Capítulo 14

Árvores binárias

As árvores da computação têm a tendência de crescer para baixo:
a raiz fica no ar enquanto as folhas se enterram no chão.

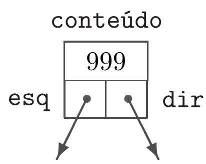
— folclore

Uma árvore binária é uma estrutura de dados mais geral que uma lista encadeada. Este capítulo introduz as operações mais simples sobre árvores binárias. O capítulo seguinte trata de uma aplicação básica.

14.1 Definição

É fácil transmitir a ideia intuitiva de árvore binária por meio de uma figura (veja Figura 14.1), mas é surpreendentemente difícil dar uma definição precisa do conceito. Uma árvore binária é um conjunto de registros (veja Apêndice E) que satisfaz certas condições, detalhadas adiante. Os registros serão chamados **nós** (poderiam também ser chamados **células**). Suporemos, por enquanto, que cada nó tem três campos: um número inteiro e dois ponteiros (veja Apêndice D) para nós. Os nós podem, então, ser definidos assim:

```
struct cel {
   int         conteúdo;
   struct cel *esq;
   struct cel *dir;
};
typedef struct cel nó;
```



O campo `conteúdo` é a "carga útil" do nó, enquanto os outros dois campos dão estrutura à árvore. O campo `esq` contém o endereço de um nó ou `NULL`. Hipótese análoga vale para o campo `dir`. Se o campo `esq` de um nó `X` é o endereço de um nó `Y`, diremos que `Y` é o **filho esquerdo** de `X`. Se `X.esq` = `NULL`, então `X` não tem filho esquerdo. Se `X.dir` = `&Y`, diremos que `Y` é o **filho direito** de `X`. Se `Y` é filho (esquerdo ou direito) de `X`, então `X` é **pai** de `Y`. Uma **folha** é um nó que não tem filho algum.

Um **ciclo** é qualquer sequência $(\mathtt{X}_0, \mathtt{X}_1, \ldots, \mathtt{X}_k)$ de nós tal que $\mathtt{X}_{i+1}$ é filho de $\mathtt{X}_i$ para $i = 0, 1, \ldots, k-1$ e $\mathtt{X}_0$ é filho de $\mathtt{X}_k$. Por exemplo, se `X.esq` = `&X` então (`X`) é um ciclo. Se `X.esq` = `&Y` e `Y.dir` = `&X` então (`X`, `Y`) é um ciclo.

Podemos agora definir o conceito central do capítulo. Uma **árvore binária** é um conjunto $\mathcal{A}$ de nós tal que (1) os filhos de cada elemento de $\mathcal{A}$ pertencem a $\mathcal{A}$, (2) todo elemento de $\mathcal{A}$ tem no máximo um pai, (3) um e apenas um dos elementos de $\mathcal{A}$ não tem pai em $\mathcal{A}$, (4) os filhos esquerdo e direito de cada elemento de $\mathcal{A}$ são distintos e (5) não há ciclos em $\mathcal{A}$. (Em geral, o programador não tem consciência dos detalhes dessa definição porque as árvores são construídas nó a nó de modo a satisfazer as condições naturalmente.) O único elemento de $\mathcal{A}$ que não tem pai em $\mathcal{A}$ é chamado **raiz** da árvore.

Suponha, por exemplo, que `P`, `X`, `Y` e `Z` são nós distintos, que `X` é filho esquerdo de `P`, que `Y` é filho esquerdo de `X`, que `Z` é filho direito de `X` e que `Y` e `Z` são folhas. Então o conjunto {`P`, `X`, `Y`, `Z`} é uma árvore binária. O conjunto {`X`, `Y`, `Z`} também é uma árvore binária.

Subárvores. Um **caminho** em uma árvore binária é qualquer sequência $(\mathtt{Y}_0, \mathtt{Y}_1, \ldots, \mathtt{Y}_k)$ de nós da árvore tal que $\mathtt{Y}_{i+1}$ é filho de $\mathtt{Y}_i$ para $i = 0, 1, \ldots, k-1$. Dizemos que $\mathtt{Y}_0$ é a **origem**, $\mathtt{Y}_k$ o **término** e k o **comprimento** do caminho. Um nó `Z` é **descendente** de um nó `X` se existe um caminho com origem `X` e término `Z`.

Para todo nó `X` de uma árvore binária, o conjunto formado por `X` e todos os seus descendentes é uma árvore binária. Dizemos que esta é a **subárvore** com raiz `X`. Se `P` é um nó, então `P.esq` é a raiz da **subárvore esquerda** de `P` e `P.dir` é a raiz da **subárvore direita** de `P`.

Endereço de uma árvore. O **endereço** de uma árvore binária é o endereço de sua raiz. (O endereço da árvore vazia é `NULL`.) Em discussões informais, é conveniente confundir árvores com seus endereços. Assim, se `r` é o endereço de uma árvore, podemos dizer "`r` é uma árvore" e "considere a árvore `r`". Isso sugere a introdução do nome alternativo `árvore` para o tipo de dados ponteiro–

para–nó:

```
typedef nó *árvore;
```

Recursão. A seguinte observação coloca em evidência a natureza recursiva das árvores binárias. Para toda árvore binária `r`, vale uma das seguintes alternativas:

1. `r` é `NULL` ou
2. `r->esq` e `r->dir` são árvores binárias.

Muitos algoritmos sobre árvores ficam mais simples quando escritos em estilo recursivo.

Exercícios

14.1.1 Dado o endereço `x` de um nó em uma árvore binária, considere a sequência de endereços que se obtém pela iteração das atribuições `x = x->esq` e `x = x->dir` em qualquer ordem. Mostre que esta sequência descreve um caminho.

14.1.2 Mostre que os nós de qualquer caminho em uma árvore binária são distintos dois a dois.

14.1.3 Sejam `X` e `Z` dois nós de uma árvore binária. Mostre que existe no máximo um caminho com origem `X` e término `Z`.

14.1.4 Sequências de parênteses. Árvores binárias têm uma relação muito íntima com certas sequências bem-formadas de parênteses (veja Seção 6.2). Discuta essa relação.

14.1.5 Expressões aritméticas. Árvores binárias podem ser usadas, de maneira muito natural, para representar expressões aritméticas (como $((a+b)*c-d)/(e-f)+g$, por exemplo). Discuta os detalhes desta representação.

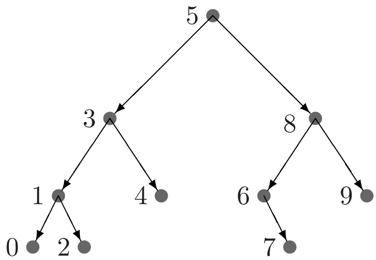


Figura 14.1: Uma árvore binária. Os nós da árvore estão numerados em ordem e-r-d.

14.2 Varredura esquerda-raiz-direita

Os nós de uma árvore binária podem ser visitados em muitas ordens diferentes. Cada ordem define uma **varredura** da árvore. Na varredura **e-r-d**, ou **esquerda-raiz-direita** (*inorder traversal*), visitamos

1. a subárvore esquerda da raiz, em ordem e-r-d,
2. depois a raiz,
3. depois a subárvore direita da raiz, em ordem e-r-d.

Eis uma função recursiva que faz a varredura e-r-d de uma árvore:

```
/* Recebe uma árvore binária r e imprime o conteúdo
 * de seus nós em ordem e-r-d. */
void Erd (árvore r) {
   if (r != NULL) {
      Erd (r->esq);
      printf ("%d\n", r->conteúdo);
      Erd (r->dir);
   }
}
```

A versão iterativa da função `Erd` usa uma pilha (veja Capítulo 6) de nós. A pilha é armazenada num vetor `p[0..t-1]` e há sempre um nó `x` pronto para ser colocado na pilha. A sequência de nós `p[0]`, `p[1]`, ..., `p[t-1]`, `x` é um roteiro do que ainda precisa ser feito: `x` representa a instrução "imprima a subárvore `x`" e cada `p[i]` representa a instrução "imprima o nó `p[i]` e em seguida a subárvore direita de `p[i]`".

```
/* Recebe uma árvore binária r e imprime o conteúdo de
 * seus nós em ordem e-r-d. Supõe que
 * a árvore não tem mais que 100 nós. */
void ErdI (árvore r) {
   nó *p[100], *x;
   int t = 0;
   x = r;
   while (x != NULL || t > 0) {
      /* o topo da pilha p[0..t-1] está em t-1 */
      if (x != NULL) {
         p[t++] = x;
```

```
            x = x->esq;
         }
         else {
            x = p[--t];
            printf ("%d\n", x->conteúdo);
            x = x->dir;
         }
      }
   }
```

As varreduras **r-e-d** (raiz-esquerda-direita ou *preorder traversal*) e **e-d-r** (esquerda-direita-raiz ou *postorder traversal*) são definidas por analogia com a varredura e-r-d.

Exercícios

14.2.1 Encontre um nó com conteúdo k em uma árvore binária.

```
                        5
                      3 5
                    1 3 5
                  0 1 3 5
                N 0 1 3 5
0                 N 1 3 5
0 1                 2 3 5
0 1               N 2 3 5
0 1 2               N 3 5
0 1 2 3               4 5
0 1 2 3             N 4 5
0 1 2 3 4             N 5
0 1 2 3 4 5             8
0 1 2 3 4 5           6 8
0 1 2 3 4 5         N 6 8
0 1 2 3 4 5 6         7 8
0 1 2 3 4 5 6       N 7 8
0 1 2 3 4 5 6 7       N 8
0 1 2 3 4 5 6 7 8       9
0 1 2 3 4 5 6 7 8     N 9
0 1 2 3 4 5 6 7 8 9     N
```

Figura 14.2: Função `ErdI` aplicada à árvore binária da Figura 14.1. Para simplificar, confundimos o conteúdo de cada nó com o seu endereço. Cada linha da tabela resume o estado de coisas no início de uma iteração: à esquerda estão os nós que já foram impressos; à direita está a pilha `x`, `p[t-1]`, ..., `p[0]`. A letra `N` representa `NULL`.

14.2.2 Calcule o número de nós de uma árvore binária.

14.2.3 Imprima as folhas de uma árvore binária em ordem e-r-d.

14.2.4 Verifique que o código abaixo é equivalente ao da função `ErdI`:

```
while (1) {
   while (x != NULL) {
      p[t++] = x;
      x = x->esq; }
   if (t == 0) break;
   x = p[--t];
   printf ("%d\n", x->conteúdo);
   x = x->dir; }
```

14.2.5 Escreva uma função que faça a varredura r-e-d de uma árvore binária. Escreva uma função que faça a varredura e-d-r de uma árvore binária.

14.2.6 Escreva uma função que receba uma árvore binária não vazia e devolva o endereço do primeiro nó da árvore na ordem e-r-d. Faça duas versões: uma iterativa e uma recursiva. Repita o exercício com “último” no lugar de “primeiro”.

14.2.7 Expressões aritméticas. Discuta a relação entre a varredura e-r-d e a notação infixa de expressões aritméticas. Discuta a relação entre a varredura e-d-r e a notação posfixa. (Veja Seção 6.3 e Exercício 14.1.5.)

14.3 Altura

A **altura de um nó** em uma árvore binária é a distância entre o nó e o seu descendente mais afastado. Mais precisamente, a altura de um nó é o comprimento do mais longo caminho que leva do nó até uma folha.

A **altura de uma árvore** é a altura de sua raiz. Por exemplo, uma árvore

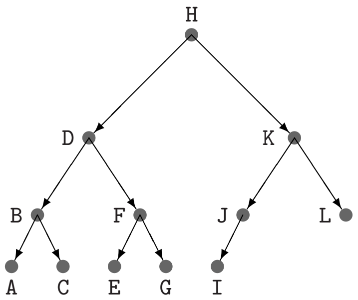


Figura 14.3: Árvore binária quase completa. (A ordem alfabética dos nós descreve uma varredura e-r-d.) A altura da árvore é $\lfloor \log_2 12 \rfloor$.

com um único nó tem altura 0 e a árvore da Figura 14.3 tem altura 3. A altura de uma árvore binária com n nós fica entre $\log_2 n$ e n: se h é a altura da árvore então

$$\lfloor \log_2 n \rfloor \;\leq\; h \;<\; n\,.$$

Uma árvore binária de altura $n-1$ é um "tronco sem galhos": cada nó tem no máximo um filho. Uma árvore binária de altura $\lfloor \log_2 n \rfloor$ é "completa" ou "quase completa": todos os "níveis" estão lotados exceto talvez o último. (Veja Exercício 1.2.4.)

Eis como a altura de uma árvore binária pode ser calculada:

```
/* Devolve a altura da árvore binária r. */
int Altura (árvore r) {
   if (r == NULL)
      return -1; /* a altura de uma árvore vazia é -1 */
   else {
      int he = Altura (r->esq);
      int hd = Altura (r->dir);
      if (he < hd) return hd + 1;
      else return he + 1;
   }
}
```

Árvores balanceadas. Uma árvore binária é **balanceada** se as subárvores esquerda e direita de cada nó tiverem aproximadamente a mesma altura. Uma árvore binária balanceada com n nós tem altura próxima de $\log_2 n$.

Muitos algoritmos sobre árvores binárias consomem tempo proporcional à altura da árvore. Por isso, convém trabalhar com árvores balanceadas. Mas é difícil manter o balanceamento se a árvore sofre inserção e remoção de nós ao longo da execução do algoritmo.

Exercícios

14.3.1 Desenhe uma árvore binária com 17 nós que tenha a menor altura possível.

14.3.2 Escreva uma função iterativa que calcule a altura de uma árvore binária.

14.3.3 Árvores AVL. Uma árvore é balanceada *no sentido AVL* se, para cada nó `x`, as alturas das subárvores esquerda e direita de `x` diferem em no máximo uma unidade. Escreva uma função que decida se uma dada árvore é balanceada no sentido AVL. Procure escrever sua função de modo que ela visite cada nó no máximo uma vez.

14.4 Nós com campo pai

Em algumas aplicações (veja seção seguinte, por exemplo) é conveniente ter acesso imediato ao pai de qualquer nó. Para isso, é preciso acrescentar um campo `pai` a cada nó:

```
struct cel {
   int         conteúdo;
   struct cel *pai;
   struct cel *esq;
   struct cel *dir;
};
typedef struct cel nó;
```

É um bom exercício escrever uma função que preencha o campo `pai` de todos os nós de uma árvore binária.

Exercícios

14.4.1 Escreva uma função que preencha corretamente todos os campos `pai` de uma árvore binária.

14.4.2 A **profundidade** de um nó em uma árvore binária é a distância entre o nó e a raiz da árvore. Mais precisamente, a profundidade de um nó `X` é o comprimento do (único) caminho que vai da raiz até `X`. Por exemplo, a profundidade da raiz é 0 e a profundidade de qualquer filho da raiz é 1. Escreva uma função que determine a profundidade de um nó dado.

14.4.3 É verdade que uma árvore binária é balanceada se e somente se todas as suas folhas têm aproximadamente a mesma profundidade?

14.4.4 Escreva uma função que imprima o conteúdo de cada nó de uma árvore binária precedido de um recuo em relação à margem esquerda do papel. Esse recuo deve ser proporcional à profundidade do nó. Veja Figura 14.4.

14.4.5 HEAP. Em que condições uma árvore binária pode ser considerada um heap (veja Seção 10.1)? Escreva uma função que transforme um max-heap em uma árvore binária quase completa. Escreva uma versão da função `SacodeHeap` (Seção 10.3) para um max-heap representado por uma árvore binária.

14.5 Nó seguinte

Suponha que `x` é o endereço de um nó de uma árvore binária. Queremos calcular o endereço do nó seguinte na ordem e-r-d. Para resolver o problema, é necessário

```
555
   333
      111
         -
         -
      444
         -
         -
   888
      -
      999
         -
         -
```

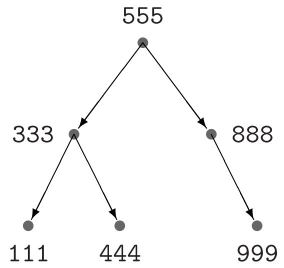


Figura 14.4: O lado esquerdo da figura é uma representação da árvore binária que está à direita. O número de espaços que precede o conteúdo de cada nó é proporcional à profundidade do nó. Os caracteres `'-'` representam `NULL`. Veja Exercício 14.4.4.

que os nós tenham um campo `pai`, conforme a seção anterior. A função abaixo devolve o endereço do nó seguinte a `x` ou devolve `NULL` se `x` é o último nó.

(Às vezes convém confundir, a título de atalho verbal, um nó com o seu endereço. Na documentação da função abaixo, por exemplo, a expressão "recebe um nó `x`" deve ser entendida como "recebe o endereço `x` de um nó". Analogamente, a expressão "devolve o nó seguinte" deve ser entendida como "devolve o endereço do nó seguinte".)

```
/* Recebe um nó x de uma árvore binária cujos nós têm
 * campo pai e devolve o nó seguinte na ordem e-r-d.
 * A função supõe que x != NULL. */
nó *Seguinte (nó *x) {
   if (x->dir != NULL) {
      nó *y = x->dir;
      while (y->esq != NULL) y = y->esq;
      return y; /* 1 */
   }
   while (x->pai != NULL && x->pai->dir == x) /* 2 */[1]
      x = x->pai; /* 3 */
   return x->pai;
}
```

[1] A expressão `x->pai->dir` equivale a `(x->pai)->dir`, conforme o Seção J.5.

Na linha `1` da função `Seguinte`, `y` é o primeiro nó, na ordem e-r-d, da subárvore direita de `x`. As linhas `2` e `3` fazem com que `x` suba na árvore enquanto for filho direito de alguém.

Exercícios

14.5.1 Escreva uma função que receba um nó `x` de uma árvore binária e encontre o nó anterior a `x` na ordem e-r-d.

14.5.2 Escreva uma função que faça varredura e-r-d de uma árvore binária usando a função `Seguinte` e a função sugerida no Exercício 14.2.6.

14.5.3 Leia o verbete *Binary tree* na Wikipedia [21].