

Capítulo 16

Árvores

Nos capítulos anteriores, examinamos as estruturas de dados que podem ser chamadas de lineares, como vetores e listas. A importância dessas estruturas é inegável, mas elas não são adequadas para representar dados que devem ser dispostos de maneira hierárquica. Por exemplo, os arquivos (documentos) que criamos num computador são armazenados dentro de uma estrutura hierárquica de diretórios (pastas). Existe um diretório base dentro do qual podemos armazenar diversos subdiretórios e arquivos. Por sua vez, dentro dos subdiretórios, podemos armazenar outros subdiretórios e arquivos e assim por diante, recursivamente.

Neste capítulo, vamos introduzir as *árvore*s, estruturas de dados adequadas para a representação de hierarquias. A forma mais natural de definir uma estrutura de árvore é usando recursividade. Uma árvore é composta por um conjunto de nós. Existe um nó r , denominado raiz, que contém zero ou mais subárvores, cujas (sub)raízes são ligadas diretamente a r . Esses nós raízes das subárvores são ditos filhos do nó pai, r . Nós com filhos são comumente chamados de *nós internos* e nós que não têm filhos são chamados de *folhas* ou *nós externos*. É tradicional desenhar as estruturas de árvores com a raiz para cima e as folhas para baixo. A Figura 16.1 exemplifica a estrutura de uma árvore.

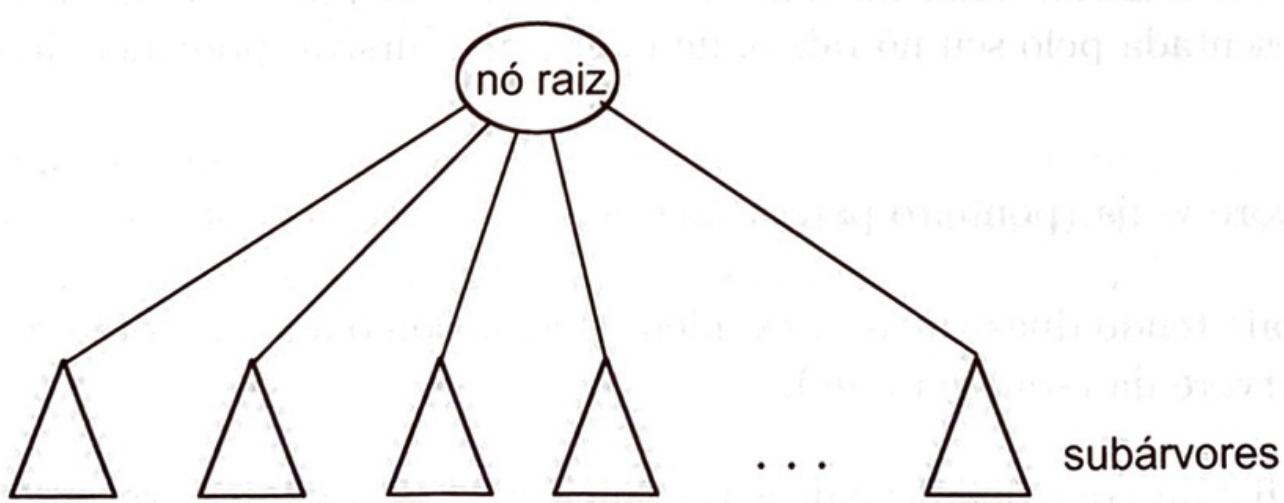


Figura 16.1: Estrutura de árvore.

Observamos que, por adotar essa forma de representação gráfica, não representamos explicitamente a direção dos ponteiros, subentendendo que eles apontam sempre do pai para os filhos.

O número de filhos permitido por nó e as informações armazenadas em cada nó diferenciam os diversos tipos de árvores existentes. Neste capítulo, estudaremos dois

tipos de árvores. Primeiro, examinaremos as árvores binárias, nas quais cada nó tem, no máximo, dois filhos. Depois, examinaremos as estruturas de árvores nas quais o número de filhos é variável. Estruturas recursivas serão usadas como base para o estudo e a implementação das operações com árvores.

16.1 Árvores binárias

Um exemplo de utilização de árvores binárias é a avaliação de expressões. Como trabalhamos com operadores que esperam um ou dois operandos, os nós da árvore para representar uma expressão têm no máximo dois filhos. Nessa árvore, os nós folhas representam operandos e os nós internos, operadores. Uma árvore que representa, por exemplo, a expressão $(3 + 6) * (4 - 1) + 5$ é ilustrada na Figura 16.2.

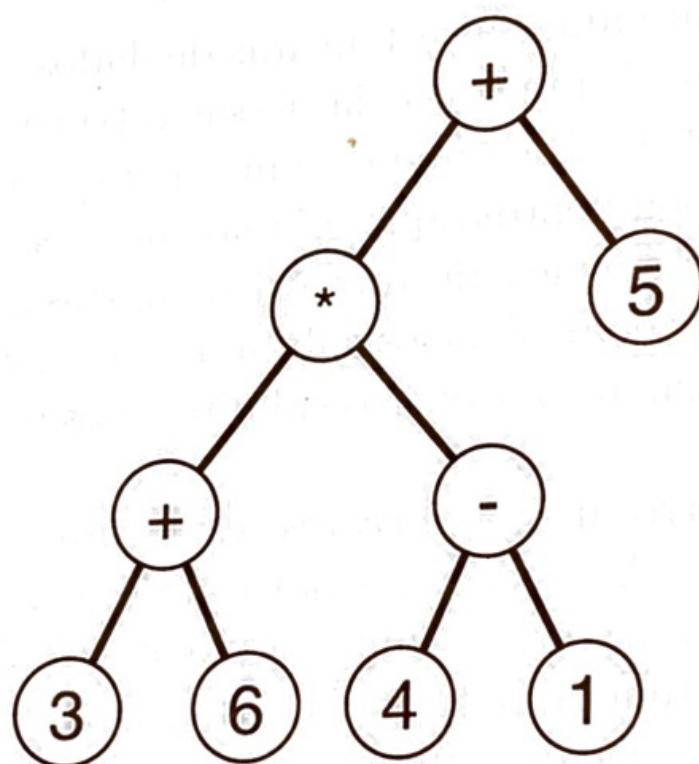


Figura 16.2: Árvore da expressão: $(3 + 6) * (4 - 1) + 5$.

Numa árvore binária, cada nó tem zero, um ou dois filhos. Podemos dizer que a árvore é representada pelo seu nó raiz e, de maneira recursiva, podemos definir a árvore como:

- uma árvore vazia (ponteiro para a raiz nulo); ou
- um nó raiz tendo duas subárvores, identificadas como a subárvore da direita (*sad*) e a subárvore da esquerda (*sae*).

A Figura 16.3 ilustra a definição de árvore binária. Essa definição recursiva será usada na construção de algoritmos e na verificação (informal) da correção e do desempenho deles.

A Figura 16.4 ilustra um exemplo de árvore binária que armazena caracteres. Os nós *a*, *b*, *c*, *d*, *e* e *f* formam uma árvore binária da seguinte maneira: a árvore é composta pelo nó *a*, pela subárvore à esquerda, formada por *b* e *d*, e pela subárvore à direita, formada por *c*, *e* e *f*. O nó *a* representa a raiz da árvore e os nós *b* e *c* as raízes das subárvores. Finalmente, os nós *d*, *e* e *f* são as folhas da árvore. Devemos notar que cada nó folha também representa uma árvore, com duas subárvores vazias.

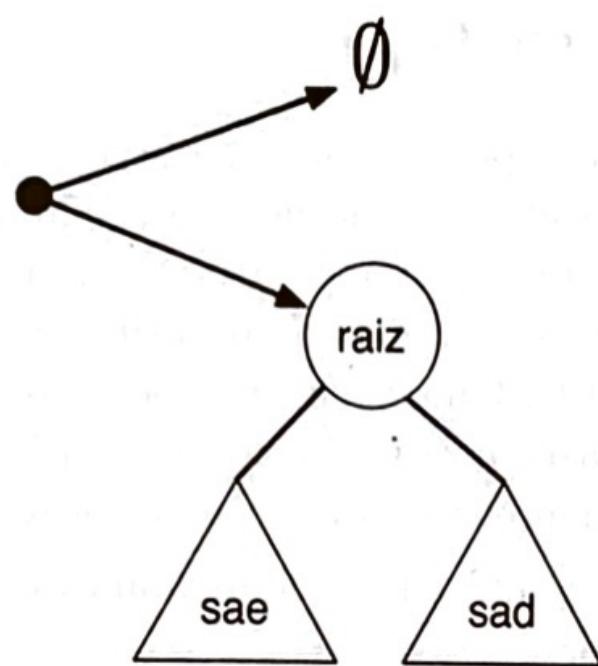


Figura 16.3: Representação esquemática da definição da estrutura de árvore binária.

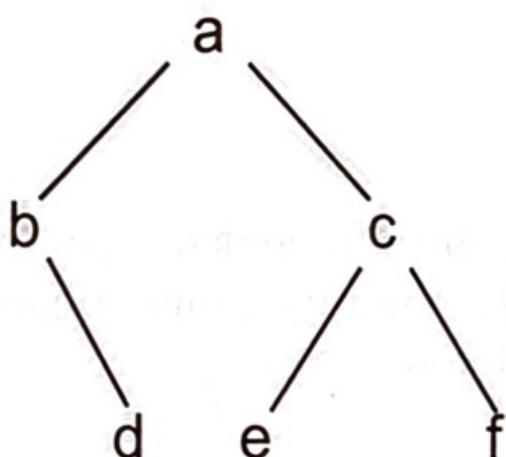


Figura 16.4: Exemplo de árvore binária.

Para descrever árvores binárias, podemos usar a seguinte notação textual: a árvore vazia é representada por `<>` e as árvores não vazias por `<raiz sae sad>`. Com essa notação, a árvore da Figura 16.4 é representada por:

```
<a<b><><d><><>><c<e><>><f><><>>>
```

Pela definição, uma subárvore de uma árvore binária é sempre especificada como sendo a *sae* ou a *sad* de uma árvore maior, e qualquer uma das duas subárvores pode ser vazia. Assim, as duas árvores da Figura 16.5 são distintas.



Figura 16.5: Duas árvores binárias distintas.

Isto também pode ser visto pelas representações textuais das duas árvores, que são, respectivamente: `<a<><>>` e `<a><><>>`.

16.1.1 Representação em C

De modo análogo ao que fizemos para as demais estruturas de dados, podemos definir um tipo para representar uma árvore binária. Para simplificar esta discussão, vamos considerar que as informações que queremos armazenar nos nós da árvore são valores de caracteres simples. Vamos, inicialmente, discutir como podemos representar uma estrutura de árvore binária em C. Que estrutura podemos usar para representar um nó da árvore? Cada nó deve armazenar três informações: a informação propriamente dita, no caso, um caractere, e dois ponteiros para as subárvores, à esquerda e à direita.

Assim, a estrutura que representa um nó de uma árvore binária que armazena valores de caracteres pode ser dada por:

```
typedef struct arvno ArvNo;
struct arvno {
    char info;
    ArvNo* esq;
    ArvNo* dir;
};
```

Podemos ainda criar um tipo abstrato de dados para representar a árvore como um todo. Para tanto, precisamos armazenar um ponteiro para o nó raiz. A partir da raiz, temos acesso aos demais nós da árvore:

```
typedef struct arb Arv;
struct arb {
    ArvNo* raiz;
};
```

Como acontece com qualquer TAD (tipo abstrato de dados), as operações que fazem sentido para uma árvore binária dependem essencialmente da forma de utilização que se pretende fazer da árvore. Nesta seção, em vez de discutir a interface do tipo abstrato para depois mostrar sua implementação, vamos optar por discutir algumas operações mostrando simultaneamente suas implementações. Ao final da seção, apresentaremos um arquivo que pode representar a interface do tipo. Nas funções que se seguem, consideraremos que existem os tipos `Arv` e `ArvNo` definidos. Como veremos, as funções que manipulam árvores são, em geral, implementadas de forma recursiva, usando a definição recursiva da estrutura.

Vamos procurar identificar e descrever apenas operações cuja utilidade seja a mais geral possível. Para a criação da estrutura da árvore, vamos considerar duas funções. Uma primeira função é responsável por criar um novo nó da árvore, recebendo como parâmetros a informação (no caso, um caractere) e as duas subárvores (que podem ser vazias, isto é, `NULL`) associadas:

```
ArvNo* arv_criano (char c, ArvNo* esq, ArvNo* dir)
{
    ArvNo* p = (ArvNo*) malloc(sizeof(ArvNo));
    p->info = c;
    p->esq = esq;
    p->dir = dir;
    return p;
}
```

A segunda função cria a representação do tipo abstrato, recebendo como parâmetro o nó raiz:

```
Arv* arv_cria (ArvNo* r)
{
    Arv* a = (Arv*) malloc(sizeof(Arv));
    a->raiz = r;
    return a;
}
```

Com as duas funções para a criação da estrutura, `cria` e `criano`, podemos criar árvores complexas. Para exemplificar, podemos verificar que a árvore ilustrada na Figura 16.4 pode ser criada pela seguinte atribuição:

```
Arv* a = arv_cria(
    arv_criano('a',
        arv_criano('b',
            NULL,
            arv_criano('d', NULL, NULL)
        ),
        arv_criano('c',
            arv_criano('e', NULL, NULL),
            arv_criano('f', NULL, NULL)
        )
    );
);
```

Outra função útil consiste em exibir o conteúdo da árvore. Essa função deve percorrer recursivamente os nós da árvore, imprimindo as informações associadas. A operação é facilmente implementada usando recursão para percorrer a estrutura da árvore. A função externa do tipo abstrato é responsável apenas por invocar a função recursiva a partir do nó raiz. Na função recursiva, para imprimir a informação de todos os nós da árvore, devemos primeiro testar se a árvore é vazia; se não for, imprimimos a informação associada à raiz e chamamos (recursivamente) a função para imprimir as subárvore a partir de suas raízes.

```
static void imprime (ArvNo* r)
{
    if (r != NULL){
        printf("%c ", r->info); /* mostra informação */
        imprime(r->esq);      /* imprime sae */
        imprime(r->dir);      /* imprime sad */
    }
}

void arv_imprime (Arv* a)
{
    imprime(a->raiz); /* imprime recursivamente a partir da raiz */
}
```

A estrutura do código para implementar esta operação será recorrente: uma função externa que recebe a árvore como parâmetro e uma função auxiliar recursiva que opera a partir da raiz da árvore. Se a função para imprimir fosse aplicada à árvore ilustrada na Figura 16.4, a saída da função seria: **a b d c e f**.

Podemos notar que a impressão não nos permite identificar a estrutura da árvore. Podemos então modificar a implementação de `imprime` de forma que a saída impressa reflita, além do conteúdo de cada nó, a estrutura da árvore, usando a notação textual introduzida anteriormente. Uma possível implementação dessa função é mostrada a seguir:

```
static void imprime (ArvNo* r)
{
    printf("<");
    if (r != NULL){
        printf("%c", r->info);      /* mostra informação */
        imprime(r->esq);          /* imprime sae */
        imprime(r->dir);          /* imprime sad */
    }
    printf(">");
}
```

Outra operação que pode ser acrescentada é a que libera a memória alocada pela estrutura da árvore. Novamente, teremos duas funções: a função recursiva para liberar os nós e a função externa para liberar a estrutura da árvore:

```
static void libera (ArvNo* r)
{
    if (r != NULL) {
        libera(r->esq);          /* libera a sae */
        libera(r->dir);          /* libera a sad */
        free(r);                  /* libera o nó raiz */
    }
}

void ary_libera (Arv* a)
{
    libera(a->raiz);           /* libera hierarquia de nós */
    free(a);                   /* libera raiz */
}
```

Note que, na função recursiva `libera`, foi fundamental liberar as subárvore antes de liberar a estrutura do nó (caso contrário, não teríamos acesso às subárvore).

Outra função que podemos considerar percorre a árvore verificando a ocorrência de determinado caractere *c* em um de seus nós. Essa função tem como retorno um valor booleano (um ou zero) indicando a ocorrência ou não do caractere na estrutura.

```
static int pertence (ArvNo* r, char c)
{
    if (r == NULL)
        return 0;      /* árvore vazia: não encontrou */
```

```

else
    return c==r->info ||
           pertence(r->esq,c) ||
           pertence(r->dir,c);
}

int arv_pertence (Arv* a, char c)
{
    return pertence(a->raiz,c);
}

```

Note que esta forma de programar usando o operador lógico `||` ("ou") faz com que a busca seja interrompida assim que o elemento for encontrado. Isto acontece porque se `c==r->info` for verdadeiro, as duas outras expressões não chegam a ser avaliadas. Analogamente, se o caractere for encontrado na subárvore da esquerda, a busca não prossegue na subárvore da direita.

Podemos dizer que a expressão:

```

return c==r->info ||
       pertence(r->esq,c) ||
       pertence(r->dir,c);

```

é equivalente a:

```

if (c == r->info)
    return 1;
else if (pertence(r->esq,c))
    return 1;
else
    return pertence(r->dir,c);

```

Nos casos práticos, na verdade, em geral não é suficiente saber se uma informação está ou não presente na estrutura; queremos ter acesso ao nó que contém a informação buscada. Podemos então considerar uma função que, em vez de ter como valor de retorno um booleano, tenha como valor de retorno o ponteiro do nó que contém a informação. Se a informação não for encontrada na estrutura, tem-se como retorno o valor `NULL`. Neste caso, a forma compacta de codificação do valor de retorno não pode ser usada:

```

static ArvNo* busca (ArvNo* r, char c)
{
    if (r == NULL)
        return NULL; /* árvore vazia: não encontrou */
    else if (c == r->info)
        return r;
    else {
        ArvNo* p = busca(r->esq,c); /* busca na sae */
        if (p != NULL)
            return p; /* encontrou na sae */
        else
            return busca(r->dir,c); /* busca na sad */
    }
}

```

```
ArvNo* arv_busca (Arv* a, char c)
{
    return busca(a->raiz, c);
}
```

Finalmente, considerando que as funções discutidas e implementadas formam a interface do tipo abstrato para representar uma árvore binária, um arquivo de interface "arv.h" pode ser dado por:

```
typedef struct arv Arv;
typedef struct arvno ArvNo;

Arv* arv_cria (void);
ArvNo* arv_criano (char c, ArvNo* e, ArvNo* d);
void arv_libera (Arv* a);
void arv_imprime (Arv* a);
int arv_pertence (Arv* a, char c);
ArvNo* arv_busca (Arv* a, char c);
```

16.1.2 Ordens de percurso em árvores binárias

A programação da operação `imprime`, vista anteriormente, seguiu a ordem empregada na definição de árvore binária para decidir a ordem em que as três ações seriam executadas: imprimimos o conteúdo da raiz, em seguida imprimimos o conteúdo da subárvore à esquerda e, então, imprimimos o conteúdo da subárvore à direita. Entretanto, dependendo da aplicação, esta ordem pode não ser a preferível. Podemos utilizar uma ordem diferente desta, por exemplo:

```
imprime(r->esq);          /* mostra sae */
imprime(r->dir);          /* mostra sad */
printf("%c ", r->info);   /* mostra raiz */
```

Muitas operações em árvores binárias envolvem o percurso de todas as subárvores, executando alguma ação de tratamento em cada nó, de forma que é comum percorrer uma árvore em uma das seguintes ordens:

- *pré-ordem*: trata raiz, percorre *sae*, percorre *sad*;
- *ordem simétrica*: percorre *sae*, trata raiz, percorre *sad*;
- *pós-ordem*: percorre *sae*, percorre *sad*, trata raiz.

Na implementação da função `libera`, por exemplo, tivemos que adotar a pós-ordem:

```
libera(r->esq);      /* libera sae */
libera(r->dir);      /* libera sad */
free(r);              /* libera raiz */
```

- Em árvores binárias de busca, como veremos, a ordem de percurso importante é a ordem simétrica. Algumas outras ordens de percurso podem ser definidas, podendo inverter a ordem de percurso entre a *sae* e a *sad*.

16.1.3 Altura de uma árvore

Uma propriedade fundamental de todas as árvores é que só existe um caminho da raiz para qualquer nó. Com isso, podemos definir a *altura* de uma árvore como sendo o comprimento do caminho da raiz até a folha mais distante. Por exemplo, a altura da árvore da Figura 16.4 é 2, e a altura das árvores da Figura 16.5 é 1. Assim, a altura de uma árvore com apenas o nó raiz é zero e, por conseguinte, dizemos que a altura de uma árvore vazia é negativa e vale -1 . Também podemos numerar os níveis em que os nós aparecem na árvore. A raiz está no nível 0, seus filhos diretos no nível 1 e assim por diante. O último nível da árvore é o nível h , sendo h a altura da árvore. A Figura 16.6 identifica os níveis e a altura de uma árvore binária.

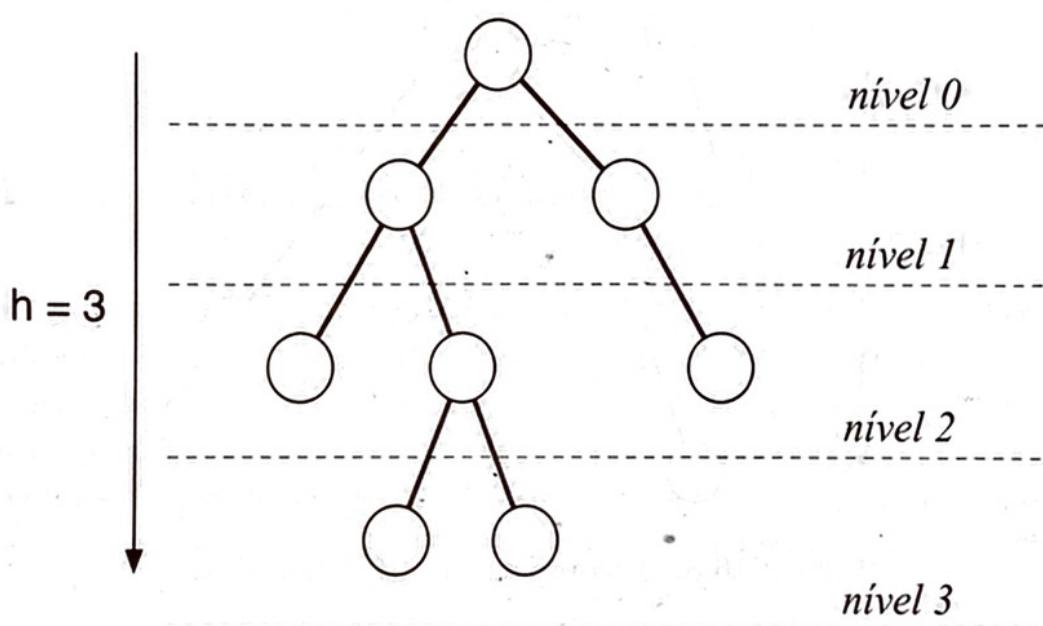


Figura 16.6: Níveis e altura de uma árvore binária.

Uma árvore é dita cheia se todos os seus nós folhas estão no mesmo nível e não se pode acrescentar nenhuma nova folha sem aumentar a altura da árvore. A Figura 16.7 ilustra uma árvore cheia. Podemos notar que numa árvore cheia temos 1 nó no nível 0, 2 nós no nível 1, 4 nós no nível 2, 8 nós no nível 3 e assim por diante. Isto é, no nível n , temos 2^n nós. Também podemos notar que o número de nós de determinado nível de uma árvore cheia é uma unidade a mais do que a soma de todos os nós dos níveis anteriores:

$$N = \sum_{i=0}^n 2^i = 2^{n+1} - 1$$

Uma árvore cheia de altura h tem, portanto, $2^{h+1} - 1$ nós.

Uma árvore é dita *degenerada* se todos os seus nós internos têm uma única subárvore associada. De fato, a estrutura hierárquica se degenera numa estrutura linear. A Figura 16.8 exemplifica árvores degeneradas. Observamos que numa árvore degenerada temos um único nó em cada nível. Assim, uma árvore degenerada de altura h tem $h + 1$ nós.

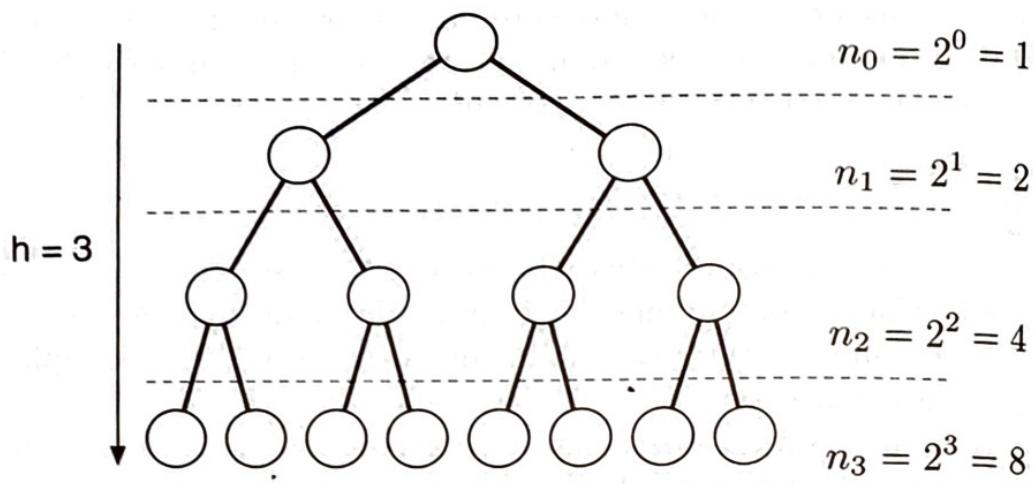


Figura 16.7: Árvore cheia de altura 3.

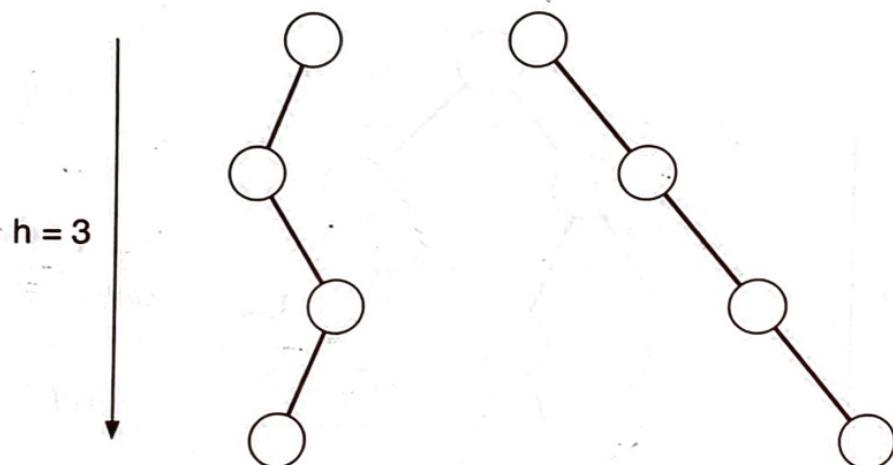


Figura 16.8: Árvores binárias degeneradas.

A altura de uma árvore é uma medida importante na avaliação da eficiência com que visitamos os nós de uma árvore. Uma árvore binária com n nós tem uma altura mínima proporcional a $\log_2 n$ (caso da árvore cheia) e uma altura máxima proporcional a n (caso da árvore degenerada). A altura indica o esforço computacional necessário para alcançar qualquer nó da árvore a partir da raiz. Quando discutirmos árvores binárias de busca, verificaremos a importância de manter as árvores com altura pequena, isto é, manter as árvores com uma distribuição de nós próxima à da árvore cheia.

Podemos pensar na implementação de uma função que calcula a altura de uma árvore binária. A implementação dessa função é simples, bastando aplicar a definição recursiva dada. Se a árvore for vazia, sua altura, por definição, vale -1 . Se a árvore não for vazia, sua altura será dada pela maior altura das subárvores, acrescida de 1 (a árvore tem um nível a mais do que suas subárvores, que é o nível da sua raiz). Uma possível implementação dessa função, que usa uma função auxiliar para calcular o máximo entre dois números inteiros, é mostrada a seguir:

```
static int max2 (int a, int b)
{
    return (a > b) ? a : b;
}
```

```

static int altura (ArvNo* r)
{
    if (r == NULL)
        return -1;
    else
        return 1 + max2(altura(r->esq), altura(r->dir));
}

int arv_altura (Arv* a)
{
    return altura(a->raiz);
}

```

Devemos notar que a altura da árvore vazia deve ser -1 para que a função funcione corretamente.

16.2 Árvore binária de busca

Como vimos, o algoritmo de busca binária tem bom desempenho computacional e deve ser usado quando temos os dados ordenados em um vetor. No entanto, se precisarmos inserir e remover elementos da estrutura e, ao mesmo tempo, dar suporte a funções de busca eficientes, a estrutura de vetor (e, consequentemente, o uso do algoritmo de busca binária em vetor) não se torna adequada. Para inserir um novo elemento num vetor ordenado, temos que rearrumar os elementos no vetor para abrir espaço para a inserção do novo elemento. Uma 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 na seção anterior 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 até $2^{h+1} - 1$ nós. 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 $O(\log n)$. A 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 acessar qualquer nó de maneira eficiente é necessário ter árvores binárias “balanceadas”, em que o número de nós da subárvore à esquerda seja próximo do número de nós da subárvore à direita, recursivamente. 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 dada por $O(n)$. 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 subárvore à esquerda (*sae*) e é sempre menor que o valor associado a qualquer nó da subárvore 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 feita de forma eficiente. 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 frequentemente chamadas de *árvores binárias de busca*.

Naturalmente, a *ordem* a que fizemos referência 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, uma função de comparação específica deve ser empregada.

16.2.1 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 assumir que não há repetição de valores associados aos nós da árvore. A Figura 16.9 ilustra uma árvore de busca de valores inteiros.

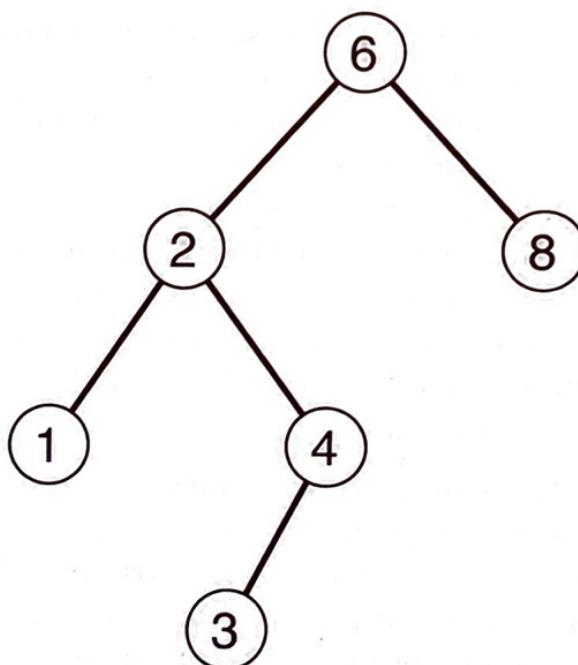


Figura 16.9: Exemplo de árvore binária de busca.

O tipo do nó da árvore binária para armazenar valores inteiros pode ser dado por:

```
struct arvno {
    int info;
    ArvNo* esq;
    ArvNo* dir;
};
```

Vamos considerar a estrutura externa que representa a árvore contendo o ponteiro para a raiz, como na seção anterior.

Supondo a existência de uma árvore binária de busca já construída, podemos imprimir os valores da árvore em ordem crescente, percorrendo os nós em ordem simétrica:

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

Esta função `imprime` é similar à que vimos para árvores binárias comuns, apenas variando a ordem de percurso, a fim de explorar a propriedade de ordenação das árvores de busca. No entanto, as operações que nos interessam 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.

16.2.2 Operação de busca

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

```
static ArvNo* busca (ArvNo* 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;
}
ArvNo* abb_busca (Arv* a, int v)
{
    return busca(a->raiz, v);
}
```

16.2.3 Operação de criação

A operação que cria uma árvore binária inicialmente vazia é simples e pode ser dada por:

```
Arv* abb_cria (void)
{
    Arv* a = (Arv*) malloc(sizeof(Arv));
    a->raiz = NULL;
    return a;
}
```

Após a criação da árvore, podemos inserir e remover elementos da estrutura.

16.2.4 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 for vazia, deve ser substituída por uma árvore cujo único nó (o nó raiz) contenha o valor v . Se a árvore não for vazia, comparamos v com o valor na raiz da árvore e inserimos v na *sae* ou na *sad*, conforme o resultado da comparação. O código a seguir ilustra a implementação dessa operação. A função recursiva tem como valor de retorno o eventual novo nó raiz da (sub)árvore.

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

void abb_insere (Arv* a, int v)
{
    a->raiz = insere(a->raiz,v);
}
```

Salientamos a necessidade de atualizar os ponteiros para as subárvores à esquerda à direita quando da chamada recursiva da função, pois a função de inserção pode alterar o valor do ponteiro para a raiz da (sub)árvore. O mesmo é feito na função exportada: a raiz da árvore é atualizada com o valor de retorno da chamada da função interna recursiva.

16.2.5 Operação de remoção

Outra operação a ser analisada é a que permite retirar determinado elemento da árvore. A função recursiva desta operação também deve ter como valor de retorno a eventual nova raiz da (sub)árvore, mas sua implementação é mais complexa que a de inserção. De novo, devemos pensar essa implementação com base na definição recursiva da árvore. Se a árvore for vazia, nada tem que ser feito, pois o elemento não está presente na árvore. Se a árvore não for vazia, comparamos o valor armazenado no nó raiz com o valor que se deseja retirar da árvore. Se o valor associado à raiz for maior que o valor a ser retirado, chamamos a função recursivamente para retirar o elemento da subárvore à esquerda. Se o valor da raiz for menor, retiramos o elemento da subárvore à direita. Finalmente, se o valor associado à raiz for igual, encontramos o elemento a ser retirado e devemos efetuar essa operação. Portanto, estaremos sempre retirando um nó raiz de uma (sub)árvore.

Nesse caso, existem três situações possíveis. A primeira, e mais simples, é quando se deseja retirar uma raiz que é folha (isto é, uma raiz que não tem filhos). Nesta condição, basta liberar a memória alocada pelo elemento e ter como valor de retorno a raiz atualizada, que passa a ser `NULL`.

A segunda situação, ainda simples, acontece quando a raiz a ser retirada possui um único filho. Retirando esse nó, a raiz da árvore passa a ser o único filho existente. A Figura 16.10 ilustra essa situação.

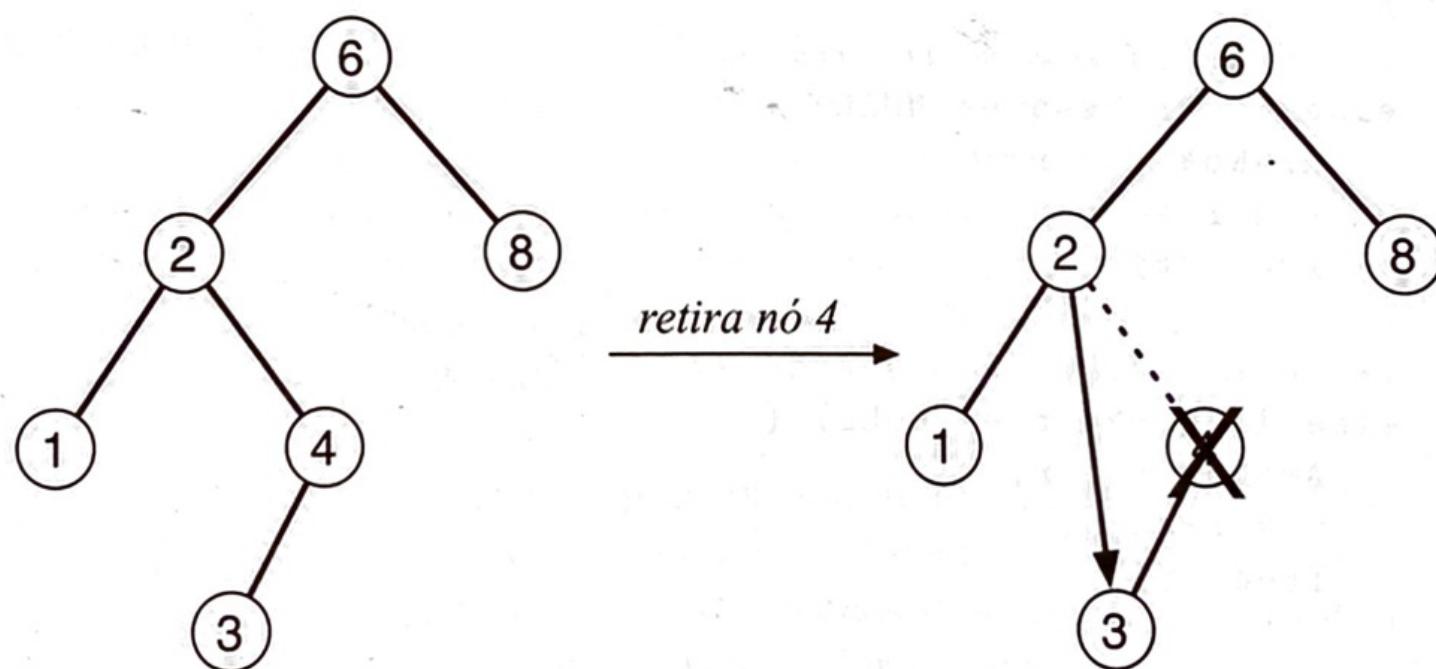


Figura 16.10: Retirada de um elemento com um único filho.

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

1. Encontramos o elemento que precede a raiz na ordenação; isto equivale a encontrar o elemento mais à direita da subárvore à esquerda.
2. Trocamos a informação da raiz com a informação do nó encontrado.
3. Retiramos da subárvore à esquerda, chamando a função recursivamente, o nó encontrado (que agora contém a informação da raiz que se deseja retirar). Observa-se que retirar o nó mais à direita é trivial, pois ele é um nó folha ou um nó com um único filho (no caso, o filho da direita nunca existe).

A Figura 16.11 ilustra o procedimento para a retirada de um nó com dois filhos. O procedimento descrito deve ser seguido para não haver violação da ordenação da árvore.

Observamos que uma operação análoga à que foi feita com o nó mais à direita da subárvore à esquerda pode ser feita com o nó mais à esquerda da subárvore à direita (que é o nó que segue a raiz na ordenação).

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

```

static ArvNo* retira (ArvNo* 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 */
        /* elemento sem filhos */
        if (r->esq == NULL && r->dir == NULL) {
            free (r);
            r = NULL;
        }
        /* só tem filho à direita */
        else if (r->esq == NULL) {
            ArvNo* t = r;
            r = r->dir;
            free (t);
        }
        /* só tem filho à esquerda */
        else if (r->dir == NULL) {
            ArvNo* t = r;
            r = r->esq;
            free (t);
        }
        /* tem os dois filhos */
        else {
            ArvNo* f = r->esq;
            while (f->dir != NULL) {
                f = f->dir;
            }
            r->info = f->info;          /* troca as informações */
            f->info = v;
            r->esq = abb_retira(r->esq, v);
        }
    }
    return r;
}
void abb_retira (Arv* a, int v)
{
    a->raiz = retira(a->raiz, v);
}

```

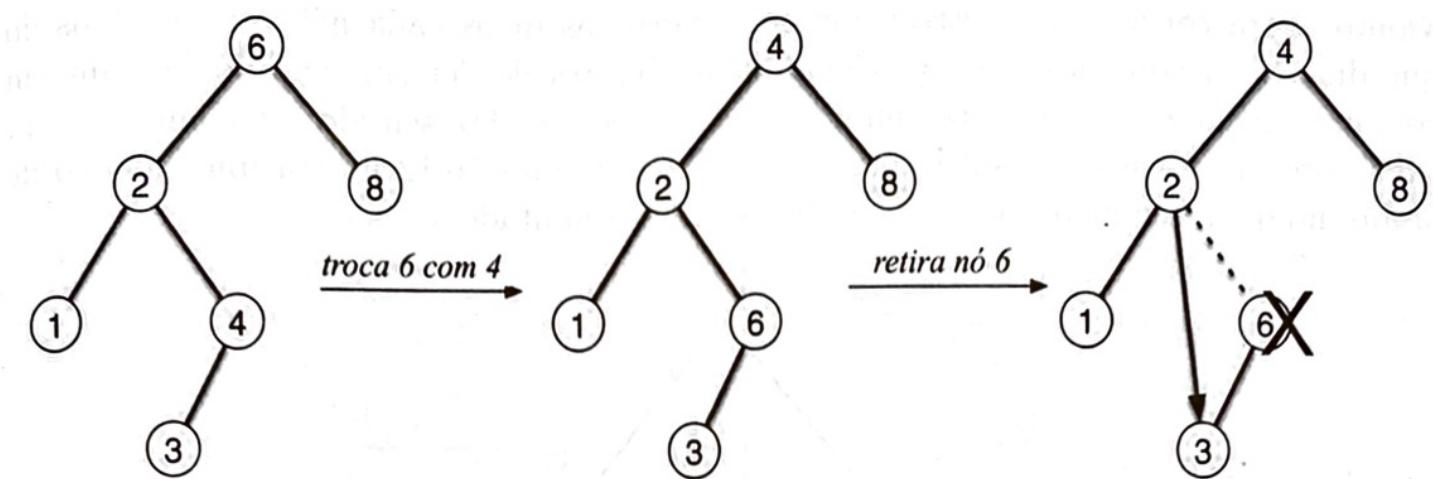


Figura 16.11: Retirada de um elemento com dois filhos.

16.2.6 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 subárvore (sempre retirando um nó da subárvore à esquerda, por exemplo). Uma estratégia que pode ser utilizada para amenizar o problema é intercalar de qual subárvore 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 subárvore de cada nó o mesmo “peso”, isto é, o número de elementos nas duas subárvore deve ser igual ou aproximadamente igual. No caso de um número de nós par, podemos aceitar uma diferença de um nó entre a subárvore à esquerda e a subárvore à direita.

A ideia central de um algoritmo para平衡ar (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*, na qual 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 subárvore seja menor ou igual a 1. Naturalmente, o processo deve continuar (recursivamente) com o balanceamento das duas subárvore de cada árvore. Um ponto a observar é que a remoção do menor (ou maior) elemento de uma árvore é mais simples do que a remoção de um elemento qualquer. A implementação desse algoritmo para balanceamento da árvore fica como sugestão de exercício. Finalmente, é importante salientar que existem diferentes estruturas de árvores mais avançadas que dão suporte a operações de busca de forma bastante eficiente.

16.3 Árvores com número variável de filhos

Nesta seção, discutiremos as estruturas de árvores com número variável de filhos. Como vimos, numa árvore binária, o número de filhos dos nós é limitado em no máximo dois.

Vamos agora considerar as estruturas de árvores nas quais cada nó pode ter mais do que duas subárvore associadas. Como as subárvore de determinado nó formam um conjunto linear e são dispostas numa ordem específica, faz sentido falar em primeira subárvore (*sa1*), segunda subárvore (*sa2*) etc. A Figura 16.12 ilustra um exemplo de árvore no qual o número máximo de filhos não está limitado a dois.

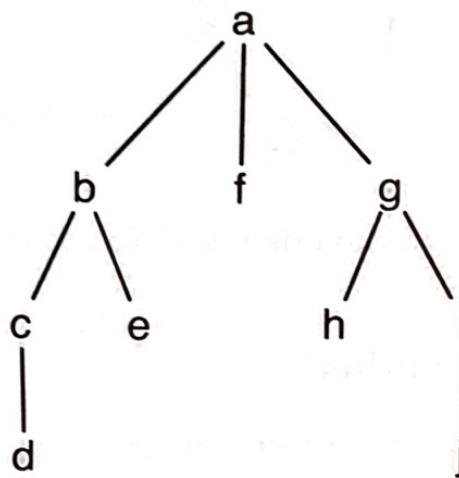


Figura 16.12: Exemplo de árvore que não é binária.

Nesse exemplo, podemos notar que o nó raiz *a* tem três subárvore, ou seja, o nó *a* tem três filhos. Os nós *b* e *g* têm dois filhos cada um; os nós *c* e *i* têm um filho cada, e os nós *d*, *e*, *h* e *j* são folhas e não têm filhos.

De forma semelhante ao que foi feito no caso das árvores binárias, podemos representar essas árvores através de notação textual, usando o seguinte formato:

<*raiz* *sa1* *sa2* ... *san*>

Com essa notação, a árvore da Figura 16.12 seria representada por:

<*a* <*b* <*c* <*d*>> <*e*>> <*f*> <*g* <*h*> <*i* <*j*>>>

16.3.1 Representação em C

Dependendo da aplicação, podemos usar diferentes estruturas para representar árvores, levando em consideração o número de filhos que cada nó pode apresentar. Se soubermos, por exemplo, que numa aplicação o número máximo de filhos que um nó pode apresentar é três, podemos montar uma estrutura com três campos de apontadores para os nós filhos, digamos, *f1*, *f2* e *f3*. Os campos não utilizados podem ser preenchidos com o valor nulo *NULL*. Prevendo um número máximo de filhos igual a três, e considerando a implementação de árvores para armazenar valores de caracteres simples, a declaração do tipo que representa o nó da árvore poderia ser:

```

typedef struct arv3no Arv3No;
struct arv3no {
    char info;
    Arv3No *f1, *f2, *f3;
};
  
```

A Figura 16.13 ilustra a representação da árvore da Figura 16.12 com esta organização.

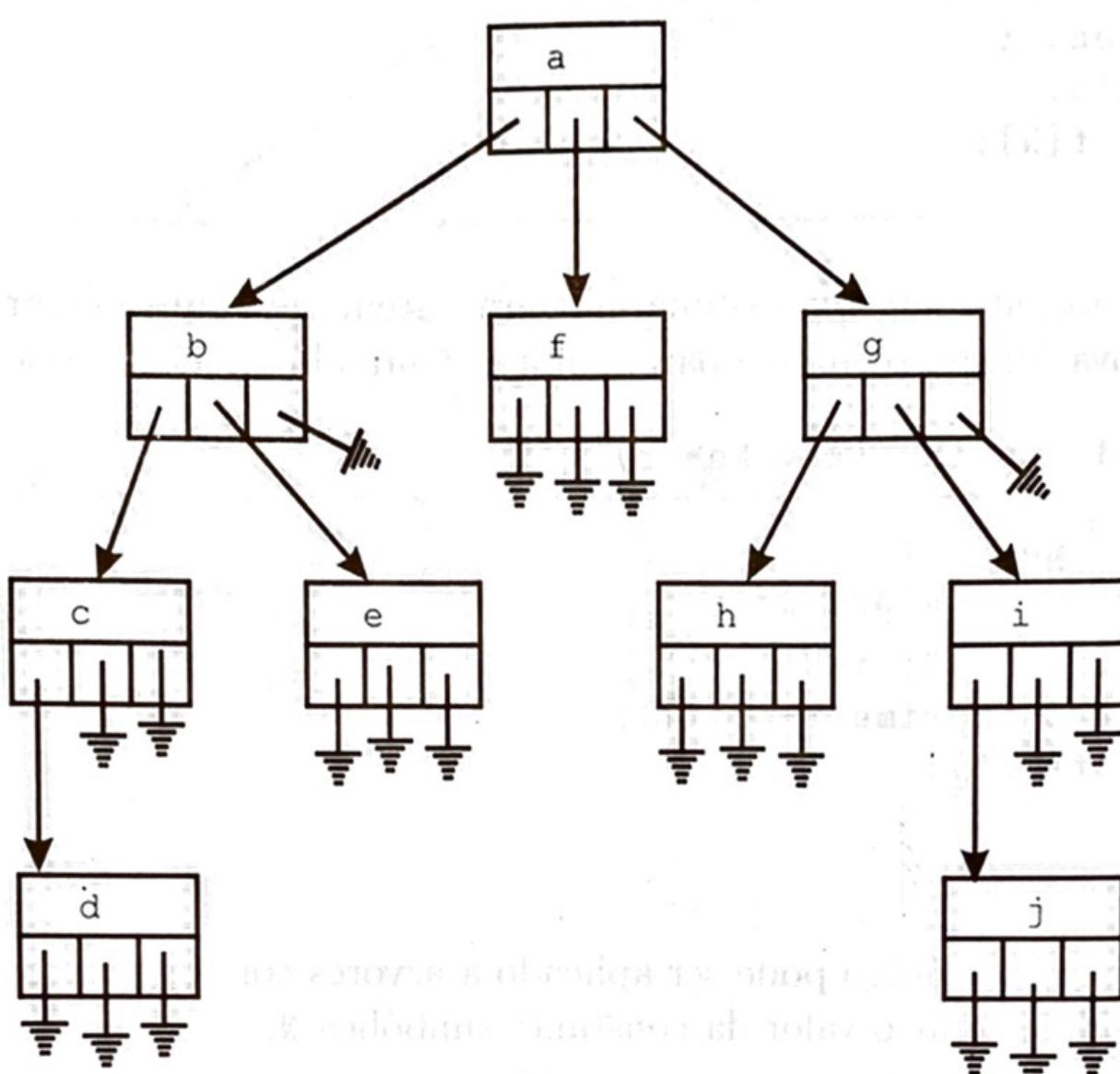


Figura 16.13: Árvore com no máximo três filhos por nó.

Para ilustrar o acesso aos elementos da árvore com essa representação em C, podemos implementar uma função recursiva para exibir a representação textual no formato mostrado, sem representar árvores vazias:

```

static void imprime (Arv3No* r)
{
    if (r != NULL) {
        printf("<%c", r->info);
        imprime(r->f1);
        imprime(r->f2);
        imprime(r->f3);
        printf(">");
    }
}
  
```

Apesar de correto, o código dessa função mostra que esta representação não é muito adequada, pois não existe uma maneira sistemática para acessar os nós filhos. Existem diversas aplicações computacionais em que precisamos trabalhar com árvores nas quais o número de filhos é limitado. Na área de computação gráfica, por exemplo, são muito utilizadas as árvores com quatro e com oito filhos por nó, conhecidas como *quadtree* e *octree*. Portanto, precisamos estruturar de maneira mais adequada os filhos dos nós da árvore (seria impraticável declarar um campo na estrutura para cada possível filho).

Uma representação mais adequada consiste em armazenar os filhos dos nós em um vetor. Assim, a representação do nó para a árvore com até três filhos passa a ser dada por:

```
#define N 3
typedef struct arv3no Arv3No;
struct arv3no {
    char info;
    Arv3No* f[N];
};
```

Com essa representação, temos uma maneira sistemática para visitar todos os filhos de um nó. A nova função recursiva para exibir o conteúdo da árvore pode ser dada por:

```
static void imprime (Arv3No* r)
{
    if (r != NULL) {
        printf("<%c", r->info);
        for (int i=0; i<N; i++)
            arv3_imprime(r->f[i]);
        printf(">");
    }
}
```

Dessa forma, o mesmo código pode ser aplicado a árvores com outros limites de número de filhos, bastando alterar o valor da constante simbólica **N**.

No entanto, em aplicações em que não existe um limite superior no número de filhos, esta técnica não é aplicável. O mesmo acontece se existe um limite no número de nós, mas esse limite é raramente alcançado, pois estariam tendo um grande desperdício de espaço de memória com os campos não utilizados. Nesses casos, precisamos, de fato, de uma estrutura de árvore que não imponha restrições ao número de filhos de cada nó. Um exemplo de aplicação é a representação de árvores de diretórios, na qual o número de filhos varia arbitrariamente.

A representação em C de uma árvore com número variável de filhos por nó pode utilizar então uma “lista de filhos”: um nó aponta apenas para seu primeiro (**prim**) filho, e cada um de seus filhos aponta para o próximo (**prox**) irmão. Desta forma, cada nó pode ter um número arbitrário de filhos. A Figura 16.14 ilustra essa representação de árvore.

Devemos notar que essa representação também permite acessar de forma sistemática os filhos de um nó, pois eles estão organizados numa estrutura de lista encadeada. Os tipos que representam uma árvore com número variável de filhos com esta estrutura de lista podem ser dados por:

```
typedef struct arvn Arvn;
typedef struct arvnno ArvnNo;

struct arvnno {
    char info;
    ArvnNo* prim; /* ponteiro para eventual primeiro filho */
    ArvnNo* prox; /* ponteiro para eventual irmão */
};

struct arvn {
    ArvnNo* raiz;
};
```

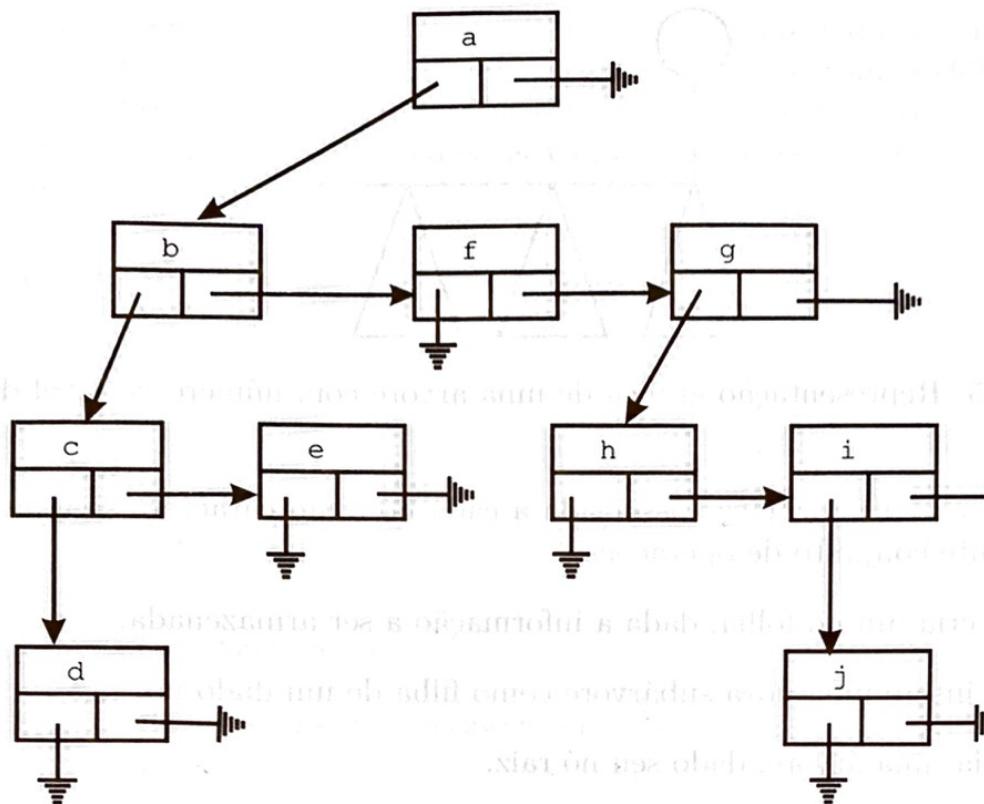


Figura 16.14: Representação de árvores com “lista de filhos”.

Cada nó, além da informação associada, guarda duas referências: uma para a primeira subárvore filha e outra para a próxima subárvore irmã. Se o nó representar uma folha da árvore, o valor de `prim` será `NULL`, pois esse nó não terá filhos. Se o nó representar o último filho de outro nó, o valor de `prox` será `NULL`, pois não existirá um próximo irmão.

As funções que manipulam esse tipo de árvore também farão uso de implementações recursivas. Na implementação dessas funções, adotaremos a seguinte definição da árvore, dado seu nó raiz. Uma árvore é composta por:

- um nó raiz; e
- zero ou mais subárvores.

A Figura 16.15 ilustra essa definição de forma esquemática. Estritamente, essa definição não trata o caso de uma subárvore vazia. Assim, uma folha de uma árvore não é um nó com subárvores vazias, como no caso da árvore binária, mas é um nó com zero subárvores. Em qualquer definição recursiva deve haver uma “condição de contorno” que permita a definição de estruturas finitas, e, no nosso caso, a definição de uma árvore se encerra nas folhas, que são identificadas como sendo nós com zero subárvores.

Como as funções que implementaremos nesta seção serão baseadas nessa definição, não será considerado o caso de árvores vazias na recursão. O caso da árvore vazia será tratado na estrutura externa que representa a árvore como um todo. Esta estratégia simplifica as implementações recursivas e não limita a utilização da estrutura.

16.3.2 Tipo abstrato de dado

Para exemplificar a implementação de funções que manipulam uma árvore com número variável de filhos, vamos considerar a criação de um tipo abstrato de dados para represen-

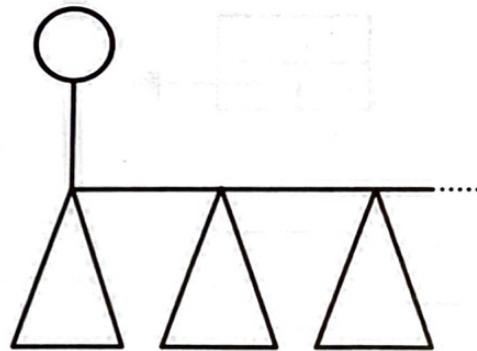


Figura 16.15: Representação gráfica de uma árvore com número variável de filhos.

tar árvores em que a informação associada a cada nó é um caractere simples. Podemos definir o seguinte conjunto de operações:

- **criano**: cria um nó folha, dada a informação a ser armazenada.
- **insere**: insere uma nova subárvore como filha de um dado nó.
- **cria**: cria uma árvore, dado seu nó raiz.
- **imprime**: percorre todos os nós e imprime suas informações.
- **busca**: verifica a ocorrência de determinado valor num dos nós da árvore.
- **libera**: libera toda a memória alocada pela árvore.

A interface do tipo pode então ser definida no arquivo ArvnNo.h dado por:

```

typedef struct arvn Arvn;
typedef struct arvnno ArvnNo;

ArvnNo* arvn_criano (char c);
void arvn_insere (ArvnNo* a, ArvnNo* sa);
Arvn* arvn_cria (ArvnNo* r);
void arvn_imprime (Arvn* a);
ArvnNo* arvn_busca (Arvn* a, char c);
void arvn_libera (Arvn* a);
  
```

Vamos então apresentar a implementação de cada uma dessas funções. A estrutura dos tipos **Arvn** e **ArvnNo** já foram apresentadas. A função para criar uma folha deve alocar o nó e inicializar seus campos, atribuindo **NULL** aos campos **prim** e **prox**, pois trata-se de um nó folha isolado.

```

ArvnNo* arvn_criano (char c)
{
    ArvnNo *a = (ArvnNo*) malloc(sizeof(ArvnNo));
    a->info = c;
    a->prim = NULL;
    a->prox = NULL;
    return a;
}
  
```

A função que insere uma nova subárvore como filha de um dado nó é muito simples. Como não vamos atribuir nenhum significado especial à posição de um nó filho, a operação de inserção pode inserir a subárvore em qualquer posição. Neste caso, vamos optar por inserir sempre no início da lista que, como já vimos, é a maneira mais simples de inserir um novo elemento numa lista encadeada.

```
void arvn_insere (ArvnNo* a, ArvnNo* sa)
{
    sa->prox = a->prim;
    a->prim = sa;
}
```

A função para criar a estrutura externa que representa a árvore, dada sua raiz, pode ser:

```
Arvn* arvn_cria (ArvnNo* r)
{
    Arvn* a = (Arvn*) malloc(sizeof(Arvn));
    a->raiz = r;
    return a;
}
```

Com essas três funções, podemos construir a árvore do exemplo da Figura 16.13 com o seguinte fragmento de código:

```
/* cria nós como folhas */
ArvnNo* a = arvn_criano('a');
ArvnNo* b = arvn_criano('b');
ArvnNo* c = arvn_criano('c');
ArvnNo* d = arvn_criano('d');
ArvnNo* e = arvn_criano('e');
ArvnNo* f = arvn_criano('f');
ArvnNo* g = arvn_criano('g');
ArvnNo* h = arvn_criano('h');
ArvnNo* i = arvn_criano('i');
ArvnNo* j = arvn_criano('j');

/* monta a hierarquia */
arvn_insere(c,d);
arvn_insere(b,e);
arvn_insere(b,c);
arvn_insere(i,j);
arvn_insere(g,i);
arvn_insere(g,h);
arvn_insere(a,g);
arvn_insere(a,f);
arvn_insere(a,b);

/* cria árvore */
Arvn* x = arvn_cria(a);
...
```

Para imprimir as informações associadas aos nós da árvore, temos duas opções para percorrer a árvore: pré-ordem (primeiro a raiz e depois as subárvore) ou pós-ordem (primeiro as subárvore e depois a raiz). Note que, neste caso, não faz sentido a ordem simétrica, uma vez que o número de subárvore é variável. Para esta função, vamos optar por imprimir o conteúdo dos nós em pré-ordem:

```
static void imprime (ArvnNo* r)
{
    printf(" <%c", r->info);
    for (ArvnNo* p=r->prim; p!=NULL; p=p->prox)
        imprime(p); /* imprime cada subárvore filha */
    printf(">");
}

void arvn_imprime (Arvn* a)
{
    if (a->raiz != NULL)
        imprime(a->raiz);
}
```

A operação para verificar a ocorrência de uma dada informação na árvore é exemplificada a seguir:

```
static ArvnNo* busca (ArvnNo* r, char c)
{
    if (r->info==c)
        return r;
    else {
        for (ArvnNo* p=r->prim; p!=NULL; p=p->prox) {
            ArvnNo* q = busca(p,c);
            if (q != NULL)
                return q;
        }
        return NULL;
    }
}

ArvnNo* arvn_busca (Arvn* a, char c)
{
    if (a->raiz == NULL)
        return NULL;
    else
        return busca(a->raiz,c);
}
```

A última operação apresentada é a que libera a memória alocada pela árvore. O único cuidado que precisamos tomar na programação da função recursiva é o de liberar as subárvore antes de liberar o espaço associado a um nó (isto é, usar pós-ordem).

```

static void libera (ArvnNo* r)
{
    ArvnNo* p = r->prim;
    while (p!=NULL) {
        ArvnNo* t = p->prox;
        libera(p);
        p = t;
    }
    free(r);
}

void arvn_libera (Arvn* a)
{
    if (a->raiz != NULL)
        libera(a->raiz);
    free(a);
}

```

16.3.3 Altura da árvore

As mesmas definições de níveis e altura que fizemos para as árvores binárias se aplicam às árvores com número variável de filhos. Assim, a árvore ilustrada na Figura 16.12 tem altura igual a 3, pois os nós *d* e *j* estão no nível 3 da árvore.

Para o cálculo da altura da árvore, devemos considerar a definição recursiva que temos usado nas nossas implementações. Desta forma, a altura da árvore será uma unidade a mais do que a maior altura entre as subárvores filhas. Portanto, precisamos computar a maior altura entre as subárvores e retornar esse valor acrescido de uma unidade. Caso o nó raiz não tenha filhos, a altura da árvore deve ser zero. Uma implementação dessa função pode ser dada por:

```

static int altura (ArvnNo* r)
{
    int hmax = -1; /* -1 para tratar caso c/ zero filhos */
    for (ArvnNo* p=r->prim; p!=NULL; p=p->prox) {
        int h = altura(p);
        if (h > hmax)
            hmax = h;
    }
    return hmax + 1;
}

int arvn_altura (Arvn* a)
{
    if (a->raiz == NULL)
        return -1;
    else
        return altura(a->raiz);
}

```

16.3.4 Topologia binária

A representação de árvores com número variável de filhos que fizemos é apenas conceitual. Concretamente, a estrutura que usamos para representar o nó da árvore adota a mesma topologia que usamos para representar o nó da árvore binária. O nó, além da informação associada, tem dois ponteiros para subárvores. O que muda é o significado que atribuímos a cada uma das subárvores referenciadas. No caso da árvore binária, uma representa a subárvore à esquerda e a outra, a subárvore à direita. Aqui, uma representa a primeira subárvore filha e a outra, a subárvore irmã. A Figura 16.16 ilustra a topologia binária de ambas as representações.



Figura 16.16: Topologia binária nós de árvores.

Feita a observação, podemos trabalhar com a definição de árvore com número variável de filhos de maneira análoga ao que fizemos para árvore binária. Podemos considerar que um nó da árvore representa:

- uma árvore vazia; ou
- um nó raiz tendo duas subárvores, identificadas como a subárvore filha e a subárvore irmã.

Com essa nova definição de árvore, podemos reescrever os códigos das funções que discutimos. Algumas funções ficam mais simples se feitas usando essa nova definição. O leitor deve optar por utilizar a definição que julgar mais adequada para resolver o problema em questão. Para ilustrar, vamos escrever a função que calcula a altura de uma árvore usando essa nova definição recursiva. Assim, a altura da árvore será o maior valor entre a altura da subárvore filha, acrescido de uma unidade, e a altura da subárvore irmã. O código dessa implementação é mostrado a seguir. Devemos notar que o caso da árvore vazia agora é considerado na recursão, pois faz parte da definição recursiva que estamos usando.

```

static int max2 (int a, int b)
{
    return (a > b) ? a : b;
}
static int altura (ArvnNo* r)
{
    if (r == NULL)
        return -1;
    else
        return max2(1+altura(r->prim), altura(r->prox));
}
int arvn_altura (Arvn* a)
{
    return altura(a->raiz);
}

```

Funções implementadas para árvores binárias que não diferenciam as árvores referenciadas podem ser diretamente aplicadas a árvores com número variável de filhos. Por exemplo, se tivermos implementada uma função para calcular o número de nós presentes em uma árvore binária, essa mesma função servirá para calcular o número de nós de uma árvore com número variável de filhos.

Exercícios

1. Considerando estruturas de árvores binárias que armazenam valores inteiros, implemente uma função que, dada uma árvore, retorne a quantidade de nós que guardam números pares. Essa função deve obedecer ao protótipo:

```
int pares (Arv* a);
```

2. Implemente uma função que retorne a quantidade de folhas de uma árvore binária. Essa função deve obedecer ao protótipo:

```
int folhas (Arv* a);
```

3. Implemente uma função que retorne a quantidade de nós de uma árvore binária que possuem apenas um filho. Essa função deve obedecer ao protótipo:

```
int um_filho (Arv* a);
```

4. Implemente uma função que compare se duas árvores binárias são iguais. Essa função deve obedecer ao protótipo:

```
int igual (Arv* a, Arv* b);
```

5. Implemente uma função que crie uma cópia de uma árvore binária. Essa função deve obedecer ao protótipo:

```
Arv* copia (Arv*a);
```

6. Implemente uma função que retorne o número de nós folhas maiores do que um valor x , considerando uma árvore binária de busca. Essa função deve obedecer ao protótipo:

```
int nfolhas_maiores (Arv*a, int x);
```

7. Implemente uma função que retorne o somatório dos valores entre x e y (considerando $x > y$) armazenados numa árvore binária de busca de valores inteiros. Essa função deve obedecer ao protótipo:

```
int soma_xy (Arv*a, int x, int y);
```

8. Implemente uma função que retorne o nível do nó que contém a informação x numa árvore binária de busca. Essa função deve obedecer ao protótipo:

```
int nivel (Arv*a, int x);
```

9. Considere uma estrutura para representar os dados de um aluno:

```
typedef struct aluno Aluno;
struct aluno {
    char nome[81];
    float p1, p2, p3; /* notas em provas */
};
```

Considere ainda uma árvore binária de busca que armazena informações de alunos, ordenada por ordem alfabética dos nomes:

```
typedef struct arv Arv;
typedef struct arvno ArvNo;
struct arvno {
    Aluno info;
    ArvNo* esq;
    ArvNo* dir;
};
struct arv {
    ArvNo* raiz;
}
```

Pede-se:

- (a) Implemente uma função para criar uma árvore vazia:

```
Arv* aa_cria (void);
```

- (b) Implemente uma função para inserir os dados de um novo aluno na estrutura:

```
void aa_insere (Arv* a, char* nome,
                 float p1, float p2, float p3);
```

- (c) Implemente uma função que, dado um nome de um aluno, retorne a média das notas das provas:

```
float aa_media (Arv* a, char* nome);
```

- (d) Implemente uma função para retirar um aluno da estrutura, dado o seu nome:

```
void aa_retira (Arv* a, char* nome);
```

- (e) Implemente uma função para liberar a estrutura da árvore de alunos:

```
void aa_libera (Arv* a);
```

10. Considerando estruturas de árvores com número variável de filhos que armazenam valores inteiros, implemente uma função que, dada uma árvore, retorne a quantidade de nós que guardam números pares. Essa função deve obedecer ao protótipo:

```
int pares (Arvn* a);
```

11. Implemente uma função que retorne a quantidade de folhas de uma árvore com número variável de filhos. Essa função deve obedecer ao protótipo:

```
int folhas (Arvn* a);
```

12. Considerando estruturas de árvores com número variável de filhos, implemente uma função que retorne a quantidade de nós que possuem apenas um filho. Essa função deve obedecer ao protótipo:

```
int um_filho (Arvn* a);
```

13. Implemente uma função que compare se duas árvores são iguais. Essa função deve obedecer ao protótipo:

```
Arvn* igual (Arvn* a, Arvn* b);
```

14. Implemente uma função que crie uma cópia de uma árvore. Essa função deve obedecer ao protótipo:

```
Arvn* copia (Arvn*a);
```