Nota de Aula 5

Bacharelado em Sistemas de Informação | Algoritmos e Estruturas de Dados 2 | Prof. Raimundo Osvaldo

Conteúdo

Árvores Binárias de Busca e Árvores AVL

Referências

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de Dados e Algoritmos em Java**. Porto Alegre: Bookman, 2013

CORMEN, Thomas H. **Algoritmos: Teoria e Prática**. Rio de Janeiro: Elsevier, 2012

ASCENCIO, Ana Fernanda Gomes; ARAÚJO, Graziela Santos de. **Estruturas de Dados:** Algoritmos, Análise de Complexidade e Implementações em Java e C/C++. São Paulo: Pearson, 2010

LAUREANO, Marcos. **Estruturas de Dados e Algoritmos em C**. São Paulo: Brasport, 2010.

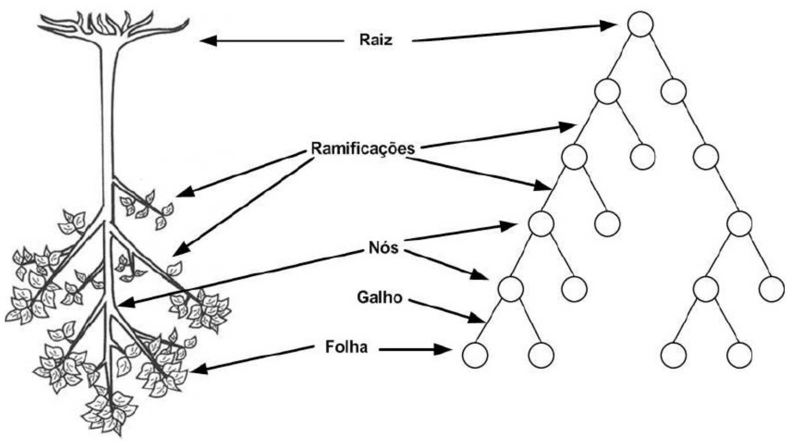
CELES, Waldemar; RANGEL, José Lucas. **Estruturas de Dados**. Apostila – PUC-RIO, 2002.

Resumo Teórico 01: O Básico de Árvores

No contexto da programação e ciência da computação, uma árvore é “uma estrutura de dados que herda as características das topologias em árvore onde os dados estão dispostos de forma hierárquica (um conjunto de dados é hierarquicamente subordinado a outro)” (Laureano, 2010)

Ainda, segunda Laureano (2010), uma árvore é composta por um elemento principal chamado raiz, que possui ligações para outros elementos, que são denominados galhos ou filhos. Estes galhos levam a outros elementos que também possuem outros galhos. O elemento que não possui galhos é conhecido como folha ou nó terminal.

Figura 1: Analogia entre árvores

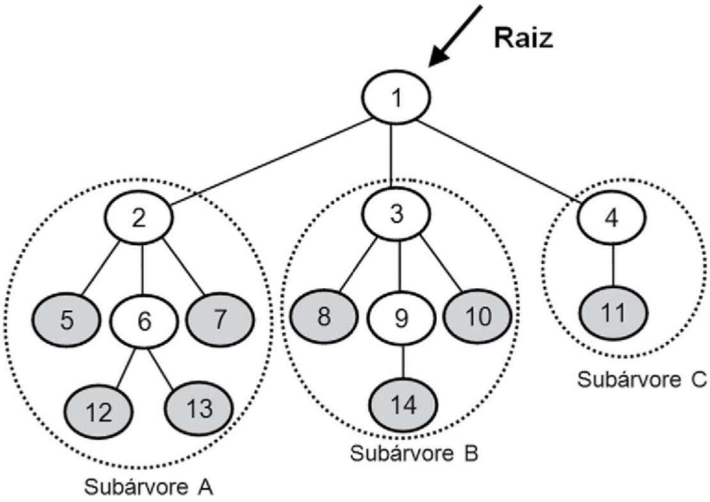


Fonte: Laureano (2010)

As estruturas de dados do tipo árvore são não lineares, isto significa que seus elementos não são armazenados de maneira sequencial nem são todos encadeados (Ascencio; Araújo, 2010). Cada Árvore possui um **nó raiz** e possivelmente várias **subárvores**. Cada subárvore também pode ser considerada uma árvore (e, nesse ponto, a definição passa a ser **recursiva**). Os nós **folhas** podem também ser chamados de **nós externos** e os demais nós podem ser chamados de **nós internos.**

A Figura 2 ilustra a representação usual de uma árvore.

Figura 2 Representação usual de uma árvore



Fonte: Ascêncio e Araújo (2010)

Podemos identificar os nós folha (fundo cinza), as subárvores A, B e C e o nó raiz 1. Além disso, observe que o nó raiz possui três subárvores. Neste caso, dizemos que seu **grau** é 3. Veja também que a subárvore C possui raiz no nó 4 e seu grau é 1. O grau de um nó representa seu número de subárvores (Ascencio; Araújo, 2010).

Árvores Binárias

Uma árvore binária é um conjunto finito de elementos ordenados conforme as seguintes propriedades (Goodrich; Tamassia, 2013):

1 Todos os nós têm no máximo dois filhos;

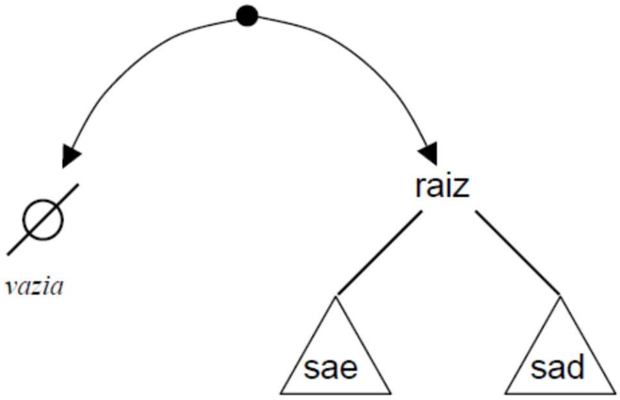
2 Cada nó é rotulado como **filho da esquerda** ou **filho da direita**;

3 O filho da esquerda precede o filho da direita na ordenação dos filhos de um nó.

A Figura 3 ilustra a definição recursiva de árvore binária. Isto é, uma árvore binária T ou é vazia ou consiste em:

- Um **nó raiz** r, que armazena um elemento;
- Uma árvore binária chamada **subárvore esquerda (sae)** de T;
- Uma árvore binária chamada **subárvore direita (sad)** de T.

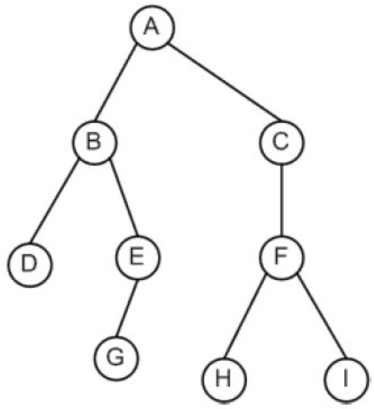
Figura 3 Representação esquemática da estrutura de uma árvore binária



Fonte: Celes e Rangel (2012)

Se vistas isoladamente, cada subárvore compõe uma árvore. As árvores onde cada nó que não seja folha numa árvore binária tem subárvore esquerda e direita não vazias são conhecidas como árvores estritamente binárias. Uma árvore estritamente binária com n folhas tem $n-1$ nós (Laureano, 2010).

Figura 4 Representação de uma árvore binária



Fonte: Laureano (2010)

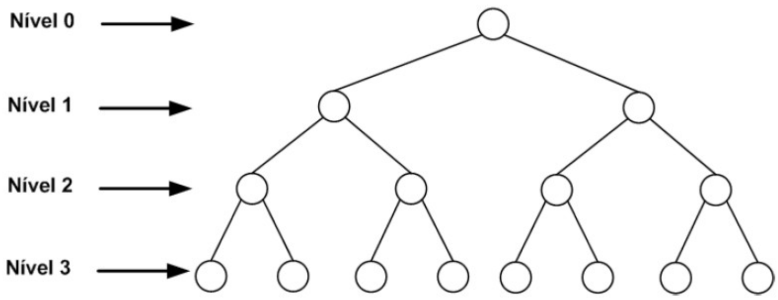
Referente à árvore mostrada na Figura 4, podemos identificar as seguintes relações: os nós B e C são **filhos** de A, ou seja, B e C são **irmãos**. Do mesmo modo, D e E são irmãos, H e I são irmãos. Veja que T_A é a subárvore enraizada em A (toda a árvore) e T_F é a subárvore enraíza em F (contém os nós F, H, I).

Uma árvore binária é **própria** ou **estritamente binária** se cada nó tem zero ou dois filhos. Na Figura 4, a árvore é **imprópria**, pois o nó E tem apenas um filho.

Algumas definições importantes (Laureano, 2010)

- **Caminho**: é uma sequência de nós consecutivos $(n_1, n_2, \ldots, n_{k-1}, n_k)$ que obedece a seguinte relação n_j é pai de n_{j+1}. k nós formam um caminho de comprimento $k-1$. Por exemplo, o comprimento do nó A até o nó G é 3.
- **Nível do nó**: O nível de um nó pode ser definido como o nó raiz de nível 0. Os outros nós têm um nível que é uma unidade a mais do que o nível do seu pai. Veja a Figura 5.

Figura 5: Árvore binária completa de nível 3



Fonte: Laureano (2010)

- **Altura ou profundidade de um nó:** A altura de um nó é o comprimento do maior caminho do nó até alguns de seus descendentes. Descendentes do nó são todos os nós que podem ser alcançados caminhando-se para baixo a partir do nó. A altura de cada uma das folhas é 1. Na Figura 4, a altura de A é 4, a altura de C é 3 e a de E e F é 2.

  A altura de uma árvore é o nível do nó mais distante da raiz.
- **Árvore Completa**: todos os nós com menos de dois filhos ficam no último e penúltimo nível.
- **Árvore Cheia**: árvore estritamente binária e completa.

Propriedades (Goodrich; Tamassia, 2010) (Ascencio; Araújo, 2010)

Em uma árvore binária, o nível 0 tem no máximo 1 nó, o nível 1 tem no máximo 2 nós e, assim por diante. Pode-se, pois, dizer que o nível d tem no máximo 2^d nós.

Vamos denotar por T uma árvore binária não vazia e por n, e, i e h o número de nós, o número de nós externos (folhas), o número de nós externos e a altura de T, respectivamente.

As seguintes propriedades são válidas:

1 $h + 1 \leq n \leq 2^{h+1} - 1$

2 $1 \leq e \leq 2^h$

3 $h \leq i \leq 2^h - 1$

4 $\log(n + 1) - 1 \leq h \leq n - 1$

Além disso, se a árvore é própria, aplicam-se as seguintes propriedades:

1 $2h + 1 \leq n \leq 2^{h+1} - 1$

2 $h + 1 \leq e \leq 2^h$

3 $e = i + 1$

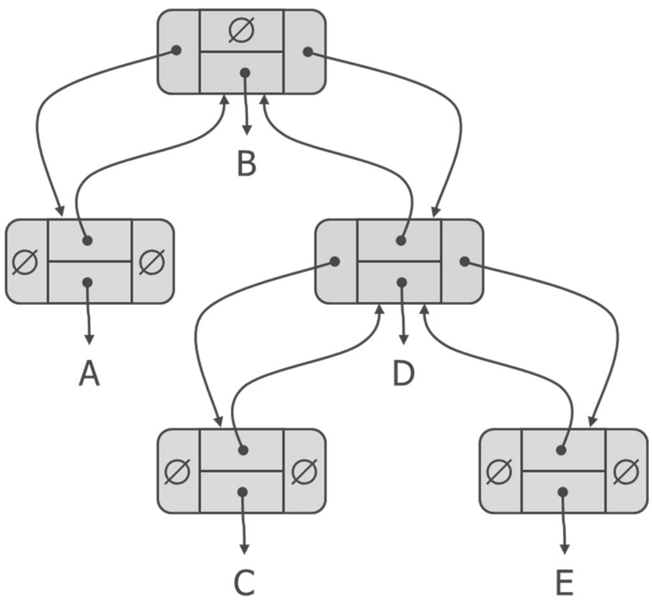
4 $\log(n + 1) - 1 \leq h \leq (n - 1)/2$

5 $n = 2e - 1$

Representação (Cormen *et al.*, 2012)

Podemos representar uma árvore binária por uma estrutura de dados ligada, na qual cada nó é um objeto. Além de uma chave de dados, cada nó contém atributos *esquerda* e *direita* (Figura 6). Podemos, ainda, acrescentar um atributo *p* que apontar para o nós pai.

Figura 6*:* Árvore binária com vários elementos



Fonte: Goodrich e Tamassia (2010)

Se um filho estiver ausente, o atributo adequado contém o valor **`null`**. O nó raiz é o único cujo pai é **`null`**.

Percurso em Árvores Binárias (Celes; Rangel, 2002)

Muitas operações em árvores binárias envolvem o percurso de todas as subárvores, executando alguma ação de tratamento em cada nó, de forma que é comum percorrer uma árvore em uma das seguintes ordens:

- **Pré-ordem**: trata raiz, percorre sae, percorre sad;
- **Em ordem**: percorre sae, trata raiz, percorre sad;
- **Pós-ordem**: percorre sae, percorre sad, trata raiz.

Se x é raiz de uma subárvore de n nós, então a chamada a uma função que percorre uma árvore binária *Em ordem* demora o tempo $\Theta(n)$.

Resumo Teórico 02: Árvores Binárias de Busca

Em uma Árvore Binária de Busca (ABB), as informações armazenadas na subárvore esquerda são menores do que informação armazenada no nó raiz, e as informações armazenadas na subárvore direita são maiores do que a informação armazenada no nó raiz.

Formalmente, as chaves em uma árvore binária de busca são sempre armazenadas de modo a satisfazer a **propriedade de árvore binária de busca:** *seja x um nó em uma árvore binária de busca. Se y é um nó na subárvore esquerda de x, então $y.chave \leq x.chave$. Se y é um nó na subárvore direita de x, então $x.chave \geq y.chave$* (Cormen *et al.*, 2012).

O objetivo de organizar dados em árvores de busca binária é facilitar a tarefa de procura de um determinado valor. A partir da raiz e de posse da informação a ser encontrada, é possível saber qual o caminho (galho) a ser percorrido até encontrar o nó desejado. Para tanto, basta verificar se o valor procurado é maior, menor ou igual ao nó que se está posicionando (Laureano, 2010)

Em resumo: o objetivo dessas árvores é minimizar o tempo de acesso no pior caso. Para isso, usa a seguinte ideia: para cada chave, separe as demais em maiores ou menores.

Operações em Árvores Binárias de Busca

Inserção

O procedimento busca pelo valor na árvore. Se o elemento não existir na árvore, é alcançada a folha, e então inserido o valor nesta posição. É examinada a raiz e introduzido um novo nó na subárvore da esquerda, se o valor novo é menor do que a raiz, ou na subárvore da direita, se o valor novo for maior do que a raiz.

```
Tree-insert(T, z){
   y = null;
   x = T.raiz;
   while (x != null){
      y = x;
      if(z.chave < x.chave)
         x = x.esquerda;
      else
         x = x.direita;
```

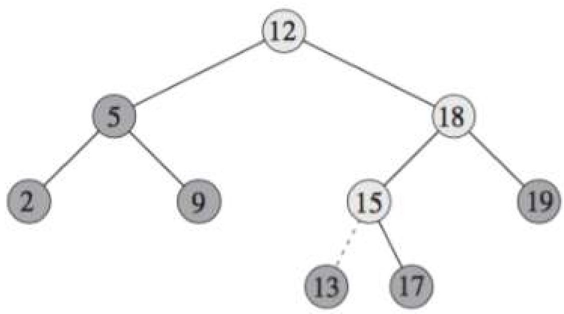
```
    }
    z.p = y;
    if(y == null)
        T.raiz = z;
    else{
        if(z.chave < y.chave)
            y.esquerda = z;
        else
            y.direita = z;
    }
}
```

É possível também fazer uma implementação recursiva

```
Tree-insert(T, z){
    if(T == null)
        T = z;
    else{
        if(z.chave < T.raiz.chave){
            T = T.raiz.esquerda;
            Tree-insert(T, z);
        }
        else{
            if(z.chave > T.raiz.chave){
                T = T.raiz.direita;
                Tree-insert(T, z);
            }
        }
    }
}
```

Vamos analisar a inserção de um valor (13) na árvore binária de busca ilustrada na Figura 07.

Figura 7*:* Inserção de um item numa árvore binária de busca



Fonte: Cormen et al. (2012)

Os nós 12 – 18 – 15 são o caminho percorrido até a posição onde o elemento é inserido.

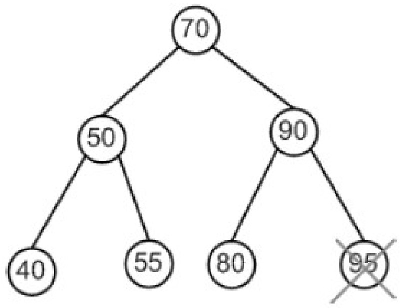
Considerando uma árvore de altura mínima, na operação de inserção, o nó sempre é inserido em uma folha, tendo que percorrer todos os nós desde a raiz, até uma folha onde o nó será inserido. Gasta-se, para isso, a altura da árvore, isto é, $O(h)$.

Remoção

A estratégia global para remover um nó z de uma árvore de busca binária T tem **três casos básicos**:

1 Se z não tem nenhum filho, apenas removemos e modificamos seu pai para substituir z por `null`.

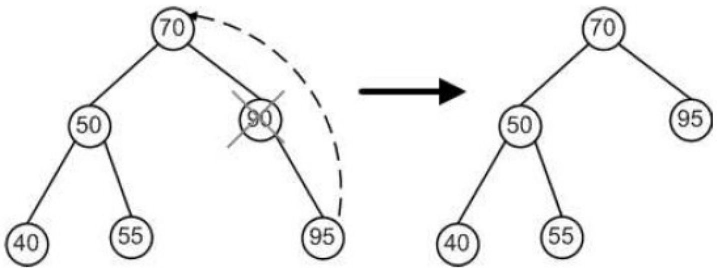
Figura 8: Remoção de um nó folha



Fonte: Laureano (2010)

2 Se o nó tem apenas um filho, elevamos esse filho para que ocupe a posição de z. Modificamos o pai de z para substituir z pelo filho de z.

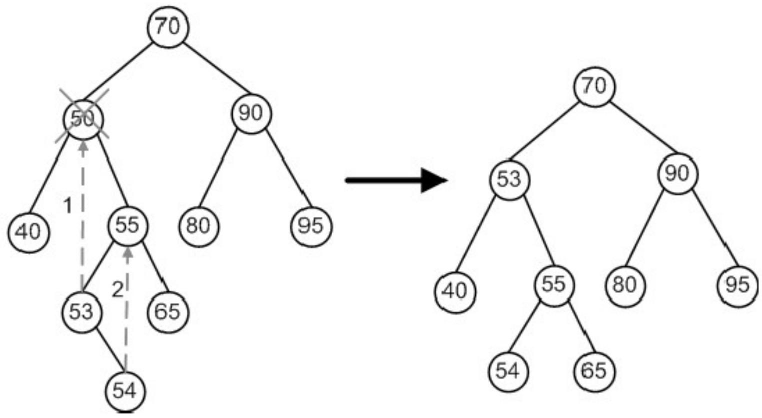
Figura 9: Remoção de um nó com um filho



Fonte: Laureano (2010)

3 Se z tiver dois filhos, encontramos o sucessor de z (y), que deve estar na subárvore direita de z, e fazemos y tomar a posição de z na árvore. Em seguida, o resto da subárvore direita original de z torna-se a nova subárvore direita de y. A subárvore esquerda de z torna-se a nova subárvore esquerda de y.

Figura 10: Remoção de um nó com dois filhos

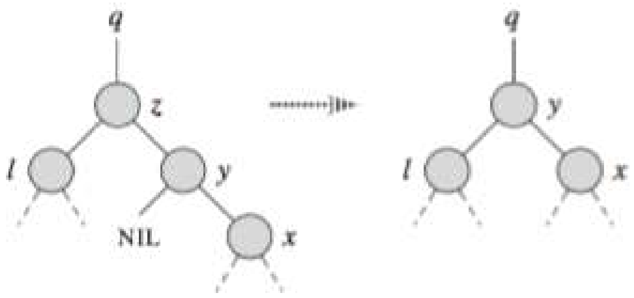


Fonte: Laureano (2010)

Esse caso pode ser visto desdobrado em dois:

3.1 O nó z tem dois filhos; seu filho à esquerda é o nó l, seu filho à direita é seu sucessor y, e o filho à direita de y é o nó x. Substituímos z por y, atualizando o filho à esquerda de y para que se torne l, mas deixando x como filho à direita de y (Figura 11)

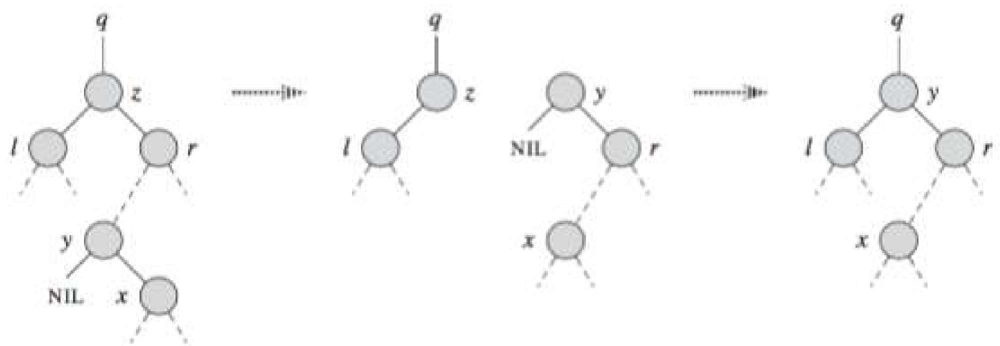
Figura 11: Eliminação de um nó de uma árvore binária de busca - Caso 3.1



Fonte: Cormen *el al.* (2012)

3.2 O nó z tem dois filhos (o filho à esquerda l e o filho à direita r), e seu sucessor $y \neq r$ encontra-se dentro da subárvore enraizada em r. Substituímos y por seu próprio filho à direita x, e definimos y como pai de r. Então, tomamos y filho de q e pai de y.

Figura 12: Eliminação de um nó de uma árvore binária de busca - caso 3.2



Fonte: Cormen *el al.* (2012)

Para movimentar subárvores dentro da árvore binária de busca, definimos um método `transplant`, que substitui a subárvore enraizada no nó u pela subárvore enraizada no nó v, o pai do nó u torna-se pai do nó v, e o pai de u acaba ficando com v como seu filho adequado.

```
transplant(T, u, v){
   if(u.p == null)
      T.raiz = v;
   else{
      if(u == u.p.esquerda)
         u.p.esquerda = v;
      else
         u.p.direita = v;
   }
   if(v != u.p)
       v.p = u.p;
}
```

A seguir vamos mostrar o código para um método que executa os quatro casos para remoção de um nó em uma árvore binária de busca.

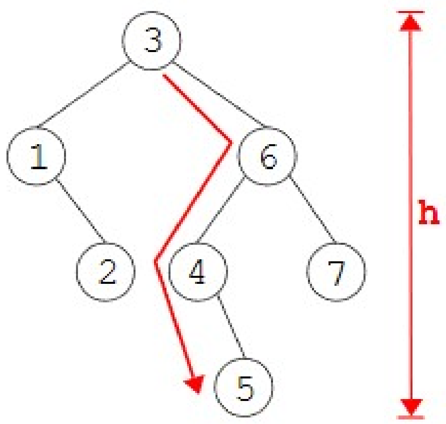
Cada linha desse método, incluindo as chamadas a `transplant`, demora tempo constante, exceto a chamada a `tree-minimum`. Assim, `tree-delete` é executado no tempo $O(h)$ em uma árvore de altura h.

```
tree-delete(T, z){
   if(z.esquerda == null)
      transplant(T, z, z.direita);
   else{
      if(z.direita == null)
         transplant(T, z, z.esquerda);
      else{
         y = tree-minimum(z.direita);
         if(y.p != z){
            transplant(T, y, y.direita);
            y.direita = z.direita;
            y.direita.p = y;
         }
         transplant(T, z.y);
         y.esquerda = z.esquerda;
         y.esquerda.p = y;
      }
   }
}
```

Busca

Consiste na procura por um nó com determinada chave em uma árvore de busca binária. A busca começa na raiz e segue um caminho simples, descendo a árvore. Esse caminho é formado pelos nós encontrados durante a recursão. Neste caso, o tempo de execução é $O(h)$, onde h é a altura da árvore:

Figura 13: Busca em uma árvore binária



Fonte: http://professor.ufabc.edu.br/~leticia.bueno/classes/aed2/materiais/avl.pdf

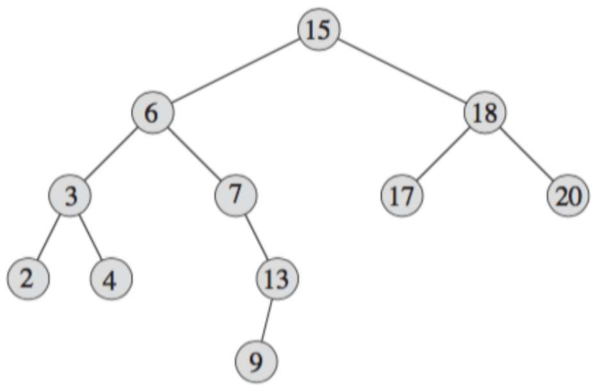
```
tree-search(x, k){
   if(x == null || k == x.chave)
      return x;
   if(k < x.chave)
      return tree-search(x.esquerda, k);
   else
      return tree-search(x.direita, k);
}
```

O mesmo procedimento pode ser reescrito de modo iterativo, que, por sinal, é mais eficiente em alguns computadores

```
iterative-tree-search(x, k){
   while(x != null && k != x.chave){
      if(k < x.chave)
         x = x.esquerda;
      else
         x = x.direita;
   }
   return x;
}
```

Para procurar a chave 13, o algoritmo segue o caminho 15 – 6 – 7 – 13 partindo da raiz (Figura 14).

Figura 14: Pesquisa numa árvore binária



Fonte: Cormen el al. (2012)

Mínimo e Máximo

É possível encontrar um elemento em uma árvore binária de busca cuja chave é o valor mínimo. Para isso, basta seguir as referências de filho da esquerda desde a raiz até encontrar um valor `null`.

```
tree-minimum(x){
   while(x.esquerda != null)
      x = x.esquerda;
   return x;
}
```

Se um nó x não tem subárvore esquerda, a chave mínima é $x.chave$. O pseudocódigo para o valor máximo é simétrico.

```
tree-maximum(x){
   while(x.direita != null)
      x = x.direita;
   return x;
}
```

Ambos os procedimentos são executado no tempo $O(h)$, onde h é a altura da árvore, uma vez que a sequência de nós forma um caminho simples descendente, a partir da raiz.

Resumo Teórico 03: Árvores AVL

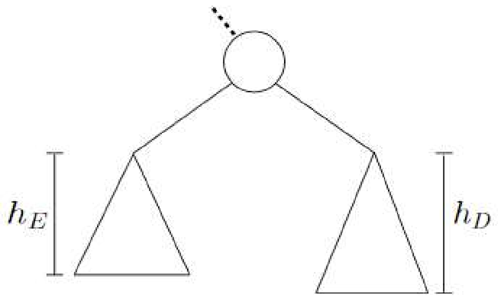
Uma árvore AVL é uma árvore binária balanceada, ou seja, é aquela na qual as alturas das subárvores esquerda e direita de cada nó diferem no máximo por um. Se a diferença de altura entre as subárvores de um nó é maior que 1 ou menor que -1, a árvore está desbalanceada.

Na implementação, para cada nó x definimos os seguintes dados:

$h_E(x)$: altura da subárvore esquerda

$h_D(x)$: altura da subárvore direita

Figura 15*:* Esquema de uma árvore AVL

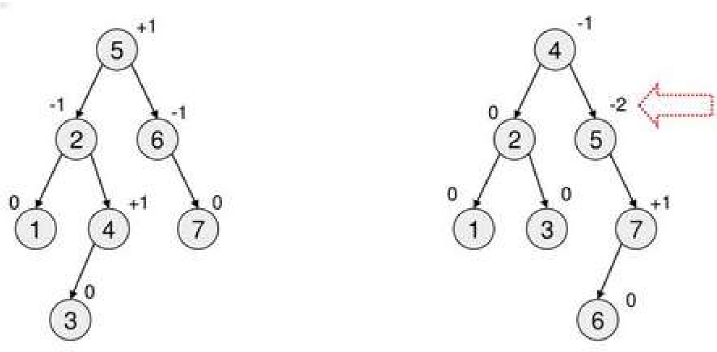


Fonte: http://professor.ufabc.edu.br/~leticia.bueno/classes/aed2/materiais/avl.pdf

Em uma árvore AVL, **para cada nó** deve ser **válida** a propriedade AVL:

$$|\boldsymbol{h_E(x) - h_D(x)}| \leq \mathbf{1}$$

Figura 16*:* Exemplo ilustrativo: a árvore da esquerda é AVL e a da direita não é AVL



Fonte: https://slideplayer.com.br/slide/14798097/

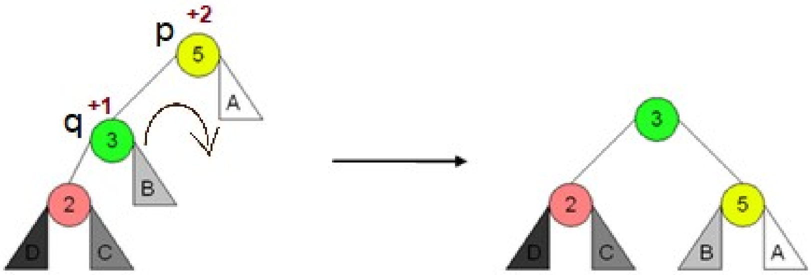
Uma consequência imediata da propriedade AVL é que uma subárvore de uma árvore AVL também é uma árvore AVL. Tal propriedade também é importante para manter a altura da árvore na ordem de $O(\log n)$, onde n é o total de nós da árvore.

As operações básicas são similares àquelas apresentadas para as árvores binárias de busca, porém para árvores AVL é preciso verificar a propriedade AVL e executar operações de reestruturação da árvore, quando ela estiver desbalanceada. Essas operações são geralmente chamadas de **rotações**.

Rotação simples à direita

Deve ser executada quando um nó tem um fator positivo (+2) e seu filho à esquerda tem fator positivo (+1). Por exemplo (Figura 17), o filho à esquerda (3) assume a posição do nó desbalanceado (5). O filho à direita de 3 passa ser o filho à esquerda de 5 e 5 passa a ser o filho à direita de 3.

Figura 17: Rotação simples à direita



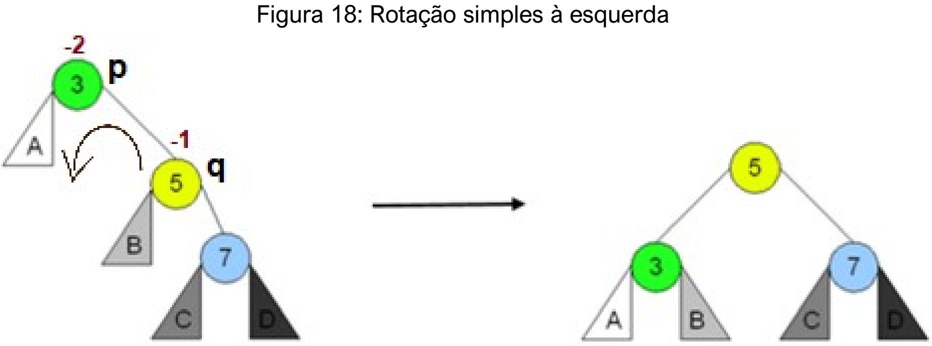
Fonte: Wikipedia

```
Rotação-direita(p){
   q = p.esquerda;
   temp = q.direita;
   q.direita = p;
   p.esquerda = temp;
   p = q;
}
```

Rotação simples à esquerda

Deve ser executada quando um nó tem um fator negativo (-2) e seu filho à direita tem fator negativo (-1). Nesse caso, o filho à direita (5) assume a posição do nó desbalanceado (3). O filho à esquerda de 5 passa ser o filho à direita de 3 e 3 passa a ser o filho à esquerda de 5.

Figura 18: Rotação simples à esquerda



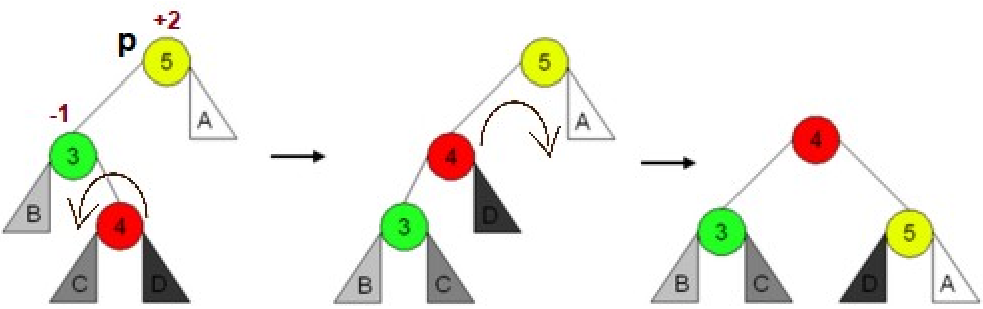
Fonte: Wikipedia

```
Rotação-esquerda(p){
   q = p.direita;
   temp = q.esquerda;
   q.esquerda = p;
   p.direita = temp;
   p = q;
}
```

Rotação dupla à direita

Deve ser executada quando um nó tem fator positivo e seu filho à esquerda tem fator negativo. Faz-se uma rotação à esquerda no nó filho da esquerda e uma rotação à direita no próprio nó.

Figura 19: Rotação dupla a direita



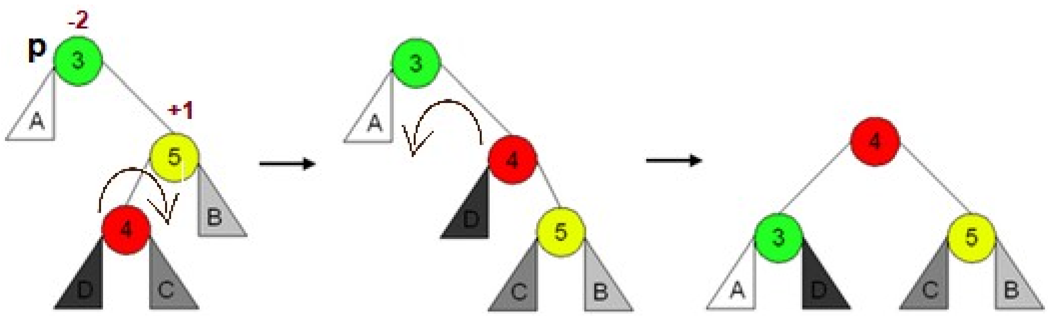
Fonte: Wikipedia

```
Rotação-dupla-direita(p){
   Rotação-esquerda(p.esquerda);
   Rotação-direita(p);
}
```

Rotação dupla à esquerda

deve ser executada quando um nó tem fator positivo e seu filho à esquerda tem fator negativo. Faz-se uma rotação à esquerda no nó filho da esquerda e uma rotação à direita no próprio nó.

Figura 20: Rotação dupla à esquerda



Fonte: Wikipedia

```
Rotação-dupla-esquerda(p){
   Rotação-direita(p.direita);
   Rotação-esquerda(p);
}
```

Operações em Árvores AVL

Inserção

A intenção é inserir um novo nó, em tempo razoável, e manter o balanceamento da árvore.

A estratégia é a seguinte:

1 Inserir como numa árvore binária comum;

2 Verificar a propriedade AVL para cada nó (atualizar os fatores de balanceamento);

3 Executar operações de reestruturação da árvore, quando necessário.

Remoção

A remoção acontece como nas árvores binárias de busca, entretanto se algum nó ocupar o lugar do nó removido, é necessário atualizar o fator de balanceamento desde o pai desse nó até a posição atual do nó. Percorre-se o caminho desde o pai do nó removido até a raiz, fazendo as operações de rotação necessárias

Operações de **busca**, **inserção** e **remoção** são realizadas no tempo $O(\log n)$, onde n é o total de nós da árvore.

Atividades Propostas

1. O professor Amongus afirma que a ordem na qual um conjunto fixo de itens é inserido em uma árvore binária de busca não interessa – sempre resulta na mesma árvore. Apresente um pequeno exemplo para demonstrar que ele está errado?
2. Insira em uma árvore AVL, itens com as chaves apresentadas nos itens a seguir (na ordem em que aparecem). Desenhe a árvore resultante da inserção, sendo que uma nova árvore deve ser desenhada quando houver uma rotação. Indique qual a rotação que foi executada.

   (a) 50, 30, 20, 70, 40, 35, 37, 38, 10, 32, 45, 42, 25, 47, 36.

   (b) 100, 80, 60, 40, 20, 70, 30, 50, 35, 45, 55, 75, 65, 73, 77
3. Escreva um algoritmo que verifica se uma dada árvore binária é do tipo AVL. Suponha já existente uma função `p.altura()` que retorna a altura de uma árvore binária referenciada por p.
4. Implemente uma versão, a seu critério, do TAD Árvore AVL.
5. Em equipe (a mesma do trabalho anterior), escolha um dos cenários a seguir e faça uma implementação simples na linguagem de sua preferência. A implementação é apenas ilustrativa e não precisa fazer uso de interface gráfica e acesso a banco de dados. A apresentação será por equipe em horário combinado com o professor. **Prazo final**: 16 de dezembro.

   Cenário 01: em um laboratório de pesquisa, vários alunos e pesquisadores utilizam máquinas que exigem autenticação local. O ambiente possui grande rotatividade, com usuários sendo adicionados e removidos constantemente. Atualmente, o cadastro é mantido em um arquivo simples, mas as consultas têm ficado lentas e difíceis de gerenciar. Para resolver isso, os estudantes devem implementar um índice eficiente de usuários utilizando uma árvore AVL, onde cada usuário é identificado por um ID numérico único que servirá de chave para inserção e busca. O registro deverá conter também o nome de login associado ao ID. O sistema deve permitir cadastrar novos usuários, remover usuários desativados, autenticar alguém buscando o ID na AVL e exibir todos os usuários em ordem crescente. Assim, o laboratório terá uma estrutura organizada, com consultas previsíveis e de alto desempenho.

Cenário 02: uma oficina mecânica mantém um estoque de peças automotivas com códigos numéricos definidos pelos fornecedores. A oficina percebeu que as consultas ao estoque se tornaram lentas, pois o sistema atual utiliza uma lista encadeada que cresce rapidamente. Os mecânicos precisam verificar com frequência se uma peça está disponível, quantas unidades há e qual o preço. O novo sistema deve utilizar uma árvore AVL para armazenar as peças usando o código como chave, garantindo que inserções, buscas e remoções sejam eficientes. Cada nó da árvore deverá incluir o código da peça, o nome, a quantidade disponível e o preço unitário. A aplicação deverá permitir cadastrar peças novas, atualizar quantidades, remover peças descontinuadas, consultar rapidamente uma peça específica e exibir o estoque completo em ordem crescente. Uma funcionalidade adicional pode listar peças que estão com estoque baixo, percorrendo a árvore e aplicando um limite configurado pelo usuário.

Cenário 03: um professor universitário precisa gerenciar uma agenda cheia de reuniões, orientações, atividades administrativas e compromissos diversos. Cada evento é associado a uma data e horário específicos, representados numericamente no formato AAAAMMDDHHMM. Como os compromissos mudam constantemente, alguns são desmarcados, outros adicionados, o professor necessita de um sistema que organize todos os eventos em ordem cronológica e permita consultas rápidas. A aplicação deve usar uma árvore AVL, onde o timestamp será a chave para manter os eventos sempre ordenados. Cada compromisso deverá ter uma descrição textual e poderá ser inserido, removido ou consultado rapidamente mesmo após muitas alterações na agenda. Além da listagem completa dos compromissos em ordem crescente, o sistema deve possibilitar consultas por intervalo de datas, percorrendo apenas as partes relevantes da árvore, garantindo desempenho mesmo em agendas extensas.

Cenário 04: um jogo simples desenvolvido por estudantes registra a pontuação de cada jogador ao final de cada partida. Como o número de partidas é grande, o ranking precisa se adaptar constantemente à chegada de novas pontuações, oferecendo uma visão atualizada da colocação de cada jogador. Para isso, será utilizada uma árvore AVL onde a pontuação numérica é a chave do nó. Cada entrada deverá armazenar também o nome do jogador e, opcionalmente, a data e horário em que a pontuação foi registrada. A aplicação deve permitir inserir novas pontuações, remover registros antigos, consultar rapidamente a posição de determinada pontuação no ranking e listar todas as pontuações ordenadas. O sistema deve ainda fornecer as maiores e menores pontuações com eficiência, explorando as extremidades da árvore. Com isso, o jogo terá um ranking dinâmico, estável e com alto desempenho.