Árvore Binária de Busca

Siang Wun Song - Universidade de São Paulo - IME/USP

MAC 5710 - Estruturas de Dados - 2008

Referência bibliográfica

Os slides sobre este assunto são parcialmente baseados nas seções sobre árvores binárias de busca do capítulo 4 do livro

- N. Wirth. Algorithms + Data Structures = Programs. Prentice Hall, 1976.

Árvore binária de busca

- Uma árvore binária de busca serve para o armazenamento de dados na memória do computador e a sua subseqüente recuperação.
- Em uma árvore binária de busca cada nó contém um campo chamado *key*, podendo haver outras informações, além dos ponteiros *left* e *right*.
- O campo *key* especifica em geral uma chave que identifica de forma única um determinado registro ou informação. Exemplos de chaves: número de identidade, número CPF, etc. Assim, suporemos que todos os valores de *key* são distintos.

Dado um valor qualquer, deseja-se localizar o nó da árvore, se houver, cujo *key* é igual ao valor dado.

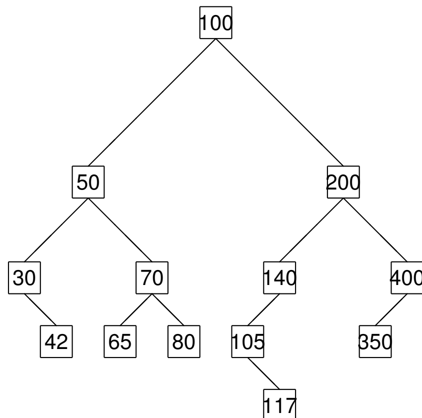
Característica da árvore binária de busca: Para todo nó x da árvore binária de busca, as seguintes condições são verficadas.

- $key(x) > key(y)$, para todo nó y da subárvore esquerda.
- $key(x) < key(y)$, para todo nó y da subárvore direita.

Exemplo de uma árvore binária de busca

Para todo nó x da árvore binária de busca:

- $key(x) \geq key(y)$, para todo nó y da subárvore esquerda.
- $key(x) < key(y)$, para todo nó y da subárvore direita.



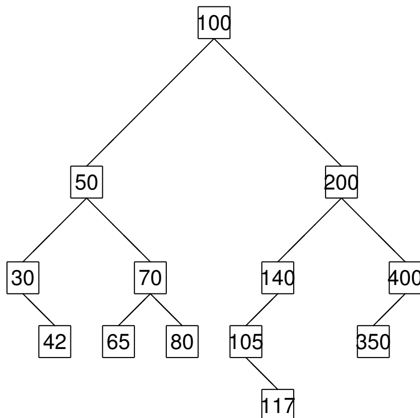
Curiosidade: experimente percorrer esta árvore em in-ordem.

30 42 50 65 70 80 100 105 117 140 200 350 400

Exemplo de uma árvore binária de busca

Para todo nó x da árvore binária de busca:

- $key(x) \geq key(y)$, para todo nó y da subárvore esquerda.
- $key(x) < key(y)$, para todo nó y da subárvore direita.



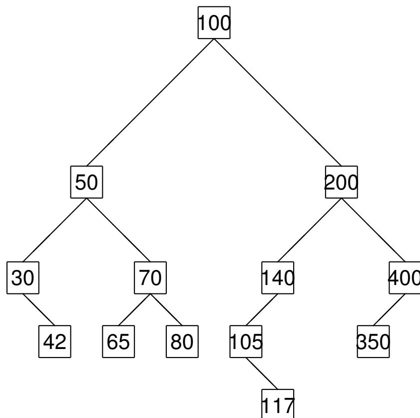
Curiosidade: experimente percorrer esta árvore em in-ordem.

30 42 50 65 70 80 100 105 117 140 200 350 400

Exemplo de uma árvore binária de busca

Para todo nó x da árvore binária de busca:

- $key(x) \geq key(y)$, para todo nó y da subárvore esquerda.
- $key(x) < key(y)$, para todo nó y da subárvore direita.



Curiosidade: experimente percorrer esta árvore em in-ordem.

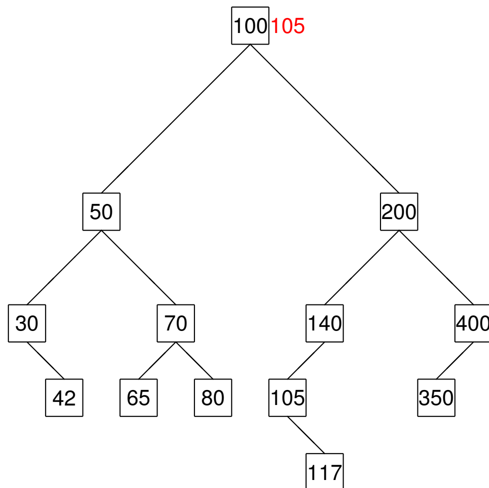
30 42 50 65 70 80 100 105 117 140 200 350 400

Busca numa árvore binária de busca

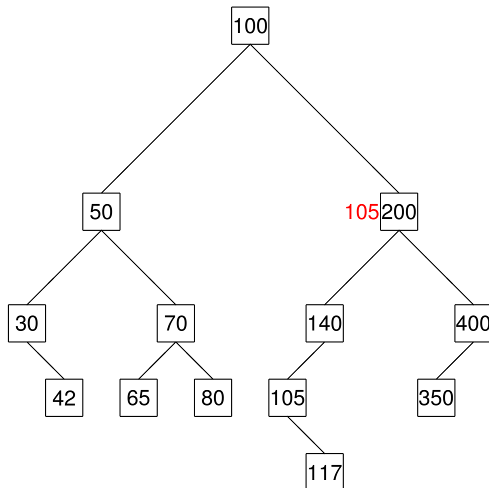
Dado um valor x, deseja-se localizar, se existir, um nó na árvore binária de busca cujo *key* seja igual a x.

- A busca começa pelo nó raiz. Quando a busca chega a um nó qualquer da árvore, ou esse nó já contém o valor procurado e a busca termina, ou ele contém um valor menor ou maior que x. Isso orienta o prosseguimento da busca em apenas uma das subárvores, podendo descartar a outra subárvore.

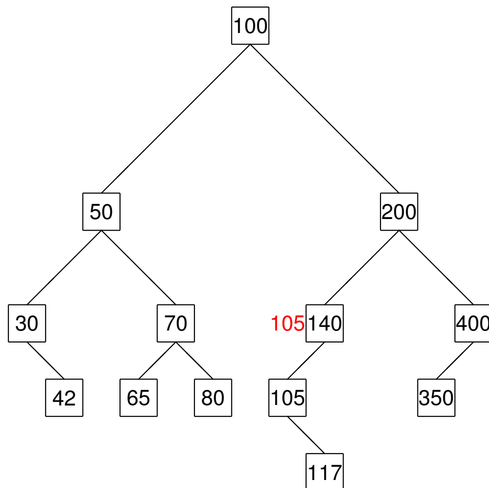
Exemplo: busca do valor 105



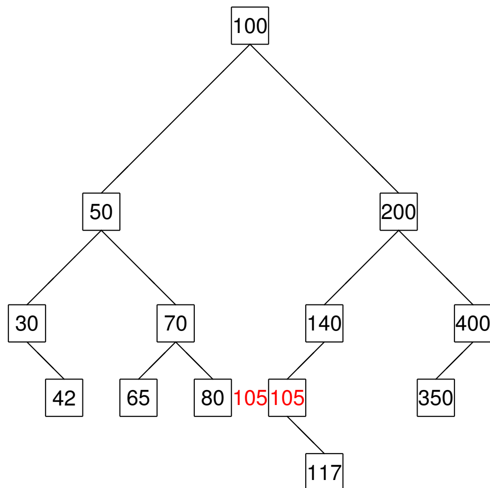
Exemplo: busca do valor 105



Exemplo: busca do valor 105



Exemplo: busca do valor 105 - achou!



Algoritmo de busca numa árvore binária de busca

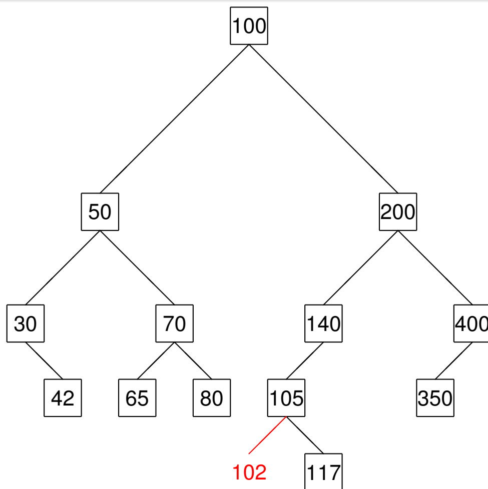
```
function loc(x:integer; t:ref):ref;
var found: boolean;
begin
    found:=false;
    while (t ≠ nil) and not found do
    begin
        if x = t∧.key then
            found:=true
        else
        if x < t∧.key then
            t:=t∧.left
        else
            t:=t∧.right
    end;
    loc:=t
end
```

Complexidade da busca em uma árvore binária de busca

Dado um valor x, deseja-se localizar, se existir, um nó na árvore binária de busca cujo *key* seja igual a x.

- A complexidade do pior caso é igual à altura da árvore. O pior caso pode ser $O(n)$, onde n é o número de nós da árvore. Quando a árvore está balanceada, a busca é eficiente e o pior caso é $O(\log n)$. Veremos o caso médio mais tarde.

Exemplo: busca do valor 102 cai no poneiro nil



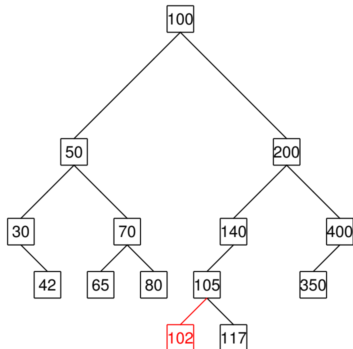
- A busca de um valor inexistente cai no ponteiro nil.
- Esse fato será usado para inserir um elemento novo na árvore binária de busca.

Inserção numa árvore binária de busca

Deseja-se inserir um novo elemento x numa árvore binária de busca.

1. Busca-se x na árvore.
2. Caso não esteja já presente, chega-se a um ponteiro nil e insere-se o novo elemento nesse lugar.

Exemplo: inserir o valor 102 na árvore do exemplo.



Algoritmo de inserção

Obs. Se o elemento a inserir já está na árvore, o algoritmo seguinte incrementa um campo chamado *count* que conta o número de ocorrências desse elemento na árvore.

```
type ref= ∧node;
     node = record
          key:integer;
          count:integer;
          left, right:ref
     end;
```

Algoritmo de inserção

```
procedure insert(x:integer; var p:ref);
{note a chamada por referência do parâmetro p}
begin
    if p=nil then
    begin {não está árvore, então inserir}
        new(p);
        begin
            p∧.key:=x;
            p∧.count:=1;
            p∧.left:=nil;
            p∧.right:=nil
        end
    end
    else if x<p∧.key then
            insert(x,p∧.left)
        else
        if x>p∧.key then
            insert(x,p∧.right)
        else
            p∧.count:=p∧.count+1
end;
```

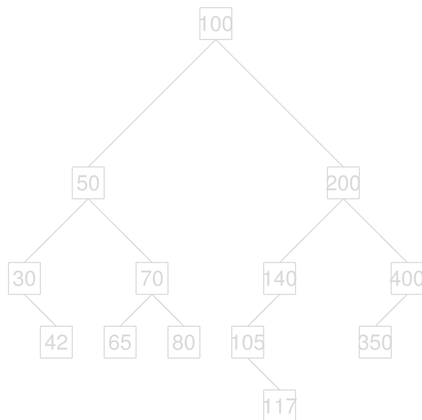
Complexidade de inserção

Seja uma árvore binária de busca de n nós e de altura h.

- A inserção precisa localizar o local para fazer a inserção.
- A complexidade de inserção é $O(h)$.
- No pior caso, a altura de uma árvore binária de busca pode ser $O(n)$.

Os seguintes valores são inseridos, nesta ordem, numa árvore binária de busca inicialmente vazia. Desenhe a árvore resultante.

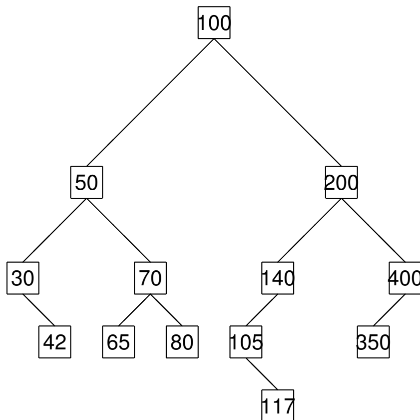
100 200 140 50 105 70 80 30 400 350 117 80 42 65



Exercícios sobre inserção na árvore binária de busca

Os seguintes valores são inseridos, nesta ordem, numa árvore binária de busca inicialmente vazia. Desenhe a árvore resultante.

100 200 140 50 105 70 80 30 400 350 117 80 42 65



Outro exercício de inserção

Agora suponha que os seguintes valores são inseridos, nesta ordem, numa árvore binária de busca inicialmente vazia. Desenhe a árvore resultante.

8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18.

Se esses mesmos valores forem inseridos numa outra ordem, uma árvore diferente pode resultar? Caso sim, mostre uma ordem de inserção que pode produzir a árvore com menor altura?

Remoção da árvore binária de busca

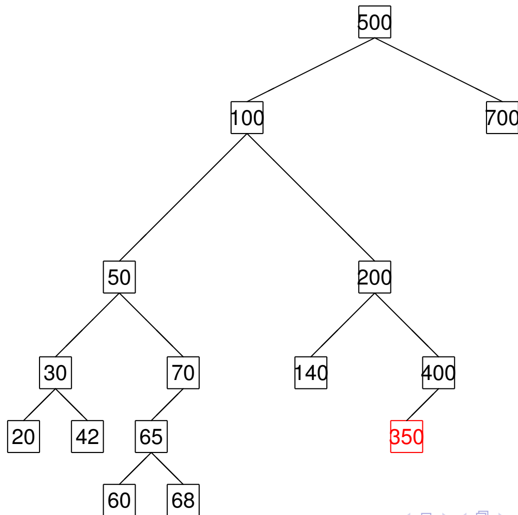
Para remover o nó cujo *key* contém um dado valor x, temos que considerar 3 casos:

1. Não existe nó na árvore com a informação x.
2. O nó contendo x tem 0 ou 1 filho.
3. O nó contendo x tem 2 filhos.

Caso 1: não tem o que fazer.

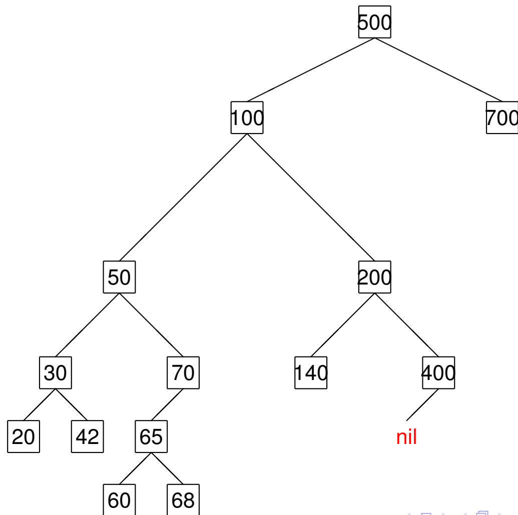
Caso 2a: o nó removido tem 0 filho

Exemplo: remover o nó com o valor 350.



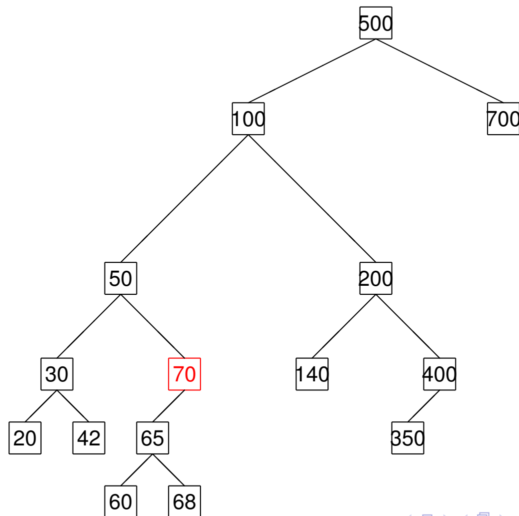
Caso 2a: o nó removido tem 0 filho

O nó pai do nó removido passa a apontar para nil.



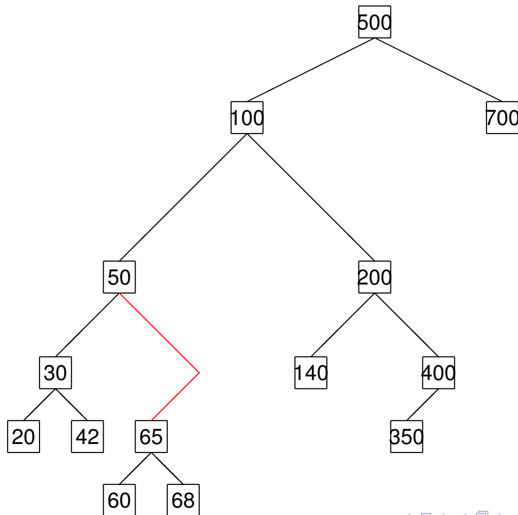
Caso 2b: o nó removido tem 1 filho

Exemplo: remover o nó com o valor 70.



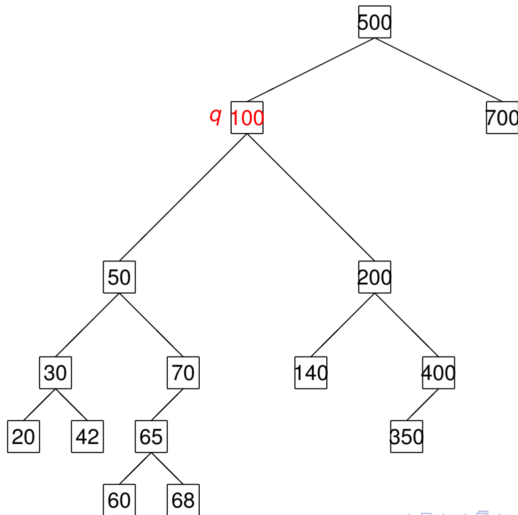
Caso 2b: o nó removido tem 1 filho

O pai do nó removido aponta para o filho do nó removido (i.e. neto).



Caso 3: o nó a ser removido tem 2 filhos

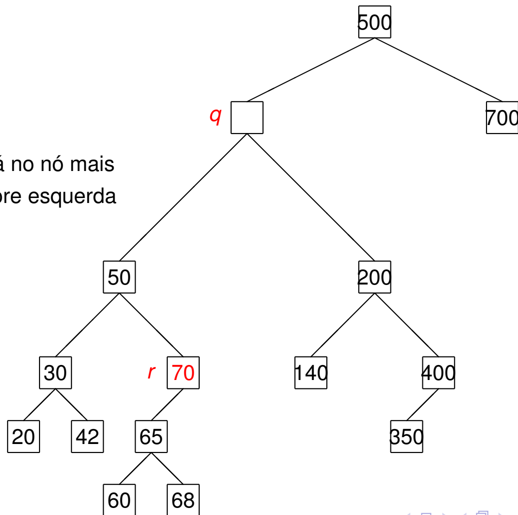
Exemplo: remover o valor 100 do nó q.



Caso 3: o nó a ser removido tem 2 filhos

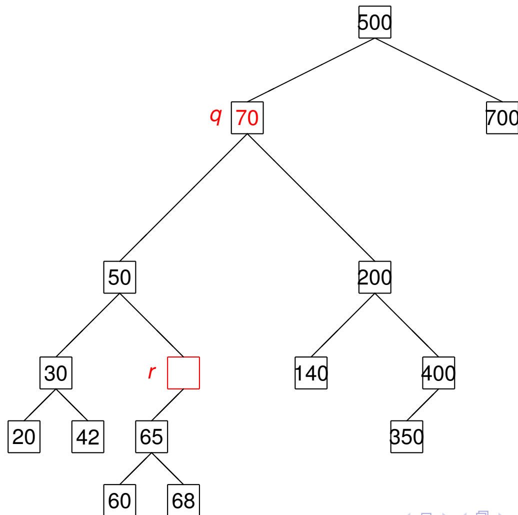
O valor de q é substituído pelo maior valor (r) da subárvore esq.

Esse valor (70) está no nó mais
à direita da subárvore esquerda



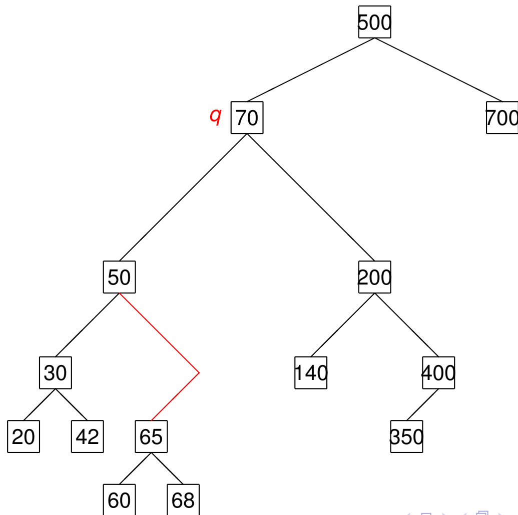
Caso 3: o nó a ser removido tem 2 filhos

Agora remove o nó vazio r, que tem um filho só: recai no caso 2.



Caso 3: o nó a ser removido tem 2 filhos

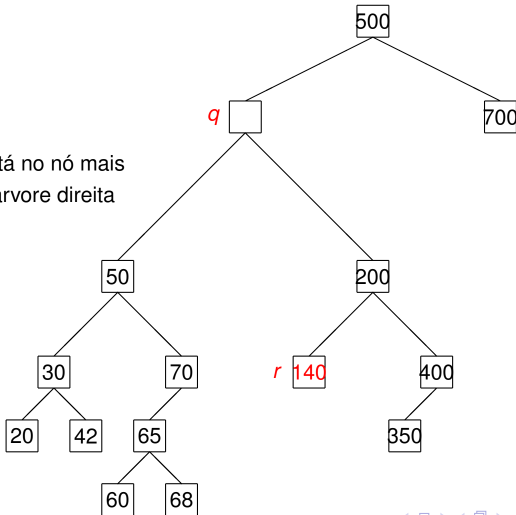
Basta proceder como no caso 2: o nó removido tem 1 filho.



Caso 3: o nó a ser removido tem 2 filhos

Poderia ter substituído o valor de q pelo menor valor (r) da subárvore direita.

Esse valor (140) está no nó mais à esquerda da subárvore direita



Algoritmo de remoção

O procedimento *delete* remove o nó contendo x da árvore apontada por p. O procedimento auxiliar *del* cuida do caso 3. As variáveis q e r são as mesmas do exemplo ilustrativo.

```
procedure delete(x:integer; var p: ref);
    var q:ref;
    procedure del(var r:ref);
    begin
        if r∧.right ≠ nil then
            del(r∧.right)
        else
        begin
            q∧.key := r∧.key;
            q∧.count := r∧.count;
            q := r;
            r := r∧.left
        end
    end;
```

Algoritmo de remoção

```
begin {início proc delete}
     if p = nil then
          writeln('Não está na árvore.')
     else
          if x < p∧.key then
               delete(x, p∧.left)
          else
               if x > p∧.key then
               delete(x, p∧.right)
               else
                    begin
                         q := p;
                         if q∧.right=nil then
                              p:=q∧.left
                         else if q∧.left=nil then
                              p:=q∧.right
                              else del(q∧.left)
                    end;
dispose(q)
end
```

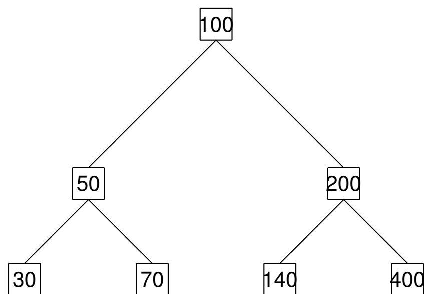
Complexidade de remoção

Seja uma árvore binária de busca de n nós e de altura h.

- A remoção precisa primeiro localizar o nó que contém a chave a ser removida.
- Durante a remoção (caso 2) é necessário localizar o nó mais à esquerda (ou mais à direita) de uma subárvore.
- A complexidade de remoção é $O(h)$.
- No pior caso, a altura de uma árvore binária de busca pode ser $O(n)$.

Exercício de remoção

Seja a árvore binária de busca.



Remova sucessivamente os valores 140, 200, 50 e 100 e desenhe a árvore após cada remoção.

Análise do algoritmo de busca

- Construída uma árvore binária de busca com n chaves e dada uma determinada chave i do conjunto das n chaves, quantas comparações são necessárias para se localizar a chave i na árvore?
- Isso depende da estrutura da árvore, que depende por sua vez da ordem em que as n chaves foram inseridas. Por exemplo, se as n chaves foram inseridas na ordem crescente das mesmas, então se obtém uma árvore degenerada que não ajuda na busca. Por outro lado, as chaves podem ser inseridas de tal forma que se resulta em uma árvore balanceada.
- A pergunta acima também depende da posição da chave i na árvore. Por exemplo, se ela estiver já na raiz, então uma comparação já basta.

No que se segue vamos analisar o algoritmo de busca, em vista dessas considerações.

Comprimento de caminho (*path length*) de um nó

Comprimento de caminho h_i (*path length*) de um nó i.

- Se o valor buscado está num nó i da árvore binária de busca, então temos que percorrer e comparar todos os nós desde a raiz até o nó i.
- Denotamos por comprimento de caminho h_i de um nó i o número de nós encontrados desde a raiz até o nó i.
- A eficiência de busca de um nó i tem a ver portanto com o comprimento de caminho h_i que mede o número de comparações realizadas até localizar o nó i.
- O comprimento de caminho de um nó numa árvore é igual ao nível do nó.

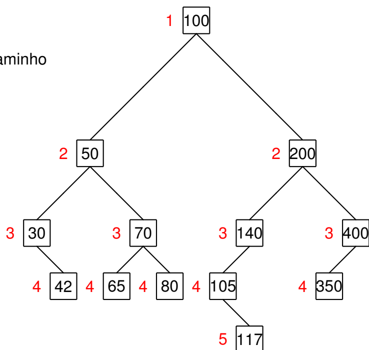
Comprimento de caminho de uma árvore de busca

Comprimento de caminho (*path length*) de uma árvore binária de busca.

- É a soma dos comprimentos de caminhos de todos os nós da árvore:

  $\sum_i h_i$ onde h_i é o comprimento de caminho do nó i.
- Expressa o número total de comparações para se buscar cada nó da árvore.

Comprimento de caminho da árvore = 42



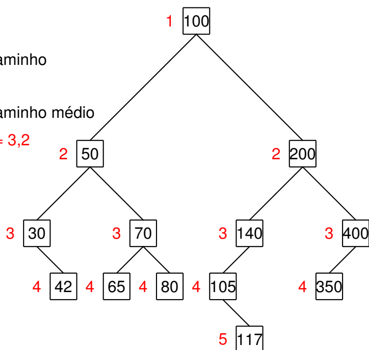
Comprimento de caminho médio a_n de uma árvore binária de busca de n nós.

- É o comprimento de caminho da árvore dividido pelo número de nós da árvore:

  $a_n = (\sum_i h_i)/n$ onde h_i é o comprimento de caminho do nó i.
- Expressa o número médio de comparações para se buscar um nó da árvore, quando cada nó tem igual chance de ser buscado.

Comprimento de caminho da árvore = 42

Comprimento de caminho médio da árvore = 42/13 = 3,2



Análise do algoritmo de busca

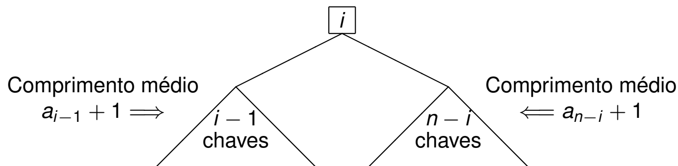
- O valor de a_n expressa o número médio de comparações para se localizar um nó da árvore, quando cada nó da árvore tem igual probabilidade de ser buscado.
- É portanto uma quantidade interessante para medir a eficiência de busca numa árvore binária de busca.
- A pergunta é quanto vale a_n?

O valor de a_n depende da estrutura da árvore construída, que depende por sua vez da ordem em que as n chaves foram inseridas. Pode-se mostrar que:

- No melhor caso, numa árvore perfeitamente balanceada, a_n é aproximadamente $\log n$.
- No pior caso, numa árvore degenerada onde cada nó só tem um filho, a_n é $(n + 1)/2$.
- Quanto vale a_n no caso médio? Primeiro temos que definir o que se entende por caso médio.

Análise do algoritmo de busca no caso médio

Sejam as chaves 1, 2, ..., n, cada uma com igual probabilidade de ser a próxima a ser inserida, numa árvore binária de busca. A árvore construída pode ter qualquer uma dessas chaves na raiz.



A probabilidade de a chave i ser a primeira a ser inserida na árvore, inicialmente vazia, é $1/n$.

Seja $a_n^{(i)}$ o valor do comprimento médio da árvore com n nós estando i na raiz. $a_n^{(i)}$ pode ser expresso em termos dos comprimentos médios de suas subárvores.

$$a_n^{(i)} = (a_{i-1} + 1)\frac{i-1}{n} + 1.\frac{1}{n} + (a_{n-i} + 1)\frac{n-i}{n}$$

Obtenção de a_n

$$
\begin{aligned}
a_n^{(i)} &= (a_{i-1}+1)\frac{i-1}{n} + 1.\frac{1}{n} + (a_{n-i}+1)\frac{n-i}{n} \\
a_n &= \frac{1}{n}\sum_{i=1}^{n}[(a_{i-1}+1)\frac{i-1}{n} + \frac{1}{n} + (a_{n-i}+1)\frac{n-i}{n}] \\
&= \ldots = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i\ a_i
\end{aligned}
$$

Pode-se escrever ainda:

$$a_n = \frac{1}{n^2}[(n^2-1)a_{n-1} - n^2 + 2n - 1]$$

Obtenção de a_n

Escrevendo em termos da função harmônica

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

chegamos à expressão

$$a_n = 2\frac{n+1}{n}H_n - 3$$

Pela fórmula de Euler:

$$H_n = \gamma + \ln(n) + \frac{1}{12n^2} + \ldots \quad \text{onde } \gamma \text{ vale aprox. 0,577.}$$

Para n grande

$$a_n \cong 2[\ln(n) + \gamma] - 3 = 2\ln n - C$$

onde C é uma constante. Portanto $a_n = O(\log n)$.

Obtenção de a_n

Para uma árvore perfeitamente balanceada, temos $a'_n = \log n$.
Para n grande:

$$\lim_{n \to \infty} \frac{a_n}{a'_n} = \frac{2 \ln n}{\log n} = 2 \ln 2 \cong 1,39$$

Assim, no caso médio, há uma melhoria (no número de comparações) de apenas 39% quando a árvore é perfeitamente balanceada.

O caso médio é encorajador, mas se voce quer ter certeza...

- A análise acima mostra que o caso médio é encorajador, com complexidade de $O(\log n)$.
- Para termos certeza de que a busca seja eficiente mesmo no pior caso, podemos usar:
  - Árvores que se reestruturam toda vez que fique desbalanceada por causa de inserções ou remoções efetuadas: e.g. Árvore AVL e árvore rubro-negra.
  - Árvore binária de busca ótima que monta uma árvore eficiente a partir de chaves e informações sobre a probabilidade de acesso de cada uma. Serve para busca, sem inserção nem remoção.