Árvores AVL

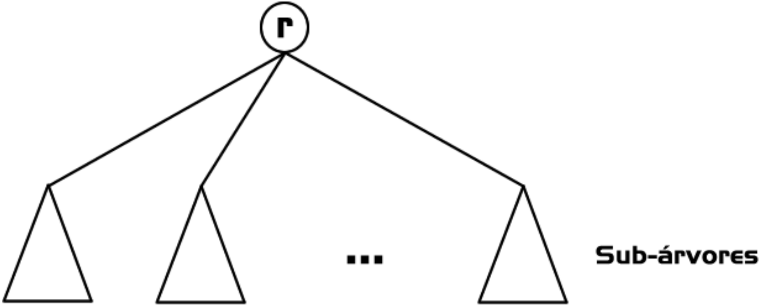
Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados 2 (AE43CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

Sumário

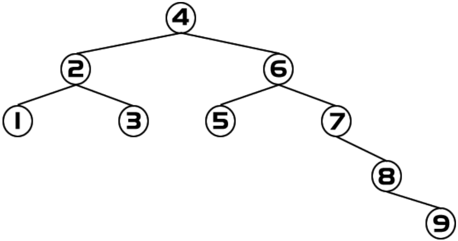
- Balanceamento
- Árvores AVL
  - Fator de balanceamento
  - Rotações
  - Operações

- Árvore



Considerações Iniciais

- As árvores binárias de busca (pesquisa) são projetadas para um acesso rápido à informação
  - Idealmente a árvore deve ser razoavelmente equilibrada
- Tempo de busca é de $O(\log n)$ para uma árvore balanceada
- Sucessivas inserções de itens podem acarretar no aumento da complexidade de tempo para $O(n)$

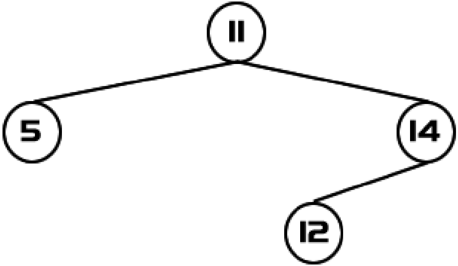


Balanceamento

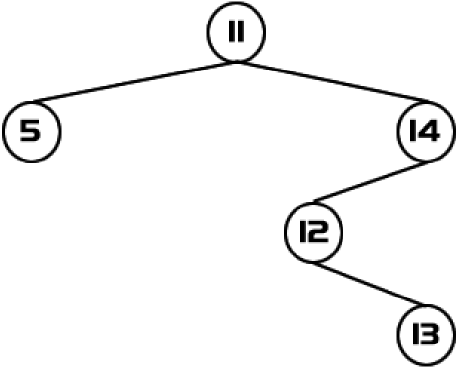
Balanceamento

- Árvores binárias de busca balanceadas minimizam o número de comparações em comparação com o pior caso ($O(n)$)
  - A altura da árvore é mantida baixa (por volta de $O(\log n)$) após sucessivas inserções
  - Uma árvore de altura h pode conter, no máximo, $2^{h+1} - 1$ elementos
  - A diferença de altura das sub-árvores direita e esquerda deve ser no máximo um (para uma árvore AVL)
- A manutenção de árvores de busca balanceadas é considerada uma tarefa complexa

- Árvore balanceada

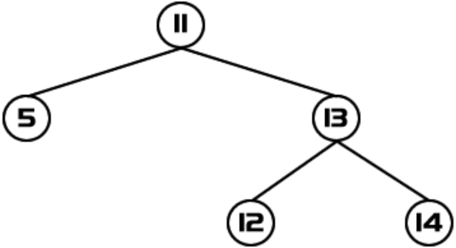


- Inserção do item 13 torna a árvore desbalanceada



Balanceamento

- Árvore rebalanceada



Balanceamento

- Exemplos de tipos de árvores binárias balanceadas:
  - Árvores AVL
  - Árvores vermelha-preta (rubro-negra)

Árvores AVL

Árvores AVL

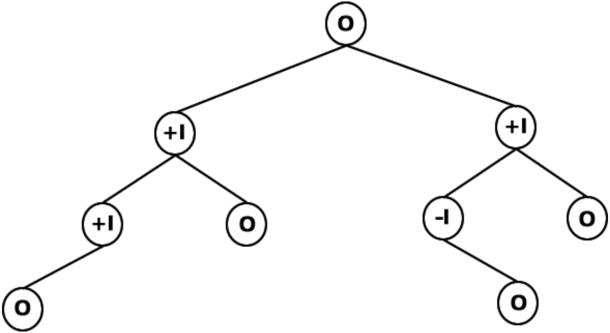
- Adelson, Velsky e Landis (1962)
- Árvore de altura balanceada
- As operações de busca, inserção e remoção podem ser realizadas a um custo de tempo $O(\log n)$
- Uma árvore vazia é uma árvore AVL

- Dada pela diferença de altura entre as sub-árvores esquerda (h_e) e direita (h_d)
  - $h_e - h_d$

- Em uma Árvore AVL, cada sub-árvore deve ter altura equilibrada (de acordo com o fator de balanceamento)

- Cada nó de uma árvore AVL deve ter um valor de fator de balanceamento
  - -1: a altura da sub-árvore direita é maior que a da esquerda
  - 0: a altura das sub-árvores direita e esquerda são iguais
  - +1: a altura da sub-árvore esquerda é maior que a da direita
- Em uma operação de inserção ou remoção, caso uma sub-árvore fique com altura menor que -1 ou maior que +1, a árvore deve ser rebalanceada

- Exemplo de árvore balanceada com fator de balanceamento em cada nó

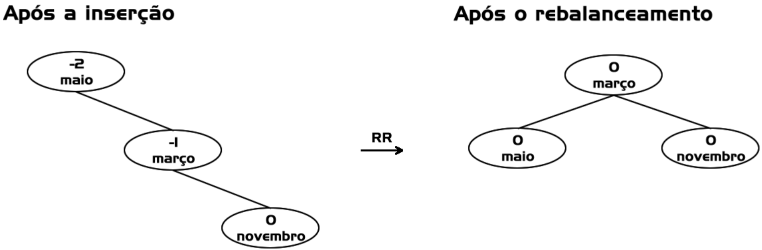


- *Left-left* (LL)
- *Right-right* (RR)
- *Left-right* (LR)
- *Right-left* (RL)

- Exemplo (quando os elementos são *strings*, levaremos em conta a sua ordem alfabética)
  - Inserção do item *maio*
    - Inicialmente, a árvore está vazia e após a inserção, o balanceamento é mantido

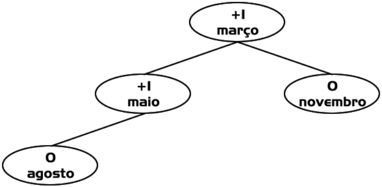
- Exemplo
  - Inserção do item *março*
    - Na ordem alfabética, *março* vem depois de *maio*, já que *mar* é maior que *mai*, ou seja, o novo item é adicionado à direita de *maio*
    - Após a inserção, a árvore ainda é mantida balanceada

- Exemplo
  - Inserção do item *novembro*
    - Na ordem alfabética, o item deve ser adicionado à direita de *março*
    - A árvore fica desbalanceada, já que o nó *maio* possui fator de balanceamento < -1
    - Como o sinal do nó desbalanceado é negativo, algum tipo de rotação à direita (R) deve ser feita
    - A raiz da sub-árvore direta do nó *maio* também possui fator de balanceamento negativo
    - A rotação que deve ser aplicada é a RR



- Exemplo
  - Inserção do item agosto
    - Como *a* é menor *m*, então agosto é inserido à esquerda do item maio
    - Após a inserção, a árvore ainda é mantida balanceada

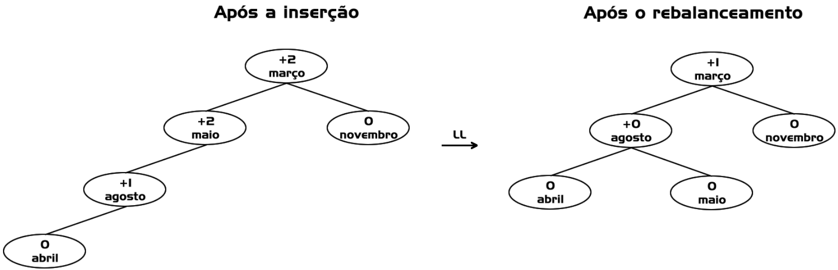
Após a inserção



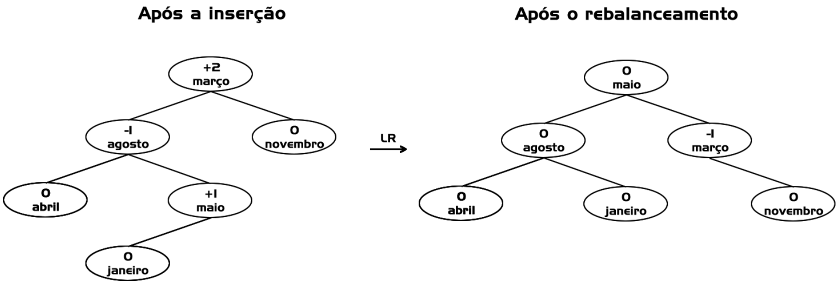
Após o rebalanceamento

Sem necessidade de rebalanceamento

- Exemplo
  - Inserção do item abril
    - Como *ab* é menor *ag*, então *abril* é inserido à esquerda do item *agosto*
    - Após a inserção, a árvore fica desbalanceada no nó *maio*, cujo fator de balanceamento é positivo, indicando que deve ser feita uma rotação à esquerda (L)
    - A raiz da sub-árvore esquerda do nó *maio* também possui fator de balanceamento positivo
    - A rotação que deve ser aplicada é a LL

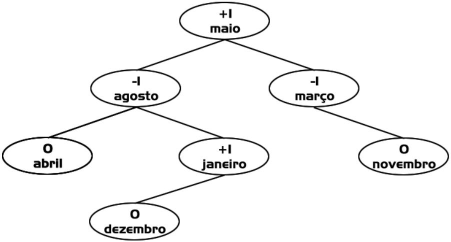


- Exemplo
  - Inserção do item *janeiro*
    - Como *j* é menor que *m* e maior que *a*, então *janeiro* de ser inserido ao lado esquerdo de *maio*
    - Com a inserção, árvore fica desbalanceada por causa do nó *março*, que possui fator de balanceamento $> +1$, ou seja, deve ser realizada alguma rotação à esquerda
    - A raiz subárvore esquerda de *março* possui fator de balanceamento negativo
    - A rotação que deve ser aplicada é a LR



- Exemplo
  - Inserção do item *dezembro*
    - Como *d* é maior que *a* e menor que *j*, então o novo item deve ser inserida ao lado esquerdo de *janeiro*
    - Após a inserção, a árvore é mantida balanceada

Após a inserção

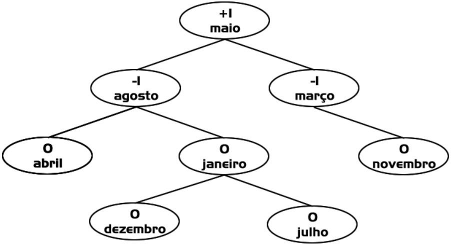


Após o rebalanceamento

Sem necessidade de
rebalanceamento

- Exemplo
  - Inserção do item *julho*
    - Como *ju* é maior que *ja*, então o novo item deve ser inserida ao lado direito de *janeiro*
    - Após a inserção, a árvore é mantida balanceada

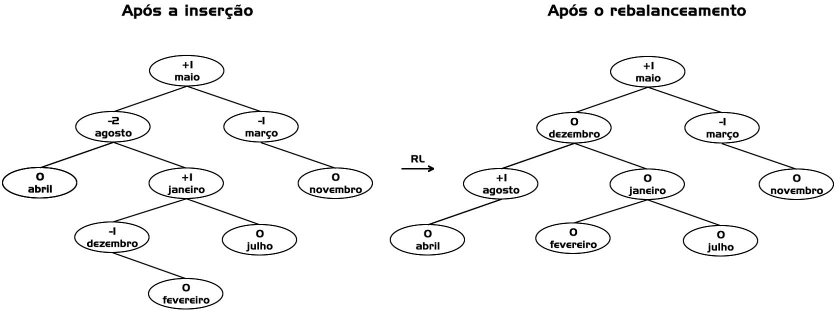
Após a inserção



Após o rebalanceamento

Sem necessidade de rebalanceamento

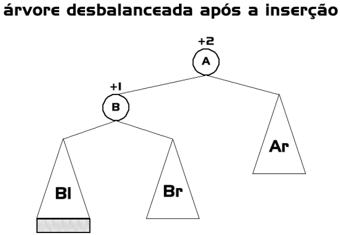
- Exemplo
  - Inserção do item *fevereiro*
    - Como *f* é menor que *j* e maior que *d*, então *fevereiro* de ser inserido ao lado esquerdo direito de *dezembro*
    - Com a inserção, árvore fica desbalanceada por causa do nó *agosto*, que possui fator de balanceamento < -1, ou seja, deve ser realizada alguma rotação à direita
    - A raiz subárvore direita de *agosto* possui fator de balanceamento positivo
    - A rotação que deve ser aplicada é a RL



- Estrutura de dados para a representação de uma árvore AVL:

```
typedef struct Pointer{
   int item;
   int bf;
   struct Pointer* right;
   struct Pointer* left;
}Node;
```

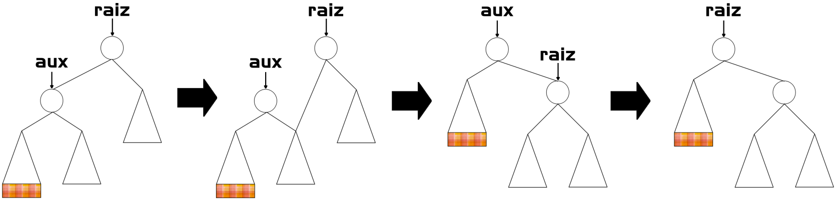
- Rotação LL



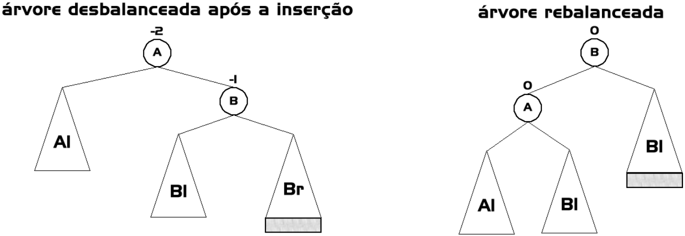
- Legenda:
  - Círculo: representa um nó
  - Triângulo: representa uma sub-árvore equilibrada
    - Nos exemplos ilustrados para as rotações **LL** e **RR**, cada sub-árvore possui a mesma altura
  - Retângulo: representa o aumento da altura de uma sub-árvore (inclusão de um novo nó)

- Rotação LL

```
Node *aux = raiz->left;
raiz->left = aux->right;
aux->right = raiz;
raiz = aux;
```



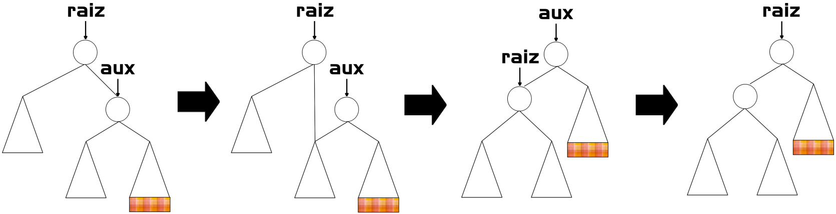
- Rotação RR



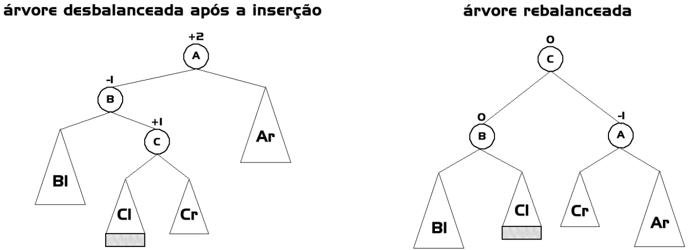
árvore desbalanceada após a inserção
-2
A
-1
B
Al
Bl
Br
árvore rebalanceada
0
B
0
A
Bl
Al
Bl

- Rotação RR

```
Node *aux = raiz->right;
raiz->right = aux->left;
aux->left = raiz;
raiz = aux;
```

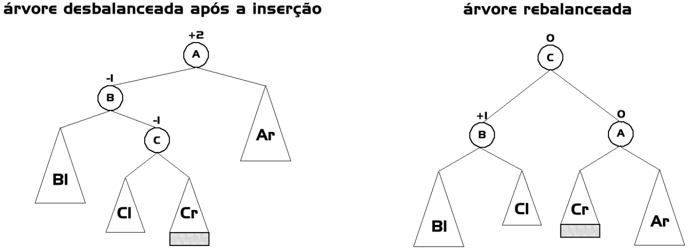


- Rotação LR: caso 1



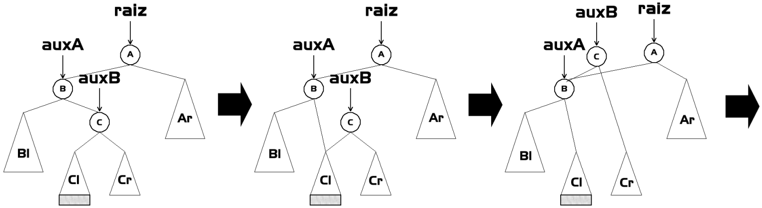
- Legenda:
  - Círculo e retângulo possuem o mesmo significado em relação aos exemplos de rotações LL e RR
  - Triângulos: representa uma sub-árvore equilibrada
    - A e B: pode-se dizer que possuem o mesmo significado em relação aos exemplos de rotações LL e RR
    - C: sub-árvore equilibrada, mas com altura brevemente menor (diferença de uma unidade) em relação às sub-árvores A e B
    - No exemplo acima, C é a raiz da sub-árvore Br

- Rotação LR: caso 2



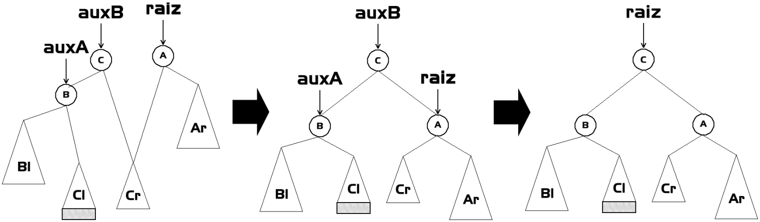
- Rotação LR

```
Node *auxA = raiz->left;
Node *auxB = auxA->right;
auxA->right = auxB->left;
auxB->left = auxA;
raiz->left = auxB->right;
auxB->right = raiz;
raiz = auxB;
```

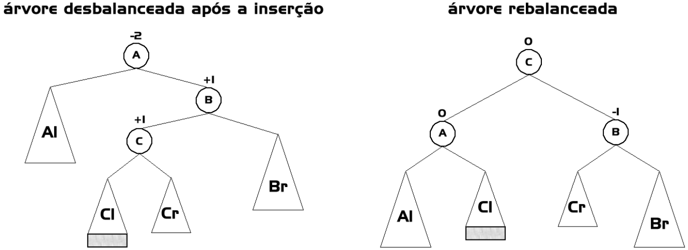


- Rotação LR

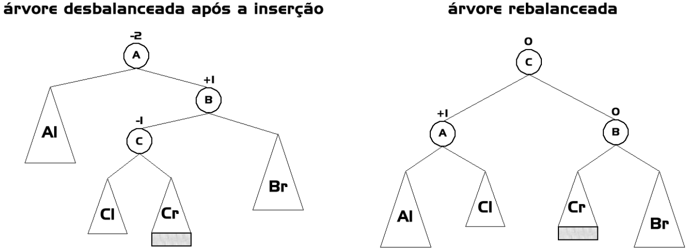
```
Node *auxA = raiz->left;
Node *auxB = auxA->right;
auxA->right = auxB->left;
auxB->left = auxA;
raiz->left = auxB->right;
auxB->right = raiz;
raiz = auxB;
```



- Rotação RL: caso 1

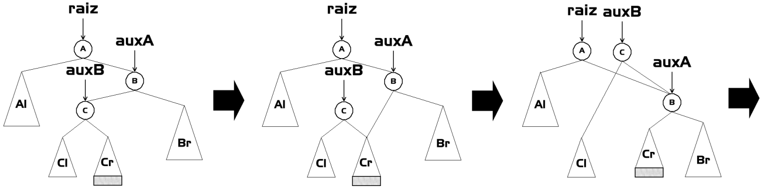


- Rotação RL: caso 2



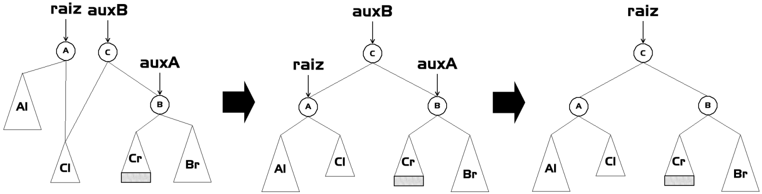
- Rotação RL

```
Node *auxA = raiz->right;
Node *auxB = auxA->left;
auxA->left = auxB->right;
auxB->right = auxA;
raiz->right = auxB->left;
auxB->left = raiz;
raiz = auxB;
```



- Rotação RL

```
Node *auxA = raiz->right;
Node *auxB = auxA->left;
auxA->left = auxB->right;
auxB->right = auxA;
raiz->right = auxB->left;
auxB->left = raiz;
raiz = auxB;
```



- A inserção e a remoção de itens em árvores AVL são realizadas da mesma forma que em árvores binárias de busca apresentadas na aula anterior
  - A diferença é que pode ser necessário rebalanceamento da árvore após essas operações

- A operação de busca, inserção e remoção tem a complexidade de tempo $O(\log n)$

```
Node rotateL(Node *raiz){
   Node *auxA = raiz->left, *auxB;

   if (auxA->fb == +1){ // Rotação LL
     raiz->left = auxA->right;
     auxA->right = raiz;
     raiz->fb = 0;
     raiz = auxA;
   }else{ Rotação RL, pois fb será negativo
     auxB = auxA->right;
     auxA->right = auxB->left;
     auxB->left = auxA;
     raiz->left = auxB->right;
     auxB->right = raiz;

     // Se a rotação LR foi para o caso 1
     if (auxB->fb == +1) raiz->fb = -1;
     else raiz->fb = 0;

     // Se a rotação LR foi para o caso 2
     if (auxB->fb == -1) auxA->fb = +1;
     else auxA->fb = 0;

     raiz = auxB;
   }
   return tree;
}
```

```
Node* rotateR(Node *raiz){
   Node *auxA = raiz->right, *auxB;

   if (auxA->fb == -1){ // Rotação RR
     raiz->right = auxA->left;
     auxA->left = raiz;
     raiz = auxA;
   }else{ // Rotação RL
     auxB = auxA->left;
     auxA->left = auxB->right;
     auxB->right = auxA;
     raiz->right = auxB->left;
     auxB->left = raiz;

     // Se a rotação RL foi para o caso 1
     if (auxB->fb == -1) raiz->fb = +1;
     else raiz->fb = 0;

     // Se a rotação RL foi para o caso 1
     if (auxB->fb == +1) auxA->fb = -1;
     else auxA->fb = 0;

     raiz = auxB;
   }
   return tree;
}
```

Árvores AVL

Operações: inserção

```
void inserir(Node* raiz, int value, int *grown){
   if(tree == NULL){
     tree = criar(value);
     *grown = 1;
   }else if (value < tree->item){
     tree->left = inserir(tree->left, value, grown);
     if (*grown){ // verificar se aumentou a árvore pelo lado esquerdo
       switch (tree->fb){
         case -1:  tree->fb = 0; *grown = 0; break;
         case 0:  tree->fb = +1; break
         case +1:  tree = rotateL(tree); tree->fb = 0; *grown = 0;
       }
     }
   }else if (value > tree->item){
     tree->right = inserir(tree->right, value, grown);
     if (*grown){
       switch (tree->fb){ // verificar se aumentou a árvore pelo lado direto
         case +1:  tree->fb = 0; *grown = 0; break;
         case 0:  tree->fb = -1; break;
         case -1:  tree = rotateR(tree); tree->fb = 0; *grown = 0;
       }
     }
   }
   return tree;
}
```

Baranauskas, J. A.
Árvores AVL – Algoritmos e Estruturas de Dados I.
Slides. Ciência da Computação FFCLRP-USP, Ribeirão Preto, 2013.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Introduction to Algorithms.
Third edition, The MIT Press, 2009.

Marin, L. O.
Árvores AVL. AE23CP – Algoritmos e Estrutura de Dados II.
Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2017.

Szwarcfiter, J.; Markenzon, L.
Estruturas de Dados e Seus Algoritmos.
LTC, 1994.

Referências II

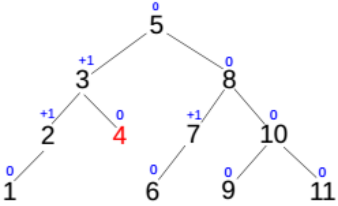
Tenenbaum, A.; Langsam, Y.
Estruturas de Dados usando C.
Pearson, 1995.

Ziviani, N.
Projeto de Algoritmos - com implementações em Java e C++.
Thomson, 2007.

Apêndice: remoção em árvores AVL

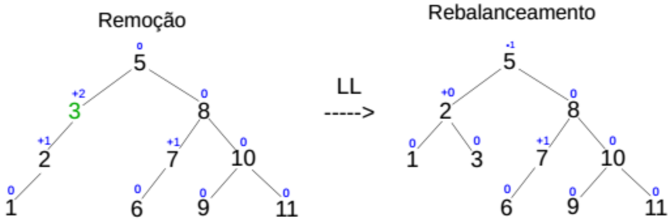
- Exemplo
  - Remoção do item 4

Remoção

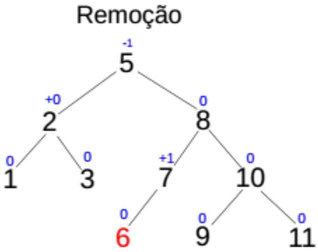


Rebalanceamento

- Exemplo
  - Remoção do item 4



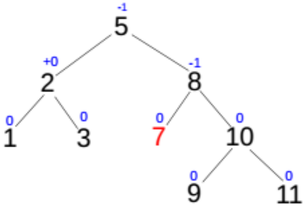
- Exemplo
  - Remoção do item 6



Rebalanceamento

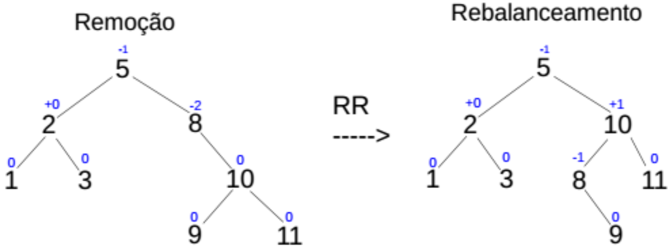
- Exemplo
  - Remoção do item 7

Remoção

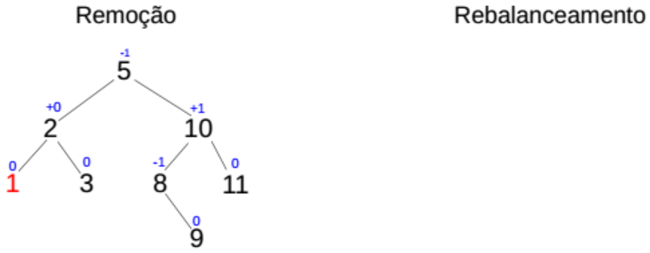


Rebalanceamento

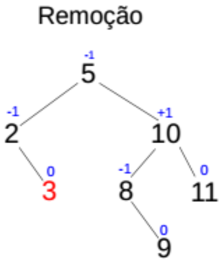
- Exemplo
    - Remoção do item 7



- Exemplo
  - Remoção do item 1



- Exemplo
  - Remoção do item 3



Rebalanceamento

- Exemplo
  - Remoção do item 3