Algorithms II, CS 502

Augmenting Data Structures

Ugur Dogrusoz

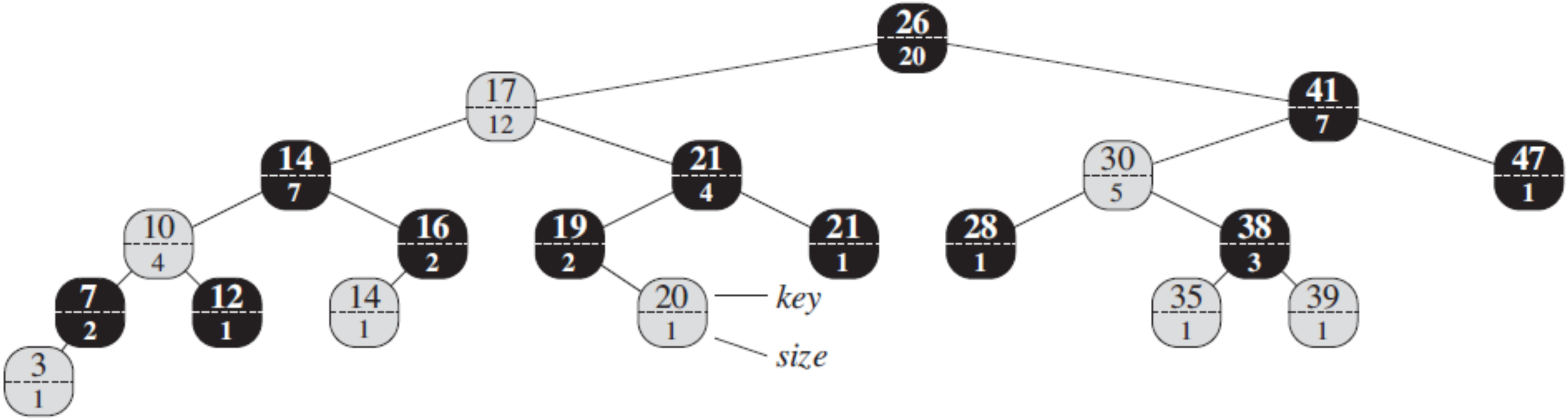
Computer Eng Dept, Bilkent Univ

Augmenting

- Often times “textbook” data structures (DS) are sufficient
  - need to modify for real-life usage of course (we rarely sort “just integers” but rather “objects based on a unique field which is an integer”)
- Frequently, will suffice to augment a textbook DS by storing additional info in it
  - to perform additional operations on the DS

Example: Dynamic order statistics

- Order statistic (OS) trees augment red-black trees:
  - Associate a size field with each node in the tree
  - **`x.size`** keeps size of subtree rooted at **`x`** (including **`x`**)
  - **`x.size = x.left.size + x.right.size + 1`**



Selecting i^{th} element

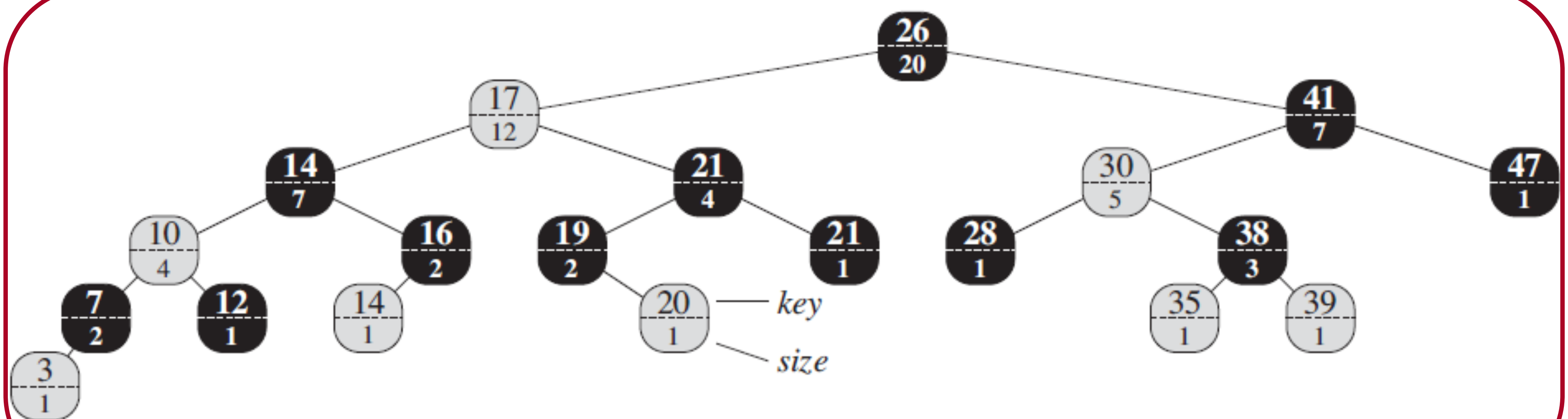
- We can use this new field to select i^{th} element in O(lg n) time

```
OS-SELECT(x, i)
r = x.left.size + 1
if i == r
    return x
elseif i < r
    return OS-SELECT(x.left, i)
else return OS-SELECT(x.right, i − r)
```

Selecting i^{th} element

- `OS-Select(T.root,17)`

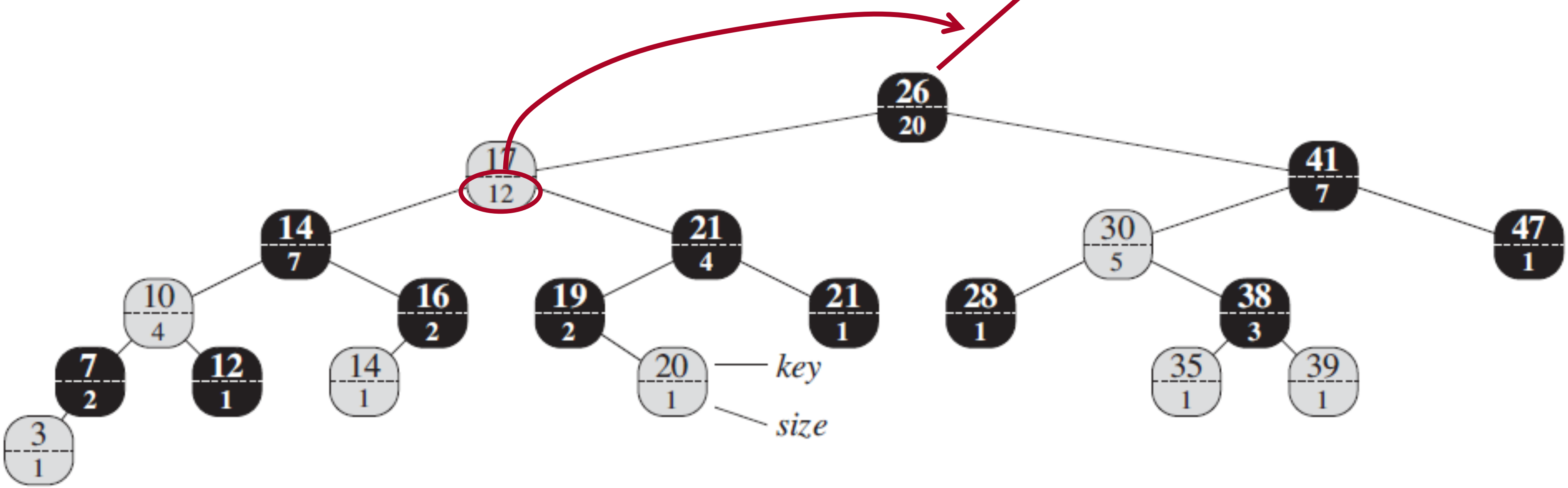
17th in



Selecting i^{th} element

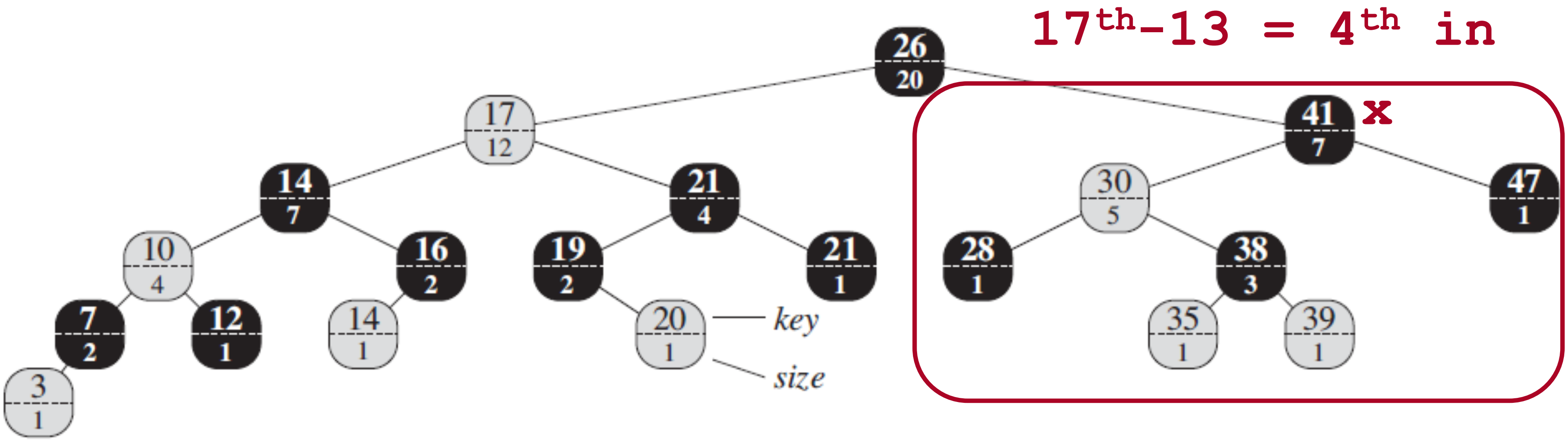
- `OS-Select(T.root,17)`

`rank(T.root)=`
`12+1=13 < 17th`



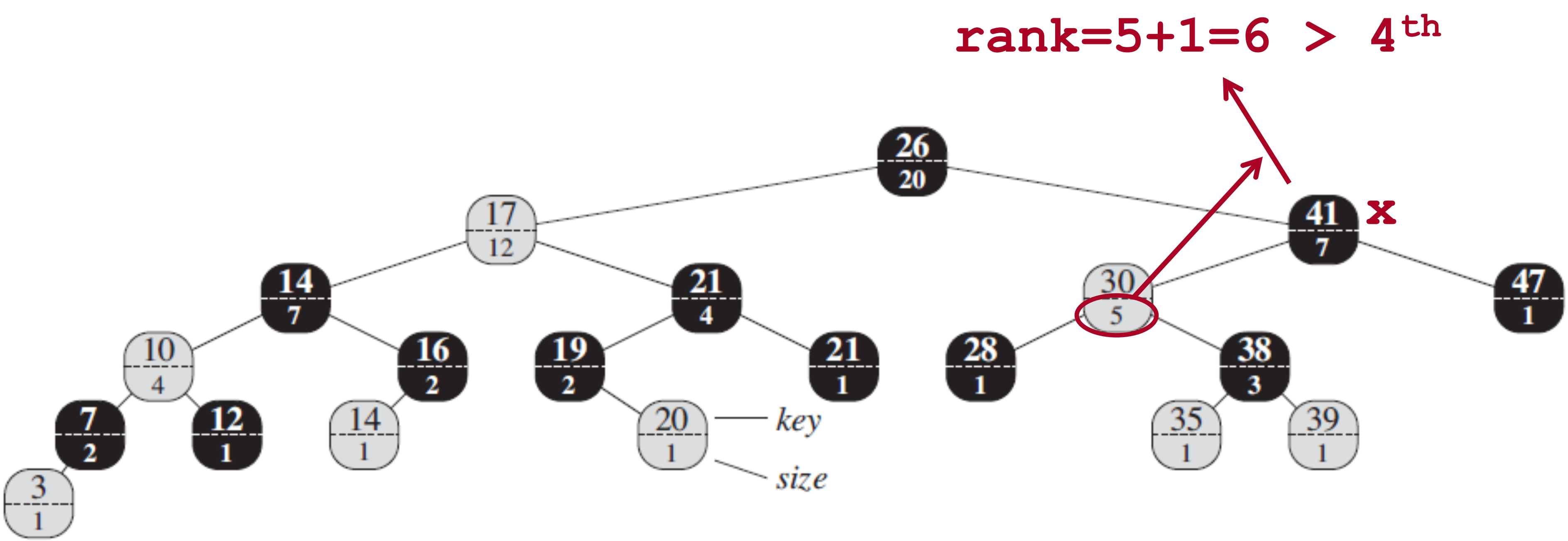
Selecting i^{th} element

- `OS-Select(x,4)`



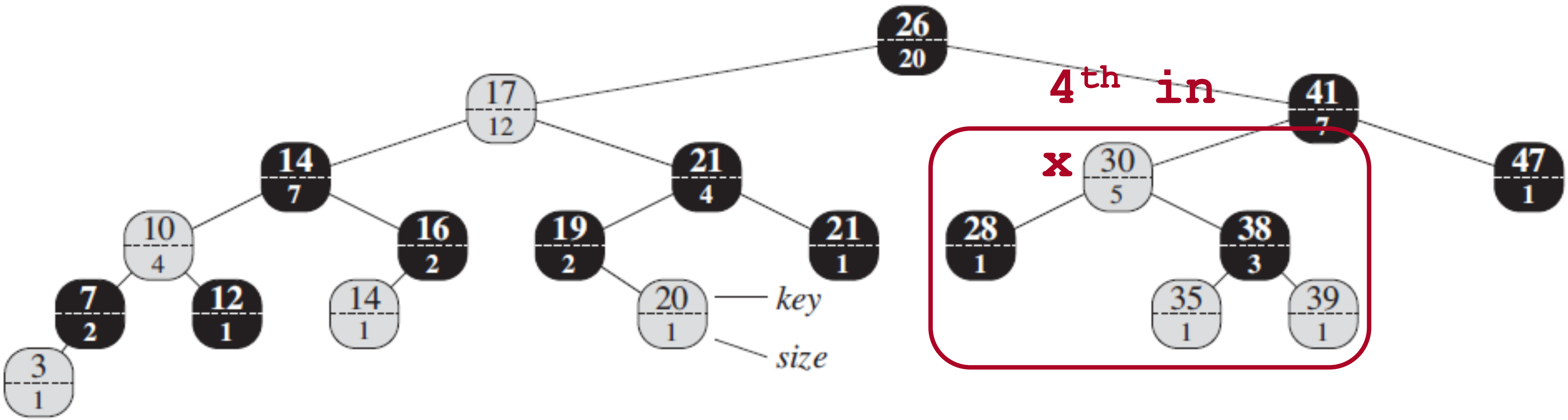
Selecting i th element

- `OS-Select(x,4)`



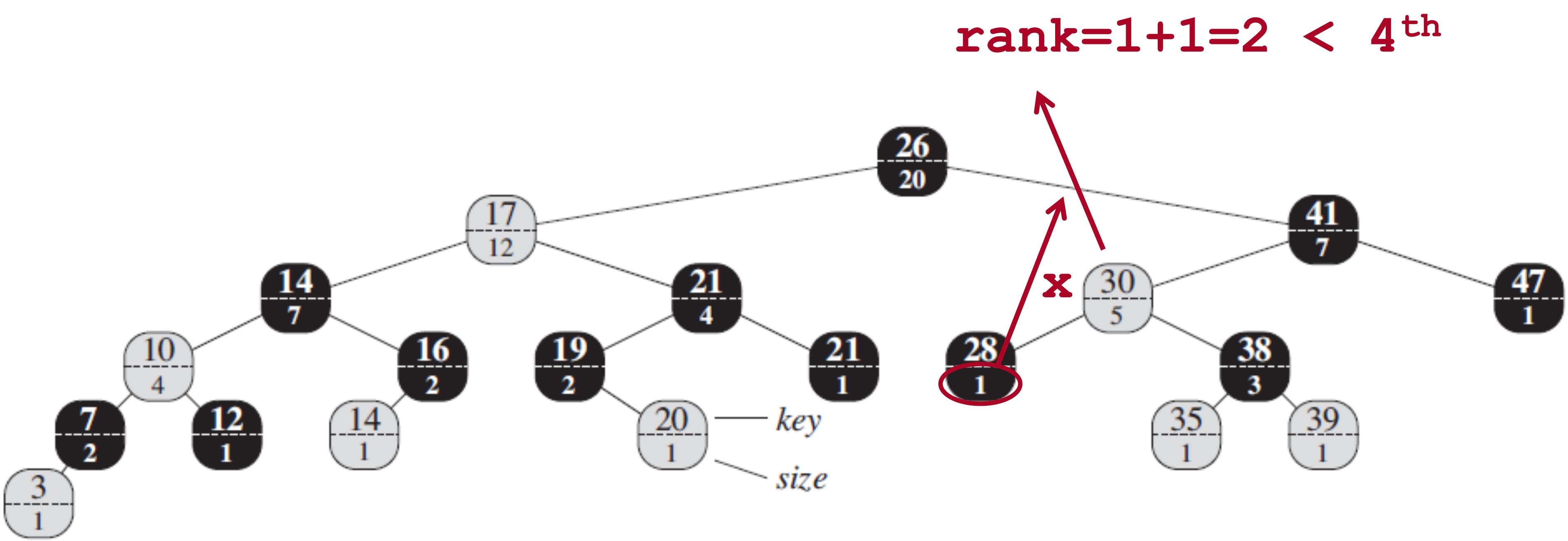
Selecting i^{th} element

- `OS-Select(x,4)`



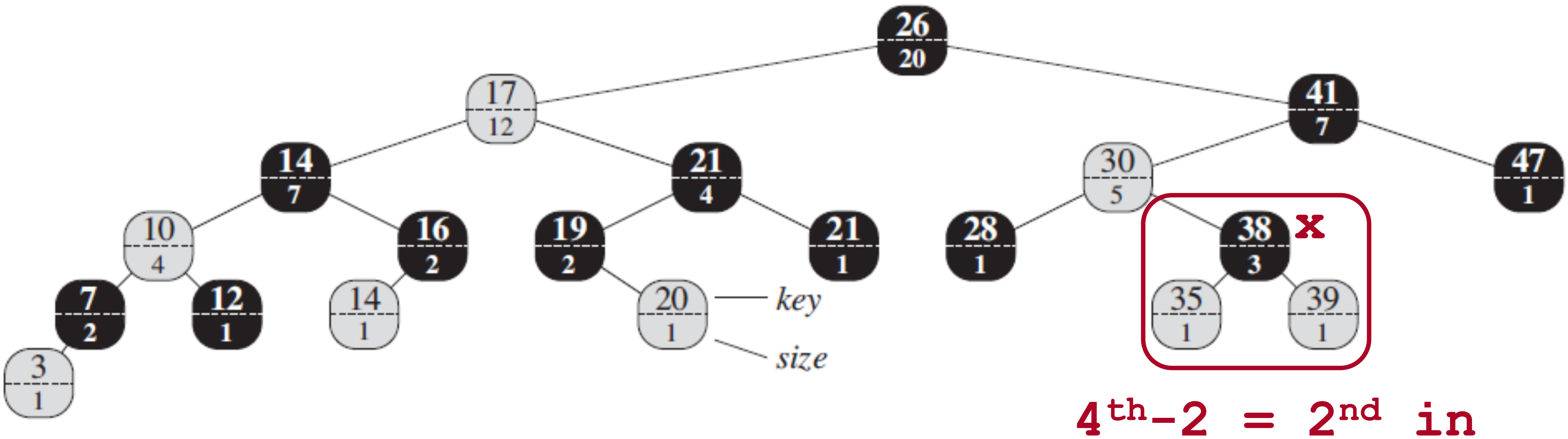
Selecting i^{th} element

- `OS-Select(x,4)`



Selecting i^{th} element

- `OS-Select(x,2)`



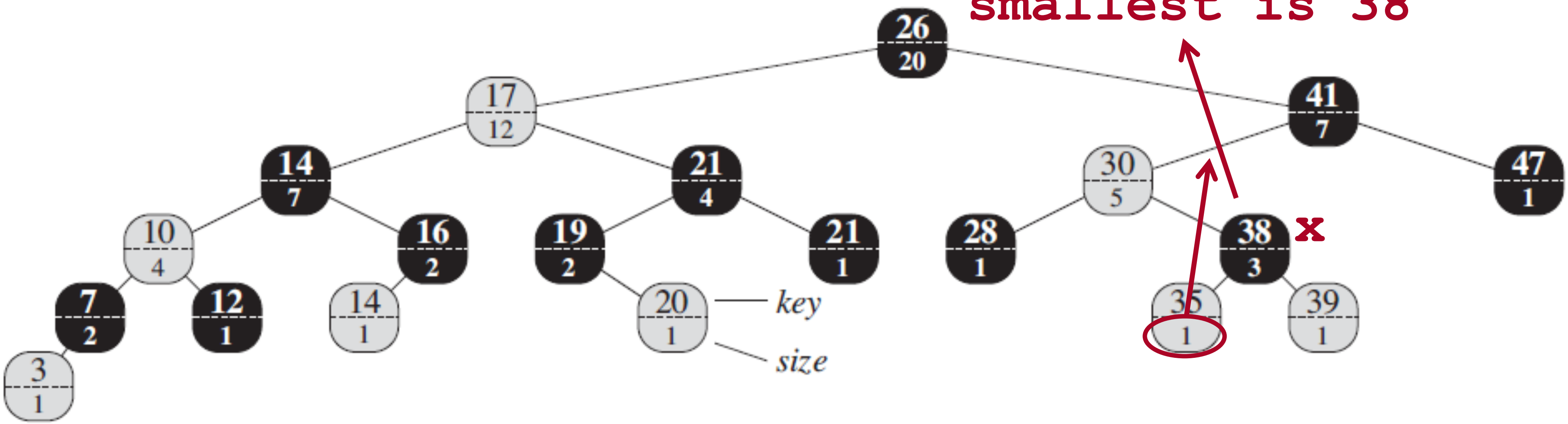
Selecting i^{th} element

- `OS-Select(x,2)`

rank=1+1=2 = 2nd

Bingo! 17th

smallest is 38



Calculating rank

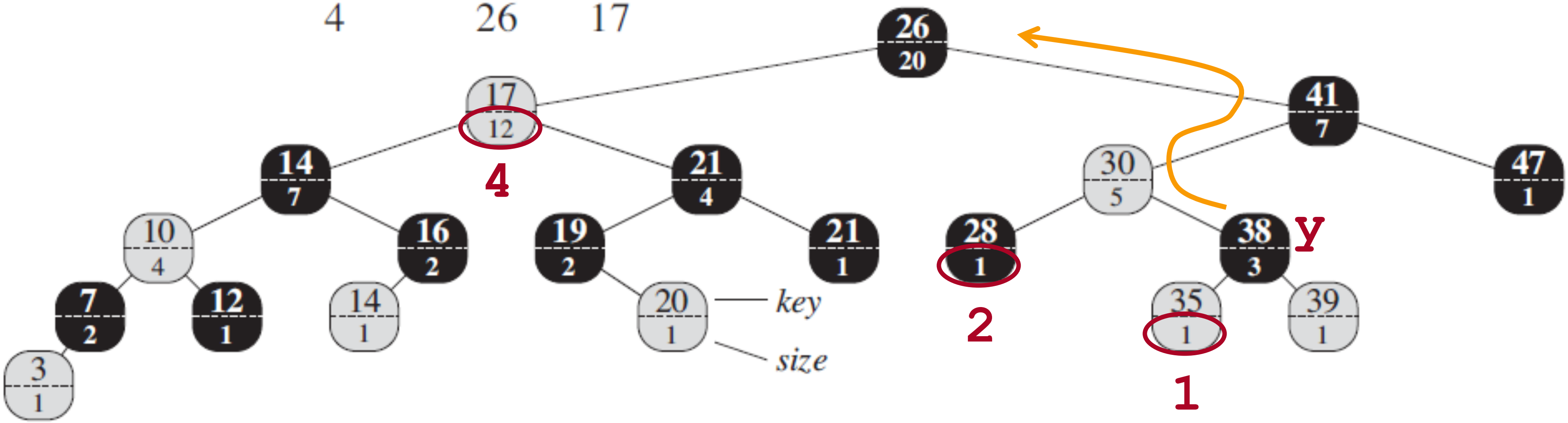
- We can use this new field to calculate rank in O(lg n) time

```
OS-RANK(T, x)
r = x.left.size + 1
y = x
while y ≠ T.root
        if y == y.p.right
                r = r + y.p.left.size + 1
        y = y.p
return r
```

Calculating rank

- `OS-Rank(T,y)`

iteration	*y.key*	*r*
1	38	2
2	30	4
3	41	4
4	26	17

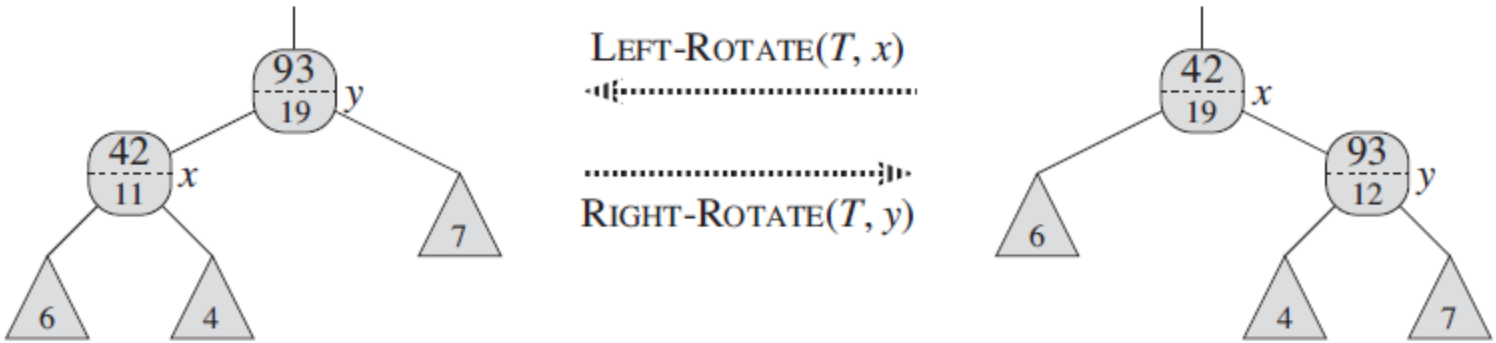


Maintaining subtree sizes: *insertion*

- First phase: go down tree to insert new node as a child of an existing node
  - Increment **`x.size`** for each node on simple path traversed from the root downward leaves
- Second phase: go up tree changing colors and rotating to maintain tree properties
  - Rotations only locally affect **`size`** attribute
  - For **`LEFT-ROTATE(T,x)`** add:
    - ```
      y.size = x.size
      x.size = x.left.size + x.right.size + 1
      ```
      ```

Maintaining subtree sizes: *insertion*

- Example update on rotations



- Additional work:
  - First phase: O(lg n)
  - Second phase: O(1)
- Overall O(lg n) is preserved

Maintaining subtree sizes: *deletion*

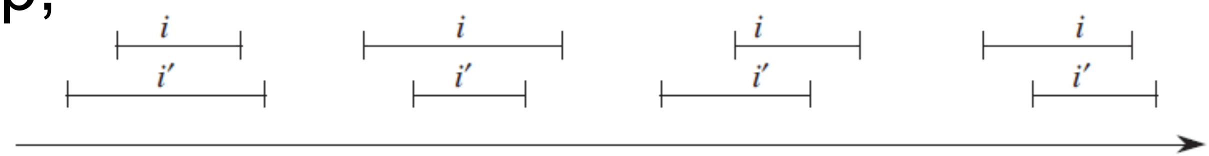
- First phase: only operates on the search tree
  - Either removes one node `y` from tree or moves upward it within the tree
  - Simply traverse a simple path from node `y` up to root, decrementing `size` attribute of each node on the path
  - Additional cost of O(lg n)
- Second phase: causes at most 3 rotations (no other structural changes)
  - Handle similar to *insertion*
  - Additional cost of O(1)
- Overall O(lg n) is preserved

How to augment a data structure

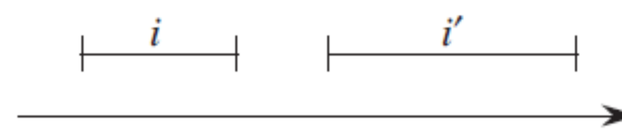
1. Choose an underlying data structure
   - Red-black trees
2. Determine additional information to maintain
   - Add the **`size`** attribute
3. Verify additional information can be maintained for basic modifying operations
   - **`insert`** and **`delete`** still in O(lg n)
4. Develop new operations
   - O(lg n) operations **`OS-Select`** and **`OS-Rank`**

Example: Dynamic set of intervals

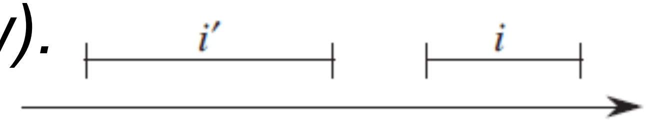
- Interval $[t_1, t_2]$ represents the set $\{t \in \Re \mid t_1 \le t \le t_2\}$
- Any two intervals *i* and *i'* satisfy interval trichotomy:
  - *i* and *i'* overlap,



  - *i* is to the left of *i'* (i.*high* < *i.low*),



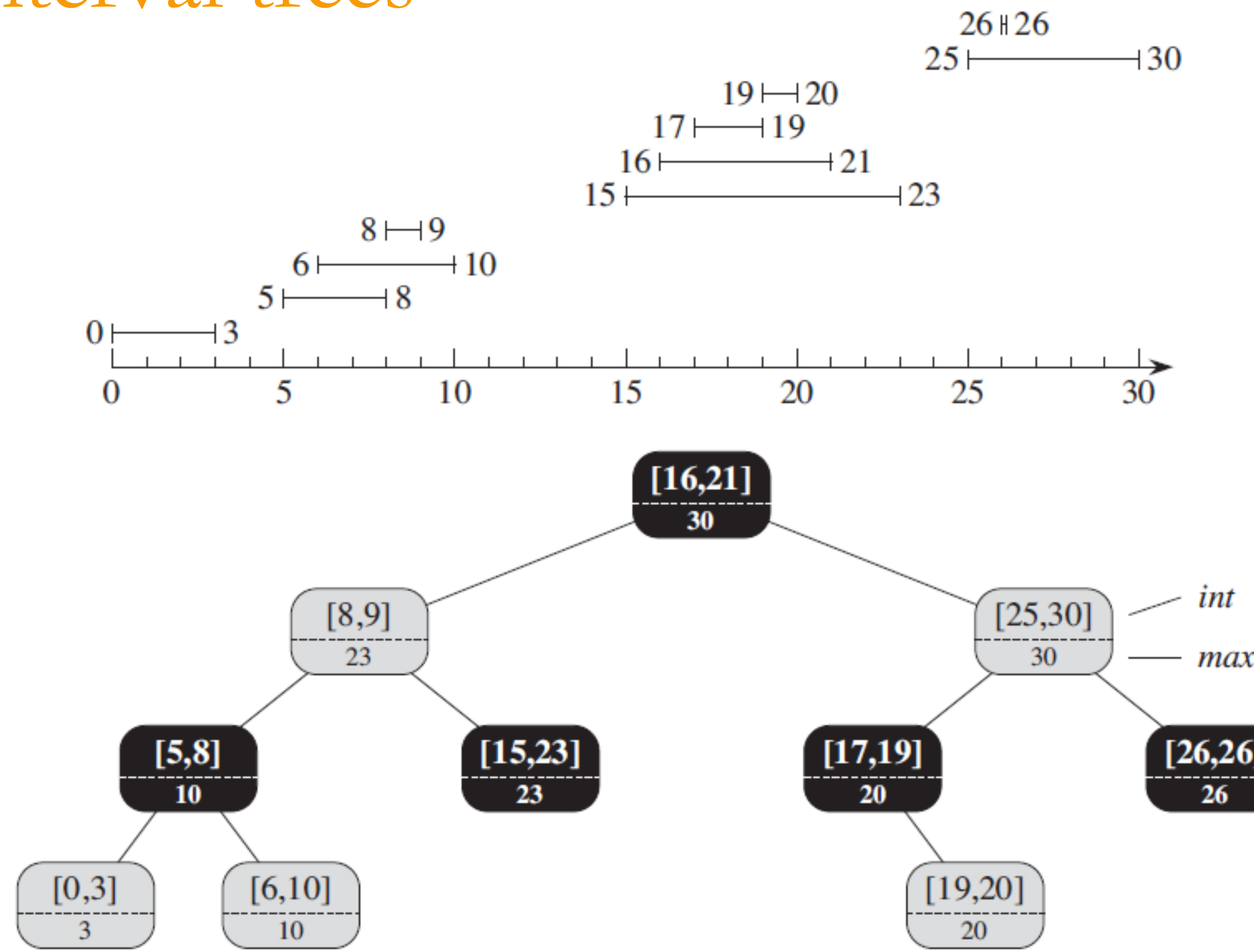
  - *i* is to the right of *i'* (i.*high* < *i.low*).



Interval trees

- A red-black tree that maintains a dynamic set of intervals, with each element **`x`** containing an interval **`x.`*`int`*** supporting
  - **`Interval-Insert(T,x)`** adds element **`x`**, whose ***`int`*** attribute contains an interval, to interval tree **`T`**
  - **`Interval-Delete(T,x)`** removes element **`x`** from interval tree **`T`**
  - **`Interval-Search(T,i)`** returns a pointer to an element **`x`** in interval tree **`T`** such that **`x.`*`int`*** overlaps interval **`i`**, **`T.nil`** otherwise

Interval trees



Interval trees

1. Choose an underlying data structure
   - Red-black trees with key of node **`x`** being ***`x.int.low`***
2. Determine additional information to maintain
   - **`x.`*`max`*`:`** max value of any interval endpoint stored in subtree rooted at **`x`**
3. Verify additional information can be maintained for basic modifying operations
   - **`insert`** and **`delete`** still in O(lg n)

Interval trees

Theorem 14.1 (Augmenting a red-black tree): *If the value of an augmenting field* `f` *for each node* `x` *depends on only the information in nodes* `x`, `x.left`, *and* `x.right`, *then we can maintain values of* `f` *in all nodes of* `T` *during insertion and deletion without asymptotically affecting O(lg n) performance of these operations*

Determine `max` value of a node as follows:

`x.max = max(x.int.high,x.left.max,x.right.max)`

In fact, rotations only take O(1) additional time

Interval trees

4. Develop new operations

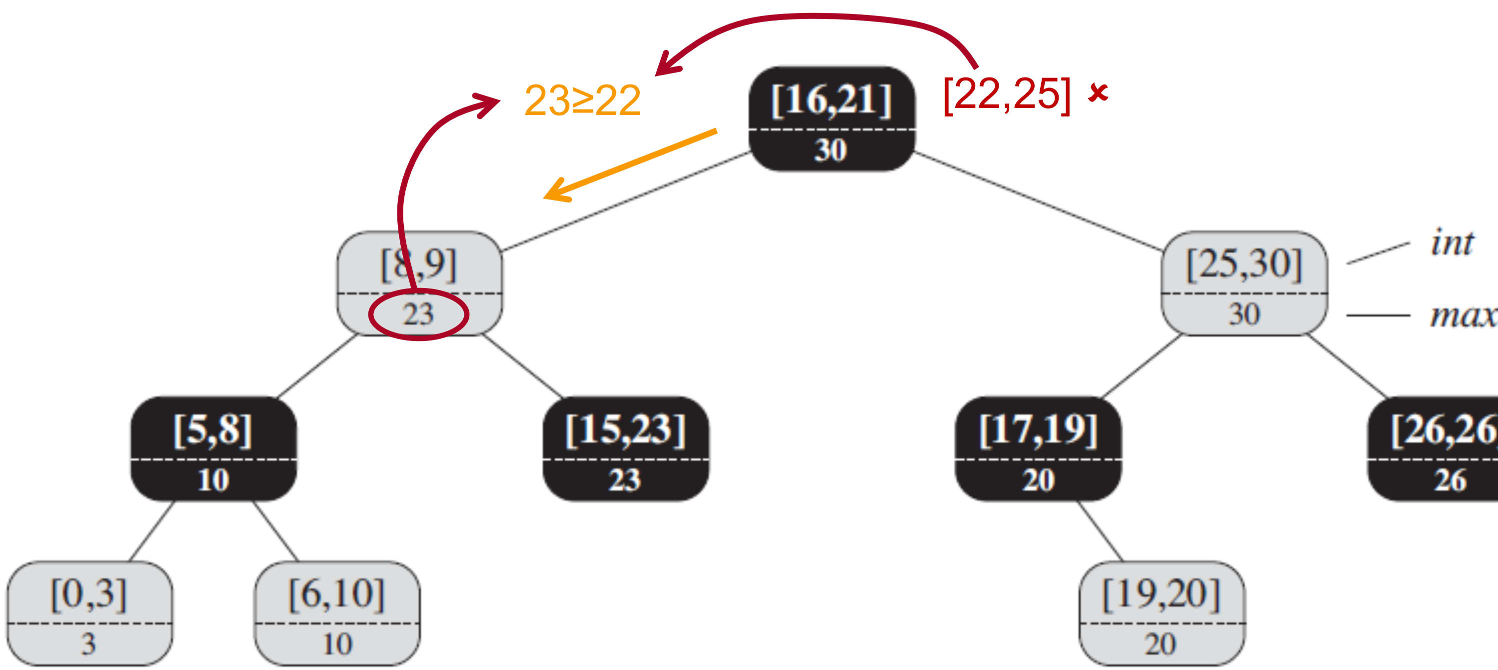
- O(lg n) operation **`Interval-Search(T,i)`** returns a node in tree **`T`** whose interval overlaps **`i`**; **`T.nil`** otherwise.

INTERVAL-SEARCH(T, i)

$x = T.root$
while $x \neq T.nil$ and i does not overlap $x.int$
    if $x.left \neq T.nil$ and $x.left.max \geq i.low$
        $x = x.left$
    else $x = x.right$
return x

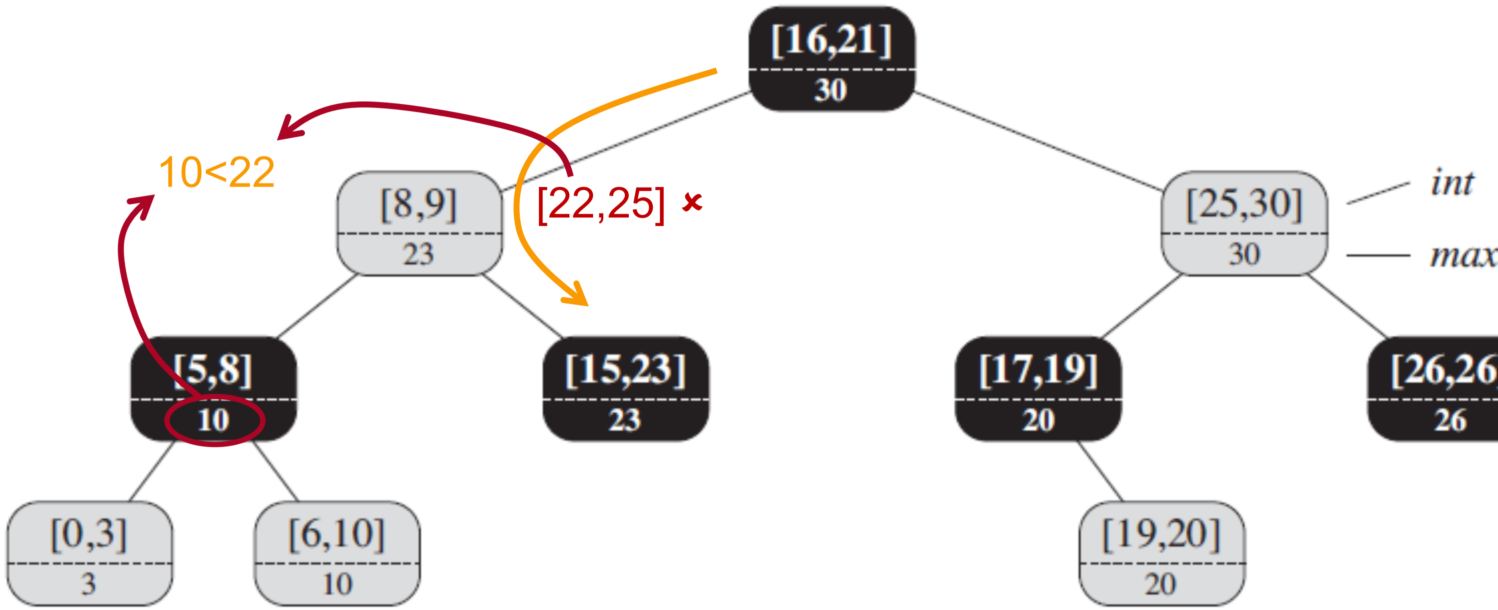
Interval trees

- **`Interval-Search(T,i=[22,25])`**



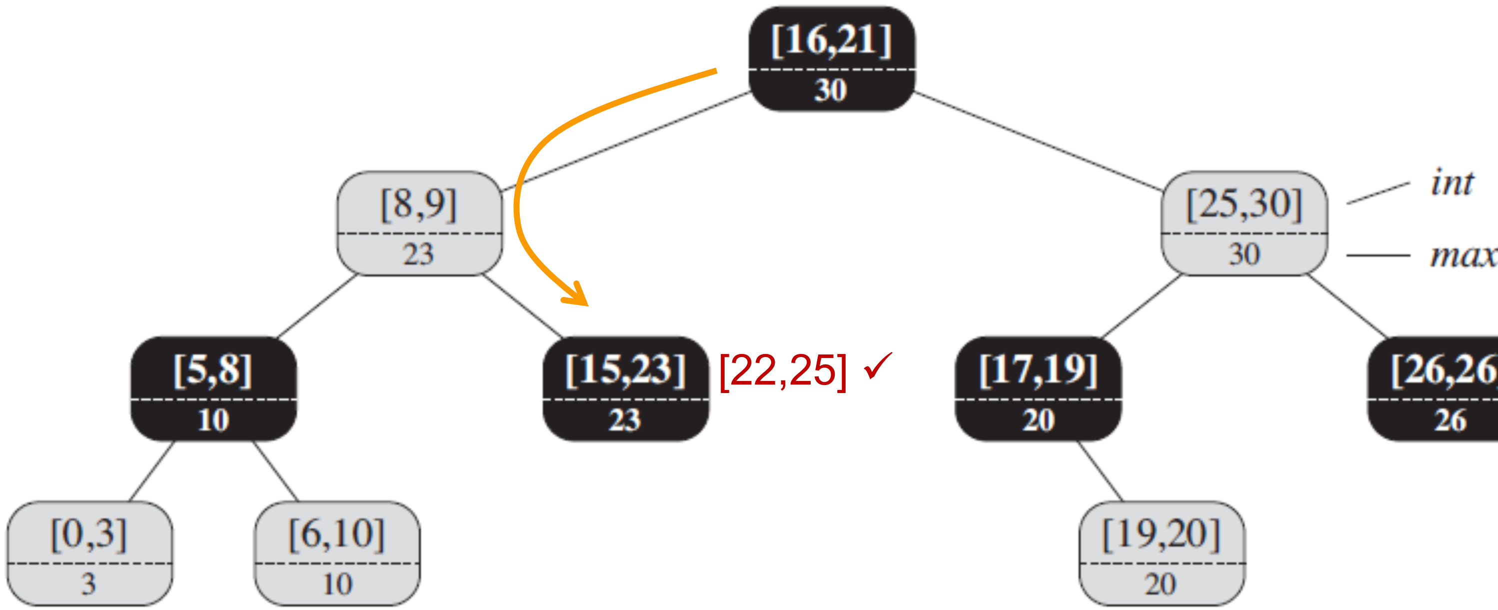
Interval trees

- `Interval-Search(T,i=[22,25])`



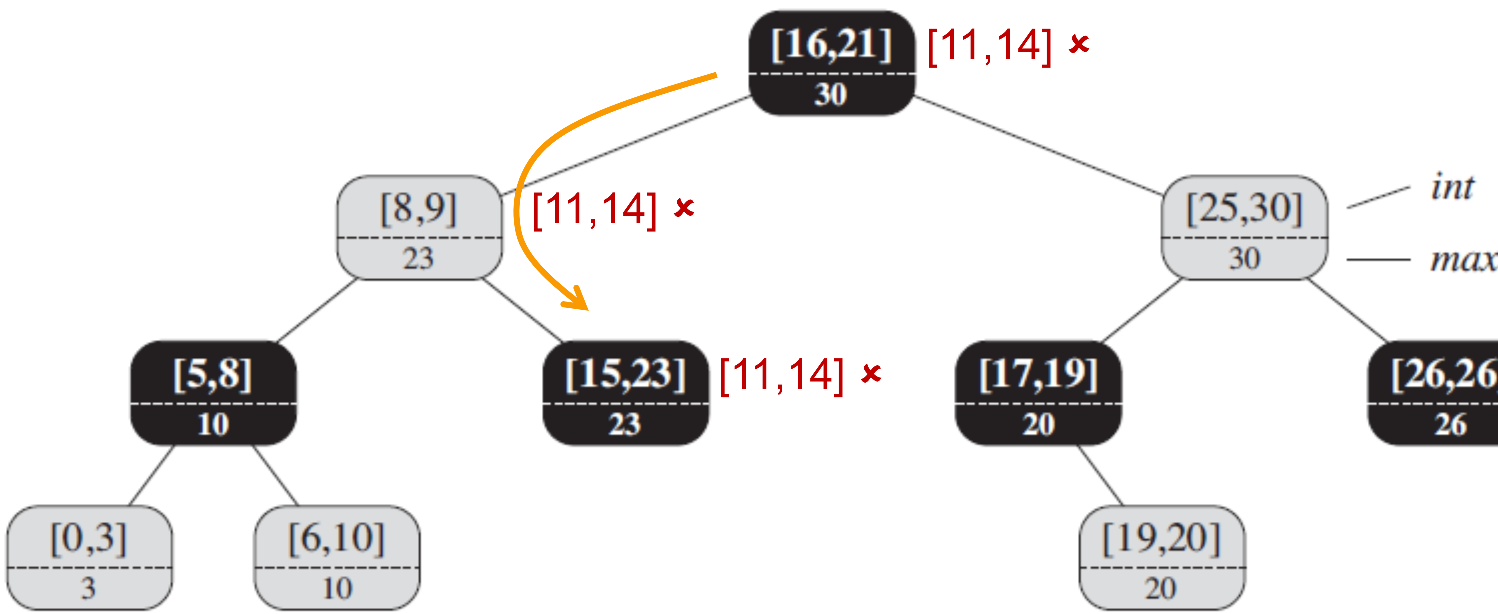
Interval trees

- `Interval-Search(T,i=[22,25])`



Interval trees

- `Interval-Search(T,i=[11,14])`



Interval trees

Theorem 14.2 (Interval-Search works correctly):

Any execution of INTERVAL-SEARCH(T,i) *either returns a node whose interval overlaps i, or it returns $T.nil$ and the tree T contains no node whose interval overlaps i.*

Proof: *Invariant:* If tree T contains an interval that overlaps i, then the subtree rooted at x contains such an interval.

- Initialization (line 1), Maintenance (line 4 or 5), Termination (line 2)

```
INTERVAL-SEARCH(T, i)
1  x = T.root
2  while x ≠ T.nil and i does not overlap x.int
3      if x.left ≠ T.nil and x.left.max ≥ i.low
4          x = x.left
5      else x = x.right
6  return x
```