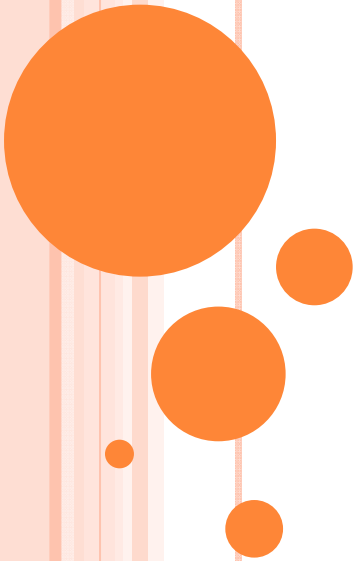
AUGMENTING DATA STRUCTURES



Autumn 2011

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

➨ 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 *ith* element of an unordered set in O(n) time

OS-Tree(order statistic tree) *T:* a structure to support finding the *ith* 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);

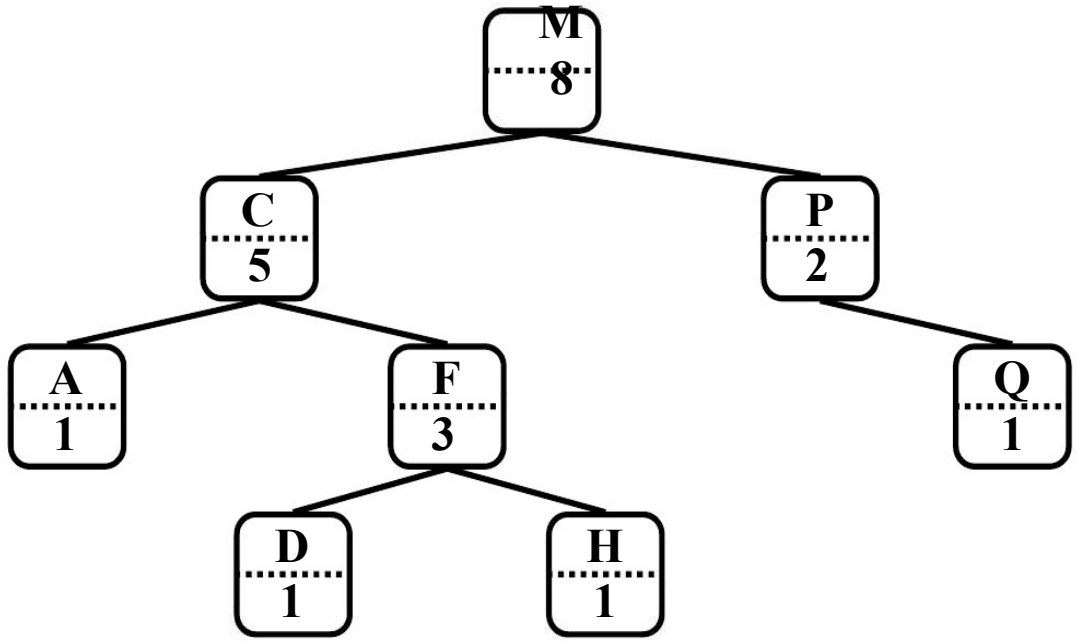
REVIEW: ORDER STATISTIC TREES

OS-Trees augment red-black trees:

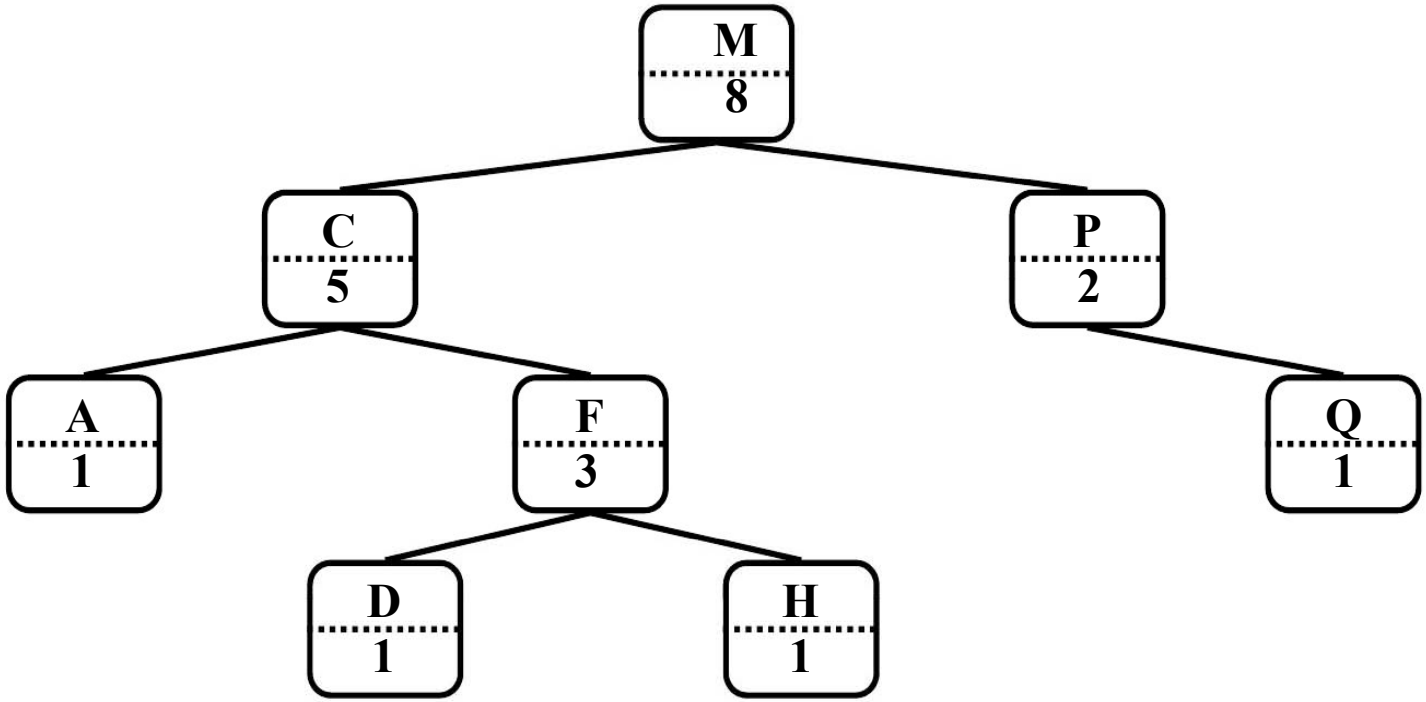
Associate a *size* field with each node in the tree

x->size records 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 *ith* element of the set?

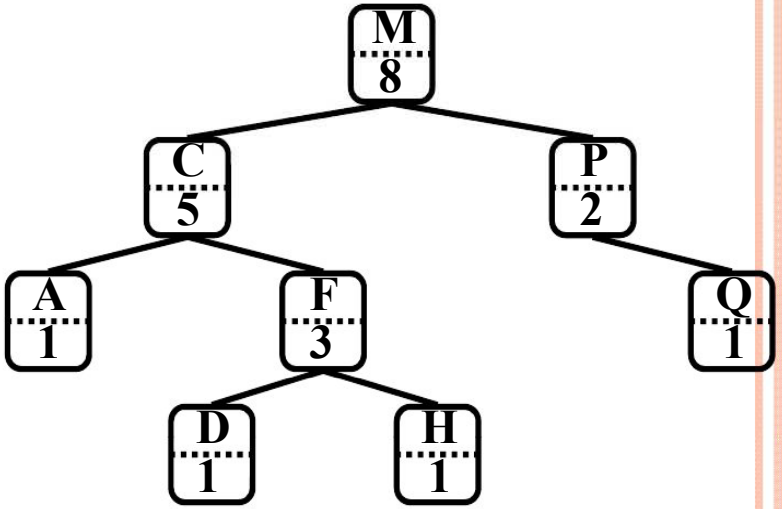
OS-SELECT

```
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);
}
```

OS-SELECT EXAMPLE

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

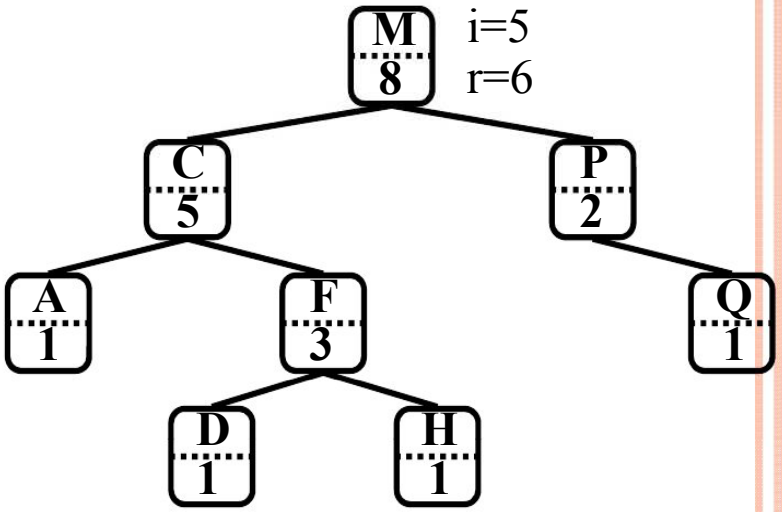
```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```



OS-SELECT EXAMPLE

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

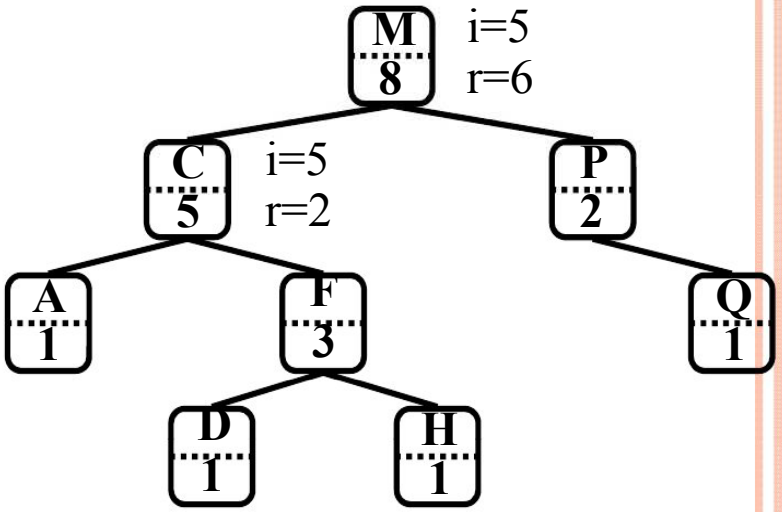
```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```



OS-SELECT EXAMPLE

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

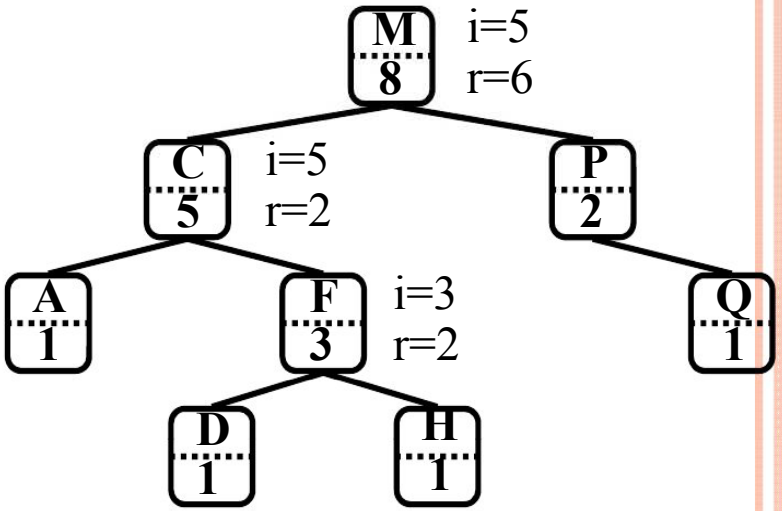
```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```



OS-SELECT EXAMPLE

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

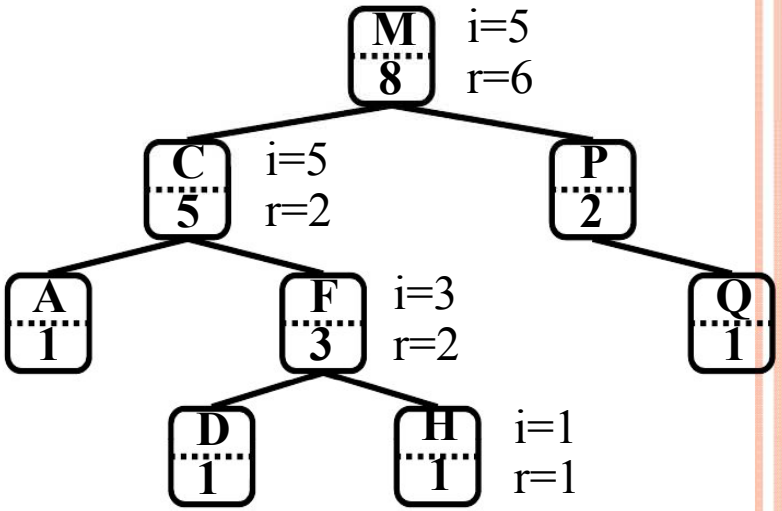
```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```



OS-SELECT EXAMPLE

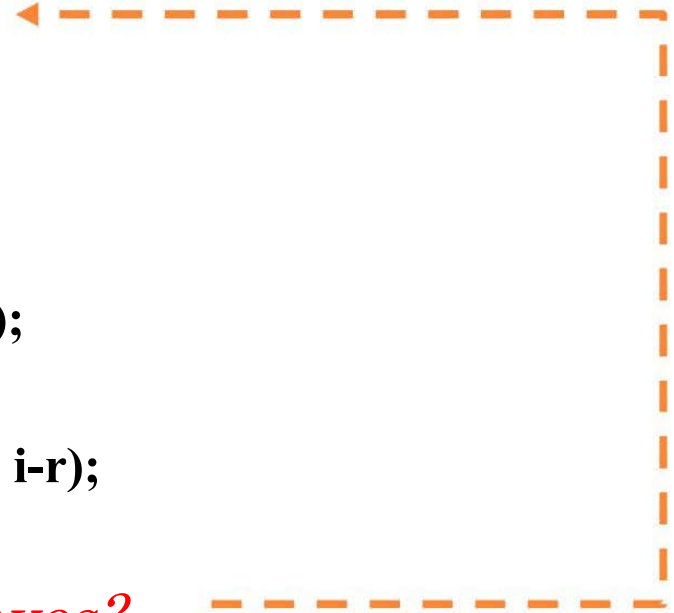
Example: show OS-Select(root, 5):

```
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);
}
```



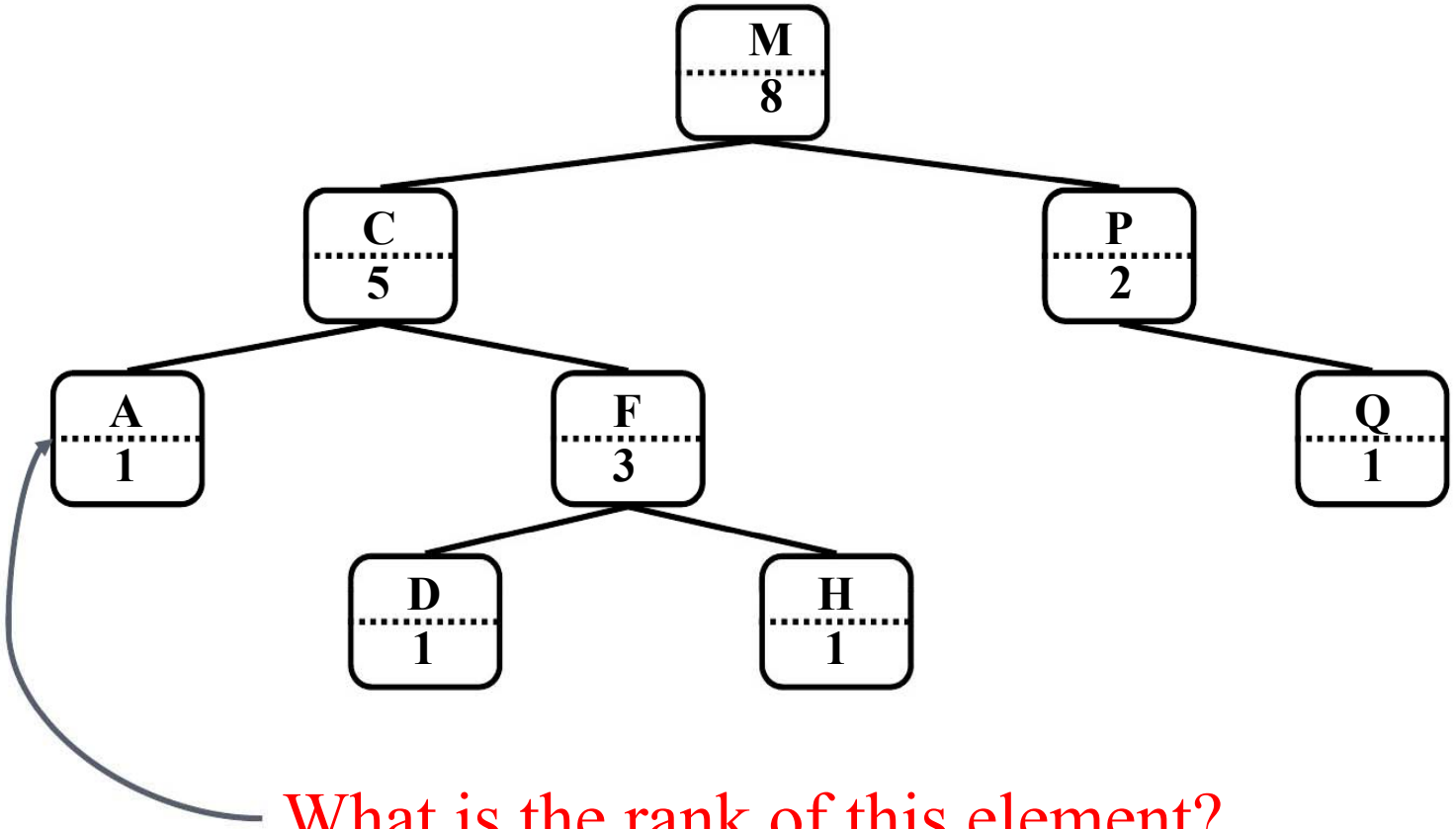
OS-SELECT: A SUBTLETY

```
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);
}
```



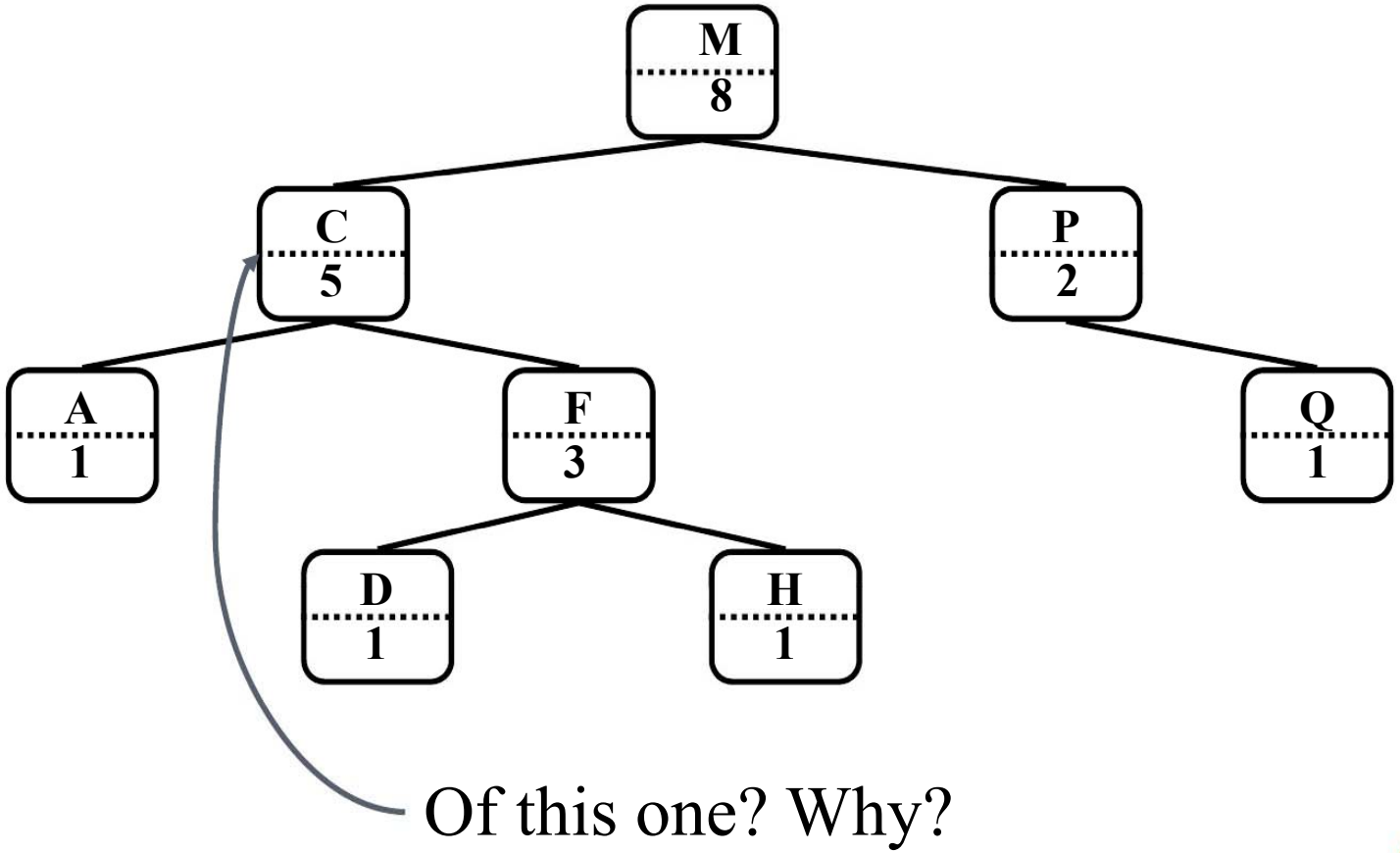
- *What happens at the leaves?*
- *How can we deal elegantly with this?*
- *What will be the running time?*

Determining The Rank Of An Element

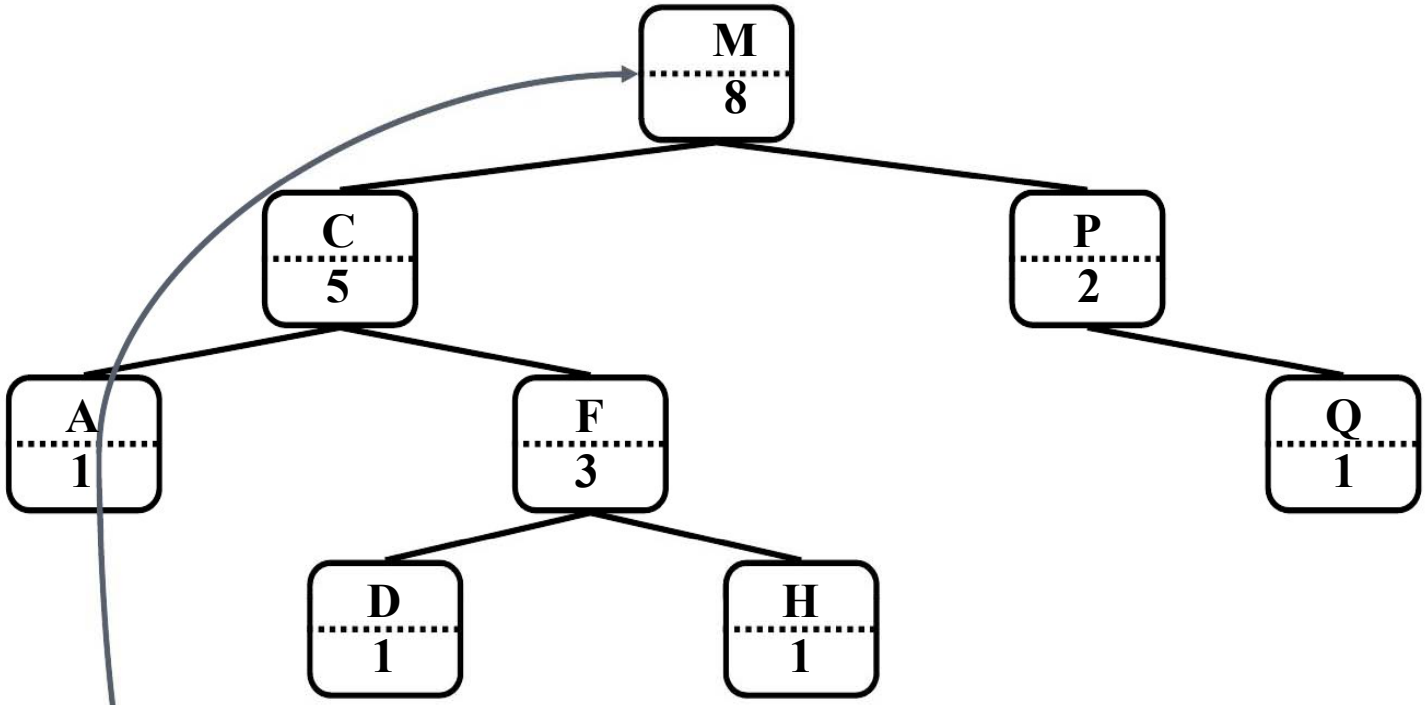


What is the rank of this element?

Determining The Rank Of An Element

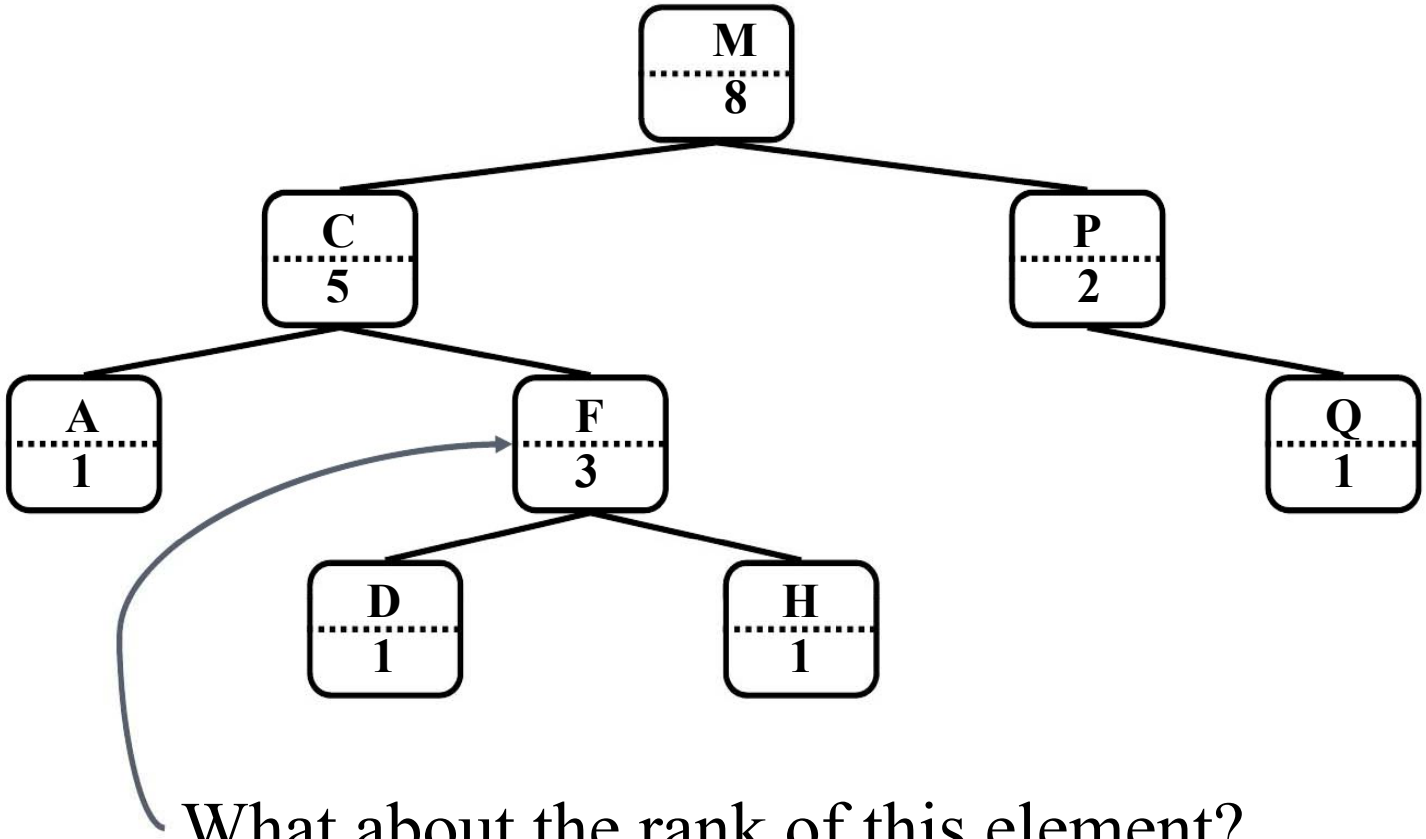


Determining The Rank Of An Element



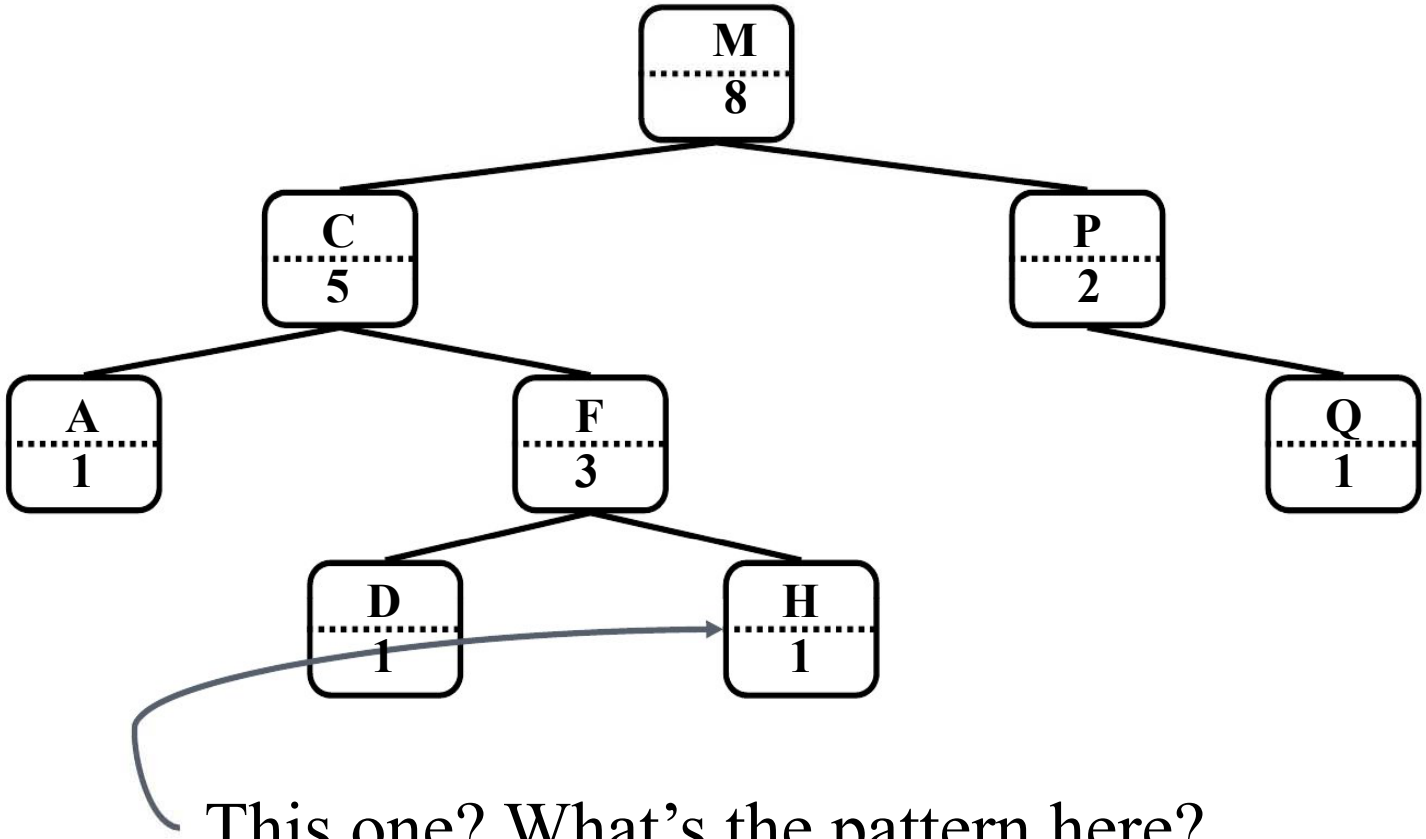
Of the root? What's the pattern here?

Determining The Rank Of An Element



What about the rank of this element?

Determining The Rank Of An 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-Trees : Maintaining Sizes

So we've shown that with subtree sizes, order statistic operations can be done in O(lg n) time

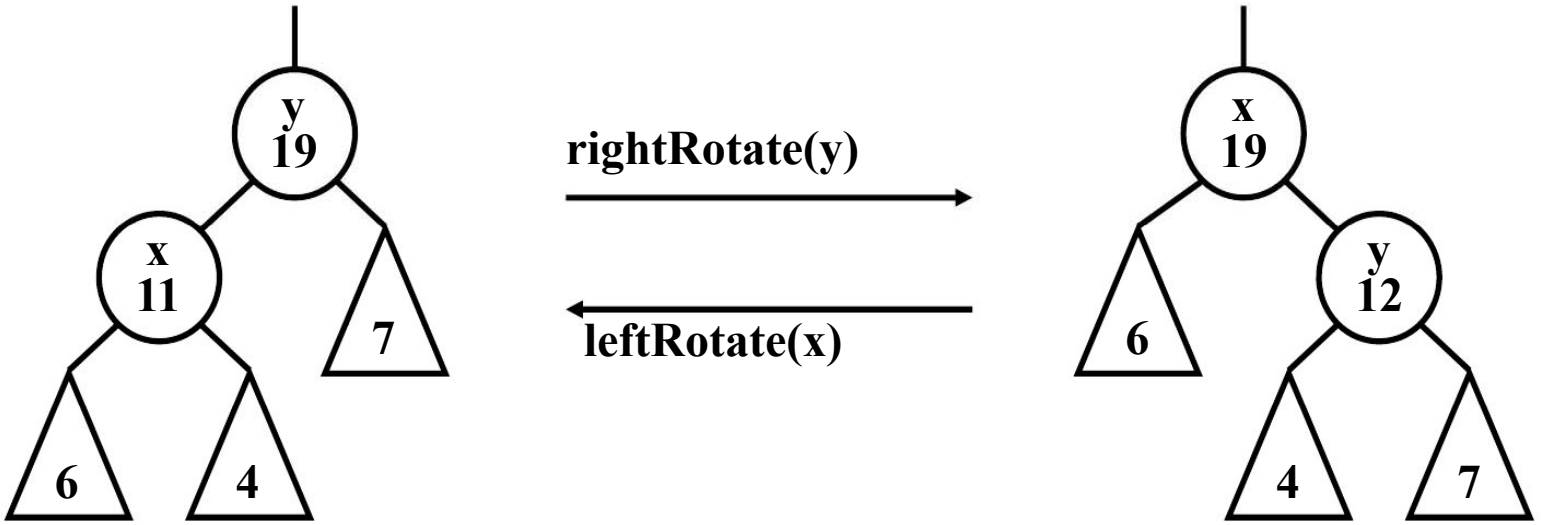
Next step: maintain sizes during Insert() and Delete() operations

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?

MAINTAINING SIZE THROUGH ROTATION



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

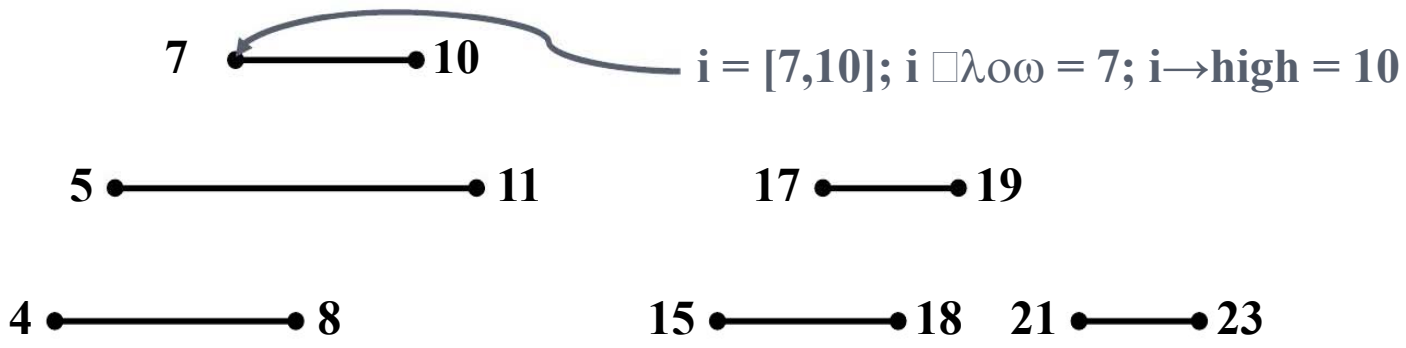
- 12 *size[y]* ← *size[x]*
- 13 *size[x]* ← *size[left[x]]* + *size[right[x]]* + 1

AUGMENTING DATA STRUCTURES: 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()

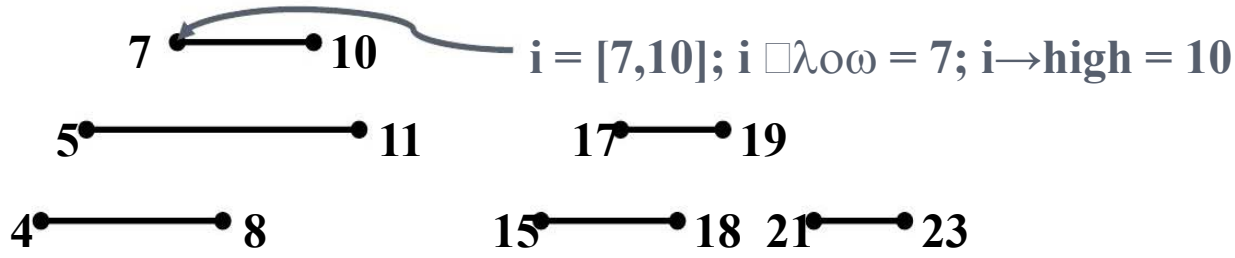
INTERVAL TREES

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



INTERVAL T REES

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



- We can represent an interval[t1,t2] as an object i, with fieldslow[i] = t1 (the **low endpoint**) and high[i] = t2 (the **high endpoint**).

We say that intervals i and i' **overlap** if $i \cap i' \neq \emptyset$,that is, if low[i] ≤ high[i'] and low[i'] ≤ high[i]. Any two intervals i and i' satisfy the **interval trichotomy**; 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])*

INTERVAL TREES

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

INTERVAL TREES

- Following the methodology:
  - Pick underlying data structure
  - Decide what additional information to store
  - Figure out how to maintain the information
  - Develop the desired new operations

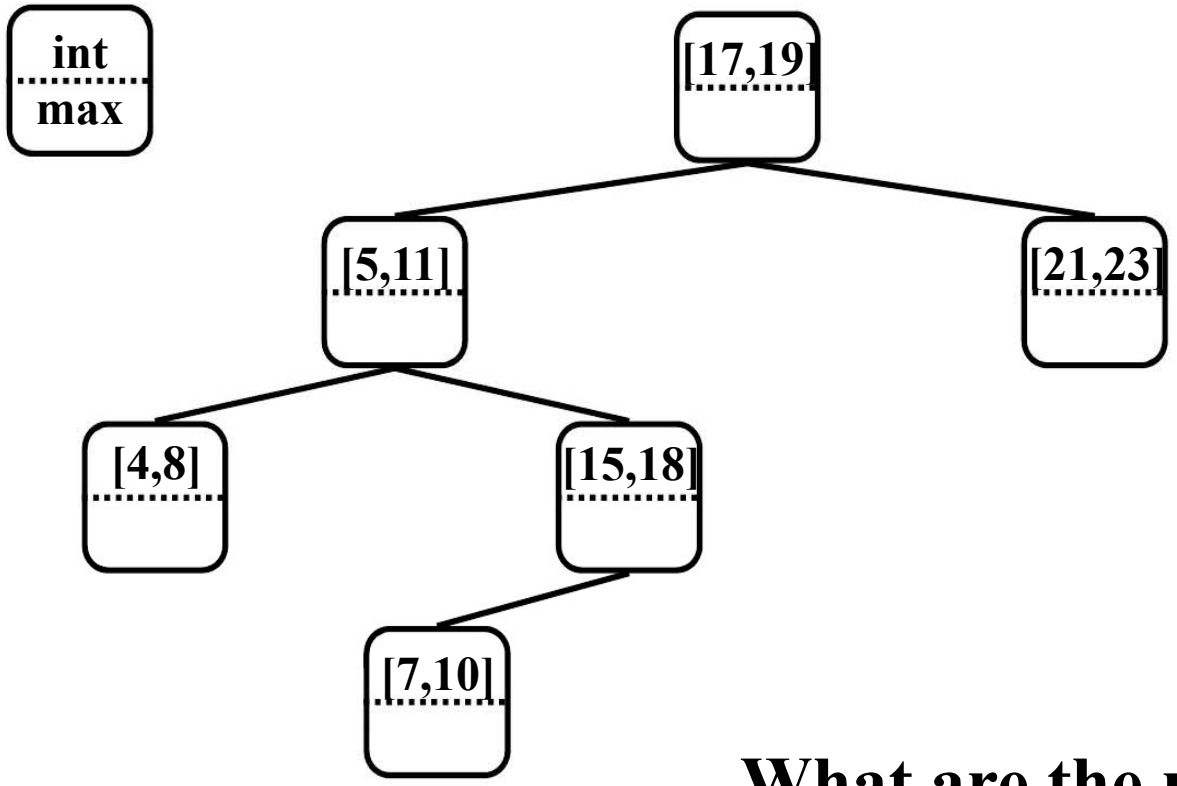
INTERVAL TREES

- Following the methodology:
  - *Pick underlying data structure*
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - Decide what additional information to store
  - Figure out how to maintain the information
  - Develop the desired new operations

INTERVAL TREES

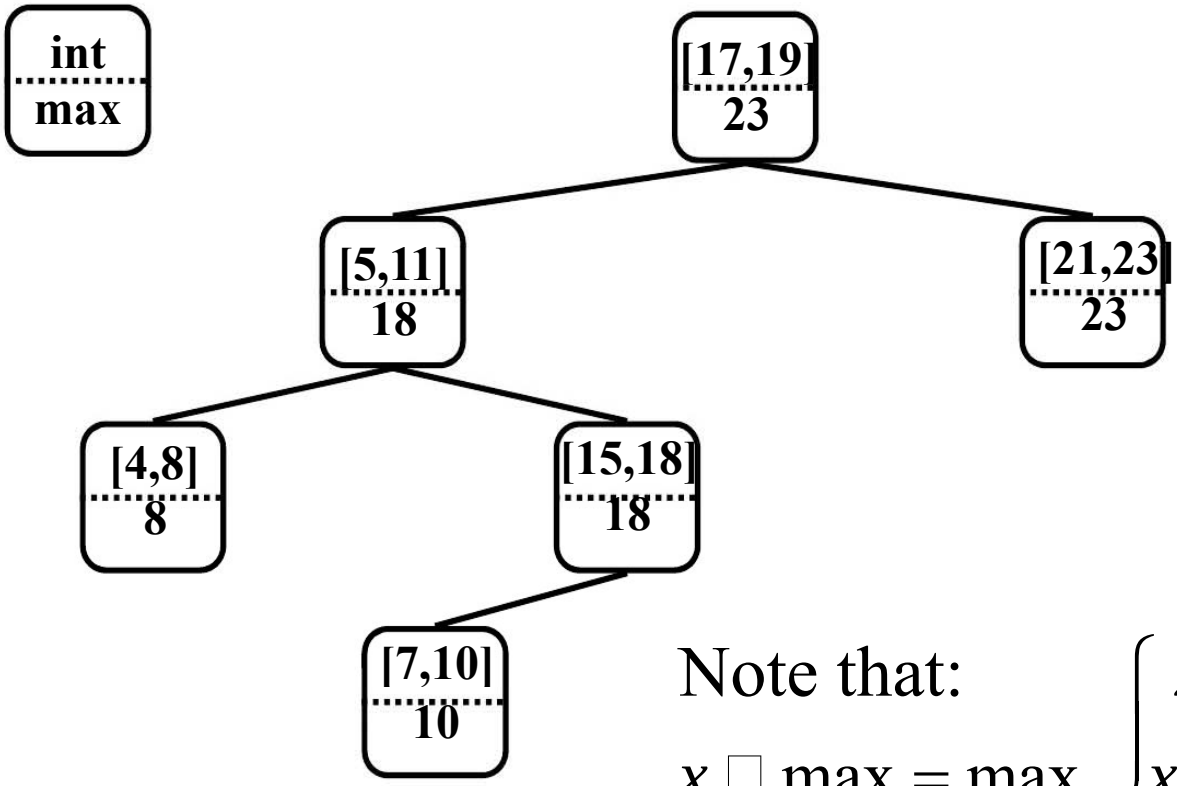
- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - *Decide what additional information to store*
    - We will store *max,* the maximum endpoint in the subtree rooted at i
  - Figure out how to maintain the information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

INTERVAL TREES



What are the max fields?

INTERVAL TREES



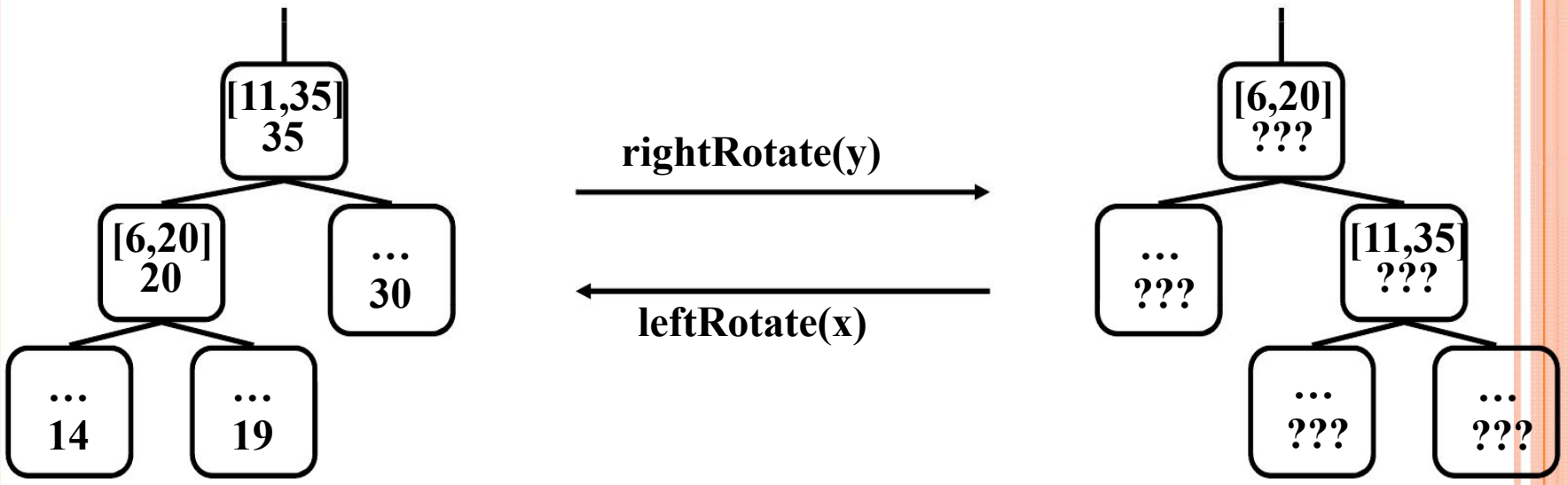
Note that:

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

INTERVAL TREES

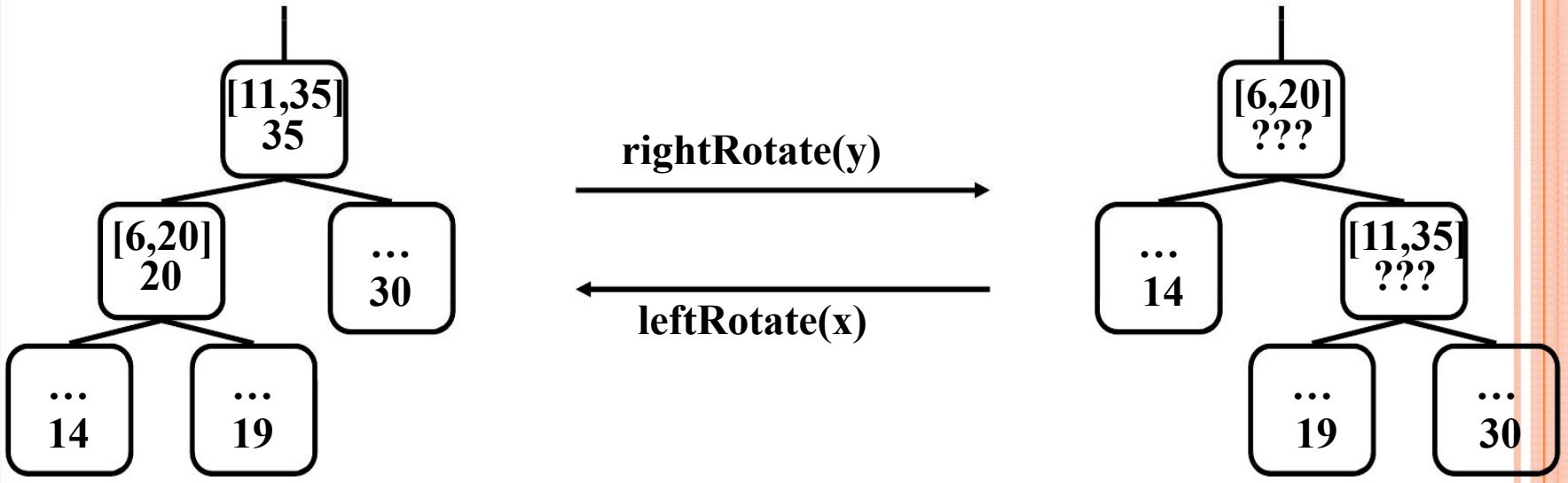
- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - Decide what additional information to store
    - Store the maximum endpoint in the subtree rooted at i
  - *Figure out how to maintain the information*
    - *How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

INTERVAL TREES



What are the new max values for the subtrees?

INTERVAL TREES

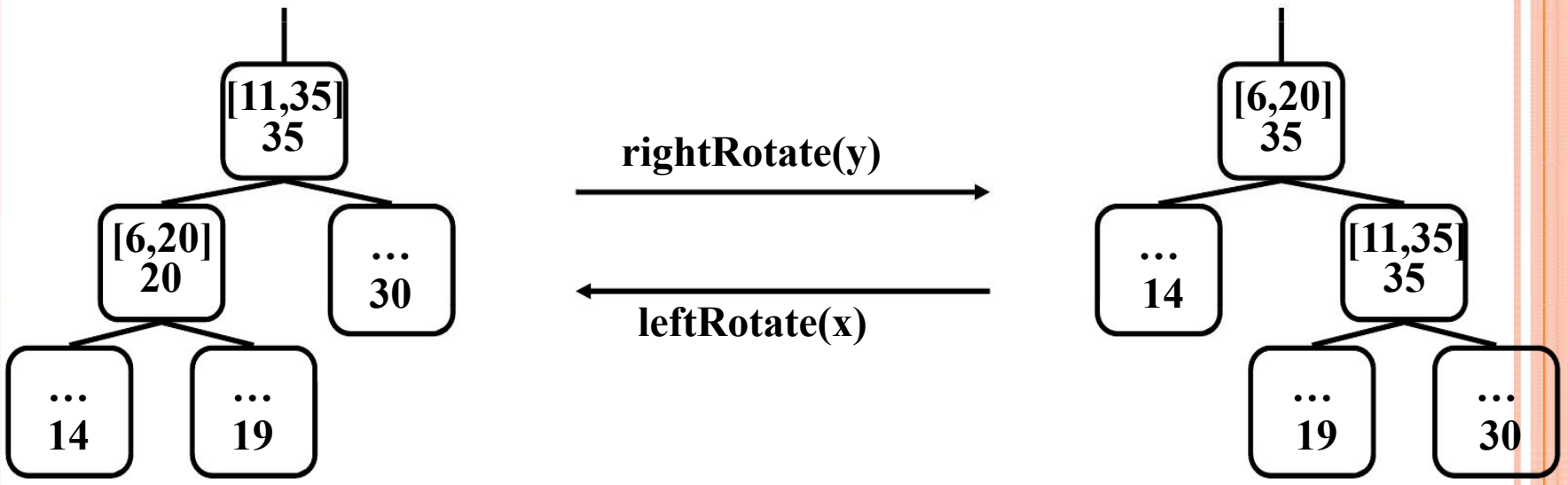


What are the new max values for the subtrees?

A: Unchanged

What are the new max values for x and y?

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

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*

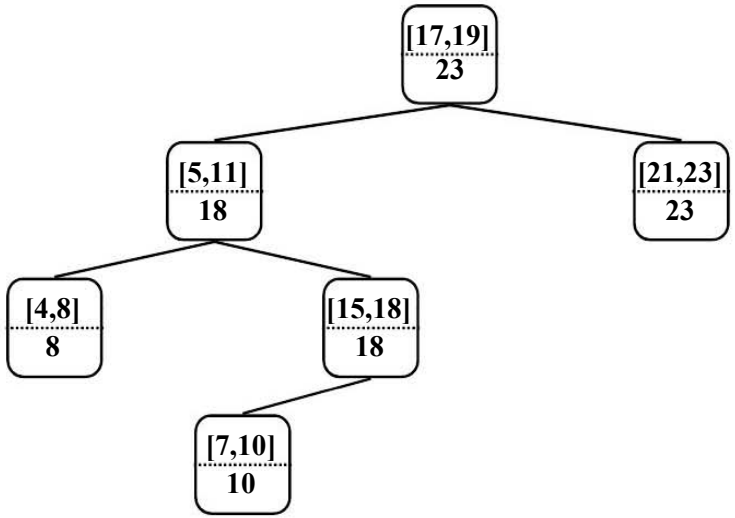
SEARCHING INTERVAL TREES

```
IntervalSearch(T, i)
{
        x = T->root;
        while (x != NULL && !overlap(i, x->interval))
                if (x->left != NULL && x->left->max ε  i->low)
                        x = x->left;
                else
                        x = x->right;
        return x
}
```

- *What will be the running time?*

IntervalSearch() Example

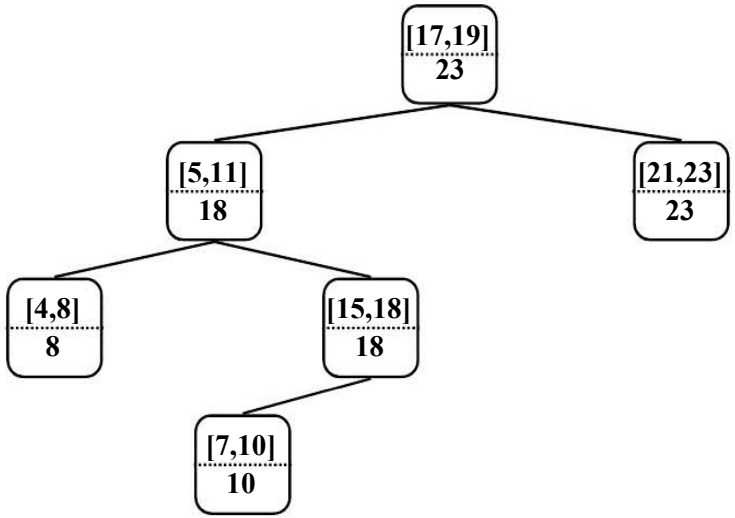
Example: search for interval overlapping [14,16]




```
IntervalSearch(T, i)
{
        x = T->root;
        while (x != NULL && !overlap(i, x->interval))
                if (x->left != NULL && x->left->max ε  i->low)
                        x = x->left;
                else
                        x = x->right;
        return x
}
```

IntervalSearch() Example

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 ε  i->low)
                        x = x->left;
                else
                        x = x->right;
        return x
}
```

Correctness of IntervalSearch()

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

Correctness of IntervalSearch()

- Case 1: 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→left = NULL, or x → left → max < x → low (Why?)
    - Thus, no overlap in left subtree!

```
while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max ε  i->low)
            x = x->left;
        else
            x = x->right;
    return x;
```

CORRECTNESS OF INTERVALSEARCH()

- Case 2: 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 →low ≤ x →left →max by branch condition
    - x →left →max = y →high for some y in left subtree
    - Since i and y dont overlap and i →low ≤ y →high,i →high < y →low
    - Since tree is sorted by low's, i →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 ε  i->low)
                x = x->left;
        else
                x = x->right;
return x;
```