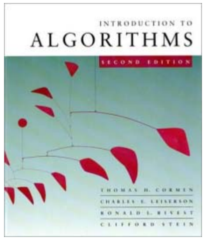
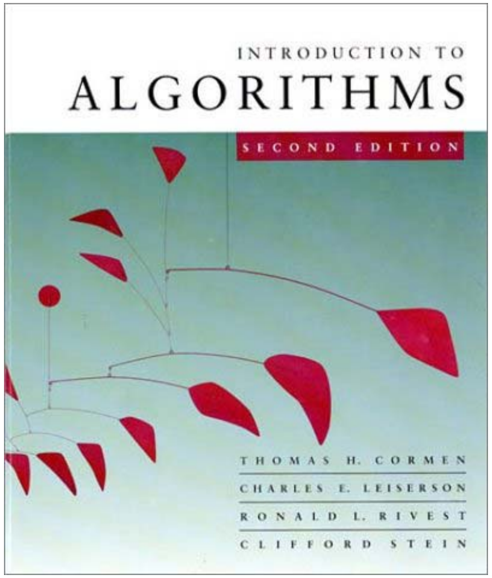


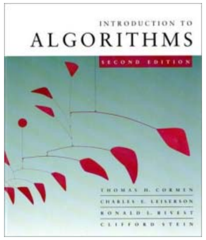
CS 5633 -- Spring 2010



Augmenting Data Structures

Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk

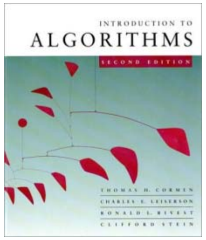


Dictionaries and Dynamic Sets

Abstract Data Type (ADT) Dictionary :

Insert (x, D):	inserts x into D	} D is a dynamic set
Delete (x, D):	deletes x from D	
Find (x, D):	finds x in D	

Popular implementation uses any balanced search tree (not necessarily binary). This way each operation takes $O(\log n)$ time.



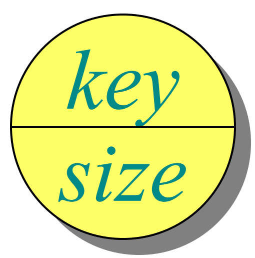
Dynamic order statistics

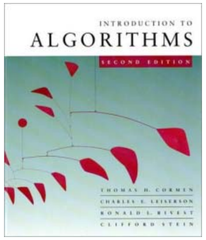
OS-SELECT(i, S): returns the ith smallest element in the dynamic set S.

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

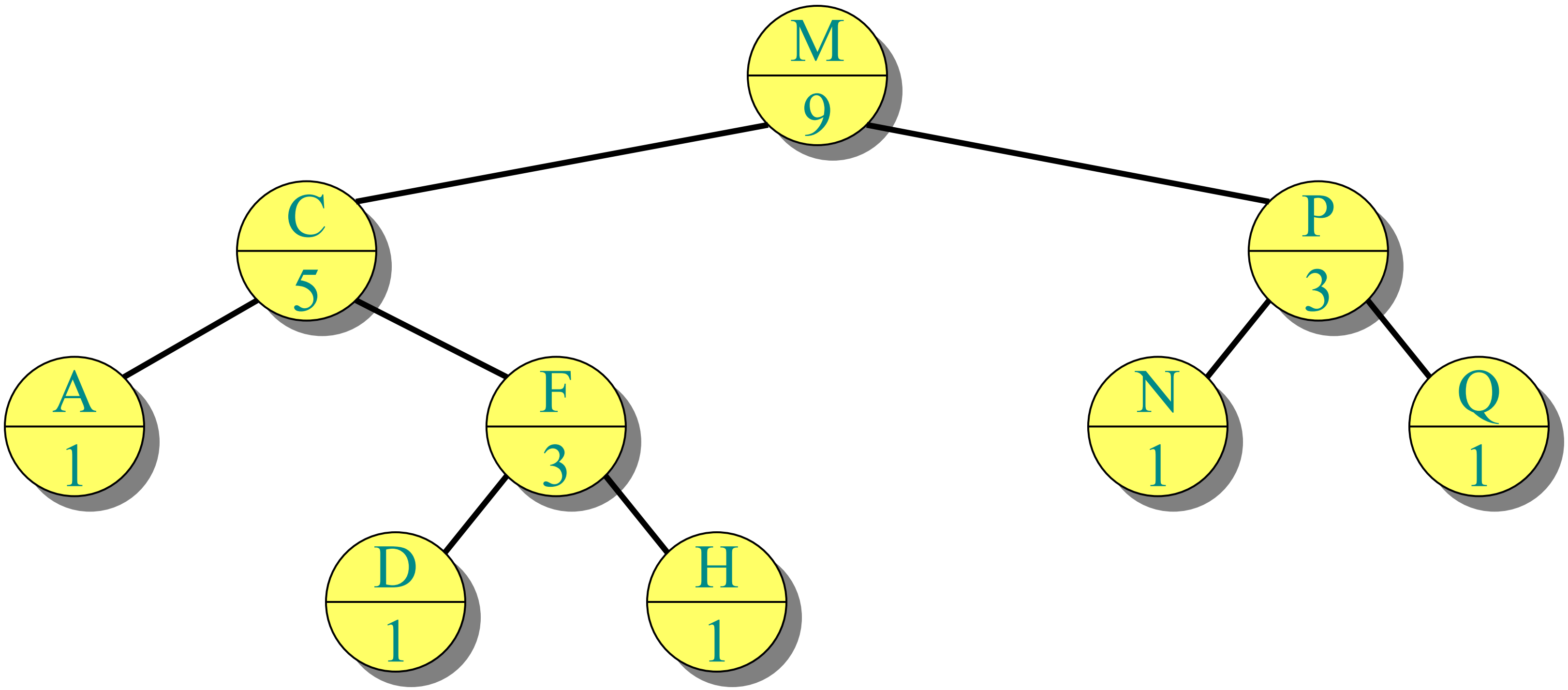
IDEA: Use a red-black tree for the set S, but keep subtree sizes in the nodes.

Notation for nodes:

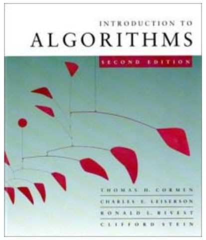




Example of an OS-tree



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



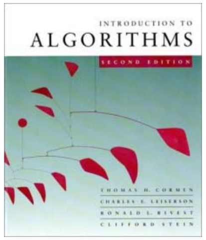
Selection

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

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

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

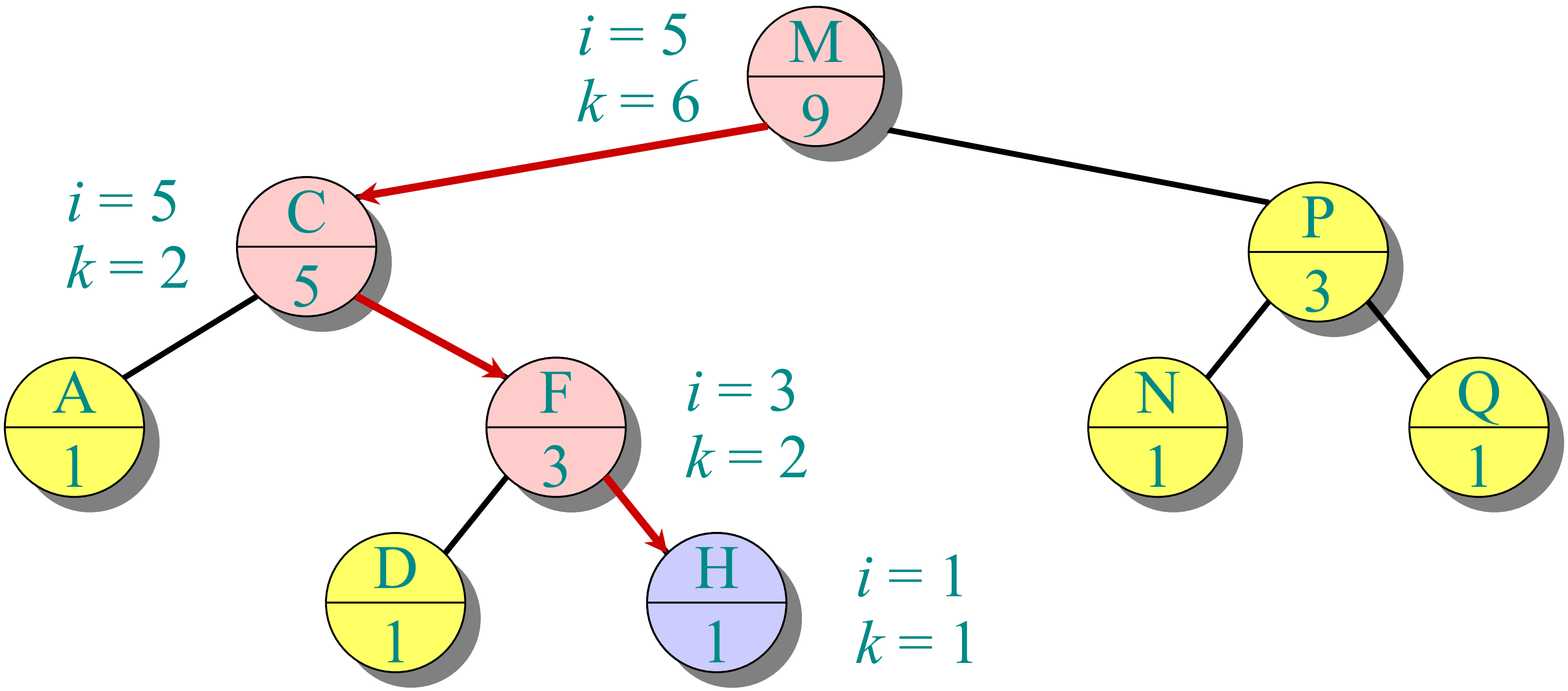
(OS-RANK is in the textbook.)



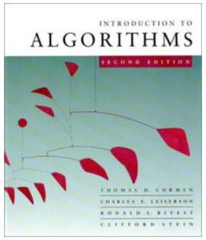
Example

OS-SELECT(x, i) ▹ ith smallest element in the subtree rooted at x
$k \leftarrow size[left[x]] + 1$ ▹ $k = \mathrm{rank}(x)$
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-SELECT$(root, 5)$



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



Data structure maintenance

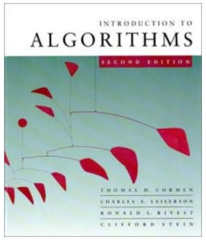
Q. Why not keep the ranks themselves in the nodes instead of subtree sizes?

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

$k \leftarrow size[left[x]] + 1 \qquad \triangleleft\ k = \mathrm{rank}(x)$

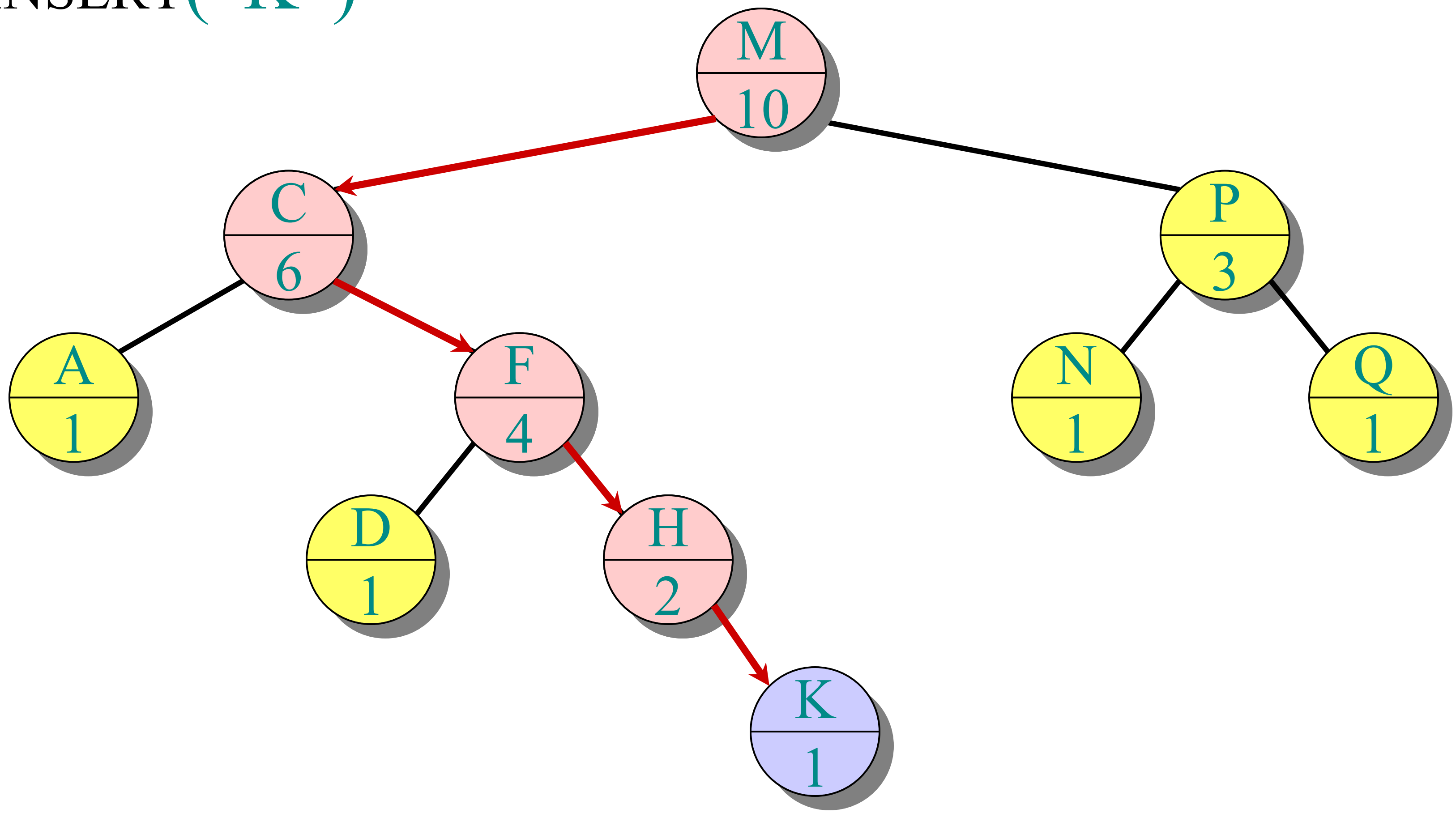
Modifying operations: INSERT and DELETE.

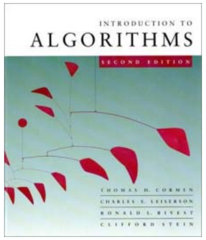
Strategy: Update subtree sizes when inserting or deleting.



Example of insertion

INSERT(“K”)



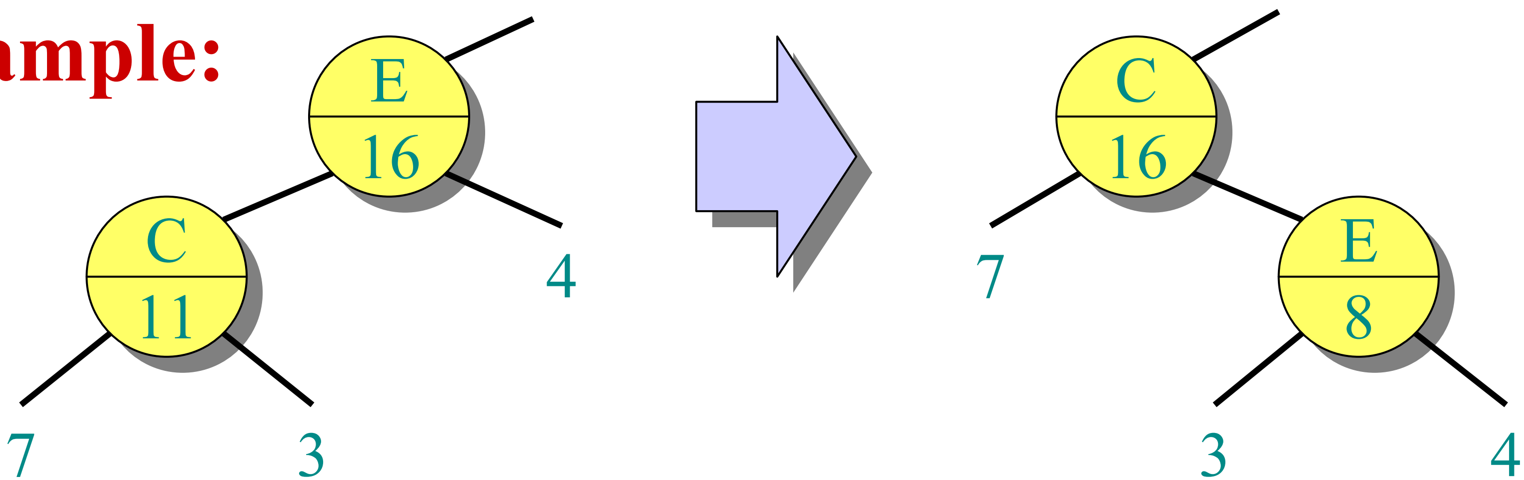


Handling rebalancing

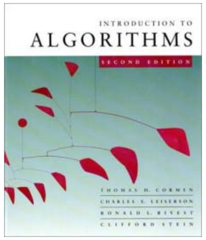
Don't forget that RB-INSERT and RB-DELETE may also need to modify the red-black tree in order to maintain balance.

- *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 $O(\log n)$ time.

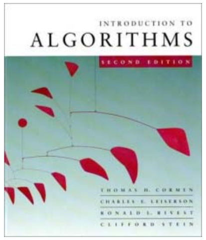


Data-structure augmentation

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

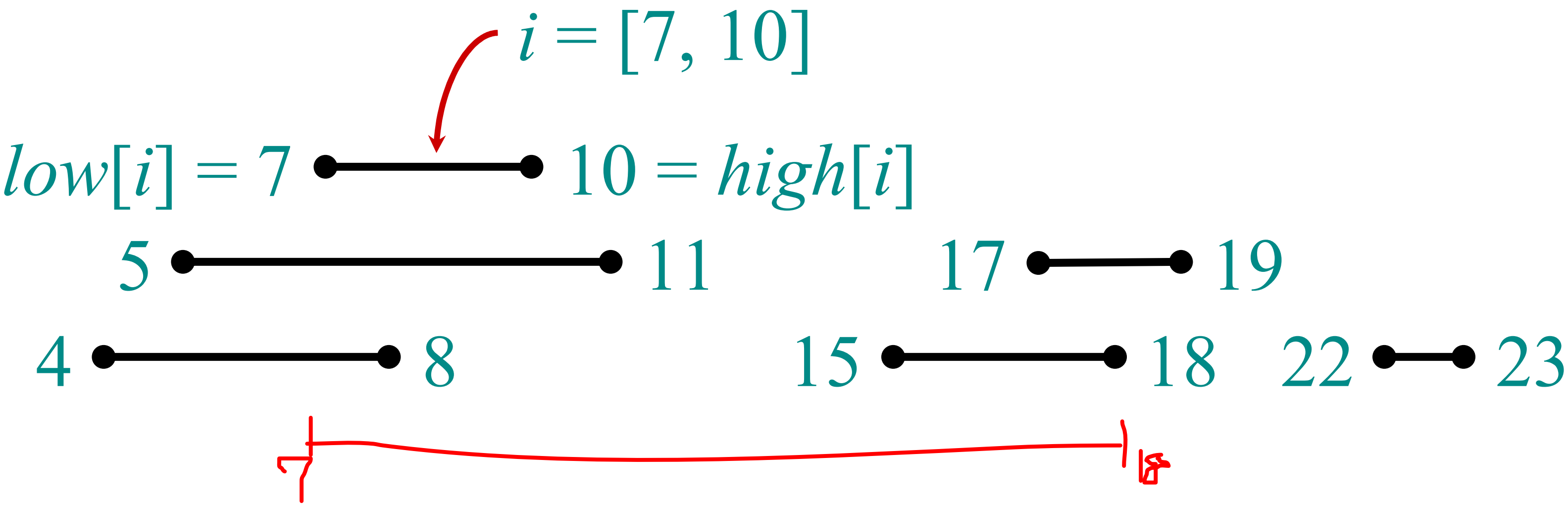
1. Choose an underlying data structure (*red-black tree*).
2. Determine additional information to be stored in the data structure (*subtree sizes*).
3. Verify that this information can be maintained for modifying operations (RB-INSERT, RB-DELETE — *don't forget rotations*).
4. Develop new dynamic-set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

These steps are guidelines, not rigid rules.



Interval trees

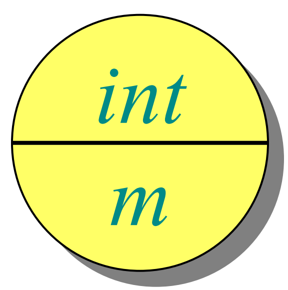
Goal: To maintain a dynamic set of intervals, such as time intervals.

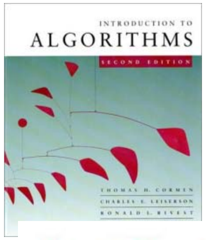


Query: For a given query interval i, find an interval in the set that overlaps i.

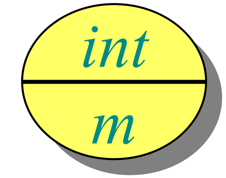
Following the methodology

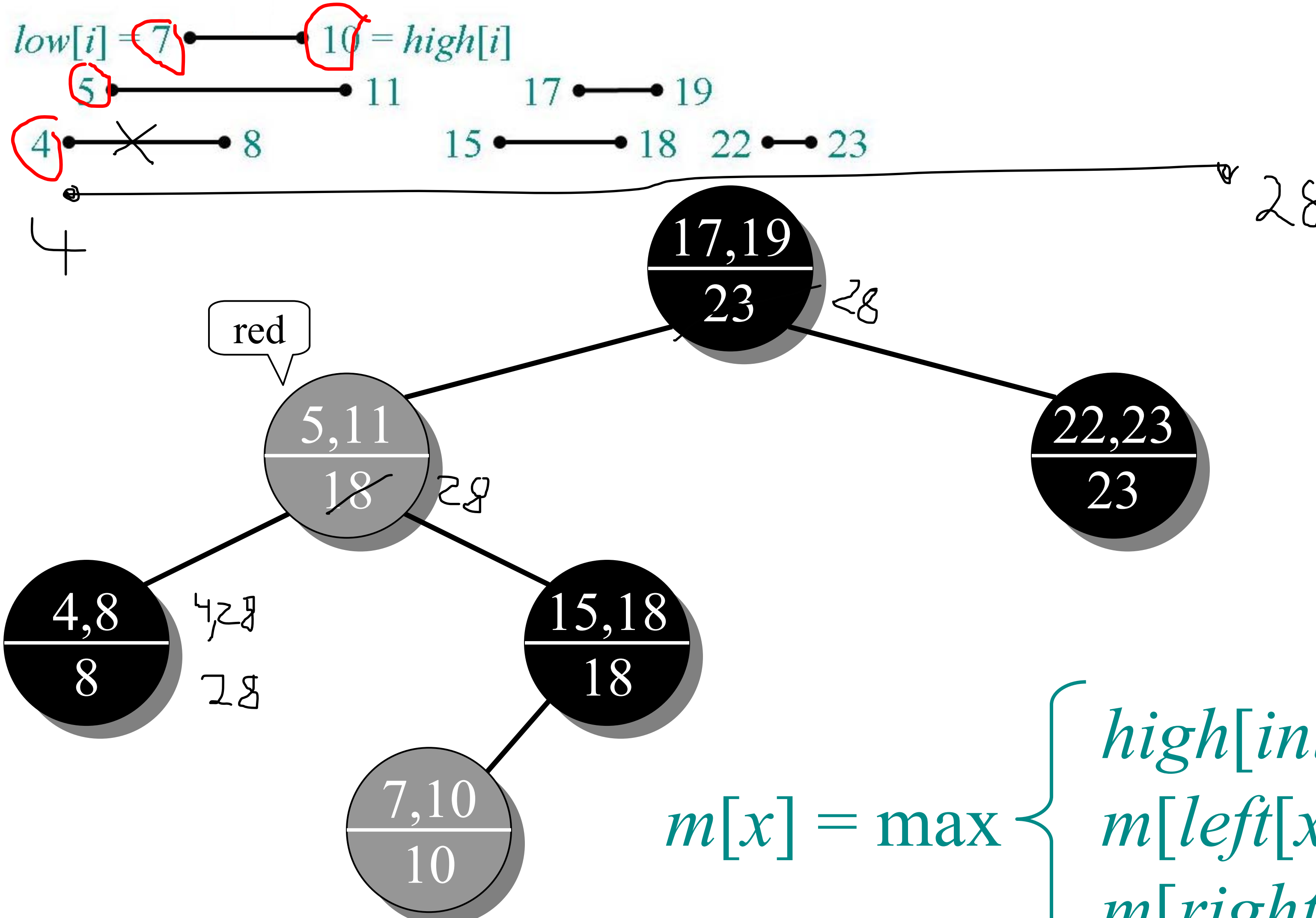
1. *Choose an underlying data structure.*
   - Red-black tree keyed on low (left) endpoint.
2. *Determine additional information to be stored in the data structure.*
   - Store in each node x the interval $int[x]$ corresponding to the key, as well as the largest value $m[x]$ of all right interval endpoints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





Example interval tree

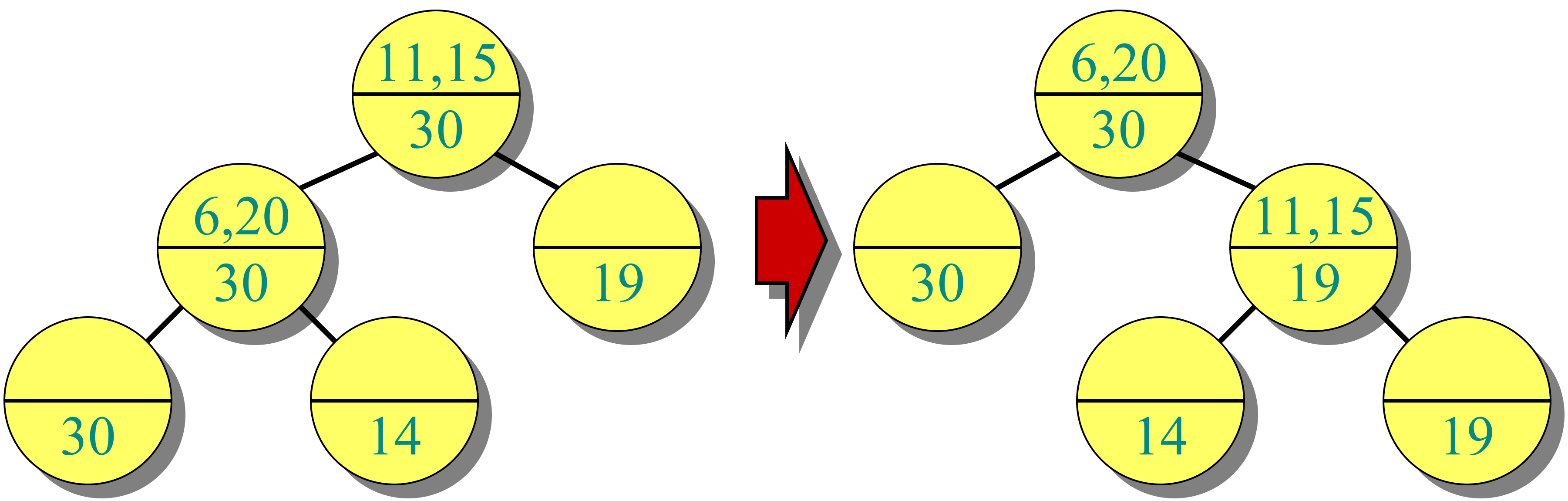




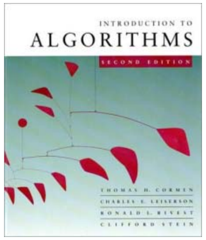
$$m[x] = \max \begin{cases} high[int[x]] \\ m[left[x]] \\ m[right[x]] \end{cases}$$

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:



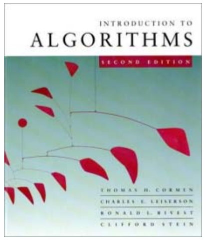
Total INSERT time $= O(\log n)$; DELETE similar.



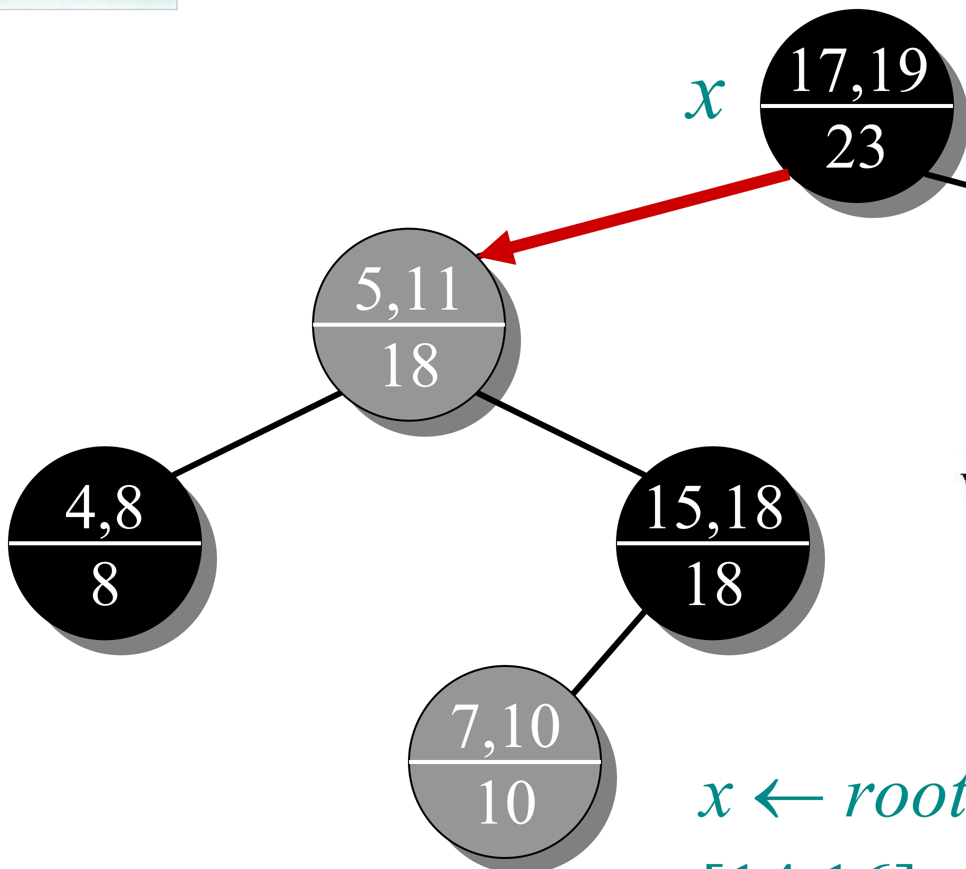
New operations

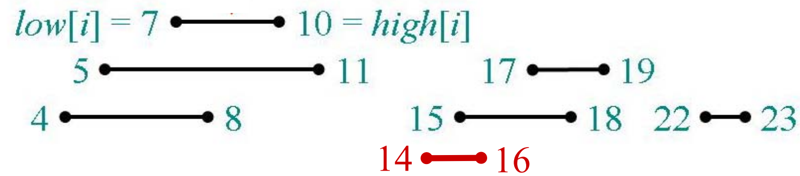
4. Develop new dynamic-set operations that use the information.

```
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])



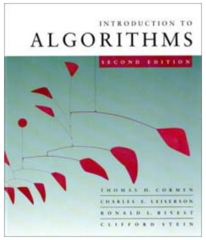


while $x \neq$ NIL and ($low[i] > high[int[x]]$
or $low[int[x]] > high[i]$)
do ▷ i and $int[x]$ don't overlap
if $left[x] \neq$ NIL and $low[i] \leq m[left[x]]$
then $x \leftarrow left[x]$
else $x \leftarrow right[x]$

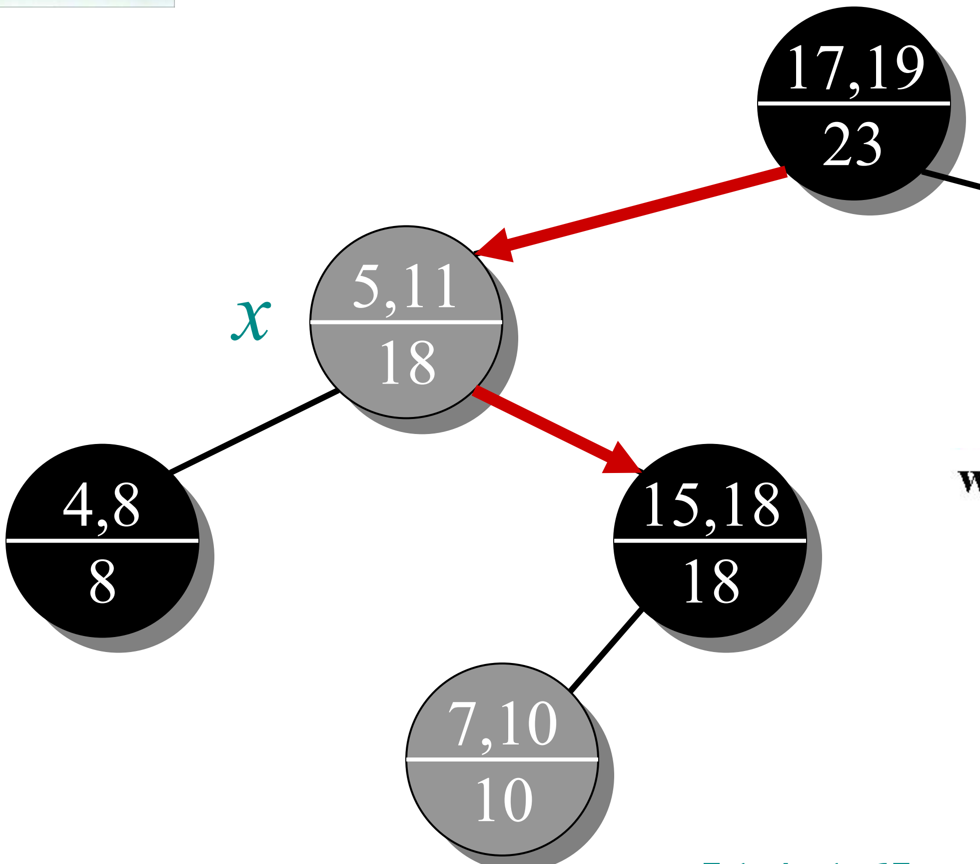
$x \leftarrow root$

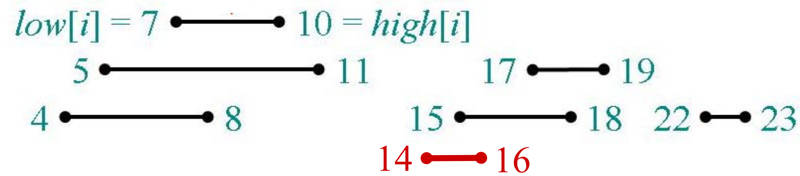
[14,16] and [17,19] don't overlap

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



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

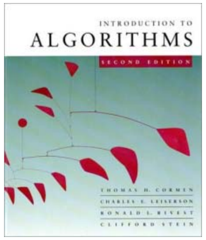




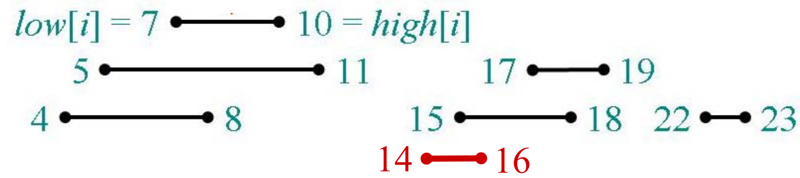
while $x \neq$ NIL and ($low[i] > high[int[x]]$
or $low[int[x]] > high[i]$)
do ▷ i and $int[x]$ don't overlap
if $left[x] \neq$ NIL and $low[i] \leq m[left[x]]$
then $x \leftarrow left[x]$
else $x \leftarrow right[x]$

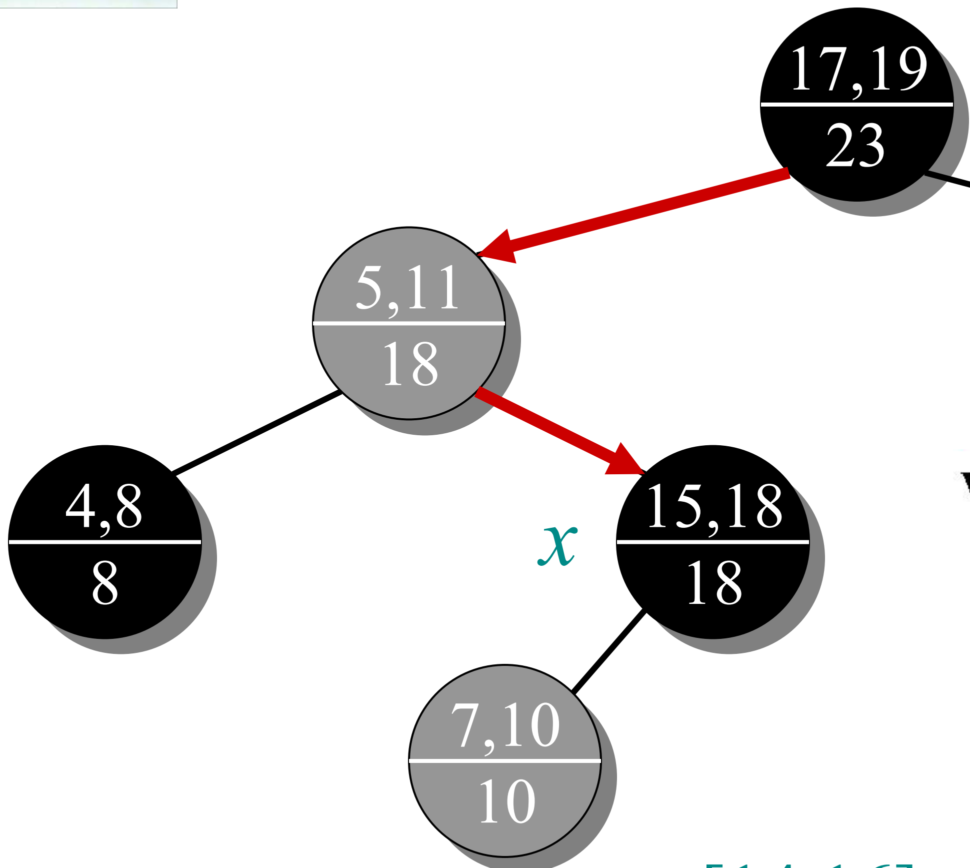
[14,16] and [5,11] don't overlap

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



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

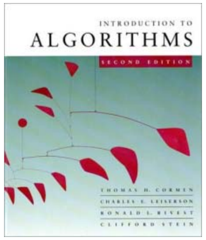




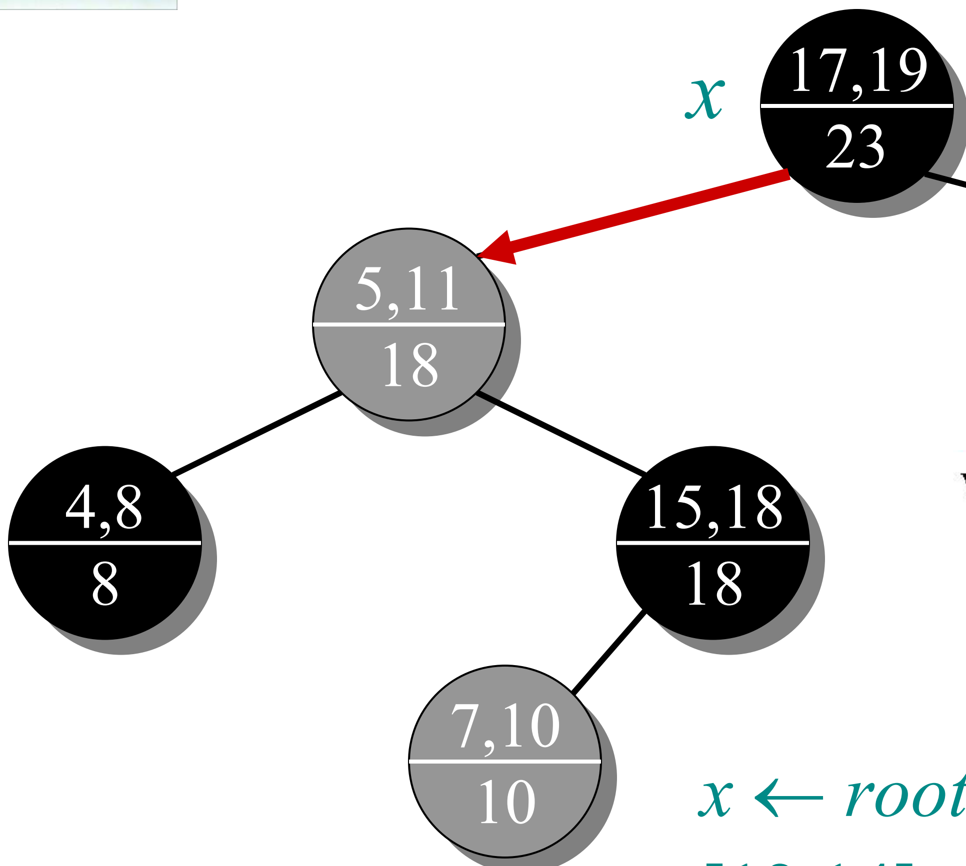
while $x \neq$ NIL and $(low[i] > high[int[x]]$
or $low[int[x]] > high[i])$
do ▷ i and $int[x]$ don't overlap
if $left[x] \neq$ NIL and $low[i] \leq m[left[x]]$
then $x \leftarrow left[x]$
else $x \leftarrow right[x]$

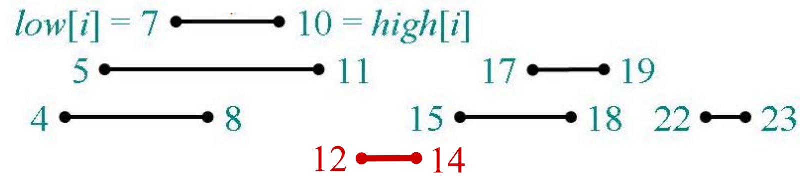
[14,16] and [15,18] overlap

return [15,18]



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



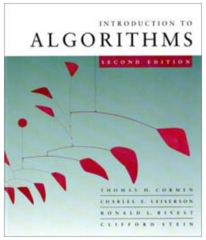


while $x \neq$ NIL and ($low[i] > high[int[x]]$
or $low[int[x]] > high[i]$)
do ▷ i and $int[x]$ don't overlap
if $left[x] \neq$ NIL and $low[i] \leq m[left[x]]$
then $x \leftarrow left[x]$
else $x \leftarrow right[x]$

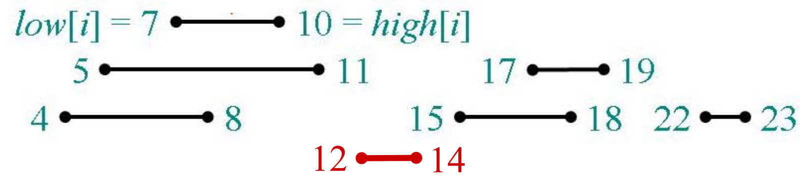
$x \leftarrow root$

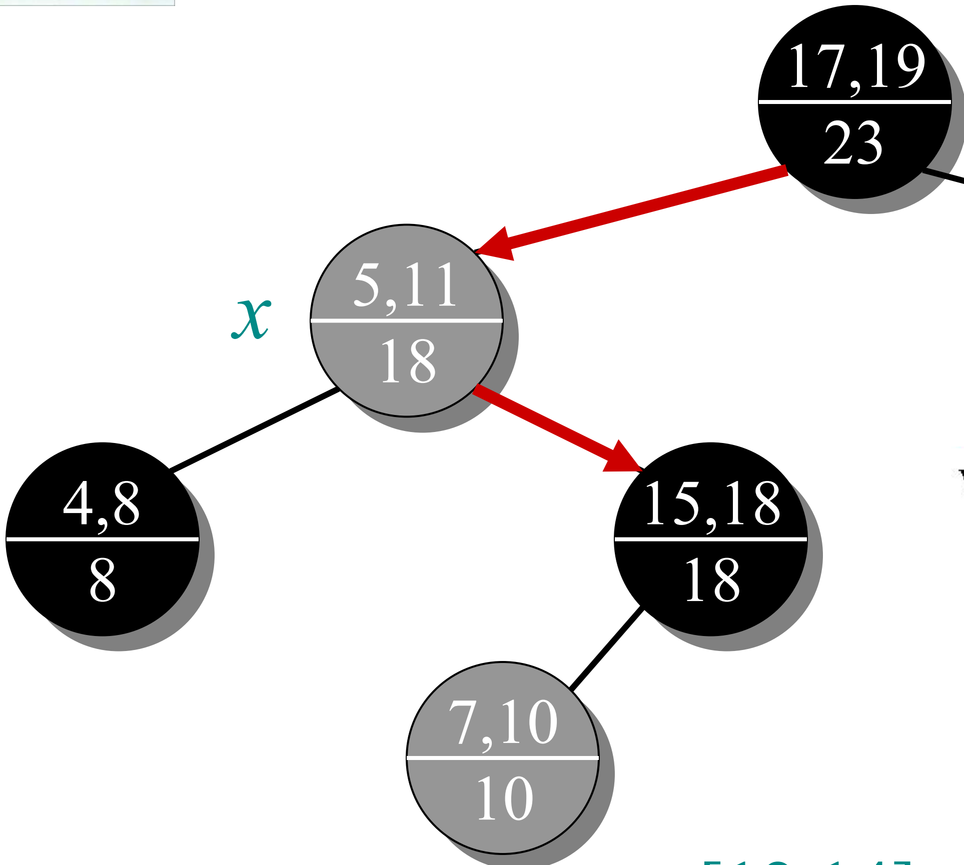
[12,14] and [17,19] don't overlap

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



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

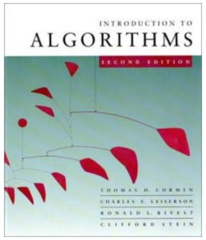




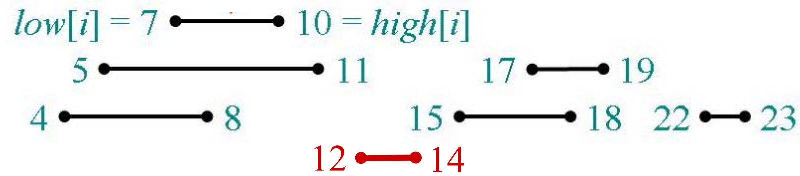
while $x \neq$ NIL and $(low[i] > high[int[x]]$
or $low[int[x]] > high[i])$
do ▷ i and $int[x]$ don't overlap
if $left[x] \neq$ NIL and $low[i] \leq m[left[x]]$
then $x \leftarrow left[x]$
else $x \leftarrow right[x]$

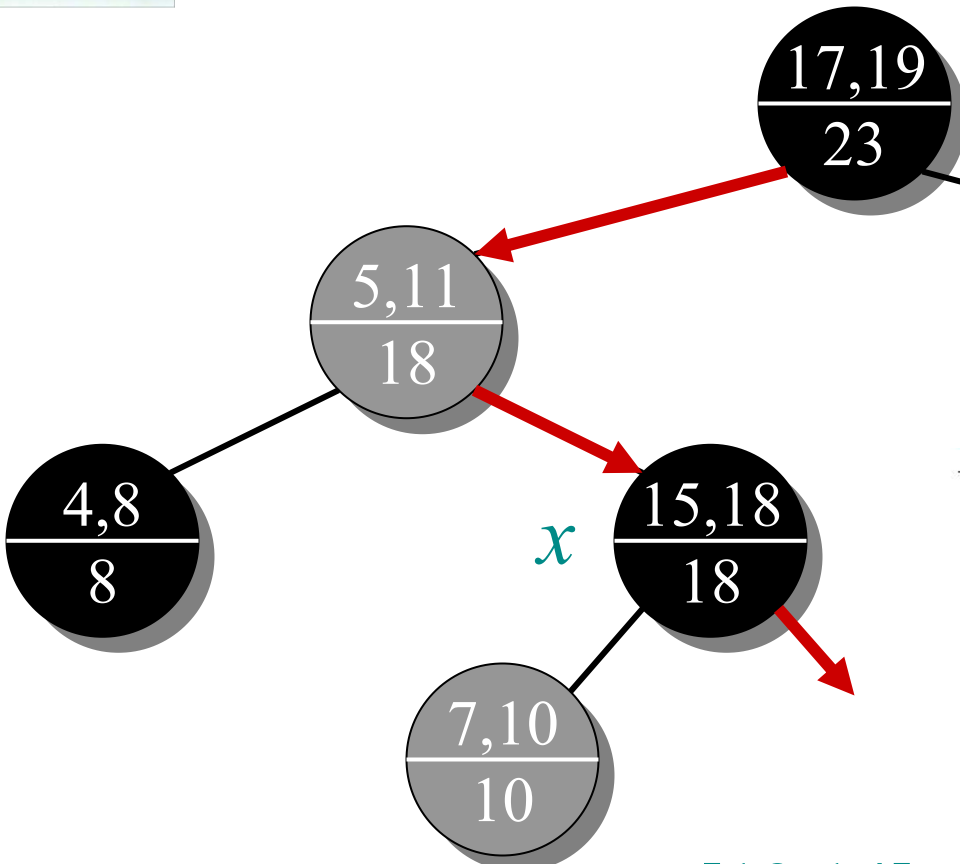
[12,14] and [5,11] don't overlap

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



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

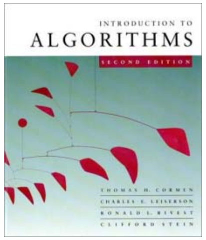




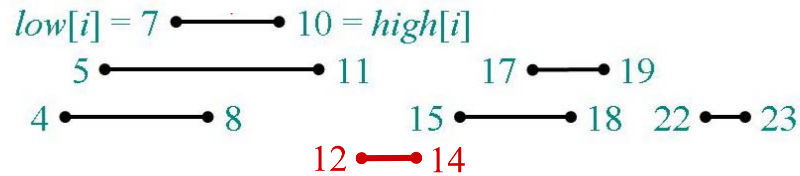
while $x \neq$ NIL and $(low[i] > high[int[x]]$
or $low[int[x]] > high[i])$
do ▹ i and $int[x]$ don't overlap
if $left[x] \neq$ NIL and $low[i] \leq m[left[x]]$
then $x \leftarrow left[x]$
else $x \leftarrow right[x]$

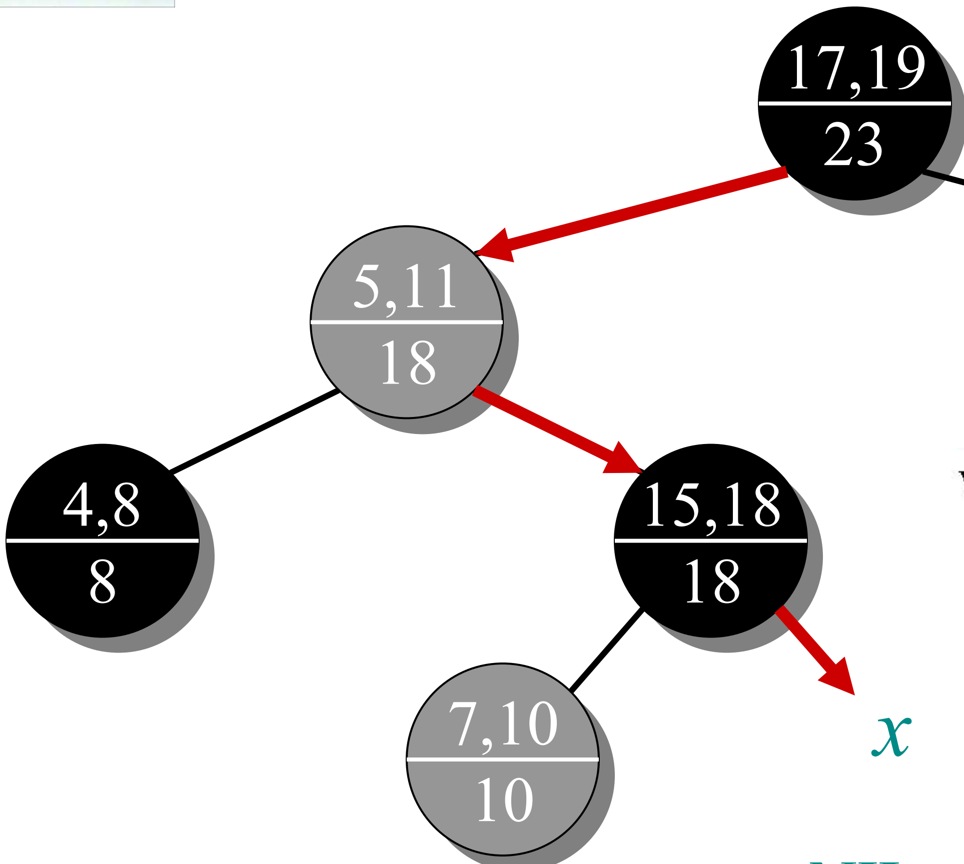
[12,14] and [15,18] don't overlap

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



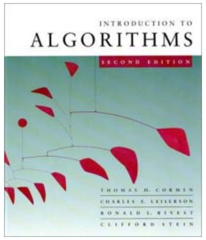
Example 2: INTERVAL-SEARCH([12,14])





while $x \neq$ NIL and $(low[i] > high[int[x]]$
  or $low[int[x]] > high[i])$
 do ▷ i and $int[x]$ don't overlap
  if $left[x] \neq$ NIL and $low[i] \leq m[left[x]]$
   then $x \leftarrow left[x]$
   else $x \leftarrow right[x]$

$x =$ NIL $\Rightarrow$ no interval that overlaps [12,14] exists



Analysis

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

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

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

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

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

- If the search goes left, then

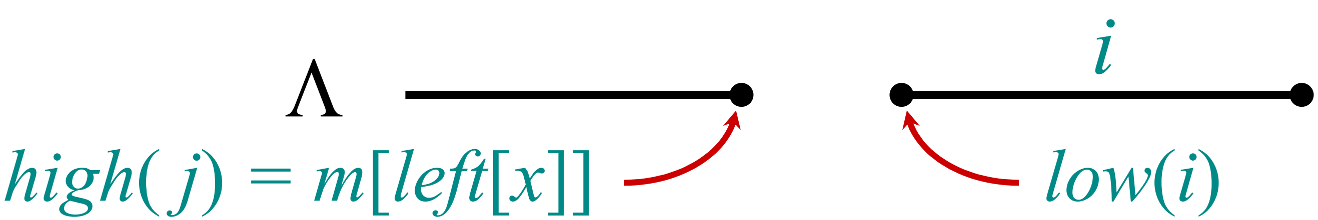
$$\{i' \in L : i' \text{ overlaps } i \} = \varnothing$$
$$\Rightarrow \{i' \in R : i' \text{ overlaps } i \} = \varnothing.$$

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.

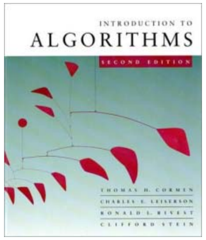
Correctness proof

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 right endpoint of some interval $j \in L$, and no other interval in L can have a larger right endpoint than $high(j)$.



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

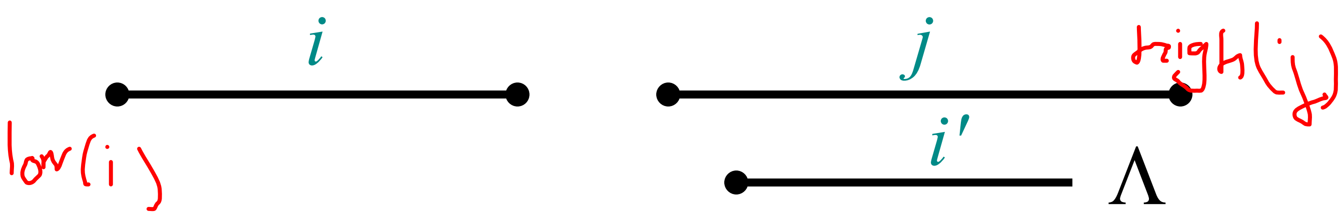


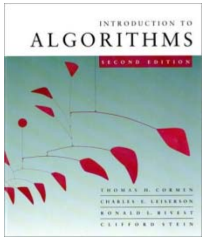
Proof (continued)

Suppose that the search goes left, and assume that

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

- 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 $\{i' \in R : i' \text{ overlaps } i\} = \emptyset$. ◻





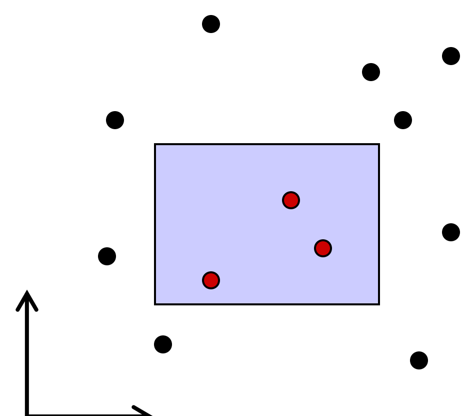
Orthogonal range searching

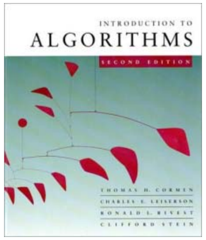
Input: n points in d dimensions

- E.g., representing a database of n records each with d numeric fields

Query: Axis-aligned ***box*** (in 2D, a rectangle)

- Report on the points inside the box:
  - Are there any points?
  - How many are there?
  - List the points.





Orthogonal range searching

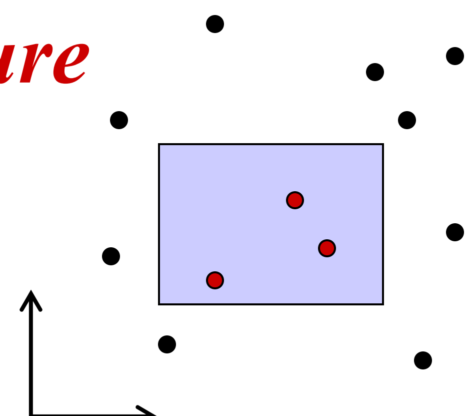
Input: *n* points in *d* dimensions

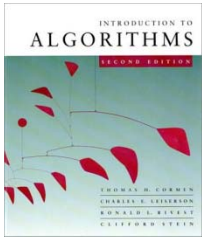
Query: Axis-aligned ***box*** (in 2D, a rectangle)

- Report on the points inside the box

Goal: Preprocess points into a data structure to support fast queries

- Primary goal: ***Static data structure***
- In 1D, we will also obtain a dynamic data structure supporting insert and delete





1D range searching

In 1D, the query is an interval:

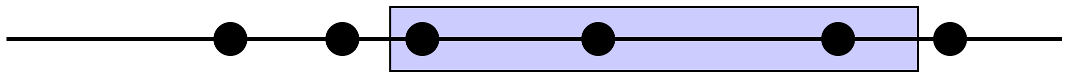
First solution:

- Sort the points and store them in an array
  - Solve query by binary search on endpoints.
  - Obtain a static structure that can list k answers in a query in $O(k + \log n)$ time.

Goal: Obtain a dynamic structure that can list k answers in a query in $O(k + \log n)$ time.

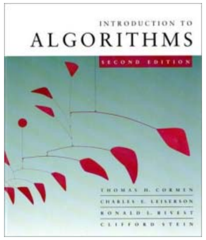
1D range searching

In 1D, the query is an interval:

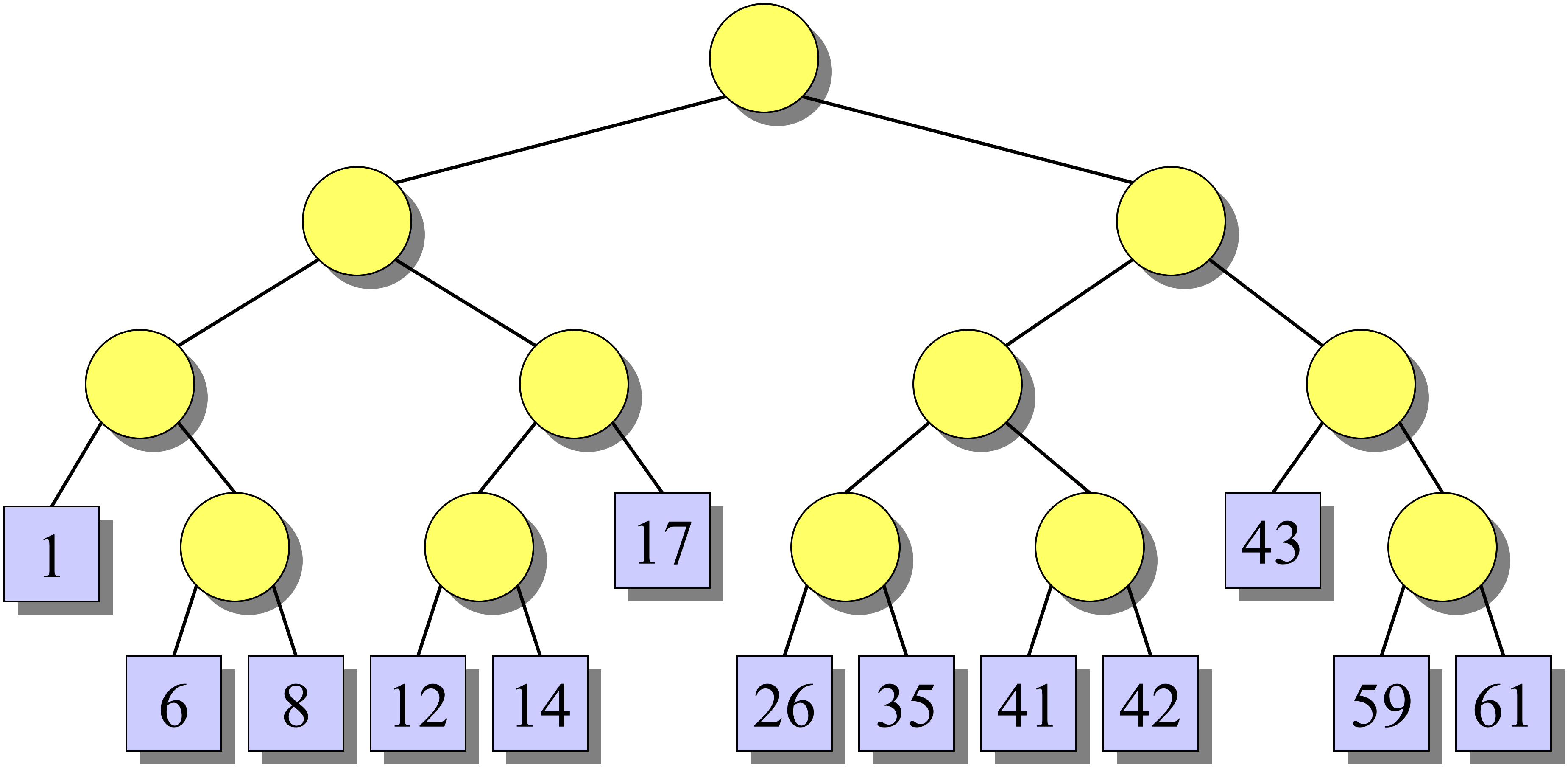


New solution that extends to higher dimensions:

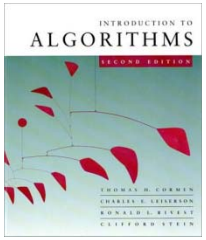
- Balanced binary search tree
  - New organization principle: Store points in the *leaves* of the tree.
  - Internal nodes store copies of the leaves to satisfy binary search property:
    - Node x stores in $key[x]$ the maximum key of any leaf in the left subtree of x.



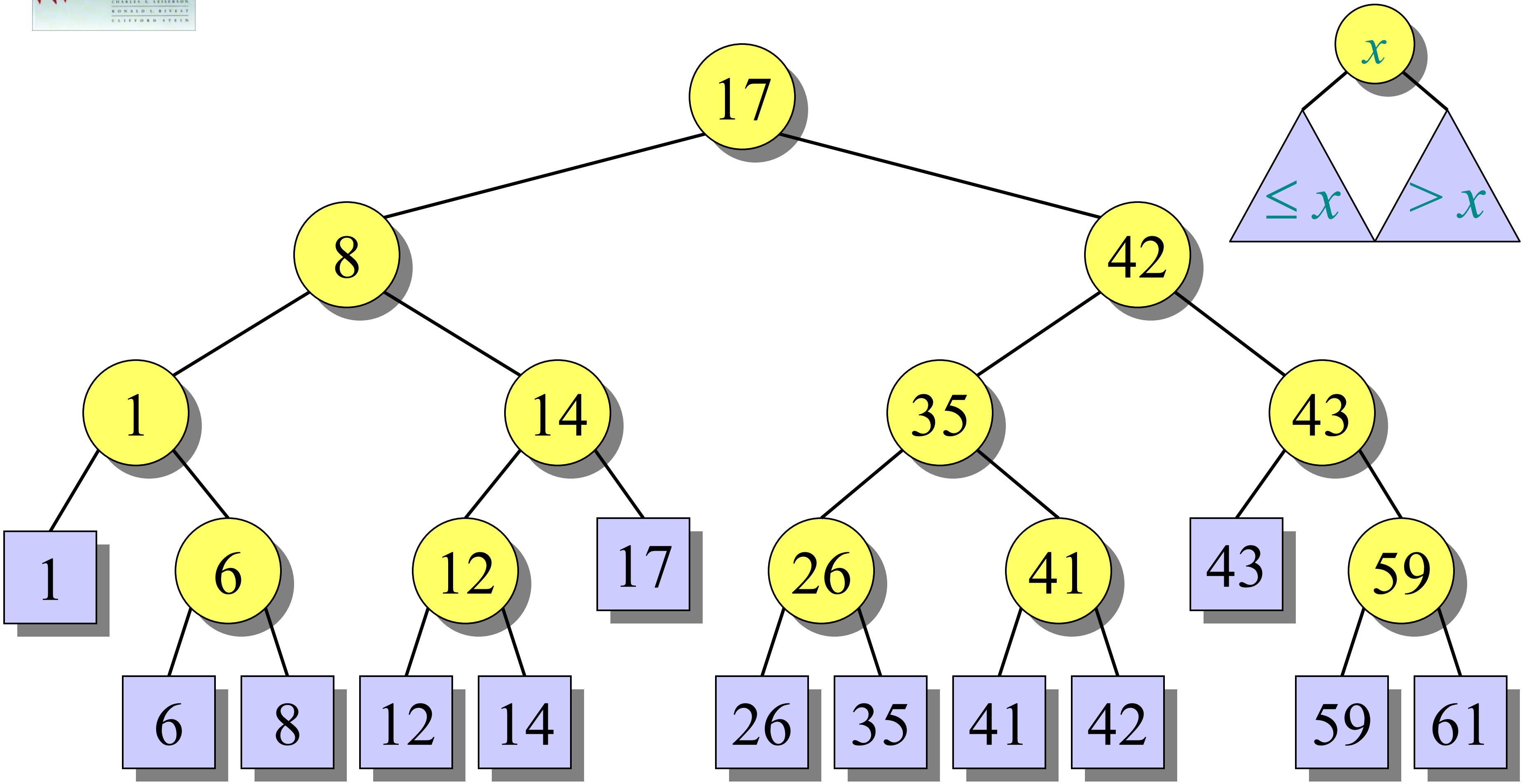
Example of a 1D range tree



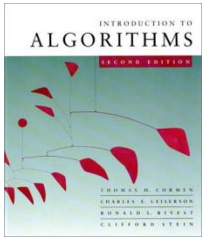
key[*x*] is the maximum key of any leaf in the left subtree of *x*.



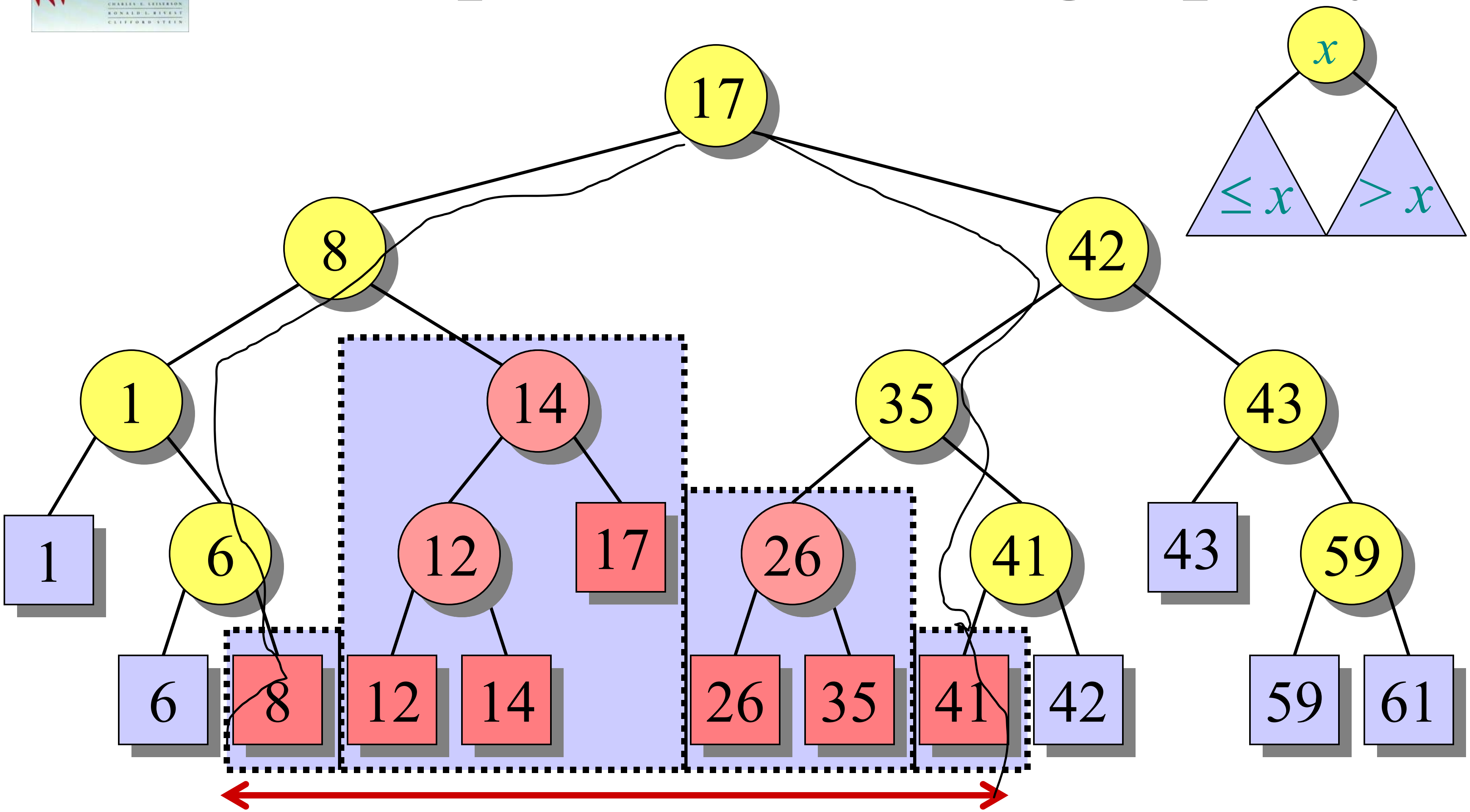
Example of a 1D range tree



key[*x*] is the maximum key of any leaf in the left subtree of *x*.

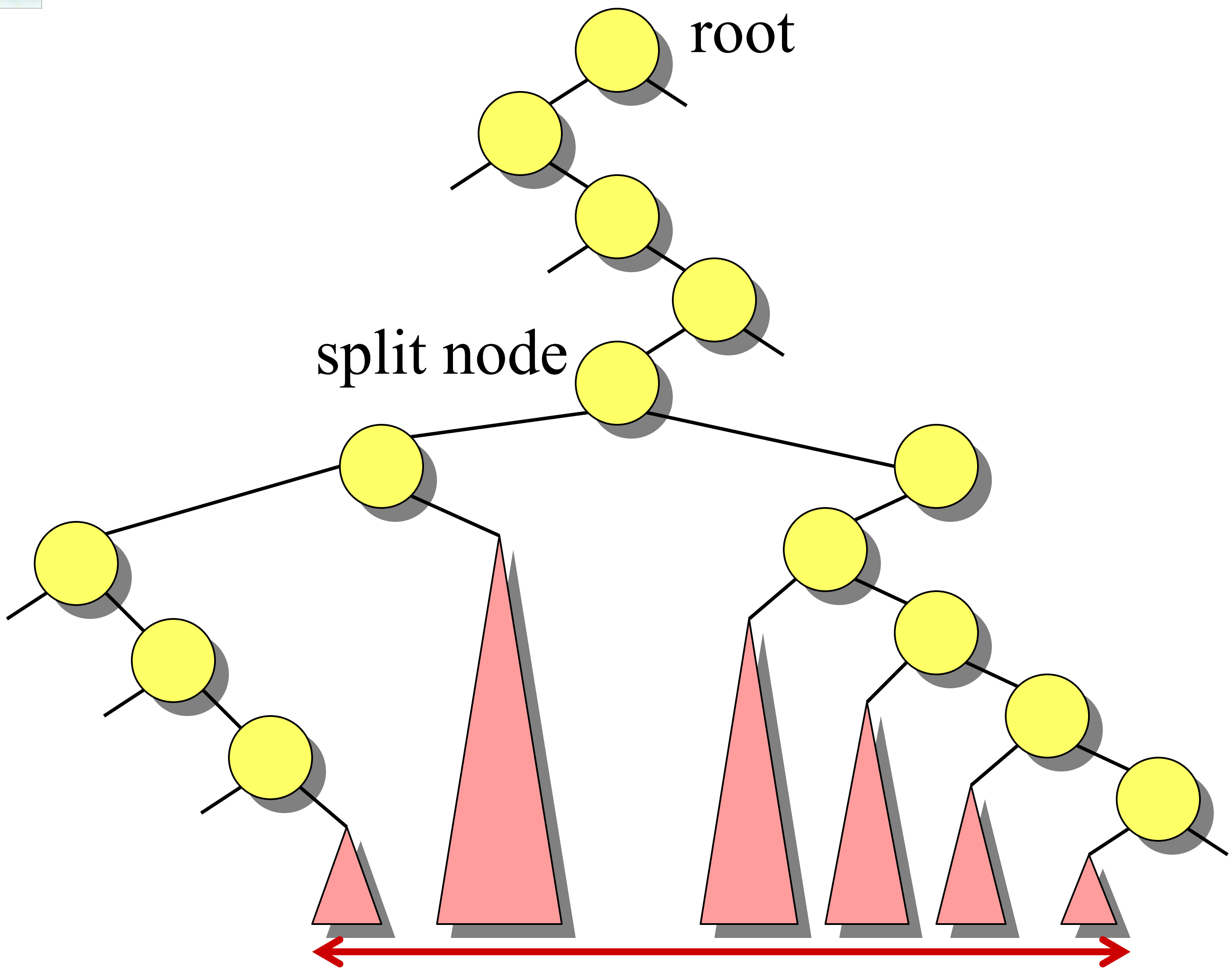


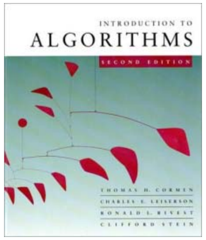
Example of a 1D range query



RANGE-QUERY([7, 41])

General 1D range query





Pseudocode, part 1: Find the split node

1D-RANGE-QUERY$(T, [x_1, x_2])$

$w \leftarrow \mathrm{root}[T]$

while w is not a leaf and $(x_2 \leq key[w]$ or $key[w] < x_1)$

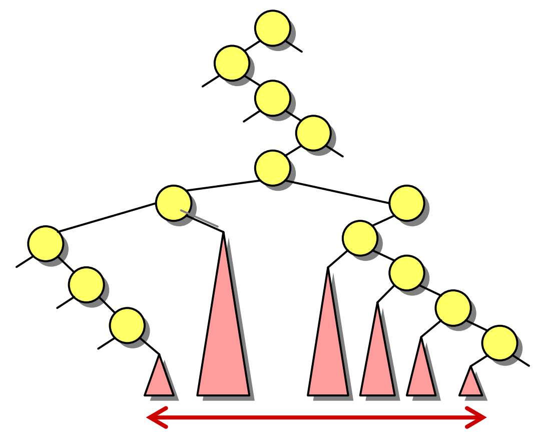
do if $x_2 \leq key[w]$

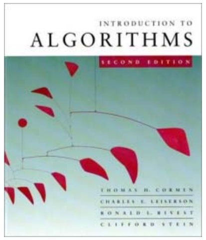
then $w \leftarrow left[w]$

else $w \leftarrow right[w]$

// w is now the split node

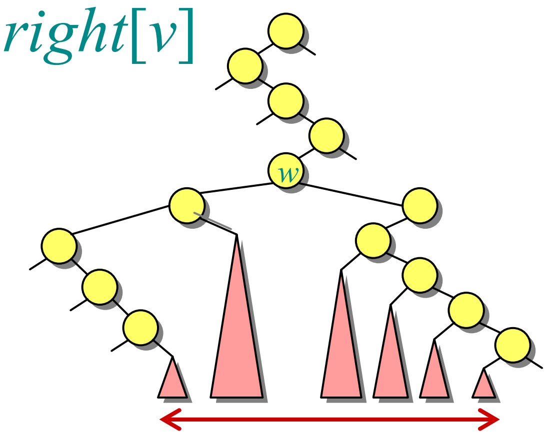
[*traverse left and right from w and report relevant subtrees*]





Pseudocode, part 2: Traverse left and right from split node

1D-RANGE-QUERY(T, $[x_1, x_2]$)
    [*find the split node*]
    // w is now the split node
    if w is a leaf
     then output the leaf w if $x_1 \leq key[w] \leq x_2$
     else $v \leftarrow left[w]$ // Left traversal
        while v is not a leaf
          do if $x_1 \leq key[v]$
              then output the subtree rooted at $right[v]$
                  $v \leftarrow left[v]$
              else $v \leftarrow right[v]$
        output the leaf v if $x_1 \leq key[v] \leq x_2$
        [*symmetrically for right traversal*]

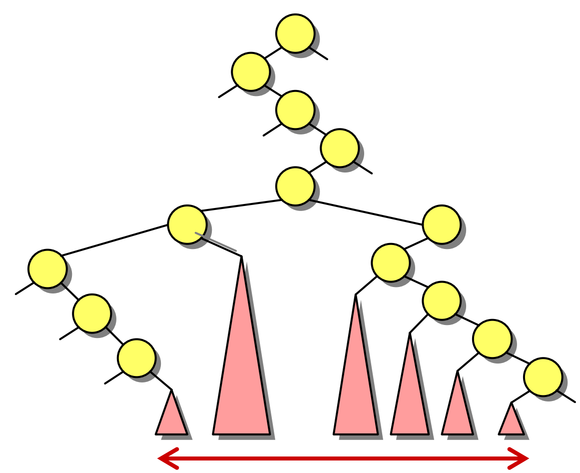


Analysis of 1D-RANGE-QUERY

Query time: Answer to range query represented by O(log n) subtrees found in O(log n) time.

Thus:

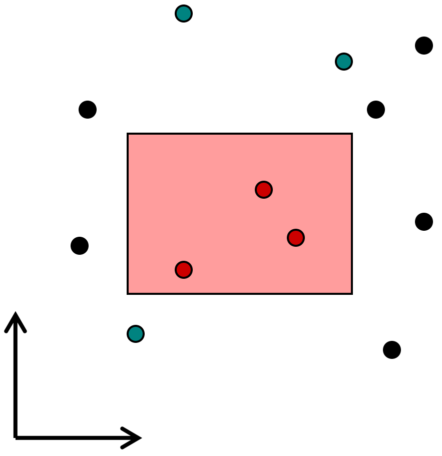
- Can test for points in interval in O(log n) time.
- Can report all k points in interval in O(k + log n) time.
- Can count points in interval in O(log n) time

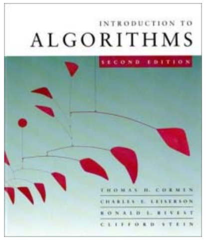


Space: O(n)

Preprocessing time: O(n log n)

2D range trees



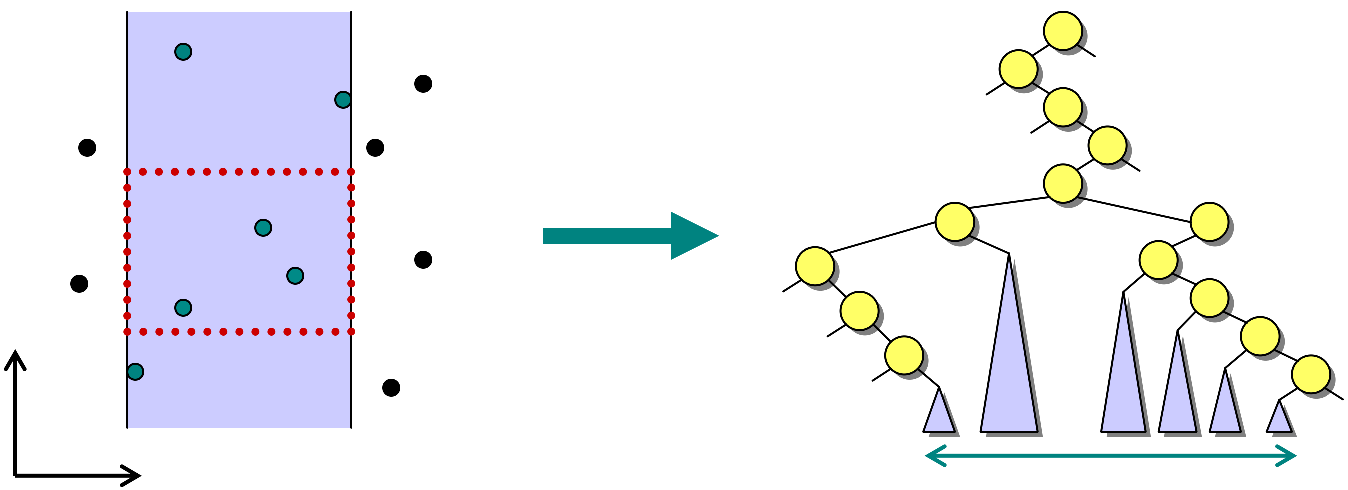


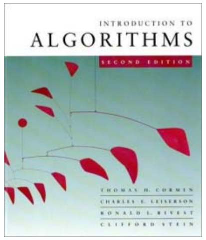
2D range trees

Store a ***primary*** 1D range tree for all the points based on *x*-coordinate.

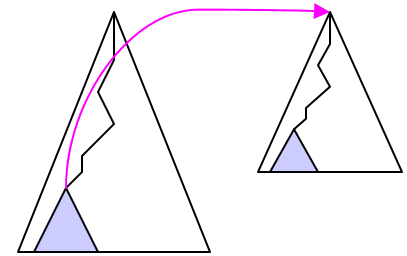
Thus in O(log *n*) time we can find O(log *n*) subtrees representing the points with proper *x*-coordinate.

How to restrict to points with proper *y*-coordinate?

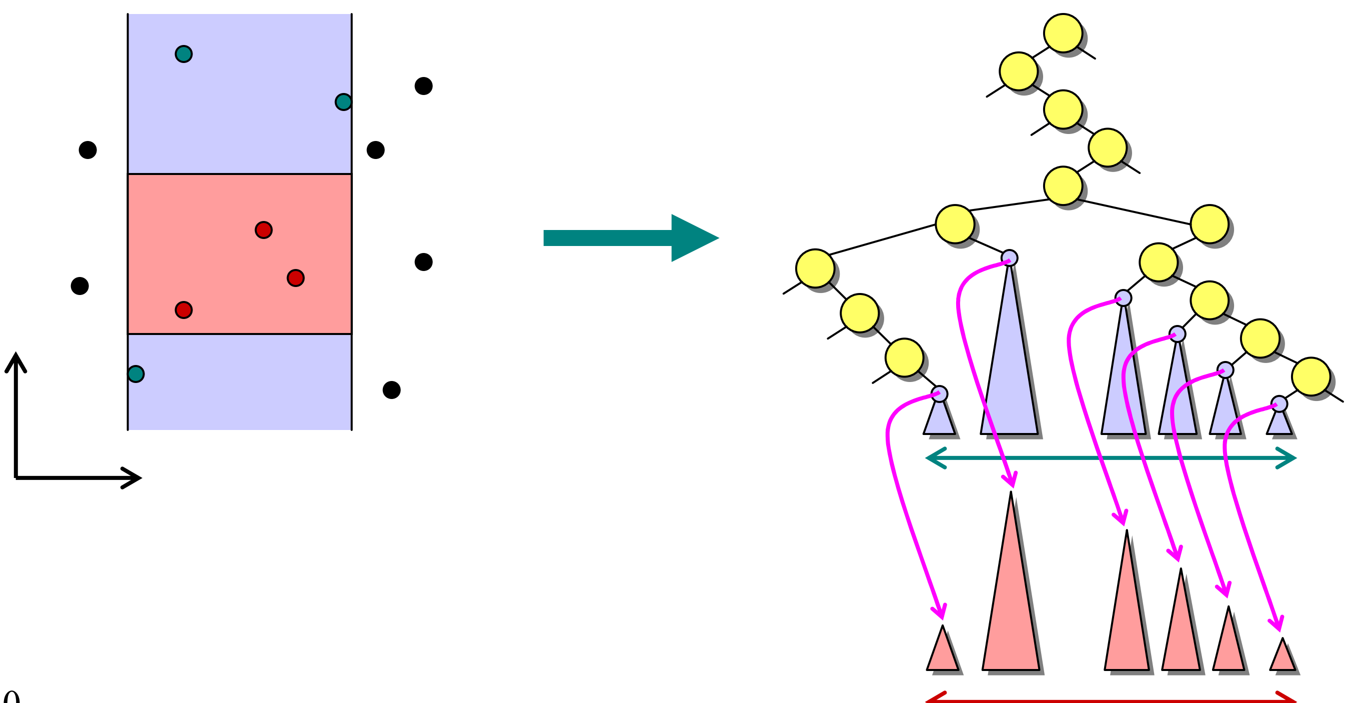


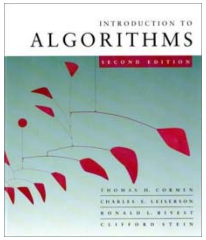


2D range trees



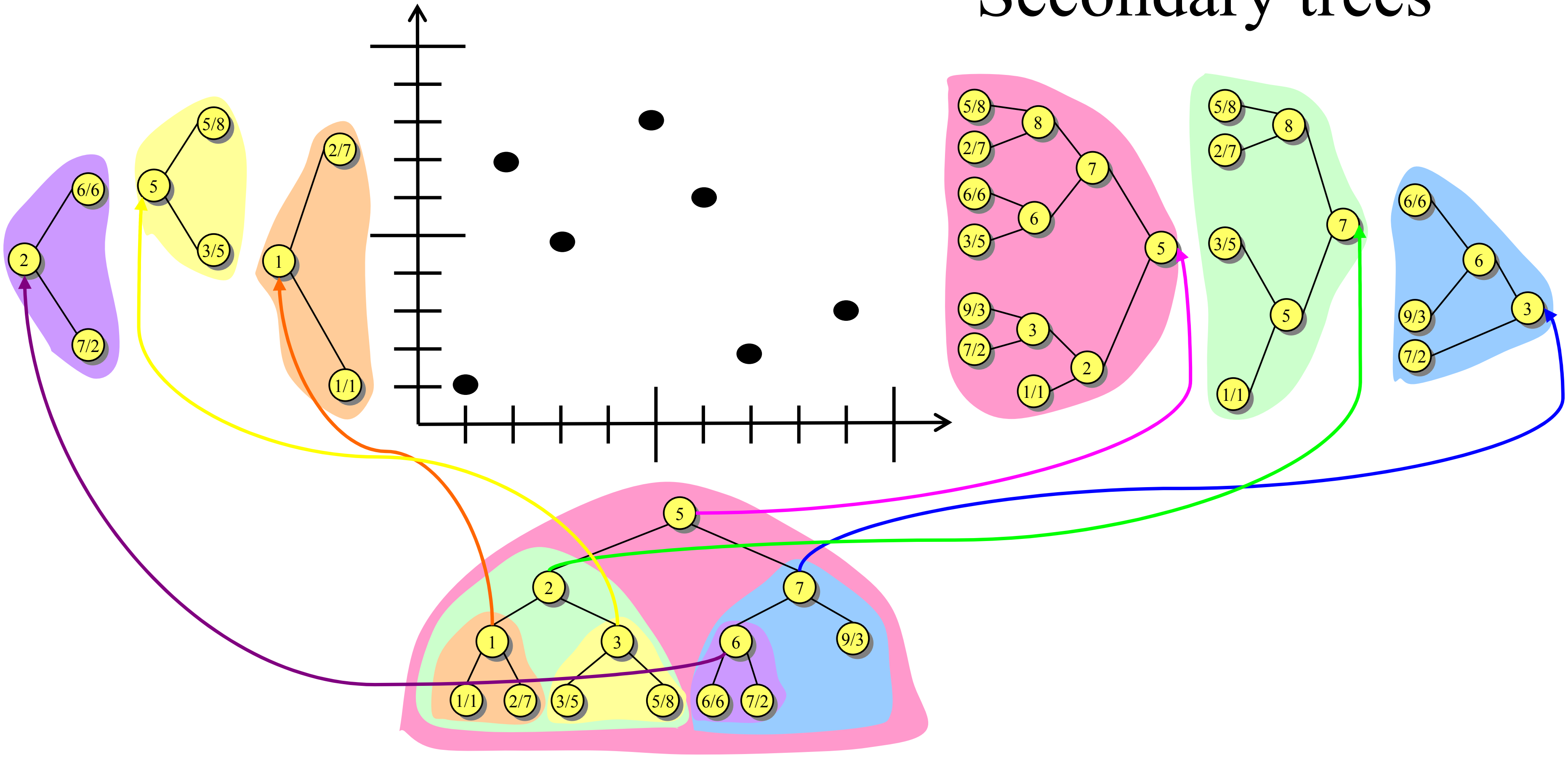
Idea: In primary 1D range tree of *x*-coordinate, **every** node stores a ***secondary*** 1D range tree based on *y*-coordinate for all points in the subtree of the node. Recursively search within each.



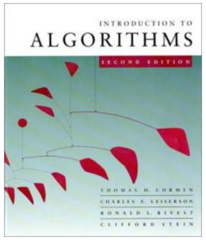


2D range tree example

Secondary trees



Primary tree

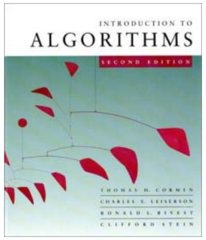


Analysis of 2D range trees

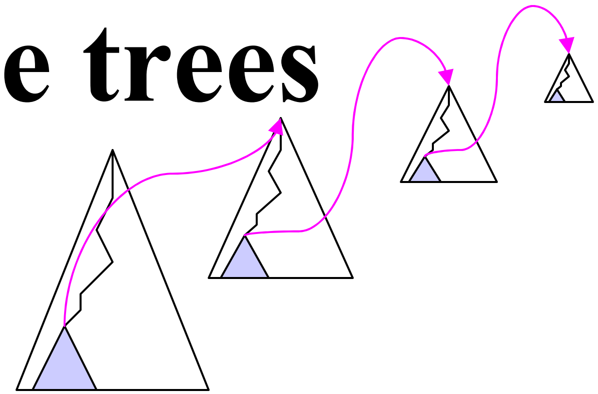
Query time: In $O(\log^2 n) = O((\log n)^2)$ time, we can represent answer to range query by $O(\log^2 n)$ subtrees. Total cost for reporting k points: $O(k + (\log n)^2)$.

Space: The secondary trees at each level of the primary tree together store a copy of the points. Also, each point is present in each secondary tree along the path from the leaf to the root. Either way, we obtain that the space is $O(n \log n)$.

Preprocessing time: $O(n \log n)$



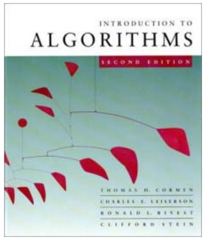
d-dimensional range trees



Each node of the secondary *y*-structure stores a tertiary *z*-structure representing the points in the subtree rooted at the node, etc.

Save one log factor using fractional cascading

Query time: $O(k + \log^d n)$ to report *k* points.
Space: $O(n \log^{d-1} n)$
Preprocessing time: $O(n \log^{d-1} n)$



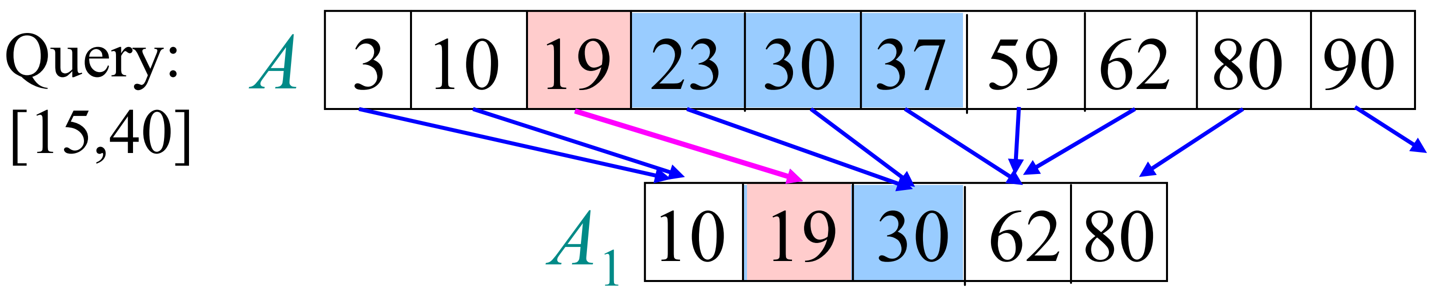
Search in Subsets

Given: Two sorted arrays A_1 and A, with $A_1 \subseteq A$
A query interval $[l,r]$

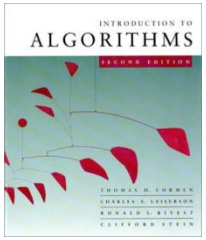
Task: Report all elements e in A_1 and A with $l \leq e \leq r$

Idea: Add pointers from A to A_1:
$\rightarrow$ For each $a \in A$ add a pointer to the smallest element $b \in A_1$ with $b \geq a$

Query: Find $l \in A$, follow pointer to A_1. Both in A and A_1 sequentially output all elements in $[l,r]$.



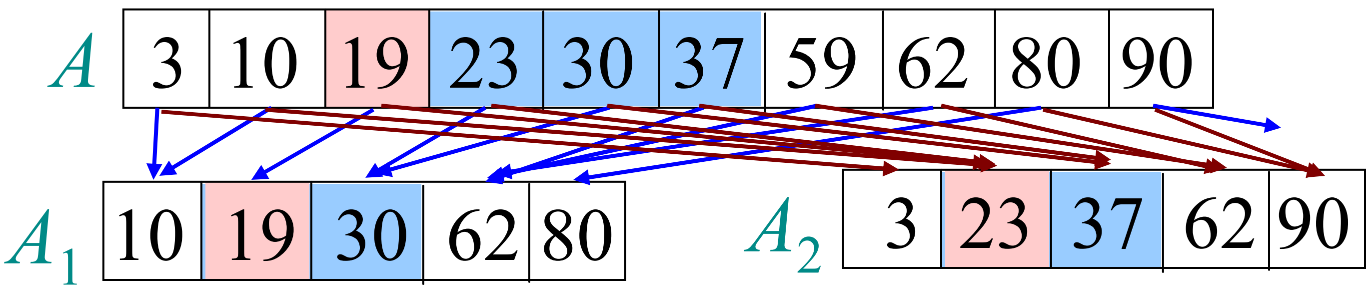
Runtime: $O((\log n + k) + (1 + k)) = O(\log n + k)$



Search in Subsets (cont.)

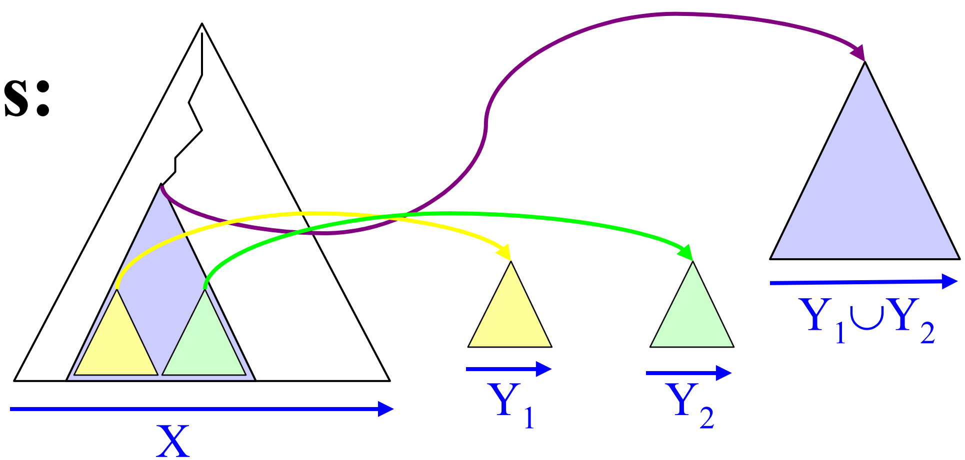
Given: Three sorted arrays A_1, A_2, and A, with $A_1 \subseteq A$ and $A_2 \subseteq A$

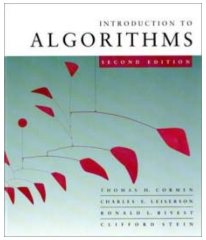
Query: [15,40]



Runtime: $O((\log n + k) + (1+k) + (1+k)) = O(\log n + k)$

Range trees:





Fractional Cascading: Layered Range Tree

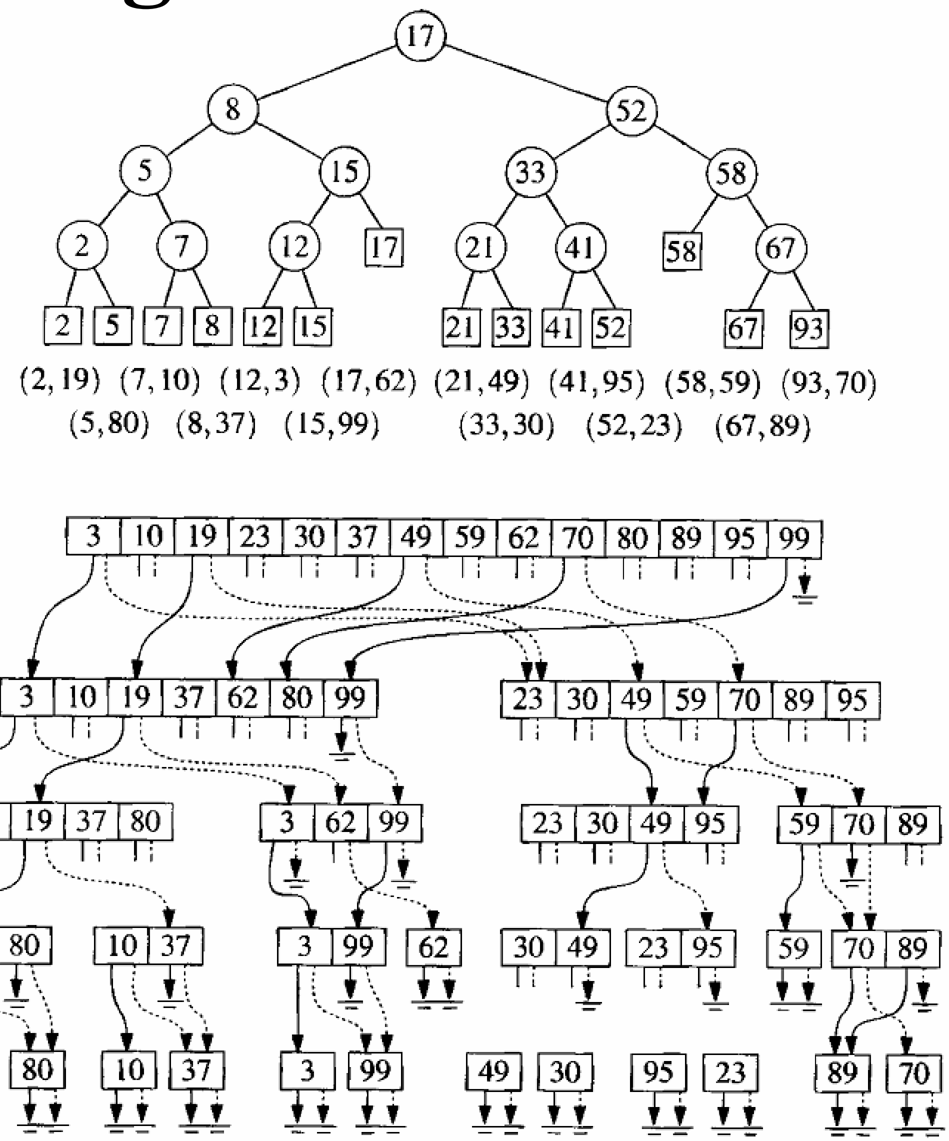
Replace 2D range tree with a layered range tree, using sorted arrays and pointers instead of the secondary range trees.

Preprocessing:

$O(n \log n)$

Query:

$O(\log n + k)$



d-dimensional range trees

Query time: $O(k + \log^{d-1} n)$ to report k points,
uses fractional cascading in the last dimension

Space: $O(n \log^{d-1} n)$

Preprocessing time: $O(n \log^{d-1} n)$

Best data structure to date:

Query time: $O(k + \log^{d-1} n)$ to report k points.

Space: $O(n (\log n / \log \log n)^{d-1})$

Preprocessing time: $O(n \log^{d-1} n)$