**MAC0122 Princípios de Desenvolvimento de Algoritmos**



# Árvores binárias de busca

Uma árvore binária de busca é uma maneira bastante popular de implementar uma tabela de símbolos. Esta página discute o conceito de árvore de busca com base nas seções 12.4, 12.5 e 12.8 do livro do Sedgewick.

## Definição

É possível fazer uma busca binária em uma lista encadeada ordenada? A rigor, a resposta é NÃO. Mas é possível imitar uma busca binária, se trocarmos a lista por uma árvore.

Uma árvore **de busca** (= *binary search tree* = *BST*) é uma árvore binária cujas chaves aparecem em ordem crescente quando a árvore é percorrida em ordem esquerda-raiz-direita. Em outras palavras, a chave de cada nó da árvore deve ser

*maior ou igual* que qualquer chave na sua subárvore esquerda e
*menor ou igual* que qualquer chave na sua subárvore direita.

## Exercícios

1. Suponha dada uma árvore binária com a seguinte propriedade: para cada nó `x` tem-se `x->l->chave` ≤ `x->chave` ≤ `x->r->chave`. Essa árvore é de busca?

## Operação de busca

Suponha que os nós da árvore têm a seguinte estrutura:

```
typedef struct node *link;
struct node {
     int  chave;
     link l, r;
} ;
```

A seguinte função faz uma busca em uma árvore binária de busca. A função recebe um inteiro `v` e devolve um nó da árvore que contém `v`, se tal nó existir. A função foi extraída do programa 12.7 de Sedgewick.

```
// Recebe uma árvore de busca h e um inteiro v.
// Devolve um nó cuja chave é igual a v.
// Devolve NULL se tal nó não existe.
//
link searchR(link h, int v) {
    int t;
    if (h == NULL) return NULL;
    t = h->chave;
    if (v == t) return h;
    if (v < t)
       return searchR(h->l, v);
    else
       return searchR(h->r, v);
}
```

Quanto tempo `STsearch` consome? O tempo é proporcional ao número de comparações entre `v` e chaves de nós. No pior caso, o número de comparações é o dobro da altura da árvore.

A altura de uma árvore com `N` nós fica entre `lg(N)` e `N`. Uma árvore é considerada **equilibrada** ou **balanceada** se sua altura é da ordem de `lg(N)`. Uma árvore é equilibrada se a maior parte de suas folhas tem a mesma profundidade,

# Exercícios

2. [Search iterativo] Escreva uma versão não recursiva da função `searchR`.
3. [Sedg 12.49, p.507] Escreva uma função que calcule o número de nós de uma árvore binária de busca que tenham chave igual a um dado número `v`.
4. A função abaixo recebe uma árvore não vazia `h` e promete devolver o endereço de um nó que tenha chave `v`. O que há de errado com a função?

```
link procura(link h, int v) {
    while (h != NULL && h->chave > v)  h = h->l;
    while (h != NULL && h->chave < v)  h = h->r;
    return h;
}
```

5. [Vetor-para-árvore] Escreva uma função que transforme um vetor crescente em uma árvore de busca razoavelmente equilibrada (tantos nós à esquerda quantos à direita de cada nó). Suponha que os elementos do vetor são inteiros. Escreva duas versões: uma iterativa e uma recursiva.
6. [Árvore-para-vetor] Escreva uma função que transforme uma árvore de busca em um vetor crescente. Cada elemento do vetor deverá conter a chave de um dos nós da árvore. Comece por alocar dinamicamente o vetor.
7. Escreva uma função `min` que receba uma árvore de busca e encontre um nó da árvore que tenha chave mínima. Faça duas versões: uma iterativa e uma recursiva.
8. [Roberts 9, ch 13, p.586] Escreva uma função que decida se uma dada árvore binária é ou não é de busca.
9. Escreva uma função que construa uma árvore aleatória de busca com `n` nós e chaves aleatórias.

# Operação de inserção

Como é possível inserir um novo nó em uma árvore de busca binária sem que ela deixe de ser uma árvore de busca binária? Eis uma solução extraída do programa 12.7 de Sedgewick.

```
// Recebe uma árvore binária de busca h e um inteiro v.
// Insere na árvore um novo nó com chave igual a v.
// Devolve o endereço da nova árvore de busca.

link insertR (link h, int v) {
    int t;
    if (h == NULL) {
       link x = malloc(sizeof *x);
       x->chave = v;
       x->l = x->r = NULL;
       return x;
    }
    t = h->chave;
    if (v < t)
       h->l = insertR (h->l, v);
    else
```

```
         h->r = insertR (h->r, v);
      return h;
}
```

Quanto tempo `insertR` consome? O tempo é proporcional ao número de comparações entre chaves. E o número de comparações é, no pior caso, proporcional à *altura* da árvore. Se a árvore for balanceada, sua altura é `lg(N)`.

## Exercícios

10. [Sedg 12.44 a 12.47, p.507] Comece com uma árvore de busca vazia. Suponha que as chaves da árvore são caracteres e não `int`s. Agora insira as chaves `E A S Y Q U E S T I O N`, nessa ordem. Quantas comparações entre chaves você realizou durante o processo? Dê uma outra sequência de inserções que resulte na mesma árvore de busca.
11. [Insert iterativo] Escreva uma versão não recursiva da função `insertR`.
12. [Search-and-insert. Sedg 12.48, p.507] Escreva uma função que procure uma dada chave `c` em uma árvore de busca não vazia. Em caso de sucesso, a função deve devolver o endereço de um nó que tem chave `c`; em caso de fracasso, a função deve inserir um novo com chave `c` na árvore (de tal forma que a árvore continue sendo de busca) e devolver o endereço do novo nó.

## Inserção na raiz de uma árvore de busca

Depois de várias operações de inserção, uma árvore de busca está, em geral, "desbalanceada". Para restabelecer o equilíbrio, usa-se o truque das rotações (program 12.11 no livro do Sedgewick):

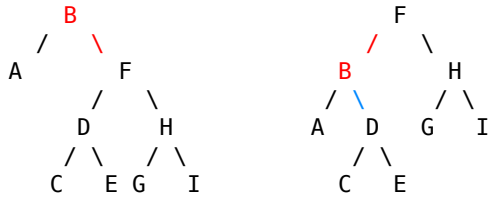
```
// Recebe uma árvore binária h tal que
// h != NULL e h->l != NULL.
// Executa uma rotação para a direita em torno de h.
// Devolve o endereço da nova raiz da árvore.

link rotR(link h)
{
   link x = h->l;
   h->l = x->r;
   x->r = h;
   return x;
}
```

```
// Recebe uma árvore binária h tal que
// h != NULL e h->r != NULL.
// Executa uma rotação para a esquerda em torno de h.
// Devolve o endereço da nova raiz da árvore.

link rotL(link h)
{
   link x = h->r;
   h->r = x->l;
   x->l = h;
   return x;
}
```

A figura mostra o efeito de uma rotação esquerda em torno da raiz `B`:




As operações de rotação são fundamentais para "re-balancear" um árvores "desbalanceadas". Não vamos estudar esse processo aqui. Vamos apenas mostrar como as rotações permitem inserir um novo nó na posição da raiz de uma árvore de busca (veja programa 12.12 de Sedgewick):

```
// Recebe uma árvore de busca h e um inteiro v.
// Insere novo nó com chave v de tal modo que
// (1) ele seja a raiz da nova árvore e
// (2) a árvore continue sendo de busca.
// Devolve o endereço da nova árvore de busca.

link insertT(link h, int v)
{
    if (h == NULL) {
       link x = malloc(sizeof *x);
       x->chave = v;
       x->l = x->r = NULL;
       return x;
    }
    if (v < h->chave) {
       h->l = insertT(h->l, v);
       h = rotR(h);
    }
    else {
       h->r = insertT(h->r, v);
       h = rotL(h);
    }
    return h;
}
```

---

## Os programas do Sedgewick

Nos exemplos acima, cada nó da árvore contém um número inteiro e nada mais (sem contar os links, é claro). Amanhã posso precisar de nós que contêm caracteres. Depois de amanhã posso precisar de nós que contêm um struct com o nome, o número e nota de um aluno. Na semana que vem posso precisar . . . Em cada um desses casos, o código das funções de busca e inserção deveria ser re-escrito. Mas os algoritmos por trás das funções são sempre os mesmos. Gostaríamos, então, de escrever um código "genérico" que sirva para todas as aplicações.

Esta é a atitude que Sedgewick adota no capítulo 12. As várias estruturas de dados são definidas de tal maneira que a alteração de uns poucos `#define` e `typedef` adapta a estrutura a diferentes usos e aplicações.

Para começar, temos um tipo-de-dados básico `Key` (com `K` maiúsculo). Nos exemplos das seções anteriores tinhamos, implicitamente,

```
typedef int Key;
```

Para comparar objetos do tipo `Key`, Sedgewick imagina duas macro-instruções para o pré-processador, que serão aplicadas a objetos do tipo `Key`:

```
#define eq(A, B) (A == B)
#define less(A, B) (A < B)
```

Em seguida, temos um tipo-de-dados `Item` que representa o "conteúdo" de cada nó. Por exemplo,

```
typedef struct {
     char *nome;
     Key   chave;
} Item;
```

(Nos exemplos das seções anteriores o tipo `Item` se confundia com `Key`.) Para extrair chaves, Sedgewick imagina uma macro `key` (com `k` minúsculo) que será aplicada a objetos do tipo `Item`:

```
#define key(A) (A.chave)
```

Para fazer o papel de "item nulo" ou "item vazio", Sedgewick imagina um objeto `NULLitem` que seja diferente de qualquer `Item` que o usuário considere válido. Por exemplo, se todas as `chave`s válidas forem não negativas, poderíamos definir `NULLitem` assim:

```
Item NULLitem;
NULLitem.chave = -1;
```

Os nós das árvores serão da forma (o prefixo `ST` lembra *search tree*):

```
typedef struct STnode *link;
struct STnode {
     Item item;
     link l, r;
} ;
```

O formato das árvores será ligeiramente diferente do que usamos acima: se um nó `x` não tem filho esquerdo, por exemplo, então em lugar de `x->l = NULL` Sedgewick faz

```
x->l = z
```

sendo `z` é um nó fixo, criado especialmente para esta finalidade:

```
link z = malloc(sizeof (struct STnode));
z->item = NULLitem;
z->l = z->r = NULL;
```

A vantagem desse circo todo é que basta mudar alguns `typedef` e alguns `#define`, e as funções de manipulação das árvores poderão ser aplicados a diferentes tipos de nós. Por exemplo, podemos ter

```
typedef char *Key;
#define eq(A, B) (strcmp(A, B) == 0)
#define less(A, B) (strcmp(A, B) < 0)
typedef struct {
     char  *nome;
     int   *numero;
     float *nota;
     Key    codigo;
} Item;
#define key(A) (A.codigo)
Item NULLitem;
NULLitem.codigo[0] = '\0';
```

**Busca.** Eis como fica a função de busca no programa 12.7 de Sedgewick. A variável global `head`

```
link head;
```

aponta para a raiz da árvore. A seguinte função (versão simplificada do programa 12.7) faz uma busca na árvore:

```
// Recebe um Key v e devolve um Item x tal que key(x) == v.
// Se tal Item não existe então a função devolve NULLitem.
// Todos os Item estão armazenados em uma árvore de busca
```

```
// cuja raiz é head.

Item STsearch (Key v) {
    return searchR (head, v);
}

Item searchR(link h, Key v) {
    Key t = key(h->item);
    if (h == z) return NULLitem;
    if eq(v, t) return h->item;
    if less(v < t)
       return searchR(h->l, v);
    else
       return searchR(h->r, v);
}
```

**Inserção.** A seguinte função (veja programa 12.7) insere um novo nó na árvore de busca:

```
// Insere na árvore de busca cuja raiz é head
// um novo nó com contéudo item.
// (A raiz da nova árvore pode ser diferente da árvore original.)

void STinsert (Item item) {
    head = insertR (head, item);
}

link insertR (link h, Item item) {
    Key v = key(item), t = key(h->item);
    if (h == z) {
       link x = malloc(sizeof *x);
       x->item = item;
       x->l = x->r = z;
       return x;
    }
    if less(v, t)
       h->l = insertR (h->l, item);
    else
       h->r = insertR (h->r, item);
    return h;
}
```

Eis uma versão não recursiva da função `STinsert` (programa 12.9):

```
void STinsert(Item item) {
    Key v = key(item);
    link p = head, x = p;
    if (head == NULL) {
        head = NEW(item, NULL, NULL);
        return;
    }
    while (x != NULL) {
        p = x;
        x = less(v < key(x->item)) ? x->l : x->r;  // agora p é pai de x
    }
    x = NEW(item, NULL, NULL);
    if less(v < key(p->item))  p->l = x;
    else  p->r = x;
}

link NEW(Item item, link l, link r) {
    link x = malloc(sizeof *x);
    x->item = item;
    x->l = l; x->r = r;
    return x;
}
```

## Exercícios

13. [Sedg. 12.52, p.508] A versão não recursiva da função `STinsert` (programa 12.9) faz uma comparação redundante para determinar o campo de `p` que deve receber o endereço do novo nó. Use ponteiros-para-links para evitar essa redundância.

## Mais exercícios

14. Se você tivesse que refazer a tarefa 7 (indexador de texto), usaria uma árvore de busca. Descreva a estrutura dos nós da árvore que você usaria.

15. [Roberts 8, p.587] Uma árvore **balanceada no sentido AVL** se, para cada nó `h`, as alturas das subárvores que têm raízes `h->l` e `h->r` diferem de 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.

---

Veja minhas notas de aula sobre árvores de busca.
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 08:03:33 BRT 2017
*Paulo Feofiloff*
*IME-USP*