アルゴリズムの設計と解析

教授:　黄　潤和　（W4022）

rhuang@hosei.ac.jp

SA:　広野　史明（A4/A8）

fumiaki.hirono.5k@stu.hosei.ac.jp

Contents (L8 – Search trees)

- Red Black Tree (review and deletion)
- 中間課題いついて

Outline

- What is Red-Black?
  赤黒木とは？
- From (2,4) trees to red-black trees
  2-4木から赤黒木へ
- Red-black tree　赤黒木
  - Insertion　挿入
    - restructuring　再構築
    - recoloring　再色付け
  - Deletion　削除
    - restructuring　再構築
    - recoloring　再色付け
    - adjustment　調整

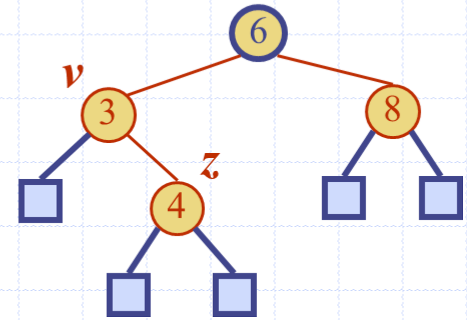
Red-Black Trees

video lecture

http://videolectures.net/mit6046jf05_demaine_lec10/

Watch about 25 minutes
to feel how a top university in the world gives lectures of CS

MIT - Massachusetts Institute of Technology



QS World University Rankings rates MIT No. 1 in 12 subjects for 2016 ...

news.mit.edu/2016/qs-world-university-rankings-rates-mit-no-1-in-... ▾ このページを訳す

2016/04/08 - QS World **University Rankings** has unveiled its lineup of the world's top universities for **2016**, by subject. **MIT** was honored with 12 No. 1 subject rankings, and 19 total top rankings (No. 5 or higher) out of 42 subjects.

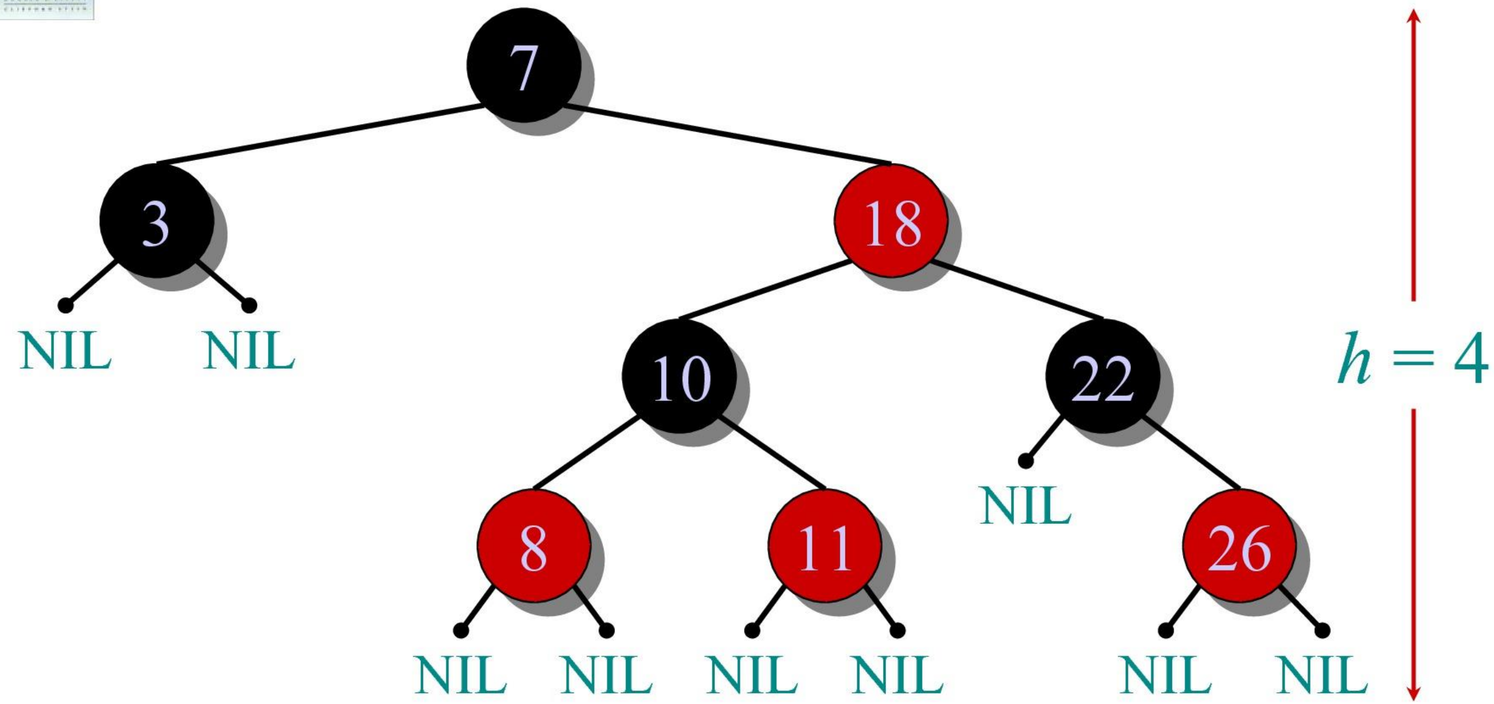
Red-black trees

This data structure requires an extra one-bit color field in each node.

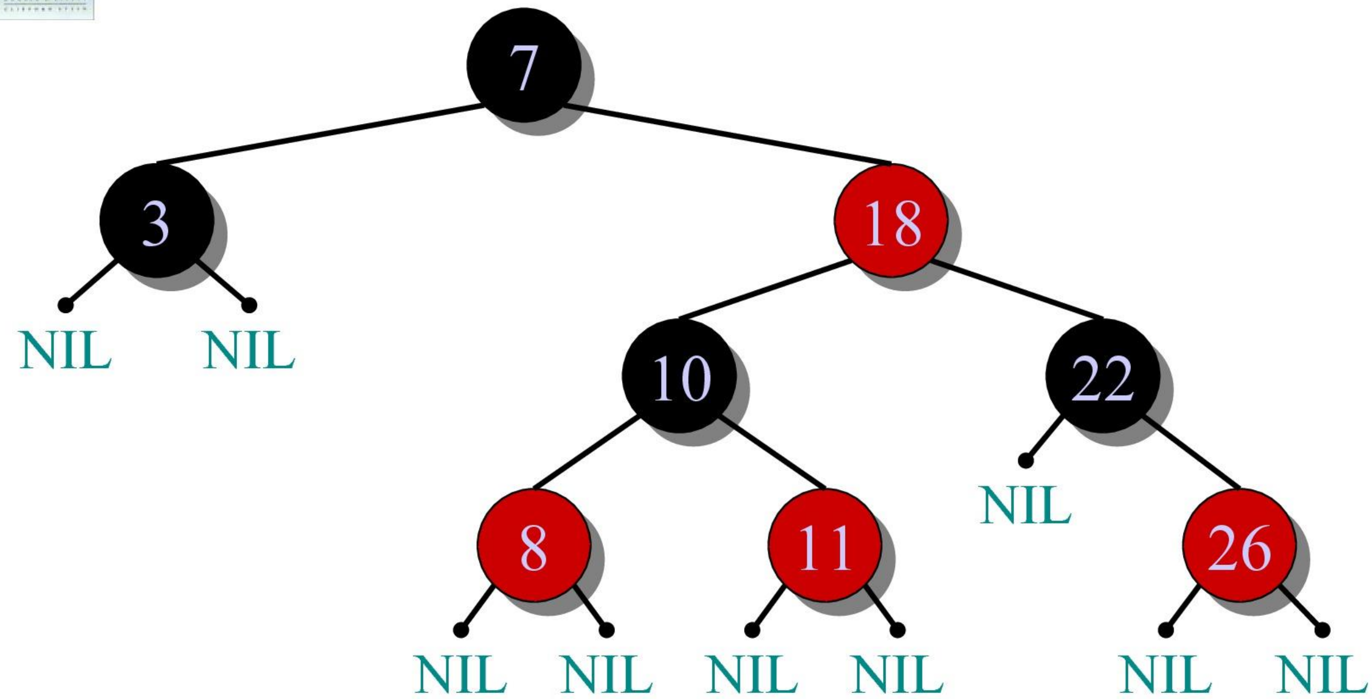
Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

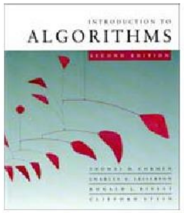
Example of a red-black tree



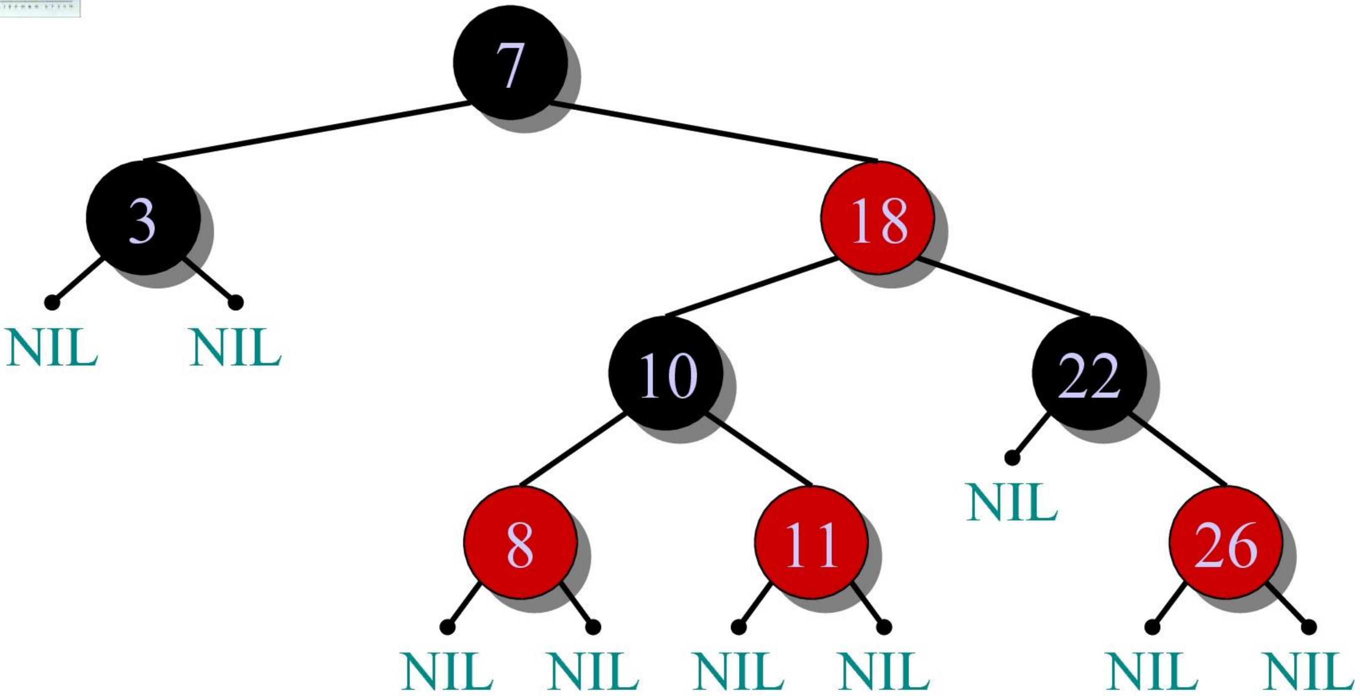
Example of a red-black tree



1. Every node is either red or black.

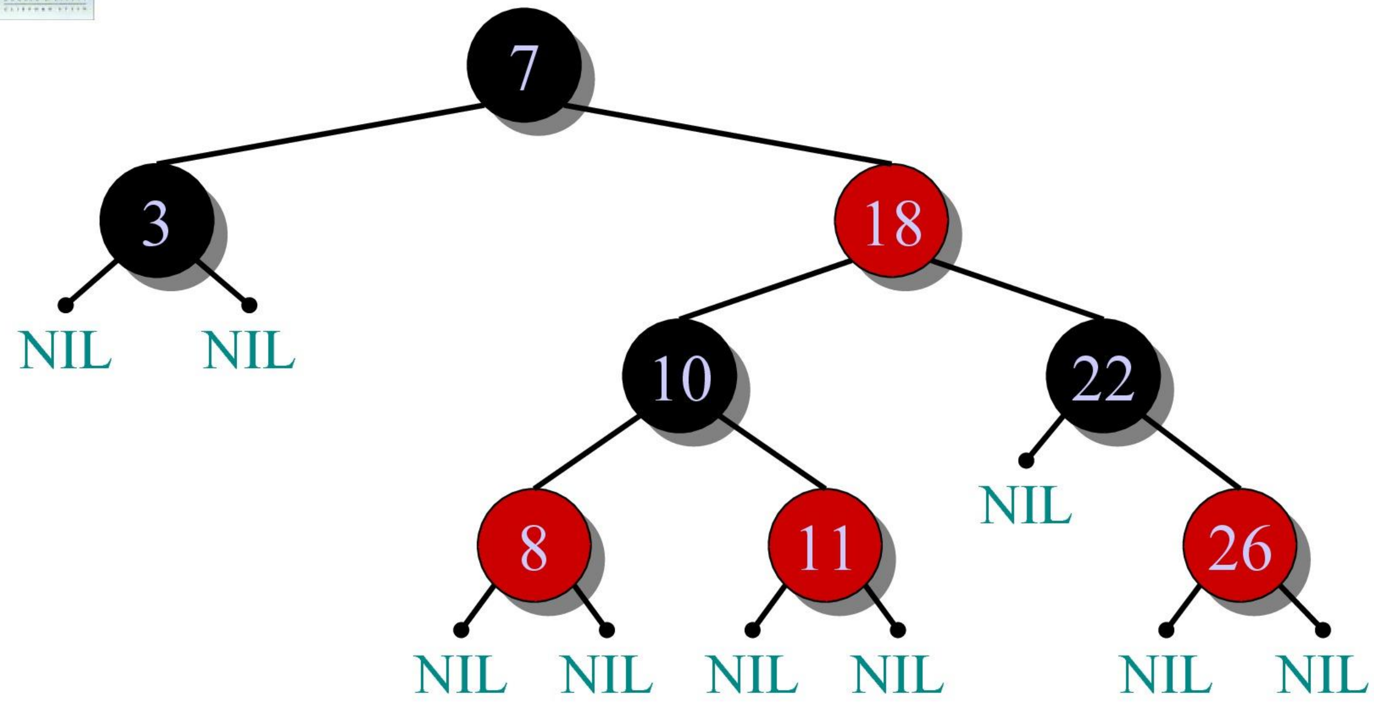


Example of a red-black tree

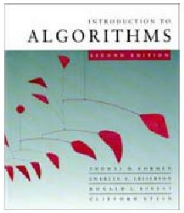


2. The root and leaves (NIL's) are black.

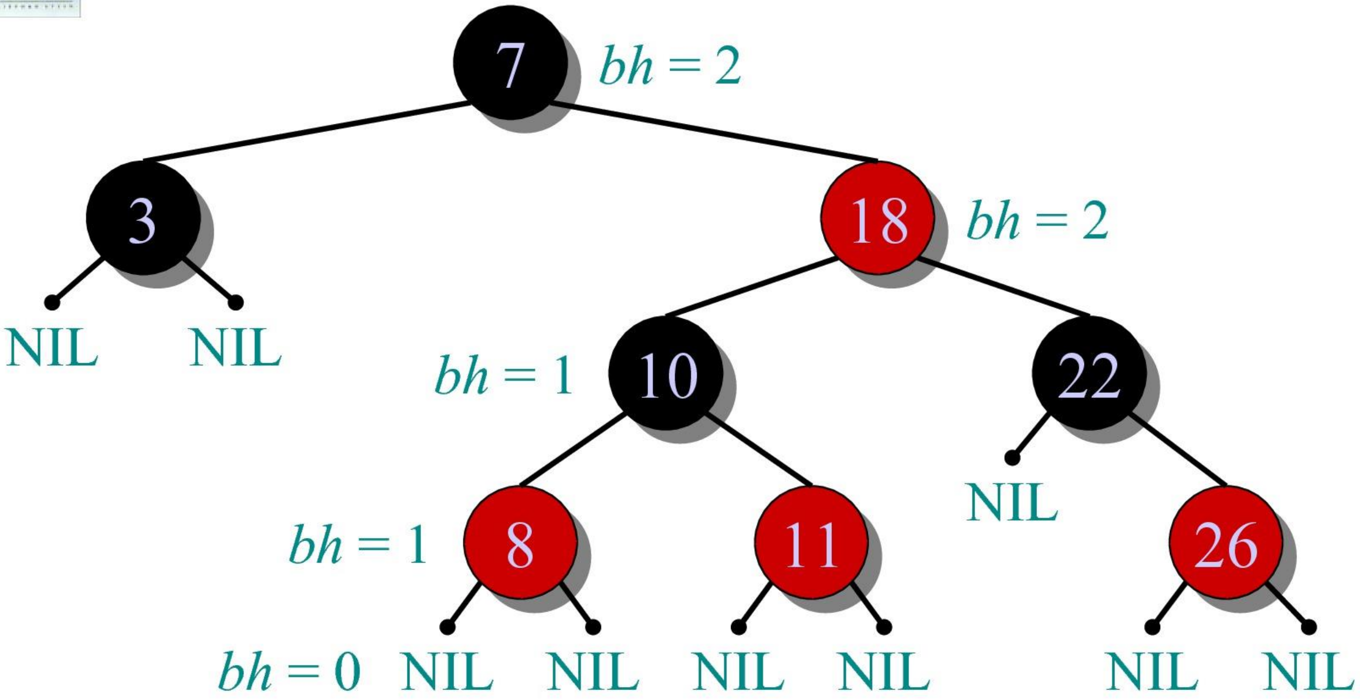
Example of a red-black tree



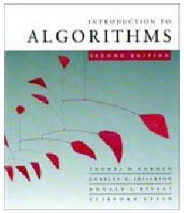
3. If a node is red, then its parent is black.



Example of a red-black tree



4. All simple paths from any node x to a descendant leaf have the same number of black nodes = *black-height*(x).



Height of a red-black tree

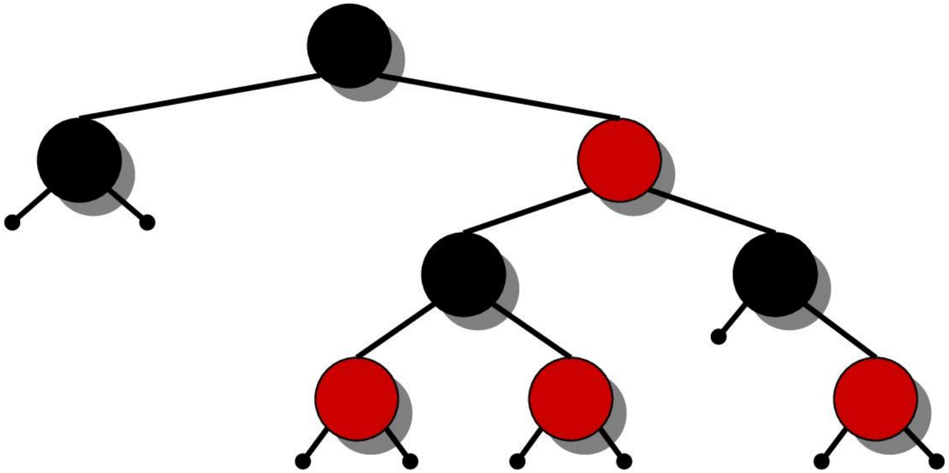
Theorem. A red-black tree with n keys has height

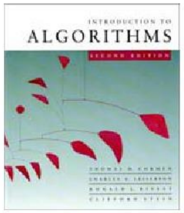
$$h \le 2 \lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

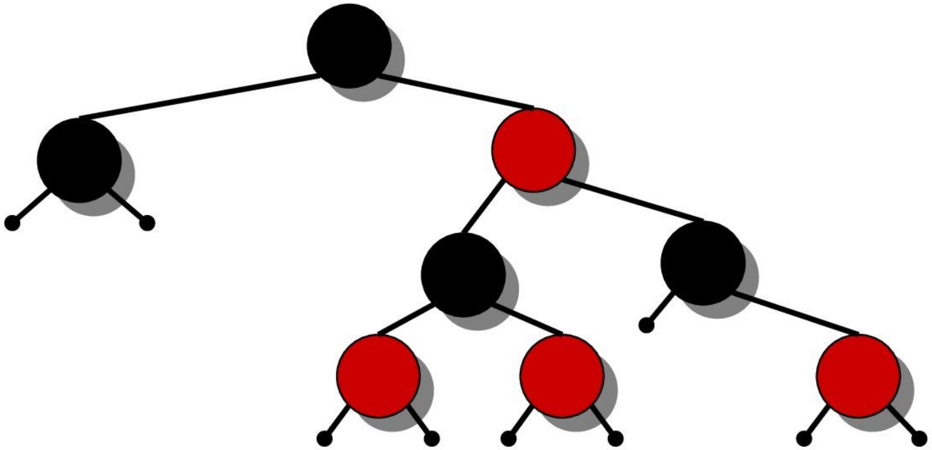
Theorem. A red-black tree with n keys has height

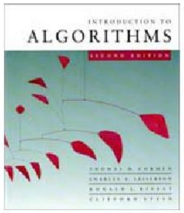
$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

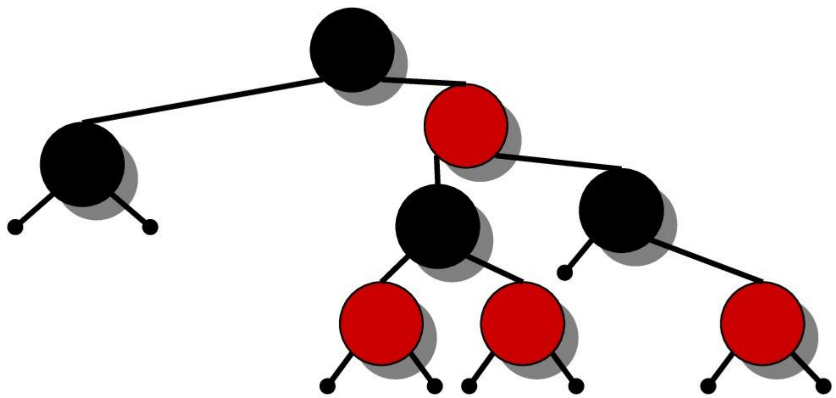
Theorem. A red-black tree with n keys has height

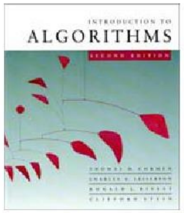
$$h \leq 2\lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

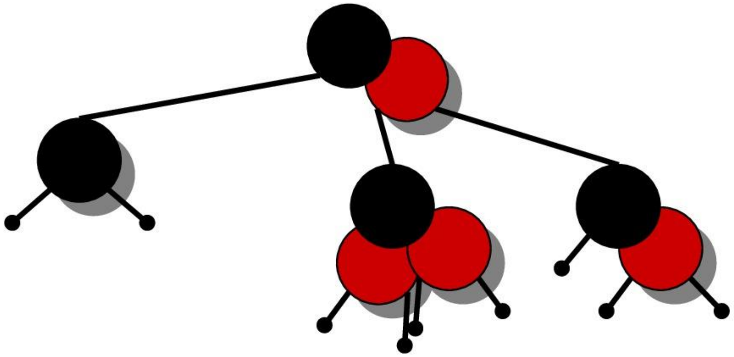
Theorem. A red-black tree with n keys has height

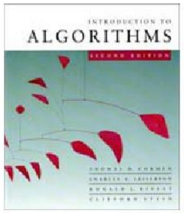
$$h \leq 2\lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

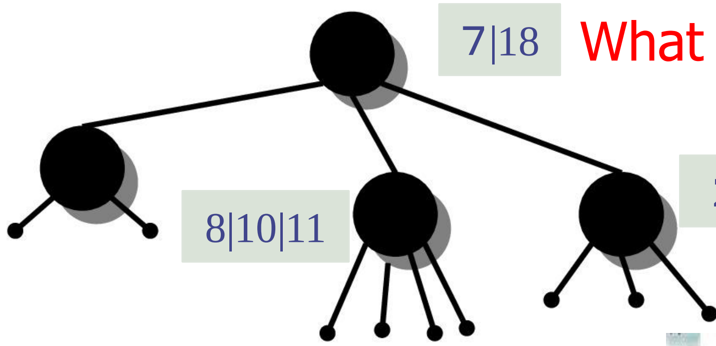
Theorem. A red-black tree with n keys has height

$$h \leq 2\lg(n+1).$$

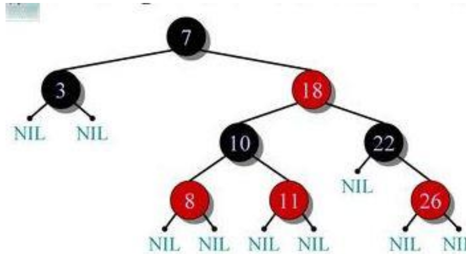
Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



What is this tree?



This is red-black tree?

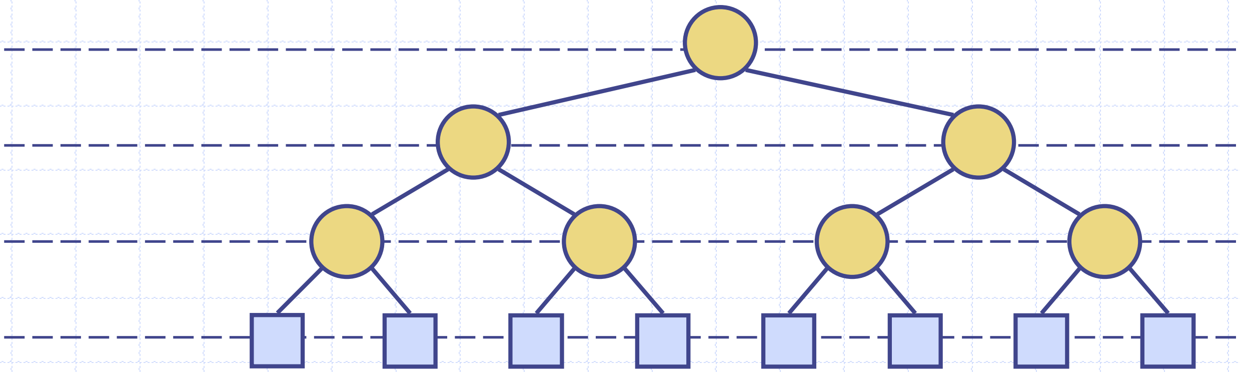
Height of a (2,4) Tree
(2,4)木の高さ

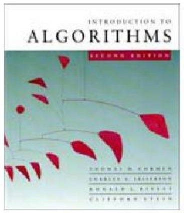
- **Theorem:** A (2,4) tree storing n items has height $O(\log_2 n)$

  Proof:
  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have

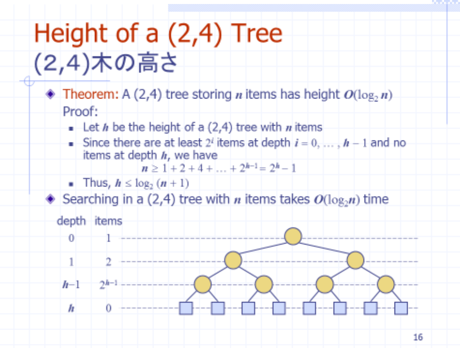
    $$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$
  - Thus, $h \leq \log_2 (n + 1)$
- Searching in a (2,4) tree with n items takes $O(\log_2 n)$ time

depth	items
0	1
1	2
$h-1$	2^{h-1}
h	0





Height of a red-black tr



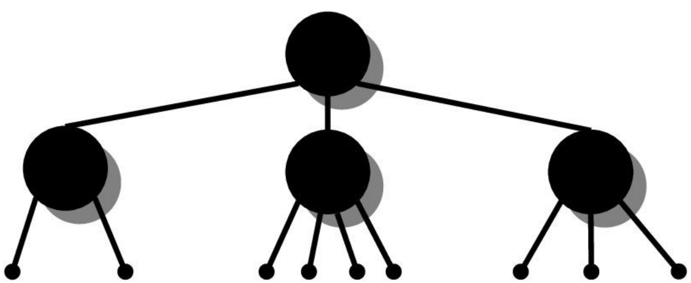
Theorem. A red-black tree with n keys has height

$$h \leq 2\lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.

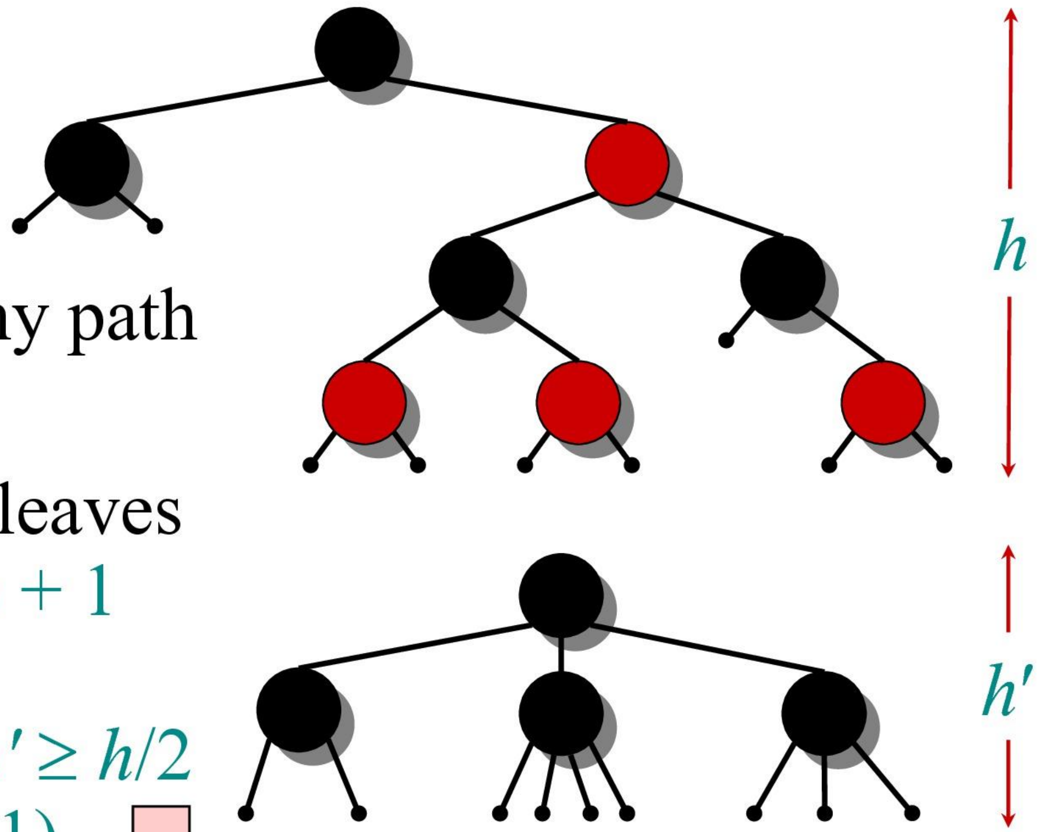


h' < = lg(n+1)

- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves.

Proof (continued) - Answer

- We have $h' \geq h/2$, since at most half the leaves on any path are red.
- The number of leaves in each tree is $n + 1$
  $\Rightarrow n + 1 \geq 2^{h'}$
  $\Rightarrow \lg(n + 1) \geq h' \geq h/2$
  $\Rightarrow h \leq 2 \lg(n + 1)$. □



- $h' \leq \log_2 (n + 1)$

Query operations

Corollary. The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\lg n)$ time on a red-black tree with n nodes.

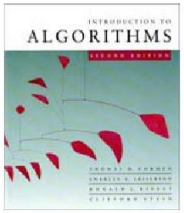
Modifying operations

The operations INSERT and DELETE cause modifications to the red-black tree:

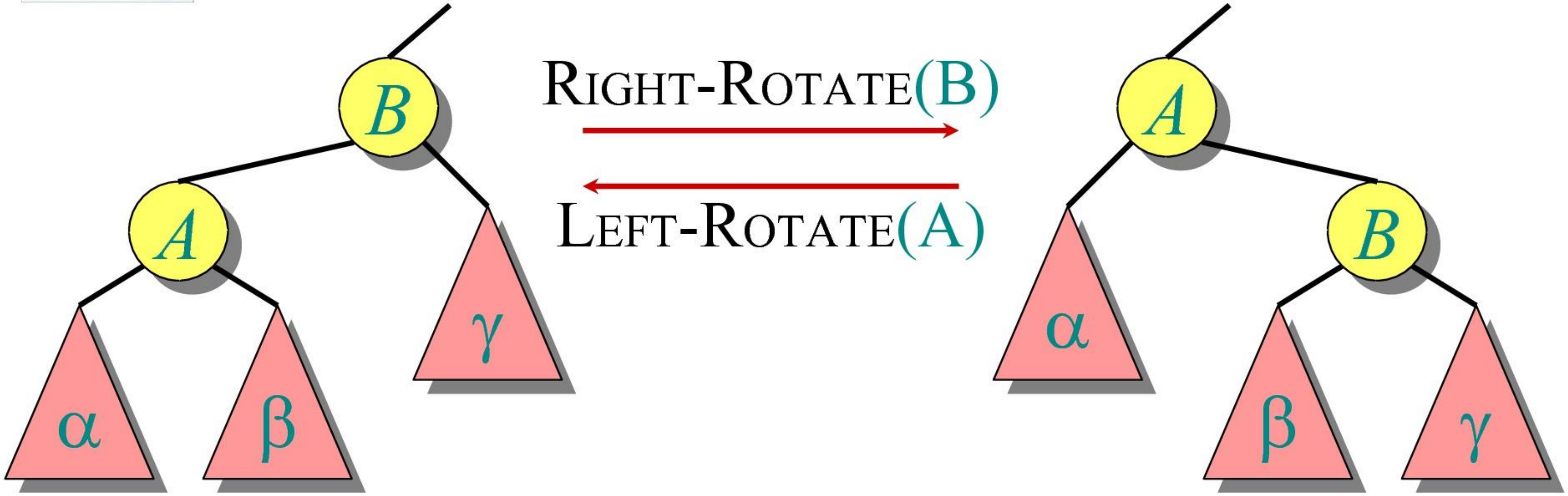
- the operation itself,
- color changes,
- restructuring the links of the tree via ***"rotations"***.

Do not forget to keep the properties
After do any operations ---------------→

Red-black properties:
1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).



Rotations



Rotations maintain the inorder ordering of keys:

- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \le A \le b \le B \le c$.

A rotation can be performed in $O(1)$ time.

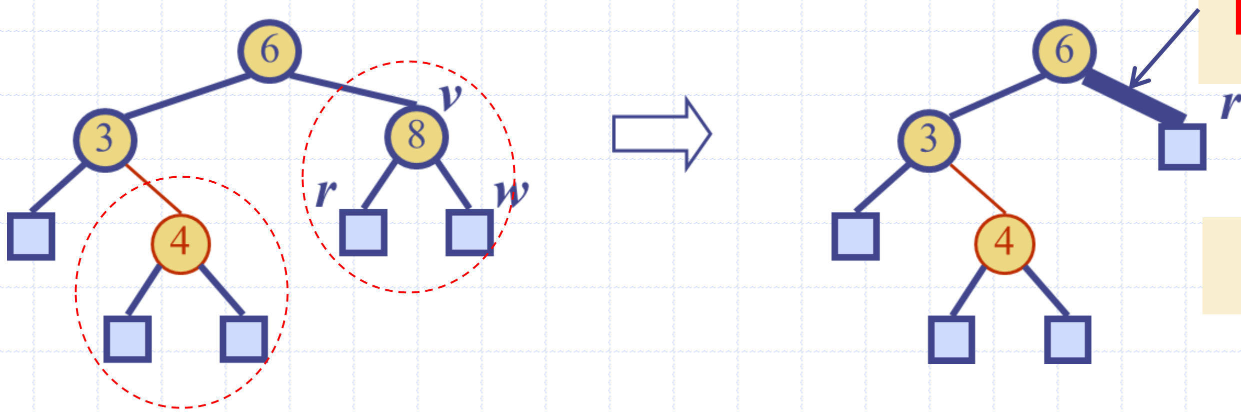
Red Black Tree deletion procedure

Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

Deletion 削除

- To perform operation remove(k), we first execute the deletion algorithm for binary search trees
  最初に2分探索木の削除アルゴリズムを用いる
- Let v be the internal node removed, w the external node removed, and r the sibling of w
  内部ノード v を削除すると、外部ノードの w と r も削除される。
  - If either v was red (r was black), no change
    or r was red (v was black), we color r black and we are done
  - Else (v and r were both black) we color r ***double black***, which is a violation of the internal property requiring a reorganization of the tree
- Example where the deletion of 8 causes a double black:



Double **black**!
Violation: Black height is changed

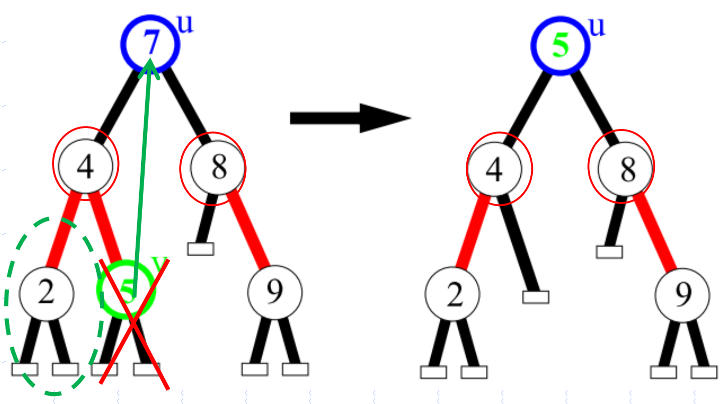
Keep black height = 2

Solution?

(1) Deletion – a node without external children (Swap → a node with external children)

If the key to be deleted is stored at a node that has no external children, we move there the key of its inorder predecessor (or successor), and delete that node instead

Example: to delete key 7, we move key 5 to node u, and delete node v

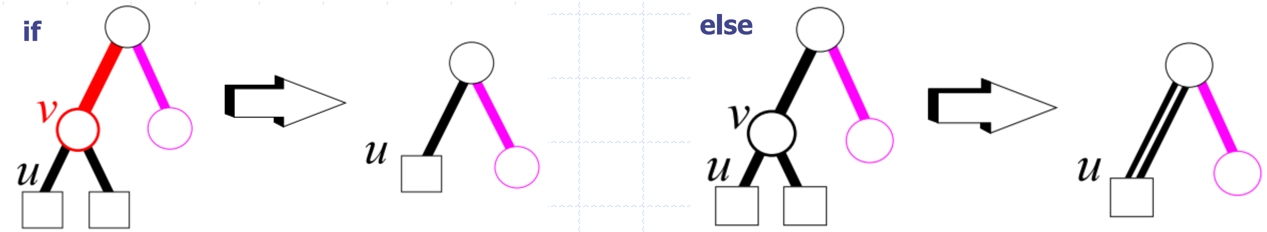


No double black

(2) Deletion – a node with external children

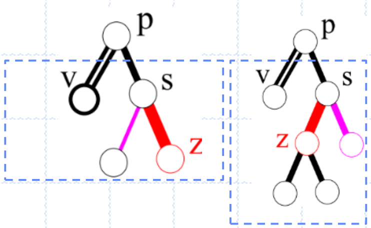
Remove v (when deleting a black node, double black problem!)

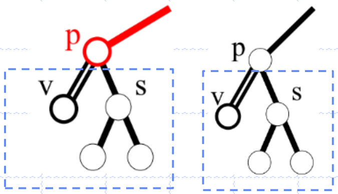
1. If v was red, color u black,
   else (v was black), color **u double black**.

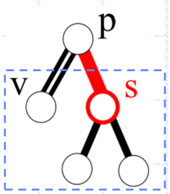


2. If double black edge exists, perform action for 3 cases:

- Case 1: black sibling ***s*** with a red child
- Case 2: black sibling ***s*** with black children
- Case 3: red sibling ***s***



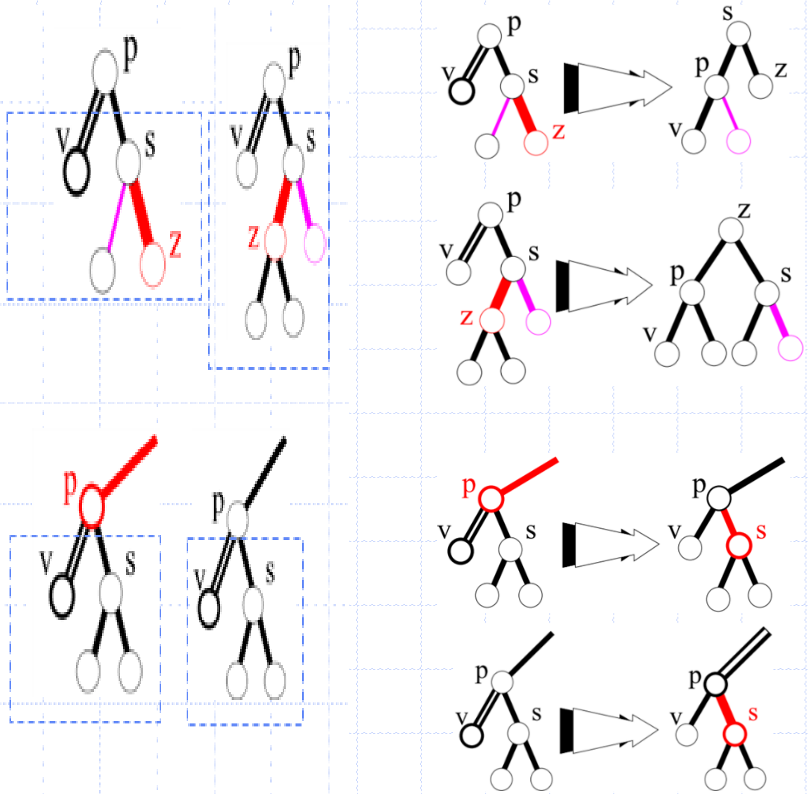




- Case 1: black sibling ***s*** with a red child

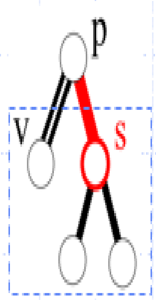
S black

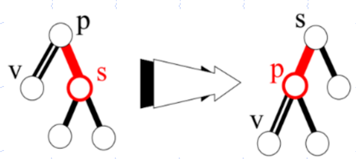
- Case 2: black sibling ***s*** with black children



- Case 3: red sibling ***s***

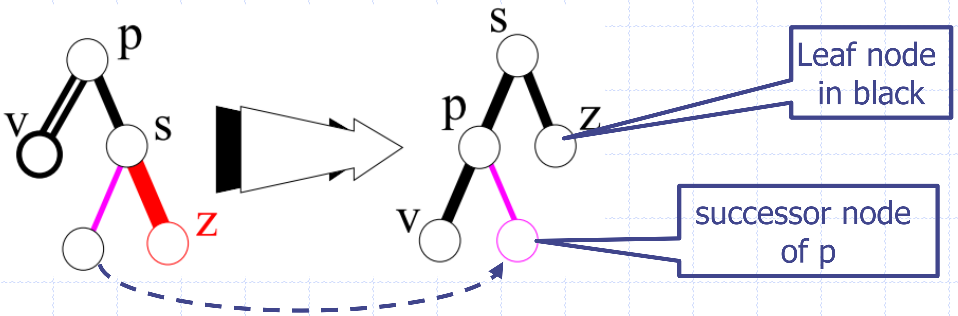
S red

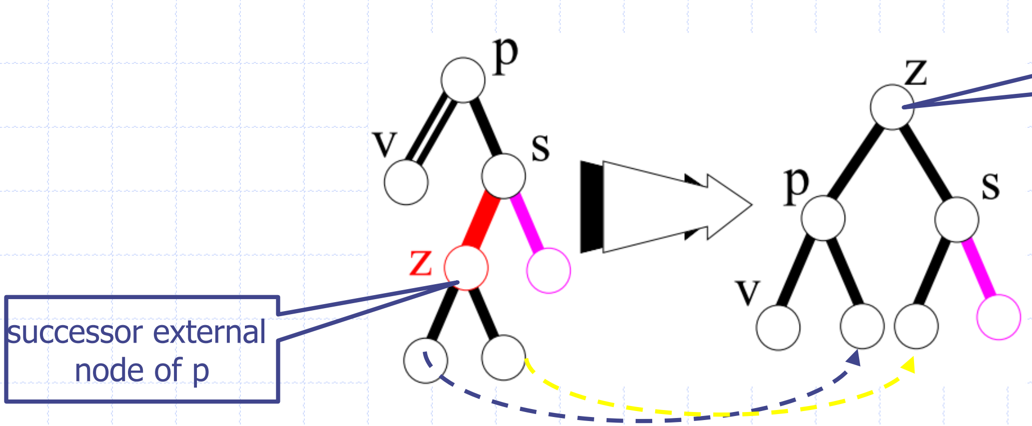




Case 1: black sibling ***s*** with a red child

- If sibling ***s*** is black and one of its children is red,
  → perform a restructuring (rotation and place children nodes in right positions)

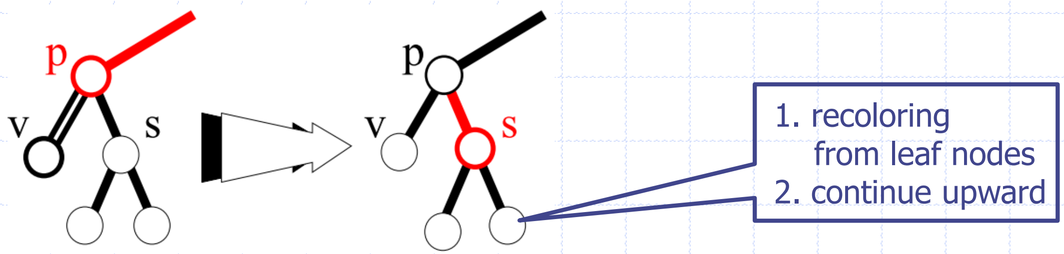


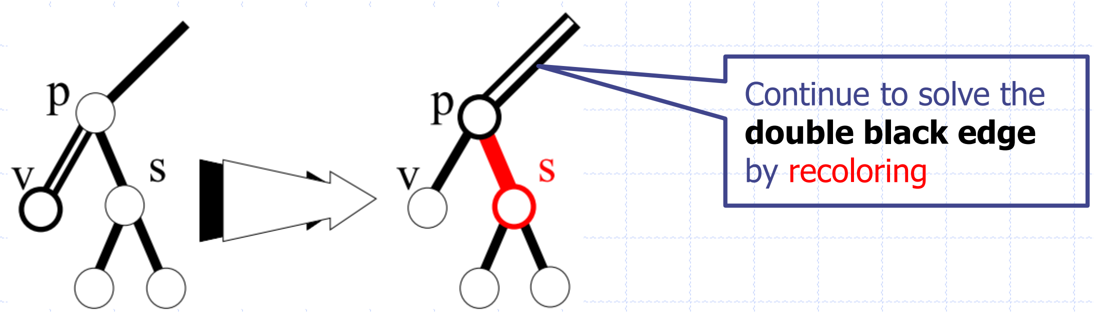


Keep node vales in order!

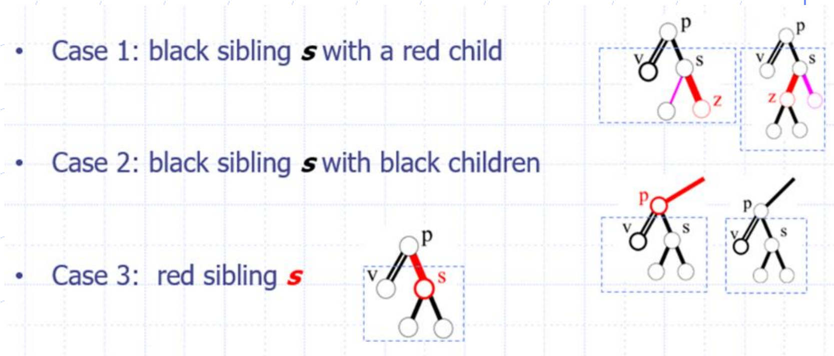
Case 2: black sibling *s* with black children

- If sibling and its children are black,
  →perform recoloring
  If parent becomes double black,
  →continue upward

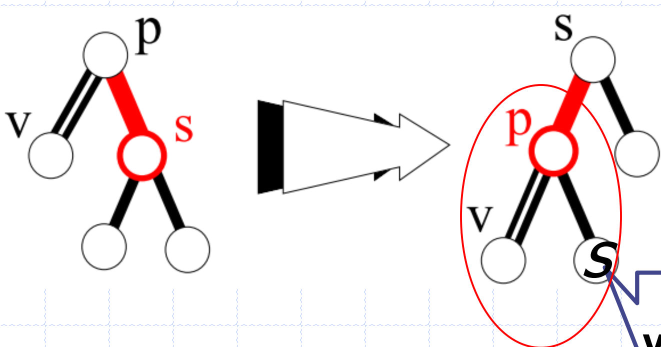




Case 3: red sibling *s*

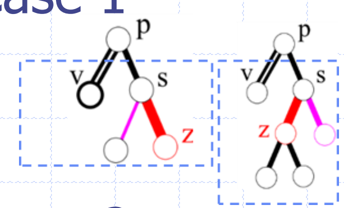


- If sibling *s* is red,
  → perform an adjustment
  then its sibling is black (Case 3 becomes Case 1 or Case 2)

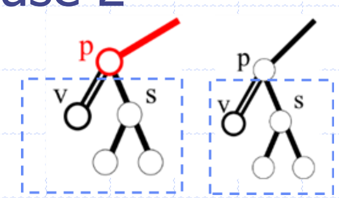


V's sibling (S) becomes **black**,
Which becomes Case 1&2

Case 1

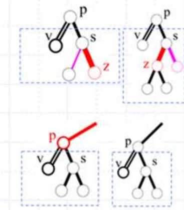


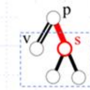
Case 2



Some examples

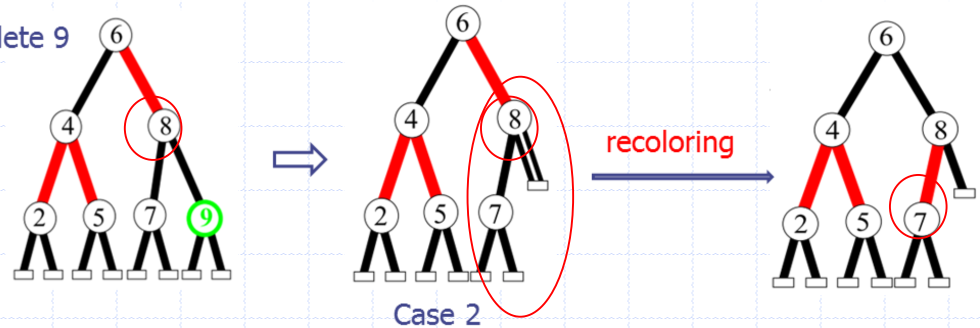
- Case 1: black sibling ***s*** with a red child
- Case 2: black sibling ***s*** with black children
- Case 3: red sibling ***s***





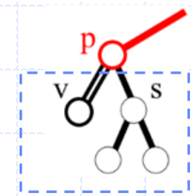
Delete 9, 8, 7

Delete 9

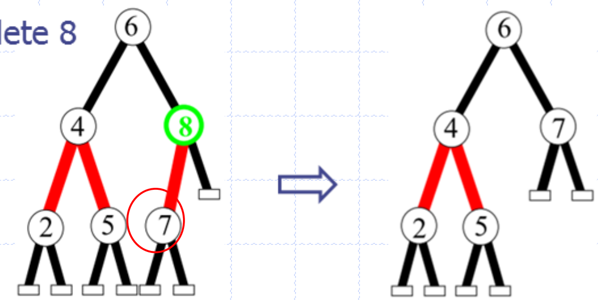


Case 2

Which case?

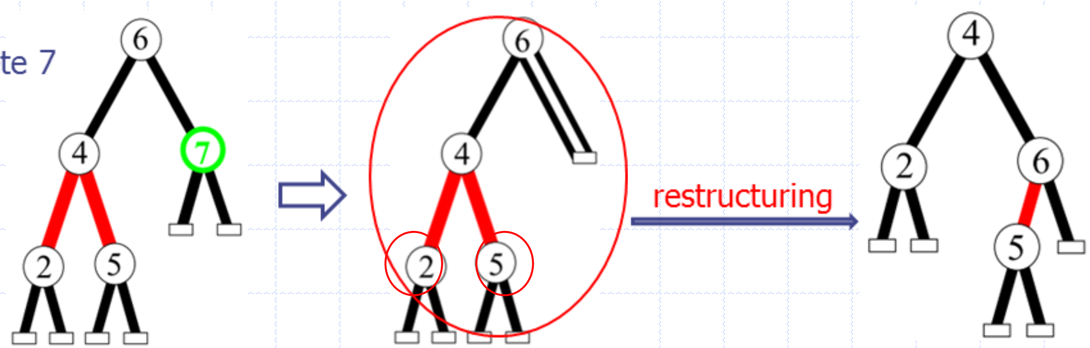


Delete 8



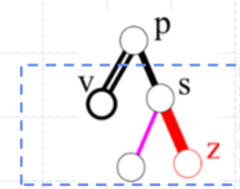
No double black,
no further process.

Delete 7

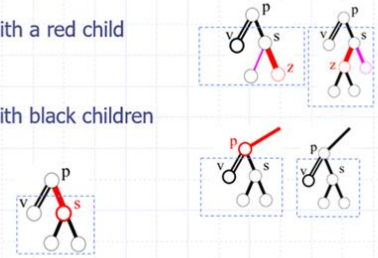


Case 1

Which case?



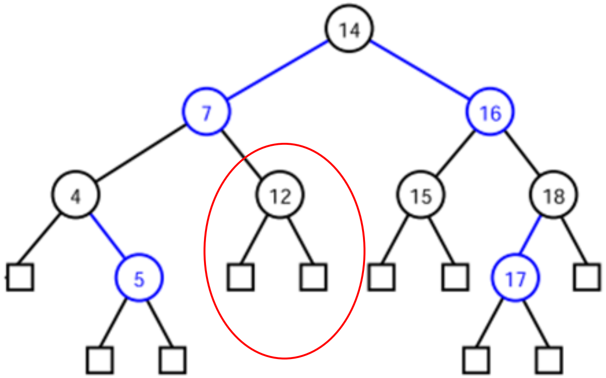
- Case 1: black sibling *s* with a red child
- Case 2: black sibling *s* with black children
- Case 3: red sibling *s*



One more example

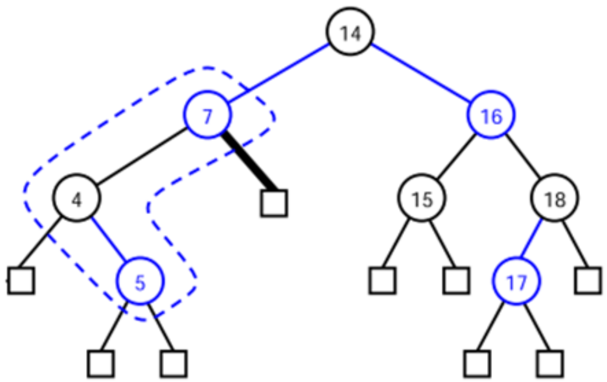
Note: Blue color nodes refer to red node

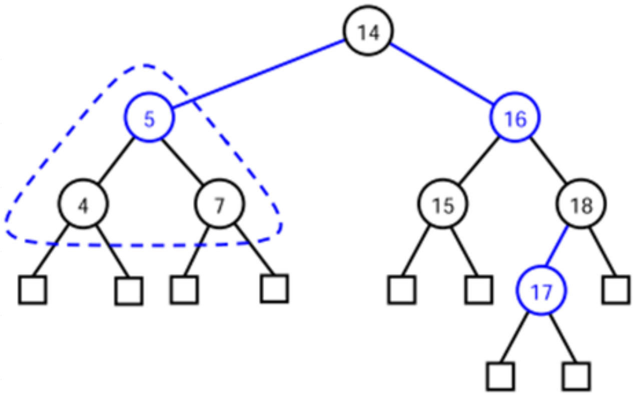
Delete 12



Q1: Which case?
- case 1
- case 2
- case 3

Q2: What operation?
- restructuring
- recoloring
- adjustment





Analysis of Deletion
挿入の分析

Algorithm *deleteItem*(*k*, *o*)

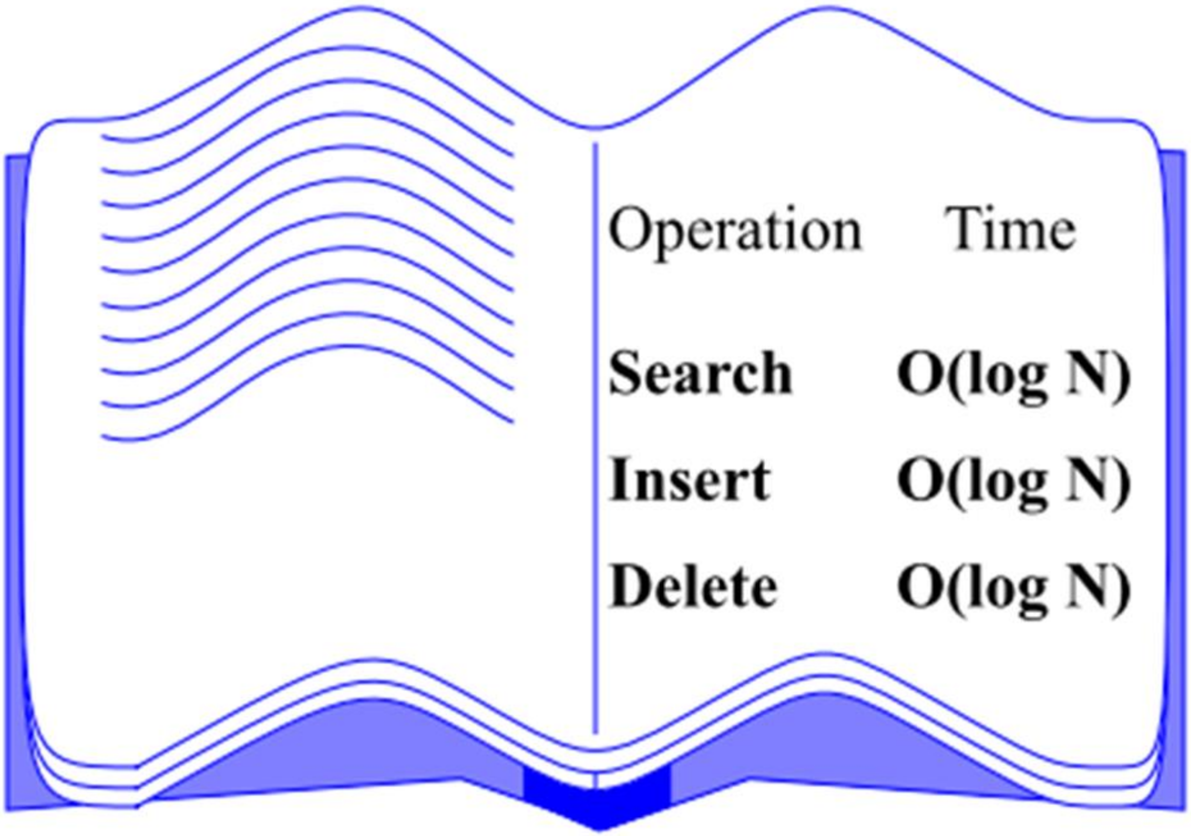
1. We search for key ***k*** to locate the node ***v***

2. We delete node ***v*** and

3. **while** *doubleBlack*(*v*)
   if *isBlack*(*sibling*(*v*))
      if *isRed*(*sibling(oneOfChildren*(*v*)))
         restructuring()
      else
         recoloring()
   else { *sibling*(*v*) is red }
      adjustment()

- Recall that a red-black tree has $O(\log n)$ height
- Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(\log n)$ time because we perform
  - at most one restructuring taking $O(1)$ time
  - $O(\log n)$ recoloring,
  - $O(\log n)$ adjustmen,
- **Thus, an deletion in a red-black tree takes $O(\log n)$ time**

Time Complexity of Red-Black Trees



Red-Black tree demo

Demo
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Java implementation in Java
And more explanations in this site
http://fujimura2.fiw-web.net/java/mutter/tree/red-black-tree.html

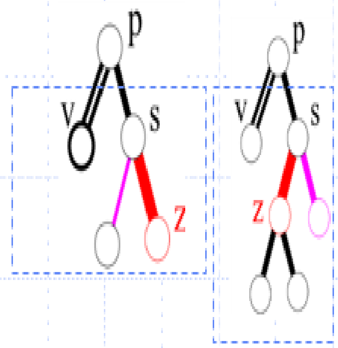
RedBlackTree.java

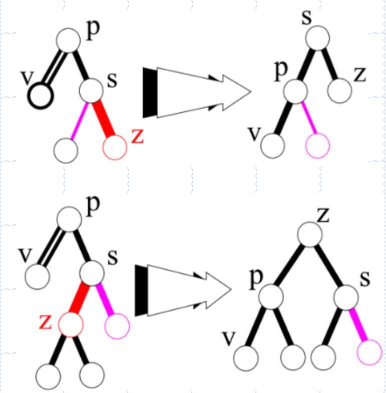
Read and understand
If possible, execute the program.

```
/**
 * 赤黒木 Red Black Tree
 *      @see <A HREF="http://www.geocities.jp/h2fujimura/mutter/tree/red-black-tree.html">赤黒木</A>
 *      @author Hikaru Fujimura
 *      @version 2005.08.31
 */
public class RedBlackTree<T extends Comparable<? super T>> extends BinaryTree<T> {

        private boolean color;
        public static boolean BLACK = true;
        public static boolean RED = false;

/** ノードの生成 */
        private RedBlackTree(T v, boolean c, RedBlackTree<T> p, RedBlackTree<T> l, RedBlackTree<T> r) {
                value = v;
                color = c;
                parent = p;
                left = l;
                right = r;
        }
```

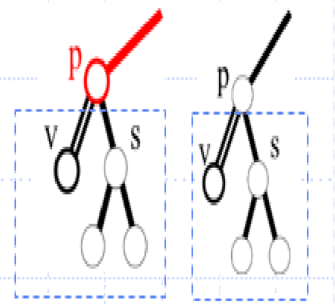
```java
/**
 * 要素 v だけをもつ２分探索木の生成します。
 * 生成した木を部分木として、NIL とおきかえるには、
 * setAsLeaf メソッドを使用してください。
 *        @param  v   根に設定する値
 */
        RedBlackTree(T v) {
                if(v==null) throw new IllegalArgumentException();
                value  = v;
                color = BLACK;
                parent = null;
                left   = new RedBlackTree<T>(null, BLACK, this, null, null);
                right  = new RedBlackTree<T>(null, BLACK, this, null, null);
        }

        private RedBlackTree(T v, boolean c) {
                this(v);
                color = c;
        }

/**
 * 空木の生成します。
 * NIL としても使われます。
 */
        RedBlackTree() {
                this(null, BLACK, null, null, null);
        }
```

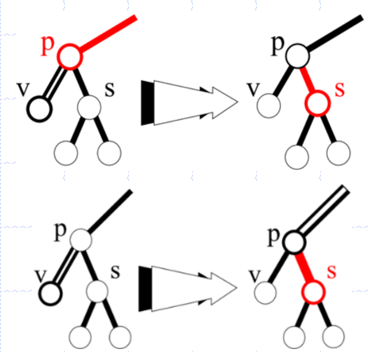
- Case 1: black sibling ***s*** with a red child



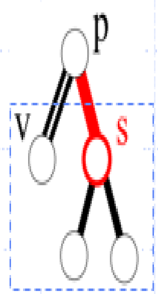


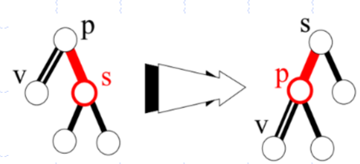
- Case 2: black sibling ***s*** with black children





- Case 3: red sibling ***s***

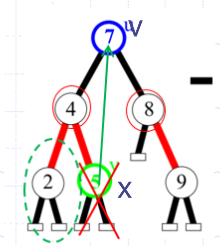




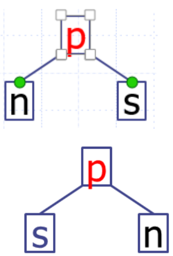
n cases
- leaf node: no children
- no left branch
- no right branch
- has two branches

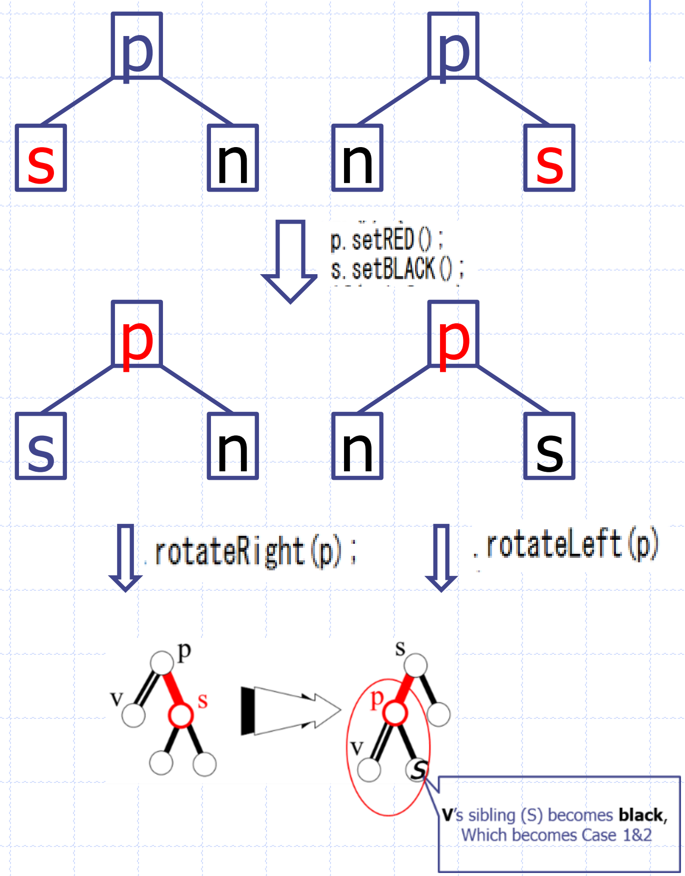
```
/**
 * この２分探索木から 値v を削除します。
 *
 *         @param v     削除する値
 *         @return      元の２分探索木（削除後）
 */
        RedBlackTree<T> delete(T v) {
                if(isNIL()) return this;                              // empty tree
                RedBlackTree<T> n = (RedBlackTree<T>)search(v);       // search
                if(n.isNIL()) return this;                            // not found
                boolean c = n.color;                                  // color of the node deleting
                if(n.left.isNIL() && n.right.isNIL()) {               // no child
                        n.value = null;                                   // change to NIL
                        n.left = null;                                    //
                        n.right = null;                                   //
                        return checkRB(n, c);                       // only parent chain
                }
                if(n.left.isNIL()) {                                  // no left subtree
                        n = (RedBlackTree<T>)n.replace(n.right);          // replace here by right subtree
                        if(n.isRoot()) return n.checkRB(n, c);
                        else           return checkRB(n, c);
                }
                if(n.right.isNIL()) {                                 // no right subtree
                        n = (RedBlackTree<T>)n.replace(n.left);           // replace here by left subtree
                        if(n.isRoot()) return n.checkRB(n, c);
                        else           return checkRB(n, c);
                }                                                      // now, t has 2 subtree
                RedBlackTree<T> x = (RedBlackTree<T>)n.left;           // get smaller subtree
                while(!x.right.isNIL()) {                              // as far as right subtree exist
                        x = (RedBlackTree<T>)x.right;                      // get greater value
                }                                                      //
                T prev = x.value;                                      // maximun value that less than v
                c = x.color;
                x = (RedBlackTree<T>)x.replace(x.left);                // replace here by left
                n.value = prev;                                        // replace v
                return checkRB(x, c);                                  // and return
        }
```

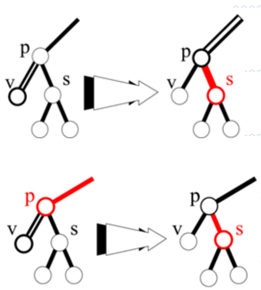
For swap



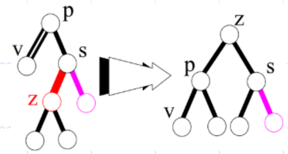
```
/** ノードの色の調整 */
        private RedBlackTree<T> checkRB(RedBlackTree<T> n, boolean c) {
//              boolean debug = n.checkOut(-325455329);
//              if(debug && c==RED) System.out.println("deleted node was RED");
                if(c==RED) return this;
                if(n.isRED()) {
                        n.color = BLACK;
                        return this;
                }
                RedBlackTree<T> tree = this;
                while(true) {
                        if(n.isRoot()) {
                                n.color = BLACK;
                                return tree;
                        }
                        RedBlackTree<T> p = (RedBlackTree<T>)n.parent;
                        RedBlackTree<T> s = (RedBlackTree<T>)n.getBrother();
//                      if(debug) System.out.println(" p:"+ p.toString()+ " s:"+s.toString());
        Case 3          if(s.isRED()) {
                                p.setRED();
                                s.setBLACK();
                                if(p.left==n) {
                                        tree = (RedBlackTree<T>)tree.rotateLeft(p);
                                        s = (RedBlackTree<T>)p.right;
                                }
                                else {
                                        tree = (RedBlackTree<T>)tree.rotateRight(p);
                                        s = (RedBlackTree<T>)p.left;
                                }
                        }
                        RedBlackTree<T> sl = (RedBlackTree<T>)s.left;
                        RedBlackTree<T> sr = (RedBlackTree<T>)s.right;
        Case 2          if(p.isBLACK() && sl.isBLACK() && sr.isBLACK()) {
                                s.setRED();
                                n = p;
                                continue;
                        }
        Case 2          if(sl.isBLACK() && sr.isBLACK()) {
                                p.setBLACK();
                                s.setRED();
                                return tree;
                        }
```







Case 1: black sibling s with a red left child

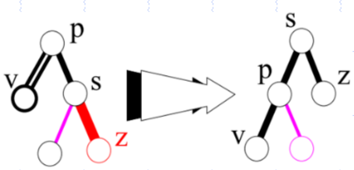



```
if(p.left==n && sl.isRED() && sr.isBLACK()) {
        sl.setBLACK();
        s.setRED();
        tree = (RedBlackTree<T>)tree.rotateRight(s);
        sr = s;
        s = sl;
        sl = null;  // don't use anymore
} else if(p.right==n && sl.isBLACK() && sr.isRED()) {
```

Case 1: the above case's mirror case

```
        sr.setBLACK();
        s.setRED();
        tree = (RedBlackTree<T>)tree.rotateLeft(s);
        sl = s;
        s = sr;
        sr = null;  // don't use this subtree anymore
}
if(p.left==n && sr.isRED()) {
```

Case 1: black sibling s with a red right child




```
        boolean t = p.color;
        p.color = s.color;
        s.color = t;
        sr.setBLACK();
        tree = (RedBlackTree<T>)tree.rotateLeft(p);
        return tree;
} else if(p.right==n && sl.isRED()) {
```

Case 1: the above case's mirror case

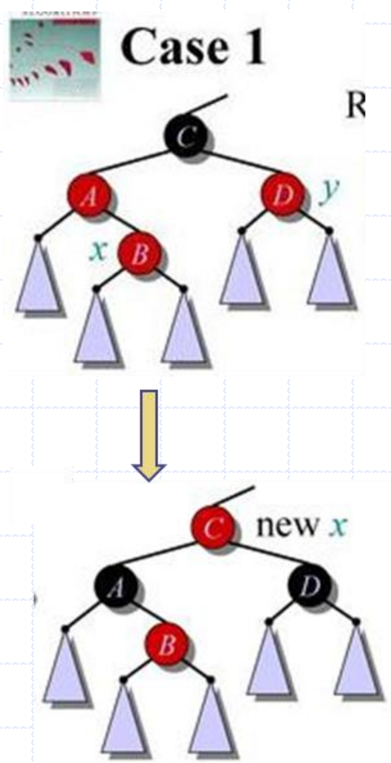
```
        boolean t = p.color;
        p.color = s.color;
        s.color = t;
        sl.setBLACK();
        tree = (RedBlackTree<T>)tree.rotateRight(p);
        return tree;
}
```

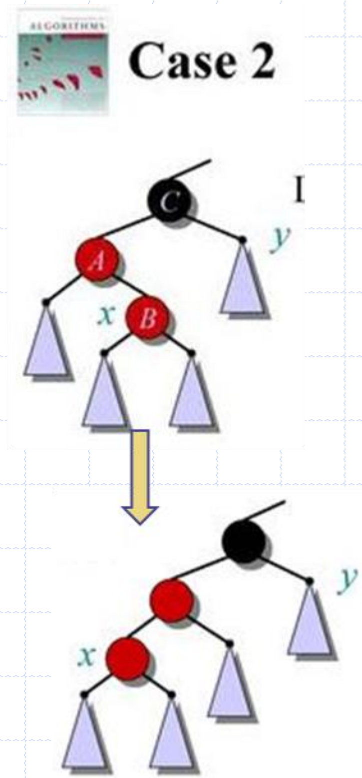
元の2分探索木、空木に挿入した場合は新しい木

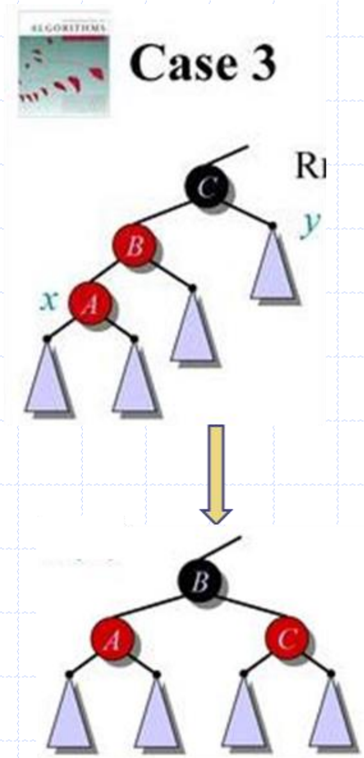
```
/**
 * この２分探索木に 値v を挿入します。
 *
 *        @param v     挿入する値
 *        @return      元の２分探索木、空木に挿入した場合は新しい木
 */
       RedBlackTree<T> insert(T v) {
               if(isNIL()) return new RedBlackTree<T>(v, BLACK); // empty tree, insert as BLACK
               BinaryTree<T> nn = search(v);                     // search v as Binary Search Tree
               RedBlackTree<T> n = (RedBlackTree<T>)nn;          //
               while(!n.isNIL()) {                               // found?
                   n = (RedBlackTree<T>)(n.right).search(v);    // yes, check right subtree
               }
               n.setAsLeaf(v, RED);                              // insert as RED node
               RedBlackTree<T> p = null;                             // parent
               RedBlackTree<T> g = null;                             // grand parent
               RedBlackTree<T> u = null;                             // uncle
```

Insertion 3 cases

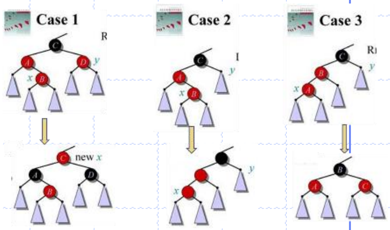
Of course, in fact, there are 6 cases since each of them has the mirror case







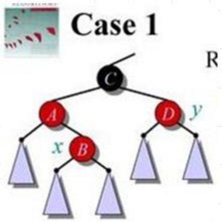
```
while(true) {
  if(n.isRoot()) {                              // if n is root
     n.setBLACK();                             // n can be BLACK always(case 1a)
        return n;                                 //
        }                                            // n is not root
        p = (RedBlackTree<T>)n.parent;               // get parent
        if(p.isBLACK()) {                            // case 1b?
                return this;                          // n can be RED, if parent is BLACK
        }
        g = (RedBlackTree<T>)p.parent;               // RED parent has parent
        u = (RedBlackTree<T>)n.getUncle();           // so, n has uncle
        if(u!=null && u.isRED()) {                  // parent and uncle are RED(case 2)
                g.setRED();                               //
                p.setBLACK();                             //
                u.setBLACK();                             //
                n = g;                                    //
                continue;                                 //
        }
        RedBlackTree<T> temp;                        // temp for exchange
        if(g.left==p && p.right==n) {                // left pattern of case 3
                g = (RedBlackTree<T>)g.rotateLeft(p);
                temp = p;
                p = n;
                n = temp;
        } else if(g.right==p && p.left==n) {         // right pattern of case 3
                g = (RedBlackTree<T>)g.rotateRight(p);
                temp = p;
                p = n;
                n = temp;
        }
        if(g.left==p && p.left==n) {                 // left pattern of case 4
                g.setRED();
                p.setBLACK();
                g = (RedBlackTree<T>)g.rotateRight(g);
        } else if(g.right==p && p.right==n) {        // right pattern of case 4
                g.setRED();
                p.setBLACK();
                g = (RedBlackTree<T>)g.rotateLeft(g);
        } else System.out.println("oops!insert to:"+this.toLongString());
        if(g.isRoot()) return g;
        else           return this;
}
```



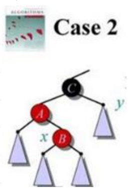
Work in class:
Please read and understand the program and mark three cases

```
while(true) {
  if(n.isRoot()) {                          // if n is root
     n.setBLACK();                          // n can be BLACK always(case 1a)
        return n;                           //
        }                                      // n is not root
        p = (RedBlackTree<T>)n.parent;         // get parent
        if(p.isBLACK()) {                      // case 1b?
               return this;                     // n can be RED, if parent is BLACK
        }
        g = (RedBlackTree<T>)p.parent;         // RED parent has parent
        u = (RedBlackTree<T>)n.getUncle();     // so, n has uncle
        if(u!=null && u.isRED()) {            // parent and uncle are RED(case 2)
               g.setRED();                          //
               p.setBLACK();                        //
               u.setBLACK();                        //
               n = g;                               //
               continue;                            //
        }
        RedBlackTree<T> temp;                  // temp for exchange
        if(g.left==p && p.right==n) {          // left pattern of case 3
               g = (RedBlackTree<T>)g.rotateLeft(p);
               temp = p;
               p = n;
               n = temp;
        } else if(g.right==p && p.left==n) {   // right pattern of case 3
               g = (RedBlackTree<T>)g.rotateRight(p);
               temp = p;
               p = n;
               n = temp;
        }
        if(g.left==p && p.left==n) {           // left pattern of case 4
               g.setRED();
               p.setBLACK();
               g = (RedBlackTree<T>)g.rotateRight(g);
        } else if(g.right==p && p.right==n) {  // right pattern of case 4
               g.setRED();
               p.setBLACK();
               g = (RedBlackTree<T>)g.rotateLeft(g);
        } else System.out.println("oops!insert to:"+this.toLongString());
        if(g.isRoot()) return g;
        else           return this;
}
```

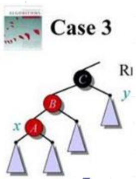
case 1

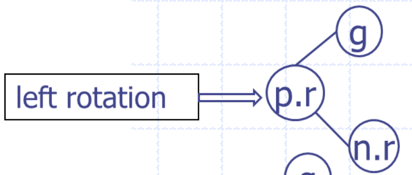


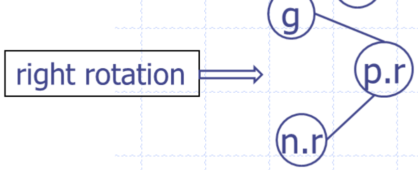
case 2
-> case 3

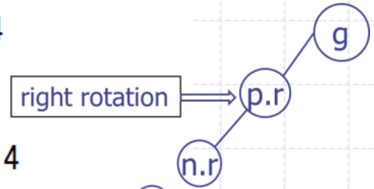


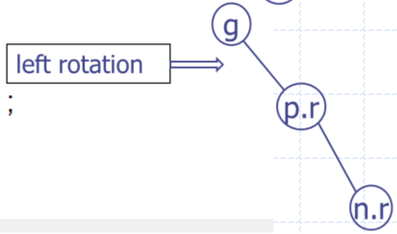
case 3





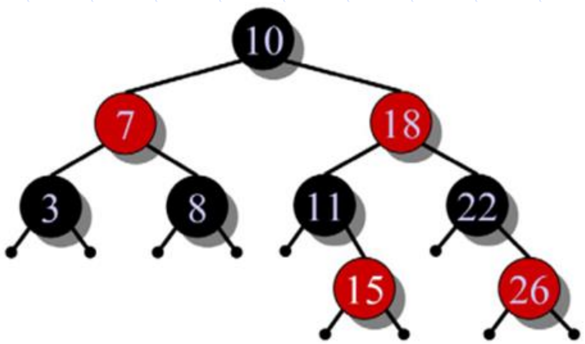






Exercise 8-1

Consider the following sequence of keys: (15, 11, 26). Delete the items with this set of keys in the order given into the red-black tree below. Draw the tree after each deletion.



キー配列について考える: (15, 11, 26)
このキーのセットを図の赤黒木に削除しなさい。
それぞれの削除後の赤黒木を描きなさい。

中間課題

1. Syllabus (2017 Syllabus)
2. L1 (Review data structures and algorithms(1))
3. L2 (Review basic algorithm analyses – Divide and Conquer)
4. L3 (Review basic algorithm analyses – Dynamic Programming)
5. L4 (Review Trees – traversal and math expressions)
6. L5 (Trees – AVL Tree, 2-3-4 Tree insertion)
7. L6 (2-3-4 Trees deletion)
8. L7 (Red-black Tree)

1. What is the Divide and Conquer algorithm and take an example to explain
2. What is the Dynamic Programming and take an example to explain
3. Redo Exercise 4.2 and 4.3
4. Proof: a 2-3-4 tree storing *n* items has height $O(\log_2 n)$ and Redo Ex 5.1
5. What are rotation and merge operations in a 2-3-4 tree deletion procedure? use examples to explain.
6. State the relation between a red-black tree and a 2-3-4 tree and Redo Ex 7.2 and do Ex 8.1
7. Summarize the 3 cases in the insertion procedure and the 3 cases in the deletion procedure of a red-black tree