Árvore Rubro-Negra

Siang Wun Song - Universidade de São Paulo - IME/USP

MAC 5710 - Estruturas de Dados - 2008

Referência bibliográfica

Os slides sobre este assunto são parcialmente baseados no capítulo 13 do livro

- T. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, second edition, The MIT Press, 2005.

Árvores binárias de busca balanceadas

- 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 $2\log(n+1)$.

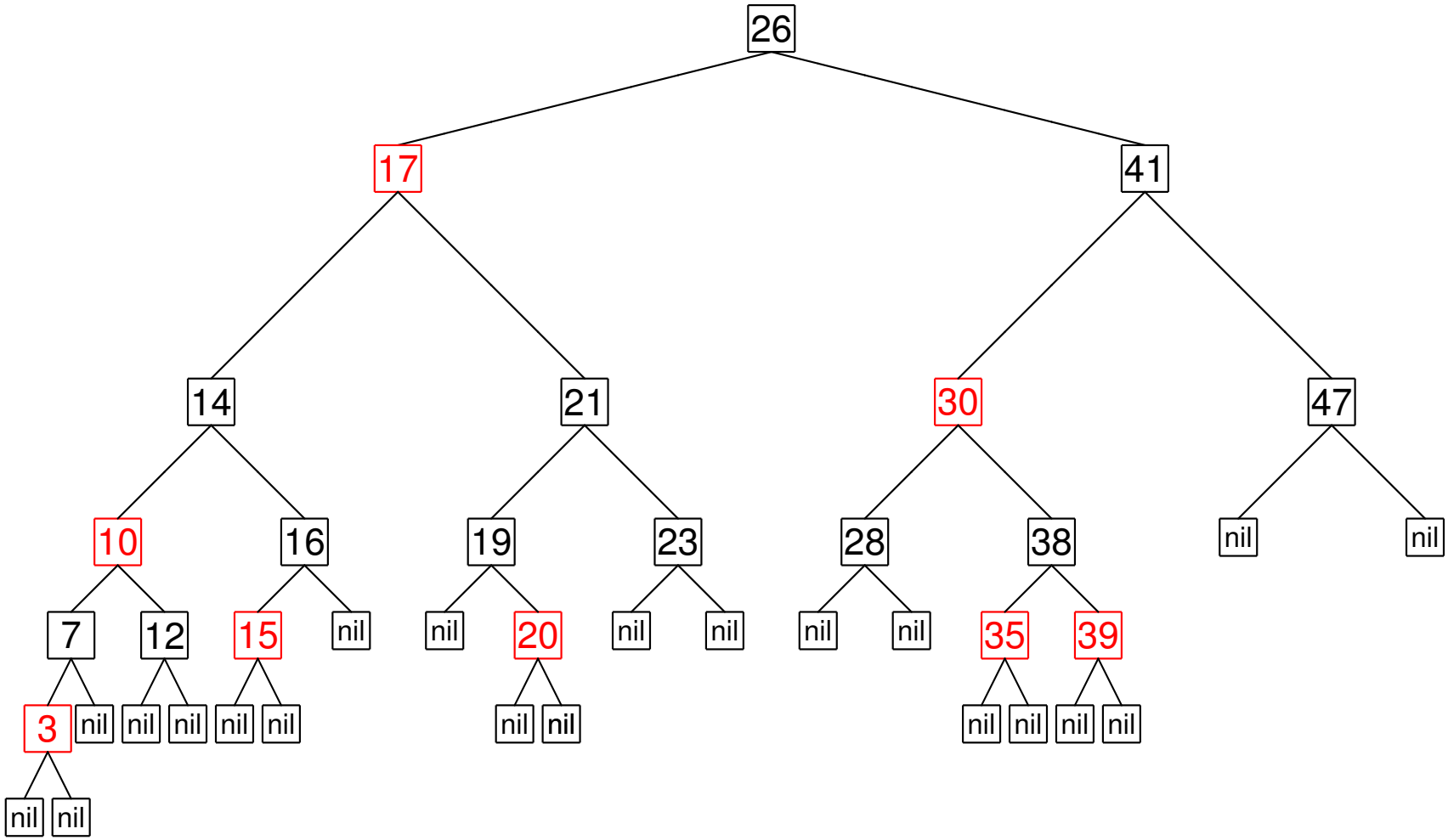
Árvore rubro-negra

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.

Uma árvore rubro-negra



Propriedades da árvore rubro-negra

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 árvore rubro-negra

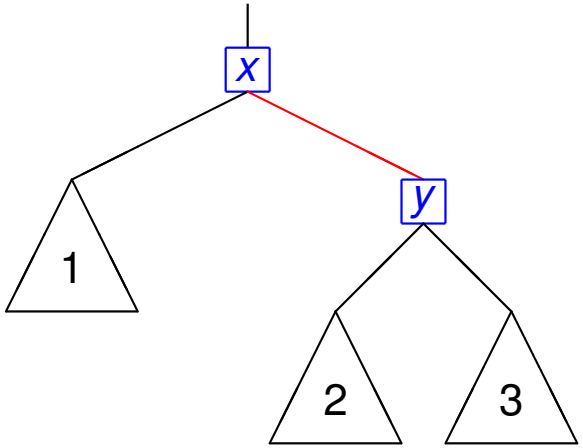
A altura h de uma árvore rubro-negra de n chaves ou nós internos é no máximo $2\log(n+1)$.

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

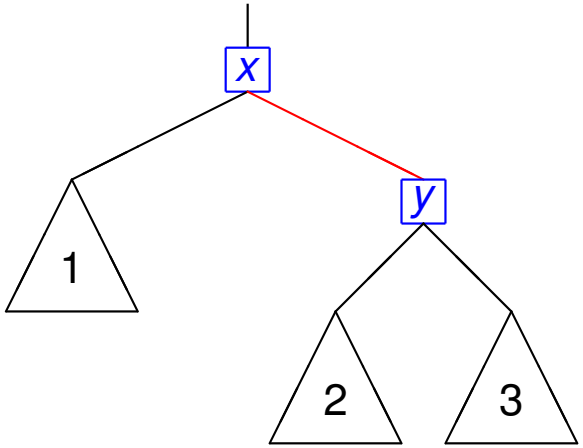
Seja uma árvore binária apontada por T



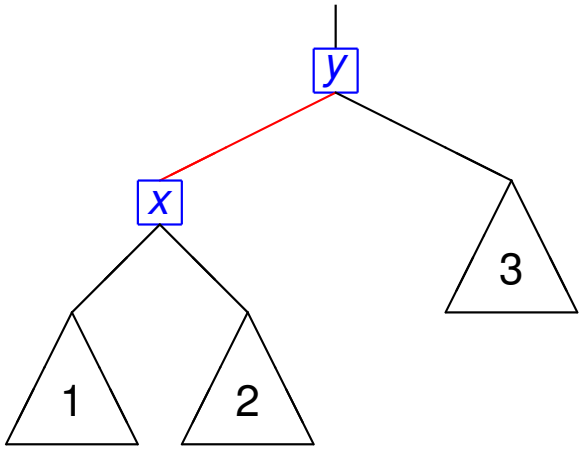
right-rotate(T, y) ⇑ ⇓ left-rotate(T, x)

Rotação: left-rotate e right-rotate

Seja uma árvore binária apontada por T

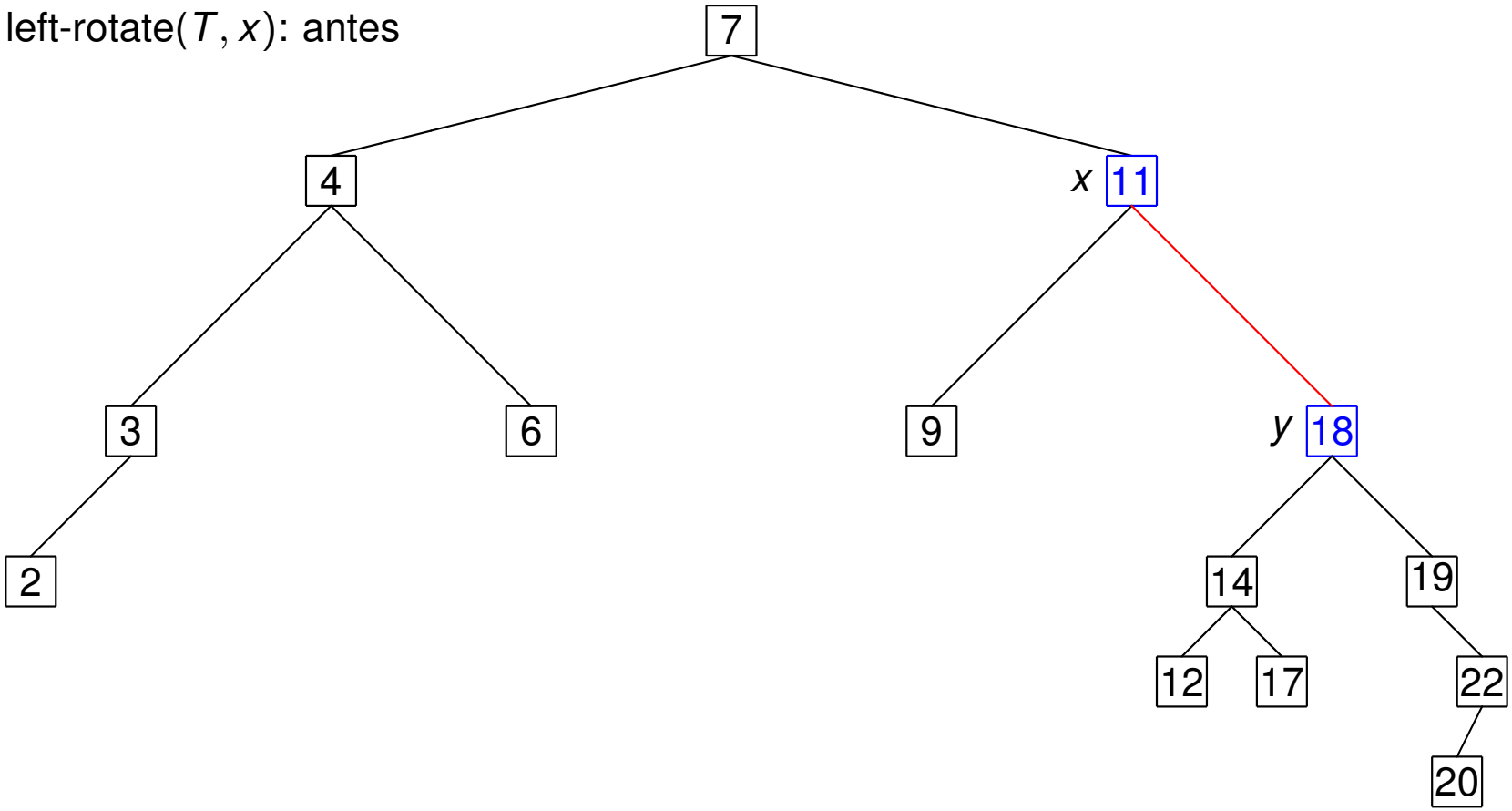


right-rotate(T, y) $\Uparrow$ $\Downarrow$ left-rotate(T, x)



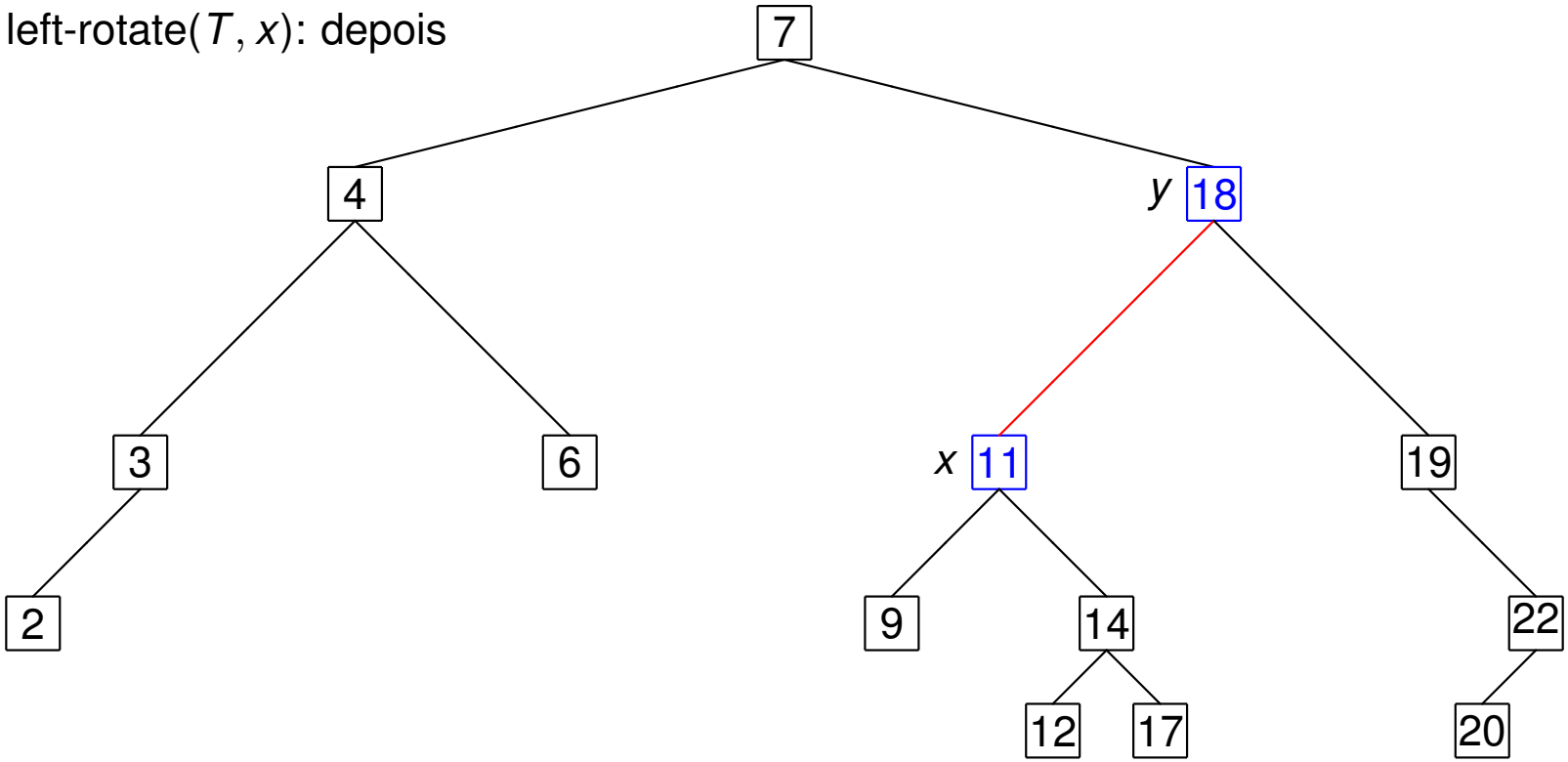
Exemplo de left-rotate

left-rotate(T, x): antes



Exemplo de left-rotate

left-rotate(T, x): depois



Algoritmo left-rotate

left-rotate(T, x):

```
 1: y ← rigft[x]
 2: right[x] ← left[y]
 3: if left[y] ≠ nil[T] then
 4:     pai[left[y]] ← x
 5: end if
 6: pai[y] ← pai[x]
 7: if pai[x] = nil[T] then
 8:     T ← y
 9: else
10:     if x = left[pai[x]] then
11:         left[pai[x]] ← y
12:     else
13:         right[pai[x]] ← y
14:     end if
15: end if
16: left[y] ← x
17: pai[x] ← y
```

O algoritmo right-rotate(T, y) é análogo e não será apresentado.

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 $O(1)$.

Algoritmo de inserção na árvore rubro-negra

RB-insert(T, z):

```
1:  y ← nil[T]
2:  x ← T
3:  while x ≠ nil[T] do
4:      y ← x
5:      if key[z] < key[x] then
6:          x ← left[x]
7:      else
8:          x ← right[x]
9:      end if
10: end while
11: pai[z] ← y
12: if y = nil[T] then
13:     T ← z
14: else
15:     if key[z] < key[y] then
16:         left[y] ← z
17:     else
18:         right[y] ← z
19:     end if
20: end if
21: left[z] ← nil[T]
22: right[z] ← nil[T]
23: cor[z] ← vermelho
24: RB-insert-fixup(T, z)
```

O algoritmo RB-insert-fixup(T, z) re-estrutura a árvore, caso necessário, através da mudança de cores de alguns nós e das operações de rotação.

Algoritmo para re-estruturar a árvore

RB-insert-fixup(T, z):

```
 1: while cor[pai[z]] = vermelho do
 2:     if pai[z] = left[pai[pai[z]]] then
 3:         y ← right[pai[pai[z]]]
 4:         if cor[y] = vermelho then
 5:             {caso1 :}cor[pai[z]] ← preto
 6:             cor[y] ← preto
 7:             cor[pai[pai[z]]] ← vermelho
 8:             z ← pai[pai[z]]
 9:         else
10:             if z = right[pai[z]] then
11:                 {caso2 :}z ← pai[z]
12:                 left-rotate(T, z)
13:             end if
14:             {caso3 :}cor[pai[z]] ← preto
15:             cor[pai[pai[z]]] ← vermelho
16:             right-rotate(T, pai[pai[z]])
17:         end if
18:     else
19:         {o mesmo do caso then (linhas 3 a 16) trocando entre si right com left}
20:     end if
21: end while
22: cor[T] ← preto
```

Corretude 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 que faz o algoritmo RB-insert-fixup

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

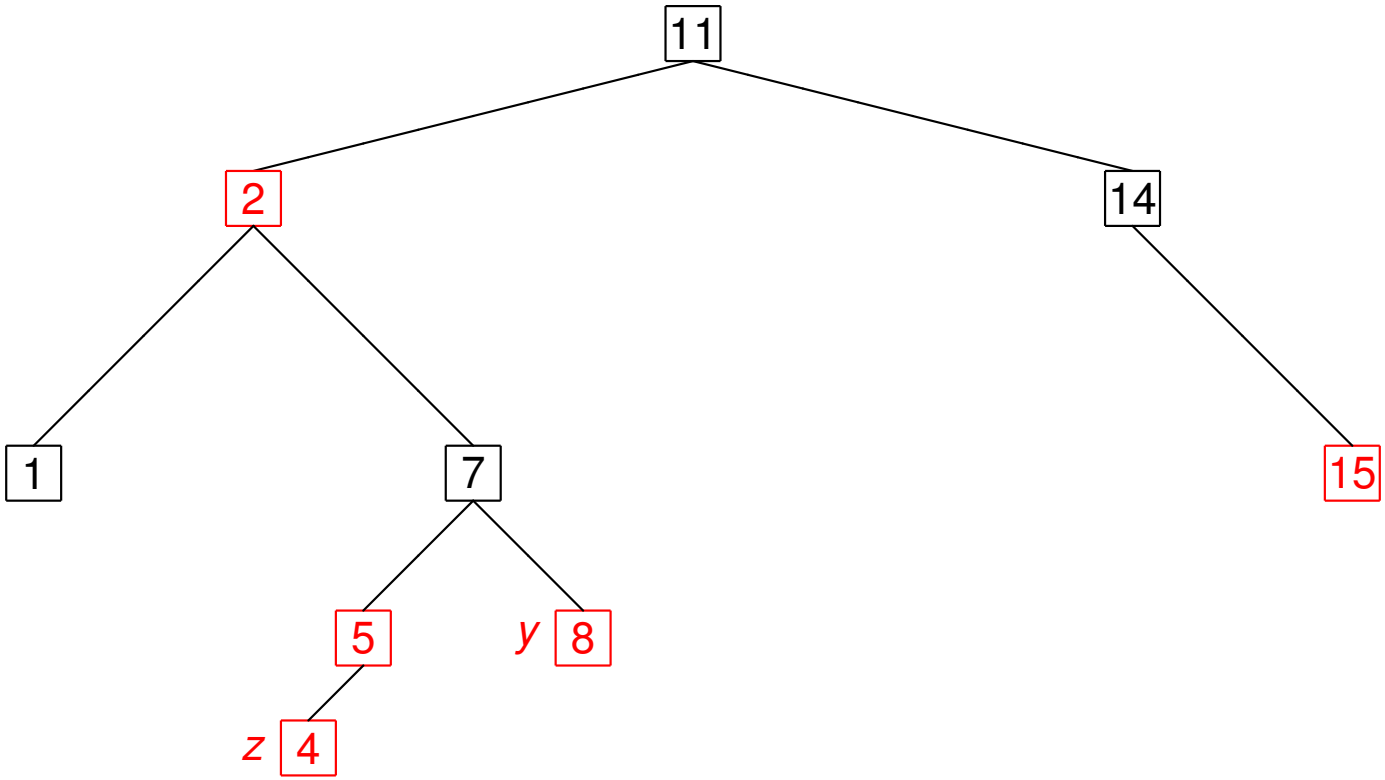
1. Nó z é vermelho.
2. Se $pai[z]$ é a raiz, então $pai[z]$ é preto.
3. Apenas uma propriedade (2 ou 4) pode estar violada. Se for a propriedade 2, então é porque z (vermelho) é a raiz. Se a propriedade violada é a 4, então é porque z e $pai[z]$ são ambos vermelhos.

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.

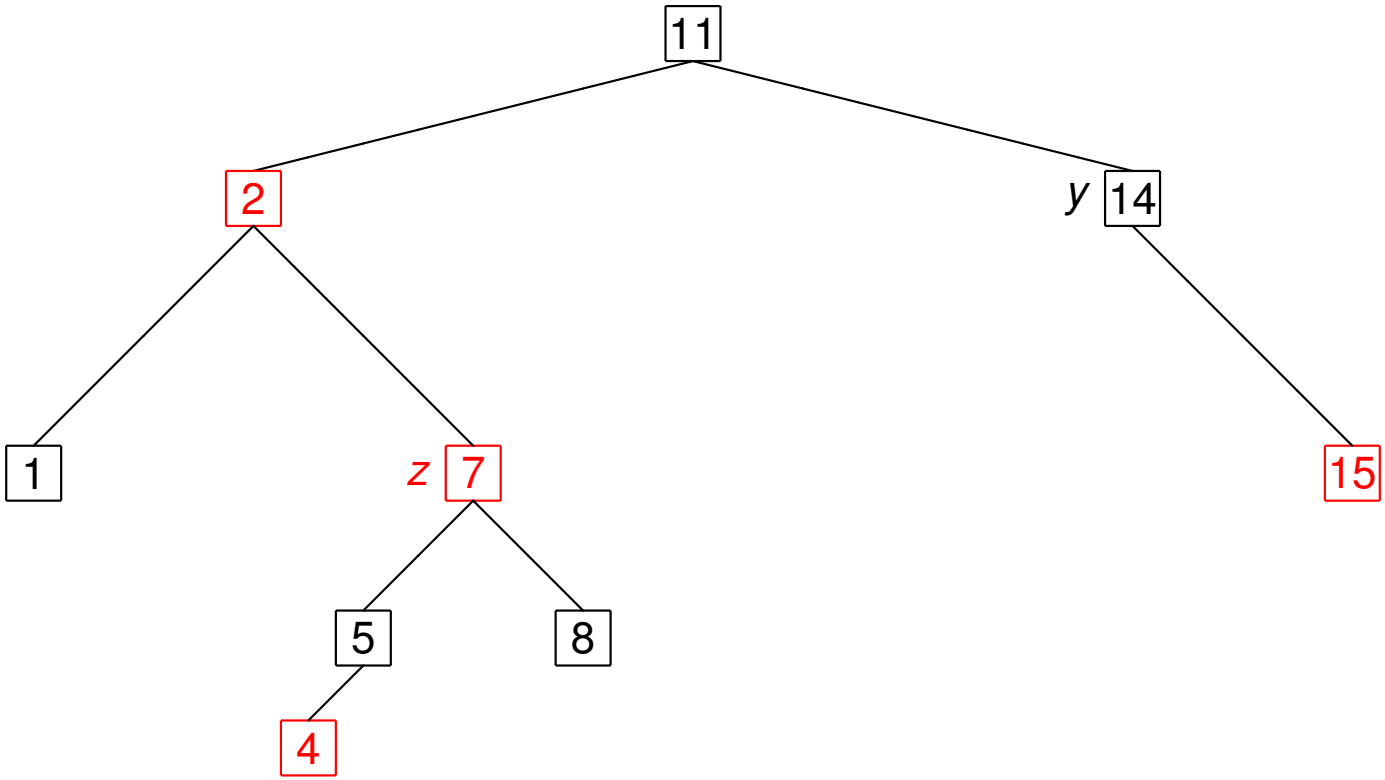
No caso 1, mudamos algumas cores e subimos z para ser $pai[pai[z]]$ e assim sucessivamente, podendo chegar até a raiz. No caso 2 e 3 mudamos algumas cores e fazemos uma ou duas rotações.

Caso 1: z tem um tio y vermelho



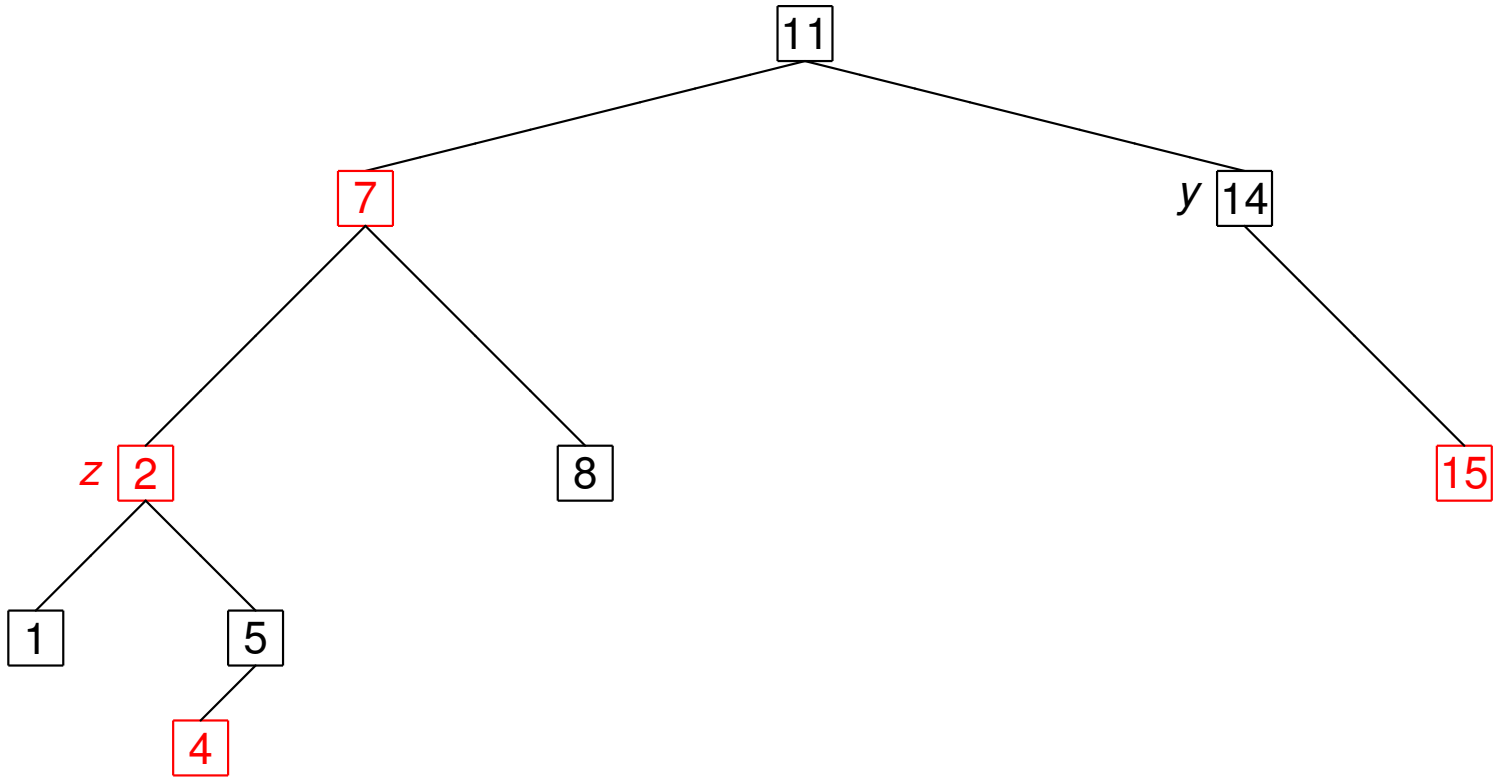
Colorimos $pai[z]$ e tio y pretos e $pai[pai[z]]$ vermelho que passa a ser o novo z

Caso 2: z tem um tio y preto e z é filho direito



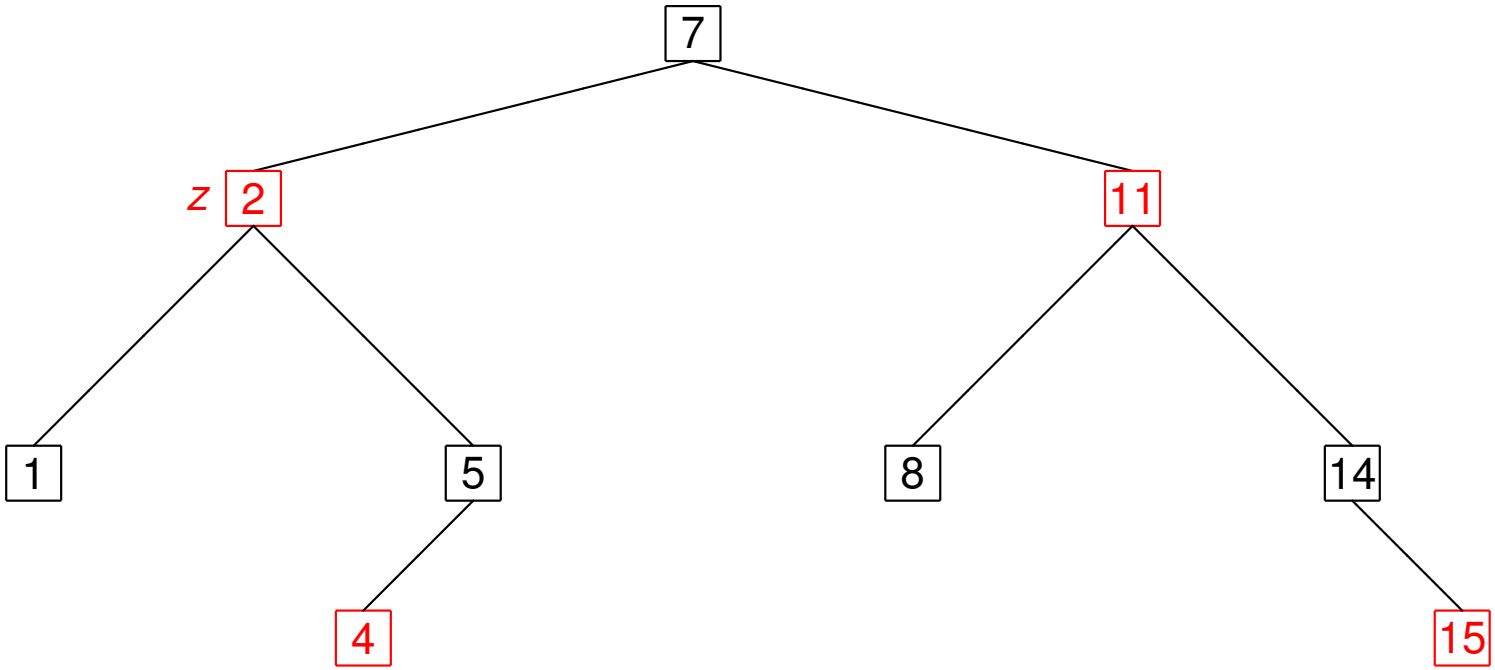
Executamos left-rotate no z transformando para caso 3

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 $pai[pai[z]]$

A 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.