Algorithms and Data Structures
Lecture notes: Red-Black Trees, Cormen Chap. 13

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

5 janvier 2022

Outline

Red-Black trees

Red-Black trees are binary search trees that are guaranteed to always have a height $h = O(\lg n)$ after the execution of operations such as insert or delete

In this lecture :

- First : describe the properties of red-black trees
- Then : prove that these guarantee h $=$ O(lg n)
- Finally : describe operations on red-black trees

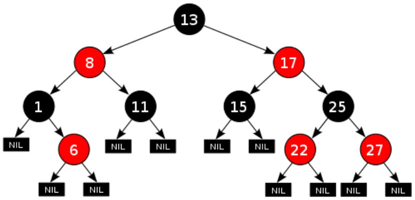
Red-black tree properties

1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL pointer) is black (every ”real” node has 2 children)
4. If a node is red, both children are black (can’t have 2 consecutive reds on a path)
5. For each node, all simple paths from the node to descendant leaves contains the same number of black nodes

Red-black tree properties

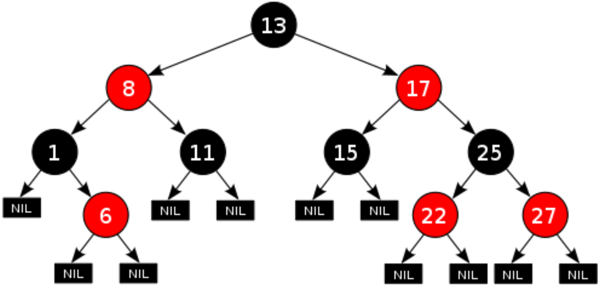
Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes



Black-height

The *black-height* (bh) of a node x is the number of black nodes, not including x, on a path from x to a leaf



A node of height h has black-height $\geq h/2$ since, by property 4, at least half of the nodes on a simple path must be black

Proving height bound of RB

Theorem

A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$

Claim : A subtree rooted at a node x contains at least $2^{bh(x)} - 1$ internal nodes

Proof by mathematical induction on height h

Basic step : x has height 0 (i.e., NIL leaf node)

- $bh(x) = 0$
- Subtree contains $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes

Proving height bound of RB

Inductive step : x has positive height and 2 children

- Each child has black-height of $bh(x)$ or $bh(x) - 1$
- The height of a child = (height of x) - 1
- Inductive hypothesis : subtrees rooted at each child contain at least $2^{bh(x)-1} - 1$ internal nodes
- Thus subtree at x contains

$$\begin{aligned} & (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 \\ = \quad & 2 \times 2^{bh(x)-1} - 1 \text{ internal nodes} \\ = \quad & 2^{bh(x)} - 1 \text{ internal nodes} \end{aligned}$$

By property 4, black height of the root r ($bh(r)$) is at least $h/2$

Thus at the root r of the red-black tree : $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$

- adding 1 on each side $n + 1 \geq 2^{h/2}$,
- taking the log on each side $\lg(n + 1) \geq h/2$,
- thus $2\lg(n + 1) \geq h$.

Operations on RB trees : Worst-Case Time

So we proved that a red-black tree has $O(\lg n)$ height

Corollary : These operations take $O(\lg n)$ time :

- Minimum(), Maximum()
- Successor(), Predecessor()
- Search()
- Insert() and Delete()
  - also take $O(\lg n)$ time
  - But will need special care since they modify tree

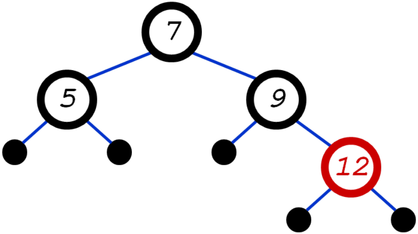
Note, the algorithms for the operations Minimum(), Maximum(), Successor(), Predecessor() and Search() are the same as for the binary search trees

Red-Black trees : an example

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

Coloring this tree :



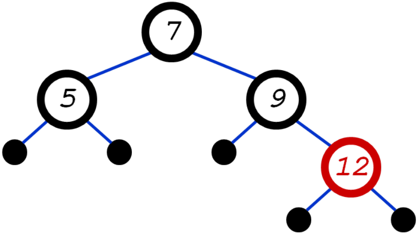
Red-Black trees : Insertion

Insertion

1. Use Tree-Insert algorithm in binary search

Insert node 8 :

- Where does it go ?



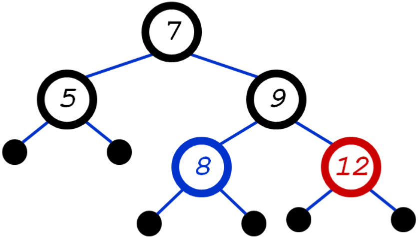
Red-Black trees : Insertion

Red-Black trees properties

5. Every path from node to descendant leaf contains the same number of black nodes

Insert node 8 :

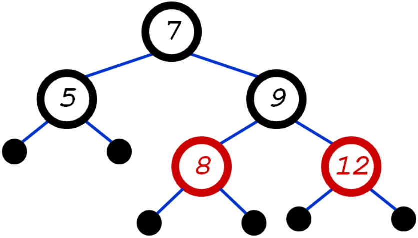
- Which color it should be ?



Red-Black trees : Insertion

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

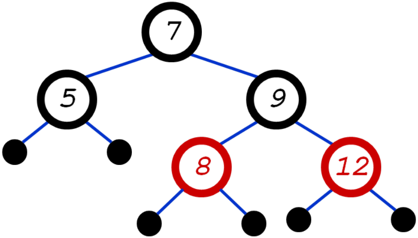


Red-Black trees : Insertion

Insertion

Insert 11 :

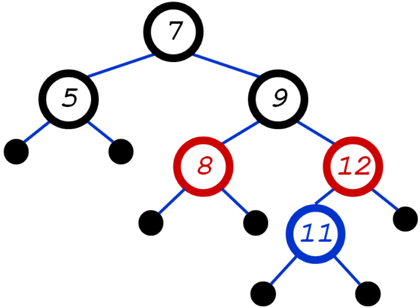
- Where node 11 goes
- Which color it should be ?



Red-Black trees : Insertion

Insertion

- Node 11 inserted



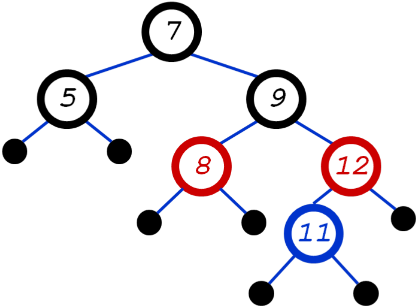
Red-Black trees : Insertion

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

Insert 11 :

- Can't be red, violate property 4
- Can't be black, violate property 5

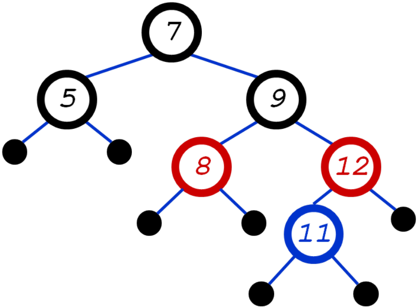


Red-Black trees : Insertion

Re-color

- The rotation operation is used to re-color the tree
- Operation described in the next slides

The solution is to re-color the tree !

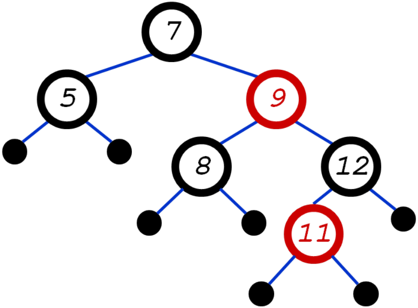


Red-Black trees : Insertion

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

Insert 10 :

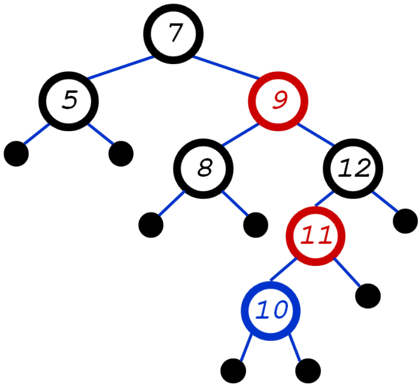


Red-Black trees : Insertion

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

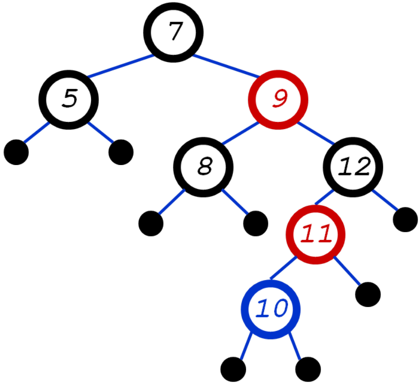
Insert 10 :



Red-Black trees : Insertion

- We cannot decide the color for 10, the tree is too unbalanced
- Must change tree structure to allow recoloring
- Goal : rebalance tree in $O(\lg n)$ time

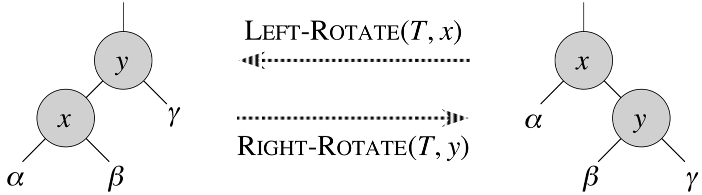
Insert 10 :



Red-Black trees : Rotation

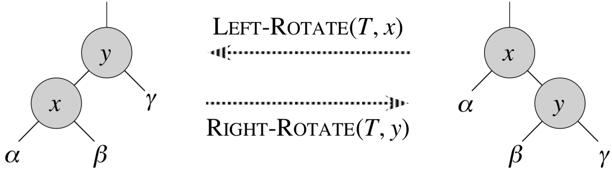
Our basic operation for changing tree structure is called rotation :

Two types of rotation operations : left rotation and right rotation

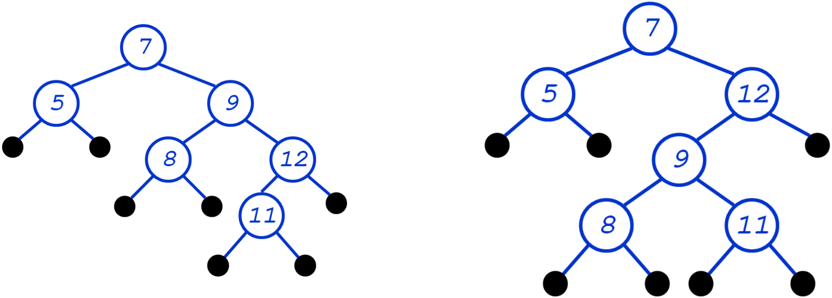


Rotation operations must preserve the inorder key ordering

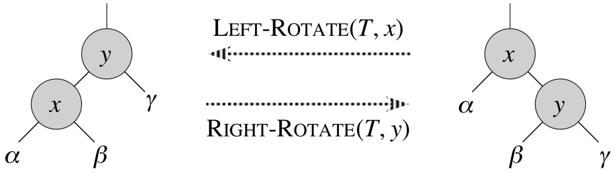
Left rotation



Rotate left about 9 :



Red-Black trees : Rotation



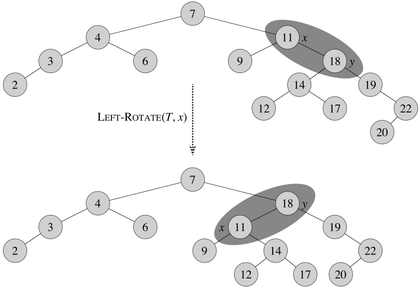
Lot of pointers manipulation

Right-Rotate(T,y) :

- x keeps its left child
- y keeps its right child
- x's right child becomes y's left child
- x's and y's parents change

Red-Black trees : Left rotation

```
Left-Rotate(T,x)
   y = x.right
   x.right = y.left
   if y.left ≠ T.nil
      y.left.p = x
   y.p = x.p
   if x.p == T.nil
      T.root = y
   elseif x == x.p.left
      x.p.left = y
   else x.p.right = y
   y.left = x
   x.p = y
```



Red-Black trees : Insertion

Red-black insertion of a node z : the basic idea

- Start by doing regular binary-search-tree insertion
- Color z red
- Only RB properties 2 (z is a root) and 4 might be violated ($z.p$ is red)
- If so, move violation up tree until a place is found where it can be fixed
- Total time will be $O(\lg n)$

Red-Black tree operation : Insertion

```
RB-Insert(T, z)
    y = T.nil
    x = T.root
    while x ≠ T.nil
        y = x
        if z.key < x.key
            x = x.left
        else x = x.right
    z.p = y
    if y == T.nil
        T.root = z
    elseif z.key < y.key
        y.left = z
    else y.right = z
    z.left = T.nil
    z.right = T.nil
    z.color = RED
    RB-Insert-Fixup(T,z)
```

Which properties of RB tree can be violated

Which of the red-black properties might be violated upon the call to RB-Insert-Fixup ?

Properties 1 and 3 still hold since both children of the newly inserted red node are the sentinels (pointers) T :nil.

Property 5 (number of black nodes is the same on every simple path from a given node) is satisfied because node z replaces the (black) sentinel, and node z is red with sentinel children.

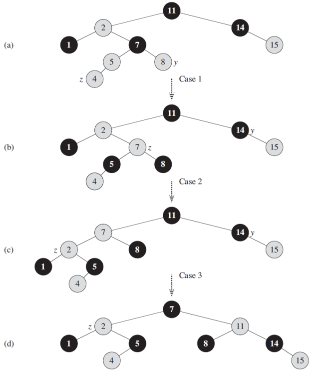
Thus, only property 2 (the root is black) or property 4 (a red node cannot have a red child) might be violated.

Both violations are due to z being colored red. Property 2 is violated if z is the root, and property 4 is violated if z's parent is red.

Remove violation property 4 : 3 cases

Let y be the sibling of $z.p$ (also called uncle in the textbook)

- Case 1 : both $z.p$ and y are red. Because $z.p.p$ is black, we can color both $z.p$ and y black, thereby fixing the problem of z and $z.p$ both being red, and we can color $z.p.p$ red, thereby maintaining property 5.
- Case 2 : y is black and z is a right child
- Case 3 : y is black and z is a left child



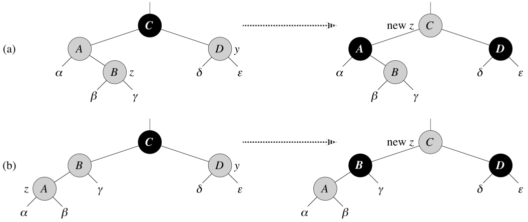
Red-Black trees : Insertion

Remove the violation of property 4 :

```
RB-Insert-Fixup(T, z)
    while z.p.color == RED
        if z.p == z.p.p.left
            y = z.p.p.right
            if y.color == RED
                z.p.color = BLACK            //case 1
                y.color = BLACK              //case 1
                z.p.p.color = RED            //case 1
                z = z.p.p                    //case 1
            elseif z == z.p.right
                    z = z.p                  //case 2
                    Left-Rotate(T,z)         //case 2
                z.p.color = BLACK            //case 3
                z.p.p.color = RED            //case 3
                Right-Rotate(T, z.p.p)       //case 3
        else (same as then clause with "right" and "left" exchanged)
    T.root.color = BLACK
```

RB Insert : Case 1

Case 1 : uncle (y) is RED

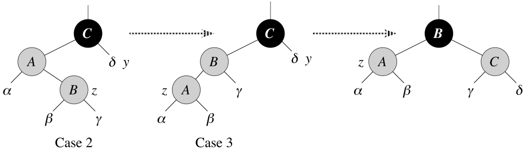



```
if y.color == RED
    z.p.color = BLACK
    y.color = BLACK
    z.p.p.color = RED
    z = z.p.p
```

- z.p.p must be black, since z and z.p are both red
- Make z.p and y black ; z and z.p are not both red. But property 5 might be violated.
- Make z.p.p red ; restores property 5
- The next iteration has z.p.p as new z(z has moved up 2 levels)

RB Insert : Case 2

Case 2 : uncle (y) is BLACK ; Node z is a right child

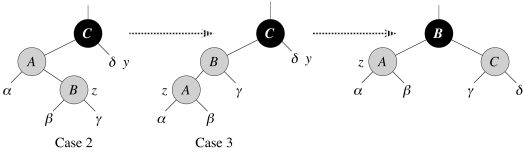



```
elseif z == z.p.right
z = z.p
Left-Rotate(T,z)
```

- Left rotate around z.p ; z is a left child, and both z and z.p are RED
- Takes us to case 3

RB Insert : Case 3

Case 3 : uncle (y) is BLACK ; Node z is a left child




```
z.p.color = BLACK
z.p.p.color = RED
Right-Rotate(T, z.p.p)
```

- Make z.p BLACK and z.p.p RED
- Then right rotate on z.p.p
- No longer have 2 reds in a row
- z.p is BLACK ; no more iterations

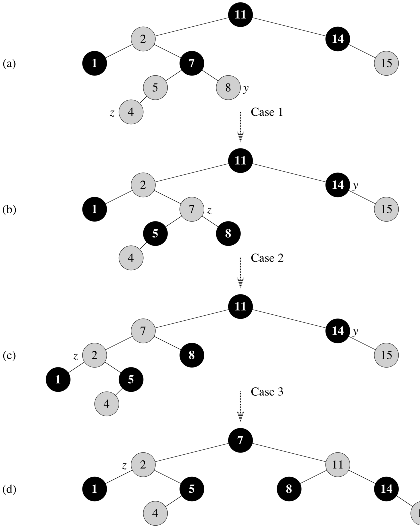
RB Insert : Cases 4-6

Cases 1-3 hold if z's parent is a left child

If z's parent is a right child, cases 4-6 are symmetric (swap left for right)

Insertion : final example

```
RB-Insert-Fixup(T, z)
    while z.p.color == RED
        if z.p == z.p.p.left
            y = z.p.p.right
            if y.color == RED
                z.p.color = BLACK      //case 1
                y.color = BLACK      //case 1
                z.p.p.color = RED      //case 1
                z = z.p.p        //case 1
            elseif z == z.p.right
                    z = z.p        //case 2
                    Left-Rotate(T,z)        //case 2
                z.p.color = BLACK        //case 3
                z.p.p.color = RED        //case 3
                Right-Rotate(T, z.p.p)        //case 3
        else (same as then clause
            with "right" and "left" exchanged)
    T.root.color = BLACK
```



RB Insert : analysis

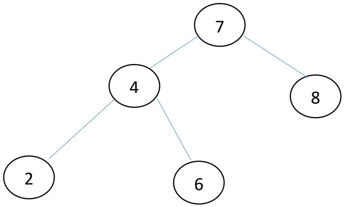
$O(\lg n)$ to get through RB-Insert up to the call of RB-Insert-Fixup

Within RB-insert-Fixup :

- Each iteration takes $O(1)$.
- Each iteration is either the last one or it moves z up 2 levels.
- $O(\lg n)$ levels, therefore $O(\lg n)$ time.
- Also note that there are at most 2 rotations overall.

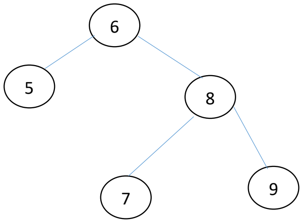
Thus, insertion into a red-black tree takes $O(\lg n)$ time.

Exercise 1



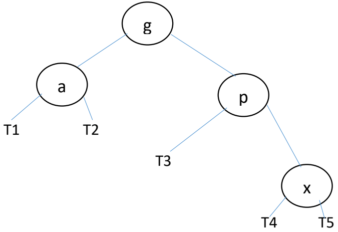
Rotate to the right around 7

Exercise 2



Rotate to the left around 6

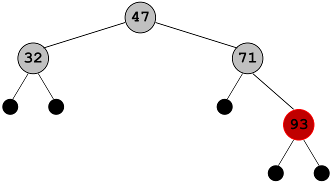
Exercise 3



Rotate to the left around g

Exercises

4. Consider the following red-black tree. Insert consecutively nodes 65, 82 and 87



5. Insert consecutively nodes 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 into an initially empty red-black tree. Show and explain what happens at each step of your insertions.

Exercises

6. Let define a *relaxed* red-black tree as a binary search tree that satisfies redblack properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree T whose root is red. If we color the root of T black but make no other changes to T, is the resulting tree a red-black tree ?
7. Construct a red-black tree of height 2 with six nodes

Exercises

8. Construct a red-black tree of height 3 with six nodes
9. Construct two different red-black trees with the same three nodes

Exercises

9. Construct two different red-black trees with the same three nodes