capítulo 4

Métodos de busca

> *Enquanto vamos em busca do incerto, perdemos o seguro.*
>
> Titus Maccius Plautus

Encontrar algo em um mar de informações torna-se cada vez mais difícil. Assim, conhecer as técnicas apropriadas para cada caso é imprescindível no desenvolvimento de algoritmos de busca.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- conhecer os métodos de busca mais utilizados;
- entender a importância do arranjo dos dados (principalmente a ordenação) e sua implicação nos métodos de busca;
- conhecer e praticar a implementação dos principais algoritmos de busca.

Para começar

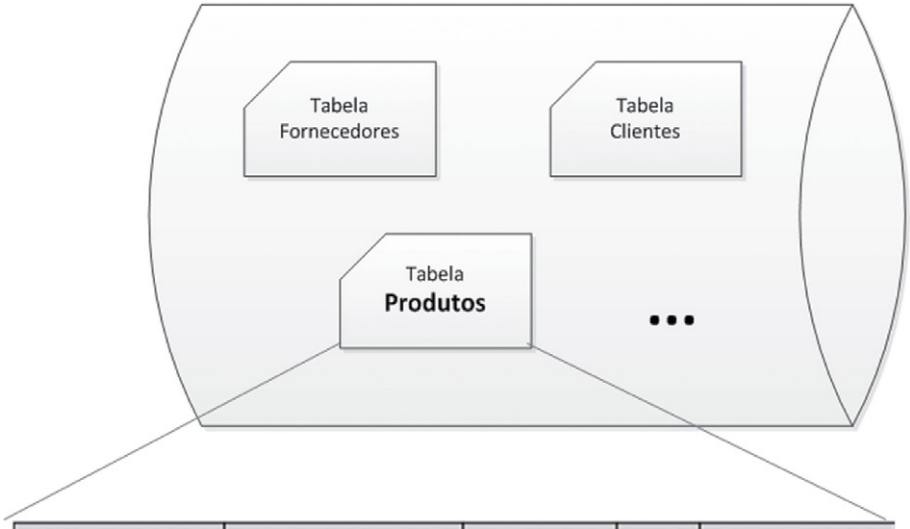
Todos nós fazemos buscas, consciente e inconscientemente, todos os dias. Fazemos isso quando, por exemplo, procuramos um restaurante onde possamos almoçar, ou uma roupa para usar ao longo do dia, ou determinado livro em uma ou mais estantes de uma biblioteca.

Também fazemos buscas mentais, internamente, em nosso cérebro, ao procurar um rosto conhecido ou o significado de uma palavra que aprendemos anteriormente.

Atualmente, guardar informações não é mais um problema. A capacidade das máquinas cresceu muito, e serviços de *cloud computing*, por exemplo, têm auxiliado ainda mais nessa tarefa. O grande problema está no processo de recuperação dessas informações, principalmente quando precisamos recuperar uma informação de qualidade, algo que possamos realmente utilizar da forma como planejamos ou necessitamos.

Vamos tomar como exemplo uma grande base de dados de produtos de uma loja de autopeças. Suponha que o cliente chegue ao balcão e pergunte se a loja tem determinada peça para vender. Geralmente, o nome ou o código do produto é utilizado como o índice ou chave de busca, dado (campo) que compõe o registro do produto e é utilizado para distingui-lo dos demais; portanto, serve de indicador para a realização da busca na base de dados. A

Figura 4.1 ilustra a organização da base de dados, com a tabela de produtos, seu registro e o campo de cada um.



Código Produto	Nome do Produto	Montadora	Qtd.	Preço Unitário
3245	Filtro de óleo	VW	4	R$ 56,00
4566	Carburador	Honda	6	R$ 356,00
4567	Carburador	Ford	9	R$ 458,00
6789	Pistão	Fiat	2	R$ 654,00
6790	Pistão	GM	10	R$ 450,00
...	...	...	...	...

FIGURA 4.1: Estrutura genérica de uma base de dados: tabelas, registros e campos.

Na Figura 4.1, as colunas "Código do produto", "Nome do produto", "Montadora", "Qtd. e Preço unitário" são os campos. Cada linha corresponde às informações sobre um produto. A esse conjunto de informações damos o nome de *registro*. Assim, cada linha é um registro. O conjunto de todos os registros dos produtos forma a tabela de produtos. Essa tabela e outras ficam alocadas dentro de uma base de dados.

Atenção

Você verá com detalhes as questões relativas à estruturação das informações em bases de dados nas disciplinas específicas da área de Banco de Dados. Por ora, os conceitos de *tabela*, *registro* e *campo* serão suficientes para as implementações dos algoritmos de busca.

Voltando ao balcão: quando o cliente pergunta sobre determinado produto – por exemplo, "carburador da Ford" –, o funcionário vai até um sistema, busca o nome ou código do produto e retorna ao cliente, dizendo: "Sim, temos! E o preço é R$ 458,00".

Aqui surgem algumas dúvidas: como o funcionário encontrou o produto especificado? Quanto tempo se passou até que o funcionário o encontrasse e respondesse ao cliente?

Possivelmente, o funcionário acessou a base de dados através de um sistema automatizado que contava com uma ferramenta de busca. Assim, inseriu a informação passada pelo cliente em um local específico no sistema, e o produto, com suas respectivas informações, foi encontrado e exibido na tela do terminal de consulta.

Isso faz com que perguntemos: como o sistema encontrou essa informação? Possivelmente, utilizando um algoritmo de busca.

E como pode ser implementado um algoritmo de busca?

Bem, pensemos em um conjunto de informações mais simples, como um vetor de 10 elementos (números inteiros), conforme ilustrado na Figura 4.2.

4	78	12	3	65	21	34	77	98	11

FIGURA 4.2: Um vetor com 10 números inteiros.

Se alguém lhe perguntasse se no vetor da Figura 4.2 existe o valor 34, você certamente responderia que sim. E poderia ser detalhista e completar dizendo que ele está na sétima posição.

Você conseguiria descrever o método que utilizou para encontrar o valor 34 no vetor?

A resposta mais frequente é: "Corri os olhos pelo vetor, da esquerda para a direita, verificando cada um dos valores, até encontrar o valor procurado: 34."

Você conseguiria descrever essa operação de busca no formato de um algoritmo (em português estruturado)? Vamos lá, tente!

Dica

Para desenvolver esse algoritmo, pense na sequência de passos que você, até de forma inconsciente, realizou para encontrar determinado número no vetor. Lembre-se de que cada detalhe é muito importante!

E então? Conseguiu?

Possivelmente, seguindo os passos descritos, você deve ter desenvolvido algo semelhante ao representado no algoritmo a seguir.

Algoritmo (em português estruturado)

Supondo um vetor *v[]*, um total de elementos, *n,* e o valor a ser encontrado, *k*, teríamos:

- percorra o vetor (da primeira à última posição);
- para cada elemento do vetor, verifique se é igual ao valor a ser encontrado;
- se for igual, retorne com a posição do vetor onde se encontra esse valor;
- caso contrário, continue procurando.

Se você chegou ao final do vetor e não encontrou o valor, retorne o valor -1.

Note que o valor -1 é retornado como uma espécie de sinalizador. Isso é feito porque não existe no vetor uma posição igual a -1. Assim, fica implícito que, ao retornar o -1, o valor não foi encontrado.

Esse método de busca é o mais simples a ser implementado, mas existem muitos outros. Neste capítulo vamos ver apenas os mais simples e utilizados: busca sequencial, busca binária e busca por tabela de espalhamento (*hashing table*). Preparado? Vamos lá!

Conhecendo a teoria para programar

Vamos iniciar com a forma mais simples de buscar um elemento em um arranjo de dados: procurando-o ao longo de todos os seus elementos – no caso de um vetor, verificando todas as suas posições. Esse método é conhecido como *busca sequencial*.

Busca sequencial

Como mencionado, para exemplificar essa forma de busca, vamos utilizar uma estrutura de dados do tipo vetor. Para tanto, o algoritmo de busca sequencial vai percorrer todo o vetor, do início ao fim, comparando o valor que se busca com cada elemento do vetor.

Se durante a busca a comparação for positiva, ou seja, se o valor for encontrado, a posição do vetor será retornada. Caso a comparação chegue ao

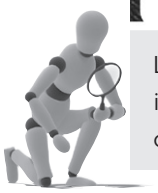
final do vetor, isso significa que o valor buscado não foi encontrado. Como uma estratégia nesse caso, o valor retornado geralmente é -1, pois o vetor não tem uma posição com esse valor.

Nós fizemos a descrição desse algoritmo em português estruturado. Vamos agora implementá-lo em linguagem C. Uma possibilidade de implementação pode ser a descrita no Código 4.1. Para tanto, faremos a implementação de uma função que receba como parâmetros um vetor *v[]* de números inteiros, um número inteiro *n,* que indica a quantidade de elementos nesse vetor, e por fim outro valor inteiro *k*, que seria o valor que se busca encontrar no vetor *v[]*. A função retorna a posição do elemento do vetor *v[]* igual a *k* (se existir) ou -1, se o valor não for encontrado. Vamos ao código!

Código 4.1

```
int buscaSeq (int v[], int n, int k)
{
       int i;
       for (i=0; i<n; i++) {                  // percorre todo o vetor v[]
         if (v[i]==k)  return i;              // se encontra um valor igual, retorna a posição i
       }
       return -1;                             // caso o valor não seja encontrado, retorna o valor -1
}
```

Para testar a função *buscaSeq()* apresentada no Código 4.1, poderíamos utilizar o vetor da Figura 4.2. Assim, se passássemos como parâmetro o vetor dessa figura, o número 10 representando o total de elementos do vetor e o número 34 como chave de busca, teríamos como retorno o valor 6, ou seja, a posição do número 34 no vetor.



Atenção

Lembre-se de que nos vetores em algumas linguagens, como C e Java, as posições iniciam em zero. Por esse motivo, o número 34, estando na sétima posição, retorna o valor 6.

Caso o valor que se pretenda procurar for o número 71, o valor de retorno da função *buscaSeq()* será -1, pois esse valor não existe no vetor da Figura 4.2. Nesse caso, ao analisar a função de busca sequencial, veremos que as 10 posições do vetor serão visitadas e comparadas com o valor-chave.

Se, por acaso, os valores do vetor estivessem ordenados (do menor para o maior), existiria uma forma de busca sequencial mais eficiente? Sim, pois poderíamos percorrer o vetor, do menor para o maior, até o ponto em que o valor buscado fosse menor que o valor do *i-ésimo* elemento do vetor. Nesse caso, quando o elemento buscado fosse menor que o valor do *i-ésimo* elemento, a função poderia retornar o valor -1 (não existe valor igual à chave de busca *k*).

Vamos tomar novamente o vetor da Figura 4.2 e organizar seus elementos em ordem crescente. O resultado pode ser visto na Figura 4.3.

Uma possibilidade de otimização do Código 4.1 (busca sequencial), supondo que os elementos do vetor estão organizados em ordem crescente, poderia ser o que é apresentado no Código 4.2.

3	4	11	12	21	34	65	77	78	98

FIGURA 4.3: Vetor de 10 elementos inteiros organizados em ordem crescente.

Código 4.2

```
int buscaSeq (int v[], int n, int k)
{
       int i;
       for (i=0; i<n; i++) {        // percorre todo o vetor v[]
         if (v[i]==k)   return i;   // se encontra um valor igual, retorna a posição i
         if (v[i]>k) return -1;     // se o i-ésimo element do vetor for maior que a chave k, retorna -1 (não existe igual).
       }
       return -1;                   // caso o valor não seja encontrado, retorna o valor -1
}
```

Observe que, no caso do vetor da Figura 4.3, se fizéssemos a busca tentando encontrar o valor 15 utilizando a função do Código 4.2, o algoritmo

faria no máximo 5 comparações, pois, quando chegasse ao quinto elemento (valor 21), verificaria que esse elemento é maior do que a chave de busca e, assim, retornaria -1 (o valor 15 não existe no vetor).

Utilizando a mesma chave de busca, porém com o algoritmo do Código 4.1, o total de buscas seria igual a 10.

Conceito

Mesmo mostrando maior eficiência para alguns casos, a busca sequencial, com seus elementos estando ou não ordenados, pode, na melhor das hipóteses, realizar apenas uma comparação, e na pior delas, *n* comparações, onde *n* é igual ao número total de elementos no vetor.

Se os elementos do vetor não estiverem ordenados, não teremos outro método de busca, a não ser o sequencial. Mas, se os elementos do vetor estiverem ordenados, existem métodos mais eficientes do que a busca sequencial ou linear. Um deles é conhecido como *busca binária*. Vamos conhecê-lo?

Busca binária

Esse método só funciona se os elementos da estrutura de dados (no caso, vetor) estiverem ordenados. Ele utiliza a ideia de "dividir para conquistar": a divisão acontece sempre comparando o valor que se busca com o elemento localizado no meio do vetor. Ao se realizar essa comparação, há três possibilidades: (1) o valor desse elemento central é igual à chave de busca; (2) o valor é menor do que a chave de busca; ou (3) o valor é maior do que a chave de busca.

No primeiro caso, basta retornar à posição central e encontramos o valor. No segundo caso, se os elementos do vetor estiverem em ordem crescente e o elemento central for menor do que a chave de busca, isso significa que o valor dessa chave, se existir, estará na parte superior do vetor e a parte inferior (da metade ao início) será totalmente ignorada. O mesmo princípio se aplica ao terceiro caso, mas desta vez o elemento que se busca, se existir, estará na parte inferior do vetor e a parte superior (da metade até o final do vetor) será totalmente ignorada.

Depois dessa primeira divisão, o novo vetor passará a conter apenas os elementos da parte selecionada, e uma nova busca será feita com o elemento central. Ocorrerá uma nova divisão e assim sucessivamente, até que se encontre o valor procurado ou se retorne um valor de divisão igual a zero

(ou seja, não existem mais elementos no processo de divisão do vetor). Nesse caso, não existe valor igual à chave de busca.

Uma possibilidade de implementação desse método de busca pode ser o do Código 4.3, a seguir.

Código 4.3

```
int buscaBin (int v[], int n, int k)
{
      int inicio=0;                          // início do vetor ou de parte do vetor (primeiro elemento)
      int fim=n-1;                           // final do vetor ou de parte do vetor (último elemento)
      int centro;                            // posição central do vetor
      while (inicio <= fim) {                // enquanto existir elementos no vetor...
        centro=inicio+(fim-inicio)/2;        // recebe a posição central do vetor
        if (k == v[centro]) return centro;   // caso (1) – encontrou o valor
        else if (k > v[centro])              // caso (2) – o valor do elemento central é menor que k
                  inicio=centro+1;           // nesse caso (2) o novo vetor passa a ser a parte superior.
             else                            // caso (3) – o valor do elemento central é maior que k
                  fim=centro-1;              // nesse caso (3) o novo vetor passa a ser a parte inferior.
      }
      return -1;                             // caso o valor não seja encontrado, retorna o valor -1
}
```

Ao executar essa função passando como parâmetro o vetor da Figura 4.3, com 10 elementos e chave de busca igual a 34 (k = 34), teríamos esta sequência de execução:

```
Vetor v[] = {3,4,11,12,21,34,65,77,78,98}
inicio=0/fim=9/centro=4
```

Depois da primeira comparação, identifica-se que o valor 34 pode estar na parte superior do vetor inicial. Assim, um novo *looping* é iniciado no comando *while*:

```
Vetor v[] = {~~3,4,11,12,21~~,34,65,77,78,98}
inicio=5/fim=9/centro=7
```

Após a segunda comparação, verifica-se que o valor 34 pode estar na parte inferior desse novo vetor. Assim, um novo *looping* é iniciado:

```
Vetor v[] = {3,4,11,12,21,34,65,77,78,98}
inicio=5/fim=6/centro=5
```

Nessa nova comparação, o valor 34 é encontrado na posição 5 (6º elemento) do vetor e, assim, retornado com sucesso.

Esquematicamente, teríamos a sequência ilustrada na Figura 4.4.

Você deve ter notado que, a cada *looping*, o vetor é dividido ao meio. Iniciou com 10 elementos, depois foi reduzido para 5, depois para 2. Nesse processo, reduziu-se gradativamente o espaço de busca até encontrar o elemento procurado ou chegar a um vetor vazio (caso em que não existe o valor procurado no vetor), e a velocidade de busca foi acelerada exponencialmente em relação ao método de busca sequencial.

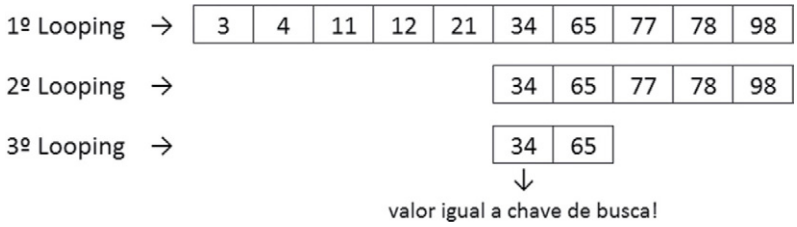


FIGURA 4.4: Sequência de execuções do algoritmo de busca binária.

Além disso, você deve ter notado que, a cada iteração do algoritmo de busca binária, um novo vetor era sugerido para realizar essa busca. Em virtude disso, poderíamos realizar a implementação do algoritmo de busca binária de forma recursiva, passando, a cada execução, um novo vetor (parte) como parâmetro. Embora a implementação não recursiva seja mais eficiente e mais adequada para esse tipo de algoritmo, a implementação recursiva é mais elegante e sucinta. O Código 4.4 ilustra uma possível implementação da função de busca binária recursiva.

Faremos uma pequena modificação na chamada da função: em vez de passar a quantidade de elementos do vetor, passaremos a posição inicial e a final.

Código 4.4

```
int buscaBinRec (int v[], int inicio, int fim, int k)
{
      int centro;                                  // posição central do vetor
      while (inicio <= fim) {                      // enquanto existir elementos no vetor...
        centro=inicio+(fim-inicio)/2;              // recebe a posição central do vetor
        if (k == v[centro]) return centro;         // caso (1) – encontrou o valor
        else if (k > v[centro])                    // caso (2) – o valor do elemento central é menor que k
              return buscaBinRec(v, centro+1, fim, k);   // caso (2) o novo vetor passa a ser a parte superior
            else                                   // caso (3) – o valor do elemento central é maior que k
              return buscaBinRec(v, inicio, centro-1, k);   // caso (3) o novo vetor é a parte inferior.
      }
      return -1;                                   // caso o valor não seja encontrado, retorna o valor -1
}
```

Agora que você já conhece os dois métodos de busca mais básicos, é hora de partir para uma implementação mais sofisticada: a busca utilizando tabelas de dispersão ou tabelas de espalhamento (em inglês, *hashing tables*).

Tabelas de dispersão

As tabelas de dispersão, ou tabelas *hashing*, consistem no armazenamento de cada elemento em determinado endereço, calculado a partir da aplicação de uma função sobre a chave de busca. Matematicamente, teríamos o seguinte:

$$e = f(c)$$

onde *e* é o endereço, *c* é a chave de busca, e *f(c)* é a função que tem como entrada a chave de busca.

Assim, o processo de pesquisa sobre elementos organizados dessa forma é similar a um acesso direto ao elemento pesquisado. A Figura 4.5 ilustra essa busca utilizando tabelas de dispersão.

A eficiência da pesquisa nesse tipo de organização dos dados depende da função de cálculo do endereço (*f(c)*). A função ideal seria aquela que pudesse gerar um endereço diferente para cada elemento da tabela (chave de busca). Entretanto, isso é praticamente inviável, principalmente pela dinâmica de atualização dos dados e aumento da quantidade de elementos na tabela.

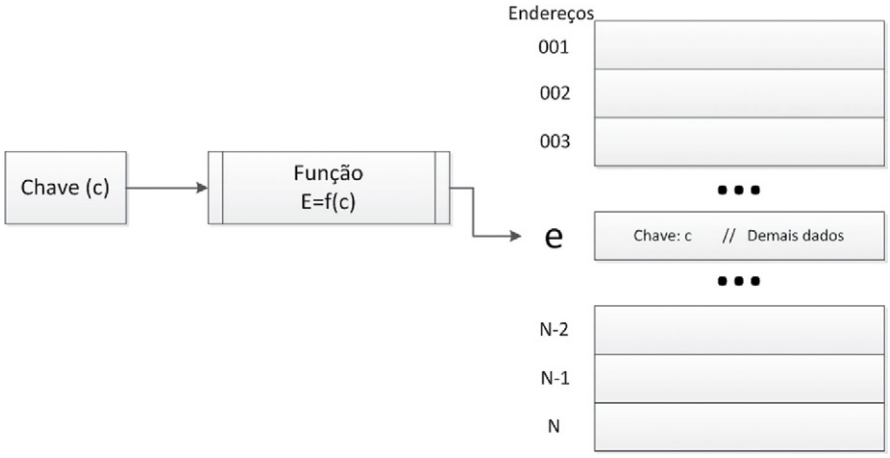


FIGURA 4.5: Processo de busca e organização dos dados em uma tabela de dispersão (*hashing table*).

Vamos entender esse conceito trabalhando com algo concreto. Supondo os dados constantes no vetor da Figura 4.6, acrescido da posição de cada elemento, teríamos:

Endereço →	0	1	2	3	4	5	6	7	8	9
Valores →	30	11	72	83	44	65	86	17	58	9

FIGURA 4.6: Um vetor de 10 elementos inteiros organizados com sua posição.

Pensando nessa tabela e na organização dos dados, poderíamos sugerir uma função de dispersão que tivesse como entrada o valor do elemento (chave de busca), e a função retornaria o endereço no vetor. Na amostra de dados, uma função possível seria:

$$e(c) = (c \; mod \; 10)$$

onde *mod* corresponde ao resto da divisão inteira da chave *c* por *10* (número de elementos da tabela). Podemos verificar se a função de dispersão proposta é eficiente testando alguns valores (Tabela 4.1).

Como pode ser visto na tabela acima, a função de dispersão proposta é efetiva para o acesso direto aos valores constantes na tabela da Figura 4.6.

Entretanto, o problema acontece quando existe possibilidade de atualização de valores. Por exemplo, vamos imaginar que queiramos mudar o valor da posição 0 de 30 para 91. A função não é mais eficiente, pois, se a aplicarmos utilizando como chave o valor 91, teremos o endereço igual a 1. Como podemos observar, o endereço 1 já está ocupado pelo valor 11.

TABELA 4.1: Aplicação dos valores das chaves de busca sobre a função de dispersão para cálculo do endereço

Chave (c)	cmod 10	e	Acesso direto?
30	30 mod 10	0	Ok!→ v[0] = 30
44	44 mod 10	4	Ok!→ v[4] = 44
9	9 mod 10	9	Ok!→ v[9] = 9
72	72 mod 10	2	Ok!→ v[2] = 72

A esse fenômeno de dois valores distintos resultarem em um mesmo endereço, quando aplicados a uma função de dispersão, chamamos de *colisão*. Trata-se de um fenômeno comum que pode ser tratado de diversas formas.

Para entender melhor o processo de colisão no mundo real, considere o seguinte: se você possui um *smartphone*, certamente tem um aplicativo de gerenciamento de contatos. Nesse aplicativo, existem várias maneiras de acessar os contatos; uma delas é escolher a letra inicial do nome. Quando você escolhe determinada letra, são exibidos todos os nomes que começam com ela. Essa é uma forma de entendermos uma tabela de dispersão. A letra inicial do nome é a entrada de uma função, que separa todos os nomes que iniciam com aquela mesma letra. Depois disso, a busca pelo nome desejado fica bem mais rápida. A Figura 4.7 ilustra essa implementação e o controle das colisões.

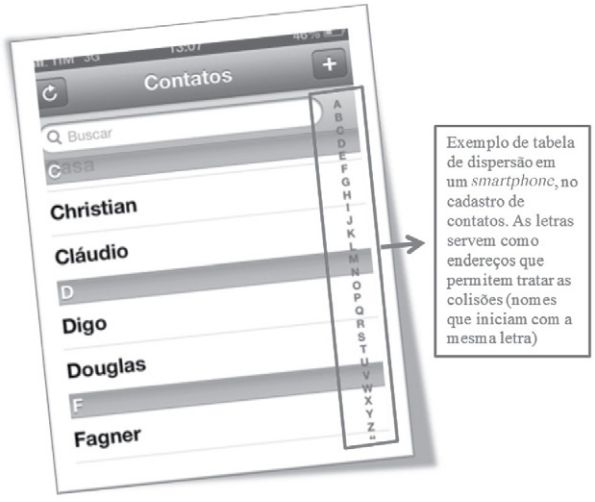


FIGURA 4.7: Tabela de dispersão e o tratamento das colisões em listas de contatos em *smartphones*.

Como já foi dito, existem várias formas de trabalhar as colisões. Uma das mais utilizadas é a implementação de listas encadeadas, estruturas de dados que utilizam alocação dinâmica na memória, a partir do endereço base. Por exemplo, pensando no vetor da Figura 4.6, poderíamos alocar outros números tomando como base uma dispersão de 10 áreas diferentes e utilizando a mesma função para cálculo do endereço. As colisões seriam tratadas com alocação dinâmica dos elementos por listas encadeadas. A Figura 4.8 ilustra essa ideia.

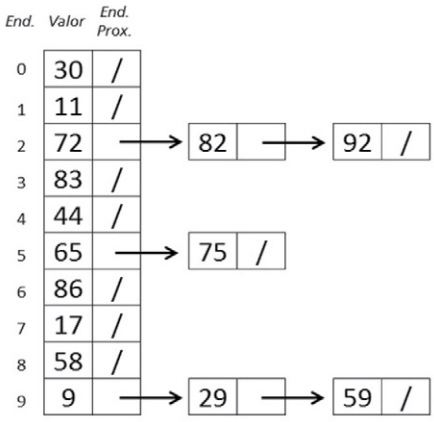


FIGURA 4.8: Tabela de dispersão e o tratamento das colisões com listas encadeadas.

Note que, quando se acessa determinado endereço (dado pela função de dispersão), pode existir um número, nenhum ou uma lista deles. Geralmente, essa lista encadeada de números é mantida em ordem crescente, o que pode facilitar o processo de busca e otimizar o processo.

Atenção

A implementação de estruturas de dados por meio de listas encadeadas será vista no Capítulo 7. Fique atento!

Você deve saber que existem muitos outros métodos e algoritmos de busca. Mais adiante, veremos, algoritmos de busca em listas encadeadas e árvores.

Vamos programar

Para ampliar a abrangência do conteúdo apresentado, teremos, nas seções seguintes, implementações nas linguagens de programação Java e Phyton.

Java

Código 4.1

```
public static int buscaSeq (int v[], int n, int k)
{
       int i;
       for (i=0; i<n; i++) {              // percorre todo o vetor v[]
           if (v[i]==k)   return i;       // se encontra um valor igual, retorna a posição i
       }
       return -1;                         // caso o valor não seja encontrado, retorna o valor -1
}
```

Código 4.2

```
public static int buscaSeq (int v[], int n, int k)
{
int i;
       for (i=0; i<n; i++) {       // percorre todo o vetor v[]
         if (v[i]==k)   return i;  // se encontra um valor igual, retorna a posição i
         if (v[i]>k) return -1;    // se o i-ésimo elemento do vetor > chave k, retorna -1 (não existe igual).
       }
       return -1;                  // caso o valor não seja encontrado, retorna o valor -1
}
```

Código 4.3

```
public static int buscaBin (int v[], int n, int k)
{
  int inicio=0;                              // início do vetor ou de parte do vetor (primeiro elemento)
  int fim=n-1;                               // final do vetor ou de parte do vetor (último elemento)
  int centro;                                // posição central do vetor
  while (inicio <= fim) {                    // enquanto existir elementos no vetor...
        centro=inicio+(fim-inicio)/2;        // recebe a posição central do vetor
        if (k == v[centro]) return centro; // caso (1) – encontrou o valor
        else if (k > v[centro])              // caso (2) – o valor do elemento central é menor que k
                 inicio=centro+1;            // nesse caso (2) o novo vetor passa a ser a parte superior.
              else                           // caso (3) – o valor do elemento central é maior que k
                 fim=centro-1;               // nesse caso (3) o novo vetor passa a ser a parte inferior.
   }
  return -1;                                 // caso o valor não seja encontrado, retorna o valor -1
}
```

Código 4.4

```
public static int buscaBinRec (int v[], int inicio, int fim, int k)
{
 int centro;                                 // posição central do vetor
 while (inicio <= fim) {                     // enquanto existir elementos no vetor...
   centro=inicio+(fim-inicio)/2;             // recebe a posição central do vetor
   if (k == v[centro]) return centro;        // caso (1) – encontrou o valor
   else if (k > v[centro])                          // caso (2) – o valor do elemento central é menor que k
         return buscaBinRec(v, centro+1, fim, k);   // caso (2) o novo vetor passa a ser a parte superior.
        else                                 // caso (3) – o valor do elemento central é maior que k
         return buscaBinRec(v, inicio, centro-1, k); // caso (3) o novo vetor é a parte inferior.
 }
 return -1;                                  // caso o valor não seja encontrado, retorna o valor -1
}
```

OU pode ser usado a seguinte função:

```
System.out.println(Arrays.binarySearch(Array, Valor));
```

Python

Código 4.1

```
def buscaSeq(v,n,k):
 i=0
 while i<n:
   if v[i]==k:
     return i
   i+=1
 return -1
```

Código 4.2

```
def buscaSeq(v,n,k):
 i=0
 while i<n:
   if v[i]==k:
     return i
   if v[i]>k:
     return -1
   i+=1
 return -1
```

Código 4.3

```
def buscaBin (v,n,k):
 inicio=0
 fim=n-1
 centro=0
 while inicio<=fim:
   centro=inicio+(fim-inicio)/2
   if k==v[centro]:
     return centro
   elif k> v[centro]:
     inicio=centro+1
   else:
     fim=centro-1
 return -1
```

Código 4.4

```
def buscaBinRec(v,inicio,fim,k):
 centro=0
 while inicio<=fim:
   centro=inicio+(fim-inicio)/2
   if k==v[centro]:
     return centro
   elif k>v[centro]:
     return buscaBinRec(v,centro+1,fim,k)
   else:
     return buscaBinRec(v,inicio,centro-1,k)
 return -1;
```

Para fixar

1. No código de busca binária abaixo estão faltando algumas informações (“##”). Com base no que você aprendeu, preencha corretamente e faça os comentários pertinentes ao que está acontecendo em cada linha do algoritmo.

```
int buscaBinaria(int v[], int n, int k) {
      int inicio = ##;
      int fim = ##;
      while (inicio ## fim-1) {
         centro = (inico+fim)/2;
         if (v[centro] ## k) return centro;
         else if (v[centro] ## k) inicio=centro;
              else fim = centro;
         }
         return ##;
      }
```

Vamos lá, você consegue!

Para saber mais

Você poderá encontrar mais informações sobre algoritmos de busca em *Algoritmos: teoria e prática* (Cormen *et al*).

Navegar é preciso

Na internet, existem muitos *sites* e páginas que tratam o tema de recursão em computação.

Um deles é o site da Wikipedia (http://pt.wikipedia.org/wiki/Algoritmo_de_busca), que define a expressão *algoritmo de busca*. Além de interessante, apresenta uma história sobre o assunto e alguns links sobre temas relacionados, como *pesquisa binária*, *pesquisa sequencial*, *pesquisa por interpolação*, *busca com retrocesso*, entre outros. Existem vários objetos de aprendizagem para a demonstração das implementações de algoritmos de busca. Selecionamos dois deles para demonstrar a busca sequencial e a busca binária:

- busca sequencial
  http://www.cse.ust.hk/learning_objects/array/linearsearch/index.html
- busca binária
  http://www.cse.ust.hk/learning_objects/array/binarysearch/index.html

Exercícios

1. Na sua opinião, que problemas poderiam surgir da utilização do código de função de busca a seguir?

```
int Busca (int vet[], int n, int k){
   int i=0;
   while (v[i] < k && i < n) ++i;
   return i;
}
```

2. Mostre que a função buscaBin, a seguir, funciona corretamente.

```
int buscaBin (int vet[], int n, int k){
   int inicio=0, fim=n, centro;
   while (inicio < fim)
     centro=(inicio+fim)/2;
     if (v[centro] < k) inicio=centro+1;
     else fim=centro;
   }
   return fim;
}
```

3. Escreva uma função recursiva de busca binária em que sejam passados apenas dois valores como parâmetros: o vetor *(v[])* e a chave de busca *(k)*.

Glossário

Cloud Computing: computação na nuvem/em nuvem. Trata-se de um modelo de computação (processamento, armazenamento e aplicações) utilizado em qualquer lugar e em qualquer equipamento, por meio da internet.

Referências bibliográficas

CORMEN, T. H. C. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.

MOKARZEL, F. C.; SOMA, N. Y. *Introdução à ciência da computação*. Rio de Janeiro: Elsevier, 2008.

SCHILDT, H. *C, completo e total*. 3. ed. São Paulo: Makron Books, 1996.

TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.

VELOSO, P. et al. *Estrutura de dados*. 4. ed. Rio de Janeiro: Campus, 1986.

O QUE VEM DEPOIS...

Agora que você já conhece os principais algoritmos de busca e entende a importância da ordenação de dados, chegou a hora de se aprofundar na forma como esses dados podem ser ordenados. No Capítulo 5 vamos ver os principais algoritmos de ordenação e suas principais aplicações. Preparado? Então, bons estudos!