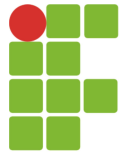
Árvores AVL

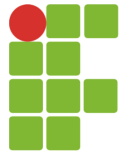
Estruturas de Dados e Algoritmos – Ciência da Computação

Prof. Daniel Saad Nogueira Nunes

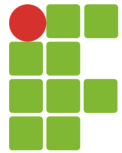
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

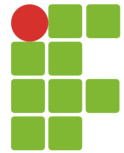


Sumário



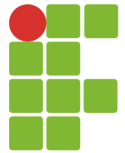
Árvores AVL

- As árvores AVL foram as primeiras BST balanceadas da literatura.
- Nomeada de acordo com os seus desenvolvedores: Georgy **A**delson-**V**elsky e Evgenii **L**andis.
- Possuem tempo de consulta/inserção/remoção limitado a $\Theta(\lg n)$.
- São estruturas que são auto-balanceáveis, tentando deixar as alturas dos nós de cada nível próximas.

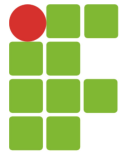


Árvore AVL

- Antes de expor os algoritmos que atuam sobre esta estrutura de dados, é necessário formalizar alguns conceitos.



Sumário

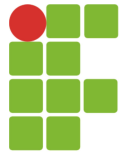


Árvore AVL

Definição ($h(x)$)

- Seja T uma árvore binária com raiz no nó x e seja uma função $ht(v)$ que nos dá a altura de um nó $v \in T$.
- A função $h(T)$ corresponde à seguinte definição:

$$h(T) = \begin{cases} 0, & \text{se } T \text{ é uma árvore vazia} \\ ht(x) + 1, & \text{caso contrário} \end{cases}$$



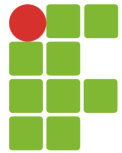
Árvore AVL

Definição (Fator de Equilíbrio)

Sejam T_1 e T_2 as subárvores da esquerda e da direita de uma árvore com raiz no nó x.

O fator de equilíbrio (*balance factor*) de x é dado como:

$$bf(x) := h(T_1) - h(T_2)$$



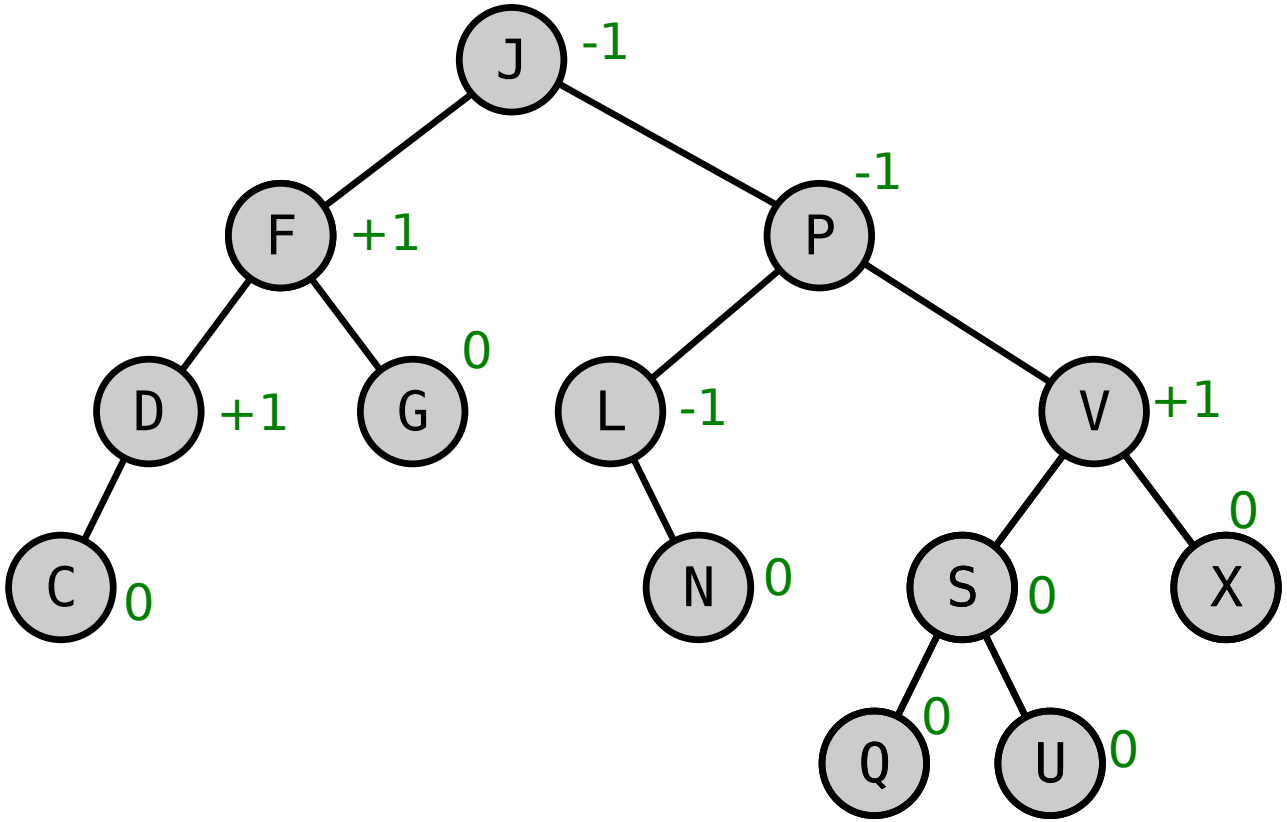
Árvore AVL

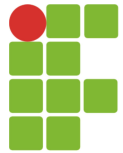
Definição (Árvore AVL)

Seja T uma árvore binária de pesquisa. T também é uma árvore AVL se e somente se:

- T é vazia; ou
- $-1 \leq bf(x) \leq 1, \forall x \in T$

Árvore AVL



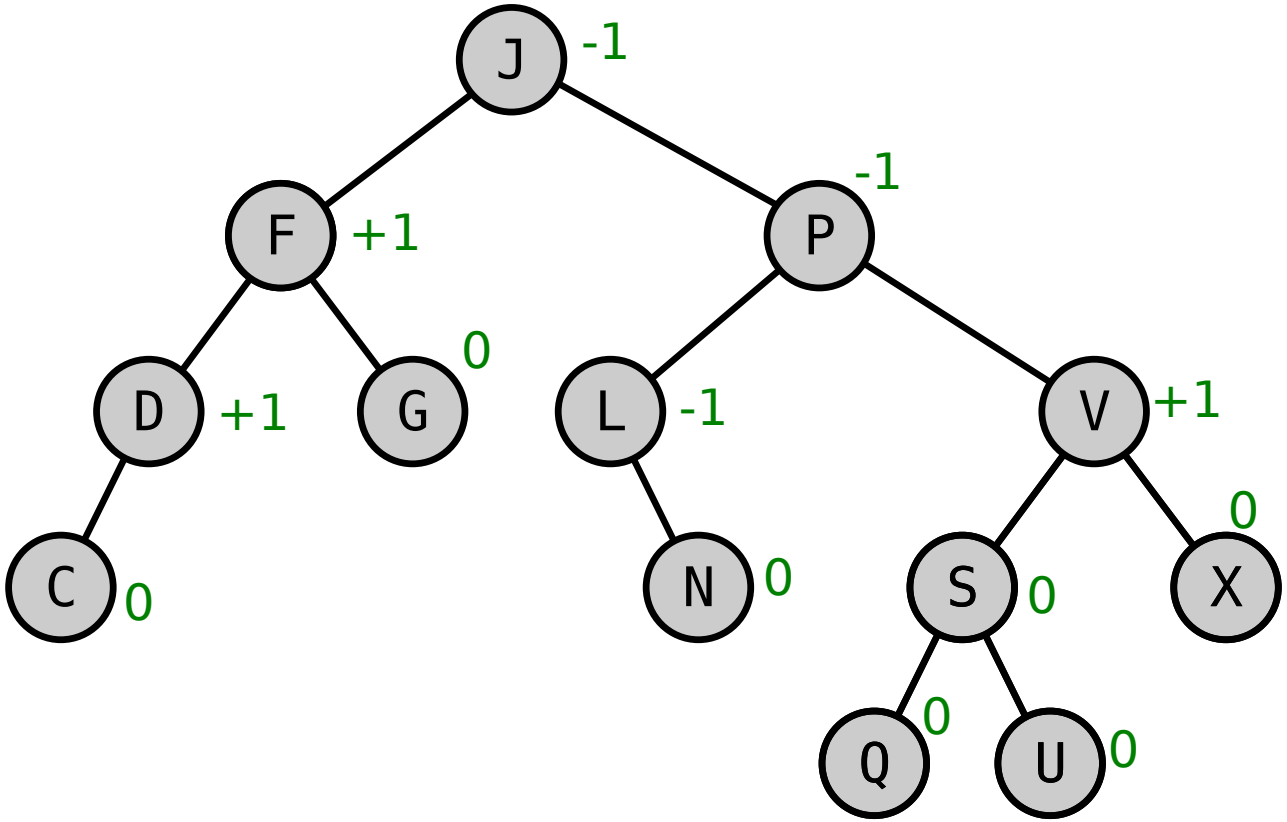


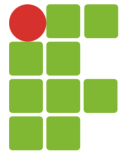
Árvore AVL

Fator de Equilíbrio

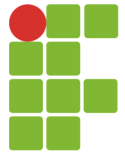
- Em árvores AVL, o fator de equilíbrio de cada nó está limitado a três possíveis valores: $\{-1, 0, 1\}$.
- Esta restrição que possibilita operações de inserção/remoção/busca serem efetuadas em tempo $\Theta(\lg n)$.

Árvore AVL



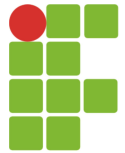


Sumário



Operações em Árvores AVL

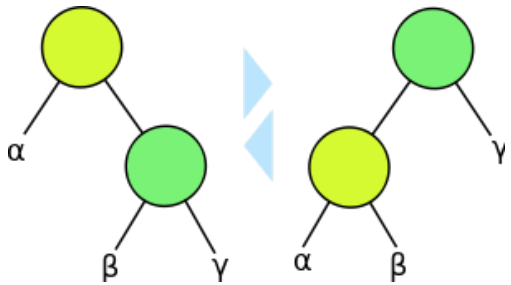
- Como árvores AVL são especializações de BST, os procedimentos de inserção, remoção e busca são bem parecidos.
- A diferença é que, após uma inserção e remoção, a árvore pode ficar desbalanceada, isto é, o fator de equilíbrio de algum nó está fora do intervalo $\{-1, 0, 1\}$.
- Assim, é necessário rebalancear a árvore.



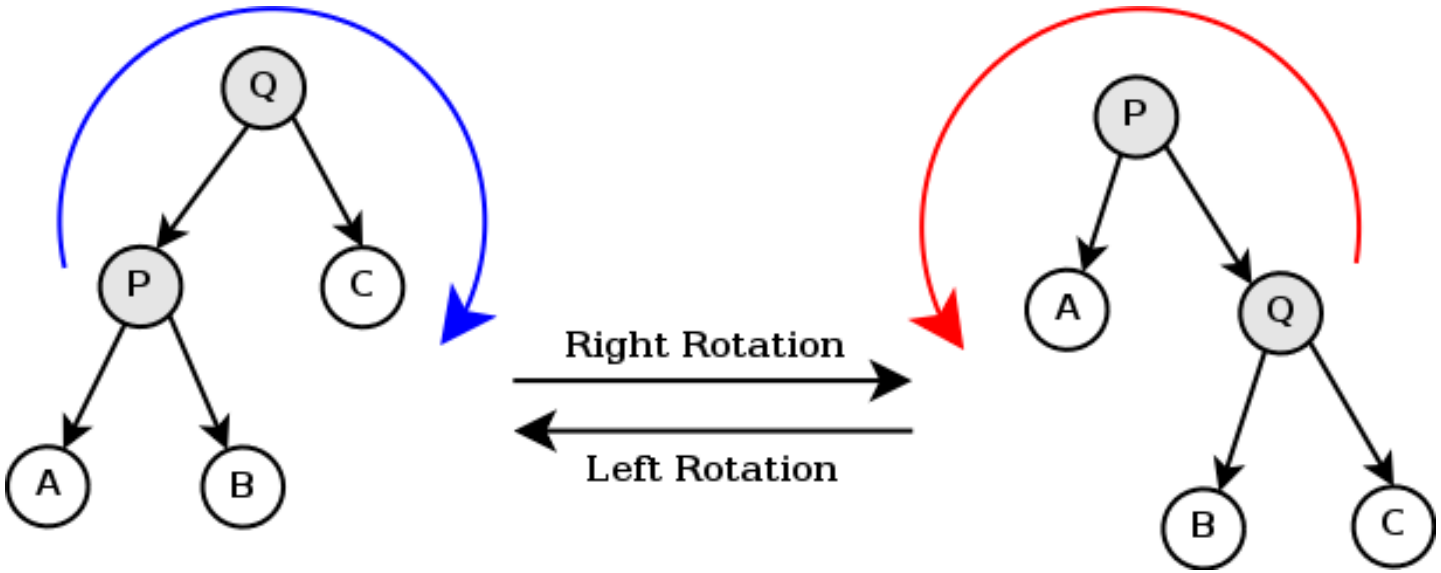
Balanceamento em Árvores AVL

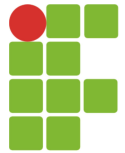
- O rebalanceamento é feito através de rotações, que podem ser:
  - Rotações para a esquerda.
  - Rotações para a direita.
- Uma rotação não interfere na propriedade de BST, isto é, a subárvore da esquerda continua os elementos com chaves menores do que a nova raiz e a subárvore da direita continua com os elementos com chaves maiores que a da nova raiz.

Rotações



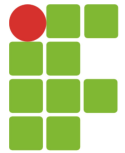
Rotações





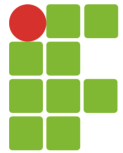
Árvores AVL

▸ Tree Rotation



Árvores AVL

- Ao final de cada inserção/remoção, a árvore deve ser atualizada.
- Para cada nó ao longo do caminho percorrido, deve-se computar:
  - A nova altura.
  - O novo fator de equilíbrio.
- Rotações para a esquerda/direita são feitas de movo a preservar os fatores de balanceamento no intervalo $\{-1, 0, 1\}$.
- Elas aumentam a altura de algumas subárvores enquanto diminuem a de outras.



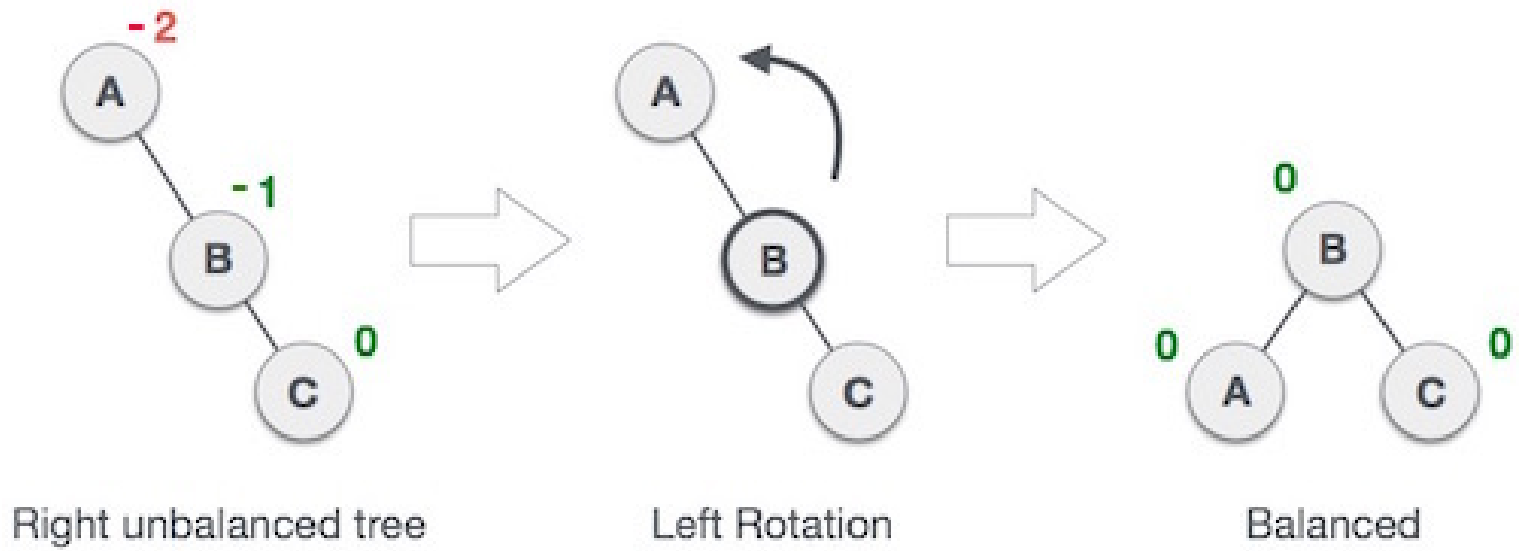
Balanceamento de Árvores AVL

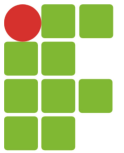
- O balanceamento das árvores AVL concentram-se em 4 casos:
  - Rotação para esquerda (L).
  - Rotação para direita (R).
  - Rotação para esquerda seguida de rotação para direita (LR).
  - Rotação para direita seguida de rotação para esquerda (RL).

Balanceamento de Árvores AVL

Caso 1 (L)

- Se a raiz da árvore tem fator de equilíbrio -2 e seu filho da direita possui fator de equilíbrio ≤ 0, uma rotação à esquerda é suficiente para balancear a árvore.

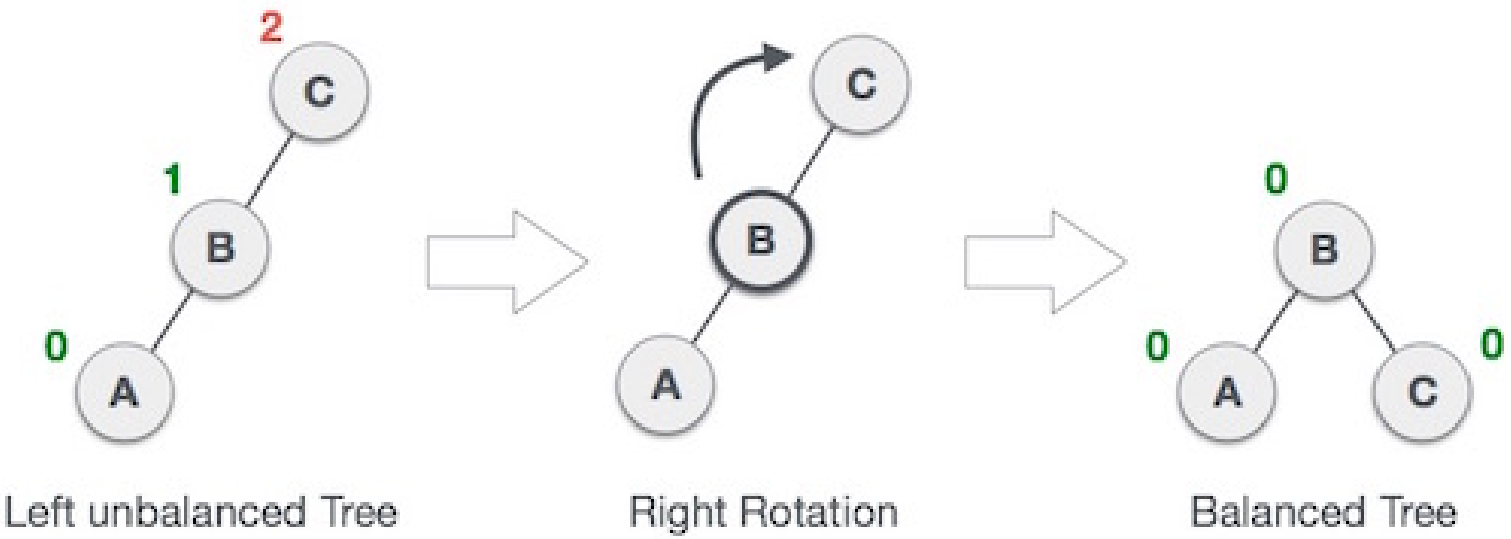


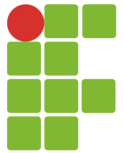


Balanceamento de Árvores AVL

Caso 2 (R)

- Se a raiz da árvore tem fator de equilíbrio 2 e seu filho da esquerda possui fator de equilíbrio ≥ 0, uma rotação à direita é suficiente para balancear a árvore.

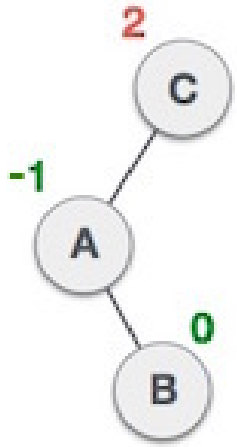


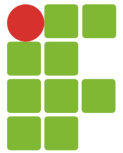


Balanceamento de Árvores AVL

Caso 3 (LR)

- Se a raiz da árvore tem fator de equilíbrio 2 e seu filho da esquerda possui fator de equilíbrio < 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.

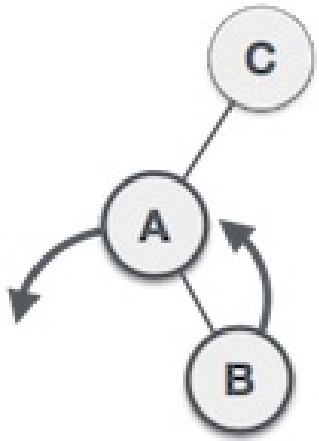


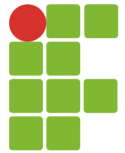


Balanceamento de Árvores AVL

Caso 3 (LR)

- Se a raiz da árvore tem fator de equilíbrio 2 e seu filho da esquerda possui fator de equilíbrio < 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.

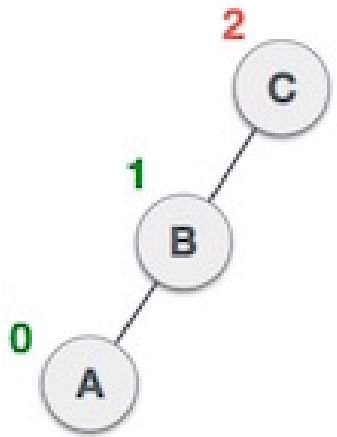


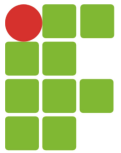


Balanceamento de Árvores AVL

Caso 3 (LR)

- Se a raiz da árvore tem fator de equilíbrio 2 e seu filho da esquerda possui fator de equilíbrio < 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.

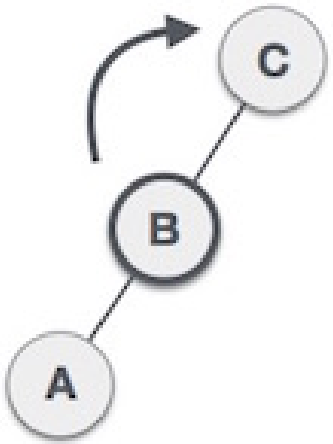


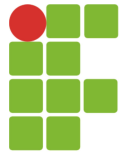


Balanceamento de Árvores AVL

Caso 3 (LR)

- Se a raiz da árvore tem fator de equilíbrio 2 e seu filho da esquerda possui fator de equilíbrio < 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.

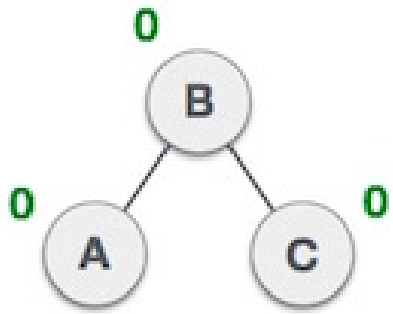


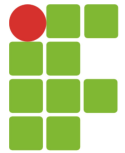


Balanceamento de Árvores AVL

Caso 3 (LR)

- Se a raiz da árvore tem fator de equilíbrio 2 e seu filho da esquerda possui fator de equilíbrio < 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.

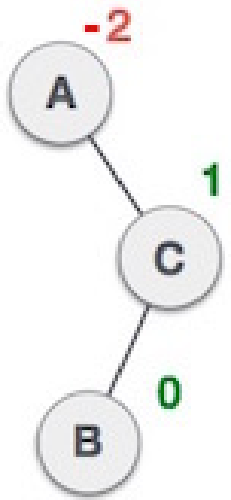


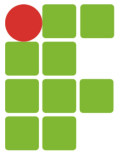


Balanceamento de Árvores AVL

Caso 4 (RL)

- Se raiz da árvore tem fator de equilíbrio -2 e seu filho da esquerda possui fator de equilíbrio > 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.

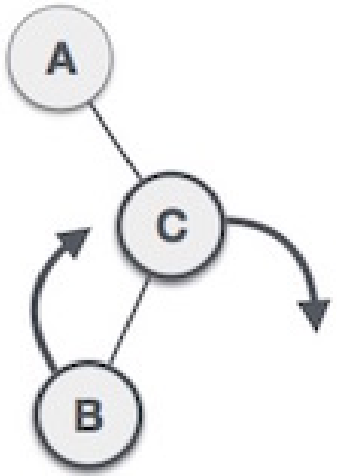


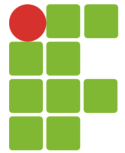


Balanceamento de Árvores AVL

Caso 4 (RL)

- Se raiz da árvore tem fator de equilíbrio -2 e seu filho da esquerda possui fator de equilíbrio > 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.

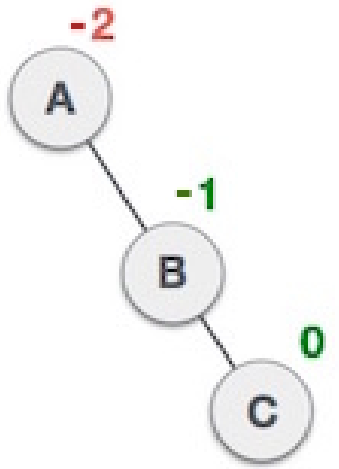


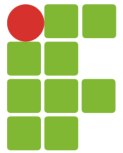


Balanceamento de Árvores AVL

Caso 4 (RL)

- Se raiz da árvore tem fator de equilíbrio -2 e seu filho da esquerda possui fator de equilíbrio > 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.

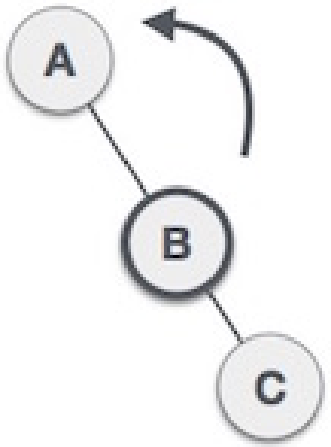


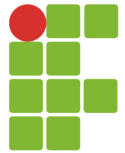


Balanceamento de Árvores AVL

Caso 4 (RL)

- Se raiz da árvore tem fator de equilíbrio -2 e seu filho da esquerda possui fator de equilíbrio > 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.

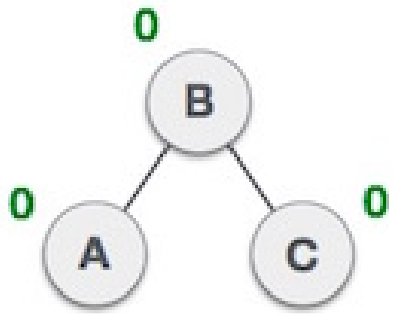


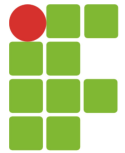


Balanceamento de Árvores AVL

Caso 4 (RL)

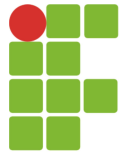
- Se raiz da árvore tem fator de equilíbrio -2 e seu filho da esquerda possui fator de equilíbrio > 0, uma rotação à esquerda seguida de uma a direita é suficiente para balancear a árvore.



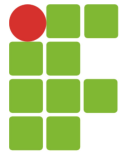


Balanceamento de Árvores AVL

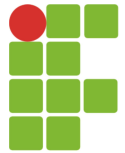
▸ AVL Applet



Sumário

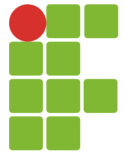


Sumário



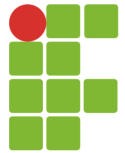
Definição

```
5  typedef void *(*avl_tree_element_constructor_fn)(void *);
6  typedef void (*avl_tree_element_destructor_fn)(void *);
7  typedef int (*avl_tree_element_compare_fn)(const void *, const void *);
```



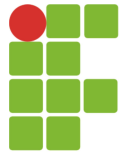
Definição

```
9  typedef struct avl_node_t {
10     void *data;
11     size_t height;
12     struct avl_node_t *left;
13     struct avl_node_t *right;
14 } avl_node_t;
```

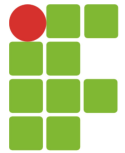


Definição

```
16  typedef struct avl_tree_t {
17      struct avl_node_t *root;
18      avl_tree_element_constructor_fn constructor;
19      avl_tree_element_destructor_fn destructor;
20      avl_tree_element_compare_fn comparator;
21      size_t size;
22  } avl_tree_t;
```

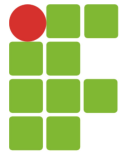


Sumário

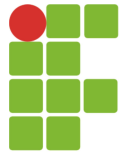


Inicialização

```
103 void avl_tree_initialize(avl_tree_t **t,
104                         avl_tree_element_constructor_fn constructor,
105                         avl_tree_element_destructor_fn destructor,
106                         avl_tree_element_compare_fn comparator) {
107     (*t) = mallocx(sizeof(avl_tree_t));
108     (*t)->root = NULL;
109     (*t)->size = 0;
110     (*t)->constructor = constructor;
111     (*t)->destructor = destructor;
112     (*t)->comparator = comparator;
113 }
```

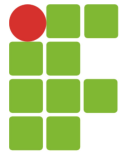


Sumário



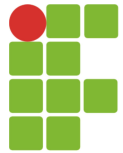
Funções auxiliares

```
252 size_t avl_tree_size(avl_tree_t *t) {
253     return t->size;
254 }
```



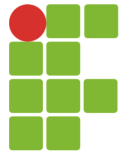
Funções auxiliares

```
167 static avl_node_t *avl_new_node(void *data,
168                                   avl_tree_element_constructor_fn constructor) {
169     avl_node_t *new_node = mallocx(sizeof(avl_node_t));
170     new_node->height = 1;
171     new_node->left = NULL;
172     new_node->right = NULL;
173     new_node->data = constructor(data);
174     return new_node;
175 }
```



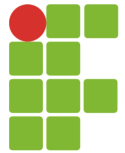
Funções auxiliares

```
129 static void avl_tree_delete_node(avl_node_t *t,
130                                    avl_tree_element_destructor_fn destructor) {
131     destructor(t->data);
132     free(t);
133 }
```



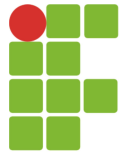
Funções auxiliares

```
89  static size_t avl_node_get_height(avl_node_t *v) {
90      if (v == NULL) {
91          return 0;
92      }
93      return v->height;
94  }
```



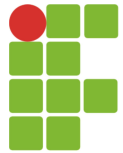
Funções auxiliares

```
41 static size_t avl_calculate_height(avl_node_t *v) {
42     size_t hl, hr;
43     if (v == NULL) {
44         return 0;
45     }
46     hl = avl_node_get_height(v->left);
47     hr = avl_node_get_height(v->right);
48     return hl > hr ? hl + 1 : hr + 1;
49 }
```



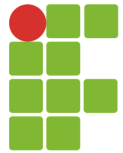
Funções auxiliares

```
96  static int avl_node_get_balance(avl_node_t *v) {
97      if (v == NULL) {
98          return 0;
99      }
100     return ((int)avl_node_get_height(v->left) - avl_node_get_height(v->right));
101 }
```



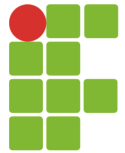
Funções auxiliares

```
59 static avl_node_t *avl_left_rotate(avl_node_t *x) {
60     assert(x != NULL);
61     avl_node_t *y = x->right;
62     assert(y != NULL);
63     x->right = y->left;
64     y->left = x;
65     x->height = avl_calculate_height(x);
66     y->height = avl_calculate_height(y);
67     return y;
68 }
```

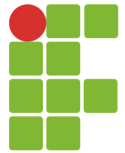


Funções auxiliares

```
78 static avl_node_t *avl_right_rotate(avl_node_t *y) {
79     assert(y != NULL);
80     avl_node_t *x = y->left;
81     assert(x != NULL);
82     y->left = x->right;
83     x->right = y;
84     y->height = avl_calculate_height(y);
85     x->height = avl_calculate_height(x);
86     return x;
87 }
```

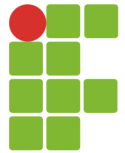


Sumário



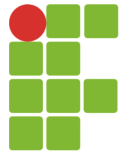
Busca

```
236 int avl_tree_find(avl_tree_t *t, void *data) {
237     return avl_tree_find_helper(t, t->root, data);
238 }
```

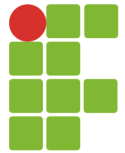


Busca

```
240 static int avl_tree_find_helper(avl_tree_t *t, avl_node_t *v, void *data) {
241     if (v == NULL) {
242         return 0;
243     }
244     if (t->comparator(data, v->data) < 0) {
245         return avl_tree_find_helper(t, v->left, data);
246     } else if (t->comparator(data, v->data) > 0) {
247         return avl_tree_find_helper(t, v->right, data);
248     }
249     return 1;
250 }
```

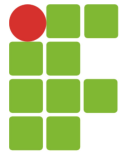


Sumário



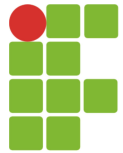
Inserção

```
135 void avl_tree_insert(avl_tree_t *t, void *data) {
136     t->root = avl_tree_insert_helper(t, t->root, data);
137 }
```



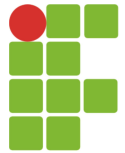
Inserção

```
139 avl_node_t *avl_tree_insert_helper(avl_tree_t *t, avl_node_t *v, void *data) {
140     if (v == NULL) {
141         v = avl_new_node(data, t->constructor);
142         t->size++;
143     } else if (t->comparator(data, v->data) < 0) {
144         v->left = avl_tree_insert_helper(t, v->left, data);
145     } else {
146         v->right = avl_tree_insert_helper(t, v->right, data);
147     }
148     v->height = avl_calculate_height(v);
149     int balance = avl_node_get_balance(v);
```

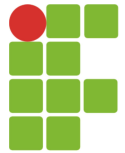


Inserção

```
150     if (balance > 1 && avl_node_get_balance(v->left) >= 0) {
151         return avl_right_rotate(v);
152     }
153     if (balance < -1 && avl_node_get_balance(v->right) <= 0) {
154         return avl_left_rotate(v);
155     }
156     if (balance > 1 && avl_node_get_balance(v->left) < 0) {
157         v->left = avl_left_rotate(v->left);
158         return avl_right_rotate(v);
159     }
160     if (balance < -1 && avl_node_get_balance(v->right) > 0) {
161         v->right = avl_right_rotate(v->right);
162         return avl_left_rotate(v);
163     }
164     return v;
165 }
```

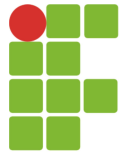


Sumário



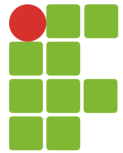
Remoção

```
177 void avl_tree_remove(avl_tree_t *t, void *data) {
178     t->root = avl_tree_remove_helper(t, t->root, data);
179 }
```



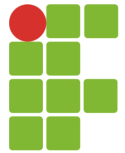
Remoção

```
181 avl_node_t *avl_tree_remove_helper(avl_tree_t *t, avl_node_t *v, void *data) {
182     if (v == NULL) {
183         return v;
184     } else if (t->comparator(data, v->data) < 0) {
185         v->left = avl_tree_remove_helper(t, v->left, data);
186     } else if (t->comparator(data, v->data) > 0) {
187         v->right = avl_tree_remove_helper(t, v->right, data);
188     } else { /*remoção do nó*/
189         /*caso 1 e caso 2, o nó é uma folha ou só tem um filho. Solução:
190          * transplantar*/
191         if (v->left == NULL) {
192             avl_node_t *tmp = v->right;
193             avl_tree_delete_node(v, t->destructor);
194             t->size--;
195             return tmp;
```



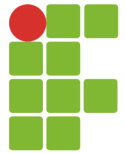
Remoção

```
196         } else if (v->right == NULL) {
197             avl_node_t *tmp = v->left;
198             avl_tree_delete_node(v, t->destructor);
199             t->size--;
200             return tmp;
201         }
```



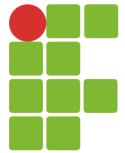
Remoção

```
202         /*caso 3, o nó tem dois filhos: achar o nó antecessor do que
203            queremos deletar. Obrigatoriamente este nó é uma folha
204            ou tem apenas um filho. Solução: colocar o valor da folha no
205            nó atual  e proceder a deletar a folha*/
206         else {
207             avl_node_t *tmp = avl_tree_find_rightmost(v->left);
208             void *swap = v->data;
209             v->data = tmp->data;
210             tmp->data = swap;
211             v->left = avl_tree_remove_helper(t, v->left, tmp->data);
212         }
213     }
```

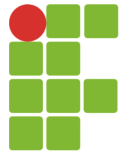


Remoção

```
214     if (v == NULL) {
215         return NULL;
216     }
217     v->height = avl_calculate_height(v);
218     int balance = avl_node_get_balance(v);
219     if (balance > 1 && avl_node_get_balance(v->left) >= 0) {
220         return avl_right_rotate(v);
221     }
222     if (balance < -1 && avl_node_get_balance(v->right) <= 0) {
223         return avl_left_rotate(v);
224     }
225     if (balance > 1 && avl_node_get_balance(v->left) < 0) {
226         v->left = avl_left_rotate(v->left);
227         return avl_right_rotate(v);
228     }
229     if (balance < -1 && avl_node_get_balance(v->right) > 0) {
230         v->right = avl_right_rotate(v->right);
231         return avl_left_rotate(v);
232     }
233     return v;
234 }
```

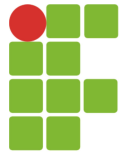


Sumário



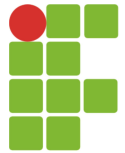
Limpeza

```
115 void avl_tree_delete(avl_tree_t **t) {
116     avl_tree_delete_helper((*t), (*t)->root);
117     free(*t);
118     (*t) = NULL;
119 }
```

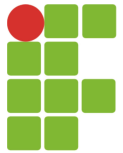


Limpeza

```
121 static void avl_tree_delete_helper(avl_tree_t *t, avl_node_t *v) {
122     if (v != NULL) {
123         avl_tree_delete_helper(t, v->left);
124         avl_tree_delete_helper(t, v->right);
125         avl_tree_delete_node(v, t->destructor);
126     }
127 }
```

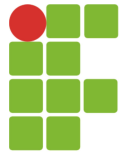


Sumário

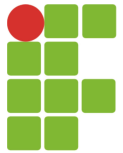


Análise

	Busca	Inserção	Remoção
BST	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
AVL	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$



Sumário



Considerações

- A árvore AVL aumenta a BST ao incluir uma estratégia de balanceamento por altura.
- Para alcançar esse objetivo, utiliza de rotações à esquerda e à direita, que conservam a propriedade de BST.
- Com esta estratégia, a altura máxima da árvore é $\Theta(\lg n)$.
- Implementá-la é um pouco mais complicado que implementar uma BST, mas o esforço vale a pena a longo prazo.