AUGMENTING DATA STRUCTURES

Prof. Zheng Zhang

Harbin Institute of Technology, Shenzhen

INTRODUCTION

- In most cases a standard data structure is sufficient (possibly provided by a software library).
- But sometimes one needs additional operations that aren't supported by any standard data structure.
- $\rightarrow$ need to design new data structure?

Not always: often augmenting an existing structure is sufficient!

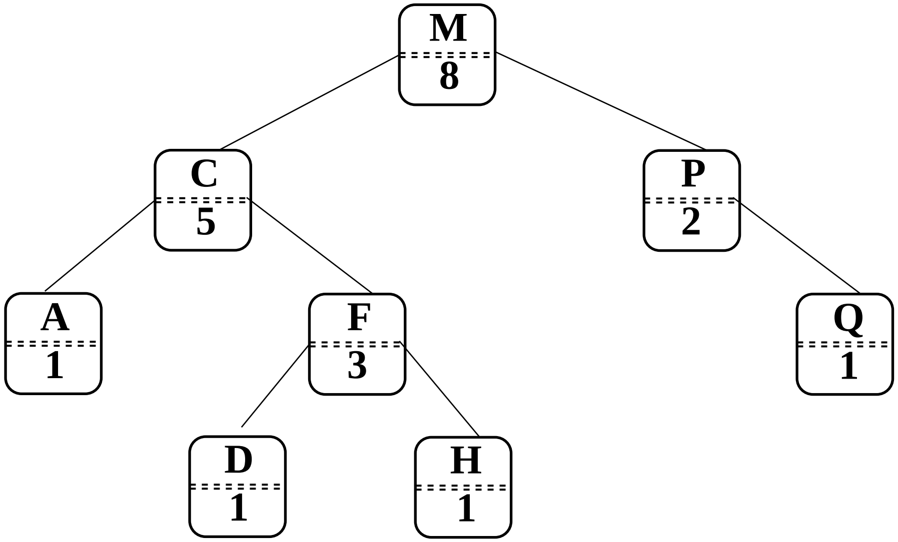
"One good thief is worth ten good scholars"

Dynamic Order Statistics

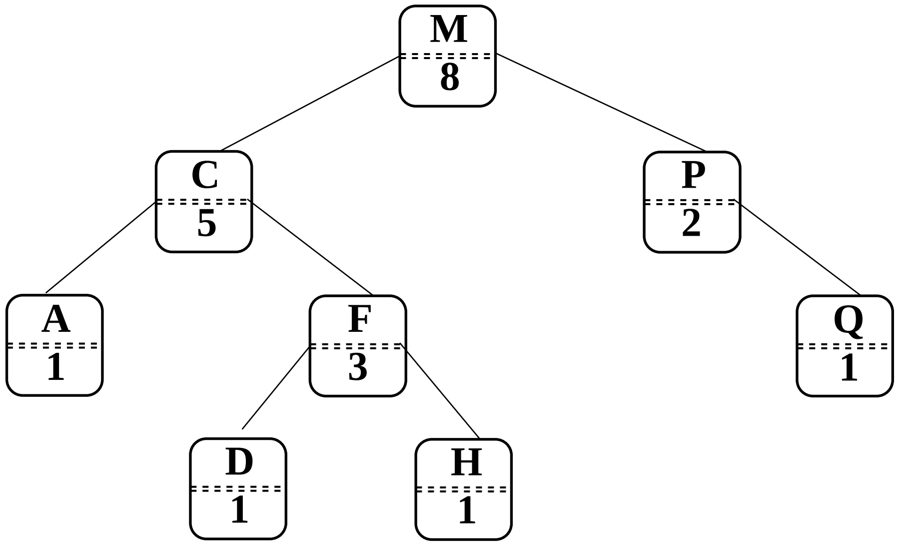
- We've seen algorithms for **finding** the *i*-th element of an unordered set in $O(n)$ time.
- **OS-Tree** (order statistic tree) T: a structure to support finding the *i*-th element of a dynamic set in $O(\lg n)$ time.
  - Support standard dynamic set operations (Insert(), Delete(), Min(), Max(), Succ(), Pred()).
  - Also support these order statistic **operations**
    - void OS-Select(root, *i*);
    - int OS-Rank(*x*);

Review-- ORDER STATISTIC TREES

- OS-Trees augment red-black trees
  - Associate a size field with each node in the tree $x \rightarrow$size.
  - Record the size of subtree rooted at x, including x itself: size[nil[T]] = 0



SELECTION ON OS-TREES



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

How can we use this property to select the *i*-th element of the set?

SELECTION ON OS-TREES

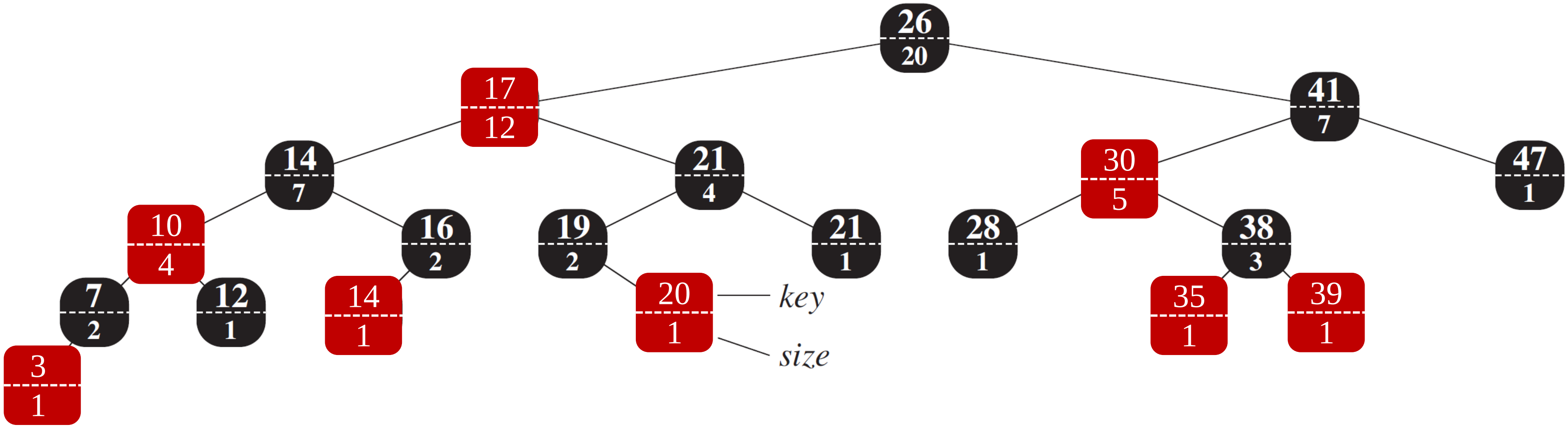


Figure 14.1 An order-statistic tree, which is an augmented red-black tree.

- Do not require keys to be distinct.
- In the presence of equal keys, the above notion of rank is not well defined.
- We remove this ambiguity for an order-statistic tree by defining the rank of an element as *the position* in an inorder walk of the tree.

Selection on OS-Trees

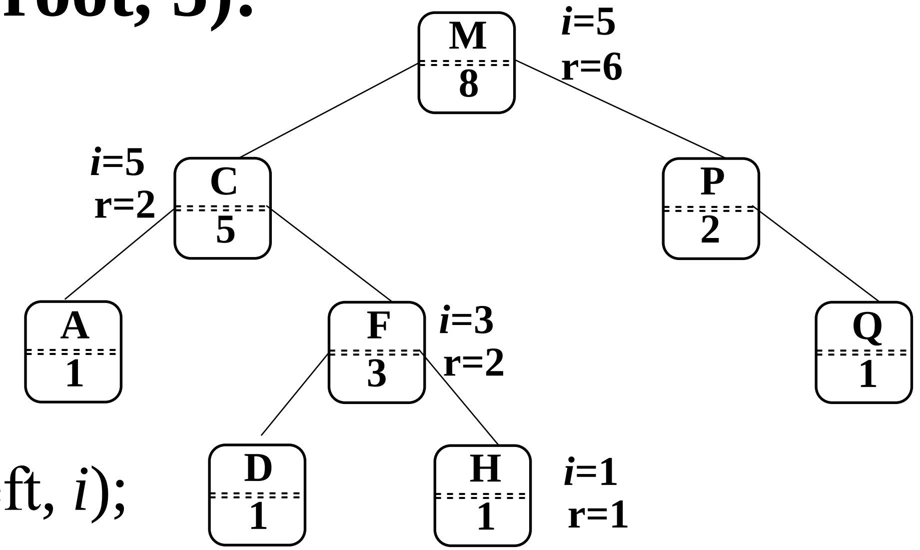
OS-Select(*x*, *i*)

```
{
    r = x→left→size + 1;
    // rank of x within the subtree rooted at x
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x→left, i);
    else
        return OS-Select(x→right, i-r);
}
```

OS-Select Examples

Example: show OS-Select(root, 5):

```
{
    r = x→left→size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x→left, i);
    else
        return OS-Select(x→right, i-r);
}
```

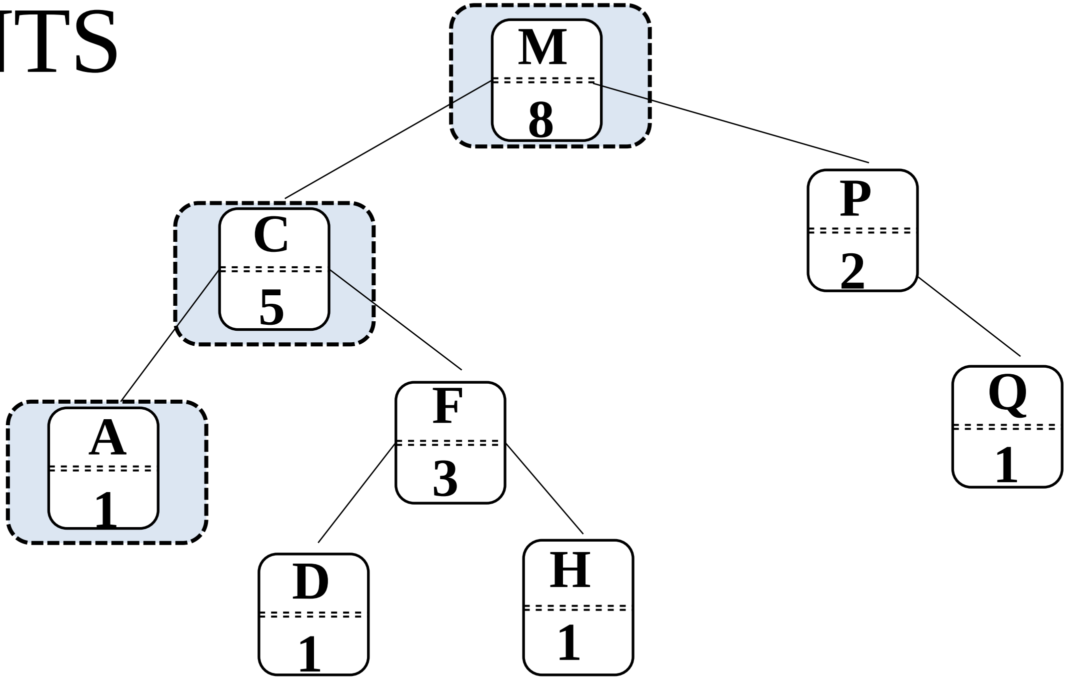


OS-Select: A Subtlety

```
OS-Select(x, i)
{
    r = x→left→size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x→left, i);
    else
        return OS-Select(x→right, i-r);
}
```

1. What happens at the leaves?
2. How can we deal elegantly with this?
3. What will be the running time?

DETERMINING THE RANK OF AN ELEMENTS

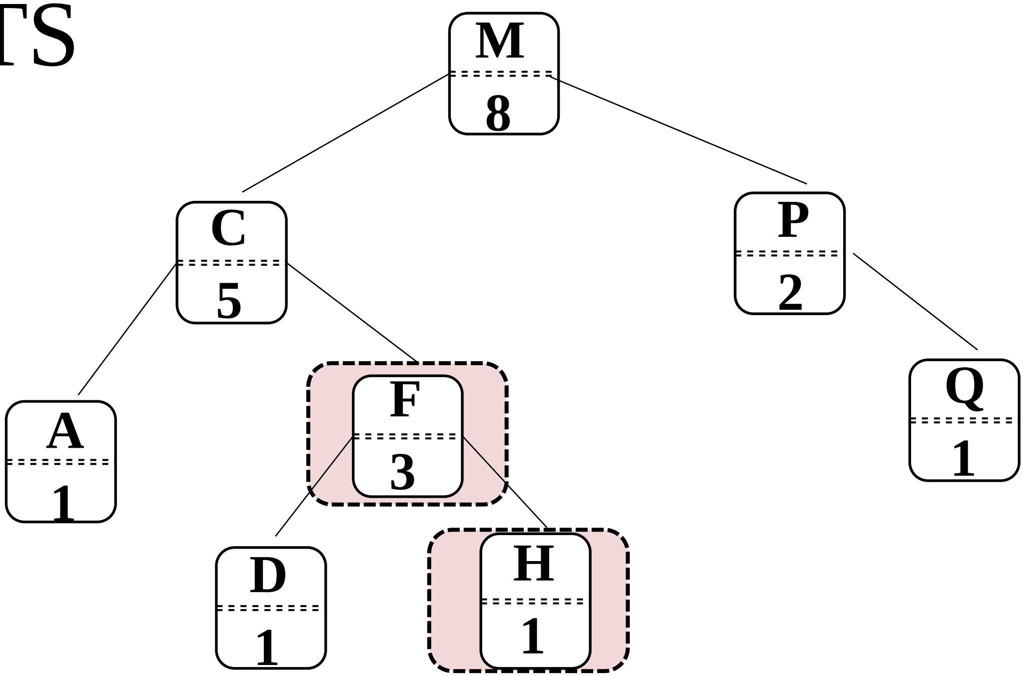


What is the rank of this element?

Of this one? Why?

Of the root? What's the pattern here?

Determining The Rank of an Elements



What is the rank of this element?

This one? What's the pattern here?

OS-Rank

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;
}
```

What will be the running time?

OS-Select Examples

Example: show OS-Select(root, 5):

{

$r = x \rightarrow left \rightarrow size + 1;$

$y = x;$

while (y != $T \rightarrow root$)

if ($y == y \rightarrow p \rightarrow right$)

$r = r + y \rightarrow p \rightarrow left \rightarrow size + 1;$

$y = y \rightarrow p;$

return r;

}

S1: $x = y = 38$ $r=2$

$y = 30$

S2: $r = 2+1+1;$

$y = 41$

S3: $r = 4+12+1$

$y = 26$

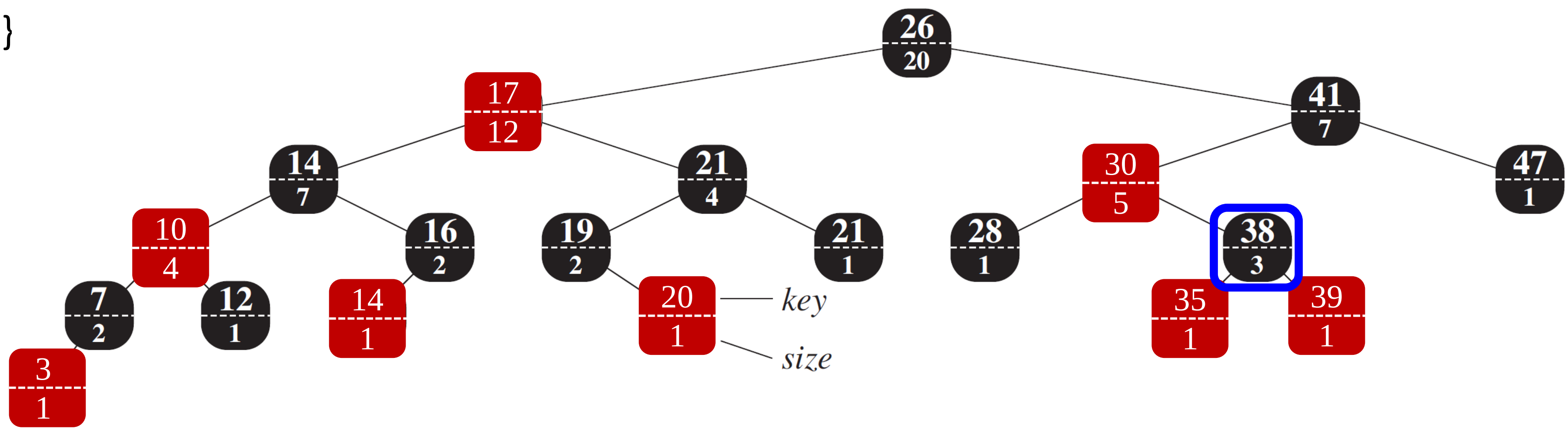


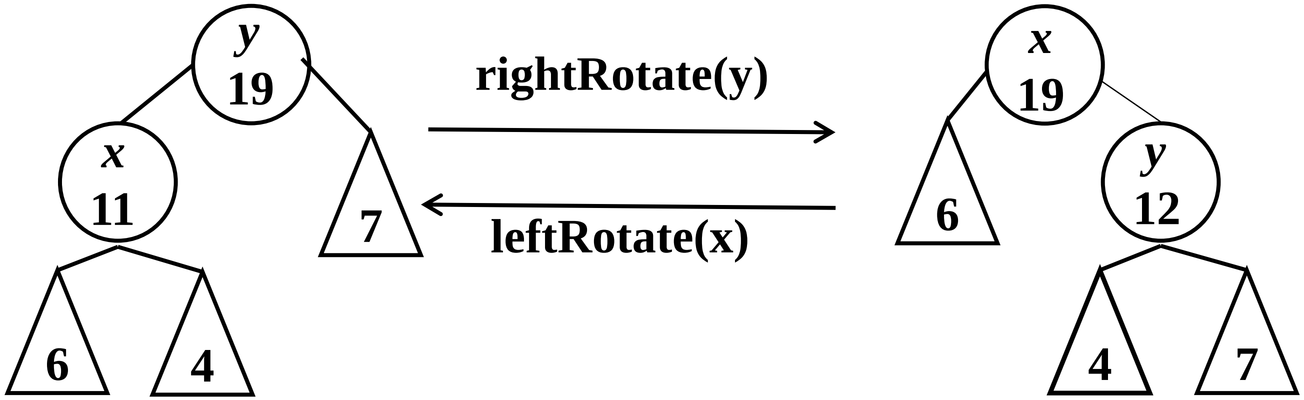
Figure 14.1 An order-statistic tree, which is an augmented red-black tree.

OS-Trees: Maintaining Sizes

- So we've shown that with subtree sizes, order statistic operations can be done in $O(\lg n)$ time.
- Next step: Maintain sizes during Insert() and Delete() operations.
  - How would we adjust the size fields during insertion on a plain binary search tree?

  A: Increment sizes of nodes traversed during search.
  - Why won't this work on red-black trees?

OS-Trees: Maintaining Sizes by Rotation



- Salient point: rotation invalidates only *x* and *y*
- Can recalculate their sizes in constant time $O(1)$
- Why?

12. size[y] $\leftarrow$ size[x]
13. size[x] $\leftarrow$ size[left[x]] + size[right[x]] + 1

Augmenting Data Structure: Methodology

- Choose underlying data structure
  - E.g., red-black trees
- Determine additional information to maintain
  - E.g., subtree sizes
- Verify that information can be maintained for operations that modify the structure
  - E.g., Insert(), Delete() (don’t forget **rotations**!)
- Develop new operations
  - E.g., OS-Rank(), OS-Select()

AUGMENTING DATA STRUCTURE: INTERVAL TREES

- The problem : maintain a set of intervals
  - E.g. , time intervals for a scheduling program:

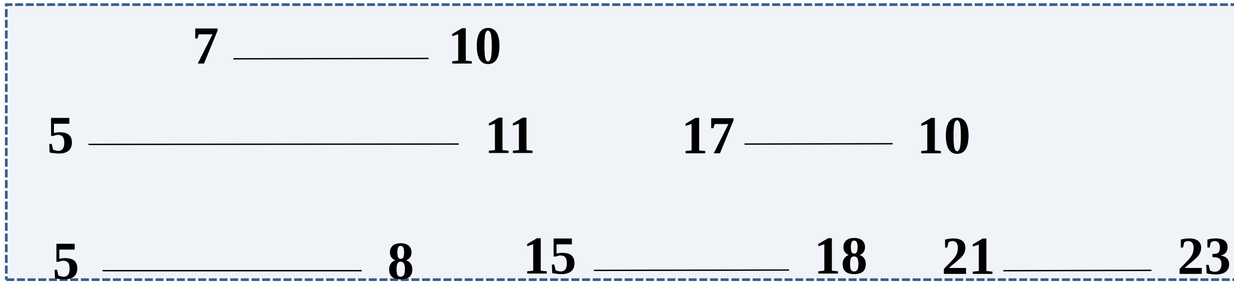
i = [7,10], $i \rightarrow$low=7, $i \rightarrow$high=10

7 ______ 10

5 ______ 11 17 ______ 10

5 ______ 8 15 ______ 18 21 ______ 23

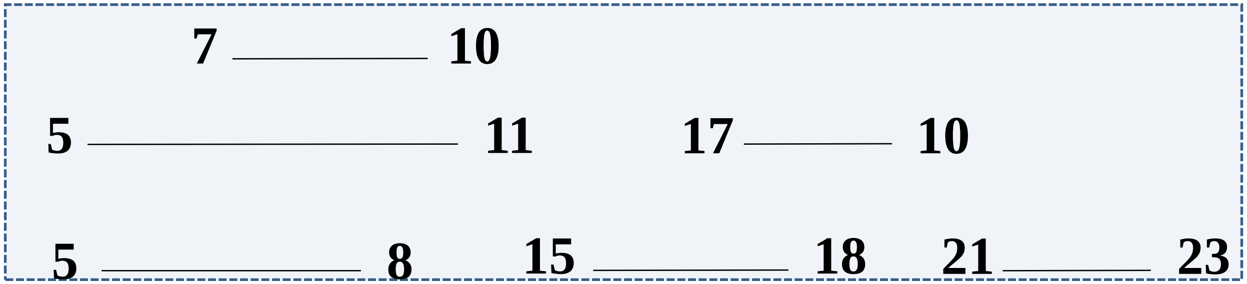
Augmenting Data Structure: Interval Trees



- The problem : maintain a set of intervals
  - E.g. , time intervals for a scheduling program
  - We can represent an interval[t_1, t_2] as an object i, with field slow[i]=t_1(**the low endpoint**) and high[i]=t_2(**the high endpoint**)
  - We say that intervals i and i' overlap if $i \cap i' \neq \emptyset$, that is,

    if low[i] $\leq$ high[i'] and low[i'] $\leq$ high[i].
  - Any two intervals i and i' satisfy the **interval trichotomy**; that exactly one of the **following three properties** holds:

AUGMENTING DATA STRUCTURE: INTERVAL TREES



- The problem : maintain a set of intervals
  - E.g. , time intervals for a scheduling program
  - Any two intervals i and i' satisfy <u>the interval trichotomy</u>; that exactly **one of** the following three properties holds:
    - **a**. i and i' overlap.
    - **b**. i is to the left of i' (*i.e.*, high[i] < low[i']).
    - **c**. i is to the right of i' (*i.e.*, high[i'] < low[i]).

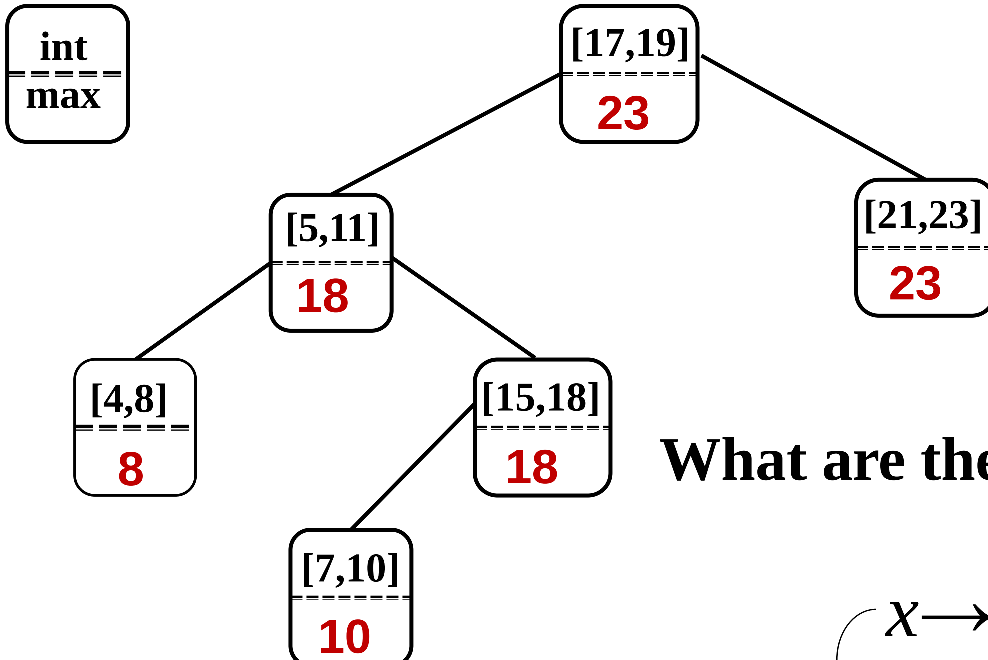
Augmenting Data Structure: Interval Trees

- Interval trees support the following operations.
  - INTERVAL-INSERT(T, x)
  - INTERVAL-DELETE(T, x)
  - INTERVAL-SEARCH(T, i)

Augmenting Data Structure: Interval Trees

- Following the methodology
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - Decide what additional information to store
    - We will store **max**, the maximum endpoint in the subtree rooted at i.
  - Figure out how to maintain the information
  - Develop the desired new operations

AUGMENTING DATA STRUCTURE: INTERVAL TREES



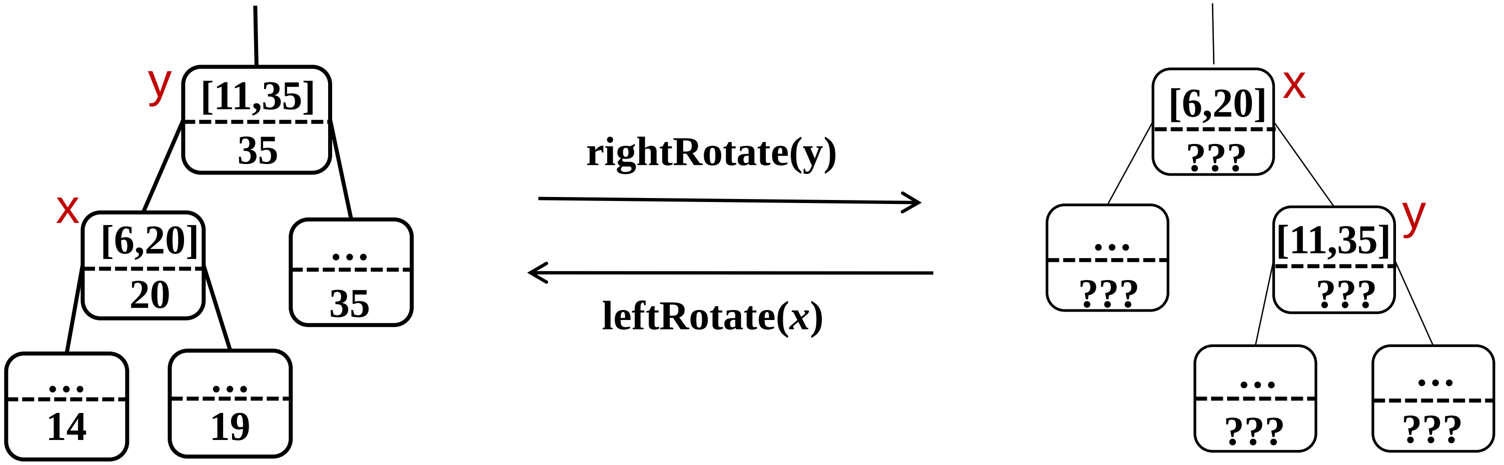
What are the max fields?

$$x\rightarrow\max = \max\begin{cases} x\rightarrow\text{high} \\ x\rightarrow\text{left}\rightarrow\max \\ x\rightarrow\text{right}\rightarrow\max \end{cases}$$

AUGMENTING DATA STRUCTURE: INTERVAL TREES

- Following the methodology
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - Decide what additional information to store
    - We will store **max**, the maximum endpoint in the subtree rooted at i.
  - Figure out how to maintain the information
    - How would we maintain max field for a BST?
    - What's different?
  - Develop the desired new operations

AUGMENTING DATA STRUCTURE: INTERVAL TREES



- What are the new max values for the **subtrees**?

  A: Unchanged

- What are the new max values for *x* and *y***?**

  A: root value unchanged, recompute other

Augmenting Data Structure: Interval Trees

- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - Decide what additional information to store
    - Store the maximum endpoint in the subtree rooted at i.
  - Figure out how to maintain the information
    - ***Insert***: update max on way down, during rotations
    - ***Delete***: similar
  - Develop the desired new operations

Searching Interval Trees

IntervalSearch(T, *i*)

{

x = T→root;

while (x != NULL && !overlap(i, x→interval))

if (x→left != NULL && x→left→max ≥ i→low)

x = x→left; ①

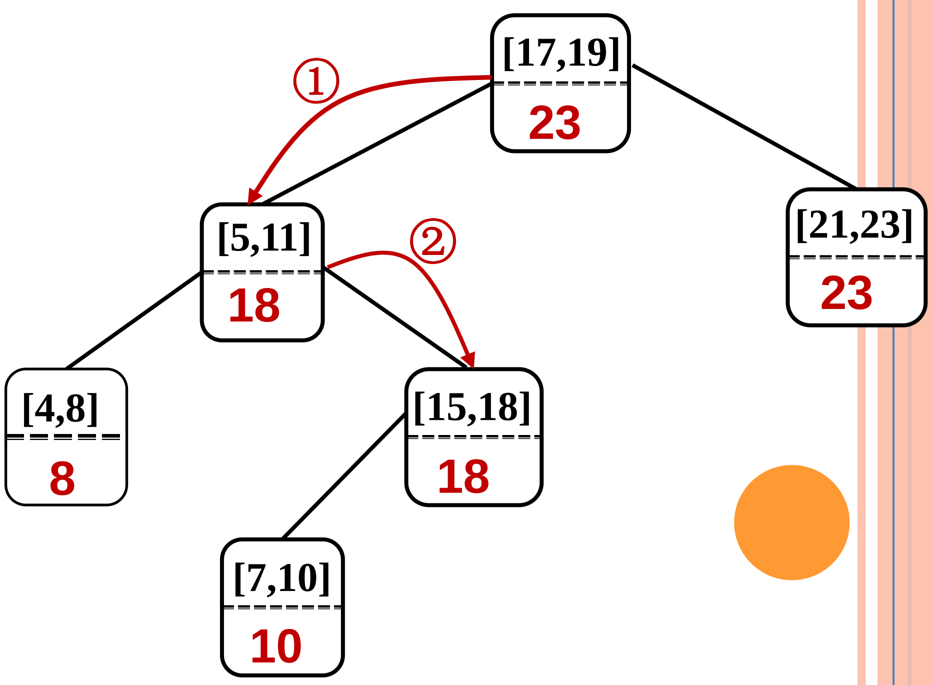
else

x = x→right; ②

return x;

}

What will be the running time?



SEARCHING INTERVAL TREES

Example: search for interval overlapping [14,16]

IntervalSearch(T, *i*)

{*x* = T→root;

while (*x* != NULL && !overlap(*i*, *x*→interval))

if (*x*→left != NULL && *x*→left→max ≥ *i*→*low*)

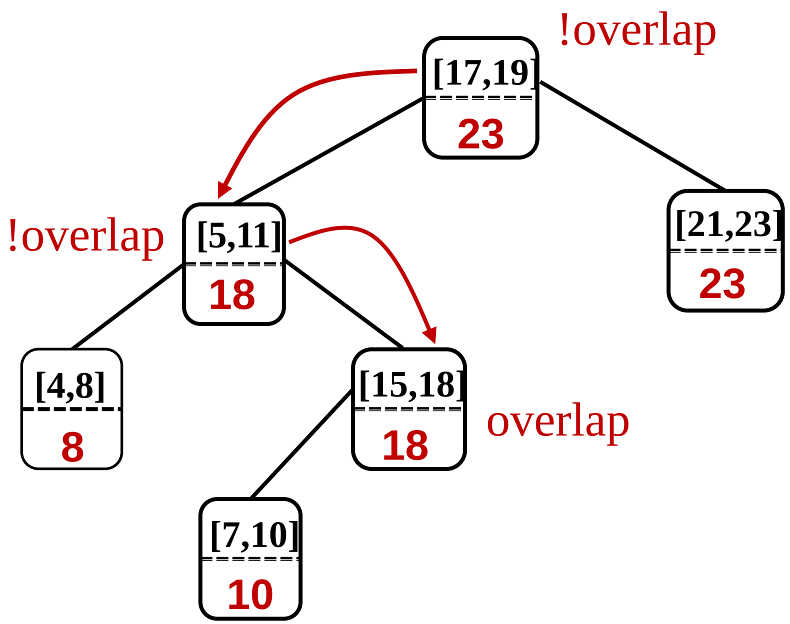
x = *x*→left;

else

x = *x*→right;

return *x*;

}



Searching Interval Trees

Example: search for interval overlapping [12,14]

IntervalSearch(T, *i*)

{x = T→root;

while (x != NULL && !overlap(i, x→interval))

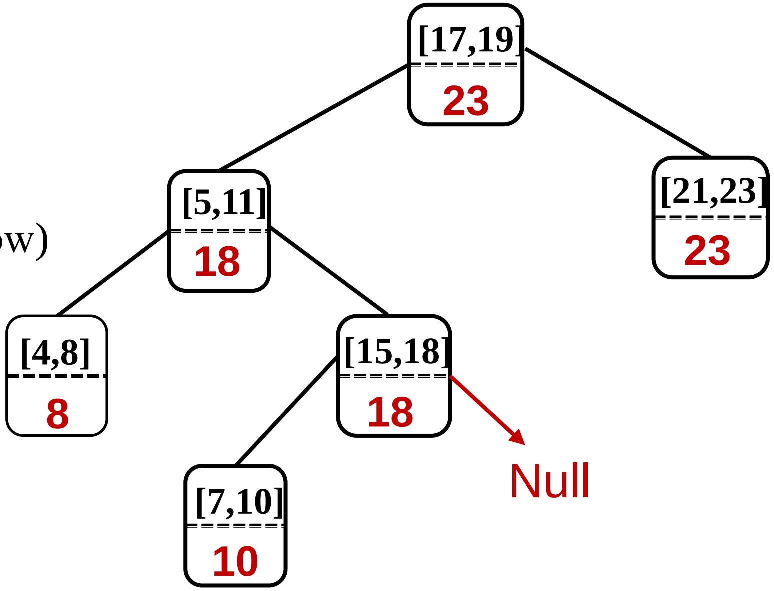
if (x→left != NULL && x→left→max ≥ i→low)

x = x→left;

else x = x→right;

return x;

}



Correctness of Interval Search()

- Key idea: need to check only 1 of node's 2 children
  - **Case 1**: search goes right
    - Show that overlap in right subtree, or no overlap at all
  - **Case 2**: search goes left
    - Show that overlap in left subtree, or no overlap at all

CORRECTNESS OF INTERVAL SEARCH()

- Key idea: need to check only 1 of node's 2 children
  - **Case 1**: search goes <u>right</u>

    If search goes right, overlap in the right subtree or no overlap in either subtree

    - If overlap in right subtree, we're done.
    - Otherwise:
      - $x \rightarrow$left = NULL, ***or*** $x \rightarrow$left$\rightarrow$max < $i \rightarrow$low (Why?)
      - Thus, no overlap in left subtree!

```
while (x != NULL && !overlap(i, x→interval)
    if (x→left != NULL && x→left→max ≥ i→low))
        x = x→left;
    else     x = x→right;
return x;}
```

Correctness of Interval Search()

- Key idea: need to check only 1 of node's 2 children
  - **Case 2**: search goes left

  If search goes left, overlap in the left subtree or no overlap in either subtree

  - If overlap in left subtree, we're done.
  - Otherwise:
    - $i \to \text{low} \le x \to \text{left} \to \text{max}$ by branch condition.
    - $x \to \text{left} \to \text{max} = i' \to \text{high}$ for some i' in x's left subtree.
    - Since i and i' don't overlap and $i \to \text{low} \le i' \to \text{high}$, $i \to \text{high} < i' \to \text{low}$
    - Since tree is sorted by low's, $i \to \text{high}$ < any low, in right subtree.
    - Thus, no overlap in right subtree.

```
while (x != NULL && !overlap(i, x→interval)
    if (x→left != NULL && x→left→max ≥ i→low))
        x = x→left;
      else    x = x→right;
 return x;}
```