# Introduction to Algorithms

Augmenting Data Structures

# Augmenting Data Structures

- This course is supposed to be about design and analysis of algorithms
- So far, we've only looked at one design technique *(What is it?)*

# Augmenting Data Structures

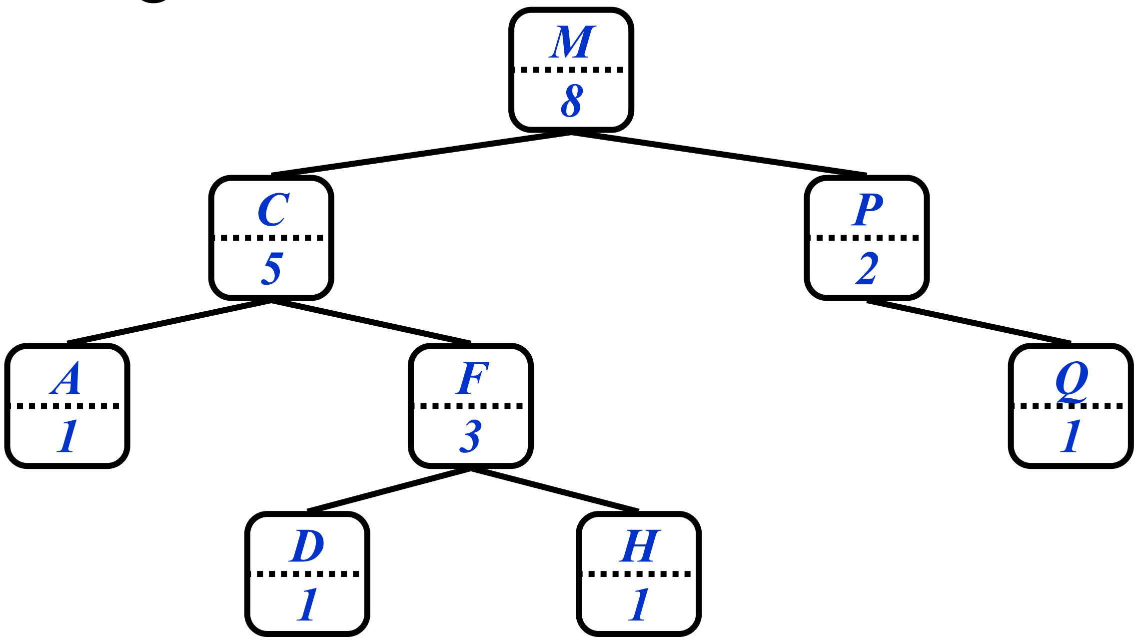
- This course is supposed to be about design and analysis of algorithms
- So far, we've only looked at one design technique: *divide and conquer*
- Next up: augmenting data structures

# Dynamic Order Statistics

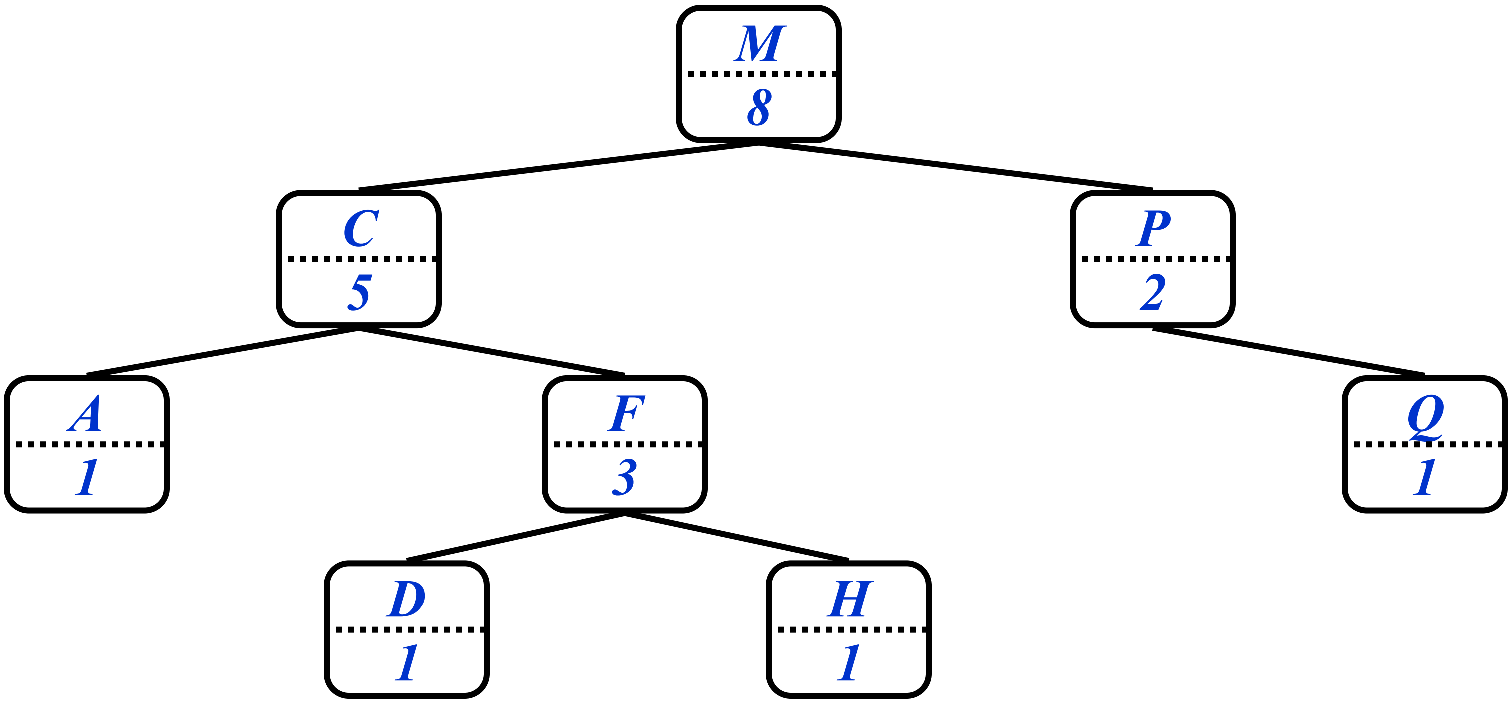
- We've seen algorithms for finding the $i$-th element of an unordered set in $O(n)$ time
- Next, a structure to support finding the $i$-th element of a dynamic set in $O(\lg n)$ time
  - *What operations do dynamic sets usually support?*
  - *What structure works well for these?*
  - *How could we use this structure for order statistics?*
  - *How might we augment it to support efficient extraction of order statistics?*

# 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:



# Selection On OS Trees




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

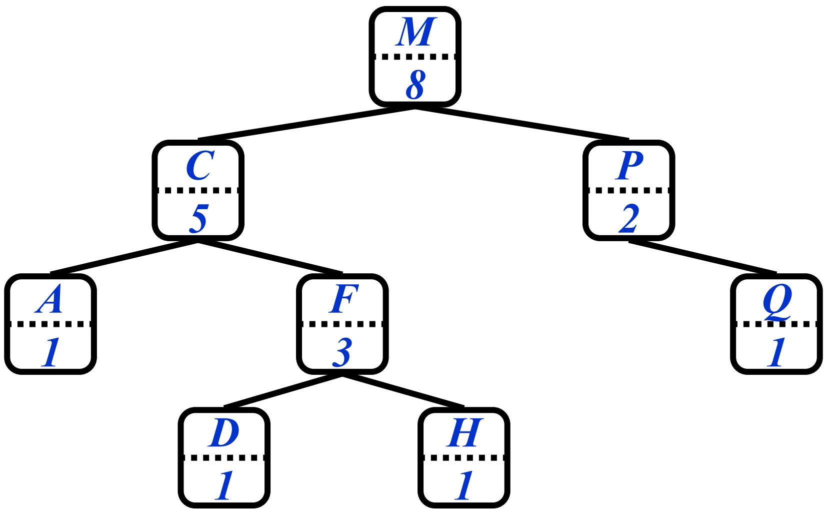
# OS-Select

```
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):

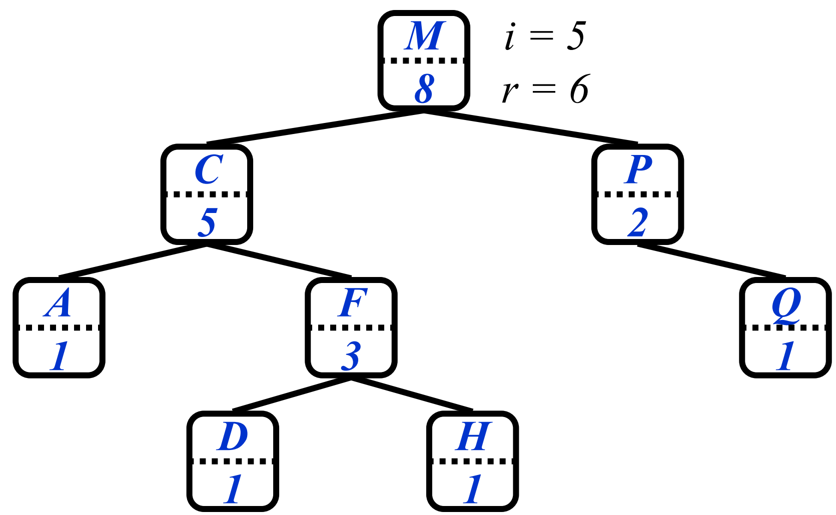
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



# OS-Select Example

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

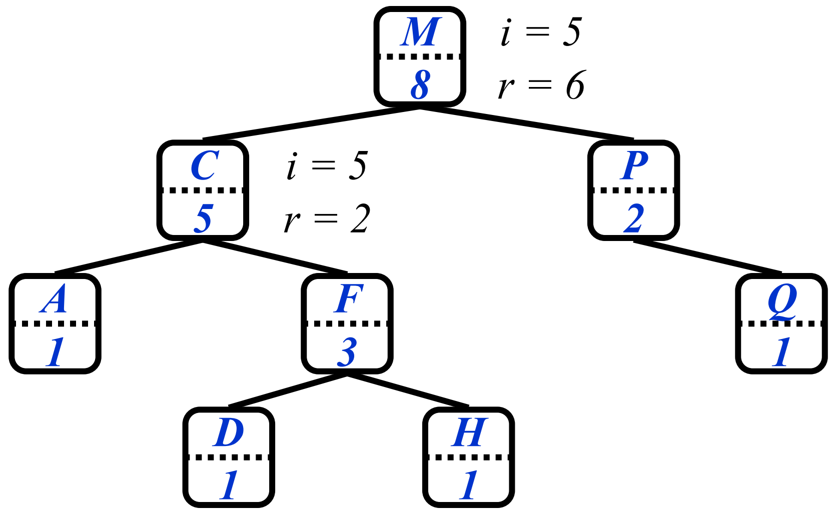
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



# OS-Select Example

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

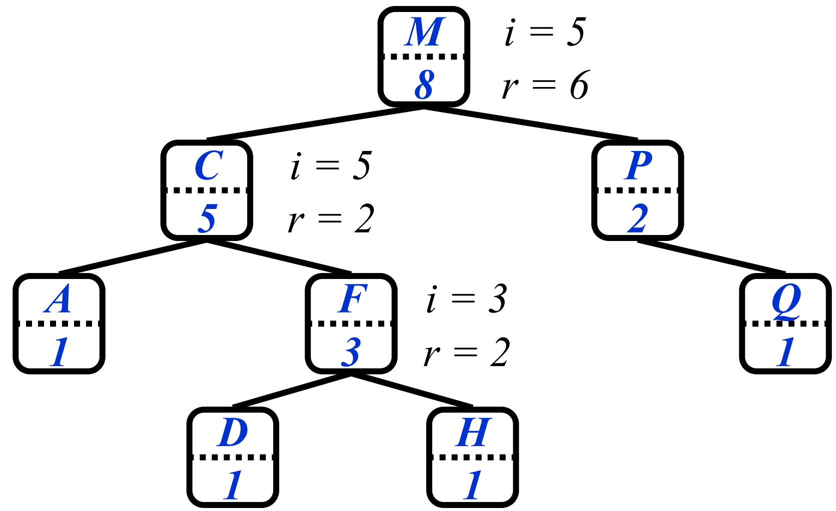
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



# OS-Select Example

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

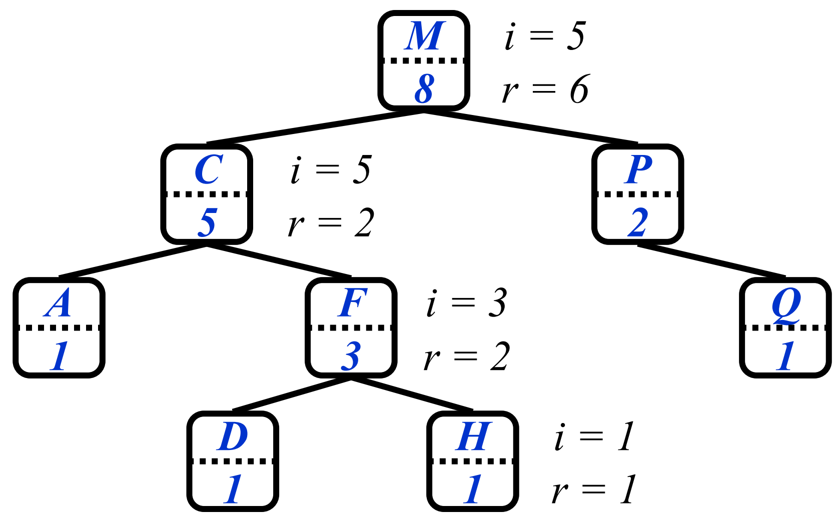
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



# 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

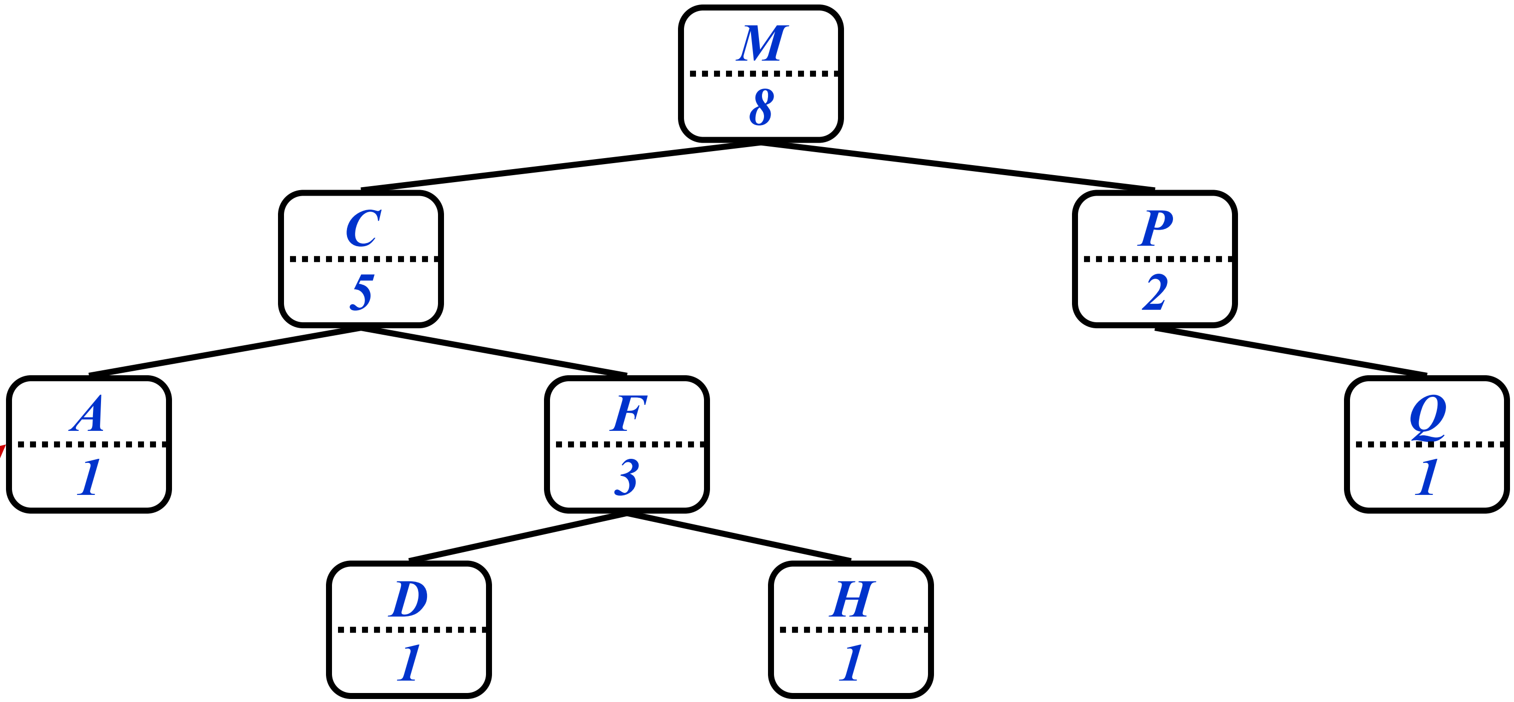
```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```

# 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);
}
```

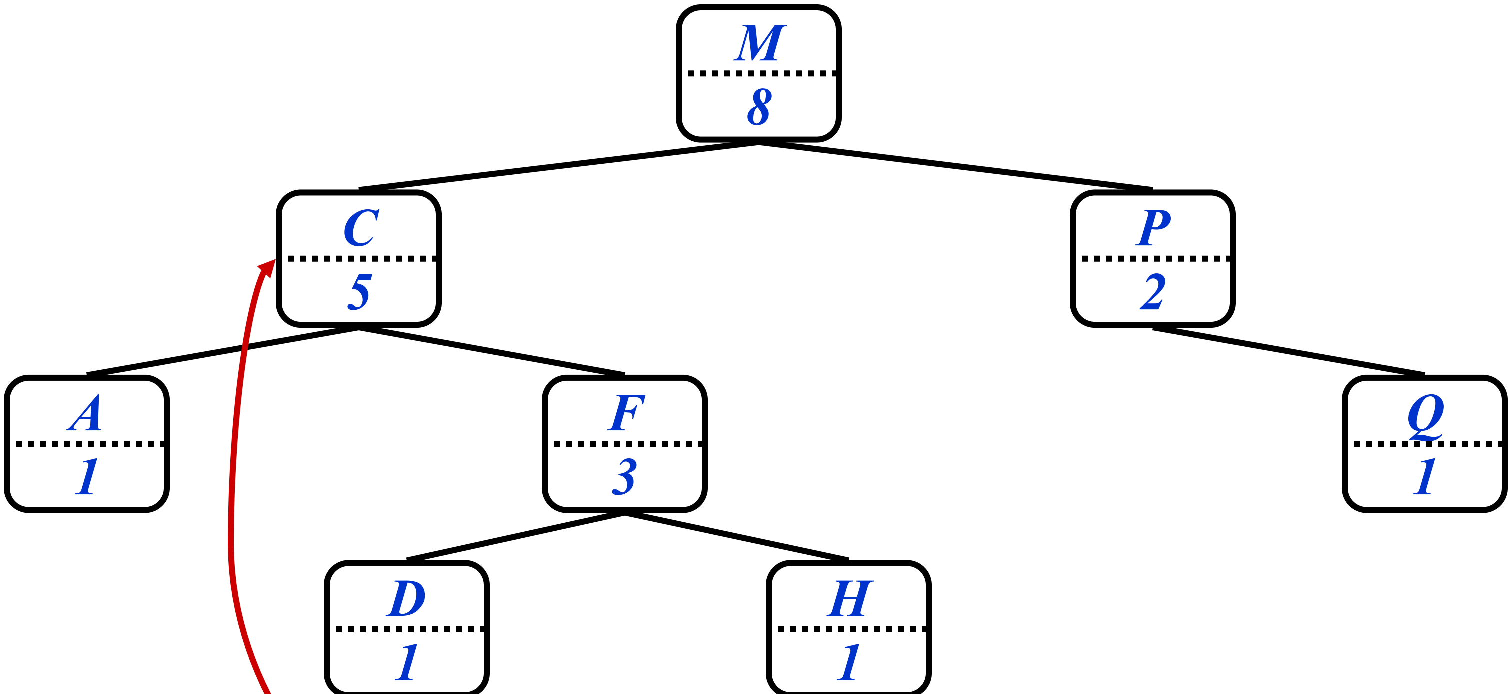
- *What will be the running time?*

# Determining The Rank Of An Element



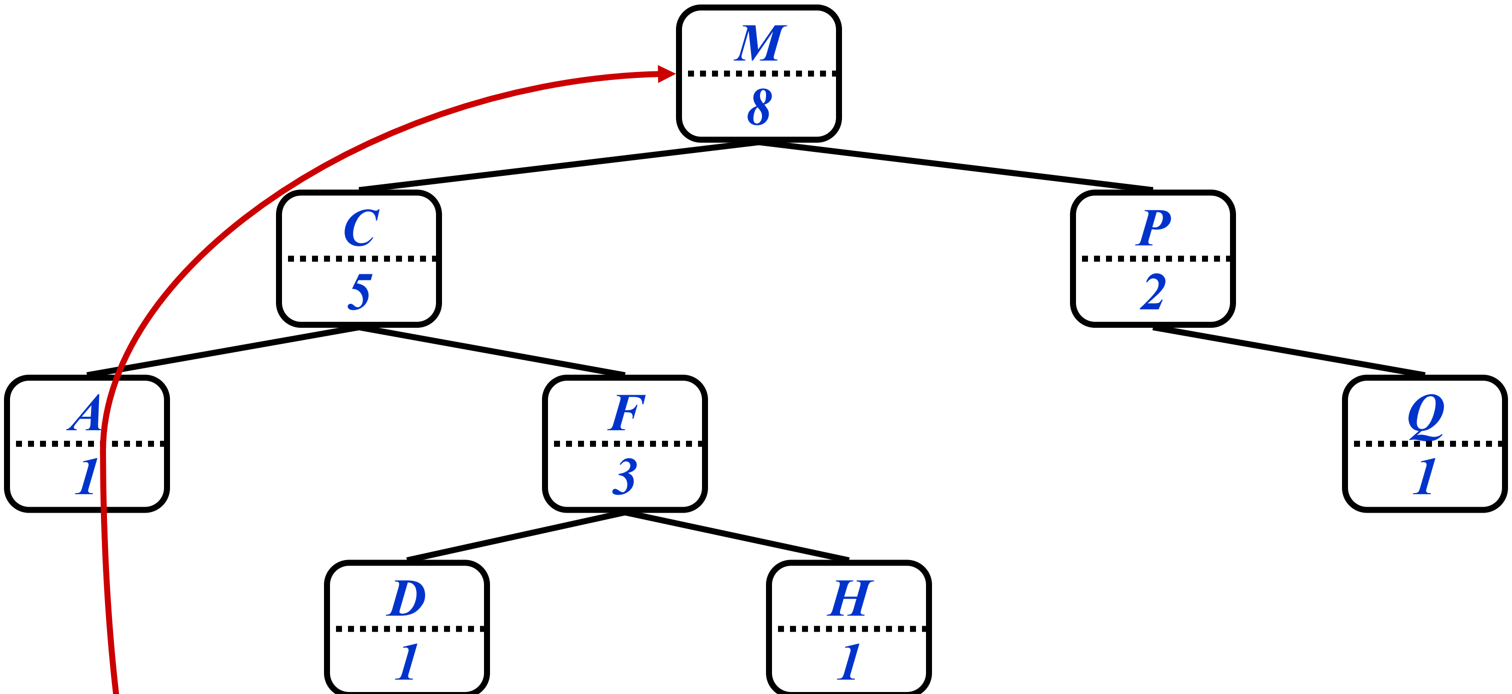

*What is the rank of this element?*

# Determining The Rank Of An Element



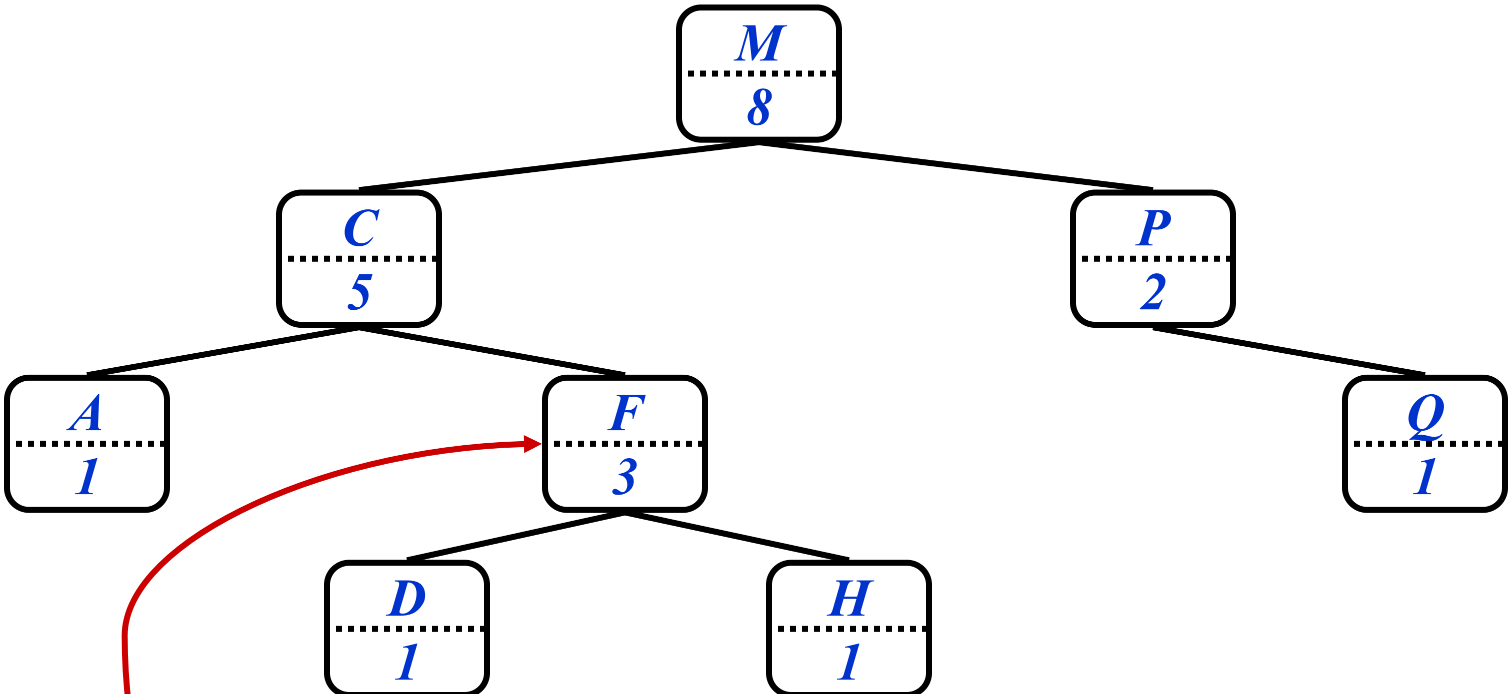

*Of this one? Why?*

# Determining The Rank Of An Element



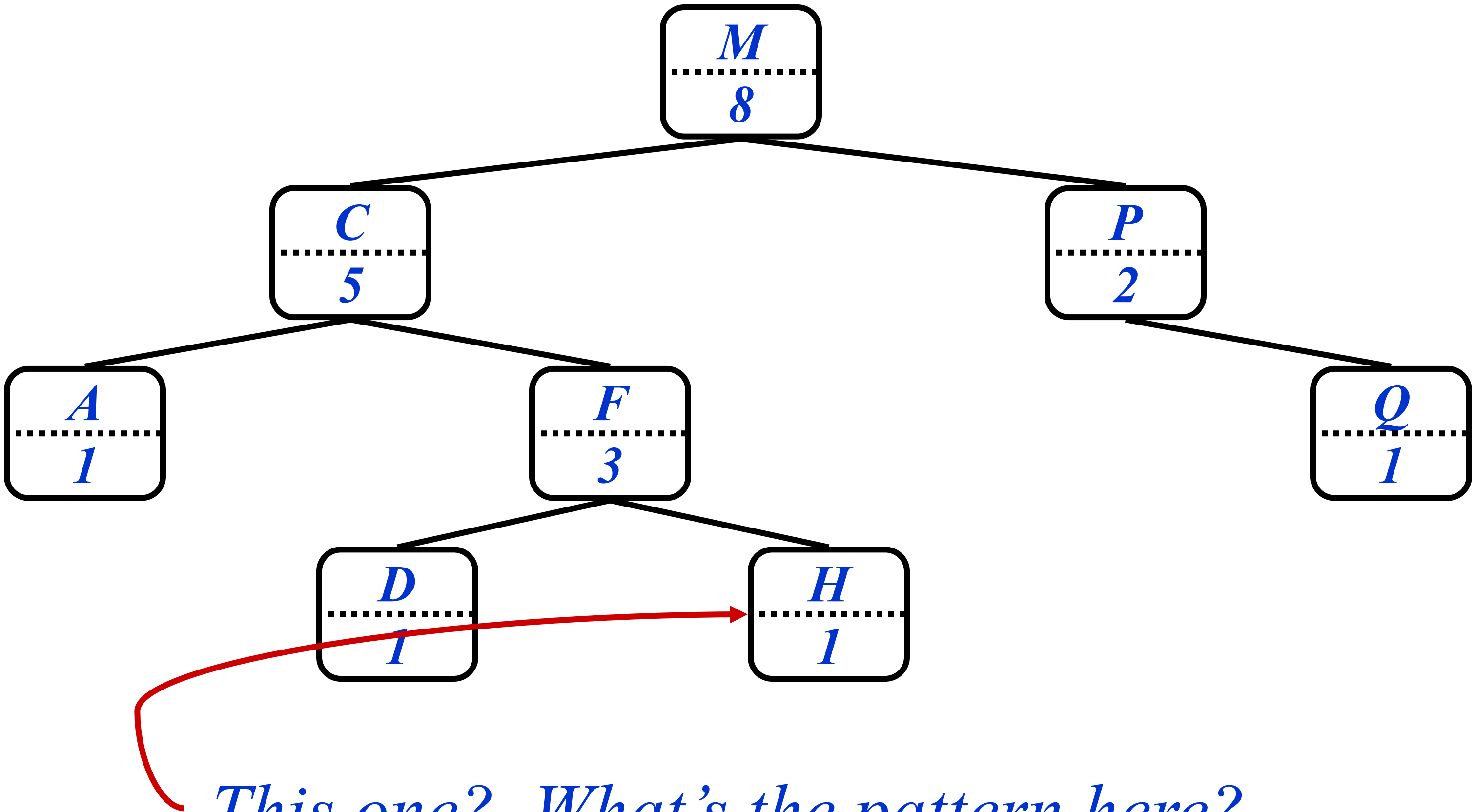

*Of the root? What's the pattern here?*

# 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?*

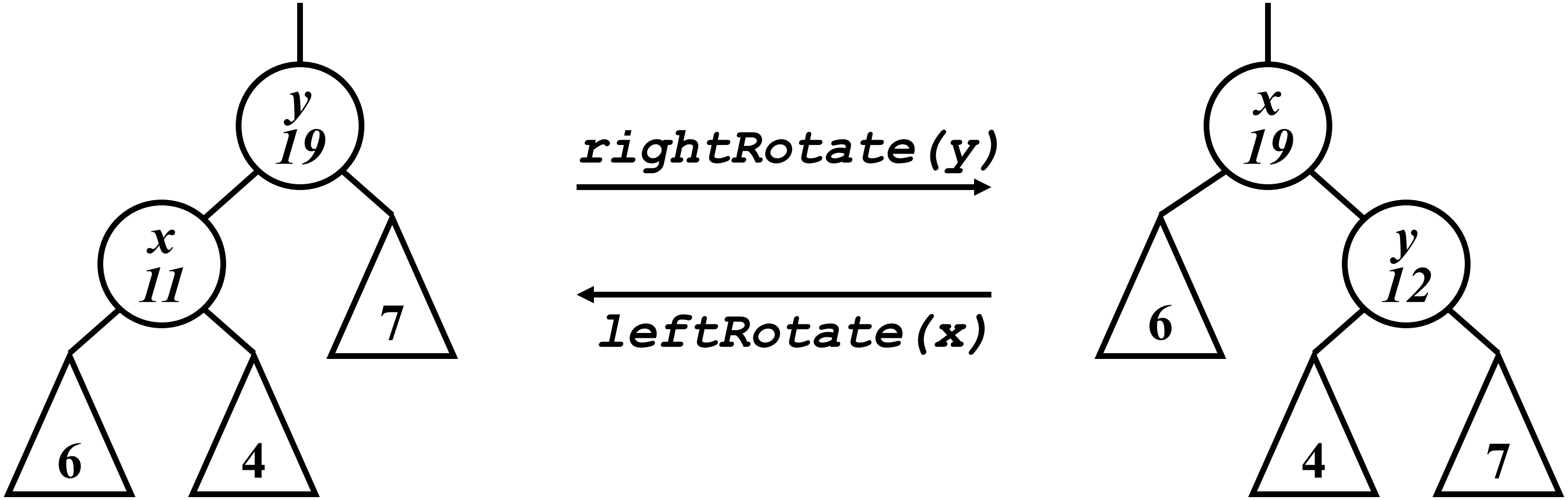
# OS-Trees: Maintaining Sizes

- 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

# 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



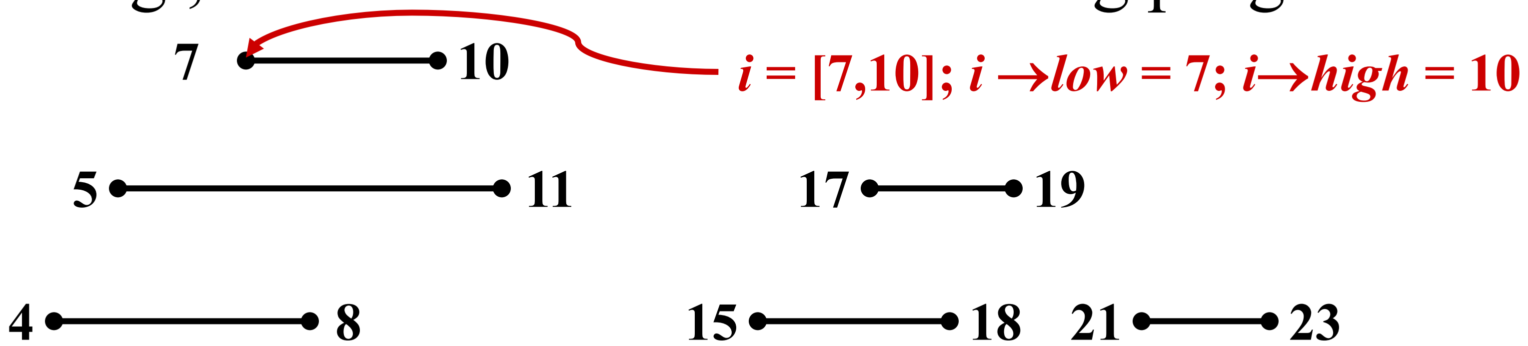

- Salient point: rotation invalidates only *x* and *y*
- Can recalculate their sizes in constant time
  - *Why?*

# 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()

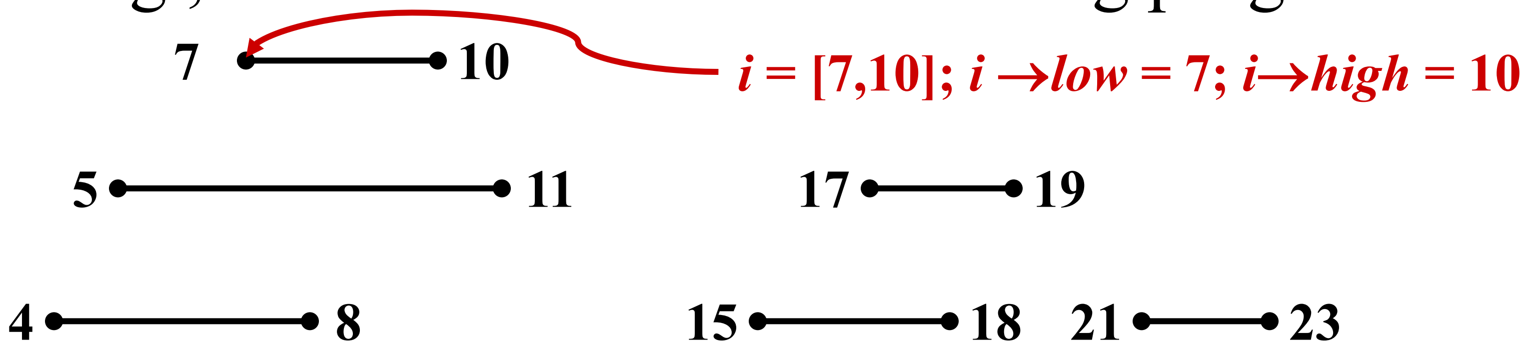
# Interval Trees

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



# Interval Trees

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




  - Query: find an interval in the set that overlaps a given query interval
    - [14,16] → [15,18]
    - [16,19] → [15,18] or [17,19]
    - [12,14] → NULL

# 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

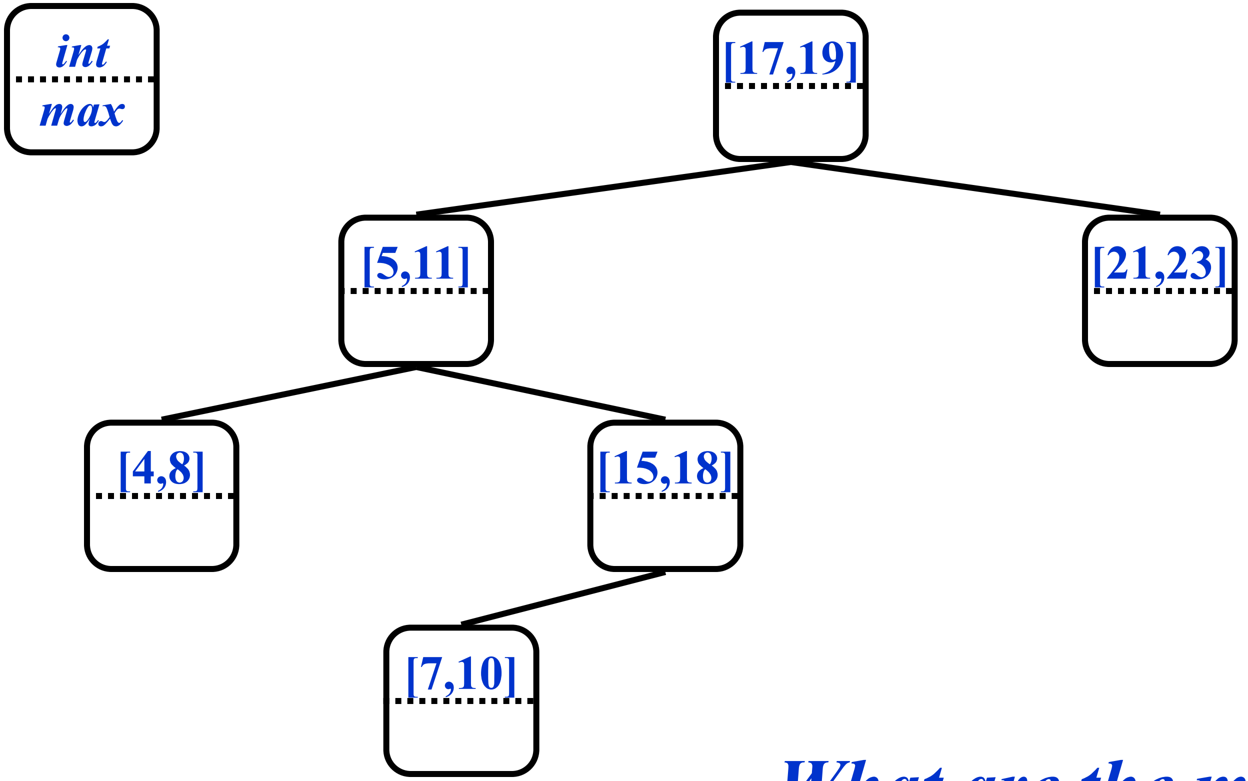
# Interval Trees

- 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

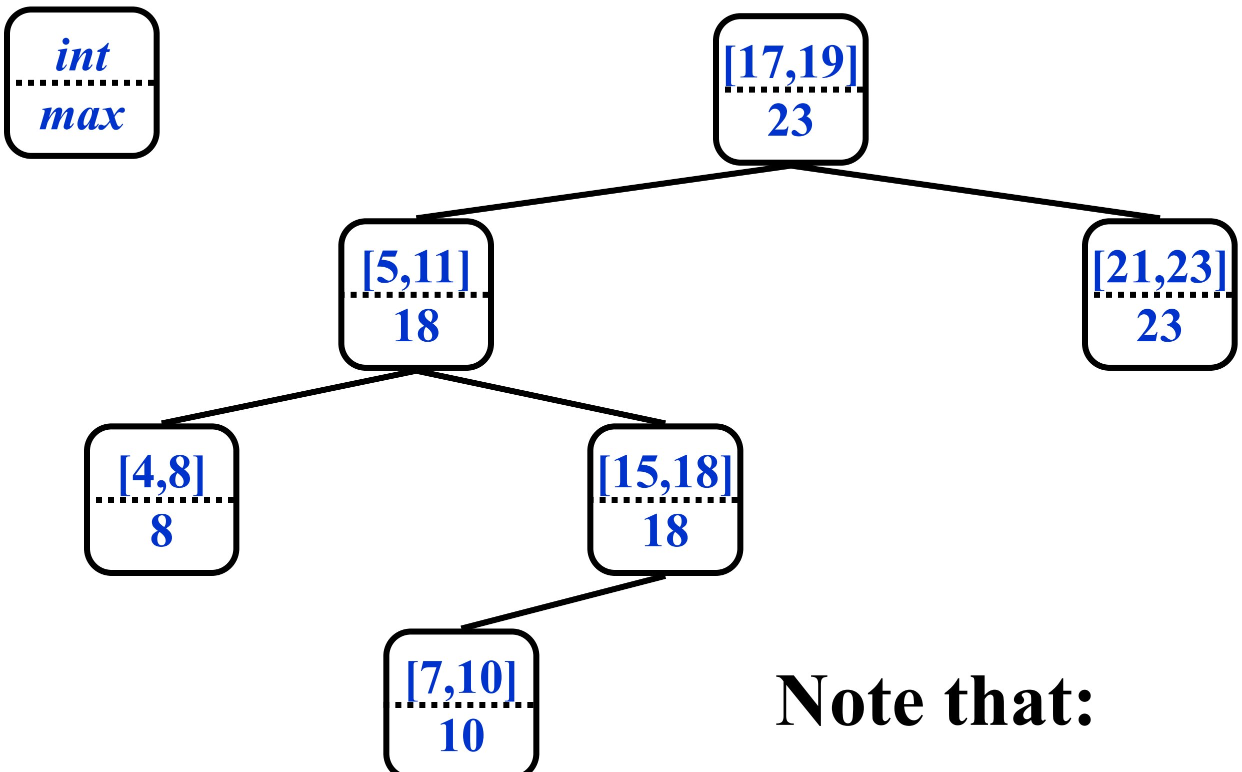
- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - *Decide what additional information to store*
    - We will store *max*, the maximum endpoint in the subtree rooted at *i*
  - Figure out how to maintain the information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

# Interval Trees




***What are the max fields?***

# Interval Trees



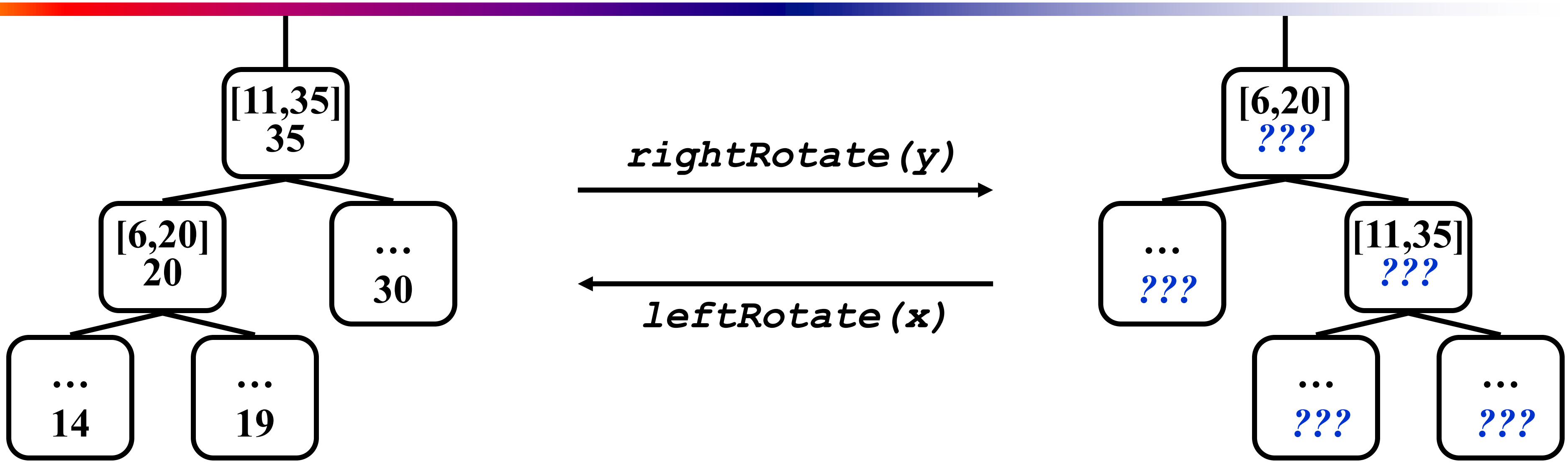

**Note that:**

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

# Interval Trees

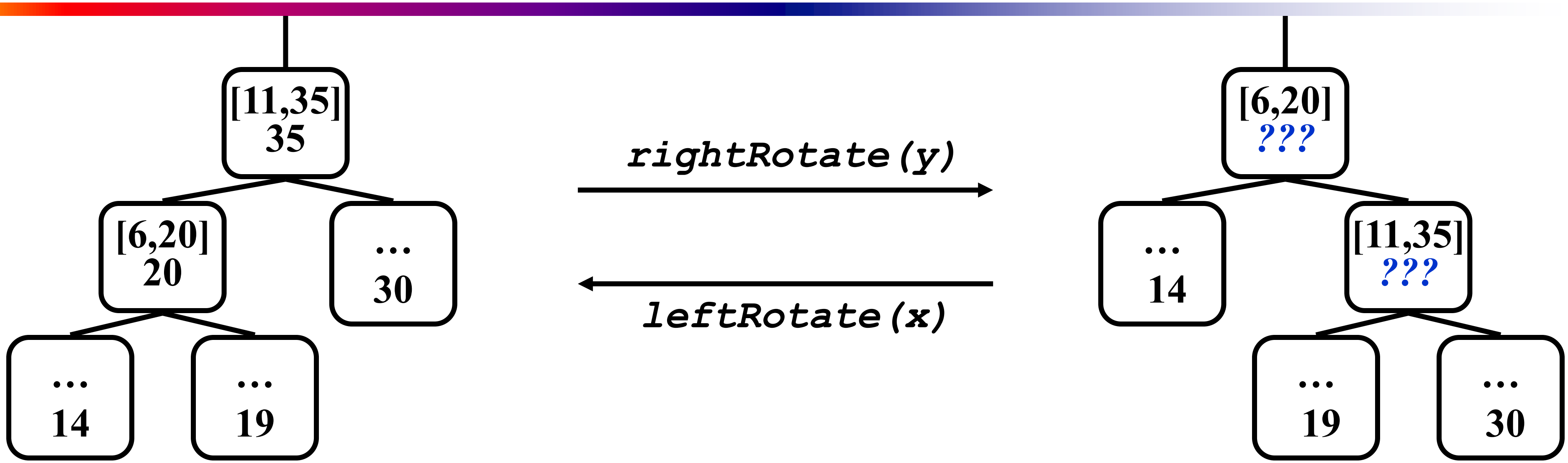
- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - Decide what additional information to store
    - Store the maximum endpoint in the subtree rooted at $i$
  - *Figure out how to maintain the information*
    -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

# Interval Trees



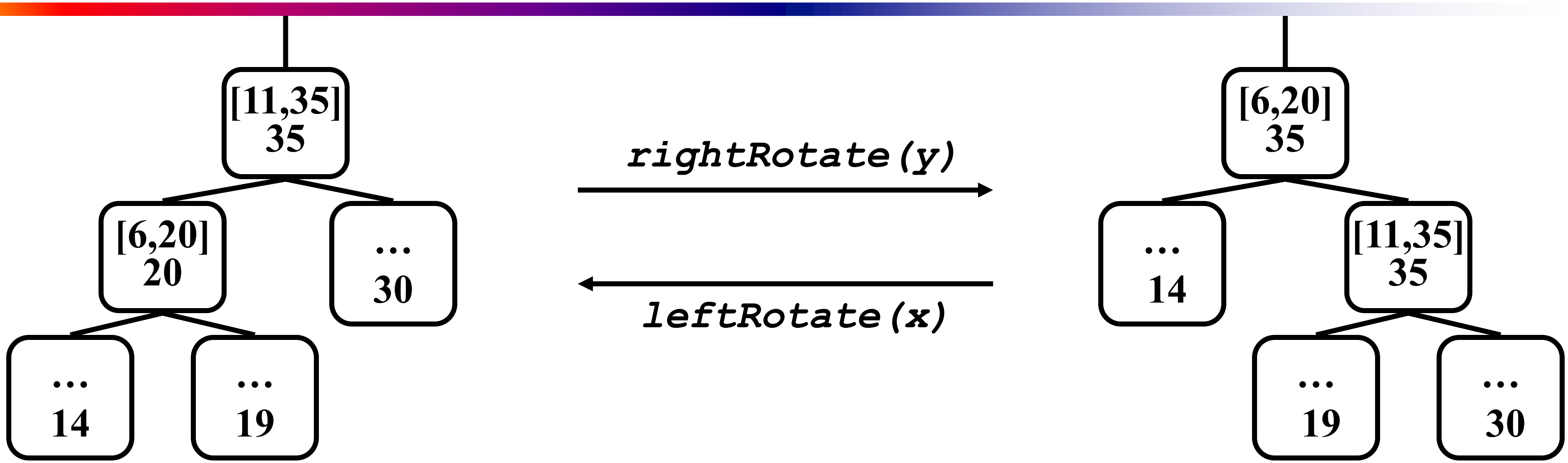

- *What are the new max values for the subtrees?*

# 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*

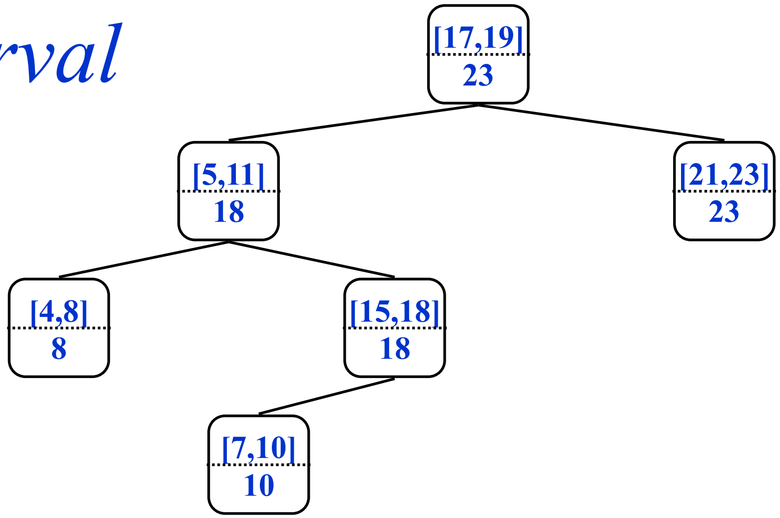
# Searching Interval Trees

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

- *What will be the running time?*

# IntervalSearch() Example

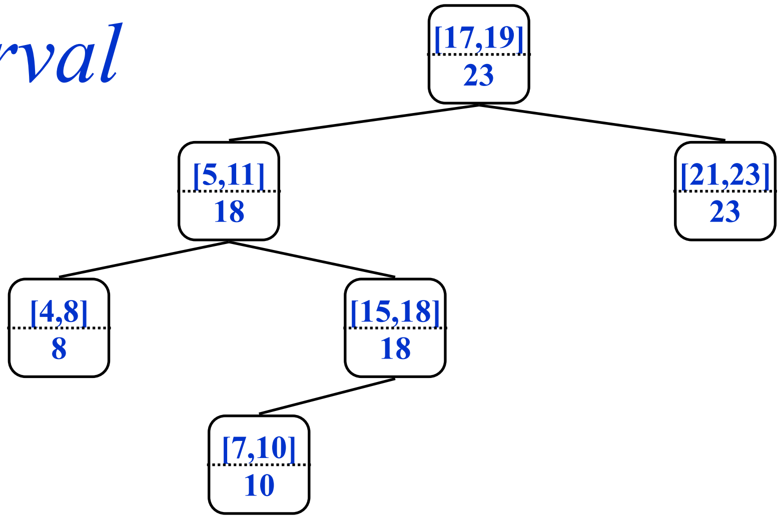
- *Example: search for interval overlapping [14,16]*




```
IntervalSearch(T, i)
{
    x = T->root;
    while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max ≥ 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 ≥ 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 ∃ 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 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:
    - $x \rightarrow left$ = 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 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 don't 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 ≥ i->low)
            x = x->left;
        else
            x = x->right;
    return x;
```