# Árvores binárias

As árvores da computação têm a curiosa tendência
de crescer para baixo...

Uma árvore binária é uma estrutura de dados mais geral que uma lista encadeada. Este capítulo introduz algumas operações básicas sobre árvores binárias. O capítulo seguinte, *Árvores binárias de busca*, trata de uma aplicação fundamental.

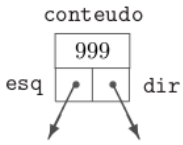
Sumário:


- Nós e seus filhos
- Árvores e subárvores
- Endereço de uma árvore e definição recursiva
- Varredura esquerda-raiz-direita
- Altura e profundidade
- Nós com campo pai
- Primeiro e último nós
- Nó seguinte e anterior (sucessor e predecessor)


## Nós e seus filhos

Uma árvore binária (= *binary tree*) é um conjunto de registros que satisfaz certas condições. As condições não serão dadas explicitamente, mas elas ficarão implicitamente claras no contexto. Os registros serão chamados *nós* (poderiam também ser chamados *células*). Cada nó tem um endereço. Suporemos, por enquanto, que cada nó tem apenas três campos: um número inteiro e dois ponteiros para nós. Os nós podem, então, ser definidos assim:

```
typedef struct reg {
   int  conteudo; // conteúdo
   noh *esq;
   noh *dir;
} noh; // nó
```

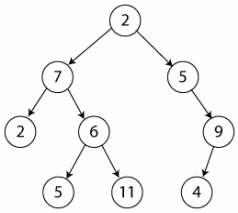



O campo `conteudo` é a carga útil do nó; os dois outros campos servem apenas para dar estrutura à árvore. O campo `esq` de cada nó contém `NULL` ou o endereço de outro nó. Uma hipótese análoga vale para o campo `dir`.

Se o campo `esq` de um nó `P` é o endereço de um nó `E`, diremos que `E` é o *filho esquerdo* de `P`. Analogamente, se `P.dir` é igual a `&D`, diremos que `D` é o *filho direito* de `P`. Se um nó `F` é filho (esquerdo ou direito) de `P`, diremos que `P` é o *pai* de `F`. Uma *folha* (= *leaf*) é um nó que não tem filho algum.

É muito conveniente confundir, verbalmente, cada nó com seu endereço. Assim, se `x` é um ponteiro para um nó (ou seja, se `x` é do tipo `*noh`), dizemos "considere o nó `x`" em lugar de "considere o nó cujo endereço é `x`".

A figura abaixo mostra dois exemplos de árvores binárias. Do lado esquerdo, temos uma curiosa árvore binária na natureza. Do lado direito, uma representação esquemática de uma árvore binária cujos nós contêm os números 2, 7, 5, etc.


## Exercícios 1

1. Dê uma lista das condições que um conjunto de nós deve satisfazer para ser considerado uma árvore binária.

# Árvores e subárvores

Suponha que `x` é um nó. Um *descendente* de `x` é qualquer nó que possa ser alcançado pela iteração das instruções `x = x->esq` e `x = x->dir` em qualquer ordem. (É claro que essas instruções só podem ser iteradas enquanto `x` for diferente de `NULL`. Estamos supondo que `NULL` é de fato atingido mais cedo ou mais tarde.)

Um nó `x` juntamente com todos os seus descendentes é uma *árvore binária*. Dizemos que `x` é a *raiz* (= *root*) da árvore. Se `x` tiver um pai, essa árvore é *subárvore* de alguma árvore maior. Se `x` é `NULL`, a árvore é *vazia*.

Para qualquer nó `x`, o nó `x->esq` é a raiz da *subárvore esquerda* de `x` e `x->dir` é a raiz da *subárvore direita* de `x`.

# Endereço de uma árvore e definição recursiva

O *endereço* de uma árvore binária é o endereço de sua raiz. É conveniente confundir, verbalmente, árvores com seus endereços: dizemos "considere a árvore `r`" em lugar de "considere a árvore cuja raiz tem endereço `r`". Essa convenção sugere a introdução do nome alternativo "`arvore`" para o tipo-de-dados ponteiro-para-nó:

```
typedef noh *arvore; // árvore
```

Dada essa convenção, podemos dar uma definição recursiva de árvore binária: um ponteiro-para-nó `r` é uma árvore binária se

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 2

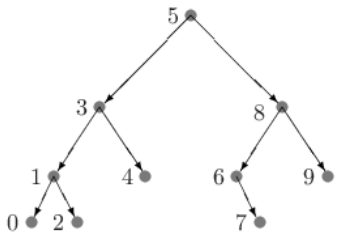
1. SEQUÊNCIAS DE PARENTESES. Árvores binárias têm uma relação muito íntima com certas sequências bem-formadas de parênteses. Discuta essa relação.
2. ★ EXPRESSÕES ARITMÉTICAS. Árvores binárias podem ser usadas, de maneira muito natural, para representar expressões aritméticas (como `((a+b)*c-d)/(e-f)+g`, por exemplo). Discuta os detalhes.

# Varredura esquerda-raiz-direita

Uma árvore binária pode ser percorrida de muitas maneiras diferentes. Uma maneira particularmente importante é a esquerda-raiz-direita, ou e-r-d, também conhecida como *inorder traversal*, ou varredura infixa, ou varredura central. A *varredura e-r-d* visita

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

nessa ordem. Na seguinte figura, os nós estão numeradas na ordem em que são visitados pela varredura e-r-d.




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

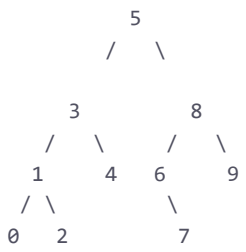
```
// Recebe a raiz r de uma árvore binária e
// imprime os conteúdos dos seus nós
// em ordem e-r-d.

void erd (arvore r) {
   if (r != NULL) {
      erd (r->esq);
      printf ("%d\n", r->conteudo);
      erd (r->dir);
   }
}
```

É um excelente exercício escrever uma versão iterativa dessa função. A versão abaixo usa uma pilha de nós. A sequência de nós na pilha é uma espécie de "roteiro" daquilo que ainda precisa ser feito: cada nó `x` na pilha é um lembrete de que ainda precisamos imprimir o conteúdo de `x` e o conteúdo da subárvore direita de `x`. O elemento no topo da pilha pode ser um `NULL`, mas os demais elementos são diferentes de `NULL`.

```
void erd_i (arvore r) {
   criapilha ();  // pilha de nós
   empilha (r);
   while (true) {
      x = desempilha ();
      if (x != NULL) {
         empilha (x);
         empilha (x->esq);
      }
      else {
         if (pilhavazia ()) break;
         x = desempilha ();
         printf ("%d\n", x->conteudo);
         empilha (x->dir);
      }
   }
   liberapilha ();
}
```

(O código ficaria ligeiramente mais simples — mas um pouco menos legível — se manipulasse a pilha diretamente.) A figura abaixo dá um exemplo de execução da função `erd_i`. Cada linha da tabela resume o estado das coisas no início de uma iteração: à esquerda está a pilha e à direita os nós impressos. O valor `NULL` é indicado por "`-`".




| pilha | |
|---|---|
| 5 | |
| 5 3 | |
| 5 3 1 | |
| 5 3 1 0 | |
| 5 3 1 0 - | |
| 5 3 1 - | 0 |
| 5 3 2 | 1 |
| 5 3 2 - | |
| 5 3 - | 2 |
| 5 4 | 3 |
| 5 4 - | |
| 5 - | 4 |
| 8 | 5 |
| 8 6 | |
| 8 6 - | |
| 8 7 | 6 |
| 8 - | 7 |
| 9 | 8 |
| 9 - | |
| - | 9 |

Os exercícios abaixo discutem duas outras ordens de varredura de uma árvore binária: a *varredura r-e-d*, que percorre a árvore em ordem raiz-esquerda-direita, e a *varredura e-d-r*, que percorre a árvore em ordem esquerda-direita-raiz. A primeira também é conhecida como *preorder traversal*, ou varredura em pré-ordem, ou varredura prefixa. A segunda também é conhecida como *postorder traversal*, ou varredura em pós-ordem, ou varredura posfixa. (A propósito, veja abaixo o exercício sobre notações infixa, posfixa e prefixa.)

## Exercícios 3

1. Considere a função `erd_i`. É verdade que a sequência de nós na pilha é um caminho que começa em algum nó e segue os ponteiros esquerdo e direito em alguma ordem?

2. Verifique que o código abaixo é equivalente ao da função `erd_i`. O código usa uma pilha de nós, todos diferentes de `NULL`, e mais um nó `x` (que é candidato a entrar na pilha mas pode ser `NULL`).

```
void erd_i (arvore r) {
   noh *x = r;
   criapilha ();
   while (true) {
      while (x != NULL) {
         empilha (x);
         x = x->esq; }
      if (pilhavazia ()) break;
      x = desempilha ();
      printf ("%d\n", x->conteudo);
      x = x->dir; }
   liberapilha (); }
```

3. Número de nós. Escreva uma função que calcule o número de nós de uma árvore binária.

4. Folhas. Escreva uma função que imprima, em ordem e-r-d, os conteúdos das *folhas* de uma árvore binária.

5. Dada uma árvore binária, encontrar o primeiro nó, na ordem e-r-d, cujo conteúdo tenha um dado valor *val*.

6. Varredura r-e-d. Escreva uma função que faça a varredura r-e-d (varredura prefixa) de uma árvore binária. (A varredura r-e-d também é conhecida como busca em profundidade ou *depth-first search*.)

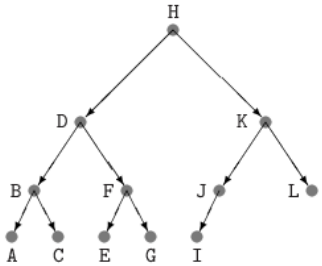
7. Varredura e-d-r. Escreva uma função que faça varredura e-d-r (varredura posfixa) de uma árvore binária.

8. Notações infixa, posfixa e prefixa. Mostre que a varredura e-r-d da árvore de uma expressão aritmética corresponde exatamente à representação infixa da expressão. Mostre que a varredura e-d-r da árvore de uma expressão aritmética corresponde exatamente à representação da expressão em notação posfixa. Mostre que a varredura r-e-d da árvore de uma expressão aritmética corresponde exatamente à notação prefixa.

## Altura e profundidade

A *altura de um nó* `x` em uma árvore binária é a distância entre `x` e o seu descendente mais afastado. Mais exatamente, a altura de `x` é o número de passos no mais longo caminho que leva de `x` até uma folha. Os caminhos a que essa definição se refere são os obtido pela iteração das instruções `x = x->esq` e `x = x->dir`, em qualquer ordem.

A *altura* (= *height*) *de uma árvore* é a altura da raiz da árvore. Uma árvore com um único nó tem altura 0. A árvore da figura tem altura 3.




Veja como a altura de uma árvore com raiz `r` pode ser calculada:

```
// Devolve a altura da árvore binária
// cuja raiz é r.

int altura (arvore r) {
   if (r == NULL)
      return -1; // altura da árvore vazia
   else {
      int he = altura (r->esq);
      int hd = altura (r->dir);
```

```
        if (he < hd) return hd + 1;
        else return he + 1;
      }
    }
```

Qual a relação entre a altura, digamos $h$, e o número de nós de uma árvore binária? Se a árvore tem $n$ nós então

$$\lg(n) \le h \le n-1,$$

onde $\lg(n)$ denota $\lfloor \log n \rfloor$, ou seja, o piso de $\log n$. Uma árvore binária de altura $n-1$ é um "tronco sem galhos": cada nó tem no máximo um filho. No outro extremo, uma árvore de altura $\lg(n)$ é "quase completa": todos os "níveis" estão lotados exceto talvez o último.

| n | lg(n) |
|---|---|
| 4 | 2 |
| 5 | 2 |
| 6 | 2 |
| 10 | 3 |
| 64 | 6 |
| 100 | 6 |
| 128 | 7 |
| 1000 | 9 |
| 1024 | 10 |
| 1000000 | 19 |

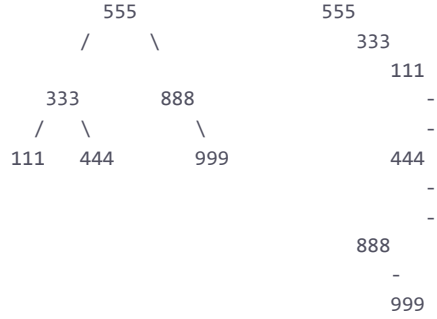
Uma árvore binária é *balanceada* (ou *equilibrada*) se, em cada um de seus nós, as subárvores esquerda e direita tiverem aproximadamente a mesma altura. Uma árvore binária balanceada com $n$ nós tem altura próxima de $\log n$. (A árvore do exemplo acima é balanceada). Sempre que possível, convém trabalhar com árvores que são balanceadas. Mas isso não é fácil se a árvore aumenta e diminui ao longo da execução do seu programa.

**Profundidade.** A *profundidade* (= *depth*) de um nó `s` em uma árvore binária com raiz `r` é a distância de `r` a `s`. Mais exatamente, a profundidade de `s` é o comprimento do único caminho que vai de `r` até `s`. Por exemplo, a profundidade de `r` é 0 e a profundidade de `r->esq` é 1.

Uma árvore é balanceada se todas as suas folhas têm aproximadamente a mesma profundidade.

## Exercícios 4

1. Desenhe uma árvore binária que com 17 nós que tenha a menor altura possível. Repita com a maior altura possível.
2. Escreva uma função iterativa para calcular a altura de uma árvore binária.
3. Escreva uma função que imprima o conteúdo de cada nó de uma árvore binária. Faça uma indentação (recuo de margem) proporcional à profundidade do nó. Segue um exemplo de árvore e sua representação (os caracteres `'-'` representam `NULL`):



-
-

4. VARREDURA POR NÍVEIS. A *varredura por níveis* (= *level-traversal*), ou *busca em largura* (= *breadth-first search*), de uma árvore binária visita todos os nós à profundidade 0, depois todos os nós à profundidade 1, depois todos os nós à profundidade 2, e assim por diante. Para cada profundidade, os nós são visitados "da esquerda para a direita". Escreva uma função que imprima os conteúdos dos nós de uma árvore na ordem da varredura por níveis. (Dica: Use uma fila de nós. Veja o cálculo de distâncias em um grafo.)
5. Digamos que $h$ é a altura e $p$ é a profundidade de um nó `x` em uma árvore binária. É verdade que $h + p$ é igual à altura da árvore?
6. Escreva uma função que determine a profundidade de um nó dado.
7. Escreva uma função que decida se uma dada árvore binária é quase completa.
8. DE HEAP PARA ÁRVORE. Escreva uma função que transforme um heap `v[1..n]` em uma árvore binária (quase completa).
9. DE ÁRVORE PARA HEAP. Uma árvore binária é *hierárquica* se `x->conteudo ≥ x->esq->conteudo` e `x->conteudo ≥ x->dir->conteudo` para todo nó `x`, desde que os filhos existam. É possível transformar qualquer uma árvore hierárquica quase completa em heap?
10. HIERARQUIZAÇÃO. Uma árvore binária é *esquerdista* se, para todo nó `x`, `x->esq == NULL` implica em `x->dir == NULL`. Uma árvore binária é *hierárquica* se `x->conteudo ≥ x->esq->conteudo` e `x->conteudo ≥ x->dir->conteudo` para todo nó `x`, desde que os filhos existam. Escreva uma função que movimente os campos `conteudo` de uma árvore esquerdista de modo a torná-la hierárquica. (Sugestão: Imite a função que transforma um vetor em heap. Em particular, imite a função `peneira`.)
11. ÁRVORE AVL. Uma árvore é balanceada *no sentido AVL* se, para cada nó `x`, as alturas das subárvores que têm raízes `x->esq` e `x->dir` diferem de no máximo 1. 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.

## Nós com campo pai

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

```
typedef struct reg {
   int         conteudo;
   struct reg *pai;
   struct reg *esq, *dir;
} noh;
```

Como a raiz da árvore não tem pai, é preciso tomar uma decisão de projeto a respeito do valor do campo `pai` nesse caso. Se `r` é a raiz da árvore, poderíamos dizer `r->pai == NULL`. Mas é melhor adotar a convenção

```
r->pai = r
```

quando `r` é a raiz. É claro que `r` será o único nó da árvore a ter essa propriedade.

## Exercícios 5

1. Escreva uma função que preencha corretamente todos os campos `pai` de uma árvore binária.
2. FILA DE PRIORIDADES. Escreva uma implementação de fila de prioridades que use como estrutura de dados subjacente uma árvore hierárquica com campos `pai`. Uma *árvore hierárquica* é uma binária quase completa tal que `x->conteudo ≥ x->esq->conteudo` e `x->conteudo ≥ x->dir->conteudo` para todo nó `x` (desde que os filhos

existam). A implementação deve ter funções para criar uma fila vazia, retirar um elemento máximo da fila, e inserir um elemento na fila. (Dica: veja a implementação de fila de prioridades baseada em heap.)

## Primeiro e último nós

Considere o seguinte problema sobre uma árvore binária: encontrar o *primeiro* nó da árvore na ordem e-r-d. É claro que o problema só faz sentido se a árvore não é vazia. Eis uma função que resolve o problema:

```
// Recebe uma árvore binária não vazia r
// e devolve o primeiro nó da árvore
// na ordem e-r-d.

noh *primeiro (arvore r) {
    while (r->esq != NULL)
       r = r->esq;
    return r;
}
```

Não é difícil fazer uma função análoga que encontre o *último* nó na ordem e-r-d.

## Nó seguinte e anterior (sucessor e predecessor)

Digamos que x é um nó de uma árvore binária. Nosso problema: calcular o nó seguinte na ordem e-r-d.

Para resolver o problema, é necessário que os nós tenham um campo `pai`. A função a seguir resolve o problema. É claro que a função só deve ser chamada com x diferente de `NULL`. A função devolve o nó seguinte a x ou devolve `NULL` se x é o último nó.

```
// Recebe o endereço de um nó x. Devolve o endereço
// do nó seguinte na ordem e-r-d ou NULL se não
// houver nó seguinte. A função supõe que x != NULL.

noh *seguinte (noh *x) {
    if (x->dir != NULL) {
       noh *y = x->dir;
       while (y->esq != NULL) y = y->esq;
       return y;                              // A
    }
    while (x->pai != NULL && x->pai->dir == x) // B
       x = x->pai;                             // B
    return x->pai;
}
```

(Note que a função não precisa saber onde está a raiz da árvore.) Na linha A, y é o primeiro nó, na ordem e-r-d, da subárvore cuja raiz é `x->dir`. As linhas B fazem com que x suba na árvore enquanto for filho direito de alguém.

## Exercícios 6

1. Escreva uma versão recursiva da função `primeiro`.
2. Escreva uma função que encontre o último nó na ordem e-r-d.

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

4. Escreva uma função que faça a varredura e-r-d de uma árvore binária usando as funções `primeiro` e `seguinte`.

---

Atualizado em 2018-08-07
https://www.ime.usp.br/~pf/algoritmos/

*DCC-IME-USP*