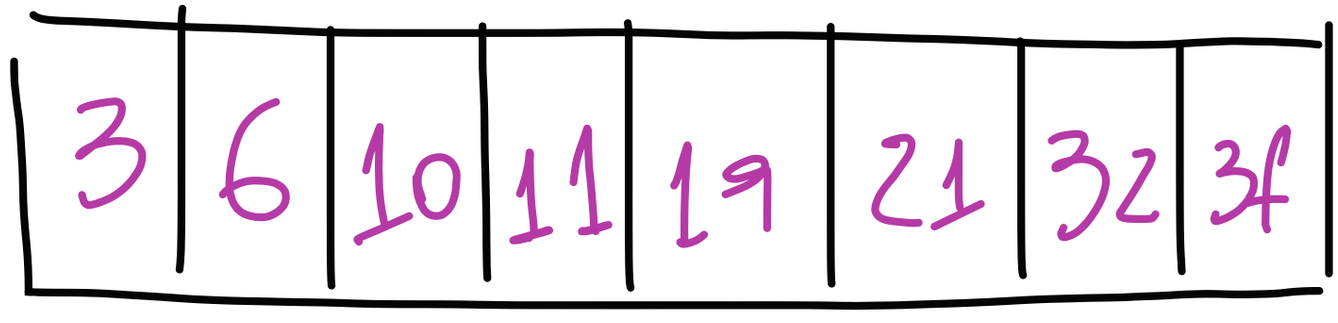
Árvores binárias de busca balanceadas:

ênfase em rubro-negras

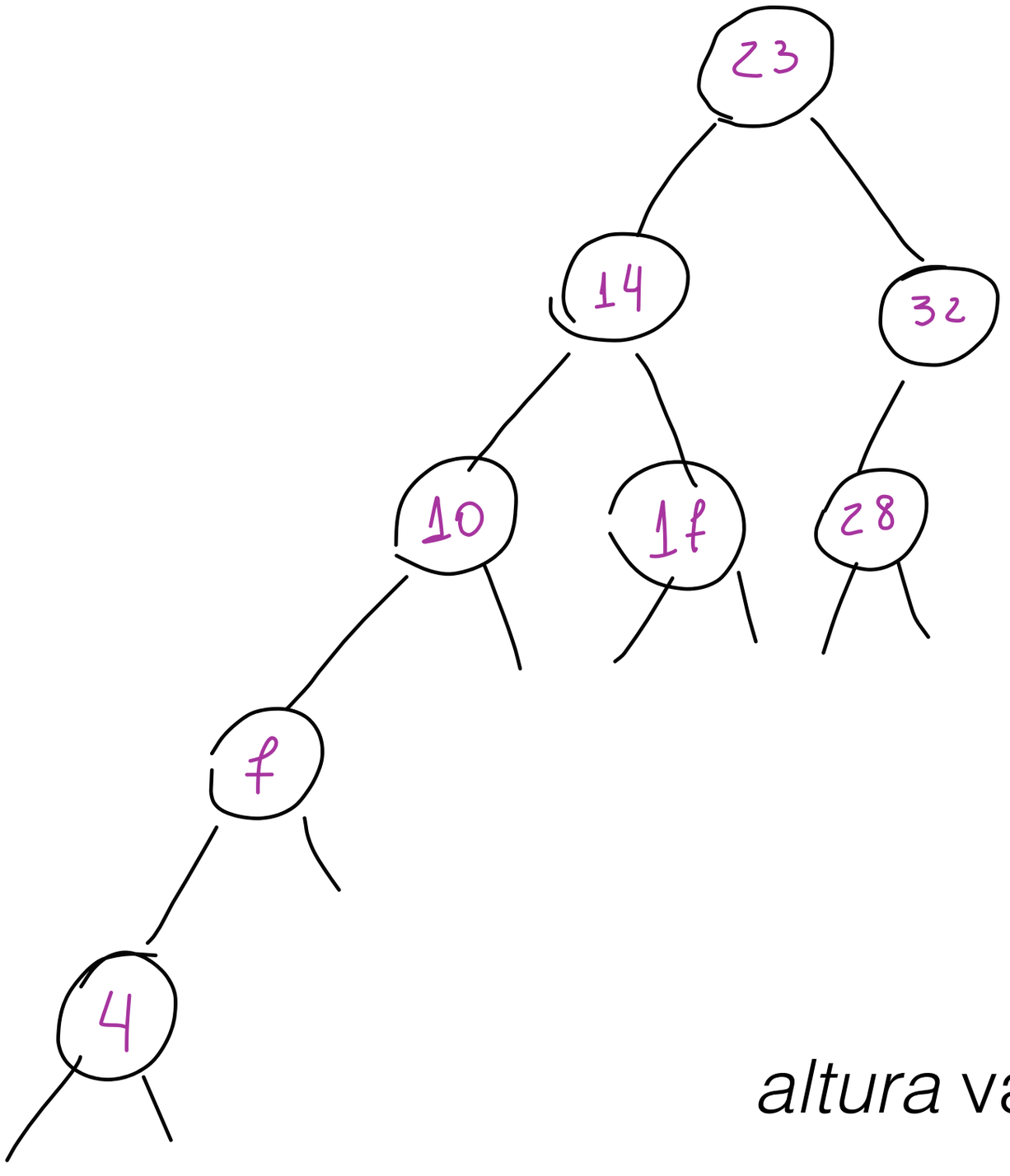
Giovani Chiachia

vetores ordenados



operações	tempo
busca	O(log n)
seleção	O(1)
min/max	O(1)
pred/sucessor	O(1)
rank	O(log n)
impressão em ordem	O(n)
inserções/remoções	O(n)

árvores binárias de busca (ABBs)



qual o tempo
no pior caso
da operação de busca
(ou inserção) em
uma ABB com *n nós*?

- ☐ $\Theta(1)$
- ☐ $\Theta(\log n)$
- ☐ $\Theta(altura)$
- ☐ $\Theta(n)$

altura varia entre ~log *n* e ~*n*

cenário

dá para fazer melhor?

operações	vetor ordenado	arv. binárias de busca (ABBs)	ABBs balanceadas
busca	O(log n)	O(altura)	O(log *n*)
seleção	O(1)	O(altura)	O(log *n*)
min/max	O(1)	O(altura)	O(log *n*)
pred/sucessor	O(1)	O(altura)	O(log *n*)
rank	O(log n)	O(altura)	O(log *n*)
imp. em ordem	O(n)	O(n)	O(*n*)
ins./remoções	O(n)	O(altura)	O(log *n*)

aplicações normalmente são dinâmicas

ABBs balanceadas

quando usar?

conjunto rico de operações
dados dinâmicos
garantia de desempenho

quando não usar?

dados mudam pouco
não requer todas essas operações

árvores balanceadas

ideia: garantir que a altura da árvore seja sempre $O(\log n)$

binárias

árvores AVL — Adelson-Velskii e Landis, 1962

árvores rubro-negras — Guibas e Sedgewick, 1978

árvores *splay* — Sleator e Tarjan, 1985

outras

árvores 2-3 — Hopcroft, 1970

árvores B — Bayer e McCreight, 1972

árvores rubro-negras

invariantes adicionais

1 - todo nó será vermelho ou negro (1 bit)

2 - a *raiz* da árvore será sempre negra

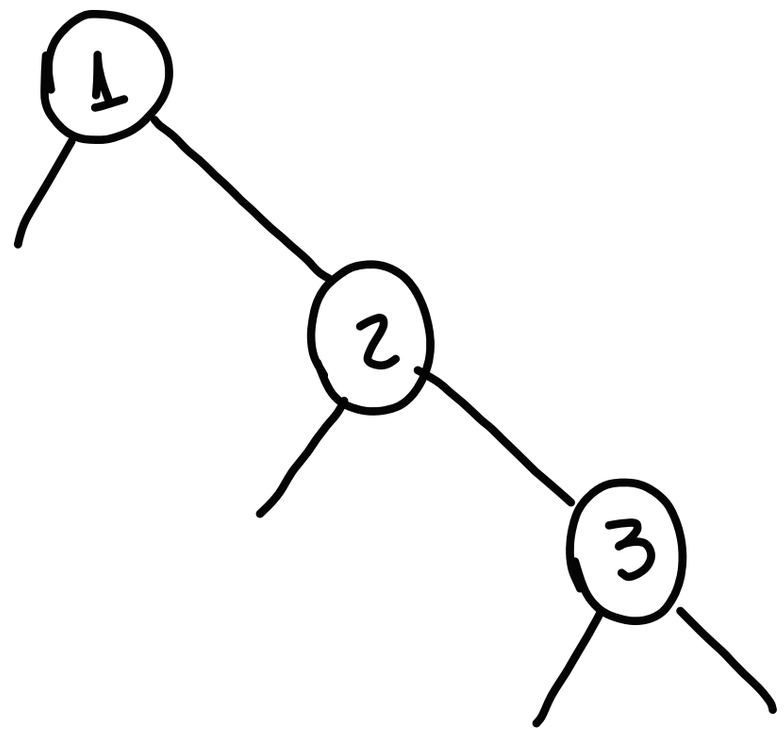
3 - não é permitido dois nós vermelhos consecutivos

4 - todos os caminhos da *raiz* até uma subárvore nula deve ter o mesmo número de nós negros

árvores rubro-negras

exemplos

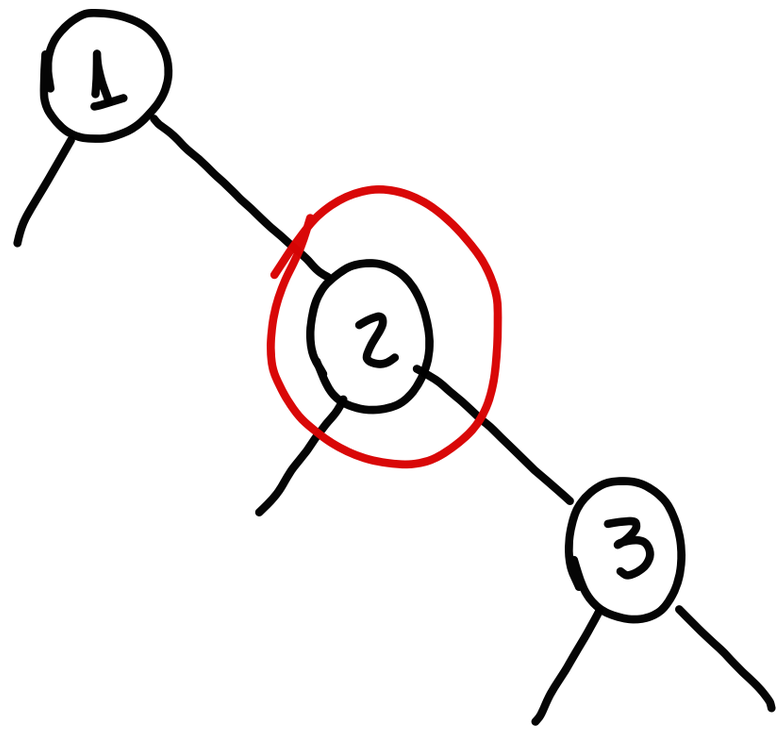
uma árvore desbalanceada com três nós não pode ser rubro-negra



árvores rubro-negras

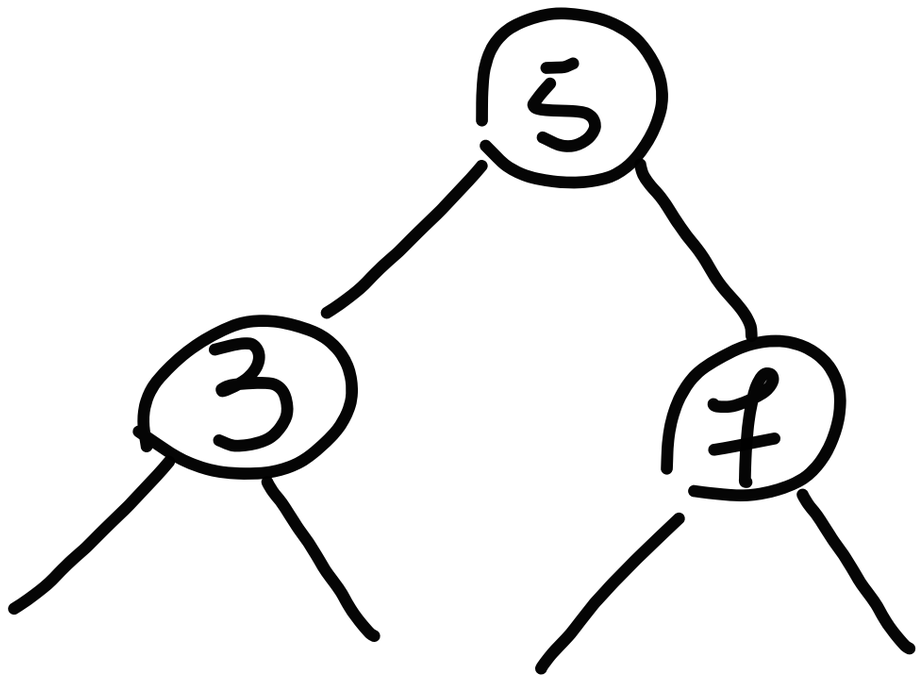
exemplos

uma árvore desbalanceada com três nós não pode ser rubro-negra



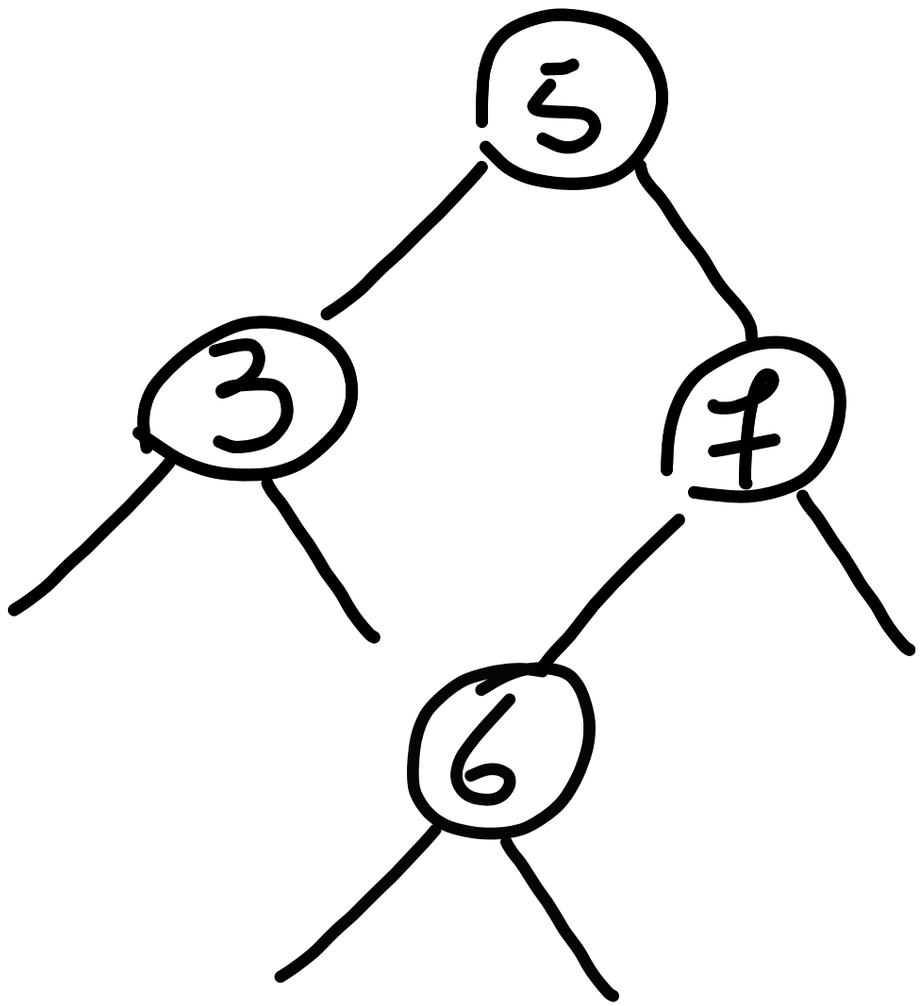
árvores rubro-negras

exemplos



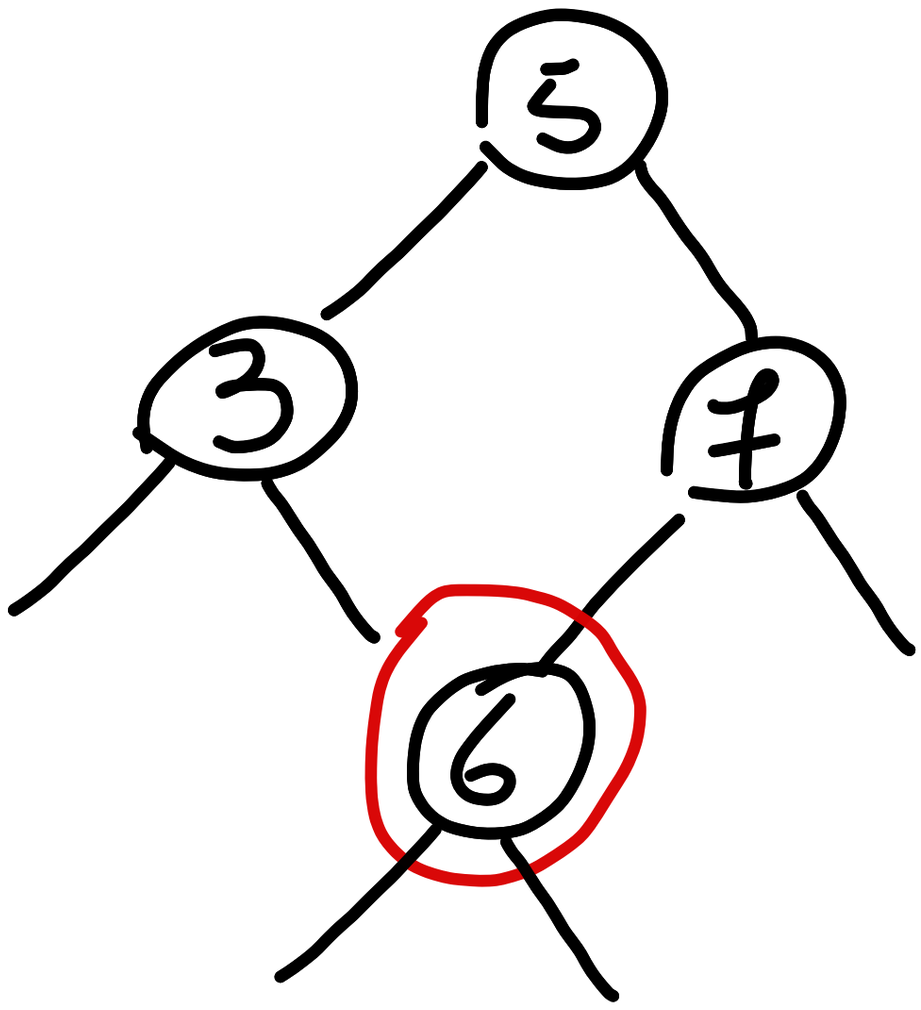
árvores rubro-negras

exemplos



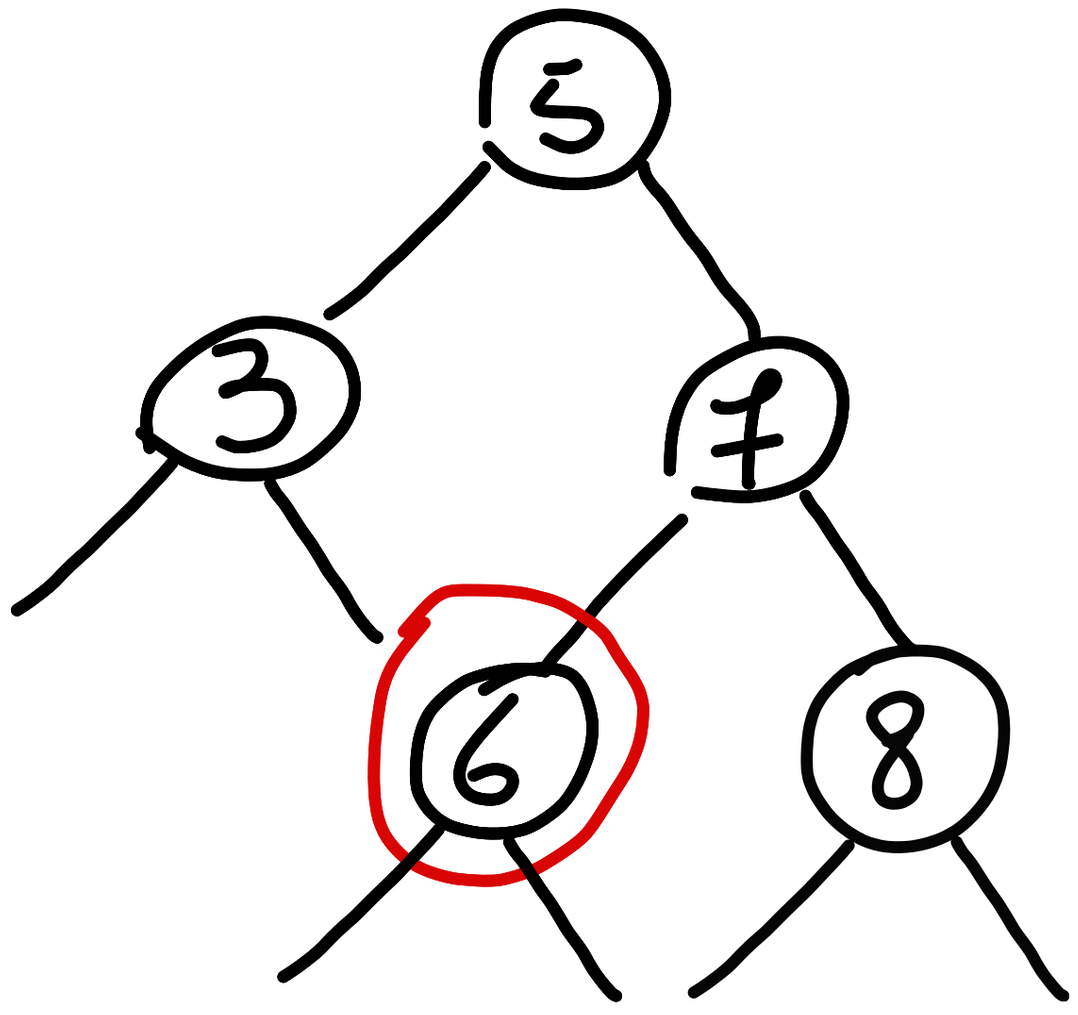
árvores rubro-negras

exemplos



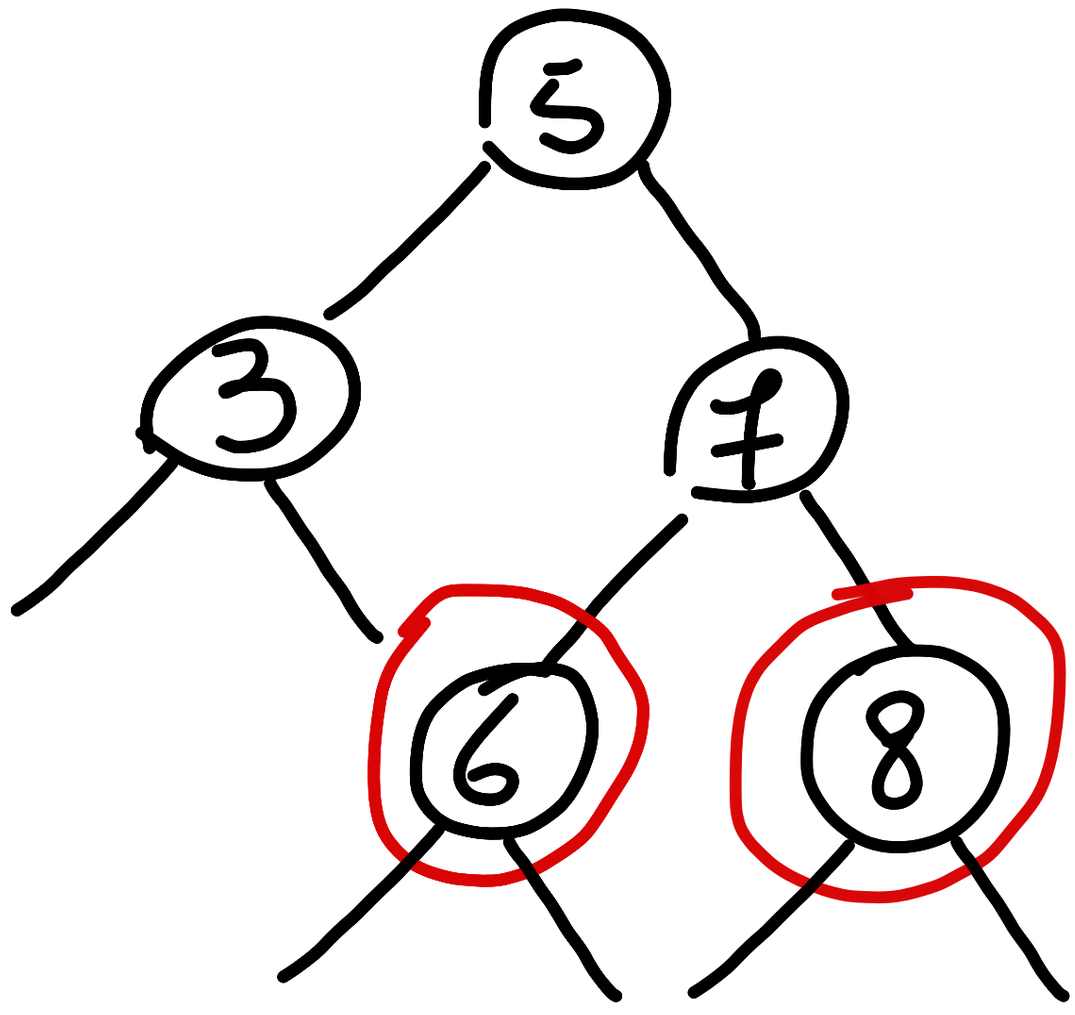
árvores rubro-negras

exemplos



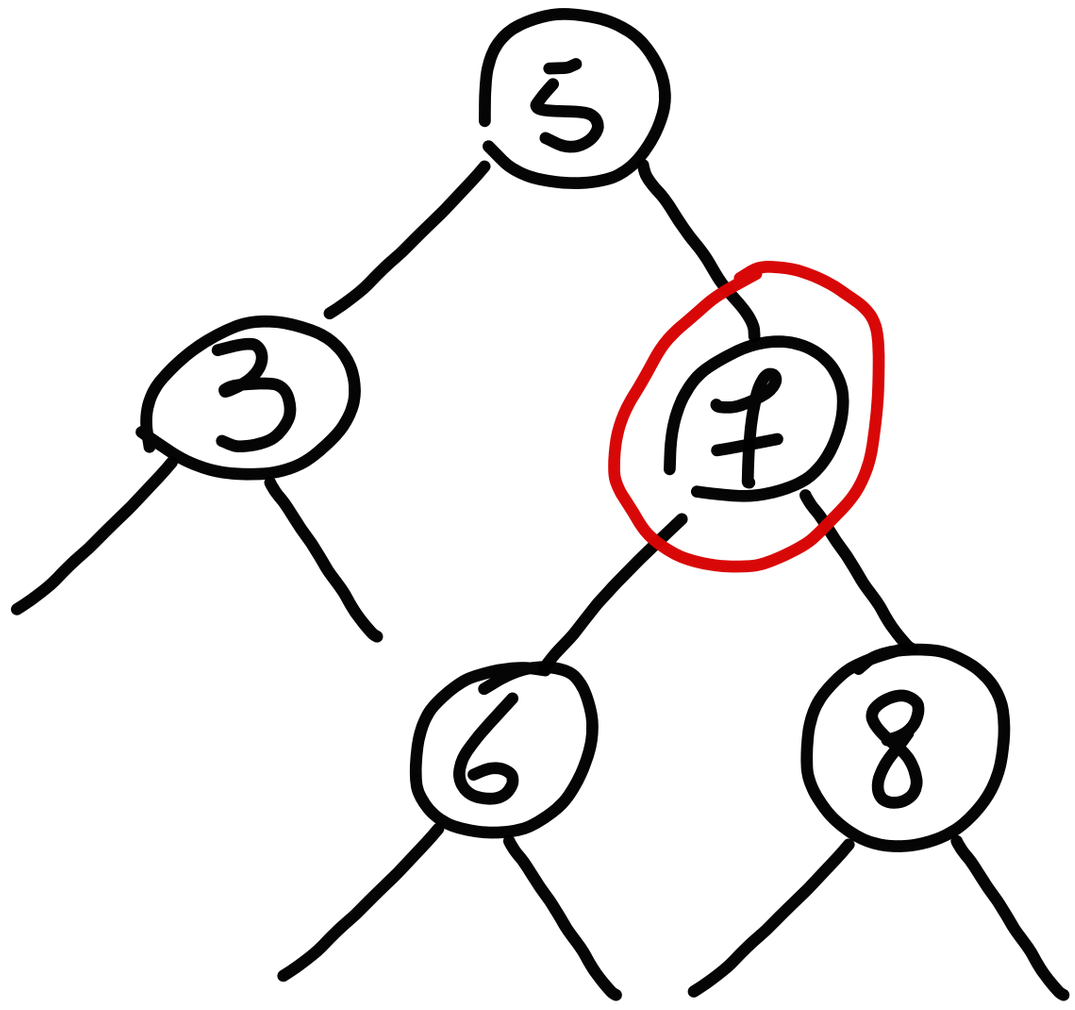
árvores rubro-negras

exemplos



árvores rubro-negras

exemplos



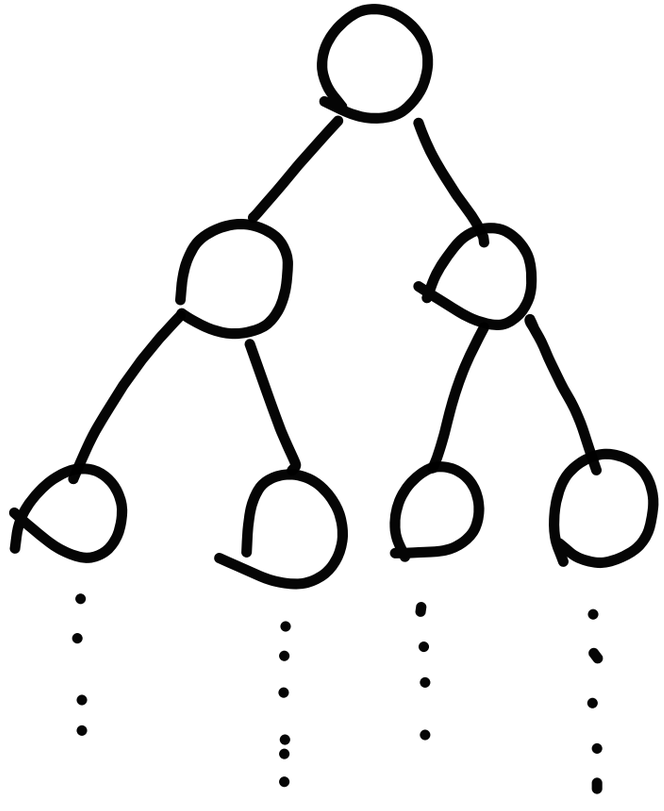
árvores rubro-negras

altura máxima

proposição: toda árvore rubro-negra com n nós tem altura $\leq 2\log(n+1)$

árvores rubro-negras

observação: se todos os caminhos *raiz-árvore-vazia*
em uma árvore binária tiverem $\geq k$ nós,
então essa árvore tem em seu topo uma
árvore perfeitamente balanceada com altura $k - 1$



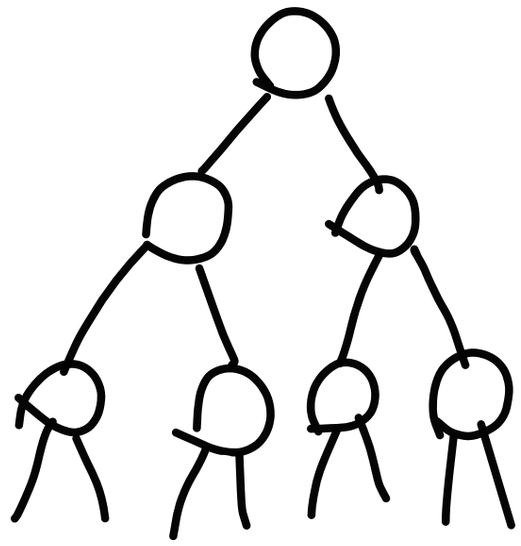
essa árvore tem $n \geq 2^k - 1$ nós

e $k \leq \log(n + 1)$

árvores rubro-negras

altura máxima

$k \leq \log(n+1)$, onde k é o $\#$ mínimo
de nós em um caminho *raiz-árvore-vazia*

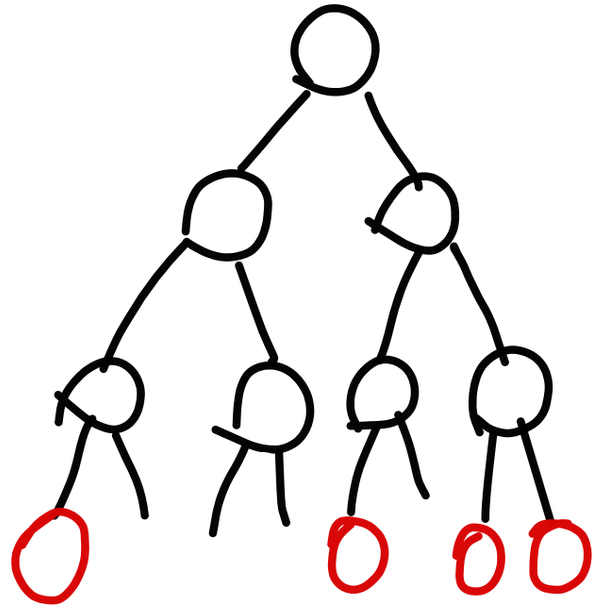


em uma árvore rubro-negra com n nós,
existe um caminho *raiz-árvore-vazia*
com $\leq \log(n+1)$ nós negros

árvores rubro-negras

altura máxima

$k \leq \log(n+1)$, onde k é o # mínimo
de nós em um caminho *raiz-árvore-vazia*



em uma árvore rubro-negra com n nós,
existe um caminho *raiz-árvore-vazia*
com $\leq \log(n+1)$ nós negros

árvores rubro-negras

altura máxima

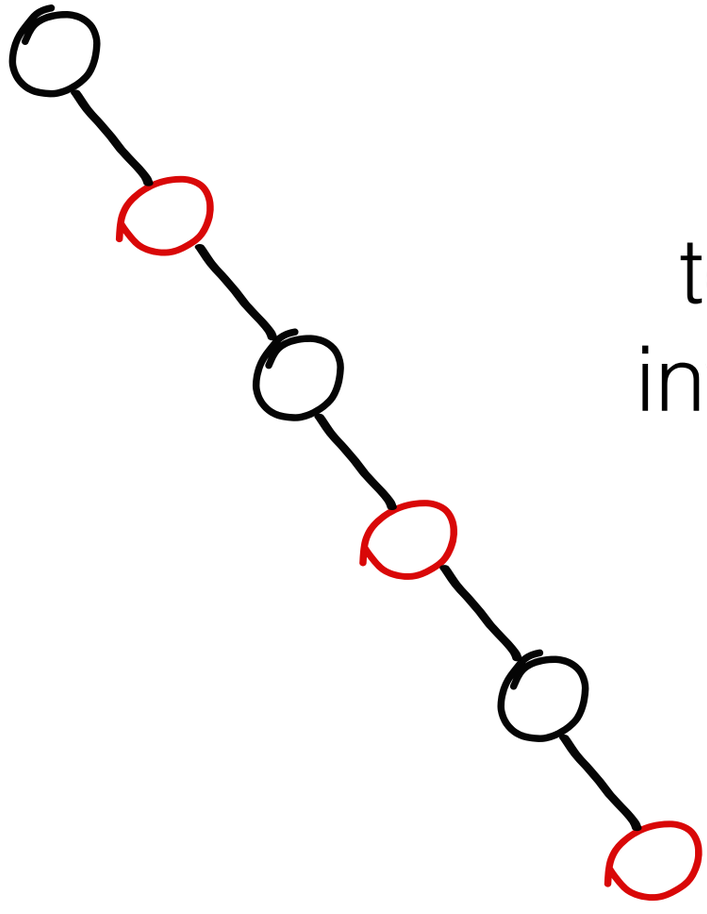
existe um caminho *raiz-árvore-vazia*
com $\leq \log(n+1)$ nós negros

pela quarta invariante, todos os caminhos
raiz-árvore-vazia tem $\leq \log(n+1)$ nós negros

árvores rubro-negras

em uma árvore rubro-negra com n nós,
se todos os caminhos *raiz-árvore-vazia*
têm $\leq \log(n+1)$ nós negros,
qual o tamanho máximo do maior caminho dessa árvore?

- ☐ $\log(n+1)$
- ☐ $\sqrt{n}$
- ☐ $2\log(n+1)$
- ☐ n



terceira
invariante

árvores rubro-negras

como manter as invariantes?

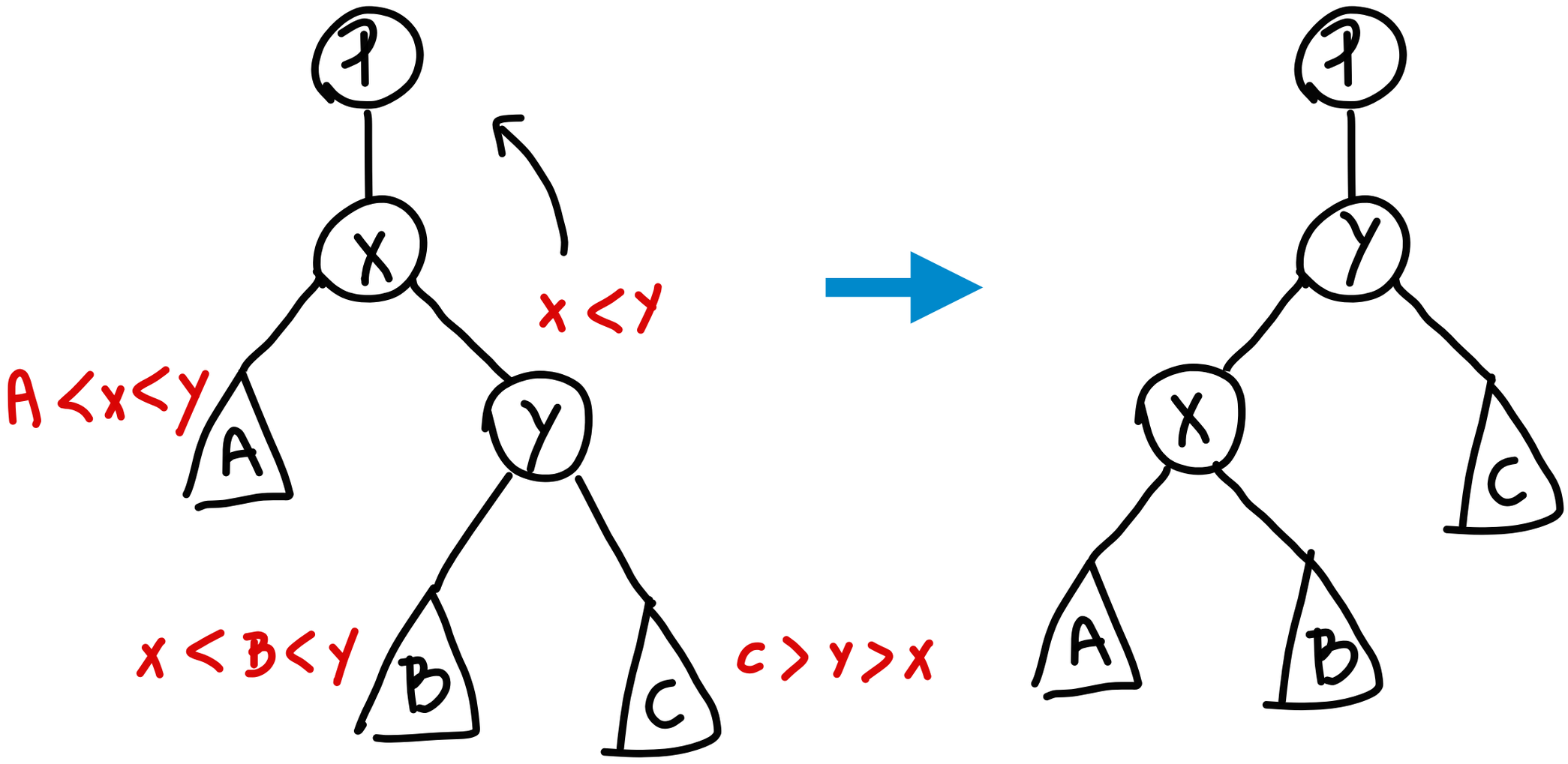
rotações para esquerda e direita

ideia

rebalancear localmente a subárvore enraizada
em um dado nó em tempo $O(1)$

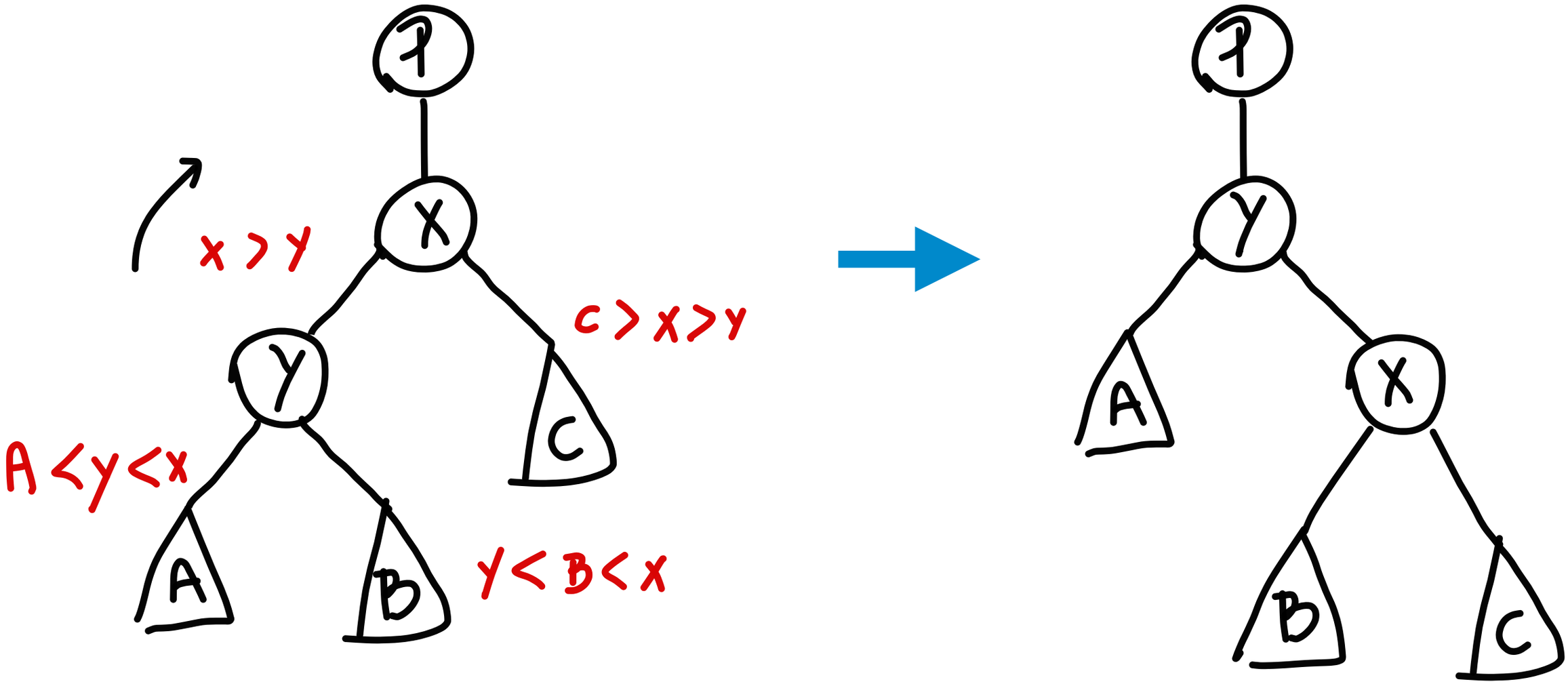
árvores balanceadas

rotação para esquerda de x e y



árvores balanceadas

rotação para direita de x e y



árvores rubro-negras

como manter as invariantes?

rotacionando / rebalanceando

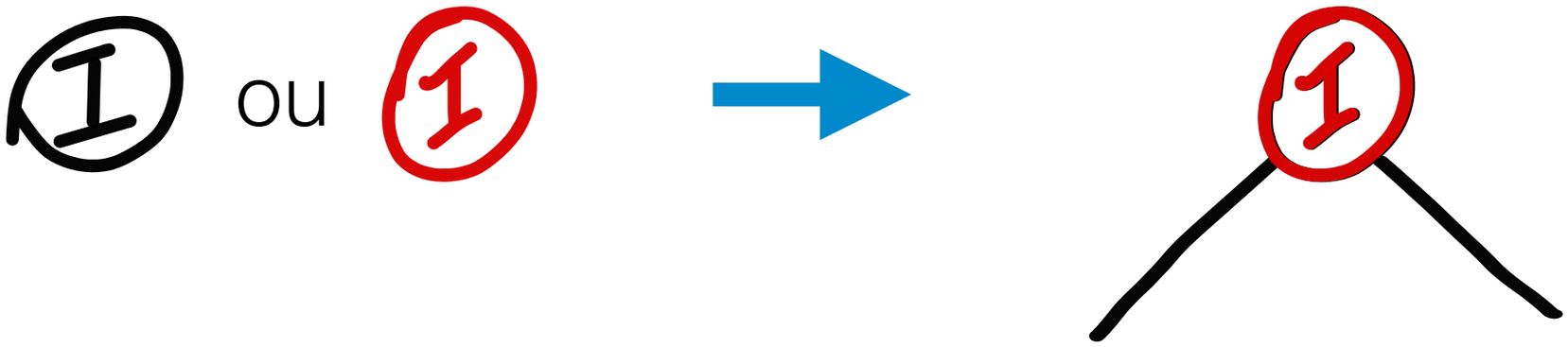
trocando cores

propagando o rebalanceamento
para cima se necessário

árvores rubro-negras

inserção

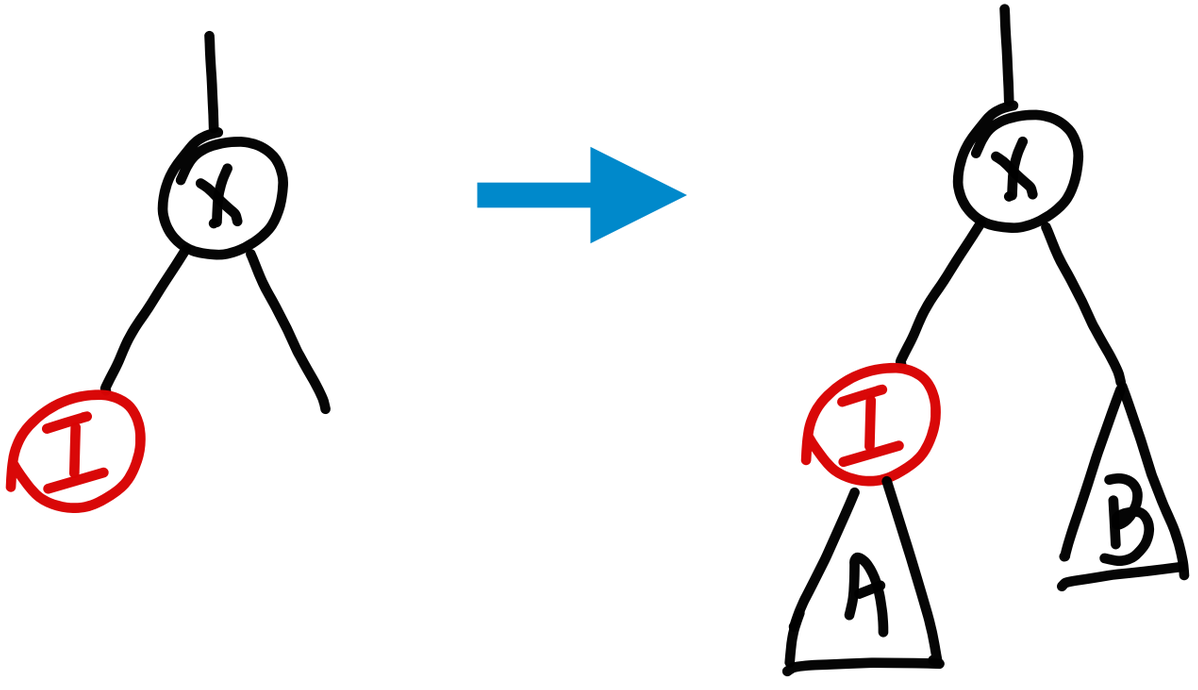
árvore vazia



árvores rubro-negras

inserção

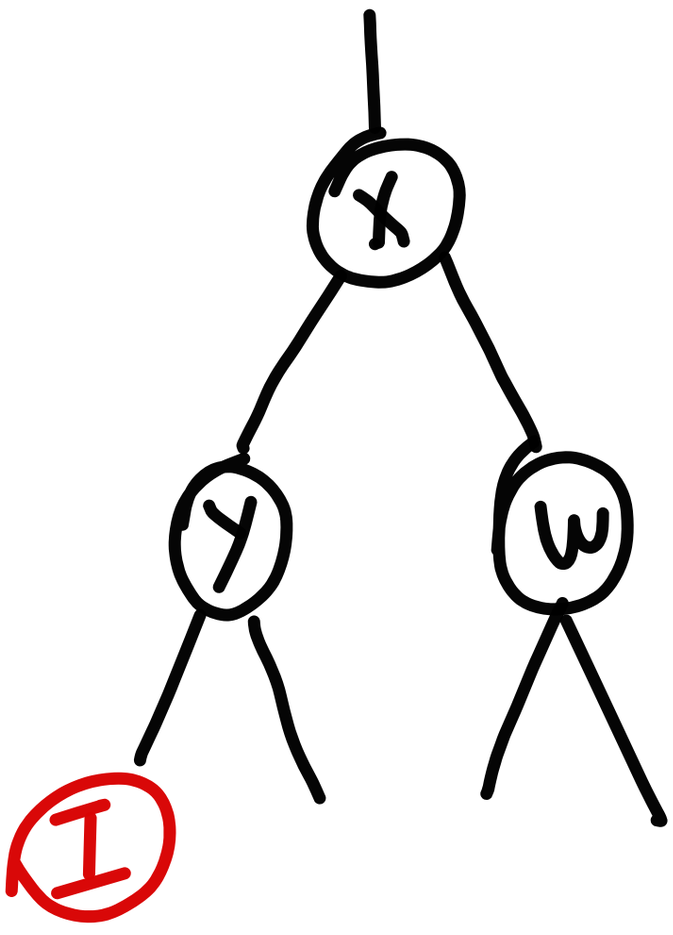
nó pai é negro



árvores rubro-negras

inserção

noção de tio

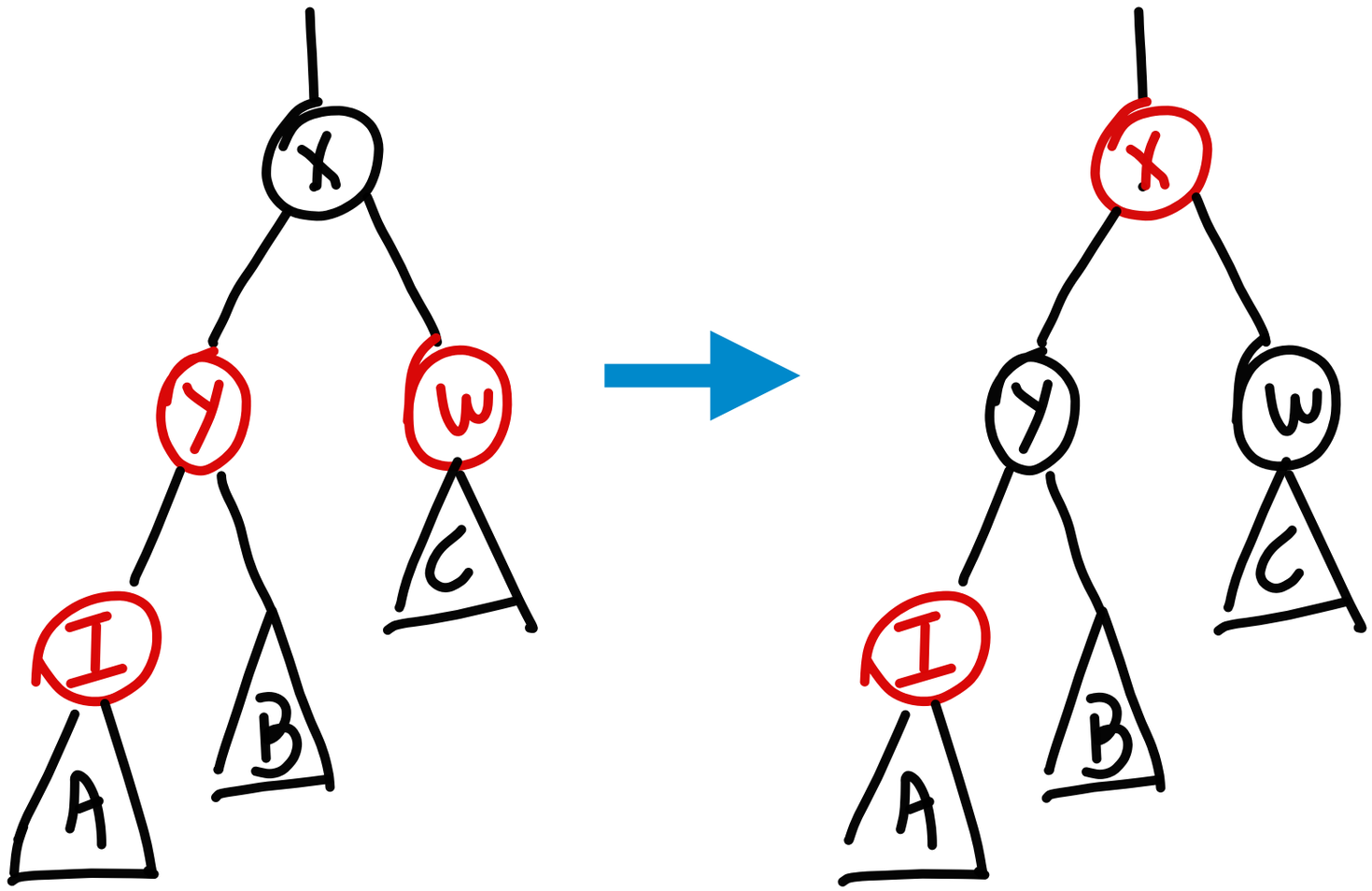


o caso simétrico
é sempre aplicável

árvores rubro-negras

inserção

I tem pai rubro e tio rubro

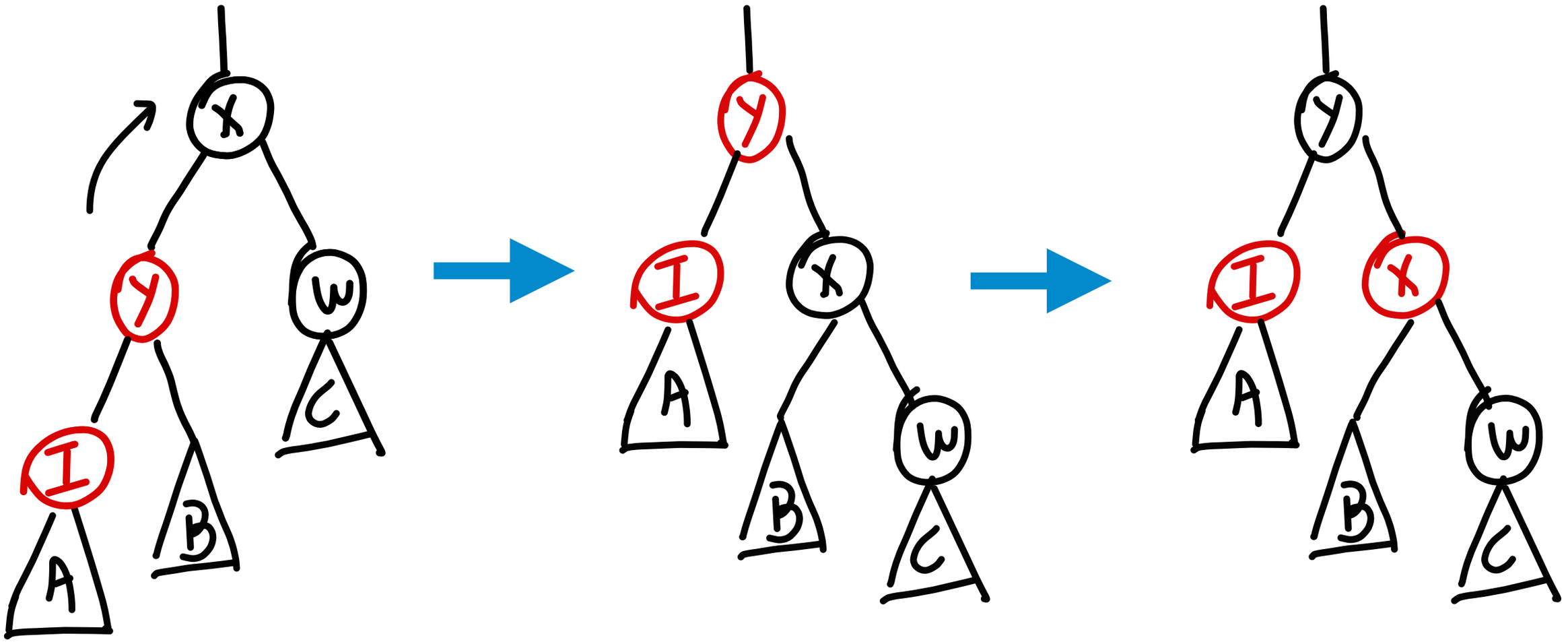


não importa de que lado I esteja

árvores rubro-negras

inserção

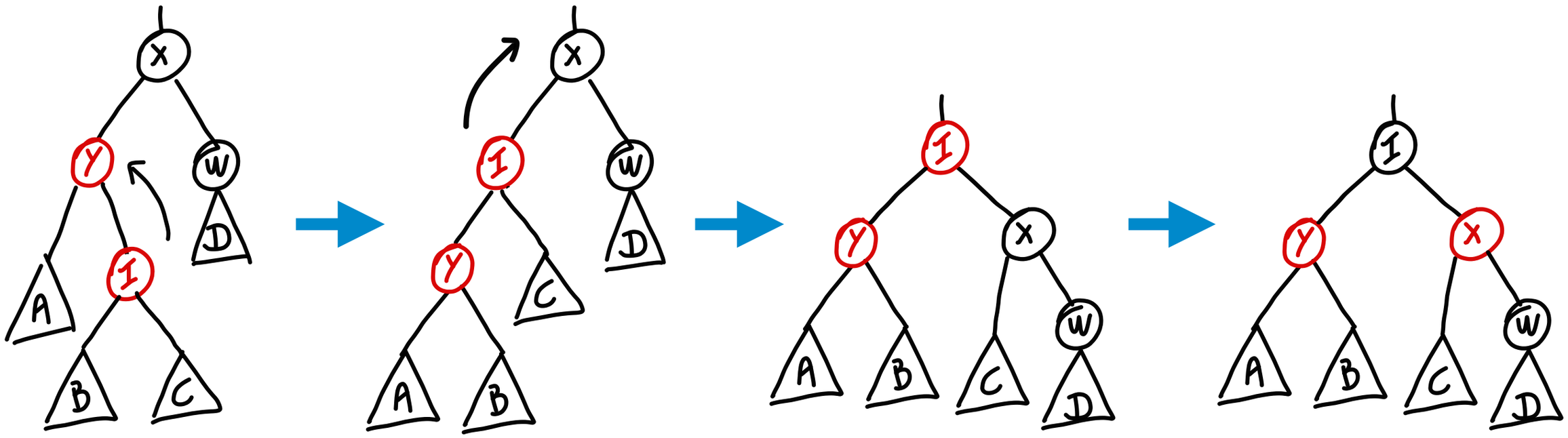
I é filho esquerdo, tem pai rubro e tio negro



árvores rubro-negras

inserção

I é filho direito, tem pai rubro e tio negro



árvores rubro-negras

implementação

http://www.opensource.apple.com/source/sudo/sudo-46/src/redblack.c

www.opensource.apple.com/source/sudo/sudo-46/src/redblack.h

```
#ifndef _SUDO_REDBLACK_H
#define _SUDO_REDBLACK_H

enum rbcolor {
    red,
    black
};

enum rbtraversal {
    preorder,
    inorder,
    postorder
};

struct rbnode {
    struct rbnode *left, *right, *parent;
    void *data;
    enum rbcolor color;
};

struct rbtree {
    int (*compar) __P((const void *, const void *));
    struct rbnode root;
    struct rbnode nil;
};

#define rbapply(t, f, c, o)     rbapply_node((t), (t)->root.left, (f), (c), (o))
#define rbisempty(t)            ((t)->root.left == &(t)->nil && (t)->root.right == &(t)->nil)
#define rbfirst(t)              ((t)->root.left)
#define rbroot(t)               (&(t)->root)
#define rbnil(t)                (&(t)->nil)

void *rbdelete                  __P((struct rbtree *, struct rbnode *));
int rbapply_node                __P((struct rbtree *, struct rbnode *,
                                    int (*)(void *, void *), void *,
                                    enum rbtraversal));
struct rbnode *rbfind           __P((struct rbtree *, void *));
struct rbnode *rbinsert         __P((struct rbtree *, void *));
struct rbtree *rbcreate         __P((int (*)(const void *, const void *)));
void rbdestroy                  __P((struct rbtree *, void (*)(void *)));

#endif /* _SUDO_REDBLACK_H */
```

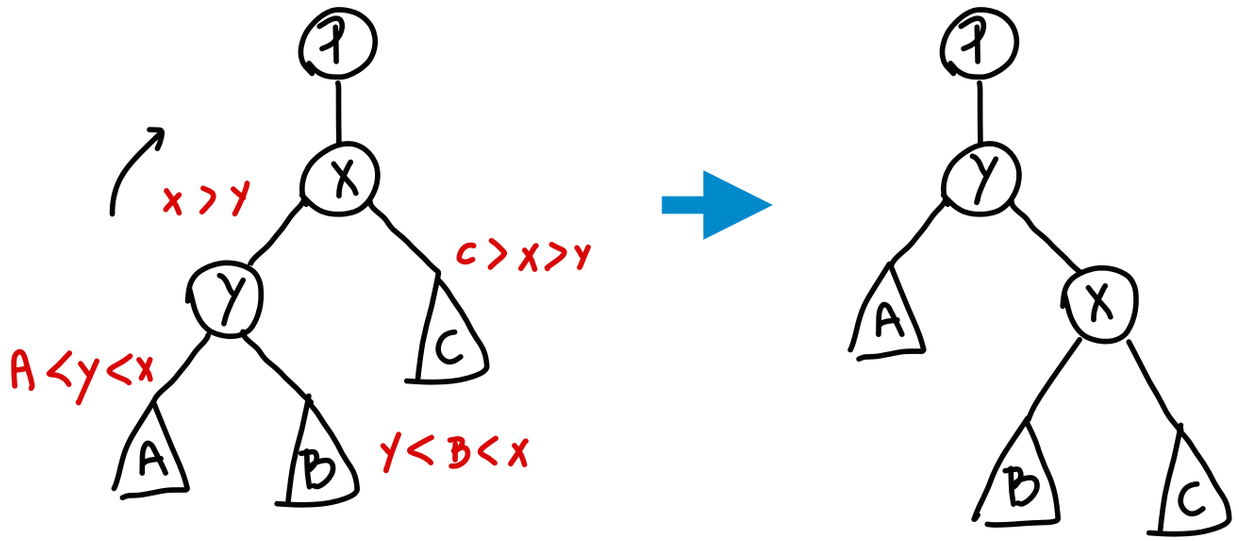
www.opensource.apple.com/source/sudo/sudo-46/src/redblack.c

```c
/*
 * Perform a right rotation starting at node.
 */
static void
rotate_right(tree, node)
    struct rbtree *tree;
    struct rbnode *node;
{
    struct rbnode *child;

    child = node->left;
    node->left = child->right;

    if (child->right != rbnil(tree))
        child->right->parent = node;
    child->parent = node->parent;

    if (node == node->parent->left)
        node->parent->left = child;
    else
        node->parent->right = child;
    child->right = node;
    node->parent = child;
}
```



www.opensource.apple.com/source/sudo/sudo-46/src/redblack.c

```
/*
 * Insert data pointer into a redblack tree.
 * Returns a NULL pointer on success.  If a node matching "data"
 * already exists, a pointer to the existant node is returned.
 */
struct rbnode *
rbinsert(tree, data)
    struct rbtree *tree;
    void *data;
{
    struct rbnode *node = rbfirst(tree);
    struct rbnode *parent = rbroot(tree);
    int res;

    /* Find correct insertion point. */
    while (node != rbnil(tree)) {
        parent = node;
        if ((res = tree->compar(data, node->data)) == 0)
            return(node);
        node = res < 0 ? node->left : node->right;
    }

    node = (struct rbnode *) emalloc(sizeof(*node));
    node->data = data;
    node->left = node->right = rbnil(tree);
    node->parent = parent;
    if (parent == rbroot(tree) || tree->compar(data, parent->data) < 0)
        parent->left = node;
    else
        parent->right = node;
    node->color = red;
```

www.opensource.apple.com/source/sudo/sudo-46/src/redblack.c

```
while (node->parent->color == red) {
    struct rbnode *uncle;
    if (node->parent == node->parent->parent->left) {
        uncle = node->parent->parent->right;
        if (uncle->color == red) {
            node->parent->color = black;
            uncle->color = black;
            node->parent->parent->color = red;
            node = node->parent->parent;
        } else /* if (uncle->color == black) */ {
            if (node == node->parent->right) {
                node = node->parent;
                rotate_left(tree, node);
            }
            node->parent->color = black;
            node->parent->parent->color = red;
            rotate_right(tree, node->parent->parent);
        }
    } else { /* if (node->parent == node->parent->parent->right) */
        uncle = node->parent->parent->left;
        if (uncle->color == red) {
            node->parent->color = black;
            uncle->color = black;
            node->parent->parent->color = red;
            node = node->parent->parent;
        } else /* if (uncle->color == black) */ {
            if (node == node->parent->left) {
                node = node->parent;
                rotate_right(tree, node);
            }
            node->parent->color = black;
            node->parent->parent->color = red;
            rotate_left(tree, node->parent->parent);
        }
    }
}
rbfirst(tree)->color = black;        /* first node is always black */
return(NULL);
}
```

árvores rubro-negras

demo

árvores rubro-negras

inserções/remoções on-the-fly

http://www.cs.usfca.edu/~galles/visualization/RedBlack.html

cenário

dá para fazer melhor?

	pior caso depois de N inserções			caso médio depois de N inserções		
estrutura de dados	busca	inserção	remoção	busca	inserção	remoção
vetor ordenado	log *n*	*n*	*n*	log *n*	*n/2*	*n/2*
árvores binárias de busca	*n*	*n*	*n*	1,39log *n*	1,39log *n*	-
árvores rubro-negras	2log *n*	2log *n*	2log *n*	1,0log	1,0log	1,0log
tabelas de espalha-mento	log	log	log	3,5	3,5	3,5

fiquem ligados!

* coeficiente desconhecido, mas muito próximo de 1,0 conforme Sedgewick, 2011
** sob espalhamento uniforme

referências

- R. Sedgewick, K. Waine, Algorithms in C. Addison-Wesley ,1990

- T. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Algoritmos - Teoria e Prática. Campus, 2002.

- D. E. Knuth, The Art of Computer Programming, Vol III: Sorting and Searching. Addison-Wesley (1978).

- Tim Houghgarden, Algorithms: Design and Analysis, Part 1. Coursera MOOC at https://class.coursera.org/algo-004/lecture

- R. Sedgewick, K. Waine, Algorithms, Part I, Coursera MOOC at https://class.coursera.org/algs4partI-004