.

Prova: No pior caso, o número de comparações efetuadas pelo algoritmo **BubbleSort** é $[(n - 1)n]/2$ (v. **Capítulo 6** do **Volume 1**). Logo o custo temporal de pior caso desse algoritmo é $\theta(n^2)$. ■

Teorema 11.2: No melhor caso, o custo temporal do algoritmo **BubbleSort** é $\theta(n)$.

Prova: Em qualquer caso, o custo temporal do corpo do laço interno é $\theta(n)$. No melhor caso, ao final da primeira execução do laço interno, a variável que indica se a tabela está ordenada não será alterada, informando que ela já está ordenada. Portanto, isoladamente, o custo temporal do laço externo nesse caso é $\theta(1)$. Usando a regra do produto (v. **Capítulo 6** do **Volume 1**), o custo temporal de melhor caso desse algoritmo é $\theta(n)$. ■

Teorema 11.3: No caso médio, o custo temporal do algoritmo **BubbleSort** é $\theta(n^2)$.

Prova: Se a tabela estiver ordenada aleatoriamente e o tamanho da tabela for suficientemente grande, cerca de metade das comparações de chaves resultarão em troca, de maneira que o laço externo só terá certeza de que a tabela está ordenada na última passagem desse laço. Logo o custo temporal desse último laço é $\theta(n)$ e, usando a regra do produto, o custo temporal no caso médio é $\theta(n^2)$. ■

O algoritmo **BubbleSort** é estável, pois dois registros só trocam de posição quando um registro com chave maior precede um registro com chave menor. Ou seja, quando dois registros possuem a mesma chave, eles não trocam de lugar.

Um resumo da avaliação desse algoritmo é apresentado na **Tabela 11–1**.

Custo temporal	❒ Melhor caso: $\theta(n)$ ❒ Caso médio: $\theta(n^2)$ ❒ Pior caso: $\theta(n^2)$
Vantagens	❒ Facilidade de implementação ❒ In loco ❒ Estável
Desvantagem	❒ Lentidão
Indicações	❒ Tabelas muito pequenas ❒ Tabelas quase ordenadas ❒ Demonstrações didáticas

Tabela 11–1: Análise Resumida de BubbleSort

11.2.2 Ordenação por Seleção Direta (SelectionSort)

Descrição

O algoritmo de **ordenação por seleção direta** (**SelectionSort**) é apresentado na **Figura 11–5**.

Algoritmo SelectionSort

Entrada/Saída: Uma tabela indexada com n elementos

1. Para cada elemento da tabela, faça o seguinte:
  1.1 Encontre o menor elemento
  1.2 Troque o elemento corrente de posição com o menor elemento

Figura 11–5: Algoritmo SelectionSort

O algoritmo **SelectionSort** usa um índice i para marcar o início da parte desordenada do array. Inicialmente o índice do primeiro elemento é atribuído a i, de maneira que a parte desordenada do array fica entre os índices i e $n - 1$, em que n é o número de elementos da tabela. O processamento principal ocorre num laço no qual, em cada iteração, o elemento com a menor chave na parte desordenada do array é trocado com o elemento no índice i. Depois da troca, i está na parte ordenada do array, de modo que se reduz o tamanho da parte desordenada incrementando i. Na expressão condicional do laço interno, a parte desordenada do array varia de i até

$n - 1$. Sabe-se que cada elemento na parte desordenada tem chave maior do que ou igual à chave de qualquer elemento na parte ordenada do array. Quando $i = n - 1$, a parte desordenada do array contém apenas um elemento e a chave desse elemento deve ser maior do que ou igual à chave de qualquer elemento na parte ordenada. Assim o elemento no índice $n - 1$ está em seu correto lugar e a ordenação está completa.

A **Figura 11–6** ilustra o uso de SELECTIONSORT na ordenação de um array de valores inteiros.

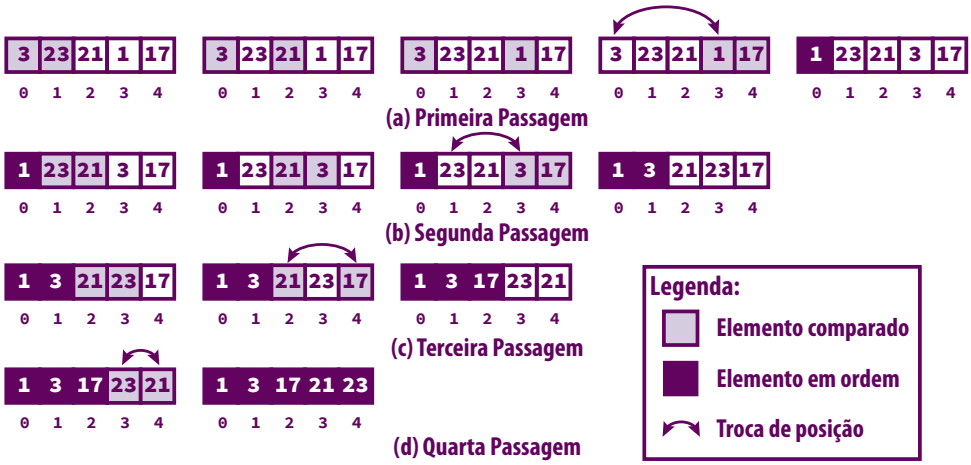

FIGURA 11–6: ORDENAÇÃO POR SELEÇÃO DIRETA (SELECTIONSORT)

Um algoritmo equivalente ao descrito na **Figura 11–5** pode efetuar a ordenação do final para o início do array colocando os maiores elementos em suas posições ordenadas a cada iteração.

Implementação

A função `SelectionSort()` ordena uma tabela usando seleção direta.

```
void SelectionSort(int tabela[], int nElem)
{
   int i, j, iMenor;

      /* A porção ordenada do array está entre os índices 0 e i - 1, */
      /* enquanto a porção desordenada está entre i e n - 1          */

      /* Obtém o menor elemento na parte desordenada e troca-o de posição */
      /* com o primeiro elemento da parte desordenada (que é i)           */
   for (i = 0; i < nElem - 1; ++i) {
         /* Supõe que o primeiro elemento da parte desordenada é o menor */
      iMenor = i;

         /* Verifica se existe um elemento menor do 'iMenor' na parte desordenada */
      for (j = i + 1; j < nElem; ++j)
            /* Se o elemento corrente for menor do que */
            /* 'iMenor', ele passará a ser o menor     */
        if (tabela[j] < tabela[iMenor])
            iMenor = j; /* O novo mínimo passa a ser j */

         /* Se foi encontrado um elemento menor do que aquele que se encontra   */
         /* na posição i, trocam-se as posições desses elementos. Caso          */
         /* contrário, o elemento na posição i já está em sua posição ordenada. */
      if (iMenor != i)
        TrocaGenerica(tabela + i, tabela + iMenor, sizeof(tabela[0]));
   }
}
```

Análise

Lema 11.1: No algoritmo SELECTIONSORT, o número de comparações de chaves efetuadas é igual a $(n^2 - n)/2$.

Prova: Na primeira execução do laço **for** interno, a instrução que efetua comparações de chaves é executada $n - 1$ vezes, na segunda execução desse laço, essa instrução é executada $n - 2$ vezes, e assim por diante, até que, na última execução desse laço, a instrução de comparação é executada apenas uma vez. Assim o número total de comparações é dado por:

$$1 + 2 + \ldots + (n - 2) + (n - 1) = (n^2 - n)/2$$ ■

Teorema 11.4: Em qualquer caso, o custo temporal do algoritmo SELECTIONSORT é $\theta(n^2)$.

Prova: De acordo com o **Lema 11.1**, o número de comparações de chaves efetuadas pelo algoritmo SELECTIONSORT é sempre $(n^2 - n)/2$, independentemente de caso. Logo, em qualquer caso, o custo temporal desse algoritmo é $\theta(n^2)$. ■

Teorema 11.5: No algoritmo SELECTIONSORT, o número máximo de trocas de posições entre elementos é $n - 1$.

Prova: O número máximo de trocas acontece quando, em cada passagem do laço externo, ocorre uma troca e o número de passagens do laço externo é $n - 1$. ■

No algoritmo SELECTIONSORT, o número de comparações de chaves é igual a $(n^2 - n)/2$ e o número máximo de trocas é $n - 1$, que é o melhor que se pode esperar de um algoritmo de ordenação que conta apenas com trocas para colocar seus elementos na posição correta. Assim esse algoritmo é bom para chaves pequenas e registros grandes.

O algoritmo SELECTIONSORT não é estável, pois, após encontrar um registro contendo a menor chave na parte desordenada da tabela, ele troca esse registro de posição com outro, de modo que esse outro registro pode ser posicionado adiante de um registro que tem a mesma chave. A **Figura 11–7** ilustra essa situação. Nessa figura, a lista que está sendo ordenada possui dois elementos com a mesma chave *3*, sendo que o elemento com fundo escurecido antecede o outro com fundo branco que possui a mesma chave. Após a troca dos elementos com chaves *1* e *3*, o elemento com fundo escurecido passou a suceder o elemento com fundo branco que possui a mesma chave. É possível tornar o algoritmo SELECTIONSORT estável evitando esse tipo de troca, mas acrescentar essa alteração no algoritmo SELECTIONSORT básico teria um custo adicional elevado.

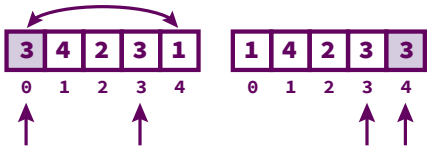

FIGURA 11–7: INSTABILIDADE DE SELECTIONSORT

A **Tabela 11–2** apresenta um resumo da avaliação desse algoritmo.

CUSTO TEMPORAL	❒ $\theta(n^2)$ nos três casos
VANTAGENS	❒ Facilidade de implementação ❒ In loco
DESVANTAGENS	❒ Lentidão ❒ Instável
INDICAÇÕES	❒ Tabelas muito pequenas ❒ Chaves pequenas e registro grandes

TABELA 11–2: ANÁLISE RESUMIDA DE SELECTIONSORT

11.2.3 Ordenação por Inserção (InsertionSort)

Descrição

O algoritmo de **ordenação por inserção** (InsertionSort) é apresentado na **Figura 11–8**.

Algoritmo InsertionSort

Entrada/Saída: Uma tabela indexada com *n* elementos

1. A partir do segundo elemento da tabela, para cada um deles, faça o seguinte:
   - **1.1** Armazene o valor do elemento corrente numa variável auxiliar
   - **1.2** Mova cada elemento maior do que a variável auxiliar uma posição adiante a partir da posição do elemento anterior ao elemento corrente
   - **1.3** Armazene o valor da variável auxiliar na posição do elemento anterior ao elemento corrente

Figura 11–8: Algoritmo InsertionSort

A **Figura 11–9** ilustra a ordenação de um array de inteiros usando o método de inserção[1].

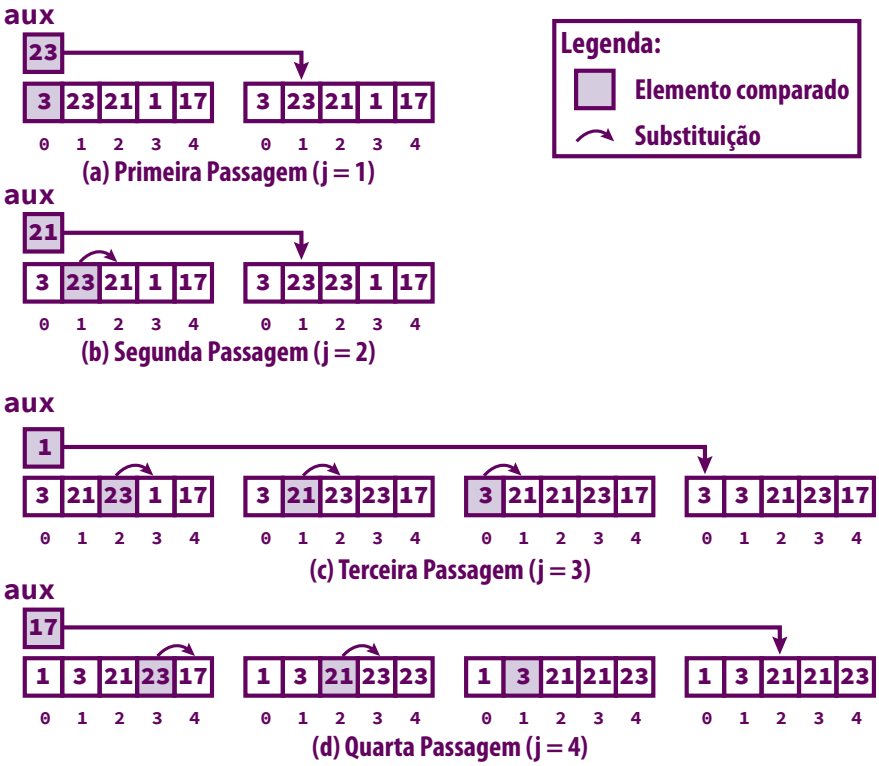

Figura 11–9: Ordenação pelo Método de Inserção (InsertionSort)

Implementação

A função `InsertionSort()` ordena uma tabela usando ordenação por inserção.

```
void InsertionSort(int tabela[], int  nElem)
{
   int i, j, aux;
```

[1] Talvez, estudar a implementação de algoritmo antes de examinar essa figura facilite o entendimento de ambas.

```
      /* Inicialmente, o array tabela[] é considerado uma tabela ordenada.   */
      /* Então, a cada iteração, os elementos entre 0 e j estarão ordenados. */

      /* Insere tabela[j] na parte ordenada da tabela */
   for (j = 1; j < nElem; ++j) {
      aux = tabela[j];

         /* Move cada elemento maior do que aux uma posição adiante */
      for (i = j-1; i >= 0 && aux < tabela[i]; --i)
         tabela[i + 1] = tabela[i];

      tabela[i + 1] = aux; /* Insere aux na posição correta */
   }
}
```

Análise

Lema 11.2: O número máximo de comparações de chaves efetuadas pelo algoritmo **InsertionSort** é $(n^2 - n)/2$.

Prova: A primeira passagem do algoritmo **InsertionSort** compara, no máximo, uma chave; a segunda passagem compara, no máximo, duas chaves; e assim por diante, de modo que, na última passagem, no máximo, são comparadas $n - 1$ chaves. Portanto tem-se que o número máximo de comparações de chaves é dado por: $1 + 2 + 3 + \ldots + n - 1$, que pode ser escrito como:

$$\sum_{i=1}^{n-1} i = \frac{(n^2 - n)}{2}$$

■

Teorema 11.6: No pior caso, o custo temporal do algoritmo **InsertionSort** é $\theta(n^2)$.

Prova: De acordo com o **Lema 11.2**, o número máximo de comparações de chaves é $\theta(n^2)$ e esse é o custo temporal de pior caso do algoritmo **InsertionSort**. ■

Teorema 11.7: No melhor caso, o custo temporal do algoritmo **InsertionSort** é $\theta(n)$.

Prova: No melhor caso, o corpo do laço interno (**Passo 1.2** do algoritmo **InsertionSort**) não é executado e o laço externo é executado $n - 1$ vezes. Logo o custo temporal é $\theta(n)$. ■

Lema 11.3: No caso médio, o algoritmo **InsertionSort** efetua $\theta(n^2)$ comparações de chaves e $\theta(n^2)$ atribuições.

Prova: Assuma que a tabela a ser ordenada está ordenada aleatoriamente, o que significa que qualquer ordenação das chaves é igualmente provável. Suponha que um elemento da tabela encontra-se na posição i. Existem $i - 1$ movimentos que podem ser efetuadas sobre esse elemento, desde uma posição até $i - 1$ posições para o início da lista. Além disso ele pode ficar onde está (pois já se encontra na posição correta). Portanto, no total, há i ações possíveis, que, por hipótese, são todas equiprováveis. Assim a probabilidade de o elemento não ser movido é $1/i$ e, nesse caso, ocorre apenas uma comparação de chaves. As demais movimentações têm probabilidade $(i - 1)/i$.

Como todas as $i - 1$ movimentações indicadas acima são igualmente prováveis, o número médio de iterações do laço interno do algoritmo **InsertionSort** é dado por:

$$\frac{1+2+\ldots+i-1}{i-1} = \frac{(i-1)\cdot i}{2} \times \frac{1}{i-1} = \frac{i}{2}$$

Em cada uma dessas iterações, ocorre uma comparação e uma atribuição. No laço externo, ocorrem mais duas atribuições, de modo que, nesse caso, o elemento de índice i requer, em média, $i/2$ comparações de chaves e $i/2 + 2$ atribuições. Quando esses dois casos são combinados, obtém-se que o número médio de comparações de chaves é dado por:

$$\frac{i}{2}\times 1+\frac{i-1}{i}\times\frac{i}{2}=\frac{2i-1}{2} \qquad [1]$$

Por sua vez, o número médio de atribuições é dado por:

$$\frac{i}{2}\times 0+\frac{i-1}{i}\times\left(\frac{i}{2}+2\right)=\frac{i+3}{2}-\frac{2}{i} \qquad [2]$$

O número total de comparações de chaves é determinado pelo somatório da expressão **[1]** e o número total de atribuições é obtido pelo somatório da expressão **[2]**, sendo que, em ambos os casos, os limites do somatório são $i = 2$ e $i = n - 1$. Não é necessário obter o resultado preciso desse somatório para concluir que o número total de comparações de chaves é $\theta(n^2)$ e o número total de atribuições também é $\theta(n^2)$. ■

Teorema 11.8: No caso médio, o custo temporal do algoritmo **InsertionSort** é $\theta(n^2)$.

Prova: A prova é consequência imediata do **Lema 11.3**. ■

Assim como **BubbleSort**, o melhor caso de **InsertionSort** também ocorre quando as chaves já estão ordenadas e o pior caso ocorre quando as chaves estão em ordem inversa.

Como **InsertionSort** requer atribuições entre chaves, em vez de trocas de chaves (que requerem três atribuições), e o número de atribuições é aproximadamente igual ao número de comparações de chaves (v. prova do **Lema 11.3**), esse algoritmo é aproximadamente duas vezes mais rápido do que **BubbleSort** para chaves distribuídas aleatoriamente, embora, nesse caso, ambos tenham custo temporal $\theta(n^2)$.

InsertionSort é o mais versátil dos algoritmos básicos de ordenação e é uma boa escolha quando a tabela a ser ordenada estiver quase ordenada ou for pequena. Ele é o algoritmo mais usado dentre aqueles com custo temporal quadrático e é frequentemente usado como algoritmo auxiliar de **QuickSort** e **MergeSort** (v. **Seção 11.3**). **InsertionSort** pode ainda ser facilmente adaptado para ordenação de listas encadeadas.

O algoritmo **InsertionSort** é estável, pois, quando um registro é inserido na parte ordenada da tabela, apenas registros maiores do que ele mudam de posição. Assim, um registro que antecedesse o registro inserido antes da inserção, continuaria a antecedê-lo depois dessa inserção.

Um resumo da avaliação do algoritmo **InsertionSort** é vista na **Tabela 11–3**.

Custo temporal	❒ Melhor caso: $\theta(n)$ ❒ Caso médio: $\theta(n^2)$ ❒ Pior caso: $\theta(n^2)$
Vantagens	❒ Simplicidade do algoritmo ❒ In loco ❒ Estável ❒ Melhor algoritmo com custo $\theta(n^2)$
Desvantagens	❒ Lentidão
Indicações	❒ Tabelas pequenas ❒ Tabelas quase ordenadas

Tabela 11–3: Análise Resumida de InsertionSort

11.3 Ordenação com Custo Temporal Linear Logarítmico

11.3.1 QuickSort

Descrição

O método de ordenação **QuickSort** consiste em dividir a tabela a ser ordenada em duas **partições**, de modo que os elementos da primeira partição tenham chaves menores do que as chaves dos elementos da segunda partição. A divisão em partições é feita comparando-se a chave de cada elemento com a chave de um elemento denominado **pivô**. Então, as duas partições da tabela são ordenadas recursivamente invocando o próprio algoritmo **QuickSort**. Esse algoritmo foi inventado por Charles Hoare em 1962 (v. **Bibliografia**) e é apresentado na **Figura 11–10**.

Algoritmo QuickSort

Entrada/Saída: Uma tabela indexada com *n* elementos

1. Se houver apenas um elemento na tabela, encerre
2. Eleja um elemento da tabela para servir como pivô
3. Divida a tabela em duas metades, sendo que na primeira metade ficam os elementos menores do que o pivô e na segunda metade ficam os elementos maiores do que o pivô
4. Ordene a primeira metade da tabela usando **QuickSort**
5. Ordene a segunda metade da tabela usando **QuickSort**

Figura 11–10: Algoritmo QuickSort

Supondo que a tabela a ser ordenada possui pelo menos um elemento, o primeiro passo do algoritmo **QuickSort** é a escolha do pivô, que é sempre o elemento que termina em sua posição final ordenada depois de cada divisão da tabela, sendo que elementos com chaves menores do que a dele ficam na partição esquerda e elementos com chaves maiores do que a dele ficam na partição direita. Há muitas maneiras usadas para escolha do pivô e a mais simples delas é elegê-lo como o primeiro elemento da tabela[2]. Entretanto essa escolha pode criar problemas de desempenho para o algoritmo **QuickSort** (v. adiante).

O problema central do método de ordenação **QuickSort** é como fazer a partição da tabela (**Passo 3** na **Figura 11–10**). O algoritmo de partição usado por **QuickSort** é apresentado na **Figura 11–11**.

Algoritmo PartiçãoDeQuickSort

Entrada/Saída: Uma tabela indexada com *n* elementos

1. Atribua o primeiro índice da tabela à variável *Esq*
2. Atribua o último índice da tabela à variável *Dir*
3. Enquanto *Esq* < *Dir*, faça:
   - **3.1** Enquanto a chave do elemento apontado por *Esq* não for maior do que o pivô, incremente *Esq*
   - **3.2** Enquanto a chave do elemento apontado por *Dir* não for menor do que o pivô, decremente *Dir*
   - **3.3** Se os elementos nos índices *Esq* e *Dir* estiverem em partições erradas, troque-os de posição

Figura 11–11: Algoritmo de Partição Usado por QuickSort

O algoritmo **PartiçãoDeQuickSort** usa dois índices, sendo um em cada metade da tabela. Inicialmente, esses índices são associados, respectivamente, ao primeiro e ao último elementos da tabela [v. **Figura 11–12 (a)**]. Na **Figura 11–12**, esses índices são representados por *Esq* e *Dir*. O algoritmo de partição avança os índices esquerdo e direito em direção um do outro até que eles se encontrem. O índice esquerdo é incrementado até

[2] Escolher o último elemento da tabela como pivô é uma abordagem equivalente que não será usada aqui.

que se encontre um elemento cuja chave é maior do que aquela do pivô [v. **Figura 11–12 (c)**]. Por outro lado, o índice direito é decrementado até que seja encontrado um elemento cuja chave é menor do que aquela do pivô [v. **Figura 11–12 (e)**]. Então o elemento que se encontra no índice direito troca de posição com o elemento que se encontra no índice esquerdo [v. **Figura 11–12 (f)**]. Como mostra a **Figura 11–12 (l)**, quando esses índices se cruzam, o índice direito indica exatamente a posição definitiva do pivô. Nesse ponto, o pivô troca de posição com o elemento no índice direito [v. **Figura 11–12 (m)**] e a presente divisão de tabela está completa com chaves menores à esquerda do pivô e chaves maiores à direita do pivô, como se vê na **Figura 11–12 (n)**.

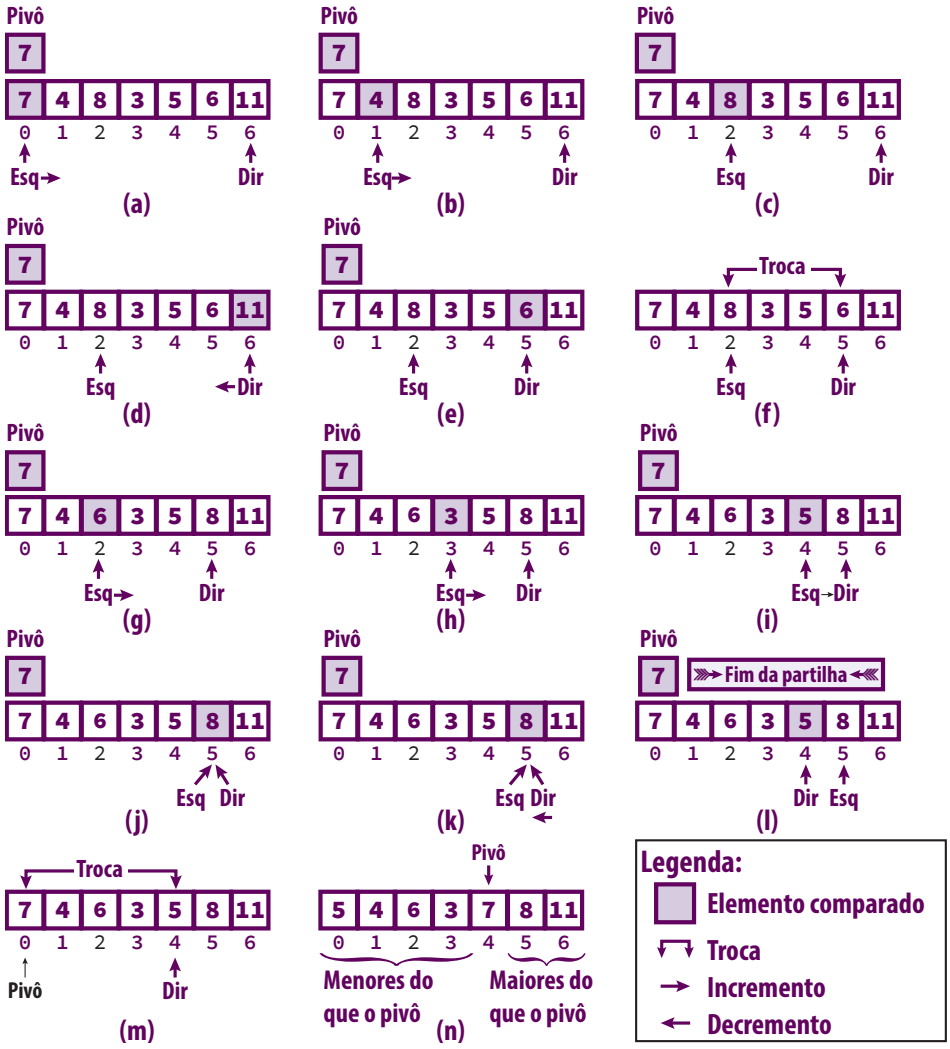

Figura 11–12: Partilhando uma Tabela com QuickSort

Após a partição, a tabela terá sido dividida em duas partes que não estão necessariamente ordenadas. Essas duas partições serão ordenadas em seguida nos próximos dois passos do algoritmo. Esses dois últimos passos invocam o próprio algoritmo **QuickSort** para ordenar cada uma dessas partições.

O pior caso de **QuickSort** ocorre quando uma tabela com n elementos é dividida numa partição contendo um único elemento (que é o pivô) e noutra com os $n - 1$ elementos restantes. Se tal divisão ocorre com cada par de subtabelas, cada elemento da tabela requer uma partição. É isso que ocorre quando a tabela já está ordenada (ou inversamente ordenada) e o pivô é seu primeiro ou último elemento, pois, em cada divisão de tabelas, o

pivô terá sempre a menor (ou a maior) chave, de modo que cada partição resultará em $n - 1$ elementos numa partição e apenas o pivô na outra partição[3].

Uma possível solução para esse problema é usar como pivô o elemento com a chave mediana entre o primeiro elemento, o último elemento e o elemento do meio da tabela. Essa abordagem é chamada **mediana de três** e é ilustrada na **Figura 11–13**[4].

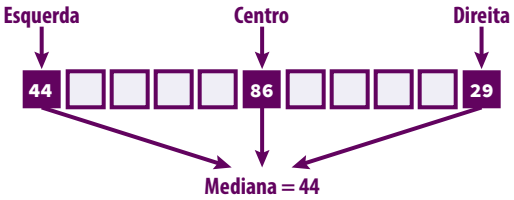

FIGURA 11–13: USO DE MEDIANA DE TRÊS EM QUICKSORT

A chave do elemento mediano de uma tabela é maior do que as chaves de uma metade dos elementos e menor do que as chaves da outra metade desses elementos. A escolha dessa chave mediana ideal resultaria numa tabela partilhada em duas subtabelas de mesmo tamanho, que seria a situação ótima para o algoritmo **QUICKSORT**. Contudo essa escolha ideal iria requerer a ordenação de toda a tabela, que é exatamente o objetivo final.

Encontrar a mediana de três elementos de uma tabela é obviamente muito mais fácil e rápido do que encontrar a mediana de todos os elementos da tabela. Mesmo assim, essa simples abordagem evita que se use como pivô a maior ou a menor chave em casos nos quais os dados já estão ordenados ou inversamente ordenados. Infelizmente, é provável que ainda existam alguns estados de ordenação patológicos nos quais a abordagem de mediana de três não funciona bem (p. ex., quando todas as chaves são iguais), mas, normalmente, essa é uma técnica rápida e efetiva de escolha de pivô.

Pode-se usar a abordagem mediana de três não apenas para selecionar o pivô, mas também para ordenar os três elementos usados nesse processo de seleção, como mostra a **Figura 11–14**. A ordenação desses três elementos influencia a chave que será usada como pivô na próxima divisão de tabela e ajuda a limitar o efeito de ordenação prévia da tabela.

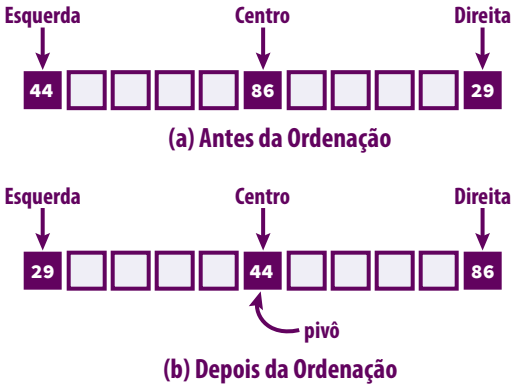

FIGURA 11–14: USO DE MEDIANA DE TRÊS COM REORDENAÇÃO EM QUICKSORT

Usando a abordagem mediana de três, o algoritmo **QUICKSORT** não funciona para partições com três ou menos itens. Nesse caso, o valor *3* é chamado **ponto de corte**[5]. Quando o tamanho de uma partição atinge o ponto de corte, ela é ordenada por outro algoritmo de ordenação diferente de **QUICKSORT** (tipicamente, **INSERTIONSORT**).

[3] O mesmo ocorre quando todas as chaves de ordenação são iguais.

[4] O leitor que não é habituado com Estatística não deve confundir mediana com média. A mediana de um conjunto finito de números é obtida ordenando-os. Se a quantidade de valores for ímpar, o valor central é a mediana. Se essa quantidade for par, considera-se a média dos valores centrais.

[5] Knuth (1997, Volume 3 — v. **Bibliografia**) recomenda um ponto de corte igual a 9.

Como o ponto de corte é relativamente bem pequeno, não importa que um algoritmo com custo temporal $\theta(n^2)$ seja usado nessa situação.

A escolha aleatória de um pivô é outra opção comumente utilizada com QuickSort. Nesse caso, o pivô é escolhido aleatoriamente entre os limites de cada subtabela resultante de uma partição. Com isso, espera-se que as partições sejam equitativas e previnam o pior caso. Quando o pivô é escolhido aleatoriamente, QuickSort torna-se um exemplo de algoritmo aleatório, pois seu desempenho depende das propriedades estatísticas do gerador de números aleatórios utilizado na escolha do pivô.

Implementação Básica

Nesta implementação, o pivô é escolhido como o primeiro elemento da tabela e a função `Quick1()`, apresentada a seguir, é a função auxiliar que implementa o algoritmo QuickSort. Os parâmetros dessa função são:

- `tabela` (entrada/saída) — tabela que será ordenada
- `inf` (entrada) — limite inferior da tabela
- `sup` (entrada) — limite superior da tabela

```
static void Quick1(int tabela[], int inf, int sup)
{
   int esq,   /* Índice da esquerda */
       dir,   /* Índice da direita  */
       iPivo; /* Índice do pivô     */

      /* Se a tabela está vazia ou contém apenas um elemento, ela já está ordenada */
   if (inf >= sup)
      return; /* A tabela já está ordenada */

      /* Nesta implementação, o pivô é sempre o primeiro elemento da tabela */
   iPivo = inf;

      /* Início da operação de partição */

   for (esq = inf, dir = sup; esq < dir; ) {
         /* Enquanto os elementos da primeira metade da tabela forem */
         /* menores do que o pivô,  incrementa-se o índice esquerdo  */
      while (tabela[esq] <= tabela[iPivo] && esq < sup)
         ++esq;

         /* Enquanto os elementos da segunda metade da tabela forem */
         /* maiores do que o pivô,  decrementa-se o índice direito  */
      while (tabela[dir] > tabela[iPivo])
         --dir;

         /* Se os elementos nos índices esq e dir estiverem  */
         /* em partições erradas, deve-se trocá-los de lugar */
      if (esq < dir)
         Troca(tabela + esq, tabela + dir, sizeof(tabela[0]));
   }

      /* Fim da operação de partição */

      /* Coloca o pivô em sua correta posição */
   Troca(tabela + iPivo, tabela + dir, sizeof(tabela[0]));

      /* Ordena recursivamente a primeira metade da tabela */
   Quick1(tabela, inf, dir - 1);

      /* Ordena recursivamente a segunda metade da tabela */
   Quick1(tabela, dir + 1, sup);
}
```

A função `Quick1()` reflete fielmente os algoritmos apresentados na **Figura 11–10** e na **Figura 11–11**. Essa função é implementada usando duas chamadas recursivas, sendo que a segunda delas é uma recursão de cauda que pode ser facilmente ser substituída por iteração (v. **Capítulo 4** do **Volume 1**), mas a primeira chamada só pode ser substituída com o uso explícito de pilha, que é incapaz de melhorar significativamente o desempenho do algoritmo.

Um detalhe, que talvez passe despercebido, referente à função `Quick1()` é que a expressão:

```
esq < sup
```

presente no primeiro laço **while** impede que sejam acessadas posições de memória além do limite superior da tabela quando seus elementos estão ordenados em ordem decrescente.

A função `QuickSort1()` abaixo é uma função acionadora (v. **Capítulo 4** do **Volume 1**) que provê uma interface para a função `Quick1()`, que de fato realiza a tarefa de ordenação.

```
void QuickSort1(int tabela[], int nElem)
{
      /* Simplesmente chama Quick1() para fazer o serviço */
   Quick1(tabela, 0, nElem-1);
}
```

Implementação com Mediana de Três e Ponto de Corte

A função `Quick2()`, encontrada no site dedicado a este livro na internet, difere da função `Quick1()` pelo fato de usar a abordagem de mediana de três com ordenação discutida acima. Além disso, a função `Quick2()` usa InsertionSort quando um ponto de corte é atingido.

Implementação com Pivô Aleatório

A função `Quick3()`, encontrada no site dedicado a este livro na internet, ordena uma tabela usando o algoritmo QuickSort com o pivô escolhido aleatoriamente.

Análise

Teorema 11.9: No pior caso, o número de comparações de chaves efetuadas por QuickSort é $n^2/2 + n/2 - 1$.

Prova: Seja $C(n)$ o número de comparações de chaves efetuadas pelo algoritmo QuickSort na ordenação de uma lista de tamanho n. Cada divisão da lista em partições compara cada elemento com o pivô, de modo que, na primeira dessas divisões, ocorrem n comparações. Se, nessa divisão, uma partição tiver tamanho t, a outra terá tamanho $n - t - 1$. Portanto pode-se escrever a seguinte relação de recorrência para representar a operação de partição:

$$C(n) = n + C(t) + C(n - t - 1) \qquad [\dagger]$$

No pior caso, tem-se que $t = 0$, de sorte que essa relação de recorrência torna-se:

$$C(n) = n + C(n - 1) \qquad [\dagger\dagger]$$

Nessa última passagem, levou-se em conta que $C(0) = 0$. Além disso, tem-se que $C(1) = 0$, pois, se uma lista só tem um elemento não há comparação de chaves (i.e., o algoritmo de partição não será sequer invocado).

A solução da relação de recorrência [††] será apresentada a seguir.

$$\begin{aligned} C(n) &= n + C(n-1) \\ &= n + n - 1 + C(n-2) \\ &= n + n - 1 + n - 2 + C(n-3) \end{aligned}$$

$$
\begin{aligned}
&= \ldots \\
&= n + n - 1 + n - 2 + \ldots + 3 + 2 + C(1) \\
&= n + n - 1 + n - 2 + \ldots + 3 + 2 + 0 \\
&= (1 + 2 + 3 + \ldots + n - 2 + n - 1 + n) - 1 \\
&= \left(\sum_{i=1}^{n} i\right) - 1 = \frac{n \cdot (n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1
\end{aligned}
$$

■

Corolário 11.1: O custo temporal do algoritmo **QuickSort** é $\theta(n^2)$ no pior caso.

Prova: Essa afirmação é consequência direta do **Teorema 11.9**. ■

Teorema 11.10: No pior caso, o número de trocas efetuadas por **QuickSort**, é $n^2/2 + n/2 - 1$.

Prova: Seja $T(n)$ o número de trocas efetuadas pelo algoritmo **QuickSort** ao ordenar uma tabela de tamanho n. O algoritmo de partição efetua n comparações. No pior caso, o pivô contém a maior (ou menor) chave da lista, de modo que o algoritmo de partição efetua n trocas (todas elas envolvendo o pivô). Assim, no pior caso, obtém-se a seguinte relação de recorrência:

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

Essa relação de recorrência é a mesma obtida na prova do **Teorema 11.9** e sua solução é $n^2/2 + n/2 - 1$.

■

Teorema 11.11: No melhor caso, o custo temporal do algoritmo **QuickSort** é $\theta(n \cdot \log n)$.

Prova: Supondo que uma partição tenha tamanho t, a relação de recorrência que se obtém é aquela representada por [†] na prova do **Teorema 11.9**.

No melhor caso, o algoritmo de partição divide a tabela em duas partes aproximadamente iguais. Isto é, uma partição terá $\lfloor (n - 1)/2 \rfloor$ elementos e a outra terá $\lceil (n - 1)/2 \rceil$ elementos. Como não se está tentando determinar o número exato de trocas ou comparações, esses números de elementos podem ser ambos aproximados por $n/2$. Desse modo, a relação de recorrência [†] torna-se:

$C(n) = n + 2 \cdot C(n/2)$

Seja k o número de vezes que a tabela é dividida até que cada partição tenha tamanho unitário (i.e., k é a profundidade da árvore de recursão — v. **Capítulo 6** do **Volume 1**). Então o custo temporal de **QuickSort** é dado por:

$$
\begin{aligned}
C(n) &= n + 2 \cdot C(n/2) \\
&= n + 2 \cdot (n/2 + 2 \cdot C(n/2^2)) \\
&= n + n + 2^2 \cdot C(n/2^2) \\
&= 2 \cdot n + 2^2 \cdot (n/2^2 + 2 \cdot C(n/2^3)) \\
&= 3 \cdot n + 2^3 \cdot C(n/2^3) \\
&= \ldots \\
&= k \cdot n + 2^k \cdot C(n/2^k)
\end{aligned}
$$

Como se supõe que k é o número de vezes que a tabela é dividida, tem-se que:

$n/2^k = 1 \Rightarrow k = \log n$

Levando isso em conta, obtém-se:

$C(n) = k \cdot n + 2^k \cdot C(n/2^k) = n \cdot \log n + n \cdot C(1) = n \cdot \log n + c \cdot n$, em que c é uma constante, visto que, quando $n = 1$, o algoritmo simplesmente retorna após verificar esse fato.

Esse último resultado mostra que o custo temporal de **QuickSort** é $\theta(n \cdot \log n)$ no melhor caso. ■

Teorema 11.12: O custo temporal do algoritmo **QuickSort** básico no caso médio é $\theta(n \cdot \log n)$.

Prova: A prova desse teorema requer conhecimento de teoria das probabilidade e, por ser relativamente longa, não será apresentada aqui. Recomenda-se que o leitor interessado nessa prova consulte Cormen et al. (2009) (v. **Bibliografia**).

Teorema 11.13: O custo temporal esperado de **QuickSort** aleatório é $\theta(n \cdot \log n)$.

Prova: A prova desse teorema também requer conhecimento de teoria das probabilidade e está fora do escopo deste livro. O leitor interessado nessa prova deve consultar Cormen et al. (2009) (v. **Bibliografia**).

O algoritmo **QuickSort** não requer tabela adicional, mas ele usa uma abordagem recursiva, de modo que registros de ativação de muitas chamadas recursivas podem ser armazenados na pilha de execução num dado instante. Assim, no melhor caso, esse algoritmo requer $\theta(\log n)$ de espaço extra para conter dados relativos a cada chamada (i.e., registros de ativação — v. **Capítulo 4** do **Volume 1**) e, no pior caso, esse algoritmo requer espaço extra com custo $\theta(n)$. Os dois próximos teoremas apresentam esses resultados formalmente.

Teorema 11.14: No melhor caso, o custo espacial do algoritmo **QuickSort** é $\theta(\log n)$.

Prova: Em cada nível de recursão apresentado na **Figura 11–15**, o algoritmo **QuickSort** efetua duas chamadas recursivas, sendo que cada uma delas é responsável pela alocação de um registro de ativação (v. **Capítulo 4** do **Volume 1**). Cada uma dessas chamadas tem custo espacial $\theta(1)$, pois ela usa a tabela original. Como há $\theta(\log n)$ chamadas recursivas, o custo espacial do algoritmo **QuickSort** no melhor caso é $\theta(\log n)$. ■

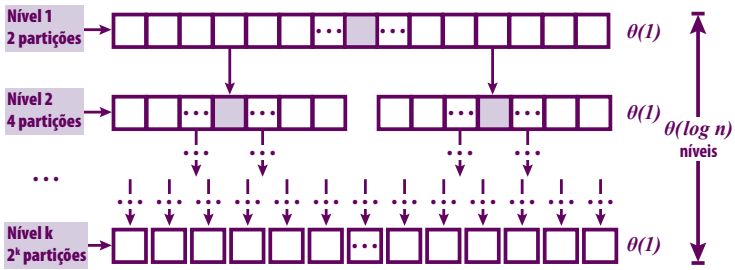

Figura 11–15: Custo Espacial do Melhor Caso de QuickSort

Teorema 11.15: No pior caso, o custo espacial do algoritmo **QuickSort** é $\theta(n)$.

Prova: A prova é semelhante àquela do **Teorema 11.14**. A diferença é que, agora, existem $\theta(n)$ chamadas recursivas (v. **Figura 11–16**). ■

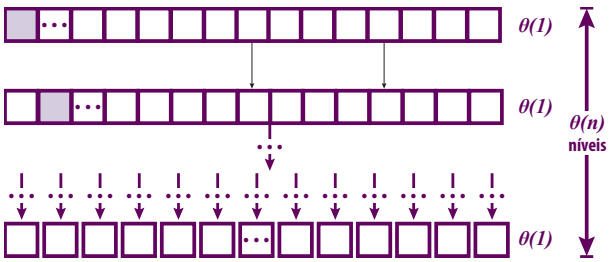

Figura 11–16: Custo Espacial do Pior Caso de QuickSort

O algoritmo **QuickSort** não é estável, pois, durante o passo de partição da tabela, os registros podem mudar de posição, de modo que, ao final, dois registros com mesmas chaves podem ter suas posições relativas iniciais trocadas.

Em geral, QUICKSORT é o melhor para tabelas grandes e, escolhendo-se cuidadosamente o pivô, pode-se tornar o pior caso muito pouco provável. A **Tabela 11–4** exibe um resumo da avaliação do algoritmo QUICKSORT.

CUSTO TEMPORAL	❒ Melhor caso: $\theta(n \cdot \log n)$ ❒ Caso médio: $\theta(n \cdot \log n)$ ❒ Pior caso: $\theta(n^2)$
VANTAGENS	❒ Muito rápido em média ❒ Pior caso é muito raro
DESVANTAGENS	❒ Eficiência depende da escolha correta do pivô ❒ Custo espacial $\theta(\log n)$ no melhor caso e $\theta(n)$ no pior caso ❒ Instável
INDICAÇÕES	❒ Situações práticas nas quais o estado inicial de ordenação da tabela seja desconhecido

TABELA 11–4: ANÁLISE RESUMIDA DE QUICKSORT

11.3.2 Ordenação por Intercalação (MergeSort)

Descrição

O algoritmo de **ordenação por intercalação** (MERGESORT) é baseado no conceito de **intercalação**. Intercalar duas tabelas ordenadas significa criar uma terceira tabela ordenada que contém todos os elementos de ambas as tabelas ordenadas. Mais precisamente, o procedimento de intercalação recebe como entrada duas tabelas já ordenadas e produz uma nova tabela ordenada contendo, em ordem, todos os elementos das tabelas de entrada. Em MERGESORT, subtabelas contendo um elemento são intercaladas em subtabelas de dois elementos, subtabelas de dois elementos são intercaladas em subtabelas de quatro elementos e assim por diante até que a tabela inteira esteja ordenada. A **Figura 11–17** ilustra um exemplo detalhado de intercalação. Nesse processo, os itens no início de cada tabela são comparados e o menor deles é acrescentado à tabela que conterá o resultado. Então o próximo item da tabela da qual o menor item foi copiado é considerado e o processo se repete até que todos os elementos de uma das tabelas ou de ambas as tabelas tenham sido levados em conta. Caso restem elementos numa das tabelas que ainda não tenham sido levados em consideração, eles são simplesmente copiados para a tabela resultante, como mostra a **Figura 11–17 (d)**.

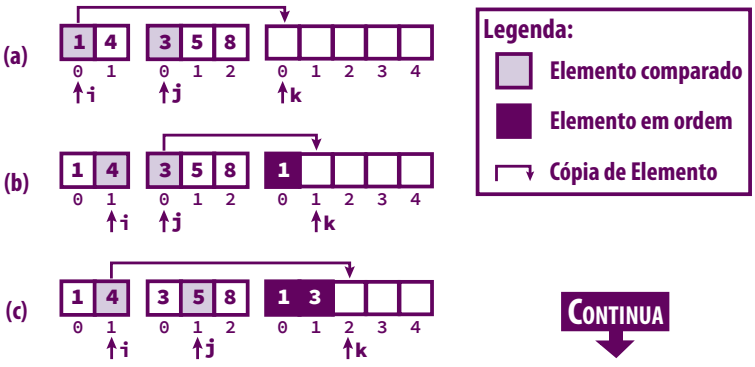

FIGURA 11–17: EXEMPLO DE INTERCALAÇÃO DE TABELAS

FIGURA 11–17 (CONT.): EXEMPLO DE INTERCALAÇÃO DE TABELAS

O algoritmo **MERGESORT** é apresentado na **Figura 11–18**.

ALGORITMO MERGESORT

ENTRADA/SAÍDA: Uma tabela indexada de *n* elementos

1. Se a tabela contiver menos de dois elementos, encerre
2. Divida a tabela em duas partes, cada uma das quais contendo a metade dos elementos da tabela original
3. Ordene a primeira subtabela usando **MERGESORT**
4. Ordene a segunda subtabela usando **MERGESORT**
5. Intercale as duas subtabelas ordenadas usando o algoritmo **INTERCALATABELAS**

FIGURA 11–18: ALGORITMO MERGESORT

O algoritmo de intercalação invocado por **MERGESORT** é exibido na **Figura 11–19**.

ALGORITMO INTERCALATABELAS

ENTRADA: Duas tabelas indexadas ordenadas

SAÍDA: Uma tabela indexada ordenada (tabela auxiliar)

1. Atribua a uma variável *i* o índice do primeiro elemento da primeira tabela (essa variável indicará a posição do elemento sob consideração na primeira tabela)
2. Atribua a uma variável *j* o índice do primeiro elemento da segunda tabela (essa variável indicará a posição do elemento sob consideração na segunda tabela)
3. Atribua a uma variável *k* o índice do primeiro elemento da tabela auxiliar (essa variável indicará a posição do elemento sob consideração na tabela auxiliar)
4. Enquanto *i* não for maior do que o índice do último elemento da primeira tabela e *j* não for maior do que o índice do último elemento da segunda tabela, faça o seguinte:
   - **4.1** Se o elemento na posição *i* da primeira tabela for menor do que ou igual ao elemento na posição *j* da segunda tabela
     - **4.1.1** Acrescente o referido elemento da primeira tabela na tabela auxiliar
     - **4.1.2** Incremente *i*
   - **4.2** Caso contrário, faça o seguinte:
     - **4.2.1** Acrescente na tabela auxiliar o elemento que se encontra na posição *j* da segunda tabela
     - **4.2.2** Incremente *j*
   - **4.3** Incremente *k*
5. Enquanto *i* não for maior do que o índice do último elemento da primeira tabela, faça:
   - **5.1** Acrescente o elemento na posição *i* da primeira tabela na tabela auxiliar

CONTINUA

FIGURA 11–19: ALGORITMO DE INTERCALAÇÃO USADO POR MERGESORT

Algoritmo IntercalaTabelas (Continuação)

5.2 Incremente *i*
5.3 Incremente *k*
6. Enquanto *j* não for maior do que o índice do último elemento da segunda tabela, faça:
6.1 Acrescente o elemento na posição *j* da segunda tabela na tabela auxiliar
6.2 Incremente *j*
6.3 Incremente *k*

Figura 11–19 (Cont.): Algoritmo de Intercalação Usado por MergeSort

A diferença principal entre **MergeSort** e **QuickSort** é que o modo como a tabela é dividida em **MergeSort** é completamente independente dos dados de entrada. **MergeSort** simplesmente divide a tabela, enquanto **QuickSort** partilha a tabela com base na chave de um determinado elemento (o pivô), que pode dividir a tabela em qualquer ponto.

A **Figura 11–20** ilustra um exemplo de ordenação efetuada pelo algoritmo **MergeSort**.

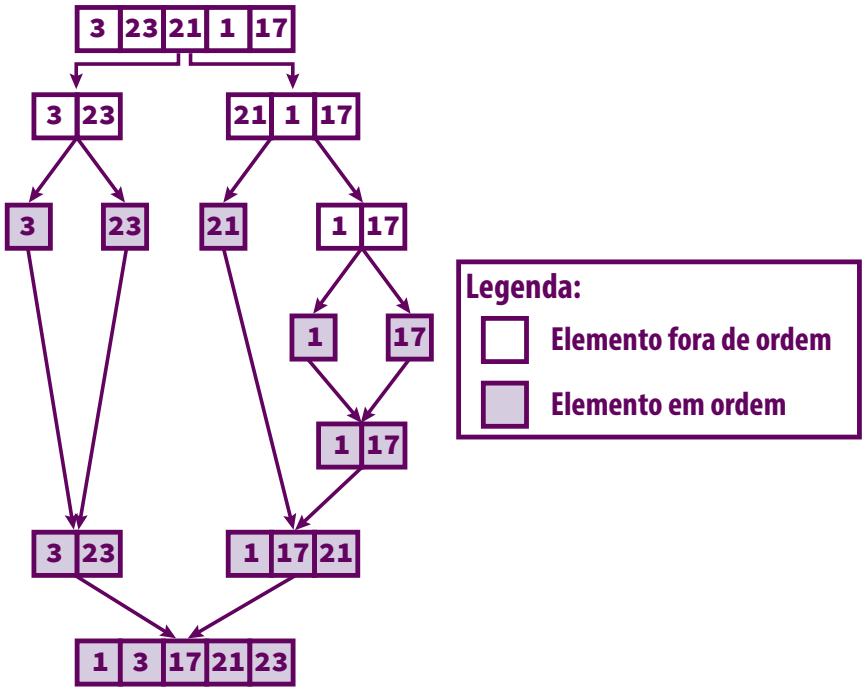

Figura 11–20: Exemplo de Ordenação Usando MergeSort

Implementação

A função `Intercala2Tabelas()` faz a intercalação de duas subtabelas e tem como parâmetros:

- `tabela` (entrada/saída) — tabela contendo as duas subtabela que serão intercaladas
- `aux` (entrada/saída) — tabela auxiliar que conterá o resultado
- `iniEsq` (entrada) — início da metade da esquerda
- `iniDir` (entrada) — início da metade da direita
- `finalDir` (entrada) — final da metade da direita

```
static void Intercala2Tabelas( int tabela[], int aux[], int iniEsq, int iniDir,
                               int finalDir )
{
   int i, finalEsq, nElementos, posAux;

   finalEsq = iniDir - 1;
   posAux = iniEsq;
   nElementos = finalDir - iniEsq + 1;

      /* Intercala as duas metades e coloca o resultado na tabela auxiliar */
   while((iniEsq <= finalEsq) && (iniDir <= finalDir))
      if(tabela[iniEsq] <= tabela[iniDir])
         aux[posAux++] = tabela[iniEsq++];
      else
         aux[posAux++] = tabela[iniDir++];

      /* Copia o restante da primeira metade */
   while(iniEsq <= finalEsq)
      aux[posAux++] = tabela[iniEsq++];

      /* Copia o restante da segunda metade */
   while(iniDir <= finalDir)
      aux[posAux++] = tabela[iniDir++];

      /* Copia a tabela auxiliar de volta na tabela original */
   for(i = 0; i < nElementos; i++,finalDir--)
      tabela[finalDir] = aux[finalDir];
}
```

A função `MergeSortAux()` ordena uma tabela usando intercalação com uma tabela auxiliar. Os parâmetros dessa função são:

- `tabela` (entrada/saída) — tabela a ser ordenada
- `aux` (entrada/saída) — tabela auxiliar
- `esquerda` (entrada) — início da metade da esquerda da tabela
- `direita` (entrada) — início da metade da direita da tabela

```
static void MergeSortAux(int *tabela, int *aux, int esquerda, int direita)
{
   int centro;

   if(esquerda < direita) {
      centro = esquerda + (direita - esquerda)/2;

         /* Ordena metade inferior da tabela */
      MergeSortAux(tabela, aux, esquerda, centro);
         /* Ordena metade superior da tabela */
      MergeSortAux(tabela, aux, centro + 1, direita);

         /* Intercala as duas metades da tabela na tabela auxiliar */
      Intercala2Tabelas(tabela, aux, esquerda, centro + 1, direita);
   }
}
```

Cuidado: A expressão utilizada para calcular o índice central da tabela poderia ser calculado usando a expressão aparentemente equivalente:

```
centro = (esquerda + direita)/2;
```

Essa última expressão pode, entretanto, causar overflow (v. **Seção 3.5.1**).

A função `MergeSort()` cria uma tabela auxiliar e chama `MergeSortAux()` para completar o serviço de ordenação.

```
void MergeSort(int tabela[], int nElem)
{
   int *aux;

      /* Aloca espaço para a tabela auxiliar */
   aux = malloc(nElem*sizeof(int));
   ASSEGURA(aux, ERRO: Impossivel alocar tabela auxiliar);

      /* Chama MergeSortAux() para fazer o serviço */
   MergeSortAux(tabela, aux, 0, nElem - 1);

   free(aux); /* Libera o espaço ocupado pela tabela auxiliar */
}
```

Ordenação de Listas Encadeadas com MergeSort

Dentre os algoritmos de ordenação discutidos neste capítulo, MERGESORT é o mais adequado para ordenar listas simplesmente encadeadas. Do mesmo modo que o algoritmo MERGESORT descrito acima, esse algoritmo aplicado à ordenação de listas encadeadas requer que a lista ora sendo ordenada seja dividida ao meio, o que tem custo temporal $\theta(n)$, já que não se pode encontrar o meio de uma lista encadeada sem que se efetue um acesso sequencial completo na lista. Porém, em compensação, diferentemente do que ocorre com o algoritmo MERGESORT original, o custo espacial de MERGESORT para listas encadeadas é $\theta(1)$. Quer dizer, ele não usa espaço adicional.

A **Figura 11–21** mostra um exemplo de intercalação de duas listas simplesmente encadeadas previamente ordenadas. Essa intercalação consiste simplesmente em comparar o primeiro elemento de cada lista e acrescentar o menor deles ao final da lista que resultará da operação. Note nessa figura que a soma do número de elementos e do número de ponteiros é o mesmo em cada passo da intercalação, o que mostra (informalmente) que o custo espacial desse algoritmo é, de fato, $\theta(1)$.

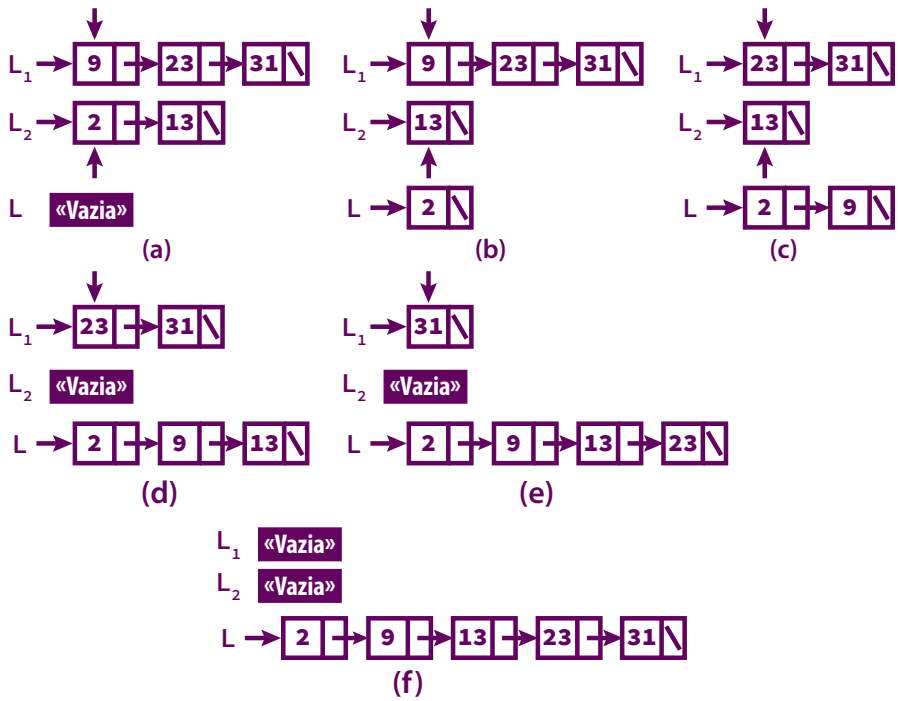

FIGURA 11–21: INTERCALAÇÃO DE DUAS LISTAS ENCADEADAS

O algoritmo **MergeSort** dedicado à ordenação de listas encadeadas segue o algoritmo apresentado na **Figura 11–22**.

Algoritmo MergeSortDeListaEncadeada

Entrada/Saída: Uma lista encadeada L

1. Se a lista L estiver vazia ou contiver apenas um elemento, encerre
2. Divida a lista em duas metades L_1 e L_2
3. Ordene a lista L_1 usando este mesmo algoritmo
4. Ordene a lista L_2 usando este mesmo algoritmo
5. Intercale as listas L_1 e L_2

Figura 11–22: Algoritmo MergeSort para Listas Encadeadas

O custo temporal do algoritmo apresentado na **Figura 11–22** é $\theta(n \log n)$ e, se não for usada recursão no passo de intercalação, o custo espacial desse algoritmo, surpreendentemente, é $\theta(1)$.

Análise

Lema 11.4: A intercalação de duas tabelas efetuada pelo algoritmo **IntercalaTabelas** resultando numa lista de tamanho n realiza, no máximo, $n - 1$ comparações de chaves.

Prova: No pior caso, cada elemento (com uma exceção) precisa ser comparado com outro antes de ser acrescentado à lista resultante da intercalação. A única exceção é o último elemento acrescentado a essa lista, que não é comparado com nenhum outro elemento. Portanto, se a lista resultante possuir n elementos, no máximo, $n - 1$ deles serão comparados. ■

Lema 11.5: O custo temporal $T(n)$ do algoritmo **MergeSort** satisfaz a seguinte relação:

$$\tfrac{1}{2}\, n \log n \leq T(n) \leq 2n \log n, \text{ para } n \geq 1$$

Prova: A relação de recorrência associada ao algoritmo **MergeSort** pode ser escrita como:

$$T(n) = \begin{cases} 0 & \text{se } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 & \text{se } n > 1 \end{cases}$$

A justificativa para essa afirmação é baseada no número de comparações necessárias para ordenar um array de tamanho n usando **MergeSort** e é a seguinte:

- O número de comparações necessárias para ordenar um array usando **MergeSort** é igual a zero quando o array só possui um elemento, visto que, nesse caso, nenhuma comparação é necessária.
- Quando o array tem mais de um elemento, o número de comparações necessárias para ordená-lo usando **MergeSort** é igual ao número de comparações necessárias para ordenar cada uma de sua partições, sendo que uma delas tem tamanho $\lfloor n/2 \rfloor$ e o tamanho da outra é $\lceil n/2 \rceil$. Além disso, deve-se acrescentar o número de comparações necessárias para intercalar esses arrays ordenados, que, de acordo com **Lema 11.4**, é $n - 1$.

O restante da prova é longa e enfadonha, de modo que sugere-se ao leitor afeito à matemática que tente completá-la (v. questão **87**). (A prova completa encontra-se no **Apêndice E**.) ■

Teorema 11.16: O custo temporal do algoritmo **MergeSort** é $\theta(n \cdot \log n)$ em qualquer caso.

Prova: Seja $T(n)$ o custo temporal do algoritmo **MergeSort**. De acordo com o **Lema 11.5**, $T(n) \geq \frac{1}{2}\, n \log n$, o que implica que $T(n)$ é $\Omega(n \cdot \log n)$. Tem-se ainda, de acordo com o mesmo lema, que $T(n) \leq 2n \log n$,

o que acarreta em $T(n)$ ser $O(n \cdot \log n)$. Como $T(n)$ é $\Omega(n \cdot \log n)$ e $T(n)$ é $O(n \cdot \log n)$, conclui-se que $T(n)$ é $\theta(n \cdot \log n)$. ■

Teorema 11.17: Em qualquer caso, o custo espacial do algoritmo MergeSort é $\theta(n)$.

Prova: A etapa de intercalação do algoritmo MergeSort usa uma tabela auxiliar com tamanho igual à soma dos tamanhos das tabelas ora sendo concatenadas. No último nível de concatenação, tem-se uma tabela de tamanho $\lceil n/2 \rceil$ e outra de tamanho $\lfloor n/2 \rfloor$, de modo que, nesse caso, o tamanho da tabela auxiliar é n, e não há tabela auxiliar com tamanho maior do que esse. Além disso, assim como faz QuickSort, o algoritmo MergeSort efetua $\theta(\log n)$ chamadas recursivas, mas o custo $\theta(n)$ devido ao uso de array auxiliar domina o custo espacial $\theta(\log n)$ decorrente do uso de recursão. Quer dizer, aqui aplica-se a regra da soma, e não a regra do produto (v. **Capítulo 6** do **Volume 1**). Portanto o custo espacial do algoritmo MergeSort é $\theta(n)$. ■

A ideia que norteia MergeSort é difícil de ser superada por qualquer outro algoritmo de ordenação em algumas situações específicas como, por exemplo, ordenação de listas encadeadas (v. **Seção 11.8.1**) e ordenação de arquivos (v. **Capítulo 12**).

Uma desvantagem do algoritmo MergeSort decorre do fato de ele requerer uma tabela auxiliar que é tão grande quanto a tabela a ser ordenada. Portanto, se essa tabela for grande e o espaço disponível em memória for crítico, esse algoritmo de ordenação pode não ser uma escolha apropriada. A **Tabela 11–5** mostra um resumo da avaliação do algoritmo MergeSort.

Custo temporal	❒ $\theta(n \cdot \log n)$ nos três casos
Vantagens	❒ No pior caso, é mais rápido do que QuickSort ❒ Mais fácil de implementar do que QuickSort ❒ Não é afetado por ordenação prévia ❒ Estável
Desvantagens	❒ Usa espaço adicional igual ao tamanho da tabela original
Indicações	❒ Quando há espaço sobrando em memória ❒ Ordenação em memória secundária

Tabela 11–5: Análise Resumida de MergeSort

11.3.3 Ordenação com Heap (HeapSort)

Descrição

No **Capítulo 10**, discutiu-se heap, uma estrutura de dados com uma característica muito especial que é o fato de sempre se saber onde encontrar seu maior elemento (ou menor elemento, dependendo do tipo de heap), que está localizado na raiz. Pode-se levar vantagem disso e usar heap para ordenação. Nesse caso, se a ordenação pretendida for em ordem crescente, como será feito aqui, usa-se um heap de máximo e se a ordenação desejada for em ordem decrescente, usa-se um heap de mínimo.

HeapSort faz parte da família de algoritmos de ordenação por seleção. Essa família de algoritmos funciona determinando o maior (ou o menor) elemento da tabela a ser ordenada e colocando-o no início (ou no final) da tabela resultante da ordenação. Esse algoritmo é apresentado na **Figura 11–23**.

Algoritmo HeapSort

Entrada/Saída: Uma tabela indexada de *n* elementos

1. Construa um heap de máximo (v. **Seção 10.2**) usando os elementos da tabela a ser ordenada.
2. Enquanto o heap contiver mais de um elemento faça o seguinte:
   - **2.1** Guarde o valor do último elemento do heap.
   - **2.2** Remova o maior elemento do heap e armazene-o no lugar do último elemento do heap. Esse último elemento (i.e., a antiga raiz) deixa de fazer parte do heap e passa a fazer parte da tabela ordenada.
   - **2.3** Insira no heap o elemento que foi guardado. (Este passo reordena o heap, como foi visto na **Seção 10.2**.)

Figura 11–23: Algoritmo HeapSort

No **Capítulo 10**, foi visto como heaps podem ser representados num array. De acordo com a propriedade estrutural, sabe-se que os elementos de um heap ocupam posições consecutivas no array. De fato, qualquer array, ordenado ou não, satisfaz a propriedade estrutural de heaps. A **Figura 11–24** mostra um array desordenado e sua árvore completa associada (que ainda não é heap).

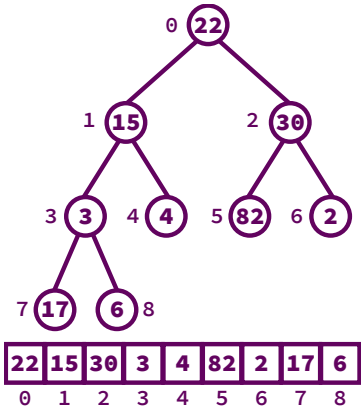

Figura 11–24: Array Desordenado e sua Árvore Completa Associada

Para transformar a árvore associada a um array num heap é necessário fazer com que ela satisfaça a propriedade de ordenação de heaps. Primeiro, verifica-se se qualquer parte da árvore já satisfaz a propriedade de ordenação. Na **Figura 11–25 (a)**, as subárvores cujas raízes contêm os valores *17*, *6*, *4*, *82* e *2* são heaps porque essas raízes são folhas.

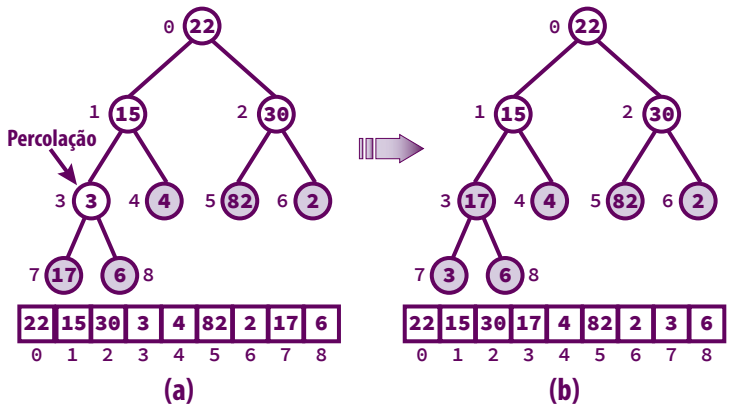

Figura 11–25: Transformação de Árvore Completa em Heap 1

A seguir, examina-se, de baixo para cima da árvore, seu primeiro nó interno, que é aquele que é pai da última folha da árvore. Como, de acordo com o **Teorema 10.1**, o índice do pai de um nó é dado por $\lfloor (i-1)/2 \rfloor$ e o índice da última folha é $n-1$, o índice desse primeiro nó é obtido usando a fórmula: $\lfloor (n-2)/2 \rfloor$. Na **Figura 11–24**, o índice desse primeiro nó é *3* (e seu conteúdo também é *3*). A subárvore enraizada nesse nó não é um heap, mas é quase um heap, pois todos os nós, exceto a raiz, dessa subárvore satisfazem a propriedade de ordenação. No **Capítulo 10**, apresentou-se o algoritmo de percolação descendente, que pode ser usada para corrigir esse problema. Dada uma árvore cujos elementos satisfazem a propriedade de ordenação de heaps exceto a raiz, esse algoritmo rearranja os nós, de modo que se garanta que a árvore será um heap. Aplicando-se esse algoritmo de percolação a todos os nós desde o pai da última folha até a raiz da árvore, ao final a árvore estará ordenada como um heap. A **Figura 11–25** mostra a percolação do pai da última folha, enquanto a **Figura 11–26** exibe as percolações dos nós que se encontram no próximo nível acima. Finalmente, a **Figura 11–27** ilustra a percolação da raiz da árvore.

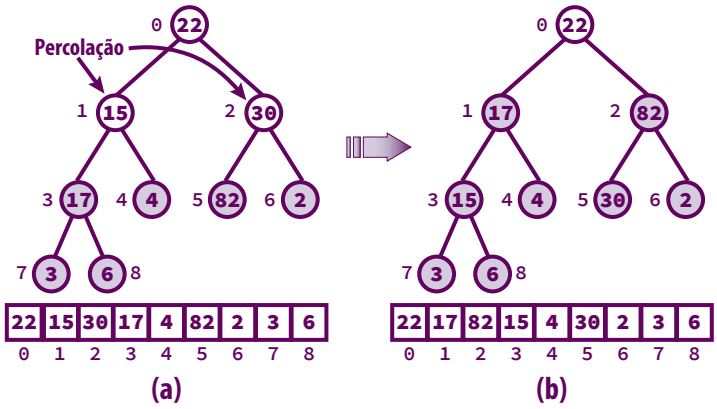

Figura 11–26: Transformação de Árvore Completa em Heap 2

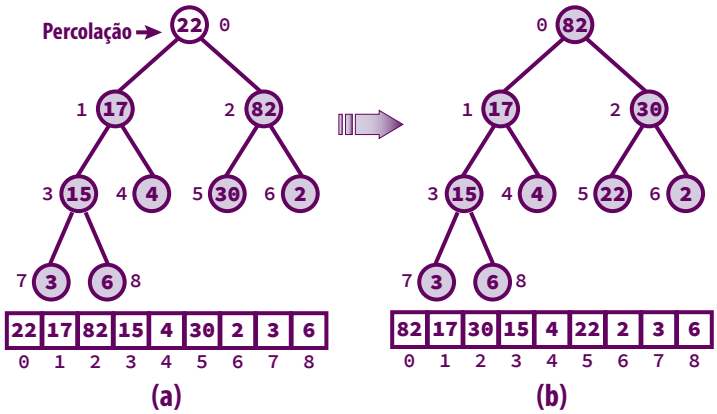

Figura 11–27: Transformação de Árvore Completa em Heap 3

Tendo construído o heap conforme foi descrito acima, resta utilizá-lo para ordenar o array (**Passo 2** do algoritmo da **Figura 11–27**). A **Figura 11–28** ilustra o processo de remoção do heap e subsequente inserção no array ordenado. Esse processo é repetido até que todos os elementos do array estejam em suas posições corretas. Isto é, até que o heap contenha apenas um único elemento, que deve ser o menor item no array. Essa localização é sua correta posição, de modo que o array agora está completamente ordenado do menor para o maior elemento. Note que em cada iteração o tamanho da porção desordenada (representada por um heap) torna-se cada vez menor e o tamanho da porção ordenada torna-se cada vez maior. No final do algoritmo, o tamanho da porção ordenada torna-se igual ao tamanho do array original. Note que o heap em **HeapSort** é apenas uma

estrutura temporária interna ao algoritmo. Ele é criado no início do processo para auxiliar a ordenação e então é gradativamente reduzido à medida que a parte ordenada do array cresce. Ao final do processo, a parte ordenada ocupa todo o array e o heap desaparece.

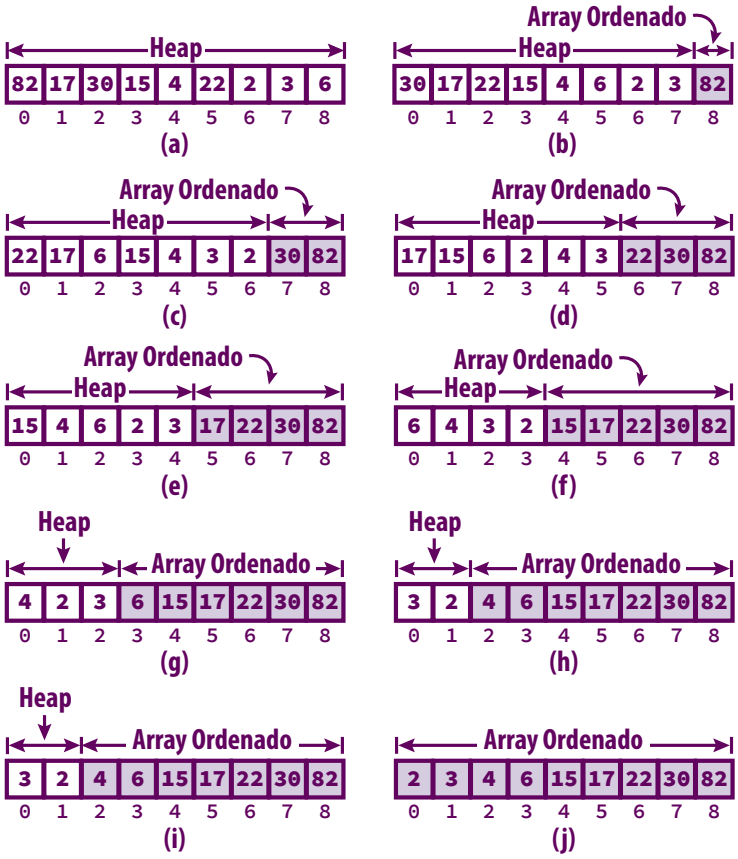

Figura 11–28: Exemplo de Ordenação Usando HeapSort

Implementação

A função `HeapSort()` a seguir ordena uma tabela usando o algoritmo HeapSort.

```
void HeapSort(int tabela[], int nElem)
{
   int i;

      /* Constrói o heap a partir do pai do último elemento da tabela */
   for(i = (nElem - 2)/2; i >= 0; i--)
      OrdenaHeap(tabela, i, nElem);

      /* Neste ponto, o heap está construído. Troca-se o primeiro elemento */
      /* do heap, diminuído de um elemento, com o último elemento e        */
      /* reordena-se o heap.                                               */
   for(i = nElem-1; i >= 1; i--) {
         /* Remove o maior elemento do heap e coloca no final da tabela */
      TrocaGenerica(tabela, tabela + i, sizeof(tabela[0]));

      OrdenaHeap(tabela, 0, i);  /* Reordena o heap */
   }
}
```

A função `OrdenaHeap()`, chamada por `HeapSort()`, é similar àquela de idêntica denominação discutida no **Capítulo 10**.

Análise

A operação óbvia de criação de um heap a partir de um array consiste em inserir cada elemento do array no heap. Como cada operação de inserção tem custo $\theta(\log n)$, construir um heap com n nós dessa maneira tem custo temporal $\theta(n \cdot \log n)$. Mas a operação de criação de heap implementada pela função `HeapSort()` tem custo temporal $\theta(n)$, como afirma o **Teorema 11.18** adiante.

Lema 11.6: Para todo $x \in \mathbb{R}$, se $0 < x < 1$, então:

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

Prova: A prova dessa afirmação é relativamente fácil, mas requer conhecimento de séries de potências e cálculo diferencial, o que está além do escopo deste livro.

Teorema 11.18: A criação de um heap tem custo temporal $\theta(n)$.

Prova: A abordagem utilizada consiste, inicialmente, em transformar em heap a subárvore cuja raiz se encontra na posição $\lfloor n/2 \rfloor - 1$. Depois transforma-se em heap a subárvore cuja raiz está na posição $\lfloor n/2 \rfloor - 2$ e assim por diante até que se tenha transformado em heap a árvore com raiz na posição 0.

Suponha que o heap é uma árvore binária repleta[6]. Então, em cada nível i dessa árvore, há 2^{i-1} nós. No último nível da árvore, não ocorre percolação, pois todos os nós nesse nível são folhas. No penúltimo nível, a percolação desce, no máximo, um nível da árvore. No antepenúltimo nível, a percolação desce, no máximo, dois níveis da árvore. E assim por diante, de modo que, sendo a altura da árvore, o número total de comparações (i.e., descidas na árvore) efetuadas nas percolações necessárias para transformar o array em heap é dado por:

$$\sum_{i=0}^{a} i \cdot 2^{a-i} = 2^a \cdot \sum_{i=0}^{a} \frac{i}{2^i} < 2^a \cdot \sum_{i=0}^{\infty} i \cdot \left(\frac{1}{2}\right)^i$$

Agora, de acordo com o **Lema 11.6**, tem-se que:

$$2^a \cdot \sum_{i=0}^{\infty} i \cdot \left(\frac{1}{2}\right)^i = 2^a \cdot 2 = 2^{a+1}$$

Portanto tem-se que o número de comparações efetuadas na ordenação do heap é menor do que 2^{a+1}. Mas, como se supõe que a árvore é repleta, sua altura a é dada por: $a = \lfloor \log n \rfloor + 1$, em que n é o número de nós da árvore. Assim esse número de comparações é menor do que $n + 1$, que é $\theta(n)$. Como o algoritmo de criação do heap deve acessar cada elemento do array, pelo menos, uma vez, tem-se ainda que seu custo temporal mínimo é $\Omega(n)$. Consequentemente, tem-se que o custo temporal de construção de um heap é $\theta(n)$. ■

Teorema 11.19: Em qualquer caso, o custo temporal do algoritmo **HeapSort** é $\theta(n \cdot \log n)$.

Prova: De acordo com o **Teorema 11.18**, custo temporal de criação do heap (**Passo 1** do algoritmo **HeapSort**) é $\theta(n)$ e, conforme o **Teorema 10.3**, o custo temporal de uma operação de remoção é $\theta(\log n)$. Como o algoritmo **HeapSort** efetua n operações de remoção, o custo temporal do **Passo 2** desse algoritmo é $\theta(n \cdot \log n)$. Usando o teorema da soma da análise assintótica, o custo temporal do algoritmo **HeapSort** é $\theta(n \cdot \log n)$. ■

[6] Essa suposição afasta a necessidade de uso de piso e teto e não prejudica a prova.

O algoritmo **HeapSort** é instável e, para entender a razão disso, considere dois registros R_1 e R_2 que tenham a mesma chave e que essa seja a maior chave encontrada na tabela. Então, se R_1 preceder R_2 antes da ordenação, após a ordenação, R_1 sucederá R_2. A **Figura 11–29** ilustra essa situação. Nessa figura, R_1 seria representado pelo elemento com fundo escurecido e R_2 seria o elemento com fundo branco que tem a mesma chave de R_1.

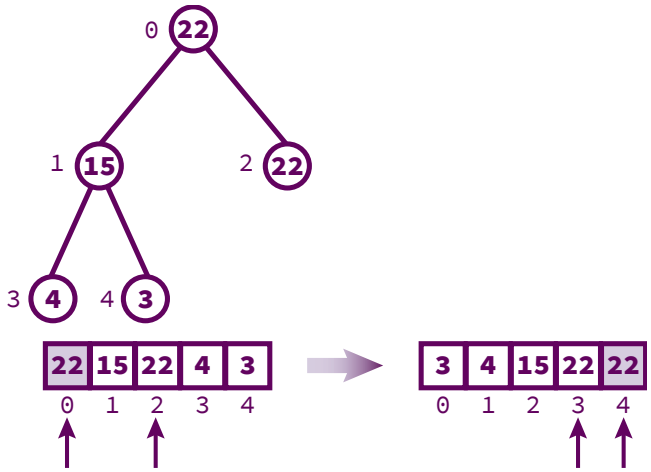

Figura 11–29: Instabilidade de HeapSort

Para arrays pequenos, o algoritmo **HeapSort** não é muito eficiente por causa do ônus associado à criação do heap, mas, para arrays grandes, ele é bem eficiente. Além disso, diferentemente de **QuickSort**, a eficiência de **HeapSort** não é afetada pela ordenação inicial dos elementos, embora, na prática, ele seja um pouco mais lento do que uma boa implementação de **QuickSort**. **HeapSort** também é eficiente em termos de espaço, pois ele não usa espaço adicional. Por outro lado, dentre os métodos de ordenação discutidos neste livro, **HeapSort** é o mais difícil de tornar estável, pois não há como ordenar de modo estável uma tabela usando as propriedades originais de um heap.

Um resumo da avaliação do algoritmo **HeapSort** é visto na **Tabela 11–6**.

Custo temporal	❒ $\theta(n \cdot \log n)$ nos três casos
Vantagens	❒ In loco ❒ Não usa ponteiros ❒ É rápido, mesmo no pior caso
Desvantagens	❒ Não é muito eficiente para tabelas pequenas ❒ Instável ❒ Desordena a tabela no primeiro passo, se ela já estiver ordenada
Indicações	❒ Tabelas que não sejam muito pequenas

Tabela 11–6: Análise Resumida de HeapSort

11.4 Ordenação com Custo Temporal Linear

Existem algoritmos capazes de ter custos temporais menores do que $\Omega(n \cdot \log n)$, mas eles requerem suposições especiais sobre as chaves de ordenação. Mesmo assim, tais situações frequentemente aparecem na prática, de modo que discuti-los vale a pena. Nesta seção, serão discutidos três desses algoritmos.

11.4.1 Ordenação por Contagem (CountingSort)

Descrição

Tipicamente, a entrada para o algoritmo de ordenação por contagem (CountingSort) é uma tabela com n elementos, cada um dos quais contém uma chave inteira não negativa cujo maior valor é k. Esse valor k deve ser conhecido antecipadamente e deve fazer parte da entrada do algoritmo. Caso contrário, se ele não for conhecido, será necessário um acesso sequencial inicial na tabela para determinar qual é esse valor.

Em resumo, o algoritmo CountingSort acessa sequencialmente os elementos da tabela de entrada e calcula o número de vezes que cada chave ocorre na tabela. Os resultados dessa contagem são armazenados num array auxiliar de contagem. Então esse array é alterado de modo que ele determine, para cada chave, a posição inicial dos elementos que tenham essa chave na tabela ordenada, que é armazenada em outro array auxiliar. Finalmente, o algoritmo acessa sequencialmente os elementos desse array ordenado, copiando cada um deles para sua posição ordenada na tabela de saída.

O algoritmo de ordenação CountingSort segue os passos descritos na **Figura 11–30**.

Algoritmo CountingSort

Entrada/Saída: Uma tabela indexada cujos elementos contêm chaves inteiras não negativas e a maior dessas chaves (k)

1. Crie um array de contagem com tamanho igual a k
2. Conte as ocorrências de cada chave e atualize os valores dos elementos do array de contagem
3. Acesse todos os elementos do array de contagem a partir do segundo elemento somando a cada um deles o valor de seu antecessor
4. Usando a tabela de entrada e o array de contagem, armazene a tabela ordenada num array auxiliar
5. Copie o conteúdo do array auxiliar para a tabela recebida como entrada

Figura 11–30: Algoritmo CountingSort

Se os elementos da tabela a ser ordenada forem *valores inteiros* (em vez de *registros contendo chaves inteiras*), o algoritmo pode ser substancialmente simplificado. Mais especificamente, os três últimos passos podem ser fundidos num único passo.

A **Figura 11–31** ilustra os quatro primeiros passos do algoritmo CountingSort.

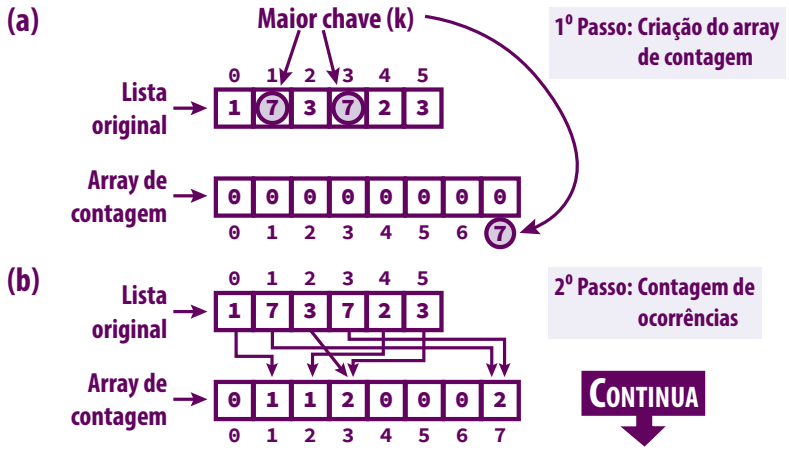

Figura 11–31: Exemplo de Ordenação Usando CountingSort 1

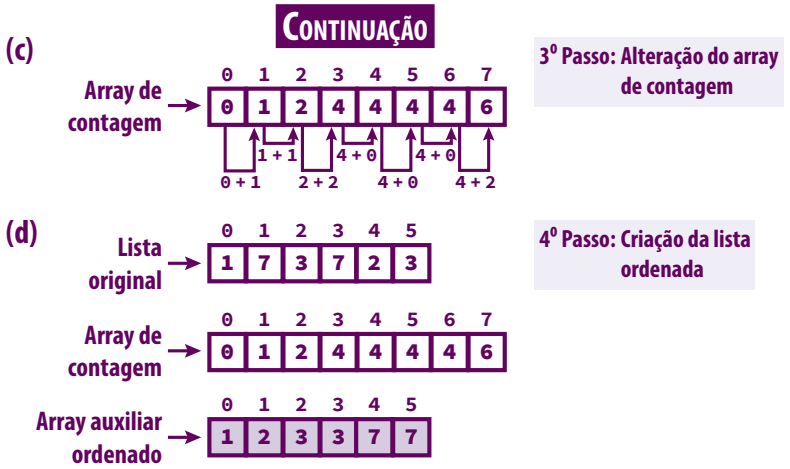

FIGURA 11–31 (CONT.): EXEMPLO DE ORDENAÇÃO USANDO COUNTINGSORT 1

Implementação

A função `CountingSort()` abaixo ordena uma tabela usando o algoritmo COUNTINGSORT.

```
void CountingSort(int tabela[], int nElem)
{
   int *resultado, /* Apontará para o array auxiliar que */
                   /* conterá os elementos ordenados     */
       *contagem,  /* Apontará para o array de contagem  */
       maiorValor, /* Maior valor da tabela */
       tamCont,    /* Tamanho do array de contagem */
       i;

      /* Encontra o maior valor da tabela */
   maiorValor = tabela[0];
   for (i = 1; i < nElem; ++i)
      if (tabela[i] > maiorValor)
         maiorValor = tabela[i];

            /*** Passo 1 ***/

   tamCont = maiorValor + 1; /* Calcula o tamanho do array de contagem */

      /* Aloca o array de contagem */
   ASSEGURA( contagem = calloc(tamCont, sizeof(int)),
             "Array de contagem nao foi alocado" );

            /*** Passo 2 ***/

      /* Efetua a contagem de elementos e armazena-a no array de contagem */
   for (i = 0; i < nElem; i++)
      ++contagem[tabela[i]];

            /*** Passo 3 ***/

      /* Altera o array de contagem de modo que contagem[i] passe */
      /* a conter as posições  desse elemento no array ordenado   */
   for (i = 1; i < tamCont; ++i)
      contagem[i] += contagem[i - 1];

            /*** Passo 4 ***/

      /* Aloca o array que conterá o resultado ordenado */
```

```
   ASSEGURA(resultado = calloc(nElem, sizeof(int)), "Array auxiliar nao foi alocado");

      /* Copia cada elemento da lista original para sua posição ordenada    */
      /* no array que conterá o resultado. Esse posicionamento é feito do   */
      /* último para o primeiro elemento para que a ordenação seja estável. */
   for (i = nElem - 1; i >= 0; --i) {
         /* Armazena um elemento da lista original em sua     */
         /* posição ordenada no array que conterá o resultado */
      resultado[contagem[tabela[i]] - 1] = tabela[i];
         /* Decrementa o valor deste elemento do array de contagem para que */
         /* o próximo elemento da lista que seja igual ao elemento corrente */
         /* seja colocado uma posição atrás                                 */
      --contagem[tabela[i]];
   }
            /*** Passo 5 ***/

      /* Copia o resultado para a tabela original */
   for (i = 0; i < nElem; ++i)
      tabela[i] = resultado[i];

      /* Libera o espaço ocupado pelos arrays auxiliares */
   free(contagem);
   free(resultado);
}
```

O modo como o **Passo 4** do algoritmo da **Figura 11–30** é realmente levado a efeito é esclarecido na **Figura 11–32**, que mostra como os dois últimos elementos da tabela original são armazenados no array auxiliar que armazena a tabela ordenada.

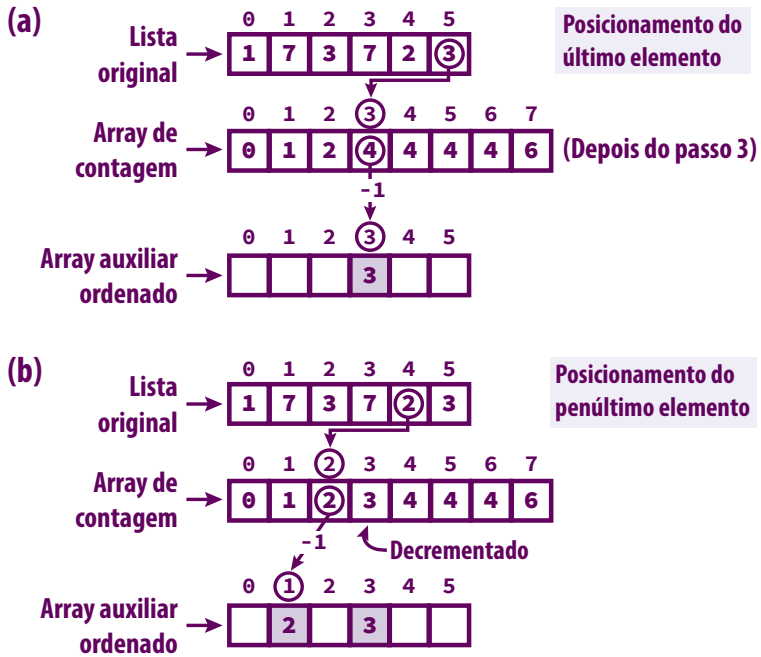

Figura 11–32: Exemplo de Ordenação Usando CountingSort 2

Análise

Teorema 11.20: Em qualquer caso, o custo temporal do algoritmo CountingSort é $\theta(n + k)$.

Prova: O custo temporal de cada passo do algoritmo CountingSort é apresentado na **Tabela 11–7**.

Passo	Custo temporal
1	$\theta(k)$
2	$\theta(n)$
3	$\theta(k)$
4	$\theta(n)$
5	$\theta(n)$

Tabela 11–7: Custo Temporal de Cada Passo do Algoritmo CountingSort

Levando em consideração o custo de cada passo do algoritmo CountingSort, conclui-se que o custo temporal total desse algoritmo é, nos três casos de análise, $\theta(n + k)$, em que n é o número de elementos da tabela de entrada e k é a maior chave dessa tabela. ■

Teorema 11.21: Em qualquer caso, o custo espacial do algoritmo CountingSort é $\theta(n + k)$.

Prova: O algoritmo CountingSort usa dois arrays auxiliares: o array de contagem tem tamanho $k + 1$ e o array que armazena temporariamente a tabela ordenada tem o mesmo tamanho n da tabela. ■

É fácil verificar que o algoritmo CountingSort é estável, pois os elementos são inseridos no array auxiliar que armazena a tabela ordenada na ordem em que eles se encontram na tabela original.

Ordenação por contagem, que funciona bem quando se tem um número limitado de chaves possíveis e há muitas duplicatas, apresenta algumas limitações. A mais significativa delas é que ele funciona apenas com chaves inteiras ou que possam ser associadas a valores inteiros. Isso ocorre porque ordenação por contagem faz uso de um array de contadores indexados por essas chaves inteiras para acompanhar quantas vezes cada uma delas ocorre.

O array de contagem usado em CountingSort tem tamanho k, o que não é um problema quando o intervalo de valores é pequeno, mas, para grandes intervalos, haverá grande desperdício de memória. Quer dizer, esse algoritmo é conveniente apenas para situações nas quais a largura desse intervalo de chaves não é significativamente maior do que o número de elementos. Por exemplo, suponha que se deseja ordenar a lista: *133*, *121*, *423*, *616*, *980*, *330*. Como o maior valor nessa lista é *980*, para ordenar uma lista com apenas 6 elementos, usa-se um array auxiliar com 980 elementos, a maioria dos quais é zero. Pior ainda, se o intervalo dos n elementos que se desejam ordenar variasse, por exemplo, de *0* a n^3, então simplesmente criar o array de contagem consumiria um tempo $\theta(n^3)$ e ordenação por contagem teria desempenho pior do que InsertionSort. Nesse último exemplo, o custo espacial seria $\theta(n^3)$, o que é significativamente maior do que o custo espacial de qualquer outro algoritmo de ordenação examinado até aqui.

Um resumo da avaliação do algoritmo CountingSort é apresentado na **Tabela 11–8**.

Custo temporal	❒ $\theta(n + k)$ nos três casos
Vantagens	❒ Muito rápido ❒ Estável
Desvantagens	❒ Depende do tipo das chaves ❒ Requer espaço adicional com custo $\theta(n + k)$
Indicações	❒ Quando as chaves são inteiras ❒ Quando o intervalo de chaves não é muito grande

Tabela 11–8: Análise Resumida de CountingSort

11.4.2 Ordenação com Coletores (BucketSort)

Descrição

O algoritmo de **ordenação com coletores** (abreviadamente, BucketSort) funciona distribuindo os elementos de uma tabela num certo número de coletores. Cada coletor é então ordenado individualmente usando um algoritmo de ordenação diferente ou invocando recursivamente o próprio algoritmo BucketSort. A **Figura 11–33** ilustra ordenação com coletores.

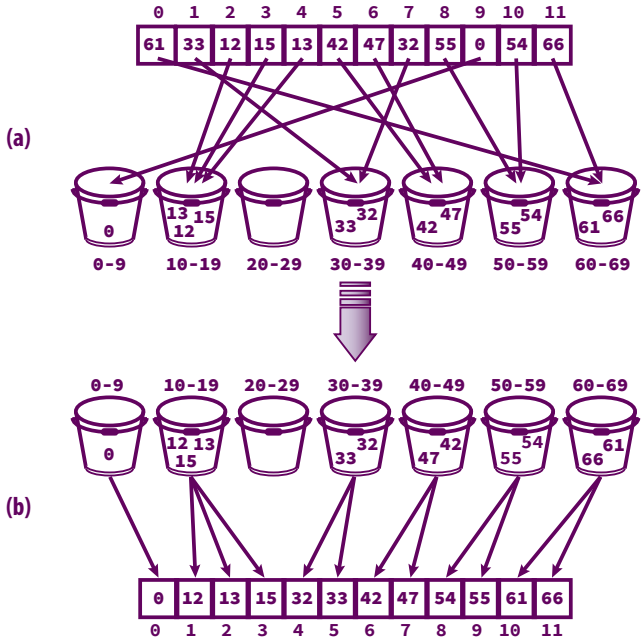

Figura 11–33: Ordenação com Coletores (BucketSort)

BucketSort é um algoritmo de **ordenação por distribuição** que pode ser implementado usando comparações. Portanto ele pode também ser considerado um algoritmo de ordenação por comparação. Precisamente, BucketSort segue os passos descritos na **Figura 11–34**.

Algoritmo BucketSort

Entrada/Saída: Uma tabela indexada cujos *n* elementos contêm chaves inteiras não negativas

1. Crie um array com *k* coletores (listas encadeadas) inicialmente vazios
2. Insira de maneira ordenada cada elemento da tabela de entrada em seu coletor (**fase de distribuição**)
3. Visite os coletores em ordem e coloque os elementos de volta na tabela original (**fase de coleta**)

Figura 11–34: Algoritmo BucketSort

BucketSort está para ordenação assim como dispersão está para busca. Isto é, ele usa os valores das chaves para ordenar os itens assim como dispersão usa os valores das chaves para determinar onde armazenar e encontrar itens.

Funções diferentes podem ser usadas para converter as chaves em índices do array de *n* coletores. Por exemplo, pode-se usar uma função para converter letras no intervalo *[A..Z]* em índices no intervalo *[0..25]*.

A variante mais comum de BucketSort opera sobre uma tabela contendo *n* chaves numéricas cujos valores estão entre *0* e algum valor máximo *M* e divide o intervalo de valores em *n* coletores, cada um dos quais com tamanho *M/n*. Se cada coletor for ordenado usando um método de ordenação razoavelmente eficiente (p. ex.,

InsertionSort), pode-se mostrar que a ordenação tem custo temporal linear. Entretanto o desempenho de BucketSort é degradado com o surgimento de agrupamentos, semelhantes àqueles que ocorrem com dispersão com endereçamento aberto (v. **Capítulo 8**). Ou seja, se muitas chaves forem concentradas num mesmo coletor, elas serão ordenadas lentamente.

Implementação

As seguintes definições de constantes e tipos são usadas pelo programa que implementa BucketSort:

```
#define N_COLETORES 8 /* Número de coletores */

   /* Número de bits mais significativos que serão obtidos de cada chave */
#define N_BITS_SIG  3

   /* Tipo de um coletor, que é uma lista simplesmente encadeada */
typedef struct rotNoLSE {
           unsigned         conteudo; /* Conteúdo efetivo do nó    */
           struct rotNoLSE *proximo;  /* Próximo elemento da lista */
       } tNoListaSE, *tBucket;
```

A função `BucketSort()` implementa o método de ordenação BucketSort.

```
void BucketSort(unsigned tabela[], unsigned nElem)
{
   tBucket  coletores[N_COLETORES]; /* Array de coletores */
   unsigned iColetor, i, j;

   for (i = 0; i < N_COLETORES; i++) /* Inicia os coletores */
      coletores[i] = NULL;

      /* Armazena as chaves em ordem nos coletores */
   for (i = 0; i < nElem; i++) {
      iColetor = MSBits(tabela[i], N_BITS_SIG); /* Obtém o índice do coletor */
      InsereEmColetor(coletores + iColetor, tabela[i]);/* Insere no devido coletor */
   }

      /* Coloca as chaves de volta na tabela */
   for (i = 0, j = 0; i < N_COLETORES; i++) {
      while (coletores[i]) {
         tabela[j++] = coletores[i]->conteudo;
         coletores[i] = coletores[i]->proximo;
      }

         /* Libera o espaço ocupado pelo coletor corrente */
      DestroiColetor(coletores + i);
   }
}
```

A função `MSBits()` chamada por `BucketSort()` é responsável pela fase de distribuição do algoritmo BucketSort (i.e., ela é responsável pela distribuição de chaves nos coletores). A função `MSBits()` assume que o maior valor de chave cabe no tipo **unsigned char** e retorna o valor inteiro correspondente aos bits extraídos. Os parâmetros dessa função e sua implementação são apresentados abaixo.

- `chave` (entrada) — a chave cujos bits serão extraídos
- `bits` (entrada) — o número de bits mais significativos que serão extraídos

```
unsigned MSBits(unsigned chave, unsigned bits)
{
   ASSEGURA(chave <= UCHAR_MAX, "A chave e' grande demais");
   ASSEGURA(bits <= CHAR_BIT, "Numero de bits e' grande demais");
```

```
      /* UCHAR_MAX mascara os primeiros bits de um valor do    */
      /* tipo char, de modo que se o valor da chave for maior  */
      /* do que um valor do tipo char, o restante é descartado */
   return (chave & UCHAR_MAX) >> (CHAR_BIT - bits);
}
```

Análise

Teorema 11.22: No melhor caso, o algoritmo **BucketSort** tem custo temporal $\theta(n + k)$, em que n é o número de chaves da tabela a ser ordenada e k é o número de coletores utilizados.

Prova: O melhor caso desse algoritmo ocorre quando as chaves são uniformemente distribuídas. Claramente, o custo temporal do **Passo 1** do algoritmo da **Figura 11–34** é $\theta(k)$. Levando em consideração a hipótese de as chaves serem uniformemente distribuídas, o número máximo de chaves que se espera encontrar num coletor é n/k. Assim, como o custo de acesso a um coletor é $\theta(1)$ (pois eles são acessados diretamente), o custo do **Passo 2** do algoritmo é $\theta(n/k)$. Usando um raciocínio similar, tem-se que o tempo despendido na remoção de um elemento de um coletor e subsequente armazenamento na tabela resultante(**Passo 3**) é dado por $c_0 \cdot n/k + c_1$, em que c_0 e c_1 são constantes que dependem de diversos fatores, tais como a implementação do algoritmo. Como há k coletores que precisam ser acessados sequencialmente, o tempo gasto no **Passo 3** é $k \cdot (c_0 \cdot n/k + c_1)$, que é $\theta(n + k)$. Logo o custo temporal do algoritmo **BucketSort** é $\theta(n + k)$. ■

Uma consequência **Teorema 11.22**, por exemplo, é que, se k é $\theta(n)$, pode-se ordenar a tabela com custo temporal $\theta(n)$.

Teorema 11.23: Em qualquer caso, o algoritmo **BucketSort** tem custo espacial $\theta(n + k)$.

Prova: O algoritmo usa um array de k coletores. Em conjunto, esses coletores devem ser capazes de armazenar os n registros da tabela a ser ordenada. ■

O algoritmo **BucketSort** é eficiente quando o número de coletores k é pequeno comparado com o número de elementos (n) da tabela, por exemplo, $k = \theta(n)$ ou $k = \theta(n \cdot log\ n)$. O desempenho de **BucketSort** se deteriora quando k cresce em comparação com n.

A descrição de **BucketSort** não garante estabilidade, mas isso não é inerente ao próprio método **BucketSort**, pois pode-se facilmente modificar essa descrição para torná-lo estável e ainda preservar seu custo temporal $\theta(n + k)$. Precisamente, se os coletores forem ordenados usando um algoritmo de ordenação que não é estável, **BucketSort** não será estável.

Um resumo da avaliação do algoritmo **BucketSort** é apresentado na **Tabela 11–9**.

Custo temporal	❒ $\theta(n + k)$ nos três casos, em que k é a largura do intervalo de chaves
Vantagens	❒ Muito rápido quando devidamente implementado
Desvantagens	❒ Usa espaço adicional $\theta(n + k)$ ❒ Restrição sobre as chaves
Indicações	❒ Tabelas cujas chaves satisfaçam os requisitos do algoritmo

Tabela 11–9: Análise Resumida de BucketSort

11.4.3 Ordenação por Base (RadixSort)

Descrição

O **algoritmo de ordenação por base** (abreviadamente, **RadixSort**) não é um algoritmo de ordenação por comparação. Esse algoritmo assume que as chaves a ser ordenadas podem usar uma **representação posicional**,

como uma chave inteira ou um string. Ou seja, diferentemente do que ocorre com outros métodos de ordenação vistos até aqui, **RadixSort** leva em consideração a estrutura das chaves. Por exemplo, uma chave inteira com quatro dígitos pode ser apresentada em sua notação posicional como:

$$milhar \times 10^3 + centena \times 10^2 + dezena \times 10^1 + unidade \times 10^0$$

A ideia que norteia **RadixSort** é dividir os valores a ser ordenados em tantos subgrupos quanto as possíveis alternativas para cada **posição** na chave. Por exemplo, se a chave for um número inteiro, cada posição (i.e., unidade, dezena, centena etc.) é um dígito com dez possibilidades que variam entre *0* e *9*. Se a chave for um string de letras e diferenciar maiúsculas e minúsculas não for importante, cada posição tem *26* possibilidades que variam de *a* a *z*. Esse número de possibilidades é chamado **base** (*radix*, em inglês).

O algoritmo **RadixSort** funciona de modo (mais ou menos) similar ao método que as pessoas usam para ordenar uma lista de nomes em ordem alfabética (nesse caso, a base é *26*): primeiro ordenam-se os nomes por sua letra inicial, depois, para cada valor de letra inicial, ordena-se pela segunda letra e assim por diante. Entretanto o algoritmo **RadixSort** faz o caminho inverso; i.e., ele começa com o dígito menos significativo (ou último caractere) e, para cada dígito, faz-se uma ordenação estável, colocando-se as chaves que possuem o mesmo valor do dígito sendo ordenado numa fila e, ao final, copia-se o resultado de volta na tabela original. Quando terminar a ordenação pelo último (mais significativo) dígito, a tabela original estará ordenada.

Esse algoritmo será ilustrado ordenando valores inteiros positivos de dois dígitos a seguir. Inicialmente, cria-se um array de filas indexado de *0* a *9*. Então todas as chaves com *0* na posição de unidade são enfileirados na fila que se encontra no índice *0* desse array. Todos os itens com *1* na posição de unidade são enfileirados na fila que se encontra no índice *1* desse array e assim por diante. Depois disso, coletam-se os itens das filas na ordem em que elas se encontram. Repete-se o processo usando a posição das dezenas e, depois, a posição das centenas. Quando os itens forem coletados pela última vez, eles estarão em ordem. Considere como exemplo a ordenação da seguinte tabela:

15	32	13	17	22	43

O primeiro passo do algoritmo **RadixSort** consiste em ordenar as chaves pela unidade, que resulta nas filas mostradas na **Figura 11–35**.

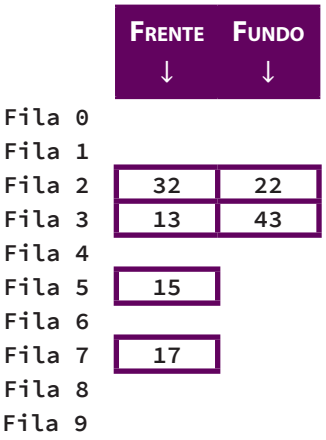

Figura 11–35: Exemplo de Ordenação Usando RadixSort 1

Após coletar as chaves de volta na tabela, o conteúdo alterado dessa tabela passa a ser:

32	22	13	43	15	17

O próximo passo consiste na coleta das chaves de acordo com suas ordenações pela dezena, o que resulta nas filas mostradas na **Figura 11–36**.

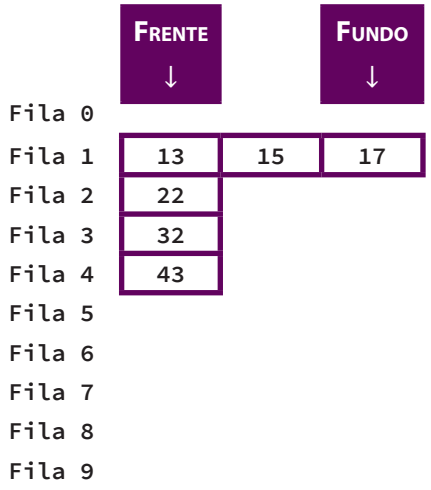

Figura 11–36: Exemplo de Ordenação Usando RadixSort 2

Após coletar as chaves de volta na tabela, seu conteúdo final ordenado passa a ser:

13	15	17	22	32	43

Para chaves inteiras, o uso de exponenciação e aritmética modular para obter o valor de cada dígito funciona bem. Diferentes tipos de dados requerem diferentes abordagens. Por exemplo, se uma chave é alfabética, deve-se considerar cada caractere e convertê-lo em um número entre *0* e *25*, se distinção entre maiúsculas e minúsculas não importa, ou entre *0* e *51*, em caso contrário. A implementação do algoritmo a ser usada depende do conjunto de caracteres sob consideração.

O algoritmo **RadixSort** segue os passos descritos na **Figura 11–37**.

Algoritmo RadixSort

Entrada/Saída: Uma tabela indexada contendo n chaves inteiras não negativas, cada uma das quais contém d dígitos

1. Crie um array contendo d filas capazes de armazenar os registros da tabela
2. Para i variando de *0* (que corresponde ao dígito menos significativo de cada chave) até $d - 2$ (que corresponde ao dígito mais significativo de cada chave), faça o seguinte:
   - **2.1** Insira cada registro da tabela na fila de índice d
   - **2.2** Esvazie as filas e armazena os registros de volta na tabela começando com a fila associada ao menor dígito e terminando com a fila associada ao maior dígito de cada chave

Figura 11–37: Algoritmo RadixSort

Implementação

As seguintes definições de constantes e tipos são usadas pela implementação de ordenação por base:

```
#define TAM_CHAVE_RADIX        5  /* Número de dígitos na chave */
#define BASE_RADIX            10  /* A base numérica utilizada  */
#define PRIMEIRO_DIGITO_RADIX '0' /* O primeiro dígito da base  */

typedef char tChaveRadix[TAM_CHAVE_RADIX + 1];

typedef struct rotNoLSE {
           tChaveRadix      chave;
           struct rotNoLSE *proximo;
        } tNoListaSE, *tListaSE;
```

A função `RadixSort()` implementa o método de ordenação RadixSort e tem como parâmetros:

- `tabela` (entrada/saída) — tabela que será ordenada
- `nChaves` (entrada) — número de chaves na tabela
- `tamChave` (entrada) — tamanho da chave (v. observações adiante)

```
void RadixSort(tChave tabela[], int nChaves, int tamChave)
{
   int       i, d, b;
   tListaSE *listas, pNo, p, q;
   char     *pDigito;

   listas = calloc(BASE_RADIX, sizeof(tListaSE));

   for (d = 0; d < tamChave; ++d) {
         /* Armazena as chaves em filas de acordo com o dígito */
         /* na posição d (contando de trás para a frente)      */
      for (i = 0; i < nChaves; ++i) {
            /* Para cada chave é alocado um nó que */
            /* será inserido em alguma das filas   */
         pNo = malloc(sizeof(tNoListaSE));
         strcpy(pNo->chave, tabela[i]);
         pNo->proximo = NULL;

            /* Faz pDigito apontar para o dígito da corrente coleta de chaves */
         pDigito = (char *)pNo->chave + tamChave - 1 - d;

            /* Determina a fila na qual a chave corrente será colocada */
         p = listas[*pDigito - PRIMEIRO_DIGITO_RADIX];
         q = NULL;

         while(p) { /* O fundo da fila é o final da lista encadeada */
            q = p;
            p = p->proximo;
         }

         if (!q) /* Esta fila está vazia */
            listas[*pDigito - PRIMEIRO_DIGITO_RADIX] = pNo;
         else /* A fila já contém pelo menos um elemento */
            q->proximo = pNo;
      }

      i = 0;

         /* Esvazia todas as filas e armazena as chaves na tabela */
         /* começando-se com a fila associada ao menor dígito e   */
         /* terminando-se com a fila associada ao maior dígito    */
      for (b = 0; b < BASE_RADIX; ++b) {
         p = listas[b];

         while (p) { /* Esvazia a fila corrente */
            strcpy(tabela[i], p->chave);
            ++i;
            q = p->proximo;
            free(p);
            p = q;
         } /* while */

            /* As filas foram todas esvaziadas. É necessário anular    */
            /* os ponteiros para os inícios das filas antes de iniciar */
            /* uma nova rodada do processo.                            */
```

```
            listas[b] = NULL;
         } /* for */
      } /* for */

      free(listas);
}
```

Na função `RadixSort()`, assume-se que os elementos do array a ser ordenado são strings numéricos em base *10* e de mesmo tamanho. Essa função não faz nenhum teste para verificar se isso realmente ocorre. O tamanho da chave não leva em consideração o caractere terminal de string (`'\0'`). Note ainda que não foi implementado o devido tratamento de exceção (p.ex., em alocação de memória) para não tornar o código ainda mais longo.

Análise

Teorema 11.24: Em qualquer caso, o custo temporal do algoritmo RadixSort é $\theta(n \cdot d)$.

Prova: Cada item da tabela é processado d vezes, em que d é o número de dígitos de cada chave. O processamento inclui extrair um valor da chave, inseri-lo numa fila, remover cada item da fila e copiá-lo de volta na tabela. Cada uma dessas operações tem custo temporal $\theta(1)$. Assim o custo temporal desse método de ordenação é $\theta(n \cdot d)$. ■

Teorema 11.25: Em qualquer caso, o custo espacial do algoritmo RadixSort é $\theta(n)$.

Prova: Os n elementos da tabela são armazenados em filas. Se as filas forem encadeadas, como na implementação apresentada acima, usa-se espaço adicional, pelo menos, para n ponteiros. ■

Se as filas forem implementadas em arrays, a quantidade de espaço gasto é proibitiva porque cada fila deve ter espaço para cada elemento.

RadixSort é menos flexível do que métodos de ordenação baseados em comparações, pois depende dos formatos das chaves. Assim é muito difícil codificá-lo com o propósito geral de lidar com todos os tipos de chaves. Por exemplo, se as chaves não forem do mesmo tamanho, é necessário um teste adicional para verificar, para cada chave, quando os dígitos são exauridos.

Um resumo da avaliação do algoritmo RadixSort é apresentado na **Tabela 11–10**.

Custo temporal	❒ $\theta(n \cdot d)$ nos três casos, sendo d o número de dígitos de cada chave
Vantagens	❒ Muito rápido quando apropriadamente codificado
Desvantagens	❒ Requer espaço adicional com custo $\theta(n)$ ❒ Desempenho depende de como as operações de comparação e seleção de dígitos são codificadas ❒ Depende do formato das chaves
Indicações	❒ Quando as chaves são inteiras com o mesmo número de dígitos ou strings com o mesmo número de caracteres

Tabela 11–10: Análise Resumida de RadixSort

11.5 Limite Inferior de Algoritmos Baseados em Comparações

O menor custo de um algoritmo de ordenação de qualquer natureza é $\theta(n)$, pois é impossível ordenar uma tabela com n elementos sem que eles sejam todos acessados. Contudo, se uma ordenação for baseada em comparação, esse limite inferior é um pouco maior como será mostrado nesta seção. Mais precisamente, se a operação

básica usada por um algoritmo de ordenação for a comparação de dois elementos, o melhor que se pode obter para uma ordenação baseada em comparações (de chaves) tem limite inferior $\Omega(n \cdot log\ n)$ no pior caso. Nesta seção será provado que não é possível existir um algoritmo de ordenação baseado em comparação de chaves que tenha custo temporal menor do que $\theta(n \cdot log\ n)$ no pior caso.

Suponha que se tenha uma lista $L = (x_0, x_1, ..., x_{n-1})$ que se deseja ordenar e assuma que todos os elementos de L são distintos. Cada comparação entre dois elementos x_i e x_j efetuada por um algoritmo de ordenação corresponde à pergunta: x_i *é menor do* x_j*?* Obviamente, só existem duas respostas possíveis para essa pergunta: *sim* ou *não*. Baseado no resultado dessa comparação, o algoritmo de ordenação pode executar uma operação de troca e, depois, executar outra comparação entre dois outros elementos de L, que, novamente, tem dois resultados possíveis e assim por diante. Consequentemente, pode-se representar um algoritmo de ordenação baseado em comparação como uma árvore de decisão A (v. **Capítulo 12** do **Volume 1**).

A **Figura 11–38** ilustra uma árvore de decisão associada à ordenação da sequência *xyz*. Nessa árvore, cada nó interno corresponde a uma comparação e as ligações de um nó para seus filhos correspondem às operações resultantes da resposta (*sim* ou *não*) de uma comparação. Essa árvore representa todas as possíveis sequências de comparações de chaves que o algoritmo de ordenação hipotético pode fazer, começando com a primeira comparação (associada à raiz) e terminando com a última comparação (associada ao pai de um nó-folha). A profundidade dessa árvore de decisão é o número de comparações no caminho mais longo desde a raiz até uma folha (que, na **Figura 11–38**, é 3) e representa exatamente o pior caso do algoritmo de ordenação. Qualquer permutação da lista de entrada deve aparecer como uma folha na árvore de decisão, pois, se uma tal permutação não for uma folha da árvore, quando o algoritmo for alimentado com essa mesma permutação, ele não terminará. Como, numa lista com n elementos, há $n!$ permutações possíveis, o número de folhas é, pelo menos, $n!$. Cada ordenação (ou **permutação**) possível dos elementos de L faz com que o algoritmo hipotético de ordenação execute uma série de comparações, caminhando desde a raiz até alguma folha de A.

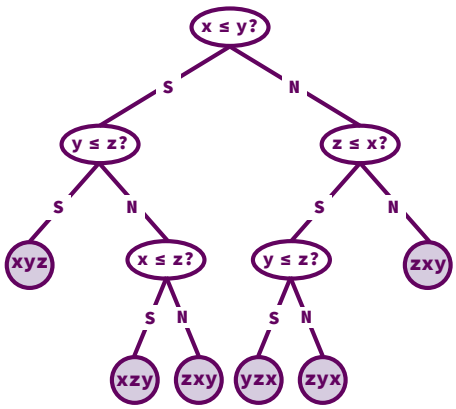

Figura 11–38: Árvore de Decisão de um Algoritmo Baseado em Comparações

Lema 11.7: Uma árvore binária de altura a possui, no máximo, 2^{a-1} folhas.

Prova: A prova será feita por indução sobre a.

Base da indução. Se $a = 1$, o lema é válido, pois, nesse caso, a árvore possui apenas um nó que é raiz e folha.

Hipótese indutiva. Suponha que o lema é válido para $a = k$; i.e., uma árvore de altura k tem no máximo 2^{k-1} folhas.

Passo indutivo. Suponha que se tenha uma árvore de altura k. De acordo com a hipótese indutiva, o número máximo de folhas dessa árvore é 2^{k-1}. Se forem acrescentados dois filhos para cada uma dessas

folhas, obtém-se uma árvore de altura $k + 1$, cujo número de máximo de filhos é $2 \cdot 2^{k-1} = 2^{k+1-1}$. Portanto, o lema vale para $k + 1$. ■

Lema 11.8: Uma árvore binária com n folhas deve ter altura mínima igual a $\lceil \log n \rceil + 1$.

Prova: De acordo com o **Lema 11.7**, tem-se que $n \leq 2^{a-1} \Rightarrow a \geq \lceil \log n \rceil + 1$. ■

Teorema 11.26: No pior caso, qualquer algoritmo de ordenação baseado em comparações deve efetuar, pelo menos, $\lceil \log (n!) \rceil + 1$ comparações para ordenar uma tabela com n elementos.

Prova: A árvore de decisão associada a um algoritmo possui, pelo menos, $n!$ folhas, sendo uma para cada permutação da tabela de entrada, como mostra a **Figura 11–39**. Para obter a permutação correta, deve-se percorrer um desses caminhos da raiz até uma folha. Portanto, no pior caso, o número de nós visitados é igual à altura mínima dessa árvore, que, de acordo com o **Lema 11.8**, é, pelo menos, $\lceil \log (n!) \rceil + 1$. Ademais, cada nó visitado corresponde a uma comparação. ■

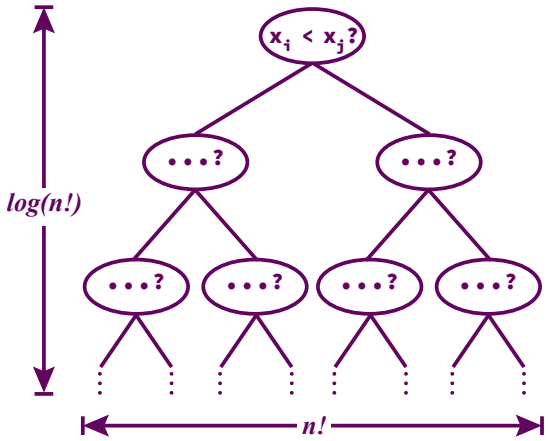

Figura 11–39: Tamanho da Árvore de Decisão de um Algoritmo Baseado em Comparações

Teorema 11.27: Qualquer algoritmo de ordenação baseado em comparação deve efetuar $\Omega(n \log n)$ comparações para ordenar n elementos no pior caso.

Prova: Usando o resultado do **Teorema 11.26**, resta mostrar que $\lceil \log (n!) \rceil + 1$ é $\Omega(n \log n)$, o que será feito a seguir.

$$
\begin{aligned}
\lceil \log (n!) \rceil &\geq \log (n!) \\
&= \log[n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1] \\
&= \log n + \log (n-1) + \log (n-2) + \ldots + \log 2 + \log 1 \\
&= \sum_{i=1}^{n} \log i \\
&= \sum_{i=1}^{n/2-1} \log i + \sum_{i=n/2}^{n} \log i \\
&\geq 0 + \sum_{i=n/2}^{n} \log \frac{n}{2} \\
&= \frac{n}{2} \cdot \log \frac{n}{2}
\end{aligned}
$$

Logo $\lceil \log (n!) \rceil$ é $\Omega(n \log n)$, o que prova o teorema. ■

O **Teorema 11.27** pode ser generalizado para mostrar que qualquer algoritmo de ordenação baseado em comparação deve ter custo temporal $\theta(n \cdot log\ n)$ em média, e não apenas no pior caso. A prova dessa afirmação é deixada como exercício para o leitor (v. questão **118** na **página 643**).

11.6 Algoritmos de Divisão e Conquista

Tanto MergeSort quanto QuickSort utilizam um paradigma de construção de algoritmos denominado **divisão e conquista**. Nesse paradigma, um algoritmo é dividido em três partes:

1. **Divisão**. Se o tamanho do problema é menor do que um certo limite previamente estabelecido, ele pode ser resolvido diretamente por outro algoritmo mais trivial e a solução obtida pode ser retornada. Caso contrário, o problema é dividido em duas ou mais porções menores.
2. **Recursão**. Nessa etapa, os subproblemas resultantes da divisão são resolvidos.
3. **Conquista**. As soluções dos subproblemas são combinadas de modo que resultem na solução final do problema original.

Usando o paradigma de divisão e conquista o algoritmo MergeSort, pode ser descrito como:

1. Divisão. Se a tabela contém menos de dois elementos encerre. Caso contrário, divida a tabela em duas partes, cada uma das quais contendo a metade dos elementos da tabela original.
2. Recursão. Recursivamente ordene as subtabelas resultantes da divisão.
3. Conquista. Intercale as duas subtabelas ordenadas.

A **Figura 11–40** ilustra as fases de divisão e conquista do algoritmo MergeSort.

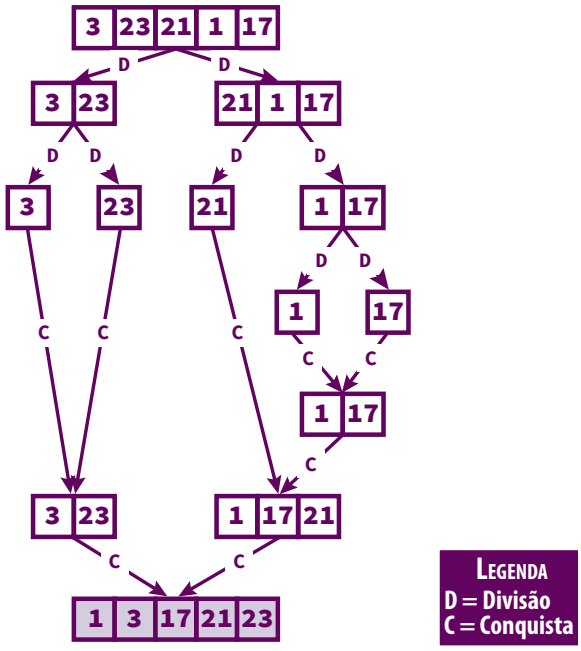

FIGURA 11–40: DIVISÃO E CONQUISTA NO ALGORITMO MERGESORT

11.7 Avaliações de Métodos de Ordenação

Neste capítulo, foram estudados métodos de ordenação, tais como InsertionSort e SelectionSort, que têm custo $\theta(n^2)$ no caso médio e no pior caso. Também foram estudados os métodos HeapSort, MergeSort e

QuickSort, que têm custos temporais $\theta(n \cdot log\ n)$. Finalmente, foram estudados algoritmos especiais de ordenação, a saber os métodos CountingSort, BucketSort e RadixSort, que são executados com custo linear, mas requerem que as chaves exibam características especiais.

Não há nenhum *melhor* algoritmo de ordenação dentre esses candidatos. O algoritmo de ordenação mais conveniente para uma aplicação particular depende de várias propriedades da aplicação que o usa. Para escolher o algoritmo de ordenação apropriado, é importante conhecer algumas propriedades dos dados de entrada. Mas, mesmo quando esse não é o caso, é possível oferecer alguma orientação com base nas propriedades dos algoritmos de ordenação que foram discutidos neste capítulo.

Esta seção avalia cada algoritmo de ordenação discutido neste capítulo usando uma abordagem prática que não tem a pretensão de ser uma lista definitiva de critérios para selecionar um algoritmo.

Para a maioria das aplicações de ordenação interna, InsertionSort, MergeSort, HeapSort ou QuickSort é o método de escolha. A decisão sobre qual método de ordenação usar depende evidentemente do tamanho da tabela a ser ordenada e do espaço disponível em memória. Ordenação com custo temporal $\theta(n^2)$ é apropriada apenas para ordenação de tabelas muito pequenas.

Uma consideração na escolha do algoritmo de ordenação mais adequado é o estado inicial de ordenação dos dados a ser ordenados (v. **página 583**). Se eles já estiverem ordenados (ou quase ordenados), BubbleSort tem custo temporal $\theta(n)$, enquanto algumas versões de QuickSort têm custo temporal $\theta(n^2)$.

Quando um algoritmo será usado poucas vezes, é preferível que ele seja fácil de entender, codificar e depurar. Por outro lado, se o algoritmo será usado muitas vezes ou com entradas muito grandes, a implementação de um algoritmo mais complicado pode compensar, se ela garantir execuções mais rápidas e que ocupem pouco espaço em memória.

11.7.1 Algoritmos com Custos Temporais Quadráticos

Embora os algoritmos mais simples tenham custos temporais $\theta(n^2)$, alguns podem ser substancialmente mais rápidos do que outros. Os algoritmos SelectionSort, InsertionSort e BubbleSort têm todos custo temporal quadrático no pior caso e no caso médio, e nenhum deles requer memória adicional. Assim seus custos temporais diferem por apenas um fator constante. BubbleSort e SelectionSort sempre efetuam exatamente o mesmo número de comparações de chaves, mas o número de comparações de chaves requerido por BubbleSort e SelectionSort é independente do estado dos dados de entrada. Por outro lado, o número de comparações requerido por InsertionSort é sensível ao estado de ordenação dos dados de entrada. Na pior das hipóteses, esse último algoritmo requer tantas comparações de chaves quanto os demais algoritmos com custo temporal $\theta(n^2)$. Porém, na melhor das hipóteses, ele requer menos comparações do que o número de elemento da tabela de entrada.

Se o algoritmo InsertionSort for bem implementado, seu custo temporal é $\theta(n + m)$, sendo m o número de inversões (i.e., o número de pares de elementos fora de ordem). Assim InsertionSort é um excelente algoritmo para ordenar tabelas pequenas, porque tabelas pequenas necessariamente têm poucas inversões. InsertionSort também é bem efetivo para ordenar tabelas que já estão quase ordenadas. Aqui, *quase* quer dizer que o número de inversões é pequeno. Mas o custo temporal $\theta(n^2)$ de InsertionSort faz com que esse algoritmo seja uma má escolha fora desses contextos especiais.

11.7.2 Algoritmos com Custos Temporais Linear Logarítmicos

O uso de espaço adicional por MergeSort torna-o menos atrativo do que HeapSort e QuickSort para tabelas que possam ser contidas inteiramente em memória principal. O algoritmo MergeSort é um bom algoritmo

para situações nas quais os dados a ser ordenados não cabem em memória principal e são armazenados em memória secundária (v. **Capítulo 12**).

QuickSort é uma excelente escolha como algoritmo de ordenação de propósito geral. Ele é implementado na função **qsort()** provida pela biblioteca padrão da linguagem C. Mas, se estabilidade for importante e houver espaço suficiente disponível, MergeSort pode ser uma opção melhor do que QuickSort, pois MergeSort tem custo temporal $\theta(n \cdot log\ n)$ garantido no pior caso. Então por que QuickSort é considerado o algoritmo mais usado na prática? A resposta a essa questão é simples: quando bem implementado, o custo de pior caso $\theta(n^2)$ desse algoritmo é muito raro de ocorrer na prática. Logo, quando o estado dos dados a ser ordenados é desconhecido e não se tem espaço sobrando em memória, o algoritmo de ordenação recomendado é QuickSort.

A análise assintótica de HeapSort sugere que, dentre os algoritmos com custo linear logarítmico, esse é o melhor método de ordenação. Mas, na realidade, esse não é bem o caso porque HeapSort é totalmente insensível ao estado de ordenação da tabela que ele recebe como entrada. Por exemplo, quando essa tabela já está ordenada, a primeira providência desse algoritmo é desordená-la para transformá-la em heap. Nessa mesma situação, o algoritmo QuickSort usando mediana de três efetua um número ínfimo de trocas de posições de elementos. Além disso, QuickSort exibe melhor localidade de referência (v. **Seção 1.5**) do que HeapSort.

11.7.3 Algoritmos com Custos Temporais Lineares

Se uma aplicação requer ordenar elementos com chaves inteiras pequenas, CountingSort, BucketSort ou RadixSort é uma excelente escolha, porque esses algoritmos funcionam com custo temporal linear. Assim, se todos os requisitos para uso desses algoritmos forem atendidos, eles devem funcionar mais rápido do que QuickSort, MergeSort ou HeapSort. Em particular, CountingSort é recomendável em situações nas quais o valor máximo de chave é significativamente menor do que o número de chaves. Strings também podem ser ordenados com custo temporal linear usando BucketSort ou RadixSort, mas esses métodos de ordenação apresentam custo espacial linear. Os algoritmos CountingSort, BucketSort e RadixSort usam espaço adicional e, dentre eles, o pior algoritmo nesse quesito é CountingSort que usa dois arrays adicionais.

11.7.4 Resumo das Avaliações

A **Tabela 11–11** compara os algoritmos de ordenação discutidos nesse capítulo em termos de custos temporais. Nessa tabela, n é o número de chaves e, no caso de CountingSort e BucketSort, k é a maior chave. No caso de RadixSort, d é o número de dígitos de cada chave.

Método	Melhor Caso	Caso Médio	Pior Caso
BubbleSort	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$
SelectionSort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$
InsertionSort	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$
QuickSort	$\theta(n \cdot log\ n)$	$\theta(n \cdot log\ n)$	$\theta(n^2)$
MergeSort	$\theta(n \cdot log\ n)$	$\theta(n \cdot log\ n)$	$\theta(n \cdot log\ n)$
HeapSort	$\theta(n \cdot log\ n)$	$\theta(n \cdot log\ n)$	$\theta(n \cdot log\ n)$
CountingSort	$\theta(n + k)$	$\theta(n + k)$	$\theta(n + k)$
BucketSort	$\theta(n + k)$	$\theta(n + k)$	$\theta(n + k)$
RadixSort	$\theta(n \cdot d)$	$\theta(n \cdot d)$	$\theta(n \cdot d)$

Tabela 11–11: Custos Temporais de Métodos de Ordenação

A **Tabela 11–12** compara os algoritmos de ordenação discutido nesse capítulo em termos de estabilidade e uso de espaço adicional. A última coluna dessa tabela apresenta o número da página na qual se inicia a descrição do respectivo método.

Método	Estável?	In Loco?	Espaço Adicional	Referência
BubbleSort	Sim	Sim	$\theta(1)$	página 584
SelectionSort	Não	Sim	$\theta(1)$	página 586
InsertionSort	Sim	Sim	$\theta(1)$	página 589
QuickSort	Não	Não	❒ $\theta(\log n)$ (melhor caso) ❒ $\theta(n)$ (pior caso)	página 592
MergeSort	Sim	Não	$\theta(n)$	página 599
HeapSort	Não	Sim	$\theta(1)$	página 605
CountingSort	Sim	Não	$\theta(n + k)$	página 611
BucketSort	Sim	Não	$\theta(n + k)$	página 615
RadixSort	Sim	Não	$\theta(n)$	página 617

Tabela 11–12: Estabilidade e Custos Espaciais de Métodos de Ordenação

11.7.5 Avaliação Experimental

Antes de usar um método de ordenação que o programador julga ser o melhor para uma determinada situação, é recomendável que ele conduza alguns experimentos para certificar-se de que sua hipótese é, de fato, válida. Um programa que usa a função **main**() exibida a seguir pode ser usado para testar vários métodos de ordenação.

```
int main(void)
{
   int   nChaves, maxRegs;
   FILE *stream;

   printf("\n>>> Numero maximo de registros (0 = todos): ");
   maxRegs = LeInteiro();

      /* Cria o arquivo binário */
   nChaves = CriaArquivoApenasCEPs( NOME_ARQUIVO_TEXTO, NOME_ARQUIVO_BIN, maxRegs );

      /* Verifica se ocorreu erro na criação do arquivo binário */
   ASSEGURA(!maxRegs || maxRegs == nChaves, "Ocorreu erro na criacao do arquivo");

      /* Apresenta o resultado da conversão */
   printf( "\n   >>> Armazenados %d registros no arquivo "
           "\"%s\" <<<\n", nChaves, NOME_ARQUIVO_BIN );

   stream = AbreArquivo(NOME_ARQUIVO_BIN, "rb"); /* Tenta abrir o arquivo binário */

   printf("\n      ***** Tamanho da tabela: %d *****\n", nChaves);

   TestaMetodo(stream, nChaves, BubbleSort, "BubbleSort");
   TestaMetodo(stream, nChaves, InsertionSort, "InsertionSort");
   TestaMetodo(stream, nChaves, SelectionSort, "SelectionSort");
   TestaMetodo(stream, nChaves, QuickSort1, "QuickSort Basico");
   TestaMetodo(stream, nChaves, QuickSort2, "QuickSort Mediana");
   TestaMetodo(stream, nChaves, QuickSort3,"QuickSort Aleatorio");
   TestaMetodo(stream, nChaves, MergeSort, "MergeSort");
```

```
   TestaMetodo(stream, nChaves, HeapSort, "HeapSort");
   putchar('\n');
   fclose(stream);
   return 0;
}
```

Essa função **main()** solicita que o usuário introduza o número de chaves com o qual o ele deseja testar os métodos de ordenação e, então, chama a função `CriaArquivoApenasCEPs()` para criar um arquivo binário contendo apenas chaves inteiras. Essas chaves são obtidas convertendo-se em números inteiros os campos `CEP` do arquivo do `CEPs.bin` (v. **Apêndice A**). Em seguida, a função **main()** chama `TestaMetodo()` para testar cada método de ordenação de interesse para o experimento. Essa última função, que será apresentada a seguir, tem como parâmetros:

- `stream` (entrada) — stream associado ao arquivo que contém as chaves que serão ordenadas
- `nChaves` (entrada) — número de elementos da tabela
- `ordena` (entrada) — ponteiro para a função de ordenação
- `nome` (entrada) — nome do método de ordenação

```
void TestaMetodo( FILE *stream, int nChaves, tFOrdena ordena, const char *nome )
{
   int  *tabela; /* Ponteiro para a tabela utilizada */
   tabela = CriaTabelaApenasCEPs(stream, nChaves);
   ExibeResultadoDeTeste(tabela, nChaves, ordena, nome);
   ExibeResultadoDeTeste(tabela, nChaves, ordena, nome);
   InverteTabela(tabela, nChaves, ComparaIntsInv);
   ExibeResultadoDeTeste(tabela, nChaves, ordena, nome);
   free(tabela);
}
```

O tipo do terceiro parâmetro da função `TestaMetodo()` um tipo de ponteiro para função de ordenação definido como:

```
typedef void (*tFOrdena) (int *, int);
```

Note que a função `TestaMetodo()` chama `ExibeResultadoDeTeste()` três vezes. Na primeira dessas chamadas, espera-se que a tabela esteja desordenada, enquanto, na segunda vez, a tabela deverá estar ordenada. Antes da terceira dessas chamadas, a função `InverteTabela()` é chamada para ordenar a tabela em ordem inversa. Essa última função simplesmente usa a função **qsort()** da biblioteca padrão de C para executar sua tarefa de ordenação e é relativamente trivial. A função `ComparaIntsInv()` cujo nome é passado como parâmetro de `InverteTabela()` semelhante à função `ComparaInts()` apresentada na **Seção 10.2.4**, porém elas comparam inteiros em ordens diferentes.

A função `CriaTabelaApenasCEPs()` lê números inteiros estocados num arquivo binário e armazena-os num array alocado dinamicamente. Essa função retorna o endereço da tabela criada e seus parâmetros são:

- `stream` (entrada) — stream associado ao arquivo
- `nEle` (entrada) — número de elementos da tabela

```
int *CriaTabelaApenasCEPs(FILE *stream, int nEle)
{
   int *tabela; /* Ponteiro para um array que armazena a tabela */
   int  i = 0, umaChave;
      /* Tenta alocar espaço para a tabela */
```

```
   tabela = calloc(nEle, sizeof(int));
   ASSEGURA( tabela, "Tabela nao foi alocada" );

   rewind(stream); /* Garante leitura a partir do início do arquivo */

      /* Lê cada número inteiro do arquivo e armazena-o na tabela */
   while (1) {
      fread(&umaChave, sizeof(umaChave), 1, stream);

      ASSEGURA(!ferror(stream), "Erro de leitura");

      if (feof(stream))
         break;

         /* Testa se o número de elementos da tabela é  */
         /* menor do que o número de valores no arquivo */
      ASSEGURA(i < nEle, "Tabela subdimensionada");

      tabela[i++] = umaChave;
   }
   return tabela;
}
```

A função `ExibeResultadoDeTeste()` chamada por `TestaMetodo()` apresenta na tela o resultado de um teste de método de ordenação e seus parâmetros são:

- `tabela` (entrada) — tabela que será ordenada
- `n` (entrada) — número de elementos da tabela
- `ordena` (entrada) — ponteiro para a função de ordenação
- `nome` (entrada) — nome do método de ordenação

```
void ExibeResultadoDeTeste(int *tabela, int n, tFOrdena ordena, const char *nome)
{
   printf("\n>>>>>>>>>>>>>>>>>>>> %s <<<<<<<<<<<<<<<<<<<<", nome);

   printf("\n*** Antes da Ordenacao: ");

   if (TabelaEstaOrdenada(tabela, n))
      printf("Tabela ordenada ***\n");
   else if (TabelaEstaEmOrdemInversa(tabela, n))
      printf("Tabela em ordem inversa ***\n");
   else
      printf("Tabela desordenada ***\n");

   printf("*** Ordenando a tabela...");
   MedidaDeTempo();

   ordena(tabela, n);

   printf(" fim da ordenacao ***");
   MedidaDeTempo();

   if (!TabelaEstaOrdenada(tabela, n))
      printf("\n\n\a>>> A tabela NAO foi ordenada <<<\n");
}
```

A função `TabelaEstaOrdenada()` chamada por `TestaMetodo()` testa se uma tabela está ordenada em ordem crescente e tem como parâmetros:

- `tabela` (entrada/saída) — tabela que será a testada
- `nElementos` (entrada) — número de elementos na tabela

Essa função retorna **1**, se a tabela estiver ordenada ou **0**, em caso contrário.

```
int TabelaEstaOrdenada(int tabela[], int nElementos)
{
   int i;

   for (i = 0; i < nElementos - 1; ++i) {
      if (tabela[i] > tabela[i + 1]) {
         return 0; /* Encontrado um par fora de ordem */
      }
   }

   return 1; /* Se ainda não houve retorno, a tabela está ordenada */
}
```

A função **TabelaEstaEmOrdemInversa()** chamada por **ExibeResultadoDeTeste()** testa se uma tabela está ordenada em ordem decrescente e é similar à função **TabelaEstaOrdenada()**.

A seguir um trecho de uma sessão de execução do programa:

```
      ***** Tamanho da tabela: 40000 *****

>>>>>>>>>>>>>>>>>>>> BubbleSort <<<<<<<<<<<<<<<<<<<<
*** Antes da Ordenacao: Tabela desordenada ***
*** Ordenando a tabela... fim da ordenacao ***
*** Tempo gasto na operacao:  4.07600 segundos ***
>>> Numero maximo de registros (0 = todos): 40000

   >>> Armazenados 40000 registros no arquivo "ApenasCEPs.bin" <<<

>>>>>>>>>>>>>>>>>>>> BubbleSort <<<<<<<<<<<<<<<<<<<<
*** Antes da Ordenacao: Tabela ordenada ***
*** Ordenando a tabela... fim da ordenacao ***
*** Tempo gasto na operacao:  0.00000 segundos ***

>>>>>>>>>>>>>>>>>>>> BubbleSort <<<<<<<<<<<<<<<<<<<<
*** Antes da Ordenacao: Tabela em ordem inversa ***
*** Ordenando a tabela... fim da ordenacao ***
*** Tempo gasto na operacao:  4.43900 segundos ***

>>>>>>>>>>>>>>>>>>>> InsertionSort <<<<<<<<<<<<<<<<<<<<
*** Antes da Ordenacao: Tabela desordenada ***
*** Ordenando a tabela... fim da ordenacao ***
*** Tempo gasto na operacao:  1.39500 segundos ***

[Trecho removido]
```

O programa descrito acima é usado com o arquivo de dados **CEPs.bin** (v. **Apêndice A**) e os resultados obtidos são aqueles mostrados na **Tabela 11–13**. Nessa tabela, **D** significa que a tabela estava desordenada antes de a ordenação acontecer. Por sua vez, **O** significa que a tabela já estava ordenada e **I** significa que a tabela estava inversamente ordenada[7].

[7] O programa foi compilado com clang versão 7.3.0 e executado num computador iMac com processador Intel Core i5 de 2.5 MHz e 4 GB de memória usando o sistema Mac OS X versão 10.11.5. O tamanho padrão da pilha de execução usado pelo sistema precisou ser aumentado para que a implementação básica de QuickSort pudesse ser executada sem ocorrência de *stack overflow*.

Número de Registros ☛		100.000	200.000
Método	Status	Tempo (s)	Tempo (s)
BubbleSort	D	29,71501	96,67553
	O	0,00029	0,00054
	I	34,25861	137,0634
InsertionSort	D	7,27757	33,82525
	O	0,00046	0,00089
	I	15,12504	60,56307
SelectionSort	D	12,38761	49,53647
	O	12,19441	48,84231
	I	12,28336	49,12317
QuickSort elementar	D	0,01435	0,03071
	O	10,85462	43,35838
	I	11,18969	44,74479
QuickSort com mediana de três	D	0,01109	0,02522
	O	0,00387	0,00818
	I	0,00722	0,01383
QuickSort aleatório	D	0,01403	0,03226
	O	0,00966	0,01875
	I	0,01051	0,01943
MergeSort	D	0,01648	0,03364
	O	0,01197	0,02507
	I	0,01279	0,02529
HeapSort	D	0,02310	0,04662
	O	0,02089	0,04188
	I	0,01855	0,04057

Tabela 11–13: Avaliação Experimental de Métodos de Ordenação

Os valores reais apresentados na **Tabela 11–13** não são tão importantes, pois eles refletem o desempenho de uma plataforma específica na qual os testes foram executados. Em vez disso, deve-se dar atenção ao desempenho relativo dos algoritmos nos conjuntos de dados correspondentes.

11.8 Exemplos de Programação

11.8.1 Ordenação de Lista Simplesmente Encadeada

Preâmbulo: Quando se deseja ordenar uma lista encadeada, o fato de ela não permitir acesso direto a seus elementos faz com que alguns algoritmos de ordenação (p. ex., QUICKSORT) apresentem baixo desempenho, enquanto outros sejam totalmente impossíveis de usar (p. ex., HEAPSORT). De qualquer modo, muitos algoritmos (p. ex., BUBBLESORT e MERGESORT) são bem adaptáveis para ordenação de listas encadeadas.

Problema: Escreva uma função que ordena uma lista simplesmente encadeada cujo conteúdo efetivo de seus nós é do tipo **int**.

Solução: Uma maneira bem simples de ordenar uma lista encadeada é usando o algoritmo BUBBLESORT, que pode ser aplicado a uma lista simplesmente encadeada como mostra a **Figura 11–41**.

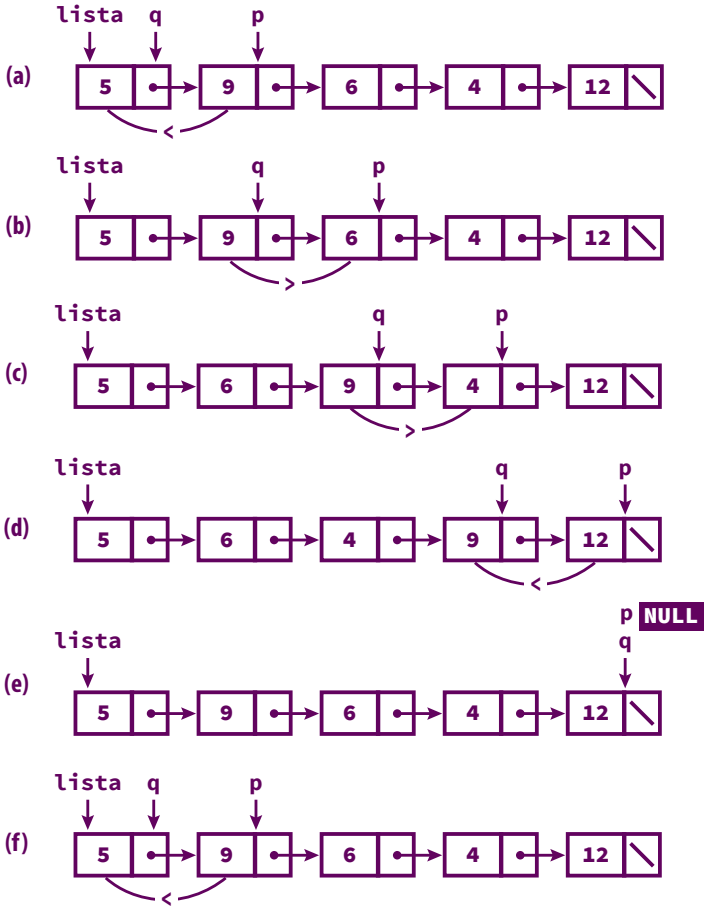

FIGURA 11–41: ORDENAÇÃO DE LISTA SIMPLESMENTE ENCADEADA

A função `OrdenaListaSE()` implementa essa abordagem. O tipo `tNoListaSE` usado nesta implementação é o mesmo usado na **Seção 11.4.2**.

```
void OrdenaListaSE(tNoListaSE *lista)
{
   tNoListaSE *p, *q; /* Ponteiros usados para visitar os nós da lista */
   int         ordenada = 0; /* Informará se a lista está ordenada */
```

```
      /* Se a lista estiver vazia ou tiver apenas um elemento, ela está ordenada */
   if (!lista || !lista->proximo)
      return;

      /* O laço encerra quando não houver troca de nós */
   while (!ordenada) {
      ordenada = 1; /* Supõe que a lista está ordenada */

      p = lista->proximo; /* p segue à frente de q */
      q = lista; /* q segue atrás de p */

         /* O laço a seguir encerra quando cada nó  */
         /* tiver sido comparado com seu antecessor */
      while (p) {
            /* Compara conteúdos dos nós apontados por q e p */
         if (q->conteudo > p->conteudo) {
            ordenada = 0; /* Nós estão fora de ordem*/

               /* Troca os conteúdos dos nós */
            TrocaGenerica(&q->conteudo, &p->conteudo, sizeof(p->conteudo));
         }

            /* Passa para os nós seguintes */
         q = p;
         p = p->proximo;
      }
   }
}
```

11.8.2 Ordenação de Ponteiros

Preâmbulo: A ordenação de registros que ocupam muito espaço em memória usando algum tipo de ordenação que troca suas posições pode consumir muito tempo apenas para mover muitos bytes de um lugar para outro cada vez que se faz uma troca de dois elementos. Pode-se reduzir esse tempo de movimentação criando-se um array de ponteiros para os registros e, então, ordenando-se esses ponteiros, em vez de ordenando-se os registros para os quais os ponteiros apontam. Depois dessa ordenação, os registros ainda ocupam os mesmos respectivos espaços em memória, mas eles podem ser acessados de modo ordenado por meio do array ordenado de ponteiros. A **Figura 11–42** ilustra essa abordagem, que é conhecida como **ordenação de ponteiros**.

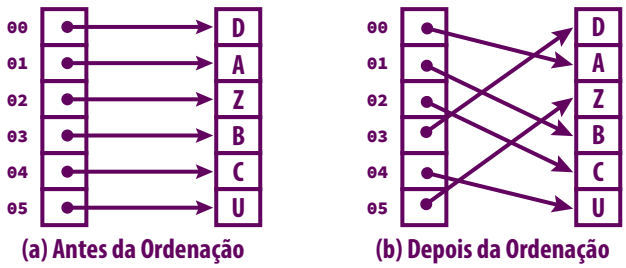

Figura 11–42: Ordenação de Ponteiros

Problema: Escreva um programa que demonstre a técnica de ordenação de ponteiros.

Solução: O primeiro passo dessa técnica é a associação de ponteiros aos elementos da tabela que se deseja acessar de modo ordenado, como faz a função `AssociaPonteiros()` a seguir.

```
tAluno **AssociaPonteiros(tAluno tabela[], int nElem)
{
   tAluno **ponteiros;
```

```
   int      i;

      /* Aloca o array de ponteiros e verifica se a alocação foi bem sucedida */
   ponteiros = malloc(nElem*sizeof(tAluno *));
   ASSEGURA(ponteiros, "Impossivel alocar array de ponteiros");

      /* Faz cada ponteiro apontar para o elemento correspondente da tabela */
   for (i = 0; i < nElem; ++i)
      ponteiros[i] = tabela + i;

   return ponteiros;
}
```

O tipo `tAluno` usado pela função `AssociaPonteiros()` é definido no **Apêndice A**.

O segundo passo da ordenação de ponteiros consiste na ordenação do array que contém os ponteiros obtidos no passo anterior, que é o que faz função `OrdenaPonteiros()` apresentada adiante. Os parâmetros dessa função são:

- `ptr` (entrada/saída) — array de ponteiros associado à tabela
- `nElem` (entrada) — número de elementos do array
- `F` (entrada) - endereço da função que compara dois elementos da tabela. Essa função deve ter a mesma especificação de retorno da função **strcmp()** da biblioteca padrão de C.

```
void OrdenaPonteiros(tAluno *ptr[], int nElem, int (*F)(void *e1, void *e2))
{
   int i, ordenado = 0;

   while (!ordenado){
      ordenado = 1; /* Supõe que a tabela está ordenada */

      for (i = 0; i < nElem - 1; i++)
            /* Compara elementos adjacentes */
         if (F(ptr[i], ptr[i + 1]) > 0) {
            ordenado = 0; /* Pelo menos um par de elementos está fora de ordem */

               /* Troca ponteiros para os elementos adjacentes */
            TrocaGenerica(ptr + i, ptr + i + 1, sizeof(ptr[0]));
         }

         --nElem; /* Mais um elementos ficou em seu devido lugar */
   }
}
```

Observação: A função `OrdenaPonteiros()` usa o algoritmo **BubbleSort** para facilitar o entendimento, mas alguns outros métodos de ordenação, como **MergeSort** (v. **Seção 11.3.2**), poderiam ter sido usados.

Para completar o programa, é necessária uma função para exibição dos registros da tabela usando o array de ponteiros ordenados. Essa função é relativamente trivial e não será exibida aqui. O programa completo pode ser encontrado no site dedicado a este livro na internet.

11.8.3 O Problema da Bandeira Holandesa

Preâmbulo: O **problema da bandeira holandesa** foi proposto por Edsger Dijkstra e, originalmente, consiste no seguinte: dada uma coleção de bolas de três cores distintas[8], deve-se arranjá-las de modo que as bolas que possuem a mesma cor permaneçam juntas. Esse problema pode ser enunciado de diversas maneiras distintas. Por exemplo, em programação, um problema análogo consiste em

[8] A bandeira holandesa apresenta três cores distintas: azul, branco e vermelho, mas, de fato, as cores das bolas não têm nenhuma importância na definição do problema.

separar em três partições um array que contém apenas três valores distintos (p. ex., *0*, *1* e *2*) de tal modo que cada partição do array contenha valores iguais. Para tornar ainda mais palpável esse último problema, considere um array contendo os elementos *0*, *1*, *1*, *2*, *1*, *0*, *2*. Esse array atende ao pressuposto do problema e, após sua solução, ele se apresentará como: *0*, *0*, *1*, *1*, *1*, *2*, *2*.

Problema: Suponha que se tenha um array de elementos do tipo **int** contendo apenas os valores *0*, *1* e *2*. Escreva uma função em C que divide o array em três partições: uma contendo apenas elementos com valores iguais a *0*, outra com elementos com valores iguais a *1* e a última com valores iguais a *2*.

Solução: Inicialmente, o array é dividido em três partições:

- Entre os índices *0* e *inf* – *1* encontram-se os valores iguais a *0*
- Entre os índices *inf* e *meio* encontram-se os valores iguais a *1*
- Entre os índices *sup* + *1* e *n* – *1* encontram-se os valores iguais a *2*
- Entre os índices *meio* e *sup* encontram-se os elementos que ainda não foram classificados

A **Figura 11–43** ilustra essas partições durante um certo ponto do processamento.

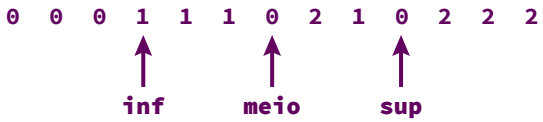

FIGURA 11–43: PROBLEMA DA BANDEIRA HOLANDESA

A solução para o problema estará completa quando todos os elementos que se encontram entre *meio* e *sup* forem movidos para suas respectivas partições, como mostra a implementação da função `SeparaEm3Particoes()` a seguir.

```
void SeparaEm3Particoes(int lista[], int n)
{
   int inf = 0,
       sup = n - 1,
       meio = 0;

      /* Coloca cada elemento em sua devida partição */
   while (meio <= sup) {
      switch (lista[meio]) {
         case 0:
            TrocaGenerica(lista + inf, lista + meio, sizeof(lista[0]));
            ++inf;
            ++meio;
            break;
         case 1:
            meio++;
            break;
         case 2:
            TrocaGenerica(lista + meio, lista + sup, sizeof(lista[0]));
            --sup;
            break;
         default:
            printf("\nValor de array diferente de 0, 1 ou 2\n");
            exit(1);
      }
   }
}
```

11.8.4 Encontrando Chaves Duplicadas Eficientemente

Problema: Suponha que se tenha uma lista indexada cujos elementos são do tipo **int**. (a) Escreva uma função em C que verifica se há algum elemento duplicado nessa lista, supondo que ela não está ordenada. (b) Escreva uma função semelhante àquela do item (a) supondo que a lista está ordenada. (c) Qual é o custo temporal no pior caso de cada uma dessas funções?

Solução de (a): A função `ContemDuplicatasListaIdx()` verifica se uma lista indexada contém pelo menos um elemento duplicado.

```
int ContemDuplicatasListaIdx(int lista[], int n)
{
   int i, j;

      /* Compara elementos */
   for (i = 0; i < n; ++i)
      for (j = i + 1; j < n; ++j)
         if (lista[i] == lista[j])
            return i; /* Encontrada uma chave duplicada */

   return -1; /* Não foi encontrada nenhuma duplicata */
}
```

Solução de (b): A função `ContemDuplicatasListaIdxOrd()` verifica se uma lista indexada ordenada contém pelo menos um elemento duplicado.

```
int ContemDuplicatasListaIdxOrd(int lista[], int n)
{
   int i;

     /* Compara elementos adjacentes */
   for (i = 0; i < n - 1; ++i)
      if (lista[i] == lista[i + 1])
         return i; /* Encontrada uma chave duplicada */

   return -1; /* Não foi encontrada nenhuma duplicata */
}
```

Solução de (c): É fácil verificar que a função `ContemDuplicatasListaIdx()` tem custo temporal $\theta(n^2)$ e que a função `ContemDuplicatasListaIdxOrd()` tem custo temporal $\theta(n)$ no pior caso.

11.8.5 Lista Bitônica

Preâmbulo: Uma lista indexada L com n elementos é **bitônica** se existe um índice i, com $0 < i < n - 1$, tal que $L[0], ..., L[i]$ é uma sequência crescente e $L[i + 1], ..., L[n - 1]$ é uma sequência decrescente. Nesse caso, o elemento que se encontra no índice i é denominado **ponto bitônico**. Por exemplo, a lista contendo os elementos *5*, *8*, *9*, *12*, *6*, *4*, *1* e *0* é uma lista bitônica e o ponto bitônico é *12*, pois os elementos *5*, *8*, *9* e *12* formam uma sequência crescente, enquanto os elementos *6*, *4*, *1* e *0* formam uma sequência decrescente.

Problema: Escreva uma função em C que encontre o índice do ponto bitônico de uma lista bitônica.

Solução: A função `PontoBitonico()` abaixo implementa aquilo que foi solicitado.

```
int PontoBitonico(int lista[], int n)
{
   int inf = 0,
       sup = n - 1,
       meio;
```

```
   while (inf < sup) {
         /* Se o array só possui um elemento,  ele é o ponto bitônico */
      if (inf == sup)
         return inf;

         /* Se o array possui dois elementos, o ponto bitônico é o maior deles */
      if (inf == sup - 1)
         return lista[inf] > lista[sup] ? inf : sup;

      /* O array possui pelo menos três elementos */

      meio = inf + (sup - inf)/2;

      if (lista[meio - 1] < lista[meio] && lista[meio] > lista[meio + 1])
         return meio;
      else if (lista[meio - 1] < lista[meio] && lista[meio] < lista[meio + 1])
         inf = meio + 1;
      else if (lista[meio - 1] > lista[meio] && lista[meio] > lista[meio + 1])
         sup = meio - 1;
      else
         return -1; /* A lista não é bitônica */
   }

   return -1; /* A lista não é bitônica */
}
```

11.9 Exercícios de Revisão

Fundamentos de Ordenação (Seção 11.1)

1. Como métodos de ordenação podem ser classificados?
2. Por que são estudados tantos métodos de ordenação?
3. O que é chave de ordenação?
4. (a) O que é ordenação por comparação? (b) Quais são os métodos de ordenação descritos neste capítulo que não usam comparação?
5. (a) Como uma pessoa (normal) ordena uma lista de nomes alfabeticamente? (b) Como esse tipo de ordenação é classificado?
6. (a) O que é um algoritmo de ordenação estável? (b) Quando é desejável que um algoritmo de ordenação apresente essa propriedade?
7. Dentre os algoritmos de ordenação discutidos neste capítulo, quais deles são estáveis?
8. O que é uma ordenação in loco?
9. (a) O que é ordenação interna? (b) O que é ordenação externa?
10. (a) O que é estado de ordenação de uma tabela? (b) Quais são os possíveis estados de ordenação que uma tabela pode apresentar?
11. (a) O fato de uma tabela já estar ordenada constitui o melhor caso de qualquer algoritmo de ordenação? (b) Quando uma tabela está inversamente ordenada constitui o pior caso de qualquer algoritmo de ordenação? (c) O fato de uma tabela estar aleatoriamente ordenada constitui o caso médio de qualquer algoritmo de ordenação?
12. Descreva o conceito de inversão no contexto de ordenação.
13. Apresente três problemas cujas soluções sejam facilitadas com o uso de ordenação informando qual é a contribuição da ordenação na solução de cada problema.
14. (a) O que é ordenação por troca? (b) Quais são os métodos de ordenação descritos neste capítulo que se enquadram nessa classificação?

Ordenação com Custo Temporal Quadrático (Seção 11.2)

15. (a) Descreva o método de ordenação por borbulhamento (**BubbleSort**). (b) Por que esse método é assim denominado?

16. Suponha que a tabela abaixo seja ordenada utilizando o algoritmo **BubbleSort**. Apresente o conteúdo dessa tabela após a terceira iteração do laço externo do referido algoritmo.

0	1	2	3	4	5	6	7	8	9
55	8	13	28	23	9	25	11	70	19

17. Mostre que, no pior caso, o número de comparações de chaves efetuadas pelo algoritmo **BubbleSort** é $n^2 - 3n + 2$.

18. Considerando o método **BubbleSort**, preencha a seguinte tabela informando quando ocorre o melhor caso desse algoritmo e qual é seu custo temporal em termos de notação ó. Repita o mesmo procedimento para o caso médio e o pior caso desse algoritmo.

Método de Borbulhamento (BubbleSort)					
Melhor Caso		Caso Médio		Pior Caso	
Quando ocorre?	Custo?	Quando ocorre?	Custo?	Quando ocorre?	Custo?

19. (a) Em que situações o uso do método de ordenação **BubbleSort** é aceitável? (b) Em que situações o uso desse método de ordenação não é aceitável?

20. Qual é o custo espacial do método **BubbleSort**?

21. (a) Descreva o método de ordenação por inserção. (b) Por que esse método é assim denominado?

22. Suponha que o array abaixo seja ordenado utilizando o algoritmo **InsertionSort**. Apresente o conteúdo desse array após a terceira iteração do laço externo do referido algoritmo.

0	1	2	3	4	5	6	7	8	9
55	8	13	28	23	9	25	11	70	19

23. Considerando o método **InsertionSort**, preencha a seguinte tabela informando quando ocorre o melhor caso desse algoritmo e qual é seu custo temporal em termos de notação ó. Repita o mesmo procedimento para o caso médio e o pior caso desse algoritmo.

Método de Inserção (InsertionSort)					
Melhor Caso		Caso Médio		Pior Caso	
Quando ocorre?	Custo?	Quando ocorre?	Custo?	Quando ocorre?	Custo?

24. (a) Em que situações o uso do método **InsertionSort** é recomendável? (b) Em que situações o uso desse método de ordenação não é recomendável?

25. Qual é o custo espacial do método **InsertionSort**?

26. (a) O que é ordenação por seleção? (b) Quais são os métodos de ordenação descritos neste capítulo que se enquadram nessa classificação?

27. (a) Descreva o método de ordenação por seleção direta. (b) Por que esse método é assim denominado?

28. Suponha que o array abaixo seja ordenado utilizando o algoritmo **SelectionSort**. Apresente o conteúdo desse array após a terceira iteração do laço externo do referido algoritmo.

0	1	2	3	4	5	6	7	8	9
55	8	13	28	23	9	25	11	70	19

29. Considerando o método **SelectionSort**, preencha a seguinte tabela informando quando ocorre o melhor caso desse algoritmo e qual é seu custo temporal em termos de notação ó. Repita o mesmo procedimento para o caso médio e o pior caso desse algoritmo.

Método de Seleção Direta (SelectionSort)					
Melhor Caso		Caso Médio		Pior Caso	
Quando ocorre?	Custo?	Quando ocorre?	Custo?	Quando ocorre?	Custo?

30. Qual é o custo espacial do método ordenação de seleção direta?

31. (a) Explique por que o método **SelectionSort** descrito neste capítulo não é estável. (b) Que modificações são necessárias para tornar o método **SelectionSort** estável?

32. Qual é o custo temporal de **InsertionSort** quando todas as chaves da tabela a ser ordenada são iguais?

33. Mostre passo a passo como é efetuada a ordenação das chaves *9*, *8*, *7*, *6*, *5*, *4*, *3*, *2*, *1* usando **InsertionSort**.

34. Qual dos três métodos **SelectionSort**, **InsertionSort** ou **BubbleSort** é executado mais rapidamente para uma tabela com todas as chaves iguais?

35. Qual dos três métodos **SelectionSort**, **InsertionSort** ou **BubbleSort** é executado mais rapidamente para uma tabela cujas chaves estão em ordem invertida?

36. (a) Qual é o número mínimo de comparações necessárias para concluir que uma lista está ordenada? (b) Qual é o número mínimo de comparações necessárias para concluir que uma lista não está ordenada?

37. O que é necessário para tornar o algoritmo **InsertionSort** estável?

38. Qual é a principal vantagem do algoritmo **SelectionSort**?

39. Por que **SelectionSort** não é considerado um algoritmo adaptável?

40. Todos os algoritmos discutidos na **Seção 11.2** apresentam custo temporal $\theta(n^2)$ no caso médio. Isso significa que, considerando uma tabela aleatoriamente ordenada, todos eles apresentarão o mesmo desempenho? Explique seu raciocínio.

41. Qual é o número de trocas efetuadas pelo algoritmo **InsertionSort** no pior caso?

42. Qual é o número máximo de atribuições efetuadas pelo algoritmo **SelectionSort**?

43. Mostre que, no pior caso, o algoritmo **BubbleSort** efetua $(n - 2)^2$ comparações de chaves.

Ordenação com Custo Temporal Linear Logarítmico (Seção 11.3)

44. (a) Descreva o método de ordenação **QuickSort**. (b) Por que esse método é assim denominado?

45. (a) O que é pivô no método **QuickSort**? (b) Qual é a importância da escolha do pivô no desempenho do método **QuickSort**?

46. O algoritmo **QuickSort** é sensível ao estado inicial de ordenação da tabela de entrada?

47. Considerando o método de ordenação **QuickSort**, preencha a seguinte tabela informando quando ocorre o melhor caso desse algoritmo e qual é seu custo temporal em termos de notação ó. Repita o mesmo procedimento para o caso médio e o pior caso desse algoritmo.

Método QuickSort					
Melhor Caso		Caso Médio		Pior Caso	
Quando ocorre?	Custo?	Quando ocorre?	Custo?	Quando ocorre?	Custo?

48. (a) Em que situações o uso do método de ordenação **QuickSort** é recomendável? (b) Em que situações o uso do método de ordenação **QuickSort** não é recomendável?

49. Qual é o custo espacial do método de ordenação **QuickSort**?

50. (a) Em que consiste o método de ordenação de árvore binária de busca? (b) Quando esse método de ordenação é vantajoso? (c) Quando esse método de ordenação não é vantajoso? (d) Qual é o pior caso desse método? (e) Qual é o custo de pior caso desse método de ordenação? (f) Qual é o custo de melhor caso desse método? (g) Qual é o custo espacial desse método?

51. Descreva o método de ordenação por intercalação (**MergeSort**).

52. Suponha que o array abaixo seja ordenado utilizando o algoritmo **MergeSort**. Apresente o conteúdo desse array logo antes da etapa de intercalação do referido algoritmo.

0	1	2	3	4	5	6	7	8	9
55	8	13	28	23	9	25	11	70	19

53. Considerando o método **MergeSort**, preencha a seguinte tabela informando quando ocorre o melhor caso desse algoritmo e qual é seu custo temporal em termos de notação ó. Faça o mesmo para o caso médio e o pior caso desse algoritmo.

Ordenação por Intercalação (MergeSort)					
Melhor Caso		Caso Médio		Pior Caso	
Quando ocorre?	Custo?	Quando ocorre?	Custo?	Quando ocorre?	Custo?

54. Qual é o custo espacial do método de ordenação por intercalação?

55. (a) Em que situações o uso do método de ordenação por intercalação é recomendável? (b) Em que situações o uso desse método não é recomendável?

56. Descreva o método **HeapSort**.

57. (a) O método **QuickSort** é estável? (b) Se sua resposta for negativa, como ele pode se tornar estável?

58. (a) Quando o algoritmo **QuickSort** apresenta custo temporal $\theta(n^2)$? (b) Como esse custo pode ser amenizado?

59. (a) Descreva a técnica mediana de três usada com **QuickSort**. (b) Para que serve essa técnica?

60. Suponha que o array abaixo represente os nós de uma árvore binária completa e que o primeiro elemento represente a raiz dessa árvore. Desenhe a referida árvore.

0	1	2	3	4	5	6	7	8	9
55	8	13	28	23	9	25	11	70	19

61. Apresente o conteúdo do array do exercício **60** quando a árvore que ele representa é transformada num heap de máximo.

62. Apresente o conteúdo do array do exercício **60** quando a árvore que ele representa é transformada num heap de mínimo.

63. Considerando o método de ordenação **HeapSort**, preencha a seguinte tabela informando quando ocorre o melhor caso desse algoritmo e qual é seu custo temporal em termos de notação ó. Faça o mesmo para o caso médio e o pior caso desse algoritmo.

Método HeapSort					
Melhor Caso		Caso Médio		Pior Caso	
Quando ocorre?	Custo?	Quando ocorre?	Custo?	Quando ocorre?	Custo?

64. Por que o algoritmo de ordenação **HeapSort** é inerentemente instável?

65. Em que situações o algoritmo **QuickSort** não é apropriado?

66. Por que o algoritmo **QuickSort** é normalmente escolhido quando não se tem noção sobre o estado de ordenação da tabela a ser ordenada?

67. (a) Existe alguma situação específica na qual a versão de **QuickSort** que usa a última chave da tabela como pivô não é adequada? (b) Por que essa versão de **QuickSort** é tão lenta quando os dados estão ordenados na ordem inversa?

68. Por que a abordagem mediana de três usada pelo algoritmo **QuickSort** não deve ser usada para partições muito pequenas?

69. O que é ponto de corte no algoritmo **QuickSort**?

70. Mostre como ocorre a ordenação dos valores *3*, *1*, *4*, *1*, *5*, *9*, *2* e *6* usando **QuickSort** com mediana de três e um ponto de corte igual a *3*.

71. Usando a implementação básica de **QuickSort**, determine o custo temporal desse algoritmo para uma tabela:
(a) Ordenada
(b) Inversamente ordenada
(c) Sem ordenação

72. Determine o custo temporal de **MergeSort** para uma tabela:
(a) Ordenada
(b) Inversamente ordenada
(c) Sem ordenação

73. Mostre como **HeapSort** ordena o array contendo os valores: *142*, *543*, *123*, *65*, *453*, *879*, *572*, *434*, *111*, *242*, *811* e *102*.

74. Se você desejar ordenar um array em ordem decrescente usando **HeapSort**, você utilizará um heap de máximo ou um heap de mínimo?

75. Qual é o custo temporal de **HeapSort** quando a tabela já está ordenada?

76. Suponha que uma tabela T_1 já esteja ordenada na ordem desejada e que outra tabela T_2 contendo os mesmos elementos da tabela T_1 esteja ordenada em ordem inversa. Qual das duas tabelas, T_1 ou T_2, é ordenada mais rapidamente por **HeapSort**?

77. Mostre como são ordenados os valores *3*, *1*, *4*, *1*, *5*, *9*, *2* e *6* usando **MergeSort**.

78. (a) Na implementação básica de **QuickSort**, qual é o custo temporal quando todas as chaves da tabela a ser ordenada são iguais? (b) E se a abordagem mediana de três for utilizada com **QuickSort**?

79. Suponha que se escolha o elemento na posição média do array como pivô. Isso torna improvável que **QuickSort** tenha custo temporal quadrático?

80. (a) Mostre, usando análise assintótica, que o algoritmo **HeapSort** tem custo temporal $\theta(n)$ quando as chaves da tabela a ser ordenada são iguais. (b) Isso não contradiz o **Teorema 11.19**? Explique sua resposta.

81. Assim como ocorre com **QuickSort**, o algoritmo **MergeSort** também efetua $\theta(\log n)$ chamadas recursivas. (a) Essa afirmação é verdadeira? (b) Se esse for o caso, por que, então, o custo espacial de **MergeSort** não é $\theta(n \cdot \log n)$ em vez de $\theta(n)$?

82. O algoritmo **QuickSort** apresenta recursão de cauda (v. **Capítulo 4** do **Volume 1**). Então, por que ele não pode ser transformado facilmente num algoritmo iterativo e apresentar custo espacial $\theta(1)$?

83. (a) Como o algoritmo **QuickSort** pode ser transformado num algoritmo iterativo (em vez de recursivo)? (b) Efetuando essa transformação o custo temporal ou espacial ser melhorado em termos de análise assintótica?

84. Que garantia oferece **HeapSort** em relação a **QuickSort**?

85. Supondo que o pivô seja o primeiro elemento, ilustre graficamente (v. **Figura 11–12**) a operação de partição usada por **QuickSort** do array `{15, 21, 10, 7, 11, 6, 9, 4, 22, 3, 5, 10}`.

86. Como se modifica **QuickSort** de modo que esse algoritmo ordene em ordem decrescente?

87. Complete a prova do **Lema 11.5**, mostrando que a relação de recorrência:

$$T(n) = \begin{cases} 0 & \text{se } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 & \text{se } n > 1 \end{cases}$$

tem uma solução $T(n)$ tal que satisfaz as seguintes relações:

(a) $T(n) \geq n \log_2 n$

(b) $T(n) \leq 2n \log_2 n$

88. Por que na análise espacial do algoritmo **MergeSort** (v. **Teorema 11.17**) utiliza-se a regra da soma, em vez da regra do produto?

Ordenação com Custo Temporal Linear (Seção 11.4)

89. (a) Descreva o método **CountingSort**. (b) Que restrições esse método de ordenação impõe às chaves da tabela a ser ordenada?

90. Suponha que um array contém números inteiros, sendo que alguns dos quais são negativos. Como se poderia ordenar esse array usando **CountingSort**?

91. Como o **Passo 4** do algoritmo **CountingSort** é realmente levado a efeito?

92. Por que um array auxiliar é necessário para armazenar a lista ordenada no algoritmo **CountingSort**? Ou, em outras palavras, por que o resultado do **Passo 4** não é copiado diretamente para o array recebido como parâmetro?

93. Como o algoritmo **CountingSort** pode ser simplificado quando os elementos do array a ser ordenado são valores inteiros?

94. Considerando o método **CountingSort**, preencha a seguinte tabela informando quando ocorre o melhor caso desse algoritmo e qual é seu custo temporal em termos de notação ó. Faça o mesmo para o caso médio e o pior caso desse algoritmo.

Método CountingSort					
Melhor Caso		Caso Médio		Pior Caso	
Quando ocorre?	Custo?	Quando ocorre?	Custo?	Quando ocorre?	Custo?

95. (a) Descreva o método **BucketSort**. (b) Que restrições esse método de ordenação impõe às chaves da tabela a ser ordenada?

96. Qual é a fase mais complicada do método **BucketSort**? Explique sua resposta.

97. O método de ordenação **BucketSort** usa comparações? Explique sua resposta.

98. Por que se diz que o uso do método **BucketSort** em ordenação é análogo ao uso de dispersão em busca?

99. Considerando o método **BucketSort**, preencha a seguinte tabela informando quando ocorre o melhor caso desse algoritmo e qual é seu custo temporal em termos de notação ó. Faça o mesmo para o caso médio e o pior caso desse algoritmo.

Método BucketSort					
Melhor Caso		Caso Médio		Pior Caso	
Quando ocorre?	Custo?	Quando ocorre?	Custo?	Quando ocorre?	Custo?

100. Compare ordenação por contagem com ordenação com coletores, apresentando as semelhanças e diferenças entre esses dois métodos de ordenação.

101. Descreva o método de ordenação por base (**RadixSort**).

102. Considerando o método **RadixSort**, preencha a seguinte tabela informando quando ocorre o melhor caso desse algoritmo e qual é seu custo temporal em termos de notação ó. Faça o mesmo para o caso médio e o pior caso desse algoritmo.

Método RadixSort					
Melhor Caso		Caso Médio		Pior Caso	
Quando ocorre?	Custo?	Quando ocorre?	Custo?	Quando ocorre?	Custo?

103. Em que situações o método **RadixSort** é viável?

104. Por que se diz que o método **RadixSort** não usa comparações?

105. Como o algoritmo **RadixSort** consegue obter custo temporal $\theta(n \cdot d)$?

106. Como o algoritmo **RadixSort** pode ser alterado de modo que ele ordene tabelas em ordem decrescente?

107. O algoritmo de ordenação **RadixSort** poderia usar um array de pilhas em vez de um array de filas? Elabore seu raciocínio.

108. Qual é a maneira mais eficiente de ordenar um milhão de inteiros de 32 bits?

109. É possível escrever uma função genérica [tal como **qsort()** da biblioteca padrão de C] para o algoritmo **BucketSort**?

110. O que há de comum entre os métodos de ordenação **CountingSort**, **BucketSort** e **RadixSort**?

111. Por que ordenação com coletores é mais eficiente para listas com chaves densamente distribuídas, enquanto ordenação por base é melhor para listas com chaves esparsamente distribuídas?

112. É correto afirmar que ordenação de palavras da língua portuguesa pode ser efetuada com **RadixSort** considerando a base igual a 26, que é o tamanho do alfabeto usado nessa língua?

113. (a) Por que o algoritmo **RadixSort** ordena o dígito menos significante antes de ordenar os dígitos mais significantes? (b) Por que esse algoritmo efetua mais de uma ordenação com coletores se é a última dessas ordenações que coloca os elementos da lista em ordem?

114. Por que o subalgoritmo de ordenação usado por **RadixSort** tem que ser estável?

115. Qual é a suposição fundamental adotada para que o algoritmo **BucketSort** tenha custo $\theta(n + k)$?

Limite Inferior para Algoritmos Baseados em Comparações (Seção 11.5)

116. No pior caso, qual é o menor custo temporal de um método que de ordenação que usa comparação?

117. (a) Por que a árvore de decisão associada a um algoritmo arbitrário baseado em comparações tem altura dada por $log_2(n!)$, em que n é o número de itens a ser ordenados? (b) Por que o número de folhas dessa árvore é $n!$?

118. Mostre que, no caso médio, qualquer algoritmo de ordenação baseado em comparações efetua pelo menos $\lfloor log_2(n!) \rfloor$ comparações.

Algoritmos de Divisão e Conquista (Seção 11.6)

119. (a) O que é um algoritmo de divisão e conquista? (b) Por que **QuickSort** e **MergeSort** são considerados algoritmos de divisão e conquista?

120. (a) Apresente outros algoritmos de divisão e conquista além daqueles apresentados neste livro. (b) Apresente dois exemplos de problemas que não podem ser resolvidos por um algoritmo de divisão e conquista.

121. Por que, apesar das aparências, busca binária não é um algoritmo de divisão e conquista?

Avaliações de Métodos de Ordenação (Seção 11.7)

122. Suponha que se deseje ordenar em ordem crescente uma tabela com *100* elementos e que tal tabela já esteja assim ordenada. Quantas comparações de chaves serão efetuadas se o algoritmo de ordenação utilizado for:
(a) **BubbleSort**
(b) **QuickSort**
(c) **SelectionSort**
(d) **InsertionSort**

123. Suponha que se deseje ordenar em ordem crescente uma tabela com *100* elementos e tal tabela esteja ordenada em ordem decrescente. Quantas comparações serão efetuadas se o algoritmo de ordenação utilizado for:
(a) **BubbleSort**
(b) **QuickSort**
(c) **SelectionSort**
(d) **InsertionSort**

124. Além de custo temporal e custo espacial, qual é o outro custo que se deve levar em consideração na análise de um algoritmo de ordenação?

125. Apresente uma situação na qual o custo de programação justifique o uso de um algoritmo de ordenação com custo temporal $\theta(n^2)$.

126. Por que é desejável que funções chamadas no corpo de uma função de ordenação sejam substituídas por instruções equivalentes que não envolvem chamadas de funções?

127. Qual é o ganho que se obtém quando se transforma uma função de ordenação recursiva numa função equivalente iterativa?

128. Por que, normalmente, não faz sentido usar análise amortizada na avaliação de algoritmos de ordenação?

129. Suponha que um algoritmo de ordenação deva satisfazer as seguintes condições:
(1) A ordenação deve ser estável
(2) As chaves estão ordenadas em ordem crescente
(3) O espaço disponível é muito restrito
Qual método de ordenação deve ser escolhido de modo a satisfazer todas essas condições?

130. Se tanto **BubbleSort** quanto **InsertionSort** têm o mesmo custo temporal $\theta(n^2)$ e **BubbleSort** é mais fácil (i.e., intuitivo) de implementar, por que se dá preferência a **InsertionSort** em detrimento à **BubbleSort**?

131. Suponha que se tenha um milhão de cadeias de DNA cada uma das quais com exatamente 100 caracteres derivados do alfabético genético $\sum = \{A, C, N, T\}$. Qual algoritmo de ordenação seria a melhor opção?

132. Preencha a segunda coluna da tabela a seguir com *E*, para algoritmos estáveis, ou *I*, para algoritmos instáveis.

Algoritmo de Ordenação	Estável?
BubbleSort	
InsertionSort	
SelectionSort	
HeapSort	
QuickSort	
BucketSort	
MergeSort	

133. Suponha que se sabe que uma lista indexada está desordenada com relação a uma determinada chave. Qual é o menor custo temporal no pior caso com que se pode verificar se essa lista contém pelo menos dois elementos com duplicidade dessa chave?

134. Dentre os algoritmos de ordenação discutidos neste capítulo, quais deles apresentam boa localidade de referência?

135. Por que **HeapSort** não é tão bom quanto parece?

Exemplos de Programação (Seção 11.8)

136. Dada uma lista simplesmente encadeada de n elementos, como se poderia ordená-la com custo temporal $\theta(n \log n)$, de modo estável e com custo espacial $\theta(1)$?

137. (a) O que é ordenação de ponteiros? (b) Quando o uso de ordenação de ponteiros é justificável?

138. Descreva o problema da bandeira holandesa e como ele pode ser resolvido.

139. (a) O que é uma lista bitônica? (b) O que é ponto bitônico?

11.10 Exercícios de Programação

EP11.1 Escreva um programa que cria uma lista indexada com *100* elementos inteiros escolhidos aleatoriamente. Então o programa ordena essa lista usando os algoritmos **BubbleSort**, **SelectionSort** e **InsertionSort** e apresenta uma tabela contendo o número de comparações efetuadas por cada algoritmo.

EP11.2 Escreva um programa que cria uma lista indexada com *100* elementos inteiros escolhidos aleatoriamente. Então o programa ordena essa lista usando os algoritmos **QuickSort**, **MergeSort** e **HeapSort** e apresenta uma tabela contendo o número de comparações efetuadas por cada algoritmo.

EP11.3 Implemente uma versão de **BubbleSort** que alterna passagens pela tabela da esquerda para direita e da direita para esquerda. Esse algoritmo mais rápido do que **BubbleSort** é chamado **BubbleSort bidirecional** e é uma ligeira variação de **BubbleSort**. **BubbleSort** bidirecional difere de **BubbleSort** tradicional no sentido de que em vez de repetidamente atravessar a tabela do início para o final, ele passa alternadamente do início para o final e então do final para o início.

EP11.4 Use ponteiros genéricos e ponteiro para função para implementar uma função genérica para o algoritmo **InsertionSort**.

EP11.5 Use ponteiros genéricos e ponteiro para função para implementar uma função genérica para o algoritmo **QuickSort**.

EP11.6 Use ponteiros genéricos e ponteiro para função para implementar uma função genérica para o algoritmo **MergeSort**.

EP11.7 Implemente uma função para ordenação de listas encadeadas usando o algoritmo **MergeSort** descrito na **Seção 11.3.2**.

EP11.8 Use ponteiros genéricos e ponteiro para função para implementar uma função genérica para o algoritmo **HeapSort**.

EP11.9 Dado um conjunto de n pontos no plano, ponto (x_i, y_i) **domina** (x_j, y_j) se $x_i > x_j$ e $y_i > y_j$. Um **ponto máximo** é um ponto que não é dominado por nenhum outro ponto no conjunto. (a) Implemente uma função que encontra todos os máximos de um conjunto de pontos. (b) Qual é o custo temporal dessa função?

EP11.10 Escreva uma função que implementa **HeapSort** de modo que ela ordena apenas itens que estão no intervalo entre `inf` e `sup`, que são passados como parâmetros para a função.

EP11.11 Escreva uma função para comparação de datas.

EP11.12 Implemente o algoritmo **MergeSort** sem usar recursão.

EP11.13 Escreva um programa que lê uma lista de palavras e as apresenta agrupadas de acordo com suas rimas. O procedimento a ser seguido é o seguinte:

(i) Leia a lista de palavras num array de strings

(ii) Inverta as letras em cada palavra (p. ex., *laranja* se torna *ajnaral*)

(iii) Ordene o array de palavras resultante

(iv) Inverta as letras em cada palavra de volta aos seus estados originais

(v) Apresente o resultado

Agora a palavra *laranja* deverá estar próxima de palavras, como *arranja* e *canja*, que rimam com *laranja*.

EP11.14 Escreva um programa que leia uma sequência de strings e ordene-os em ordem crescente, ignorando distinção entre letras maiúsculas e minúsculas.

EP11.15 Escreva um programa para ordenar palavras da língua portuguesa.

EP11.16 Suponha que você tenha um array de n elementos contendo apenas duas chaves distintas, cujos valores são *verdadeiro* e *falso*. Implemente uma função que apresente custo temporal $\theta(n)$ para ordenar esse array de modo que todos os elementos falsos precedem os elementos verdadeiros. Não é permitido o uso de espaço adicional [i.e., o custo espacial deve ser $\theta(1)$].

EP11.17 Escreva um programa que cria uma lista indexada aleatória com *100* elementos inteiros. Então o programa ordena essa lista usando três versões de **QuickSort**: (1) o pivô é o primeiro elemento da lista, (2) o pivô é o último elemento da lista e (3) o pivô é o primeiro elemento que se encontra na metade da lista. O programa deve apresentar uma tabela contendo o número de comparações efetuadas por cada algoritmo.

EP11.18 Escreva um programa que cria uma lista indexada aleatória com *100* elementos inteiros. Então o programa ordena essa lista usando três versões de **QuickSort**, cada uma das quais usa mediana de três e ponto de corte igual a: (1) 3, (2) 10 e (3) 20. O programa deve apresentar uma tabela contendo o número de comparações efetuadas por cada algoritmo.

EP11.19 Dados dois vetores de mesmo comprimento no espaço de dimensão n, encontre uma permutação desses vetores de modo que o produto interno deles é tão pequeno quanto possível.

EP11.20 Pode-se estender o esquema de ordenação de ponteiros apresentado na **Seção 11.8.2** para manter uma grande tabela ordenada em mais de uma chave. Os dados podem ser fisicamente armazenados de acordo com a primária chave e arrays auxiliares podem conter ponteiros para os mesmos dados ordenados em chaves secundárias. Implemente.

EP11.21 Suponha que se tenha um array cujos elementos são do tipo **int**. (a) Escreva uma função em C que encontra os dois elementos desse array cuja soma seja mais próxima de zero. (b) Escreva uma função semelhante àquela do item (a) supondo que o array está ordenado. (c) Qual é o custo temporal de cada uma dessas funções?

EP11.22 Suponha que se tenha uma lista indexada cujos elementos são do tipo **int**. (a) Escreva uma função em C que verifica se há três elementos tais que a soma deles seja igual a um certo valor x. (b) Escreva uma função semelhante àquela do item (a) supondo que a lista está ordenada. (c) Qual é o custo temporal de cada uma dessas funções?

EP11.23 Suponha que se tenha uma lista indexada cujos elementos são do tipo **int** e na qual existem elementos repetidos. (a) Escreva uma função em C que verifica qual é o valor que mais aparece nessa lista. (b) Escreva uma função semelhante àquela do item (a) supondo que a lista está ordenada. (c) Qual é o custo temporal de cada uma dessas funções?

EP11.24 Suponha que um array de inteiros é ordenado e começa com valores negativos e termina com valores positivos. Escreva uma função em C que retorne o índice do primeiro número positivo de tal modo que ela tenha custo temporal $\theta(\log n)$.

EP11.25 Suponha que se tenha um array de elementos do tipo **int** inicialmente ordenado. Assuma que esse array tenha sofrido um número desconhecido de rotações. Apresente uma função em C que encontra um elemento nesse array com custo temporal $\theta(\log n)$. [**Observação:** Uma rotação de um array faz com que o último elemento passe a ocupar a primeira posição do array, o primeiro elemento passe a ocupar a segunda posição e assim por diante.]

EP11.26 Escreva uma função com custo temporal $\theta(n \log n)$ que encontra a mediana de um array de elementos do tipo **int**.

EP11.27 Um elemento de uma lista indexada com n elementos é uma **maioria** se seu número de ocorrência é maior do que $n/2$. Escreva uma função em C que retorne o valor da maioria de uma lista indexada com n elementos do tipo **int**, se tal elemento existir.

EP11.28 Escreva uma função em C que cria duas partições de um array de elementos do tipo **int**, de modo que a primeira partição contém apenas números pares e a segunda partição contém apenas números ímpares.

EP11.29 Existem duas vantagens no uso de base *2* na implementação de **RadixSort** para a ordenação de inteiros: (1) o uso de divisão pode ser substituído por uma operação de baixo nível que é bem mais eficiente e (2) apenas duas filas são necessárias. Implemente o algoritmo de ordenação para a base *2* descrito a seguir:

1. Inicie uma variável i com 1
2. Inicie duas filas F_0 e F_1
3. Para cada item x_j da lista a ser ordenada, aplique a operação de conjunção sobre bits entre ele e i ; i.e., efetue a operação x_j & i
4. Se o resultado de x_j & i for 0, acrescente x_j à lista F_0; caso contrário, acrescente-o à lista F_1
5. Escreva os elementos da fila F_0 seguidos dos elementos de F_1 na lista a ser ordenada
6. Se i estiver em sua posição mais à esquerda (p. ex., *1000 0000*, se os números que estão sendo ordenados forem de 8 bit), encerre
7. Caso contrário, efetue uma operação de deslocamento esquerdo $i << 2$ e volte ao **Passo 2**

EP11.30 Suponha que se tenha uma lista indexada cujos elementos são do tipo int. (a) Escreva uma função em C que verifica se há dois elementos tais que a soma deles seja igual a um certo valor x. (b) Qual é o custo temporal dessa função? Escreva uma função semelhante àquela do item supondo que a lista está ordenada e que tenha custo temporal linear.

EP11.31 Escreva um programa semelhante àquele apresentado na **Seção 11.7.5** para testar os métodos de ordenação **CountingSort**, **BucketSort** e **RadixSort**.