

Capítulo 11

Busca e ordenação

Em diversas aplicações, os dados devem ser armazenados obedecendo a uma determinada ordem. Alguns algoritmos podem explorar a ordenação dos dados para operar de maneira mais eficiente, do ponto de vista de desempenho computacional. Para ordenar os dados, temos basicamente duas alternativas: ou inserimos os elementos na estrutura de dados, respeitando a ordenação (dizemos que a ordenação é garantida por construção) ou, a partir de um conjunto de dados já criado, aplicamos um algoritmo para ordenar seus elementos. Neste capítulo, vamos apresentar algoritmos de busca em vetor e analisar os benefícios de fazer busca em conjuntos ordenados. Em seguida, vamos discutir dois algoritmos de ordenação de vetor que podem ser empregados em aplicações computacionais.

11.1 Busca em vetor

Nesta seção, discutiremos estratégias para efetuar a busca de um elemento em um vetor. A operação de busca é encontrada com muita frequência em aplicações computacionais. Por exemplo, um programa de controle de estoque pode buscar, dado um código numérico ou um nome, a descrição e as características de determinado produto. Se tivermos um grande número de produtos cadastrados, o método para efetuar a busca deverá ser eficiente, caso contrário a busca poderá ser muito demorada, inviabilizando a operação.

Vamos considerar que temos nossos dados armazenados em um vetor e discutir os algoritmos de busca que podemos empregar.

11.1.1 Busca linear

A forma mais simples de fazer uma busca em um vetor consiste em percorrer o vetor, elemento a elemento, verificando se o elemento de interesse é igual a um dos elementos do vetor. Esse algoritmo pode ser implementado conforme ilustrado pelo código a seguir, considerando um vetor de números inteiros. A função apresentada tem como valor de retorno o índice do vetor no qual foi encontrado o elemento; se o elemento não for encontrado, o valor de retorno é -1 .

```

int busca (int n, int* vet, int x)
{
    int i;
    for (int i=0; i<n; ++i)
        if (x == vet[i])
            return i; /* elemento encontrado */
    return -1; /* percorreu todo o vetor e não encontrou elemento */
}

```

Esse algoritmo de busca é extremamente simples, mas será muito ineficiente quando o número de elementos no vetor for muito grande. Isto porque o algoritmo (a função, no caso) pode ter que percorrer todos os elementos do vetor para verificar se determinado elemento está ou não presente. Dizemos que, no pior caso, será necessário realizar n comparações, onde n representa o número de elementos no vetor. Portanto, o desempenho computacional desse algoritmo varia linearmente em relação ao tamanho do problema. Por isso, chamamos esse algoritmo de *busca linear*.

Em geral, usamos a notação “Big-O” para expressar como a complexidade de um algoritmo varia com o tamanho do problema. Assim, nesse caso em que o tempo computacional varia de forma linear com o tamanho do problema, dizemos que se trata de um algoritmo de ordem linear e expressamos sua complexidade computacional por $O(n)$.

Além do pior caso, podemos analisar o caso médio, isto é, o caso que ocorre na média. Já vimos que o algoritmo em questão requer n comparações quando o elemento não está presente no vetor. E no caso do elemento estar presente, quantas operações de comparação são, em média, necessárias? Na média, podemos concluir que são, necessárias $n/2$ comparações. Em termos de ordem de complexidade, no entanto, continuamos a ter uma variação linear, isto é, $O(n)$, pois dizemos que $O(kn)$, onde k é uma constante relativamente pequena, é igual a $O(n)$.

Em diversas aplicações reais, precisamos de algoritmos de busca mais eficientes. Seria possível melhorar a eficiência do algoritmo de busca mostrado? Infelizmente, se os elementos estiverem armazenados em uma ordem aleatória no vetor, não temos como melhorar o algoritmo de busca, pois precisamos verificar todos os elementos. No entanto, se assumirmos, por exemplo, que os elementos estão armazenados em ordem crescente, podemos concluir que um elemento não está presente no vetor assim que acharmos um elemento maior, pois se o elemento que buscamos estivesse presente ele precederia um elemento maior na ordem do vetor.

O código a seguir ilustra a implementação da busca linear, assumindo que os elementos do vetor estão ordenados (vamos assumir ordem crescente).

```

int busca_ord (int n, int* vet, int elem)
{
    int i;
    for (int i=0; i<n; ++i) {
        if (elem == vet[i])
            return i; /* elemento encontrado */
        else if (elem < vet[i])
            return -1; /* interrompe busca */
    }
    return -1; /* percorreu todo o vetor e não encontrou elemento */
}

```

No caso do elemento procurado não pertencer ao vetor, esse segundo algoritmo apresenta um desempenho ligeiramente superior ao primeiro, mas a ordem dessa versão do algoritmo continua sendo linear – $O(n)$. No entanto, se os elementos do vetor estão ordenados, existe um algoritmo muito mais eficiente, que será apresentado a seguir.

11.1.2 Busca binária

Se os elementos do vetor estiverem ordenados, podemos aplicar um algoritmo mais eficiente para realizar a busca. Trata-se do algoritmo de *busca binária*. A ideia do algoritmo é testar o elemento que buscamos com o valor do elemento armazenado no meio do vetor. Se o elemento que buscamos for menor que o elemento do meio, sabemos que, se o elemento estiver presente no vetor, ele estará na primeira parte do vetor; se for maior, estará na segunda parte do vetor; se for igual, achamos o elemento no vetor. Se concluirmos que o elemento está numa das partes do vetor, repetimos o procedimento considerando apenas a parte que restou: comparamos o elemento que buscamos com o elemento armazenado no meio dessa parte. Esse procedimento é continuamente repetido, subdividindo a parte de interesse, até encontrarmos o elemento ou chegarmos a uma parte do vetor com tamanho zero.

O código a seguir ilustra uma implementação de busca binária num vetor de valores inteiros ordenados de forma crescente.

```
int busca_bin (int n, int* vet, int elem)
{
    /* no início consideramos todo o vetor */
    int ini = 0;
    int fim = n-1; int meio;
    /* enquanto a parte restante for maior que zero */
    while (ini <= fim) {
        int meio = (ini + fim) / 2;
        if (elem < vet[meio])
            fim = meio - 1;          /* ajusta posição final */
        else if (elem > vet[meio])
            ini = meio + 1;         /* ajusta posição inicial */
        else
            return meio;           /* elemento encontrado */
    }

    return -1; /* não encontrou: restou parte de tamanho zero */
}
```

O desempenho desse algoritmo é muito superior ao da busca linear. Novamente, o pior caso caracteriza-se pela situação do elemento que buscamos não estar no vetor. Quantas vezes precisamos repetir o procedimento de subdivisão para concluir que o elemento não está presente no vetor? Inicialmente, todo o vetor é considerado; a cada repetição, a parte considerada na busca é dividida à metade. A tabela a seguir mostra o tamanho do vetor ao fim de cada repetição do laço do algoritmo.

Repetição	Tamanho do problema
0	n
1	$n/2$
2	$n/4$
3	$n/8$
...	...
$\log n$	1

Sendo assim necessárias $\log n$ repetições. Como fazemos um número constante de comparações a cada repetição (duas comparações por ciclo), podemos concluir que a ordem de complexidade desse algoritmo é $O(\log n)$.

O algoritmo de busca binária consiste em repetir o mesmo procedimento repetidamente, podendo ser naturalmente implementado de forma recursiva. Embora a implementação não recursiva seja mais eficiente e mais adequada para esse algoritmo, a implementação recursiva vale a pena ser apresentada. Na implementação recursiva, temos dois casos a serem tratados. No primeiro, a busca deve continuar na primeira metade do vetor; logo, chamamos a função recursivamente passando como parâmetros o número de elementos dessa primeira parte restante e o mesmo ponteiro para o primeiro elemento, pois a primeira parte tem o mesmo primeiro elemento do que o vetor como um todo. No segundo caso, a busca deve continuar apenas na segunda parte do vetor; logo, passamos na chamada recursiva, além do número de elementos restantes, um ponteiro para o primeiro elemento dessa segunda parte. Para simplificar, uma primeira versão apenas informa se o elemento pertence ou não ao vetor, tendo como valor de retorno falso (0) ou verdadeiro (1). Uma possível implementação usando essa estratégia é mostrada a seguir.

```
int pertence_rec (int n, int* vet, int elem)
{
    int meio;
    if (n <= 0) /* testa condição de contorno: tamanho zero */
        return 0;
    else {
        meio = n / 2;           /* deve buscar o elemento do meio */
        if (elem < vet[meio])
            return pertence_rec (meio, vet, elem);
        else if (elem > vet[meio])
            return pertence_rec (n-1-meio, &vet[meio+1], elem);
        else
            return 1;           /* elemento encontrado */
    }
}
```

Em particular, devemos notar a expressão `&vet[meio+1]` que, como sabemos, resulta no ponteiro para o primeiro elemento da segunda parte do vetor.

Se quisermos que a função tenha como valor de retorno o índice do elemento, devemos acertar o valor retornado pela chamada recursiva na segunda parte do vetor. Uma implementação dessa função de busca é apresentada a seguir:

```

int busca_bin_rec (int n, int* vet, int elem)
{
    if (n <= 0) /* testa condição de contorno: tamanho zero */
        return -1;
    else {
        int meio = n / 2; /* deve buscar o elemento do meio */
        if (elem < vet[meio])
            return busca_bin_rec(meio, vet, elem);
        else if (elem > vet[meio])
        {
            int r = busca_bin_rec(n-1-meio, &vet[meio+1], elem);
            if (r<0) return -1;
            else      return meio+1+r;
        }
        else
            return meio; /* elemento encontrado */
    }
}

```

Note a correção do índice quando o elemento é encontrado na segunda parte do vetor: $meio + 1 + r$. Esta correção é necessária pois a chamada recursiva retorna o índice do elemento na segunda parte do vetor. Para se chegar ao índice desse mesmo elemento no vetor como um todo, temos que acrescentar o número de elementos do vetor que precede essa segunda parte.

11.2 Ordenação de vetor

Devido ao seu uso muito frequente, é importante ter à disposição algoritmos de ordenação (*sorting*) eficientes, tanto em termos de tempo (devem ser rápidos) como em termos de espaço (devem ocupar pouca memória durante a execução). Vamos descrever os algoritmos de ordenação considerando o seguinte cenário:

- A entrada é um vetor cujos elementos precisam ser ordenados.
- A saída é o mesmo vetor com seus elementos na ordem especificada.

Portanto, vamos discutir ordenação de vetores *in place*, isto é, os elementos do vetor original são repositionados para respeitarem a ordem desejada. Como veremos, os algoritmos de ordenação podem ser aplicados a qualquer informação, desde que exista uma ordem definida entre os elementos. Podemos, por exemplo, ordenar um vetor de valores inteiros adotando uma ordem crescente ou decrescente; podemos também aplicar algoritmos de ordenação em vetores que guardam informações mais complexas; por exemplo, um vetor que guarda os dados relativos a alunos de uma turma, com nome, número de matrícula etc. Nesse caso, a ordem entre os elementos tem que ser definida usando uma das informações do aluno como chave da ordenação: alunos ordenados pelo nome, alunos ordenados pelo número de matrícula etc.

Nos casos em que a informação é composta por diversos campos, raramente se encontra toda a informação relevante sobre os elementos do vetor no próprio vetor; em vez disso, cada componente do vetor pode conter apenas um ponteiro para a informação propriamente dita, que pode ficar em outra posição na memória. Assim, a ordenação pode

ser feita sem a necessidade de mover grandes quantidades de informações para rearrumar os elementos do vetor na ordem desejada. Para trocar a ordem entre dois elementos, apenas os ponteiros são trocados. Em muitos casos, devido ao grande volume, as informações podem ficar em um arquivo de disco e o elemento do vetor ser apenas uma referência para a posição da informação nesse arquivo.

Neste capítulo, examinaremos os algoritmos de ordenação conhecidos como *ordenação bolha* e *ordenação rápida*, ou, mais precisamente, versões simplificadas desses algoritmos.

11.3 Ordenação bolha

O algoritmo de ordenação bolha (*bubble sort*) recebeu esse nome pela imagem pitoresca usada para descrevê-lo: os maiores elementos do vetor são mais leves, e sobem como bolhas até suas posições corretas. A ideia fundamental é fazer uma série de comparações entre os elementos do vetor. Quando dois elementos adjacentes estão fora de ordem, há uma inversão e esses dois elementos são trocados de posição, ficando em ordem relativa correta. Assim, o primeiro elemento é comparado com o segundo. Se uma inversão for encontrada, a troca é feita. Em seguida, independentemente de se houve ou não troca após a primeira comparação, o segundo elemento é comparado com o terceiro, e, caso uma inversão seja encontrada, a troca é feita. O processo continua até que o penúltimo elemento seja comparado com o último. Com este processo, garante-se que os elementos maiores “subam” no vetor e que o elemento de maior valor do vetor seja levado para a última posição. A ordenação continua, posicionando o segundo maior elemento, o terceiro etc., até que todo o vetor esteja ordenado. O conceito do que é *maior*, naturalmente, depende do critério de ordenação adotado. Se, por exemplo, quisermos ordenar valores numéricos em ordem decrescente, os menores valores serão considerados “maiores” nesta descrição. Para simplificar a explicação, vamos considerar inicialmente que estamos ordenando valores em ordem crescente.

Para exemplificar, vamos considerar que os elementos do vetor que queremos ordenar sejam valores inteiros. Assim, consideremos a ordenação do seguinte vetor:

25 86 48 92 57 12 33 37 • Vetor original

A cada passada do algoritmo, compara-se todos os pares de elementos adjacentes, fazendo a troca de valores quando inversões são detectadas. Na primeira passada, para o vetor em questão, compara-se o primeiro elemento com o segundo (25 e 86); como não há inversão (a ordem relativa entre estes dois elementos já está correta), não há troca. Em seguida, compara-se o segundo elemento com o terceiro (86 e 48); neste caso, há inversão, e os valores dos dois elementos são trocados. Em seguida, compara-se o terceiro elemento, agora com o valor 86, com o quarto, com valor 92; não há inversão e não se faz a troca. Continuando, o quarto elemento é comparado com o quinto (92 e 57); neste caso, novamente detecta-se uma inversão e a troca é feita. O procedimento continua até que o penúltimo elemento seja comparado com o último. Neste exemplo, em cada uma destas comparações subsequentes, haverá inversão, e a troca será realizada. Isto porque o valor 92 é o maior elemento do vetor, sendo levado para a última posição do vetor. As trocas dessa primeira passada são ilustradas a seguir. Cada linha mostra o resultado de trocas de valores adjacentes:

25 86 48 92 57 12 33 37 *Início do passo: i=7*

48 86

57 92

12 92

33 92

37 92

25 48 86 57 12 33 37 92 *Final do primeiro passo*

A parte sabidamente já ordenada do vetor está sublinhada. O procedimento deve continuar, agora considerando-se o vetor com um elemento a menos, pois o maior elemento já está na posição correta. Os próximos passos realizam as trocas indicadas a seguir:

25 48 86 57 12 33 37 92 *Início do passo: i=6*

57 86

 12 86

33 86

37 86

25 48 57 12 33 37 86 92 *Início do passo: i=5*

12 57

33 57

37 57

25 48 12 33 37 57 86 92 *Início do passo: i=4*

12 48

33 48

37 48

25 12 33 37 48 57 86 92 *Início do passo: i=3*

12 25

12 25 33 37 48 57 86 92 *Início do passo: i=2*

12 25 33 37 48 57 86 92 *Início do passo: i=1*

12 25 33 37 48 57 86 92 *Fim do último passo*

Na realidade, após a troca dos valores 25 com 12, o vetor se encontra totalmente ordenado, mas esse fato não é levado em consideração por essa versão do algoritmo, e o procedimento continua até que reste apenas um elemento no vetor, e, consequentemente, que o vetor esteja todo ordenado.

Uma função que implementa esse algoritmo é apresentada a seguir. A função recebe como parâmetros o número de elementos e o ponteiro do primeiro elemento do vetor que se deseja ordenar. Vamos considerar a ordenação de um vetor de valores inteiros.

```
/* Ordenação bolha */
void bolha (int n, int* v)
{
    for (int i=n-1; i>=1; --i) {
        for (int j=0; j<i; ++j) {
            if (v[j]>v[j+1]) {
                int temp = v[j]; /* troca */
                v[j] = v[j+1];
                v[j+1] = temp;
            }
        }
    }
}
```

Note que o laço da variável i apenas controla o número de pares de elementos adjacentes que devem ser comparados. Na ilustração do algoritmo em funcionamento mostrada, cada passo foi identificado pelo valor desta variável.

Para evitar que o processo continue mesmo depois de o vetor estar ordenado, podemos interromper o processo quando houver uma passagem inteira sem trocas, usando uma variante do algoritmo apresentado:

```
/* Ordenação bolha (2a. versão) */
void bolha (int n, int* v)
{
    for (int i=n-1; i>0; --i) {
        int troca = 0;
        for (int j=0; j<i; ++j) {
            if (v[j]>v[j+1]) {
                int temp = v[j]; /* troca */
                v[j] = v[j+1];
                v[j+1] = temp;
                troca = 1;
            }
        }
        if (troca == 0) /* se não houve troca */
            return;
    }
}
```

A variável troca guarda o valor 0 quando uma passada do vetor (no `for` interno) se faz sem nenhuma troca. Isto significa que não foi encontrada nenhuma inversão entre os elementos adjacentes e que, portanto, o vetor está ordenado.

O esforço computacional despendido pela ordenação de um vetor pode ser determinado pelo número de comparações, que serve também para estimar o número máximo de trocas que podem ser realizadas. Na primeira passada, fazemos $n - 1$ comparações; na segunda, $n - 2$; na terceira, $n - 3$ e assim por diante. Logo, o tempo total gasto pelo algoritmo é proporcional a $(n - 1) + (n - 2) + \dots + 2 + 1$. A soma desses termos

é proporcional a n^2 . Portanto, o desempenho computacional desse algoritmo varia de forma quadrática em relação ao tamanho do problema. Dizemos que o algoritmo tem ordem de complexidade $O(n^2)$.

No melhor caso, quando o vetor fornecido estiver quase ordenado, o procedimento pode ser capaz de ordenar numa única passada. Esse fato, no entanto, não pode ser usado para fazer uma análise de desempenho do algoritmo, pois o melhor caso representa uma situação muito particular.

Uma função cliente para testar esse algoritmo pode ser dada simplesmente por:

```
#include <stdio.h>
int main (void)
{
    int v[8] = {25, 48, 37, 12, 57, 86, 33, 92};
    bolha(8, v);
    printf("Vetor ordenado:");
    for (int i=0; i<8; ++i)
        printf("%d ", v[i]);
    printf("\n");
    return 0;
}
```

11.3.1 Implementação recursiva

Analisando a forma com que a ordenação bolha funciona, verificamos que o algoritmo procura resolver o problema da ordenação por partes. Inicialmente, o algoritmo coloca em sua posição correta (no final do vetor) o maior elemento, e o problema restante é semelhante ao inicial, só que com um vetor com menos elementos, formado pelos elementos $v[0], \dots, v[n-2]$.

Com base nessa observação, é fácil implementar um algoritmo de ordenação bolha recursivo. Embora não seja a forma mais adequada de implementar esse algoritmo, o entendimento desse algoritmo recursivo nos ajudará a entender a ideia por trás do algoritmo de ordenação rápida que veremos mais adiante.

O algoritmo recursivo de ordenação bolha posiciona o elemento de maior valor e chama, recursivamente, o algoritmo para ordenar o vetor restante, com $n - 1$ elementos.

```
/* Ordenação bolha recursiva */
void bolha_rec (int n, int* v)
{
    int troca = 0;
    for (int j=0; j<n-1; ++j) {
        if (v[j]>v[j+1]) {
            int temp = v[j]; /* troca */
            v[j] = v[j+1];
            v[j+1] = temp;
            troca = 1;
        }
    }
    if (troca != 0) /* houve troca */
        bolha_rec(n-1, v);
}
```

11.3.2 Algoritmo genérico

O mesmo algoritmo de ordenação bolha pode ser aplicado a vetores que guardam outras informações. O código escrito anteriormente pode ser reaproveitado, com exceção de alguns detalhes. Primeiro, a assinatura da função deve ser alterada, pois deixamos de ter um vetor de inteiros; segundo, a forma de comparação entre os elementos também deve ser alterada, pois não podemos, por exemplo, comparar duas cadeias de caracteres usando simplesmente o operador relacional “maior que” ($>$).

Para aumentar o potencial de reutilização do nosso código, podemos reescrever o algoritmo de ordenação apresentado, tornando-o independente da informação armazenada no vetor. Vamos inicialmente discutir como podemos abstrair a função de comparação. O mesmo algoritmo para ordenação de inteiros apresentado pode ser reescrito usando-se uma função auxiliar que faz a comparação. Em vez de comparar diretamente dois elementos com o operador “maior que”, usamos uma função auxiliar que, dados dois elementos, verifica se o primeiro é maior que o segundo.

```

/* Função auxiliar de comparação */
static int compara (int a, int b)
{
    if (a > b) return 1;
    else        return 0;
}

/* Ordenação bolha (3a. versão) */
void bolha (int n, int* v)
{
    int temp, i, j, troca;
    for (int i=n-1; i>0; --i) {
        int troca = 0;
        for (int j=0; j<i; ++j) {
            if (compara(v[j],v[j+1])) {
                int temp = v[j]; /* troca */
                v[j] = v[j+1];
                v[j+1] = temp;
                troca = 1;
            }
        }
        if (troca == 0) /* se não houve troca */
            return;
    }
}

```

Dessa forma, já aumentamos o potencial de reutilização do algoritmo. Podemos, por exemplo, arrumar os elementos em ordem decrescente simplesmente reescrevendo a função `compara`. A ideia fundamental é escrever uma função de comparação que recebe dois elementos e verifica se há uma inversão de ordem entre o primeiro e o segundo. Assim, se tivéssemos um vetor de cadeia de caracteres para ordenar, poderíamos usar a seguinte função de comparação.

```

static int compara (char* a, char* b)
{
  if (strcmp(a,b) > 0) → (0) se as strings a e b são iguais
    return 1;
  else → (< 0) se o conteúdo da string a é menor que b
    return 0; → (> 0) se o conteúdo da string a é maior que b
}
  
```

Consideremos agora um vetor de ponteiros para a estrutura Aluno:

```

typedef struct aluno Aluno;
struct aluno {
  char nome[81];
  char mat[8];
  char turma;
  char email[41];
};
  
```

abc e abd
retorna -1

Uma função de comparação, neste caso, receberia como parâmetros dois ponteiros para a estrutura que representa um aluno e, considerando uma ordenação que usa o nome do aluno como chave de comparação, poderia ter a seguinte implementação:

```

static int compara (Aluno* a, Aluno* b)
{
  if (strcmp(a->nome, b->nome) > 0)
    return 1;
  else
    return 0;
}
  
```

Portanto, o uso de uma função auxiliar para realizar a comparação entre os elementos ajuda na obtenção de um código reutilizável. No entanto, só isso não é suficiente. Para o mesmo código poder ser aplicado a qualquer tipo de informação armazenada no vetor, precisamos tornar a implementação independente do tipo do elemento, isto é, precisamos tornar tanto a própria função de ordenação (`bolha`) quanto a assinatura da função de comparação (`compara`) independentes do tipo do elemento.

Em C, a forma de generalizar o tipo é usar `void*`. Escreveremos o código de ordenação considerando que temos um ponteiro de qualquer tipo e passaremos para a função de comparação dois ponteiros genéricos, um para cada elemento que se deseja comparar. A função de ordenação, no entanto, precisa percorrer o vetor e, para tanto, precisamos passar para a função uma informação adicional – o tamanho, em número de bytes, de cada elemento. A assinatura da função de ordenação poderia então ser dada por:

```
void bolha (int n, void* v, int tam);
```

A função de ordenação por sua vez, receberia dois ponteiros genéricos:

```
int compara (void* a, void* b);
```

```

static int compara (char* a, char* b)
{
    if (strcmp(a,b) > 0) → (0) se as strings a e b são iguais
        return 1;
    else
        return 0;
}
    (< 0) se o conteúdo da string a é menor que b
    (> 0) se o conteúdo da string a é maior que b

```

Consideremos agora um vetor de ponteiros para a estrutura Aluno:

```

typedef struct aluno Aluno;
struct aluno {
    char nome[81];
    char mat[8];
    char turma;
    char email[41];
};

```

abc e abd
retorna -1

Uma função de comparação, neste caso, receberia como parâmetros dois ponteiros para a estrutura que representa um aluno e, considerando uma ordenação que usa o nome do aluno como chave de comparação, poderia ter a seguinte implementação:

```

static int compara (Aluno* a, Aluno* b)
{
    if (strcmp(a->nome, b->nome) > 0)
        return 1;
    else
        return 0;
}

```

Portanto, o uso de uma função auxiliar para realizar a comparação entre os elementos ajuda na obtenção de um código reutilizável. No entanto, só isso não é suficiente. Para o mesmo código poder ser aplicado a qualquer tipo de informação armazenada no vetor, precisamos tornar a implementação independente do tipo do elemento, isto é, precisamos tornar tanto a própria função de ordenação (`bolha`) quanto a assinatura da função de comparação (`compara`) independentes do tipo do elemento.

Em C, a forma de generalizar o tipo é usar `void*`. Escreveremos o código de ordenação considerando que temos um ponteiro de qualquer tipo e passaremos para a função de comparação dois ponteiros genéricos, um para cada elemento que se deseja comparar. A função de ordenação, no entanto, precisa percorrer o vetor e, para tanto, precisamos passar para a função uma informação adicional – o tamanho, em número de bytes, de cada elemento. A assinatura da função de ordenação poderia então ser dada por:

```
void bolha (int n, void* v, int tam);
```

A função de ordenação por sua vez, receberia dois ponteiros genéricos:

```
int compara (void* a, void* b);
```

Assim, se estamos ordenando vetores de inteiros, escrevemos a nossa função de comparação convertendo o ponteiro genérico para um ponteiro de inteiro e fazendo o teste apropriado:

```
/* função de comparação para inteiros */
static int compara (void* a, void* b)
{
    int* p1 = (int*) a; /* converte para o tipo específico */
    int* p2 = (int*) b; /* converte para o tipo específico */

    if ((*p1) > (*p2))
        return 1;
    else
        return 0;
}
```

Se os elementos do vetor fossem ponteiros para a estrutura aluno, a função de comparação poderia ser:

```
/* função de comparação para ponteiros de alunos */
static int compara (void* a, void* b)
{
    Aluno** p1 = (Aluno**) a;
    Aluno** p2 = (Aluno**) b;

    if (strcmp((*p1)->nome, (*p2)->nome) > 0)
        return 1;
    else
        return 0;
}
```

Note que a assinatura da função (o protótipo) é a mesma, e serve para elementos de qualquer tipo, pois usa o tipo genérico `void*`.

Como vimos, o código da função de ordenação necessita percorrer os elementos do vetor. O acesso a determinado elemento `i` do vetor não pode mais ser feito diretamente por `v[i]`, pois `v` agora é do tipo `void*` e não pode ser indexado (o compilador não sabe quantos bytes ocupa cada elemento). Para acessar um elemento qualquer, dado o endereço do vetor, devemos avançar esse endereço de `i * tam` bytes para termos o endereço do elemento `i`. Podemos então escrever uma função auxiliar que faz esse incremento de endereço. Essa função recebe como parâmetros o endereço inicial do vetor (do tipo `void*`), o índice do elemento cujo endereço se quer alcançar e o tamanho (em bytes) de cada elemento. A função retorna o endereço do elemento especificado (também do tipo `void*`). Uma parte útil, porém necessária, dessa função é que, para incrementar o endereço genérico de determinado número de bytes, precisamos antes, temporariamente, converter esse ponteiro genérico para ponteiro de caractere. Assim, a cada incremento do endereço, seu valor avança 1 byte (pois um caractere ocupa 1 byte). O código dessa função auxiliar pode ser dado por:

```

static void* acessa(void* v, int i, int tam)
{
    char* t = (char*)v;
    return (void*)(t + tam*i);
}

```

A função de ordenação identifica a ocorrência de inversões entre elementos e realiza uma troca entre os valores. O código que realiza a troca também tem que ser pensado de forma genérica, pois, como não sabemos o tipo de cada elemento, não temos como declarar a variável temporária para poder realizar a troca. Uma alternativa é fazer a troca dos valores byte a byte (ou caractere a caractere). Para tanto, podemos definir outra função auxiliar que recebe os ponteiros genéricos dos dois elementos que devem ter seus valores trocados, além do tamanho de cada elemento.

```

static void troca(void* a, void* b, int tam)
{
    char* v1 = (char*)a;
    char* v2 = (char*)b;
    for (int i=0; i<tam; ++i) {
        char temp = v1[i];
        v1[i] = v2[i];
        v2[i] = temp;
    }
}

```

Uma alternativa seria usar a função `memcpy` (definida na biblioteca `string.h`), que copia determinada quantidade de bytes de uma posição da memória para outra. Neste caso, no entanto, seria necessário alocar o espaço de memória temporário para realizar a troca:

```

static void troca(void* a, void* b, int tam)
{
    void* temp = malloc(tam);
    memcpy(temp, a, tam);
    memcpy(a, b, tam);
    memcpy(b, temp, tam);
    free(temp);
}

```

memcpy(destino, origem, tamanho)

Para evitar a alocação e a liberação da memória temporária a cada troca, a função de ordenação pode ser responsável por alocar, no início, e liberar, no fim, esta área temporária, e passar seu endereço como parâmetro adicional para a função `troca`. Vamos adotar aqui a função que faz a troca caractere a caractere, para simplificar.

Podemos então escrever o código da nossa função de ordenação genérica. Falta, no entanto, um último detalhe. As funções auxiliares `acessa` e `troca` são realmente genéricas, e independem da informação efetivamente armazenada no vetor. Porém, a função de comparação deve ser especializada para cada tipo de informação, conforme ilustramos antes. A assinatura dessa função é genérica, mas a sua implementação deve, naturalmente, levar em conta a informação armazenada para que a comparação tenha sentido. Por-

tanto, para generalizar a implementação da função de ordenação, não podemos chamar uma função de comparação específica. A solução é passar, via parâmetro, qual função *callback* de comparação deve ser chamada. A função de comparação tem a assinatura:

```
int compara (void*, void*);
```

Com isso, a assinatura da função genérica de ordenação, recebendo a *callback* como parâmetro, passa a ser:

```
void bolha_gen (int n, void* v, int tam, int(*cmp)(void*,void*));
```

onde *cmp* representa a variável do tipo ponteiro para a função de comparação. Em C, o nome de uma função representa o ponteiro da função. Esse ponteiro, passado como parâmetro, possibilita chamar a função indiretamente.

Agora sim, podemos escrever nossa função de ordenação genérica:

```
/* Ordenação bolha (genérica) */
void bolha_gen (int n, void* v, int tam, int(*cmp)(void*,void*))
{
    for (int i=n-1; i>0; --i) {
        int fez_troca = 0;
        for (int j=0; j<i; ++j) {
            void* p1 = acessa(v,j,tam);
            void* p2 = acessa(v,j+1,tam);
            if (cmp(p1,p2)) {
                troca(p1,p2,tam);
                fez_troca = 1;
            }
        }
        if (fez_troca == 0) /* não houve troca */
            return;
    }
}
```

Esse código genérico pode ser usado para ordenar vetores com qualquer informação. Para exemplificar, vamos usá-lo para ordenar um vetor de números reais. Para isso, temos que escrever o código da função que faz a comparação, agora especializada para números reais:

```
static int compara_reais (void* a, void* b)
{
    float* p1 = (float*) a;
    float* p2 = (float*) b;
    if ((*p1) > (*p2))
        return 1;
    else
        return 0;
}
```

Podemos, então, chamar a função para ordenar um vetor v de n números reais:

```
...
bolha_gen(n, v, sizeof(float), compara_reais);
...
```

11.4 Ordenação rápida

Assim como o algoritmo anterior, o algoritmo de ordenação rápida (*quick sort*), que iremos discutir agora, procura resolver o problema da ordenação por partes. No entanto, enquanto o algoritmo de ordenação bolha coloca em sua posição (no final do vetor) o maior elemento, a ordenação rápida faz isso com um elemento arbitrário x , chamado de *pivô*. Por exemplo, podemos escolher como pivô o primeiro elemento do vetor e posicioná-lo em sua posição correta numa primeira passada.

Suponha que esse elemento, x , deva ocupar a posição i do vetor, de acordo com a ordenação, isto é, que esta seja a sua posição definitiva no vetor. Sem ordenar o vetor completamente, esse fato pode ser reconhecido quando todos os elementos $v[0], \dots, v[i-1]$ são menores que x e todos os elementos $v[i+1], \dots, v[n-1]$ são maiores que x . Supondo que x já esteja na sua posição correta, com índice i , há dois problemas menores para serem resolvidos: ordenar os (sub) vetores formados por $v[0], \dots, v[i-1]$ e por $v[i+1], \dots, v[n-1]$. Esses subproblemas são resolvidos (recursivamente) de forma semelhante, cada vez com vetores menores, e o processo continua até que os vetores que devem ser ordenados tenham zero ou um elemento, caso em que sua ordenação já está correta.

A grande vantagem deste algoritmo é que ele pode ser muito eficiente. O melhor caso ocorre quando o elemento pivô representa o valor mediano do conjunto dos elementos do vetor. Se isso acontece, após o deslocamento do pivô para a sua posição, restarão dois subvetores para serem ordenados, ambos com o número de elementos reduzido à metade, em relação ao vetor original. Pode-se mostrar que, neste melhor caso, o esforço computacional do algoritmo é proporcional a $n \log(n)$, e dizemos que o algoritmo é $O(n \log(n))$, com desempenho muito superior ao $O(n^2)$ apresentado pelo algoritmo de ordenação bolha. Infelizmente, não temos como garantir que o pivô seja o valor mediano. No pior caso, o pivô pode sempre ser, por exemplo, o maior elemento, e recaímos no algoritmo de ordenação bolha ($O(n^2)$). No entanto, é possível mostrar que a ordenação rápida ainda apresenta, no caso médio, um desempenho $O(n \log(n))$.

A versão do algoritmo de ordenação rápida que vamos apresentar aqui usa o primeiro elemento como pivô: $x = v[0]$. O processo compara os elementos $v[1], v[2], \dots$ até encontrar um elemento $v[a] > x$. Então, a partir do final do vetor, compara os elementos $v[n-1], v[n-2], \dots$ até encontrar um elemento $v[b] \leq x$. Nesse ponto, $v[a]$ e $v[b]$ são trocados e o procedimento continua, para cima a partir de $v[a+1]$ e para baixo a partir de $v[b-1]$. Em algum momento, o procedimento termina, porque os pontos de busca se encontrarão ($b < a$). Nesse momento, a posição correta de x está definida e os valores $v[0]$ e $v[b]$ são trocados.

Vamos discutir detalhadamente este procedimento de partição do vetor com um exemplo. Considere o vetor v de inteiros a seguir:

No início, temos $x = v[0]$, $a=1$ e $b=n-1$, onde n representa o número de elementos do vetor. As localizações dos índices a e b no vetor são mostradas a seguir:

37	25	44	33	64	86	18	12	92	48

 a b *é que*

Nosso objetivo é incrementar o índice a enquanto $v[a] > x$ e decrementar o índice b enquanto $v[b] \leq x$. Considerando nosso exemplo, podemos incrementar a até que $v[a]$ seja 44 e decrementar b até que $v[b]$ seja 12. Neste ponto, trocamos $v[a]$ com $v[b]$, incrementando a e decrementando b :

37	25	44	33	64	86	18	12	92	48

 a b

37	25	12	33	64	86	18	44	92	48

 a b

O procedimento então continua, incrementando a e decrementando b , até que $v[a]$ seja 64 ($v[b]$ já é menor que o pivô e, portanto, não é decrementado). Novamente, os valores de $v[a]$ e $v[b]$ são trocados, a é incrementado, b é decrementado.

37	25	12	33	64	86	18	44	92	48

 a b

37	25	12	33	18	86	64	44	92	48

 ab

O procedimento continua (no caso, decrementando b) até que os índices se cruzam ($b < a$):

37	25	12	33	18	86	64	44	92	48

 b a

Neste ponto, temos a localização do pivô dada pelo índice b : $v[b]$ deve ocupar a primeira posição do vetor e o pivô ocupar a posição de $v[b]$:

18	25	12	33	<u>37</u>	86	64	44	92	48

 x

Com isso, temos a partição do vetor concluída. Todos os elementos à esquerda de x são menores que o pivô e todos os elementos à direita são maiores. Se ordenarmos o subvetor à esquerda e o subvetor à direita, teremos o vetor inteiro ordenado. Para ordenar os subvetores, usamos o mesmo procedimento, particionando os elementos. O processo se repete, recursivamente, até que os subvetores restantes tenham zero ou um elemento. A partição do subvetor à esquerda é ilustrada a seguir:

18 25 12 33

a b

18 25 12 33

a b

18 25 12 33

b a

12 18 25 33

Devemos observar que, para deslocar o índice **a** para a direita, fizemos o teste:

```
while (a < n && v[a] <= x)
```

enquanto, para deslocar o índice **b** para a esquerda, fizemos apenas:

```
while (v[b] > x)
```

O teste adicional no deslocamento para a direita é necessário porque o pivô pode ser o elemento de maior valor, nunca ocorrendo a situação **v[a]** <= **x**, o que nos faria acessar posições além dos limites do vetor. No deslocamento para a esquerda, um teste adicional do tipo **b** >= 0 não é necessário, pois, na nossa implementação, **v[0]** é o pivô, impedindo que **b** assuma valores negativos (teremos, pelo menos, **v[0] == x**).

11.5 Algoritmos genéricos da biblioteca padrão

A biblioteca padrão de C disponibiliza algoritmos genéricos de busca e ordenação em vetores. Nesta seção, vamos inicialmente analisar o algoritmo de ordenação e, em seguida, o de busca.

11.5.1 Algoritmo genérico de ordenação

A ordenação rápida é o algoritmo de ordenação mais utilizado no desenvolvimento de aplicações. Por sua grande utilidade, a biblioteca padrão de C disponibiliza, via interface `stdlib.h`, uma função que ordena vetores usando esse algoritmo. A função disponibilizada pela biblioteca independe do tipo de informação armazenada no vetor. A implementação dessa função genérica segue os princípios discutidos na implementação do algoritmo de ordenação bolha genérico da seção anterior. O protótipo da função `qsort` disponibilizada pela biblioteca pode ser simplificada por:

```
void qsort (void *v, int n, int tam,
            int (*cmp)(const void*, const void*));
```

Os parâmetros de entrada dessa função são:

- **v**: ponteiro para o primeiro elemento do vetor que se deseja ordenar. Como não se sabe, *a priori*, o tipo dos elementos do vetor, temos um ponteiro genérico `void*`.
- **n**: número de elementos do vetor.
- **tam**: tamanho, em bytes, de cada elemento do vetor.
- **cmp**: ponteiro para a função responsável por comparar dois elementos do vetor.

Como dissemos, o nome de uma função representa o ponteiro da função. Esse ponteiro pode ser passado como parâmetro, possibilitando chamar a função indiretamente. Como era de se esperar, a biblioteca não sabe comparar dois elementos do vetor (ela desconhece o tipo desses elementos). Fica a cargo do cliente da função de ordenação escrever a função de comparação. Essa função de comparação tem que ter o seguinte protótipo:

```
int _nome_ (const void*, const void*);
```

O parâmetro `cmp` recebido pela função `qsort` é um ponteiro para uma função com esse protótipo. Assim, para usar a função de ordenação da biblioteca, temos que escrever uma função que receba dois ponteiros genéricos, `const void*`, os quais representam ponteiros para os dois elementos que se deseja comparar. O modificador de tipo `const` aparece no protótipo apenas para garantir que essa função não modificará os valores dos elementos (devem ser tratados como valores constantes). Essa função deve ter como valor de retorno < 0 , 0 , ou > 0 , dependendo de se o primeiro elemento for menor, igual ou maior que o segundo, respectivamente, de acordo com o critério de ordenação adotado. Isto é, a função deve retornar um número negativo se a ordem relativa entre os dois elementos já estiver correta.

Para ilustrar a utilização da função `qsort`, vamos considerar alguns exemplos. O código a seguir ilustra a utilização da função para ordenar valores reais. Nesse caso, os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `float`.

```
/* Ilustra uso do algoritmo qsort() para ordenar vetores */
#include <stdio.h>
#include <stdlib.h>
/* função de comparação de reais */
static int comp_reais (const void* p1, const void* p2)
{
    /* converte ponteiros genéricos para ponteiros de float */
    float *f1 = (float*)p1;
    float *f2 = (float*)p2;

    /* dados os ponteiros de float, faz a comparação */
    if (*f1 < *f2) return -1;
    else if (*f1 > *f2) return 1;
    else return 0;
}

/* programa que faz a ordenação de um vetor */
int main (void)
{
    float v[8] = {25.6, 48.3, 37.7, 12.1, 57.4, 86.6, 33.3, 92.8};

    qsort(v, 8, sizeof(float), comp_reais);

    printf("Vetor ordenado: ");
    for (int i=0; i<8; ++i)
        printf("%g ", v[i]);
    printf("\n");

    return 0;
}
```

Vamos agora considerar que temos um vetor de *alunos* e que desejamos ordenar o vetor usando o nome do aluno como chave de comparação. A estrutura que representa um aluno pode ser dada por:

```

typedef struct aluno Aluno;
struct aluno {
    char nome[81];
    char mat[8];
    char turma;
    char email[41];
    float nota;
};

```

Vamos analisar duas situações. Na primeira, consideraremos a existência de um vetor de estrutura (por exemplo, `Aluno vet[N];`). Neste caso, cada elemento do vetor é do tipo `Aluno` e os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `Aluno`. Essa função de comparação pode ser dada por:

```

/* Função de comparação: elemento é do tipo Aluno */
static int comp_alunos (const void* p1, const void* p2)
{
    /* converte ponteiros genéricos para ponteiros de Aluno */
    Aluno *a1 = (Aluno*)p1;
    Aluno *a2 = (Aluno*)p2;

    /* dados os ponteiros de Aluno, faz a comparação */
    return strcmp(a1->nome,a2->nome);
}

```

Um programa para testar a ordenação pode ser:

```

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

int main (void)
{
    Aluno v[4] = {
        {"Nome C", "mat C", 'C', "c@mail.com"}, 
        {"Nome B", "mat B", 'B', "b@mail.com"}, 
        {"Nome D", "mat D", 'D', "d@mail.com"}, 
        {"Nome A", "mat A", 'A', "a@mail.com"}, 
    };

    qsort(v,4,sizeof(Aluno),comp_alunos);

    for (int i=0; i<4; ++i)
        printf("%s\n", v[i].nome);

    return 0;
}

```

Numa segunda situação, podemos considerar que temos um vetor de ponteiros para a estrutura `aluno` (por exemplo, `Aluno* vet[N];`). Agora, cada elemento do vetor é

um ponteiro para o tipo `Aluno` e a função de comparação tem que tratar uma indireção a mais. Aqui, os dois ponteiros genéricos passados para a função de comparação representam *ponteiros de ponteiros para Aluno*.

```
/* Função de comparação: elemento é do tipo Aluno */
static int comp_alunos_ptr (const void* p1, const void* p2)
{
    /* converte p/ ponteiros de ponteiros de Aluno */
    Aluno **a1 = (Aluno**)p1;
    Aluno **a2 = (Aluno**)p2;

    /* dados os ponteiros de ponteiro de Aluno, faz a comparação */
    return strcmp((*a1)->nome, (*a2)->nome);
}
```

Uma função `main` para testar a ordenação pode ser:

```
int main (void)
{
    Aluno v[4] = {
        {"Nome C", "mat C", 'C', "c@mail.com"}, 
        {"Nome B", "mat B", 'B', "b@mail.com"}, 
        {"Nome D", "mat D", 'D', "d@mail.com"}, 
        {"Nome A", "mat A", 'A', "a@mail.com"}};
    Alunos* p[4] = {&v[0], &v[1], &v[2], &v[3]};
    qsort(p, 4, sizeof(Aluno*), comp_alunos_ptr);
    for (int i=0; i<4; ++i)
        printf("%s\n", p[i]->nome);
    return 0;
}
```

Como último exemplo de função de comparação, vamos considerar uma ordenação com critério de desempate. Queremos agora ordenar nosso vetor de ponteiros para estrutura de aluno em ordem decrescente de nota, sendo que alunos com notas iguais devem ser dispostos em ordem alfabética. Portanto, ordem alfabética representa o critério de desempate da ordenação por nota. A codificação desta função de comparação não apresenta dificuldade e por ser ilustrada por:

```
/* Função de comparação: elemento é do tipo Aluno */
static int comp_notas_nomes (const void* p1, const void* p2)
{
    Aluno **a1 = (Aluno**)p1;
    Aluno **a2 = (Aluno**)p2;

    if ((*a1)->nota > (*a2)->nota) return -1;
    else if ((*a1)->nota < (*a2)->nota) return 1;
    else return strcmp((*a1)->nome, (*a2)->nome);
}
```

11.5.2 Algoritmo genérico de busca

A biblioteca padrão de C disponibiliza, também via interface `stdlib.h`, uma função que faz a busca binária de um elemento num vetor. A função disponibilizada pela biblioteca independe do tipo de informação armazenada no vetor. A implementação dessa função genérica segue os mesmos princípios discutidos anteriormente. O protótipo da função da biblioteca padrão que faz a busca binária é:

```
void* bsearch (void* info, void *v, int n, int tam,
               int (*cmp)(const void*, const void*));
```

Se o elemento for encontrado no vetor, a função tem como valor de retorno o endereço do elemento no vetor; caso o elemento não seja encontrado, o valor de retorno é `NULL`. De modo análogo à função `qsort`, os parâmetros de entrada dessa função são:

- `info`: ponteiro para a informação que se deseja buscar no vetor – representa a chave de busca.
- `v`: ponteiro para o primeiro elemento do vetor no qual a busca será feita. Os elementos do vetor têm que estar ordenados, segundo o critério de ordenação adotado pela função de comparação descrita a seguir.
- `n`: número de elementos do vetor.
- `tam`: tamanho, em bytes, de cada elemento do vetor.
- `cmp`: ponteiro para a função responsável por comparar a informação que se busca com um elemento do vetor. O primeiro parâmetro dessa função é sempre o endereço da informação que se busca (o mesmo endereço passado para a função `bsearch`) e o segundo é um ponteiro para um dos elementos do vetor. O critério de comparação adotado por essa função deve ser compatível com o critério de ordenação do vetor. Essa função deve ter como valor de retorno `< 0`, `0` ou `> 0`, dependendo de se a informação que se busca for menor, igual ou maior que a informação armazenada no elemento, respectivamente.

Para ilustrar a utilização da função `bsearch`, vamos inicialmente considerar um vetor de valores inteiros em ordem crescente. Nesse caso, os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `int`:

```
/* função de comparação de inteiros */
static int comp_int (const void* p1, const void* p2)
{
    /* converte ponteiros genéricos para ponteiros de int */
    int *info = (int*)p1;
    int *elem = (int*)p2;

    /* dados os ponteiros de int, faz a comparação */
    if (*info < *elem) return -1;
    else if (*info > *elem) return 1;
    else return 0;
}
```

```

/* programa que faz a busca em um vetor */
int main (void)
{
    int v[8] = {12,25,33,37,48,57,86,92};
    int e = 57;      /* informação que se deseja buscar */
    int* p;

    p = (int*)bsearch(&e,v,8,sizeof(int),comp_int);
    if (p == NULL)
        printf("Elemento não encontrado.\n");
    else
        printf("Elemento encontrado no indice: %d\n", (int)(p-v));
    return 0;
}

```

Devemos notar que o índice do elemento, se encontrado no vetor, pode ser extraído subtraindo-se o ponteiro do elemento do ponteiro do primeiro elemento ($p - v$). Essa aritmética de ponteiros é válida, aqui, porque podemos garantir que ambos os ponteiros armazenam endereços de memória de um mesmo vetor. A diferença entre os ponteiros representa a “distância” em que os elementos estão armazenados na memória.

Vamos agora considerar uma busca num vetor de ponteiros para alunos. A estrutura que representa um aluno pode ser dada por:

```

struct aluno {
    char nome[81];
    char mat[8];
    char turma;
    char email[41];
};

typedef struct aluno Aluno;

```

Considerando que o vetor está ordenado segundo os nomes dos alunos, podemos buscar a ocorrência de determinado aluno passando para a função de busca um *nome* e o vetor. A função de comparação, então, receberá dois ponteiros que referenciam tipos distintos: um ponteiro para uma cadeia de caracteres e um ponteiro para um elemento do vetor (no caso, será um ponteiro para ponteiro de aluno, ou seja, um *Aluno ***).

```

/* Função de comparação: char* e Aluno** */
static int comp_alunos (const void* p1, const void* p2)
{
    /* converte ponteiros genéricos para ponteiros específicos */
    char* s = (char*)p1;
    Aluno **pa = (Aluno**)p2;

    /* faz a comparação */
    return strcmp(s,(*pa)->nome);
}

```

Conforme observamos, o tipo de informação a ser buscada nem sempre é igual ao tipo do elemento; para dados complexos, em geral não é. A informação buscada geralmente representa um campo da estrutura armazenada no vetor (ou da estrutura apontada por elementos do vetor).

Exercícios

1. Considere um tipo que representa um funcionário de uma empresa, definido pela estrutura a seguir:

```
typedef struct Funcionario Funcionario;
struct Funcionario {
    char nome[81]; /* nome do funcionário */
    float valor_hora; /* valor da hora de trabalho em Reais */
    int horas_mes; /* horas trabalhadas em um mês */
};
```

Escreva uma função que faça uma busca binária em um vetor de ponteiros para o tipo `Funcionario`, cujos elementos estão em ordem alfabética dos nomes dos funcionários. Essa função deve receber como parâmetros o número de funcionários, o vetor e o nome do funcionário que se deseja buscar, e deve ter como valor de retorno um ponteiro para o registro do funcionário procurado. Se não houver um funcionário com o nome procurado, a função deve retornar `NULL`. A função deve obedecer ao seguinte protótipo:

```
Funcionario* busca (int n, Funcionario** v, char* nome);
```

2. Considere um cadastro de pessoas que armazena nome e data de nascimento (representada por três inteiros: *dia*, *mês*, *ano*). O cadastro é representado por um vetor de ponteiros para a estrutura a seguir, ordenado em ordem crescente de data de nascimento:

```
typedef struct Pessoa Pessoa;
struct Pessoa {
    char nome[81]; /* nome */
    int dia, mes, ano; /* data de nascimento */
};
```

Aplicando a técnica de busca binária (usando ou não a função `bsearch` da biblioteca padrão), implemente uma função que verifique se alguém nasceu numa determinada data. A função deve receber o número de elementos no vetor, o vetor de ponteiros e a data que se deseja fazer a busca. Caso seja encontrada uma pessoa que tenha nascido na data especificada, a função deve retornar o ponteiro para a estrutura `Pessoa` correspondente. Se não existir uma pessoa no cadastro que tenha nascido na data especificada, a função deve retornar `NULL`. A função deve obedecer ao seguinte protótipo:

```
Pessoa* busca (int n, Pessoa** v, int dia, int mes, int ano);
```

3. Considere um vetor de inteiros cujos valores estão armazenados em ordem crescente. Usando como base o algoritmo de busca binária, escreva uma função que, dado

Se o vetor é um valor inteiro x , retorne o elemento do vetor que possui valor mais próximo de x . Caso x seja equidistante de dois elementos do vetor, sua função deve retornar o valor do menor deles. Por exemplo, considerando o vetor $\{3, 7, 10, 14, 16\}$, sua função deve retornar os valores indicados a seguir para os diferentes casos listados:

Valor de x	Valor retornado
11	10
5	3
14	14
13	14

Considerando que o valor de x é maior ou igual ao primeiro elemento do vetor e menor ou igual ao último elemento do vetor. Sua função deve ter o seguinte protótipo:

```
int mais_proximo (int n, int* vet, int x);
```

(vet *contém, x é o valor buscado)

4. Considere a estrutura Aluno a seguir:

```
typedef struct aluno Aluno;
struct aluno {
    char nome[81];
    float nota;
};
```

Usando ordenação bolha, implemente uma função que ordene um vetor de alunos em ordem alfabética dos nomes, seguindo o protótipo:

```
void ordem_alfabetica (int n, Aluno* pv);
```

Escreva uma função *main* para testar sua função.

5. Considerando novamente o tipo *Funcionario* definido na Questão 1, escreva uma função que imprima os nomes dos cinco funcionários com maiores salários, sendo o salário de um funcionário igual ao número de horas trabalhadas no mês vezes o valor de sua hora de trabalho. Assuma que não existem dois funcionários com o mesmo salário. A função deve receber como parâmetros o número total de funcionários e um vetor que armazena ponteiros para estruturas do tipo *Funcionario*, de acordo com o protótipo definido a seguir:

```
void imprime_marajas (int n, Funcionario** vet);
```

Dica: Sua função pode primeiro ordenar o vetor em ordem decrescente de salários, usando alguma das técnicas de ordenação apresentadas.

6. Considere um tipo que represente um aluno de um curso, definido pela estrutura a seguir:

```
typedef struct aluno Aluno;
struct aluno {
    char mat[8]; /* matrícula do aluno */
    char nome[81]; /* nome do aluno */
    float cr; /* coeficiente de rendimento */
};
```

Considere ainda um vetor de estrutura `Aluno` com um número ímpar de elementos. Implemente uma função que imprima na tela o nome de um aluno *mediano*. O aluno mediano é definido da seguinte forma: dos alunos restantes, metade tem coeficiente de rendimento menor ou igual ao do aluno mediano e metade tem coeficiente de rendimento maior ou igual ao do aluno mediano. A função deve obedecer ao seguinte protótipo:

```
void mediano (int n, Aluno* v);
```

Dica: Ordene o vetor em ordem de coeficiente de rendimento e considere o aluno no meio do vetor.

7. Altere o algoritmo de ordenação rápida discutido, aplicado a um vetor de inteiros, para que use como pivô o valor mediano entre o primeiro, o último e o elemento do meio do vetor. Neste caso, se o vetor tiver apenas dois elementos, deve-se colocá-los em ordem sem chamar a função recursivamente. Para facilitar a adaptação do algoritmo apresentado, pode-se usar a seguinte estratégia: ache o valor mediano entre os três elementos, troque o valor do elemento mediano com o primeiro elemento (se o primeiro já não for o mediano) e execute o mesmo procedimento usando o primeiro elemento como pivô da partição. Esta estratégia para escolher o pivô tende a melhorar substancialmente o desempenho do algoritmo na prática, pois, em geral, resulta em partições mais equilibradas.
8. Considere que uma empresa quer ter acesso a lista dos aniversários de seus funcionários. Cada funcionário, neste vetor, será identificado pelo nome e dia/mês de aniversário.

```
typedef struct funcionario Funcionario;
struct funcionario {
    char nome[81];
    int dia, mes;
};
```

- (a) Implemente uma função que receba um vetor de ponteiros para o tipo apresentado e ordene o vetor em ordem crescente de dia de aniversário. Ou seja, use como critério de ordenação o mês; para valores de mês iguais, use o dia. Se dois ou mais funcionários fizerem aniversário na mesma data, use a ordem alfabética do nome como critério de desempate. Sua função de ordenação deve

obrigatoriamente fazer uso da função `qsort` da biblioteca padrão. Sua função deve receber o número de funcionários e o vetor de ponteiros como parâmetros.

- (b) Implemente uma função que receba um vetor de ponteiros para os funcionários, ordenado conforme o critério citado, e exiba na tela os nomes dos aniversariantes do mês especificado. Sua função deve receber como parâmetros o número de funcionários, o vetor de ponteiros (já ordenado) e o mês que se deseja consultar. Utilize o algoritmo de busca binária para procurar um funcionário que faça aniversário no mês especificado. A partir desta posição, percorra os elementos vizinhos no vetor para achar o início e o fim dos funcionários que fazem aniversário no mesmo mês, e exiba seus nomes na tela, em ordem do dia do aniversário.
- (c) Escreva uma função *main* para testar suas funções. A função deve definir um vetor de ponteiros para funcionários com os dados numa ordem qualquer, chamar a função de ordenação e, então, chamar a função para exibir os aniversariantes de uma data.

Parte II

Estruturas Dinâmicas