MC-202
Árvores Balanceadas

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

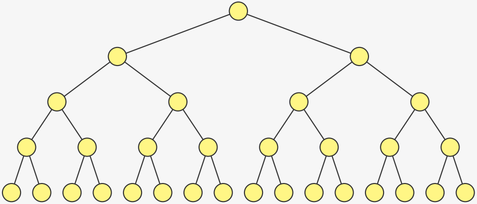
2º semestre/2023

Eficiência da busca, inserção e remoção

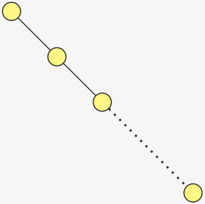
Qual é o tempo da busca, inserção e remoção em ABBs?

- depende da altura da árvore...

Ex: 31 nós



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



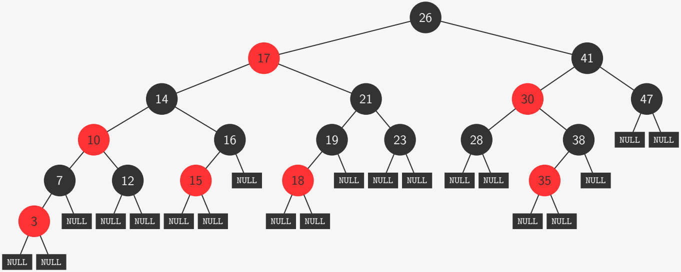
Pior árvore: n

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 $\mathrm{O}(\lg n)$

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, $h \leq 2 \lg(n + 1) = \mathrm{O}(\lg n)$

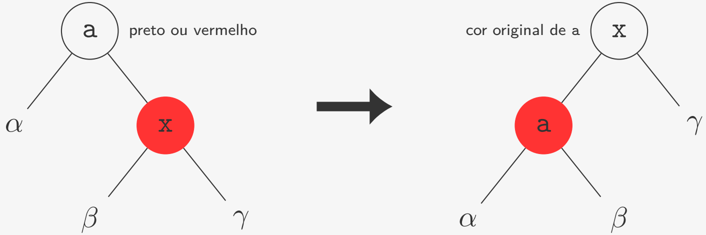
Alterando a Struct e testando a cor

```c
enum cor {VERMELHO, PRETO};

typedef struct no *p_no;

struct no {
  int chave;
  enum cor cor;
  p_no esq, dir;
};
```

```c
int ehVermelho(p_no x) {
  if (x == NULL)
    return 0;
  return x->cor == VERMELHO;
}
```

```c
int ehPreto(p_no x) {
  if (x == NULL)
    return 1;
  return x->cor == PRETO;
}
```

Rotação para a esquerda

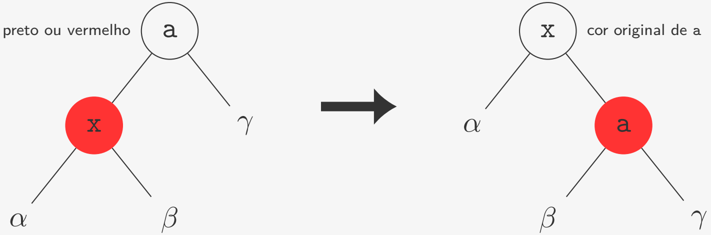



```
p_no rotaciona_para_esquerda(p_no raiz) {
  p_no x = raiz->dir;
  raiz->dir = x->esq;
  x->esq = raiz;
  x->cor = raiz->cor;
  raiz->cor = VERMELHO;
  return x;
}
```

Note que a rotação:

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

Rotação para a direita

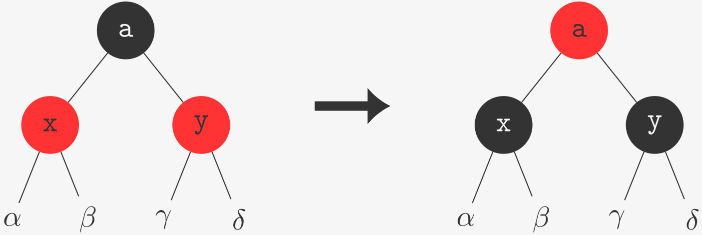



```
p_no rotaciona_para_direita(p_no raiz) {
  p_no x = raiz->esq;
  raiz->esq = x->dir;
  x->dir = raiz;
  x->cor = raiz->cor;
  raiz->cor = VERMELHO;
  return x;
}
```

Note que a rotação:

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

Subindo a cor




```c
void sobe_vermelho(p_no raiz) {
  raiz->cor = VERMELHO;
  raiz->esq->cor = PRETO;
  raiz->dir->cor = PRETO;
}
```

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

```
p_no inserir_rec(p_no raiz, int chave) {
  p_no novo;
  if (raiz == NULL) {
    novo = malloc(sizeof(struct no));
    novo->esq = novo->dir = NULL;
    novo->chave = chave;
    novo->cor = VERMELHO;
    return novo;
  }
  if (chave < raiz->chave)
    raiz->esq = inserir_rec(raiz->esq, chave);
  else
    raiz->dir = inserir_rec(raiz->dir, chave);
  /* corrige a árvore */
  return raiz;
}

p_no inserir(p_no raiz, int chave) {
  raiz = inserir_rec(raiz, chave);
  raiz->cor = PRETO;
  return raiz;
}
```

Mantém a raiz preta

Corrigindo

- Iremos corrigir cada propriedade
  - supomos que as subárvores esquerda e direita já satisfazem todas propriedade, 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**:

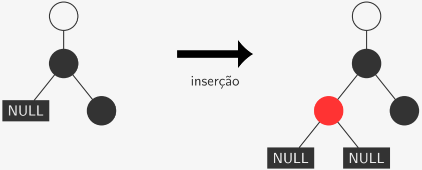
```
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
  raiz = rotaciona_para_direita(raiz);
```

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

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

Inserção — Caso 1

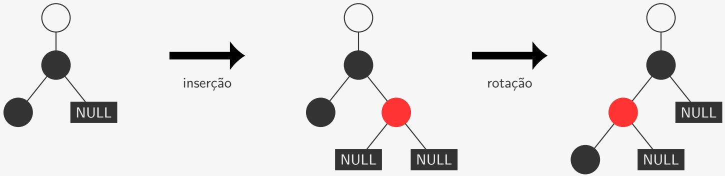
- Inserimos no filho esquerdo
- Nó atual é **preto**
  - não sabemos a cor do seu pai
  - nem se ele é o filho esquerdo ou direito
- Filho direito é **preto** (tem que ser — por que?)




```c
/* corrige a árvore */
if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
  raiz = rotaciona_para_esquerda(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
  raiz = rotaciona_para_direita(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
  sobe_vermelho(raiz);
```

Inserção — Caso 2

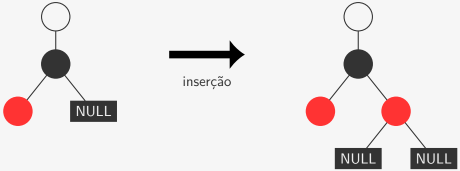
- Inserimos no filho direito
- Nó atual é **preto**
  - não sabemos a cor do seu pai
  - nem se ele é o filho esquerdo ou direito
- Filho esquerdo é **preto**




```c
/* corrige a árvore */
if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
  raiz = rotaciona_para_esquerda(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
  raiz = rotaciona_para_direita(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
  sobe_vermelho(raiz);
```

Inserção — Caso 3

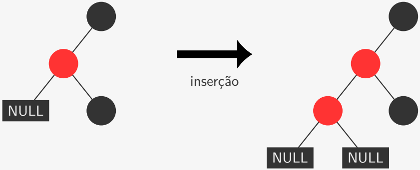
- Inserimos no filho direito
- Nó atual é **preto**
  - não sabemos a cor do seu pai
  - nem se ele é o filho esquerdo ou direito
- Filho esquerdo é **vermelho**




```
/* corrige a árvore */
if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
  raiz = rotaciona_para_esquerda(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
  raiz = rotaciona_para_direita(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
  sobe_vermelho(raiz);
```

Inserção — Caso 4

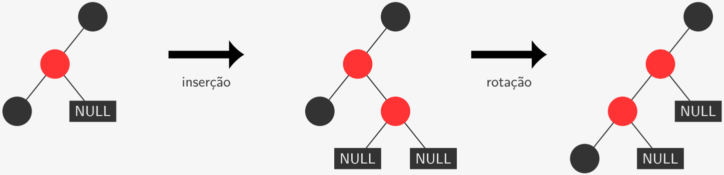
- Inserimos no filho esquerdo
- Nó atual é **vermelho**
  - seu pai é **preto** (ele não é a raiz — por que?)
  - é o filho esquerdo (por que?)




```
/* corrige a árvore */
if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
  raiz = rotaciona_para_esquerda(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
  raiz = rotaciona_para_direita(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
  sobe_vermelho(raiz);
```

Inserção — Caso 5

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



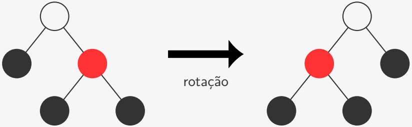

```
/* corrige a árvore */
if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
  raiz = rotaciona_para_esquerda(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
  raiz = rotaciona_para_direita(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
  sobe_vermelho(raiz);
```

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



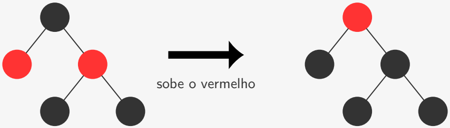

```c
/* corrige a árvore */
if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
  raiz = rotaciona_para_esquerda(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
  raiz = rotaciona_para_direita(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
  sobe_vermelho(raiz);
```

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

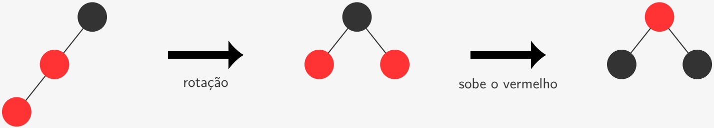



```
/* corrige a árvore */
if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
  raiz = rotaciona_para_esquerda(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
  raiz = rotaciona_para_direita(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
  sobe_vermelho(raiz);
```

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**




```c
/* corrige a árvore */
if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
  raiz = rotaciona_para_esquerda(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
  raiz = rotaciona_para_direita(raiz);
if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
  sobe_vermelho(raiz);
```

Inserção — Implementação

```
p_no inserir_rec(p_no raiz, int chave) {
  p_no novo;
  if (raiz == NULL) {
    novo = malloc(sizeof(struct no));
    novo->esq = novo->dir = NULL;
    novo->chave = chave;
    novo->cor = VERMELHO;
    return novo;
  }
  if (chave < raiz->chave)
    raiz->esq = inserir_rec(raiz->esq, chave);
  else
    raiz->dir = inserir_rec(raiz->dir, chave);
  /* corrige a árvore */
  if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
    raiz = rotaciona_para_esquerda(raiz);
  if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
    raiz = rotaciona_para_direita(raiz);
  if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
    sobe_vermelho(raiz);
  return raiz;
}
```

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 $\mathrm{O}(\lg n)$

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 $\mathrm{O}(\lg n)$

Árvores AVL:

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

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): $\mathrm{O}(\lg n)$

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 $\mathrm{O}((n+m)\lg(n+m))$