

Árvores binárias de busca

David Déharbe*

Semestre 03.2 - versão 1.1

Sumário

1	Introdução	2
2	Definição	2
3	Implementação	3
4	Busca	3
5	Inserção	4
6	Remoção	5
7	Construção de ABB otimal no caso de acesso uniforme	5
8	Critérios de avaliação de ABBs	6
8.1	O comprimento de caminho interno	6
8.2	O comprimento de caminho externo	6

Lista de Figuras

1	Algoritmos de busca em ABB	4
2	Algoritmo de inserção em ABB	5
3	Algoritmos de remoção em ABB	6
4	Etapas na construção de uma ABB otimal	7

Resumo

Esse capítulo foi preparado com base os livros *Estruturas de Dados e seus Algoritmos* de Jaime Szwarcfiter e Lilian Markenzon e *Algorithms, Data Structures and Problem Solving* de Mark Allen Weiss.

*Copyright © 2003 David Déharbe and Universidade Federal do Rio Grande do Norte. Todos os direitos reservados.

1 Introdução

As árvores binárias de busca (ABB) podem ser empregados para armazenar coleções de dados que podem ser ordenados entre si e para os quais é preciso prover operações de busca, inserção e remoção. Uma ABB é uma árvore binária, onde cada dado é guardado em um nó. O princípio básico de repartição dos dados é que os nós da subárvore esquerda armazenam as chaves inferiores à chave da raiz, e os nós da subárvore direita armazenam as chaves superiores à raiz.

A maneira como os dados são repartidos na ABB tem uma grande influência no desempenho das operações de busca. Por exemplo, uma busca numa árvore binária zigzague é nada mais que uma busca numa lista ordenada, cujo desempenho é uma função linear do número de itens armazenados. Se a árvore for cheia, o tempo de busca será limitado pela altura da árvore, que é uma função logarítmica do número de dados armazenados. Além de procurar minimizar a altura da árvore, um outro objetivo é localizar os dados mais frequentemente acessados perto da raiz da árvore.

Neste capítulo apresentaremos as árvores binárias de busca e abordaremos várias maneiras de organizar árvores binárias de busca de tal maneira a otimizar o tempo de busca de um dado.

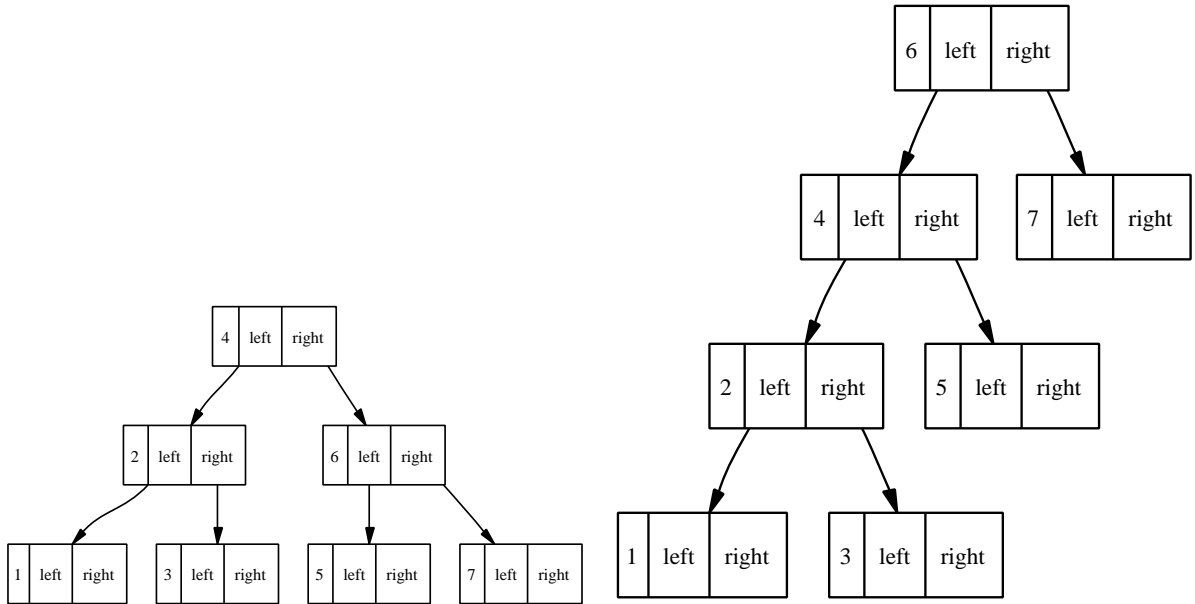
2 Definição

Seja $C = \{c_1, \dots, c_i\}$ um conjunto ordenado de chaves tal que $c_1 < \dots < c_i$. O objetivo da busca é verificar se um objeto o , com chave s , é elemento de S ou não. C é armazenado numa árvore binária de busca, que deve satisfazer a definição seguinte.

Definição 1 (Árvore binária de busca) *Seja $C = \{c_1, \dots, c_i\}$ um conjunto ordenado de chaves tal que $c_1 < \dots < c_i$. Uma árvore binária de busca (ou ABB) A para C é tal que:*

- *A tem i nós, e cada nó corresponde a uma chave distinta, ou seja o rótulo de cada nó n pertence a C , e cada chave c_j de C é rótulo de algum nó de A . Denotamos $r(n)$ o rótulo do nó n . Seja n um nó de A .*
- *Para qualquer subárvore (n, A_e^n, A_d^n) de A ,*
 - *$\forall n_e \in A_e^n : r(n_e) < r(n)$, ou seja todos os nós da subárvore esquerda de n armazenam uma chave inferior à de n .*
 - *$\forall n_d \in A_d^n : r(n_d) > r(n)$, ou seja todos os nós da subárvore direita de n armazenam uma chave superior à de n .*

Para um conjunto C com mais de um elemento, há várias ABB diferentes possíveis. Por exemplo, seja $C = \{1, 2, 3, 4, 5, 6, 7\}$, duas ABB possíveis armazenando C são:



3 Implementação

Geralmente, nas representações computacionais de uma árvore, os conceitos de árvore (não vazia), e raiz dessa árvore são representados pelo mesmo objeto. Adicionalmente, os algoritmos usando árvores binárias de busca precisam saber, dada uma árvore, (e portanto, dado o seu nó raiz), qual a sub-árvore esquerda e qual a sub-árvore direita da raiz. Seja \mathcal{K} o domínio das chaves representadas e $\mathcal{T}_{\mathcal{K}}$ o domínio das árvores binárias de busca sobre \mathcal{K} . Denotaremos $key : \mathcal{T}_{\mathcal{K}} \rightarrow \mathcal{K}$, $left : \mathcal{T}_{\mathcal{K}} \rightarrow \mathcal{T}_{\mathcal{K}}$, e $right : \mathcal{T}_{\mathcal{K}} \rightarrow \mathcal{T}_{\mathcal{K}}$ as funções que, a uma árvore binária t associam a chave da sua raiz, sua sub-árvore esquerda e sua sub-árvore direita. Denotamos t_{\perp} a árvore vazia de $\mathcal{T}_{\mathcal{K}}$.

Para implementar essas funções, pode se usar, para cada nó, uma estrutura comportando os dados associados com esse nó e uma referência para a sub-árvore esquerda e para a sub-árvore direita. Uma outra possibilidade é uso de vetores para essas informações, quando o domínio das chaves \mathcal{K} pode ser mapeado sobre um domínio de vetor (intervalo $[1, n]$).

4 Busca

De uma certa forma, os nós da ABB já estão ordenados. Na busca de uma chave c , basta comparar essa chave com a chave c_r da raiz para saber se está na subárvore esquerda (caso $c < c_r$), na subárvore direita (caso $c > c_r$) ou na raiz (caso $c = c_r$). Nota que se $c < c_r$ e a subárvore esquerda está vazia, então c não está na árvore. Dualmente, se $c > c_r$ e a subárvore direita está vazia, então c também não está na árvore. O algoritmo de busca apresentado na figura 1(a) é baseado nessas observações.

Também é útil ter um algoritmo que retorna o nó que armazena a chave procurada, caso essa for presente, ou a do nó armazenando a maior chave inferior à chave procurada (ou a menor chave superior à chave procurada) caso contrário. Esse algoritmo é similar ao precedente e é apresentado na figura 1(b). Uma vez obtido o resultado, basta testar se o resultado não é a árvore vazia e comparar a chave da raiz do resultado com a chave procurada para saber se ela está presente na árvore ou não.

```

01  present?( $x : \mathcal{K}, t : \mathcal{T}_{\mathcal{K}}$ ) : Bool
02    if  $t = t_{\perp}$  then
03       $\rightarrow false$ 
04    elseif  $x = key[t]$  then
05       $\rightarrow true$ 
06    elseif  $x < key[t]$  then
07       $\rightarrow present?(x, left(t))$ 
08    else
09       $\rightarrow present?(x, right(t))$ 

```

(a) com retorno booleano

```

01  find'( $x : \mathcal{K}, t : \mathcal{T}_{\mathcal{K}}$ ) : Bool
02    if  $x = key[t]$  then
03       $\rightarrow t$ 
04    elseif  $x < key[t]$  then
05      if  $left(t) = t_{\perp}$  then
06         $\rightarrow t$ 
07      else
08         $\rightarrow find'(x, left(t))$ 
09    else
10      if  $right(t) = t_{\perp}$  then
11         $\rightarrow t$ 
12      else
13         $\rightarrow find'(x, right(t))$ 
14  find( $x : \mathcal{K}, t : \mathcal{T}_{\mathcal{K}}$ ) : Bool
15    if  $t = t_{\perp}$  then
16       $\rightarrow t$ 
17    else
18       $\rightarrow find'(x, t)$ 

```

(b) com retorno de nó

Figura 1: Algoritmos de busca em ABB

A complexidade do algoritmo de busca é igual a o número de vezes que esse algoritmo, que é recursivo, é chamado. No pior caso, o algoritmo é chamado i vezes, onde i é o número de nós na ABB. Isso acontece se a ABB for uma árvore zigzague e a chave procurada está na (única) folha da ABB.

Em geral, para uma ABB qualquer, o pior caso é quando a busca deve chegar a folha mais distante da raiz, ou seja a complexidade da busca, no pior caso, é igual à altura da ABB. Para uma quantidade dada de chaves, e consequentemente de nós, a árvore de altura mínima deve ser uma árvore binária completa. Portanto, árvores binárias completas são ABBS ótimas para a operação de busca. Neste caso, a complexidade da busca, no pior caso, é linear em $\log i$, a altura da árvore (ver a proposição 3 do capítulo anterior).

5 Inserção

O algoritmo de inserção de uma chave numa ABB é apresentado na figura 2. Tem como entrada a chave x a inserir e a árvore t onde a inserção deve ser operada. O resultado é um valor booleano que indica se a operação foi realmente efetuada (quando o valor x não estava presente anteriormente). Primeiro, é efetuada uma busca para verificar que a chave não está já armazenada na árvore (linha 03). Caso contrário, o algoritmo de busca retorna o endereço do nó que armazena a maior chave inferior (ou a menor chave superior) à chave a inserir. Caso não foi encontrado o valor x , é criada uma nova folha (linha 05) que passa a ser a subárvore direita (ou a subárvore esquerda) deste nó (linhas 06-09).

O custo do algoritmo de inserção é o do algoritmo de busca mais uma número constante de operações. Sua complexidade assintótica portanto é a mesma.

```

01  inserir( $x : \mathcal{K}, t : \mathcal{T}_{\mathcal{K}}$ ) : Bool
02      var ins, new :  $\mathcal{T}_{\mathcal{K}}$ 
03      ins  $\leftarrow$  find( $x, t$ )
04      if  $ins = t_{\perp} \vee key[ins] \neq x$  then
05           $key[new] \leftarrow x$ 
06          if  $left[ins] = t_{\perp}$  then
07               $left[ins] = new$ 
08          else
09               $right[ins] = new$ 
10           $\rightarrow true$ 
11      else
12           $\rightarrow false$ 

```

Figura 2: Algoritmo de inserção em ABB

6 Remoção

A operação de remoção numa ABB é a mais complicada, pois, no caso de um nó interno, a remoção desconecta partes da ABB, o que deve ser evitado. O algoritmo de remoção é dado na figura 3. Tem como parâmetros x a chave a remover, e t a ABB onde a remoção deve ser realizada. O resultado é uma ABB que contém todas as chaves que t continha, com a exceção de x .

O algoritmo deve considerar os seguintes casos:

- Se k fica numa folha, é o caso mais simples pois não ocorre desconexão da ABB, e é suficiente remover esse nó.
- Se k fica num nó interior n com uma sub-árvore vazia, basta conectar essa sub-árvore ao pai, no lugar de n , que é removido.
- Quando k fica num nó interior com dois filhos, a remoção é mais complicada. Uma solução é substituir a chave de n com a menor chave da subárvore direita de n , e remover o nó n' que armazenava essa chave. Observa que, como n' é uma folha (a mais à esquerda da subárvore direita), não há problemas realizar essa remoção auxiliar. O algoritmo *remover'* da figura 3 realiza esse papel: tem como parâmetro uma ABB t não vazia, e retorna uma ABB t' e uma chave k tal que k é a menor chave de t , e que t' comporta exatamente o conjunto de chaves de t menos a chave k .
- Em todos os casos, se n é a raiz o tratamento é ligeiramente diferente, pois n não tem pai.

O algoritmo *remover* de um valor x numa árvore t , na figura 3 busca recursivamente o nó onde fica o valor a remover (linhas 05–10), até encontrar uma sub-árvore vazia (linhas 03–04, caso onde x não pertencia a t), ou um nó que comporta o valor x (linhas 11–26). Note que o algoritmo constroi também recursivamente a árvore resultante t' .

No caso de uma remoção efetiva, a árvore t pode haver uma das suas sub-árvores vazias, quando t é uma folha ou um nó interior com uma das sub-árvores vazias. Nessa situação, o resultado é a outra sub-árvore (linhas 11–16). Se nenhuma das sub-árvores é vazia, então a o valor de t e sua sub-árvore direita são alterados e atribuídos respectivamente a menor chave da sua sub-árvore direita $right[t]$, e a árvore resultando da remoção desta chave nessa sub-árvore direita. Ambos valores são computados pela função auxiliar *remover'* (linhas 17–18).

O algoritmo auxiliar *remover'* tem como papel, dada uma ABB não vazia t , calcular o par (t', k) , onde k é a menor chave de t , e t' o resultado da remoção de k de t . Esse algoritmo também trabalha de forma recursiva,

<pre> 01 remover($x : \mathcal{K}, t : \mathcal{T}_{\mathcal{K}}$) : $\mathcal{T}_{\mathcal{K}}$ 02 var $t' : \mathcal{T}_{\mathcal{K}}$ 03 if $t = t_{\perp}$ then 04 $t' \leftarrow t_{\perp}$ 05 elsif $key[t] < x$ then 06 $right[t] \leftarrow remover(x, t)$ 07 $t' \leftarrow t$ 08 elsif $key[t] > x$ then 09 $left[t] \leftarrow remover(x, t)$ 10 $t' \leftarrow t$ 11 elsif $left[t] = t_{\perp}$ then 12 $t' \leftarrow right[t]$ 13 $free(t)$ 14 elsif $right[t] = t_{\perp}$ then 15 $t' \leftarrow left[t]$ 16 $free(t)$ 17 else 18 $right[t], key[t] \leftarrow remover'(right[t])$ 19 $t' \leftarrow t$ 20 $\rightarrow t'$ </pre>	<pre> 21 remover'($t : \mathcal{T}_{\mathcal{K}}$) : $\mathcal{T}_{\mathcal{K}} \times \mathcal{K}$ 22 var $t' : \mathcal{T}_{\mathcal{K}}; k : \mathcal{K}$ 23 if $left(t) = t_{\perp}$ then 24 $t', k \leftarrow right[t], key[t]$ 25 $free(t)$ 26 $\rightarrow t', k$ 27 else 28 $left[t], k \leftarrow remover'(left[t])$ 29 $\rightarrow t, k$ 30 </pre>
---	---

Figura 3: Algoritmos de remoção em ABB

procurando pela menor chave de t (linhas 27–30). O caso de base ocorre quando t não tem sub-árvore esquerda. Nessa situação o resultado é o par composto pela sub-árvore direita de t e a chave da raiz de t (linhas 23–26).

7 Construção de ABB otimal no caso de acesso uniforme

Para construir uma ABB otimal (ou seja, completa), as operações de inserção devem ser efetuada conforme a ordem das chaves. Como as inserções sempre ocorrem à nível de folha, deve-se primeiro inserir as chaves que ficaram mais perto da raiz. Se temos um conjunto de chaves $\{c_1, \dots, c_i\}$ tal que $c_1 < \dots < c_i$, deve se inserir o valor médio das chaves, inserir o conjunto das chaves inferior a esse valor, e inserir o conjunto das chaves superior a esse valor.

Exemplo Considera que o conjunto de chaves é $\mathcal{K} = \{1, 2, 3, 4, 5, 6, 7\}$. Uma ordem de inserção otimal é: 4, 2, 6, 1, 7, 5, 3. Os estados correspondentes da ABB são apresentados na figura 4

8 Critérios de avaliação de ABBs

8.1 O comprimento de caminho interno

Um primeiro critério, usado para avaliar uma ABB em relação à operação de busca, é o comprimento de caminho interno, que é a soma dos níveis dos nós da árvore.

Definição 2 (comprimento de caminho interno) *Seja A uma ABB não vazia para um conjunto de chaves $C = \{c_1, \dots, c_i\}$. Seja n_j o nó de A armazenando a chave c_j . O comprimento de caminho interno de A , notado*

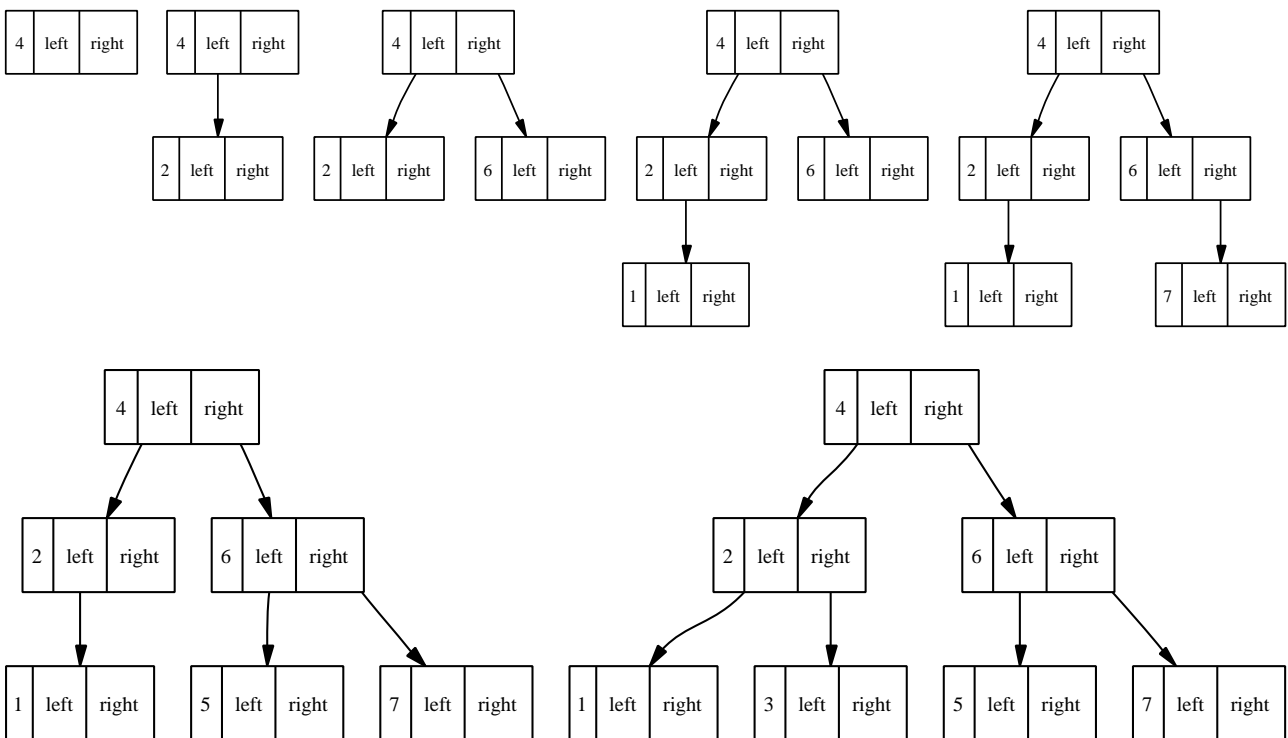


Figura 4: Etapas na construção de uma ABB ótima

$I(A)$ é:

$$I(A) = \sum_{j=1}^i \nu(n_j)$$

Consequentemente, $I(A)/i$ é o número médio de comparações numa busca bem sucedida numa ABB.

Exemplo

8.2 O comprimento de caminho externo

Um segundo critério permite avaliar uma ABB em relação as buscas mal-sucedidas. Seja R o domínio de todas as chaves possíveis ($C \subseteq R$). Sejam

$$\begin{aligned} R_0 &= \{x \in R \mid x < c_1\} \\ R_j &= \{x \in R \mid c_j < x < c_{j+1}\} \\ R_i &= \{x \in R \mid x > c_i\} \end{aligned}$$

Esses conjuntos correspondem aos intervalos de chaves que não se encontram na ABB e, portanto, resultam em buscas mal-sucedidas. Em todo caso, uma busca mal-sucedida se termina numa árvore vazia. Para avaliar o desempenho em caso de busca mal-sucedida numa ABB A , considera-se uma árvore binária A' , onde as subárvores vazias são substituídas por folhas rotuladas com o intervalo dos valores correspondendo as buscas mal-sucedidas em A se terminando na subárvore vazia correspondente. Chamamos A' de árvore das buscas mal-sucedidas da ABB A . Observa que se A tem i nós, A' tem $i + 1$ folhas (proposição 2 do capítulo sobre árvores).

Numa busca sem sucesso numa ABB A corresponde a um caminho da raiz até uma folha f_k da árvore binária A' correspondente. O número de comparações no caso de uma busca mal-sucedida em A terminando na subárvore vazia correspondente à folha f_k é $\nu(l_k) - 1$.

Definição 3 (comprimento de caminho externo) *Seja A uma ABB não vazia e A' a árvore binária das buscas mal sucedidas correspondente. A tem i nós e A' tem $i + 1$ folhas f_0, \dots, f_i . O comprimento de caminho externo de A , notado $E(A)$ é:*

$$E(A) = \sum_{k=0}^i \nu(f_k) - 1$$

Se $|R_k|$ é o número de elementos no intervalo R_k e f_k é a folha da árvore das buscas mal-sucedidas correspondendo a R_k , o tempo médio de busca no caso de uma busca mal-sucedida é:

$$\frac{\sum_{k=1}^i |R_k| \cdot \nu(f_k)}{\sum_{k=1}^i |R_k|}$$

Exemplo

Lema 1 *Seja A uma ABB não vazia, então o comprimento de caminho interno e externo são relacionados pela equação seguinte:*

$$E(A) = I(A) + n$$

Prova:

A prova pode ser feita usando uma indução sobre i , o número de nós.

Caso de base $i = 1$, $I(A) = 1$ e $E(A) = 2$, e a relação é verificada.

Caso geral Suponha que a relação seja verdadeira para qualquer ABB com até $i - 1$ nós. A tem i nós. Considera uma ABB A' , construída a partir de A removendo uma folha f . A' tem $i - 1$ nós.

$$E(A') = I(A') + i - 1$$

De um outro lado, como A' foi obtido de A retirando a folha f , então

$$\begin{aligned} I(A') &= I(A) - \nu(f) \text{ e} \\ E(A') &= E(A) - (\nu(f) + 1) \end{aligned}$$

Portanto, temos

$$\begin{aligned} E(A) - (\nu(f) + 1) &= I(A) - \nu(f) + i - 1 \\ E(A) - \nu(f) - 1 &= I(A) - \nu(f) + i - 1 \\ E(A) &= I(A) + i \end{aligned}$$

■