16. Busca

W. Celes e J. L. Rangel

Neste capítulo, discutiremos diferentes estratégias para efetuarmos a busca de um elemento num determinado conjunto de dados. A operação de busca é encontrada com muita freqüência em aplicações computacionais, sendo portanto importante estudar estratégias distintas para efetuá-la. Por exemplo, um programa de controle de estoque pode buscar, dado um código numérico ou um nome, a descrição e as características de um determinado produto. Se temos um grande número de produtos cadastrados, o método para efetuar a busca deve ser eficiente, caso contrário a busca pode ser muito demorada, inviabilizando sua utilização.

Neste capítulo, estudaremos algumas estratégias de busca. Inicialmente, consideraremos que temos nossos dados armazenados em um vetor e discutiremos os algoritmos de busca que podemos empregar. A seguir, discutiremos a utilização de árvores binárias de busca, que são estruturas de árvores projetadas para darem suporte a operações de busca de forma eficiente. No próximo capítulo, discutiremos as estruturas conhecidas como tabelas de dispersão (*hash*) que podem, como veremos, realizar buscas de forma extremamente eficiente, fazendo uso de espaço de memória adicional.

16.1. Busca em Vetor

Nesta seção, apresentaremos os algoritmos de busca em vetor. Dado um vetor `vet` com `n` elementos, desejamos saber se um determinado elemento `elem` está ou não presente no vetor.

Busca linear

A forma mais simples de fazermos uma busca num vetor consiste em percorrermos 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-se 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 elem)
{
   int i;

   for (i=0; i<n; i++) {
      if (elem == vet[i])
         return i;       /* elemento encontrado */
   }

   /* percorreu todo o vetor e não encontrou elemento */
   return -1;
}
```

Esse algoritmo de busca é extremamente simples, mas pode ser 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 que um 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 com relação ao tamanho do problema – chamamos esse algoritmo de *busca linear*.

Em geral, usamos a notação "Big-O" para expressarmos como a complexidade de um algoritmo varia com o tamanho do problema. Assim, nesse caso em que o tempo computacional varia linearmente com o tamanho do problema, dizemos que trata-se de um algoritmo de ordem linear e expressamos isto escrevendo *O(n)*.

No melhor caso, se dermos sorte do elemento procurado ocupar a primeira posição do vetor, o algoritmo acima necessitaria de apenas uma única comparação. Esse fato, no entanto, não pode ser usado para fazermos uma análise de desempenho do algoritmo, pois o melhor caso representa um situação muito particular.

Além do pior caso, devemos 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(k n)*, onde *k* é uma constante, é igual a *O(n)*.

Em diversas aplicações reais, precisamos de algoritmos de busca mais eficientes. Seria possível melhorarmos a eficiência do algoritmo de busca mostrado acima? 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 se acharmos um elemento maior, pois se o elemento que buscamos estivesse presente ele precederia um elemento maior na ordem do vetor.

O código abaixo 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 (i=0; i<n; i++) {
      if (elem == vet[i])
         return i;               /* elemento encontrado */
      else if (elem < vet[i])
         return -1;              /* interrompe busca */
   }

   /* percorreu todo o vetor e não encontrou elemento */
   return -1;
}
```

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.

Busca binária

No caso dos elementos do vetor estarem em ordem, podemos aplicar um algoritmo mais eficiente para realizarmos a busca. Trata-se do algoritmo de *busca binária*. A idéia 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. Este 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 inicio consideramos todo o vetor */
   int ini = 0;
   int fim = n-1;
   int meio;

   /* enquanto a parte restante for maior que zero */
   while (ini <= fim) {
      meio = (ini + fim) / 2;
      if (elem < vet[meio])
         fim = meio - 1;          /* ajusta posicão final */
      else if (elem > vet[meio])
         ini = meio + 1;          /* ajusta posicão inicial */
      else
         return meio;             /* elemento encontrado */
   }

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

O desempenho desse algoritmo é muito superior ao de 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 concluirmos que o elemento não está presente no vetor? A cada repetição, a parte considerada na busca é dividida à metade. A tabela abaixo mostra o tamanho do vetor a cada repetição do laço do algoritmo.

Repetição	Tamanho do problema
1	*n*
2	*n/2*
3	*n/4*
...	...
log n	*1*

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

O algoritmo de busca binária consiste em repetirmos o mesmo procedimento recursivamente, podendo ser naturalmente implementado de forma recursiva. Embora a implementação não recursiva seja mais eficiente e mais adequada para esse algoritmo, a implementação recursiva é mais sucinta e 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, a função de busca 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 busca_bin_rec (int n, int* vet, int elem)
{
   /* testa condição de contorno: parte com tamanho zero */
   if (n <= 0)
      return 0;
   else {
      /* deve buscar o elemento entre os índices 0 e n-1 */
      int meio = (n - 1) / 2;

      if (elem < vet[meio])
         return busca_bin_rec(meio,vet,elem);
      else if (elem > vet[meio])
         return busca_bin_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 num 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 com essa modificação é apresentada abaixo:

```
int busca_bin_rec (int n, int* vet, int elem)
{
   /* testa condição de contorno: parte com tamanho zero */
   if (n <= 0)
      return -1;
   else {
      /* deve buscar o elemento entre os índices 0 e n-1 */
      int meio = (n - 1) / 2;

      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 */
   }
}
```

Algoritmo genérico

A biblioteca padrão de C disponibiliza, via a 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 no capítulo anterior. O protótipo da função de busca binária da biblioteca 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`. Análogo a função `qsort`, apresentada no capítulo anterior, os parâmetros de entrada dessa função são:

- `info`: o ponteiro para a informação que se deseja buscar no vetor – representa a *chave de busca*;
- `v`: o ponteiro para o primeiro elemento do vetor onde a busca será feita. Os elementos do vetor têm que estar ordenados, segundo o critério de ordenação adotado pela função de comparação descrita abaixo.
- `n`: o número de elementos do vetor.
- `tam`: o tamanho, em bytes, de cada elemento do vetor.
- `cmp`: o ponteiro para a função responsável por comparar a informação buscada e um elemento do vetor. O primeiro parâmetro dessa função é sempre o endereço da informação buscada, 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 `-1`, `0`, ou `1`, dependendo se a informação buscada 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. Neste caso, os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `int`.

```
/* Ilustra uso do algoritmo bsearch */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de inteiros */
int comp_int (const void* p1, const void* p2)
{
   /* converte ponteiros genéricos para ponteiros de int */
   int *i1 = (int*)p1;
   int *i2 = (int*)p2;
   /* dados os ponteiros de int, faz a comparação */
   if (*i1 < *i2) return -1;
   else if (*i1 > *i2) 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 nao encontrado.\n");
   else
      printf("Elemento encontrado no indice: %d\n", 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 pois 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 que queremos efetuar 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 um determinado aluno passando para a função de busca um nome e o vetor. A função de comparação então receberá dois ponteiros: 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** */
int comp_alunos (const void* p2, 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).

Devemos finalmente salientar que se tivermos os dados armazenados em uma lista encadeada, só temos a alternativa de implementar um algoritmo de busca linear, mesmo se os elementos estiverem ordenados. Portanto, lista encadeada não é uma boa opção para estruturarmos nossos dados, se desejarmos realizar muitas operações de busca. A estrutura dinâmica apropriada para a realização de busca é a árvore binária de busca que será discutida a seguir.

16.2. Árvore binária de busca

Como vimos, o algoritmo de busca binária apresentado na seção anterior apresenta bom desempenho computacional e deve ser usado quando temos os dados ordenados armazenados num vetor. No entanto, se precisarmos inserir e remover elementos da estrutura, e ao mesmo tempo dar suporte a eficientes funções de busca, a estrutura de vetor (e, conseqüentemente, o uso do algoritmo de busca binária) não se torna adequada. Para inserirmos um novo elemento num vetor ordenado, temos que re-arrumar os elementos no vetor, para abrir espaço para a inserção do novo elemento. Situação análoga ocorre quando removemos um elemento do vetor. Precisamos portanto de uma estrutura dinâmica que dê suporte a operações de busca.

Um dos resultados que apresentamos anteriormente foi o da relação entre o número de nós de uma árvore binária e sua altura. A cada nível, o número (potencial) de nós vai dobrando, de maneira que uma árvore binária de altura *h* pode ter um número de nós dado por:

$$1 + 2 + 2^2 + \ldots + 2^{h-1} + 2^h = 2^{h+1}-1$$

Assim, dizemos que uma árvore binária de altura *h* pode ter no máximo $O(2^h)$ nós, ou, pelo outro lado, que uma árvore binária com *n* nós pode ter uma altura mínima de $O(\log n)$. Essa relação entre o número de nós e a altura mínima da árvore é importante porque se as condições forem favoráveis, podemos alcançar qualquer um dos *n* nós de uma árvore binária a partir da raiz em, no máximo, $O(\log n)$ passos. Se tivéssemos os *n* nós em uma lista linear, o número máximo de passos seria $O(n)$, e, para os valores de *n* encontrados na prática, $\log n$ é muito menor do que *n*.

A altura de uma árvore é, certamente, uma medida do tempo necessário para encontrar um dado nó. No entanto, é importante observar que para acessarmos qualquer nó de maneira eficiente é necessário termos árvores binárias "balanceadas", em que os nós internos têm todos, ou quase todos, o máximo número de filhos, no caso *2*. Lembramos que o número mínimo de nós de uma árvore binária de altura *h* é *h+1*, de forma que a altura máxima de uma árvore com *n* nós é $O(n)$. Esse caso extremo corresponde à árvore "degenerada", em que todos os nós têm apenas *1* filho, com exceção da (única) folha.

As árvores binárias que serão consideradas nesta seção têm uma propriedade fundamental: o valor associado à raiz é sempre maior que o valor associado a qualquer nó da sub-árvore à esquerda (*sae*), e é sempre menor que o valor associado a qualquer nó da sub-árvore à direita (*sad*). Essa propriedade garante que, quando a árvore é percorrida em ordem simétrica (*sae* - *raiz* - *sad*), os valores são encontrados em ordem crescente.

Uma variação possível permite que haja repetição de valores na árvore: o valor associado à raiz é sempre maior que o valor associado a qualquer nó da *sae*, e é sempre menor ou igual ao valor associado a qualquer nó da *sad*. Nesse caso, como a repetição de valores é permitida, quando a árvore é percorrida em ordem simétrica, os valores são encontrados em ordem não decrescente.

Usando essa propriedade de ordem, a busca de um valor em uma árvore pode ser simplificada. Para procurar um valor numa árvore, comparamos o valor que buscamos com o valor associado à raiz. Em caso de igualdade, o valor foi encontrado; se o valor dado for menor que o valor associado à raiz, a busca continua na *sae*; caso contrário, se o valor associado à raiz for menor, a busca continua na *sad*. Por essa razão, estas árvores são freqüentemente chamadas de *árvores binárias de busca*.

Naturalmente, a ordem a que fizemos referência acima é dependente da aplicação. Se a informação a ser armazenada em cada nó da árvore for um número inteiro podemos usar o habitual operador relacional *menor que* ("<"). Porém, se tivermos que considerar casos em que a informação é mais complexa, já vimos que uma função de comparação deve ser definida pelo programador, especificamente para cada caso.

Operações em árvores binárias de busca

Para exemplificar a implementação de operações em árvores binárias de busca, vamos considerar o caso em que a informação associada a um nó é um número inteiro, e não vamos considerar a possibilidade de repetição de valores associados aos nós da árvore. A Figura 16.1 ilustra uma árvore de busca de valores inteiros.

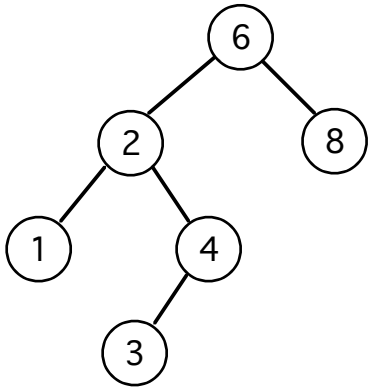

Figura 16.1: Exemplo de árvore binária de busca.

O tipo da árvore binária pode então ser dado por:

```
struct arv {
   int info;
   struct arv* esq;
   struct arv* dir;
};

typedef struct arv Arv;
```

A árvore é representada pelo ponteiro para o nó raiz. A árvore vazia é inicializada atribuindo-se `NULL` a variável que representa a árvore. Uma função simples de inicialização é mostrada abaixo:

```
Arv* init (void)
{
   return NULL;
}
```

Uma vez construída uma árvore de busca, podemos imprimir os valores da árvore em ordem crescente percorrendo os nós em ordem simétrica:

```
void imprime (Arv* a)
{
   if (a != NULL) {
      imprime(a->esq);
      printf("%d\n",a->info);
      imprime(a->dir);
   }
}
```

Essas são funções análogas às vistas para árvores binárias comuns, pois não exploram a propriedade de ordenação das árvores de busca. No entanto, as operações que nos interessa analisar em detalhes são:

- `busca`: função que busca um elemento na árvore;
- `insere`: função que insere um novo elemento na árvore;
- `retira`: função que retira um elemento da árvore.

Operação de busca

A operação para buscar um elemento na árvore explora a propriedade de ordenação da árvore, tendo um desempenho computacional proporcional a sua altura (*O(log n)* para o caso de árvore balanceada). Uma implementação da função de busca é dada por:

```
Arv* busca (Arv* r, int v)
{
   if (r == NULL) return NULL;
   else if (r->info > v) return busca (r->esq, v);
   else if (r->info < v) return busca (r->dir, v);
   else return r;
}
```

Operação de inserção

A operação de inserção adiciona um elemento na árvore na posição correta para que a propriedade fundamental seja mantida. Para inserir um valor `v` em uma árvore usamos sua estrutura recursiva, e a ordenação especificada na propriedade fundamental. Se a (sub-) árvore é vazia, deve ser substituída por uma árvore cujo único nó (o nó raiz) contém o valor `v`. Se a árvore não é vazia, comparamos `v` com o valor na raiz da árvore, e inserimos `v` na *sae* ou na *sad*, conforme o resultado da comparação. A função abaixo ilustra a implementação dessa operação. A função tem como valor de retorno o eventual novo nó raiz da (sub-) árvore.

```
Arv* insere (Arv* a, int v)
{
   if (a==NULL) {
      a = (Arv*)malloc(sizeof(Arv));
      a->info = v;
      a->esq = a->dir = NULL;
   }
   else if (v < a->info)
      a->esq = insere(a->esq,v);
   else  /* v < a->info */
      a->dir = insere(a->dir,v);
   return a;
}
```

Operação de remoção

Outra operação a ser analisada é a que permite retirar um determinado elemento da árvore. Essa operação é um pouco mais complexa que a de inserção. Existem três situações possíveis. A primeira, e mais simples, é quando se deseja retirar um elemento que é folha da árvore (isto é, um elemento que não tem filhos). Neste caso, basta retirar o elemento da árvore e atualizar o pai, pois seu filho não existe mais.

A segunda situação, ainda simples, acontece quando o nó a ser retirado possui um único filho. Para retirar esse elemento é necessário antes acertar o ponteiro do pai, "pulando" o nó: o único neto passa a ser filho direto. A Figura 16.2 ilustra esse procedimento.

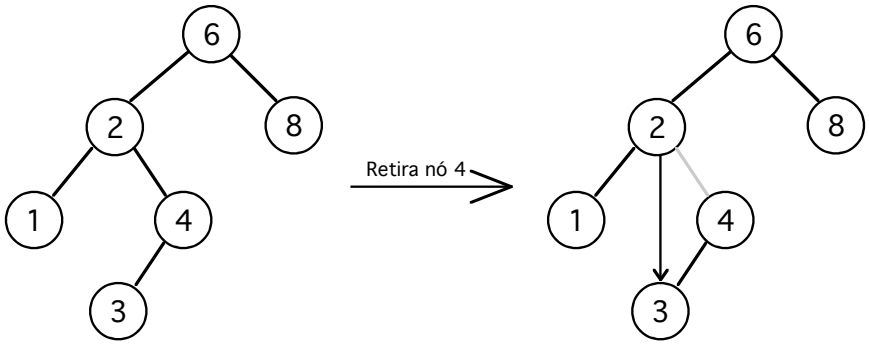

Figura 16.2: Retirada de um elemento com um único filho.

O caso complicado ocorre quando o nó a ser retirado tem dois filhos. Para poder retirar esse nó da árvore, devemos proceder da seguinte forma:

a) encontramos o elemento que precede o elemento a ser retirado na ordenação. Isto equivale a encontrar o elemento mais à direita da sub-árvore à esquerda;
b) trocamos a informação do nó a ser retirado com a informação do nó encontrado;
c) retiramos o nó encontrado (que agora contém a informação do nó que se deseja retirar). Observa-se que retirar o nó mais à direita é trivial, pois esse é um nó folha ou um nó com um único filho (no caso, o filho da direita nunca existe).

O procedimento descrito acima deve ser seguido para não haver violação da ordenação da árvore. Observamos que, análogo ao que foi feito com o nó mais à direita da sub-árvore à esquerda, pode ser feito com o nó mais à esquerda da sub-árvore à direita (que é o nó que segue o nó a ser retirado na ordenação).

A Figura 16.3 exemplifica a retirada de um nó com dois filhos. Na figura é mostrada a estratégia de retirar o elemento que precede o elemento a ser retirado na ordenação.

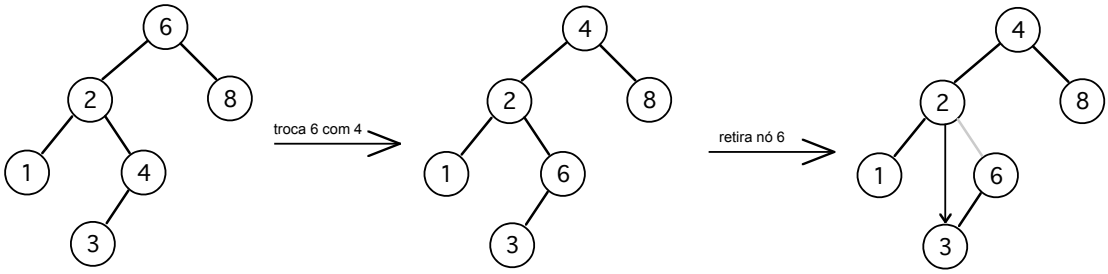

Figura 16.3: Exemplo da operação para retirar o elemento com informação igual a 6.

O código abaixo ilustra a implementação da função para retirar um elemento da árvore binária de busca. A função tem como valor de retorno a eventual nova raiz da (sub-) árvore.

```
Arv* retira (Arv* r, int v)
{
   if (r == NULL)
      return NULL;
   else if (r->info > v)
      r->esq = retira(r->esq, v);
   else if (r->info < v)
      r->dir = retira(r->dir, v);
   else {          /* achou o elemento */
      if (r->esq == NULL && r->dir == NULL) { /* elemento sem filhos */
         free (r);
         r = NULL;
      }
      else if (r->esq == NULL) {        /* só tem filho à direita */
         Arv* t = r;
         r = r->dir;
         free (t);
      }
      else if (r->dir == NULL) {        /* só tem filho à esquerda */
         Arv* t = r;
         r = r->esq;
         free (t);
      }
      else {                            /* tem os dois filhos */
         Arv* pai = r;
         Arv* f = r->esq;
         while (f->dir != NULL) {
            pai = f;
            f = f->dir;
         }
         r->info = f->info;            /* troca as informações */
         f->info = v;
         r->esq = retira(r->esq,v);
      }
   }
   return r;
}
```

Exercício: Escreva um programa que utilize as funções de árvore binária de busca mostradas acima.

Árvores balanceadas

É fácil prever que, após várias operações de inserção/remoção, a árvore tende a ficar desbalanceada, já que essas operações, conforme descritas, não garantem o balanceamento. Em especial, nota-se que a função de remoção favorece uma das sub-árvores (sempre retirando um nó da sub-árvore à esquerda, por exemplo). Uma estratégia que pode ser utilizada para amenizar o problema é intercalar de qual sub-árvore será retirado o nó. No entanto, isso ainda não garante o balanceamento da árvore.

Para que seja possível usar árvores binárias de busca mantendo sempre a altura das árvores no mínimo, ou próximo dele, é necessário um processo de inserção e remoção de nós mais complicado, que mantenha as árvores "balanceadas", ou "equilibradas", tendo as duas sub-árvores de cada nó o mesmo "peso", ou pesos aproximadamente iguais. No caso de um número de nós par, podemos aceitar uma diferença de um nó entre a *sae* (sub-árvore à esquerda) e a *sad* (sub-árvore à direita).

A idéia central de um algoritmo para balancear (equilibrar) uma árvore binária de busca pode ser a seguinte: se tivermos uma árvore com m elementos na sae, e $n \geq m + 2$ elementos na *sad*, podemos tornar a árvore menos desequilibrada movendo o valor da raiz para a *sae*, onde ele se tornará o maior valor, e movendo o menor elemento da *sad* para a raiz. Dessa forma, a árvore continua com os mesmos elementos na mesma ordem. A situação em que a *sad* tem menos elementos que a *sae* é semelhante. Esse processo pode ser repetido até que a diferença entre os números de elementos das duas sub-árvores seja menor ou igual a 1. Naturalmente, o processo deve continuar (recursivamente) com o balanceamento das duas sub-árvores de cada árvore. Um ponto a observar é que remoção do menor (ou maior) elemento de uma árvore é mais simples do que a remoção de um elemento qualquer.

Exercício: Implemente o algoritmo para balanceamento de árvore binária descrito acima.