

# Árvores balanceadas

David Déharbe\*  
david@dimap.ufrn.br

18 de dezembro de 2003

## Sumário

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Árvores AVL</b>	<b>2</b>
2.1	Embasamento . . . . .	2
2.2	Inserção . . . . .	3
2.2.1	Rotação simples . . . . .	4
2.2.2	Rotação dupla . . . . .	6
2.2.3	Algoritmos . . . . .	7
2.3	Remoção . . . . .	9
2.3.1	Estudo . . . . .	9
2.3.2	Algoritmo . . . . .	10
<b>3</b>	<b>Árvores rubro-negra</b>	<b>13</b>
3.1	Embasamento . . . . .	13
3.2	Inserção . . . . .	16
3.2.1	Implementação da inserção . . . . .	18
3.3	Remoção . . . . .	21

## Lista de Figuras

1	Exemplos de árvore AVL e não AVL . . . . .	3
2	Inserções a esquerda resultando em árvores não AVL . . . . .	4
3	Inserções a direita resultando em árvores não AVL . . . . .	5
4	Remanejamento de (a) por rotação simples a direita . . . . .	5
5	Remanejamento de (c) por rotação simples a esquerda . . . . .	6
6	Remanejamento de (b) por rotação dupla a direita . . . . .	7
7	Declaração do tipo dos nós de uma árvore AVL . . . . .	8
8	Algoritmo de rotação simples a direita . . . . .	9
9	Algoritmo de rotação dupla a direita . . . . .	10
10	Algoritmo de inserção numa árvore AVL . . . . .	11

---

\*O uso, impressão, cópia, crítica, desse material é livre em instituições da rede pública de ensino, assim como em instituições com fins não lucrativas. Esse material não pode ser usado em instituições privadas com fins lucrativas sem a autorização do autor.

11	Exemplo de remoção numa árvore AVL . . . . .	12
12	Algoritmo de remanejamento a direita de uma árvore AVL . . . . .	13
13	Algoritmo para remover o mínimo de uma árvore AVL . . . . .	13
14	Algoritmo de remoção numa árvore AVL . . . . .	14
15	Exemplo de árvore rubro-negra . . . . .	15
16	Inversão de cores para reestabelecer condições depois de uma inserção . . . . .	16
17	Reequilíbrio por rotação e inversão de cores depois de uma inserção: subcaso 1 . . . . .	17
18	Reequilíbrio por rotação e inversão de cores depois de uma inserção: subcaso 2 . . . . .	17
19	Definição dos tipos para implementar árvores rubro-negras . . . . .	18
20	Definição funções auxiliares . . . . .	18
21	Função auxiliar de inserção . . . . .	19
22	A função principal de inserção . . . . .	20
23	Função auxiliar de balanceamento . . . . .	20
24	Remanejamento após remoção – caso (1) . . . . .	22
25	Remanejamento após remoção – caso (2) . . . . .	22
26	Remanejamento após remoção – caso (3) . . . . .	23
27	Remanejamento após remoção – caso (4) . . . . .	23

## 1 Introdução

Vimos, em nosso estudo sobre árvores binárias de busca, que é possível construir uma árvore binária de busca ótima quando o conjunto das chaves a ser inseridas é conhecido. No caso de uma ABB ótima, o tempo de acesso é logarítmico no número de nós.

Porém, em muitas aplicações, essa condição não é satisfeita: dados são inseridos e removidos dinamicamente ao decorrer da execução. No caso geral das ABBs, o resultado dos algoritmos de inserção e remoção não era ótimo. No pior caso, a ABB pode ficar igual a uma lista, com tempo de acesso linear.

Árvores balanceadas formam uma classe de árvores binárias de buscas que podem ser usadas em aplicações onde o conjunto de chaves não é conhecido de antemão, ou pode evoluir ao decorrer da utilização da ABB. Árvores balanceadas garantem, por construção, uma altura logarítmica no número de elementos armazenados.

Nesse capítulo estudaremos árvores balanceadas. Árvores balanceadas são classes de árvores tais que não existe desequilíbrio entre as subárvores esquerda e direita. Isso tem como consequência que a altura da árvore é logarítmica em função do número de nós, o que garante um desempenho satisfatório para as operações de busca. Algoritmos de inserção e remoção devem garantir que a árvore permanece balanceada, o que necessita geralmente remanejar a posição dos valores na árvore.

As árvores AVL e as árvores rubro-negra são dois tipos de árvores balanceadas estudadas nesse capítulo.

## 2 Árvores AVL

### 2.1 Embasamento

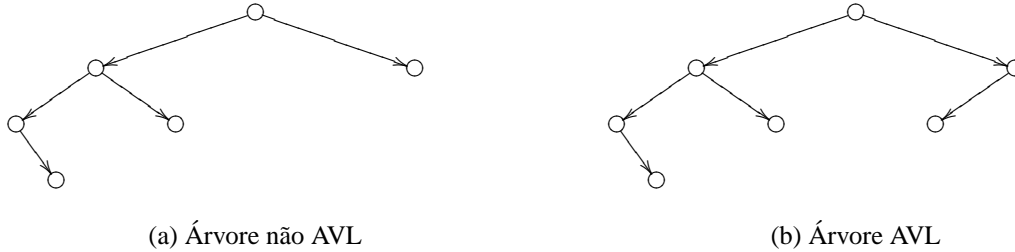
As árvores AVL, propostas por Adel'son-Vel'skiĭ e Landis em 1962, foram historicamente a primeira estrutura de dados a oferecer operações de inserção, remoção e busca em tempo logarítmico.

**Definição 1 (Árvore AVL, propriedade AVL)** *Uma árvore binária de busca  $A$  é uma árvore AVL, ou tem a propriedade AVL, quando, para qualquer nó  $n$  de  $A$ , as alturas de suas duas subárvores diferem no máximo de 1:*

$$|\alpha(A_n^e) - \alpha(A_n^d)| \leq 1.$$

Na figura 1 são desenhadas duas árvores binárias de buscas. A de esquerda não é AVL, pois a altura da subárvore direita da raiz é 1 e a da esquerda é 3.

**Figura 1** Exemplos de árvore AVL e não AVL.



**Proposição 1** A altura de uma árvore AVL  $A$  com  $i$  nós verifica a relação seguinte:

$$\alpha(A) < 1,4404 \ln(i + 2) - 0,328$$

**Prova:**

A prova dessa proposição consiste em calcular o número mínimo de nós de uma árvore AVL com uma altura dada. Seja  $\min(a)$  o número mínimo de vértices que possui uma árvore AVL de altura  $a$ .

$\min(0) = 0$ : a árvore vazia tem 0 vértice.

$\min(1) = 1$ : a árvore de altura 1 tem exatamente 1 vértice.

$\forall a > 1 \bullet \min(a) = \min(a-1) + \min(a-2)$ : a árvore de altura  $a$  com um número mínimo de vértices tem uma raiz, e duas subárvores, uma de altura  $a-1$ , com o número mínimo de nós, e uma de altura  $a-2$ , também com o número mínimo de nós.

O resto desta prova é baseado nas propriedades matemáticas da sequência  $\min$  e é deixado em exercício. ■

Portanto, a altura de uma árvore AVL é uma função logarítmica do número de nós e a operação de busca tem complexidade logarítmica.

A operação de busca numa árvore AVL é idêntica à de uma árvore binária de busca geral. No caso das operações de inserção e remoção, deve-se verificar, após a operação se a árvore permanece AVL, ou seja: que é uma árvore binária de busca<sup>1</sup> e que a diferença entre as alturas das subárvores esquerda e direita de cada nó é, no máximo, igual a 1.

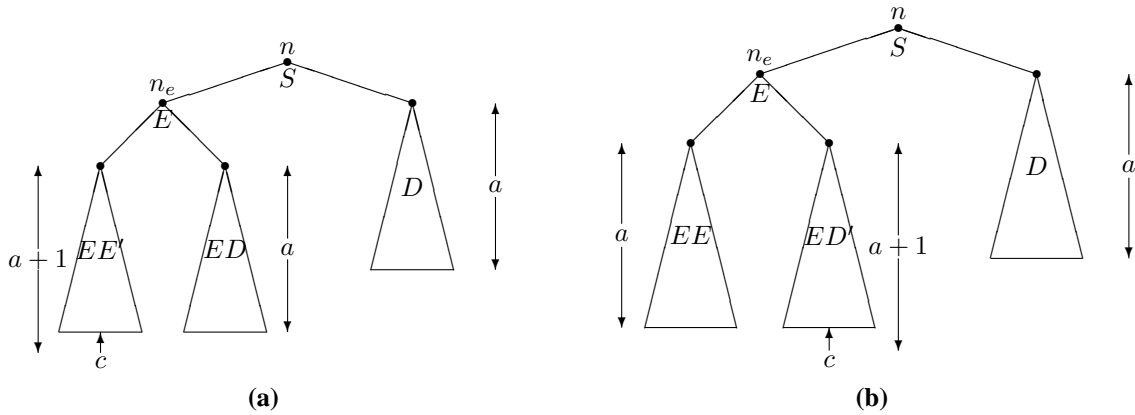
## 2.2 Inserção

Seja  $c$  uma nova chave a inserir numa árvore AVL  $A$ . Primeiro,  $c$  é inserido como folha de  $A$ , seguindo o algoritmo do caso geral das ABBs. Se depois da inserção a árvore permanece AVL, então nada mais tem a fazer. Caso contrário, é necessário remanejar a árvore.

<sup>1</sup>Para cada subárvore não vazia  $(n, A_n^e, A_n^d)$ :

$$\forall n' \bullet (n' \in A_n^e \Rightarrow r(n') < r(n)) \wedge (n' \in A_n^d \Rightarrow r(n') > r(n))$$

onde  $r$  é a função que retorna a chave associada a um nó.

**Figura 2** Inserções a esquerda resultando em árvores não AVL

Para que o resultado de uma inserção não tenha a propriedade AVL, é preciso ter pelo menos um nó da árvore inicial tal que uma subárvore tem altura estritamente superior à da outra, e que a inserção seja realizada nesta subárvore. Seja  $n$  o nó de maior nível que enraiza uma subárvore  $S$  que não tem propriedade AVL depois da inserção. Em consequência, essa árvore  $S$  tem uma altura estritamente superior a 0, e não é vazia. Supondo que  $E$  é a subárvore esquerda de  $S$  que tem altura maior que  $D$ , a subárvore direita de  $S$ . Seja  $a$  a altura de  $D$ , logo  $a + 1$  é a altura de  $E$ . Seja  $n_e$  a raiz de  $E$ ,  $EE$  e  $ED$  são as subárvores a esquerda e a direita de  $n_e$ . Necessariamente,  $a$  é a altura da mais alta das subárvores de  $n_e$ . Há dois casos possíveis (figura 2):

- (a) A inserção ocorre em  $EE$ . Como, por hipótese o resultado da inserção não é AVL, as alturas de  $EE$  antes e depois da inserção devem ser  $a$  e  $a + 1$ . Qual é a altura de  $ED$ ? Como a árvore era inicialmente balanceada, a altura de  $EE$  é  $a$  ou  $a - 1$ . Mas não pode ser  $a - 1$ , pois, neste caso,  $n_e$  é um nó que enraiza uma subárvore que não tem a propriedade AVL e seu nível é superior ao de  $n$ , o que contradiz a hipótese. Logo a altura de  $EE$  é  $a$ .

O novo nó é inserido na subárvore esquerda de  $n_e$ . Depois desta inserção, a altura da subárvore esquerda de  $n$  é  $a + 2$  e a altura da subárvore direita é  $a$ . Para corrigir esse desequilíbrio, a árvore é remanejada usando uma transformação chamada de rotação simples a direita (apresentada na seção 2.2.1).

- (b) A inserção ocorre em  $ED$ . Como, por hipótese o resultado da inserção não é AVL, as alturas de  $ED$  antes e depois da inserção devem ser  $a$  e  $a + 1$ . Para as mesmas razões que as citadas no item anterior a altura de  $EE$  é  $a$ .

Como o novo nó é inserido em  $ED$ , a altura de  $E$  torna-se igual a  $a + 2$ , enquanto a altura de  $D$  permaneceu igual a  $a$ . O equilíbrio é reestabelecido usando uma transformação chamada de rotação dupla a direita (apresentada na seção 2.2.2).

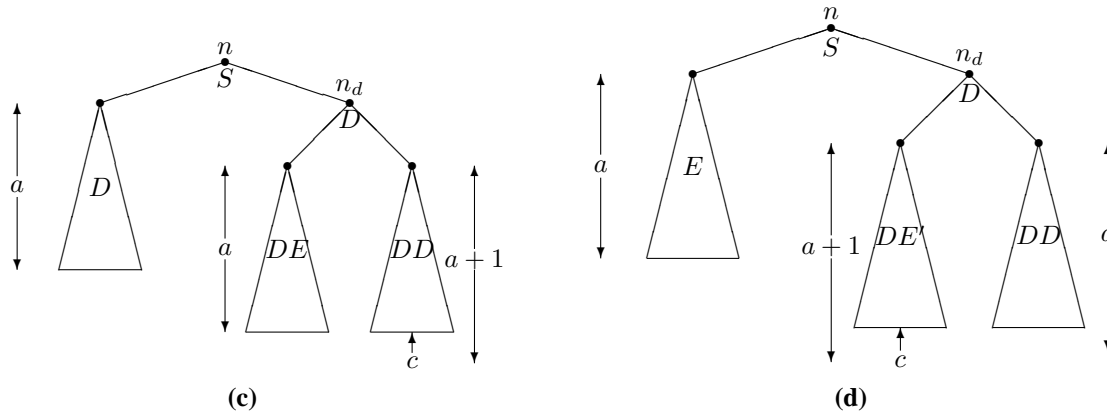
Além das inserções a esquerda ilustradas na figura 2, há o caso simétrico onde é a subárvore direita da raiz que está inicialmente mais alta que a subárvore esquerda. Essas transformações são ilustradas na figura 3:

- (c) a inserção a direita, simétrica da inserção (a).  
 (d) a inserção a direita, simétrica da inserção (b).

### 2.2.1 Rotação simples

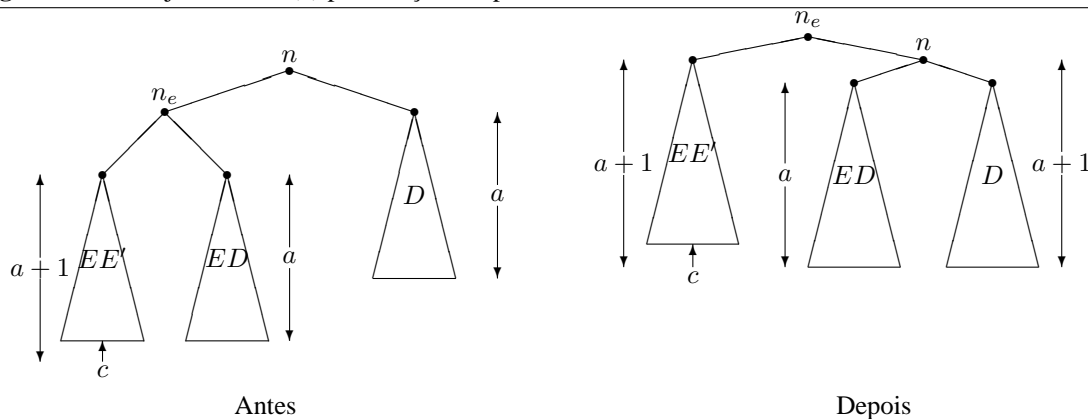
Para discutir a rotação simples, trataremos do caso particular da rotação simples a direita. A rotação simples a esquerda é a operação simétrica.

A operação de rotação simples a direita segue os passos seguintes

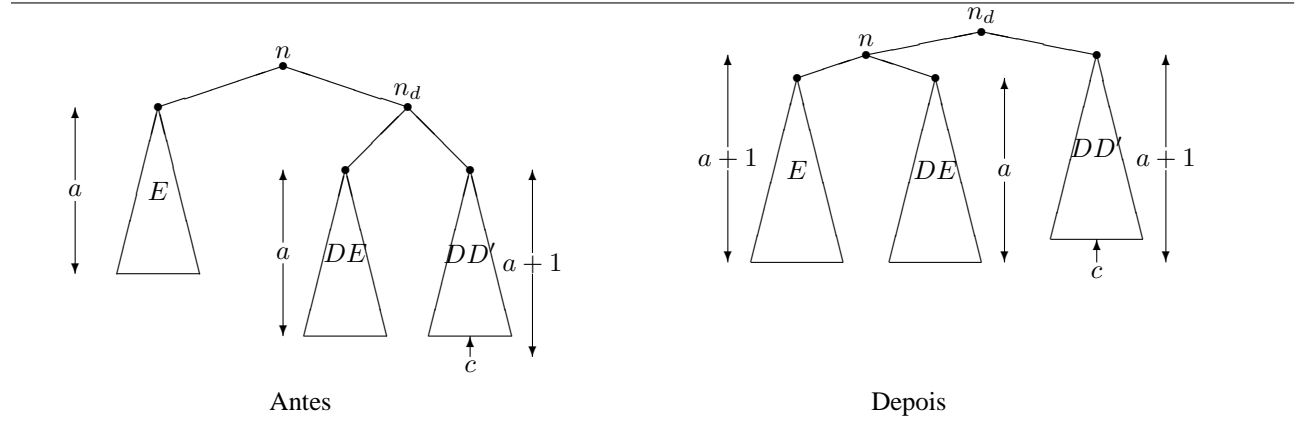
**Figura 3** Inserções a direita resultando em árvores não AVL

1.  $n_e$  é colocado na raiz,
2.  $EE$  permanece a subárvore esquerda de  $n_e$ ;
3.  $n$  torna-se a raiz da subárvore direita de  $n_e$ ;
4.  $ED$  é a nova subárvore esquerda de  $n$ ;
5.  $D$  permanece a subárvore direita de  $n$ .

O resultado dessas transformações é ilustrado na figura 4. O leitor é incentivado a verificar que o resultado é bem uma árvore binária de busca (ou seja, tem que mostrar que, para cada nó  $n$ , o rótulo de  $n$  é superior (inferior) aos rótulos dos nós na subárvore esquerda (direita)). É também uma árvore AVL, pois as subárvores de  $n_e$  e  $n$  têm a mesma altura, e  $EE$ ,  $ED$  e  $D$  são, por hipótese árvores AVL. Portanto todas as subárvores têm a propriedade AVL.

**Figura 4** Remanejamento de (a) por rotação simples a direita

A operação simétrica é a rotação simples a esquerda, ilustrada na figura 5 e corresponde ao remanejamento no caso (c) da figura 3. Note que, em todos os casos, depois de inserir e remanejar, a altura da subárvore volta a seu valor inicial ( $a+2$ ). Como, antes da inserção, quando  $S$  tinha altura  $a+2$ ,  $A$  tinha a propriedade AVL, depois da inserção,  $A$  permanece AVL.

**Figura 5** Remanejamento de (c) por rotação simples a esquerda

### 2.2.2 Rotação dupla

Em alguns casos, uma rotação simples a direita não é adequada. Basicamente esses casos correspondem à inserções a direita de um filho esquerdo de um nó cujo balanço é -1, ou a esquerda de um filho direito de um nó cujo balanço é 1.

No caso (b) da figura 2, uma rotação simples a direita não funciona e uma operação, chamada de rotação dupla a direita, é necessária. Seja  $ED'$  o resultado da inserção em  $ED$ . Observe que  $ED'$  tem altura  $a + 1$ , onde  $a \geq 0$ , e possui pelo menos um nó. Seja  $n_{ed}$  a raiz de  $ED'$ , e  $EDE'$ ,  $EDD'$  suas subárvores esquerda e direita.

**Lema 1** Antes da inserção (b), ou  $ED$  era vazia, ou as alturas das suas subárvores  $EDE$  e  $EDD$  verificavam a relação seguinte:

$$\alpha(EDE) = \alpha(EDD) = a - 1.$$

**Prova:**

No caso onde  $ED$  não era vazia antes da inserção, por hipótese,  $\alpha(ED) = a$  e depois da inserção,  $\alpha(ED') = a + 1$ , e  $n$  é a raiz da subárvore de maior nível que perde a propriedade AVL com a inserção de  $c$ .

Consequentemente, antes da inserção, pelo menos um de  $EDE$  e  $EDD$  tem altura  $a - 1$  e o outro pode ter altura  $a - 2$  ou  $a - 1$ . Suponha que  $\alpha(EDD) = a - 1$  e  $\alpha(EDE) = a - 2$  (o argumento para o caso simétrico é o mesmo). Então:

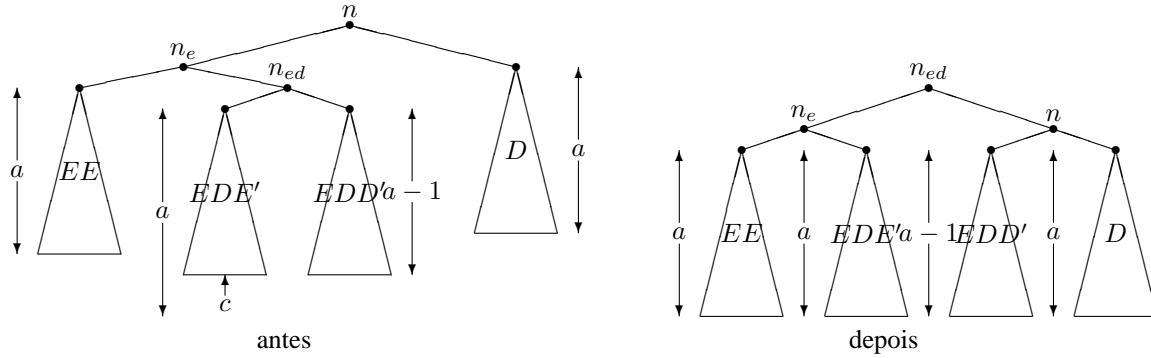
- se a inserção for em  $EDD$ , depois da inserção, temos  $\alpha(EDD') = a$ ,  $\alpha(EDE') = a - 2$  e a subárvore  $ED'$  enraizada em  $n_{ed}$  não é AVL. Como  $\nu(n_{ed}) > \nu(n)$  temos uma contradição.
- se a inserção for em  $EDE$ , depois da inserção, temos  $\alpha(EDE') = a - 1$ ,  $\alpha(EDD') = a - 1$  e  $\alpha(ED') = a$ , o que é uma contradição com a hipótese inicial que  $\alpha(ED') = a + 1$ .

Em conclusão, a suposição  $\alpha(EDD) = a - 1$  e  $\alpha(EDE) = a - 2$  leva a uma contradição. Por simetria se  $\alpha(EDD) = a - 2$  e  $\alpha(EDE) = a - 1$ , também leva a uma contradição. A única solução possível então é  $\alpha(EDD) = \alpha(EDE) = a - 1$ .

Na figura 6 (parte antes), desenhamos a inserção da chave  $c$  na subárvore esquerda de  $n_{ed}$ . O caso onde a chave é inserida na subárvore direita é tratado da mesma maneira e não é apresentado. O caso onde a subárvore  $ED$  é vazia também não é representado.

O remanejamento por rotação dupla a direita é composto das etapas seguintes:

Figura 6 Remanejamento de (b) por rotação dupla a direita



1. O nó  $n_{ed}$  é a nova raiz,  $n_e$  é seu filho esquerdo, e  $n$  seu filho direito.
2. As subárvores esquerda e direita de  $n_e$  são respectivamente  $EE$  e  $EDE$ .
3. As subárvores esquerda e direita de  $n$  são respectivamente  $EDD$  e  $D$ .

Deixamos ao leitor a tarefa de verificar que o resultado da rotação dupla a direita é uma árvore binária de busca.

Por hipótese  $EE$ ,  $EDE$ ,  $EDD$  e  $D$  têm a propriedade AVL e são de altura  $a$ . Em consequência, as subárvores enraizadas em  $n_e$  e  $n$  tem a propriedade AVL e altura  $a + 1$ . Em conclusão a árvore enraizada em  $n_{ed}$  tem a propriedade AVL. Repare que, depois de inserir e remanejar, a altura da subárvore  $S$  de novo volta a seu valor inicial ( $a + 2$ ), portanto a árvore  $A$  permanece uma árvore AVL.

O algoritmo implementando essas operações é baseado na observação que uma rotação dupla é equivalente a duas rotações simples consecutivas (uma a esquerda, enraizada em  $n_e$ , seguida de uma a direita, enraizada em  $n$ ).

### 2.2.3 Algoritmos

Seja  $A$  uma árvore AVL, com raiz  $r$  e  $i$  nós, e  $c$  uma chave a inserir. O procedimento a seguir é o seguinte:

1. Uma busca é efetuada para verificar se  $c$  já está presente. Caso for, o procedimento para. Caso contrário, a busca retorna o nó onde deve ser inserido  $c$ , e o tratamento procede com as etapas seguintes. Para esse passo de busca, podemos usar o algoritmo Busca para árvores binárias de busca.
2. Um novo nó  $n$  é criado e inserido, usando o procedimento usual.
3. Deve-se verificar se a inserção do nó desequilibrou alguma subárvore. Caso negativo, o procedimento para. Caso positivo, é preciso reequilibrar a árvore usando a operação de rotação apropriada.

Estudamos agora como verificar se a inserção introduz um desequilíbrio na árvore. Uma primeira observação é que se há uma subárvore não balanceada, ela necessariamente tem como raiz um nó que está no caminho da raiz  $r$  até o nó criado  $n$ . Lembre que há uma quantidade logarítmica em  $i$  de nós nesse caminho. Uma segunda observação é que, como uma operação seguida de um remanejamento não altera a altura da árvore, um único remanejamento é necessário.

Um procedimento possível é então, para cada nó ancestral de  $n$ , determinar a altura das subárvores e, se há desequilíbrio, aplicar a rotação apropriada. Porém, o cálculo da altura de uma subárvore é uma função linear do número de nós. Portanto o custo desse passo de inserção é uma função do tipo  $O(i \cdot \ln i)$ , que é superior ao custo logarítmico desejado.

Uma primeira solução possível é, em cada nó, armazenar num campo, chamado `alt`, a altura da subárvore que ele enraiza. Quando  $n$  é inserido, basta incrementar esse valor e verificar se há necessidade de efetuar uma rotação em cada nó no caminho de  $r$  até  $n$ . Também, não deve se esquecer de reatualizar o valor de `alt` depois do remanejamento.

Uma segunda solução é, para cada nó  $m$ , no lugar do campo `tamanho`, ter um campo `bal` =  $\alpha(A_m^d) - \alpha(A_m^e)$ , que indica o balanço entre as duas subárvores, e no caso das árvores AVL, pode ser igual a:

- $-1$  quando a subárvore direita tem um nó a menos que a subárvore esquerda,
- $0$  quando as duas subárvores tem o mesmo número de nós,
- $1$  quando a subárvore direita tem um nó a mais que a subárvore esquerda.

**Figura 7** Declaração do tipo dos nós de uma árvore AVL

```

1  /* AVL é o tipo concreto C das árvores AVL */
2  typedef struct nóAVL * AVL;
3  /* nóAVL é a estrutura dos nós de uma árvore AVL */
4  struct nóAVL {
5      t chave;
6      AVL esq; /* sub-árvore esquerda */
7      AVL dir; /* sub-árvore direita */
8      int bal; /* -1, 0 ou 1: é alt(dir) - alt(esq) */
9  };

```

Quando um nó é inserido a esquerda de um nó  $m$ , e essa inserção aumenta a altura da subárvore esquerda, o balanço é decrementado. Caso se torne igual a  $-2$ ,  $m$  é desequilibrado e uma rotação a direita é necessária. Simetricamente, quando um nó é inserido a direita de  $m$ , e a altura da subárvore direita aumenta, o balanço é incrementado. Se esse balanço se torna igual a  $2$ , então  $m$  é desequilibrado e uma rotação a esquerda é necessária.

Seja  $p$  o nó pai do novo nó  $n$ .

O tratamento da inserção a esquerda é apresentado a seguir (o tratamento para as inserções a direita é simétrico e não é apresentado):

- Se o balanço de  $p = 1$  antes da inserção, o balanço se torna igual a  $0$ . A altura da subárvore enraizada em  $p$  não foi alterada assim como também as alturas das subárvores de  $p$  até  $r$ .
- Se o balanço de  $p = 0$  antes da inserção, o balanço se torna igual a  $-1$  e a altura da subárvore enraizada em  $p$  foi incrementada pela inserção. É necessário continuar a examinar os ancestrais de  $p$ . Observe que se  $p = r$ , então o procedimento para.
- Se o balanço de  $p = -1$  antes da inserção, o balanço se torna igual a  $-2$  e uma rotação é necessária. Depois da rotação a altura da subárvore volta a seu valor inicial. Não é necessário examinar os ancestrais e o procedimento para.

Seguem os algoritmos de rotação simples e dupla a direita. Esses algoritmos realizam também a atualização dos campos `bal` dos nós remanejados. Note que o parâmetro `raiz` é passado por referência e seu valor modificado pelos algoritmos. Os algoritmos de rotação a esquerda são simétricos e deixados como exercícios ao leitor.

Finalmente, o algoritmo de inserção é dado a seguir. Esse algoritmo tem como parâmetros o valor a inserir e um ponteiro para a raiz da árvore corrente. O algoritmo retorna um valor booleano que indica se houve um aumento da altura da árvore. Também, o parâmetro `raiz`, que é passado por referência, pode ser alterado pelo algoritmo.

Inicialmente é testado se foi alcançado uma folha, caso onde a criação de um novo nó deve se operar. A raiz da árvore é o nó criado, e a altura passou de  $0$  até  $1$ , portanto o resultado é verdadeiro.

Em seguida é testado se o valor já está armazenado, caso nada seja feito, como a altura não é modificada, o resultado é falso.

Finalmente, quando o valor é diferente do valor armazenado na raiz, a inserção é realizada na subárvore correspondente. Quando a inserção na subárvore altera a altura, o valor do balanço da subárvore é testado para avaliar se a



**Figura 8** Algoritmo de rotação simples a direita

---

```

1 // ***** Definição de função *****
2 // Nome: RotaçãoSimplesADireita
3 // Parâmetros: AVL & raiz
4 // Retorno: void
5 // Descrição:
6 // Efetua um remanejamento da árvore referenciada por "ptraiiz" seguindo o padrão de rotação simples
7 // a direita. O parâmetro é passado por referência e aponta para o resultado depois da chamada.
8 // Pré-condição: Antes da chamada ptraiiz deve ser o endereço de uma árvore binária de busca
9 // com a forma da figura 2 (a).
10 void RotaçãoSimplesADireita(AVL & raiz)
11 {
12     AVL ne = raiz->esq;
13     raiz->esq = ne->dir;
14     ne->dir = raiz;
15     ne->bal = raiz->bal = 0;
16     raiz = ne;
17 }
```

---

necessidade de remanejar. Isso é o caso quando o balanço é 1 e a altura da subárvore direita aumentou, ou quando o balanço é 0 e a altura da subárvore esquerda aumentou. Note que quando há uma inserção seguida de um remanejamento, a altura não é modificada.

Observe finalmente que quando o balanço é inicialmente 0 e a inserção numa subárvore aumenta a altura dessa subárvore, não precisa fazer remanejamento. Porém, deve-se indicar que a altura da árvore aumentou, retornando o valor verdadeiro.

### Exercícios

1. Inserir, em sequência, os valores 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 numa árvore AVL inicialmente vazia. Desenhar, para cada inserção, a árvore inicial e final, e a árvore antes e depois da cada operação de rotação.
2. Caracterizar a família de árvores AVL tais que a inserção de qualquer nó provoca um desequilíbrio.

## 2.3 Remoção

### 2.3.1 Estudo

A operação de remoção numa árvore AVL também pode ser feita num número logarítmico de passos. O passo inicial da remoção consiste em seguir o procedimento aplicado no caso geral de uma ABB. É sempre uma folha que é eliminada da árvore (mesmo quando o valor removido está num nó interior, neste caso o conteúdo de uma folha é transferida nesse nó e a folha é removida). Portanto, a altura das subárvores que contém está folha pode ser alterada, e pode ser necessário reequilibrar a árvore, com operações de rotação.

Porém ao contrário da inserção, pode ser preciso fazer mais de uma rotação. A razão é a seguinte: quando uma inserção deve ser seguida de uma rotação, a altura da subárvore remanejada não é alterada. Essa propriedade não é verificada pela operação de remoção.

A remoção de uma chave  $c$  numa árvore  $A$  introduz um desequilíbrio se algum subárvore  $(n, A_n^e, A_n^d)$  torna-se desequilibrado ou seja, depois de remover o nó armazenando a chave  $c$ , temos  $\alpha(A_n^e) = \alpha(A_n^d) + 2$  ou  $\alpha(A_n^e) + 2 = \alpha(A_n^d)$ .

Por exemplo, na figura 11, a remoção do valor 47 introduz, na árvore (1), um desequilíbrio na subárvore enraizada no nó rotulado com 32 (2). Uma rotação dupla a direita nesta subárvore reestabelece o balanço, mais a altura da subárvore resultante é menor que a altura da subárvore inicial, e a subárvore enraizada em 40 fica desequilibrada (3). Uma rotação dupla a esquerda reestabelece o equilíbrio desta subárvore (4).

**Figura 9** Algoritmo de rotação dupla a direita

---

```

1  AVL RotaçãoDuplaADireita(AVL & raiz)
2  {
3      AVL ne = raiz->esq;
4      AVL ned = ne->dir;
5      raiz->esq = ned->dir;
6      ne->dir = ned->esq;
7      ned->esq = ne;
8      ned->dir = raiz;
9      if (ned->bal == 1) {
10         raiz->bal = 0;
11         ne->bal = -1;
12     } else if (ned->bal == 0) {
13         raiz->bal = 0;
14         ne->bal = 0;
15     } else {
16         raiz->bal = 1;
17         ne->bal = 0;
18     }
19     ned->bal = 0;
20     raiz = ned;
21 }

```

---

### 2.3.2 Algoritmo

O algoritmo de remoção de um valor  $v$  numa árvore AVL  $A$  procede de maneira similar com o de remoção numa árvore binária qualquer:

- O primeiro passo é buscar o valor  $v$  a remover na árvore. Se o valor não é encontrado, então o algoritmo termina.
- O segundo passo é remover da árvore o nó  $n$  carregando o valor  $v$  a remover. Como no caso geral das ABBs, há 3 subcasos a tratar:
  1.  $n$  é uma folha;
  2.  $n$  é interior e só tem um filho;
  3.  $n$  é interior e tem dois filhos.

Os dois primeiros casos são tratados da mesma maneira; o tratamento no terceiro caso consiste em trocar o valor do nó a remover o menor valor da subárvore direita e eliminar a folha carregando este valor.

- O último passo é reequilibrar  $A$  depois da remoção. Esse passo pode ser realizado da mesma maneira que no algoritmo da inserção, ou seja, para todo nó  $m$  entre  $n$  e a raiz  $r$ , tem que verificar se um desequilíbrio foi inserido, e neste caso operar a rotação correspondente.

Para implementar esse algoritmo, usaremos alguns algoritmos auxiliares. `RemoverMínimo` remove o nó carregando o menor valor numa árvore (ou seja a folha a mais a esquerda). `RemanejarDir` e `RemanejarEsq` são usadas para reequilibrar uma árvore quando a remoção de um nó diminui a altura de uma subárvore.

A função `RemanejarDir` tem como parâmetros um booleano `diminui` que indica se a altura da subárvore direita da raiz diminuiu, e `raiz`, um ponteiro para a raiz da árvore. A função realiza as rotações necessárias para reequilibrar a árvore depois de da remoção de um nó na subárvore direita. O valor de retorno é booleano e é verdadeiro quando a altura da árvore diminuiu, falso caso contrário. Essa função também atualiza o valor do balanço dos nós envolvidos.

O algoritmo de remanejamento depois de remoção na subárvore esquerda, que chamaremos `RemanejarEsq`, é estruturado exatamente como `RemanejarDir` e é deixado como exercício ao leitor.

**Figura 10** Algoritmo de inserção numa árvore AVL

---

```

1  bool Inserir (t val, AVL & raiz)
2  {
3      if (raiz == 0) {
4          raiz = novo nóAVL (val);
5          return true;
6      } else if (val == raiz->val) { // valor já presente
7          return false;
8      } else if (val < raiz->val) {
9          if (Inserir(val, raiz->esq)) {
10             switch (raiz->bal) {
11                 case 1: raiz->bal = 0;
12                     return false;
13                 case 0: raiz->bal = -1;
14                     return true;
15                 case -1: if (raiz->esq->bal == -1)
16                         RotaçãoSimplesADireita(raiz);
17                     else
18                         RotaçãoDuplaADireita(raiz);
19                     return false;
20             }
21         }
22     } else {
23         if (Inserir(val, raiz->dir)) {
24             switch (raiz->bal) {
25                 case 1: if (raiz->dir->bal == 1)
26                         RotaçãoSimplesAEsquerda(raiz);
27                     else
28                         RotaçãoDuplaAEsquerda(raiz);
29                     return false;
30                 case 0: raiz->bal = 1;
31                     return true;
32                 case -1: raiz->bal = 0;
33                     return false;
34             }
35         }
36     }
37 }

```

---

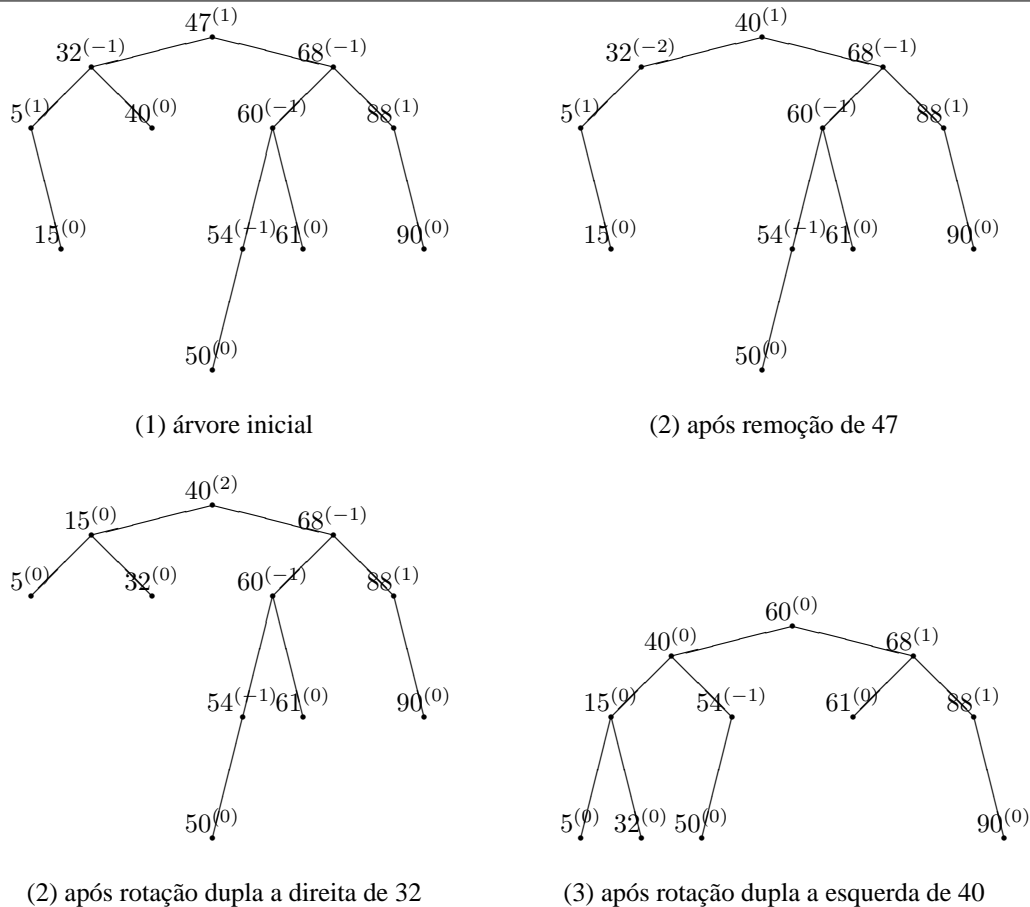
O algoritmo de remoção do nó com o menor valor é chamado *RemoverMínimo*. É parecido com o algoritmo usado para a ABBs, com o tratamento dos desequilíbrios a mais. Ele tem um parâmetro por referência, *raiz*, que é um ponteiro para o nó raiz da árvore. O resultado é verdadeiro quando a remoção do nó mínimo diminui a altura da árvore. O algoritmo é recursivo e opera da maneira seguinte:

**linhas 3-4** Se a árvore é vazia, nada tem a fazer e o resultado é falso.

**linhas 5-7** Se a árvore não é vazia e tem uma subárvore esquerda, tem que remover o mínimo da subárvore esquerda, e eventualmente reequilibrar a subárvore. Como um nó foi removido na subárvore esquerda, a função *RemanejarEsq* é chamada.

**linhas 8-11** Se a árvore não é vazia e não tem subárvore esquerda, então deve se remover o nó na raiz. A nova raiz é a raiz da subárvore direita. Neste caso a altura da árvore diminui e o resultado da operação é verdadeiro.

Os três passos (busca, remoção, remanejamento) do algoritmo de remoção são realizados pela função *Remoção*, dada na figura 14, que procede de maneira recursiva. Os parâmetros são *raiz*, a raiz da subárvore onde o valor vai ser removido, e *valor*, o valor a remover. O tipo retorno da função *Remoção* é o tipo booleano. O valor de retorno é

**Figura 11** Exemplo de remoção numa árvore AVL

verdadeiro quando a altura da árvore descende recursivamente na árvore até encontrar a folha (linha 7) ou achar que ela não está presente (linhas 3 e 4).

Quando o valor é encontrado, ocorre a remoção propriamente dita. Nas linhas 8 a 17, tratam-se os casos onde o valor se encontra numa folha ou num nó interior com um filho só. Nas linhas 18 a 25, trata-se o caso onde o valor está rotulando um nó interior com duas folhas. Nas linhas 19 a 22, o valor mínimo da subárvore direita é procurado e atribuído como rótulo da raiz. Na linha 23, é chamada a função `RemoverMínimo`, dada na figura 13, que remove o nó armazenando esse valor mínimo. Essa função retornará verdadeiro caso a altura da subárvore direita é diminuída, falso caso contrário. A função `RemanejarEsq` é então chamada, e procede as rotações necessárias caso a altura da subárvore direita foi diminuída.

O último caso a tratar é quando o valor rotulando a raiz é diferente do valor a remover (linhas 5-6 e 26-27). Nesta situação, o valor é removido da subárvore correspondente, usando a própria função `Remoção`, e uma das funções de remanejamento `RemanejarEsq` e `RemanejarDir`, é chamada, conforme o caso.

### Exercícios

- Remover a chave 60 da árvore AVL da figura 1.

**Figura 12** Algoritmo de remanejamento a direita de uma árvore AVL

---

```

1  bool RemanejarDir(bool diminui, AVL & raiz)
2  {
3      if (diminui) {
4          if (raiz->bal == -1) {
5              if (raiz->esq->bal == -1 || raiz->esq->bal == 0)
6                  RotaçãoSimplesADireita(raiz);
7              else
8                  RotaçãoDuplaADireita(raiz);
9              return true;
10         } else if (raiz->bal == 0) {
11             raiz->bal = -1;
12             return false;
13         } else {
14             raiz->bal = 0;
15             return true;
16         } else {
17             return true;
18         }
19     }

```

---

**Figura 13** Algoritmo para remover o mínimo de uma árvore AVL

---

```

1  bool RemoverMínimo(AVL & raiz)
2  {
3      if (raiz == 0) {
4          return false;
5      } else if (raiz->esq != 0) {
6          return RemanejarEsq(RemoverMínimo(raiz->esq), raiz);
7      }
8      AVL tmp = raiz;
9      raiz = raiz->dir;
10     delete tmp;
11     return true;
12 }

```

---

- Caracterizar a família de árvores AVL tais que a remoção de qualquer nó provoca um desequilíbrio.
- Dê uma família de árvores AVL tal que a remoção de nós leva a fazer um número logarítmico de remanejamentos.
- No algoritmo de remoção, para remover um valor  $v$  que está rotulando um nó interior  $n$  com duas subárvores, substitui-se este valor com o menor valor  $v_{min}$  da subárvore direita de  $n$ , e é a folha rotulada com  $v_{min}$  que está efetivamente liberada.

Reescrever o algoritmo de remoção de tal maneira que o valor  $v$  seja substituído com o maior valor  $v_{max}$  da subárvore esquerda de  $n$ , e que seja removida folha rotulada com  $v_{max}$ .

## 3 Árvores rubro-negra

### 3.1 Embasamento

As árvores rubro-negra (do inglês *Red-Black trees*) foram inventadas em 1972, 10 anos depois das árvores AVL, por Bayer sob o nome B-árvores binárias simétricas. As árvores rubro-negra são também árvores binárias de busca balanceadas que tem algoritmos de inserção e remoção de complexidade logarítmica no número de nós. Porém, elas são geralmente mais usadas pois tem implementações mais eficientes.

**Figura 14** Algoritmo de remoção numa árvore AVL

---

```

1  bool Remoção (t valor, AVL & raiz)
2  {
3      if (raiz == 0)
4          return false;
5      if (valor < raiz->valor)
6          return RemanejarEsq(Remoção(valor, raiz->esq), raiz);
7      if (valor == raiz->valor) { // caso de remoção
8          // folha ou nó interior com 1 subárvore
9          if (raiz->esq == 0 || raiz->dir == 0) {
10             if (raiz->esq == 0) {
11                 raiz = raiz->dir;
12                 return true;
13             } else {
14                 raiz = raiz->esq;
15                 return true;
16             }
17             // nó interior com 2 subárvores
18         } else {
19             AVL ptr = raiz->dir;
20             while (ptr->esq != 0)
21                 ptr = ptr->esq;
22             raiz->val = ptr->val;
23             return RemanejarDir(RemoverMínimo(raiz->dir), raiz);
24         }
25     }
26     if (valor > raiz->valor)
27         return RemanejarDir(Remoção(valor, raiz->dir), raiz);
28 }

```

---

**Definição 2** Uma árvore rubro-negra (ARN) é uma árvore binária de busca que tem as seguintes propriedades:

1. Cada nó tem uma cor que é rubro ou negro. Por convenção, uma árvore não vazia (ou sub-árvore) tem a cor de sua raiz e uma árvore vazia é negra.
2. A raiz é negra.
3. Qualquer caminho da raiz até uma subárvore vazia tem um número igual de nós negros.
4. As sub-árvores de um nó rubro são negras.

Uma propriedade óbvia resultando da quarta condição é que num caminho da raiz até uma subárvore vazia não pode haver dois nós rubros consecutivos.

A figura 15 contém dois exemplos de árvores binárias de busca coloridas em rubro e negro. A árvore de esquerda satisfaz todas as condições e é rubro-negra, enquanto a árvore da direita não satisfaz a terceira condição, pois só há dois nós negros no caminho da raiz até a sub-árvore esquerda do nó com a chave 88.

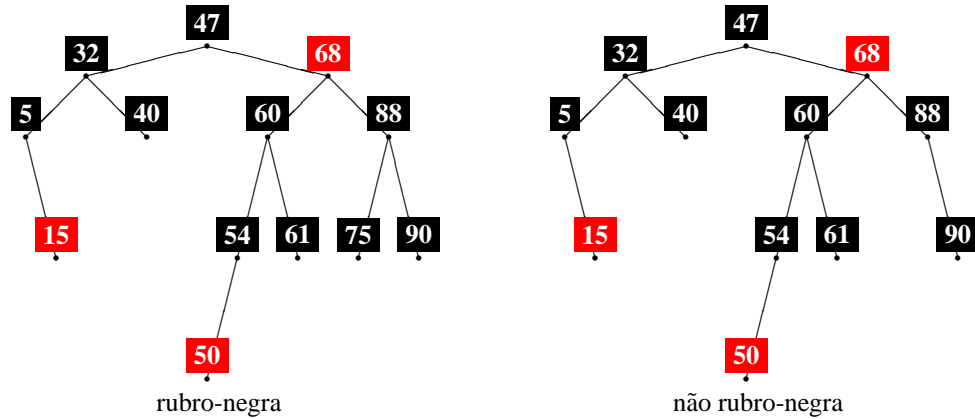
**Definição 3 (Altura negra)** A altura negra de uma árvore rubro-negra  $A$ , denotada  $an(A)$  é o número de nós negros que se encontram nos caminhos da raiz até uma folha.

Observa que, pela terceira condição da definição de árvore rubro-negra, esse número é bem definido. No caso da árvore da figura 15, a altura negra é 3.

**Exercícios** Seja um conjunto  $S$  de 30 chaves distintas.

1. Desenhar uma árvore rubro-negra armazenando  $S$  e de altura negra 3.

Figura 15 Exemplo de árvore rubro-negra



2. Desenhar uma árvore rubro-negra armazenando  $S$  e de altura negra 4.
3. Qual é a maior altura negra possível para uma árvore rubro-negra armazenando  $S$  ?
4. Qual é a menor altura negra possível para uma árvore rubro-negra armazenando  $S$  ?
5. Qual é a maior altura negra possível para uma árvore rubro-negra armazenando  $S$  ?
6. Quais são a maior e a menor altura negra possíveis para uma árvore negra armazenado um conjunto de  $n$  chaves distintas ?

**Proposição 2** A altura de qualquer árvore rubro-negra é logarítmica no número de chaves armazenadas.

**Prova:**

Seja  $A$  uma árvore rubro-negra qualquer,  $n$  o número de chaves armazenadas ( $n$  é o número de nós da árvore), e  $a$  a sua altura. Como a árvore é rubro-negra, ela satisfaz a terceira condição da definição que diz que qualquer caminho da raiz até uma subárvore vazia contém o mesmo número de nós negros. Seja  $p = an(A)$  esse número, por indução sobre  $p$ , podemos mostrar que  $n \geq 2^p - 1$  (caso onde todos os nós são negros).

Também, como todos os caminhos começam com um nó negro e que não pode haver dois nós rubros consecutivos, a altura máxima  $\alpha$  de uma árvore rubro-negra é  $2p$ .

Consequentemente, temos  $\alpha \leq 2p$  e  $n \geq 2^p - 1$ , logo:

$$\begin{aligned}
 n + 1 &\geq 2^p \\
 \ln(n + 1) &\geq p \\
 2 \ln(n + 1) &\geq 2p \\
 2 \ln(n + 1) &\geq \alpha
 \end{aligned}$$

■

**Lema 2** A busca nas árvores rubro-negra tem complexidade logarítmica.

**Prova:**

A proposição 2 diz que a altura de qualquer árvore rubro-negra é logarítmica ( $a = O(\log n)$ ). Como as árvores rubro-negra são árvores binárias de busca e o algoritmo de busca nas árvores binárias de busca é linear na altura da árvore ( $O(a)$ ), a complexidade da busca em árvores rubro-negra é logarítmica no número de nós ( $O(\log n)$ ). ■

Como para árvores AVL, a busca é realizada usando o algoritmo geral das árvores binárias de busca, mas alguns ajustes devem ser feitos para o tratamento da inserção e da remoção. Um trabalho adicional de reequilíbrio também deve ser feito para garantir que a árvore resultante verifique as propriedades das árvores rubro-negras.

**3.2 Inserção**

A operação de inserção numa árvore rubro-negra começa da mesma maneira que a inserção numa árvore binária de busca, através de uma nova folha inserida no lugar adequado. Devemos agora considerar como completar este algoritmo para tornar a árvore resultante rubro-negra, se possível fazendo um número logarítmico de passos.

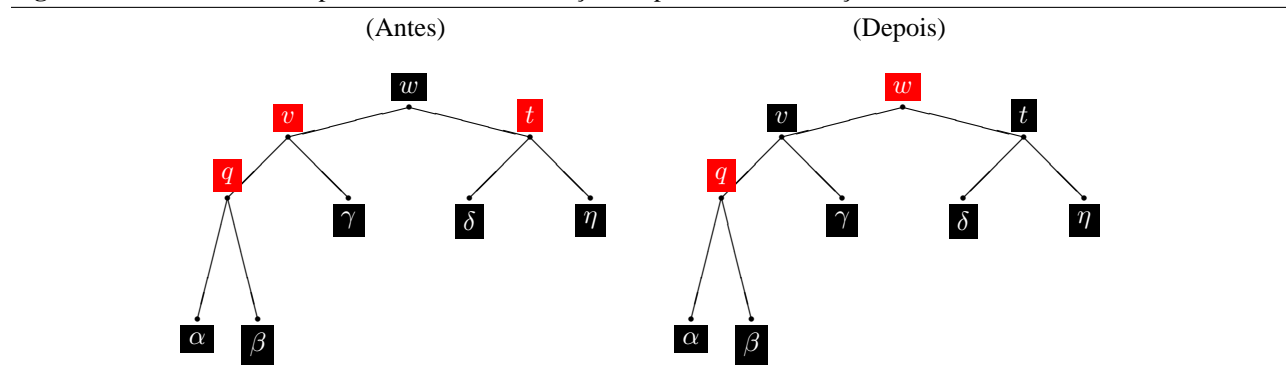
Por razões de eficiência, a cor default utilizada para inserir uma folha é rubro (se fosse negro, a terceira condição ficaria automaticamente invalidada). A segunda condição diz que a raiz é negra. Supondo que a inserção não foi feita numa árvore vazia, esta condição ficou válida. Se a inserção foi feita numa árvore vazia, basta trocar o cor da raiz para preto.

A quarta condição diz que não pode haver dois nós rubros consecutivos. Seja  $q$  a folha inserida,  $v$  o pai de  $q$ . Se  $v$  é negro, então a árvore resultante da inserção satisfaz também a terceira condição e é uma árvore rubro-negra.

Caso  $v$  for rubro,  $v$  não pode ser a raiz da árvore. Portanto o pai de  $v$  existe e é denotado  $w$ . Como, antes da inserção, a árvore era rubro-negra,  $w$  é negro, pela quarta condição. Seja  $t$  a outra subárvore de  $w$  (o “tio” da folha  $q$ ). Temos agora os dois casos seguintes:

**$t$  é rubro** Logo  $t$  não é vazia e suas duas subárvores são negras. O procedimento consiste em inverter as cores de  $v$ ,  $t$  e  $w$ . A situação é resumida na figura 16, onde  $\alpha$  até  $\eta$  denotam subárvores, eventualmente vazias. Agora estudamos as condições da definição no resultado:

**Figura 16** Inversão de cores para reestabelecer condições depois de uma inserção



1. Todos os nós são rubros ou negros.
2. Se  $w$  era a raiz, ela não é mais negra. Basta reverter-la para a cor negra para tornar essa condição válida.
3. Se  $w$  não for a raiz, seu pai pode ser rubro ou negro. Se ele for rubro então a terceira condição não é válida. Basta repetir o procedimento de inversão com  $v$  e  $w$  e seu pai para tornar a condição verdadeira. Esse



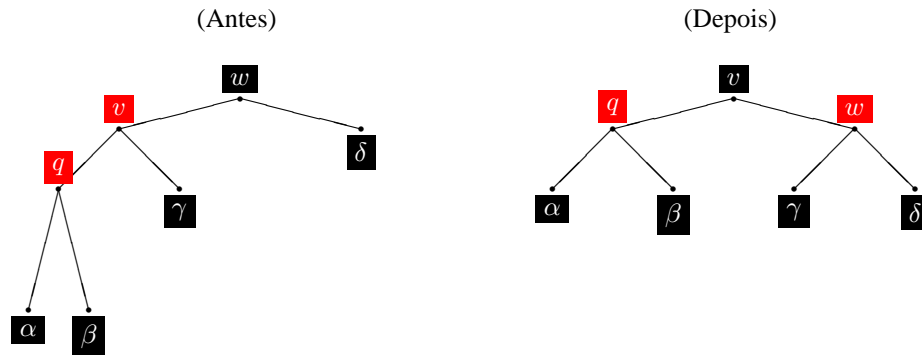
procedimento pode ter que ser repetido até encontrar a raiz da árvore. No pior caso, é preciso fazer um número logarítmico de inversões.

4. O número de nós negros nos caminhos da raiz até as subfolhas não foi alterado por essa transformação, portanto a quarta condição fica válida.

**$t$  é negro** Neste caso, uma operação de rotação, seguida eventualmente de inversão de cores, é suficiente para equilibrar a árvore depois da inserção. Há quatro subcasos a considerar, que são discutidos a seguir:

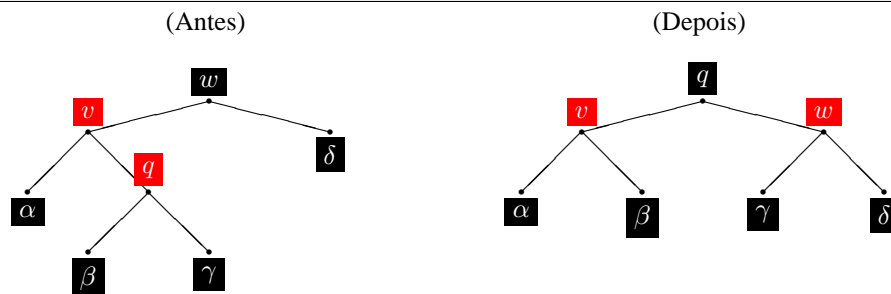
1. Se  $q$  é filho esquerdo de  $v$  e  $v$  é filho esquerdo de  $w$ , é realizada uma rotação simples a direita e as cores de  $v$  e  $w$  são invertidas (ver figura 17).

**Figura 17** Reequilíbrio por rotação e inversão de cores depois de uma inserção: subcaso 1



2. Se  $q$  é filho direito de  $v$  e  $v$  é filho esquerdo de  $w$ , é realizada uma rotação dupla a esquerda e as cores de  $q$  e  $w$  são invertidas (ver figura 18).

**Figura 18** Reequilíbrio por rotação e inversão de cores depois de uma inserção: subcaso 2



3. Se  $q$  é filho direito de  $v$  e  $v$  é filho direito de  $w$ , a configuração é simétrica da do subcaso 1 e é tratada de maneira simétrica.
4. Se  $q$  é filho direito de  $v$  e  $v$  é filho esquerdo de  $w$ , a configuração é simétrica da do subcaso 2 e é tratada de maneira simétrica.

Em todos os subcasos da configuração onde  $t$  é negro a árvore resultante da rotação e da inversão de cores é rubro-negra, pois o único desequilíbrio era a seqüência dos dois nós rubros  $v$  e  $q$ . Consequentemente, uma única operação de

rotação é suficiente para reestabelecer o equilíbrio. O algoritmo de inserção em árvores rubro-negra é composto de uma fase de inserção e de uma fase de balanceamento. A complexidade da inserção, que é a da inserção em árvore binária de busca, é logarítmica. O pior caso da fase de balanceamento é se tiver que aplicar a inversão de cores até a raiz. Como o tamanho do caminho da raiz até qualquer folha é logarítmico, o número de operações também é logarítmico. Em conclusão, a complexidade da inserção em árvores rubro-negras é logarítmica.

### Exercícios

1. Desenhar as figuras dos subcasos 3 e 4 da configuração onde o vértice  $t$  é negro.
2. Desenhar o resultado da inserção da chave 43 na árvore rubro-negra da figura 15.
3. Seja uma árvore rubro-negra inicialmente vazia. Inserir, em ordem crescente, o seguinte conjunto de chaves: {5, 15, 32, 47, 50, 54, 60, 61, 68, 88, 90}. Desenhar a árvore rubro-negra resultante.

#### 3.2.1 Implementação da inserção

Primeiro, definimos os tipos de dados necessários a implementação das árvores rubro-negras. A definição desses tipos é dada na figura 19. O tipo enumerado `cor` é usado para definir a cor dos vértices. O tipo dos vértices é `arn`. Observe que valores deste tipo podem indistintamente denotar um vértice ou a sub-árvore enraizada neste vértice. Um vértice é composto da sua chave, das sub-árvores esquerdas e direitas, e da cor do vértice.

**Figura 19** Definição dos tipos para implementar árvores rubro-negras

```

1  typedef enum {RUBRO, NEGRO} cor;
2  typedef struct arn_ * arn;
3
4  struct arn_ {
5      t chave;
6      arn esq;
7      arn dir;
8      cor tipo;
9  };

```

Em seguida, definimos, na figura 20 dois predicados `Rubro` e `Negro` para testar a cor de um vértice. Esses predicados também se aplicam a subárvores, e em particular a subárvores vazias, que são consideradas como enraizadas por um vértice negro. Essa figura contém também a definição de uma rotina de criação de novos vértices `ARN`; observa que os vértices criados por essa função são sempre folhas rubras.

**Figura 20** Definição funções auxiliares

```

10 static boolean ÉRubro(arn a) {return a && (a->cor == RUBRO);}
11 static boolean ÉNegro(arn a) {return !a || (a->cor == NEGRO);}
12 static arn Criar(t chave)
13 {
14     arn res = malloc(sizeof(struct arn_));
15     res->chave = chave;
16     res->esq = 0;
17     res->dir = 0;
18     res->tipo = RUBRO;
19     return res;
20 }

```

Em seguida, na figura 21 temos uma rotina auxiliar de inserção. Os parâmetros desta função são:

- `chave`, a chave a inserir;

- $v$ , o vértice corrente;
- $w$ , o pai de  $v$ ;
- $r$ , o avô de  $v$ ;
- $raiz$  é o endereço da raiz da árvore (esse parâmetro é passado por referência).

Todos os parâmetros de tipo nó são passados por referência.

Caso um dos nós não existir, então o valor do parâmetro é 0 (o ponteiro nulo). Se a árvore for vazia, o valor de  $raiz$  é o endereço nulo. O resultado da função `Inserir` é um inteiro que indica qual é o número de nós rubros consecutivos na árvore. Esse resultado é:

O código da função `Inserir` é o código da inserção em árvore binária de busca adicionado com algumas instruções adicionais (linhas 36 e 44 até 58) para propagar as informações sobre o equilíbrio e chamadas a função `Remanejar` quando essas informações de equilíbrio indicam que a situação pode necessitar um remanejamento da árvore.

**Figura 21** Função auxiliar de inserção

---

```

21  boolean
22  Inserir
23  (const int chave, arn & q, arn & v, arn & w)
24  {
25      if (q == 0) {
26          q = Criar(chave);
27          return true;
28      } else if (chave != q->chave) {
29          boolean e;
30          if (chave < q->chave) {
31              e = Inserir(chave, q->esq, q, v);
32          } else {
33              e = Inserir(chave, q->dir, q, v);
34          }
35          if (ÉRubro(q)) {
36              if (e) {
37                  if (chave < q->chave) {
38                      Remanejar(q->esq, q, v);
39                      return false;
40                  } else {
41                      Remanejar(q->dir, q, v);
42                      return false;
43                  }
44              } else {
45                  return true;
46              }
47          } else {
48              return false;
49          }
50      } else {
51          return ÉRubro(q);
52      }
53  }
54  }
55  }
56  }
```

---

A definição da função principal de inserção é dada na figura 22. Ela tem como parâmetro uma árvore rubro-negra (passado por referência) e uma chave a inserir. Essa função basicamente consiste em chamar a função auxiliar com os parâmetros apropriados (linha 61). Para tratar o caso de inserções numa árvore vazia ou de inserções que trocaram a cor da raiz para rubro, a função de inserção sistematicamente coloca a cor da raiz em negro (linha 63).

Finalmente, a definição da função auxiliar de balanceamento é dada na figura 23. Os parâmetros desta função são

**Figura 22** A função principal de inserção

---

```

57 void arnInserir (t chave, arn raiz)
58 {
59     if (raiz == 0) {
60         raiz = Criar(chave);
61     } else {
62         arn pai = 0, avo = 0;
63         Inserir(chave, raiz, pai, avo)
64     }
65     raiz->cor = NEGRO;
66 }

```

---

- q o nó corrente;
- v o pai de q;
- w o avô de q;
- ptraiz o endereço da raiz da árvore (é uma passagem de parâmetros por referência).

O resultado desta função é 0 ou 1 e tem o mesmo significado que para a função `Inserir`. O código basicamente implementa todos os casos examinados anteriormente. A função de remanejamento utiliza funções auxiliares de rotação, parecidas com a de árvores AVL (sem atualização do valor de balanço).

**Figura 23** Função auxiliar de balanceamento

---

```

67 void Remanejar (arn & q, arn & v, arn & w)
68 {
69     arn t;
70     if (v == w->esq) t = w->dir;
71     else t = w->esq;
72     if (Rubro(t)) {
73         t->cor = v->cor = NEGRO;
74         w->cor = RUBRO;
75     } else {
76         if (q == v->esq && v == w->esq ||
77             q == v->dir && v == w->dir) {
78             v->cor = NEGRO;
79             w->cor = RUBRO;
80             if (q == v->esq) RotaçãoSimplesADireita(w);
81             else RotaçãoSimplesAEsquerda(w);
82         } else {
83             q->cor = NEGRO;
84             w->cor = RUBRO;
85             if (v == w->esq) RotaçãoDuplaADireita(w);
86             else RotaçãoDuplaAEsquerda(w);
87         }
88     }
89 }

```

---

### Exercícios

- Utilizar a função `arnInserir` da figura 22 para inserir, em seqüência, as chaves 5, 15, 32, 47, 50, 54, 60, 61, 68, 88, 90 numa árvore rubro-negra inicialmente vazia.

- O algoritmo de inserção dado, no pior caso, tem que fazer um número logarítmico de inversão de cores. Para evitar isso, basta garantir, que quando uma folha nova  $q$  é criada, o nó tio  $t$  (ver figura 16) não é rubro. Neste caso, é só criar a folha e fazer eventualmente uma única rotação simples ou dupla.

Projetar um algoritmo de inserção que, enquanto desce recursivamente a árvore procurando o lugar de inserção, cada vez que encontra um nó  $x$  com dois filhos rubros, torna  $x$  rubro e seus filhos negros e, caso o pai de  $x$  for rubro também, realiza uma rotação simples ou dupla para equilibrar a árvore.

Verificar então que na inserção da folha, uma única rotação é suficiente para tornar a árvore rubro-negra.

### 3.3 Remoção

A remoção em árvores rubro-negra pode ser realizada também com um número logarítmico de operações. O procedimento de remoção é composto de uma etapa de remoção em árvore binária de busca seguido de uma etapa de balanceamento, caso as propriedades rubro-negras teriam sido destruídas durante a operação.

No caso da remoção em árvores binárias de busca gerais, se a chave removida está guardado num nó com dois filhos, há uma etapa anterior de troca de valores de tal maneira que, no final, sempre há remoção efetiva de um nó com no máximo um filho.

Se o nó removido for rubro, a árvore fica rubro-negra, pois todas as condições da definição ficam válidas:

1. Os nós resultantes tem cor rubra ou negra.
2. A raiz, que era negra, não foi removida.
3. Nenhum nó negro foi removido, portanto todos os caminhos da raiz até uma folha tem um número igual de nós negros.
4. Os filhos de todos os nós rubros não removidos não foram alterados e portanto ficam negros.

#### Exercícios

Remover todas as chaves armazenadas em nós rubros da árvore da figura 15.

Se o nó removido for negro, o número de nós de pelo menos um caminho foi decrementado e consequentemente a terceira condição ficou inválida. Quando isto acontece, dois tipos de solução são possíveis:

**remoção efetiva** Através de um número logarítmico de operações, a remoção efetiva reestabelece as propriedades para que a árvore seja rubro-negra. Essas operações são detalhadas em seguida.

**remoção preguiçosa** A remoção preguiçosa consiste em marcar o nó como removido, mas sém tira-lo da árvore. Nenhum remanejamento é necessário. Em compensação, os algoritmos de inserção e busca devem ser modificados para levar em conta que alguns nós da árvore devem ser considerados como ausentes. A adoção desta solução é possível quando as árvores rubro-negras são usadas no contexto de uma aplicação com poucas operações de remoção.

#### Exercícios

Projetar os algoritmos de inserção, busca e remoção para árvores rubro-negra com remoção preguiçosa.

#### Remoção efetiva

Quando o nó a remover  $y$  é negro, todos os caminhos da raiz até uma folha passando por esse nó tem um nó negro a menos. Seja  $x$  o nó que passará a ocupar a posição de  $y$  na árvore. O problema da remoção efetiva é resolvido atribuindo negro à cor de  $x$ . Assim permanece igual a altura negra de todos os caminhos contendo  $x$ , antes e depois da inserção.

Porém, se  $x$  já era negro, ela agora passa a ser dois vezes negro, o que torna inválida a primeira condição da definição, e é preciso remanejar a árvore para eliminar essa situação.

No caso de  $x$  ser a raiz, então basta torná-lo simplesmente negro: a altura negra de todos os caminhos da árvore é decrementada, e a terceira condição permanece verdadeira.

Caso  $x$  não seja a raiz, seja  $v$  seu pai, e  $w$  seu irmão. A seguir é considerado o caso de  $x$  ser o filho esquerdo de  $v$ . O outro caso é simétrico e é omitido.

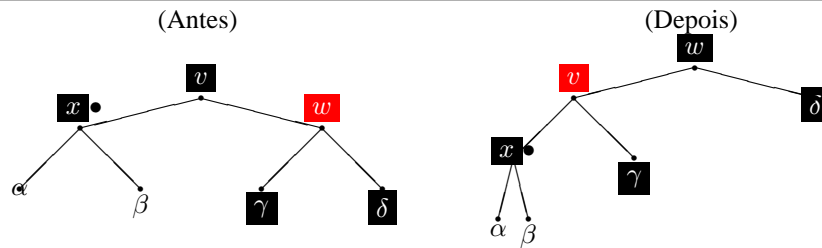
As figuras 24 até 27 ilustram como a árvore rubro-negra deve ser remanejada para reestabelecer as propriedades da definição. Nessas figuras, nós anotados com o símbolo  $\bullet$  tem um ponto negro a mais. Quando um nó aparece com um índice  $c$  (ou  $c'$ ), a convenção é que a cor deste nó é  $c$  (ou  $c'$ ). Em todas as remanejamentos, o número de nós negros da raiz da sub-árvore até todas as sub-árvores ( $\alpha, \beta, \dots$ ) deve ser preservado.

São apresentados a seguir quatro sub-casos. No primeiro caso,  $w$  é rubro; nos demais três,  $w$  é negro, e cada um corresponde a uma coloração diferente dos filhos de  $w$ .

1. O primeiro caso, ilustrado na figura 24 considera a situação onde  $w$  é rubro<sup>2</sup>. Nesta situação, é realizada uma rotação simples a esquerda de  $v$ , e as cores de  $v$  e  $w$  são modificadas.

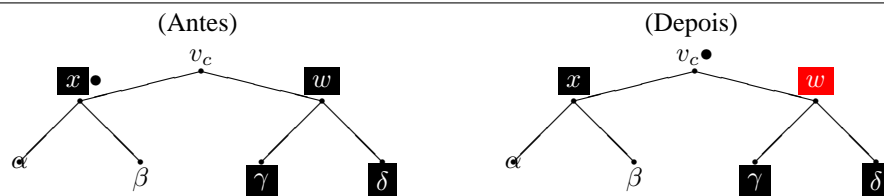
O resultado desta modificação é que  $x$  permanece duplamente negro. Porém, o seu irmão agora também é negro, e o tratamento de um dos casos apresentados a seguir deve ser aplicado.

**Figura 24** Remanejamento após remoção – caso (1)



2. O segundo caso configura a situação onde ambas sub-árvores de  $w$  são negras e é ilustrado na figura 25. Este remanejamento consiste em subir um ponto negro dos nós  $x$  e  $w$ , que passam a ser negro e rubro respectivamente, no nó  $v$ . Se ele era anteriormente rubro, ele torna-se negro. Se ele era anteriormente negro, ele torna-se duplamente negro, e um novo remanejamento é necessário no nível superior.

**Figura 25** Remanejamento após remoção – caso (2)

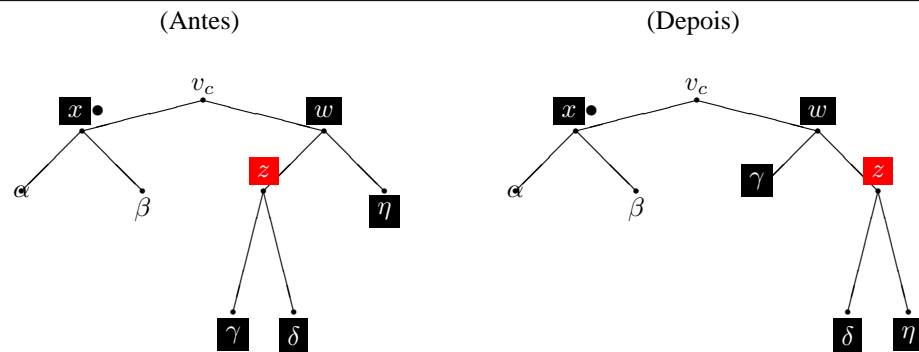


3. No terceiro caso, ilustrado na figura 26, a sub-árvore esquerda de  $w$  é rubra, e a direita é negra. Seja  $z$  o filho esquerdo de  $w$ . É então realizada uma rotação simples a direita de  $w$ , e uma inversão das cores de  $w$  e de  $w$ .

<sup>2</sup>Nesta figura, assim também como nas seguintes, utilizaremos como convenção de não colocar cor a um sub-árvore quando o tratamento é independente desta cor (por exemplo,  $\alpha$ ).

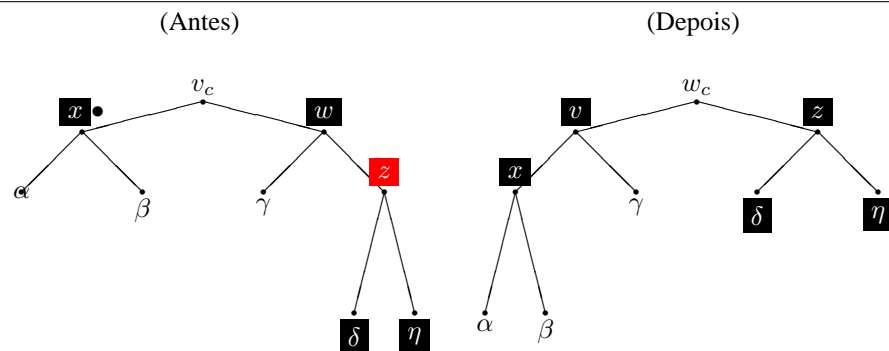
O nó  $x$  permanece duplamente negro, mas configura-se agora uma situação diferente, onde a sub-árvore direita  $w$  é rubra, cujo tratamento é apresentado a seguir.

**Figura 26** Remanejamento após remoção – caso (3)



4. O quarto e último caso corresponde portanto à situação onde a sub-árvore direita de  $w$  é rubra. Seja  $z$  o filho direito de  $w$ . A solução consiste em fazer uma rotação simples a esquerda de  $v$ , atribuir aos nós  $v$  e  $z$  a cor negra, e a  $w$  a cor que era a de  $v$ .

**Figura 27** Remanejamento após remoção – caso (4)



### Exercícios

1. Mostrar que os remanejamentos das figuras 24 até 25 não alteram o número de nós negros nos caminhos das sub-árvores.
2. Na remoção efetiva, quais são os diferentes casos possíveis de remanejamento quando  $x$  é filho direito de  $v$  na árvore?
3. Realizar operações de remoção efetiva em um caso concreto de árvores rubro-negra.
4. Desenhar o algoritmo que realiza a remoção efetiva em árvores rubro-negra.