Lecture 4

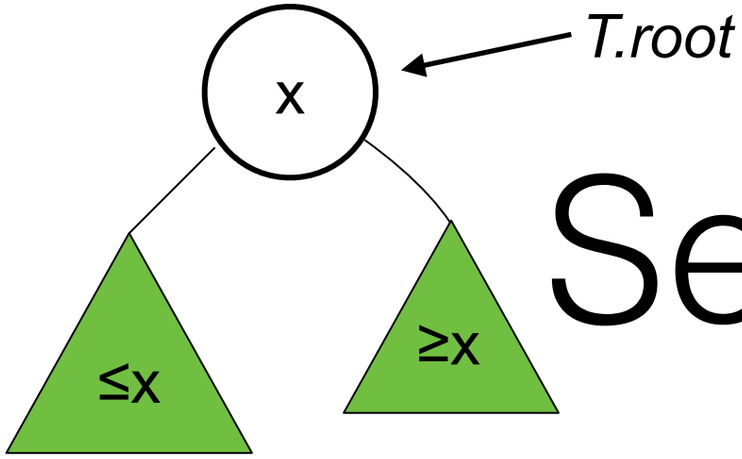
SEARCH
INSERT
DELETE

MINIMUM
MAXIMUM
SUCCESSOR
PREDECESSOR

How are you feeling today?

Linux Weekly News:

“There are a number of **red-black trees** in use in the kernel. The deadline and CFQ I/O schedulers employ **rbtrees** to track requests; the packet CD/DVD driver does the same. The high-resolution timer code uses an **rbtree** to organize outstanding timer requests. The ext3 filesystem tracks directory entries in a **red-black tree**. Virtual memory areas (VMAs) are tracked with **red-black trees**, as are epoll file descriptors, cryptographic keys, and network packets in the “hierarchical token bucket” scheduler.”



Self Balancing Trees

BST property

Balanced: height is O(log n)!

$O(\log n)$ height !!! Is for wow factor

2-3 Trees

2-3-4 Trees

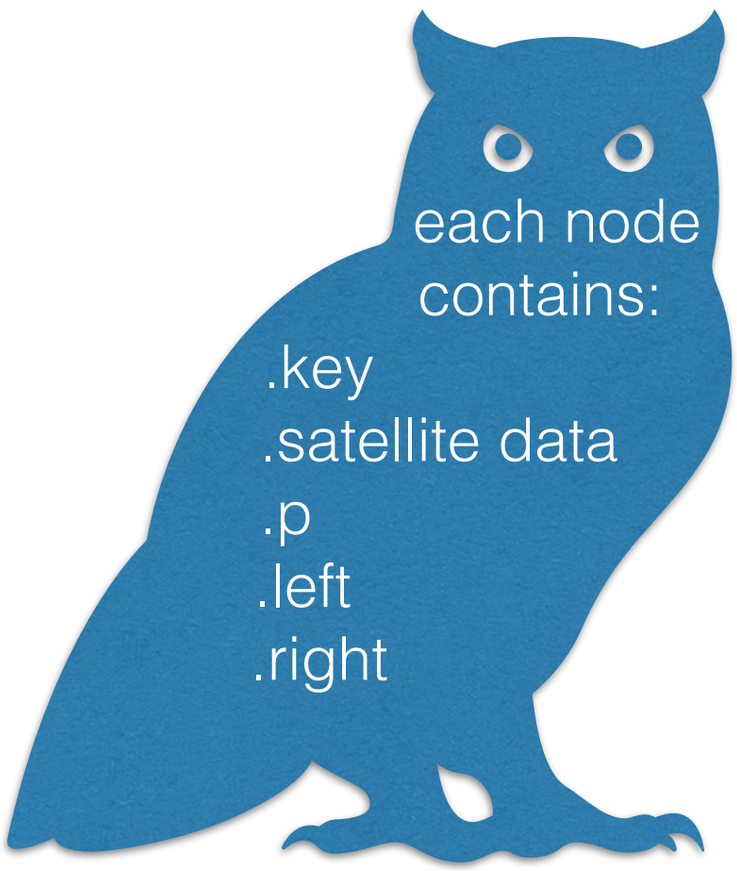
Red-Black Trees

Augmenting Data Types

a-b Trees

b-Trees

Running time to insert, delete and find in a Binary Search Tree?

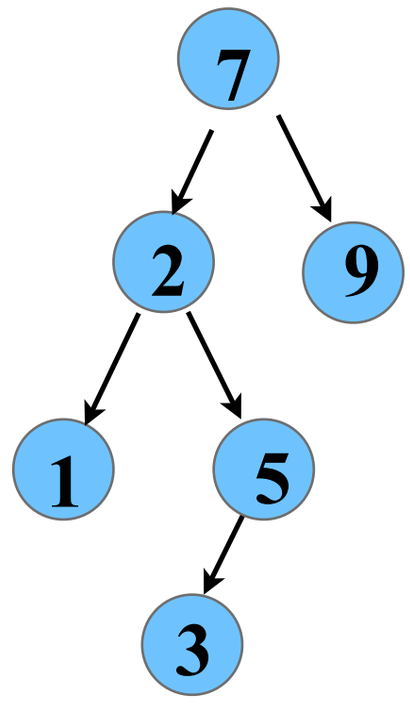


How long does it take to insert into a binary search tree in the worst case?

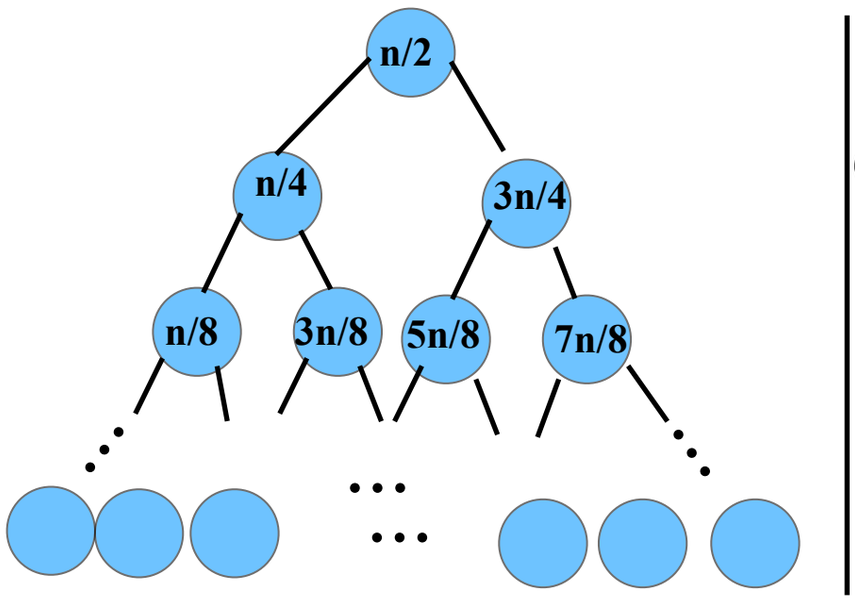
- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- None of the above

Inserting into a binary search tree

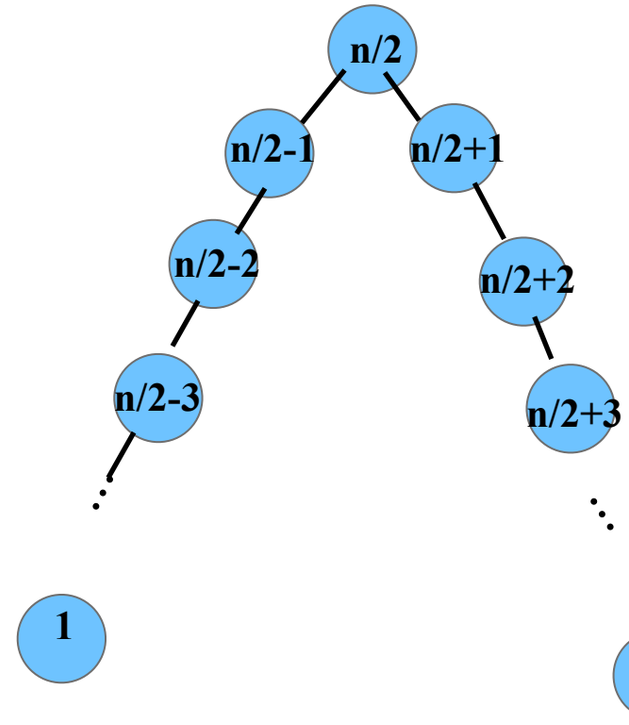
7 2 5 9 1 3

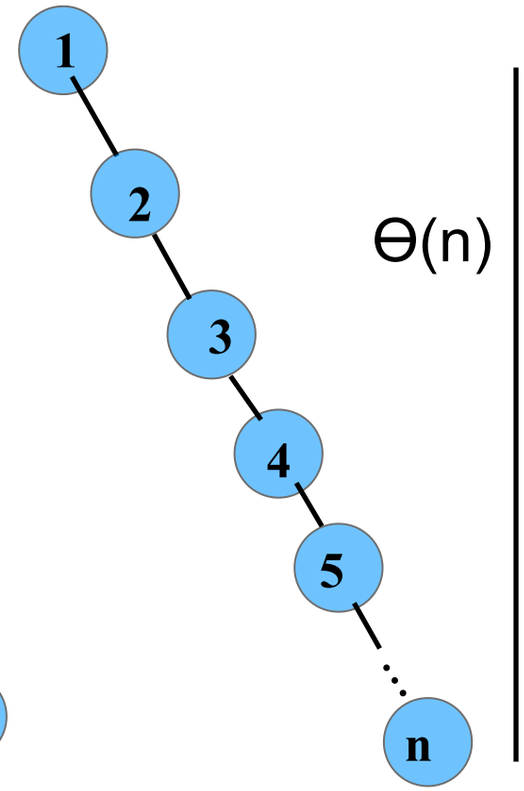


Why do we care about the height of a binary search tree?



Θ(log(n))





Θ(n)

In a *balanced* binary search tree it takes O(log(n)) *worst* case time for the operations:

- INSERT
- DELETE
- SEARCH
- PREDECESSOR
- SUCCESSOR
- MAXIMUM
- MINIMUM

In a binary search trees it takes O(h) *worst* case time to

- INSERT
- DELETE
- SEARCH
- PREDECESSOR
- SUCCESSOR
- MAXIMUM
- MINIMUM

Θ(n) for these trees

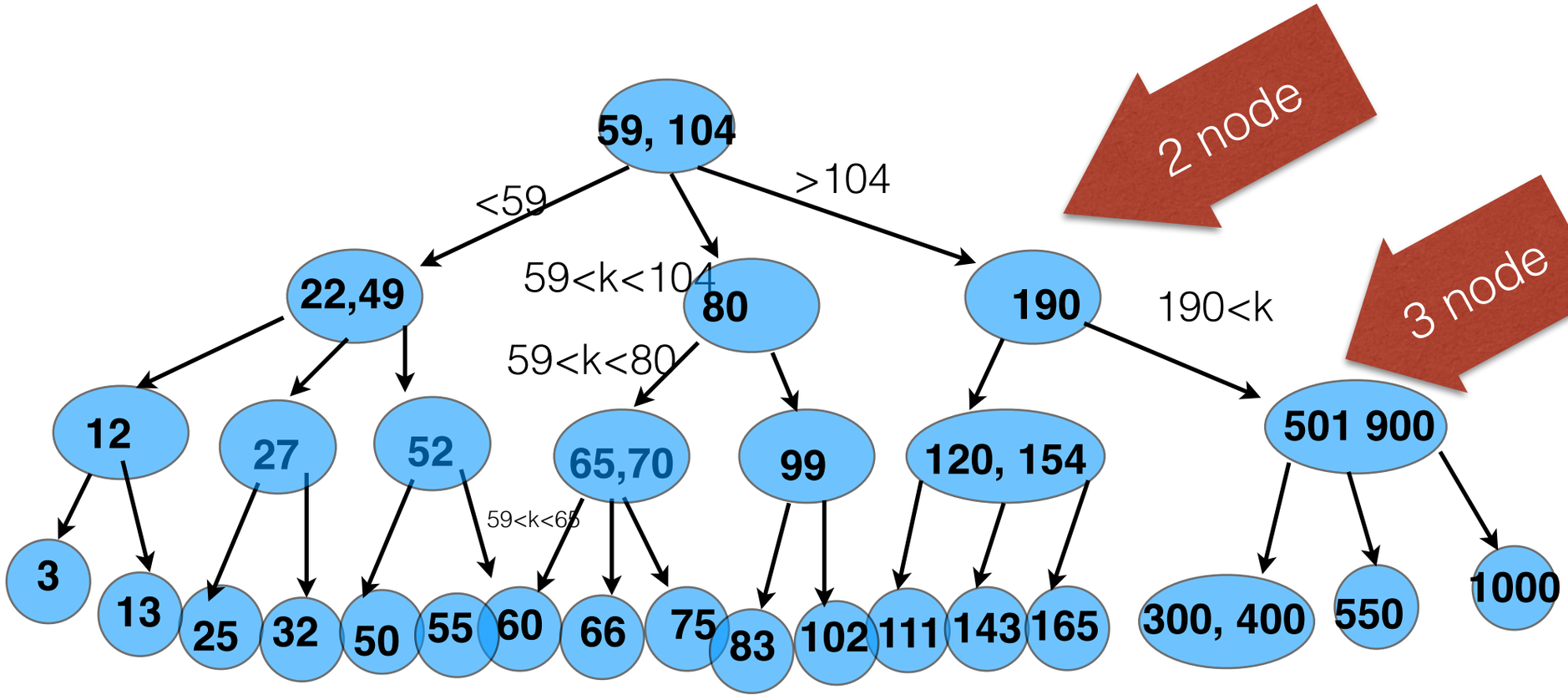
Question: How can we avoid $\Theta(n)$ searches?

Answer: Make sure our trees have height ~ log n

Can we make a perfectly balanced tree?

Break out rooms

What if we allowed our nodes to hold more than one key?



A perfectly balanced 2-3 tree all leaf nodes are at the *same distance* to the root

$$\lfloor \log_3 n \rfloor \quad <= h <= \quad \lfloor \log_2 n \rfloor$$

2-3 Tree

Not a binary Search Tree

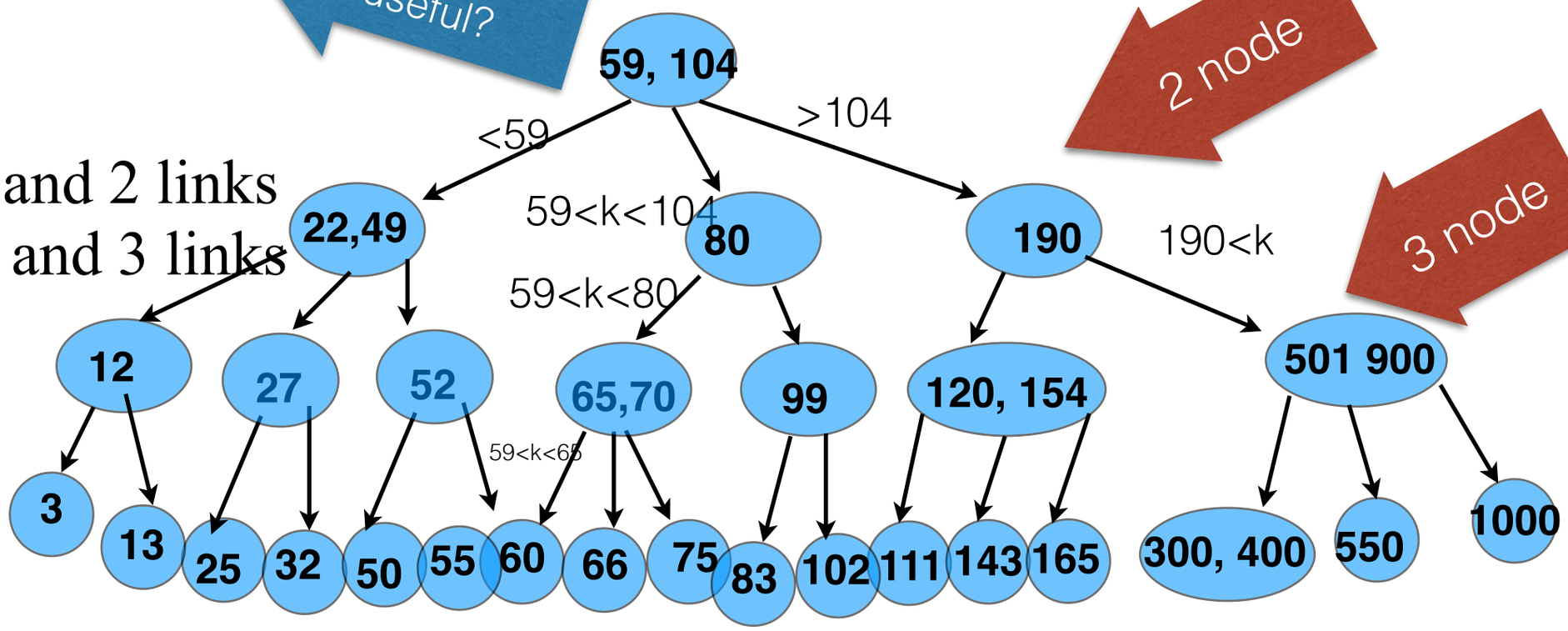
2-3 *search* tree properties

- at most 3 children
- all external nodes have same depth
- height of tree is Θ(log n)

How is this useful?

2-3 tree

- empty
- 2 node - one key and 2 links
- 3 node: two keys and 3 links

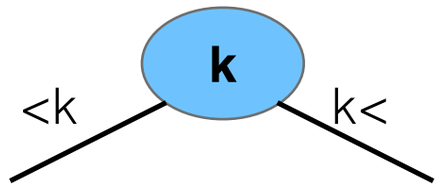


A perfectly balanced 2-3 tree all leaf nodes are at the *same distance* to the root

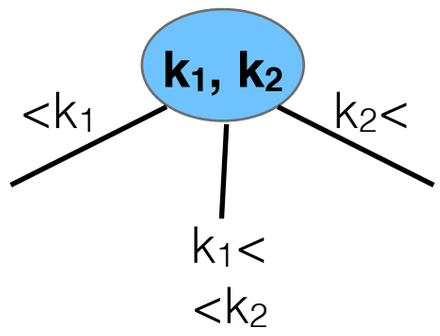
$$\lfloor \log_3 n \rfloor \quad <= h <= \quad \lfloor \log_2 n \rfloor$$

2-3 Tree Nodes

- **2-node:**
  Same as a binary node
  1 key and **2** links



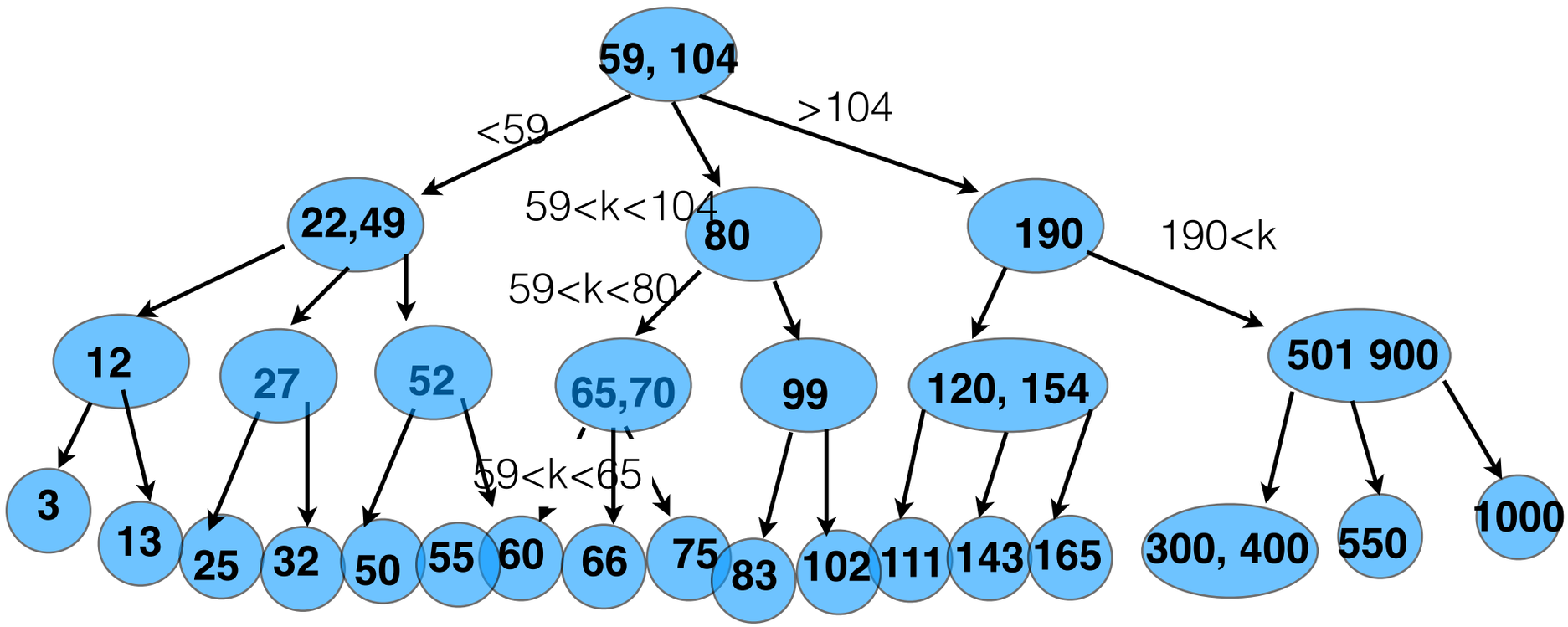
- **3-node:**
  2 keys and **3** links



2-3 Tree

Search (same as for BST)

- Compare search key with keys in node
- If search key equals a key in the node - done
- O/w follow the link that corresponds the the interval containing the search key

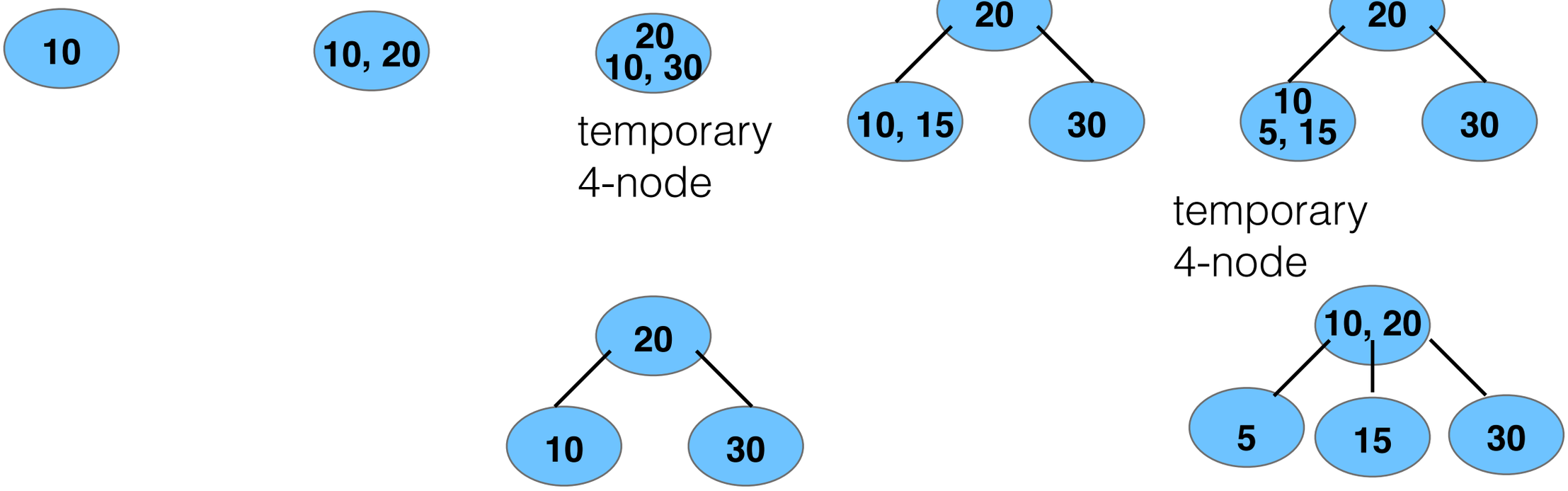


2-3 Tree

Insert

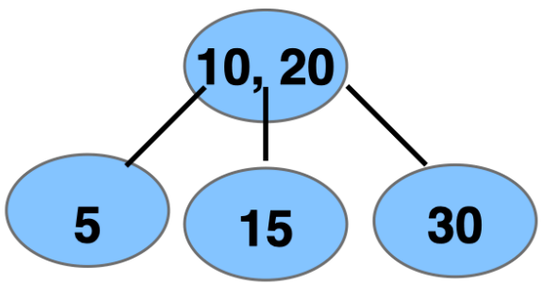
- Perform the same actions as for an unsuccessful search till you come to a leaf
- If the leaf contains 1 key: convert to 2-keys
- If the leaf contains 2 keys: split node moving up the middle value “up”

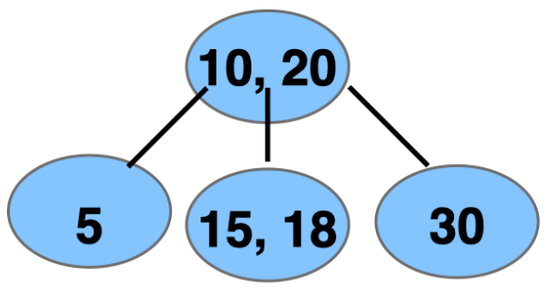
INSERT 10 INSERT 20 INSERT 30 INSERT 15 INSERT 5



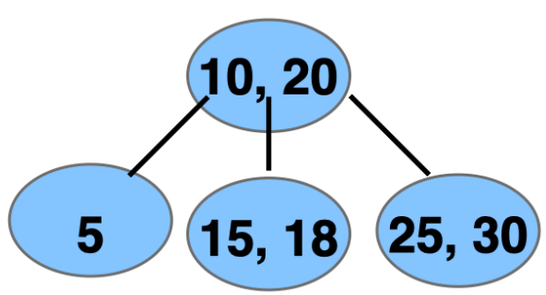
2-3 Tree

INSERT 18

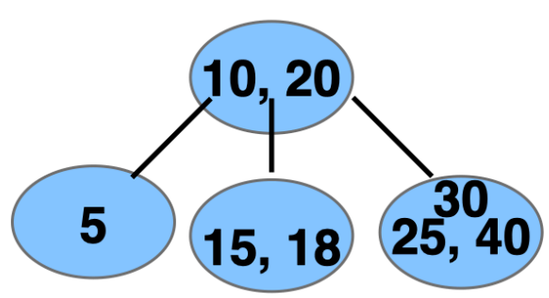




INSERT 25

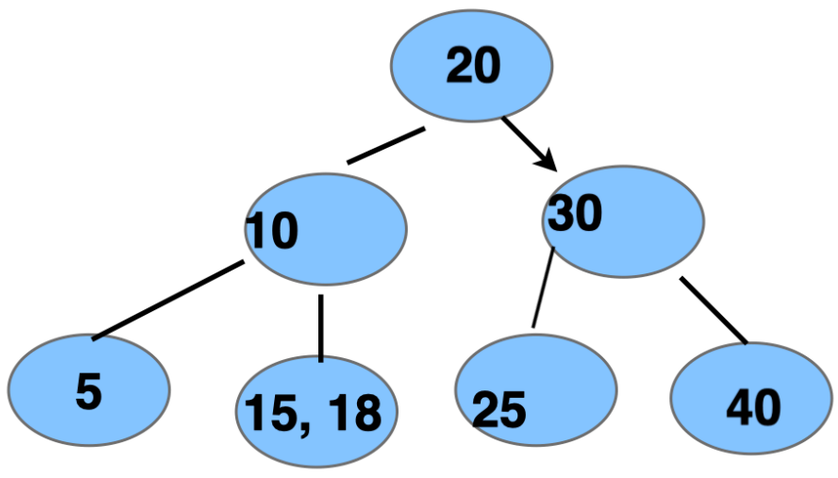


INSERT 40



temporary
4-node

- always insert into an existing node
- always maintain depth condition
- Split node if a 4-node, v, is created:
  - replace node v by creating two nodes v' and v'':
    1. v' gets the smallest key in v
    2. v'' gets the largest key in v
  - send middle key in v "up" the tree
    1. if v is the root - create a new node to hold the middle key
    2. o/w add middle key to parent (which may need to be split)



Global Properties

local transformations preserve

- tree is ordered
- perfectly balanced

} global properties

Worst case?

- Search(T, k) $O(\log n)$
- Insert(T, x) $O(\log n)$
- Delete(T, k) $O(\log n)$

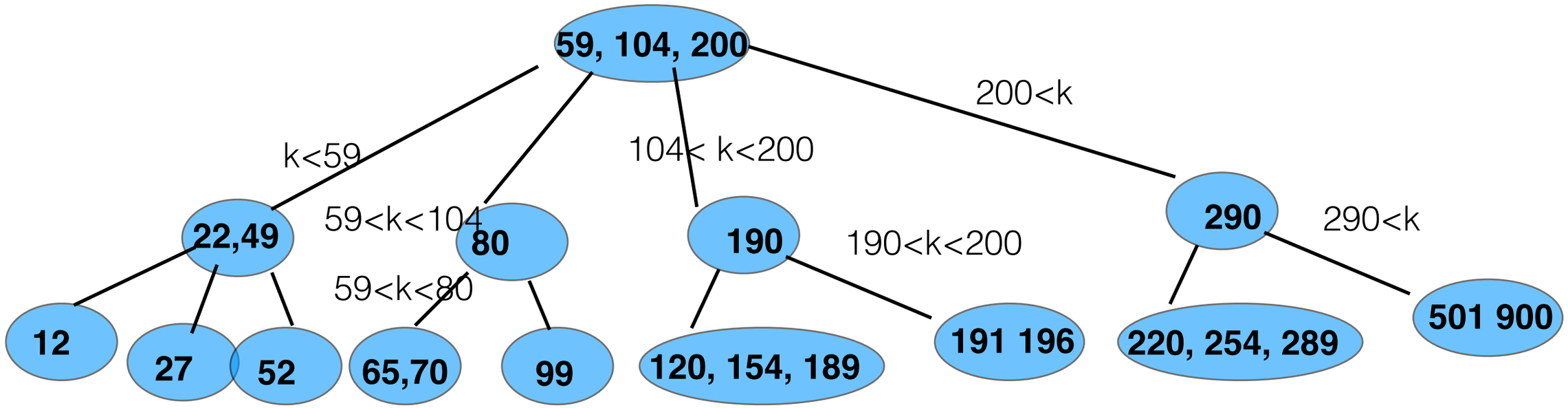
…Oops

- This is not so easy to implement
  - distinct data types: 2-nodes, 3-nodes
  - search would have to test which type of node
  - copy links other info from one type of node
  - convert nodes from one type to another

- Avoids the worse case behavior, but the overhead makes slower than standard BST search and insert…

What if we allowed our nodes to hold 1,2, or 3 keys? (We will use the ideas here in another data structure.)

2-4 Trees
(AKA 2-3-4 Trees)



A perfectly balanced 2-4 tree all leaf nodes are at the same distance to the root

$$\lfloor \log_4 n \rfloor <= h <= \lfloor \log_2 n \rfloor$$

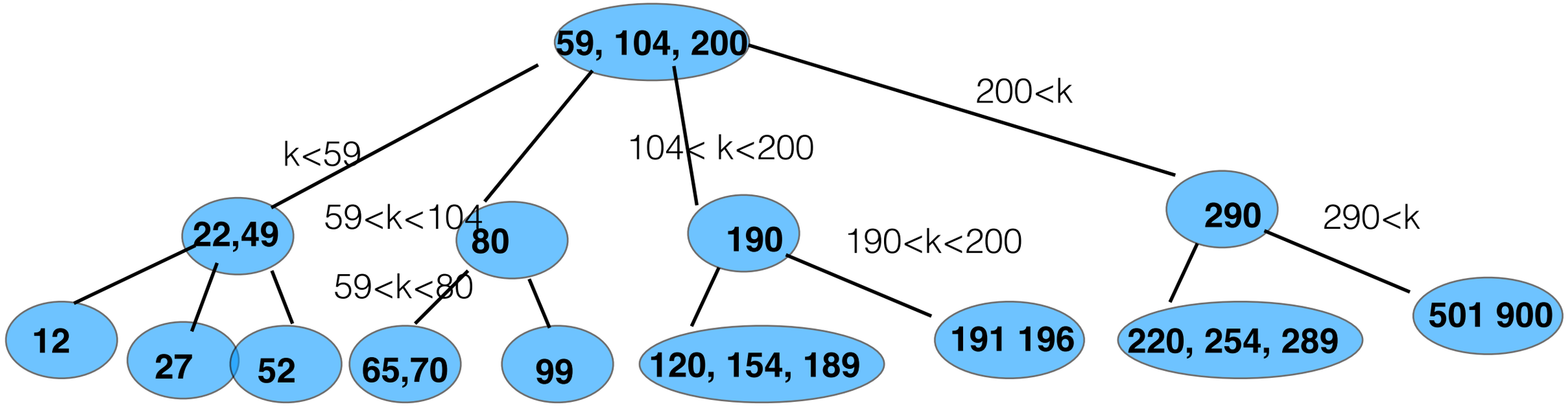
2-4 Trees (AKA 2-3-4 Trees)

2-4 *search* tree properties

- at most 4 children
- all external nodes have same depth
- height of tree is Θ(log n)

2-4 search tree

- empty
- 2 node - one key and 2 links
- 3 node: two keys and 3 links
- 4 node: three keys and 4 links

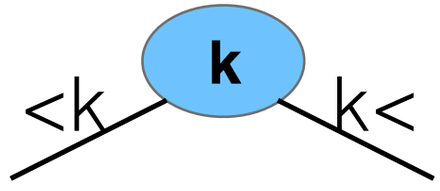


A perfectly balanced 2-4 tree all leaf nodes are at the same distance to the root

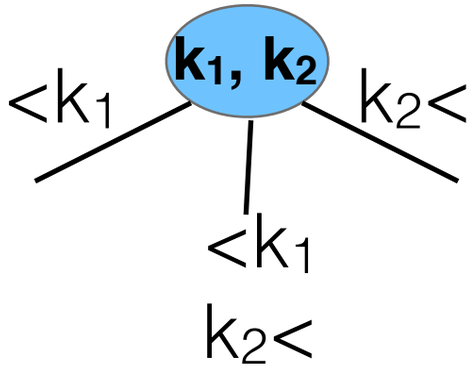
$$\lfloor \log_4 n \rfloor <= h <= \lfloor \log_2 n \rfloor$$

2-4 Tree Nodes

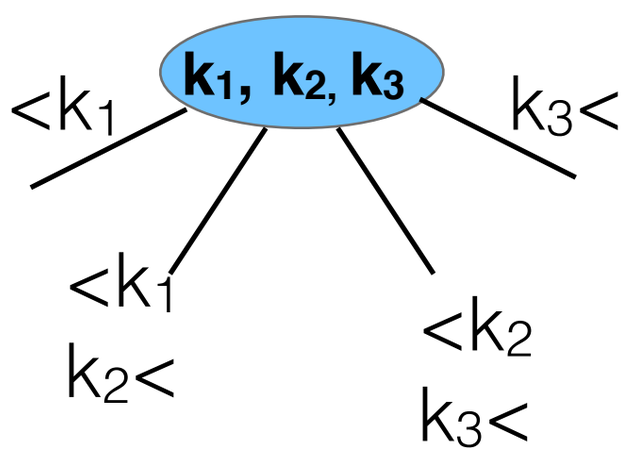
- **2-node:**
  Same as a binary node
  1 key and **2** links



- **3-node:**
  2 keys and **3** links



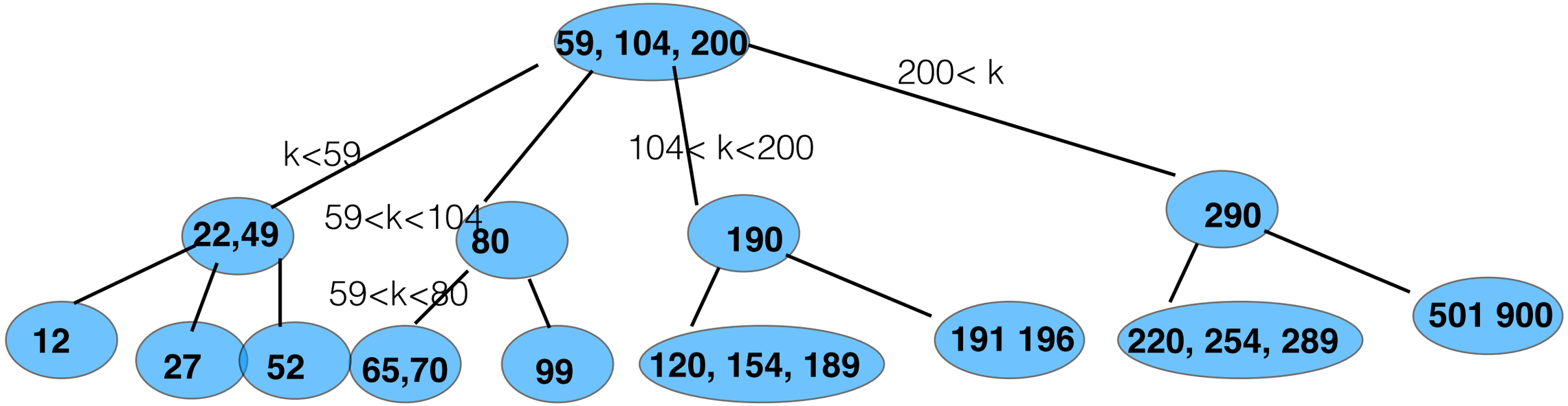
- **4-node:**
  3 keys and **4** links



2-4 Tree Search

Search (same as for BST and 2-3 trees)

- Compare search key with keys in node
- If search key equals a key in the node - done
- Ow follow the link that corresponds the the interval containing the search key

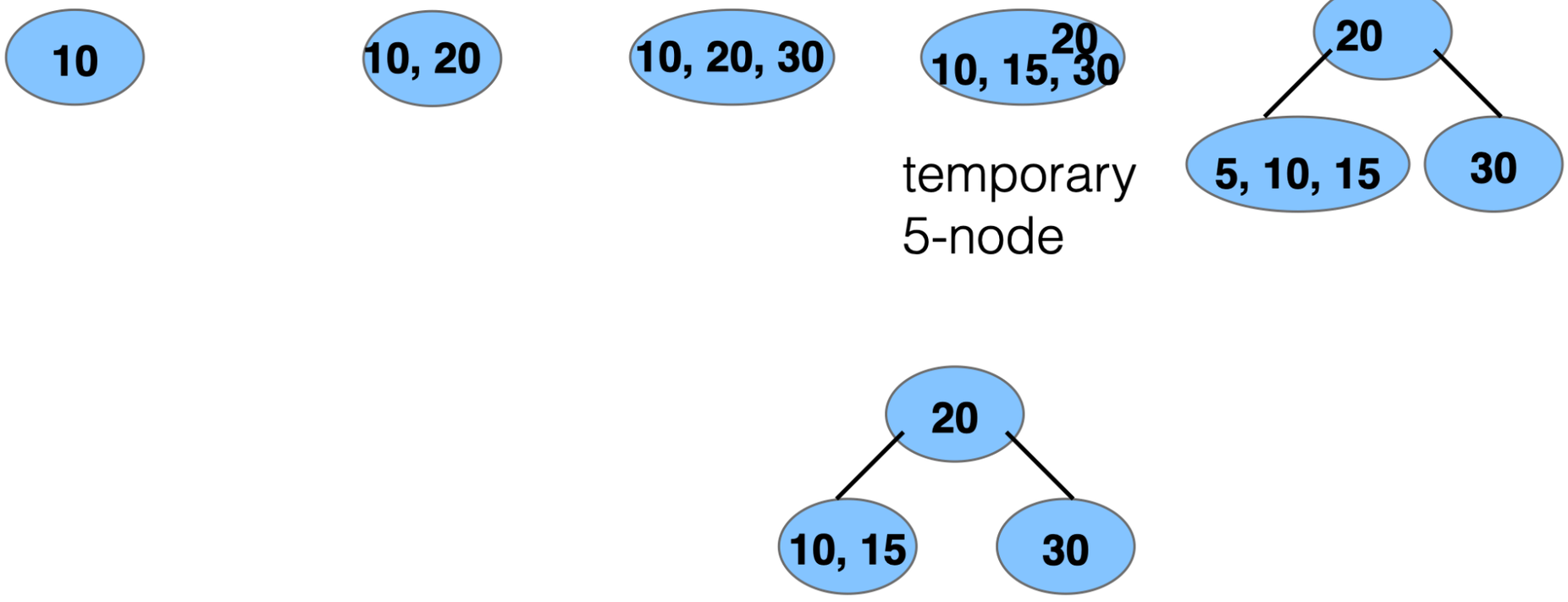


2-4 Tree Insertion Example

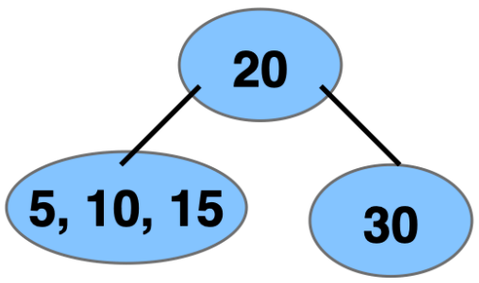
Insert

- Perform the same actions as for an unsuccessful search till you come to a leaf
- If the leaf contains 1 key: convert to 2-keys
- If the leaf contains 2 keys: convert to 3-keys
- If the leaf contains 3 keys: split node moving up the *middle* value "up"

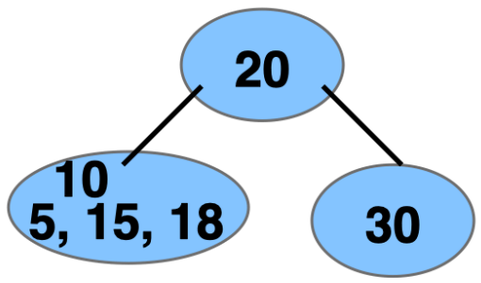
INSERT 10 INSERT 20 INSERT 30 INSERT 15 INSERT 5



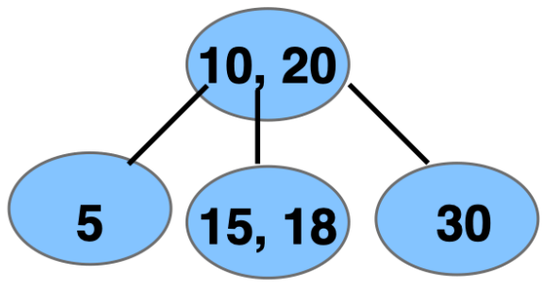
2-4 Tree Insertion Example Cont.



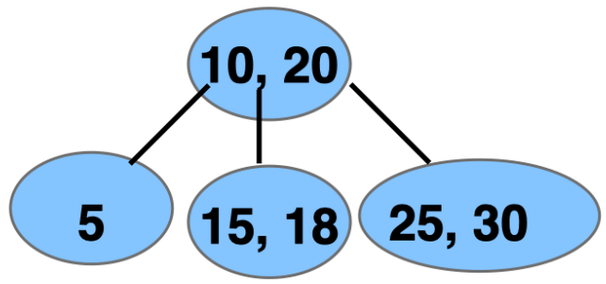
INSERT 18



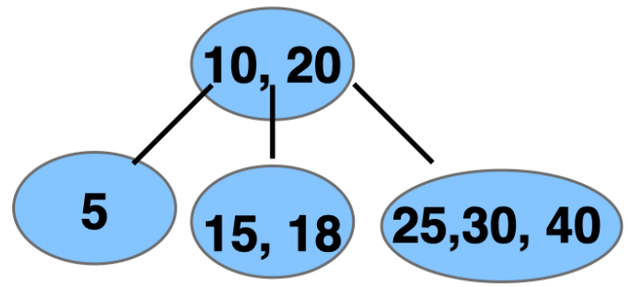
temporary
5-node



INSERT 25

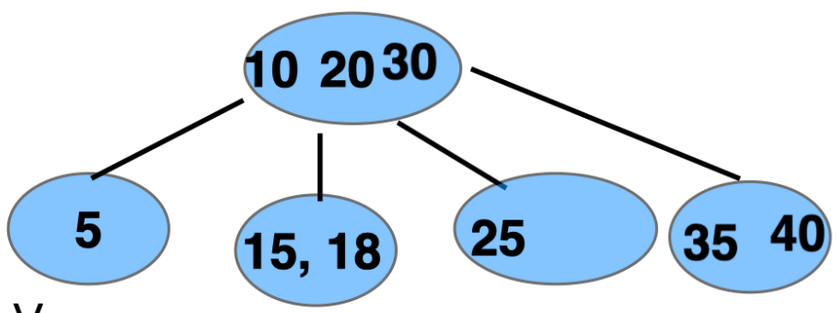


INSERT 40



- always insert into an existing node
- always maintain depth condition
- Split node if a 5-node, v, is created:
  - replace node v by creating two nodes v' and v'':
    1. v'' gets the key(s) smaller than *middle* key in v
    2. v'' gets the key(s) larger than the *middle* key in v
  - send *middle* key in v "up" the tree
    1. if v is the root - create a new node to hold the middle key
    2. o/w add middle key to parent (which may need to be split)

INSERT 35



Global Properties

local transformations preserve

- tree is ordered
- perfectly balanced

} global properties

Worst case?

- Search(T, k) $O(\log n)$
- Insert(T, x) $O(\log n)$
- Delete(T, k) $O(\log n)$

…Oops

- This is not so easy to implement
  - distinct data types: 2-nodes, 3-nodes, 4-nodes
  - search would have to test which type of node
  - copy links other info from one type of node
  - convert nodes from one type to another

- Avoids the worse case behavior, but the overhead makes slower than standard BST search and insert…

Lets relax the idea that the tree has to be perfectly balanced

Break out rooms

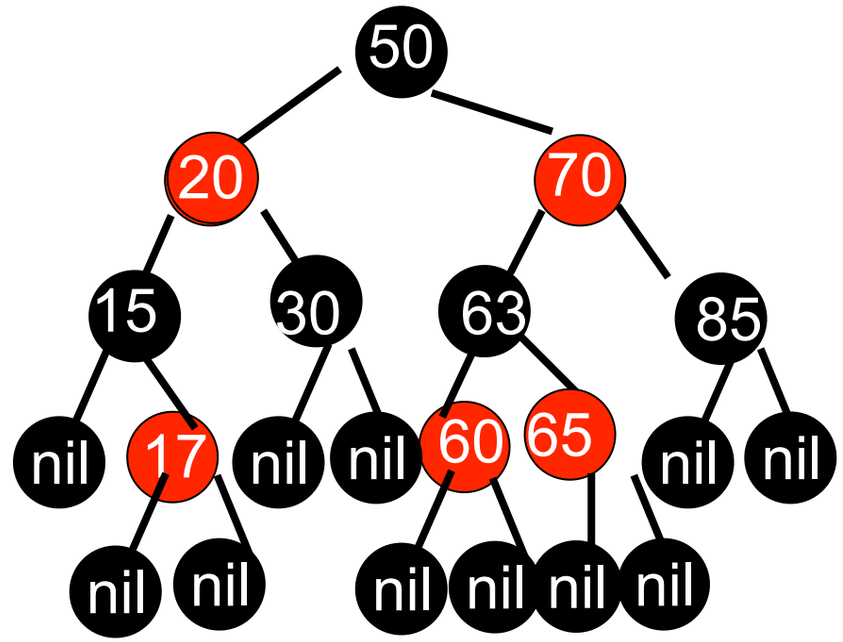
Red-Black Trees

So popular - it has its own youtube video!

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

Red-Black Tree

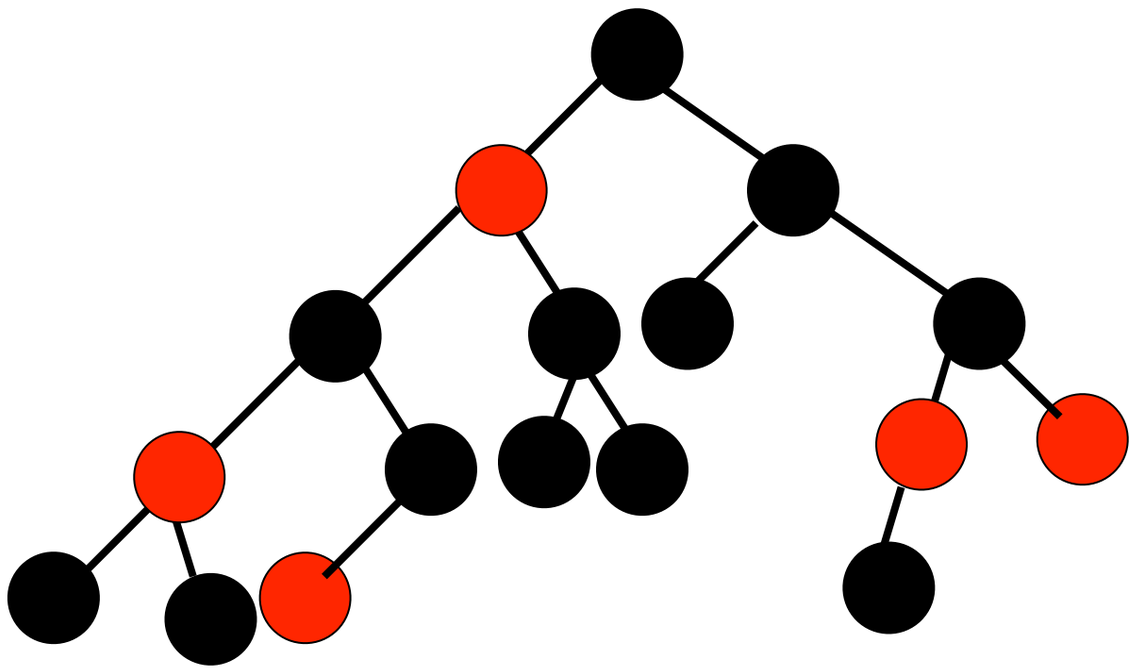
A Red-Black tree is a binary search tree that obeys 5 properties:



1) Every node is colored red or black
2) The root is black
3) Every leaf (T.nil) is black
   and doesn't contain any data
4) Both children of a red node are black
5) [black property] For every node in the tree, all paths from that node down to a leaf (nil node) have the same number of black nodes along the path

We won't show the nil nodes most of the time - but don't forget they exist

Not a Red-Black Tree

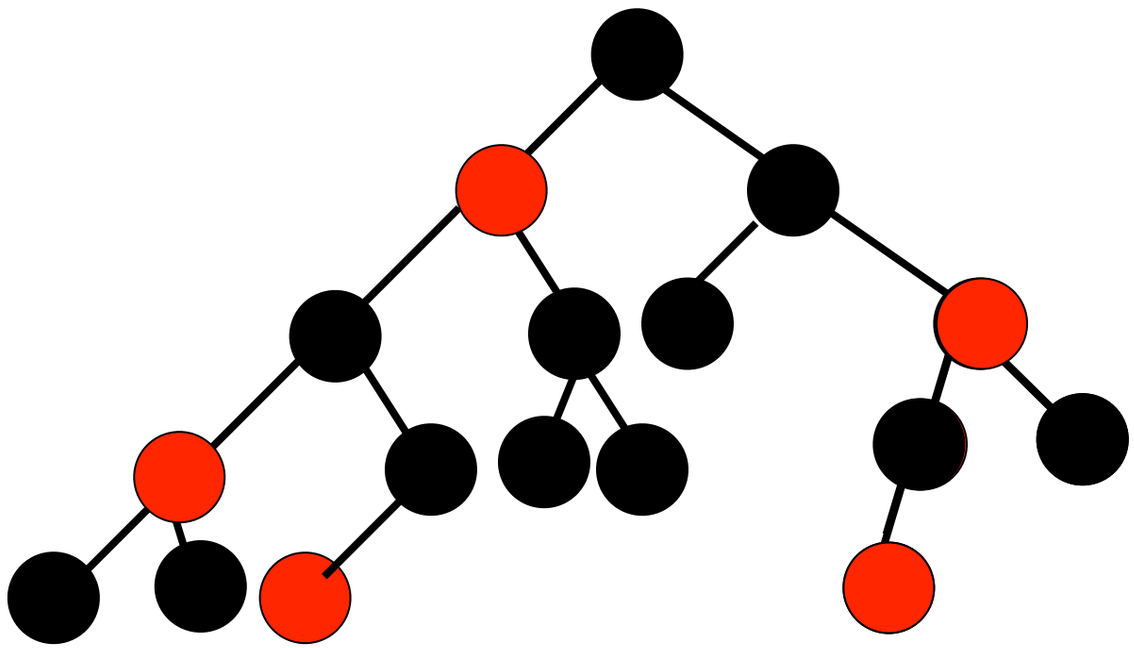


Nil nodes not shown on this slide

1) Every node is colored red or black
2) The root is black
3) Every leaf (T.nil) is black
   and doesn't contain any data
4) Both children of a red node are black
5) [black property] For every node in the tree, all paths from that node down to a leaf (nil node) have the same number of black nodes along the path

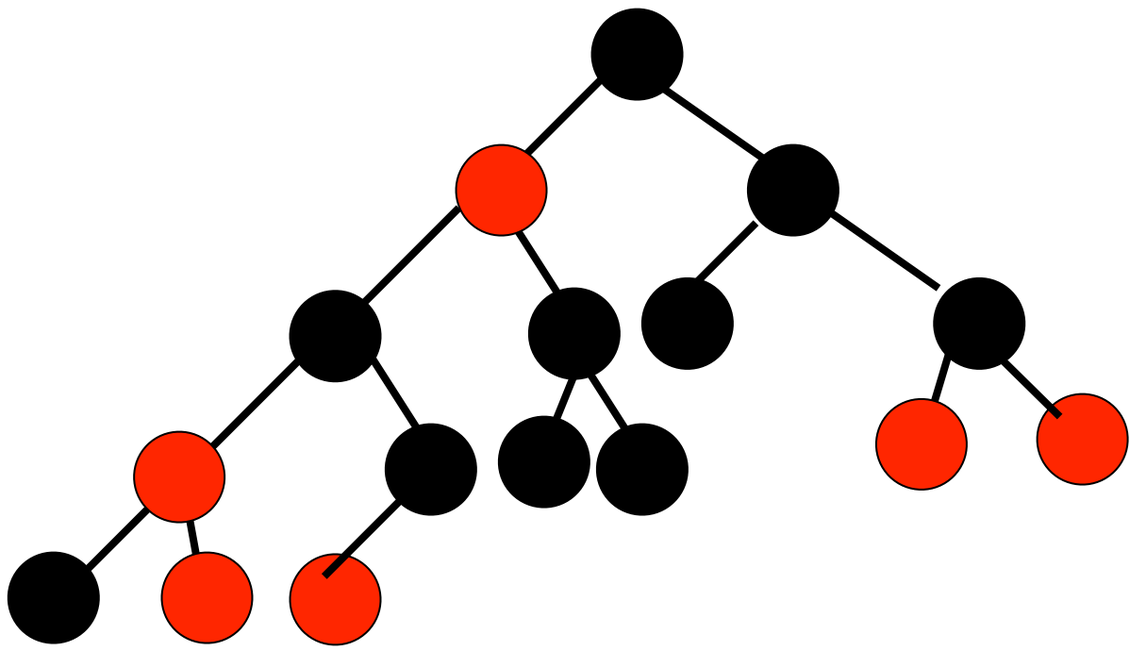
Not a Red-Black Tree

Nil nodes not shown on this slide



1) Every node is colored red or black
2) The root is black
3) Every leaf (T.nil) is black
   and doesn't contain any data
4) Both children of a red node are black
5) [black property] For every node in the tree, all paths from that node down to a leaf (nil node) have the same number of black nodes along the path

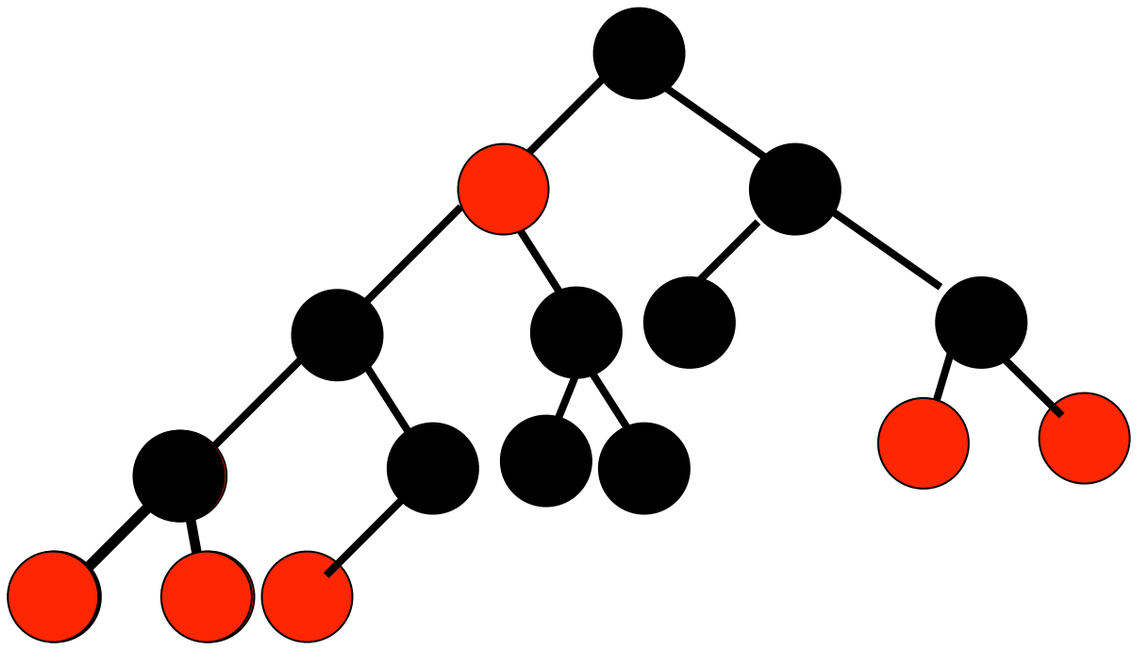
Not a Red-Black Tree



Nil nodes not shown on this slide

1) Every node is colored red or black
2) The root is black
3) Every leaf (T.nil) is black
   and doesn't contain any data
4) Both children of a red node are black
5) [black property] For every node in the tree, all paths from that node down to a leaf (nil node) have the same number of black nodes along the path

A Red-Black Tree

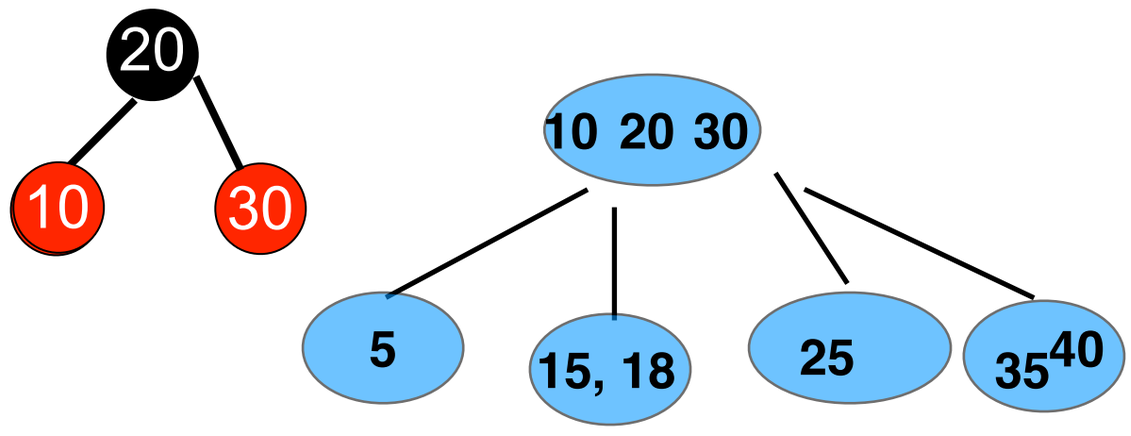


Nil nodes not shown on this slide

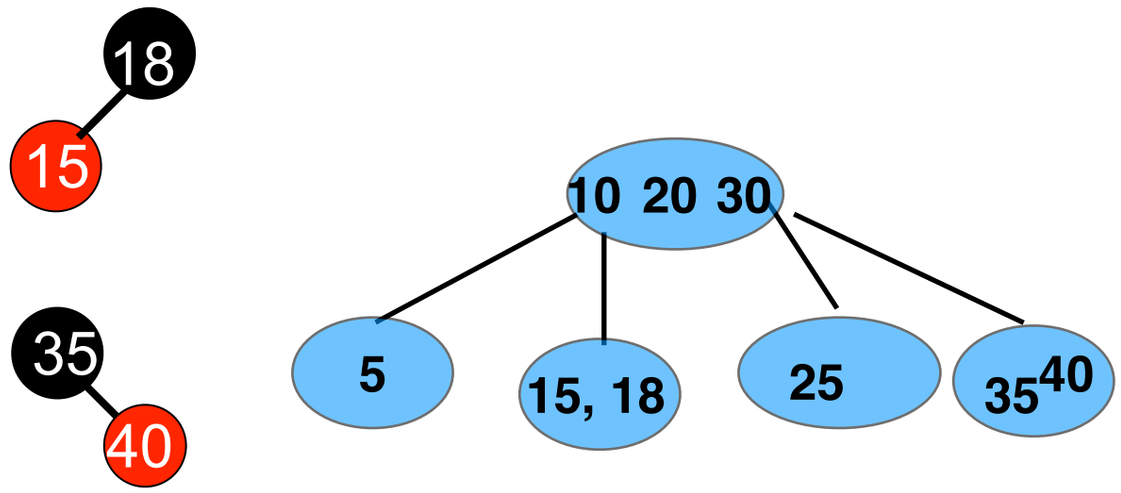
1) Every node is colored red or black
2) The root is black
3) Every leaf (T.nil) is black
   and doesn't contain any data
4) Both children of a red node are black
5) [black property] For every node in the tree, all paths from that node down to a leaf (nil node) have the same number of black nodes along the path

Creating a Red-Black BST from a 2-3-4 Tree

Encoding 4-nodes as *three* nodes: two red, one black. The black node will be the parent of the red children



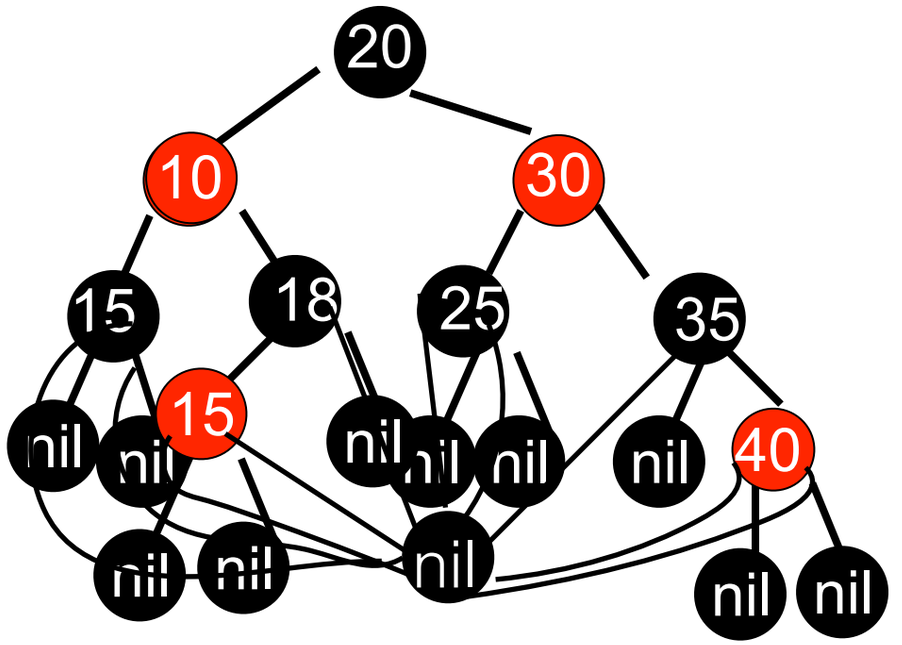
Encoding 3-nodes as *two* nodes: one red, one black. The red node will be the left (or right) child of the black node



Encoding 2-nodes as *one* node: color the node black

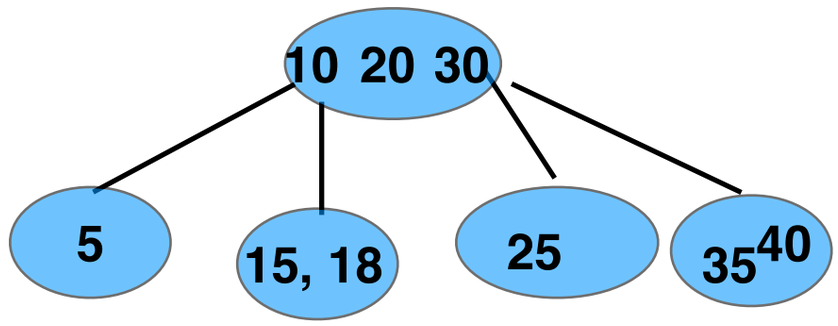
Creating a Red-Black BST from a 2-3-4 Tree Cont.

- Now we can use the BST *search* code *without any changes*! (The only code we need to change is when an item inserted or deleted.)



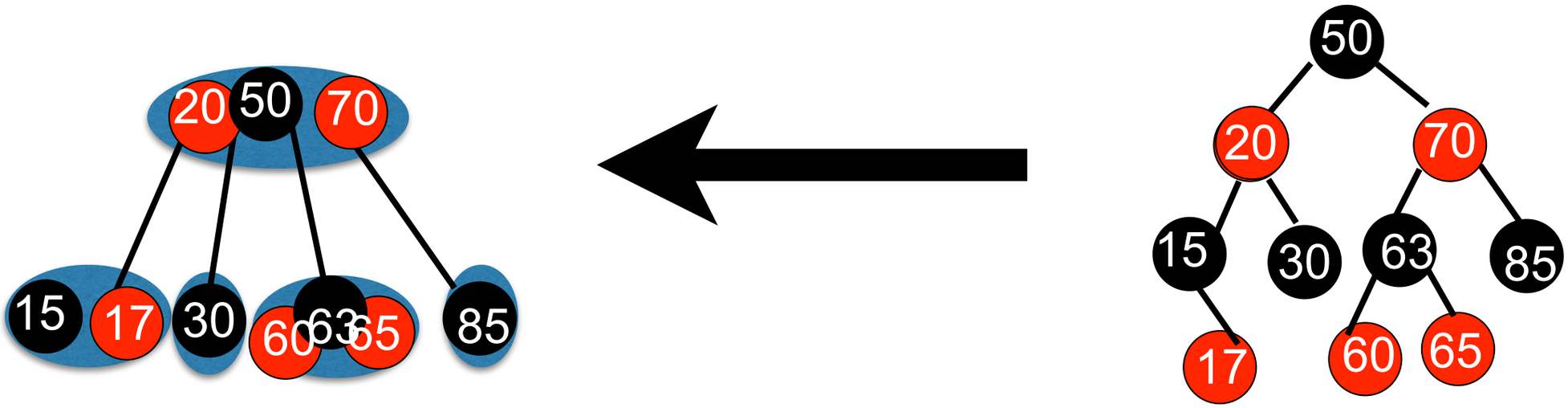
Additional details:

- all leaves are empty (nil) and are colored black.
- Sometimes I will show the empty leaf nodes using a coding trick. The *coding trick is to make a singly sentinel, T.nil for all the leaves, and for T.nil to also be the root's parent.*



Creating a 2-4 tree from a red-black tree

merge every red node into its black parent

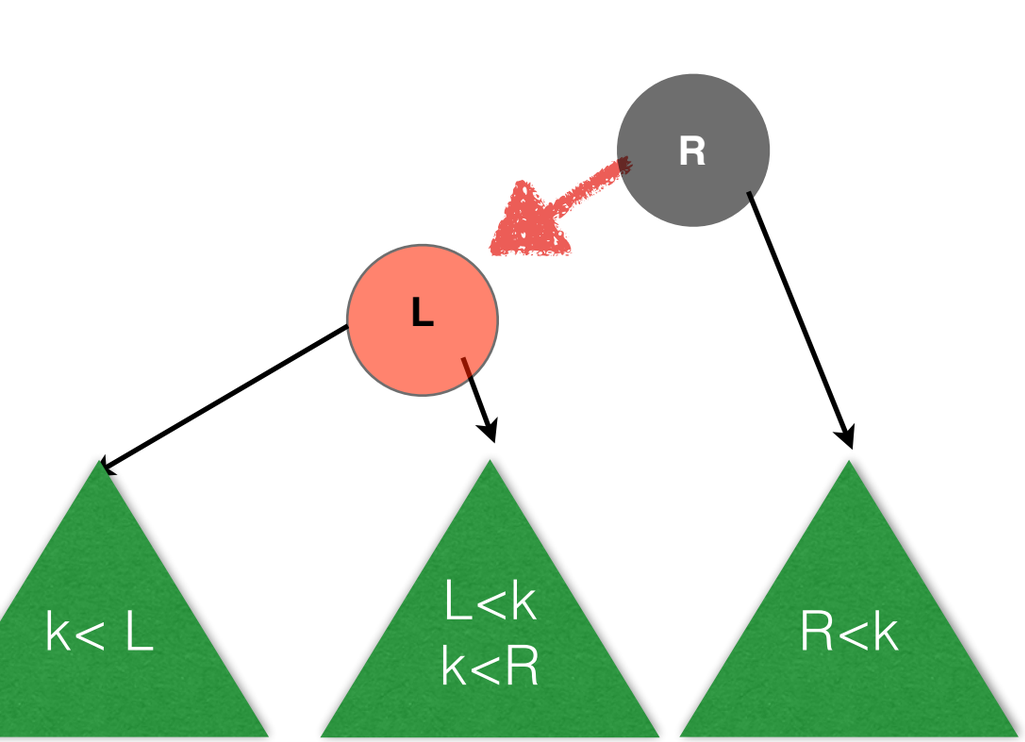


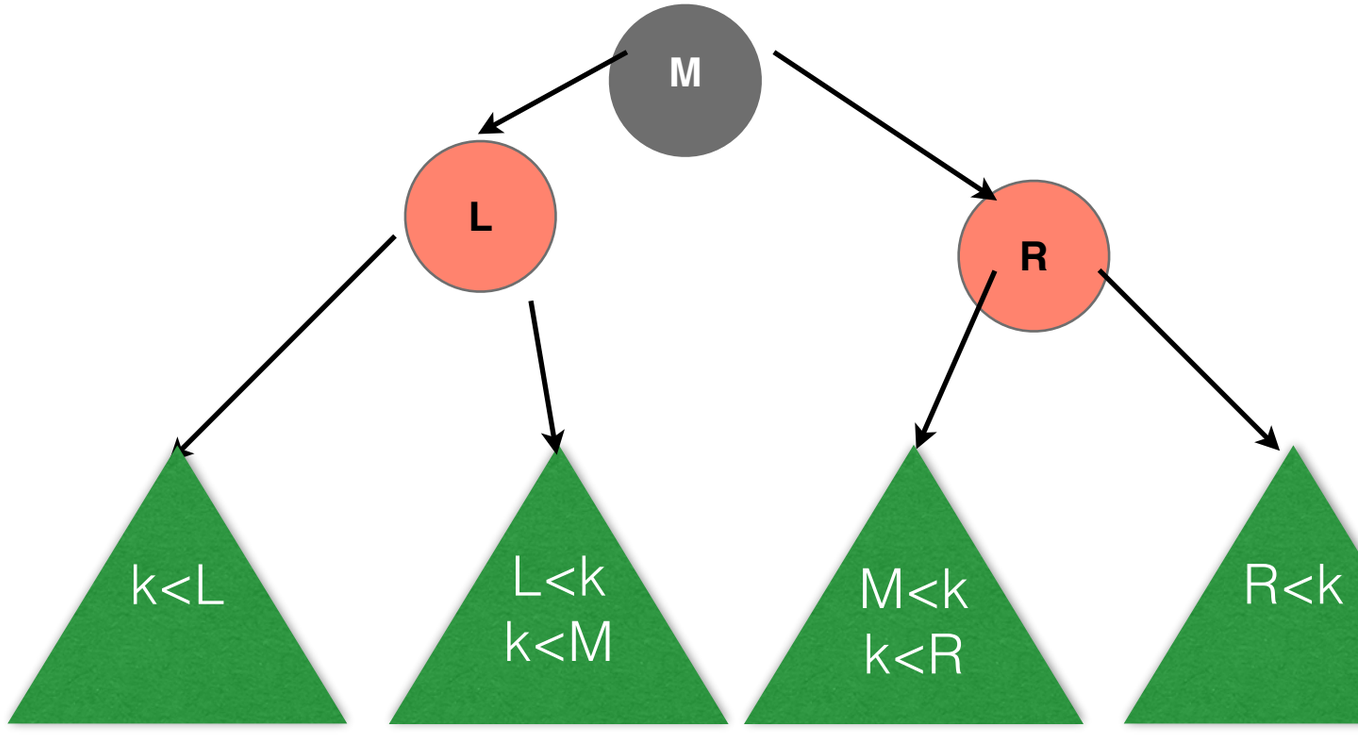
2-3-4 Tree

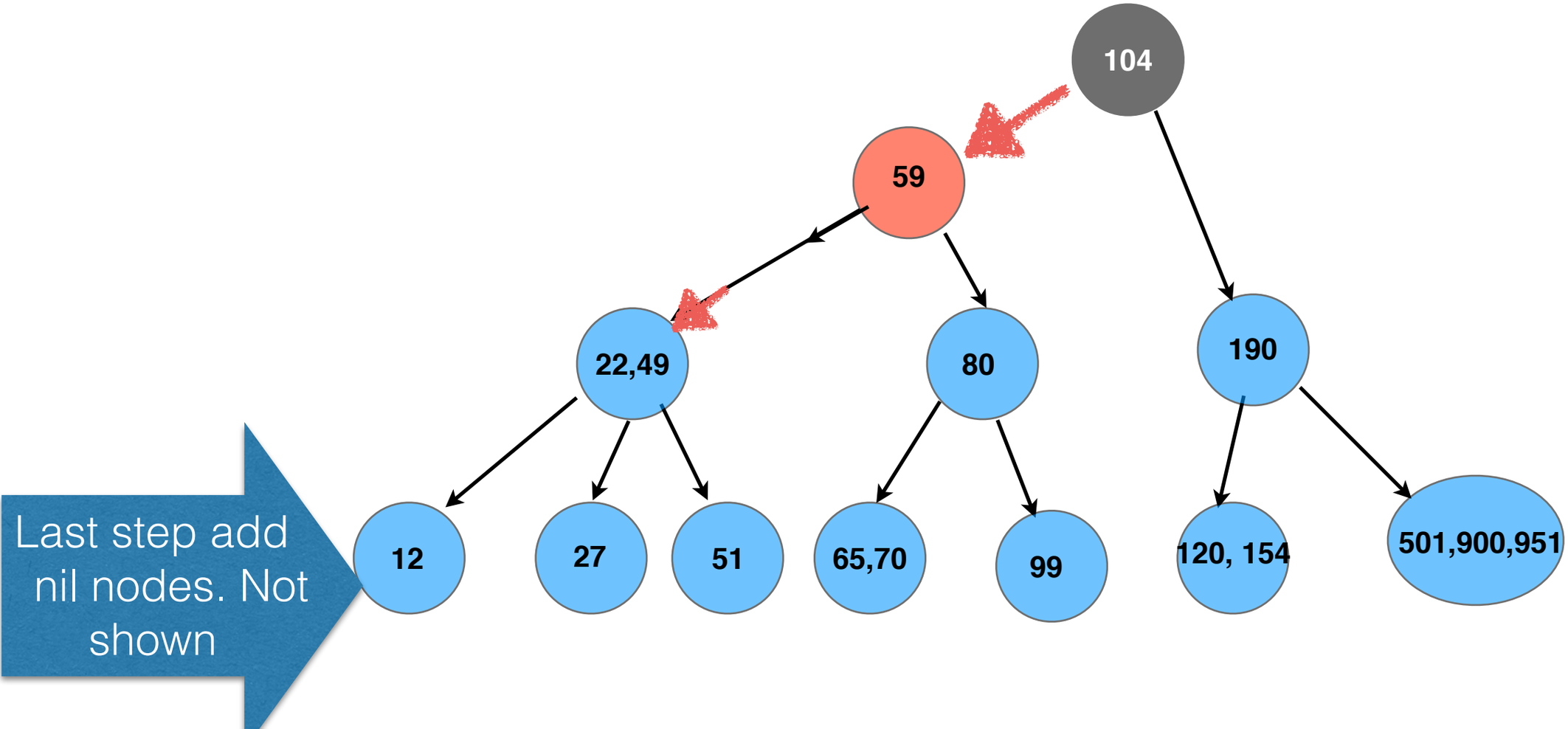
change so that we don't merge every node to parent

- every node has 2, 3 or 4 children (i.e. 1,2, or 3 keys)
- every leaf has exactly the same depth
- maximum height same as red-black tree

Transformation to a red-black tree





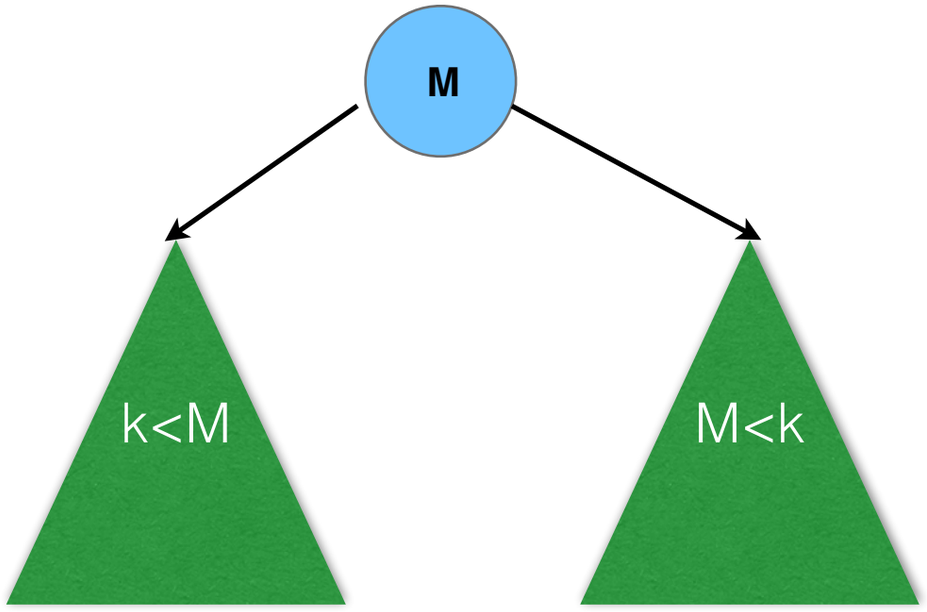


Converting between 2-4 Trees and Red-Black Trees

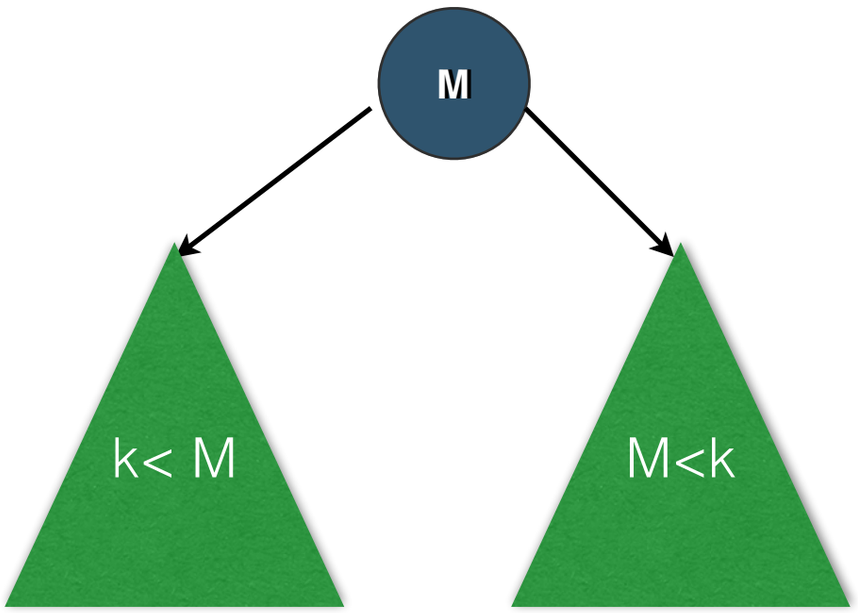
Transformation of a 2-node

2-4 tree

Color the node **black**

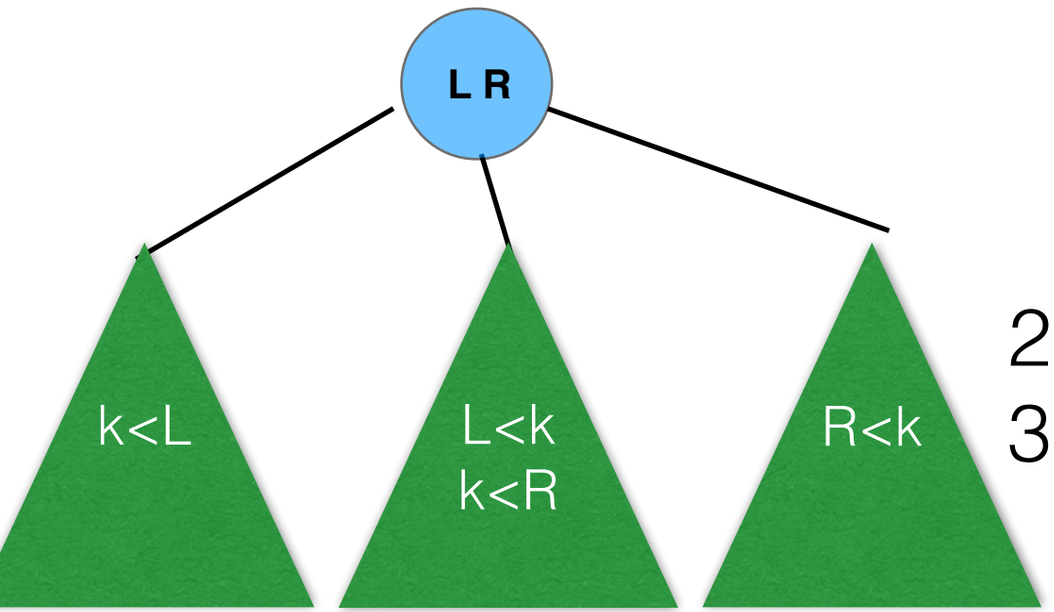


converting the 2-3 tree to a red-black tree



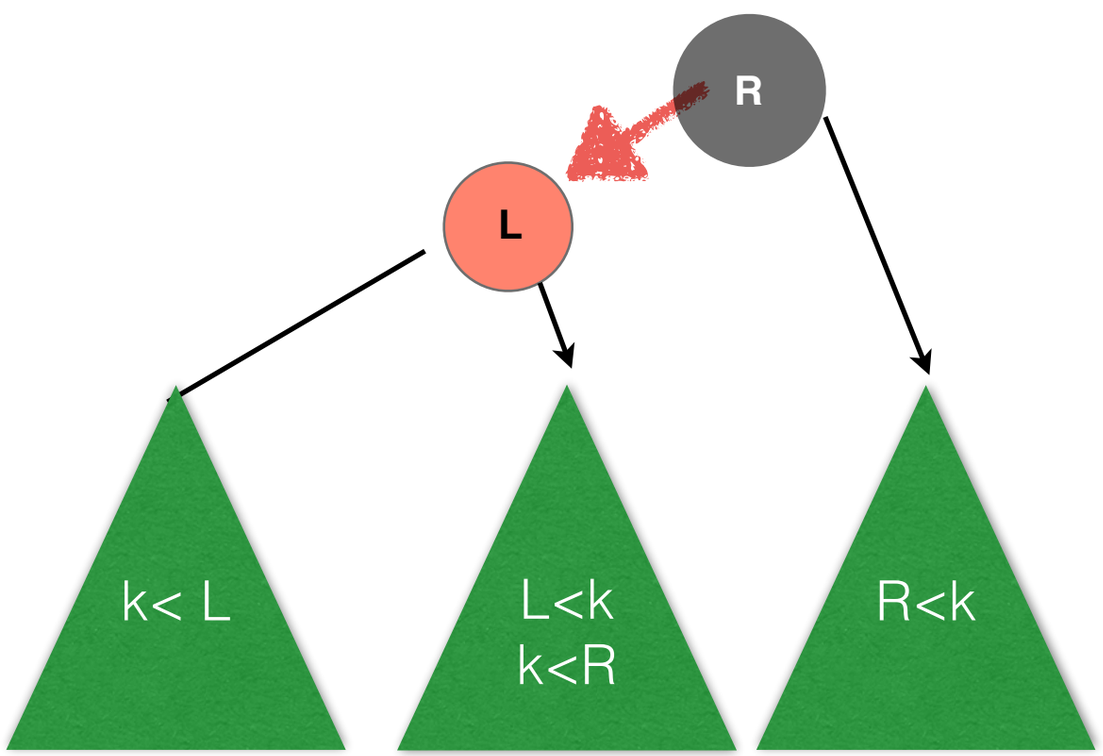
Transformation of a 3-node

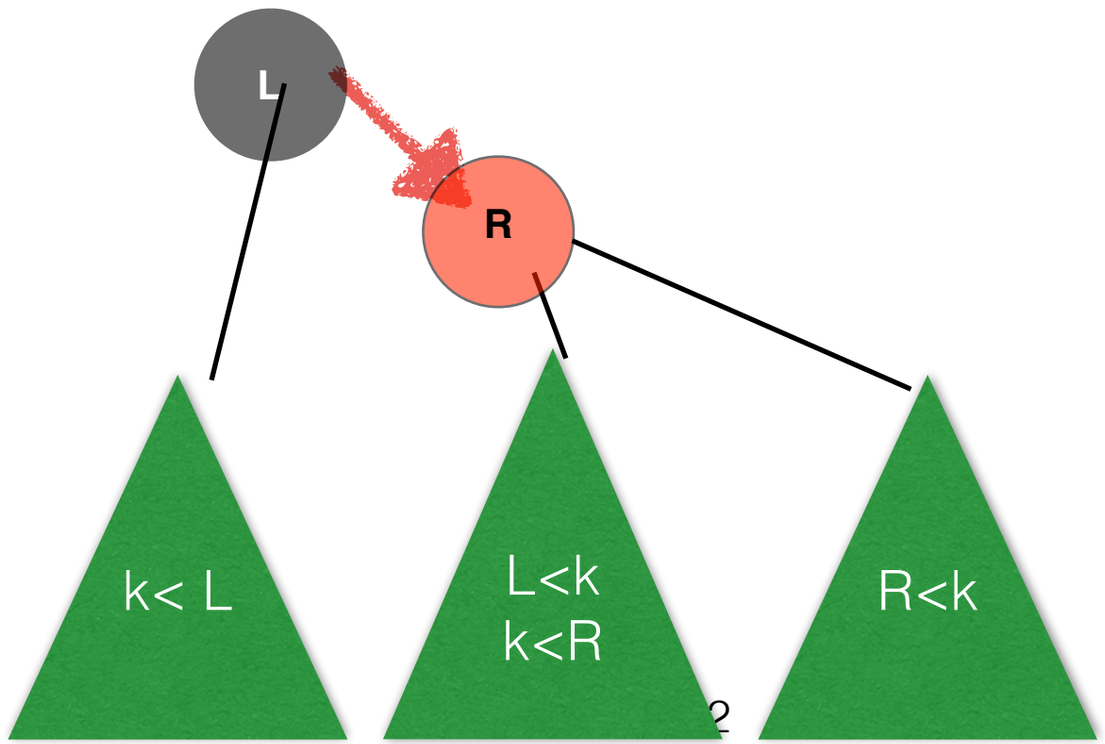
2-4 tree



1. Make 2 new nodes from the 3-node
   Two ways, either
   - Make **L** the parent of **R**, or
   - Make **R** the parent of **L**
2. Reattach subtrees
3. Color the parent **black** and the child **red**

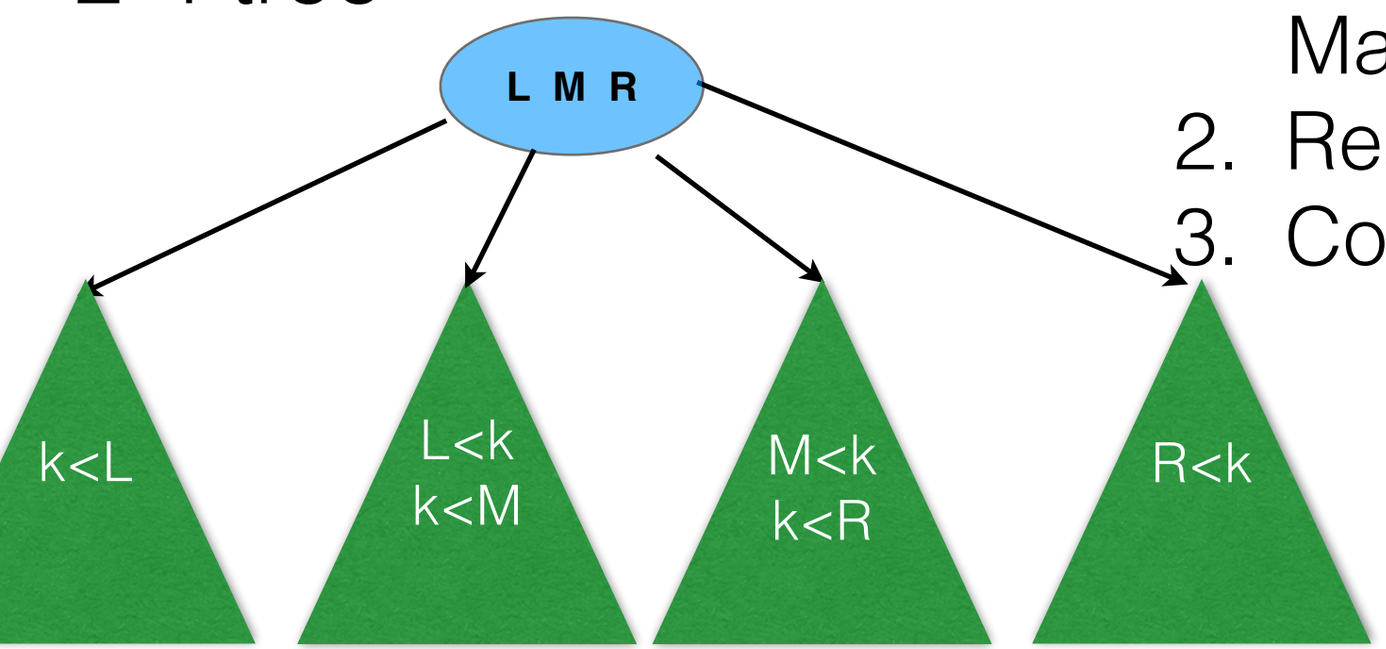
converting the 2-3 tree to a red-black tree





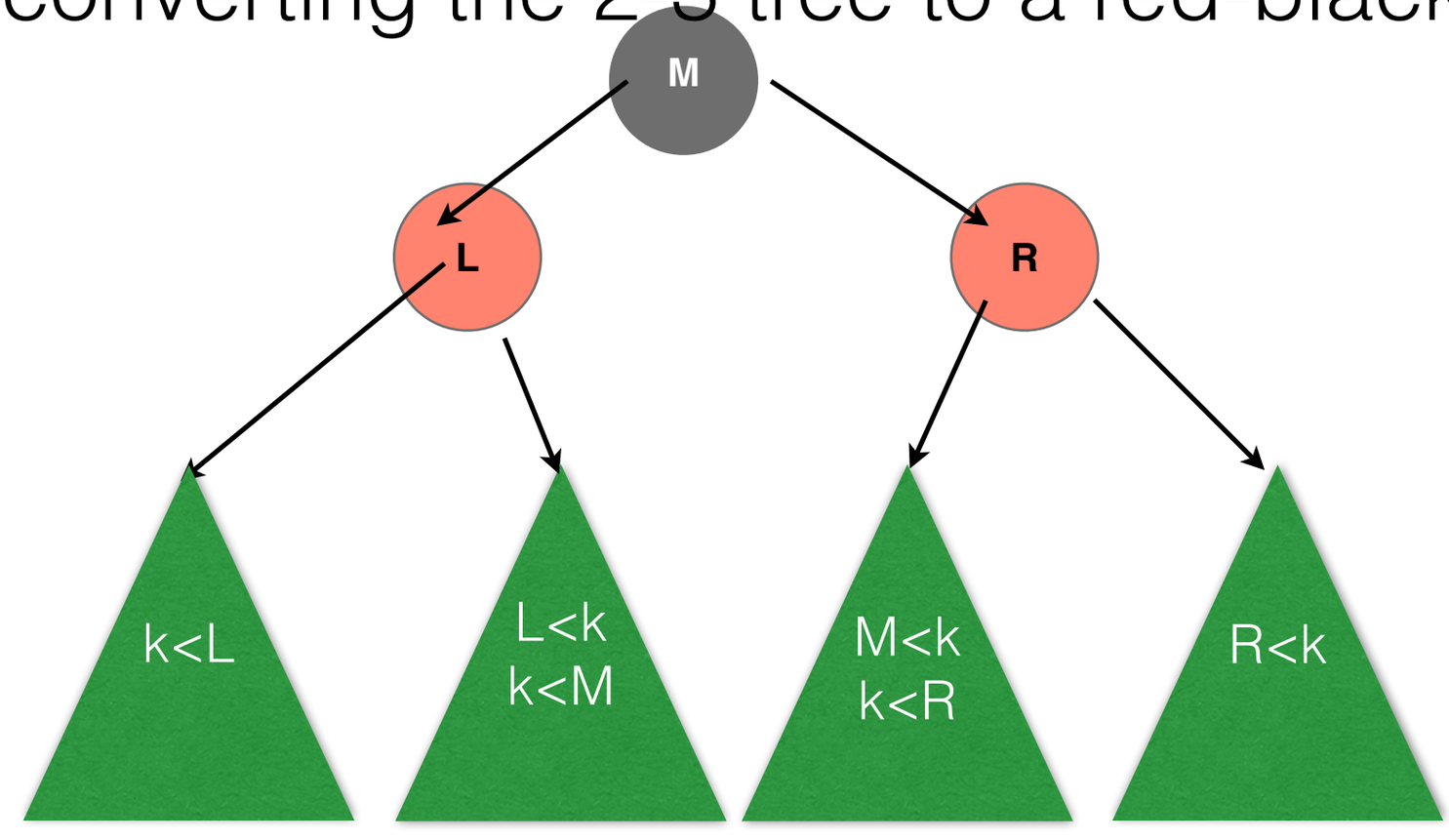
Transformation of a 4 node

2-4 tree



1. Make 3 new nodes from the 4-node
   Make **M** the parent of **L** and **R**
2. Reattach subtrees
3. Color **M** black and color **L, R** red

converting the 2-3 tree to a red-black tree



Note that the conversion from a 2-4 tree was not unique!

Note it is also possible to convert at 2-3 tree to a red-black tree

Conversion produces a valid Red-Black Tree (except for adding the nil nodes)!

- **Property 1**: Every node is colored red or black
- **Property 2**: The root is black

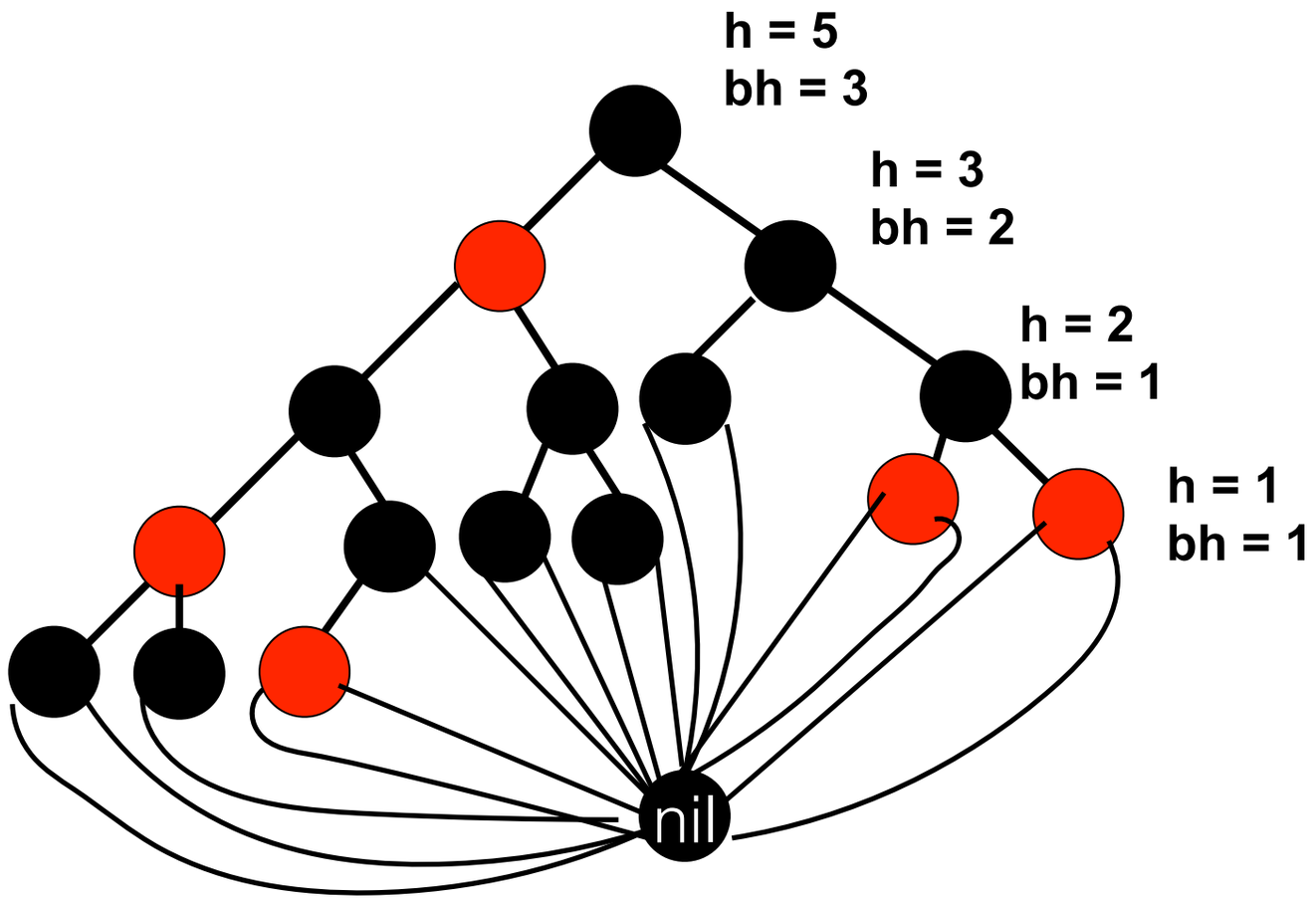
  Follows directly from the conversion of the root of the 2-4 tree
- **Property 3**: Every leaf (T.nil) is black
- **Property 4**: Every red node has black children

  Every time we created a red node we created a black parent
- **Property 5**: For every node, all paths from that node down to a leaf (nil) node contain the same number of black nodes

For every node we converted, we produced one black node which had between 0 and 2 red nodes as its children (depending on if the 2-4 node was a 2-node, 3-node, or 4-node).

A 2-4 tree (and its subtrees) are perfectly balanced, thus the number of black nodes on any path from any node down to a leaf node encounters the same number of black nodes.

Red-Black Tree

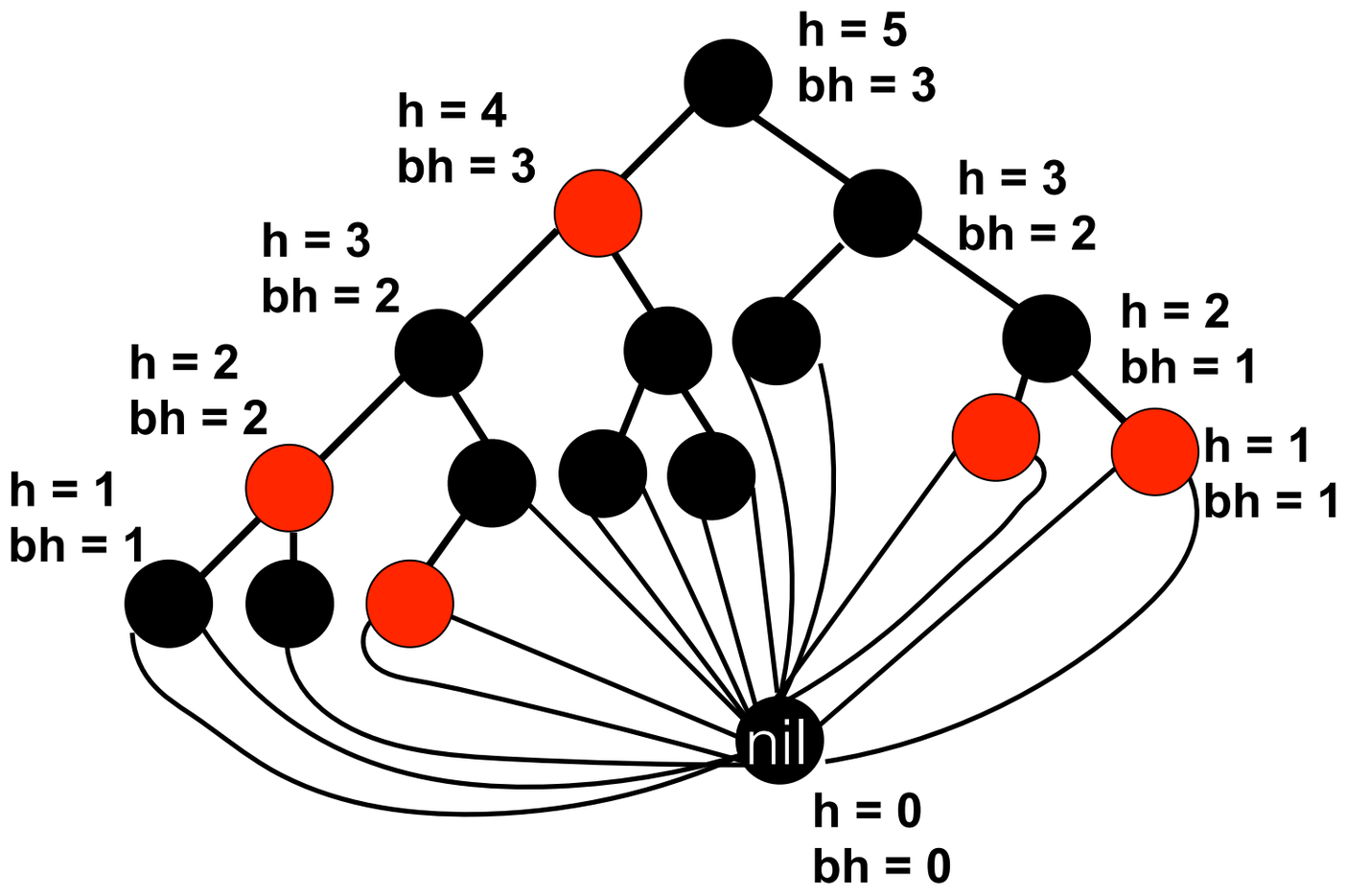


Height of a node: # edges in a longest path to a leaf.

Black Height of a node of a node x: bh(x) is # black nodes encountered on a path to a leaf (nil) node, not including the node itself. By the property that for any node, all paths from this node to a leaf contain the same number of black nodes - the black height is well defined

When drawing a tree, we will omit the leaf (nil) nodes

Red-Black Tree

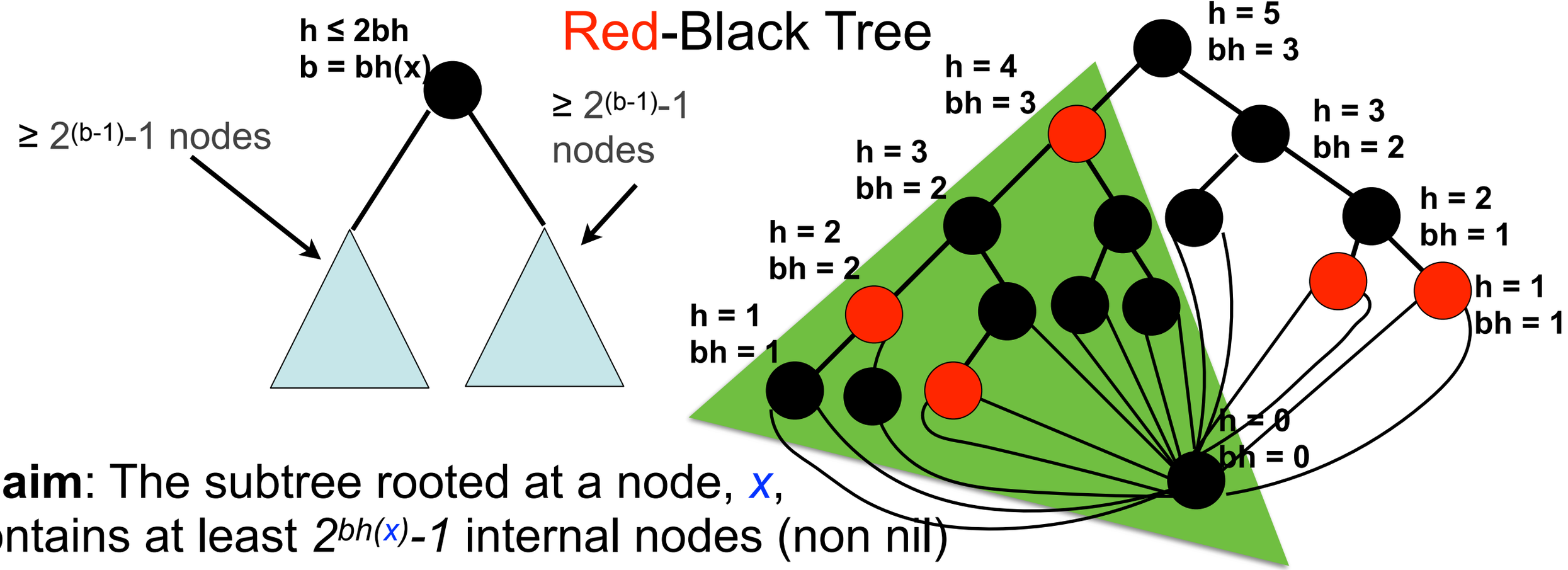


Claim: A node with height *h* has black-height at least *h/2*.

h ≤ 2*black-height

Proof: By the property that the children of a red node are black we know that at least *1/2* the nodes on the path from the node to a leaf are black. Thus the number of black nodes is at least *h/2*.

Red-Black Tree



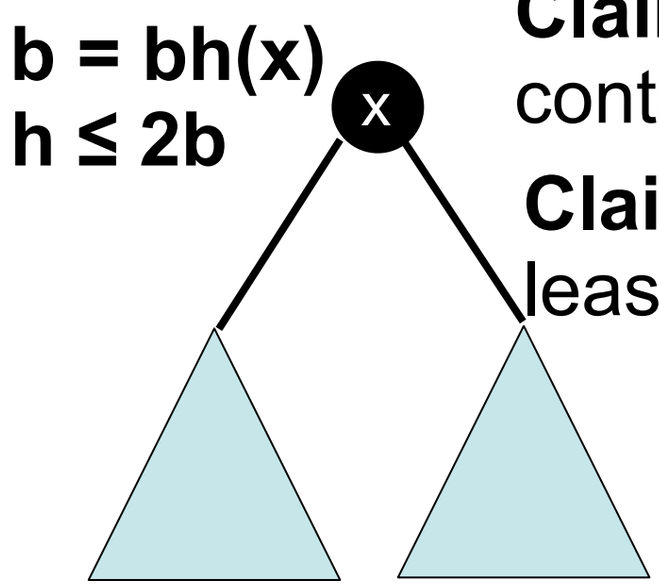
Claim: The subtree rooted at a node, *x*, contains at least $2^{bh(x)}-1$ internal nodes (non nil)

Proof: By induction on the height of *x*.

Basis: If *x* has height *0*, it is a leaf (nil node) and *bh(x)=0.* The subtree rooted at *x* has *0* internal nodes. $2^0 - 1 = 0$

Inductive hypothesis: A node y of height at most h-1 contains at least $2^{bh(y)}-1$ internal nodes

Inductive step: If x has height h and **b**=bh(x), then its children have height at most h-1.
If the child is red it has black-height **b**=bh(x), otherwise it has black-height **b-1=**bh(x)-1. By the inductive hypothesis each child has at least $2^{bh(x)-1}-1$ internal nodes. Thus the subtree rooted at x contains at least $2(2^{bh(x)-1}-1)+1 = 2^{bh(x)} - 1$ internal nodes



Claim: The subtree rooted at a node, *x*, contains at least $2^{bh(x)}-1$ internal nodes (non nil)

Claim: A node with height *h* has black-height at least *h/2*.

Lemma: A red-black tree with n internal nodes has height at most 2lg(n+1)

Proof: Let **h** be the height, and **b**=bh(x) be the black height of the root node = x.

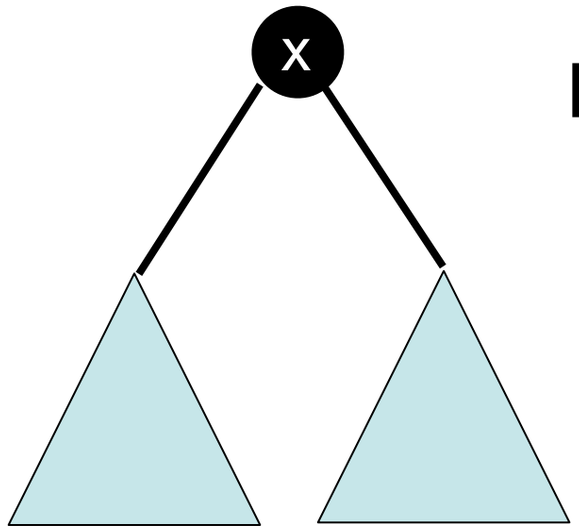
By the previous two claims $n \geq 2^{b} - 1 \geq 2^{h/2} - 1$.

Thus $n + 1 \geq 2^{b} \geq 2^{h/2}$ ➜ $\lg(n + 1) \geq h/2$

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

What is the worst case time to search for an item in a red-black tree?

$O(1)$	**160247**
$O(\log n)$	**160712**
$O(n)$	**160826**
$O(n \log n)$	**161205**
None of the above	**161207**



h ≤ 2log(n+1)

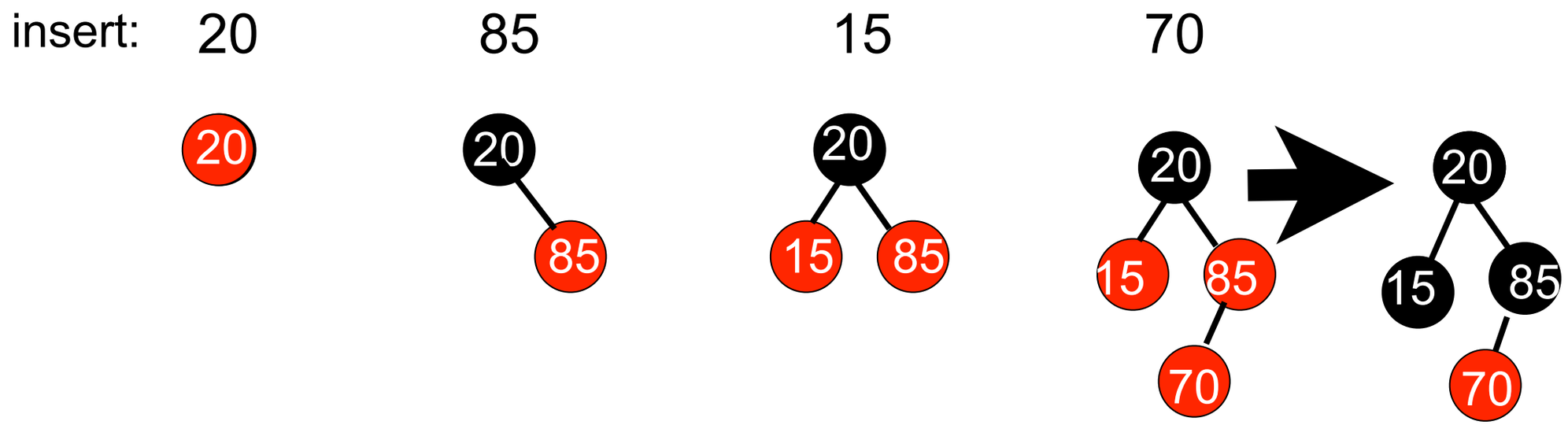
Corollary: On a red-black tree with n nodes, we can implement dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in worst case O(log n) time

Insert and Delete: Modify the Tree

- insert adds a node, delete removes a node (we won't discuss delete - see book pages 323-329)
- Modifications:
  - color changes
  - links in the tree are changed by "rotations"

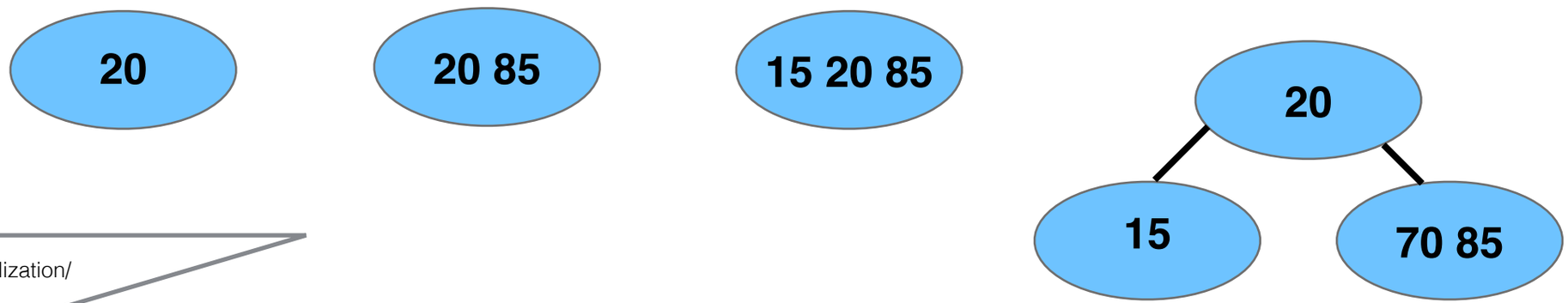
Insertion into a Red-Black Tree

Idea: Insert k into the tree similar to how you would a BST (except we use T.nil instead of NIL). Color the node red. A red-red violation may have occurred. Move the violation up the tree till it can be fixed by rotations and recoloring



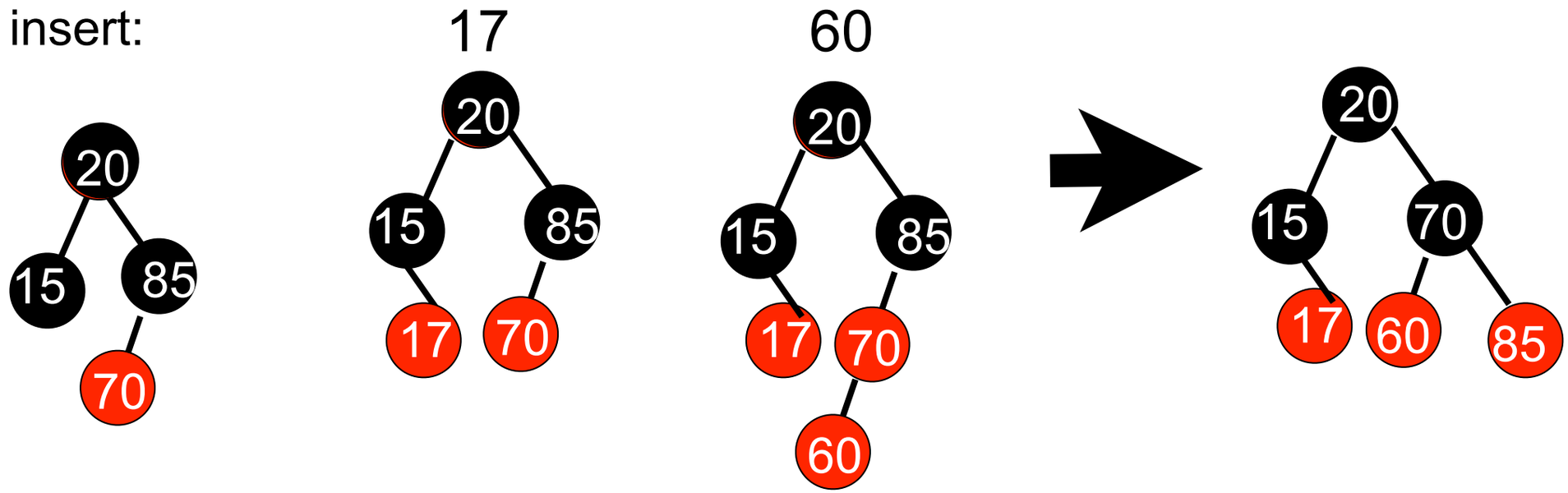
Double Red Problem
case 1 sibling of parent is red

2-4 Tree

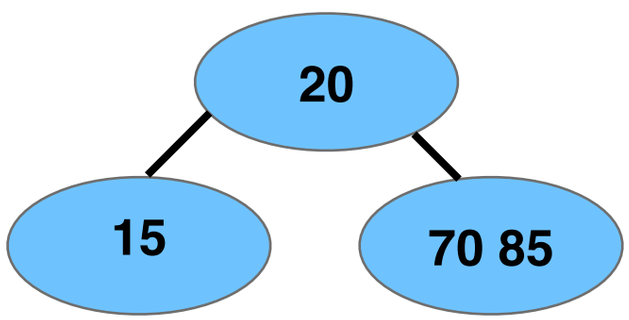


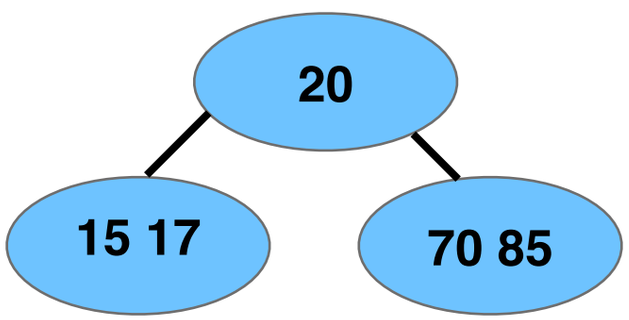
Visualization:
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Insertion into a Red-Black Tree



2-4 Tree

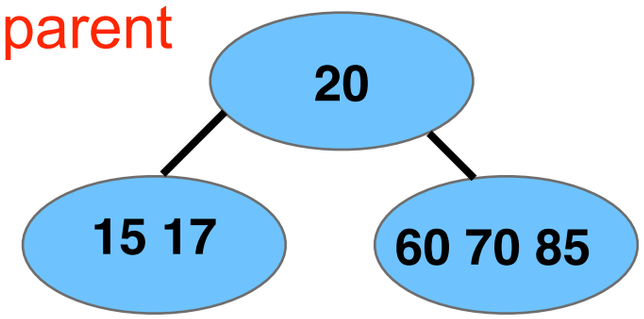




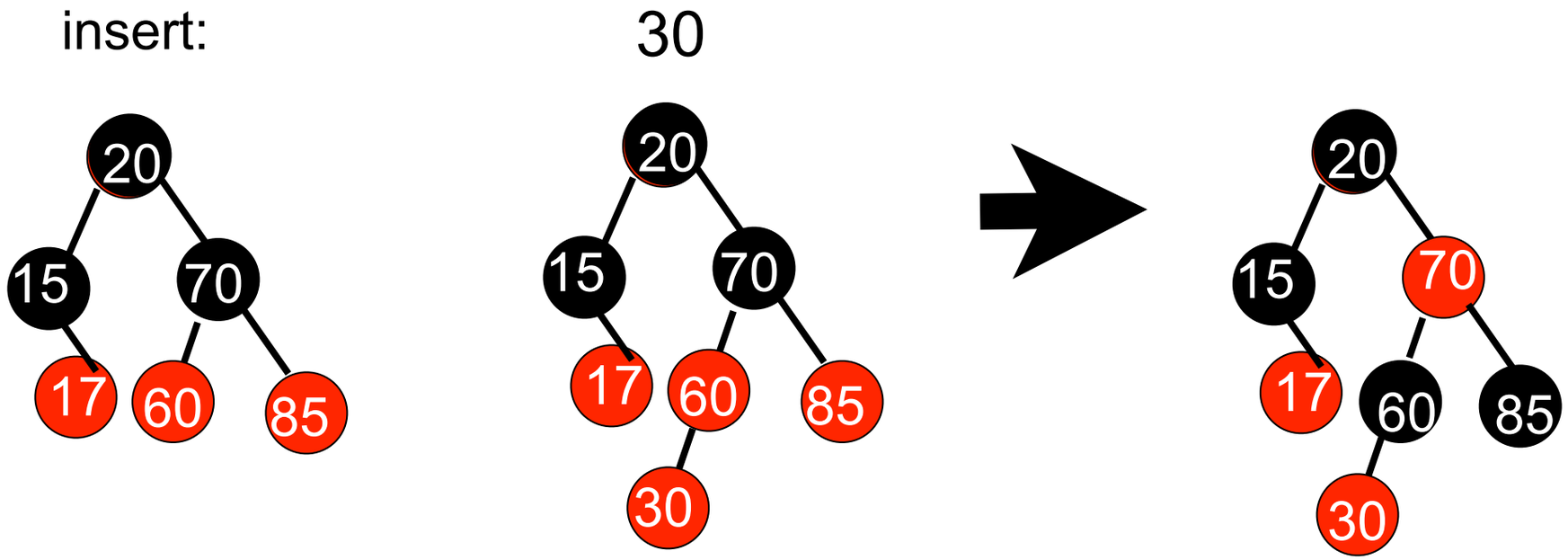
Double Red Problem

case 3 sibling of parent is BLACK (or nil)

Right Rotate

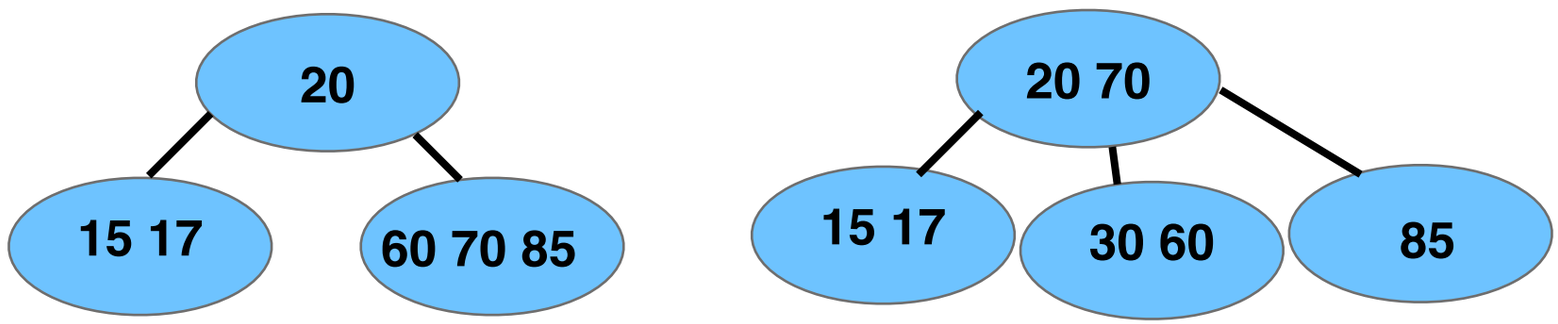


Insertion into a Red-Black Tree

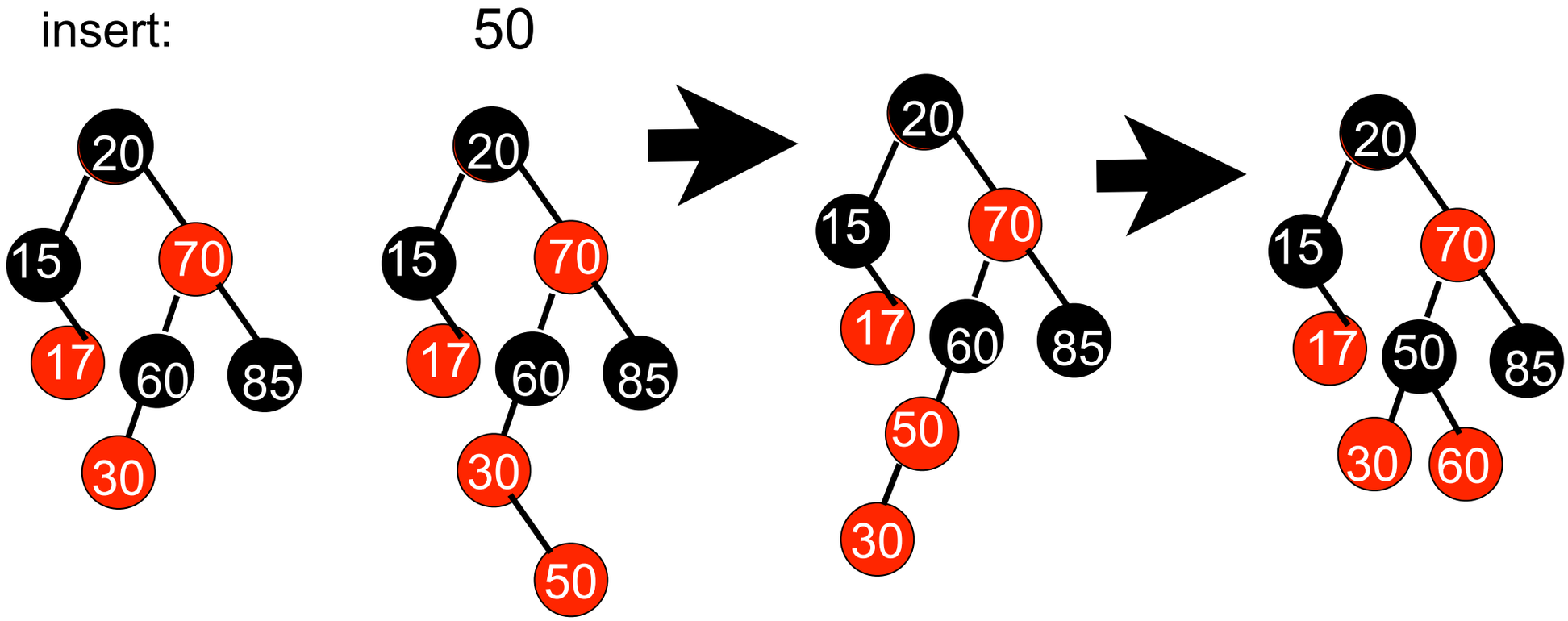


Double Red Problem
case 1 sibling of parent
is RED

2-4 Tree



Insertion into a Red-Black Tree

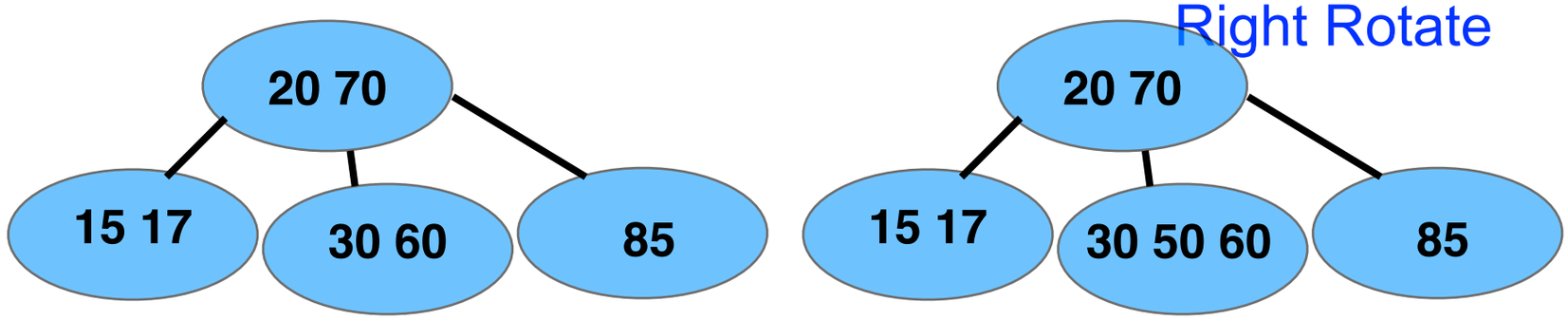


Double Red Problem
case 2 sibling of parent
is BLACK (or nil)

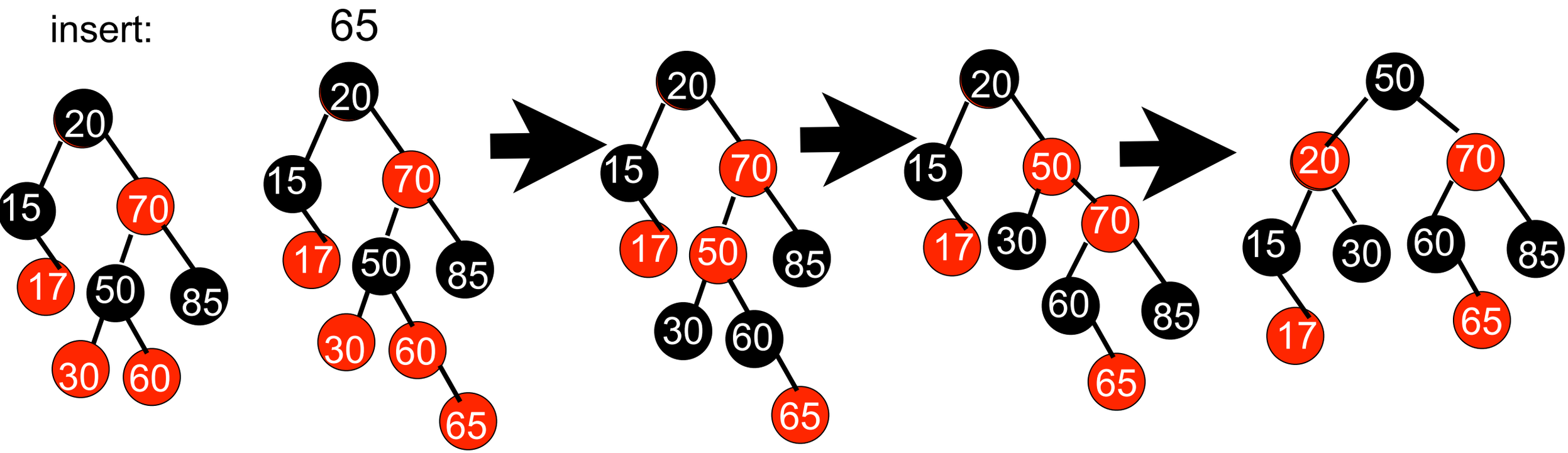
Double Red Problem
case 3 sibling of parent
is BLACK (or nil)

Right Rotate

2-4 Tree



Insertion into a Red-Black Tree

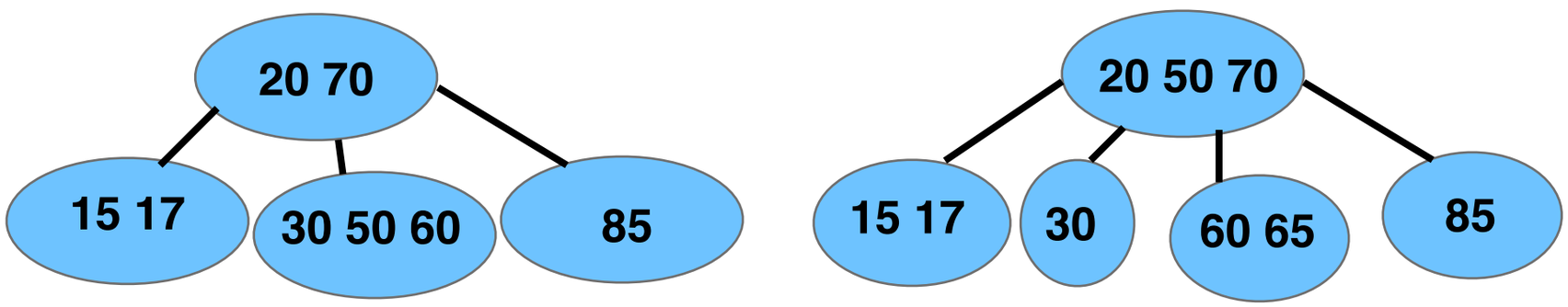


Double Red Problem
case 1 sibling of parent is RED

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

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

2-4 Tree



OPERATIONS TO FIX A RED-BLACK TREE

ROTATIONS, left and right:

- used to maintain the red-black trees as a balanced binary search tree
- change only local pointers
- maintains the binary search tree property

COLOR CHANGES

Rotate Node

LEFT-ROTATE assumes:

x.right ≠ T.nil

root's parent is T.*nil*

LEFT-ROTATE(T, x)

y = x.*right*

Update pointers to and from left child of y

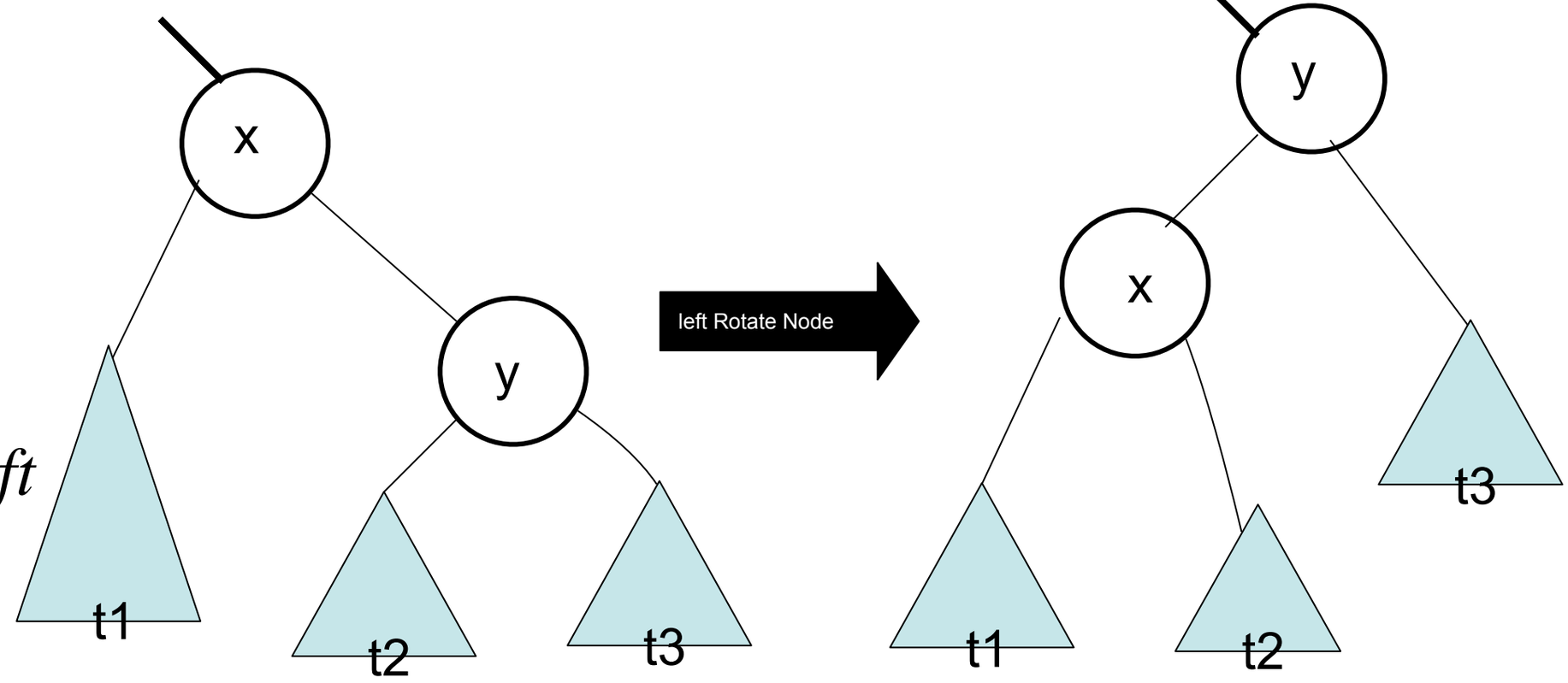
x.*right* = y.*left*
if y.*left* ≠ T.*nil*
  y.*left*.*p* = x

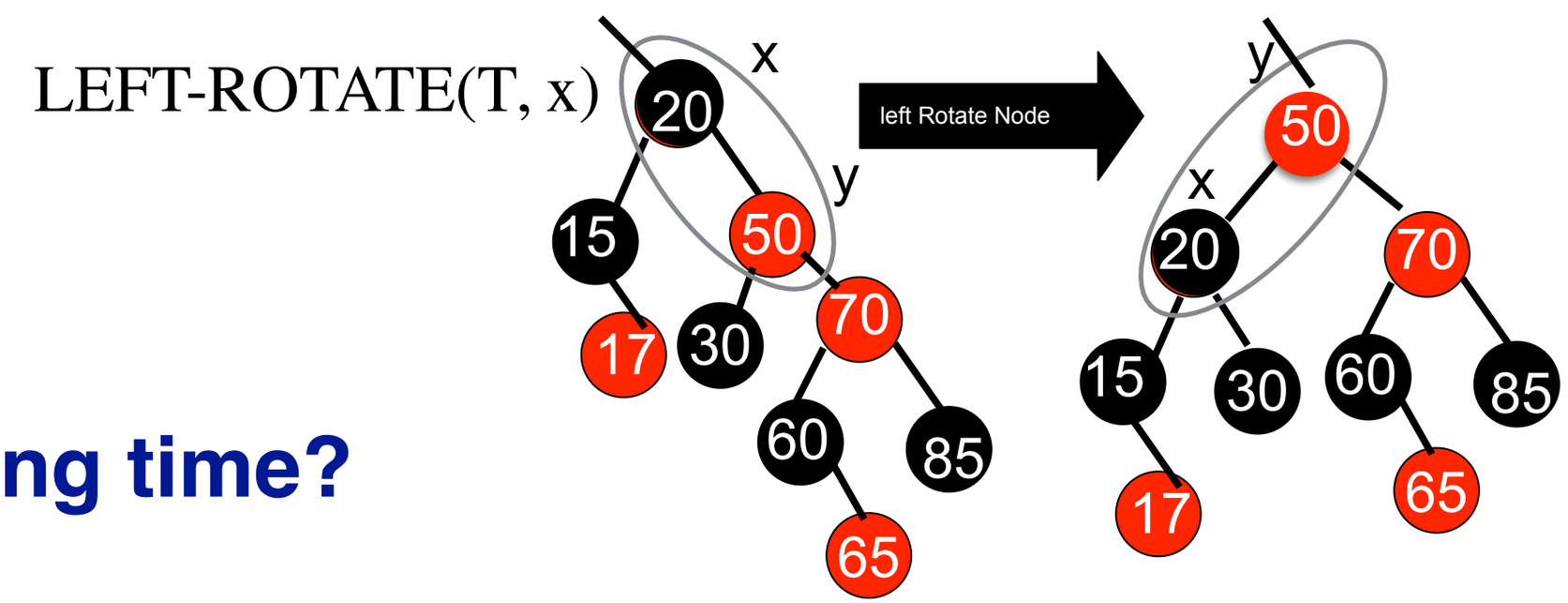
Attach y to its parent

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

Attach x to y

y.*left* = x
x.*p* = *y*





No color changes

Running time?

Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

Insertion

RB-INSERT(T, z)

y = T.nil // *y is the parent of x*

x =T.root

Find location of new node

while x ≠ *T.nil*

  y=x

  if *z.key* < *x.key*

    x = x.left

  else *x = x.right*

Attach z to parent and to z

z.p = y

if *y == T.nil*

  T.root = z //*tree T was empty*

else if *z.key* < *y.key*

  y.left = z

else *y.right = z*

Attach z to nil node and add color to z

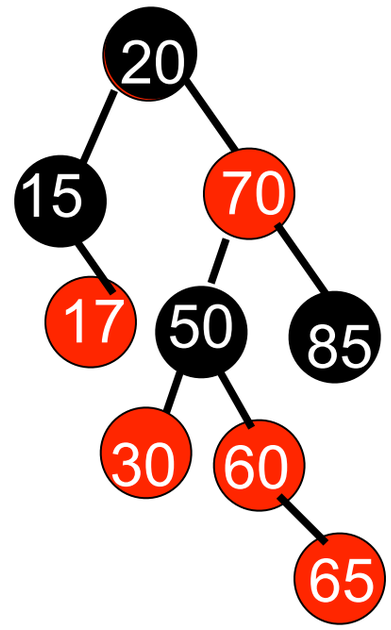
z.left = T.*nil*

z.*right* = T.*nil*

z.color = RED

RB-INSERT-FIXUP(T,z)

// *additional code needed*

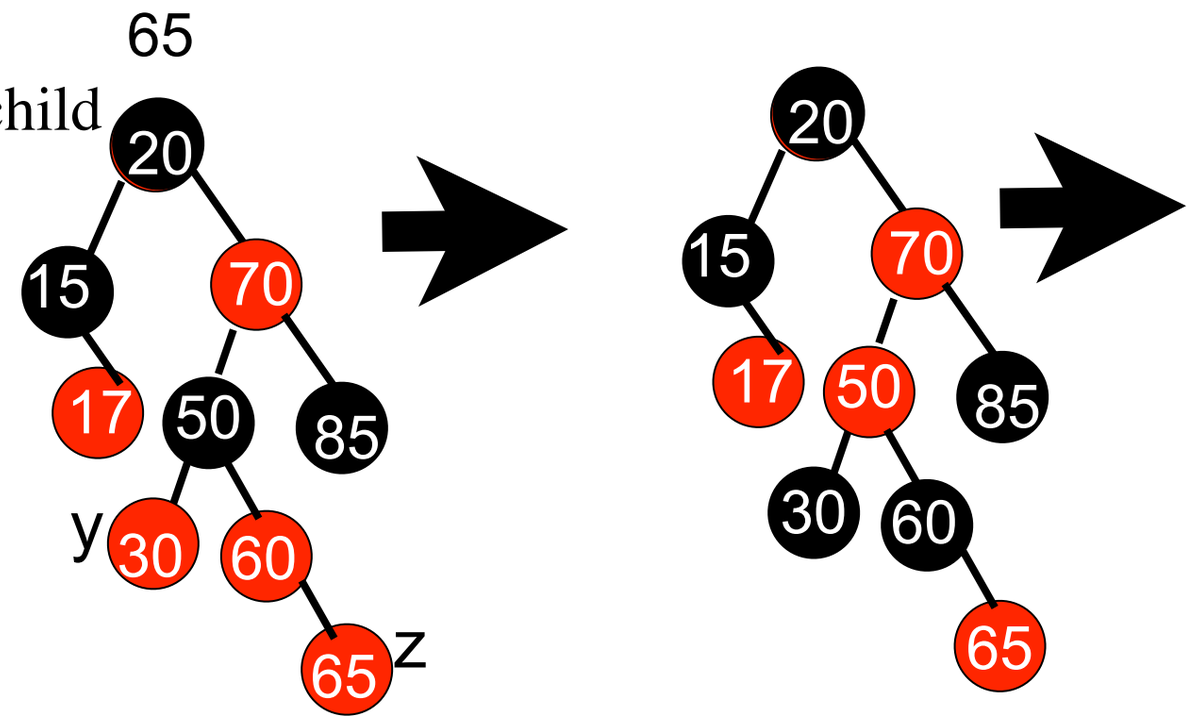


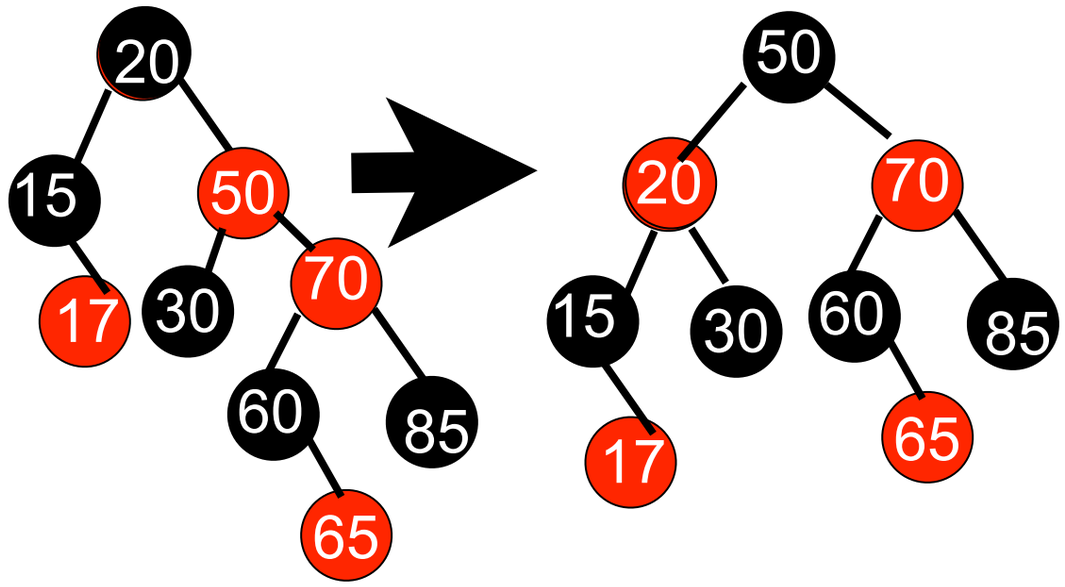
Running time?

Fixing the violations

```
RB-INSERT-FIXUP (T,z)
  while z.p.color == RED
    if z.p == z.p.p.right   // parent is a right child
      y = z.p.p.left     // parent's sibling
      if y.color == RED       //  case 1
        z.p.color = BLACK
        y.color = BLACK
        z.p.p.color = RED
        z = z.p.p
      else                    //  case 2 or case 3
        if  z == z.p.left    // z is a left child,  case 2
          z = z.p
          RIGHT-ROTATE(T, z)
        z.p.color = BLACK
        z.p.p.color = RED                          //  case 3
        LEFT-ROTATE(T, z.p.p)
    else (same as the
T.root.color = BLA
```

The parent become the child after rotation





Running time?

Proof sketch that we maintain the Red-Black tree properties

See pg's 318-321 for a more complete proof

Loop Invariant:

At the start of each iteration of the **while** loop, node z is red.
There is at most one red-black tree violation either:

- z is a red root
- z and z.*p* are both red (red node with red child violation)

Initialization:

We started with a red-black tree before we added the new red node z, If there is a red-red violation, it has to involve z and its parent, or z is the root and needs to be colored black

Termination:

The loop terminates because z.*p* is black. Thus, there is no red-red violation. We do not know the value of T.root.color The last line of the function assigns the root to be black. Thus when the function ends there is no violation

Maintenance:

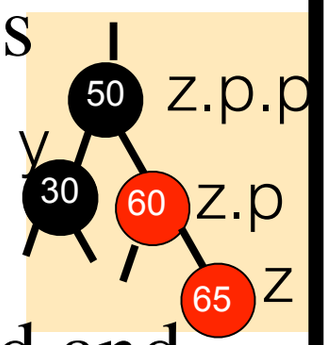
We stop when z's parent is black (by definition, the root's parents' color is defined to be black), thus if we enter the loop, then z and its parent are both red (and no other violation occurs in the red-black tree).

There are 6 cases to consider. 3 of the cases are symmetric, so WLOG we only consider the 3 cases where z.p is the right child of z.p.p

Maintenance cont:

WLOG we only consider the 3 cases where z.p is the right child of z.p.p

let y = z.p.p.left // y is z's uncle



z.p.p is black, since z and z.p are red and there is only one red-red violation

Loop Invariant: At the start of each iteration of the **while** loop,

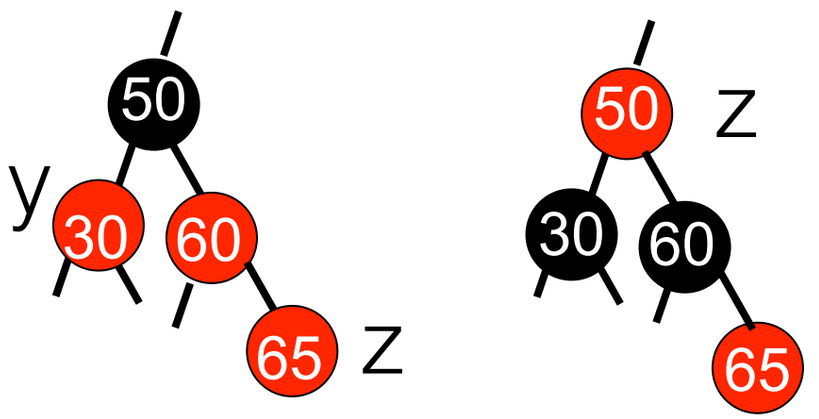
a) node z is red.

b) There is *at most* one red-red violation:

- z is a red root
- z and z.*p* are both red

<u>Case 1</u> y is red and z.p.p is black

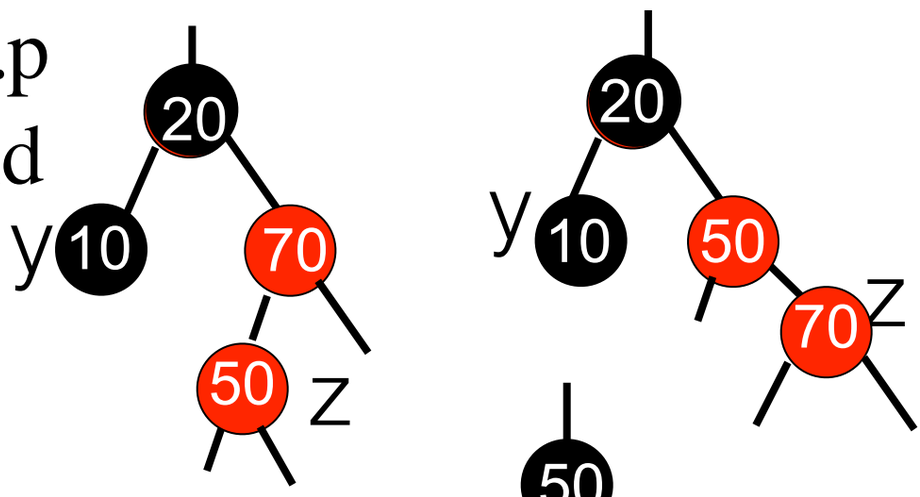
make y and z.p black and z.p.p red. Thus black height property is maintained, but we might have created a red-red violation between z.p.p and its parent. Assign z = z.p.p (now z is red again)



<u>Case 2</u> y is black and z is a left child. Set z = z.p

Right rotate around z. Now z is now a right child

Both z and z.p are red. Only one case 3 red-red violation

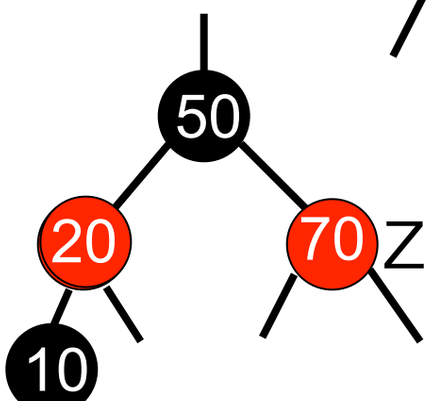


<u>Case 3</u> y is black, z is a right child

make z.p black and z.p.p red

left rotate on z.p.p ☞ no red-red violation

z.p is black ☞ no more iterations



Running time?

Conclusion

Red-Black trees allow us to implement all basic set operations in O(log n) time

At most 2 rotations and possibly some color changes to fix up a red-black tree after insertion