MC202 - Estruturas de Dados

Árvores Rubro-Negras

Emilio Francesquini
francesquini@ic.unicamp.br

Instituto de Computação - UNICAMP

Aulas 16,17 e 18 - maio de 2017

Disclaimer

- Esses slides foram preparados para um curso de Estrutura de Dados ministrado na UNICAMP
- Este material pode ser usado livremente desde que sejam mantidos os créditos dos autores e da instituição.
- Os exemplos apresentados aqui foram retirados do livro texto CLRS
- Alguns slides foram baseados nos slides preparados pela Profa. L. S. Assis

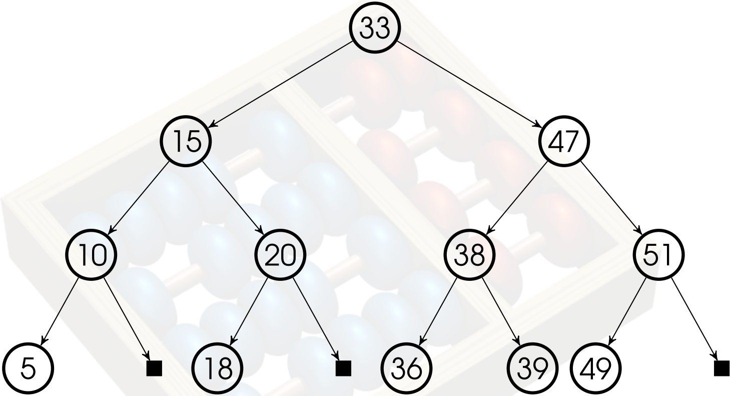
UNICAMP

Recap.: Árvores binárias de busca

- Uma árvore binária é chamada árvore binária de busca se para **cada nó p** vale que:
  - todo nó da sub-árvore esquerda de p tem chave menor que a chave de p;
  - todo nó da sub-árvore direita de p tem chave maior que a chave de p.

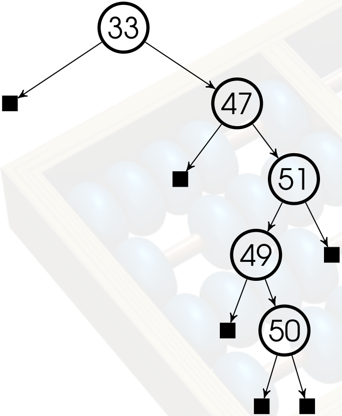
UNICAMP

Recap.: Árvores binárias de busca



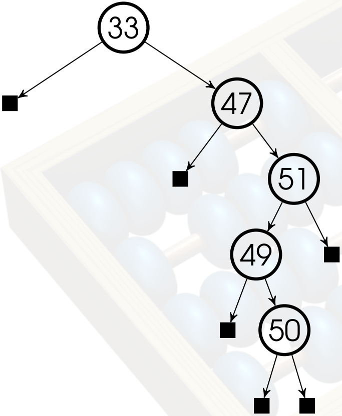
É uma **árvore de busca binária.**

Recap.: Árvores binárias de busca



É uma árvore binária de busca?

Recap.: Árvores binárias de busca



É uma árvore binária de busca?

Sim

Mas há algo diferente da anterior...

UNICAMP

Árvores Balanceadas de Busca

- Arvores de busca com garantias de altura máxima
- Em outras palavras, árvores balanceadas tem a garantia de que a sua altura é no máximo O(lg n)
- Árvores AVL e árvores **rubro-negras** são exemplos de árvores balanceadas de busca

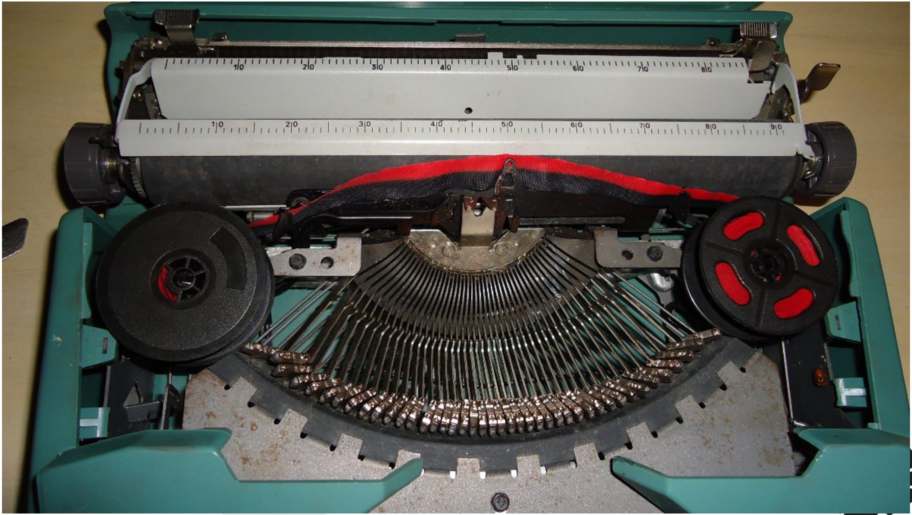
UNICAMP

Árvores Rubro-Negras

- **Rubro-Negras** são árvores de busca balanceadas
- Também chamadas de árvores vermelho-preto (do inglês *red-black trees*
- Receberam este nome em um artigo de Guibas e Sedgewick em 1978
  - Supostamente pois impressora apenas era capaz de imprimir vermelho e preto...

UNICAMP

Árvores Rubro-Negras



Árvores Rubro-Negras

- Uma árvore rubro-negra é uma árvore de busca binária, logo segue todas as regras:
  - todo nó da sub-árvore esquerda de um nó p tem chave menor que a chave de p;
  - todo nó da sub-árvore direita de um nó p tem chave maior que a chave de p.
- Além disto, cada nó de uma árvore rubro negra tem os seguintes campos:
  - **cor** – Indica se o nó é vermelho ou preto
  - **chave** (ou valor) – Conteúdo do nó
  - **dir**, **esq** – Sub-árvores direita e esquerda
  - **pai** – Apontador para o pai do nó
- Se o filho ou o pai de um nó não existir, o campo é preenchido com NULL

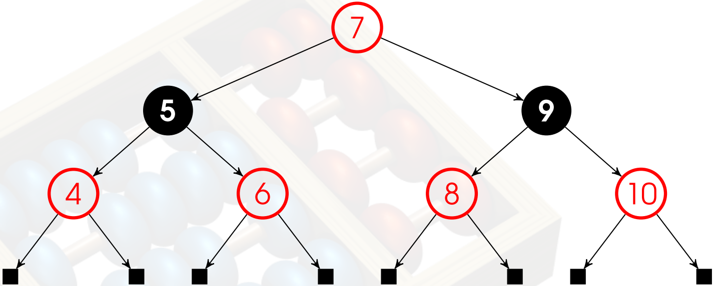
UNICAMP

Árvores Rubro-Negras

- Porém, uma árvore rubro-negra tem algumas regras adicionais:
  - Regra 1: Um nó é **vermelho** ou é **preto**
  - Regra 2: A raiz é **preta**
  - Regra 3: Toda folha (NULL) é **preta**
  - Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
  - Regra 5: Para cada nó p, todos os caminhos desde p até as folhas contêm o mesmo número de nós **pretos**

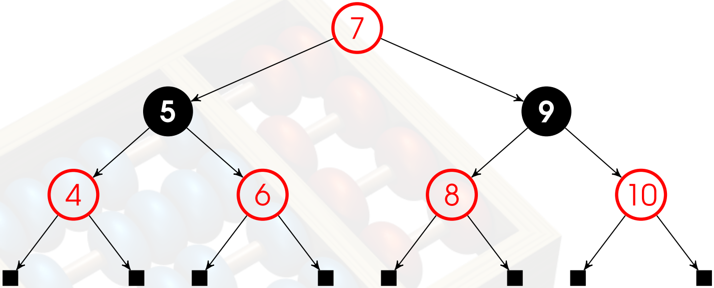
UNICAMP

Árvores Rubro-Negras - Reconhecendo 1



É uma árvore **rubro-negra**?

Árvores Rubro-Negras - Reconhecendo 1

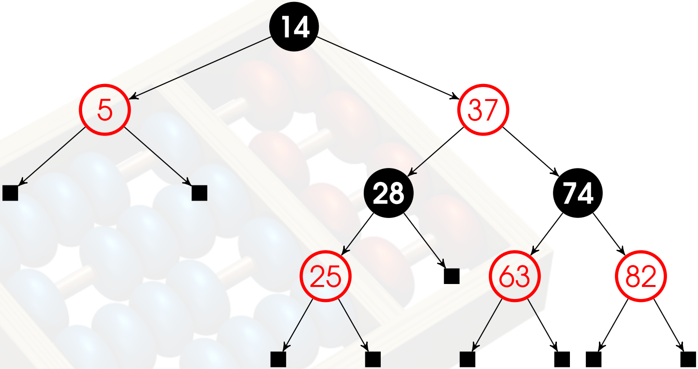


É uma árvore **rubro-negra**?

Não! Viola a Regra 2

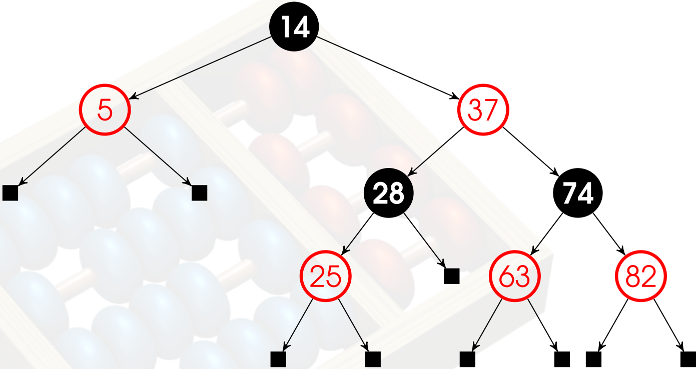
UNICAMP

Árvores Rubro-Negras - Reconhecendo 2



É uma árvore **rubro-negra**?

Árvores Rubro-Negras - Reconhecendo 2

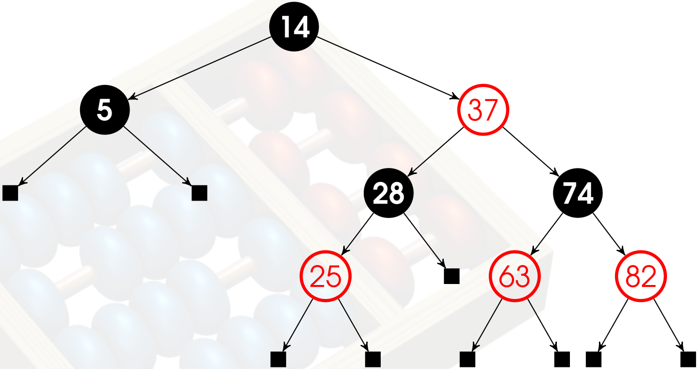


É uma árvore **rubro-negra**?

Não! Viola a Regra 5

UNICAMP

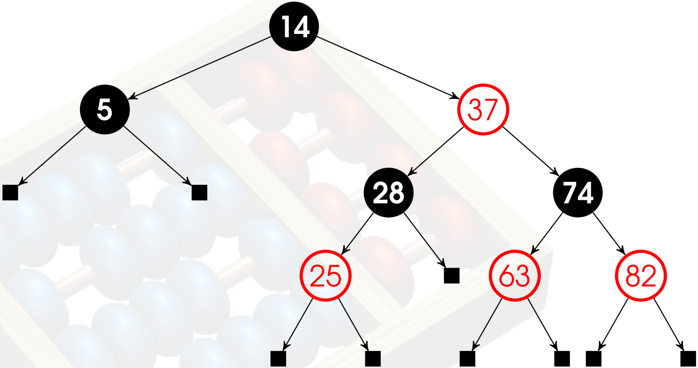
Árvores Rubro-Negras - Reconhecendo 3



É uma árvore **rubro-negra**?

UNICAMP

Árvores Rubro-Negras - Reconhecendo 3

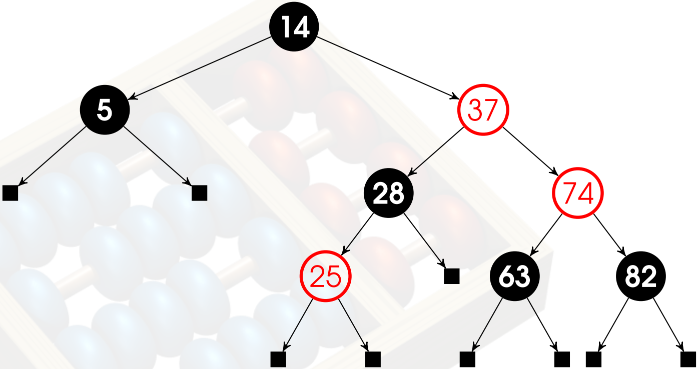


É uma árvore **rubro-negra**?

Sim!

UNICAMP

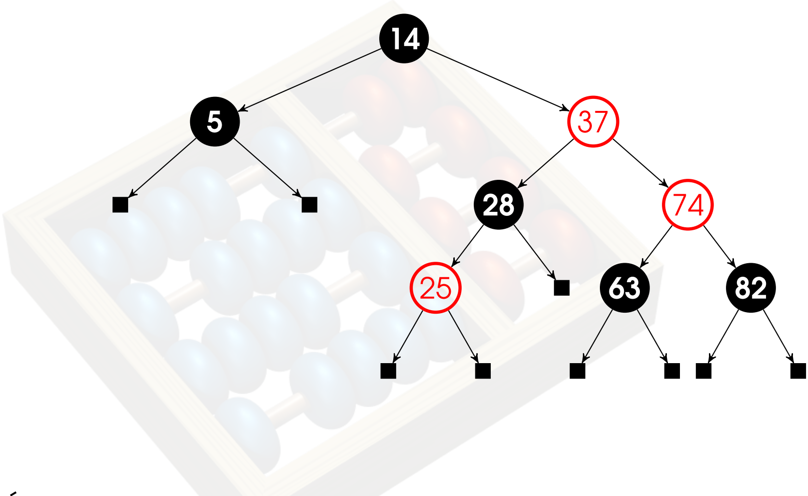
Árvores Rubro-Negras - Reconhecendo 4



É uma árvore **rubro-negra**?

UNICAMP

Árvores Rubro-Negras - Reconhecendo 4

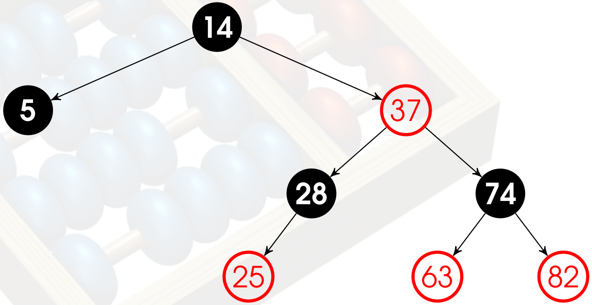


É uma árvore **rubro-negra**?

Não! Viola a Regra 4

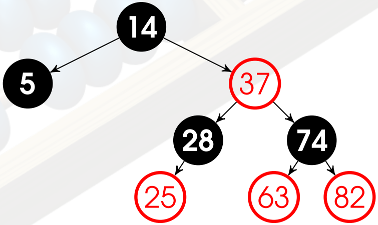
Árvores Rubro-Negras - Representação

Por simplicidade, no restante dos slides vamos representar as árvores omitindo as folhas nulas (pretas). Uma árvore válida seria então:



Árvores Rubro-Negras

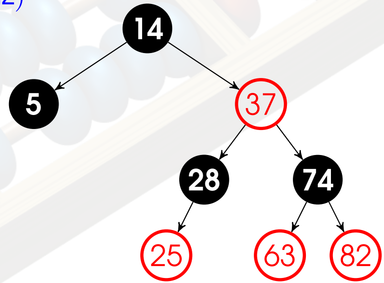
- Ok! Já vimos todas as regras de uma árvore rubro-negra. Mas qual a razão de tudo isso? Como isso nos ajuda?
- Restringindo a maneira que os nós podem ser coloridos do caminho da raíz até qualquer uma das suas folhas, as árvores rubro-negras:
  - **Garantem** que nenhum dos caminhos será maior que 2x o comprimento de qualquer outro
  - **Garantem** que a árvore é aproximadamente **balanceada**



Altura de preto

A **altura de preto** de um nó n é definida como o número de nós pretos (sem incluir o próprio n) visitados em qualquer caminho de n até as folhas.

- A altura de preto do nó n é denotada por $H_p(n)$
- Pela Regra 5, $H_p(n)$ é bem definida para todos os nós da árvore
- A altura de preto da árvore rubro-negra é definida como sendo a $H_p(\text{raíz})$



Lema 1 - Altura máxima

Lema 1: A altura máxima de uma árvore rubro-negra com **n** nós internos tem altura máxima de $2lg(n+1)$

- A prova pode ser feita por indução utilizando a H_p dos nós da árvore. Veja o Lema 13.1 do CLRS para a prova completa.

Corolário: As operações de *Busca, Mínimo, Máximo, Sucessor e Predecessor* podem ser efetuadas em tempo **O($lg(n)$)**

UNICAMP

Operações em árvores rubro-negras

Busca – A busca que estamos acostumados funciona sem modificações

Inserção e Remoção – Mantêm **apenas** as propriedades de árvores binárias de busca, mas não mantém as propriedades rubro-negras

Veja animação de operações em:
`http://tommikaikkonen.github.io/rbtree`

Comparação entre estruturas de dados

	Tempo Assintótico (limitante superior) *					
	Vetor	Vetor Ord.	Lista Ligada	Lista Ligada Ord.	Árv. Busca	Árv. Bal. Busca
Busca	n	lg(n)	n	n	n	lg(n)
Inserção	1	n	1	n	n	lg(n)
Remoção	n	n	1	1	n	lg(n)

* Esses são os tempos das operações nessas estruturas de dados tal qual elas foram apresentadas neste curso. Em alguns casos específicos é possível melhorar estes tempos. Ainda assim a árvore binária balanceada de busca continua sendo uma das estruturas mais eficientes dentre todas as que já vimos.

UNICAMP

Código do que temos até agora...

```
enum _CorNo {Vermelho, Preto};
typedef enum _CorNo CorNo;

struct _noArvRN {
    int chave;
    CorNo cor;
    struct _noArvRN *dir, *esq, *pai;
};
typedef struct _noArvRN NoArvRN;

NoArvRN *busca (int valor, NoArvRN *raiz) {
    if (!raiz || raiz->chave == valor)
        return raiz;
    return (valor > raiz->chave) ?
        busca(valor, raiz->dir) :
        busca(valor, raiz->esq);
}
```

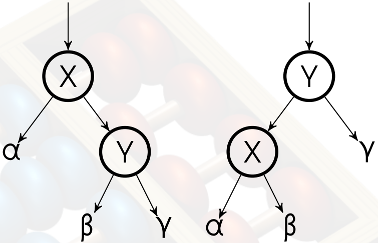
Rotações

Antes de vermos como fazer inserções em uma árvore rubro-negra, é preciso apresentar o conceito de **rotações**.

- Usamos as rotações para consertar (parte do) o estrago feito pelas operações já conhecidas de inserção e remoção nas propriedades rubro-negras
- O resto do estrago é consertado utilizando recoloração de nós
- Rotações são operações locais: alteram um número pequeno e constante de ponteiros

UNICAMP

Rotações

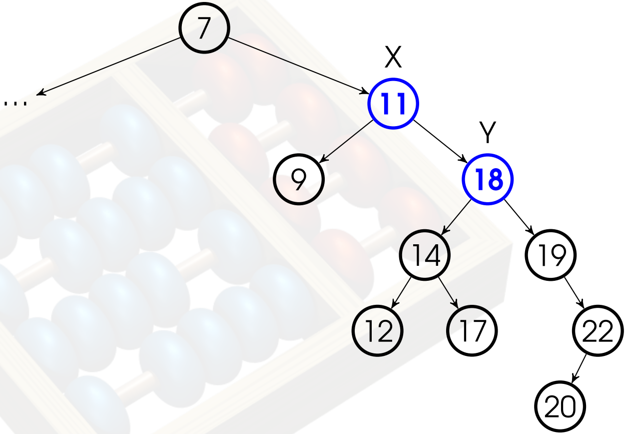


Rotação à esquerda, com pivô **X**

Rotação à direita, com pivô **Y**

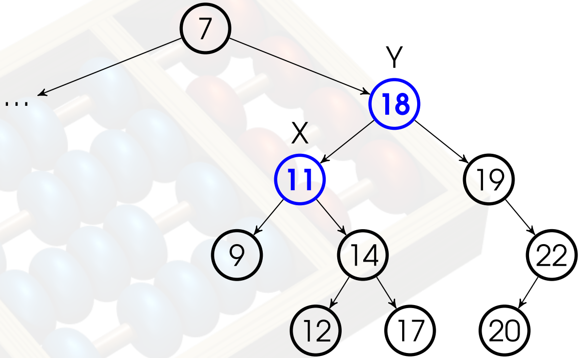
UNICAMP

Rotações



Operação de rotação a esquerda usando X como pivô...

Rotações



Rotação à esquerda - Código

```
void rotacaoAEsquerda(NoArvRN **raiz, NoArvRN *x) {
    NoArvRN *y;
    y = x->dir;
    x->dir = y->esq;
    y->esq->pai = x;
    y->pai = x->pai;
    if (x->pai == NULL) {
        *raiz = y;
    } else if (x == x->pai->esq) {
        x->pai->esq = y;
    } else {
        x->pai->dir = y;
    }
    y->esq = x;
    x->pai = y;
}
```

Este código assume que `x->dir` não é nulo e que o pai da raíz é nulo.

UNICAMP

Resumo

- Um nó que satisfaz as regras de uma árvore rubro-negra é denominado equilibrado, caso contrário é dito desequilibrado
- Em uma árvore rubro-negra todos os nós estão equilibrados
- Uma consequência direta das propriedades é que em qualquer caminho da raiz até uma folha não existem dois nós vermelhos consecutivos
- Cada vez que uma operação de modificação for realizada na árvore, o conjunto de propriedades é verificado em busca de violações
  - Caso alguma propriedade tenha sido quebrada, realizamos rotações e ajustamos as cores dos nós para que todas as propriedades continuem válidas

UNICAMP

Inserções em árvores rubro-negras

- Árvores rubro-negras são árvores de busca binária com algumas regras adicionais
  - Regra 1: Um nó é **vermelho** ou é **preto**
  - Regra 2: A raiz é **preta**
  - Regra 3: Toda folha (NULL) é **preta**
  - Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
  - Regra 5: Para cada nó p, todos os caminhos desde p até as folhas contêm o mesmo número de nós **pretos**
- Se utilizarmos o algoritmo que já conhecemos para a inserção em uma árvore rubro-negra, corremos o risco de quebrar algumas dessas regras. Mas quais?

UNICAMP

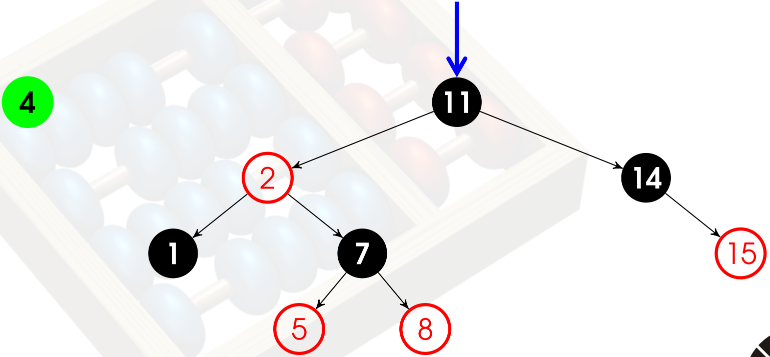
Inserções em árvores rubro-negras

Revisando a inserção em árvores binárias

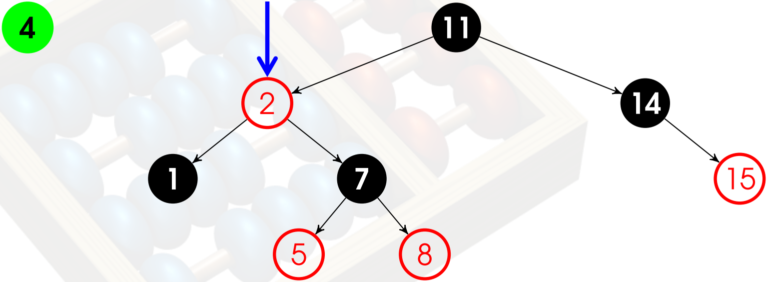
```
ArvBB insere(ArvBB r, No *novo) {
    No *p, *q; /* pai e filho */
    if (r == NULL) return novo; /* nova raiz */
    q = r;
    while (q != NULL) {
        p = q;
        if (novo->chave < q->chave)
            q = q->esq;
        else
            q = q->dir;
    }
    if (novo->chave < p->chave)
        p->esq = novo;
    else
        p->dir = novo;
    return r;
}
```

Revisão Inserções em Árv. Bin. de Busca

Revisando a inserção em árvores binárias

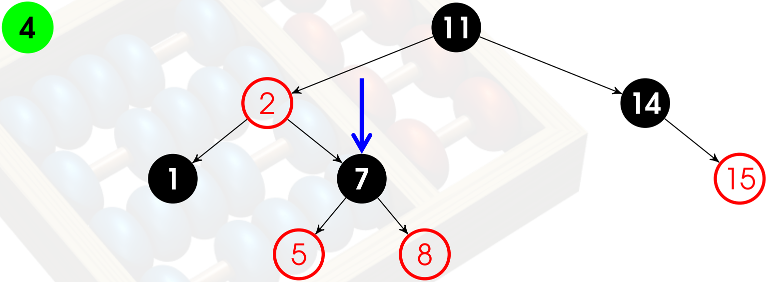


Revisando a inserção em árvores binárias



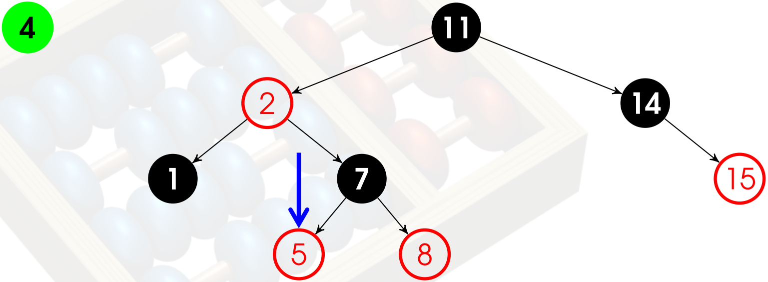
Revisão Inserções em Árv. Bin. de Busca

Revisando a inserção em árvores binárias



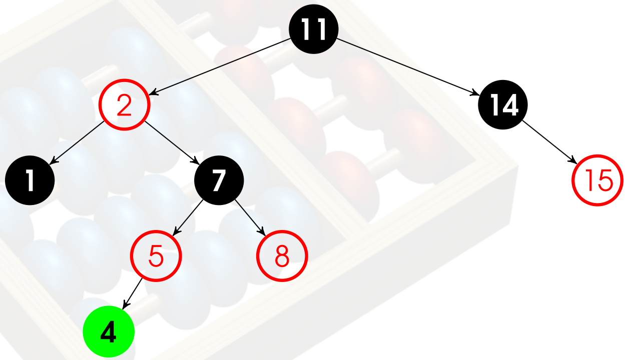
Revisão Inserções em Árv. Bin. de Busca

Revisando a inserção em árvores binárias



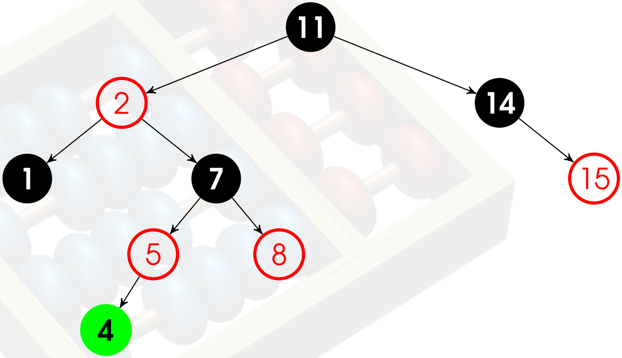
Revisão Inserções em Árv. Bin. de Busca

Revisando a inserção em árvores binárias



Revisão Inserções em Árv. Bin. de Busca

Revisando a inserção em árvores binárias

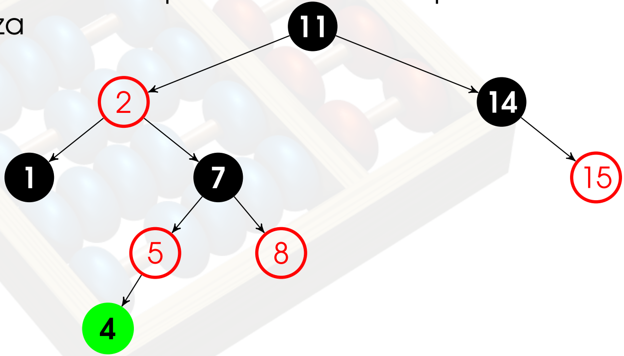


Quebra a **Regra 1** – Todo nó deve ser vermelho ou preto

Inserções em árvores rubro-negras

É preciso decidir, qual faz menos mal, colocar um nó vermelho ou um preto?

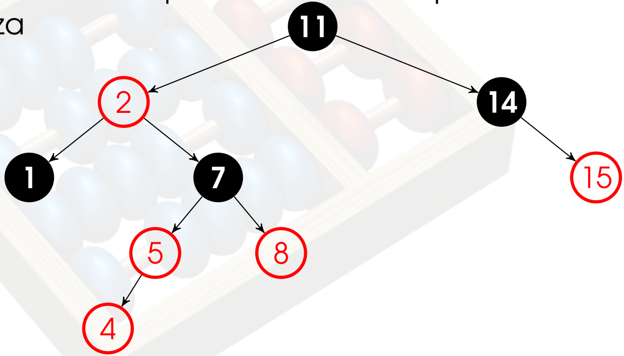
- O vermelho pode não quebrar nada
- O preto vai desequilibrar a altura de preto da raíz com certeza



Inserções em árvores rubro-negras

É preciso decidir, qual faz menos mal, colocar um nó vermelho ou um preto?

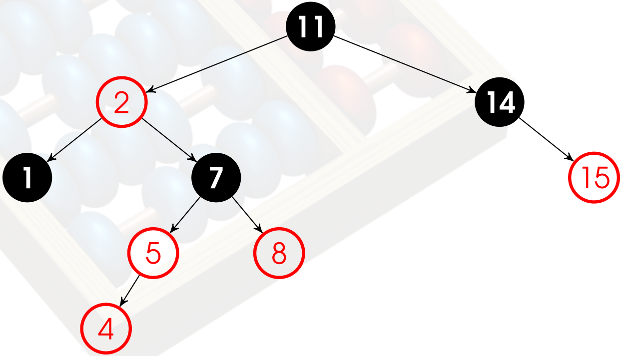
- O vermelho pode não quebrar nada
- O preto vai desequilibrar a altura de preto da raíz com certeza



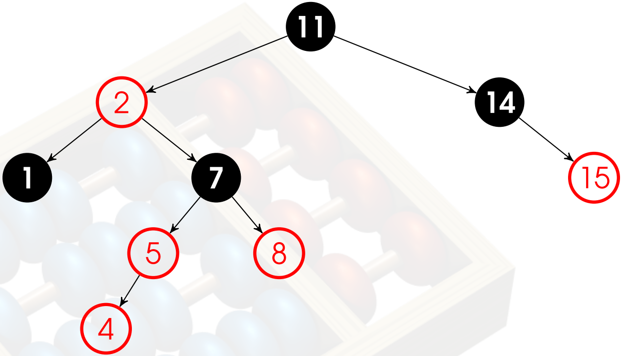
UNICAMP

Inserções em árvores rubro-negras

- Regra 1 resolvida, sempre insiro um nó com a **cor vemelha**
- E agora, qual regra eu quebrei?
- Melhor ainda, quais regras eu **poderia ter quebrado**?



Avaliando quebras de Regras

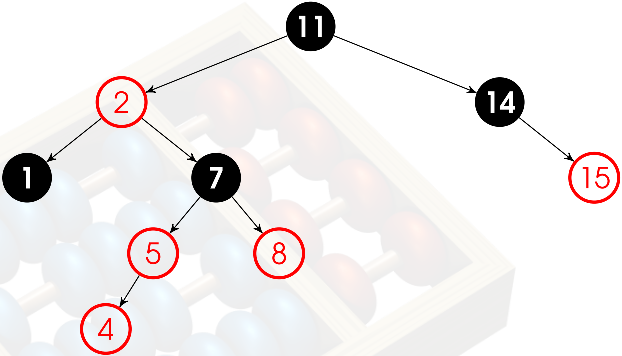


Colinha de regras

- Regra 1: Um nó é **vermelho** ou é **preto**
- Regra 2: A raiz é **preta**
- Regra 3: Toda folha (NULL) é **preta**
- Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
- Regra 5: Para cada nó p, todos os caminhos desde p até as folhas contêm o mesmo número de nós **pretos**

UNICAMP

Avaliando quebras de Regras

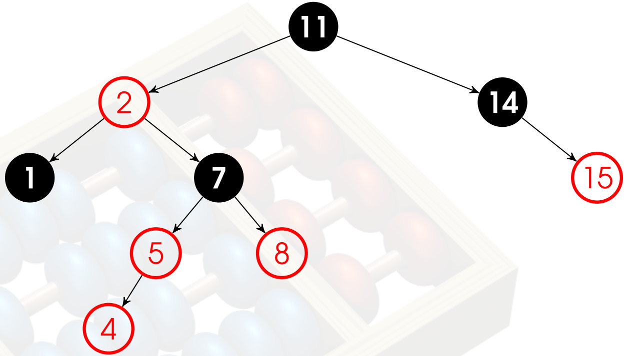


Colinha de regras

- ~~Regra 1: Um nó é **vermelho** ou é **preto**~~
- Regra 2: A raiz é **preta**
- Regra 3: Toda folha (NULL) é **preta**
- Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
- Regra 5: Para cada nó p, todos os caminhos desde p até as folhas contêm o mesmo número de nós **pretos**

UNICAMP

Avaliando quebras de Regras

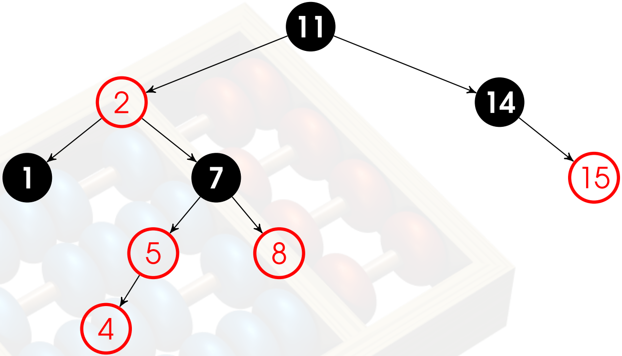


Colinha de regras

- ~~Regra 1: Um nó é **vermelho** ou é **preto**~~
- Regra 2: A raiz é **preta**
- ~~Regra 3: Toda folha (NULL) é **preta**~~
- Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
- Regra 5: Para cada nó p, todos os caminhos desde p até as folhas contêm o mesmo número de nós **pretos**

UNICAMP

Avaliando quebras de Regras

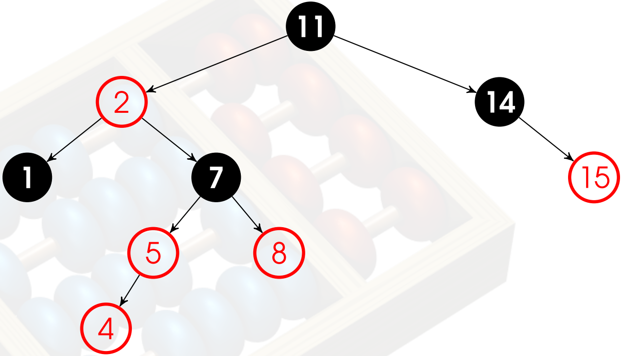


Colinha de regras

- ~~Regra 1: Um nó é **vermelho** ou é **preto**~~
- Regra 2: A raiz é **preta**
- ~~Regra 3: Toda folha (NULL) é **preta**~~
- Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
- ~~Regra 5: Para cada nó p, todos os caminhos desde p até as folhas contêm o mesmo número de nós **pretos**~~

UNICAMP

Inserções em árvores rubro-negras



Ok! Sabemos que podemos quebrar apenas a Regra 2 e a Regra 4. Vamos escrever o código.

Inserções em árvores rubro-negras

- Lembrete da estrutura de dados da árvore:

```
enum _CorNo {Vermelho, Preto};
typedef enum _CorNo CorNo;

struct _noArvRN {
    int chave;
    CorNo cor;
    struct _noArvRN *dir, *esq, *pai;
};
typedef struct _noArvRN NoArvRN;
```

UNICAMP

Inserções em árvores rubro-negras

```
NoArvRN *insereArvRN(NoArvRN *raiz, NoArvRN *novo) {
    NoArvRN *novaRaiz = raiz;
    NoArvRN *pai = NULL;
    NoArvRN *atual = raiz;
    //busca a posicao na árvore
    while (atual != NULL) {
        pai = atual;
        atual = (novo->chave < atual->chave) ? atual->esq :atual->dir;
    }
    novo->pai = pai;
    if (pai == NULL) novaRaiz = novo;
    else if (novo->chave < pai->chave) pai->esq = novo;
    else pai->dir = novo;
    novo->esq = NULL;
    novo->dir = NULL;
    novo->cor = Vermelho;
    consertaInsercaoArvRN(&novaRaiz, novo);
    return novaRaiz;
}
```

UNICAMP

Quebra na regra 2

Resolvendo a quebra na Regra 2

- Basta pintar o nó de preto.
- Isto não quebra nenhuma outra regra. **Por que?**

Inserções em árvores rubro-negras

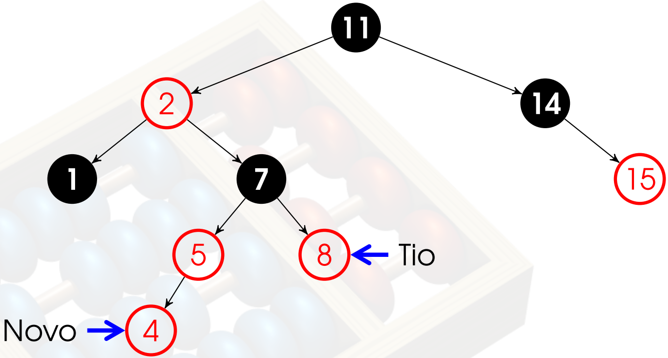
O código (incompleto) da função de conserto da árvore para a Regra 2 fica então:

```
//versão ainda incompleta
NoArvRN *consertaInsercaoArvRN(NoArvRN **raiz, NoArvRN *novo) {
    if (*raiz == novo) (*raiz)->cor = Preto; //Conserta quebra regra 2
}
```

Vamos para a Regra 4...

UNICAMP

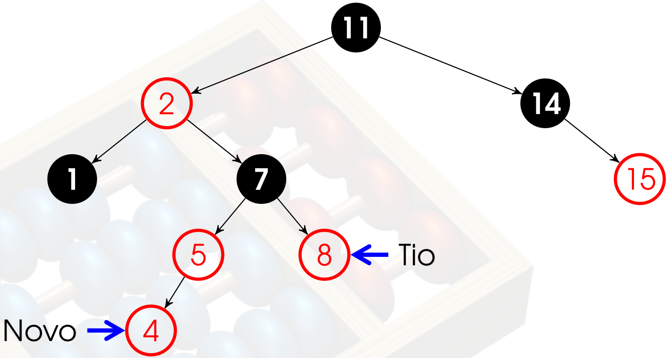
Inserções em árvores rubro-negras



Existem 3 casos que precisamos tratar

- Caso 1 – O tio de `novo` é **vermelho**
- Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita
- Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

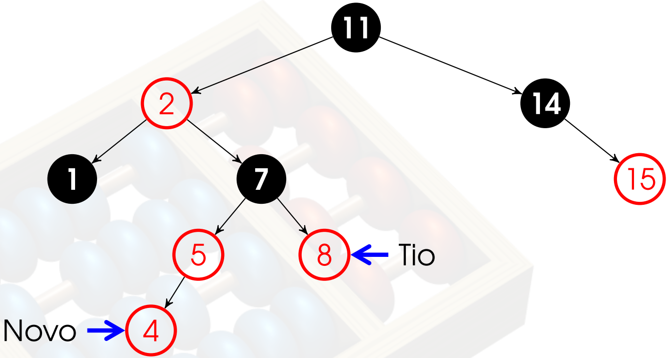
Inserções em árvores rubro-negras



Existem 3 casos que precisamos tratar

- Caso 1 – O tio de `novo` é **vermelho**
- ~~Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita~~
- Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

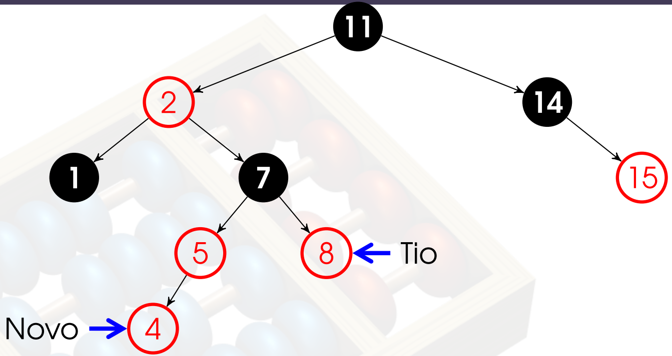
Inserções em árvores rubro-negras



Existem 3 casos que precisamos tratar

- Caso 1 – O tio de `novo` é **vermelho**
- ~~Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita~~
- ~~Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda~~

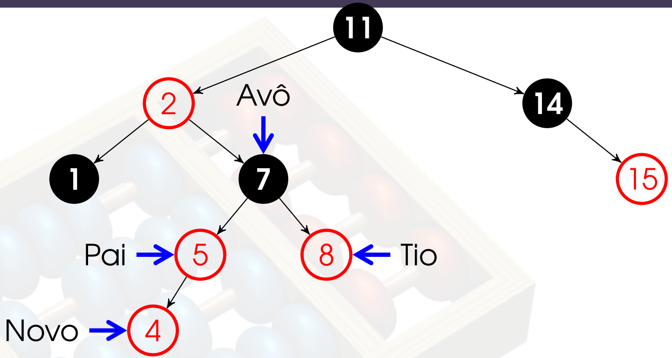
Inserções em árvores rubro-negras



- O pulo do gato é que quando resolvemos um caso, ou criamos um outro em outro nó mais alto da árvore, ou fazemos algumas rotações que resolvem o problema.
- Como a árvore tem O(lg n) níveis, precisamos fazer no máximo O(lg n) consertos até bater na raíz e terminar.

UNICAMP

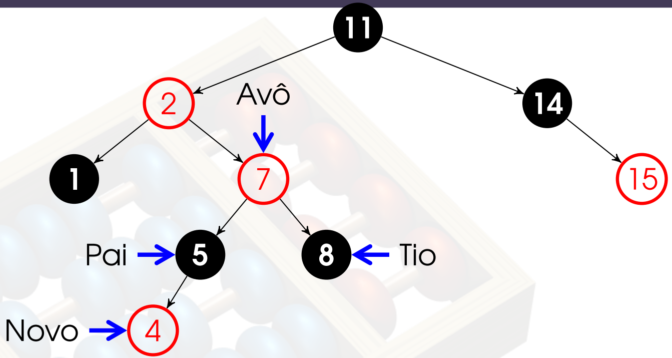
Inserções – Caso 1



Caso 1 – O **tio** de **novo** é **vermelho**

- Se o **tio** é vermelho o **avô** obrigatoriamente é **preto** (PQ?)
- Troque a cor do **pai** e **tio** para **preto**
- Troque a cor do **avô** para **vermelho**
- Neste ponto, consertamos o problema no novo, mas possivelmente estragamos o **avô**

Inserções – Caso 1

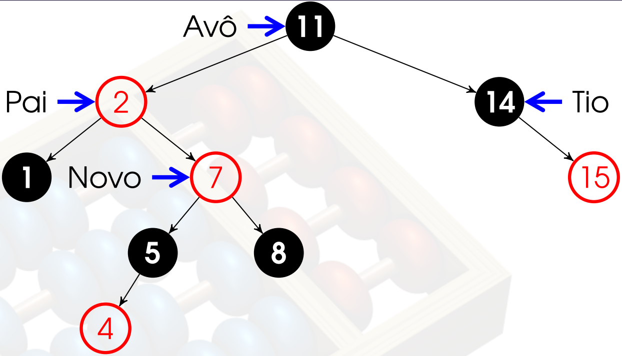


Caso 1 – O **tio** de **novo** é **vermelho**

- Se o **tio** é vermelho o **avô** obrigatoriamente é **preto** (PQ?)
- Troque a cor do **pai** e **tio** para **preto**
- Troque a cor do **avô** para **vermelho**
- Neste ponto, consertamos o problema no novo, mas possivelmente estragamos o **avô**

UNICAMP

Inserções em árvores rubro-negras

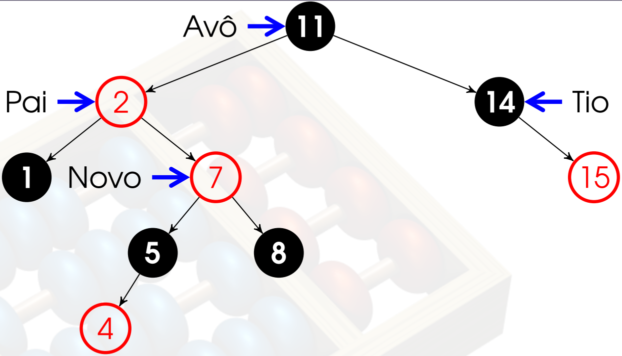


Continuamos o conserto, agora tomando como referência o antigo avô, o nó com o valor 7.

- Caso 1 – O tio de `novo` é **vermelho**
- Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita
- Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

UNICAMP

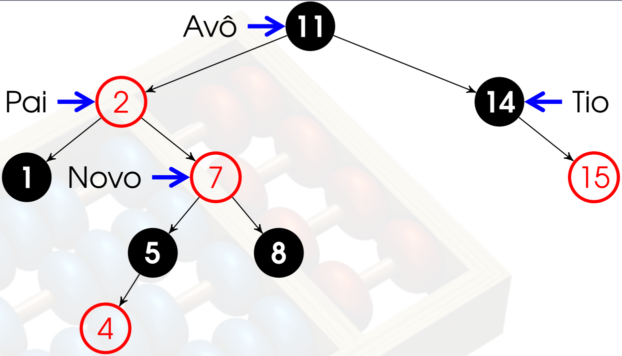
Inserções em árvores rubro-negras



Continuamos o conserto, agora tomando como referência o antigo avô, o nó com o valor 7.

- Caso 1 – O tio de `novo` é vermelho
- Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita
- Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

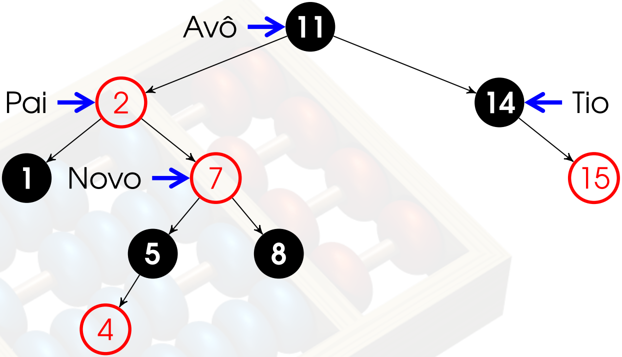
Inserções em árvores rubro-negras



Continuamos o conserto, agora tomando como referência o antigo avô, o nó com o valor 7.

- Caso 1 – O tio de `novo` é vermelho
- Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita
- Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

Inserções – Caso 2

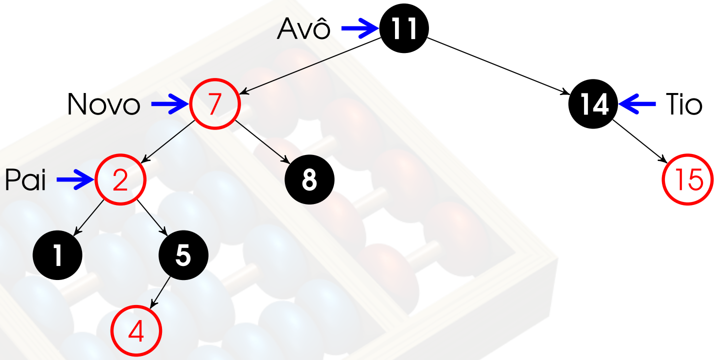


Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita

- Executa uma rotação à esquerda tendo como pivô o pai
- Neste ponto, consertamos o problema no novo, mas possivelmente estragamos o pai

UNICAMP

Inserções – Caso 2

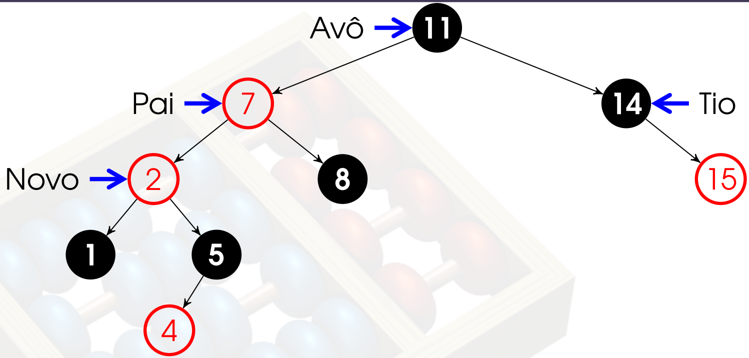


Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita

- Executa uma rotação à esquerda tendo como pivô o pai
- Neste ponto, consertamos o problema no novo, mas possivelmente estragamos o **pai**

UNICAMP

Inserções em árvores rubro-negras

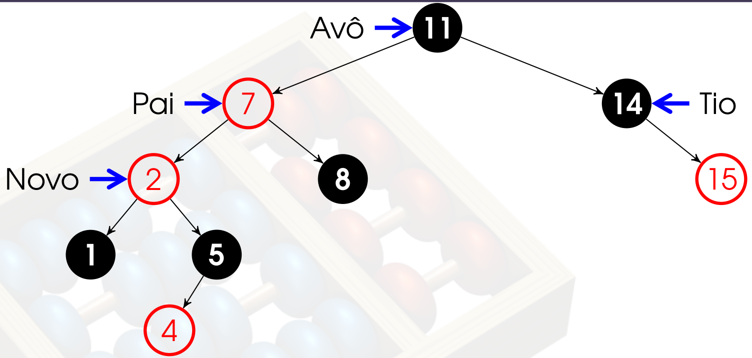


Continuamos o conserto, agora tomando como referência o antigo pai, o nó com o valor 2.

- Caso 1 – O tio de `novo` é **vermelho**
- Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita
- Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

UNICAMP

Inserções em árvores rubro-negras

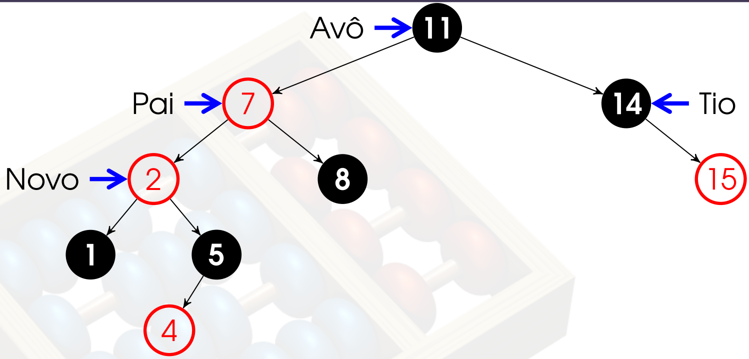


Continuamos o conserto, agora tomando como referência o antigo pai, o nó com o valor 2.

- Caso 1 – O tio de `novo` é vermelho
- Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita
- Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

UNICAMP

Inserções em árvores rubro-negras

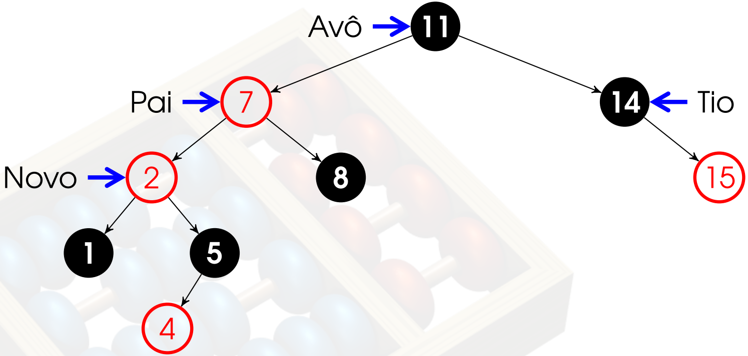


Continuamos o conserto, agora tomando como referência o antigo pai, o nó com o valor 2.

- Caso 1 – O tio de `novo` é vermelho
- Caso 2 – O tio de `novo` é **preto** e `novo` é filho da direita
- Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

UNICAMP

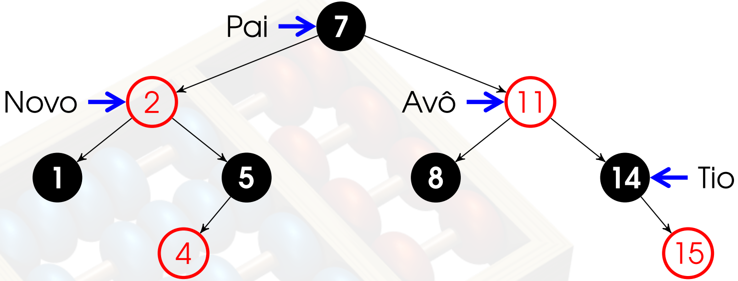
Inserções – Caso 3



Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

- Troca a cor do pai para **preto**
- Troca a cor do avô para **vermelho**
- Executa uma rotação à direita tendo como pivô o avô
- Neste ponto a árvore voltou a ser uma árvore rubro-negra válida

Inserções – Caso 3



Caso 3 – O tio de `novo` é **preto** e `novo` é filho da esquerda

- Troca a cor do pai para **preto**
- Troca a cor do avô para **vermelho**
- Executa uma rotação à direita tendo como pivô o avô
- Neste ponto a árvore voltou a ser uma árvore rubro-negra válida

UNICAMP

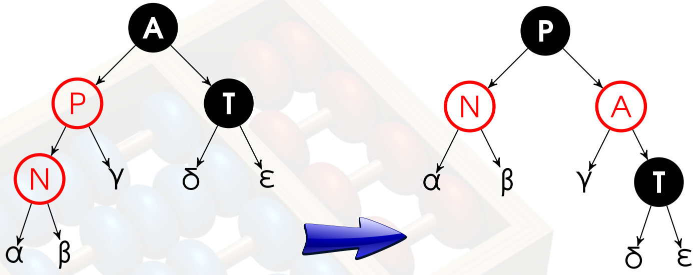
Casos 2 e 3 – o tio do nó inserido é preto

No Caso 2 e no Caso 3

- Apenas consideramos um lado da árvore, o caso onde o **pai** é filho esquerdo do **avô**.
- A resolução dos demais casos é simétrica
- Se o **tio** é preto, então existem 4 configurações possíveis:
  1. Configuração Esq-Esq (**pai** é filho esquerdo do **avô** e **novo** é filho esquerdo de **pai**)
  2. Configuração Esq-Dir (**pai** é filho esquerdo do **avô** e **novo** é filho direito de **pai**)
  3. Configuração Dir-Dir (espelho da 1)
  4. Configuração Dir-Esq (espelho da 2)

UNICAMP

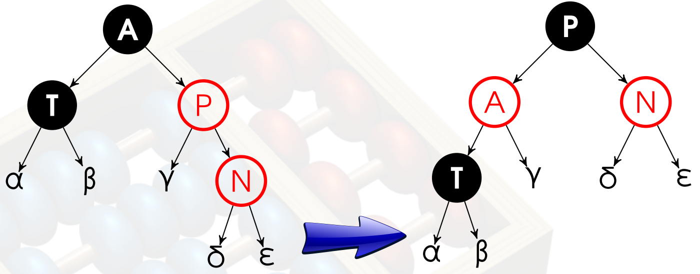
Configuração Esq-Esq



A = Avô; P = Pai; T = Tio; N = Novo; α, β, γ, δ, ε = Subárvores

1. Efetua rotação à direita em torno do avô
2. Troca as cores do avô e pai

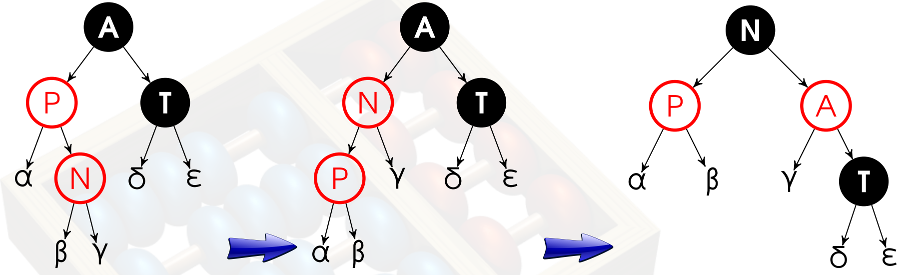
Configuração Dir-Dir



A = Avô; P = Pai; T = Tio; N = Novo; α, β, γ, δ, ε = Subárvores

1. Efetua rotação à esquerda em torno do avô
2. Troca as cores do avô e pai

Configuração Esq-Dir

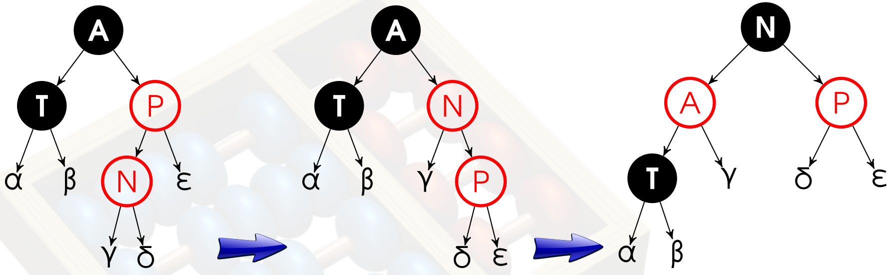


A = Avô; P = Pai; T = Tio; N = Novo; α, β, γ, δ, ε = Subárvores

1. Efetua rotação à esquerda em torno do pai
2. Aplica a configuração Esq-Esq

UNICAMP

Configuração Dir-Esq



A = Avô; P = Pai; T = Tio; N = Novo; α, β, γ, δ, ε = Subárvores

1. Efetua rotação à direita em torno do pai
2. Aplica a configuração Dir-Dir

UNICAMP

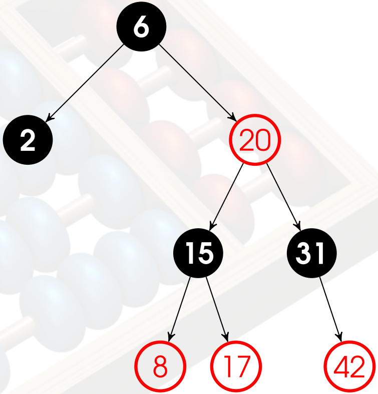
Código do Conserto Pós-Inserção

```
NoArvRN *consertaInsercaoArvRN(NoArvRN **raiz, NoArvRN *novo) {
    while (novo->pai->cor == Vermelho) {
        if (novo->pai == novo->pai->pai->esq) {
            NoArvRN *tio = novo->pai->pai->dir;
            if (tio->cor == Vermelho) {
                novo->pai->cor = Preto; tio->cor = Preto; //Caso 1
                novo->pai->pai = Vermelho; //Caso 1
                novo = novo->pai->pai; //Caso 1
            } else {
                if (novo == novo->pai->dir) {
                    novo = novo->pai; //Caso 2
                    rotacaoAEsquerda(raiz, novo); //Caso 2
                }
                novo->pai->cor = Preto; novo->pai->pai = Vermelho;//Caso 3
                rotacaoADireita(raiz, novo->p->p); //Caso 3
            } else {/*Código simétrico ao código acima*/}
        }}
    (*raiz)->cor = Preto; //Conserta possível quebra regra 2
}
```

UNICAMP

Exercício

Insira na árvore abaixo as seguintes chaves: 1, 5, 19



Remoções em árvores rubro-negras

- Assim como no caso da **inserção**, nós utilizaremos rotaçôes e recolorações para manter as propriedades da árvore rubro negra
  - Contudo, a **remoção** de nós de uma árvore rubro-negra exige um pouco mais de trabalho
- Durante a **inserção**, baseamos as nossas operações de readaptação na cor do tio
  - Já durante a **remoção** nós nos baseamos na cor do irmão do nó para decidir qual caso aplicar.
- Durante a **inserção** o principal problema que enfrentamos era um duplo nó vermelho
  - Durante a **remoção**, se retirarmos um nó preto, estamos "estragando" a propriedade de altura de preto da árvore

UNICAMP

Revisão – Remoção em Árv. de Busca

Problema: dada uma árvore binária de busca r e uma chave k, remover o nó com chave k (se existir) de r de modo que árvore binária resultante continue sendo uma árvore binária de busca.

Como vimos nas aulas passadas, isto é mais difícil do que a inserção. Para resolver este problema tratamos a princípio a remoção de uma raíz e depois partimos para a resolução de um problema mais geral.

Revisão – Remoção em Árv. de Busca

- Caso 1: a raíz não tem filhos. A árvore torna-se vazia.
- Caso 2: a raíz tem um único filho. Seu filho torna-se a nova raíz.
- Caso 3: a raíz tem dois filhos. Neste caso, tomamos o nó que sucede (ou precede) a raíz no percurso inordem *(e-r-d)* como a nova raíz.

UNICAMP

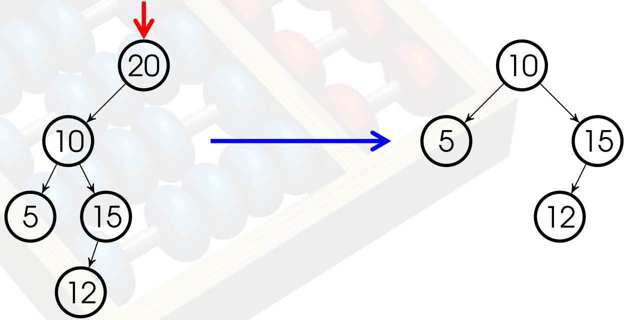
Revisão – Remoção em Árv. de Busca

- Caso 1: a raíz não tem filhos. A árvore torna-se vazia.

13 $\longrightarrow$ NULL

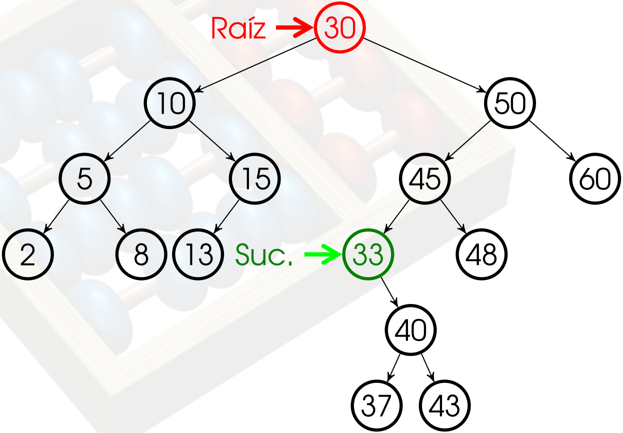
Revisão – Remoção em Árv. de Busca

- Caso 2: a raíz tem um único filho. Seu filho torna-se a nova raíz.



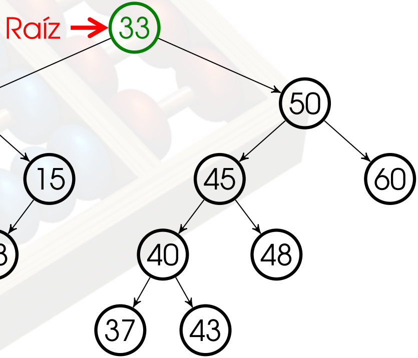
Revisão – Remoção em Árv. de Busca

- Caso 3: a raíz tem dois filhos. Neste caso, tomamos o nó que sucede (ou precede) a raíz no percurso inordem *(e-r-d)* como a nova raíz.

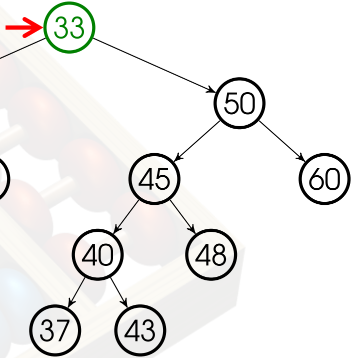


Revisão – Remoção em Árv. de Busca

- Caso 3: a raíz tem dois filhos. Neste caso, tomamos o nó que sucede (ou precede) a raíz no percurso inordem *(e-r-d)* como a nova raíz.



Revisão – Remoção em Árv. de Busca



Outra maneira de interpretar o Caso 3 é a seguinte: copiamos a informação do nó sucessor para a raiz e removemos o nó copiado (ou seja, reduzimos ao Caso 1 ou Caso 2).

UNICAMP

Revisão – Remoção em Árv. de Busca

Consulte os slides da Aula 12 para mais detalhes sobre a remoção de nós em árvores binárias de busca

Remoção em árvores Rubro-Negras

- Passos
    - Encontre o nó v a ser removido
    - Remova o nó v da árvore (use o algoritmo que acabamos de rever)
    - Conserte as propriedades da árvore rubro-negra
- Assim como as outras operações básicas em uma árvore rubro-negra com n nós, a remoção também tem complexidade de $O(\lg n)$

UNICAMP

Remoção ARN – Problemas

Que problemas podemos criar quando copiamos o valor do sucessor para a posição do nó a ser removido e removemos o sucessor (original/copiado)?

- **Remoção de um nó vermelho**: Não causa problemas no balanceamento.
  - Nenhuma altura preta mudou
  - Nenhum nó vermelho se tornou adjacente
  - Como o nó é vermelho, ele não era raíz e portanto a raíz permanece preta
- **Remoção de um nó preto**: Pode causar problema **"duplo preto"**

UNICAMP

Remoção ARN – Consertando a árvore

- Se o nó removido for **preto**
  1. Se `suc` era raíz e um filho vermelho de `suc` se torna raíz quebramos a Regra 2
  2. Se tanto `x` quanto `suc->pai` (que agora também é `x->pai`) eram vermelhos então violamos a Regra 4
  3. A remoção de `suc` faz com que qualquer caminho que continha `suc` anteriormente ter um nó preto a menos. Desse modo quebramos a Regra 5

Colinha de regras

- Regra 1: Um nó é **vermelho** ou é **preto**
- Regra 2: A raiz é **preta**
- Regra 3: Toda folha (NULL) é **preta**
- Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
- Regra 5: Para cada nó p, todos os caminhos desde p até as folhas contêm o mesmo número de nós **pretos**

UNICAMP

Remoção ARN – Consertando a árvore

Assim como na inserção onde tratamos diferentes casos de violações, vamos tratar de 4 casos diferentes para consertar a árvore após uma remoção

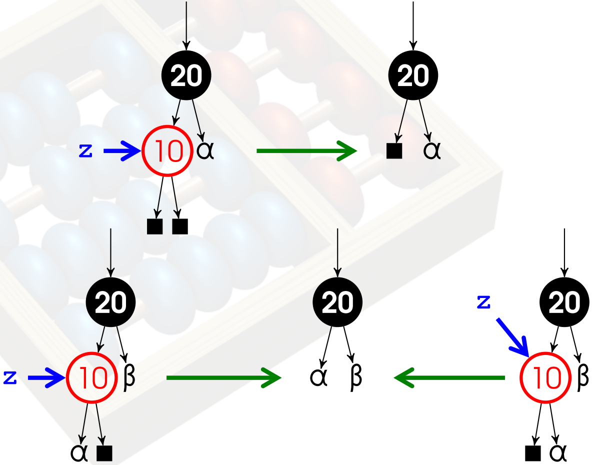
Caso	**Nó a ser removido** `z`	**Sucessor** `suc`
Caso 1	Vermelho	Vermelho
Caso 2	**Preto**	Vermelho
Caso 3	**Preto**	**Preto**
Caso 4	Vermelho	**Preto**

UNICAMP

Remoção ARN – Caso 1

Caso 1 O nó a ser removido `z` é vermelho e possui 0 ou 1 filho

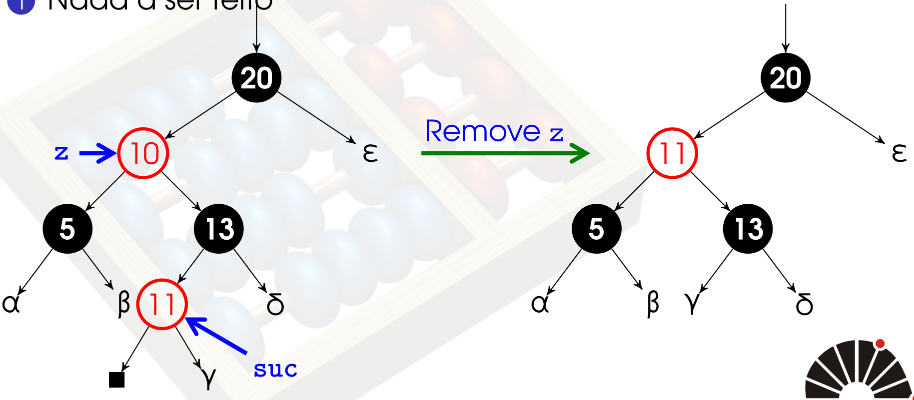
1. Remova o nó diretamente



Remoção ARN – Caso 1

Caso 1 O nó a ser removido `z` é vermelho e `suc`, sucessor de `z` também é vermelho

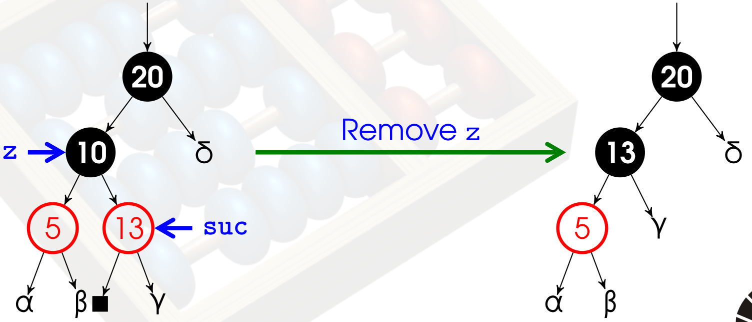
1. Nada a ser feito



Remoção ARN – Caso 2

Caso 2 O nó a ser removido `z` é **preto** e `suc`, sucessor de `z` é vermelho

1. Pinte `suc` de **preto**



Remoção ARN – Caso 3

- O problema do duplo preto ocorre quando retiramos um nó **preto**
- O duplo preto nada mais é do que uma forma de compensar a falta do nó removido na altura de preto da árvore
- Existem 4 casos a tratar caso o nó `suc` a ser removido for **preto**

UNICAMP

Remoção ARN – Caso 3

Usaremos a seguinte nomenclatura para os nós envolvidos no processo de remoção:

- `z` – O nó a ser removido
- `suc` – Onde `suc = z` se z possui 0 ou 1 filho. Caso contrário `suc = sucessor (z)`
- `x` – O filho de `suc` antes de qualquer modificação, ou `NULL` caso `suc` não tenha filhos
- `w` – o tio de `x` (irmão de `suc`) antes da remoção de `z`

UNICAMP

Remoção ARN – Caso 3

- Além disso, durante o conserto da árvore, quando considerarmos o nó `x` vamos conta-lo como **preto duas vezes**
- A ideia é que o `x` possa receber sempre a contagem de preto do pai para manter a Regra 5.
- Nos casos em que `x` é vermelho (ainda assim o contamos como preto no cálculo da altura de preto), basta pintar `x` de **preto** para finalizar o conserto da árvore
- `x` aponta para um duplo preto, o que nos diz que o nó `w` existe (Pq?)

UNICAMP

Remoção ARN – Caso 3

Resumo das variações do Caso 3: `z` **preto** e `suc` **preto**

- **Caso 3.1** – O irmão `w` de `x` é vermelho
- **Caso 3.2** – O irmão `w` de `x` é **preto**, e ambos os filhos de `w` são **pretos**
- **Caso 3.3** – O irmão `w` de `x` é **preto**, o filho esquerdo de `w` é vermelho e o filho da direta de `w` é **preto**
- **Caso 3.4** – O irmão `w` de `x` é **preto**, e o filho direito de `w` é vermelho

UNICAMP

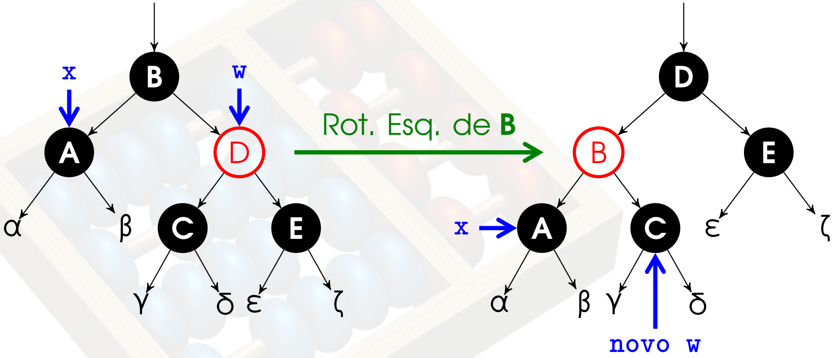
Remoção ARN – Caso 3.1

Caso 3.1 – O irmão `w` de `x` é vermelho

- Claramente o pai deles é vermelho
- Como `w` é vermelho, ambos os filhos são **pretos**. Logo:
  - Trocamos as cores de `w` e `x.pai`
  - Efetuamos uma rotação à esquerda tendo como pivô `x.pai`
- Essas alterações não violam nenhuma regra da árvore rubro-negra
  - Mas transformam o Caso 3.1 no Caso 3.2, Caso 3.3 ou Caso 3.4

UNICAMP

Remoção ARN – Caso 3.1



Caso 3.1 – O irmão `w` de `x` é vermelho

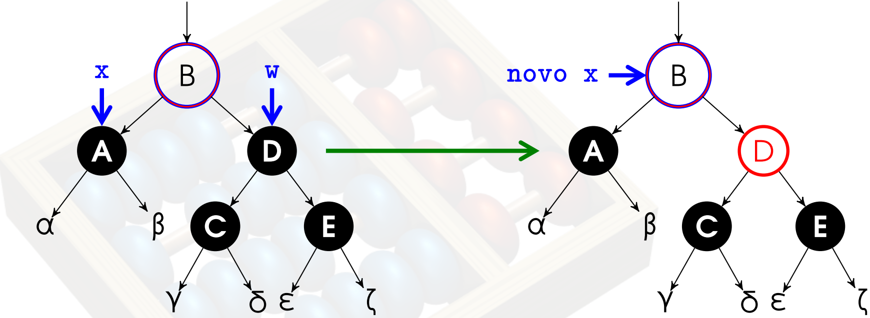
Remoção ARN – Caso 3.2

Caso 3.2 – O irmão `w` de `x` é **preto**, e ambos os filhos de `w` são **pretos**

- Retiramos um preto de `x` e um de `w` deixando `x` com apenas um preto e deixando `w` vermelho
- Para compensar a remoção de um preto de `x` e de `w`, adicionamos um preto extra no `x.pai` que era orinalmente **preto** ou vermelho
- Jogamos a bomba para cima, tratando o `x.pai` como sendo o **novo** `x`

UNICAMP

Remoção ARN – Caso 3.2



Caso 3.2 – O irmão `w` de `x` é **preto**, e ambos os filhos de `w` são **pretos**

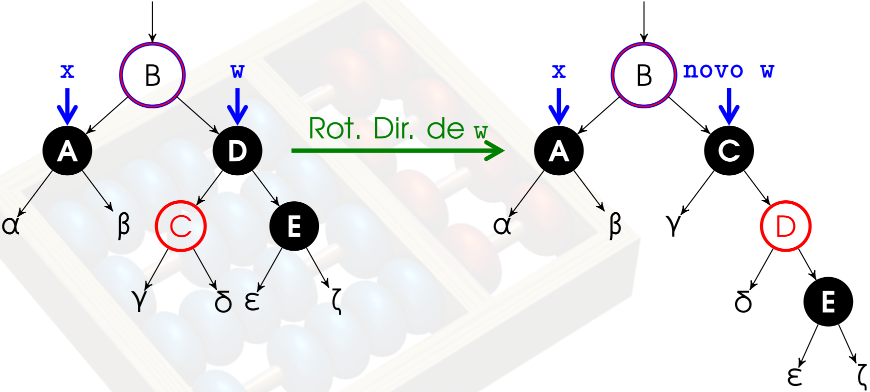
Remoção ARN – Caso 3.3

Caso 3.3 – O irmão `w` de `x` é **preto**, o filho esquerdo de `w` é vermelho e o filho da direta de `w` é **preto**

- Trocamos as cores de `w` e de seu filho esquerdo
- Rotaciona árvore à direita usando como pivô `w`
- Essas operações não introduzem violações
- Neste ponto o novo irmão `w` de `x` é um nó **preto** com um filho da direita vermelho. Estamos, portanto, no Caso 3.4.

UNICAMP

Remoção ARN – Caso 3.3



Caso 3.3 – O irmão `w` de `x` é **preto**, o filho esquerdo de `w` é vermelho e o filho da direta de `w` é **preto**

UNICAMP

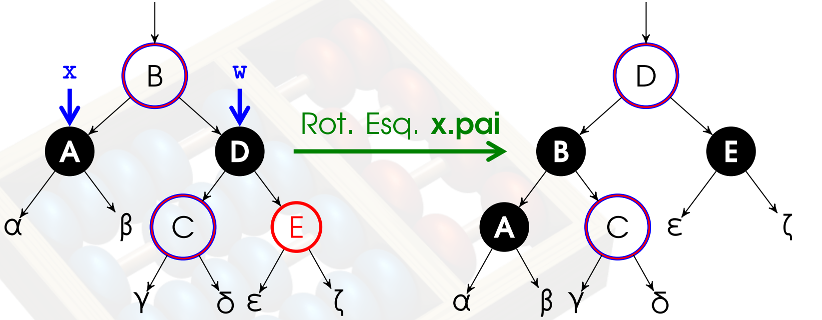
Remoção ARN – Caso 3.4

Caso 3.4 – O irmão `w` de `x` é **preto**, e o filho direito de `w` é vermelho

- Rotaciona árvore à esquerda usando como pivô `x.pai`
- `w` é pintado com a cor de `x.pai`
- Pinta o `x.pai` de preto
- Pinta o filho direito de `w` de preto

UNICAMP

Remoção ARN – Caso 3.4



`novo x` = Raíz

Caso 3.4 – O irmão `w` de `x` é **preto**, e o filho direito de `w` é vermelho

UNICAMP

Remoção ARN – Caso 4

Caso 4 O nó a ser removido `z` é vermelho e `x`, irmão de `suc`, é **preto**

1. Pinte `x` de vermelho
2. Resolva usando o Caso 3

Remoção ARN – Código

```
NoArvRN *removeArvRN(NoArvRN **raiz, NoArvRN *z) {
  NoArvRN *suc, x;
  if (z->esq==NULL || z->dir==NULL) suc = z;//determina substituto de z
  else suc = sucessor(z);
  if (y->esq != NULL) x = y->esq;//determina filho do substituto
  else x = y->dir;
  if (x) x->pai = y->pai;//acerta pai do filho de suc
  if (y->pai == NULL) {//Se suc for raiz, conserta a árvore
      (*raiz) = x;
  } else {//Senão acerta os filhos do pai de suc
      if (suc == suc->pai->esq)
           p->suc->esq = x; //suc é filho esquerdo
      else p->suc->dir = x;//suc é filho direto
  }
  if (suc != z) z->chave = suc->chave;//copia dados de suc em z
  if (suc->cor == PRETO)
      consertaRemocaoArvRN(raiz, x);
  free(suc);
}
```

Remoção ARN – Código

```
void consertaRemocaoArvRN (NoArvRN **raiz, NoArvRN *x) {
  while(x != *raiz && x->cor == Preto) {
      if (x == x->pai->esq) {
          NoArvRN *w = x->pai->dir;
          if (w->cor == Vermelho) { //Caso 3.1: w Vermelho
              w->cor = Preto;
              x->pai->cor = Vermelho;
              rotacaoAEsquerda(raiz, x->pai);
              w = x->pai->dir;
          }
          if (w->esq->cor == Preto && w->dir->cor == PRETO) {//Caso 3.2: w
              w->cor = Vermelho;
              x = x->pai;
          } else { //Caso 3.3 e 3.4
              //...
```

Remoção ARN – Código

```
        //...
        } else { //Caso 3.3 e 3.4
            if (w->dir->cor == Preto) {
                w->esq->cor = Preto;
                w->cor = Vermelho;
                rotacaoADireita(raiz, w);
                w = x->pai->dir;
            }
            w->cor = x->pai->cor;
            x->pai->cor = Preto;
            w->dir->cor = Preto;
            rotacaoAEsquerda(raiz, x->pai);
            x = *raiz;
        }
    } else {...}//Simétrico à cláusula anterior com dir e esq trocados
  }
  x->cor = Preto;
}
```

AVL vs. ARN

- **AVL**
  - Primeira árvore binária de busca com balanceamento proposta por Adelson-Velskii e Landis em 1962
  - Altura entre log(n+1) e 1.44404 * log(n+2) -0.328, ou seja, O(lg n)
- **ARN**
  - Proposta por Guibas e Sedgewick em 1978.
  - Altura até 2 * log (n + 1), logo, O (lg n)

UNICAMP

AVL vs. ARN

- Na teoria ambas tem a mesma complexidade de inserção, remoção e busca (O(lg n))
- Na prática a árvore AVL é mais rápida para buscas e mais lenta para inserção e remoção
- As árvores AVL são mais rigidamente balanceadas do que as árvores rubro-negras, o que permite uma operação de busca mais rápida mas também copromete o desempenho das operações de modificação
- Por **exemplo**, uma operação de remoção pode exigir O(lg n) rotações na AVL no pior caso, enquanto uma árvore rubro-negra se arrumaria com apenas 3

UNICAMP

AVL vs. ARN

Quando usar AVL ou ARN?

- Se a aplicação da árvore realiza de forma mais intensa operações de busca, é mais apropriado o uso de uma árvore AVL
- Se operações de modificação são mais comuns, o melhor é usar uma árvore rubro-negra
- Via de regra árvores rubro negras são mais utilizadas em bibliotecas