

SILVIO DO LAGO PEREIRA



# ESTRUTURAS DE DADOS EM C

UMA ABORDAGEM  
DIDÁTICA

 **Érica** | **Saraiva**



# ÁRVORES

Este capítulo introduz a estrutura de dados árvore, descreve as principais operações que esse tipo de dados suporta e mostra como implementá-la com alocação dinâmica encadeada.

## 13.1 Fundamentos

Uma *árvore*  $A$  é uma coleção de  $n \geq 0$  nós organizados de forma hierárquica. Se  $n = 0$ , então  $A$  é uma *árvore vazia*; senão,  $A$  tem as seguintes propriedades:

- Existe um nó especial em  $A$ , denominado *raiz* de  $A$ , a partir do qual todos os demais nós de  $A$  podem ser acessados.
- Os demais nós de  $A$  são divididos em  $k$  coleções disjuntas,  $A_1, A_2, \dots, A_k$ , denominadas *subárvores* de  $A$ .
- As subárvores  $A_1, A_2, \dots, A_k$  também são árvores.

Por definição, árvores são estruturas recursivas. Quando a raiz de uma árvore é removida, o que sobra é uma coleção de árvores. Por exemplo, quando a raiz  $a$  é removida da árvore da Figura 13.1, obtemos as árvores com raízes  $b$  e  $c$ , respectivamente. Então, cada nó numa árvore  $A$  é raiz de uma subárvore de  $A$ .

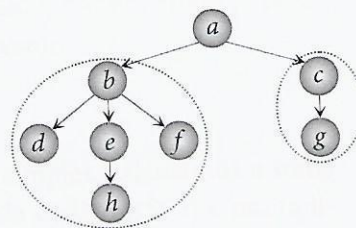


Figura 13.1 | Árvore e subárvores.

Seja  $A$  uma árvore com raiz  $r_0$  e subárvores  $A_1, A_2, \dots, A_k$ , cujas raízes são  $r_1, r_2, \dots, r_k$ , respectivamente. Então,  $r_0$  é *pai* de  $r_1, r_2, \dots, r_k$  ou, equivalentemente,  $r_1, r_2, \dots, r_k$  são *filhos* de  $r_0$ . A raiz  $r_0$  é o único nó de  $A$  que não tem pai. Um nó que não tem filhos é uma *folha*. Se uma árvore  $A$  tem apenas um nó, esse nó é raiz e folha de  $A$ , ao mesmo tempo. Por exemplo, na árvore da Figura 13.1, o nó  $a$  é pai dos nós  $b$  e  $c$ ; o nó  $g$  é filho do nó  $c$ ; o nó  $a$  é a raiz da árvore e os nós  $d, f, g$  e  $h$  são as folhas da árvore.

O número de filhos de um nó é o *grau do nó*. Por exemplo, na Figura 13.1, o grau do nó  $a$  é 2, o grau de  $b$  é 3, o grau de  $c$  e  $e$  é 1 e o grau dos demais nós é 0. O *grau de uma árvore* é o máximo grau de seus nós. Por exemplo, o grau da árvore na Figura 13.1 é 3, pois nessa árvore cada nó tem no máximo três filhos.

O *nível* da raiz de uma árvore é 1. O *nível* dos filhos de um nó num nível  $h$  é  $h + 1$ . A *altura* de uma árvore é o máximo nível de seus nós (ou 0, se a árvore é vazia). Por exemplo, a altura da árvore na Figura 13.1 é 4; a altura da subárvore enraizada em  $b$  é 3 e a altura da subárvore enraizada em  $c$  é 2.

Há várias aplicações de árvores em computação. Por exemplo, num sistema operacional, árvores são usadas para organizar arquivos em subdiretórios.

## 13.2 Árvores binárias

*Árvore binária* é uma árvore de grau 2, ou seja, uma árvore em que todo nó tem no máximo dois filhos. Numa árvore binária, há distinção entre as subárvores *esquerda* e *direita*. Por exemplo, as árvores na Figura 13.2 são distintas, pois uma tem a subárvore direita vazia e a outra tem a subárvore esquerda vazia.



Figura 13.2 | Duas árvores binárias distintas.

Para criar uma árvore binária, é preciso definir a *estrutura* de seus nós e o tipo de *ponteiro* que será usado para apontar a sua raiz, como na Figura 13.3.

```
#define fmt "%d " // formato de exibicao dos itens
typedef int Item; // tipo dos itens na arvore
typedef struct arv { // estrutura dos nos da arvore
    struct arv *esq;
    Item item;
    struct arv *dir;
} *Arv; // tipo de ponteiro para arvore
```

Figura 13.3 | Definições para criação de árvore binária.



O tipo `Item`, definido como `int`, indica que os itens na árvore binária serão números inteiros. A constante `fnt` indica o formato de exibição de um item da árvore e, portanto, deve ser compatível com o tipo `Item`.

O tipo `Arv`, definido como ponteiro para `struct arv`, serve para declarar um ponteiro para árvore binária (isto é, um ponteiro que aponta a *raiz* da árvore). Se `R` aponta uma árvore binária, `R->item` é o item na raiz dessa árvore, `R->esq` aponta a sua subárvore esquerda e `R->dir` aponta a sua subárvore direita.

### 13.2.1 Criação de árvores binárias

A função que cria um nó de árvore binária, definida na Figura 13.4, aloca uma estrutura de árvore (do tipo `struct arv`), inicia seus campos com os valores que recebe como parâmetros e, no fim, devolve o endereço dessa estrutura.

```
Arv arv(Arv e, Item x, Arv d) {
    Arv n = malloc(sizeof(struct arv));
    n->esq = e;
    n->item = x;
    n->dir = d;
    return n;
}
```

Figura 13.4 | Função para criação de um nó de árvore binária.

Por exemplo, a chamada `arv(NULL, 1, NULL)` cria uma árvore cuja raiz guarda o item 1 e cujas subárvores esquerda e direita são vazias (isto é, cria uma folha). Analogamente, a execução do comando a seguir cria a árvore da Figura 13.5.

```
Arv R = arv(arv(arv(NULL,
    4,
    NULL),
    2,
    arv(NULL,
    5,
    NULL)),
    1,
    arv(NULL,
    3,
    arv(NULL,
    6,
    NULL)));
```

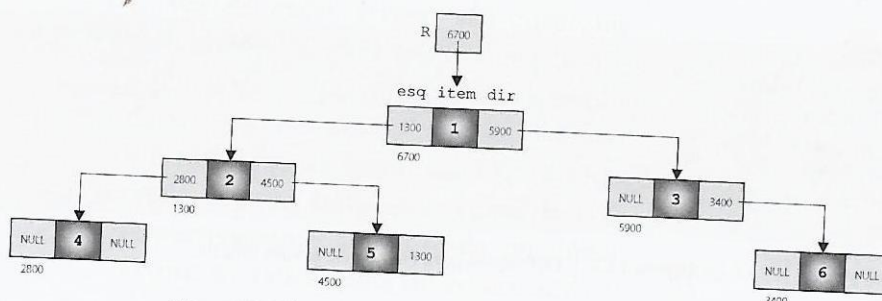


Figura 13.5 | Representação de árvore binária na memória.

### 13.2.2 Percursos em árvores binárias

Um *percurso* é uma forma sistemática de *visitar* e *processar* os nós de uma árvore.

O *percurso em largura* visita os nós de uma árvore, em nível, da esquerda para a direita. Por exemplo, o percurso em largura visita os nós da árvore na Figura 13.5 nessa ordem: 1, 2, 3, 4, 5, 6. Para percorrer uma árvore em largura, criamos uma fila contendo apenas o endereço de seu nó raiz. A partir daí, enquanto a fila não estiver vazia: o endereço de um nó é removido da fila, o item nesse nó é exibido e os endereços de seus filhos são inseridos na fila. A função para percurso em largura, definida na Figura 13.6, usa o tipo *Fila*, definido no Capítulo 4.

```
void emnivel(Arv A) {
    if( A==NULL ) return;
    Fila F = fila(MAX); // fila de Arv
    enfileira(A,F);
    while( !vaziaf(F) ) {
        Arv A = desenfileira(F);
        printf(fmt,A->item);
        if( A->esq != NULL ) enfileira(A->esq,F);
        if( A->dir != NULL ) enfileira(A->dir,F);
    }
}
```

Figura 13.6 | Função para percurso em largura (ou em nível).

Na função *emnivel()*, a constante *MAX* depende da altura da árvore percorrida. Para percorrer uma árvore binária de altura  $h > 0$ , com todos os níveis completos, *MAX* deve ser definido como  $2^{h-1}$ . Essa é uma limitação do percurso em largura.

Um *percurso em profundidade* pode ser de três tipos básicos:

- **Em-ordem:** percorre a subárvore esquerda, depois visita a raiz da árvore e, finalmente, percorre a subárvore direita. Por exemplo, o percurso em-ordem visita os nós da árvore na Figura 13.5 nessa ordem: 4, 2, 5, 1, 3, 6.
- **Pré-ordem:** visita a raiz da árvore, depois percorre a subárvore esquerda e, finalmente, percorre a subárvore direita. Por exemplo, o percurso pré-ordem visita os nós da árvore na Figura 13.5 nessa ordem: 1, 2, 4, 5, 3, 6.
- **Pós-ordem:** percorre a subárvore esquerda, depois percorre a subárvore direita e, finalmente, visita a raiz da árvore. Por exemplo, o percurso pós-ordem visita os nós da árvore na Figura 13.5 nessa ordem: 4, 5, 2, 6, 3, 1.

A vantagem dos percursos em profundidade é que eles podem ser facilmente implementados de forma recursiva. Além disso, enquanto o espaço de memória usado pelo percurso em largura cresce *exponencialmente* em função da altura da árvore percorrida, o espaço de memória usado pelos percursos em profundidade cresce apenas *linearmente* em função dessa altura. As funções para os percursos em profundidade são



definidas nas Figuras 13.7, 13.8 e 13.9. Nelas, o processamento dos nós consiste apenas em sua exibição no vídeo.

```
void emordem(Arv A) {  
    if( A==NULL ) return;  
    emordem(A->esq);  
    printf(fmt,A->item);  
    emordem(A->dir);  
}
```

Figura 13.7 | Função para percurso em-ordem.

```
void preordem(Arv A) {  
    if( A==NULL ) return;  
    printf(fmt,A->item);  
    preordem(A->esq);  
    preordem(A->dir);  
}
```

Figura 13.8 | Função para percurso pré-ordem.

```
void posordem(Arv A) {  
    if( A==NULL ) return;  
    posordem(A->esq);  
    posordem(A->dir);  
    printf(fmt,A->item);  
}
```

Figura 13.9 | Função para percurso pós-ordem.

### 13.2.3 Destruição de árvore binária

A função para destruição de árvore binária, definida na Figura 13.10, usa um percurso *pós-ordem*: primeiro ela destrói a subárvore esquerda, depois ela destrói a subárvore direita e, por fim, ela desaloca a raiz da árvore. Note que esse é o único percurso em profundidade que resolve o problema. Se, por exemplo, fosse usado um percurso pré-ordem, a raiz da árvore seria desalocada antes que suas subárvores fossem destruídas; portanto, não seria mais possível destruir as subárvores, pois os endereços de suas raízes seriam perdidos com o nó raiz.

```
void destroi(Arv *A) {  
    if( *A == NULL ) return;  
    destroi(&(*A)->esq);  
    destroi(&(*A)->dir);  
    free(*A);  
    *A = NULL;  
}
```

Figura 13.10 | Função para destruição de árvore binária.

De fato, para cada tipo de operação feita com uma árvore binária, em geral, há um único tipo de percurso que pode produzir o resultado desejado.

### 13.3 Árvores de busca binária

Uma *árvore de busca binária*  $A$  é uma árvore binária vazia ou, então, uma árvore binária cuja raiz armazena um item  $r$  e tem as seguintes propriedades:

- Todo item na subárvore esquerda de  $A$  é *menor ou igual* a  $r$ .
- Todo item na subárvore direita de  $A$  é *maior* que  $r$ .
- Cada subárvore de  $A$  é uma árvore de busca binária.

Uma consequência importante dessas propriedades é que a *projeção* de uma árvore de busca binária sempre produz uma sequência *ordenada* de itens. Por exemplo, a Figura 13.11 mostra uma árvore de busca binária e sua projeção.

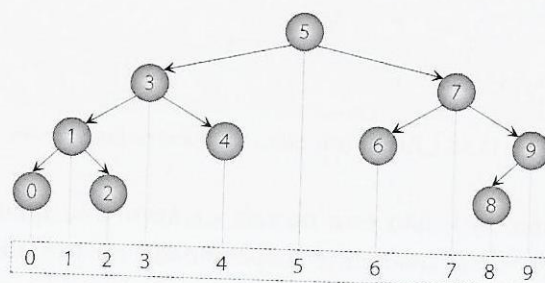


Figura 13.11 | Uma árvore de busca binária e sua projeção (ordenada).

#### 13.3.1 Inserção em árvore de busca binária

A função para inserir um novo item numa árvore de busca binária, definida na Figura 13.12, usa um percurso *pré-ordem*: primeiro ela verifica se a árvore está vazia e, caso esteja, ela coloca o novo item na raiz da árvore; se não, se o novo item for menor ou igual àquele na raiz da árvore, recursivamente, ela o insere na subárvore esquerda; caso contrário, recursivamente, ela o insere na subárvore direita. Note que, como o ponteiro para a árvore pode ser alterado durante uma inserção, ele deve ser passado por referência.

```
void ins(Item x, Arv *A) {
    if( *A == NULL ) *A = arv(NULL, x, NULL);
    else if( x <= (*A)->item ) ins(x, &(*A)->esq);
    else ins(x, &(*A)->dir);
}
```

Figura 13.12 | Função para inserção em árvore de busca binária.



A função `ins()` garante a criação de uma árvore de busca binária, independentemente da ordem em que suas chamadas são feitas no programa. Por exemplo, a árvore na Figura 13.11 pode ser criada e exibida do seguinte modo:

```
Arv R = NULL;  
ins(5, &R); ins(7, &R); ins(3, &R); ins(9, &R); ins(1, &R);  
ins(6, &R); ins(4, &R); ins(8, &R); ins(0, &R); ins(2, &R);  
emordem(R);
```

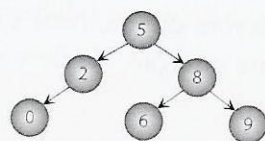
### 13.3.2 Busca em árvore de busca binária

A busca de um item numa árvore de busca binária também é simples. Dados um item  $x$  e um ponteiro  $A$  para uma árvore de busca binária, a função definida na Figura 13.13 devolve 1 (*verdade*), se  $x$  está em  $A$ , e 0 (*falso*) se  $x$  não está em  $A$ .

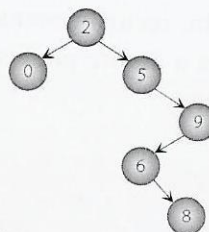
```
int busca(Item x, Arv A) {  
    if( A == NULL ) return 0;  
    if( x == A->item ) return 1;  
    if( x <= A->item ) return busca(x, A->esq);  
    else return busca(x, A->dir);  
}
```

Figura 13.13 | Função para busca em árvore de busca binária.

A cada comparação, se  $x$  não está na raiz da árvore  $A$ , apenas uma das subárvores de  $A$  precisa ser recursivamente inspecionada no próximo passo. Portanto, se  $A$  for uma árvore *balanceada* (isto é, se cada nó de  $A$  tem aproximadamente o mesmo número de descendentes à esquerda e à direita), a eficiência da busca em  $A$  é a mesma da busca binária (*vide* Capítulo 8). Por outro lado, se  $A$  for uma árvore *degenerada* (isto é, muito desbalanceada), a eficiência da busca em  $A$  é a mesma da busca linear (*vide* Capítulo 8). Na prática, se uma árvore de busca binária é criada pela inserção de itens em ordem aleatória, ela tende a ser balanceada. A Figura 13.14 mostra exemplos de árvores balanceada e degenerada.



(a) árvore balanceada



(b) árvore degenerada

Figura 13.14 | Árvores balanceada e degenerada.



### 13.3.3 Remoção em árvore de busca binária

A remoção em árvore de busca binária é mais difícil do que a inserção e a busca. Para simplificar, primeiro vamos considerar a remoção de um item *máximo* da árvore de busca binária. Para isso, partindo da raiz da árvore, seguimos sempre o ponteiro à direita, enquanto ele não for `NULL`. Quando chegarmos a um nó cujo ponteiro à direita é `NULL`, estaremos no nó que guarda um item máximo da árvore (Figura 13.14). Então, basta remover esse nó e devolver o item que ele armazena.

A função que remove um item máximo de uma árvore de busca binária é definida na Figura 13.15. Note que o nó que guarda um item máximo da árvore de busca binária *não* pode ter filho à direita (senão, ele não seria máximo), mas ele pode ter um filho à esquerda (Figura 13.14b). Portanto, quando esse nó é removido, o ponteiro que o apontava deve passar a apontar seu filho à esquerda (caso ele não tenha filho à esquerda, o ponteiro que o apontava ficará nulo).

```
Item remmax(Arv *A) {
    if( *A == NULL ) abort();
    while( (*A)->dir != NULL ) A = &(*A)->dir;
    Arv n = *A;
    Item x = n->item;
    *A = n->esq;
    free(n);
    return x;
}
```

Figura 13.15 | Função que remove e devolve um item máximo de uma árvore de busca binária.

Agora, vamos considerar a remoção de um item  $x$ , que está na *raiz* da árvore de busca binária  $A$ . Então, há três casos possíveis:

- Se o nó apontado por  $A$  *não tem filhos*, então ele deve ser desalocado e o ponteiro  $A$  deve ser anulado (isto é, a árvore fica vazia).
- Se o nó apontado por  $A$  *tem um único filho*, então ele deve ser desalocado e o ponteiro  $A$  deve passar a apontar esse filho.
- Se o nó apontado por  $A$  *tem dois filhos*, então o item nesse nó deve ser substituído por um item máximo removido de sua subárvore esquerda.

Os dois primeiros casos podem ser resolvidos diretamente e o terceiro caso pode ser facilmente resolvido com a função `remmax()`. Para entender a estratégia adotada nesse último caso, basta lembrar que, numa árvore de busca binária  $A$ , cada item na subárvore esquerda de  $A$  é menor que todo item na subárvore direita de  $A$ . Portanto, substituindo o item na raiz de  $A$  por um item máximo removido de sua subárvore esquerda, podemos garantir que a árvore resultante da remoção do item  $x$  ainda será uma árvore de busca binária (como mostra a Figura 13.16), ou seja:

- Todo nó à esquerda de A terá um item menor ou igual àquele em A.
- Todo nó à direita de A terá um item maior que aquele em A.

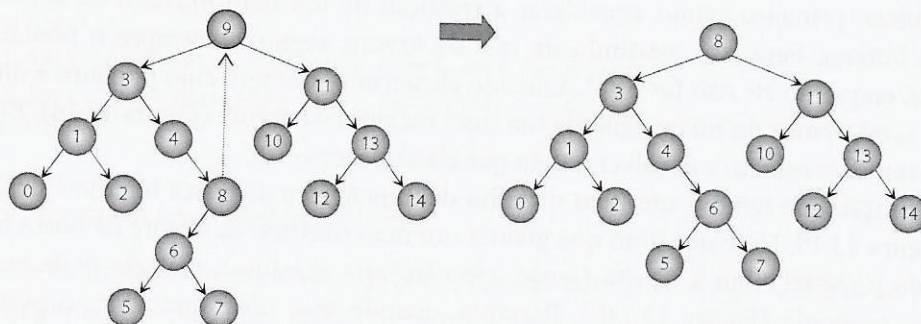


Figura 13.16 | Remoção de um nó com dois filhos.

Para generalizar essa ideia, precisamos considerar que o item a ser removido pode estar em qualquer nó da árvore, não apenas na raiz. Nesse caso, antes de removê-lo, precisamos encontrá-lo na árvore. Para isso, basta adaptar a lógica da função de busca em árvore (Figura 13.13). A função que remove um item  $x$  qualquer de uma árvore de busca binária  $A$  é definida na Figura 13.17.

```
void rem(Item x, Arv *A) {
    if( *A == NULL ) return;
    if(x == (*A)->item) {
        Arv n = *A;
        if( n->esq == NULL ) *A = n->dir;
        else if ( n->dir == NULL ) *A = n->esq;
        else n->item = remmax(&n->esq);
        if( n != *A ) free(n);
    }
    else if( x <= (*A)->item ) rem(x, &(*A)->esq);
    else rem(x, &(*A)->dir);
}
```

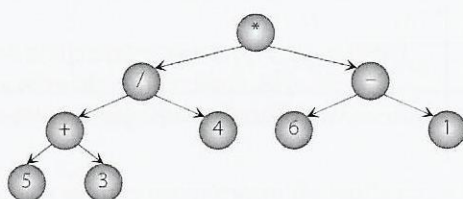
Figura 13.17 | Função para remoção em árvore de busca binária.

## Exercícios

- 13.1** Crie o arquivo `arv.h`, com os tipos e funções para árvores definidos nesse capítulo (exceto a função `emnivel()`, que depende do tipo `Fila`), e use esse arquivo num programa que cria e exibe a árvore da Figura 13.5.
- 13.2** Crie a função `nos(A)`, que devolve o total de *nós* na árvore binária  $A$ .
- 13.3** Crie a função `folhas(A)`, que devolve o total de *folhas* na árvore binária  $A$ .



- 13.4** Crie a função `altura(A)`, que devolve a *altura* da árvore binária *A*.
- 13.5** Crie a função `tem(A, x)`, que informa se a árvore binária *A* tem o item *x*.
- 13.6** Uma árvore *A* é *estritamente binária* se cada nó em *A* é uma folha ou tem dois filhos. Crie a função `eb(A)`, que informa se a árvore *A* é estritamente binária.
- 13.7** Duas árvores binárias *A* e *B* são *iguais* se elas têm a mesma forma e os mesmos itens. Crie a função `igual(A, B)`, que informa se *A* é igual a *B*.
- 13.8** Uma expressão aritmética pode ser representada por uma árvore binária cuja raiz é uma operação e cujas subárvores são operandos. Por exemplo, a expressão  $((5+3)/4) * (6-1)$  pode ser representada como na figura a seguir. Crie a função `valor(A)`, que avalia uma expressão aritmética representada por uma árvore binária *A* (cujos nós guardam números inteiros).



- 13.9** Crie a função `exibe_dec(A)`, que exibe os itens de uma árvore de busca binária em ordem decrescente.