

Design and Analysis of Algorithms

CSE 5311

Lecture 11 Augmenting Data Structures

Song Jiang, Ph.D.

Department of Computer Science and Engineering



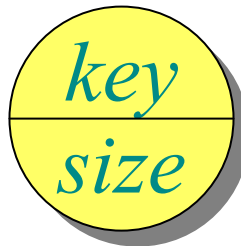
Dynamic order statistics

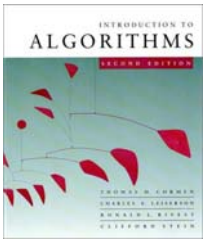
OS-SELECT(i, S): returns the i th smallest element in the dynamic set S .

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

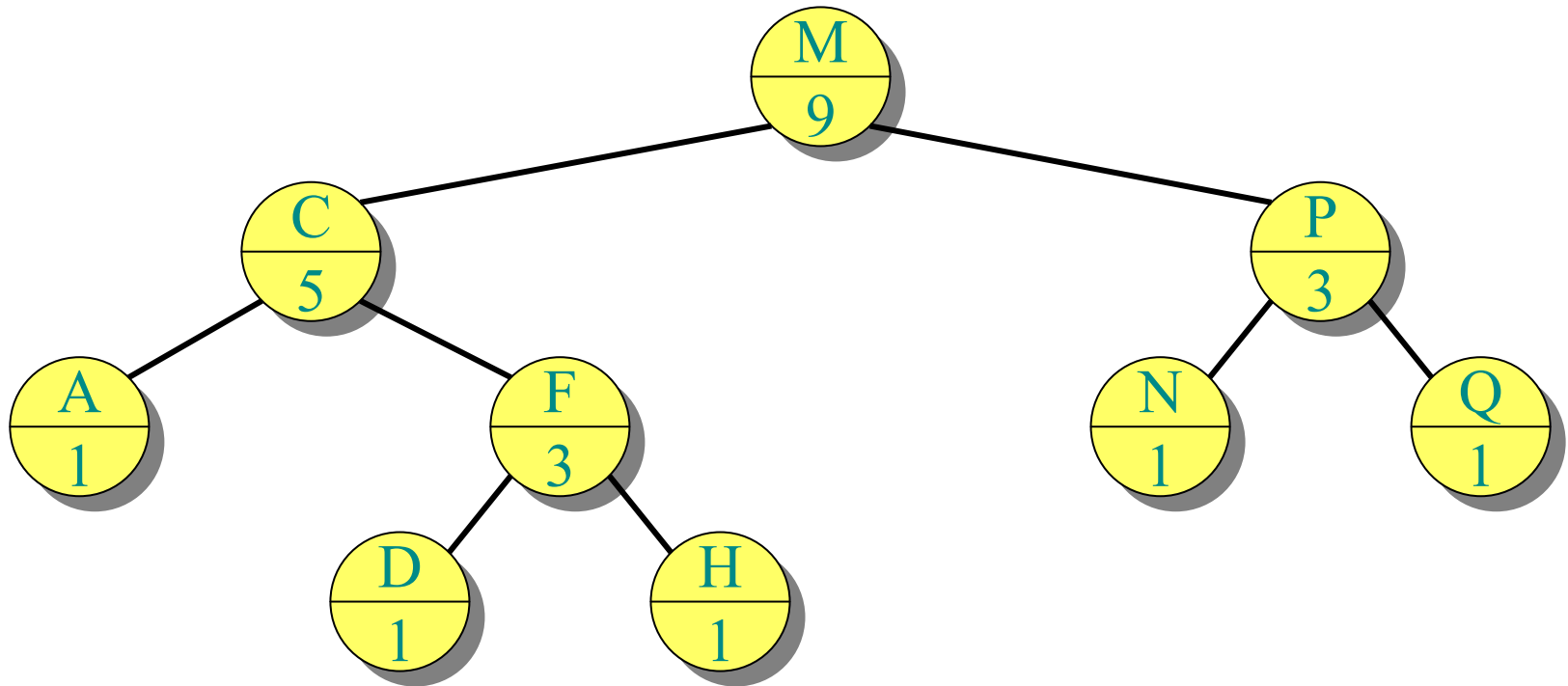
IDEA: Use a red-black tree for the set S , but keep subtree sizes in the nodes.

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 $size[NIL] = 0$.

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

$k \leftarrow size[left[x]] + 1 \quad \triangleright k = rank(x)$

if $i = k$ **then return** x

if $i < k$

then return `OS-SELECT($left[x], i$)`

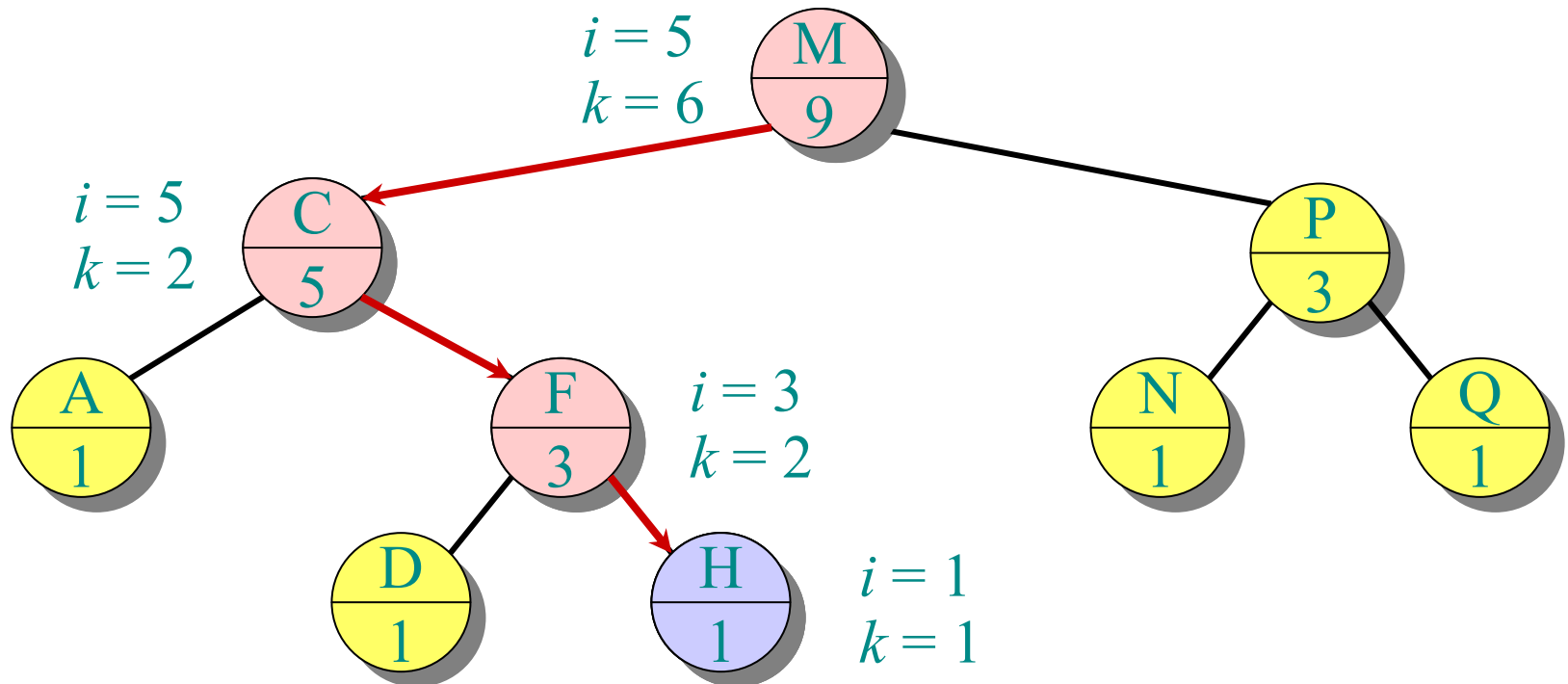
else return `OS-SELECT($right[x], i - k$)`

(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.

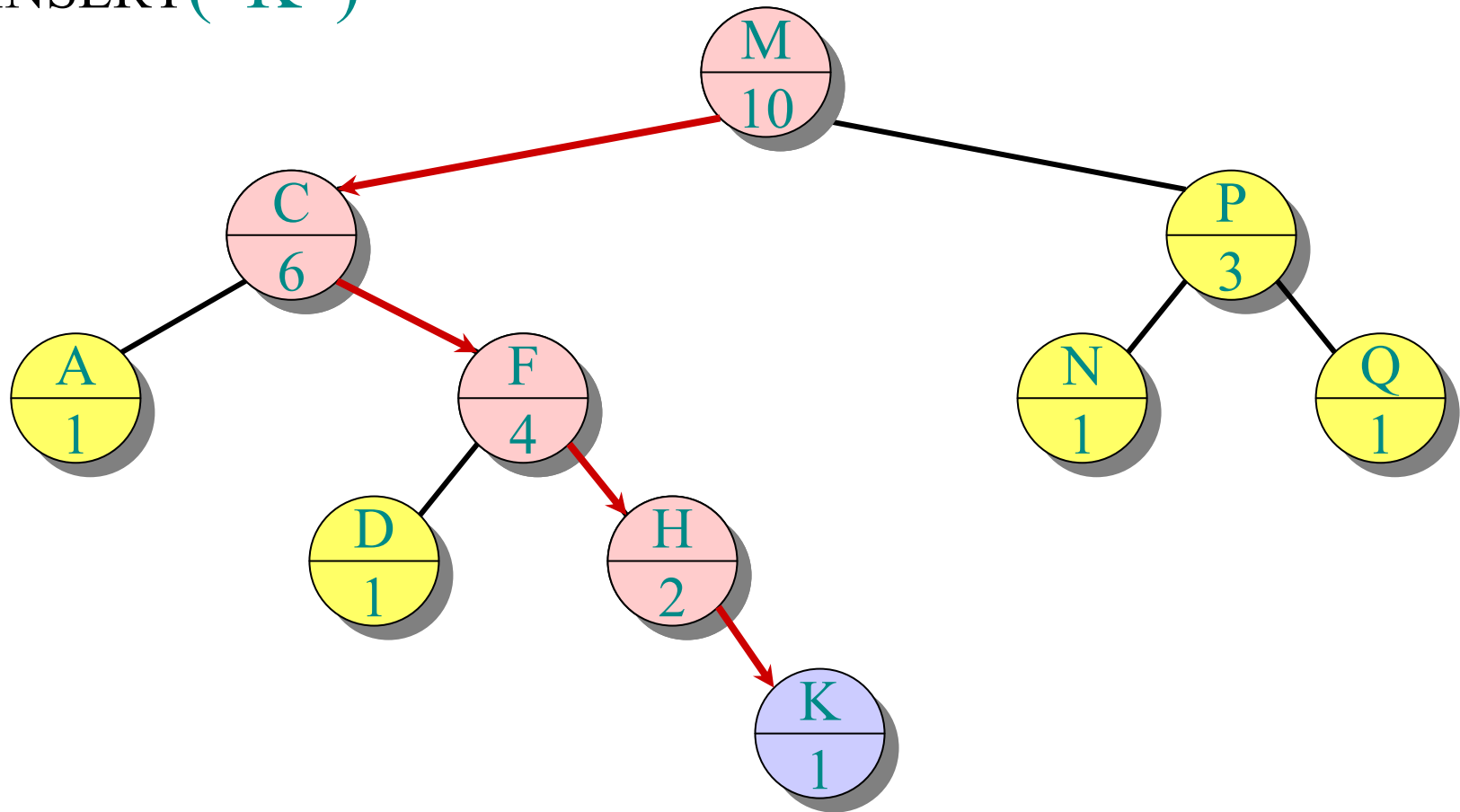
Modifying operations: INSERT and DELETE.

Strategy: Update subtree sizes when inserting or deleting.



Example of insertion

INSERT("K")



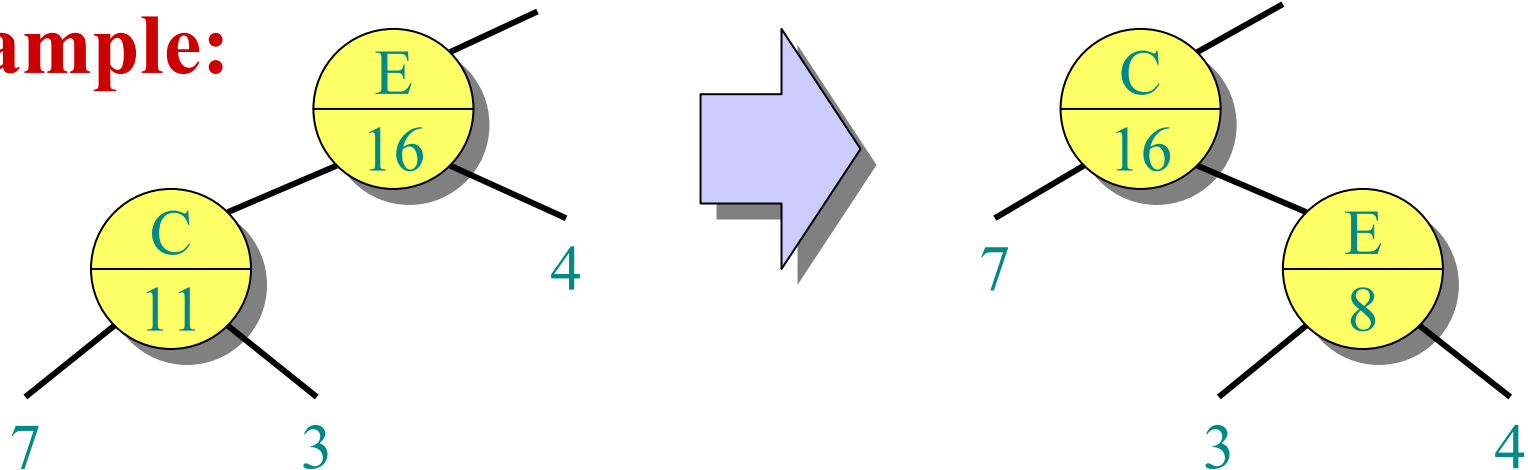


Handling rebalancing

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:



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



Data-structure augmentation

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

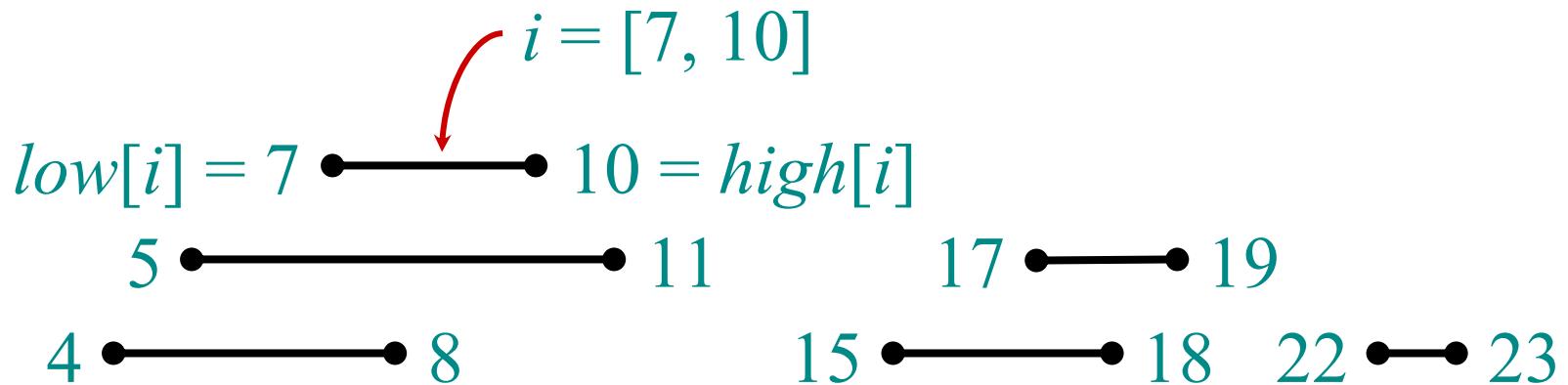
1. Choose an underlying data structure (*red-black trees*).
2. Determine additional information to be stored in the data structure (*subtree sizes*).
3. Verify that this information can be maintained for modifying operations (*RB-INSERT, RB-DELETE — don't forget rotations*).
4. Develop new dynamic-set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

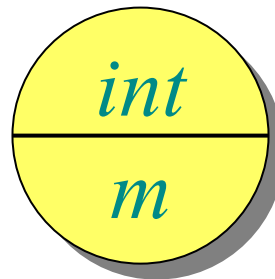


Query: For a given query interval i , find an interval in the set that overlaps i .



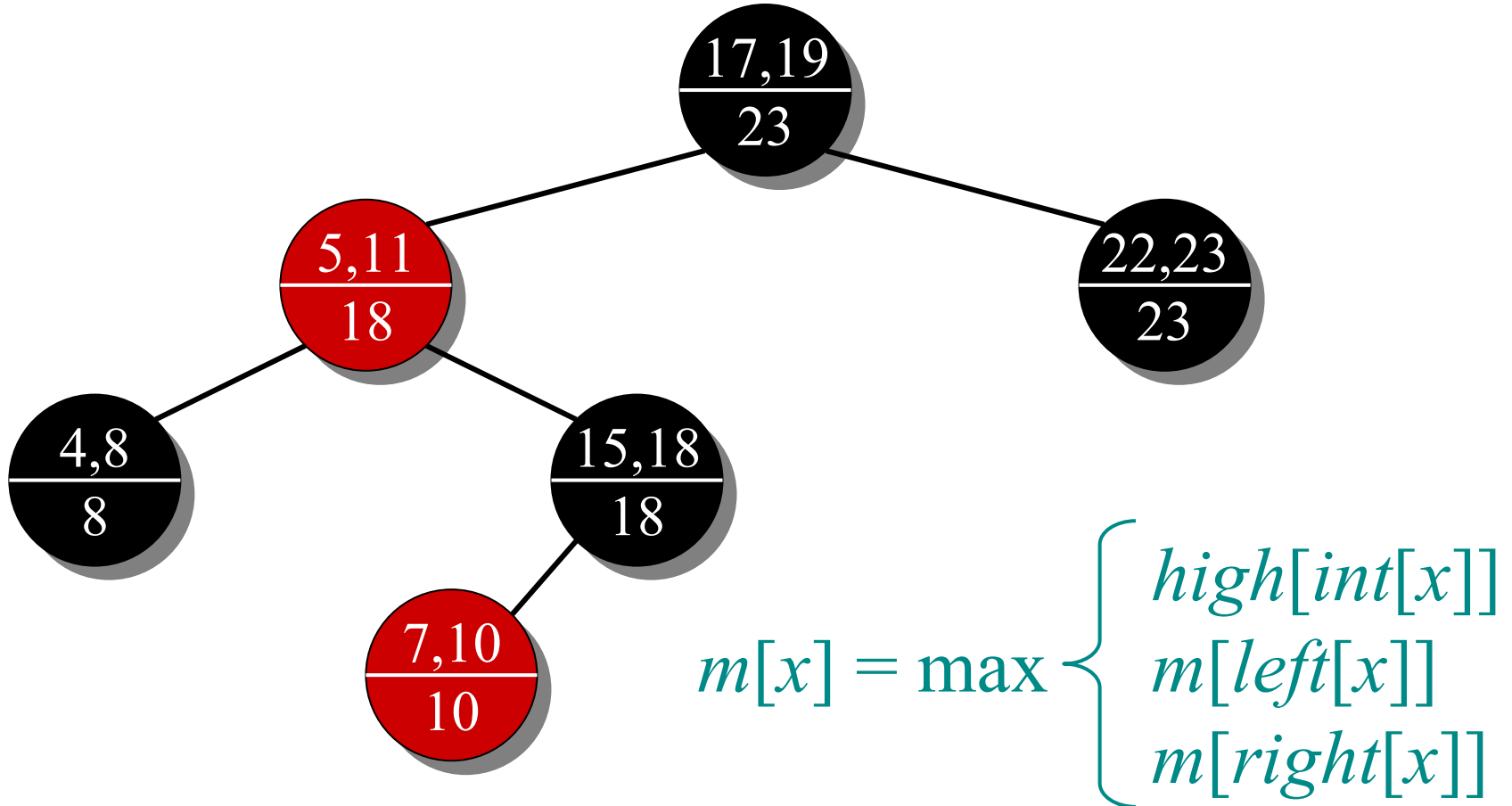
Following the methodology

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

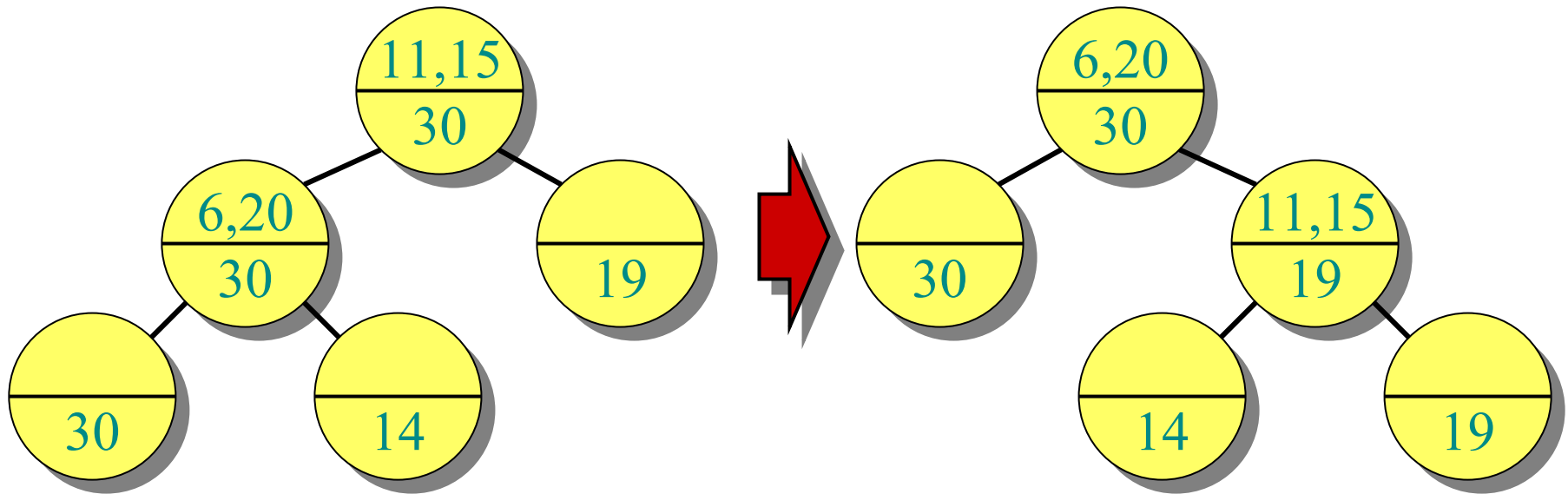




Modifying operations

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)

$$x \leftarrow root$$

```
while  $x \neq \text{NIL}$  and ( $low[i] > high[int[x]]$   
or  $low[int[x]] > high[i]$ )
```

do \triangleright i and $int[x]$ don't overlap

if $left[x] \neq \text{NIL}$ and $low[i] \leq m[left[x]]$

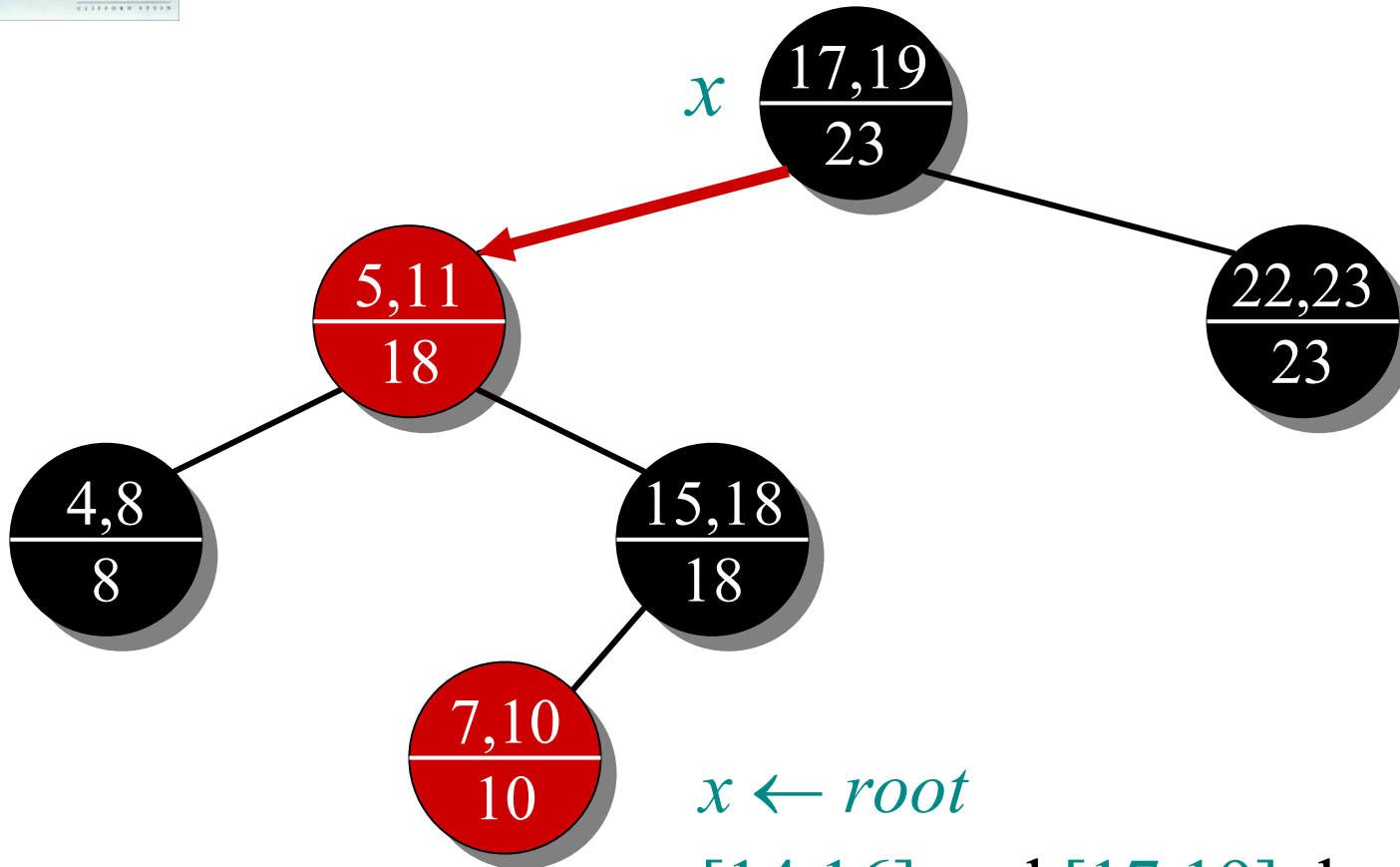
then $x \leftarrow left[x]$

```
else  $x \leftarrow right[x]$ 
```

```
return  $x$ 
```



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



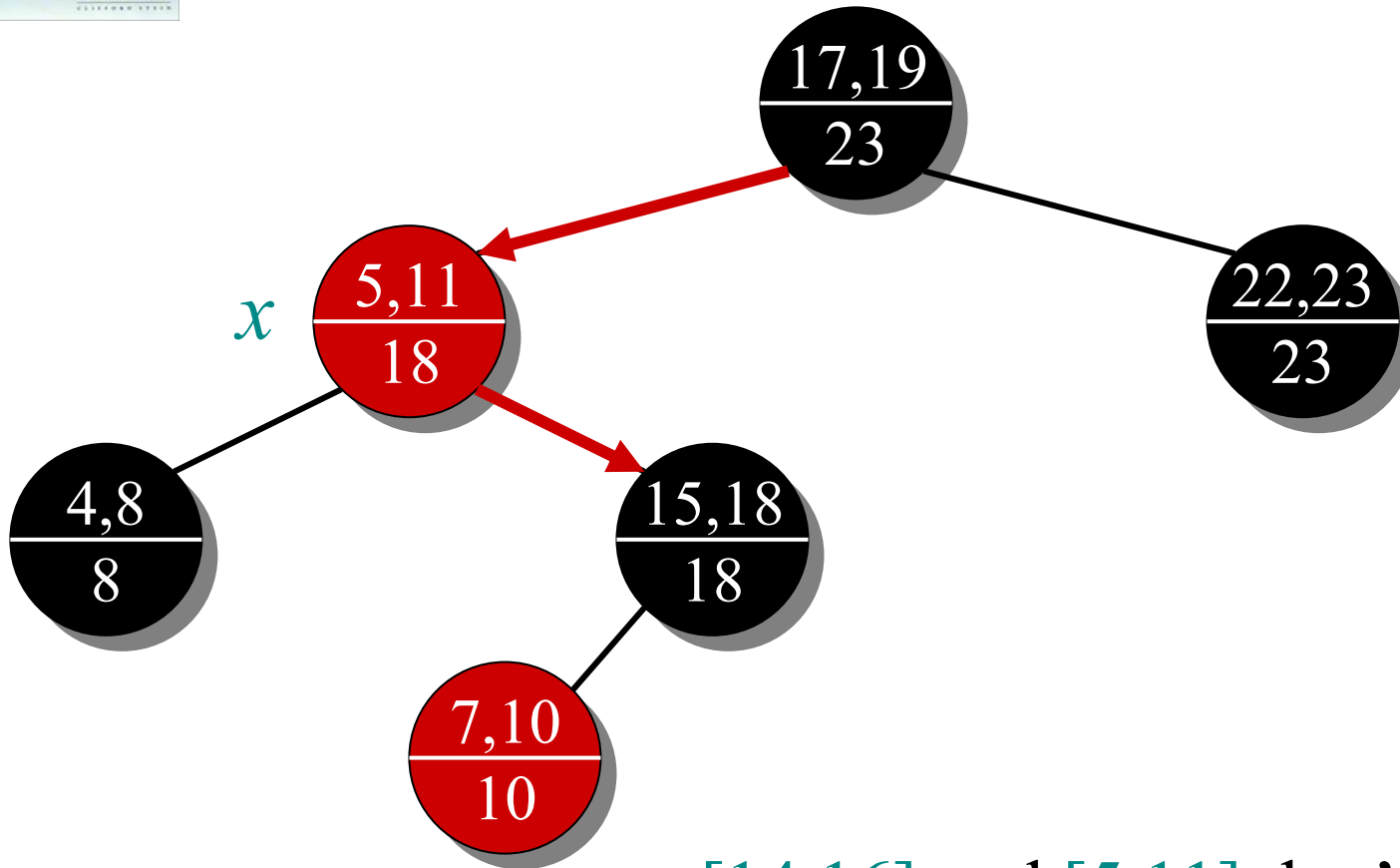
$x \leftarrow \text{root}$

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

$14 \leq 18 \Rightarrow x \leftarrow \text{left}[x]$



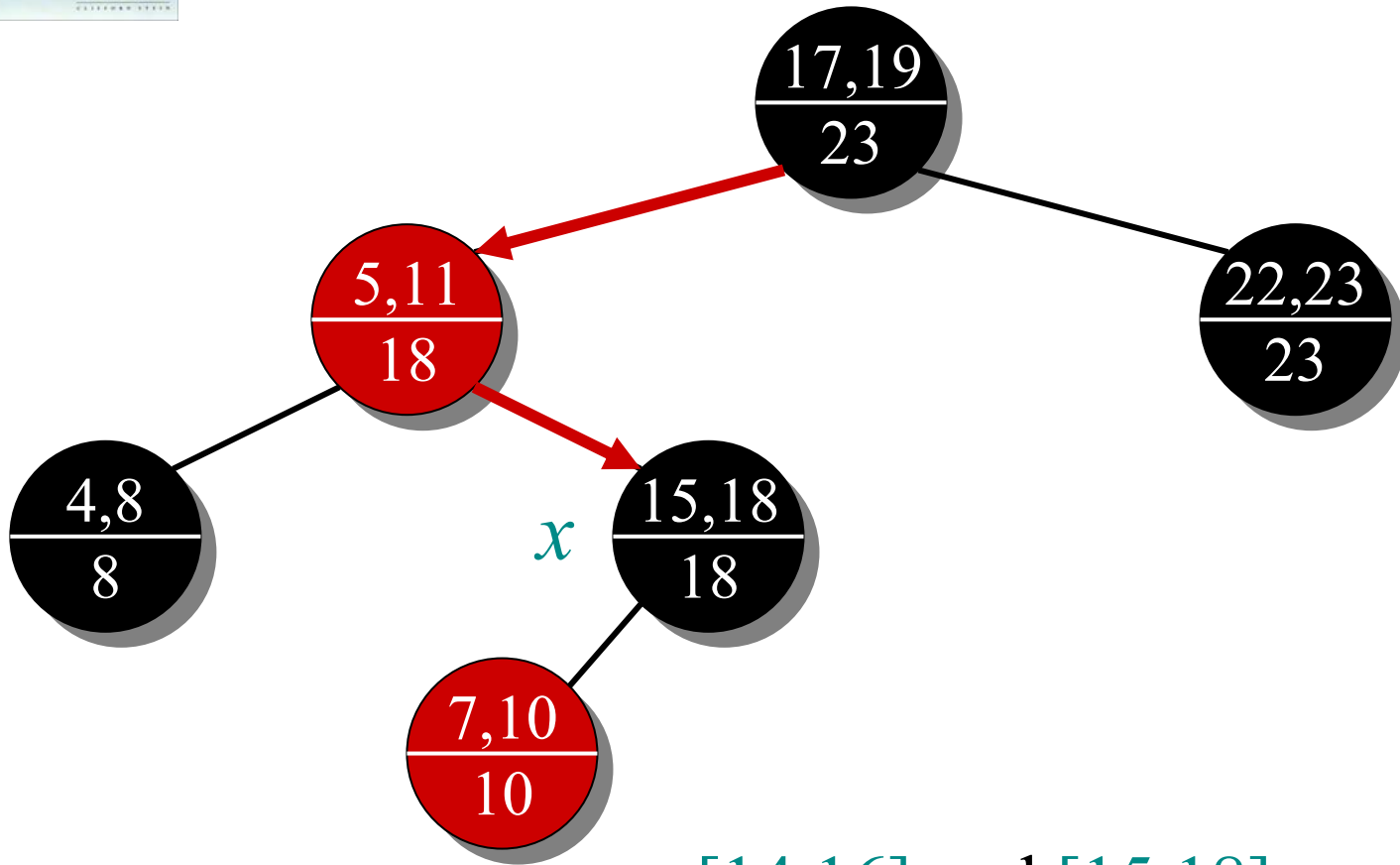
Example 1: INTERVAL-SEARCH([14,16])



[14,16] and [5,11] don't overlap
 $14 > 8 \Rightarrow x \leftarrow \text{right}[x]$



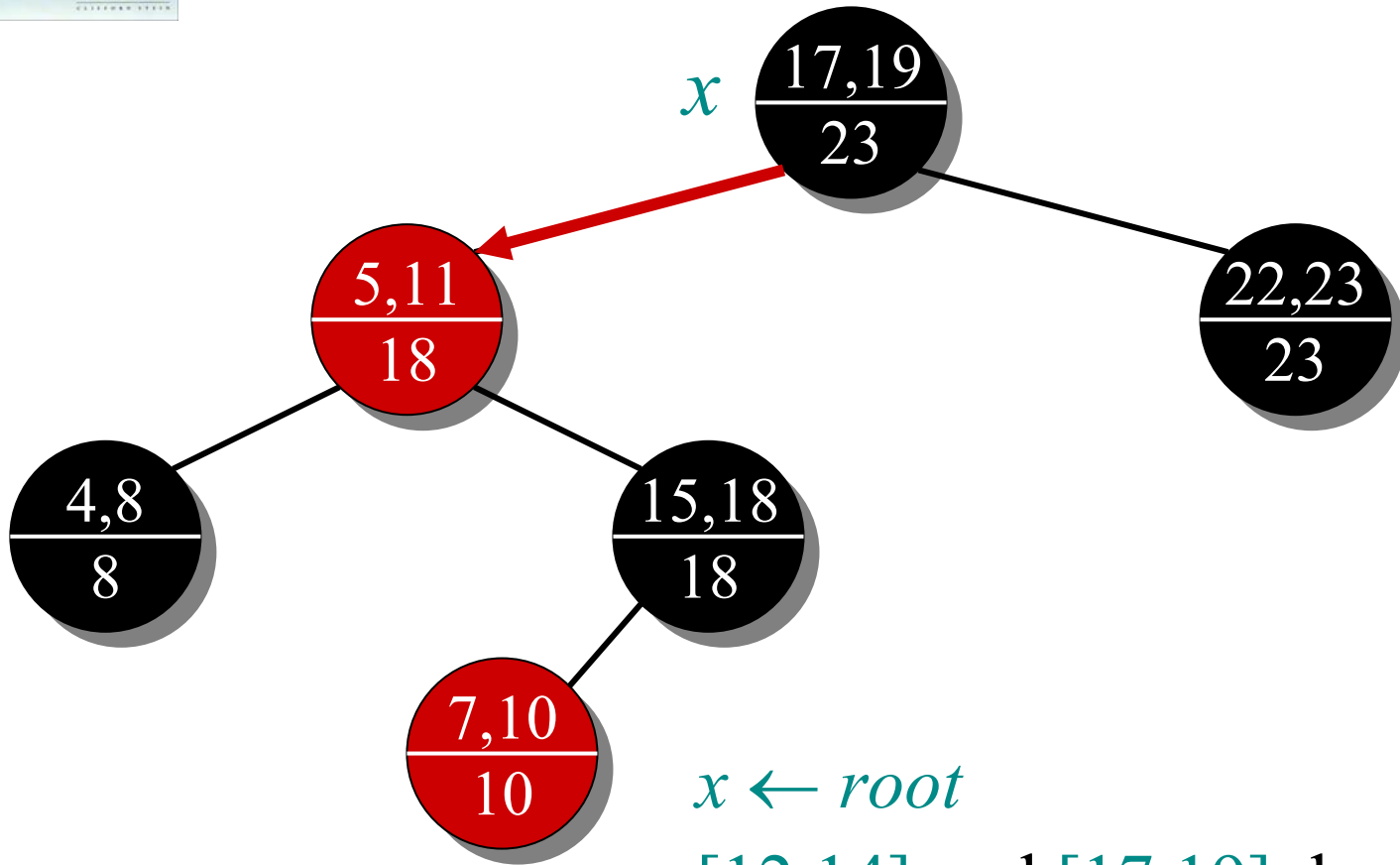
Example 1: INTERVAL-SEARCH([14,16])



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



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



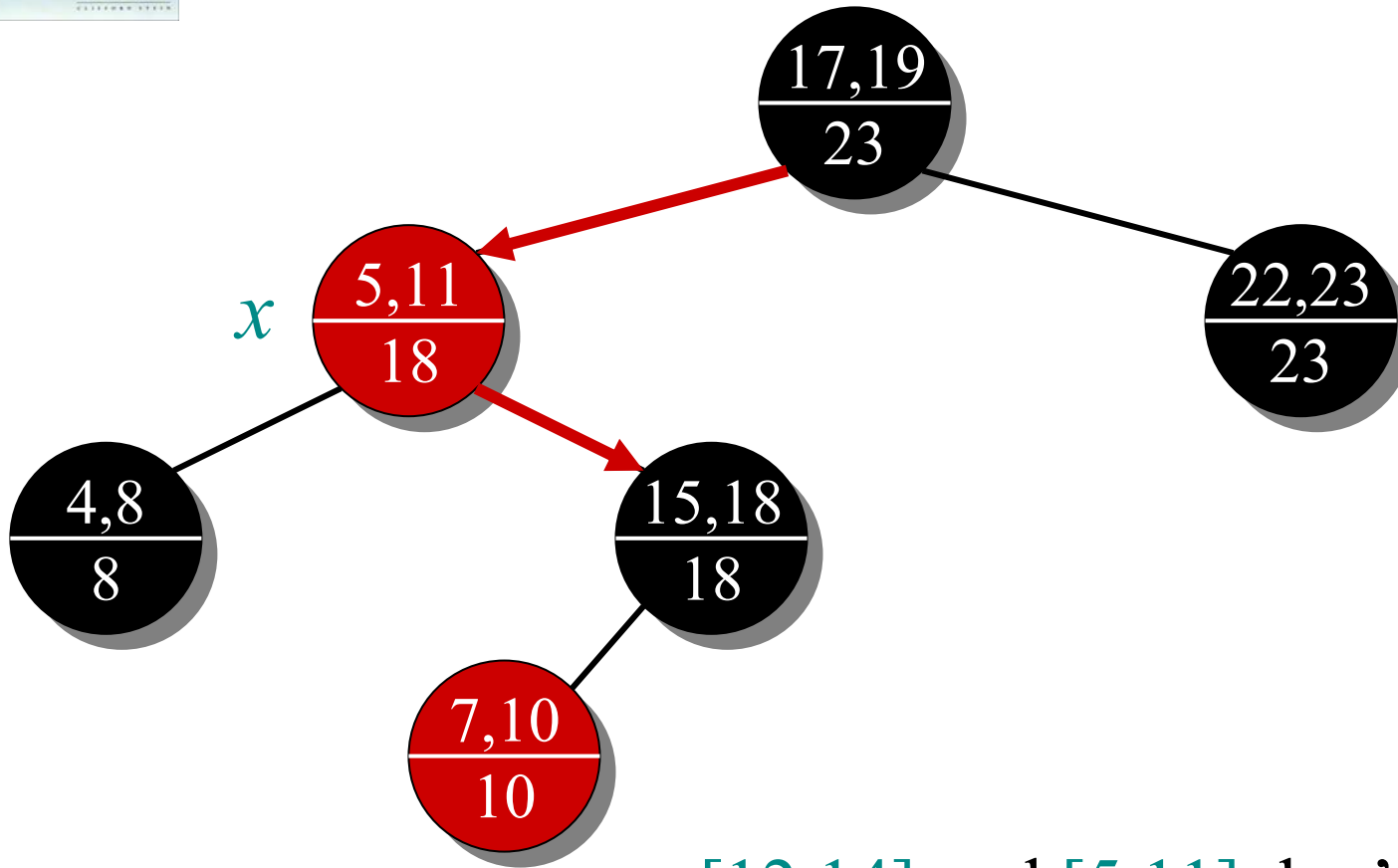
$x \leftarrow \text{root}$

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

$12 \leq 18 \Rightarrow x \leftarrow \text{left}[x]$



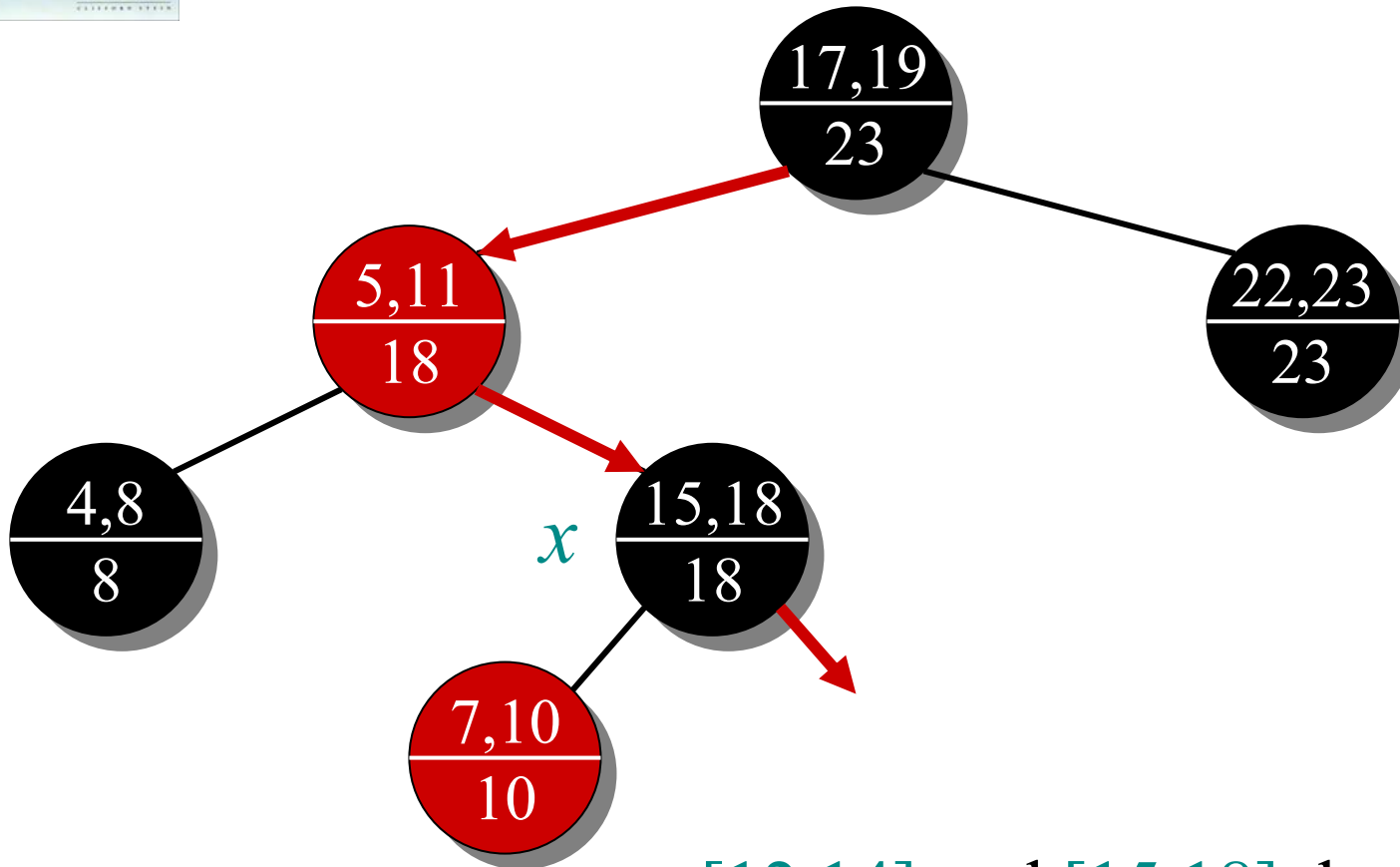
Example 2: INTERVAL-SEARCH([12,14])



[12,14] and [5,11] don't overlap
 $12 > 8 \Rightarrow x \leftarrow \text{right}[x]$



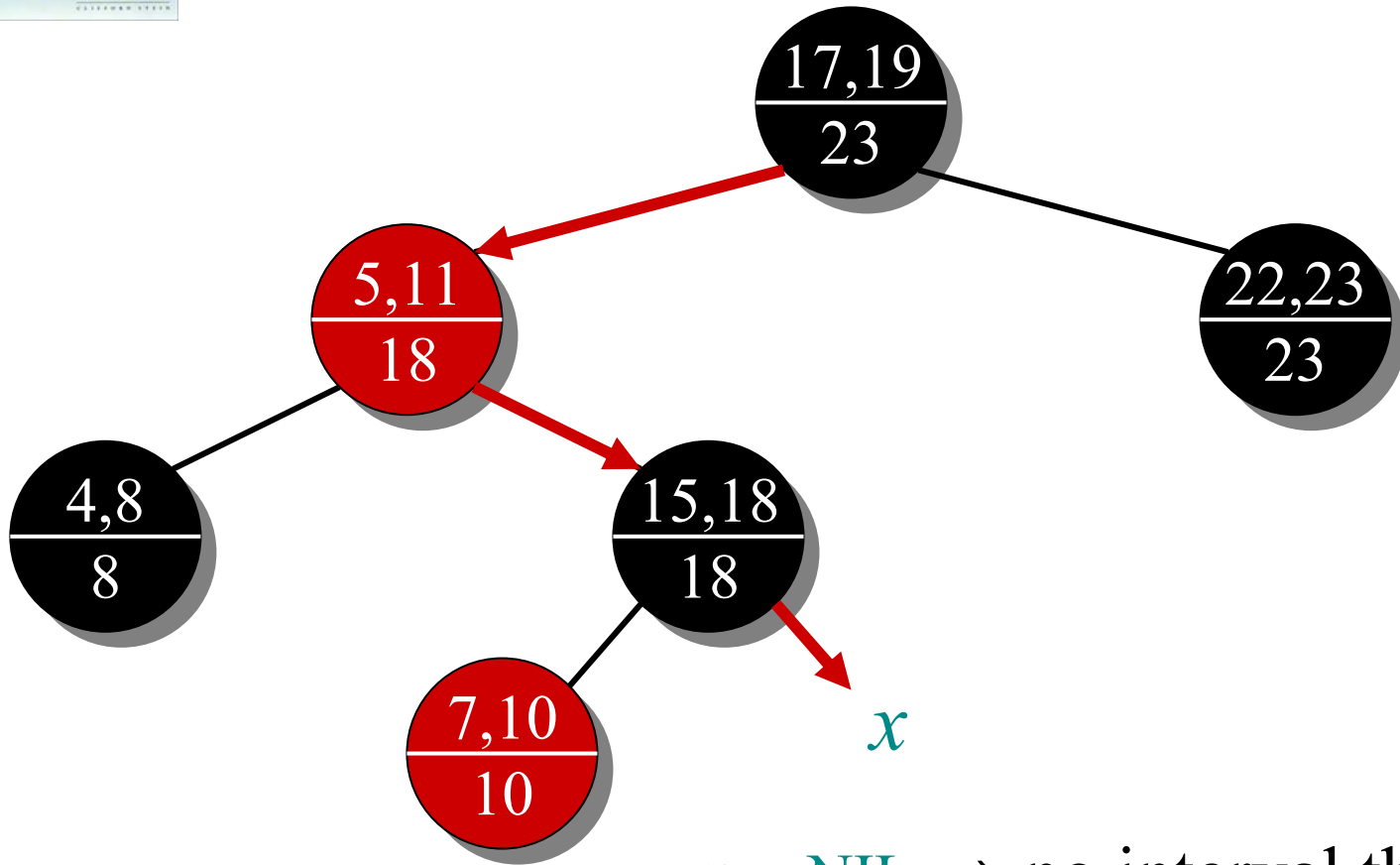
Example 2: INTERVAL-SEARCH([12,14])



$[12,14]$ and $[15,18]$ don't overlap
 $12 > 10 \Rightarrow x \leftarrow \text{right}[x]$



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



$x = \text{NIL} \Rightarrow$ 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.

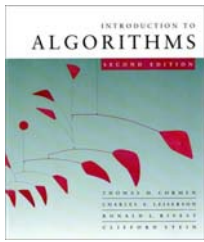
List *all* overlapping intervals:

- 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

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

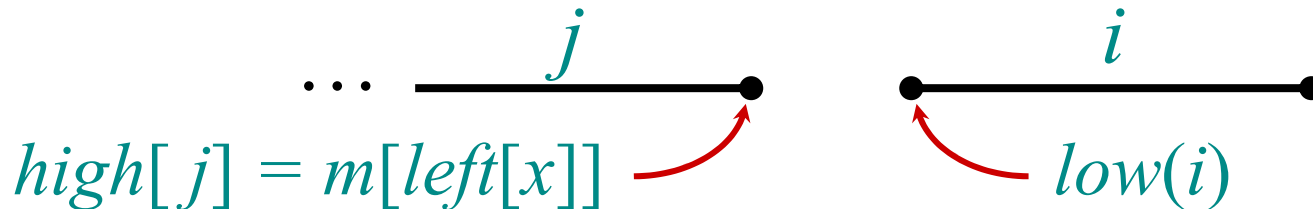
In other words, it's always safe to take only 1 of the 2 children: we'll either find something, or nothing was to be found.



Correctness proof

Proof. Suppose first that the search goes right.

- If $left[x] = \text{NIL}$, 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 high endpoint of some interval $j \in L$, and no other interval in L can have a larger high endpoint than $high[j]$.



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



Proof (continued)

Suppose that the search goes left, and assume that

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

- Then, the code dictates that $low[i] \leq m[left[x]] = high[j]$ 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 $low[j] \leq low[i']$.
- But then $\{i' \in R : i' \text{ overlaps } i\} = \emptyset$. □

