Árvores Rubro - Negras

Árvore rubro-negra

Uma **árvore rubro-negra** é um tipo de árvore de busca binária balanceada. A estrutura original foi inventada em 1972 por que a chamou de "Árvores Binárias B Simétricas", mas adquiriu este nome moderno em um artigo de 1978 escrito por Leonidas J. Guibas e Robert Sedgewick.

Ela é complexa, mas tem um bom pior-caso de tempo de execução para suas operações e é eficiente na prática: pode-se buscar, inserir, e remover em tempo O(log *n*), onde *n* é o número total de elementos da árvore.

De maneira simplificada, uma árvore rubro-negra é uma árvore de busca binária que insere e remove de forma inteligente, para assegurar que a árvore permaneça aproximadamente balanceada.

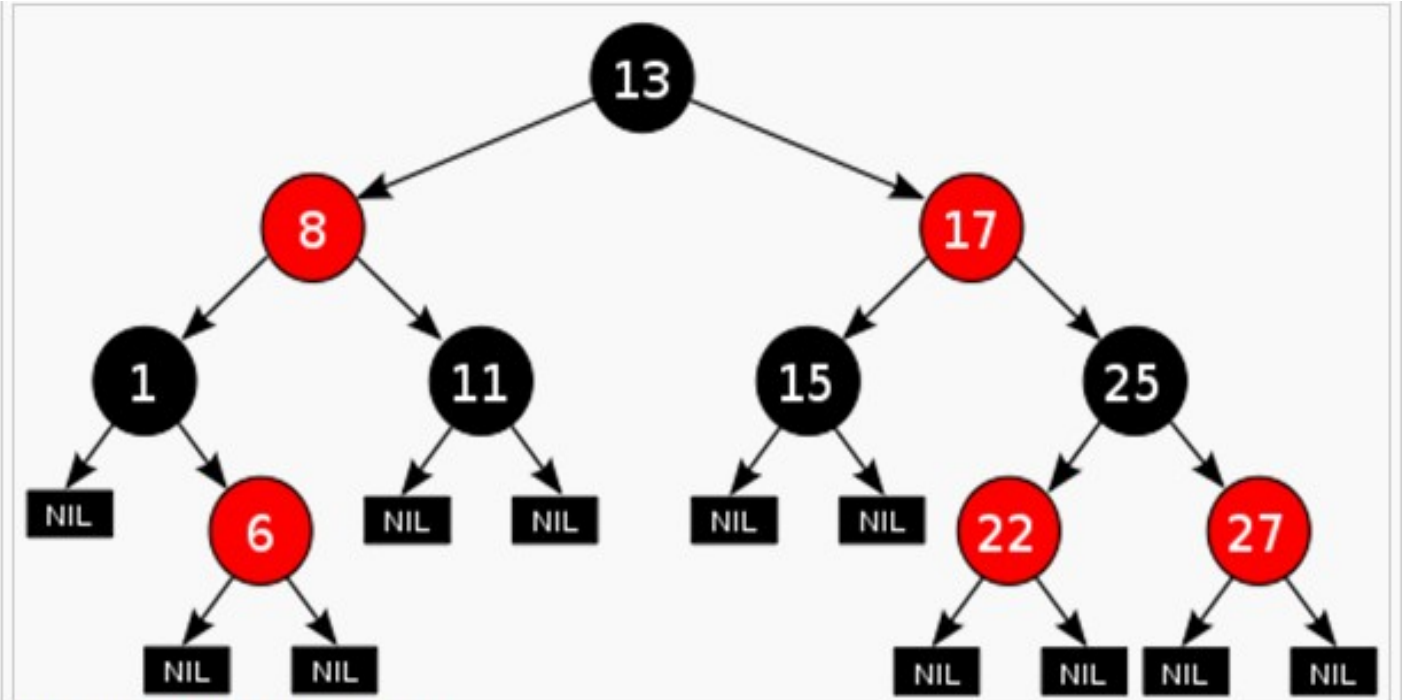
Árvore rubro-negra

Uma árvore rubro-negra é uma árvore de busca binária onde cada nó tem um atributo de *cor*, *vermelho* ou *preto*. Além dos requisitos ordinários impostos pelas árvores de busca binárias, as árvores rubro-negras tem os seguintes requisitos adicionais: um nó é vermelho ou preto.

A raiz é preta. (Esta regra é usada em algumas definições. Como a raiz pode sempre ser alterada de vermelho para preto, mas não sendo válido o oposto, esta regra tem pouco efeito na análise.). Todas as folhas(null) são pretas. Ambos os filhos de todos os nós vermelhos são pretos. Todo caminho de um dado nó para qualquer de seus nós folhas descendentes contem o mesmo número de nós pretos.

Essas regras asseguram uma propriedade crítica das árvores rubro-negras: que o caminho mais longo da raiz a qualquer folha não seja mais do que duas vezes o caminho mais curto da raiz a qualquer outra folha naquela árvore. O resultado é que a árvore **é aproximadamente balanceada**. Como as operações de inserção, remoção, e busca de valores necessitam de tempo de pior caso proporcional à altura da árvore, este limite proporcional a altura permite que árvores rubro-negras sejam eficientes no pior caso, diferentemente de árvore de busca binária.

Árvore rubro-negra



Um exemplo de árvore rubro-negra

Árvores rubro-negras

- Numa árvore binária de busca com n chaves e de altura h, as operações de busca, inserção e remoção têm complexidade de tempo $O(h)$.
- No pior caso, a altura de uma árvore binária de busca pode ser $O(n)$. No caso médio, vimos que a altura é $O(\log n)$.
- Árvores AVL, árvores 2-3 e árvores rubro-negras são alguns tipos de árvores binárias de busca ditas balanceadas com altura $O(\log n)$.
- Todas essas árvores são projetadas para busca de dados armazenados na memória principal (RAM).
- As árvores 2-3 são generalizadas para as chamadas B-árvores, que veremos mais tarde, para busca eficiente de dados armazenados em memória secundária (disco rígido).
- Veremos a árvore rubro-negra cuja altura é no máximo igual a $2\log(n+1)$.

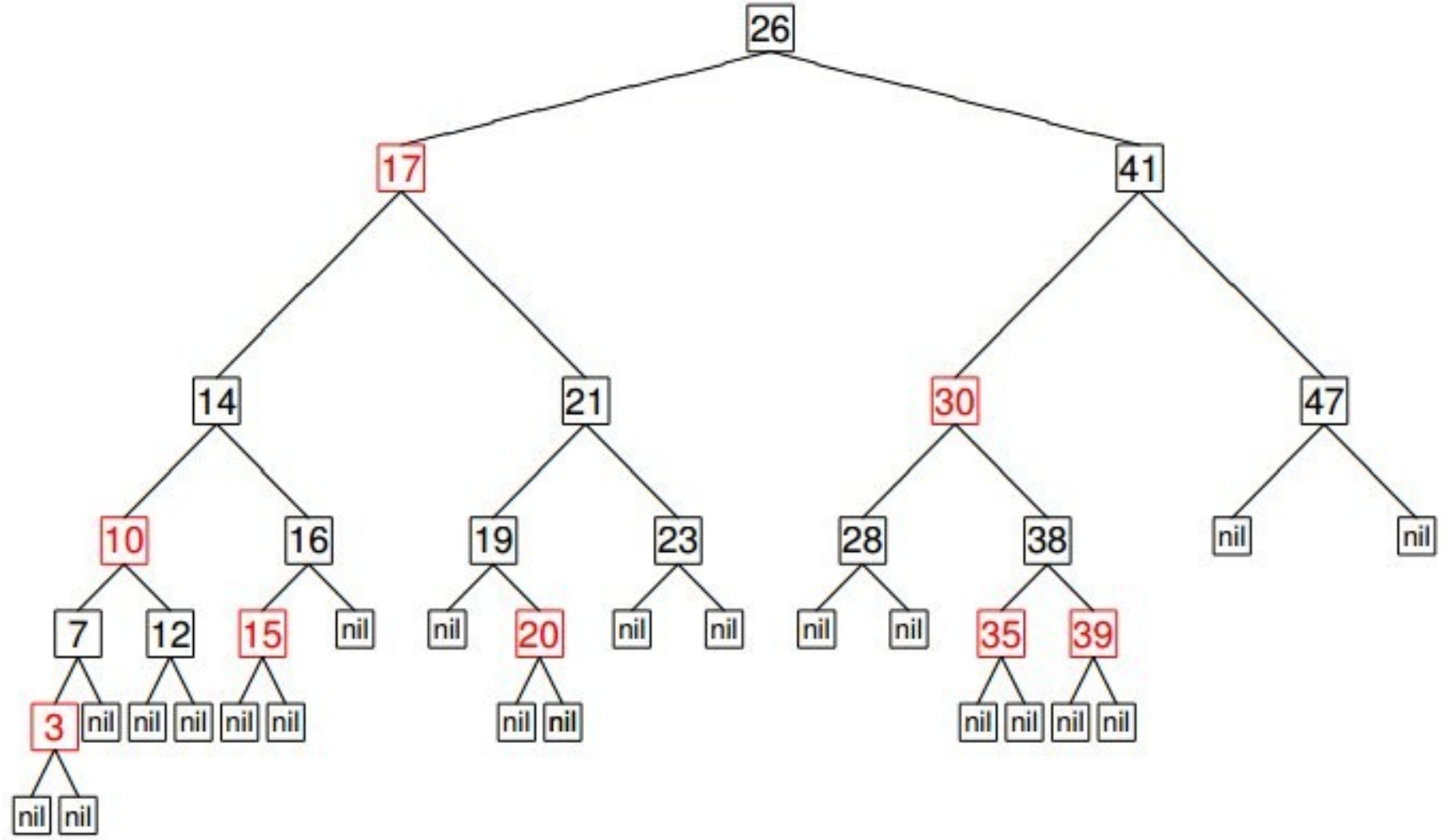
Árvores rubro-negras

Uma árvore rubro-negra é uma árvore binária de busca em que cada nó é constituído dos seguintes campos:

- cor (1 bit): pode ser vermelho ou preto.
- key (e.g. inteiro): indica o valor de uma chave.
- left, right: ponteiros que apontam para a subárvore esquerda e direita, resp.
- pai: ponteiro que aponta para o nó pai. O campo pai do nó raiz aponta para nil.

O ponteiro pai facilita a operação da árvore rubro-negra, conforme será visto a seguir.

Árvores rubro-negras



Propriedades da Árvores rubro-negras

Uma árvore rubro-negra é uma árvore binária de busca, com algumas propriedades adicionais.
Quando um nó não possui um filho (esquerdo ou direito) então vamos supor que ao invés de apontar para nil, ele aponta para um nó fictício, que será uma folha da árvore. Assim, todos os nós internos contêm chaves e todas as folhas são nós fictícios.
As propriedades da árvore rubro-negra são

1. Todo nó da árvore ou é vermelho ou é preto.
2. A raiz é preta.
3. Toda folha (nil) é preta.
4. Se um nó é vermelho, então ambos os filhos são pretos.
5. Para todo nó, todos os caminhos do nó até as folhas descendentes contêm o mesmo número de nós pretos.

Consideração Prática

Considere uma árvore rubro-negra e o ponteiro T apontando para a raiz da árvore.

- Os nós internos de uma árvore rubro-negra representam as chaves.
- As folhas são apenas nós fictícios colocados no lugar dos ponteiros nil.
- Assim, dedicaremos nossa atenção aos nós internos.
- Por economia de espaço, ao invés de representar todas as folhas, podemos fazer todos os ponteiros nil apontarem para uma mesma folha, chamada nó sentinela, que será indicado por $nil[T]$.

Altura de uma árvores rubro-negra

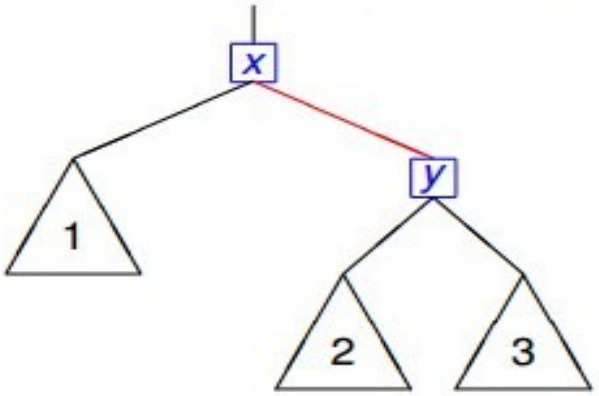
A altura h de uma árvore rubro-negra de n chaves ou nós internos é no máximo $2\log(n+1)$.

A prova é por indução. Ver detalhes no livro de Cormen et al.

- Esse resultado mostra a importância e utilidade de uma árvore rubro-negra, pois veremos que a busca, inserção e remoção têm complexidade de tempo de $O(h) = O(\log n)$.
- Inserções e remoções feitas numa árvore rubro-negra pode modificar a sua estrutura. Precisamos garantir que nenhuma das propriedades de árvore rubro-negra seja violada.
- Para isso podemos ter que mudar a estrutura da árvore e as cores de alguns dos nós da árvore. A mudança da estrutura da árvore é feita por dois tipos de rotações em ramos da árvore:
  - left-rotate e
  - right-rotate.

Rotação: left-rotate e right-rotate

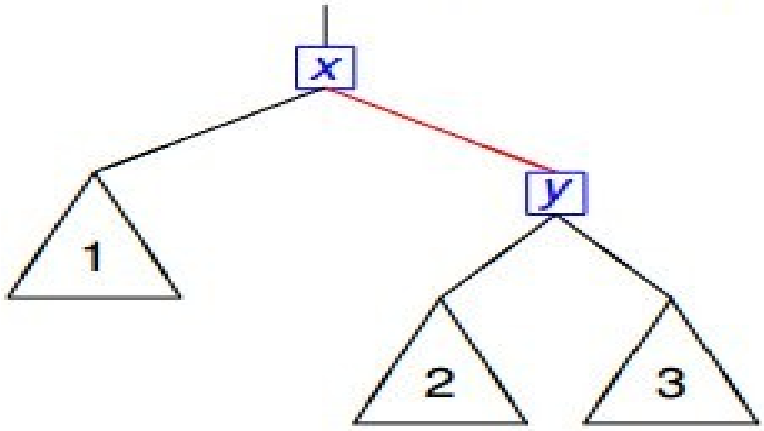
Seja uma árvore binária apontada por T



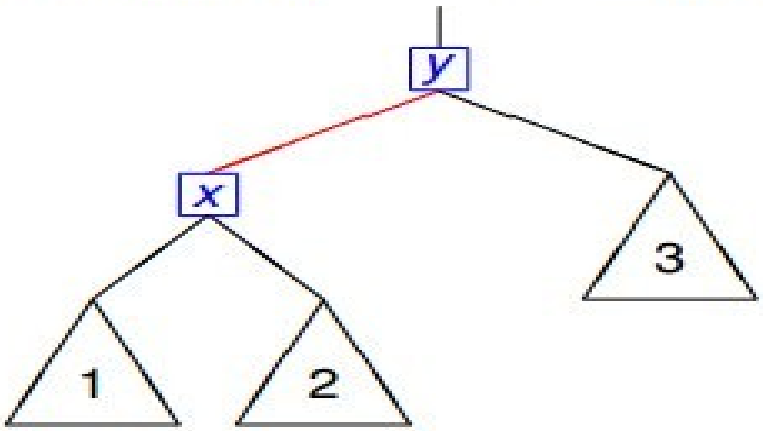
right-rotate(T, y) ⇑ ⇓ left-rotate(T, x)

Rotação: left-rotate e right-rotate

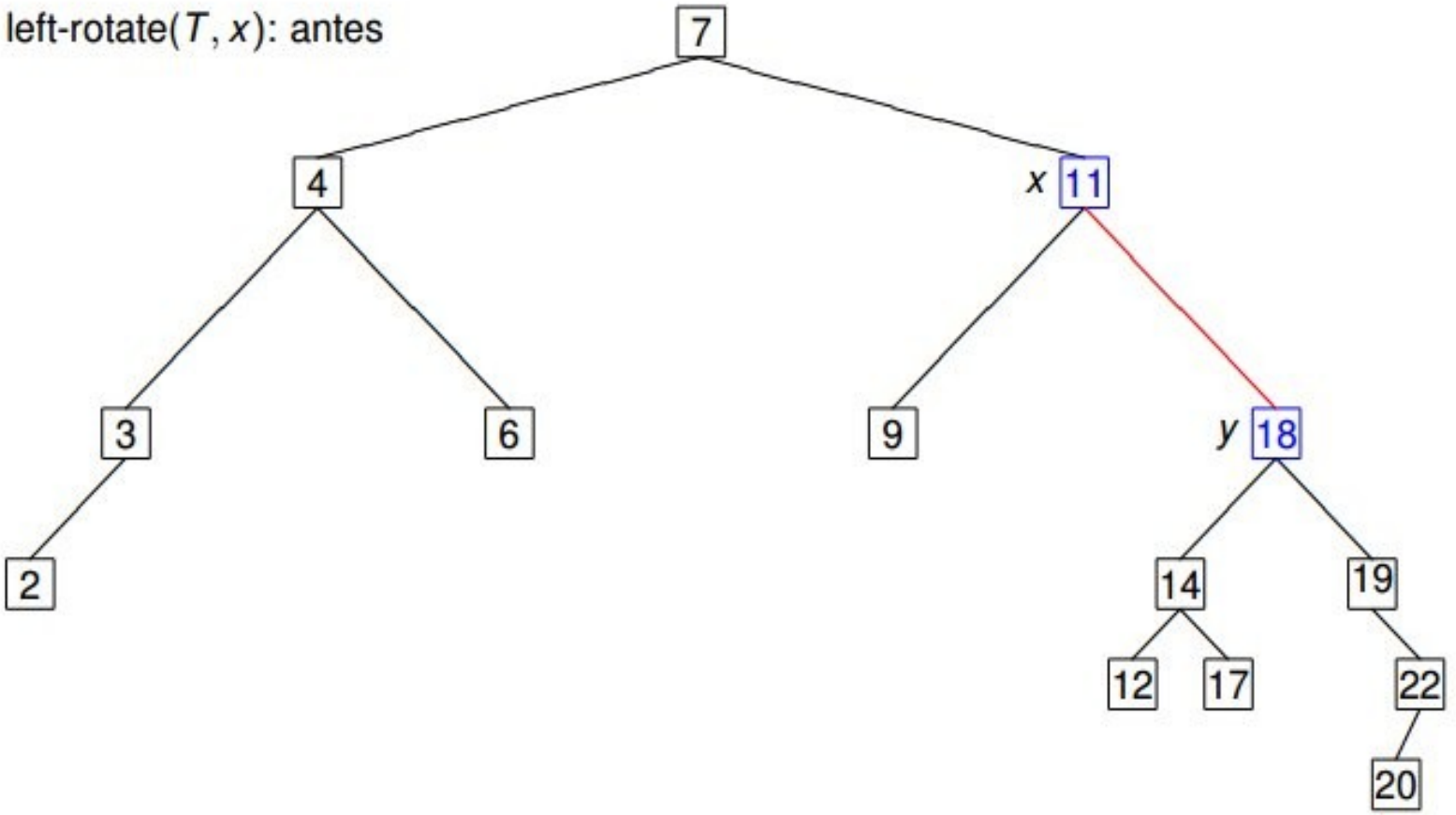
Seja uma árvore binária apontada por T



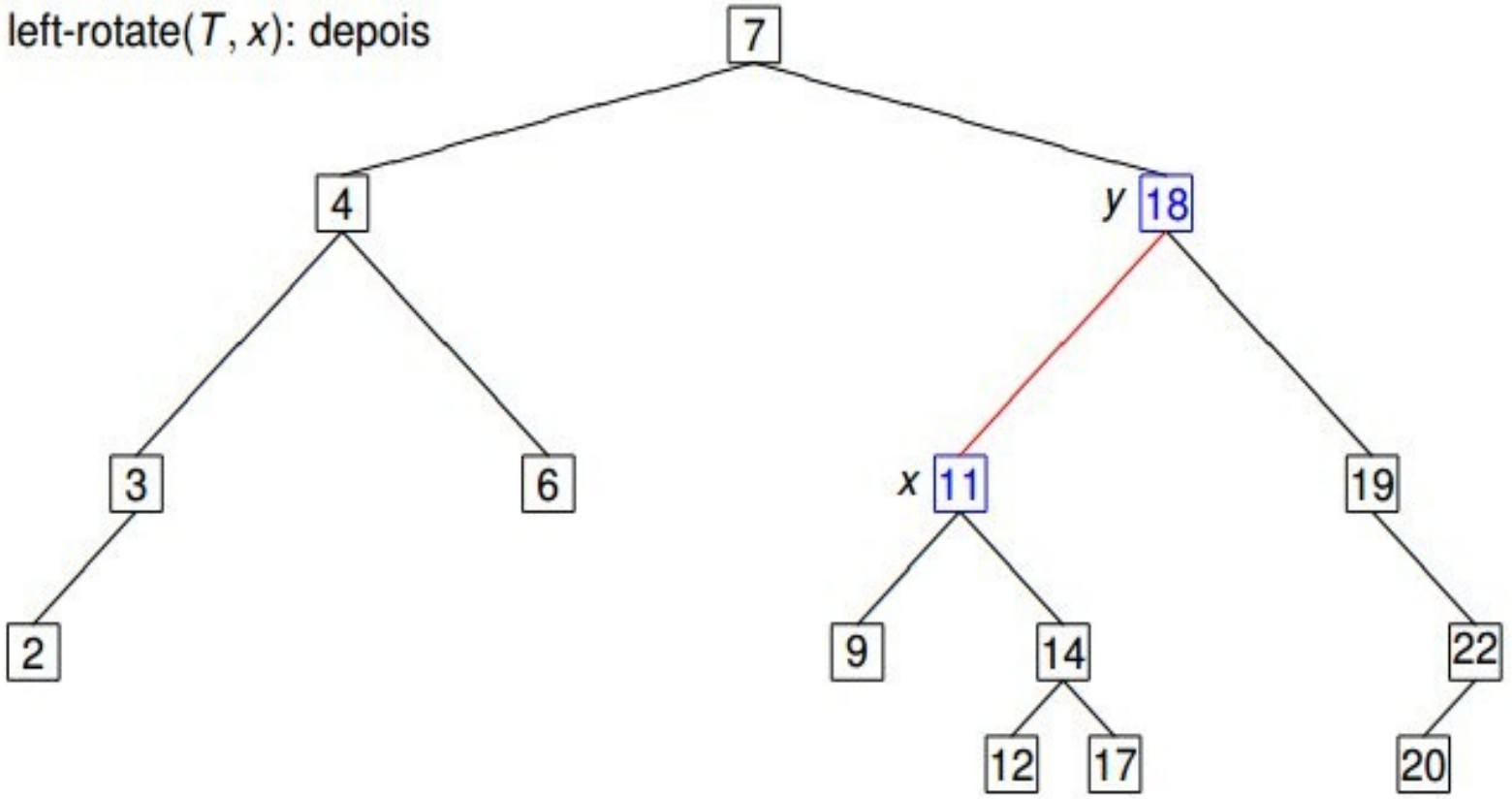
right-rotate(T, y) ⇑ ⇓ left-rotate(T, x)



Exemplo de left-rotate



Exemplo de left-rotate



Complexidade do algoritmo left-rotate

- O algoritmo de rotação (left-rotate e right-rotate) muda alguns ponteiros da árvore, preservando a propriedade de árvore binária de busca.
- Leva tempo constante $O(1)$.

Corretura do algoritmo de inserção

O algoritmo de inserção não viola as propriedades de árvore rubro-negra.

1. Todo nó da árvore ou é vermelho ou é preto.
2. A raiz é preta.
3. Toda folha (nil) é preta.
4. Se um nó é vermelho, então ambos os filhos são pretos.
5. Para todo nó, todos os caminhos do nó até as folhas descendentes contêm o mesmo número de nós pretos.

O algoritmo de inserção substitui um nó sentinela (preto) por um novo nó interno vermelho *z* contendo o valor novo inserido. Este nó aponta, por sua vez, a dois nós sentinela (preto), à esquerda e à direita.

Após a inserção, mas antes de executar RB-insert-fixup, apenas as propriedades 2 e 4 podem estar violadas: a propriedade 2 é violada se *z* é a raiz e a propriedade 4 é violada se o pai de *z* é vermelho. O algoritmo RB-insert-fixup repara essa eventual violação.

O algoritmo RB-insert-fixup

No início da iteração do laço **while** temos 3 invariantes:

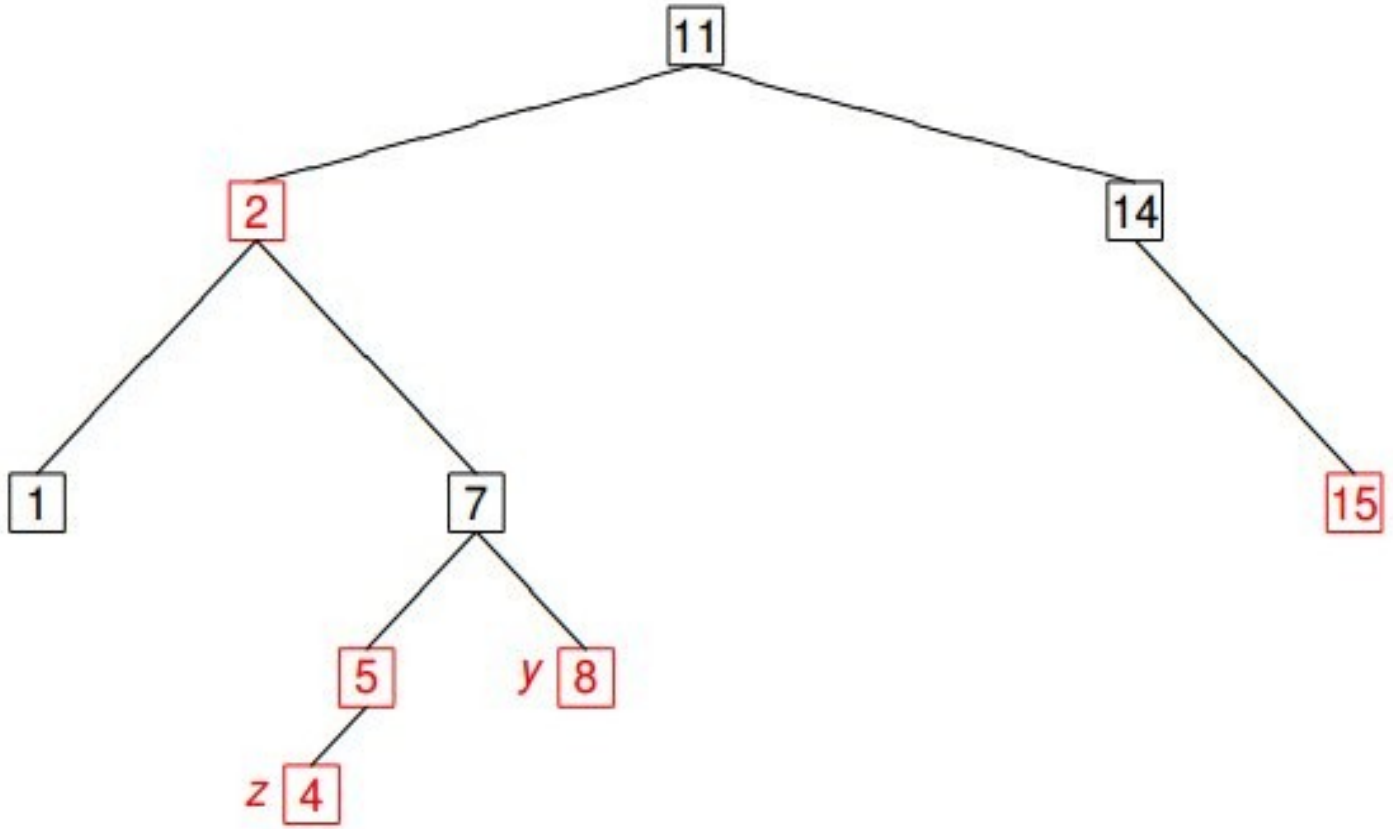
1. Nó z é vermelho.
2. Se $pai[z]$ é a raiz, então $pai[z]$ é preto.
3. Apenas uma propriedade (2 ou 4) pode estar violada. Se for a propriedade 2, então é porque z (vermelho) é a raiz. Se a propriedade violada é a 4, então é porque z e $pai[z]$ são ambos vermelhos.

Há três casos a considerar quando z e $pai[z]$ são vermelhos. Existe $pai[pai[z]]$ pois $pai[z]$ sendo vermelho não pode ser a raiz.

- Caso 1: z tem um tio y vermelho.
- Caso 2: z tem um tio y preto e z é filho direito.
- Caso 3: z tem um tio y preto e z é filho esquerdo.

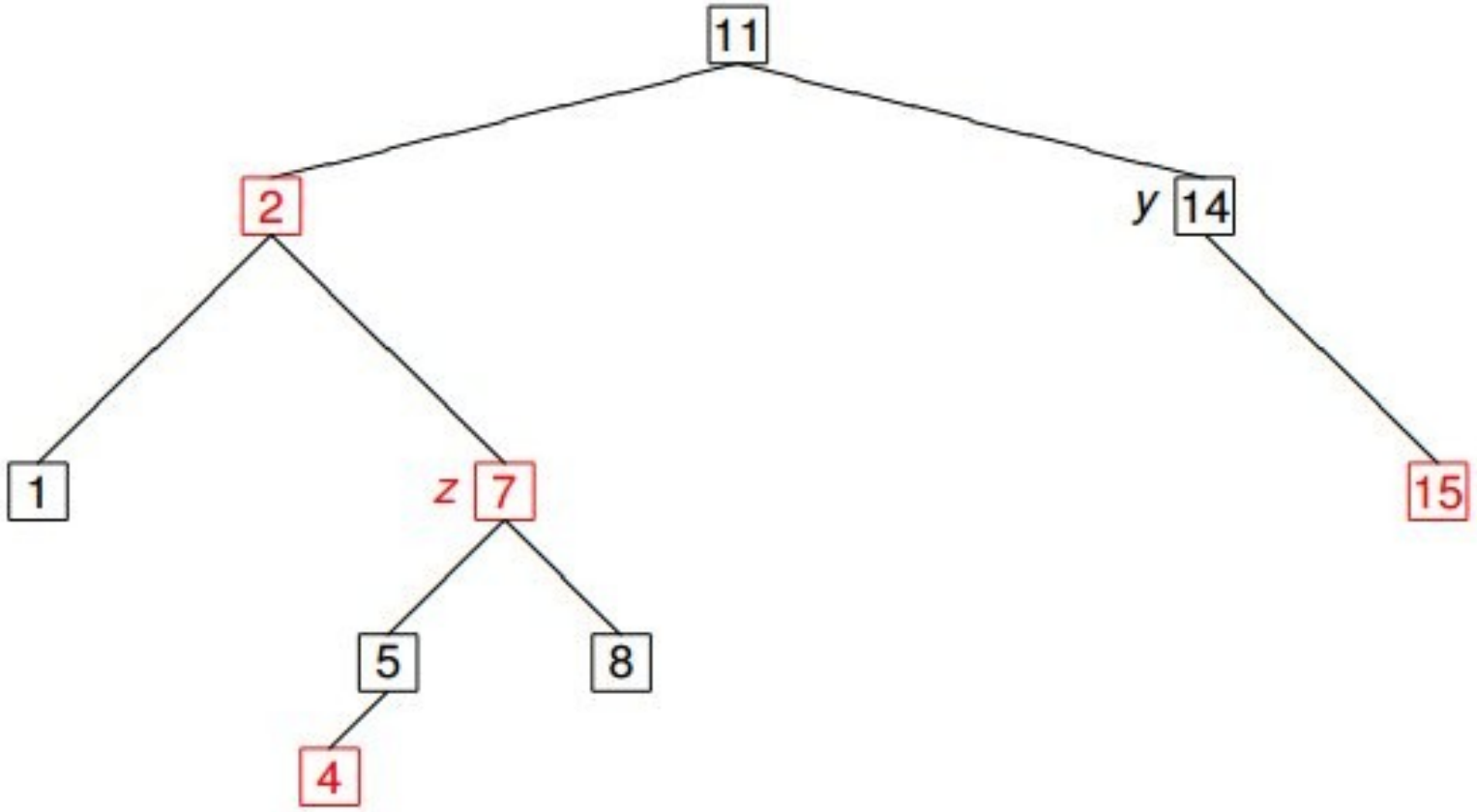
No caso 1, mudamos algumas cores e subimos z para ser $pai[pai[z]]$ e assim sucessivamente, podendo chegar até a raiz. No caso 2 e 3 mudamos algumas cores e fazemos uma ou duas rotações.

Caso 1 : Z tem um tio Y vermelho



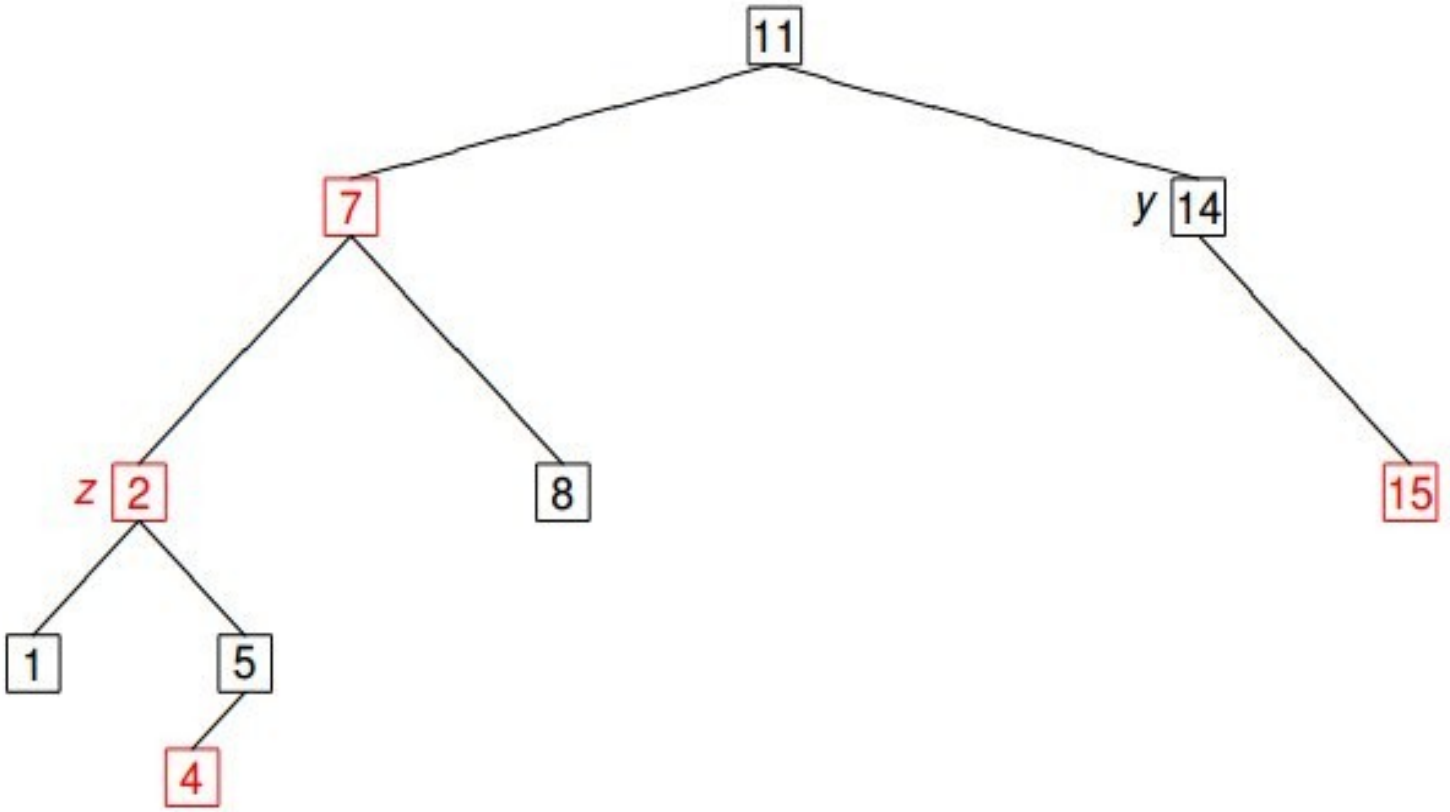
Colorimos *pai*[*z*] e tio *y* pretos e *pai*[*pai*[*z*]] vermelho que passa a ser o novo *z*

Caso 2 : Z tem um tio preto e Z é filho direito



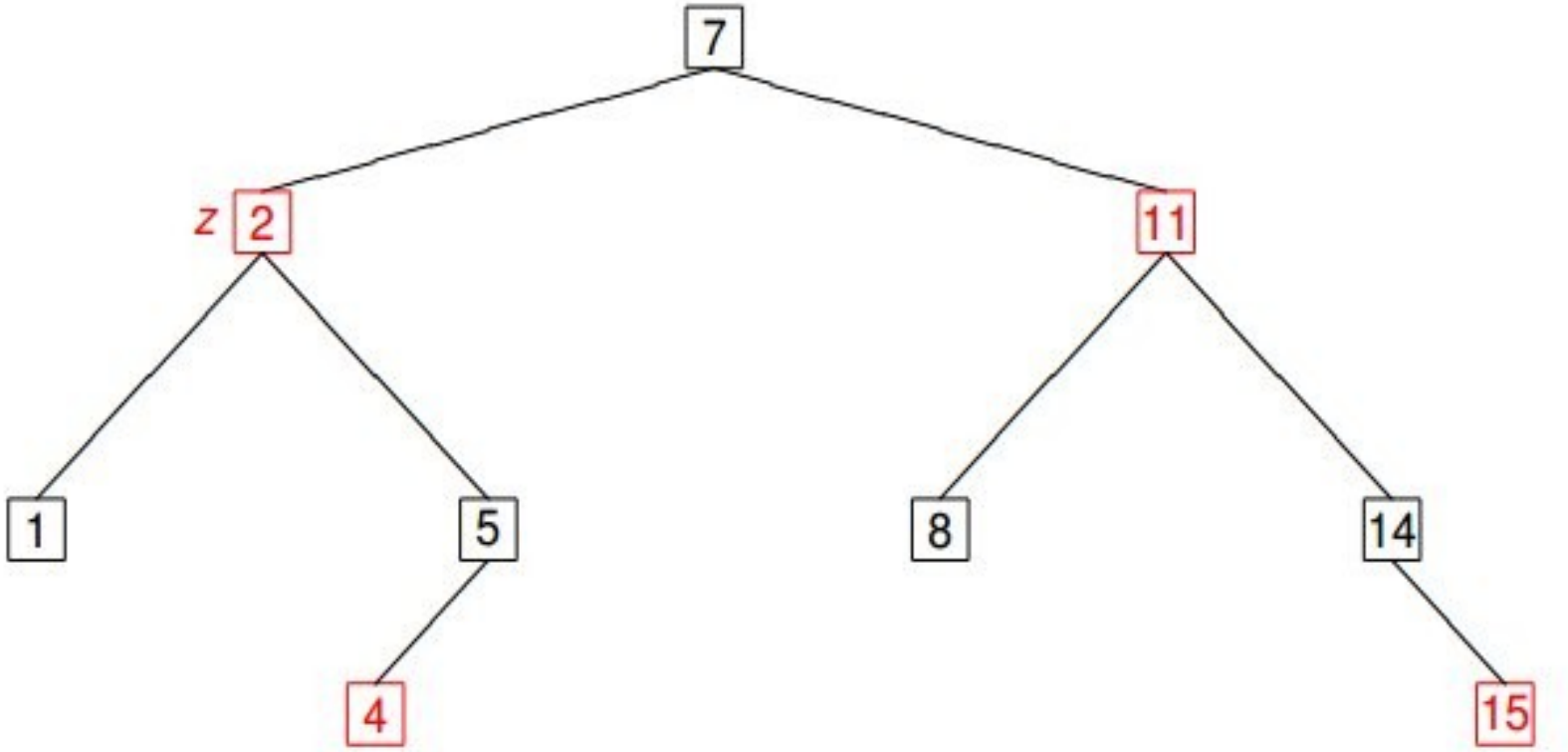
Executamos left-rotate no *z* transformando para caso 3

Caso 3 : Z tem um tio Y preto e Z é filho esquerdo



Colorimos *pai*[*z*] preto e *pai*[*pai*[*z*]] vermelho e executamos right-rotate no *pai*[*pai*[*z*]]

Árvore rubro-negra final



Complexidade do algoritmo de inserção

- A altura de uma árvore rubro-negra de n chaves é $O(\log n)$.
- O algoritmo RB-insert (linhas 1 a 23), sem contar RB-insert-fixup (linha 24), é $O(\log n)$.
- No algoritmo RB-insert-fixup, o laço **while** repete somente se caso 1 é executado e neste caso o ponteiro z "sobe" de dois níveis na árvore. O número de vezes que o laço pode ser executado é portanto $O(\log n)$.
- Se o caso 2 for executado, então o caso 3 será executado em seguida. Após a execução do caso 3, $pai[z]$ fica preto e o laço **while** termina.
- O algoritmo de inserção é portanto $O(\log n)$.

Remoção da árvore rubro-negra

- O algoritmo de remoção RB-delete remove o nó de forma análoga ao algoritmo de remoção em uma árvore binária de busca.
- No final da remoção o algoritmo chama RB-delete-fixup que, caso necessário, muda as cores de alguns nós e re-estrutura a árvore por meio de rotações. O algoritmo é um pouco mais complexo que a inserção e há 4 casos a considerar.
- O algoritmo de remoção é $O(\log n)$.
- Omitimos os detalhes que podem ser consultados no livro de Cormen et al.