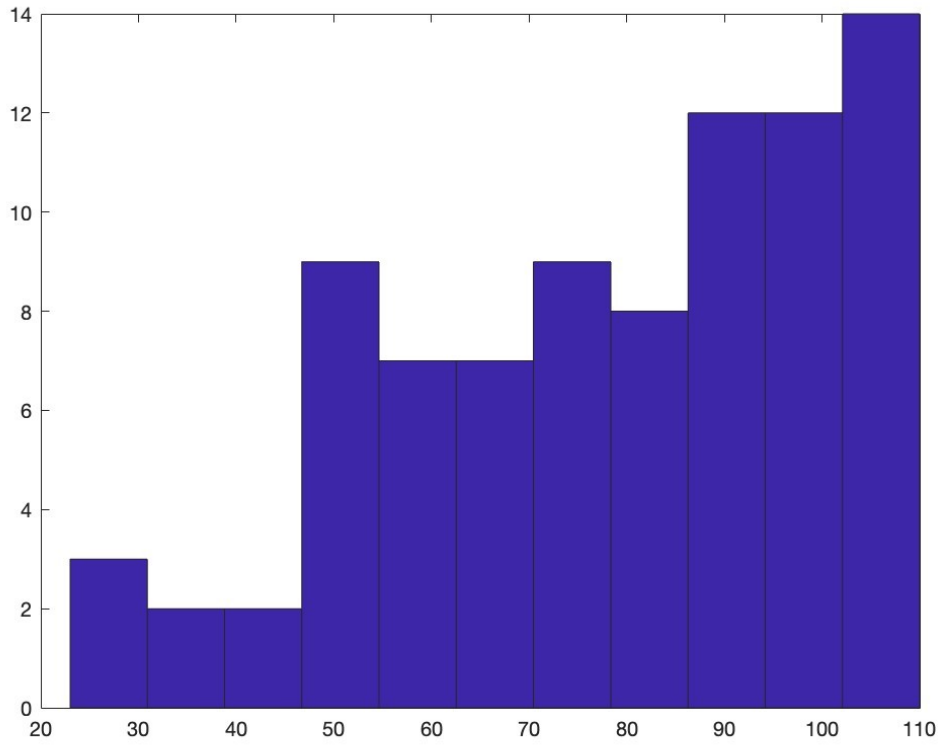
Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

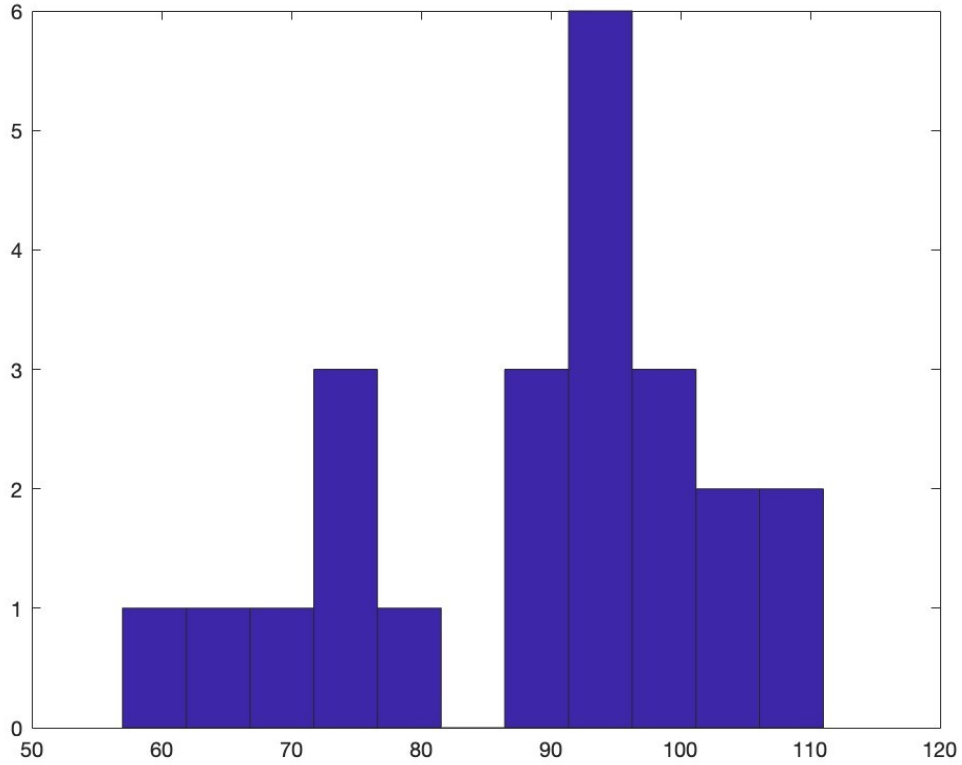
Lecture 13

Midterm Results - CS 477



Min = 23 Max = 110 Average = 78.2

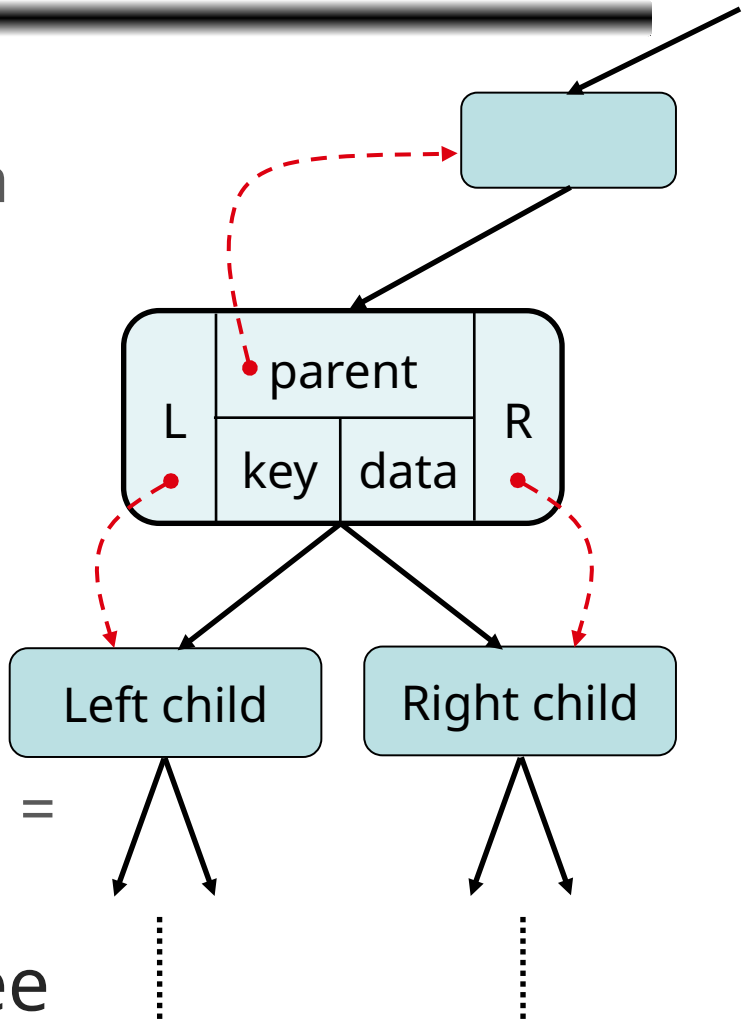
Midterm Results - CS 677



Min = 57 Max = 111 Average = 89.26

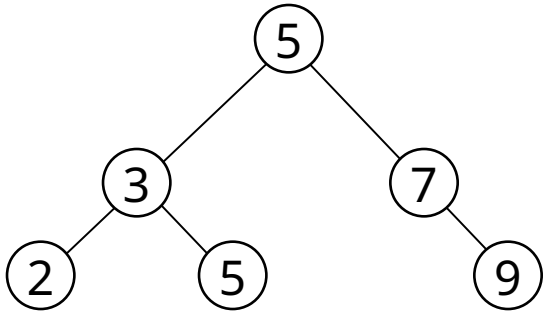
Binary Search Trees

- Tree representation:
  - A linked data structure in which each node is an object
- Node representation:
  - Key field
  - Satellite data
  - Left: pointer to left child
  - Right: pointer to right child
  - p: pointer to parent (p [root [T]] = NIL)
- Satisfies the binary search tree property



Binary Search Tree Example

- Binary search tree property:
  - If y is in left subtree of x,
    then key [y] ≤ key [x]
  - If y is in right subtree of x,
    then key [y] ≥ key [x]



Binary Search Trees

- Support many dynamic set operations
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- Running time of basic operations on binary search trees
  - On average: $\Theta(\lg n)$
    - The expected height of the tree is lgn
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of n nodes

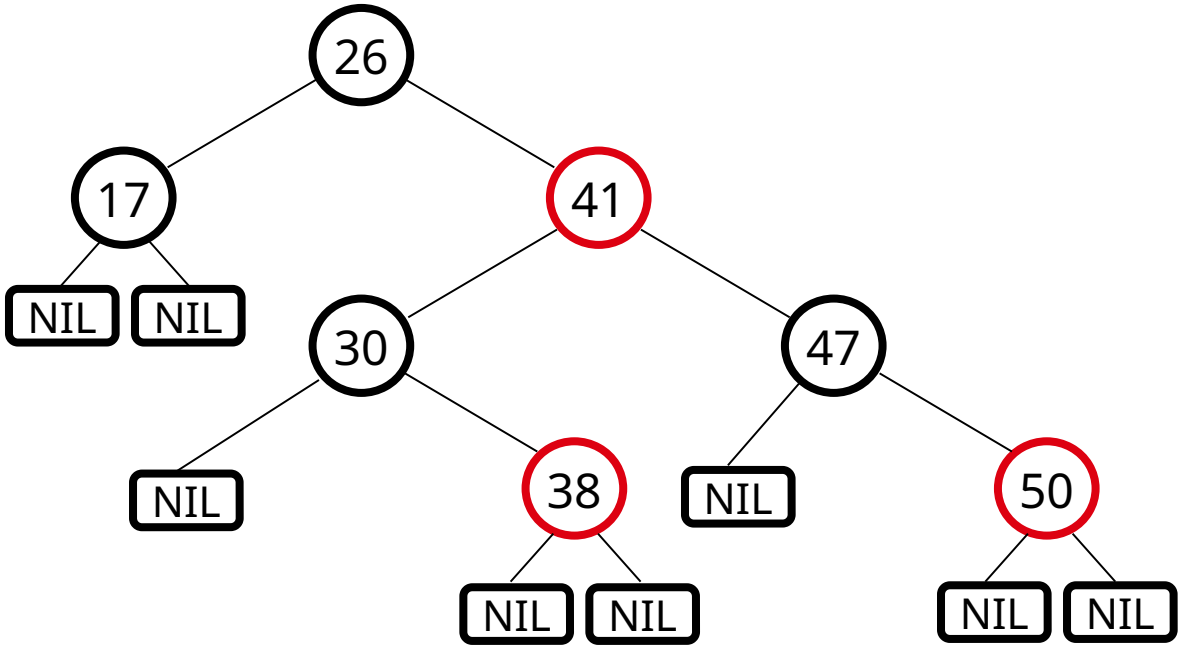
Red-Black Trees

- "Balanced" binary trees guarantee an O(lgn) running time on the basic dynamic-set operations
- Red-black tree
  - Binary tree with an additional attribute for its nodes: color which can be **red** or **black**
  - Constrains the way nodes can be colored on any path from the root to a leaf
    - Ensures that no path is more than twice as long as another ⇒ the tree is balanced
  - The nodes inherit all the other attributes from the binary-search trees: key, left, right, p

Red-Black Trees Properties

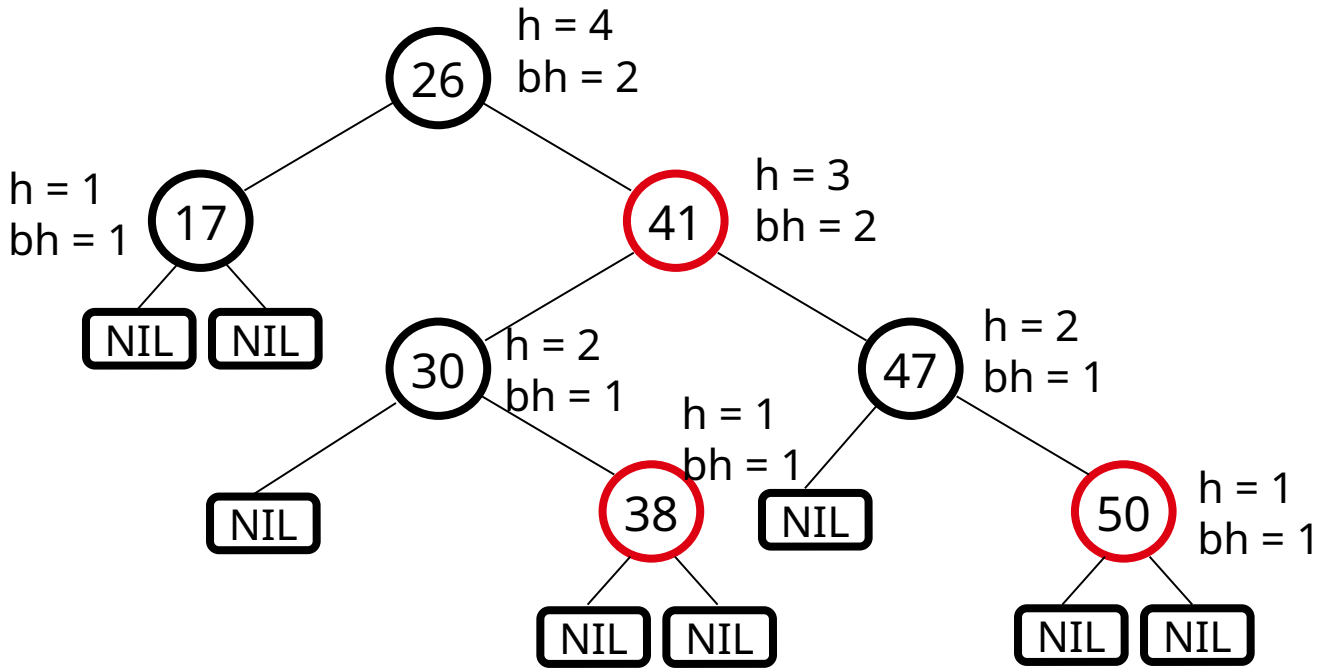
1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is red, then both its children are black
   - No two red nodes in a row on a simple path from the root to a leaf
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Example: RED-BLACK TREE



- For convenience we use a sentinel NIL[T] to represent all the NIL nodes at the leafs
  - NIL[T] has the same fields as an ordinary node
  - Color[NIL[T]] = BLACK
  - The other fields may be set to arbitrary values

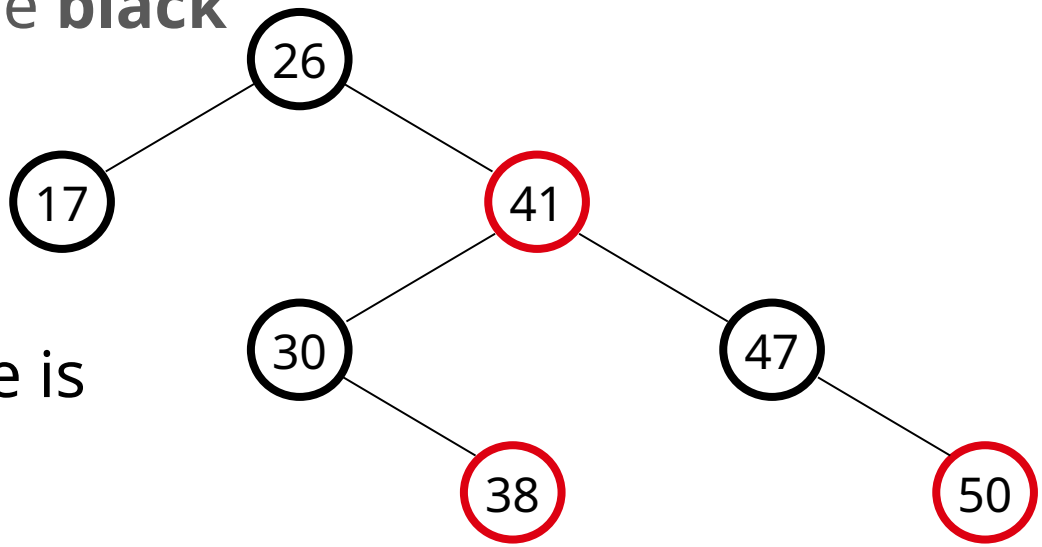
Black-Height of a Node



- **Height of a node:** the number of edges in a longest path to a leaf
- **Black-height** of a node x: bh(x) is the number of black nodes (including NIL) on a path from x to leaf, not counting x

Properties of Red-Black Trees

- **Claim**
  - Any node with height h has black-height ≥ h/2
- **Proof**
  - By property 4, there are at most h/2 **red** nodes on the path from the node to a leaf
  - Hence at least h/2 are **black**



Property 4: if a node is red then both its children are black

Properties of Red-Black Trees

Claim

The subtree rooted at any node x contains at least

$2^{bh(x)} - 1$ internal nodes

Proof: By induction on height of x

Basis: height[x] = 0 ⇒

x is a leaf (NIL[T]) ⇒

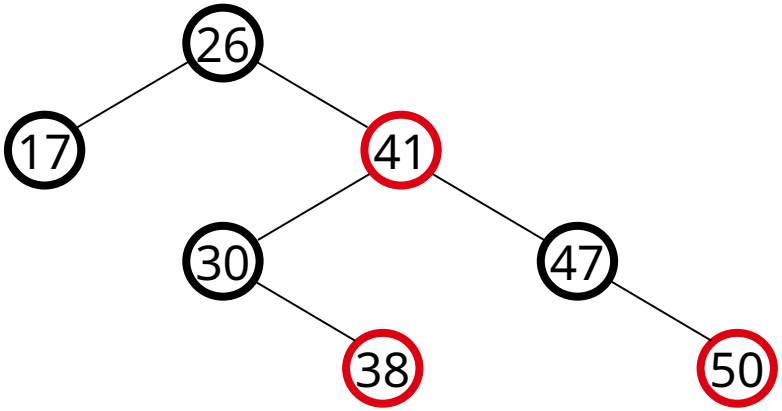
bh(x) = 0 ⇒

# of internal nodes: $2^0 - 1 = 0$

Properties of Red-Black Trees

Inductive step:

- Let height(x) = h and bh(x) = b
- Any child y of x has:
  - bh (y) = b (if the child is **red**), or
  - bh (y) = b - 1 (if the child is **black**)



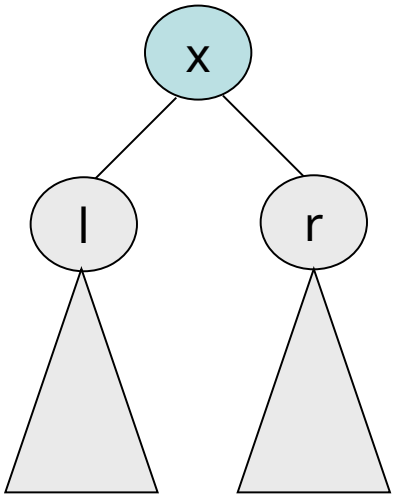
Properties of Red-Black Trees

- Want to prove:
  - The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes
- Assume true for children of x:
  - Their subtrees contain at least $2^{bh(x) - 1} - 1$ internal nodes
- The subtree rooted at x contains at least:

  $(2^{bh(x) - 1} - 1) + (2^{bh(x) - 1} - 1) + 1 =$

  $2 \cdot (2^{bh(x) - 1} - 1) + 1 =$

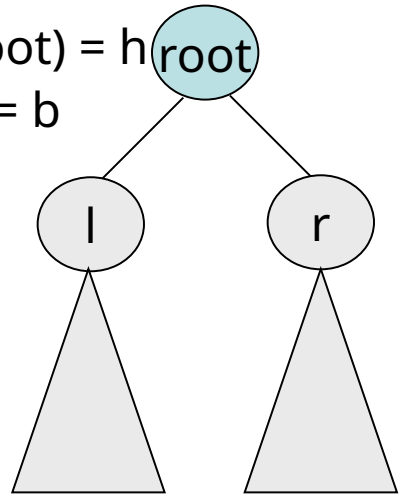
  $2^{bh(x)} - 1$ internal nodes



Properties of Red-Black Trees

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

Proof:



$$n \geq 2^b - 1 \geq 2^{h/2} - 1$$

number n of internal nodes

since b ≥ h/2

- Add 1 to all sides and then take logs:

$$n + 1 \geq 2^b \geq 2^{h/2}$$

$$\lg(n + 1) \geq h/2 \Rightarrow$$

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

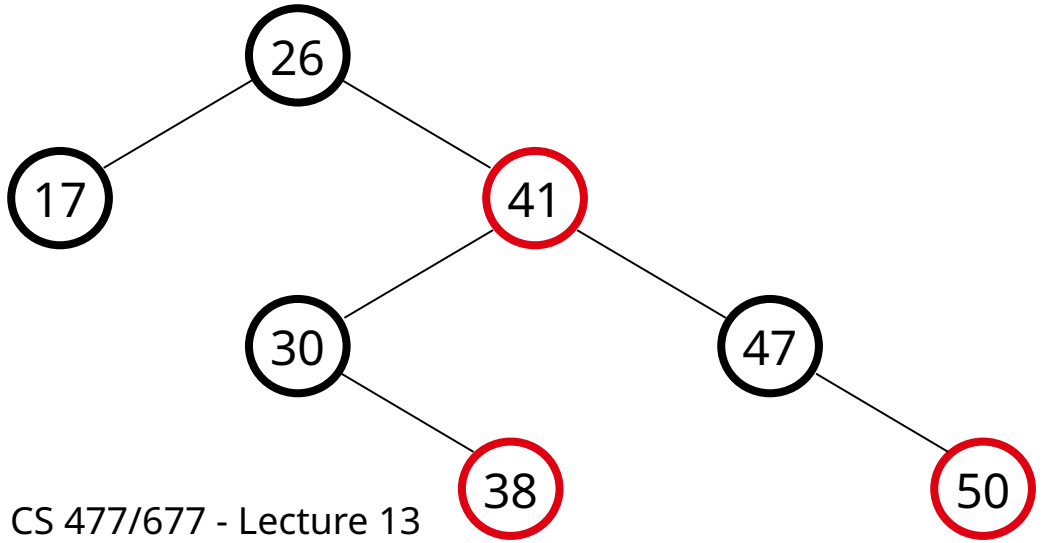
Operations on Red-Black Trees

- The non-modifying binary-search tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in O(h) time
  - They take O(lgn) time on red-black trees
- What about TREE-INSERT and TREE-DELETE?
  - They will still run in O(lgn)
  - We have to guarantee that the modified tree will still be a red-black tree

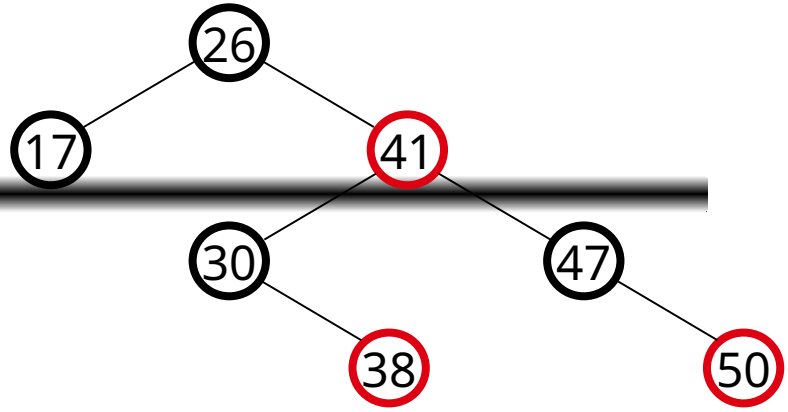
INSERT

INSERT: what color to make the new node?

- Red? Let's insert 35!
  - Property 4: if a node is red, then both its children are black
- Black? Let's insert 14!
  - Property 5: all paths from a node to its leaves contain the same number of black nodes



DELETE



DELETE: what color was the node that was removed? **Red?**

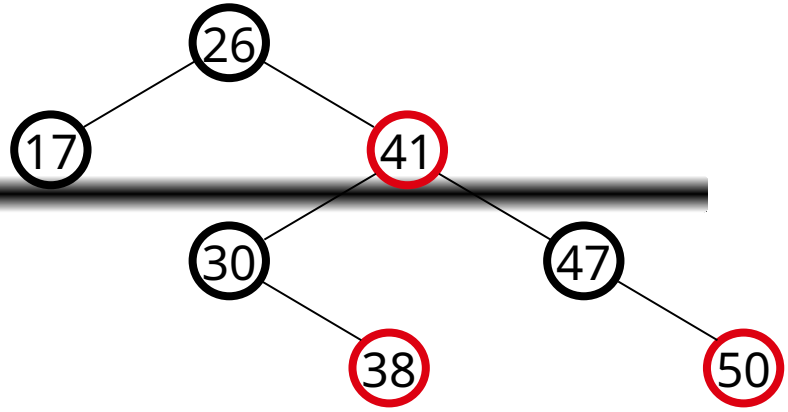
1. Every node is either **red** or **black** OK!
2. The root is **black** OK!
3. Every leaf (NIL) is **black** OK!
4. If a node is red, then both its children are black

OK! Does not change any black heights

OK! Does not create two red nodes in a row

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

DELETE



DELETE: what color was the node that was removed? **Black?**

1. Every node is either **red** or **black** OK!
2. The root is **black** Not OK! If removing the root and the child that replaces it is **red**
3. Every leaf (NIL) is **black** OK!
4. If a node is red, then both its children are black

Not OK! Could change the black heights of some nodes

Not OK! Could create two red nodes in a row

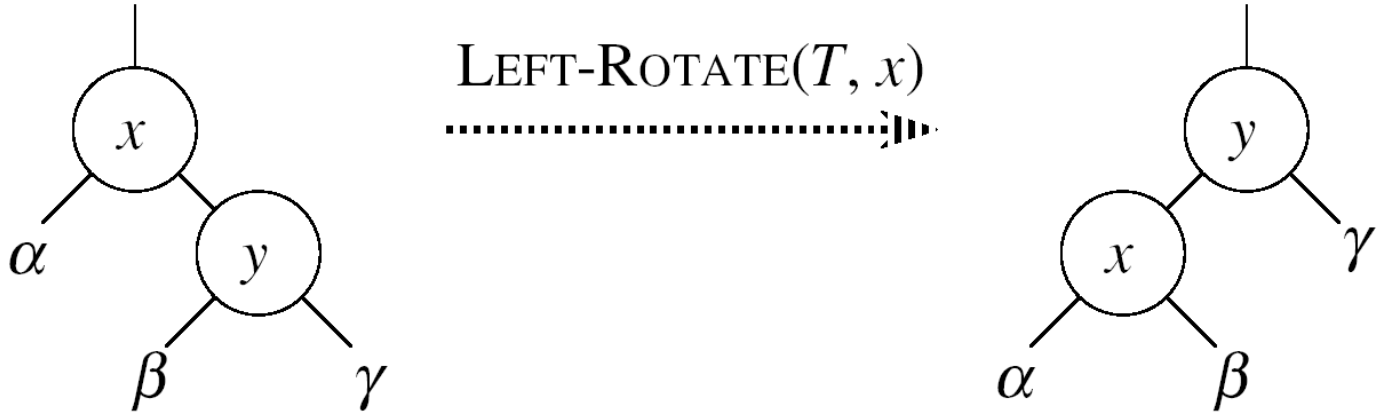
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Rotations

- Operations for restructuring the tree after insert and delete operations on red-black trees
- Rotations take a red-black tree and a node within the tree and:
  - Together with some node re-coloring they help restore the red-black tree property
  - Change some of the pointer structure
  - Do not change the binary-search tree property
- Two types of rotations:
  - Left & right rotations

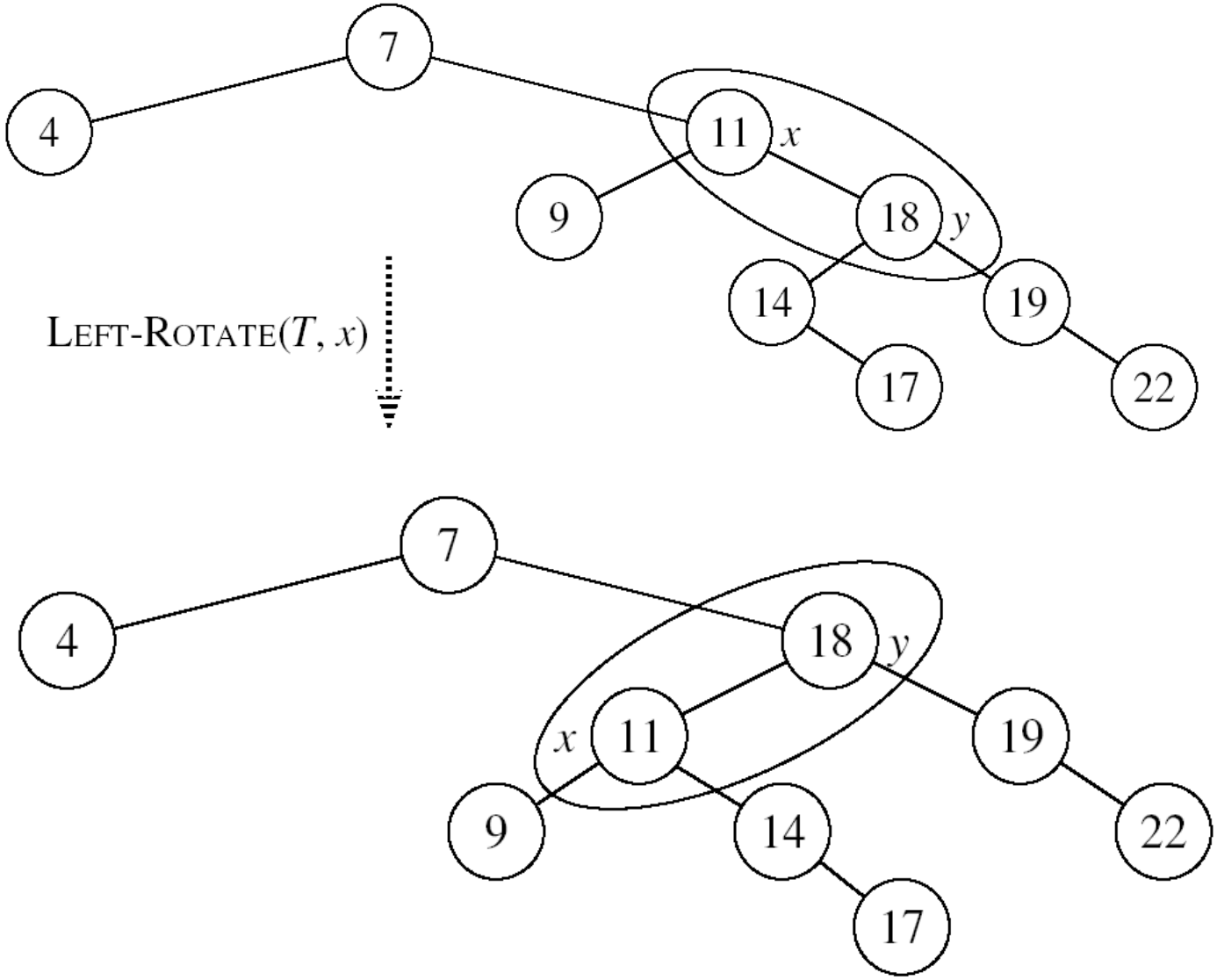
Left Rotations

- Assumption for a left rotation on a node x:
  - The right child of x (y) is not NIL



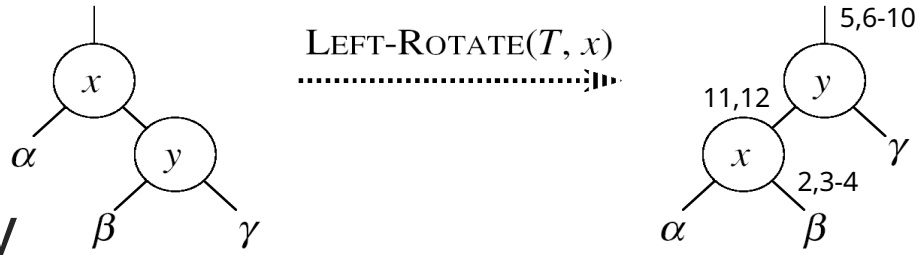
- Idea:
  - Pivots around the link from x to y
  - Makes y the new root of the subtree
  - x becomes y's left child
  - y's left child becomes x's right child

Example: LEFT-ROTATE



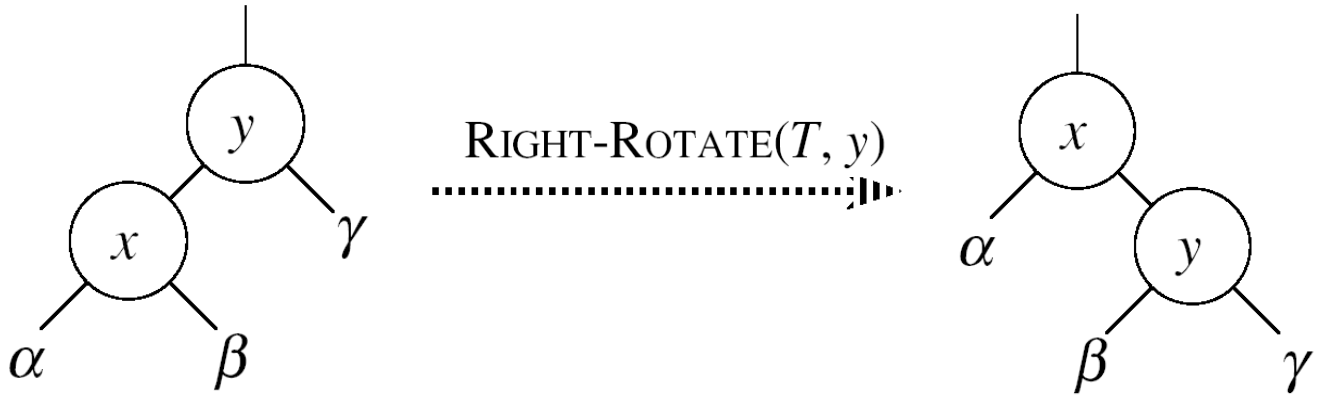
LEFT-ROTATE(T, x)

1. y ← right[x] ►Set y
2. right[x] ← left[y] ► y's left subtree becomes x's right subtree
3. **if** left[y] ≠ NIL
4. **then** p[left[y]] ← x ► Set the parent relation from left[y] to x
5. p[y] ← p[x] ► The parent of x becomes the parent of y
6. **if** p[x] = NIL
7. **then** root[T] ← y
8. **else if** x = left[p[x]]
9. **then** left[p[x]] ← y
10. **else** right[p[x]] ← y
11. left[y] ← x ► Put x on y's left
12. p[x] ← y ► y becomes x's parent



Right Rotations

- Assumption for a right rotation on a node x:
  - The left child of y (x) is not NIL

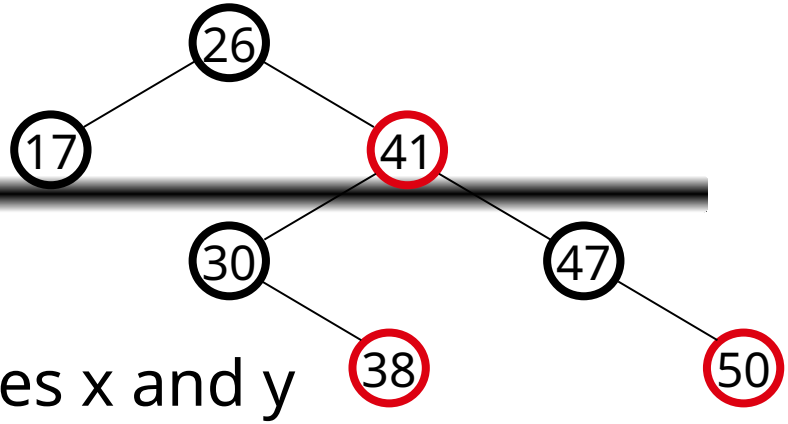


- Idea:
  - Pivots around the link from y to x
  - Makes x the new root of the subtree
  - y becomes x's right child
  - x's right child becomes y's left child

Insertion

- Goal:
  - Insert a new node z into a red-black tree
- Idea:
  - Insert node z into the tree as for an ordinary binary search tree
  - Color the node **red**
  - Restore the red-black tree properties
    - Use an auxiliary procedure RB-INSERT-FIXUP

RB-INSERT(T, z)



1. y ← NIL
2. x ← root[T]

- Initialize nodes x and y
- Throughout the algorithm y points to the parent of x

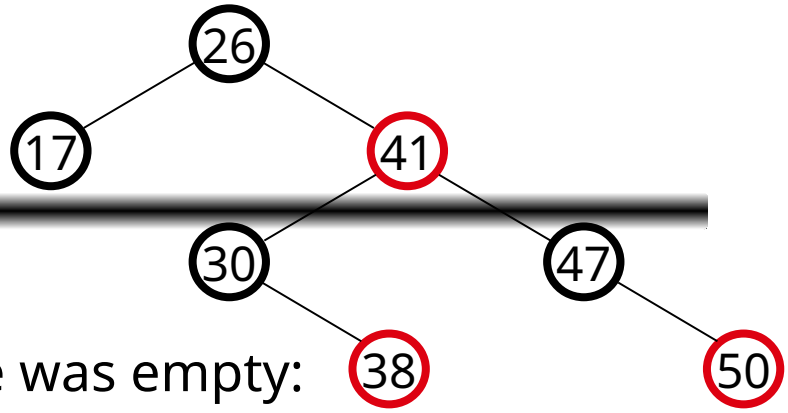
3. **while** x ≠ NIL
4. **do** y ← x
5. **if** key[z] < key[x]
6. **then** x ← left[x]
7. **else** x ← right[x]

- Go down the tree until reaching a leaf
- At that point y is the parent of the node to be inserted

8. p[z] ← y

- Sets the parent of z to be y

RB-INSERT(T, z)




```
9.  if y = NIL
10.     then root[T] ← z
```

The tree was empty: set the new node to be the root

```
11.     else if key[z] < key[y]
12.             then left[y] ← z
13.             else right[y] ← z
```

Otherwise, set z to be the left or right child of y, depending on whether the inserted node is smaller or larger than y's key

```
14. left[z] ← NIL
15. right[z] ← NIL
16. color[z] ← RED
```

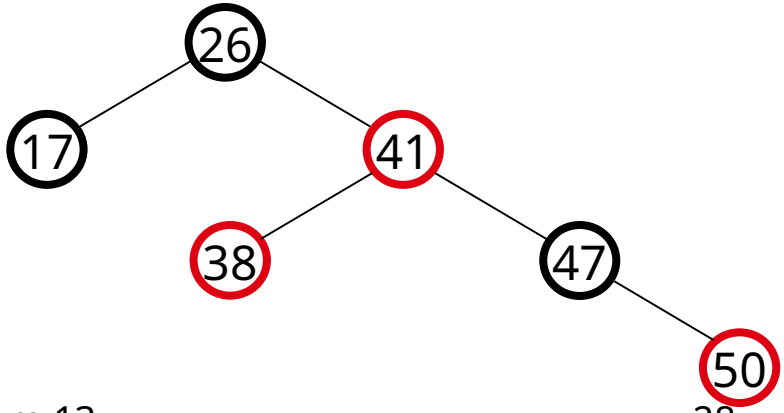
Set the fields of the newly added node

```
17. RB-INSERT-FIXUP(T, z)
```

Fix any inconsistencies that could have been introduced by adding this new red node

RB Properties Affected by Insert

1. Every node is either **red** or **black** — OK!
2. The root is **black** — If z is the root ⇒ not OK
3. Every leaf (NIL) is **black** — OK!
4. If a node is red, then both its children are black — If p(z) is red ⇒ not OK (z and p(z) are both red)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes — OK!

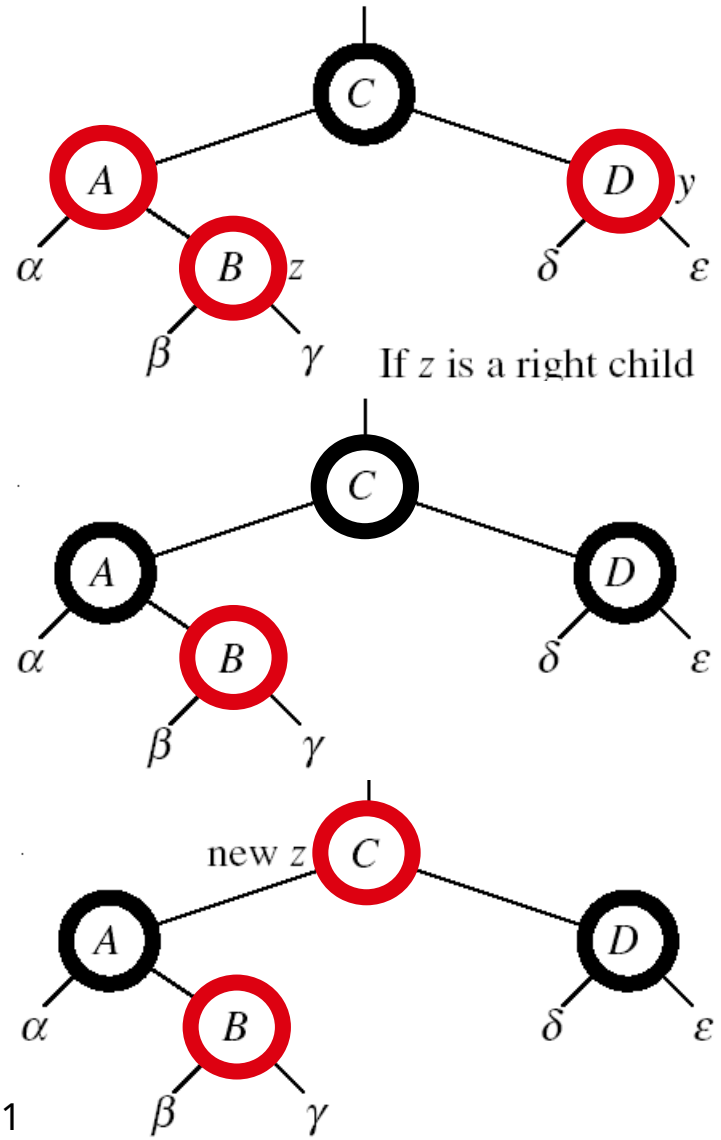


RB-INSERT-FIXUP – Case 1

z's "uncle" (y) is **red**

Idea: (z is a right child)

- p[p[z]] (z's grandparent) must be black: z and p[z] are both red
- Color p[z] **black**
- Color y **black**
- Color p[p[z]] **red**
  - Push the **red** node up the tree
- Make z = p[p[z]]



RB-INSERT-FIXUP – Case 1

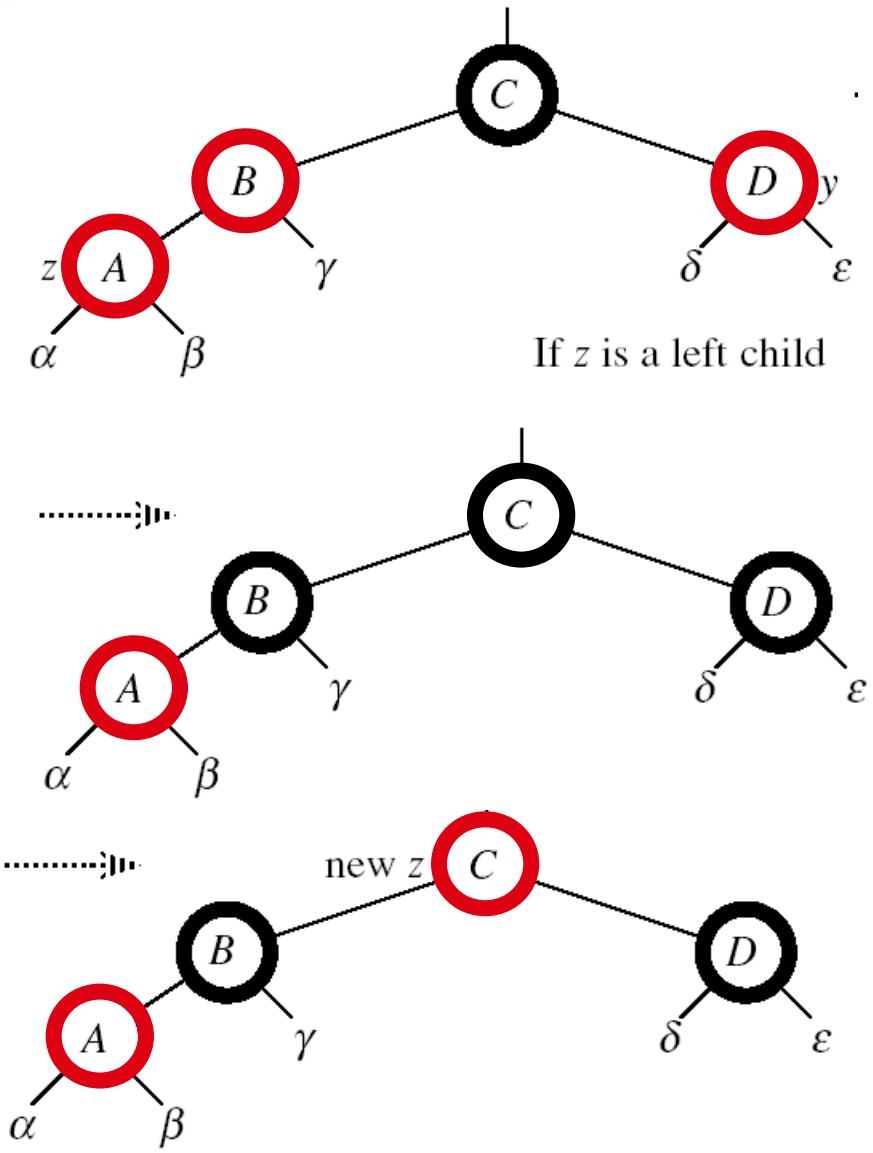
z's "uncle" (y) is **red**

Idea: (z is a left child)

- p[p[z]] (z's grandparent) must be black: z and p[z] are both red
- color[p[z]] ← **black**
- color[y] ← **black**
- color p[p[z]] ← **red**
- z = p[p[z]]

Case1

  - Push the **red** node up the tree



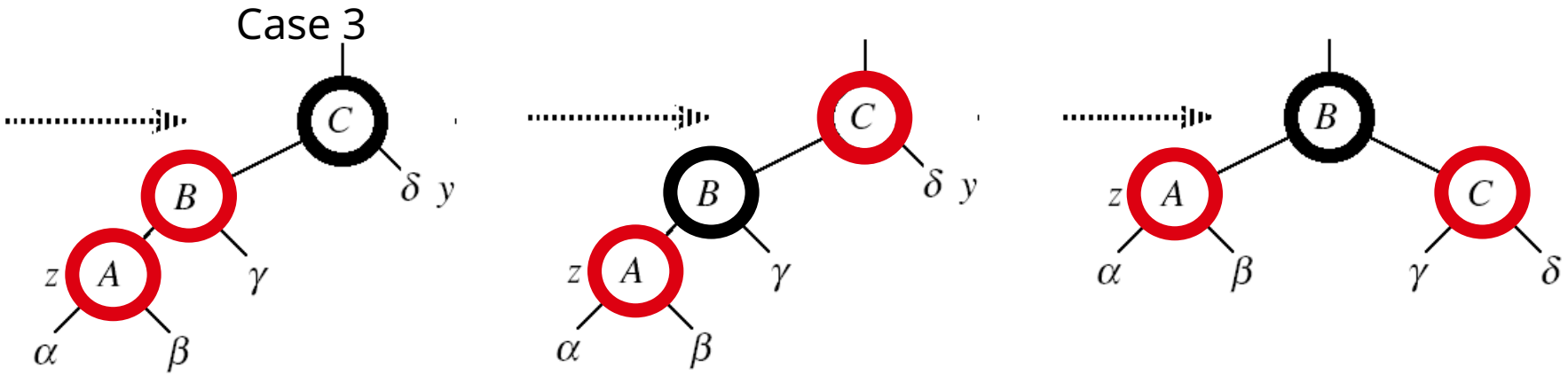
RB-INSERT-FIXUP – Case 3

Case 3:

- z's "uncle" (y) is **black**
- z is a left child

Idea:

- color[p[z]] ← black
- color[p[p[z]]] ← red
- RIGHT-ROTATE(T, p[p[z]]) Case3
- No longer have 2 reds in a row
- p[z] is now black



RB-INSERT-FIXUP – Case 2

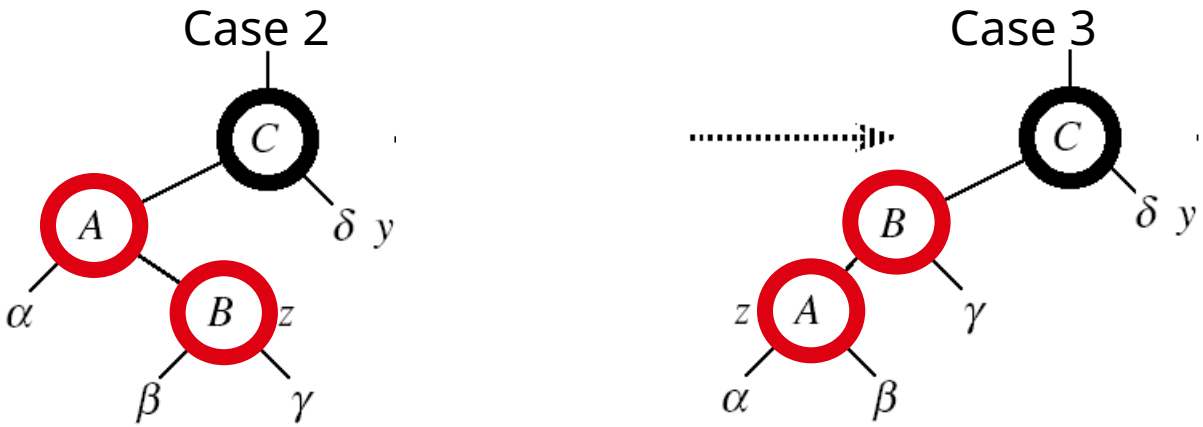
Case 2:

- z's "uncle" (y) is **black**
- z is a right child

Idea:

- z ← p[z]
- LEFT-ROTATE(T, z) Case2

⇒ now z is a left child, and both z and p[z] are red ⇒ case 3



RB-INSERT-FIXUP(T, z)

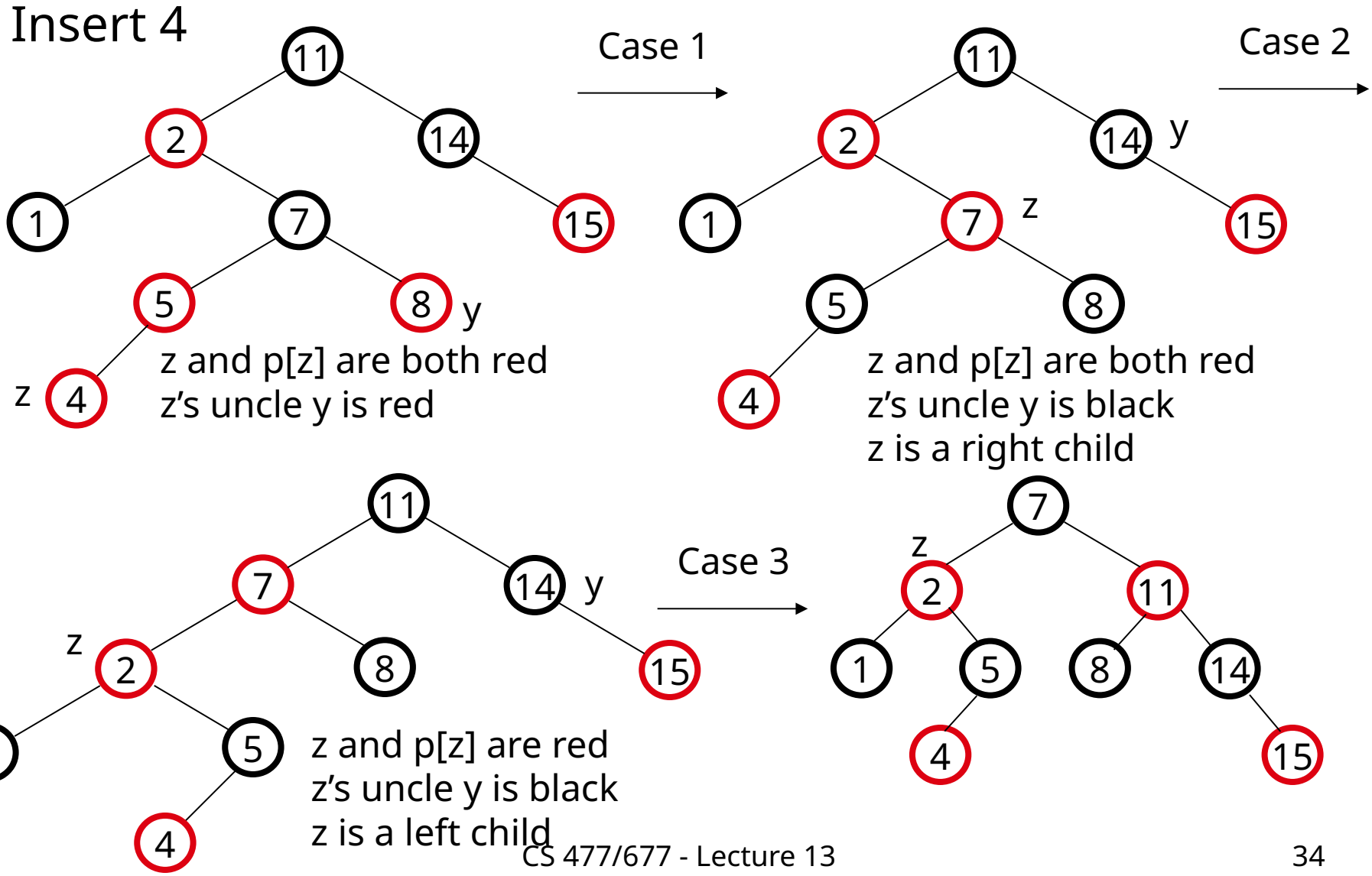
1. **while** color[p[z]] = RED
2. **do if** p[z] = left[p[p[z]]]
3. **then** y ← right[p[p[z]]]
4. **if** color[y] = RED
5. **then Case1**
6. **else if** z = right[p[z]]
7. **then Case2**
8. **Case3**
9. **else** (same as **then** clause with "right" and "left" exchanged)
10. color[root[T]] ← BLACK

The while loop repeats only when case1 is executed: O(lgn) times

Set the value of x's "uncle"

We just inserted the root, or the red node reached the root

Example


Insert 4
11
2
14
1
7
15
5
8
y
z
4
z and p[z] are both red
z's uncle y is red
Case 1
11
2
14
y
1
7
z
15
5
8
4
z and p[z] are both red
z's uncle y is black
z is a right child
Case 2
11
7
14
y
z
2
8
15
1
5
4
z and p[z] are red
z's uncle y is black
z is a left child
Case 3
7
z
2
11
1
5
8
14
4
15

Analysis of RB-INSERT

- Inserting the new element into the tree O(lgn)
- RB-INSERT-FIXUP
  - The while loop repeats only if CASE 1 is executed
  - The number of times the while loop can be executed is O(lgn)
- Total running time of RB-INSERT: O(lgn)

Red-Black Trees - Summary

- Operations on red-black trees:
  - SEARCH O(h)
  - PREDECESSOR O(h)
  - SUCCESOR O(h)
  - MINIMUM O(h)
  - MAXIMUM O(h)
  - INSERT O(h)
  - DELETE O(h)
- Red-black trees guarantee that the height of the tree will be O(lgn)

Augmenting Data Structures

- Let's look at two new problems:
  - Dynamic order statistic
  - Interval search
- It is unusual to have to design all-new data structures from scratch
  - Typically: store additional information in an already known data structure
  - The augmented data structure can support new operations
- We need to correctly maintain the new information without loss of efficiency

Dynamic Order Statistics

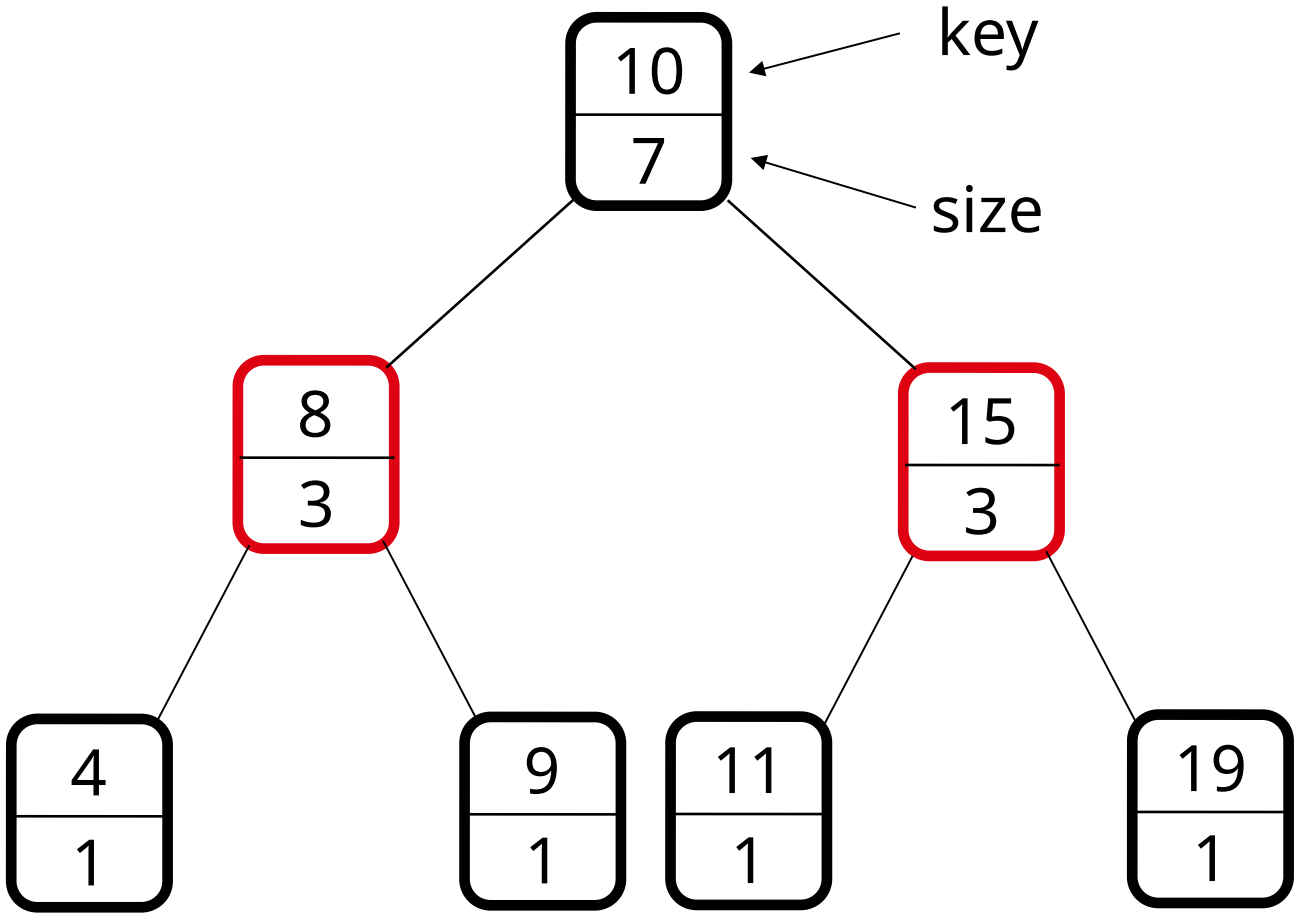
- Def.: the i-th order statistic of a set of n elements, where $i \in \{1, 2, ..., n\}$ is the element with the i-th smallest key.
- We can retrieve an order statistic from an unordered set:
  - Using: RANDOMIZED-SELECT
  - In: O(n) time
- We will show that:
  - With red-black trees we can achieve this in O(lgn)
  - Finding the **rank** of an element takes also O(lgn)

Order-Statistic Tree

- Def.: **Order-statistic tree:** a red-black tree with additional information stored in each node
- Node representation:
  - Usual fields: key[x], color[x], p[x], left[x], right[x]
  - Additional field: size[x] that contains the number of (internal) nodes in the subtree rooted at x (including x itself)
- For any internal node of the tree:

$$size[x] = size[left[x]] + size[right[x]] + 1$$

Example: Order-Statistic Tree



OS-SELECT

Goal:

- Given an order-statistic tree, return a pointer to the node containing the i-th smallest key in the subtree rooted at x

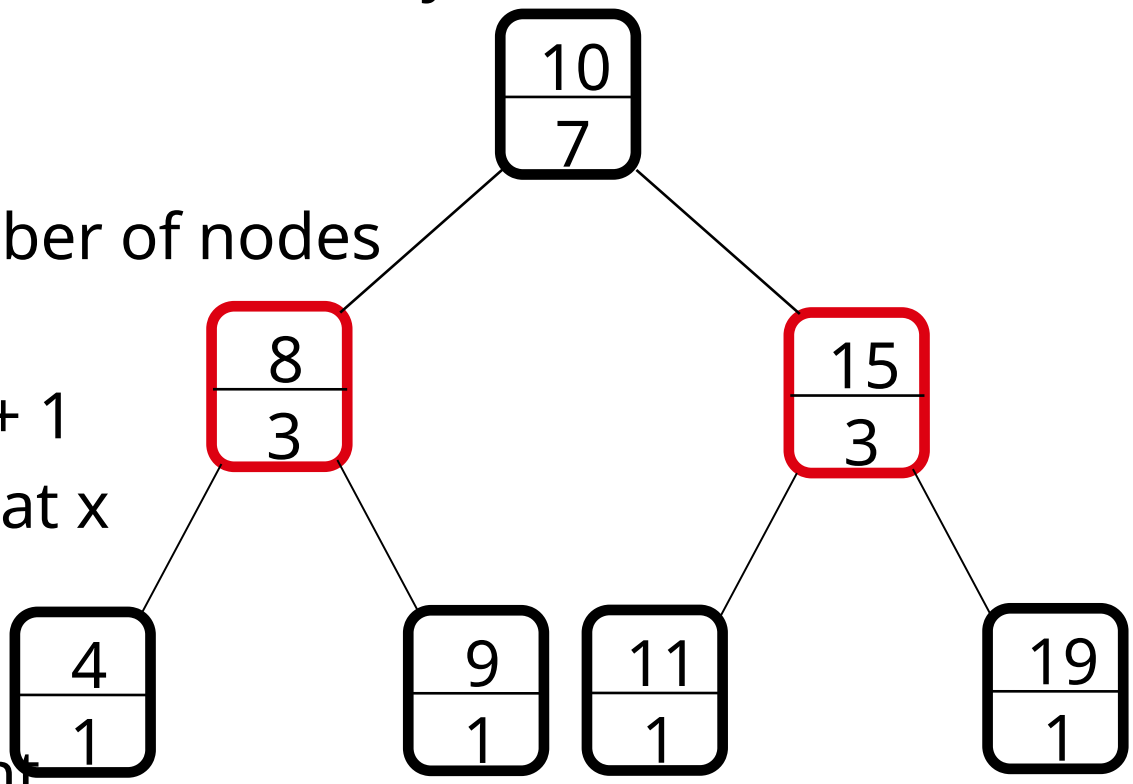
Idea:

- size[left[x]] = the number of nodes

that are smaller than x

- rank'[x] = size[left[x]] + 1

  in the subtree rooted at x
- If i = rank'[x] Done!
- If i < rank'[x]: look left
- If i > rank'[x]: look right



OS-SELECT(x, i)

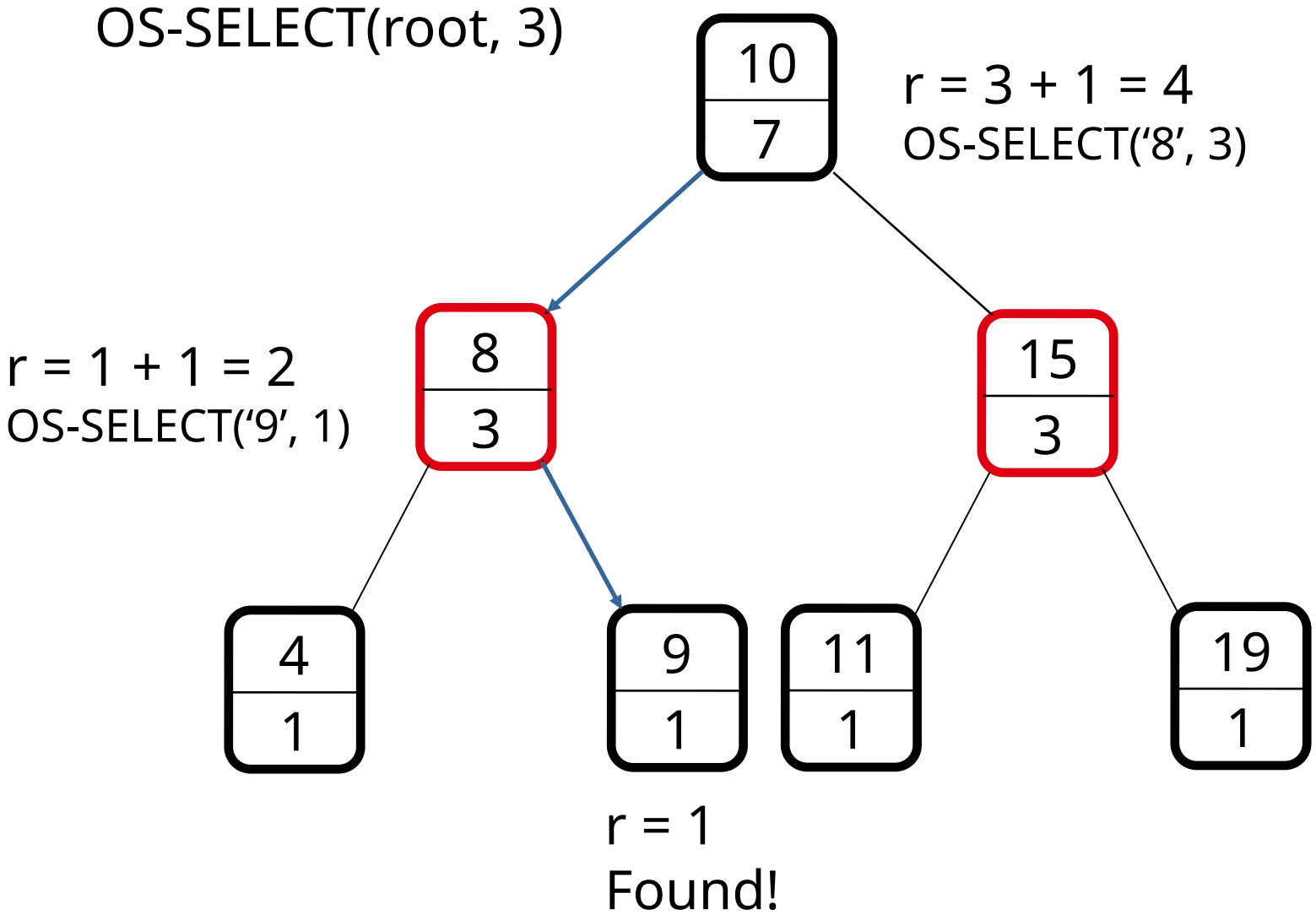
1. r ← size[left[x]] + 1
2. **if** i = r
3. **then return** x
4. **elseif** i < r
5. **then return** OS-SELECT(left[x], i)
6. **else return** OS-SELECT(right[x], i - r)

► compute the rank of x within the subtree rooted at x

Initial call: OS-SELECT(root[T], i)

Running time: O(lgn)

Example: OS-SELECT



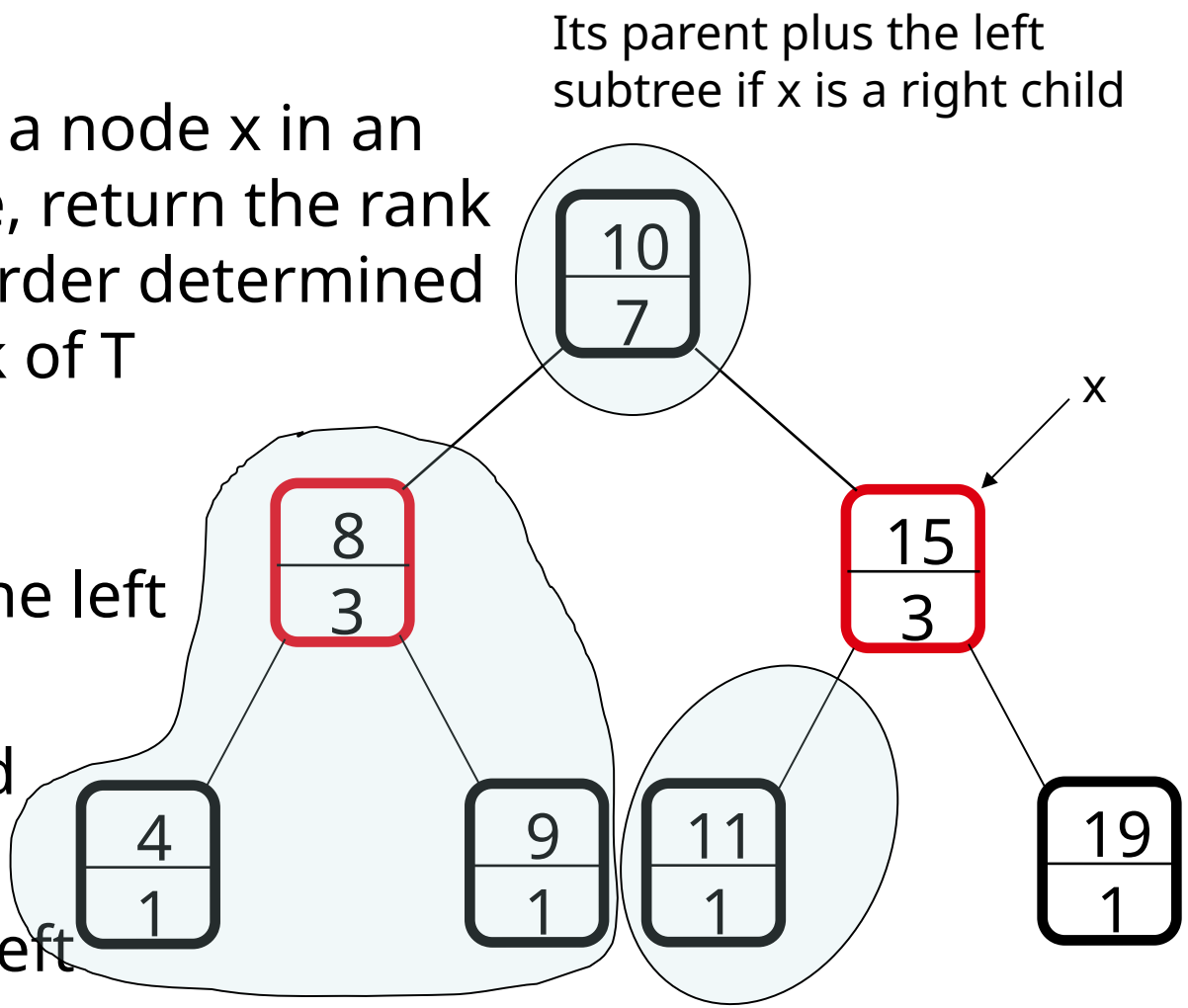
OS-RANK

Goal:

- Given a pointer to a node x in an order-statistic tree, return the rank of x in the linear order determined by an inorder walk of T

Idea:

- Add elements in the left subtree
- Go up the tree and if a right child: add the elements in the left subtree of the parent + 1



OS-RANK(T, x)

1. r ← size[left[x]] + 1
2. y ← x
3. **while** y ≠ root[T]
4. **do if** y = right[p[y]]
5. **then** r ← r + size[left[p[y]]] + 1
6. y ← p[y]
7. **return** r

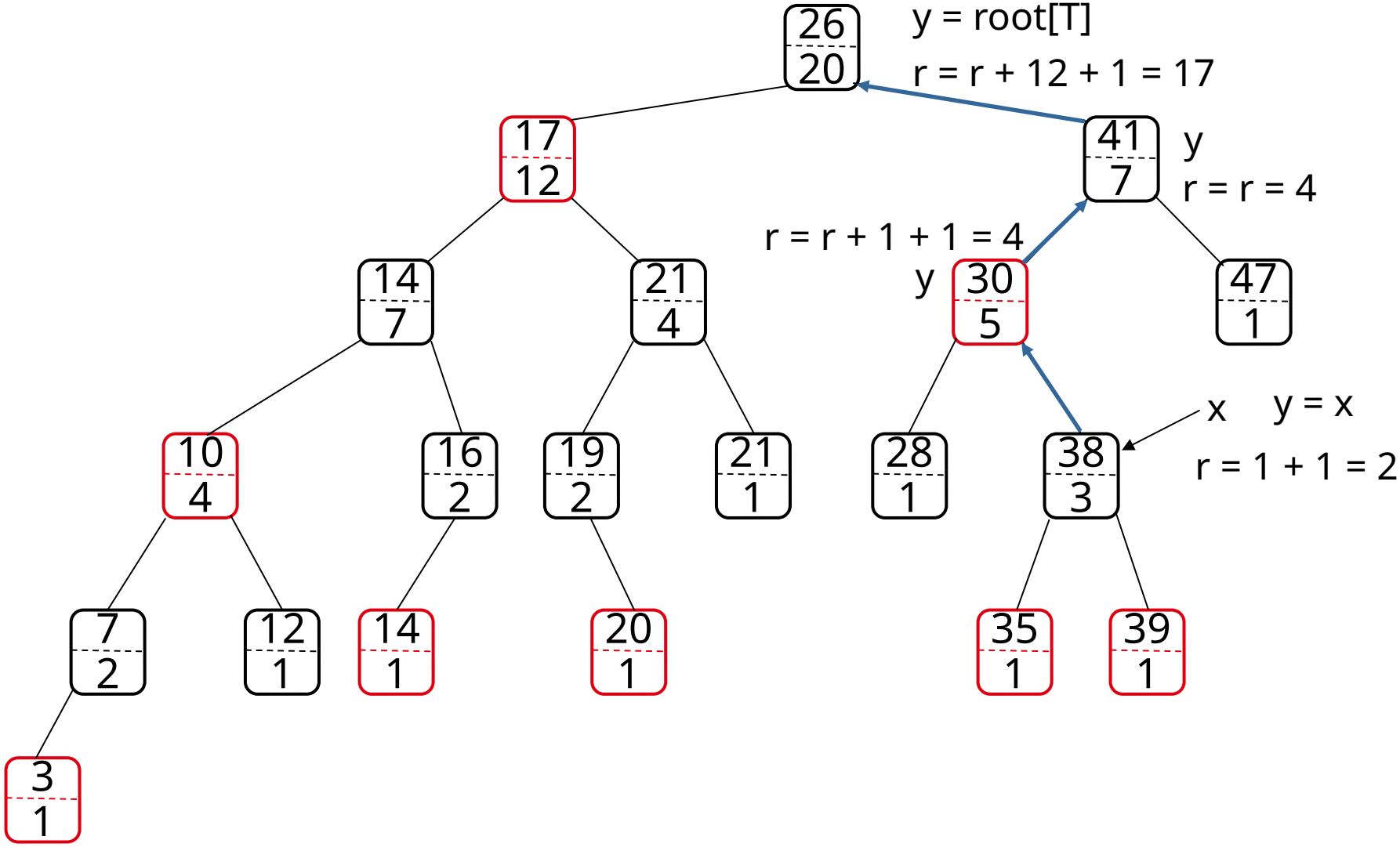
Add to the rank the elements in its left subtree + 1 for itself

Set y as a pointer that will traverse the tree

If a right child add the size of the parent's left subtree + 1 for the parent

Running time: O(lgn)

Example: OS-RANK



Maintaining Subtree Sizes

- We need to maintain the size field during INSERT and DELETE operations
- Need to maintain them efficiently
- Otherwise, might have to recompute all size fields, at a cost of $\Omega(n)$

Maintaining Size for OS-INSERT

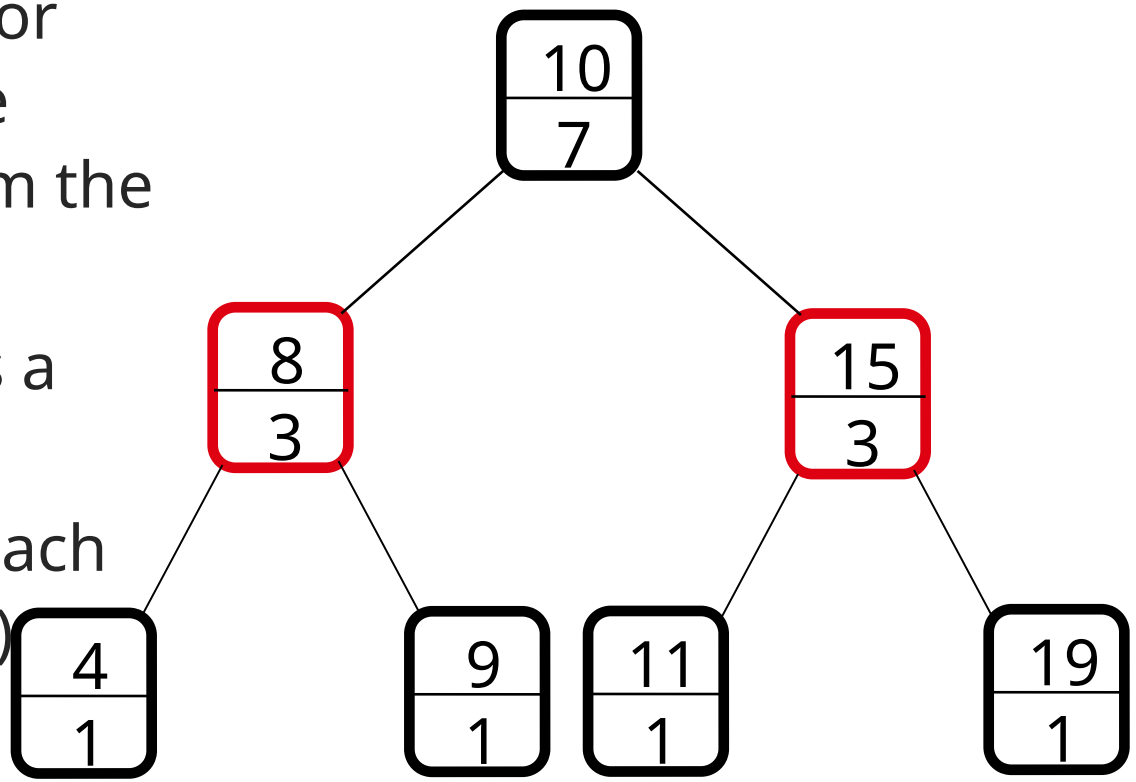
- Insert in a red-black tree has two stages
  1. Perform a binary-search tree insert
  2. Perform rotations and change node colors to restore red-black tree properties

OS-INSERT

Idea for maintaining the size field during insert

Phase 1 (going down):

- Increment size[x] for each node x on the traversed path from the root to the leaves
- The new node gets a size of 1
- Constant work at each node, so still O(lgn)

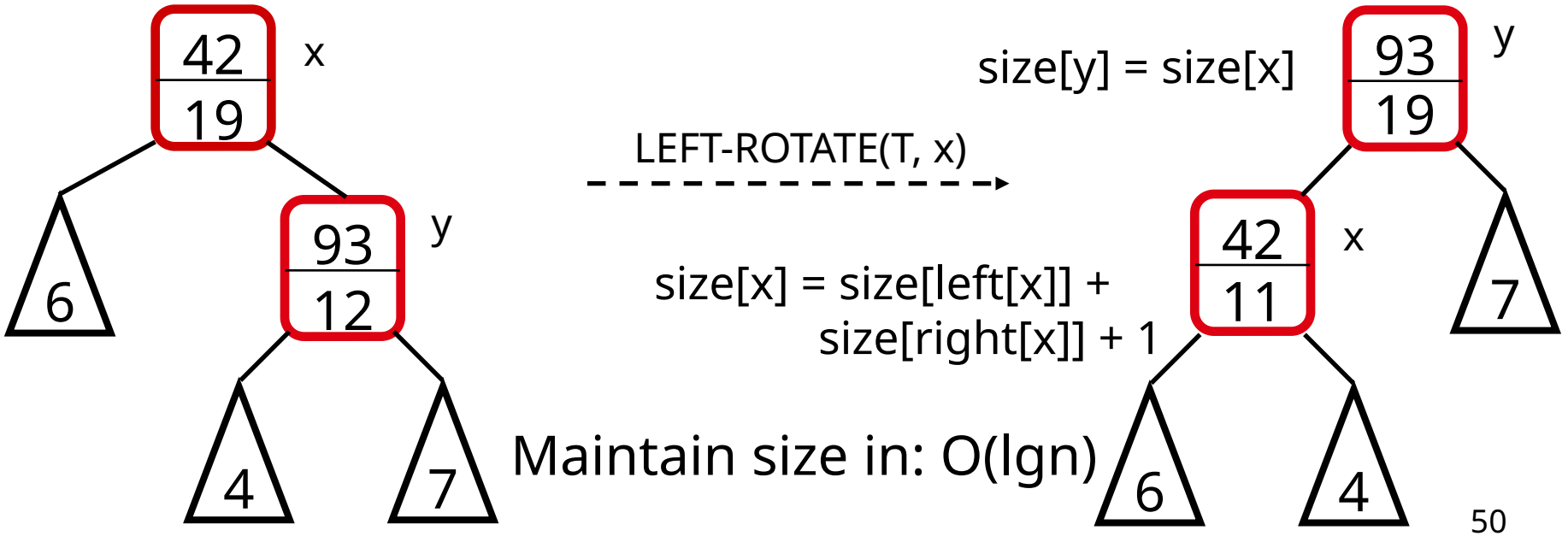


OS-INSERT

Idea for maintaining the size field during insert

Phase 2 (going up):

- During RB-INSERT-FIXUP there are:
  - O(lgn) changes in node colors
  - At most two rotations Rotations affect the subtree sizes !!



Maintain size in: O(lgn)

Augmenting a Data Structure

1. Choose an underlying data structure

   ⇒ Red-black trees

2. Determine additional information to maintain

   ⇒ size[x]

3. Verify that we can maintain additional information for existing data structure operations

   ⇒ Shown how to maintain size during modifying operations

4. Develop new operations

   ⇒ Developed OS-RANK and OS-SELECT

Augmenting Red-Black Trees

Theorem: Let f be a field that augments a red-black tree. If the contents of f for a node can be computed using only the information in x, left[x], right[x] ⇒ we can maintain the values of f in all nodes during insertion and deletion, without affecting their O(lgn) running time.

Examples

1. Can we augment a RBT with size[x]?

   Yes: size[x] = size[left[x]] + size[right[x]] + 1

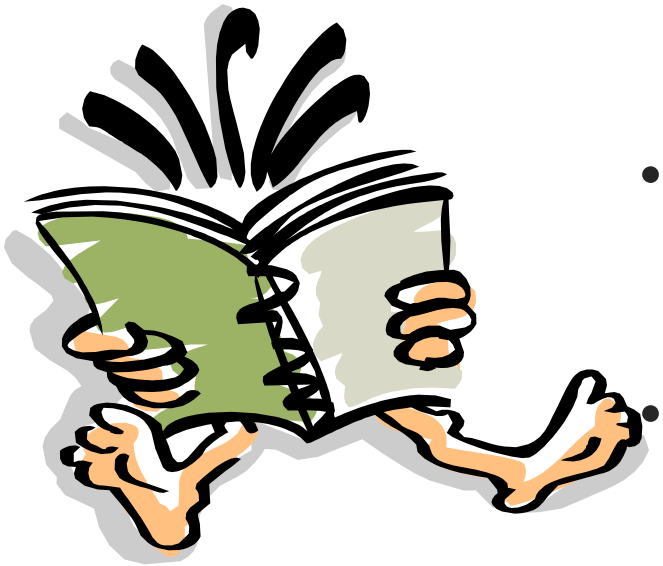
2. Can we augment a RBT with height[x]?

   Yes: height[x] = 1 + max(height[left[x]],
   height[right[x]])

3. Can we augment a RBT with rank[x]?

   No, inserting a new minimum will cause all n rank values to change

Readings



- For this lecture
  - Sections 6.3, 6,5
  - Chapter 13
- Coming next
  - Chapter 17