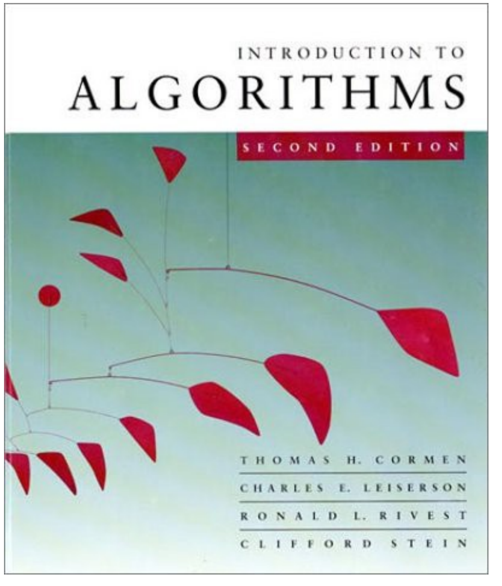
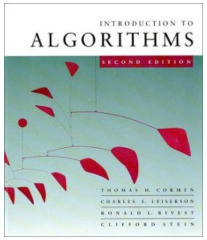
Introduction to Algorithms

6.046J/18.401J/SMA5503



Lecture 7

Based on slides by Prof. Erik Demaine

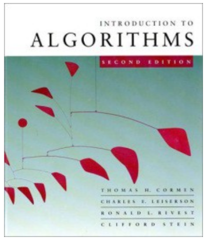


Balanced search trees

Balanced search tree: A search-tree data structure for which a height of $O(\lg n)$ is guaranteed when implementing a dynamic set of n items.

Examples:

- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees



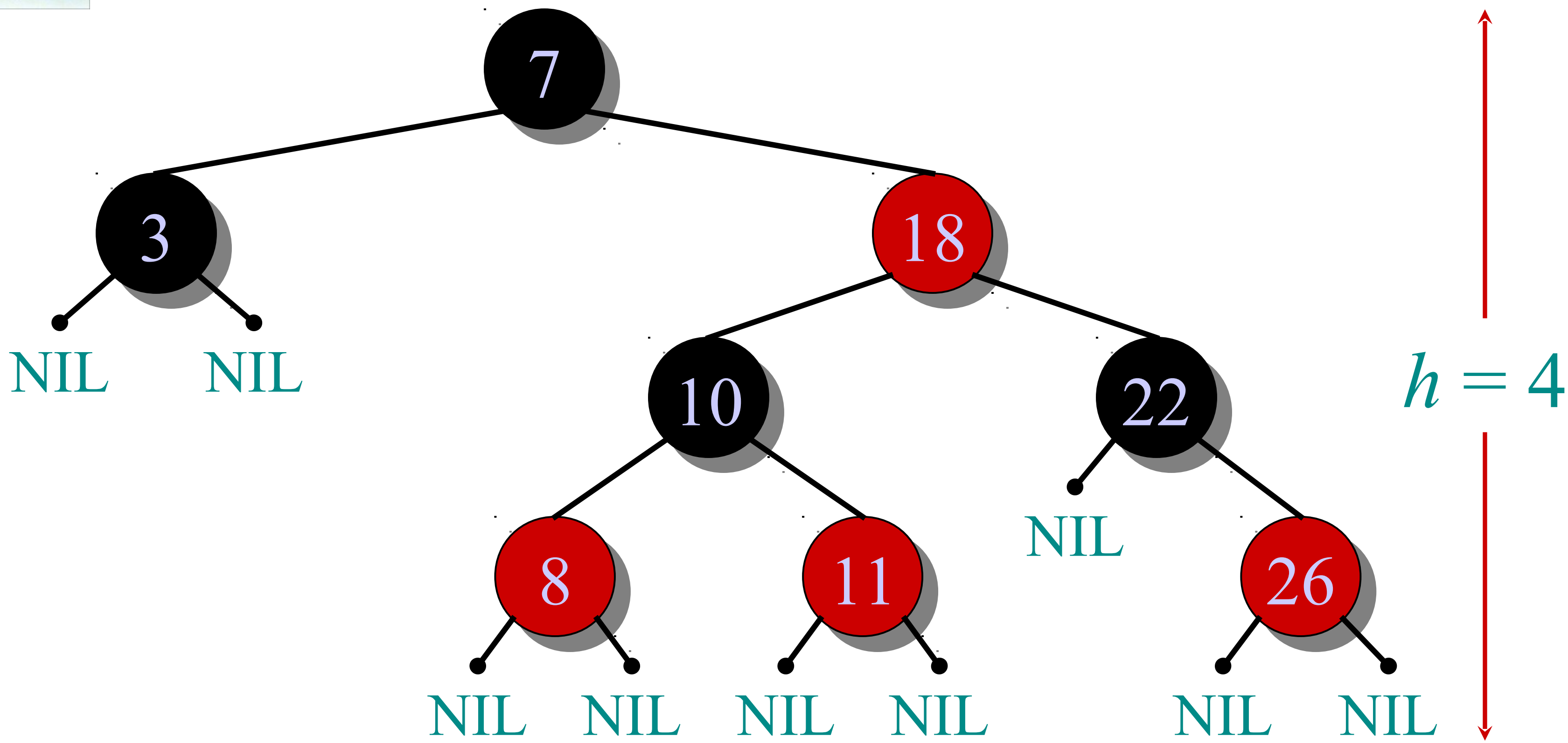
Red-black trees

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

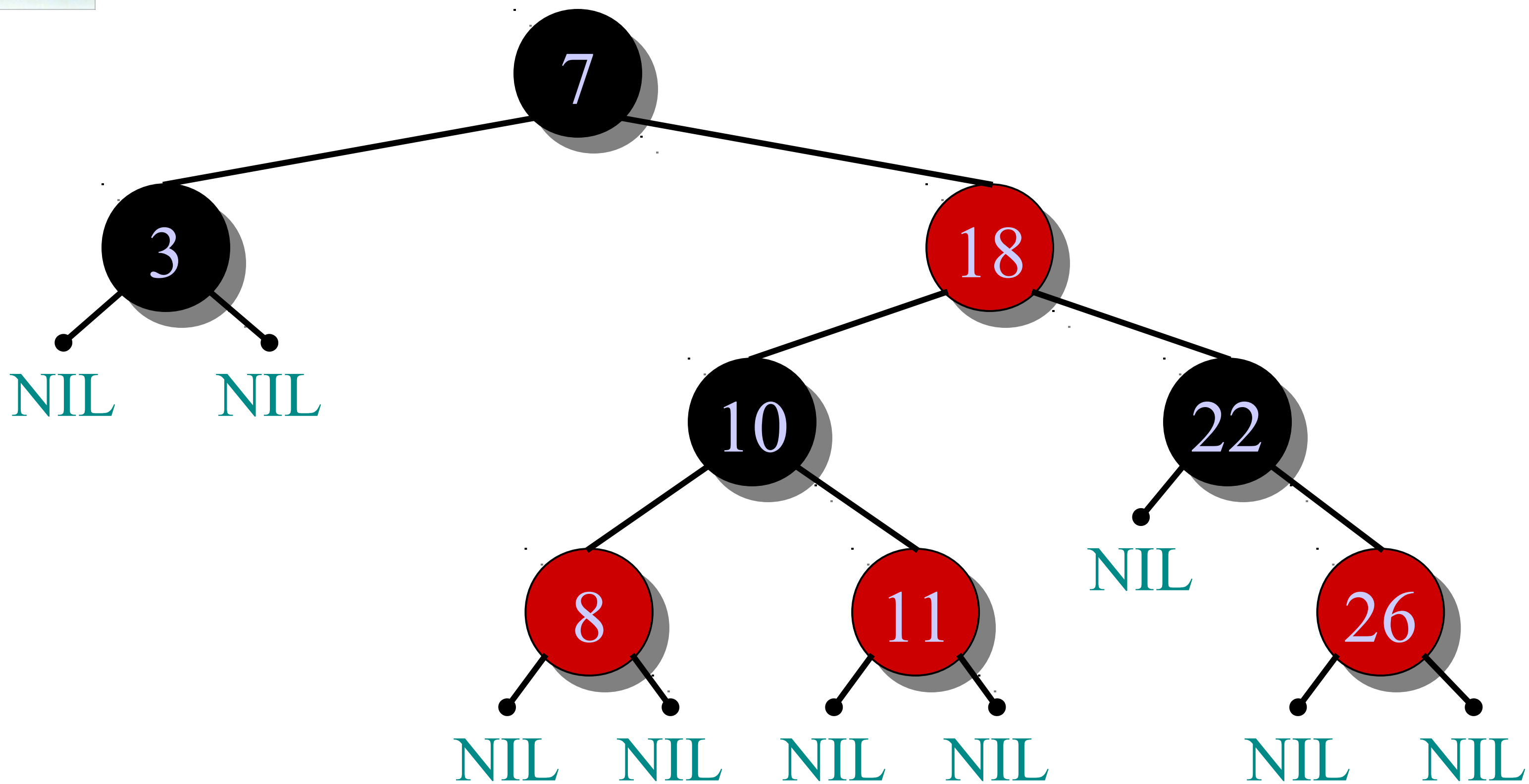
Red-black properties:

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

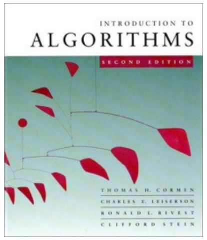
Example of a red-black tree



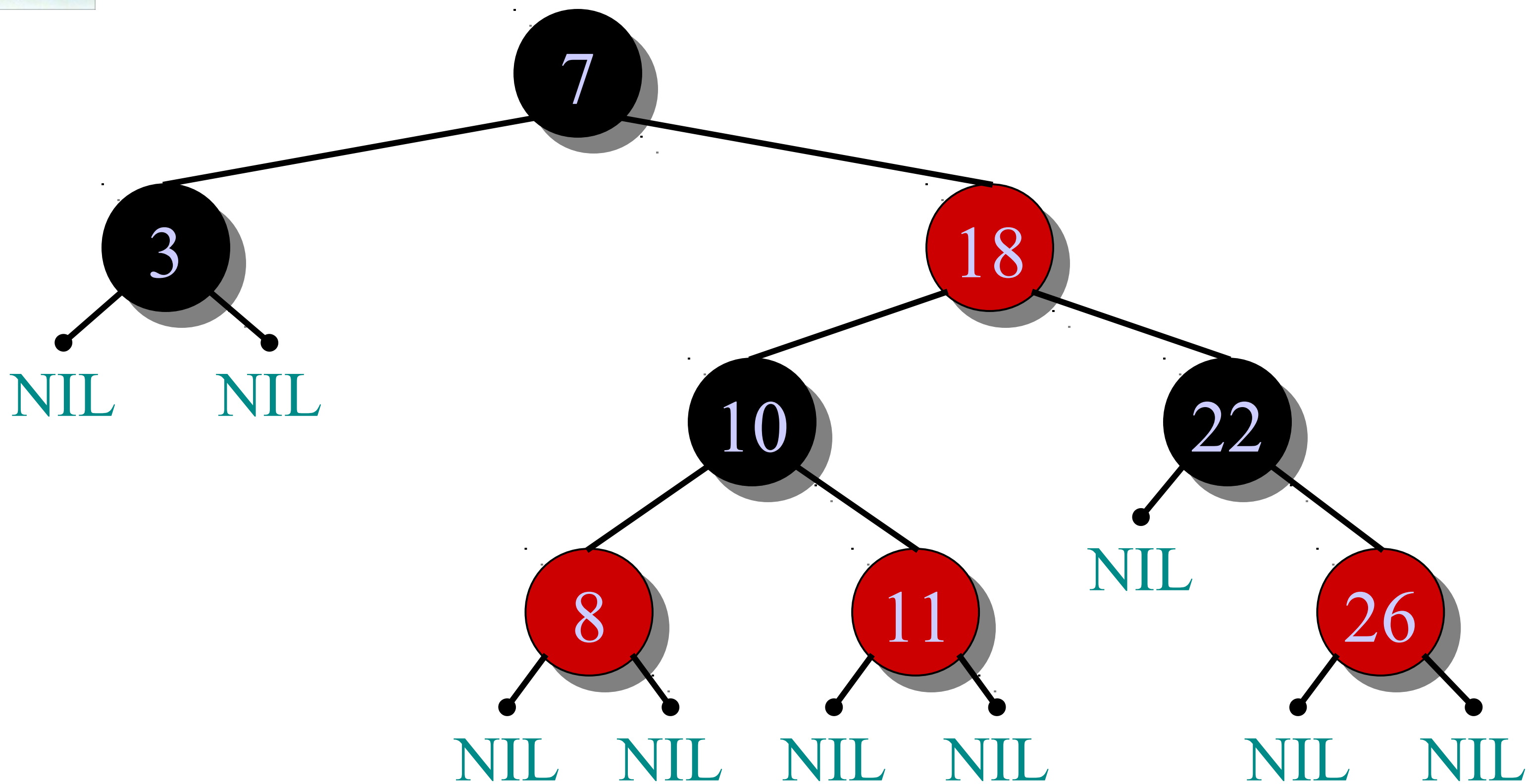
Example of a red-black tree



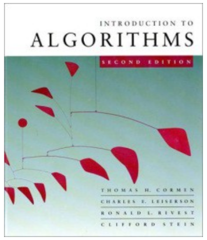
1. Every node is either red or black.



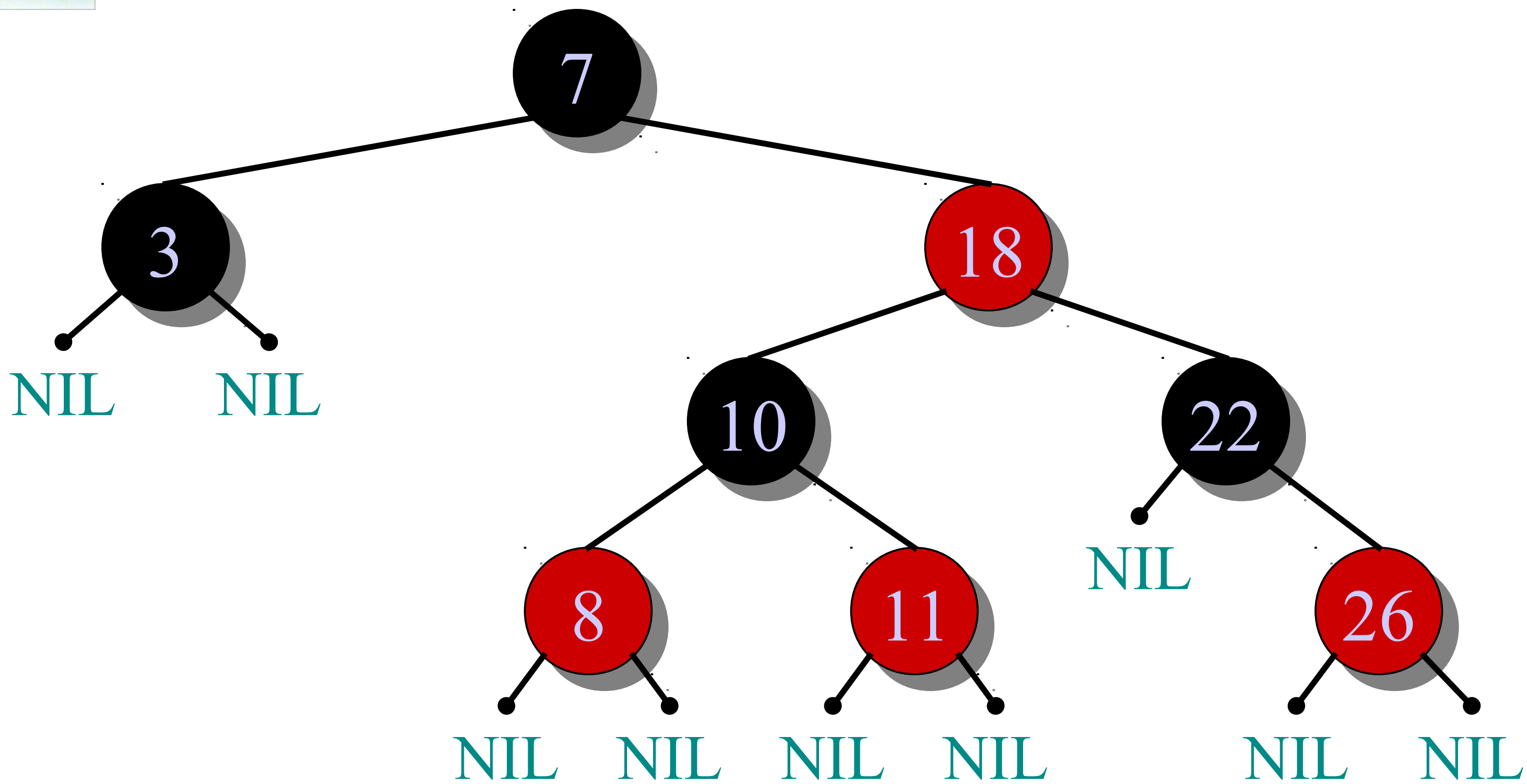
Example of a red-black tree



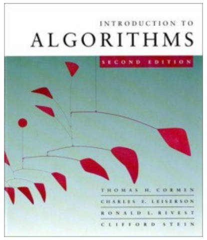
2. The root and leaves (NIL's) are black.



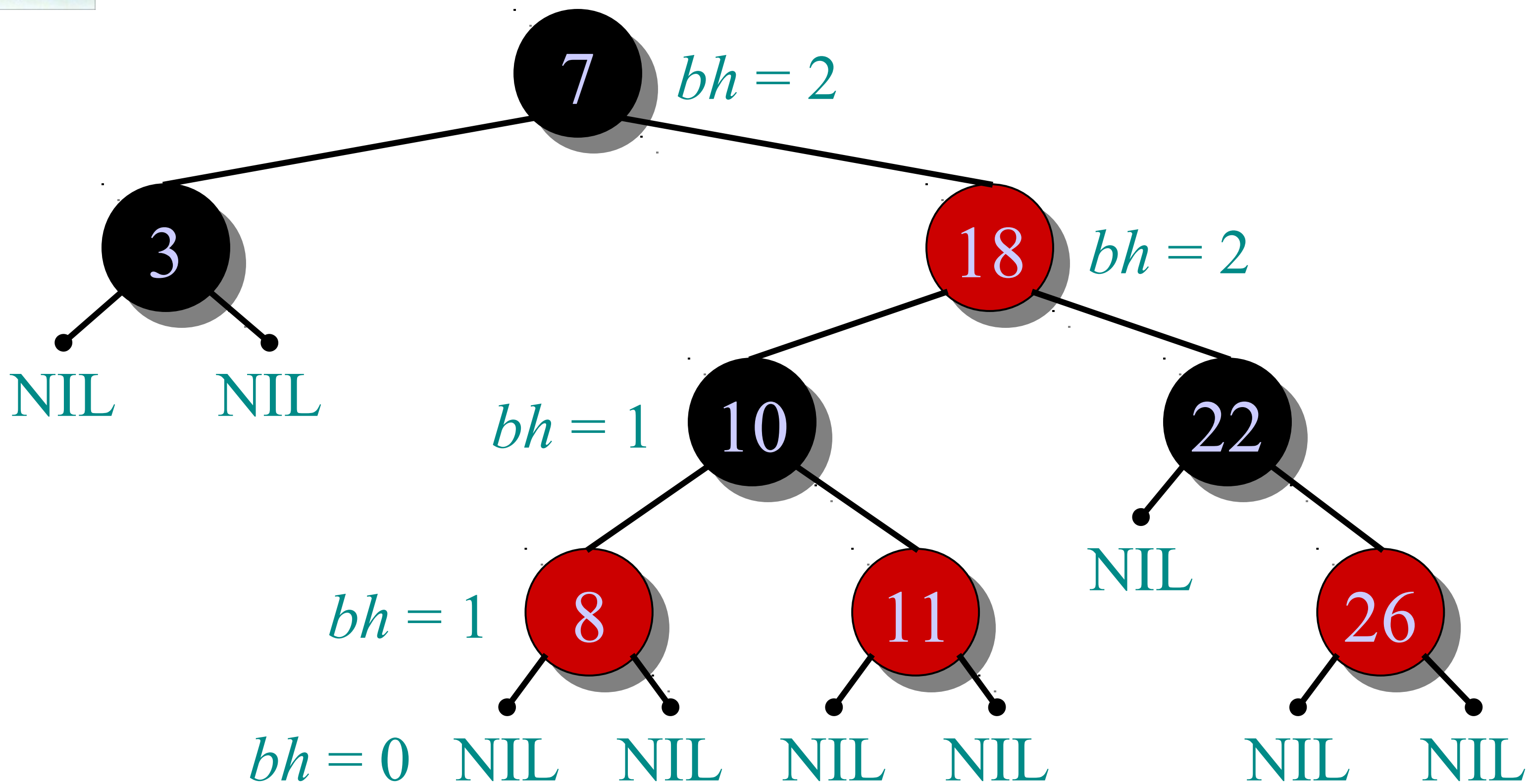
Example of a red-black tree



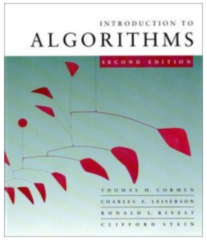
3. If a node is red, then its parent is black.



Example of a red-black tree



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



Height of a red-black tree

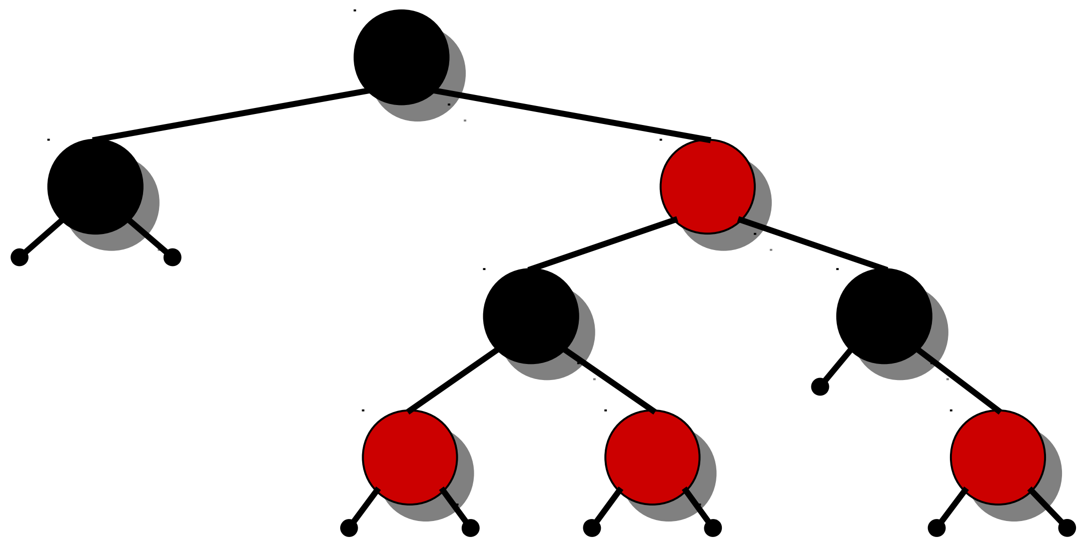
Theorem. A red-black tree with n keys has height

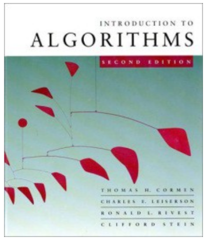
$$h \le 2 \lg(n + 1).$$

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

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

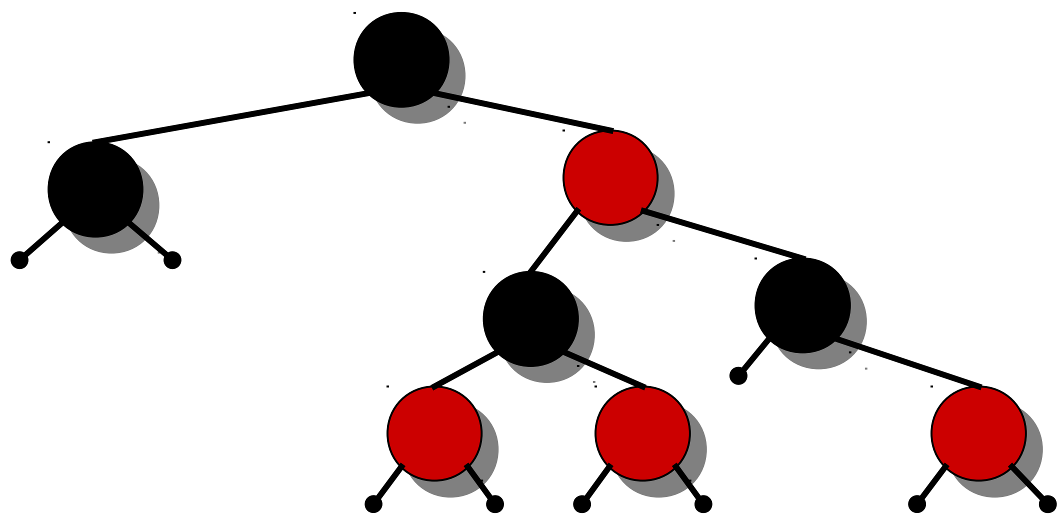
Theorem. A red-black tree with n keys has height

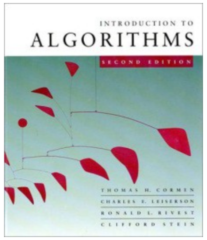
$$h \leq 2\lg(n+1).$$

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

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

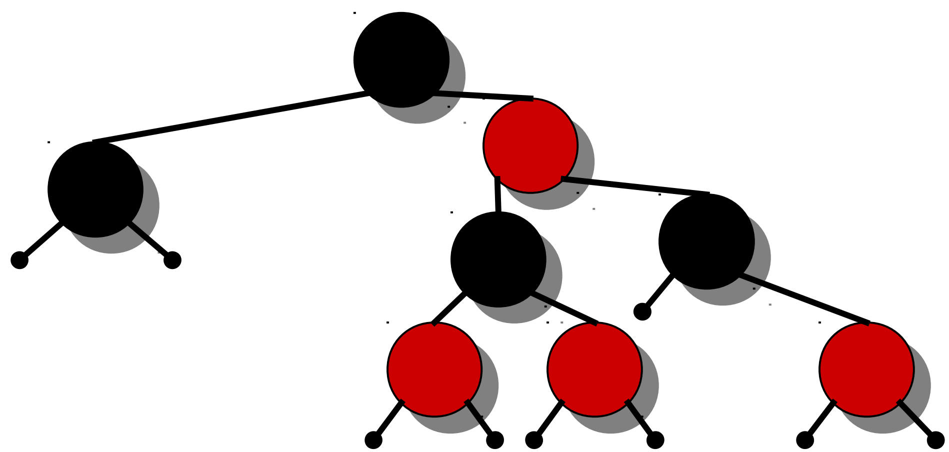
Theorem. A red-black tree with n keys has height

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

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

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

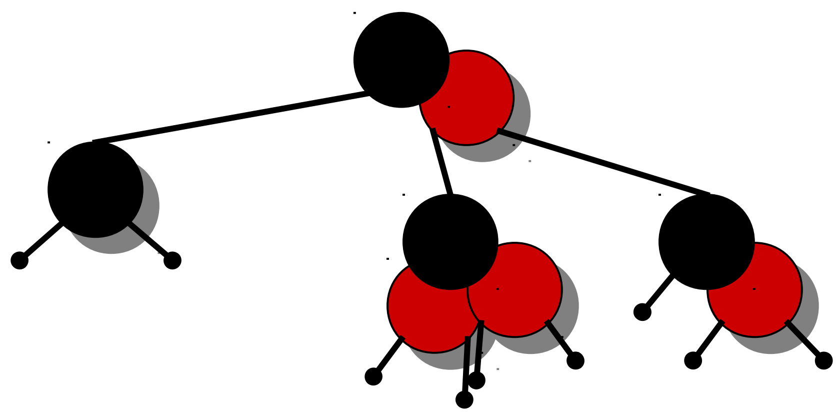
Theorem. A red-black tree with n keys has height

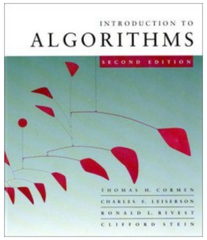
$$h \le 2\lg(n+1).$$

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

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

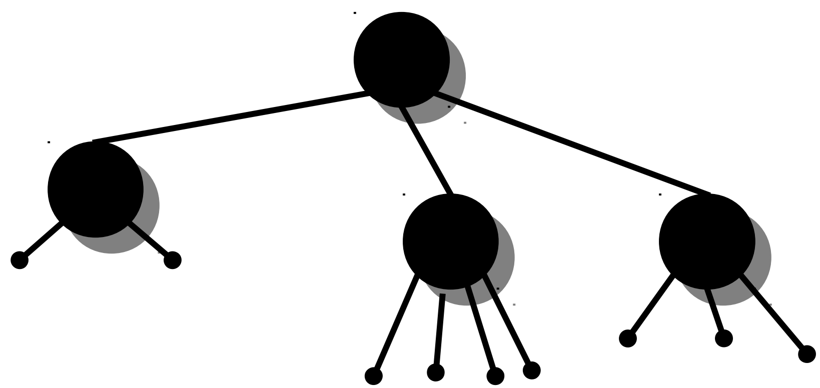
Theorem. A red-black tree with n keys has height

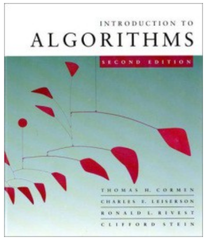
$$h \le 2 \lg(n + 1).$$

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

INTUITION:

- Merge red nodes into their black parents.





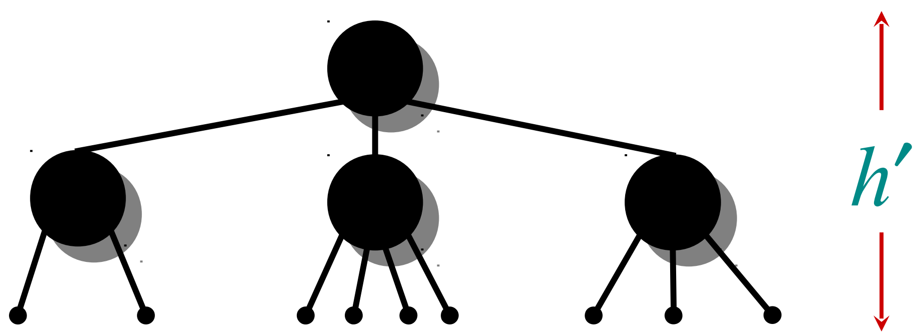
Height of a red-black tree

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

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

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

INTUITION:



- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves.

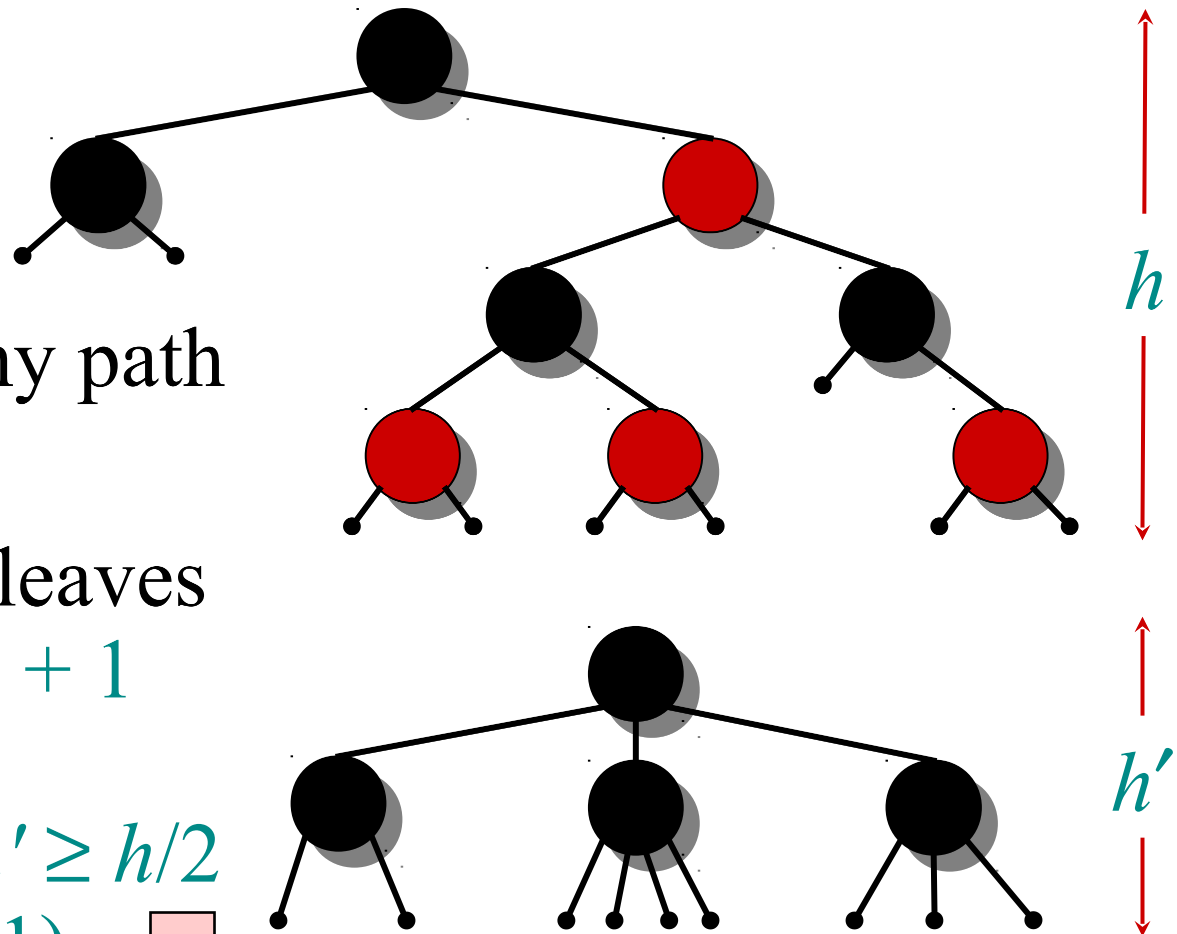
Proof (continued)

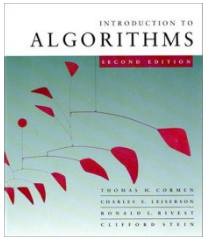
- We have $h' \geq h/2$, since at most half the leaves on any path are red.
- The number of leaves in each tree is $n + 1$

  $\Rightarrow n + 1 \geq 2^{h'}$

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

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





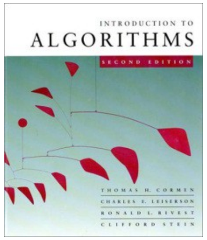
Query operations

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

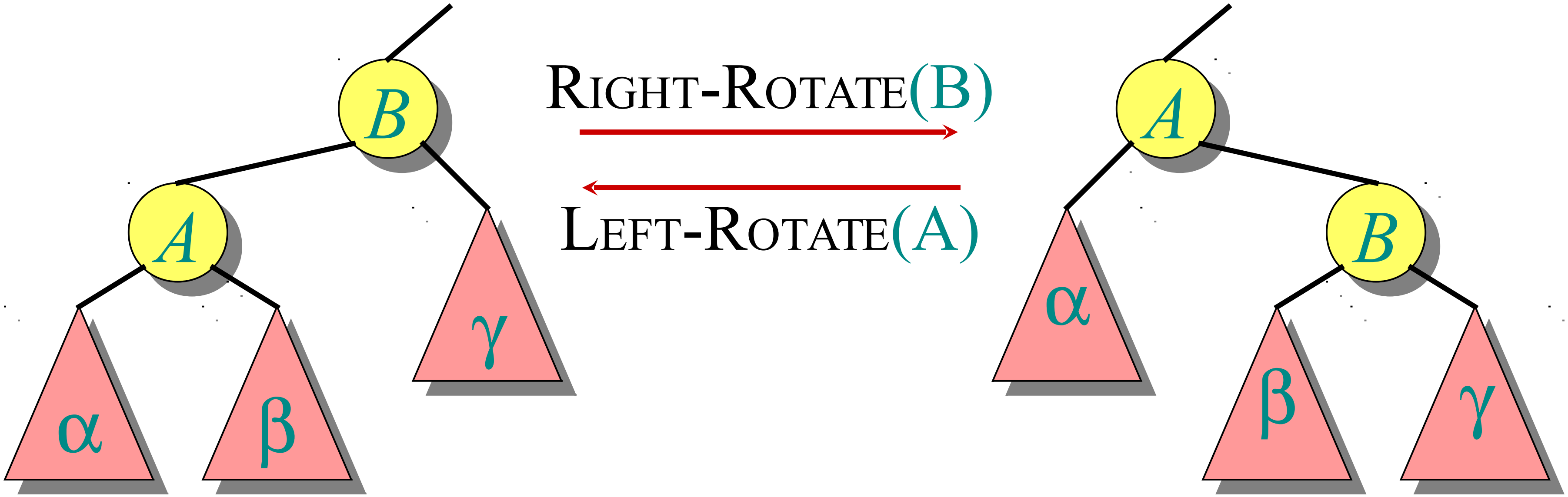
Modifying operations

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

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



Rotations



Rotations maintain the inorder ordering of keys:

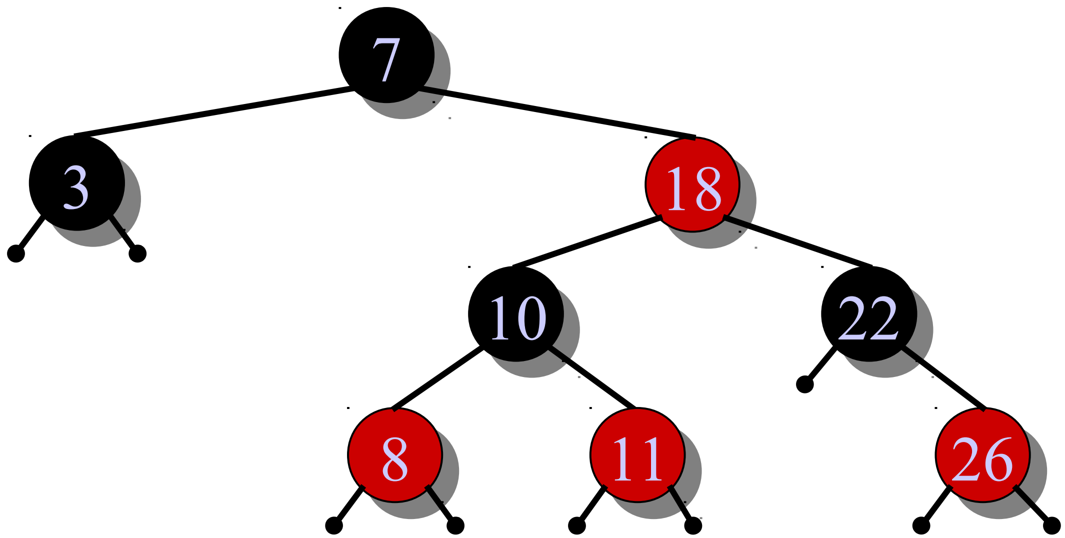
- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c$.

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

Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

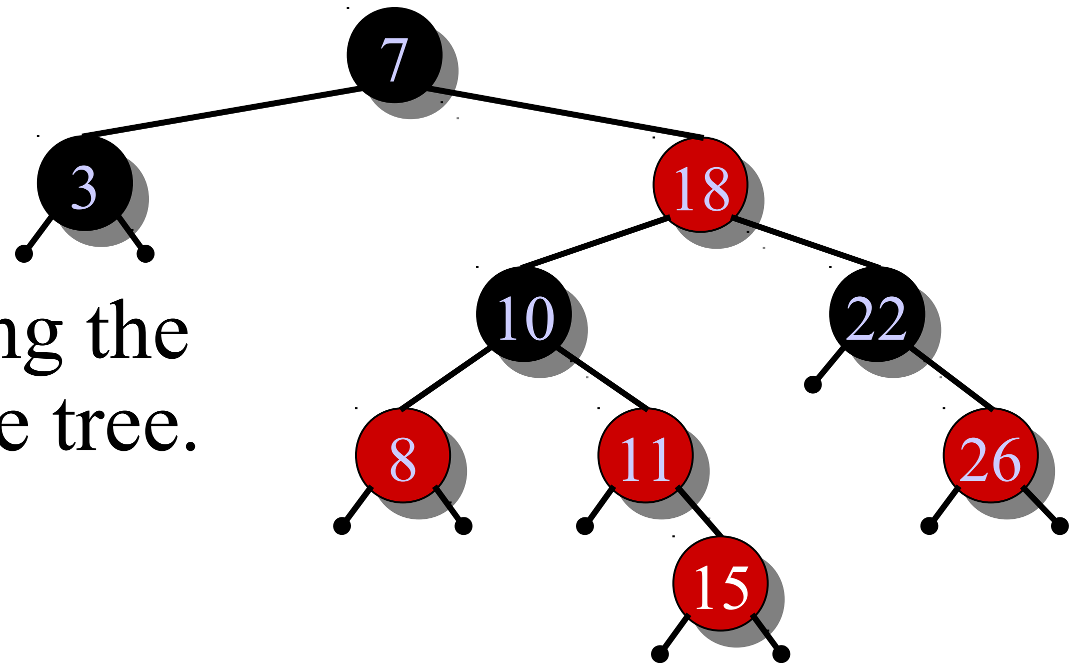


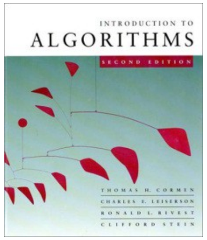
Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.



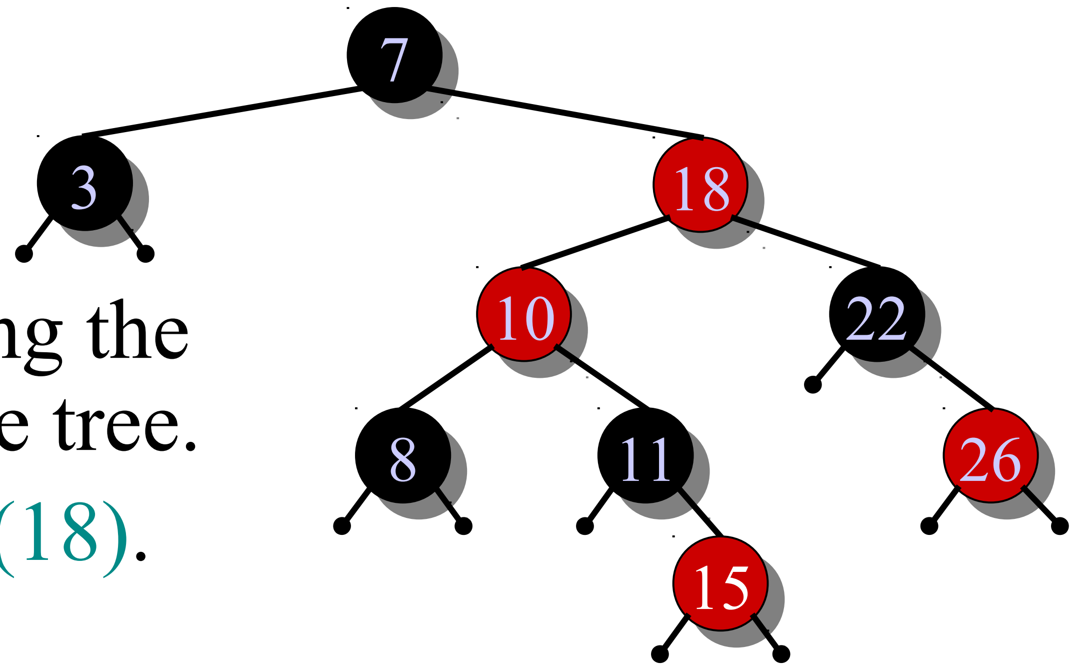


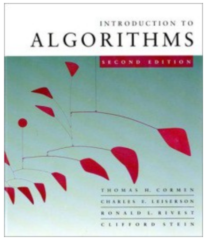
Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).



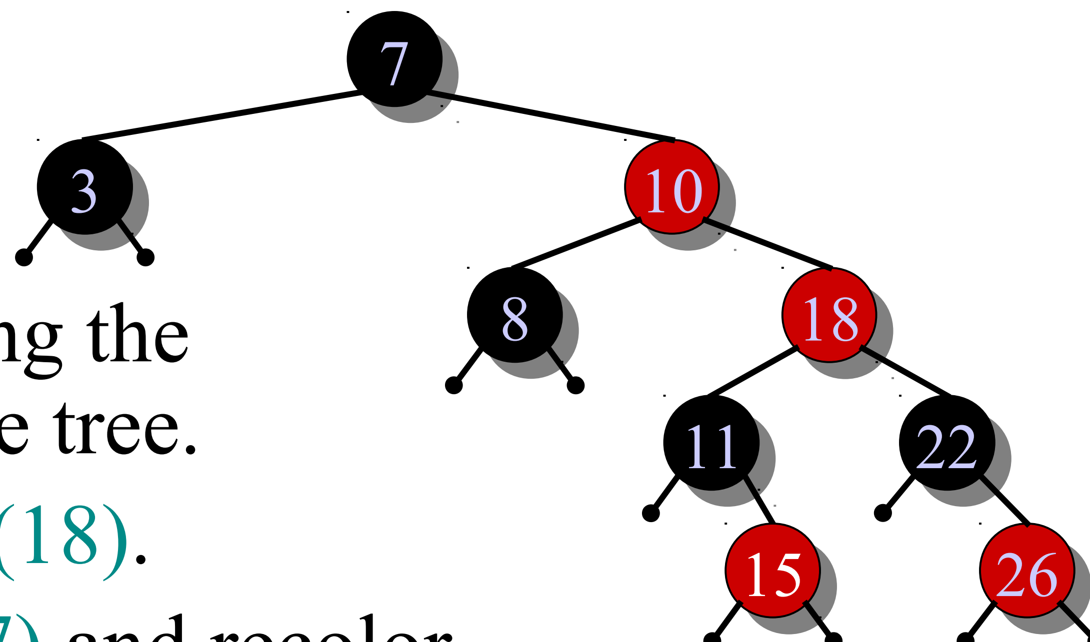


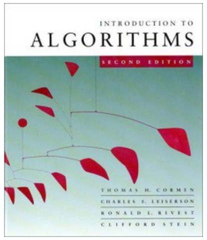
Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.



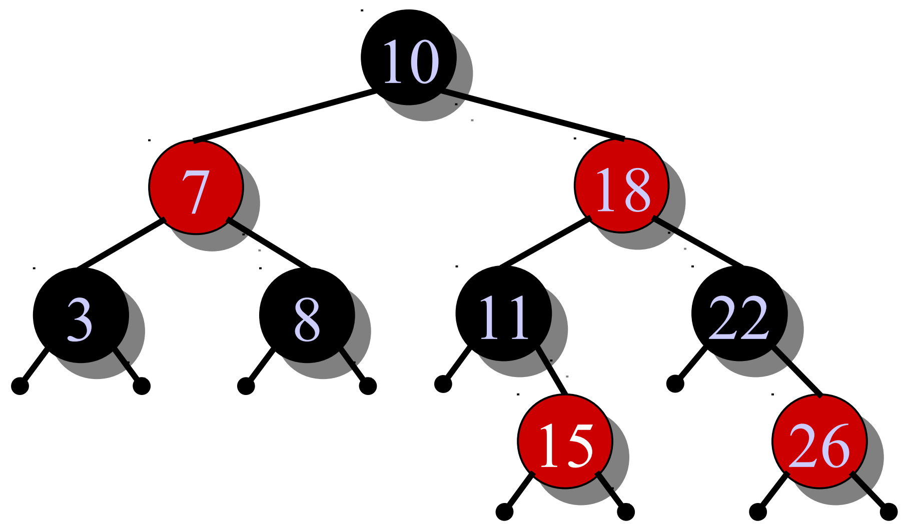


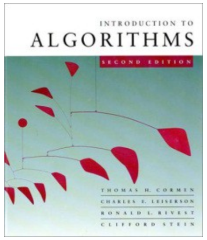
Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.

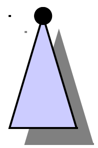


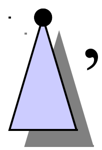


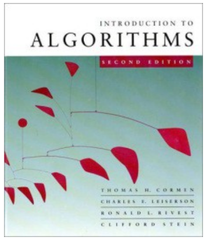
Pseudocode

```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] ← RED      ▷ only RB property 3 can be violated
    while x ≠ root[T] and color[p[x]] = RED
        do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]    ▷ y = aunt/uncle of x
                 if color[y] = RED
                  then ⟨Case 1⟩
                  else  if x = right[p[x]]
                          then ⟨Case 2⟩   ▷ Case 2 falls into Case 3
                       ⟨Case 3⟩
            else ⟨“then” clause with “left” and “right” swapped⟩
    color[root[T]] ← BLACK
```

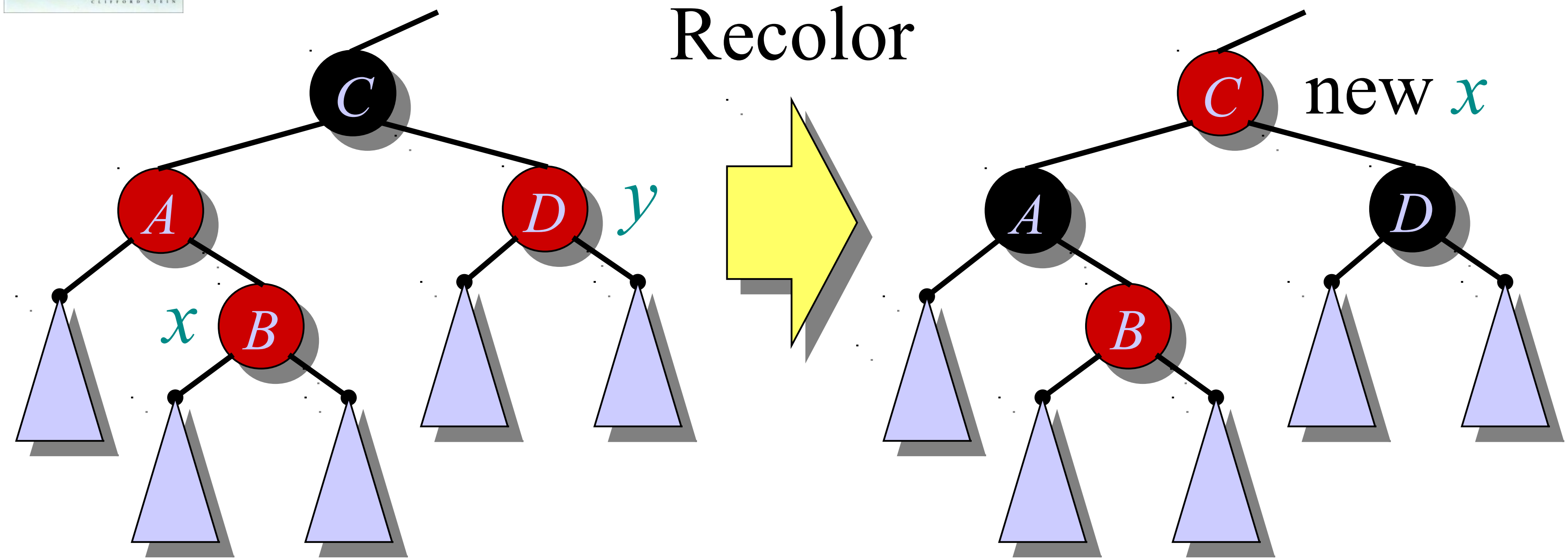
Graphical notation

Let  denote a subtree with a black root.

All 's have the same black-height.

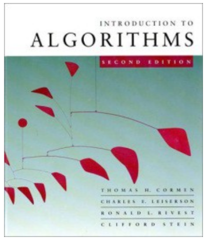


Case 1

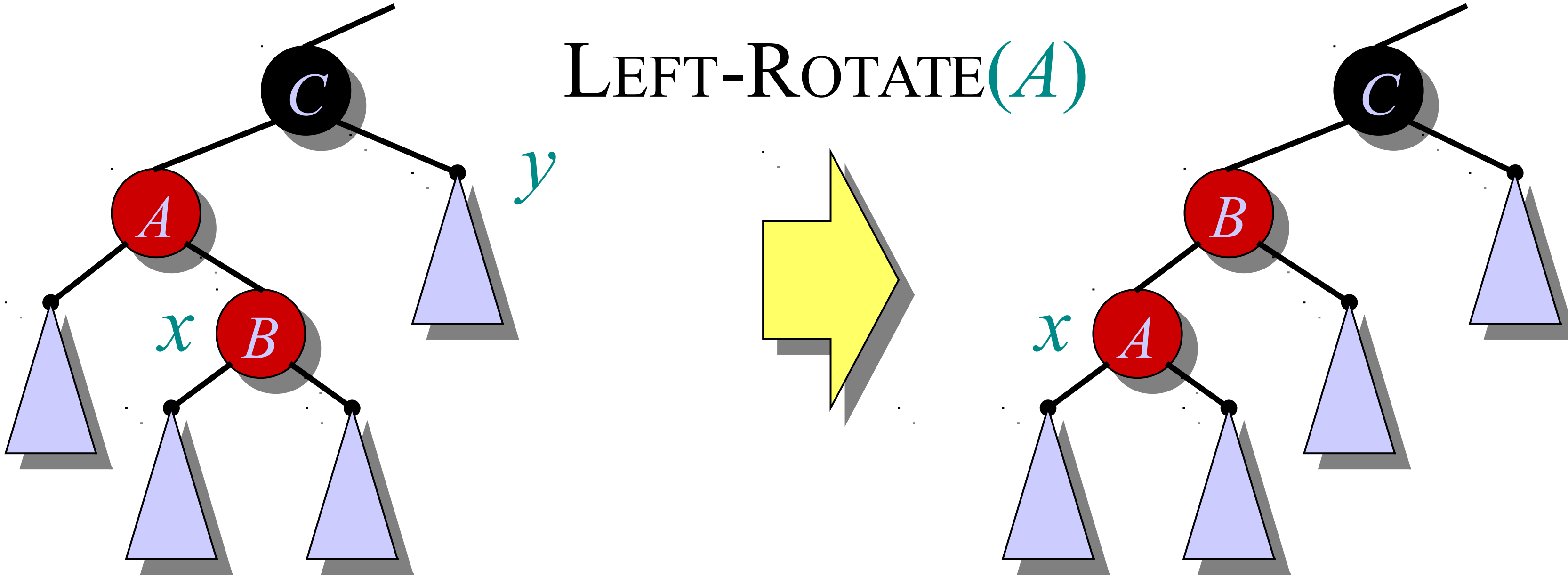


(Or, children of *A* are swapped.)

Push *C*'s black onto *A* and *D*, and recurse, since *C*'s parent may be red.

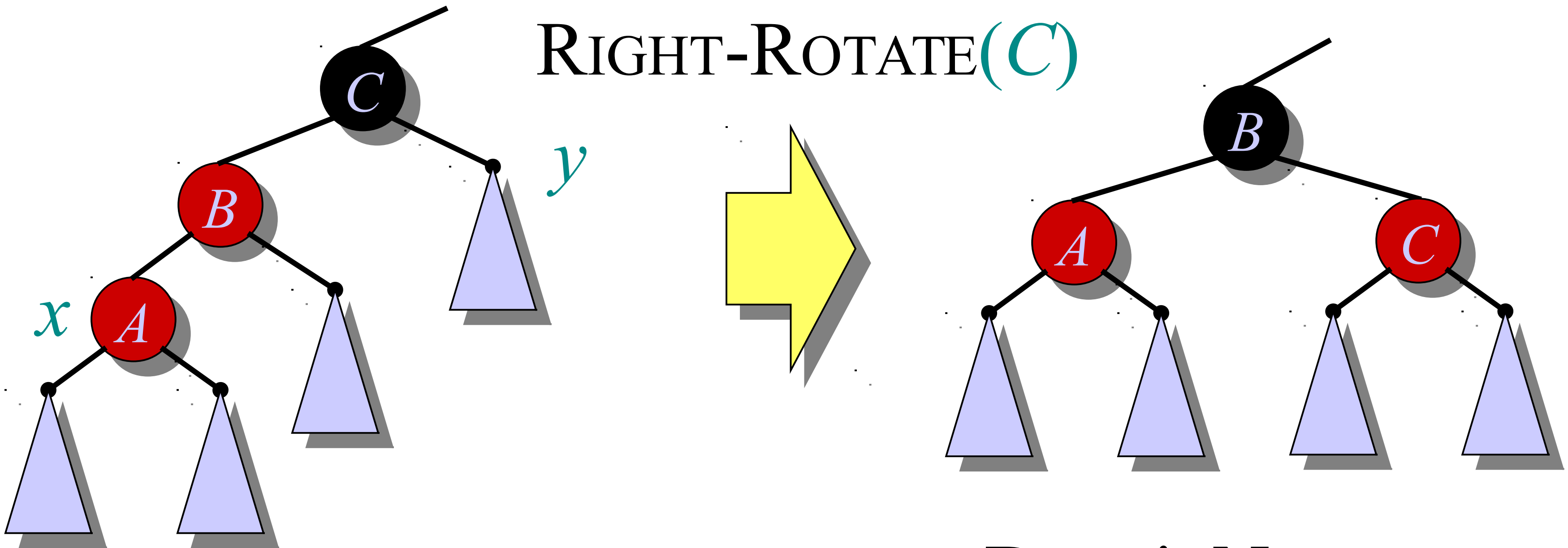


Case 2

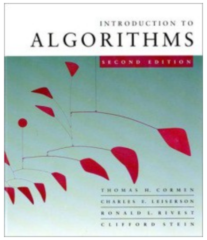


Transform to Case 3.

Case 3



Done! No more violations of RB property 3 are possible.



Analysis

- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

Running time: $O(\lg n)$ with $O(1)$ rotations.

RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT (see textbook).

Augmenting BST's

Adding more functionality to a BST by adding a bit more information

Examples:

- *Dynamic order statistics*
- *Interval trees*

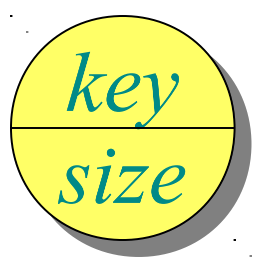
Dynamic order statistics

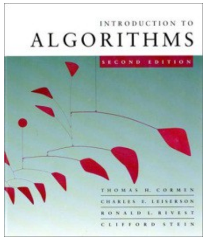
OS-SELECT(i, S): returns the ith smallest element in the dynamic set S.

OS-RANK(x, S): returns the rank of $x \in S$ in the sorted order of S's elements.

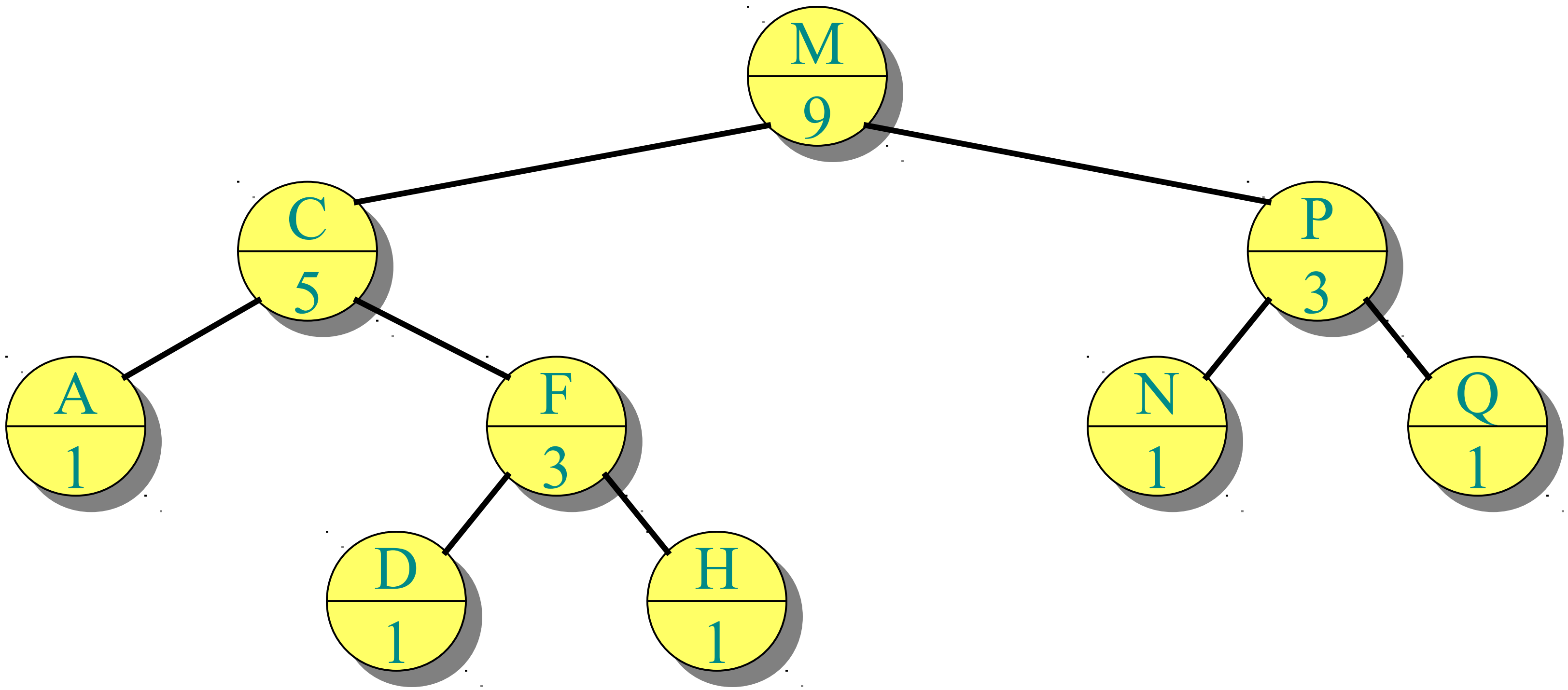
IDEA: Use a red-black tree for the set S, but keep subtree sizes in the nodes.

Notation for nodes:

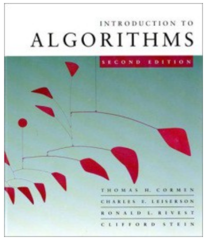




Example of an OS-tree



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



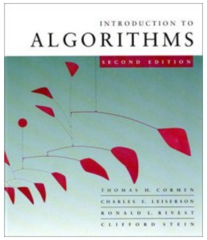
Selection

Implementation trick: Use a ***sentinel*** (dummy record) for NIL such that $size[\text{NIL}] = 0$.

OS-SELECT(x, i) $\triangleright$ ith smallest element in the subtree rooted at x

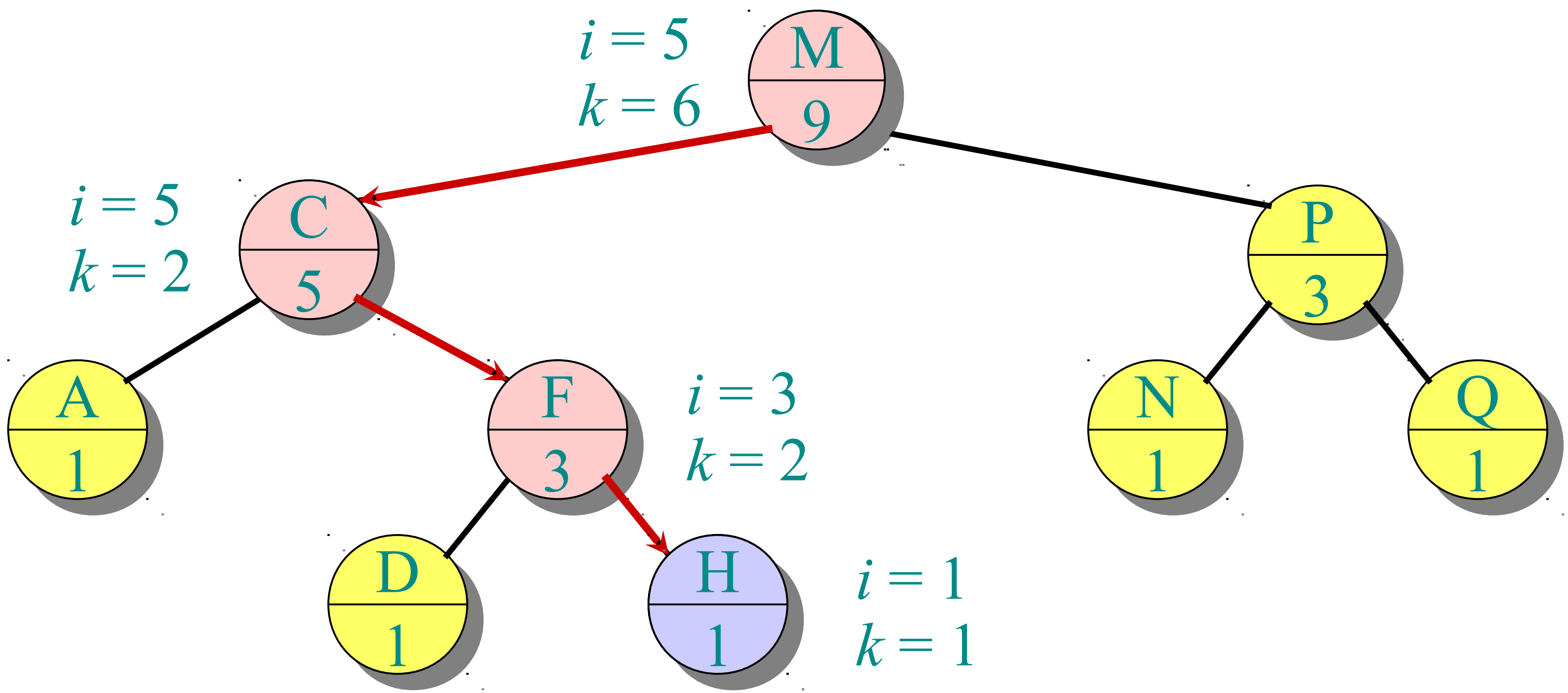
$k \leftarrow size[left[x]] + 1$ $\triangleright$ $k = \text{rank}(x)$
if $i = k$ **then return** x
if $i < k$
  then return OS-SELECT$(left[x], i)$
  else return OS-SELECT$(right[x], i - k)$

(OS-RANK is in the textbook.)



Example

OS-SELECT(*root*, 5)



Running time $= O(h) = O(\lg n)$ for red-black trees.

Data structure maintenance

Q. Why not keep the ranks themselves in the nodes instead of subtree sizes?

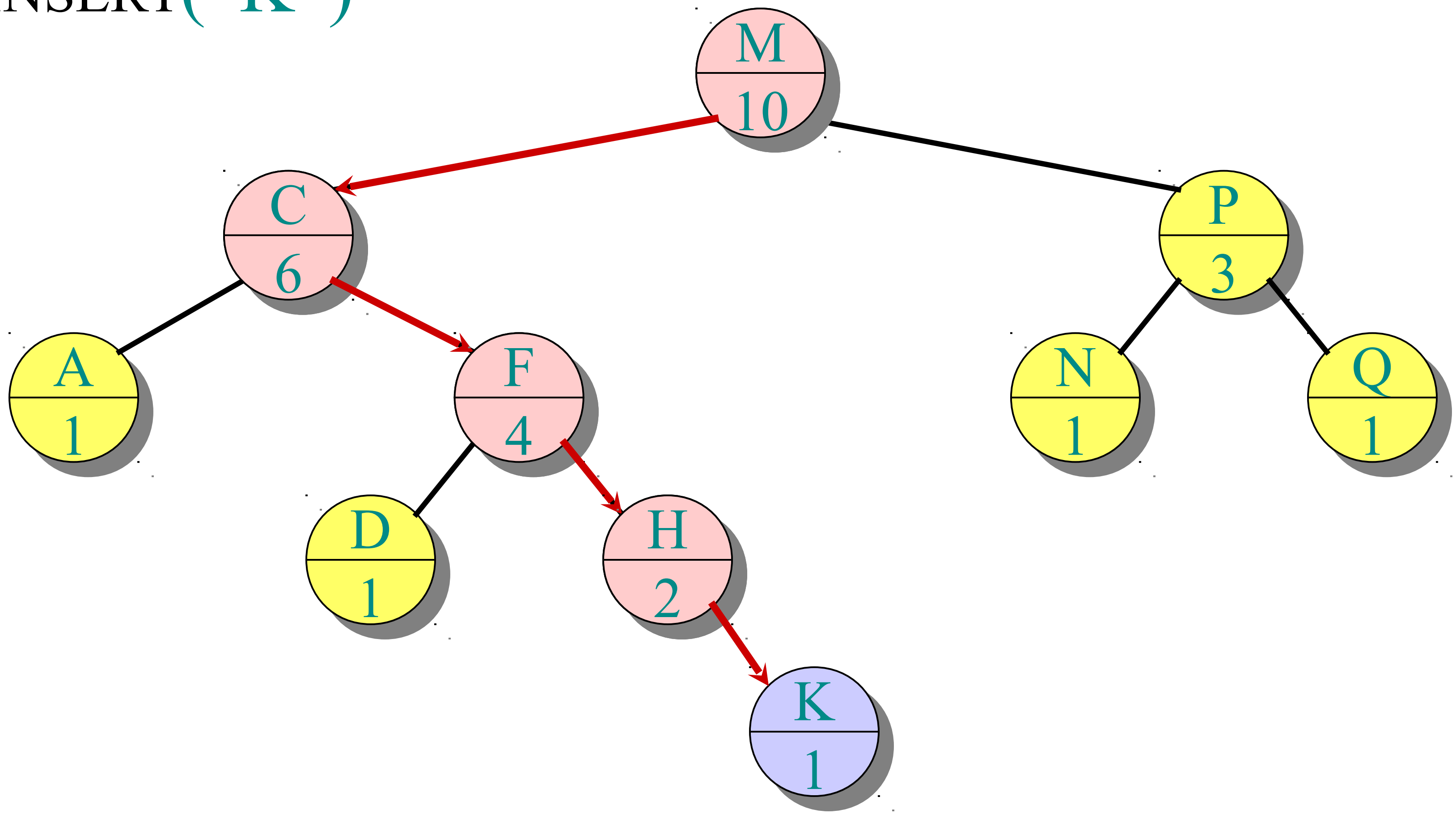
A. They are hard to maintain when the red-black tree is modified.

Modifying operations: INSERT and DELETE.

Strategy: Update subtree sizes when inserting or deleting.

Example of insertion

INSERT("K")

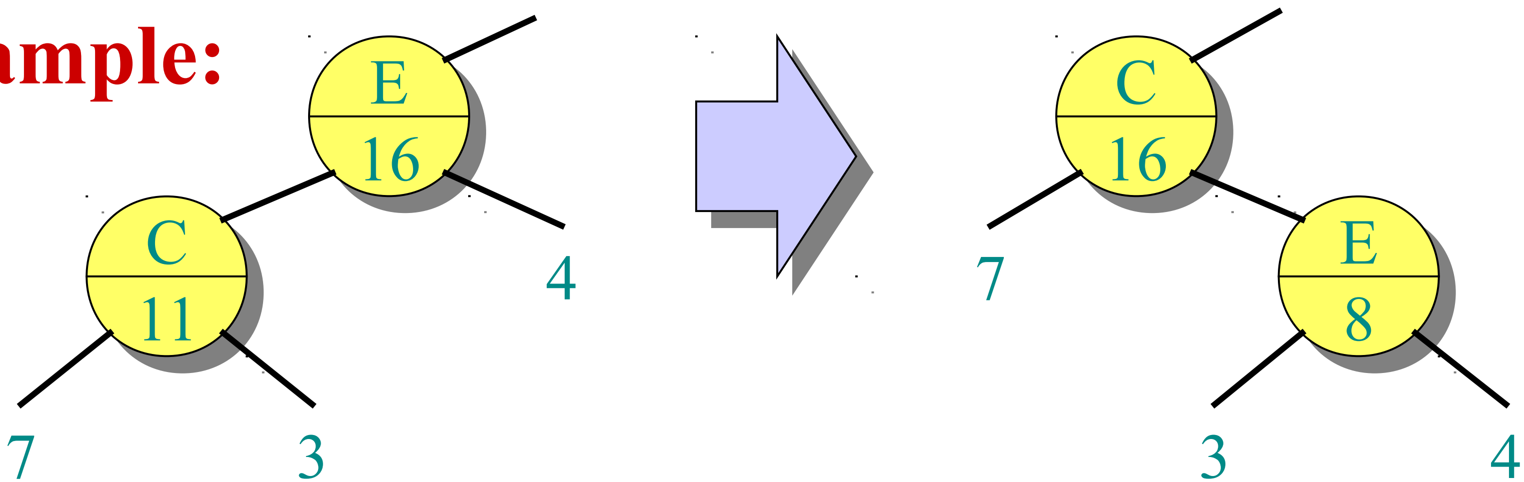


Handling rebalancing

Don't forget that RB-INSERT and RB-DELETE may also need to modify the red-black tree in order to maintain balance.

- *Recolorings*: no effect on subtree sizes.
- *Rotations*: fix up subtree sizes in $O(1)$ time.

Example:



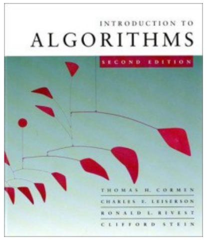
∴ RB-INSERT and RB-DELETE still run in $O(\lg n)$ time.

Data-structure augmentation

Methodology: (*e.g., order-statistics trees*)

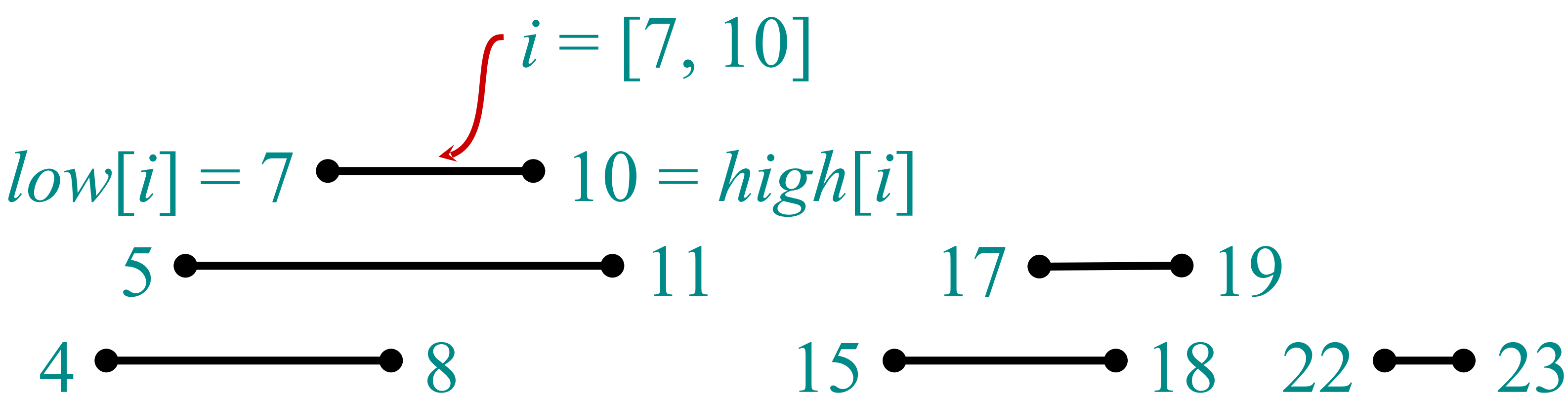
1. Choose an underlying data structure (*red-black trees*).
2. Determine additional information to be stored in the data structure (*subtree sizes*).
3. Verify that this information can be maintained for modifying operations (*RB-Insert, RB-Delete — don't forget rotations*).
4. Develop new dynamic-set operations that use the information (*OS-Select and OS-Rank*).

These steps are guidelines, not rigid rules.

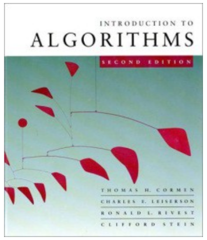


Interval trees

Goal: To maintain a dynamic set of intervals, such as time intervals.

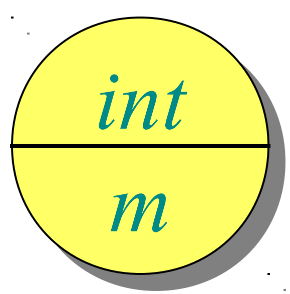


Query: For a given query interval i, find an interval in the set that overlaps i.

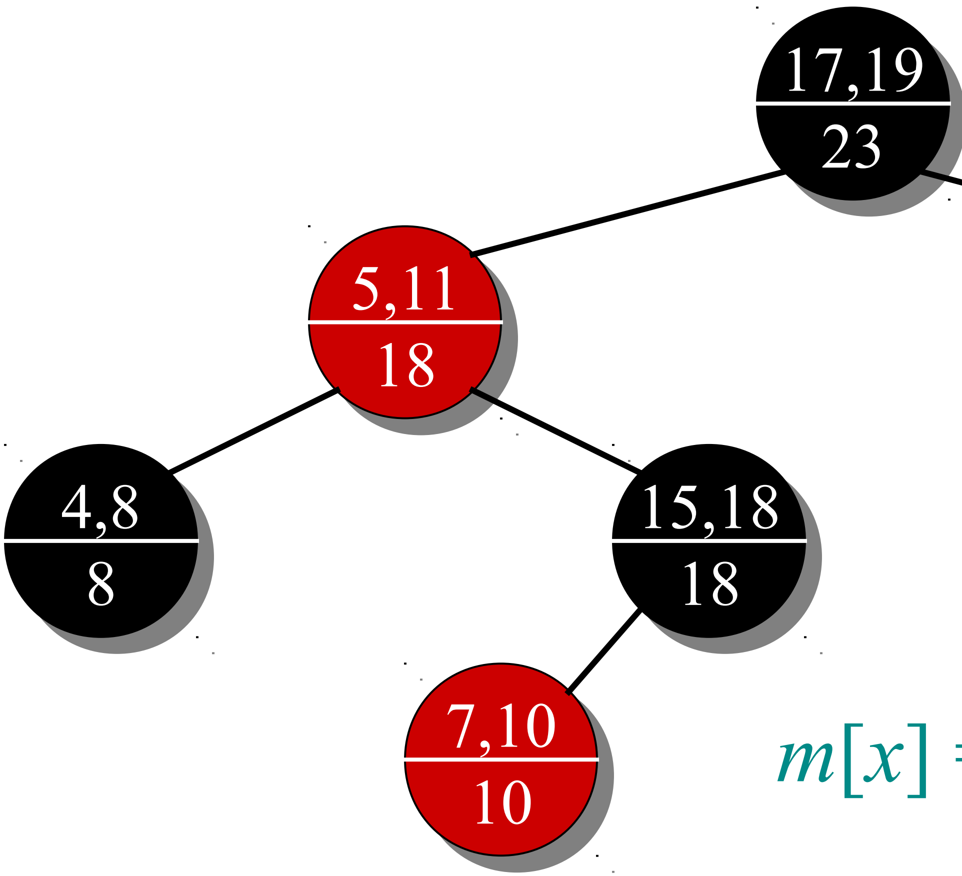


Following the methodology

1. *Choose an underlying data structure.*
   - Red-black tree keyed on low (left) endpoint.
2. *Determine additional information to be stored in the data structure.*
   - Store in each node x the largest value $m[x]$ in the subtree rooted at x, as well as the interval $int[x]$ corresponding to the key.



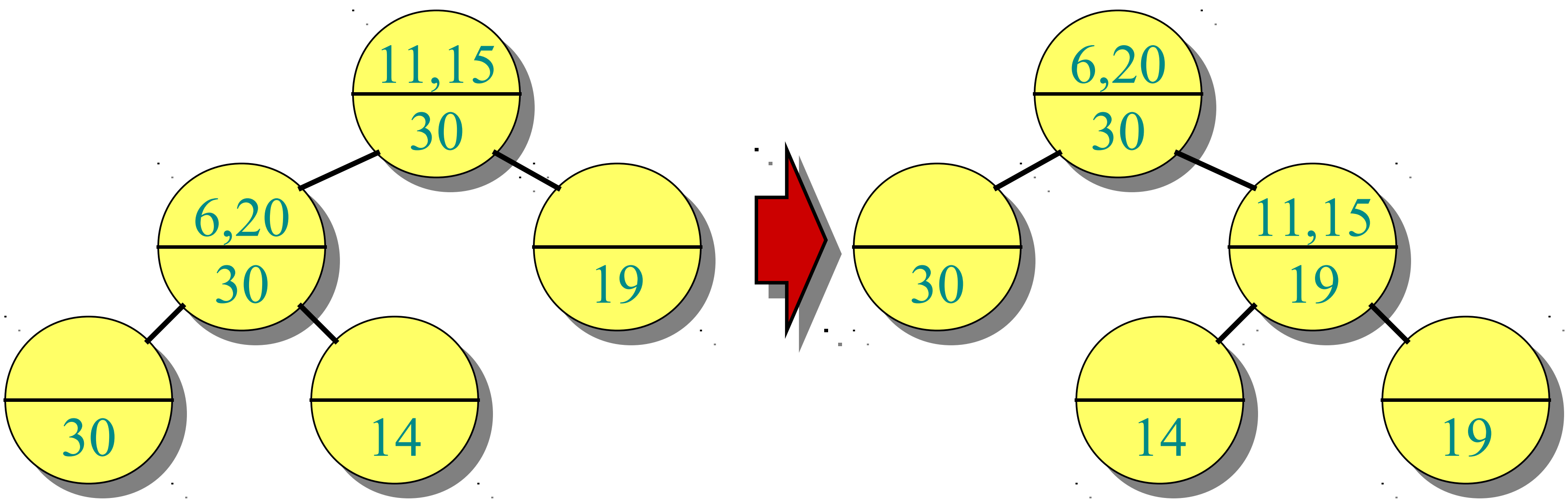
Example interval tree



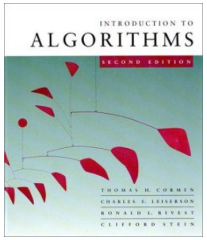
$$m[x] = \max \begin{cases} high[int[x]] \\ m[left[x]] \\ m[right[x]] \end{cases}$$

Modifying operations

3. *Verify that this information can be maintained for modifying operations.*
   - INSERT: Fix m's on the way down.
   - Rotations — Fixup $= O(1)$ time per rotation:



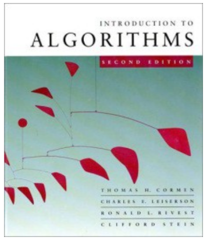
Total INSERT time $= O(\lg n)$; DELETE similar.



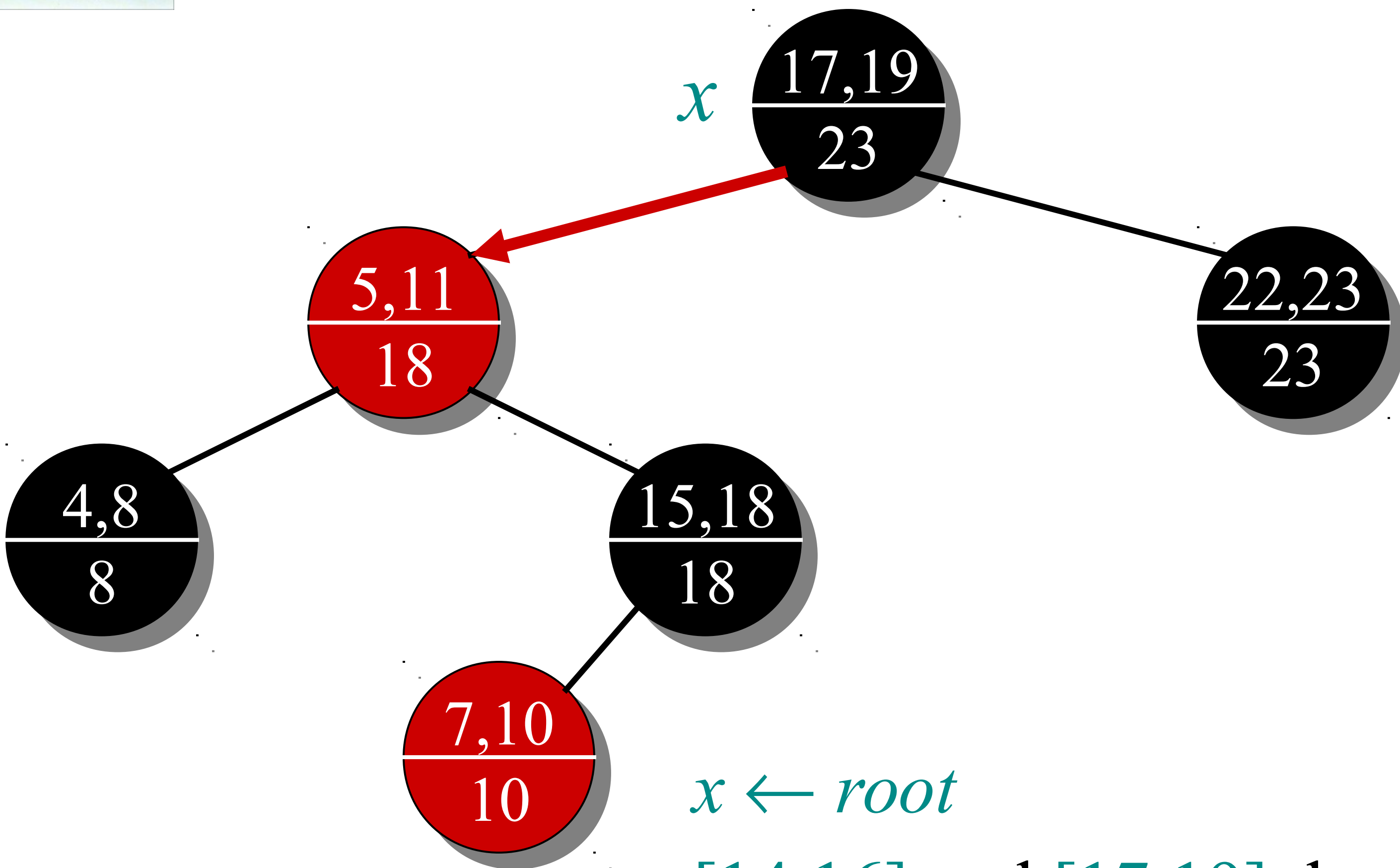
New operations

4. Develop new dynamic-set operations that use the information.

```
INTERVAL-SEARCH(i)
    x ← root
    while x ≠ NIL and (low[i] > high[int[x]]
                         or low[int[x]] > high[i])
        do ▷ i and int[x] don't overlap
           if left[x] ≠ NIL and low[i] ≤ m[left[x]]
              then x ← left[x]
              else x ← right[x]
    return x
```



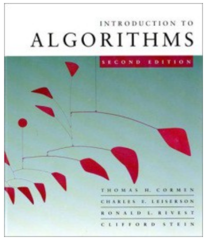
Example 1: INTERVAL-SEARCH([14,16])



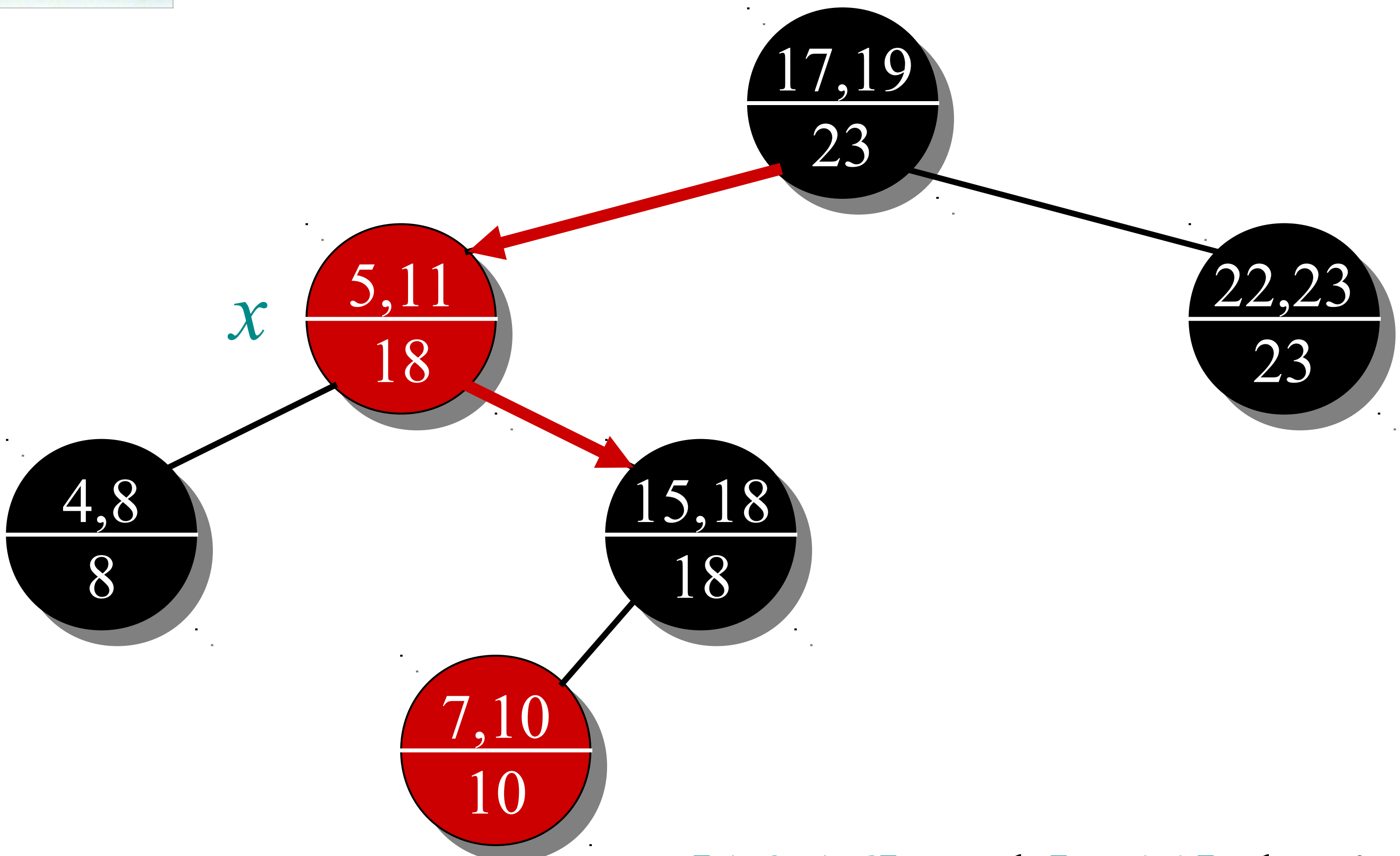
$x \leftarrow root$

[14,16] and [17,19] don't overlap

$14 \leq 18 \Rightarrow x \leftarrow left[x]$



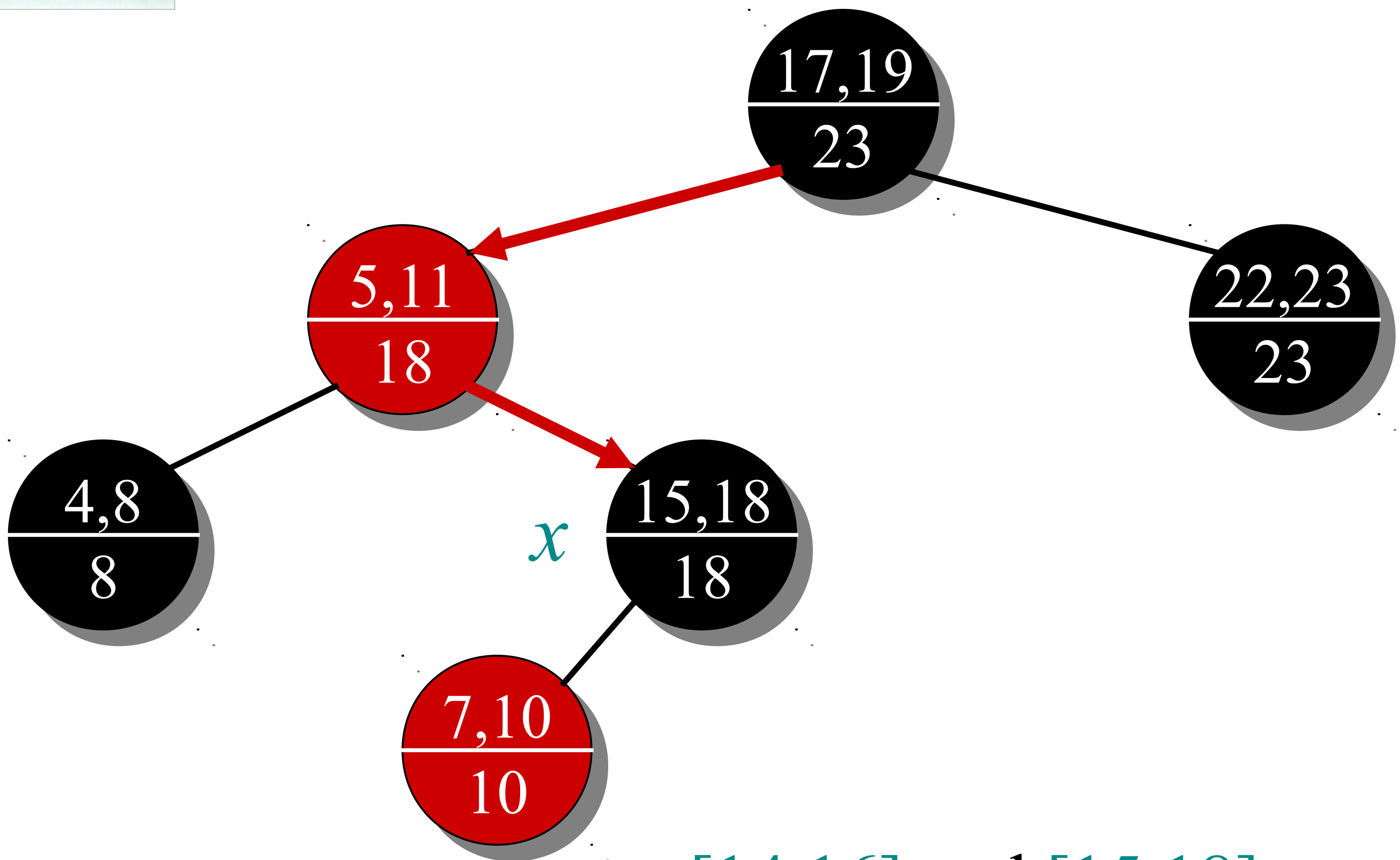
Example 1: INTERVAL-SEARCH([14,16])



[14,16] and [5,11] don't overlap

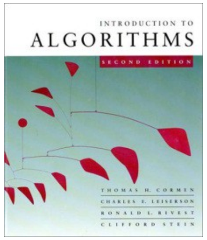
$14 > 8 \Rightarrow x \leftarrow right[x]$

Example 1: INTERVAL-SEARCH([14,16])

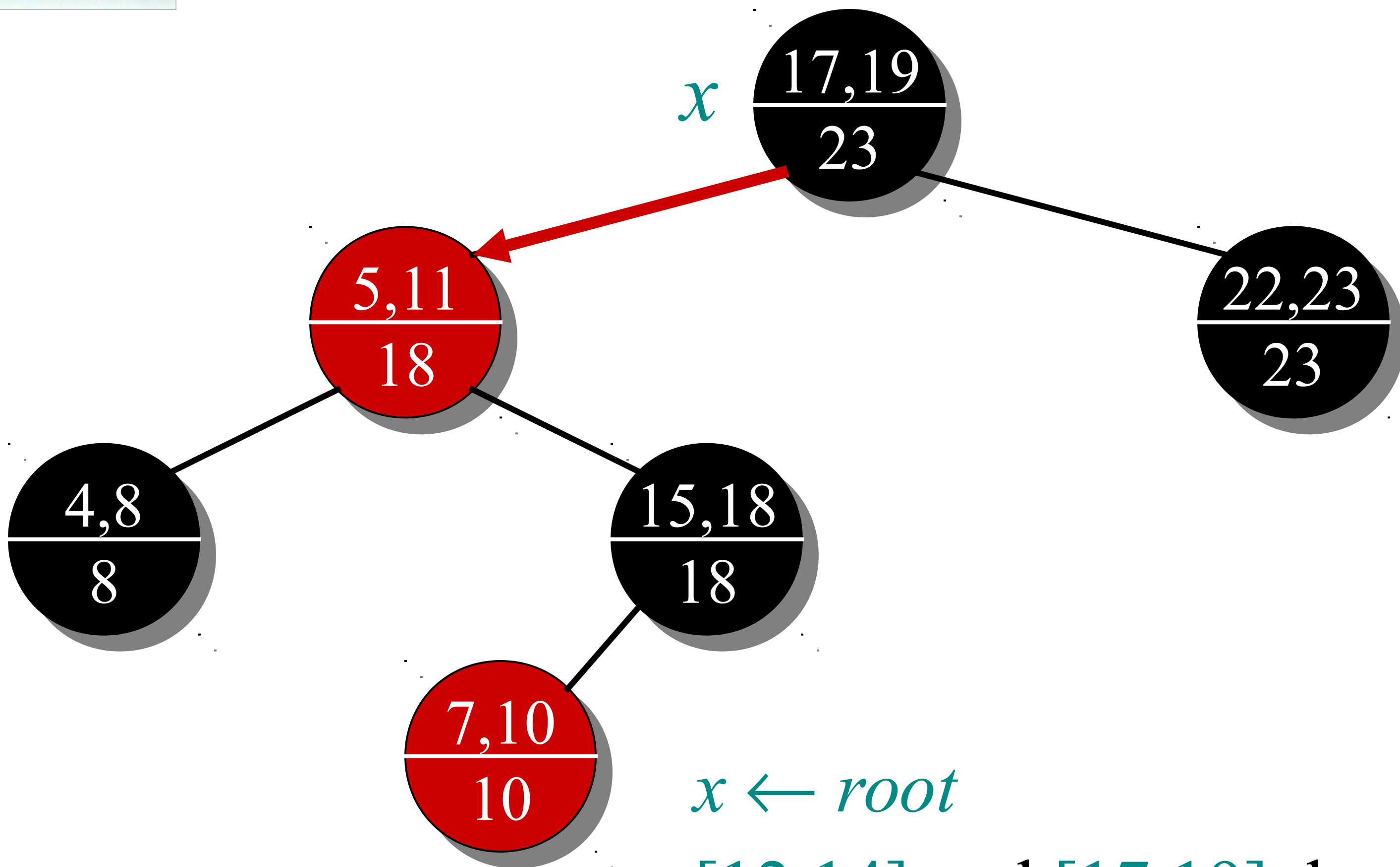


[14,16] and [15,18] overlap

return [15,18]



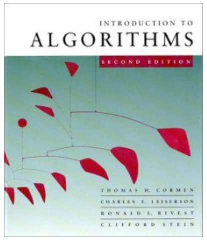
Example 2: INTERVAL-SEARCH([12,14])



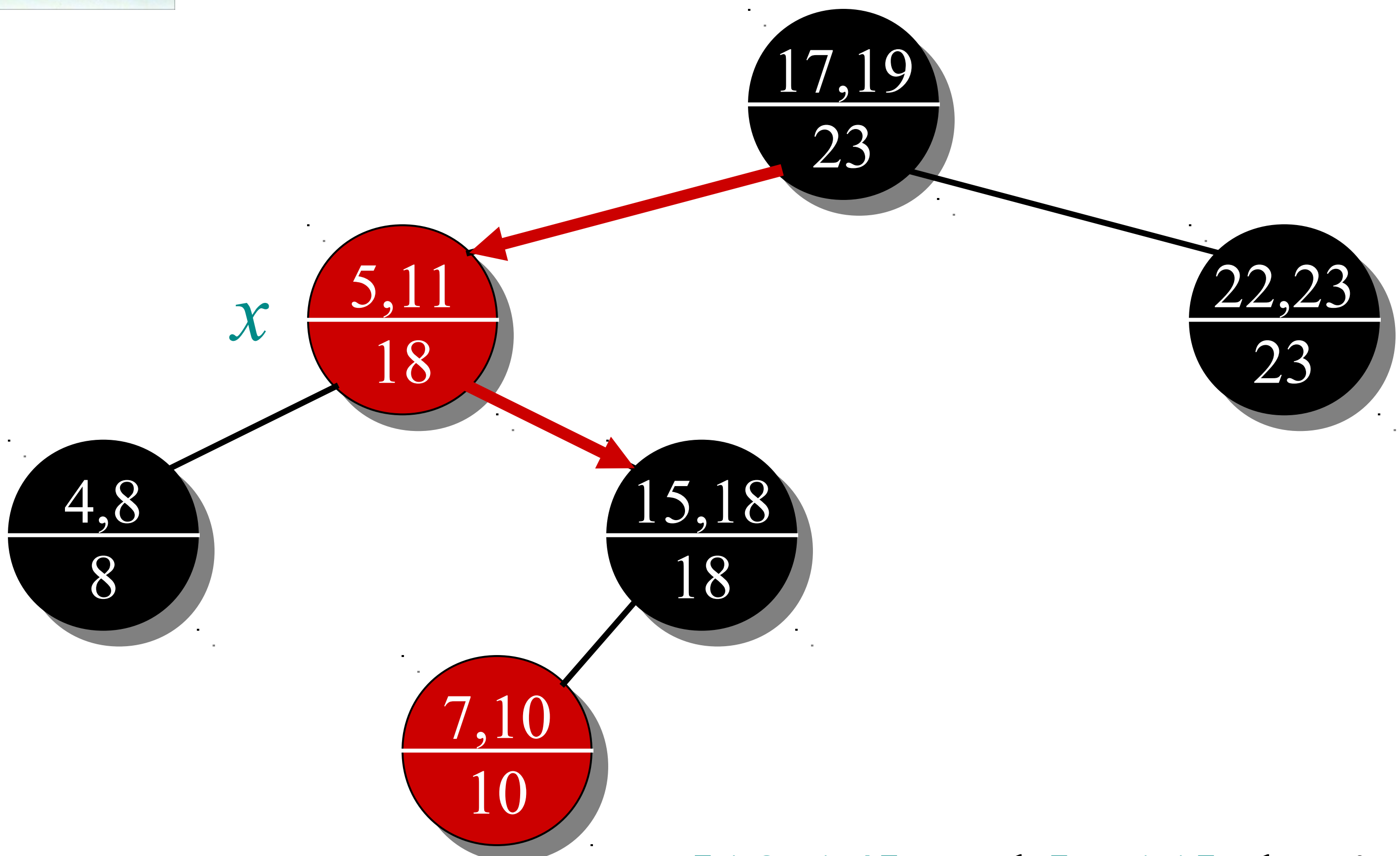
$x \leftarrow root$

[12,14] and [17,19] don't overlap

$12 \leq 18 \Rightarrow x \leftarrow left[x]$



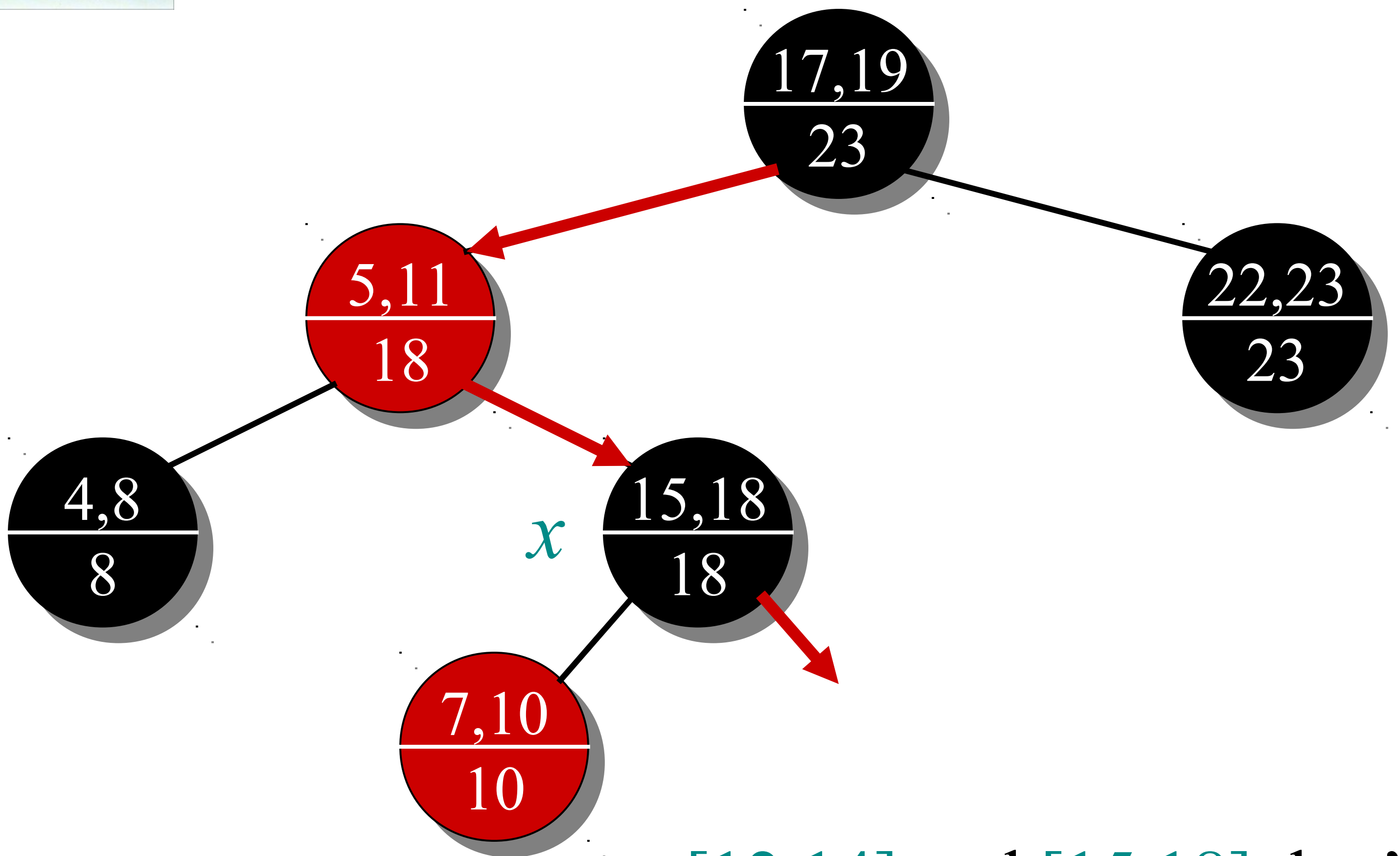
Example 2: INTERVAL-SEARCH([12,14])



[12,14] and [5,11] don't overlap

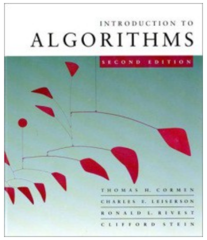
$12 > 8 \Rightarrow x \leftarrow right[x]$

Example 2: INTERVAL-SEARCH([12,14])

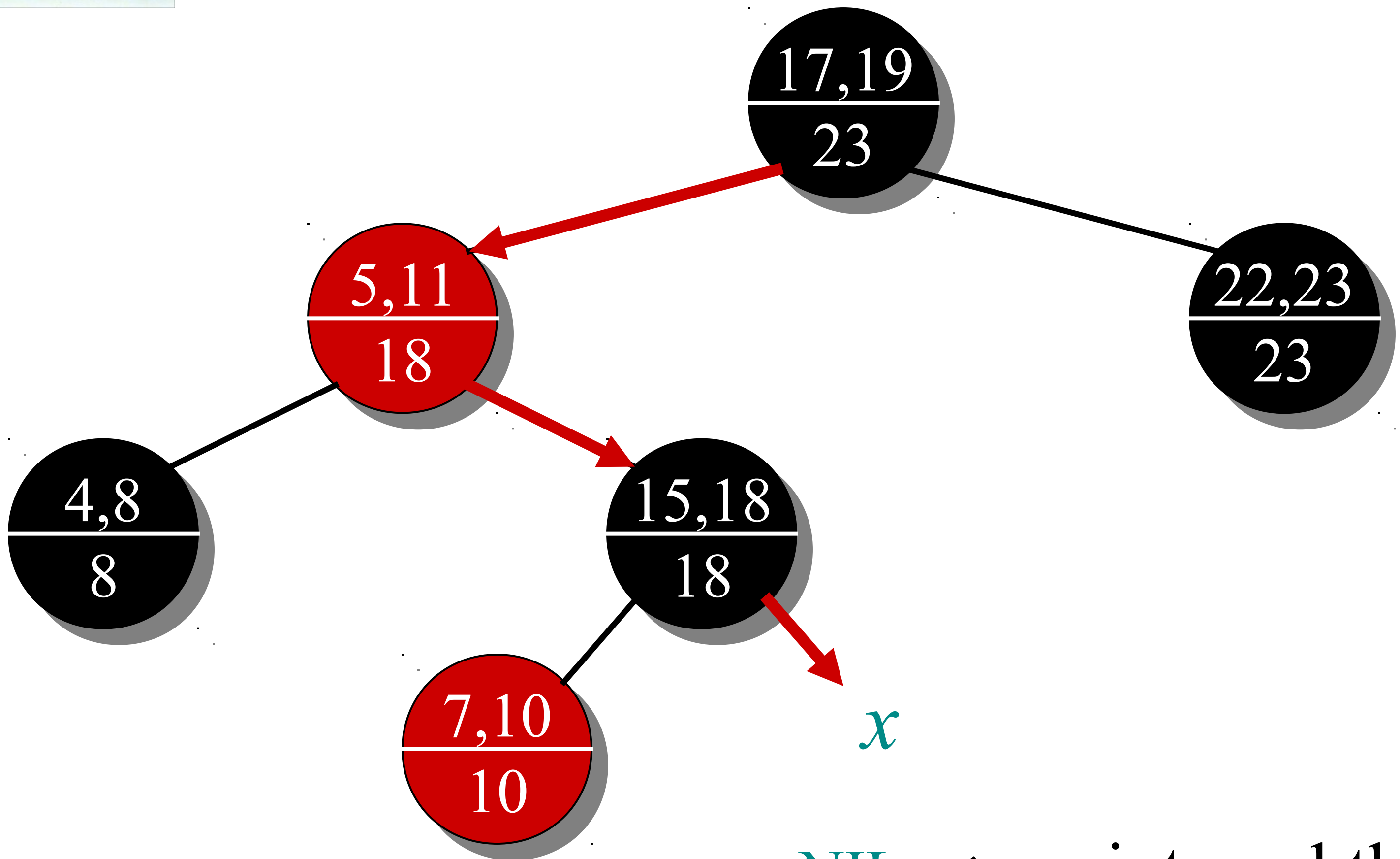


[12,14] and [15,18] don’t overlap

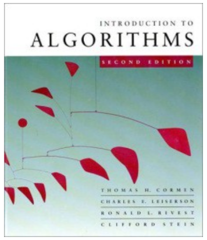
$12 > 10 \Rightarrow x \leftarrow right[x]$



Example 2: INTERVAL-SEARCH([12,14])



x = NIL ⇒ no interval that overlaps [12,14] exists



Analysis

Time $= O(h) = O(\lg n)$, since INTERVAL-SEARCH does constant work at each level as it follows a simple path down the tree.

List *all* overlapping intervals:

- Search, list, delete, repeat.
- Insert them all again at the end.

Time $= O(k \lg n)$, where k is the total number of overlapping intervals.

This is an ***output-sensitive*** bound.

Best algorithm to date: $O(k + \lg n)$.

Correctness

Theorem. Let L be the set of intervals in the left subtree of node x, and let R be the set of intervals in x's right subtree.

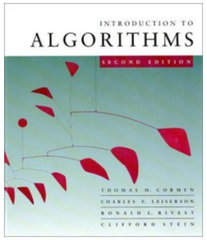
- If the search goes right, then

$$\{ i' \in L : i' \text{ overlaps } i \} = \emptyset .$$

- If the search goes left, then

$$\{i' \in L : i' \text{ overlaps } i \} = \emptyset$$
$$\Rightarrow \{i' \in R : i' \text{ overlaps } i \} = \emptyset .$$

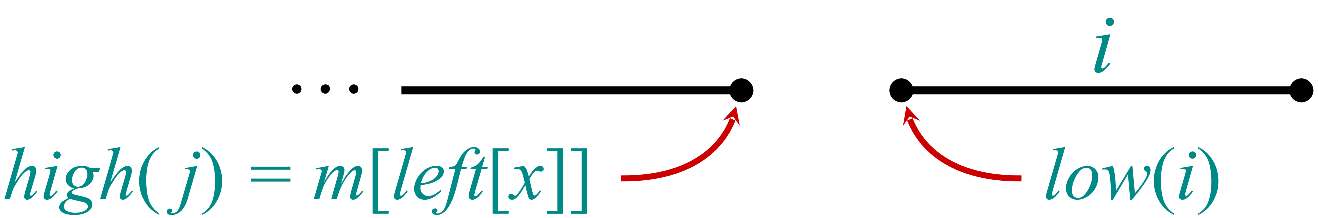
In other words, it's always safe to take only 1 of the 2 children: we'll either find something, or nothing was to be found.



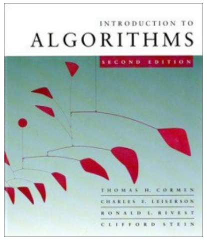
Correctness proof

Proof. Suppose first that the search goes right.

- If $left[x] = \mathrm{NIL}$, then we're done, since $L = \emptyset$.
- Otherwise, the code dictates that we must have $low[i] > m[left[x]]$. The value $m[left[x]]$ corresponds to the right endpoint of some interval $j \in L$, and no other interval in L can have a larger right endpoint than $high(j)$.



- Therefore, $\{i' \in L : i' \text{ overlaps } i\} = \emptyset$.



Proof (continued)

Suppose that the search goes left, and assume that

$\{i' \in L : i'$ overlaps $i\} = \varnothing$.

- Then, the code dictates that $low[i] \le m[left[x]] = high[j]$ for some $j \in L$.
- Since $j \in L$, it does not overlap i, and hence $high[i] < low[j]$.
- But, the binary-search-tree property implies that for all $i' \in R$, we have $low[j] \le low[i']$.
- But then $\{i' \in R : i'$ overlaps $i\} = \varnothing$. ∎