Estruturas de Dados

Árvores Rubro-Negras

Aula 08

Prof. Felipe A. Louza

UFU

Roteiro

1. Árvores Rubro-Negras
2. Busca
3. Inserção
4. Remoção
5. Outras Árvores Balanceadas
6. Referências

Roteiro

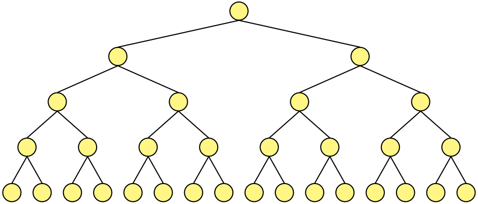
1. Árvores Rubro-Negras
2. Busca
3. Inserção
4. Remoção
5. Outras Árvores Balanceadas
6. Referências

Eficiência da busca, inserção e remoção

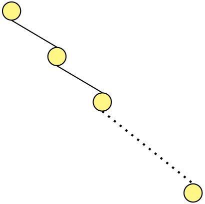
Qual é o tempo da busca, inserção e remoção em ABBs?

- depende da altura da árvore...

Ex: 31 nós



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



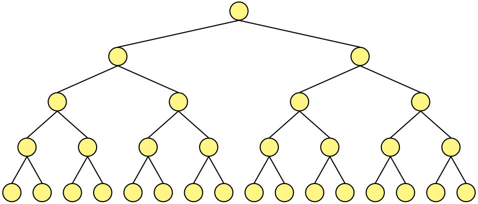
Pior árvore: n

Eficiência da busca, inserção e remoção

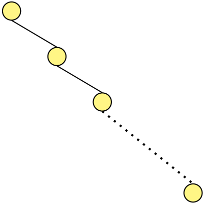
Qual é o tempo da busca, inserção e remoção em ABBs?

- depende da altura da árvore...

Ex: 31 nós



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

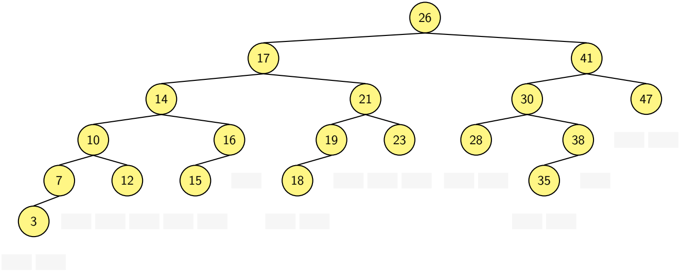


Pior árvore: n

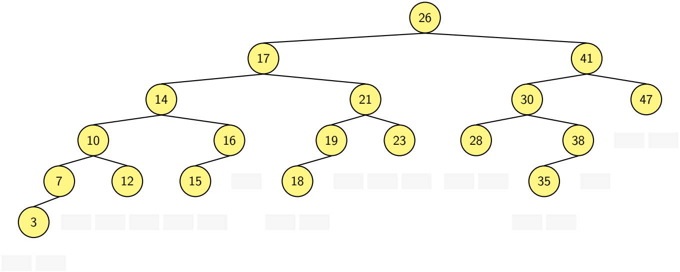
Veremos uma árvore **balanceada**

- Não é a melhor árvore possível, mas é "quase"
- Operações em $\mathrm{O}(\lg n)$

Árvores Rubro-Negras à Esquerda

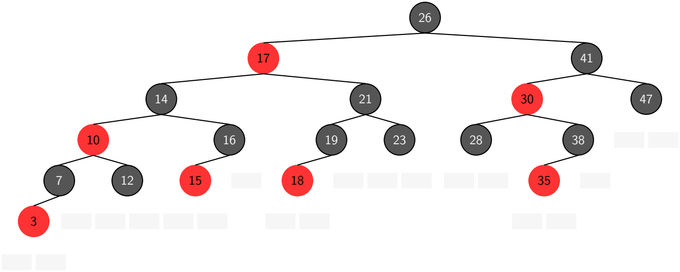


Árvores Rubro-Negras à Esquerda



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

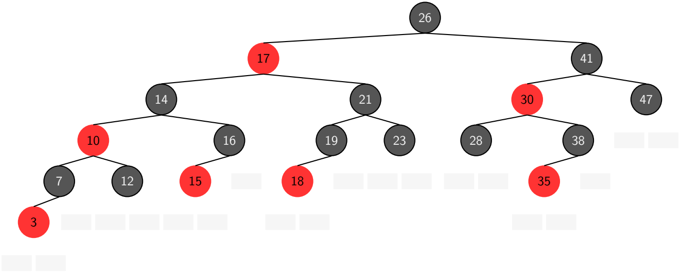
Árvores Rubro-Negras à Esquerda



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

1. Todo nó é ou **vermelho** ou **preto**,

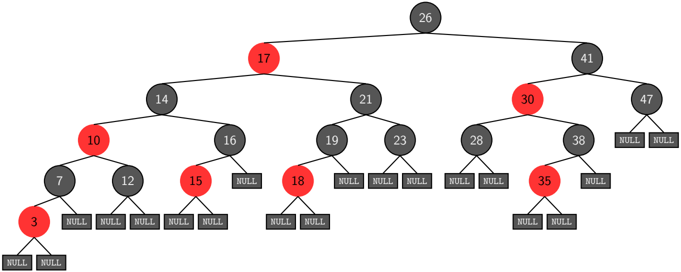
Árvores Rubro-Negras à Esquerda



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

1. Todo nó é ou **vermelho** ou **preto**,
2. A raiz é **preta**

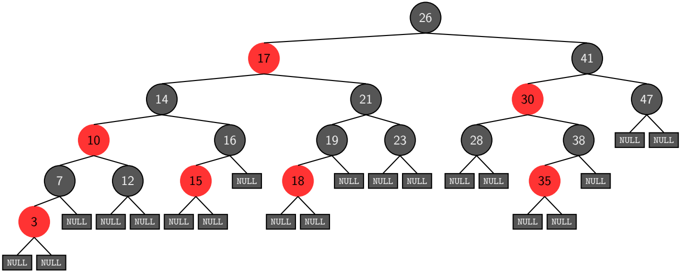
Árvores Rubro-Negras à Esquerda



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

1. Todo nó é ou **vermelho** ou **preto**,
2. A raiz é **preta**
3. As folhas (agora) são `NULL` e tem cor **preta**

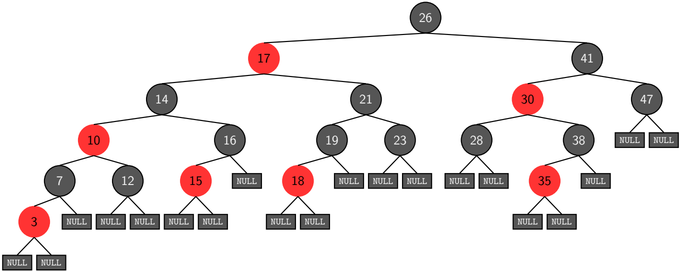
Árvores Rubro-Negras à Esquerda



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

1. Todo nó é ou **vermelho** ou **preto**,
2. A raiz é **preta**
3. As folhas (agora) são `NULL` e tem cor **preta**
4. Se um nó é **vermelho**, seus dois filhos são **pretos**

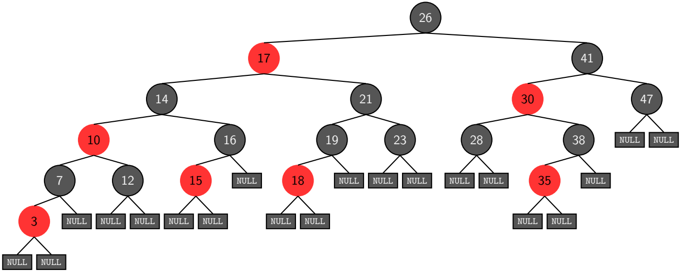
Árvores Rubro-Negras à Esquerda



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

1. Todo nó é ou **vermelho** ou **preto**,
2. A raiz é **preta**
3. As folhas (agora) 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, à esquerda)

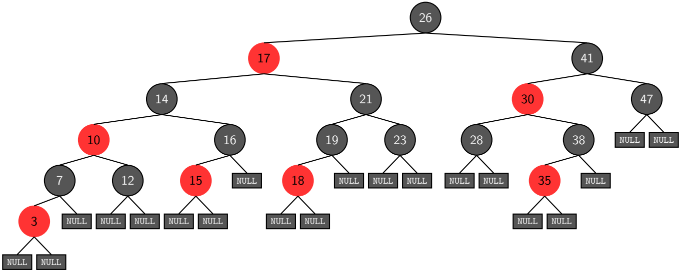
Árvores Rubro-Negras à Esquerda



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

1. Todo nó é ou **vermelho** ou **preto**,
2. A raiz é **preta**
3. As folhas (agora) 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, à esquerda)
5. Em cada nó, todo caminho dele para uma de suas folhas tem a mesma quantidade de nós **pretos** (não contamos o próprio nó)

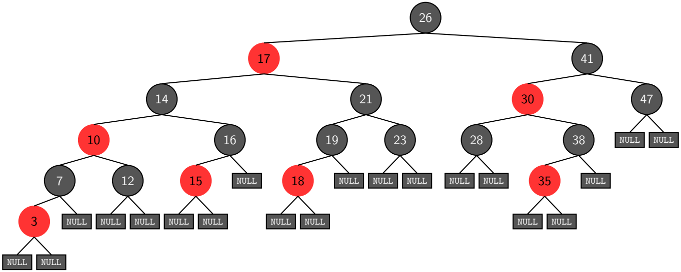
Árvores Rubro-Negras à Esquerda



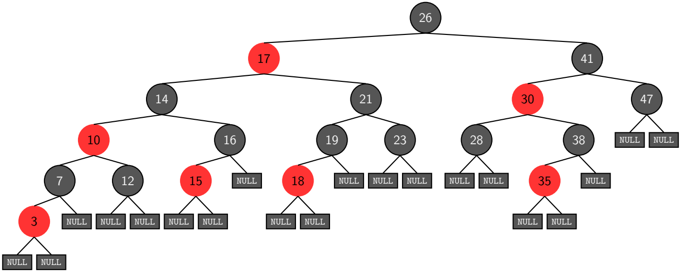
Uma árvore **rubro-negra** à esquerda é uma ABB tal que:

1. Todo nó é ou **vermelho** ou **preto**,
2. A raiz é **preta**
3. As folhas (agora) 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, à esquerda)
5. Em cada nó, todo caminho dele para uma de suas folhas tem a mesma quantidade de nós **pretos** (não contamos o próprio nó)
   - É a altura-**negra** do nó

Árvores Rubro-Negras à Esquerda

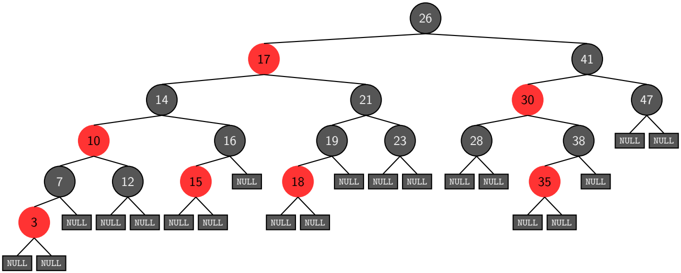


Árvores Rubro-Negras à Esquerda



Seja *bh* a altura-**negra** da árvore.

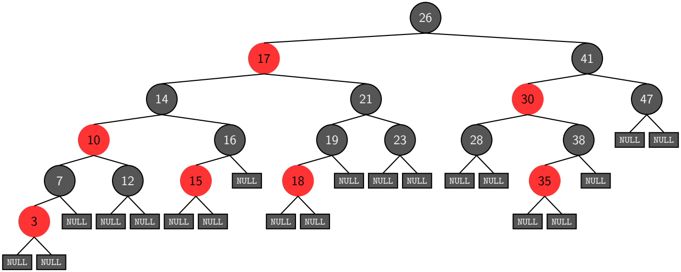
Árvores Rubro-Negras à Esquerda



Seja *bh* a altura-**negra** da árvore.

- A árvore tem pelo menos $2^{bh} - 1$ nós não nulos, $n \geq 2^{bh} - 1$ (*prova por indução*)

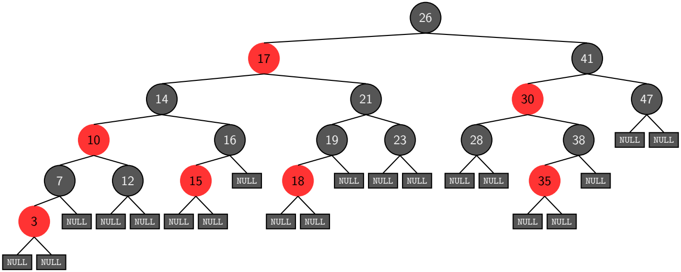
Árvores Rubro-Negras à Esquerda



Seja bh a altura-**negra** da árvore.

- A árvore tem pelo menos $2^{bh} - 1$ nós não nulos, $n \geq 2^{bh} - 1$ (*prova por indução*)
- A altura-**negra** bh é pelo menos metade da altura h, $bh \geq h/2$

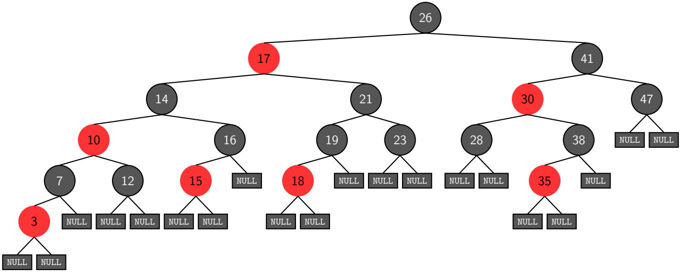
Árvores Rubro-Negras à Esquerda



Seja *bh* a altura-**negra** da árvore.

- A árvore tem pelo menos $2^{bh} - 1$ nós não nulos, $n \geq 2^{bh} - 1$ (*prova por indução*)
- A altura-**negra** *bh* é pelo menos metade da altura *h*, $bh \geq h/2$
  - Não existe nó **vermelho** com filho **vermelho**

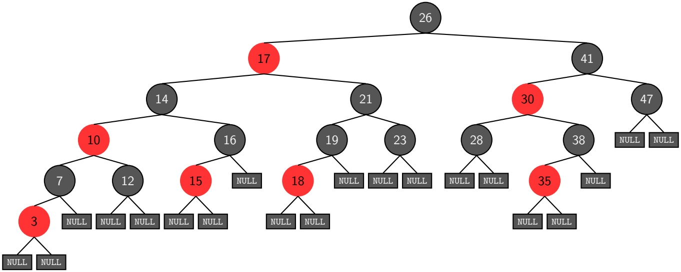
Árvores Rubro-Negras à Esquerda



Seja bh a altura-**negra** da árvore.

- A árvore tem pelo menos $2^{bh} - 1$ nós não nulos, $n \geq 2^{bh} - 1$ (*prova por indução*)
- A altura-**negra** bh é pelo menos metade da altura h, $bh \geq h/2$
  - Não existe nó **vermelho** com filho **vermelho**
  - O número de nós não nulos n é $n \geq 2^{bh} - 1 \geq 2^{h/2} - 1$

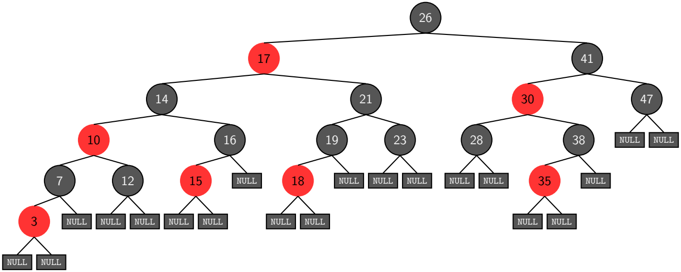
Árvores Rubro-Negras à Esquerda



Seja bh a altura-**negra** da árvore.

- A árvore tem pelo menos $2^{bh} - 1$ nós não nulos, $n \geq 2^{bh} - 1$ (*prova por indução*)
- A altura-**negra** bh é pelo menos metade da altura h, $bh \geq h/2$
  - 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, $h \leq 2\lg(n+1)$

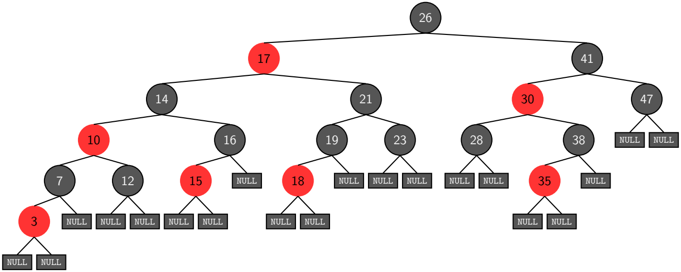
Árvores Rubro-Negras à Esquerda



Seja bh a altura-**negra** da árvore.

- A árvore tem pelo menos $2^{bh} - 1$ nós não nulos, $n \geq 2^{bh} - 1$ (*prova por indução*)
- A altura-**negra** bh é pelo menos metade da altura h, $bh \geq h/2$
  - 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, $h \leq 2 \lg(n + 1)$
- A altura h da árvore com n nós é no máximo $2 \lg(n + 1) = \mathrm{O}(\log n)$

Árvores Rubro-Negras à Esquerda



Seja bh a altura-**negra** da árvore.

- A árvore tem pelo menos $2^{bh} - 1$ nós não nulos, $n \geq 2^{bh} - 1$ (*prova por indução*)
- A altura-**negra** bh é pelo menos metade da altura h, $bh \geq h/2$
  - 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, $h \leq 2\lg(n+1)$
- A altura h da árvore com n nós é no máximo $2\lg(n+1) = \mathrm{O}(\log n)$
  - No pior caso h é duas vezes maior que a altura da "melhor" árvore

TAD - Interface

Cliente → Interface → Implementação

arn.h

```c
#ifndef ARN_H
#define ARN_H

enum Cor {VERMELHO, PRETO};

//Dados
typedef struct No {
  int chave;
  enum Cor cor;
  struct No *esq, *dir;
} No;
//Funções
No* criar_arvore();
void destruir_arvore(No **p);

void imprimir_arvore(No *p, int h);
void imprimir_inordem(No *p);

No*  inserir(No *p, int chave);
void remover(No **p, int chave);

#endif
```

- Praticamente a mesma interface vista para ABBs.

ARN - Novas funções

Cliente → Interface → Implementação

arn.c

```c
int ehVermelho(No *x) {
  if (x == NULL)
    return 0;
return x->cor == VERMELHO;
}
```

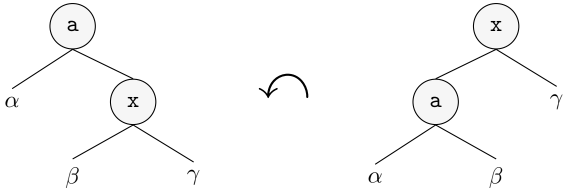
```c
int ehPreto(No *x) {
  if (x == NULL)
    return 1;
return x->cor == PRETO;
}
```

- Testando a cor de um nó.

Rotações em árvores

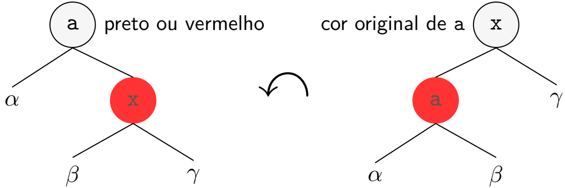
Uma rotação é uma operação que altera a estrutura de uma ABB sem interferir na ordem das chaves.

- não estraga as propriedades da árvore



Rotações são utilizadas em diferentes árvores balanceadas.

Rotação para a esquerda



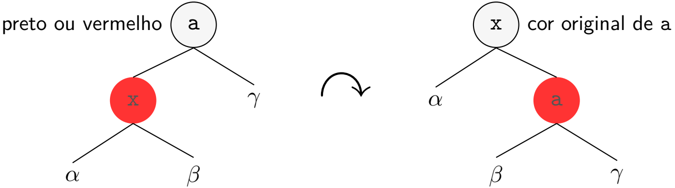

```
No* rotaciona_para_esquerda(No *p) { //retorna a raiz da nova sub-árvore
  No* x = p->dir;
  p->dir = x->esq;
  x->esq = p;
  x->cor = p->cor;
  p->cor = VERMELHO;
return x; //a altura-negra de x continua a mesma
}
```

Note que a rotação não estraga a propriedade da altura-**negra** de x

---

Vamos usar essa função para corrigir a árvore após inserções.

Rotação para a direita

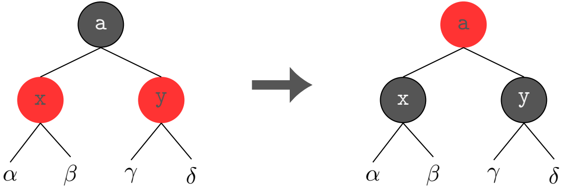



```c
No* rotaciona_para_direita(No *p) { //retorna a raiz da nova sub-árvore
  No* x = p->esq;
  p->esq = x->dir;
  x->dir = p;
  x->cor = p->cor;
  p->cor = VERMELHO;
return x; //a altura-negra de x continua a mesma
}
```

Note que a rotação não estraga a propriedade da altura-**negra** de x

Vamos usar essa função para corrigir a árvore após inserções.

Subindo a cor




```c
void sobe_vermelho(No *p) { //caso tenha 2 filhos vermelhos
  p->cor = VERMELHO;
  p->esq->cor = PRETO;
  p->dir->cor = PRETO;
}
```

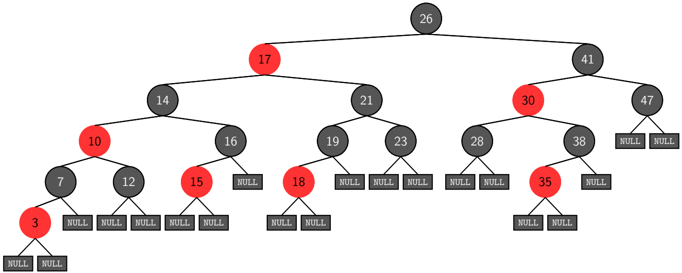
Subir a cor não estraga a propriedade da altura negra

- mas pode pintar a raiz de vermelho

Roteiro

1. Árvores Rubro-Negras
2. Busca
3. Inserção
4. Remoção
5. Outras Árvores Balanceadas
6. Referências

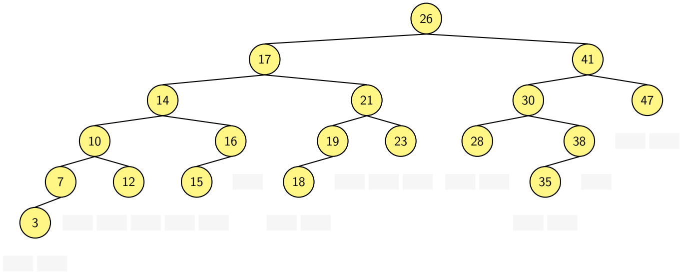
ARN - Busca por um valor



arn.c

```c
No* buscar(No *p, int chave) {
  if (p == NULL || chave == p->chave)
    return p;
  if (chave < p->chave)
    return buscar(p->esq, chave);
  else
    return buscar(p->dir, chave);
}
```

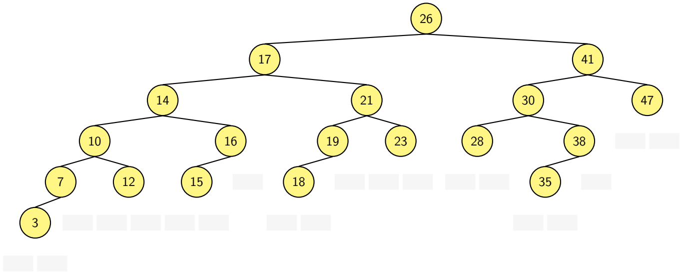
ARN - Busca por um valor



arn.c

```c
No* buscar(No *p, int chave) {
  if (p == NULL || chave == p->chave)
    return p;
  if (chave < p->chave)
    return buscar(p->esq, chave);
  else
    return buscar(p->dir, chave);
}
```

ARN - Busca por um valor



arn.c

```c
No* buscar(No *p, int chave) {
  if (p == NULL || chave == p->chave)
    return p;
  if (chave < p->chave)
    return buscar(p->esq, chave);
  else
    return buscar(p->dir, chave);
}
```

- Custo computacional: $O(\lg n)$

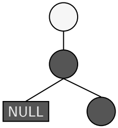
Roteiro

1. Árvores Rubro-Negras
2. Busca
3. Inserção
4. Remoção
5. Outras Árvores Balanceadas
6. Referências

Inserindo

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

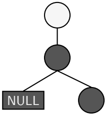
```
No* inserir_rec(No *p, int chave) {
  if (p == NULL) {
    No* novo = malloc(sizeof(No));
    novo->esq = novo->dir = NULL;
    novo->chave = chave;
    novo->cor = VERMELHO; //mantém a propriedade da altura negra
    return novo;
  }
  if (chave < p->chave) p->esq = inserir_rec(p->esq, chave);
  else p->dir = inserir_rec(p->dir, chave);
  /* corrige a árvore */
return p;
}
```




```
No* inserir(No *raiz, int chave) {
  raiz = inserir_corrigindo(raiz, chave);
  raiz->cor = PRETO; //pode ser que a raiz fique vermelha (corrigimos)
return raiz;
}
```

Inserção - Caso 1

- Nó atual é **preto** e vamos inserir no filho esquerdo
  - 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?)

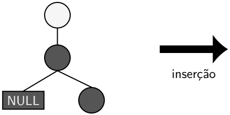



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: não faz nada.

Inserção - Caso 1

- Nó atual é **preto** e vamos inserir no filho esquerdo
  - 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?)

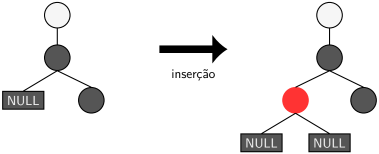



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: não faz nada.

Inserção - Caso 1

- Nó atual é **preto** e vamos inserir no filho esquerdo
  - 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?)

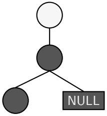



```c
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: não faz nada.

Inserção - Caso 2

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

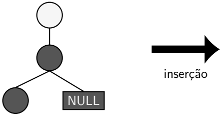



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: rotaciona para a esquerda.

Inserção - Caso 2

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

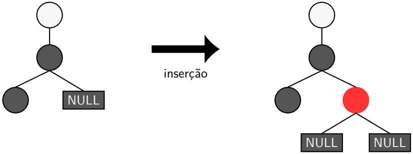



```c
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: rotaciona para a esquerda.

Inserção - Caso 2

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

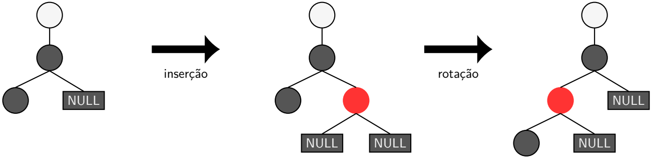



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: rotaciona para a esquerda.

Inserção - Caso 2

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

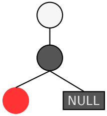



```c
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: rotaciona para a esquerda.

Inserção - Caso 3

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

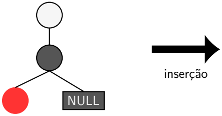



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: sobe_vermelho, pode gerar problema no pai (resolvido depois).

Inserção - Caso 3

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

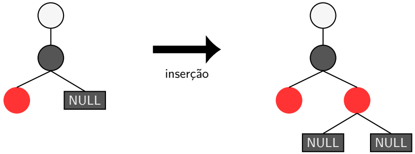



```c
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: sobe_vermelho, pode gerar problema no pai (resolvido depois).

Inserção - Caso 3

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

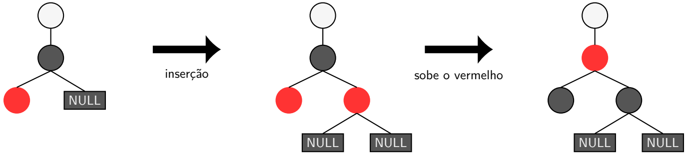



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: sobe_vermelho, pode gerar problema no pai (resolvido depois).

Inserção - Caso 3

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

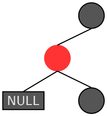



```c
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: sobe_vermelho, pode gerar problema no pai (resolvido depois).

Inserção - Caso 4

- Nó atual é **vermelho** e vamos inserir no filho esquerdo
  - seu pai é **preto** (ele não é a raiz - por que?)
  - é o filho esquerdo (por que?)
- Filho esquerdo é `NULL`

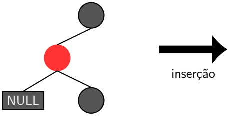



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: o código **não arruma** o problema. Será resolvido depois pelo nó pai.

Inserção - Caso 4

- Nó atual é **vermelho** e vamos inserir no filho esquerdo
  - seu pai é **preto** (ele não é a raiz - por que?)
  - é o filho esquerdo (por que?)
- Filho esquerdo é `NULL`

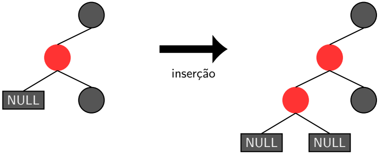



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: o código **não arruma** o problema. Será resolvido depois pelo nó pai.

Inserção - Caso 4

- Nó atual é **vermelho** e vamos inserir no filho esquerdo
  - seu pai é **preto** (ele não é a raiz - por que?)
  - é o filho esquerdo (por que?)
- Filho esquerdo é `NULL`

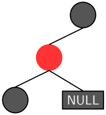



```c
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: o código **não arruma** o problema. Será resolvido depois pelo nó pai.

Inserção - Caso 5

- Nó atual é **vermelho** e vamos inserir no filho direito
  - seu pai é **preto** (ele não é a raiz)
  - é o filho esquerdo
- Filho direito é `NULL`

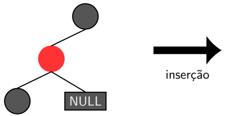



```c
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: rotaciona para esquerda. Novamente, o problema será resolvido depois pelo nó pai.

Inserção - Caso 5

- Nó atual é **vermelho** e vamos inserir no filho direito
  - seu pai é **preto** (ele não é a raiz)
  - é o filho esquerdo
- Filho direito é `NULL`

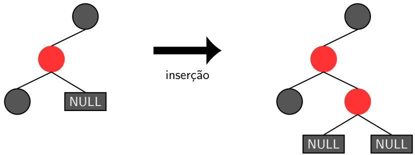



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: rotaciona para esquerda. Novamente, o problema será resolvido depois pelo nó pai.

Inserção - Caso 5

- Nó atual é **vermelho** e vamos inserir no filho direito
  - seu pai é **preto** (ele não é a raiz)
  - é o filho esquerdo
- Filho direito é `NULL`

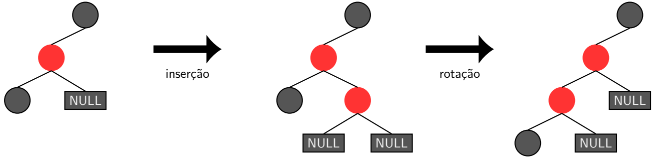



```
/* corrige a árvore */
if (ehVermelho(p->dir) && ehPreto(p->esq))
  p = rotaciona_para_esquerda(p);
if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
  p = rotaciona_para_direita(p);
if (ehVermelho(p->esq) && ehVermelho(p->dir))
  sobe_vermelho(p);
```

Corrigir árvore: rotaciona para esquerda. Novamente, o problema será resolvido depois pelo nó pai.

Inserção - Caso 5

- Nó atual é **vermelho** e vamos inserir no filho direito
  - seu pai é **preto** (ele não é a raiz)
  - é o filho esquerdo
- Filho direito é `NULL`




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

Corrigir árvore: rotaciona para esquerda. Novamente, o problema será resolvido depois pelo nó pai.

Resolvendo problemas no pai (parte 1)

Quais problemas sobraram para o pai resolver?

- Talvez o filho direito seja **vermelho** (não é à esquerda)
- Só pode ter acontecido porque a cor **vermelha** subiu (caso 3)

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

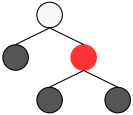
Corrigir árvore: rotaciona para esquerda.

Resolvendo problemas no pai (parte 1)

Quais problemas sobraram para o pai resolver?

- Talvez o filho direito seja **vermelho** (não é à esquerda)
- Só pode ter acontecido porque a cor **vermelha** subiu (caso 3)

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



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

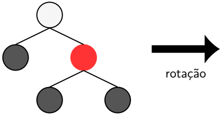
Corrigir árvore: rotaciona para esquerda.

Resolvendo problemas no pai (parte 1)

Quais problemas sobraram para o pai resolver?

- Talvez o filho direito seja **vermelho** (não é à esquerda)
- Só pode ter acontecido porque a cor **vermelha** subiu (caso 3)

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




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

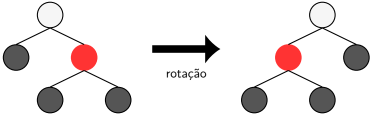
Corrigir árvore: rotaciona para esquerda.

Resolvendo problemas no pai (parte 1)

Quais problemas sobraram para o pai resolver?

- Talvez o filho direito seja **vermelho** (não é à esquerda)
- Só pode ter acontecido porque a cor **vermelha** subiu (caso 3)

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




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

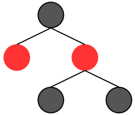
Corrigir árvore: rotaciona para esquerda.

Resolvendo problemas no pai (parte 2)

Quais problemas sobraram para o pai resolver?

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

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



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

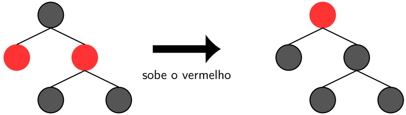
Corrigir árvore: subir_vermelho.

Resolvendo problemas no pai (parte 2)

Quais problemas sobraram para o pai resolver?

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

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




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

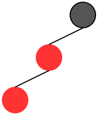
Corrigir árvore: subir_vermelho.

Resolvendo problemas no pai (parte 3)

Quais problemas sobraram para o pai resolver?

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

Situação causada pelo Caso 4.



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

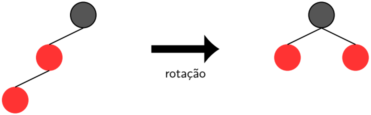
Corrigir árvore: rotaciona para direita e subir_vermelho.

Resolvendo problemas no pai (parte 3)

Quais problemas sobraram para o pai resolver?

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

Situação causada pelo Caso 4.




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

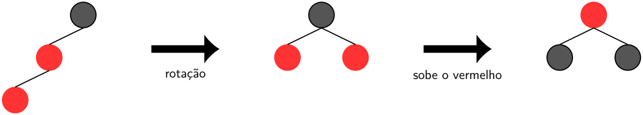
Corrigir árvore: rotaciona para direita e subir_vermelho.

Resolvendo problemas no pai (parte 3)

Quais problemas sobraram para o pai resolver?

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

Situação causada pelo Caso 4.




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

Corrigir árvore: rotaciona para direita e subir_vermelho.

Inserção - Implementação

```c
No* inserir_corrigindo(No *p, int chave) {
  No* novo;
  if (p == NULL) {
    novo = malloc(sizeof(No));
    novo->esq = novo->dir = NULL;
    novo->chave = chave;
    novo->cor = VERMELHO;
    return novo;
  }
  if (chave < p->chave) p->esq = inserir_corrigindo(p->esq, chave);
  else p->dir = inserir_corrigindo(p->dir, chave);
  /* corrige a árvore */
  if (ehVermelho(p->dir) && ehPreto(p->esq))
    p = rotaciona_para_esquerda(p);
  if (ehVermelho(p->esq) && ehVermelho(p->esq->esq))
    p = rotaciona_para_direita(p);
  if (ehVermelho(p->esq) && ehVermelho(p->dir))
    sobe_vermelho(p);
return p;
}
```

Cliente

exemplo1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "arn.h"
```

```c
int main() {
  int i, v[10] = {8, 3, 1, 7, 13, 10, 14, 12, 4, 5};
  No *T = criar_arvore();
  for (i = 0; i < 10; i++)
    T = inserir(T, v[i]);
  imprimir_arvore(T, 0);
  imprimir_inordem(T);
  printf("\n");
  destruir_arvore(&T);
return 0;
}
```

Makefile

Vamos usar o Makefile para compilar:

```
exemplo1: exemplo1.c arn.o
  gcc $^ -o $@
```

Vamos executar:

```
$ ./exemplo1
-- 14 (1)
- 13 (1)
-- 12 (1)
--- 10 (0)
 8 (1)
-- 7 (1)
- 5 (1)
--- 4 (1)
-- 3 (0)
--- 1 (1)
1 3 4 5 7 8 10 12 13 14
```

Testar `inserir_rec()`, sem correção.

Makefile

Vamos usar o Makefile para compilar:

```
exemplo1: exemplo2.c arn.o
  gcc $^ -o $@
```

Vamos executar:

```
$ ./exemplo2
- 6 (1)
 5 (1)
-- 4 (1)
--- 3 (0)
- 2 (0)
-- 1 (1)
1 2 3 4 5 6
```

Outro exemplo.

Roteiro

1. Árvores Rubro-Negras
2. Busca
3. Inserção
4. Remoção
5. Outras Árvores Balanceadas
6. Referências

Remoção

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

- Mas não veremos aqui...

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

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

Realizamos a busca por um valor:

- removemos o nó correspondente
- voltamos (recursivamente) corrigindo a árvore.

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

Realizamos a busca por um valor:

- removemos o nó correspondente
- voltamos (recursivamente) corrigindo a árvore.

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** à esquerda suportam as seguintes operações:

- Busca
- Inserção
- Remoção

todas em tempo $O(\lg n)$

Rubro-Negras - Conclusão

As árvores **rubro-negras** à esquerda 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

Rubro-Negras - Conclusão

As árvores **rubro-negras** à esquerda 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`

Roteiro

1. Árvores Rubro-Negras
2. Busca
3. Inserção
4. Remoção
5. Outras Árvores Balanceadas
6. Referências

Outras árvores balanceadas

Existem também outras ABBs balanceadas:

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

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 $O(\lg n)$

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

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

Fim

Dúvidas?

Roteiro

1 Árvores Rubro-Negras

2 Busca

3 Inserção

4 Remoção

5 Outras Árvores Balanceadas

6 Referências

Referências

1. Materiais adaptados dos slides do Prof. Rafael C. S. Schouery, da Universidade Estadual de Campinas.
2. Feofiloff, Paulo. Algoritmos em linguagem C. Elsevier Brasil, 2009.