Árvores binárias

Este é um resumo de parte das seções 5.4 (Trees, p.216), 5.6 (Tree traversal, p.230), 5.7 (Recursive binary-tree algorithms, p.235) e 12.5 (Binary Search Trees) do livro do Sedgewick.

Definição

Uma **árvore binária** (= *binary tree*) é formada de nós; cada nó tem um certo conteúdo (por exemplo, um número inteiro) e os endereços (das raízes) de duas subárvores: uma *esquerda* e uma *direita*. Eis um exemplo de nó:

```
typedef struct node *link;

struct node {
   int  item;  // conteúdo do nó
   link l, r;  // 'l' de "left" e 'r' de "right"
} ;
```

Termos técnicos importantes: *raiz* de uma árvore, *filho* de um nó, *pai* de um nó, *folha* de uma árvore, *nó interno* de uma árvore, *nível* de um nó.

Em geral, quando dizemos "um nó `x`" devemos entender que `x` é o endereço de um nó. Nesses termos, o filho esquerdo de um nó `x` é `x->l` e o filho direito é `x->r`. Um nó `x` é uma folha se não tem filhos, ou seja, se `x->l` e `x->r` valem `NULL`.

Para ilustrar o conceito de árvore, eis uma pequena função (veja programa 5.17, p.236, do Sedgewick) que calcula o número de nós de uma árvore binária.

```
// Esta função devolve o número de nós
// da árvore binária cuja raiz é h.

int count(link h) {

    if (h == NULL) return 0;
    return count(h->l) + count(h->r) + 1;
}
```

Exercícios

1. [Sedg 5.59, p.225] Escreva uma função recursiva que receba uma árvore binária `ab` e um número `x` e remova da árvore todas as folhas que tenham `item` igual a `x`.

Altura de um nó e altura de uma árvore

A **altura** (= *height*) *de um nó* `h` em uma árvore binária é a "distância" entre `h` e o seu descendente mais afastado. Mas precisamente, a altura de `h` é o número de links no mais longo caminho que leva de `h` até uma folha. Os caminhos a que essa definição se refere são os obtido pela iteração dos comandos `x = x->l` e `x = x->r`, em qualquer ordem. Exemplo: a altura de uma folha é `0`.

A função abaixo (veja programa 5.17, p.236, do Sedgewick) calcula a altura de um nó `h`. Ela dá uma resposta razoável até mesmo quando `h` é `NULL`.

```
// Devolve o altura de um nó h em uma árvore binária.

int height(link h) {

    int u, v;
    if (h == NULL) return -1;
    u = height(h->l);
    v = height(h->r);
    if (u > v) return u+1;
```

```
    else return v+1;
}
```

A **altura de uma árvore** é a altura de sua raiz. A altura de uma árvore com `N` nós pode variar de `lg(N)` até `N-1`. (Como de hábito, `lg` é uma abreviatura de $\log_2$.)

Exercícios

2. Escreva uma função não recursiva que calcule o número de nós de uma árvore binária.
3. Mostre que toda árvore binária com `N` nós tem altura maior ou igual ao piso de `lg(N)`.
4. [Profundidade] A **profundidade** (= *depth*) de um nó `x` em uma árvore binária com raiz `h` é a "distância" `x` e `h`. Mais precisamente, a profundidade de `x` é o comprimento do (único) caminho que vai de `h` até `x`. Por exemplo, a profundidade de `h` é `0` e a profundidade de `h->l` é `1`. Escreva uma função que determine a profundidade de um nó dado em relação à raiz da árvore.
5. Suponha que cada nó da árvore tem um campo `depth` do tipo `int`. Preencha o campo de cada nó com a altura do nó.
6. [Códigos de nós] Um caminho que vai da raiz de uma árvore até um nó pode ser representado por uma sequência de `0`s e `1`s: toda vez que o caminho "desce para a esquerda" temos um `0`; toda vez que "desce para a direita" temos um `1`. Diremos que essa sequência de `0`s e `1`s é o **código do nó**. Suponha agora que todo nó de nossa árvore tem um campo adicional `cod` capaz de armazenar uma cadeia de caracteres. Escreva uma função que preencha o campo `cod` de cada nó com o código do nó.
7. [Reconstrução] Suponha dados os códigos de todas as folhas de uma árvore binária. Escreva uma função que reconstrua a árvore a partir desses códigos das folhas.

Como percorrer uma árvore

O seguinte exemplo é uma versão simplificada do programa 5.14, p.231, do Sedgewick. Ele imprime o `item` de cada nó da árvore binária cuja raiz é `h` (o "h" é inicial de "here").

```
// Imprime o item de cada nó de uma árvore binária h,
// que tem nós do tipo node.


void imprime (link h) {
    if (h == NULL) return;
    printf("%d\n", h->item);
    imprime(h->l);
    imprime(h->r);
}
```

Essa função percorre a árvore em ordem **raiz-esquerda-direita** (= *preorder*). Se as três últimas instruções forem trocadas por

```
    imprime(h->l);
    printf("%d\n", h->item);
    imprime(h->r);
```

a árvore será percorrida em ordem **esquerda-raiz-direita** (= *inorder*). Se as três últimas instruções forem trocadas por

```
    imprime(h->l);
    imprime(h->r);
    printf("%d\n", h->item);
```

a árvore será percorrida em ordem **esquerda-direita-raiz** (= *postorder*).

Exercícios

8. Escreva uma função que encontre o nó de uma árvore binária cujo `item` tem um dado valor.

9. [Sedg 5.86] Escreva uma função que calcule o número de folhas de uma árvore binária. Faça três versões: uma que percorra a árvore em *inorder*, outra que percorra a árvore em *preorder* e outra que percorra a árvore em *postorder*.

Versão não recursiva dos algoritmos de percurso

Abaixo temos uma versão simplificada do programa 5.15, p.233, de Sedgewick. Ela recebe uma árvore `h` não vazia (ou seja, `h != NULL`) e imprime o conteúdo de cada nó. Nossa solução usa as funções de manipulação de pilha que discutimos em outro capítulo. Ela supõe que a árvore não é vazia e tem 100 nós ou menos; na verdade, basta apenas que a altura da árvore não passe de 100.

```
// Imprime o item de cada nó de uma árvore binária h.
// A função supõe que h != NULL e que a altura da árvore
// não passa de 100.
//
void imprime_red (link h) {
    STACKinit(100);
    STACKpush(h);
    while (!STACKempty()) {
       h = STACKpop();
       printf("%d\n", h->item);
       if (h->r != NULL) STACKpush(h->r);
       if (h->l != NULL) STACKpush(h->l);
    }
}
```

Esta função percorre a árvore na ordem raiz-esquerda-direita, ou seja, em *preorder*. Ela usa uma pilha de nós (todos diferentes de `NULL`) para gerenciar o andamento do algoritmo. Todo nó `x` na pilha representa o comando "imprima os nós da árvore cuja raiz é `x`".

No código abaixo, a pilha é implementada em um vetor `pilha[0..t]`, sendo `t` o índice do topo da pilha:

```
// Imprime o item de cada nó de uma árvore binária h.
// A função supõe que h != NULL.



void imprime_red (link h) {
    link *pilha;
    int t;

    pilha = malloc((1+height(h)) * sizeof (link))
    pilha[t=0] = h;
    while (t >= 0) {
       h = pilha[t--];
       printf("%d\n", h->item);
       if (h->r != NULL) pilha[++t] = h->r;
       if (h->l != NULL) pilha[++t] = h->l;
    }
    free(pilha);
}
```

Note que `pilha[i] != NULL` para todo `i` entre `0` e `t`.

Exercícios

10. [Inorder não recursivo. Sedg 5.82, p.235] Escreva uma versão iterativa do `imprime` que percorra a árvore na ordem esquerda-raiz-direita (= *inorder*).
11. [Postorder não recursivo. Sedg 5.83, p.235] Escreva uma versão iterativa do `imprime` que percorra a árvore na ordem esquerda-direita-raiz (= *postorder*). (Cuidado!)
12. Escreva uma função que calcule a soma dos conteúdos (campos `item`) dos nós de uma árvore binária. Percorra a árvore em ordem esquerda-raiz-direita (= *inorder*).

Percorrendo a árvore "por níveis"

Os nós podem ser percorridos em uma quarta ordem, diferente da raiz-esquerda-direita, da esquerda-raiz-direita e da esquerda-direita-raiz. Para fazer isso, basta usar uma fila no lugar de uma pilha. (Veja programa 5.16, p.235, de Sedgewick.)

```
// Imprime o item de cada nó de uma árvore binária h.
// A função supõe que h != NULL.



void imprime (link h) {
    link *fila;
    int i, f;

    fila = malloc(count(h) * sizeof (link));
    fila[0] = h;
    i = 0; f = 1;
    while (f > i) {
       h = fila[i++];
       printf("%d\n", h->item);
       if (h->l != NULL) fila[f++] = h->l;
       if (h->r != NULL) fila[f++] = h->r;
    }
    free(fila);
}
```

A função usa uma fila implementada em um vetor `fila[i..f-1]`: o índice do primeiro da fila é `i` e o índice do último é `f-1`. Todos os elementos da fila são diferentes de `NULL`.

Desenho de uma árvore

O programa 5.18, p.237, de Sedgewick faz um desenho de uma árvore binária. A função `show` supõe que o `item` de cada nó é do tipo `char` e não do tipo `int` como acima.

```
// A função show faz um desenho esquerda-direita-raiz
// da árvore x. O desenho terá uma margem esquerda de
// 3b espaços.



void show(link x, int b) {
    if (x == NULL) {
        printnode('*', b);
        return;
    }
    show(x->r, b+1);
    printnode(x->item, b);
    show(x->l, b+1);
}



// A função auxiliar printnode imprime o caracter
// c precedido de 3b espaços e seguido de uma mudança
// de linha.



void printnode(char c, int b) {
    int i;
    for (i = 0; i < b; i++) printf("   ");
    printf("%c\n", c);
}
```

Eis uma amostra do resultado de `show(x,0)`. Troquei os espaços em branco por "-" para facilitar a leitura.

```
------*
---H
------------*
---------G
------------*
------F
---------*
E
------*
---D
------------*
---------C
------------*
------B
------------*
```

```
---------A
------------*
```

Eis o resultado da impressão da mesma árvore em ordem raiz-esquerda-direita. Troquei os espaços em branco por "-" para facilitar a leitura.

```
E
---D
------B
---------A
------------*
------------*
---------C
------------*
------------*
------*
---H
------F
---------*
---------G
------------*
------------*
------*
```

Para obter isso, troque os três últimos comandos de `show` por

```
    printnode(x->item, b);
    show(x->r, b+1);
    show(x->l, b+1);
```

Construção de um torneio

O programa 5.19, p.238, de Sedgewick ilustra a construção de uma árvore binária. Diremos que uma árvore binária é um *torneio* se cada nó que não seja uma folha contém uma cópia do maior dos `items` de seus dois filhos.

```
// A função max recebe um vetor não vazio a[p..q]
// (portanto p <= q) e constroi um torneio cujas folhas
// são a[p],..,a[q].  A função devolve a raiz do torneio.

link max(int a[], int p, int q) {
    int m, u, v;
    link x;

    m = (p + q) / 2;
    x = malloc(sizeof *x);
    if (p == q) {
       x->l = x->r = NULL;
       x->item = a[m];
       return x;
    }
    x->l = max(a, p, m);
    x->r = max(a, m+1, q);
    u = x->l->item;
    v = x->r->item;
    if (u > v)  x->item = u;
    else  x->item = v;
    return x;
}
```

Compare com o programa 5.6 de Sedgewick.

Exercícios

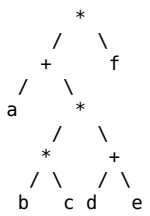
13. Aplique a função `max` acima ao vetor 1 2 3 4 5 .
14. [Sedg 5.91, p.241] Escreva uma função recursiva que remova de um torneio todas as folhas que contenham uma dada chave. (Veja acima o exercício Sedg 5.59.)
15. [Busca binária] Escreva uma função que contrua a árvore binária que representa todas as possíveis buscas binárias em um vetor crescente `a[p..r]`. Cada nó da árvore deverá conter o índice do vetor envolvido em uma comparação com a chave procurada.

16. Escreva uma função que construa uma árvore binária aleatória com `n` nós e chaves aleatórias.

Árvore de expressão aritmética

Vamos considerar aqui mais um exemplo de construção de árvore binária. Desta vez, a árvore será uma representação de uma expressão aritmética.

Suponha que temos uma expressão aritmética cujos operadores são todos binários. Mais concretamente, suponha que os operadores são soma (+) e multiplicação (*). Suponha também, para simplificar, que os operandos são nomes de variáveis, cada um consistindo de uma única letra. Uma expressão aritmética pode ser muito bem representada por uma árvore binária: as folhas da árvore são operandos e os nós internos são operadores.



Se a árvore for lida em ordem esquerda-raiz-direita, teremos a expressão em notação infixa. Se for lida em ordem esquerda-direita-raiz, teremos a expressão em notação posfixa. Se for lida em ordem raiz-esquerda-direita-raiz, teremos a expressão em notação prefixa.

infixa	`(a+(b*c)*(d+e))*f`
posfixa	`abc*de+*+f*`
prefixa	`*+a**bc+def`

O programa 5.20, p.240, de Sedgewick, faz o serviço inverso: transforma a expressão prefixa (não vazia, é claro) em uma árvore binária. Se a expressão consiste em um única letra, a árvore terá um único nó; se a expressão for algo como `*ab`, a árvore terá uma raiz e duas folhas.

Suponha que a expressão prefixa está armazenada em um vetor global de caracteres `a[i..]`, sendo `i` uma variável global.

```
typedef struct Tnode *link;
struct Tnode {
   char token;
   link l, r;
} ;

char *a;
int i;

// A função parse atua sobre a expressão prefixa a[i..].
// Os operadores são '+' e '*', cada variável tem
// um só caracter, e não há espaços entre os caracteres.
// A função transforma a expressão em uma árvore binária
// e devolve a raiz da árvore.



link parse() {
    char t;
    link x;

    t = a[i++];
    x = malloc(sizeof *x);
    x->token = t;
    if (t == '+' || t == '*') {
        x->l = parse();
        x->r = parse();
    }
    else x->l = x->r = NULL;
    return x;
}
```

Exercícios

17. Escreva uma função que calcule o valor da expressão aritmética representada por uma árvore sendo dodos os valores das variáveis. Suponha que os valores das variáveis são dados em um vetor do tipo `int` indexado por letras.

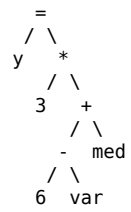
18. Escreva uma função que receba uma expressão aritmética em notação infixa e construa a correspondente árvore. Suponha que a expressão só envolve os operadores `'+'` e `'*'` e operandos que consistem em uma só letra. A página sobre pilhas pode ser útil.

Valor de uma expressão aritmética

O capítulo 14 do livro de Roberts discute a implementação de processador de expressões aritméticas. Vamos examinar aqui apenas

- a estrutura das árvores que representam expressões (mais rica e completa que aquela usada acima) e
- as funções que calculam o valor de uma expressão.

Nossas expressões aritméticas admitem os operadores `=, +, -, *, /` e admitem operandos que podem ser números inteiros, nomes de variáveis e, é claro, sub-expressões. Exemplo:



Eis a declaração do tipo de uma expressão:

```
typedef char *string;


// Type: expression
// ----------------
// This type is used to represent the abstract notion of an
// expression, such as one you might encounter in a C program.
// An expression is defined recursively to be one of the
// following:
//    1. A constant
//    2. A string representing the name of a variable
//    3. Two expressions combined by an operator
//
typedef struct node *expression


// Type: exptype
// -------------
// This enumeration type is used to differentiate the three
// expression types: constants, variables, and subexpressions.
//
typedef enum {Constant, Variable, Subexpression} exptype;
```

Para representar os nós da árvore vamos usar uma estrutura que envolve um `union` (da linguagem C):

```
// Type: node
// ----------
// An expression is represented as tree. The contents of each
// node consists of a tagged union that allows the node to
// have multiple representations [interpretations?].
//
struct node {
   exptype type;
   union {
      int constRep;      // a constant
      string varRep;     // name of a variable
      struct {
         char       op;  // '=' or '+' or '-' or '*' ou '/'
         expression lhs; // left subexpression
         expression rhs; // right subexpression
      } subexpRep;
   } contents;
} ;
```

Finalmente, eis as funções que calculam o valor de uma expressão:

```
// Function: EvalExp
// Usage: value = EvalExp(exp);
// ----------------------------
// Returns the value of the expression exp. (The function
// assumes that the values of all variables have been
// already loaded into the appropriate table.)
//
int EvalExp(expression exp) {
   switch (exp->type) {
      case Constant:
         return exp->contents.constRep;
      case Variable:
```

```
         return GetVariableValue(exp->contents.varRep);
      case Subexpression:
         return EvalSubExp(exp);
   }
}

// Returns the value of the subexpression exp. (The values
// of all variables must have been already loaded into the
// appropriate table.)
//
static int EvalSubExp(expression exp) {
   char op;
   expression leftexp, rightexp;
   int leftval, rightval;
   op = exp->contents.subexpRep.op;
   leftexp = exp->contents.subexpRep.lhs;
   rightexp = exp->contents.subexpRep.rhs;
   if (op = '=') {
      rightval = EvalExp(rightexp);
      SetVariableValue(leftexp->contents.varRep, rightval);
      return rightval;
   }
   leftval = EvalExp(leftexp);
   rightval = EvalExp(rightexp);
   switch (op) {
      case '+': return leftval + rightval;
      case '-': return leftval - rightval;
      case '*': return leftval * rightval;
      case '/': return leftval / rightval;
   }
}


// Prototypes of auxiliary functions:

// Returns the value of variable var.
int GetVariableValue(string var) ;

// Sets the value of variable var to val.
int SetVariableValue(string var, int val) ;
```

Mais exercícios

19. Escreva uma função que receba um vetor `a[1..n]`, interprete esse vetor como um heap, e construa a correspondente árvore binária.

20. [Sedg 12.54, p.511] O *comprimento interno* de uma árvore binária é a soma dos comprimentos dos caminhos que levam da raiz a cada uma das folhas. Escreva um programa recursivo que calcule o comprimento interno de uma árvore binária dada.

21. [Sedg 12.63, p.514, índices no lugar de ponteitos] Árvores binárias podem ser implementadas com índices no lugar de ponteiros, da seguinte maneira:

    teremos três vetores "paralelos", `item[1..N]`, `l[1..N]` e `r[1..N];` para cada índice `i`, `l[i]` é o índice do filho esquerdo de `i` e `r[i]` é o índice do filho direito.

    Exercício: escreva todas as funções desse capítulo para a implementação que acabamos de sugerir. [Essa implementação tem suas vantagens porque reduz o tempo consumido pelas sucessivas chamadas de malloc durante a construçÃo da árvore. Mas exige que o número total de nós seja conhecidono antes que a árvore comece a ser construída.]

Compressão de arquivos

[Esse material está no capitulo 22 da 2-a edição do livro de Sedgewick.] Suponha dada uma cadeia de caracteres, digamos

```
bafeabacaadefa
```

Cada caracter é representado por 8 *bits*:

símbolo gráfico	caracter ASCII	bits
a	97	`01100001`
b	98	`01100010`

c	99	01100011
d	100	01100100
e	101	01100101
f	102	01100110

Portanto, nossa cadeia de caracteres é representada pela seguinte cadeia de bits:

```
011000100110000101100110011001010110000101100010011000010110001101100001011000010110010001100101011001100011
```

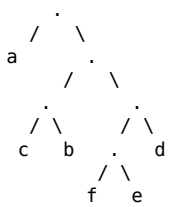
Suponha agora que adotemos uma codificação com número variável de *bits* — poucos *bits* para as letras mais frequentes e muitos *bits* para letras raras:

símbolo gráfico	bits
a	0
b	101
c	100
d	111
e	1101
f	1100

Agora podemos representar a cadeia `bafeabacaadefa` por uma cadeia de *bits* bastante curta:

```
1010110011010101010000111110111000
```

Note que não temos separadores entre as subcadeias de *bits* que representam os vários caracteres. Apesar disso, a cadeia de *bits* pode ser decodificada sem ambiguidades. Essa é uma propriedade interessante e valiosa de nossa tabela de códigos. A propriedade decorre do seguinte fato: o código pode ser representado por uma árvore cujas folhas são os caracteres:



Para determinar o código de um caracter *x*, comece na raiz e caminhe até *x*; toda vez que descer para a esquerda, acrescente um `0` ao código de *x*; toda vez que descer para a direita, acrescente um `1`. [Veja exercício sobre códigos de nós].

PROBLEMA: Dada uma cadeia de caracteres, construir uma tabela de codificação que codifique a cadeia de caracteres usando o menor número possível de *bits*.

Eis um algoritmo que resolve o problema. Suponha que cada caracter *x* ocorre *f(x)* vezes na cadeia de caracteres. Então o seguinte algoritmo produz uma codificação ótima: construa uma árvore binária cujas chaves são números inteiros; comece com um nó para cada caracter *x*, sendo *f(x)* a chave do nó; seja *x* um nó que tem chave mínima; seja *y* um nó que tem a segunda menor chave; faça com que *x* e *y* sejam os filhos de um novo nó *z*; a chave do novo nó será *f(x)+f(y)*; os nós *x* e *y* "saem do jogo" e o nó *z* "entra no jogo"; repita o processo até que todas as subárvores se juntem. A árvore resultante á conhecida como *árvore de Huffman* da cadeia de caracteres original.

Exemplo: Suponha que nossa cadeia só contém os caracteres `a`, `b`, `c`, `d`, `e`, `f`. Suponha que o número de ocorrências de cada caracter é dado pela tabela:

x	a	b	c	d	e	f
f(x)	45	13	12	16	9	5

Aplique o algoritmo. Verifique que a árvore é exatamente aquela da figura acima.

Exercício: Escreva uma função que receba uma cadeia de caracteres e construa uma árvore de Huffman para essa cadeia.

Veja também o exercício sobre reconstrução da árvore de códigos.

Esse material sobre codificação e compressão de arquivos pode ser encontrado no capítulo 22 da 2-a edição do livro do Sedgewick. Também pode ser encontrado no livro *Introduction to Algorithms* de Cormen, Leiserson, Rivest e Stein (há uma edição do livro em português).

Veja minhas notas de aula sobre o árvores binárias.
Veja também os capítulos 13 e 14 do livro de Roberts: eles são excelentes!

URL of this site: www.ime.usp.br/~pf/mac0122-2002/
Last modified: Mon Oct 9 08:03:27 BRT 2017
Paulo Feofiloff
IME-USP