

MC-202

Árvores Balanceadas

Marcelo S. Reis
msreis@ic.unicamp.br

Universidade Estadual de Campinas

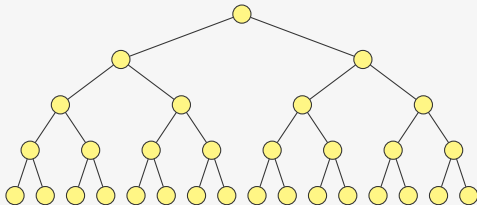
Segundo semestre de 2023

Eficiência da busca, inserção e remoção

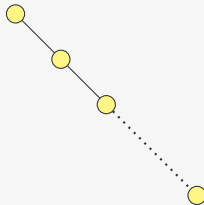
Qual é o tempo da busca, inserção e remoção em ABBs?

- depende da altura da árvore...

Ex: 31 nós



Melhor árvore: $\lceil \lg n + 1 \rceil$



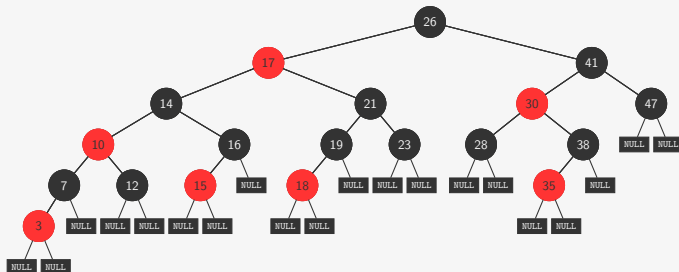
Pior árvore: n

Para ter a pior árvore basta inserir em ordem crescente...

Veremos uma árvore **balanceada**

- Não é a melhor árvore possível, mas é “quase”
- Operações em $O(\lg n)$

Árvores Rubro-Negras Esquerdistas



Uma árvore **rubro-negra** esquerdista é uma ABB tal que:

1. Todo nó é ou **vermelho** ou **preto**
2. A raiz é **preta**
3. As folhas são **NULL** e tem cor **preta**
4. Se um nó é **vermelho**, seus dois filhos são **pretos**
 - ele é o filho esquerdo do seu pai (por isso, *esquerdista*)
5. Em cada nó, todo caminho dele para uma de suas folhas descendentes tem a mesma quantidade de nós **pretos**
 - Não contamos o nó
 - É a altura-**negra** do nó

Altura da Árvore Rubro-Negra Esquerdista

Seja bh a altura-**negra** da árvore.

A árvore tem pelo menos $2^{bh} - 1$ nós não nulos

Para provar, basta utilizar indução matemática:

- Se $bh = 0$
 - a árvore é apenas uma folha NULL
 - tem exatamente $2^{bh} - 1 = 0$ nós não nulos
- Se $bh > 0$
 - seus filhos têm altura-**negra** pelo menos $bh - 1$
 - cada subárvore tem pelo menos $2^{bh-1} - 1$ nós não nulos
 - a árvore tem pelo menos $2(2^{bh-1} - 1) + 1$ nós não nulos
 - ou seja, tem pelo menos $2^{bh} - 1$ nós não nulos

A altura-**negra** bh é pelo menos metade da altura h da árvore

- Não existe nó **vermelho** com filho **vermelho**
- O número de nós não nulos n é $n \geq 2^{bh} - 1 \geq 2^{h/2} - 1$
- Ou seja, $h \leq 2 \lg(n + 1) = O(\lg n)$

Alterando a Struct e testando a cor

```
1 enum cor {VERMELHO, PRETO};
2
3 typedef struct no *p_no;
4
5 struct no {
6     int chave;
7     enum cor cor;
8     p_no esq, dir;
9 };
```

```
1 int ehVermelho(p_no x) {
2     if (x == NULL)
3         return 0;
4     return x->cor == VERMELHO;
5 }
```

```
1 int ehPreto(p_no x) {
2     if (x == NULL)
3         return 1;
4     return x->cor == PRETO;
5 }
```

Rotação para a esquerda

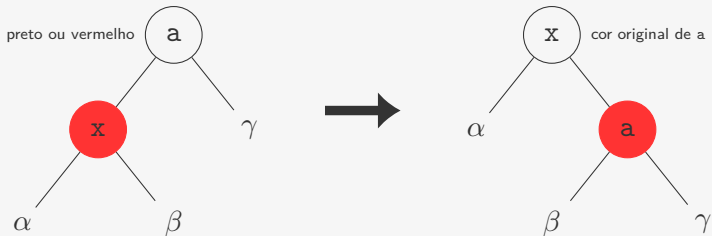


```
1 p_no rotaciona_para_esquerda(p_no raiz) {  
2   p_no x = raiz->dir;  
3   raiz->dir = x->esq;  
4   x->esq = raiz;  
5   x->cor = raiz->cor;  
6   raiz->cor = VERMELHO;  
7   return x;  
8 }
```

Note que a rotação:

- não estraga a propriedade de busca
- não estraga a propriedade da altura negra

Rotação para a direita

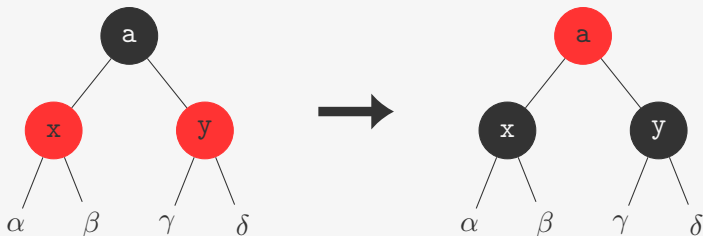


```
1 p_no rotaciona_para_direita(p_no raiz) {  
2   p_no x = raiz->esq;  
3   raiz->esq = x->dir;  
4   x->dir = raiz;  
5   x->cor = raiz->cor;  
6   raiz->cor = VERMELHO;  
7   return x;  
8 }
```

Note que a rotação:

- não estraga a propriedade de busca
- não estraga a propriedade da altura negra

Subindo a cor



```
1 void sobe_vermelho(p_no raiz) {  
2     raiz->cor = VERMELHO;  
3     raiz->esq->cor = PRETO;  
4     raiz->dir->cor = PRETO;  
5 }
```

Subir a cor não estraga a propriedade da altura negra

- mas pode pintar a raiz de vermelho

Inserindo

Inserimos como em uma ABB, mas precisamos manter as propriedades da árvore **rubro-negra** esquerdista

```
1 p_no inserir_rec(p_no raiz, int chave) {
2     p_no novo;
3     if (raiz == NULL) {
4         novo = malloc(sizeof(struct no));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         novo->cor = VERMELHO;
8         return novo;
9     }
10    if (chave < raiz->chave)
11        raiz->esq = inserir_rec(raiz->esq, chave);
12    else
13        raiz->dir = inserir_rec(raiz->dir, chave);
14    /* corrige a árvore */
15    return raiz;
16 }
17
18 p_no inserir(p_no raiz, int chave) {
19     raiz = inserir_rec(raiz, chave);
20     raiz->cor = PRETO; ← Mantém a raiz preta
21     return raiz;
22 }
```

Corrigindo

- Iremos corrigir cada propriedade
 - supomos que as subárvores esquerda e direita já satisfazem todas propriedades, com exceção da raiz **preta**
 - e corrigimos as propriedades de **raiz**, uma por vez

Não queremos que só o filho direito seja **vermelho**:

```
if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))  
    raiz = rotaciona_para_esquerda(raiz);
```

Nem que um nó **vermelho** seja filho esquerdo de nó **vermelho**:

```
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))  
    raiz = rotaciona_para_direita(raiz);
```

Também não queremos que ambos filhos sejam **vermelhos**:

```
if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))  
    sobe_vermelho(raiz);
```

Inserção - Caso 1

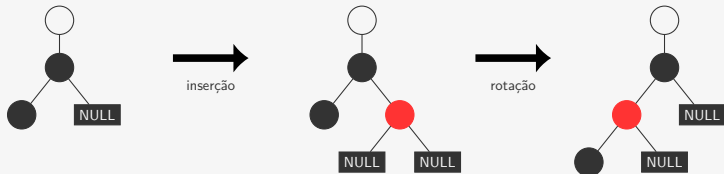
- Nó atual é **preto**
 - não sabemos a cor do seu pai
 - nem se ele é o filho esquerdo ou direito
- Filho direito é **preto** (tem que ser - por que?)
- Inserimos no filho esquerdo



```
1  /* corrige a árvore */  
2  if (ehVermelho(raiz->dir) && ehPreto(raiz->esq)) ←  
3      raiz = rotaciona_para_esquerda(raiz);  
4  if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) ←  
5      raiz = rotaciona_para_direita(raiz);  
6  if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) ←  
7      sobe_vermelho(raiz);
```

Inserção - Caso 2

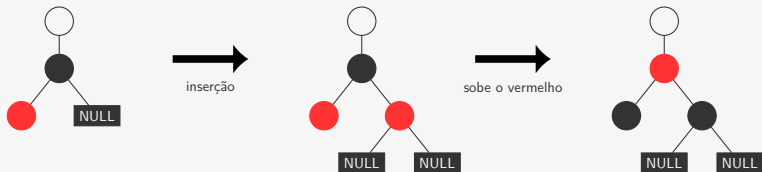
- Nó atual é **preto**
 - não sabemos a cor do seu pai
 - nem se ele é o filho esquerdo ou direito
- Filho esquerdo é **preto**
- Inserimos no filho direito



```
1  /* corrige a árvore */
2  if (ehVermelho(raiz->dir) && ehPreto(raiz->esq)) ←
3      raiz = rotaciona_para_esquerda(raiz); ←
4  if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) ←
5      raiz = rotaciona_para_direita(raiz);
6  if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) ←
7      sobe_vermelho(raiz);
```

Inserção - Caso 3

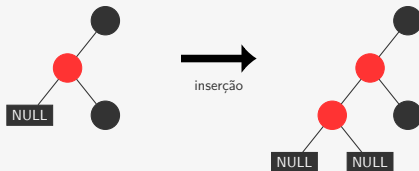
- Nó atual é **preto**
 - não sabemos a cor do seu pai
 - nem se ele é o filho esquerdo ou direito
- Filho esquerdo é **vermelho**
- Inserimos no filho direito



```
1  /* corrige a árvore */  
2  if (ehVermelho(raiz->dir) && ehPreto(raiz->esq)) ←  
3      raiz = rotaciona_para_esquerda(raiz);  
4  if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) ←  
5      raiz = rotaciona_para_direita(raiz);  
6  if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) ←  
7      sobe_vermelho(raiz); ←
```

Inserção - Caso 4

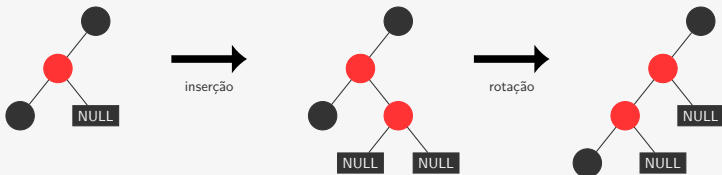
- Nó atual é **vermelho**
 - seu pai é **preto** (ele não é a raiz - por que?)
 - é o filho esquerdo (por que?)
- Inserimos no filho esquerdo



```
1  /* corrige a árvore */  
2  if (ehVermelho(raiz->dir) && ehPreto(raiz->esq)) ←  
3      raiz = rotaciona_para_esquerda(raiz);  
4  if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) ←  
5      raiz = rotaciona_para_direita(raiz);  
6  if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) ←  
7      sobe_vermelho(raiz);
```

Inserção - Caso 5

- Nó atual é **vermelho**
 - seu pai é **preto** (ele não é a raiz)
 - é o filho esquerdo
- inserimos no filho direito



```
1  /* corrige a árvore */  
2  if (ehVermelho(raiz->dir) && ehPreto(raiz->esq)) ←  
3      raiz = rotaciona_para_esquerda(raiz); ←  
4  if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) ←  
5      raiz = rotaciona_para_direita(raiz);  
6  if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) ←  
7      sobe_vermelho(raiz);
```

Resolvendo problemas no pai

Quais problemas sobraram para o pai resolver?

- Talvez o filho direito seja **vermelho** (não é esquerdista)
- Só pode ter acontecido porque a cor **vermelha** subiu

Se o filho esquerdo for **preto**, basta rotacionar para a esquerda



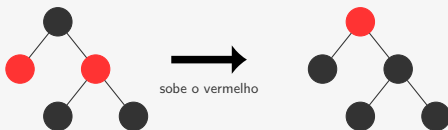
```
1  /* corrige a árvore */
2  if (ehVermelho(raiz->dir) && ehPreto(raiz->esq)) ←
3      raiz = rotaciona_para_esquerda(raiz); ←
4  if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) ←
5      raiz = rotaciona_para_direita(raiz);
6  if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) ←
7      sobe_vermelho(raiz);
```


Resolvendo problemas no pai

Quais problemas sobraram para o pai resolver?

- Talvez o filho direito seja **vermelho** (não é esquerdista)
- Só pode ter acontecido porque a cor **vermelha** subiu

Se o filho esquerdo for **vermelho**, basta subir a cor

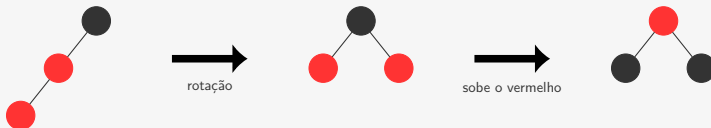


```
1  /* corrige a árvore */
2  if (ehVermelho(raiz->dir) && ehPreto(raiz->esq)) ←
3      raiz = rotaciona_para_esquerda(raiz);
4  if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) ←
5      raiz = rotaciona_para_direita(raiz);
6  if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) ←
7      sobe_vermelho(raiz); ←
```

Resolvendo problemas no pai

Quais problemas sobraram para o pai resolver?

- Talvez o filho esquerdo seja **vermelho**
- E o neto mais a esquerda seja **vermelho**



```
1  /* corrige a árvore */  
2  if (ehVermelho(raiz->dir) && ehPreto(raiz->esq)) ←  
3      raiz = rotaciona_para_esquerda(raiz);  
4  if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) ←  
5      raiz = rotaciona_para_direita(raiz); ←  
6  if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) ←  
7      sobe_vermelho(raiz); ←
```

Inserção - Implementação

```
1 p_no inserir_rec(p_no raiz, int chave) {
2     p_no novo;
3     if (raiz == NULL) {
4         novo = malloc(sizeof(struct no));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         novo->cor = VERMELHO;
8         return novo;
9     }
10    if (chave < raiz->chave)
11        raiz->esq = inserir_rec(raiz->esq, chave);
12    else
13        raiz->dir = inserir_rec(raiz->dir, chave);
14    /* corrige a árvore */
15    if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
16        raiz = rotaciona_para_esquerda(raiz);
17    if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
18        raiz = rotaciona_para_direita(raiz);
19    if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
20        sobe_vermelho(raiz);
21    return raiz;
22 }
```

Remoção

É possível fazer remoções em árvores **rubro-negras**

- Mas não veremos aqui...

A ideia é basicamente a mesma:

- encontrar operações que corrijam a árvore
- operações locais que mantêm as propriedades globais

Sugestão de leitura:

- Sedgewick e Wayne, Algorithms, 4th Edition, Addison-Wesley Professional, 2011.
- Cormen, Leiserson, Rivest e Stein, Introduction to Algorithms, Third Edition, MIT Press, 2009.

Rubro-Negras - Conclusão

As árvores **rubro-negras** esquerdistas suportam as seguintes operações:

- Busca
- Inserção
- Remoção

todas em tempo $O(\lg n)$

É uma variante da árvore **rubro-negra** com menos operações para corrigir a árvore na inserção e na remoção

Árvores **rubro-negras** são usadas como a árvore padrão no C++, no JAVA e no kernel do Linux

Outras árvores balanceadas

Existem também outras ABBs balanceadas:

- Uma árvore balanceada é uma árvore com altura $O(\lg n)$

Árvores AVL:

- A altura das subárvores pode variar de no máximo 1
- Tem altura $O(\lg n)$

ABB aleatorizada:

- Decide de maneira aleatória como inserir o nó
 - inserção normal como folha
 - inserção na raiz - rotações trazem o nó até a raiz
- Altura “média” (esperada): $O(\lg n)$

Árvores Splay:

- Sobe os nós no caminho da busca/inserção
- Nós mais acessados ficam mais próximos da raiz
- Não é balanceada, mas o custo de m inserções/buscas em uma árvore Splay com n nós é $O((n + m) \lg(n + m))$