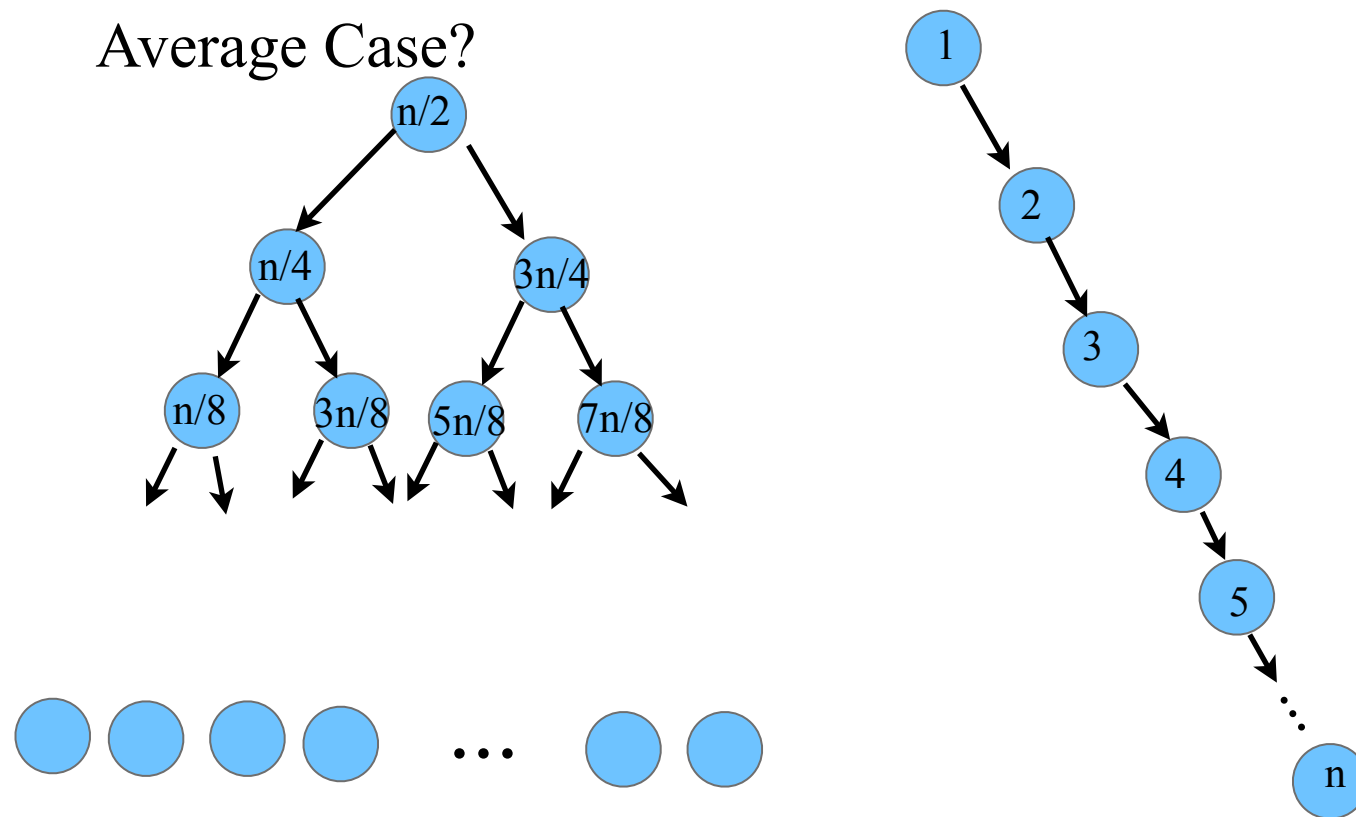
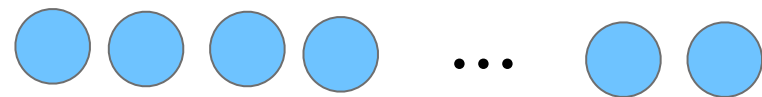
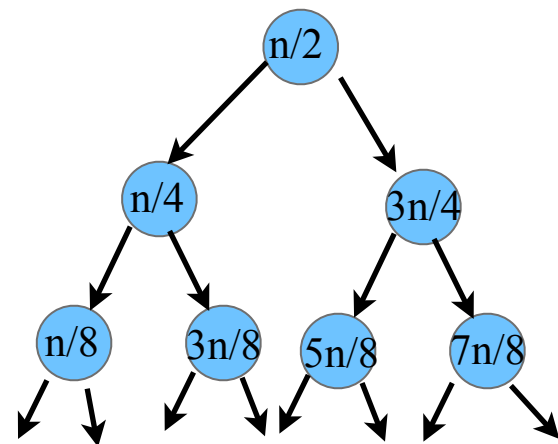


Motivation

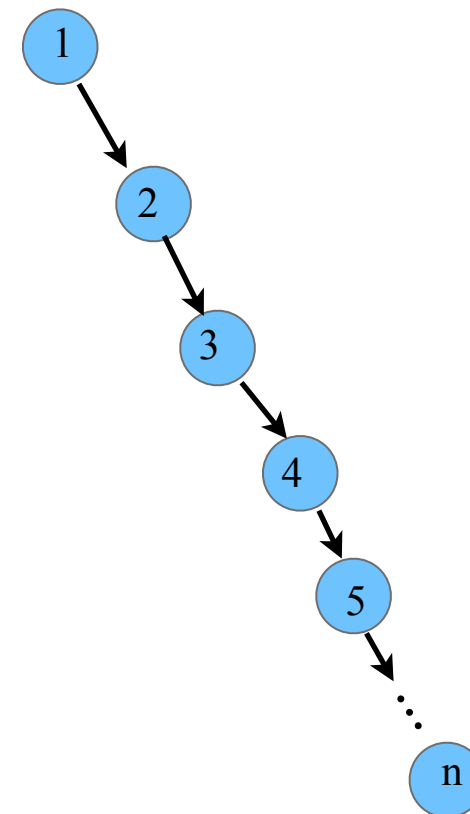
Finding an item in a binary search tree:
How long does find take in the worst case?
Best case?
Average Case?



Why do we care about the height of a binary search tree?



In a balanced tree it takes $O(\log(n))$ time to search for an item



In an unbalanced tree it takes $O(n)$ time to search for an item

Lecture 16 cont

Red-Black Trees (Building a Balanced Binary Search Tree)


So popular - it has its own youtube video!

<http://www.youtube.com/watch?v=vDHFF4wjWYU>


Red-Black trees are a type of binary search tree guaranteed to have logarithmic depth

Red-Black Tree

A **Red**-Black tree is a **binary search tree** that obeys 4 properties:

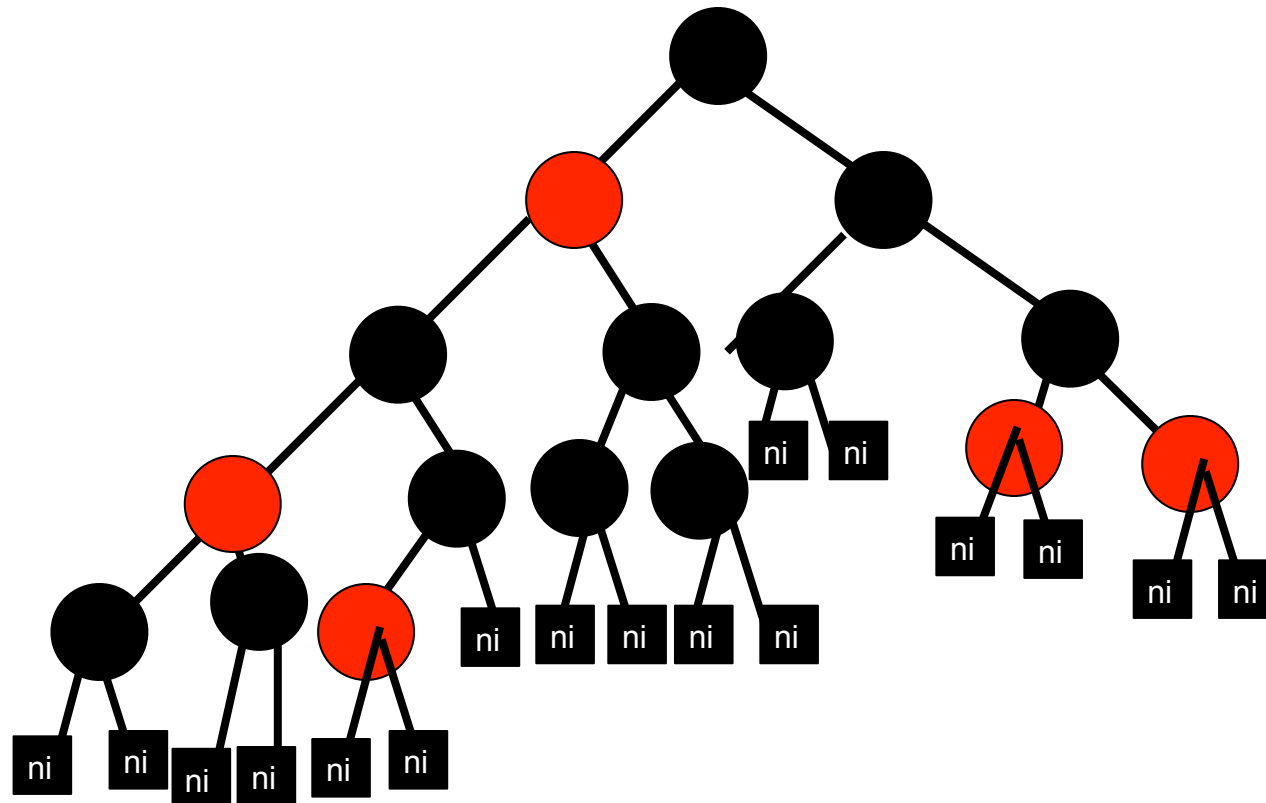
- 1) Every node is colored **red** or black
 - 2) All children of a **red node** are black
 - 3) [black property] For every node in the tree, all paths from that node down to a nullptr have the same number of black nodes along the path
 - 4) The root is black (This property is non - essential to maintain a balanced tree)
- 
- red-black
balancing
rules

NOTE: We adopt the convention that nullptrs are viewed as pseudo node and are black. i.e. we will call them nil nodes. They do not contain any data.

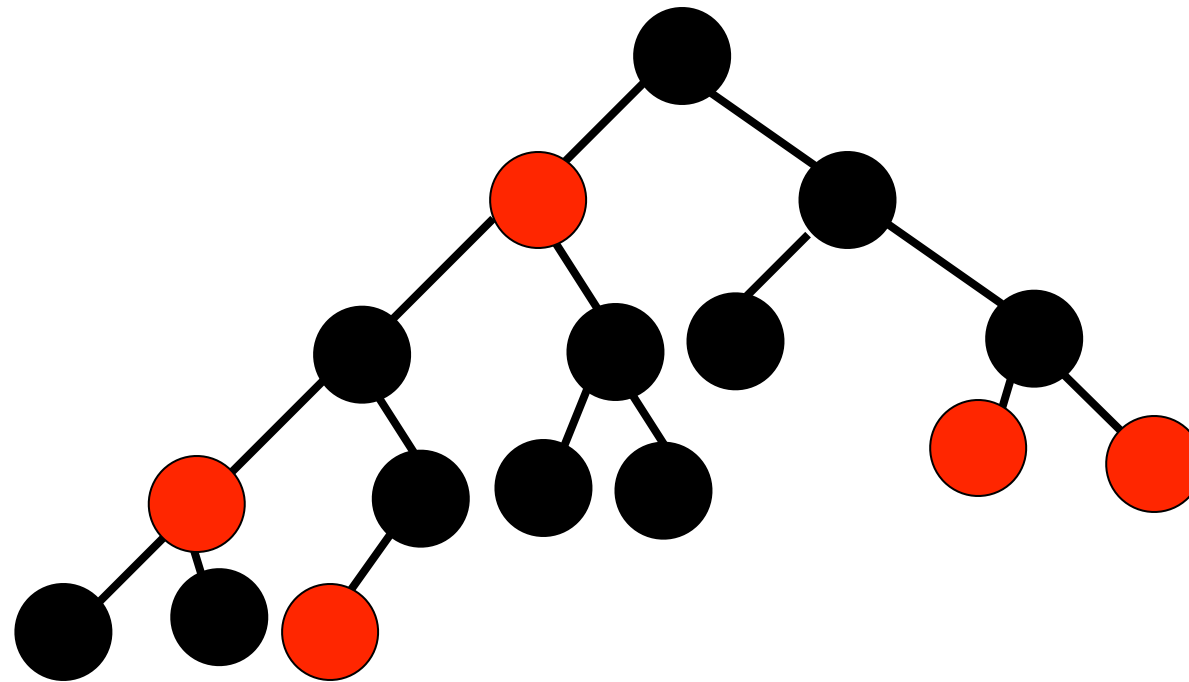


creates
a
cleaner
case
analysis

Red-Black Tree

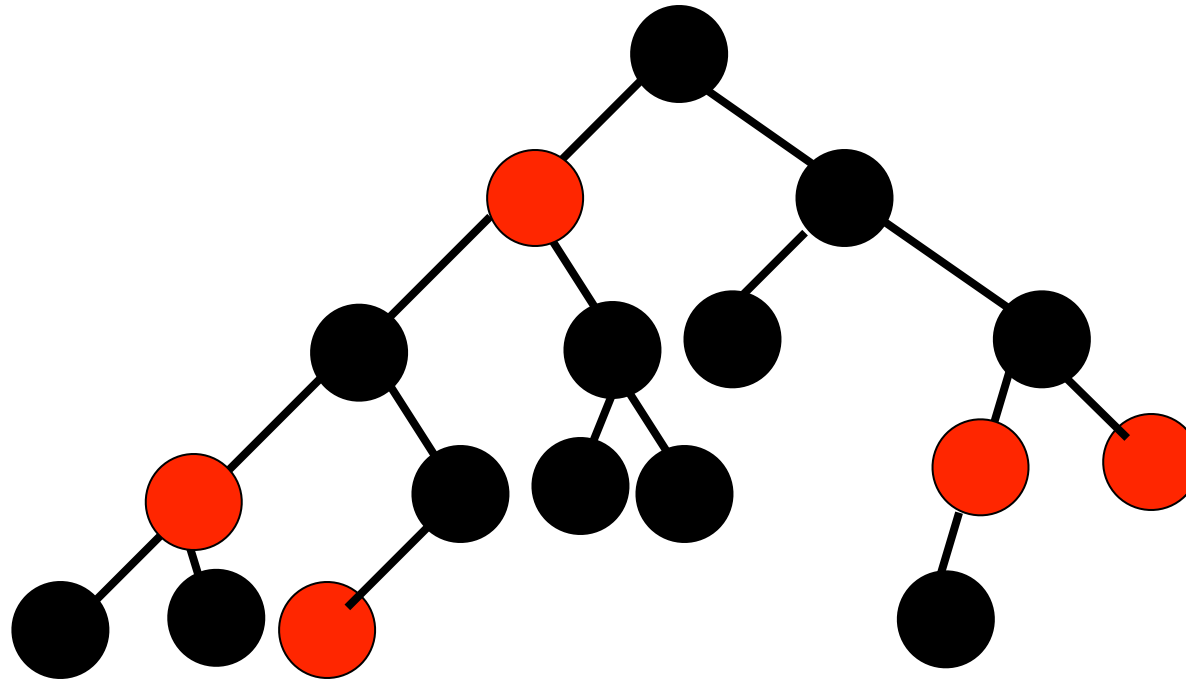


Red-Black Tree

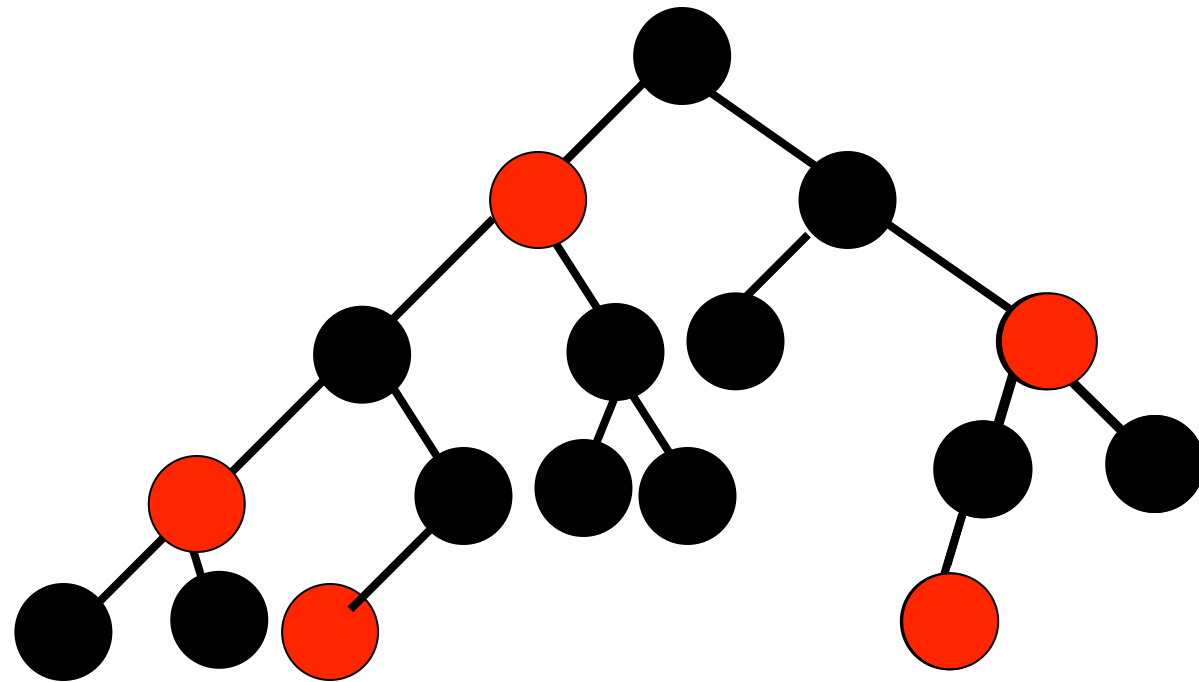


When drawing a tree, we will omit the nil nodes

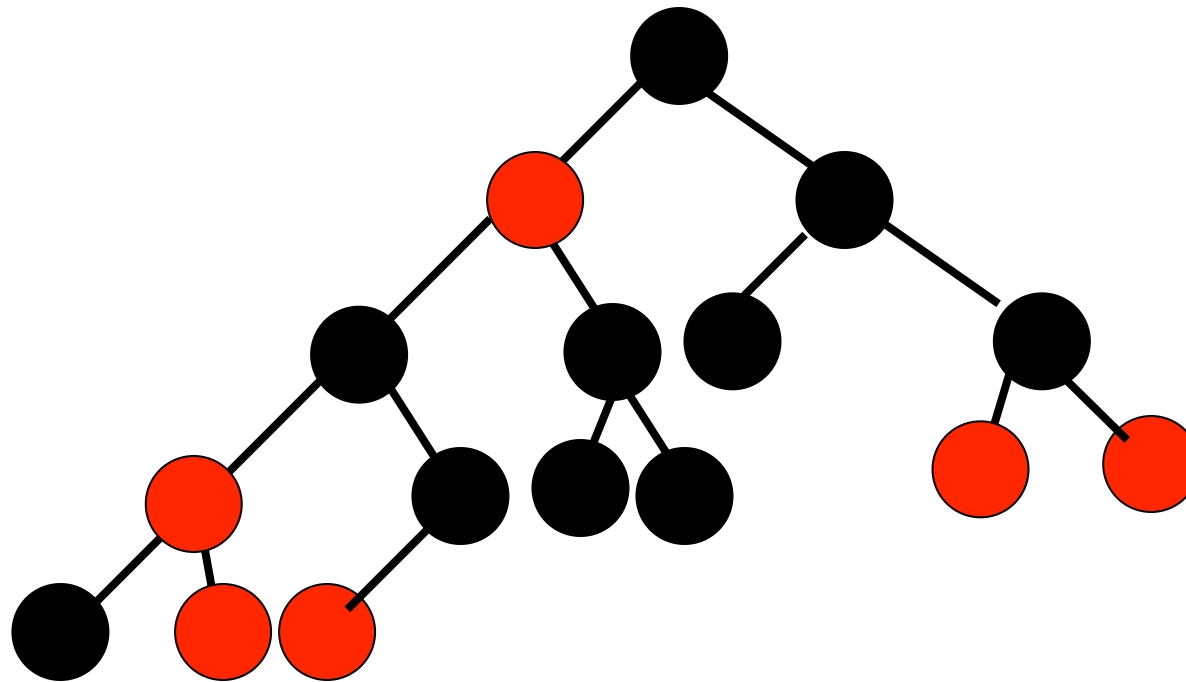
Not a Red-Black Tree



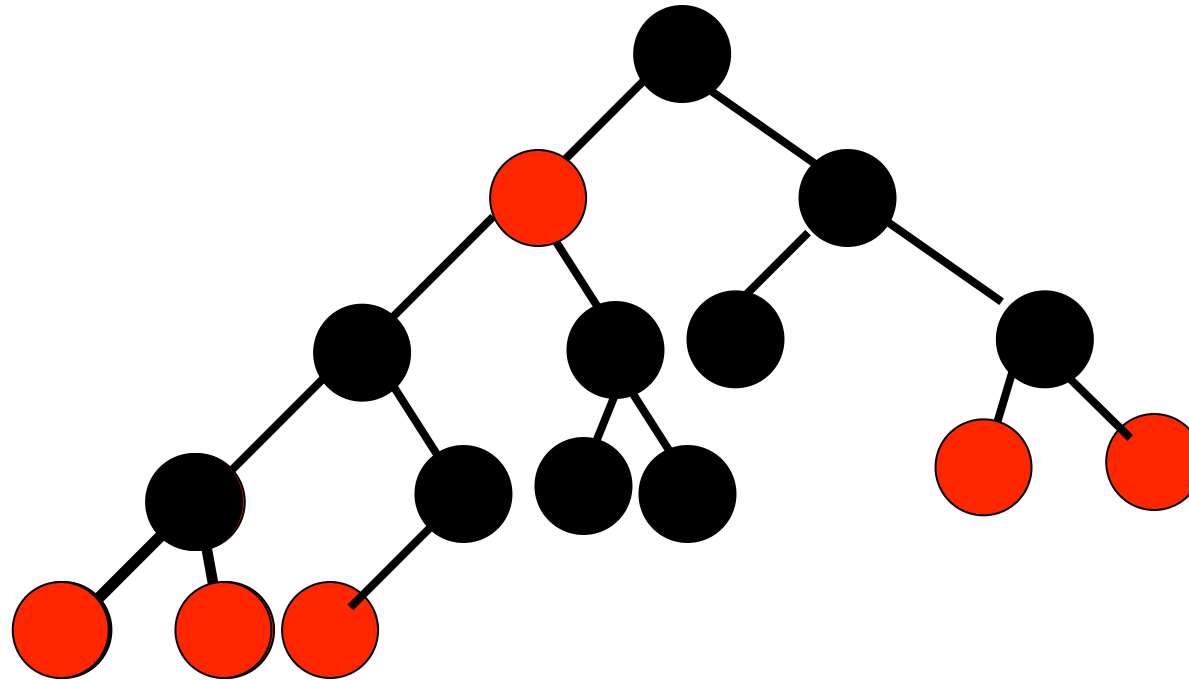
A Red-Black Tree



Not a Red-Black Tree



A Red-Black Tree



Black nodes in a Red-Black Tree

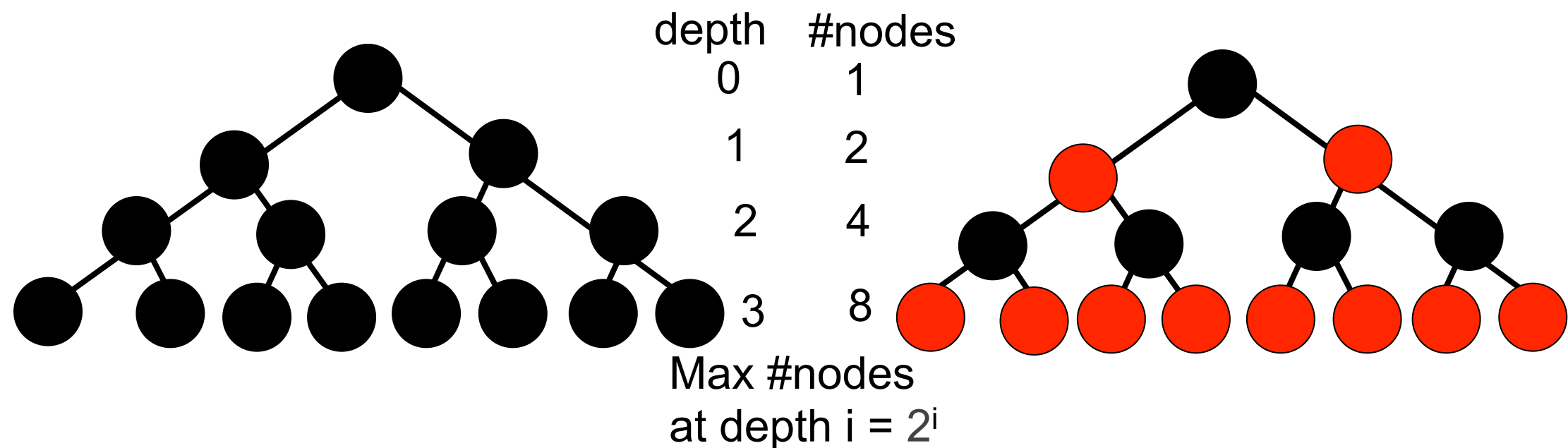
Let bh of a node be the number of black nodes on a path from the node to nil (including nil) but not including the node itself

We will prove on the next slide that any RedBlack Tree has at least $= 2^{bh}-1$ internal nodes.

Any RedBlack Tree has at most bh red nodes on a path from the root to a leaf.

Every red node has a black parent.

Any RedBlack Tree with Black height bh has at height at most $2bh$.



$$\#black\ nodes \quad 1 + 2 + 4 + 8 = 2^4 - 1$$

$$\#black\ nodes \quad 1 + 2 + 4 + 8 + \dots + 2^{bh-1} = 2^{bh} - 1$$

$$\#black\ nodes \quad 1 + 4 > 2^2 - 1$$

#Black nodes in a Red-Black Subtree

Any Red-Black tree with black height bh , has at least $2^{bh}-1$ internal (i.e. non nil) nodes.

The proof of this fact is based on induction.

Induction Hypothesis: Any subtree rooted at x of a Red-Black tree has 2^c-1 internal (i.e. non nil) nodes where c is the black height of the node x .

Base Case: If the height, is zero. Then x is a leaf and the black height is zero $2^c-1 = 2^0-1 = 0$

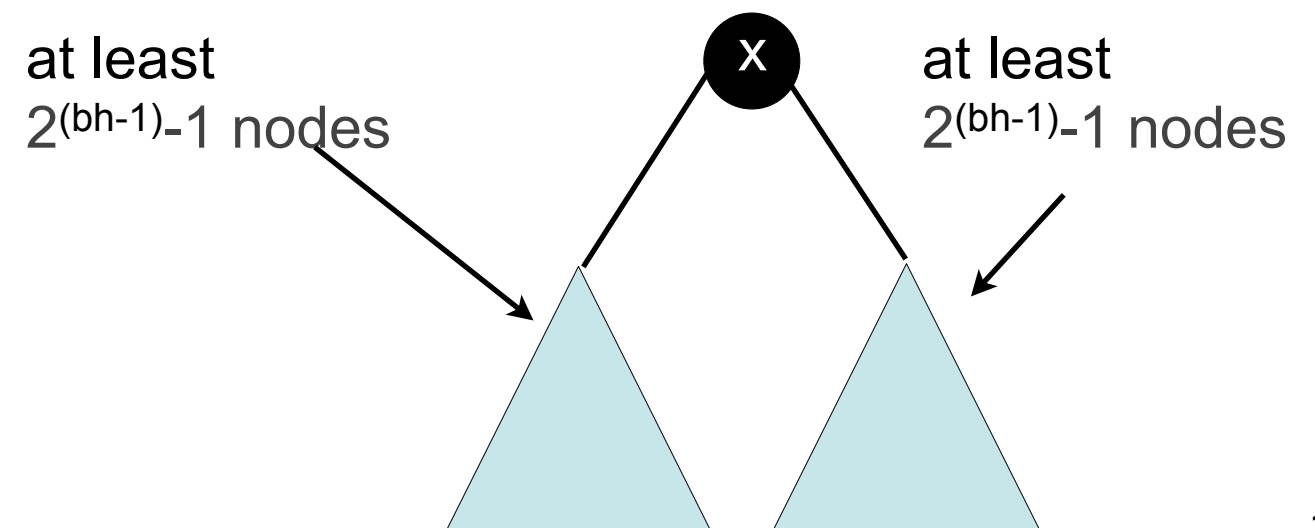
Induction Step:

We assume the result is true for any subtree of a Red-Black tree, whose height is less than the height of x .

Let bh be the black height of x .

Thus x 's left and right subtree have black height at least $(bh-1)$, and consequently each subtree has at least $2^{(bh-1)}-1$ internal nodes

Therefore the subtree rooted at x has at least $2^{(bh-1)}-1 + 2^{(bh-1)}-1 + 1 = 2^{bh} - 1$ internal nodes



Height and Size of **RB** Trees

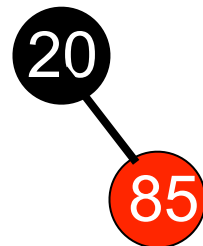
- If every path from the root down to nil has bh black nodes, then tree has at least $2^{bh}-1$ **internal** nodes (we showed this fact by induction)
 - $\Rightarrow N \geq 2^{bh}-1$ (since number nodes \geq number black nodes)
 - $\Rightarrow \log(N + 1) \geq bh$
 - $\Rightarrow \log(N + 1) \geq \frac{1}{2} \text{ height of tree}$ (since $bh \geq \frac{1}{2} \text{ height}$)
 - $\Rightarrow \text{Height of tree} \leq 2 \log (N+1) = O(\log N)$
- RED**-BLACK TREES HAVE $O(\log N)$ height

Insertion into a Red-Black Tree

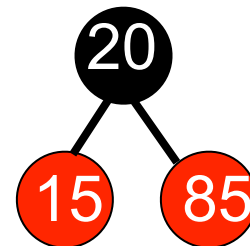
insert: 20



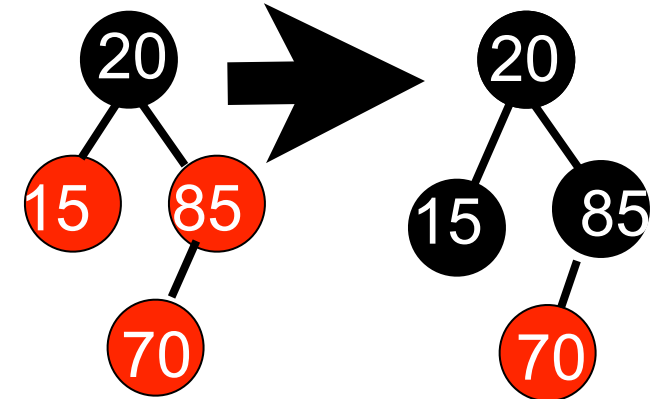
85



15



70

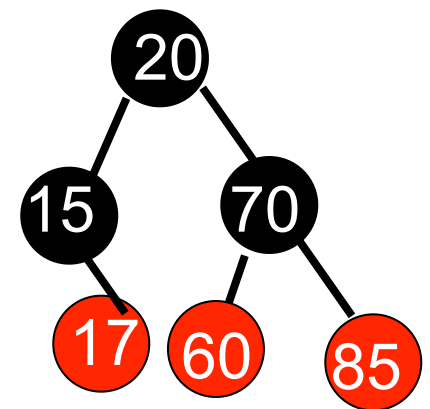
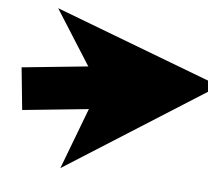
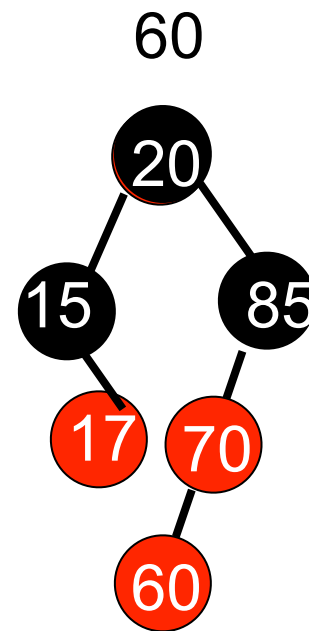
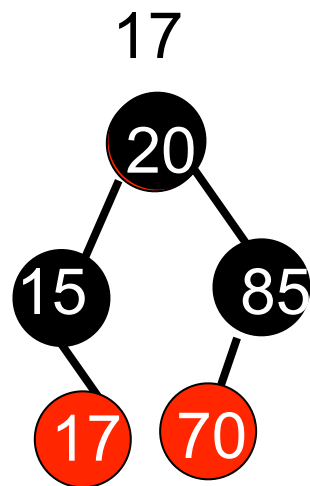
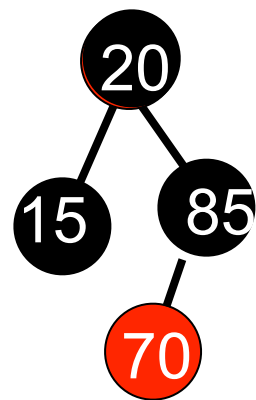


We will categorize the different types of red-red violations.
Case 1 will be distinguished by the parent node of the red-red violation having a red sibling

Double Red Problem
case 1 sibling of parent
is red

Insertion into a Red-Black Tree

insert:



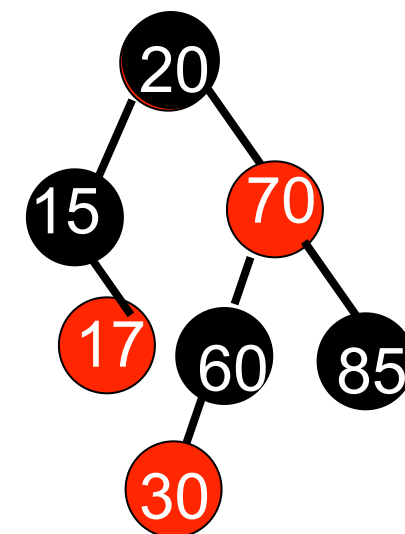
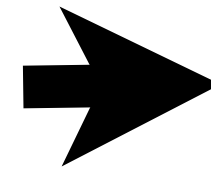
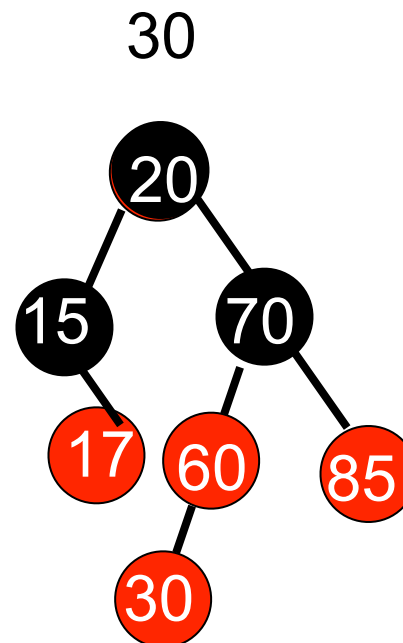
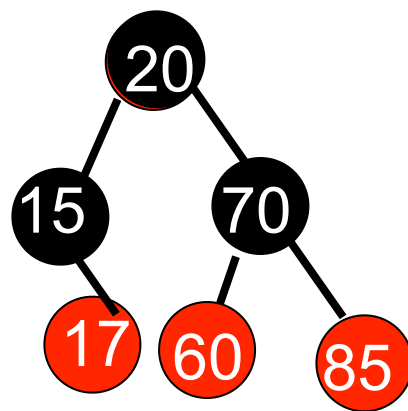
Double Red Problem

case 3 sibling of parent
is BLACK (or nil)

Right Rotate

Insertion into a Red-Black Tree

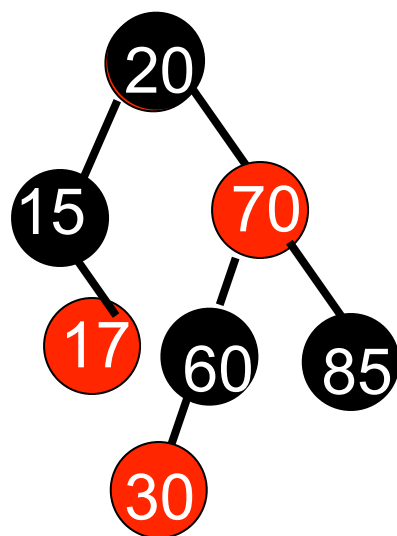
insert:



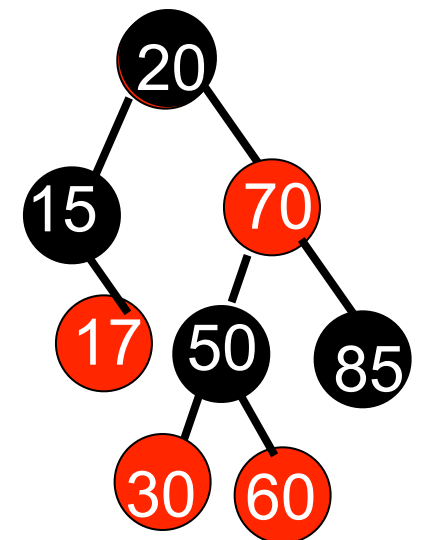
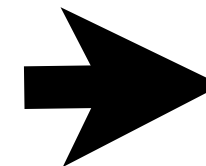
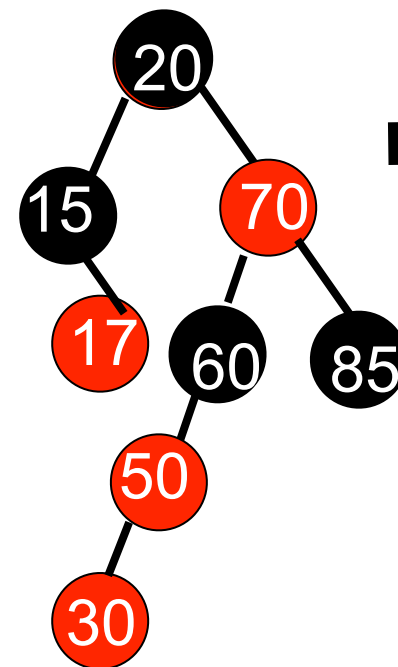
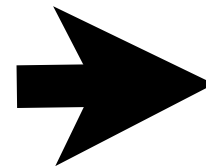
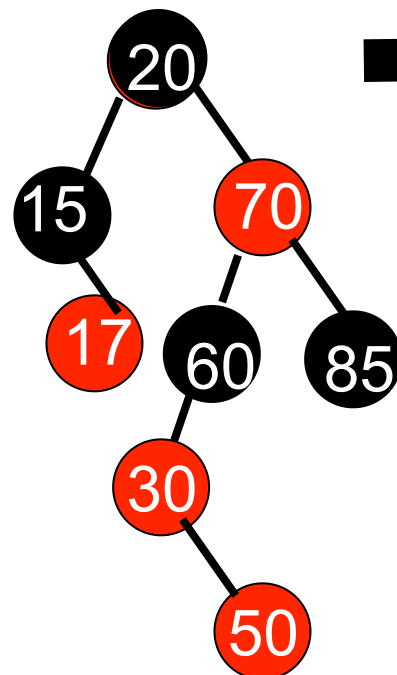
Double Red Problem
case 1 sibling of parent
is RED

Insertion into a Red-Black Tree

insert:



50



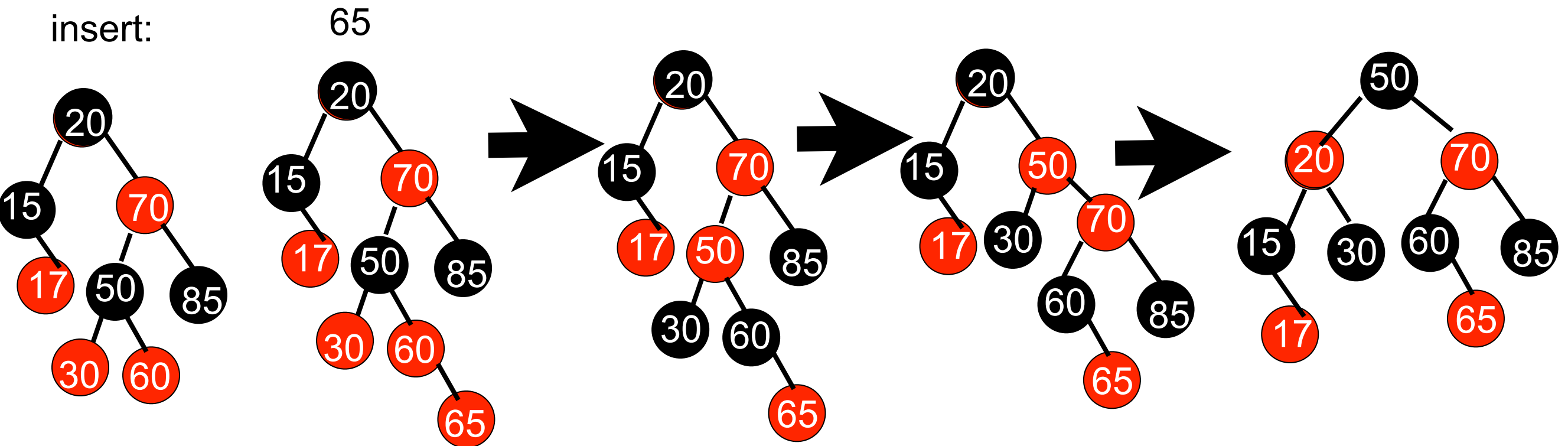
Double Red Problem
case 2 sibling of parent
is BLACK (or nil)
Left Rotate

Double Red Problem
case 3 sibling of parent
is BLACK (or nil)
Right Rotate

If a red-red violation occurs, we call it a case 2 or case 3 if the sibling of the parent is a black or nil node. Look for later slides to help you distinguish a case 2 from a case 3

Insertion into a Red-Black Tree

insert:

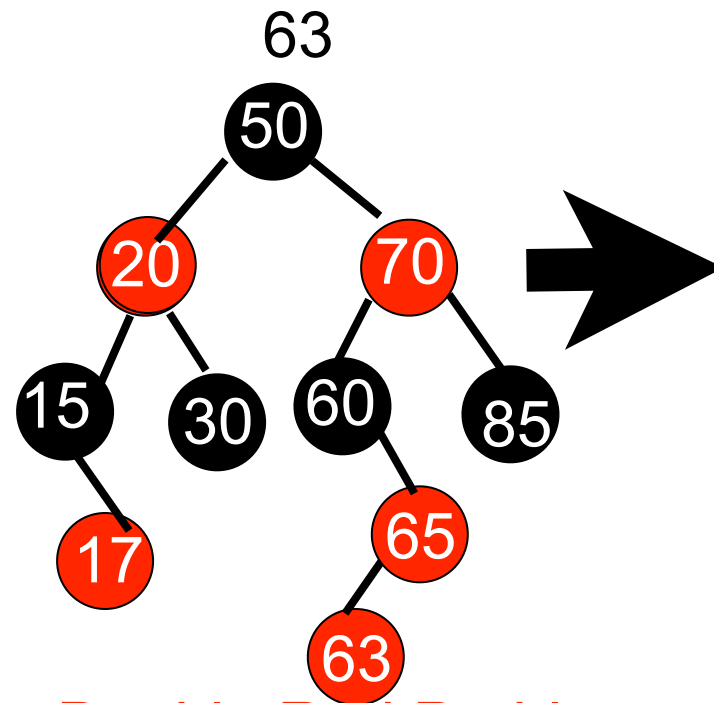
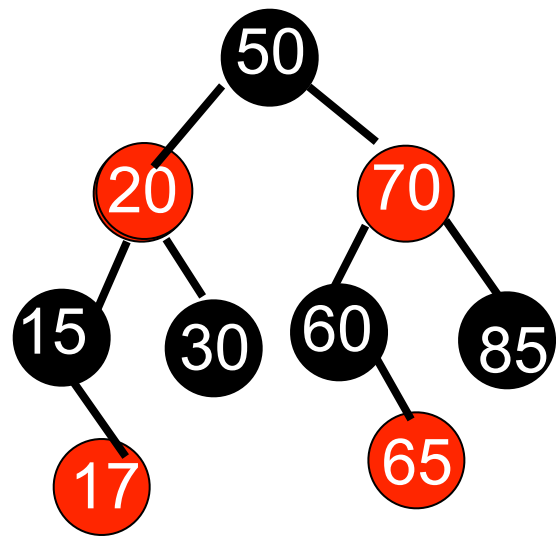


Double Red Problem
case 1 sibling of
parent is RED

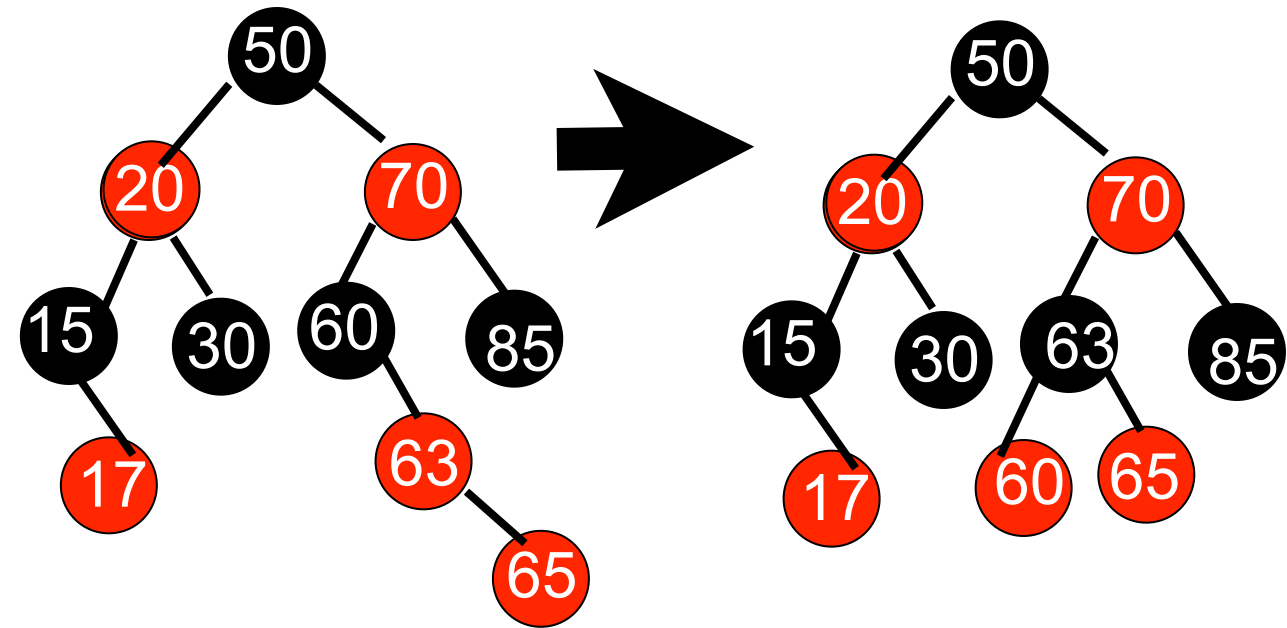
Double Red Problem
case 2 sibling of
parent is BLACK
or NULL

Double Red Problem
case 3 sibling of
parent is BLACK
or NULL

insert:



Double Red Problem
case 2 sibling of
 parent is BLACK
 or NULL



Double Red Problem
case 3 sibling of
 parent is BLACK
 or NULL

Bottom-Up Insertion into RB tree

- First insert as you would in standard binary search tree
- Color new node red
- If parent is black, done
- If parent is red, have red-red violation.
 - Fix by recoloring and (maybe) rotation.
 - May just move red-red violation further up the tree.
 - If so, repeat. Continue until
 - no red-red violation
 - red-red violation is root and one of its children. Color root black.
- Bottom-up insertion is only partially covered in our textbook

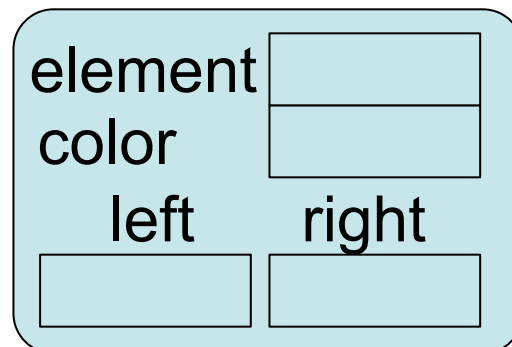
A Node of the Red-Black Tree

```
template <class Comparable>
class RedBlackNode
{
```

```
    Comparable    element;
    RedBlackNode *left;
    RedBlackNode *right;
    int            color;
```

```
    RedBlackNode( const Comparable & theElement = Comparable( ),
                  RedBlackNode *lt = nullptr, RedBlackNode *rt = nullptr,
                  int c = RedBlackTree<Comparable>::BLACK )
: element( theElement ), left( lt ), right( rt ), color( c ) { }
    friend class RedBlackTree<Comparable>;
};
```

```
template< class
Comparable>
class RedBlackTree
{
    enum { RED, BLACK };
};
```



For simplicity we will represent a node by:

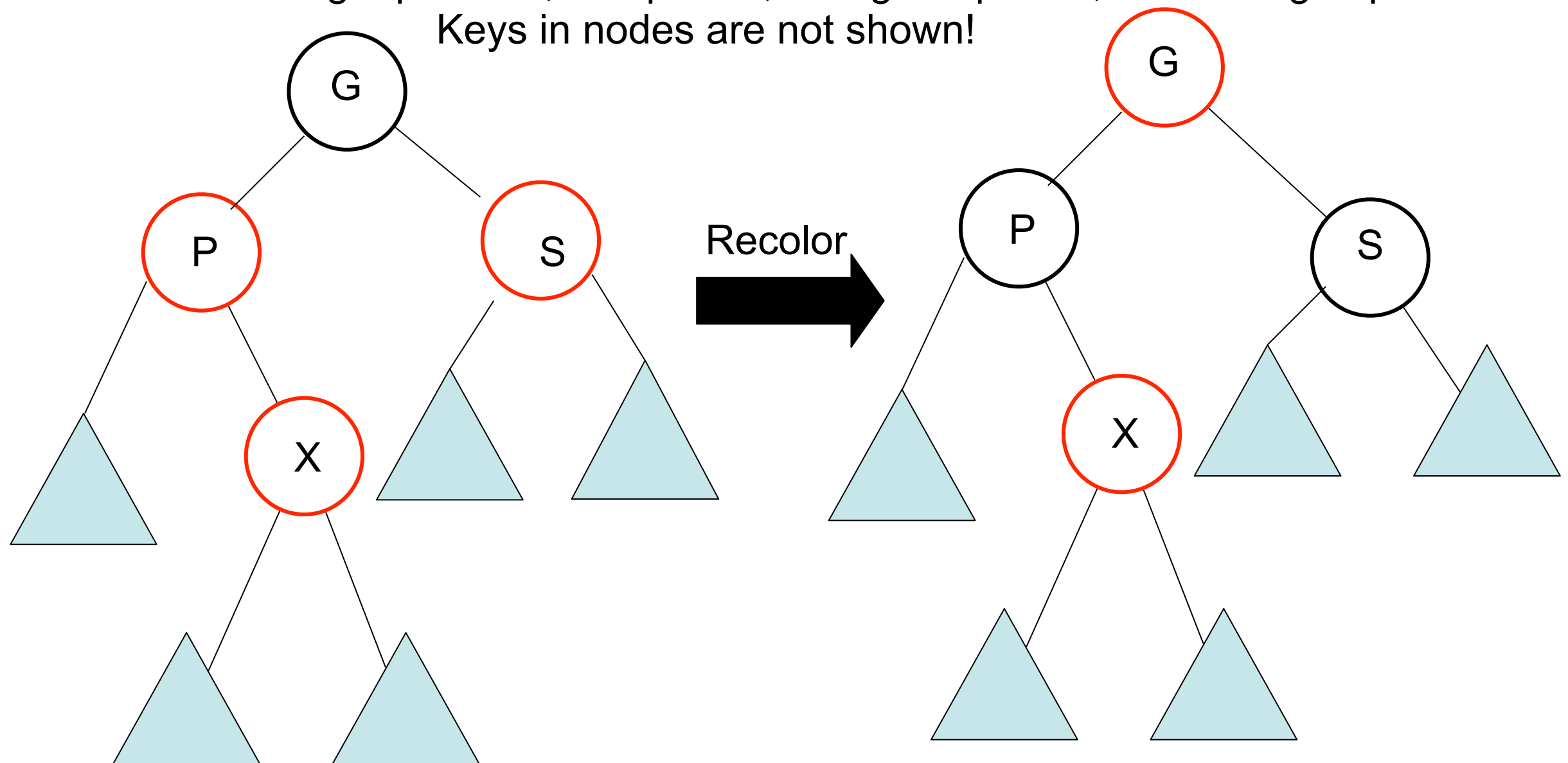


Fixing a red-red violation

- Call inserted node **X**
- May need to manipulate the following nodes:
 - P: Parent of **X**
 - G: Grandparent of **X**
 - S: Sibling of parent of **X** (uncle)
- 3 Cases –
 - Case 1 is easy (but may move problem up tree),
 - Case 3 requires one rotation
 - Case 2 you do one rotation and get to Case 3 (for a total of 2 rotations)
- When applying the following, if S is nil, treat it as a black node

Case 1: Sibling of Parent is Red

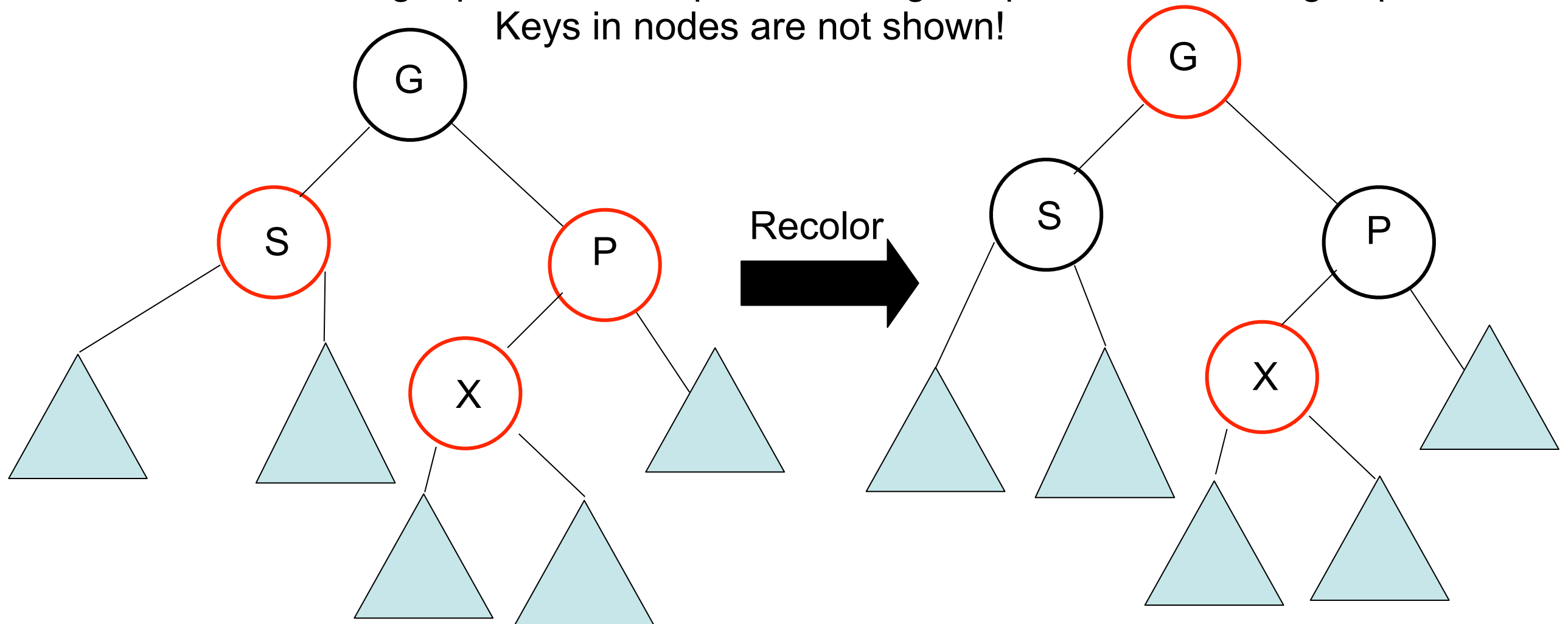
X is node causing a problem, P is parent, G is grandparent, S is sibling of parent
Keys in nodes are not shown!



After recolor: If G is root, color it black.
If parent of G is **red**, continue repair
with G being the new **X**.

Another Example of Case 1: Sibling of Parent is Red

X is node causing a problem, P is parent, G is grandparent, S is sibling of parent
Keys in nodes are not shown!



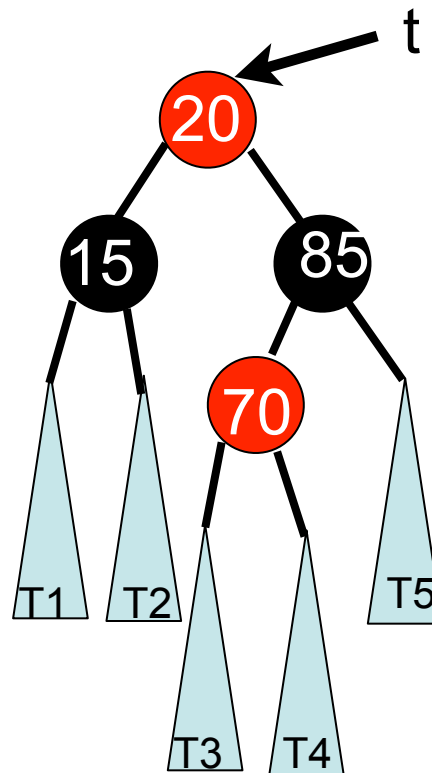
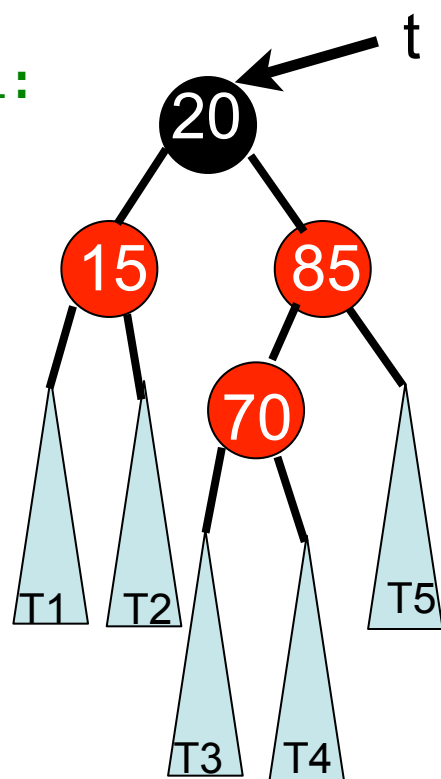
After recolor: If G is root, color it black.
If parent of G is red, continue repair
with G being the new X.

Sibling of parent is red

// Flips node and children's color

```
template <class Comparable>
void RedBlackTree<Comparable>::colorFlip( Node * t )
{
    t->color = RED;
    t->left->color = BLACK;
    t->right->color = BLACK;
}
```

Case 1:

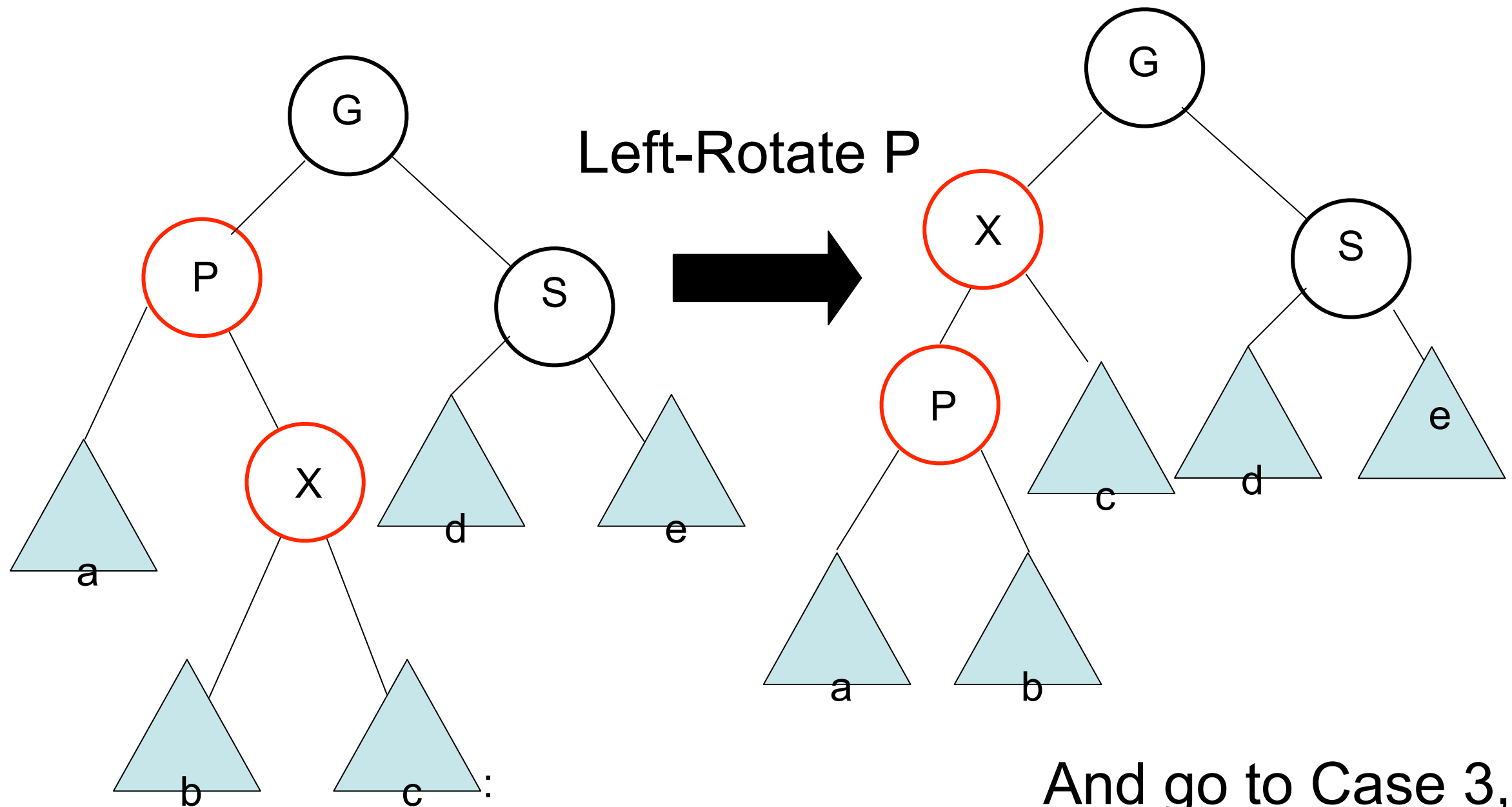


```
template< class
Comparable>
class RedBlackTree
{
    enum { RED, BLACK };
};
```

```
template <class Comparable>
class RedBlackNode
{
    Comparable    element;
    RedBlackNode *left;
    RedBlackNode *right;
    int           color;

    // constructors removed
};
```

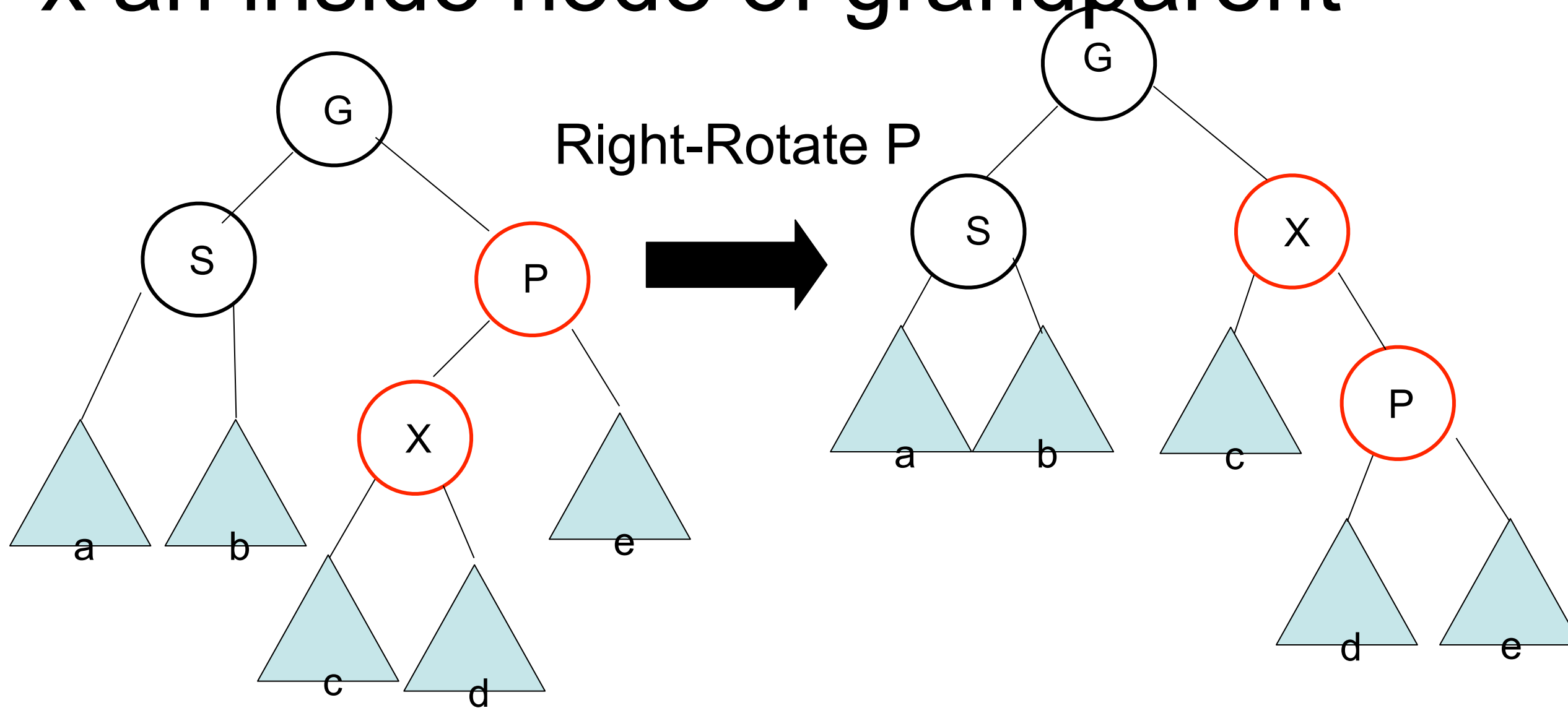
Case 2: Sibling of Parent is Black, **X** an inside node of grandparent



And go to Case 3,
with P as new **X**

Here we left-rotate P. Right rotate in Case 2 if P, **X** are in right subtree of G.

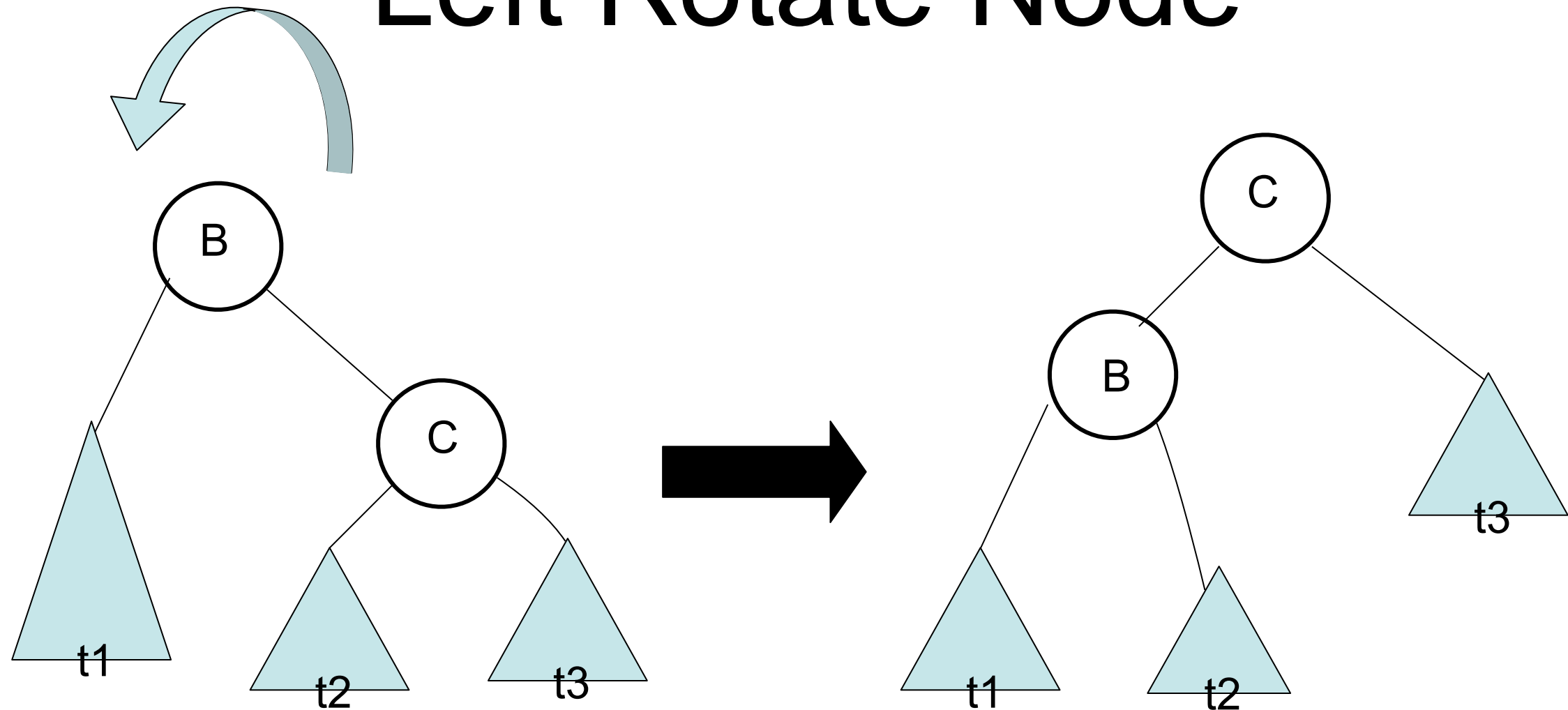
The Other Case 2: Sibling of Parent is Black, x an inside node of grandparent



And go to Case 3,
with P as new X

Here we right-rotate P.

Left Rotate Node



In text book, this is called “rotate node with right child”

// Rotate binary tree node with right child

```
template <class Comparable>
```

```
void RedBlackTree<Comparable>::leftRotate( Node * & k1 ) const
```

```
{
```

```
    Node *k2 = k1->right;
```

```
    k1->right = k2->left;
```

```
    k2->left = k1;
```

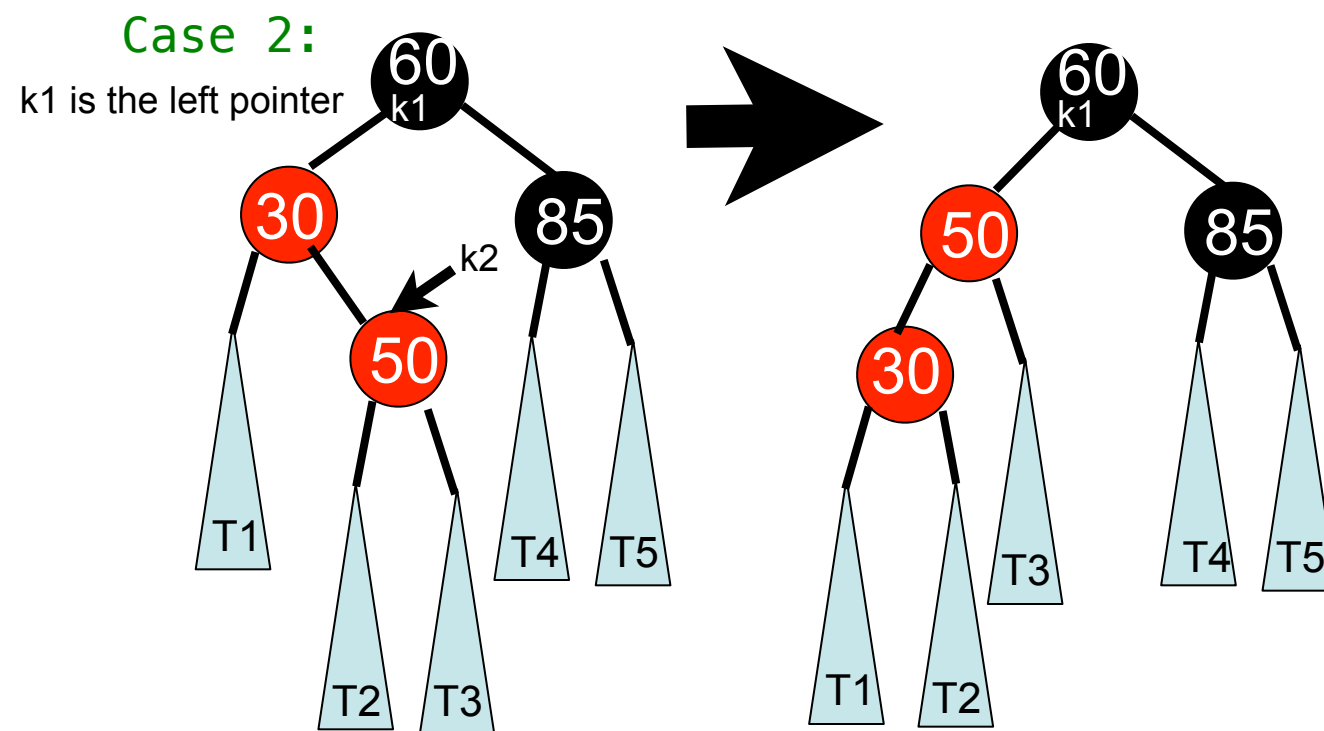
```
    k1 = k2;
```

```
}
```

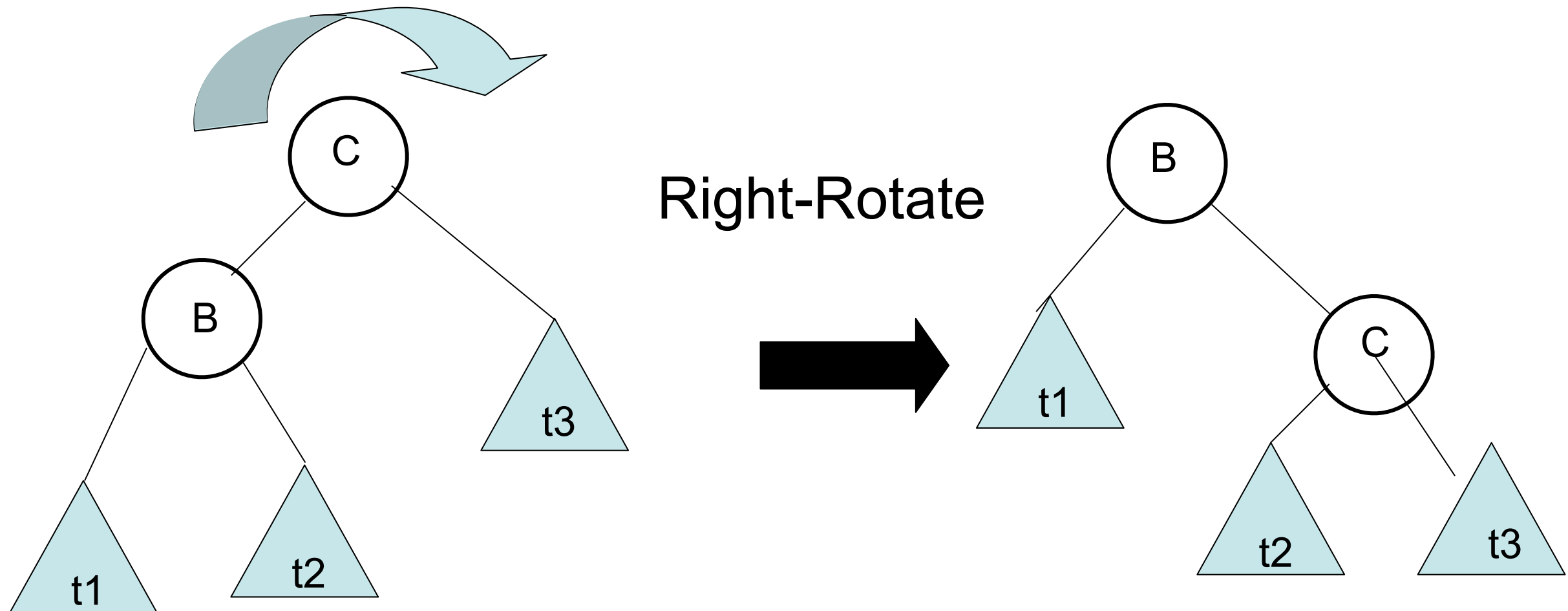
```
template< class
Comparable>
class RedBlackTree
{
    enum { RED, BLACK };
};
```

```
template <class Comparable>
class RedBlackNode
{
    Comparable    element;
    RedBlackNode *left;
    RedBlackNode *right;
    int           color;

    // constructors removed
};
```



Right Rotate Node



In text book, this is called “rotate node with left child”

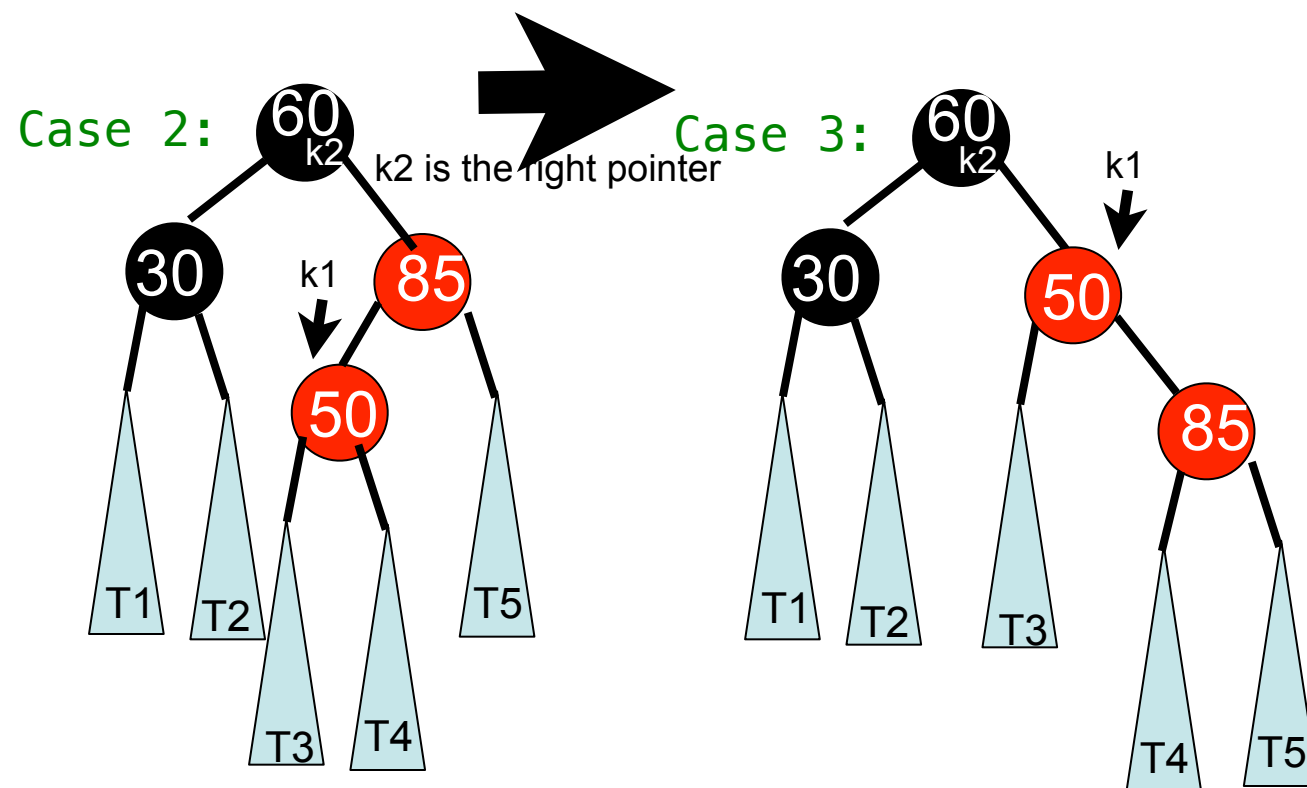
Rotate with Left Child

// Rotate binary tree node with left child.

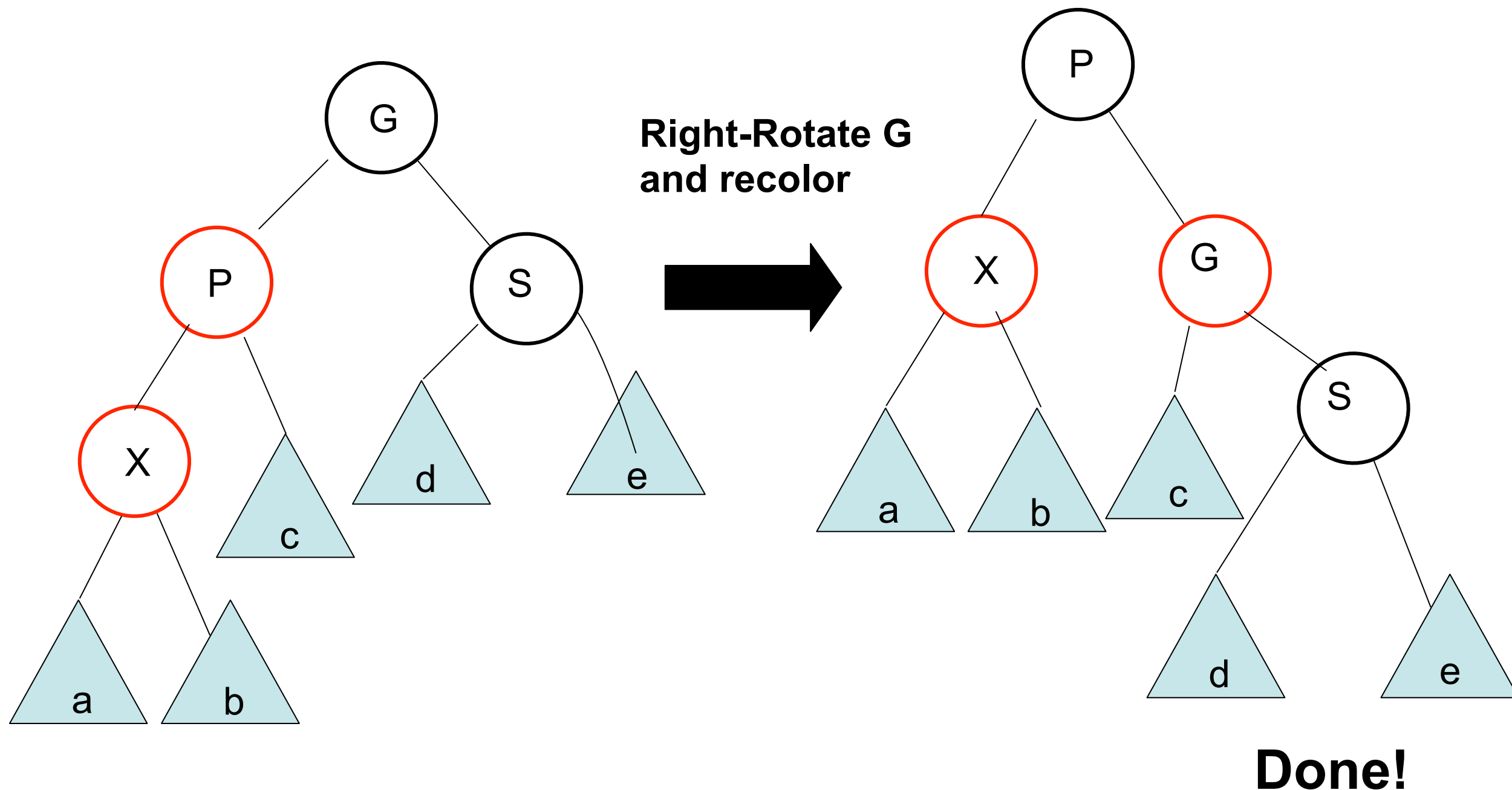
```
template <class Comparable>
```

```
void RedBlackTree<Comparable>::rightRotate( Node * & k2 ) const  
{
```

```
}
```

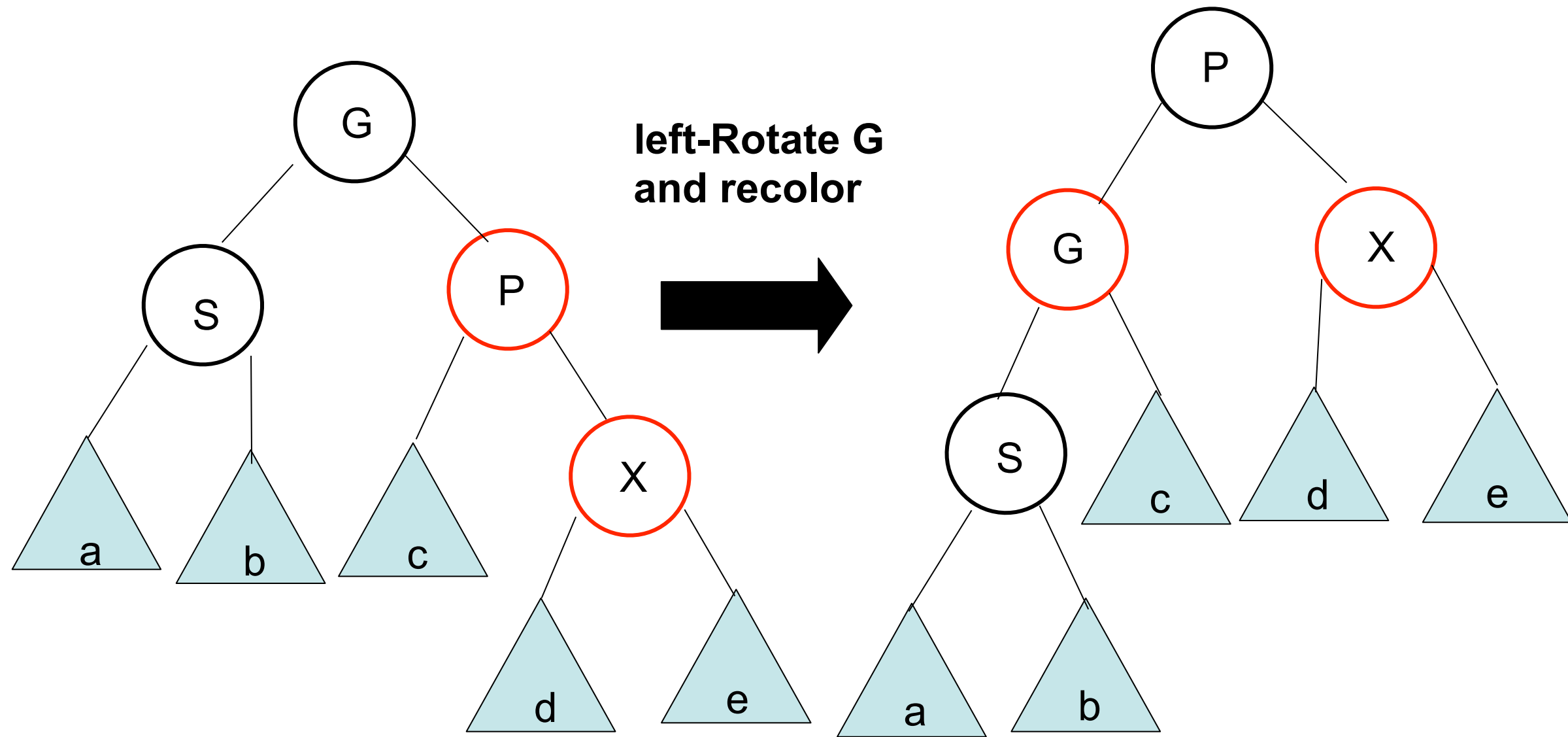


Case 3: Sibling of Parent is Black, x an outside node of grandparent



Here we right-rotate G. Left-rotate in Case 3 if P,X are in right subtree of G.

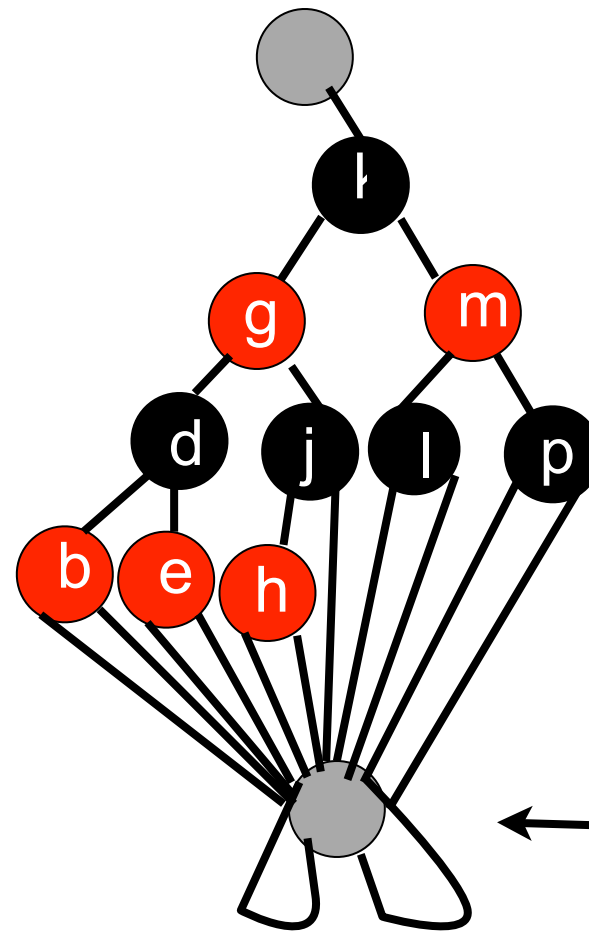
The OtherCase 3: Sibling of Parent is Black, x an outside node of grandparent



Done!

Here we left-rotate G.

Trick to Simplify the Coding of the **Red**-Black Tree



Coding trick:
nullNode points
to itself

← actually colored BLACK

Insert (very basic)

```
template <class Comparable>
void RedBlackTree<Comparable>::insert( const Comparable & x, Node * & t )
{
```

```
    if( t == nullNode )
    {
        t = new Node( x, nullNode, nullNode, RED );
        return;
    }
```

```
    if( x < t->element)
        insert( x, t->left);
    else if ( x > t->element )
        insert(x, t->right);
    else if ( t->element == x )
        throw DuplicateItemException( );
```

// Case 1: t is grand parent

```
if( t->left->color == RED && t->right->color == RED ){
    colorFlip(t); // might do unnecessary change....
    return;
}
```

// Case 2: t is grand parent

```
if ( t->left->color == RED && t->left->right->color == RED )
    leftRotate( t->left );
if ( t->right->color == RED && t->right->left->color == RED )
    rightRotate( t->right );
```

// Case 3: t is grand parent

```
if ( t->left->color == RED && t->left->left->color == RED )
    rightRotateRecolor( t );
if ( t->right->color == RED && t->right->right->color == RED )
    leftRotateRecolor( t );
}
```

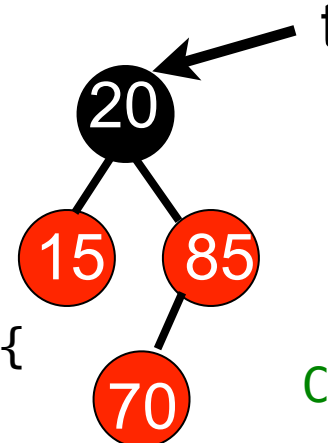
The driver for
the insert method
ensures the root
node is black

Check out these other implementations of
red-black trees:

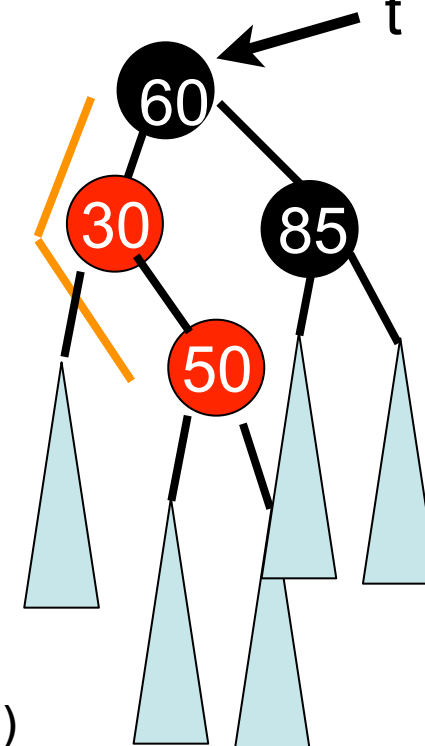
<http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf>

http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx

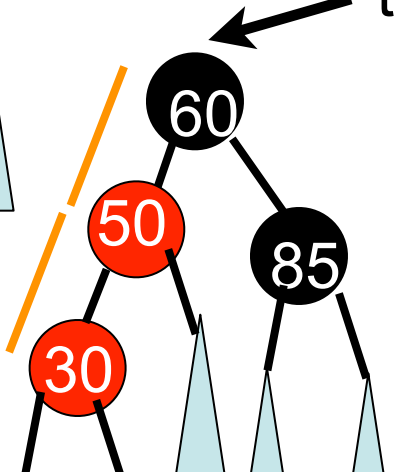
Case 1:



Case 2:



Case 3:



Maps

- Objects in a set can be inserted or deleted, but keys cannot be modified, since modification may destroy sorted order
- Maps associate unique key with data
- Can insert/delete (key,data) pairs
- Can modify data in an object that's in the map
- Cannot modify key of object in the map

Pair

```
class pair
{
public:
    Type1 first;
    Type2 second;
    pair (const Type1 & f=Type1(), const Type2 & s = Type2())
        : first(f), second( s){}
};
```

```
pair<int, int> mypair;
mypair.first = 3;
mypair.second= 9;
cout << mypair.first<< mypair.second;
```

Maps

- Objects in a set can be inserted or deleted, but keys cannot be modified, since modification may destroy sorted order
- Maps associate unique key with data
- Can insert/delete (key,data) pairs
- Can modify data in an object that's in the map
- Cannot modify key of object in the map

Map - Bidirectional Iterator

- `m.insert(pair)` $O(\log(n))$
- `m.find(key)` $O(\log(n))$
- `m.size()` $O(1)$
- `m.begin()` $O(1)$
- `m.end()` $O(1)$
- `m.lower_bound(key)` $O(\log(n))$
- `m.upper_bound(key)` $O(\log(n))$
- `m[key]` $O(\log(n))$
- `m.clear()` $O(n)$
- `m.erase(key) & m.erase(iterator)` $O(\log(n))$
 $O(1)$ amortized


```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main ( ) {
//code directly/adapted from:  Essential C++ by Stanley B. Lippman
    map<string,int> words;

    words[ "vermeer" ] = 1;

    string tword;
    while (cin >> tword)
    {
        words[tword]++;
    }

    cout <<  words[ "rembrant"]; // now rembrant is in
        the map with default value (i.e. 0)
```

```

map<string, int>::iterator it = words.begin();
for( ; it != words.end(); ++it)
    cout << "key: " << it->first << "value: "
        << it->second << endl;

int count = 0;
if (!(count == words[ "vermeer" ] ))
    //vermeer not present
    // but not just testing a word adds it to the map....

it = words.find("vermeer");
if (it != words.end())
    count = it->second;

string search_word( "vermeer");
if(words.count(search_word)) // true if word is present in the map
    count = words[search.word];

return 0;
}

```

```
map<int,int> mapOfIntInt;
map<int,int>::iterator itrM;
cout<<mapOfIntInt.size() << "is the initial size.\n";
for (int i=0; i<10; i++)
    mapOfIntInt.insert(pair<int,int>(i,rand()%10));

cout<< mapOfIntInt.size();
cout <<"The key 3, has value " << mapOfIntInt[3]<<endl;

for (itrM=mapOfIntInt.begin(); itrM!=mapOfIntInt.end(); ++itrM)
    cout << "(" << itrM->first << ", " << itrM->second << ")" << " ";
cout << endl;

mapOfIntInt.erase(1);

mapOfIntInt.erase(12);
```

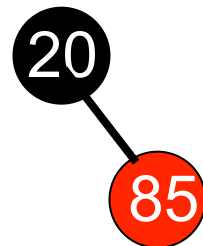
data structure	build	insert	find
vector	$O(n)$	$O(1)$ _{amortized}	$O(n)$
sorted vector	$O(n \log n)$	$O(n)$	$O(\log n)$
set or map	$O(n \log n)$	$O(\log n)$	$O(\log n)$
list	$O(n)$	$O(1)$	$O(n)$
sorted list	$O(n \log n)$	$O(n)$	$O(n)$

Insertion into a Red-Black Tree

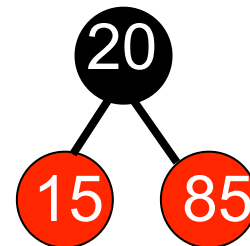
insert: 20



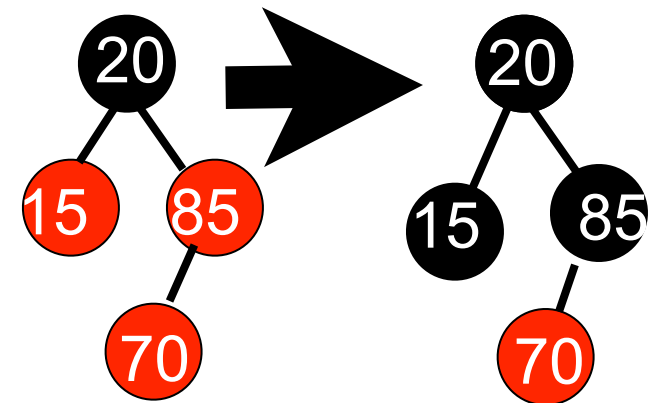
85



15



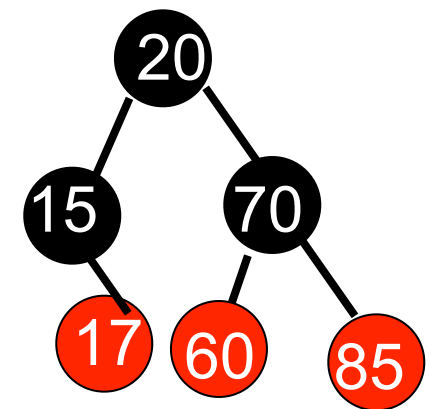
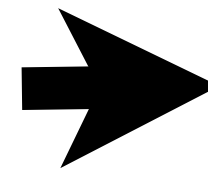
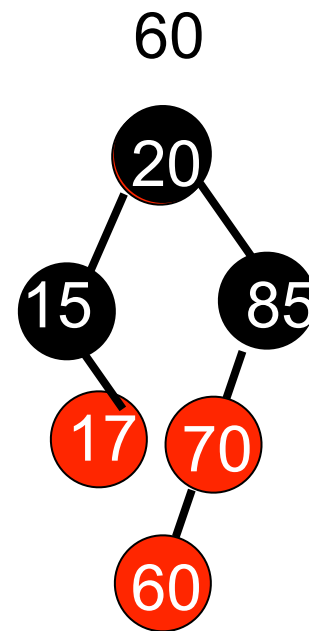
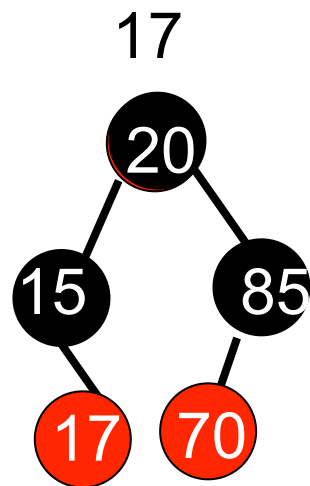
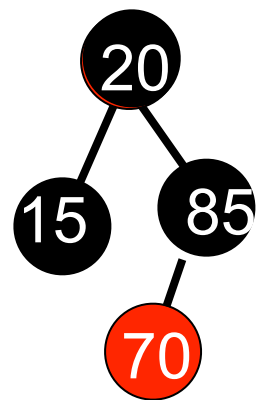
70



Double Red Problem
case 1 sibling of parent
is red

Insertion into a Red-Black Tree

insert:



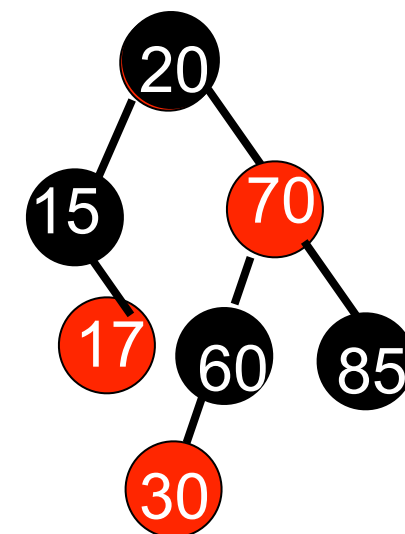
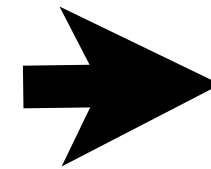
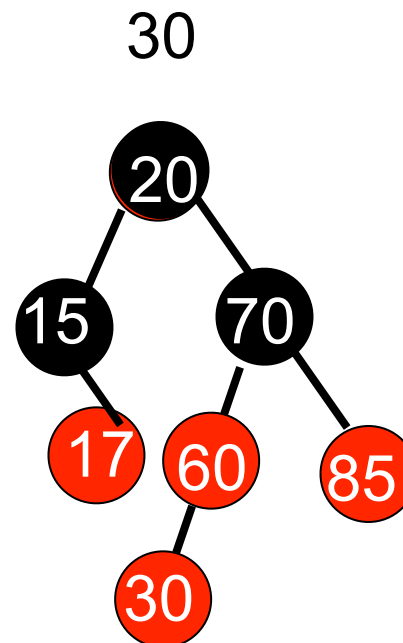
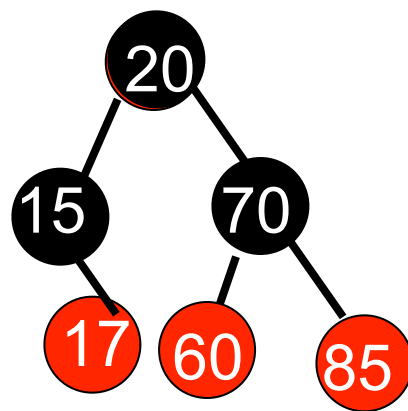
Double Red Problem

case 3 sibling of parent
is BLACK or nil

Right Rotate

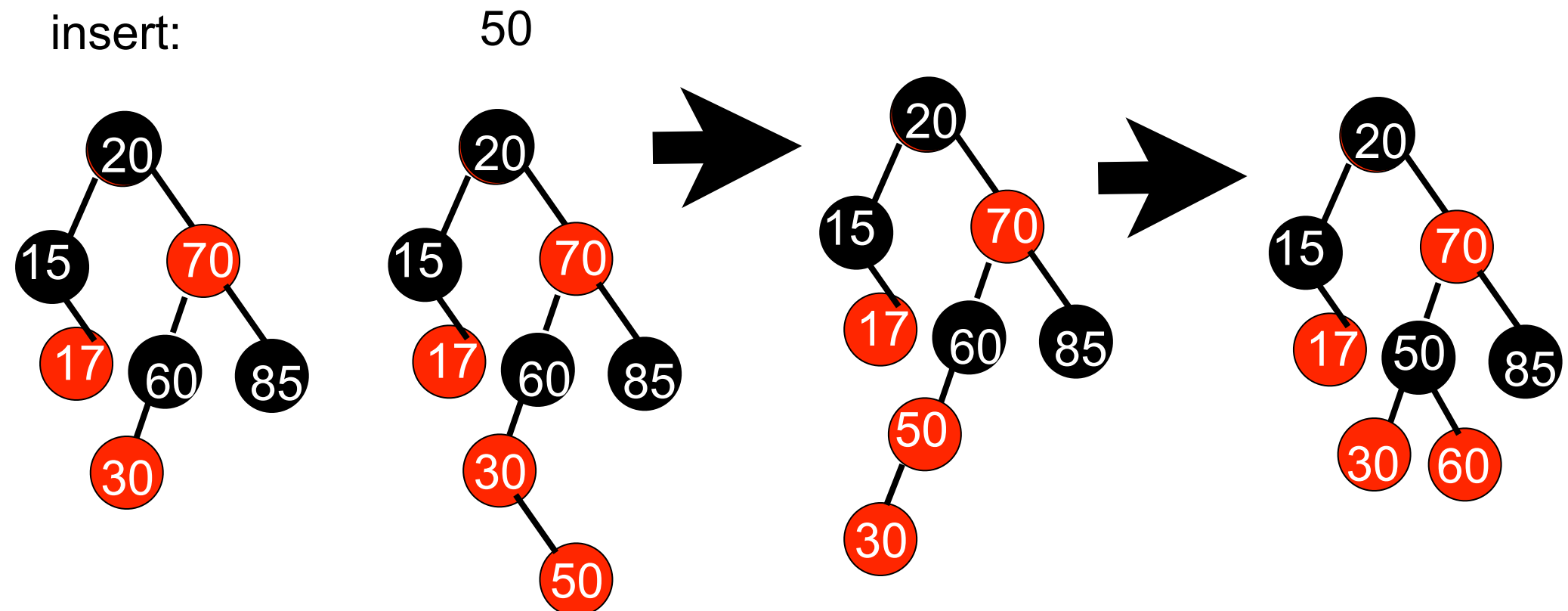
Insertion into a Red-Black Tree

insert:



Double Red Problem
case 1 sibling of parent
is RED

Insertion into a Red-Black Tree

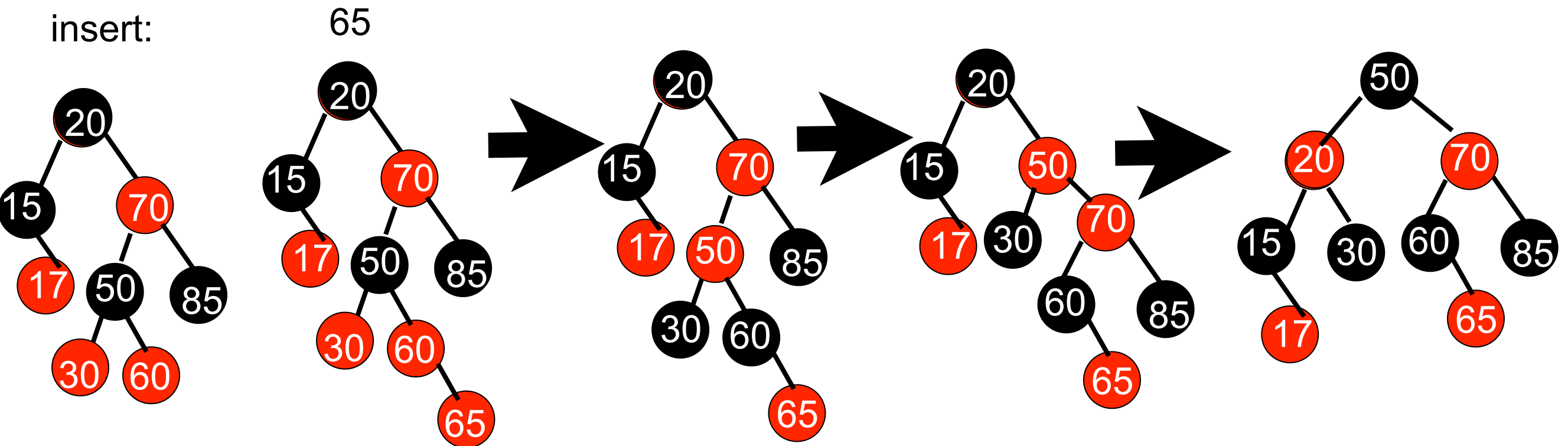


Double Red Problem
case 2 sibling of parent
is BLACK or nil

Double Red Problem
case 3 sibling of parent
is BLACK or nil
Right Rotate

Insertion into a Red-Black Tree

insert:

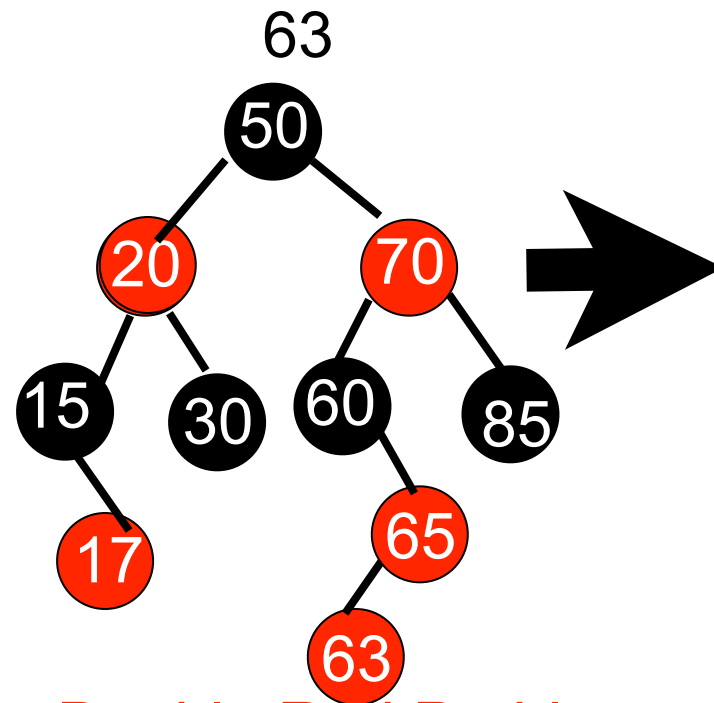
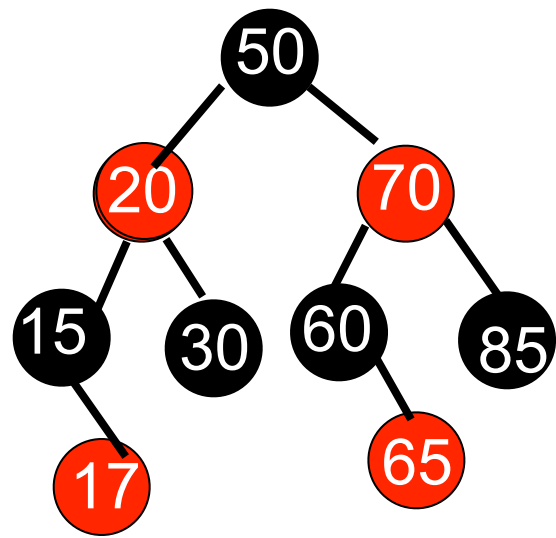


Double Red Problem
case 1 sibling of
parent is RED

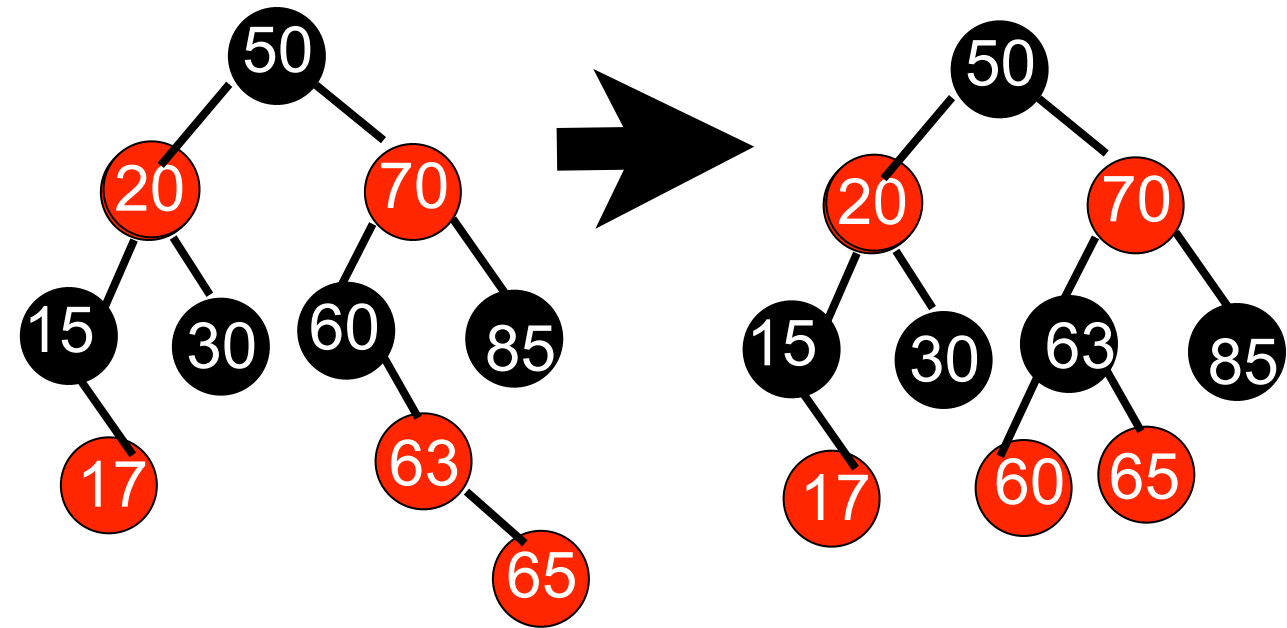
Double Red Problem
case 2 sibling of
parent is BLACK
or nil

Double Red Problem
case 3 sibling of
parent is BLACK
or nil

insert:

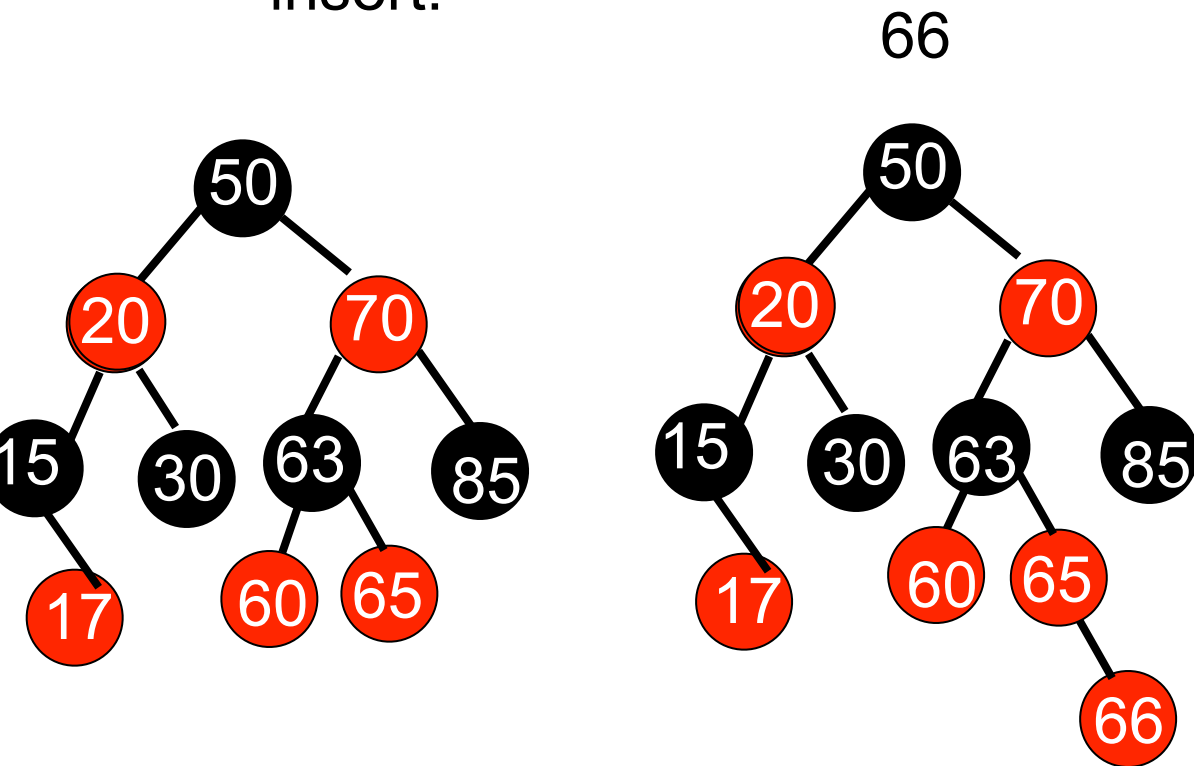


Double Red Problem
case 2 sibling of
 parent is BLACK
 or nil



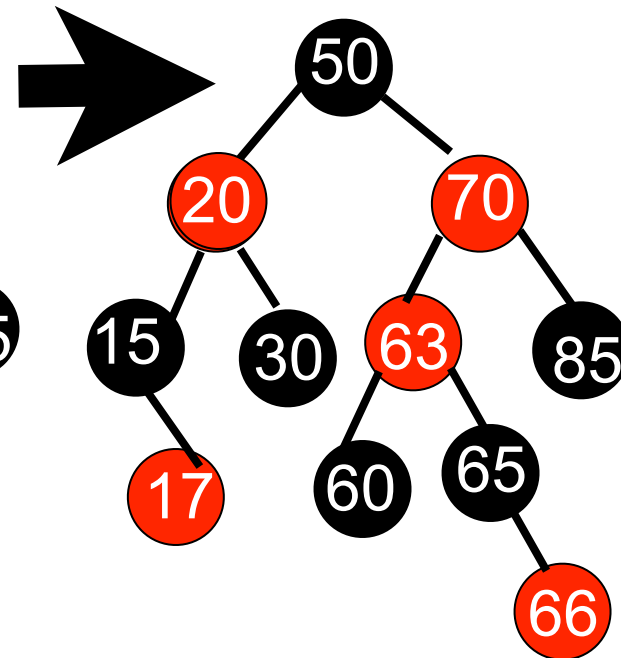
Double Red Problem
case 3 sibling of
 parent is BLACK
 or nil

insert:



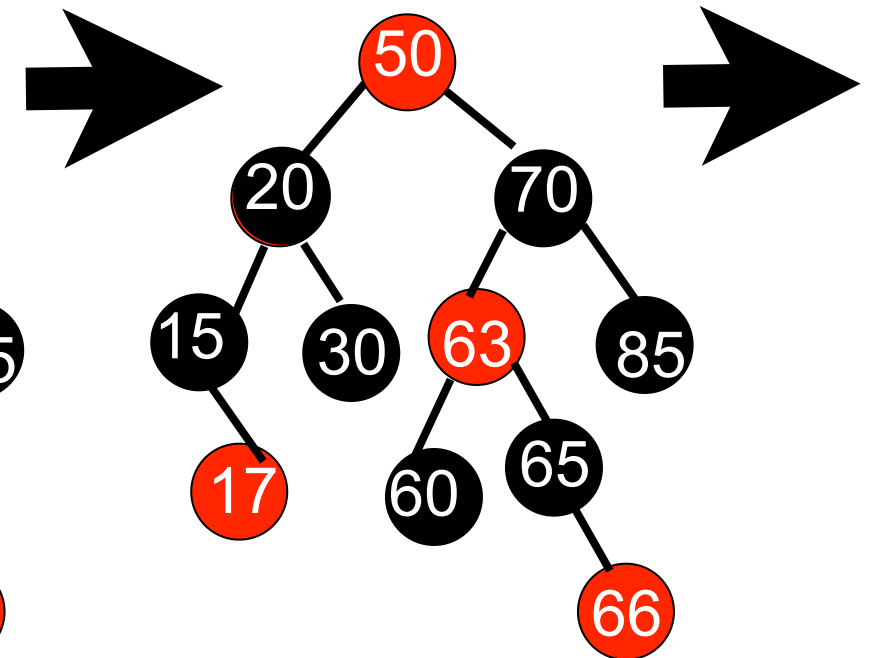
Double Red Problem

case 1 sibling of
parent is RED



Double Red Problem

case 1 sibling of
parent is RED



Root is not red

