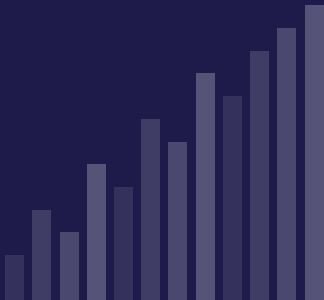

AULA 25 · BLOCO 3: TABELAS HASH E ÁRVORES

Árvores rubro-negras

Seções 13.1 a 13.3

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Enunciar as cinco propriedades rubro-negras
- ▶ Provar que a altura é no máximo $2 \lg(n + 1)$
- ▶ Executar RB-INSERT-Fixup nos três casos
- ▶ Medir altura e rotações e conferir as propriedades a cada experimento



Teoria

Propriedades, altura negra, lema 13.1, inserção e conserto



Prática em C

Rubro-negra em C, como no livro, com sentinela



Experimento

Altura em ordem e aleatória; rotações por inserção



Conexões

Linux, bibliotecas padrão, memória virtual e redes

As cinco propriedades

1. Todo nó é vermelho ou preto.
2. A raiz é preta.
3. Toda folha (NIL) é preta.
4. Um nó vermelho tem os dois filhos pretos.
5. Todo caminho de um nó até as folhas abaixo dele tem o mesmo número de nós pretos: a **altura negra** $bh(x)$.

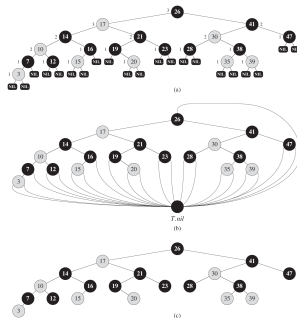


Figura 13.1 de Cormen et al. (2009) — (a) árvore rubro-negra com folhas NIL; (b) com uma sentinela $T.nil$; (c) sem desenhar as folhas.

Lema 13.1: $\text{altura} \leq 2 \lg(n + 1)$

- ▶ A subárvore de x tem pelo menos $2^{bh(x)} - 1$ nós internos (indução na altura).
- ▶ Pela propriedade 4, pelo menos metade dos nós de qualquer caminho da raiz a uma folha é preta: $bh(\text{raiz}) \geq h/2$.
- ▶ Logo $n \geq 2^{h/2} - 1$, ou seja, $h \leq 2 \lg(n + 1)$.

O caminho mais longo tem no máximo o **dobro** do mais curto: um balanceamento mais frouxo que o da AVL, mas mais barato de manter.

Busca, mínimo, máximo, sucessor e predecessor: $O(\lg n)$ sem mudança alguma, porque uma rubro-negra é uma ABB.

Inserção: o conserto

Insere-se z **vermelho**, como numa ABB. Só a propriedade 4 (ou a 2) pode quebrar.

Enquanto o pai de z for vermelho:

1. **Tio vermelho:** pinta pai e tio de preto, avô de vermelho, sobe z dois níveis.
2. **Tio preto, z filho "de dentro":** rotação no pai: vira o caso 3.
3. **Tio preto, z filho "de fora":** recolore e gira o avô. Fim.

No máximo **2 rotações** por inserção.

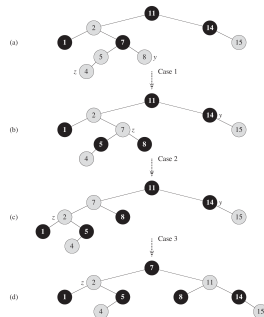


Figura 13.4 de Cormen et al. (2009) — RB-INSERT-Fixup após inserir 4: caso 1 (recolorir), caso 2 (girar o pai) e caso 3 (recolorir e girar o avô).



Prática em C

- ▶ Sentinela nil preta: folhas e pai da raiz
- ▶ Rotações e RB-INSERT-Fixup linha a linha do livro
- ▶ Teste com as chaves da Figura 13.4

Árvore rubro-negra em C (1/7)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Uma sentinela NIL (preta) faz o papel de todas as folhas e do pai da raiz. */
5  typedef enum { VERMELHO, PRETO } Cor;
6  typedef struct No { int chave; Cor cor; struct No *esq, *dir, *pai; } No;
7  typedef struct { No *raiz, *nil; } Arvore;
8
9  void gira_esquerda(Arvore *T, No *x) {           // LEFT-ROTATE
10     No *y = x->dir;
11     x->dir = y->esq;
12     if (y->esq != T->nil) y->esq->pai = x;
13     y->pai = x->pai;
14     if (x->pai == T->nil) T->raiz = y;
15     else if (x == x->pai->esq) x->pai->esq = y;
16     else x->pai->dir = y;
```

Árvore rubro-negra em C (2/7)

```
17     y->esq = x;
18     x->pai = y;
19 }
20
21 void gira_direita(Arvore *T, No *y) {           // RIGHT-ROTATE (espelho)
22     No *x = y->esq;
23     y->esq = x->dir;
24     if (x->dir != T->nil) x->dir->pai = y;
25     x->pai = y->pai;
26     if (y->pai == T->nil) T->raiz = x;
27     else if (y == y->pai->dir) y->pai->dir = x;
28     else y->pai->esq = x;
29     x->dir = y;
30     y->pai = x;
31 }
```

Árvore rubro-negra em C (3/7)

```
33 void conserta(Arvore *T, No *z) {           // RB-INSERT-FIXUP
34     while (z->pai->cor == VERMELHO) {
35         if (z->pai == z->pai->pai->esq) {
36             No *y = z->pai->pai->dir;         // tio
37             if (y->cor == VERMELHO) {       // caso 1: recolore e sobe
38                 z->pai->cor = PRETO;
39                 y->cor = PRETO;
40                 z->pai->pai->cor = VERMELHO;
41                 z = z->pai->pai;
42             } else {
43                 if (z == z->pai->dir) {       // caso 2: vira o caso 3
44                     z = z->pai;
45                     gira_esquerda(T, z);
46                 }
47                 z->pai->cor = PRETO;         // caso 3: recolore e gira
48                 z->pai->pai->cor = VERMELHO;
```

Árvore rubro-negra em C (4/7)

```
49         gira_direita(T, z->pai->pai);
50     }
51 } else {                                     // espelho: pai é filho direito
52     No *y = z->pai->pai->esq;
53     if (y->cor == VERMELHO) {
54         z->pai->cor = PRETO;
55         y->cor = PRETO;
56         z->pai->pai->cor = VERMELHO;
57         z = z->pai->pai;
58     } else {
59         if (z == z->pai->esq) {
60             z = z->pai;
61             gira_direita(T, z);
62         }
63         z->pai->cor = PRETO;
```

Árvore rubro-negra em C (5/7)

```
64         z->pai->pai->cor = VERMELHO;
65         gira_esquerda(T, z->pai->pai);
66     }
67 }
68 }
69 T->raiz->cor = PRETO;
70 }
71
72 void insere(Arvore *T, int k) { // RB-INSERT
73     No *z = malloc(sizeof(No)), *y = T->nil, *x = T->raiz;
74     z->chave = k;
75     while (x != T->nil) { y = x; x = k < x->chave ? x->esq : x->dir; }
76     z->pai = y;
77     if (y == T->nil) T->raiz = z;
78     else if (k < y->chave) y->esq = z;
79     else y->dir = z;
```

Árvore rubro-negra em C (6/7)

```
80     z->esq = z->dir = T->nil;
81     z->cor = VERMELHO;
82     conserta(T, z);
83 }
84
85 /* bh(raiz): pretos de raiz até NIL, sem contar a raiz e contando o NIL.
86    A raiz é preta, então basta contar os pretos de um caminho, raiz incluída. */
87 int altura_negra(Arvore *T) {
88     int bh = 0;
89     for (No *x = T->raiz; x != T->nil; x = x->esq)
90         if (x->cor == PRETO) bh++;
91     return bh;
92 }
93
94 void libera(Arvore *T, No *x) {
95     if (x == T->nil) return;
```

Árvore rubro-negra em C (7/7)

```
96     libera(T, x->esq);
97     libera(T, x->dir);
98     free(x);
99 }
100
101 int main(void) {
102     No nil = {0, PRETO, NULL, NULL, NULL};
103     Arvore T = {&nil, &nil};
104     int k;
105     while (scanf("%d", &k) == 1) insere(&T, k);
106     if (T.raiz == T.nil) return 1;
107     printf("raiz %d (%s), altura negra %d\n", T.raiz->chave,
108           T.raiz->cor == PRETO ? "preta" : "vermelha", altura_negra(&T));
109     libera(&T, T.raiz);
110     return 0;
111 }
```



Experimento

- ▶ Propriedades conferidas em toda árvore (aborta se falhar)
- ▶ Chaves em ordem e aleatórias, n até 2^{21}
- ▶ Rotações por inserção

Altura garantida e conserto barato

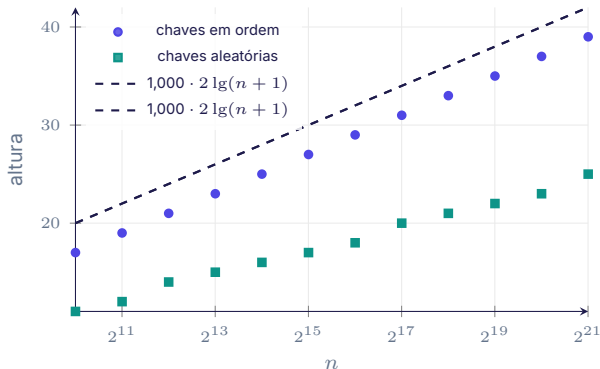
Hipótese. Pelo lema 13.1, a altura fica abaixo de $2 \lg(n+1)$ em **qualquer** ordem de inserção. Cada inserção faz no máximo 2 rotações; em média, bem menos, e o número deve ser **constante**, independente de n .

Método

1. $n = 2^{10}, \dots, 2^{21}$
2. Conferência: altura negra igual em todos os caminhos e nenhum vermelho com filho vermelho
3. Chaves em ordem crescente e chaves aleatórias
4. Contador de rotações

EXPERIMENTO

Altura da rubro-negra



n	medido	razão
1.024	17,0	0,850
4.096	21,0	0,875
16.384	25,0	0,893
65.536	29,0	0,906
262.144	33,0	0,917
2.097.152	39,0	0,929

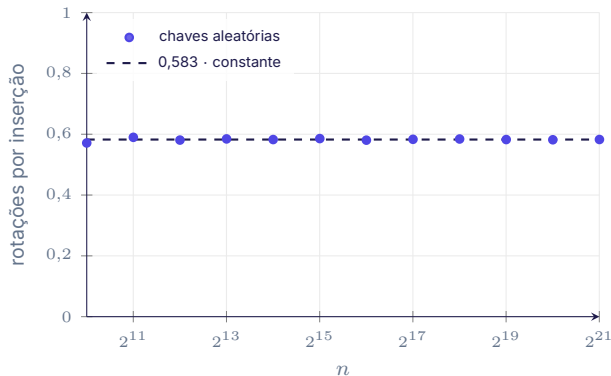
razão = medido $\div$ $2 \lg(n + 1)$

✓ sempre $\leq$ o limite

Em ordem, a altura fica perto do limite ($2 \lg n - 3$); com chaves aleatórias, perto de $\lg n + 4$.

🧪 EXPERIMENTO

Rotações por inserção



n	medido	razão
1.024	0,571	0,571
4.096	0,581	0,581
16.384	0,583	0,583
65.536	0,581	0,581
262.144	0,584	0,584
2.097.152	0,583	0,583

razão = medido $\div$ constante

✓ desvio máx. 0,3%

Cerca de 0,58 rotação por inserção, menos que a AVL (0,7).

EXPERIMENTO

O que o experimento mostrou

As propriedades valeram em todas as árvores, e a altura ficou sempre abaixo de $2\lg(n+1)$, mesmo com chaves em ordem. O conserto custa $\approx 0,58$ rotação por inserção: menos que a AVL, ao preço de árvores um pouco mais altas. É o equilíbrio que levou as bibliotecas padrão a escolher a rubro-negra.

➤ Reproduza: `gcc -O2 aula25_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Kernel Linux
- ▶ Bibliotecas padrão
- ▶ Redes
- ▶ Estruturas persistentes

Onde isso aparece na computação



Kernel Linux

O escalonador CFS ordenava processos numa rubro-negra pelo tempo de CPU usado; a memória virtual guarda as áreas mapeadas de cada processo numa delas.



Bibliotecas padrão

`std::map` e `std::set` do C++ e `TreeMap` do Java são rubro-negras; o `HashMap` do Java vira rubro-negra quando uma lista fica longa.



Redes

Temporizadores de alta resolução e tabelas de conexões usam rubro-negras para inserções e remoções frequentes com pior caso garantido.



Estruturas persistentes

Rubro-negras funcionais (Okasaki) são a base de mapas imutáveis em Scala e Clojure: versões antigas continuam válidas a custo $O(\lg n)$.

Para casa

Exercícios

1. (13.1-1) Desenhe a ABB completa de altura 3 com as chaves 1..15 e pinte-a com alturas negras 2, 3 e 4.
2. (13.1-5) Mostre que o caminho mais longo de x a uma folha tem no máximo o dobro do mais curto.
3. (13.3-2) Mostre as árvores após inserir 41, 38, 31, 12, 19, 8 numa rubro-negra vazia.
4. (13.3-5) Mostre que, com $n > 1$ inserções, a árvore tem pelo menos um nó vermelho.
5. Compare no experimento a altura média da AVL (aula 24) e da rubro-negra com as mesmas chaves.

Leitura

Cormen et al., seções 13.1 a 13.3.

Código

josedihego.net/estruturas