Seminário / Estrutura de Dados
Red-Black Tree

Carlos Pinheiro
Dayvson Sales
Ellena Oliveira
Victor Accete

Outubro/2016

Self-balancing trees

- What if we could make a tree that rotates less than AVL but still is always balanced?

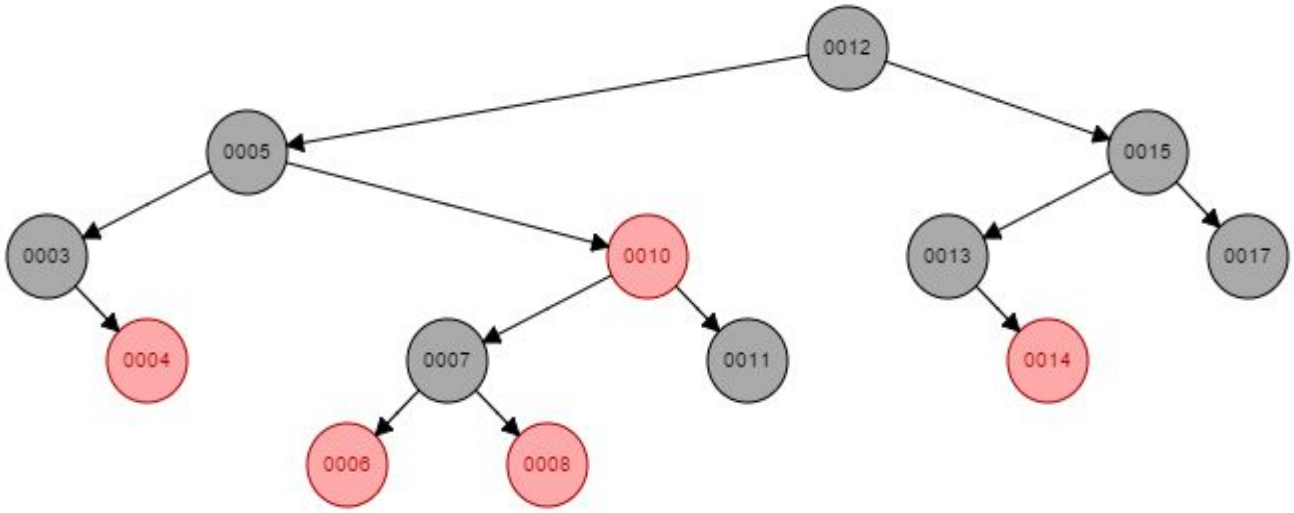
- We saw that an AVL tree can:
  - Insert in O(logn)
  - Search in O(logn)

- Is there a way to change the balancing rules so that we will make less rotations while inserting?

Red-Black Tree

- Is a self-balancing binary search tree, like an AVL tree, but with different balancing rules and different properties

- We will still have:
  - Search: O(logn)
  - Insertion: O(logn)



Red-Black Tree

- Red-black trees are similar to AVL trees, but provide faster real-time worst case performance for insertion and deletion

- The trade-off is a slightly slower (but still O(log n)) lookup time.

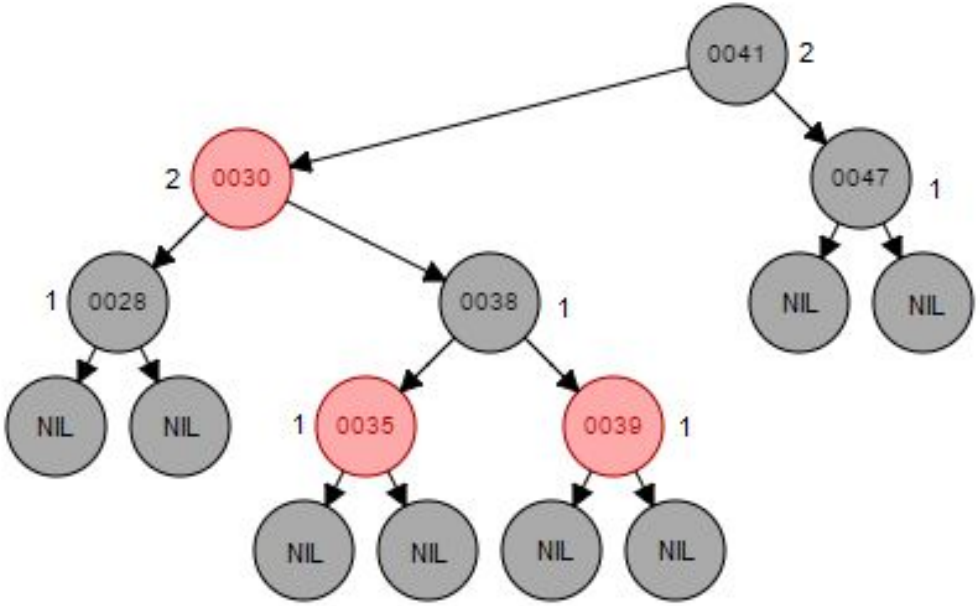
Properties

1. A node is either red or black
2. The root is black
3. Every leaf (NIL) is black (Sentinel)
4. If a node is red, then both its children are black – which implies that we can't have two adjacent red nodes
5. Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes

To preserve these properties we will need to perform rotations ↺ and recolourings!

Quick definition

- We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the black-height of the node, denoted bh(x)



Structs

```
#define NIL tree->nil
#define ROOT tree->root
#define black 'b'
#define red 'r'
```

```
struct rb_node{
    int key;
    char color;
    rbNode* parent;
    rbNode* left;
    rbNode* right;
};
```

```
struct rb_tree{
    rbNode* root;
    rbNode* nil;
};
```

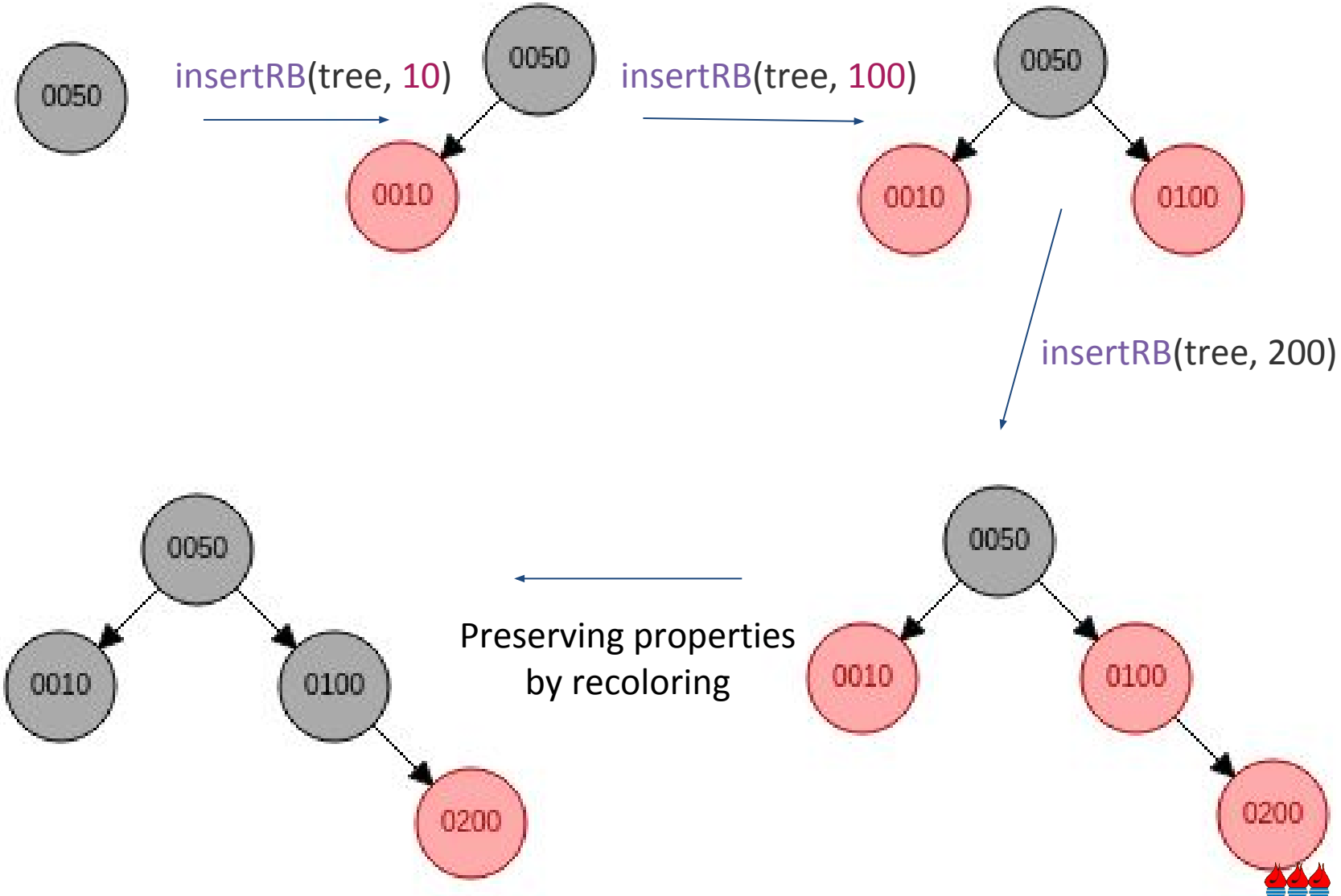
Red-Black Tree ADT

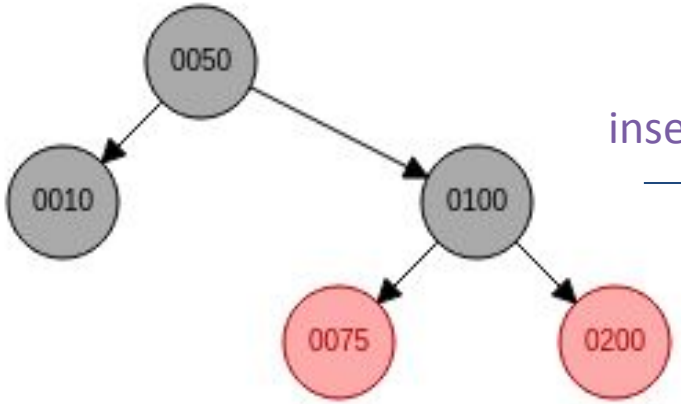
```
rbTree* createTree();

void insertRB(rbTree* tree, int key);

void insertTree(rbTree* tree, rbNode* z);

void rotateLeft(rbTree* tree, rbNode* x);

void rotateRight(rbTree* tree, rbNode* y);

bool searchRB(rbTree* tree, int key);

void printPreOrder(rbTree* tree, rbNode* x);

void printInOrder(rbTree* tree, rbNode* x);

void printPostOrder(rbTree* tree, rbNode* x);
```

```
void insertRB(rbTree* tree, int key){
   /* Allocate new node, set key and color it to red
         Insert in tree as a binary tree */
   insertTree(tree, x);
   /* fix while x->parent != tree->root and x->parent->color == red*/
   while(x->parent->color == red){
     if (x->parent == x->parent->parent->left){
     /* x's parent is a left child, y is x's uncle*/
        y = x->parent->parent->right;
        /* Case 1: x's uncle is red */
        if(y->color == red){
           x->parent->color = black;
           y->color = black;
           x->parent->parent->color = red;
           x = x->parent->parent;
        } else{
           /* Case 2: x's uncle is black and x is a right child */
           if(x == x->parent->right){
              x = x->parent;
              rotateLeft(tree, x);
           }
           /* Case 3: x's uncle is black and x is a left child */
           x->parent->color = black;
           x->parent->parent->color = red;
           rotateRight(tree, x->parent->parent);
        }
     } else /* Same as if with left and right exchanged */
   }
   ROOT->color = black;
}
```

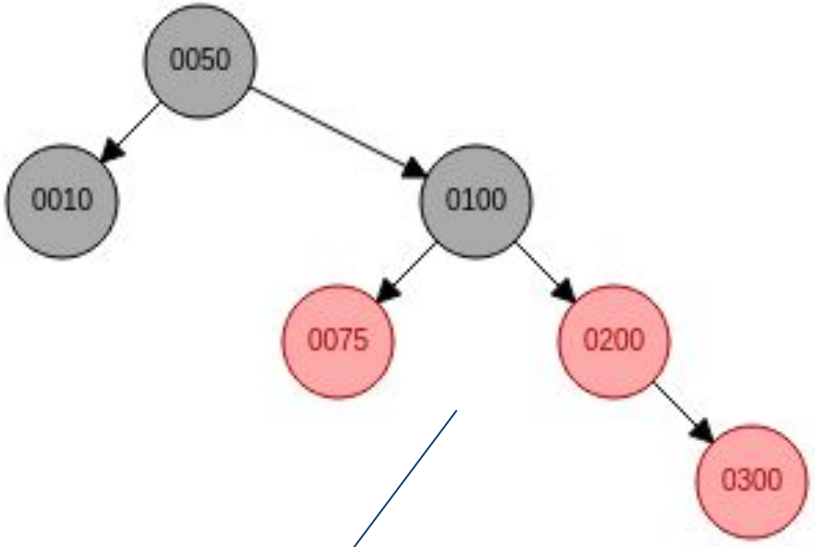
Insertion

Insertion

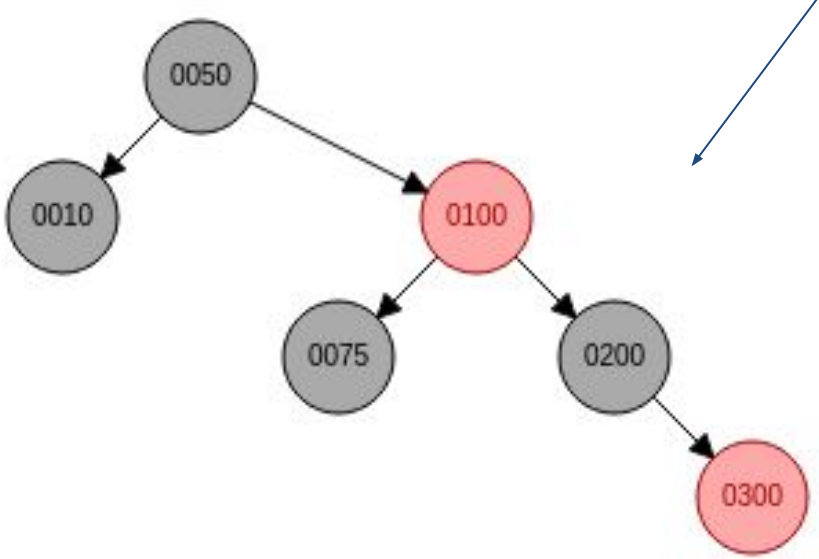

0050
insertRB(tree, 10)
0050
0010
insertRB(tree, 100)
0050
0010
0100
insertRB(tree, 200)
0050
0010
0100
0200
Preserving properties
by recoloring
0050
0010
0100
0200



insertRB(tree, 300)

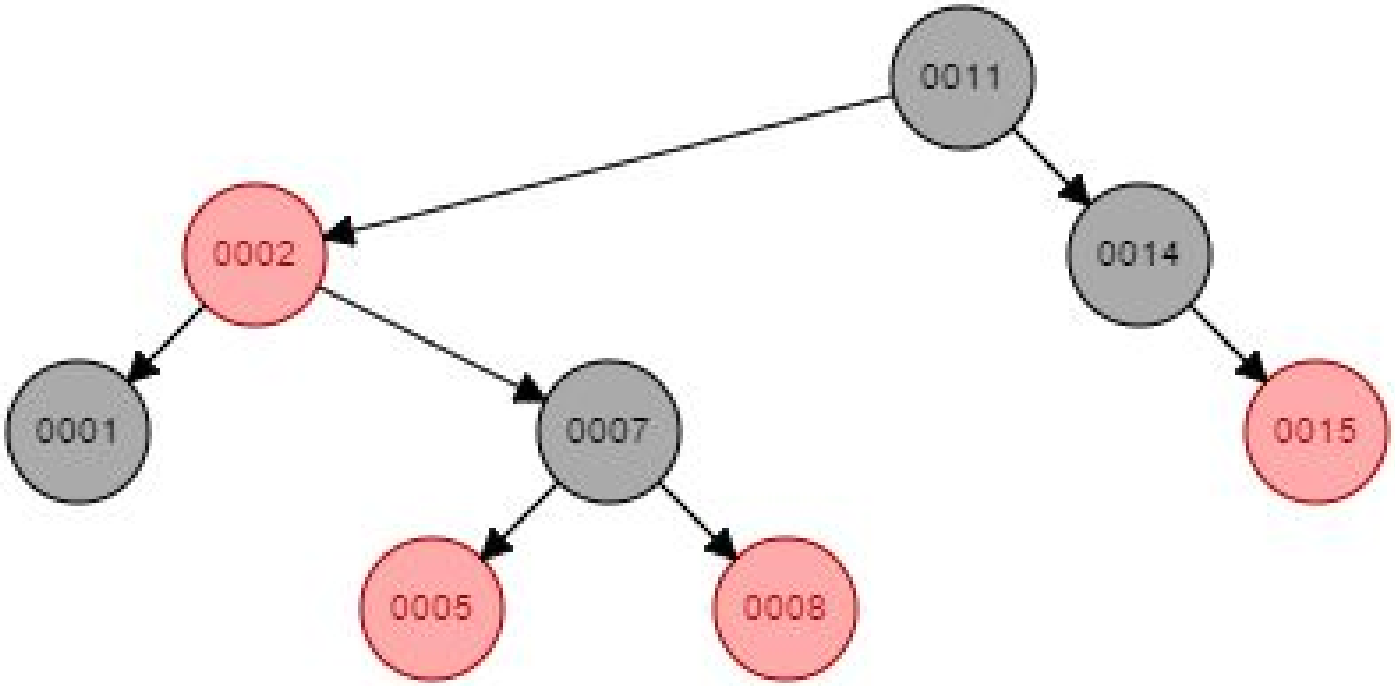


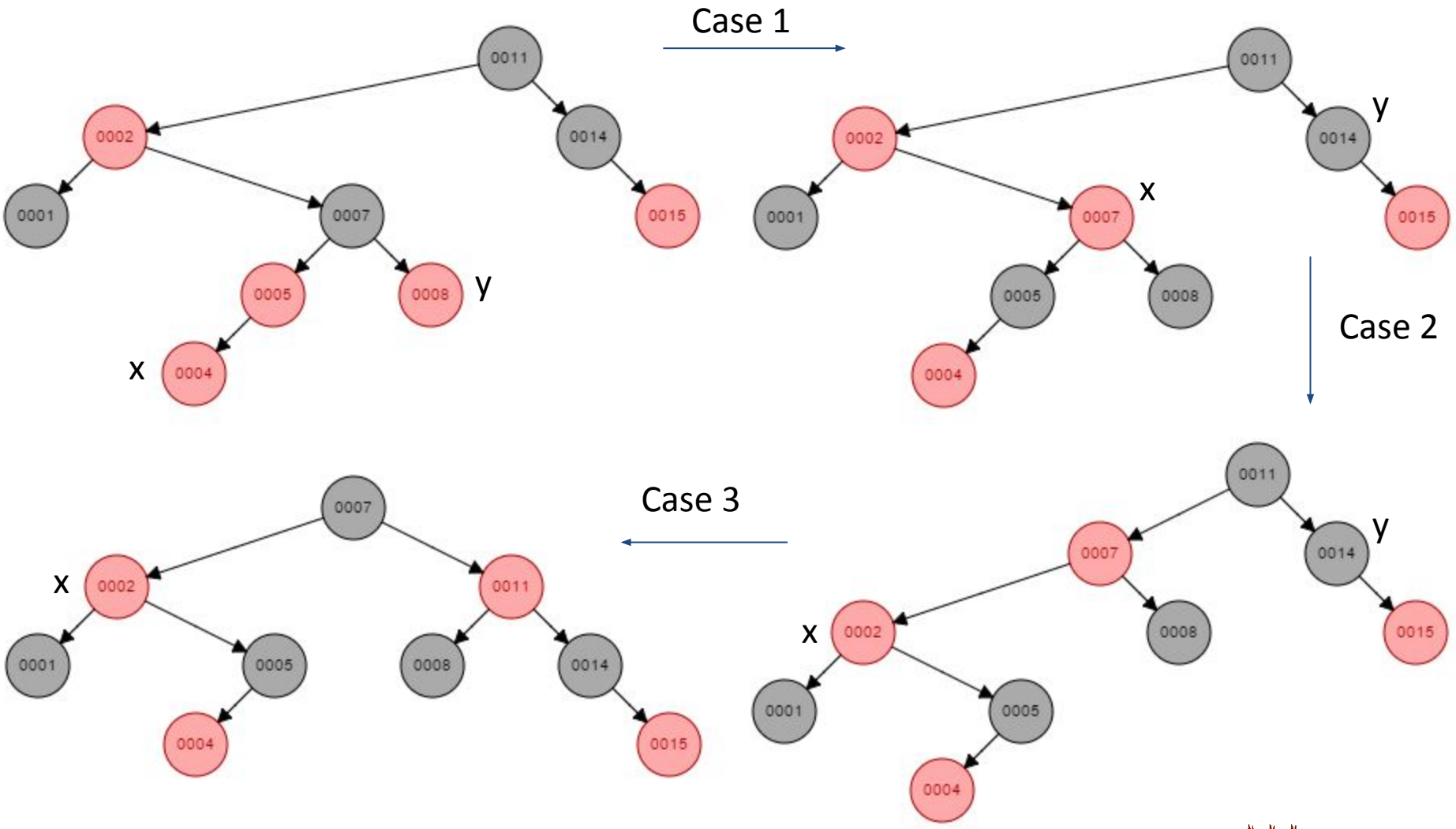
Preserving properties
by recoloring



Insertion

- An insertion that covers three cases
- Let the node 11 below be tree root,
- then insertRB(tree, 4);




Case 1
0011
0002
0014
0001
0007
0015
0005
0008
y
x
0004
0011
y
0002
0014
x
0001
0007
0015
0005
0008
0004
Case 2
Case 3
0007
x
0002
0011
0001
0005
0008
0014
0004
0015
0011
y
0007
0014
x
0002
0008
0015
0001
0005
0004

So…

- We've seen that AVL tree isn't the only way to keep a binary search tree balanced
- Red-Black trees also achieve that purpose, in a less rigid way!
- The tree would be slightly less balanced, however, the insertion would be slightly faster

Where are RB Trees used?

- RB Tree is probably the most popular self-balancing tree implementation
- Java: java.util.TreeMap , java.util.TreeSet
- C++ STL: map, multimap, multiset
- Linux kernel: completely fair scheduler, linux/rbtree


References

- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C.. Introduction to Algorithms. Cambridge: MIT, 2009.
- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
- https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-10-red-black-trees-rotations-insertions-deletions/lec10.pdf