Advanced Data Structures

Medians and Order Statistics

Order Statistics

- The ith *order statistic* in a set of n elements is the ith smallest element
- The *minimum* is thus the 1st order statistic
- The *maximum* is the nth order statistic
- The *median* is the $n/2$ order statistic
  - If n is even, there are 2 medians
- *How can we calculate order statistics?*
- *What is the running time?*

Order Statistics

- *How many comparisons are needed to find the minimum element in a set? The maximum?*
- *Can we find the minimum and maximum with less than twice the cost?*
- Yes:
  - Walk through elements by pairs
    - Compare each element in pair to the other
    - Compare the largest to maximum, smallest to minimum
  - Total cost: 3 comparisons per 2 elements = O(3n /2)

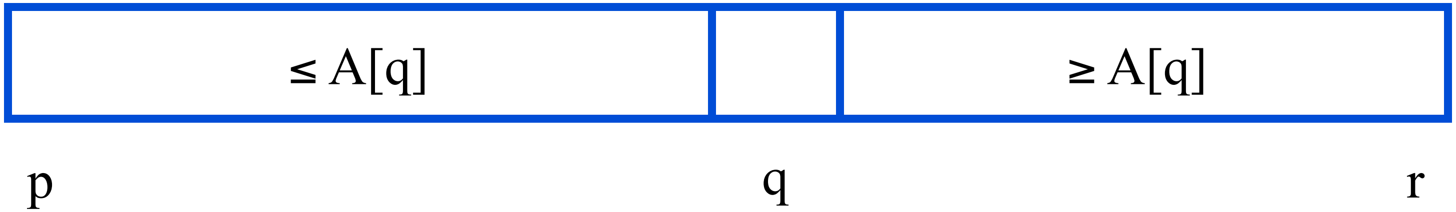
Finding Order Statistics: The Selection Problem

- A more interesting problem is *selection*: finding the *i*th smallest element of a set
- We will show:
  - A practical randomized algorithm with O(n) expected running time
  - A cool algorithm of theoretical interest only with O(n) worst-case running time

Randomized Selection

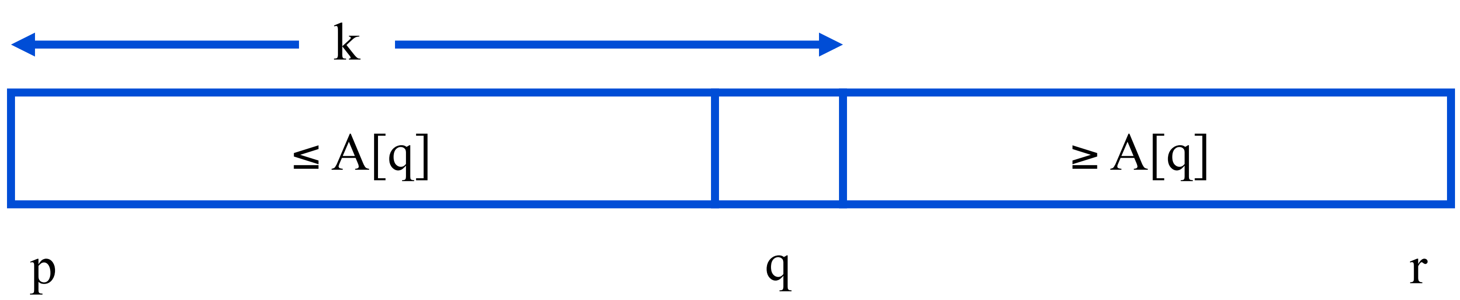
- Key idea: use partition() from quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)

q = RandomizedPartition(A, p, r)



Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A, q+1, r, i-k);
```



Randomized Selection

- Analyzing **`RandomizedSelect()`**
  - Worst case: partition always 0:n-1

    $T(n) = T(n-1) + O(n)$ ***= ???***

    $= O(n^2)$ (arithmetic series)

    - No better than sorting!
  - "Best" case: suppose a 9:1 partition

    $T(n) = T(9n/10) + O(n)$ ***= ???***

    $= O(n)$ (Master Theorem, case 3)

    - Better than sorting!
    - *What if this had been a 99:1 split?*

Randomized Selection

- Average case
  - For upper bound, assume *i*th element always falls in larger side of partition:

$$T(n) \le \frac{1}{n}\sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

What happened here?

  - Let's show that T(n) = O(n) by substitution

Randomized Selection

- Assume T(n) ≤ cn for sufficiently large c:

$$T(n) \le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

The recurrence we started with

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} ck + \Theta(n)$$

Substitute T(n) ≤ cn for T(k)

$$= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right) + \Theta(n)$$

"Split" the recurrence

$$= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \Theta(n)$$

Expand arithmetic series

$$= c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$

Multiply it out

Randomized Selection

- Assume T(n) ≤ cn for sufficiently large c:

$$T(n) \le c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$ *The recurrence so far*

$$= cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n)$$ *Multiply it out*

$$= cn - \frac{cn}{4} - \frac{c}{2} + \Theta(n)$$ *Subtract c/2*

$$= cn - \left(\frac{cn}{4} + \frac{c}{2} - \Theta(n)\right)$$ *Rearrange the arithmetic*

$$\le cn \quad \text{(if c is big enough)}$$ *What we set out to prove*

Worst-Case Linear-Time Selection

- Randomized algorithm works well in practice
- What follows is a worst-case linear time algorithm, really of theoretical interest only
- Basic idea:
  - Generate a good partitioning element
  - Call this element x

Worst-Case Linear-Time Selection

- The algorithm in words:
  1. Divide *n* elements into groups of 5
  2. Find median of each group (*How? How long?*)
  3. Use Select() recursively to find median *x* of the $\lfloor n/5 \rfloor$ medians
  4. Partition the *n* elements around *x*. Let $k = \text{rank}(x)$
  5. **if** (i == k) **then** return x

     if (i < k) **then** use Select() recursively to find *i*th smallest element in first partition

     else (i > k) use Select() recursively to find (*i*-*k*)th smallest element in last partition

Worst-Case Linear-Time Selection

- *How many of the 5-element medians are ≤ x?*
  - At least 1/2 of the medians = $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$
- *How many elements are ≤ x?*
  - At least $3 \lfloor n/10 \rfloor$ elements
- For large *n*, $3 \lfloor n/10 \rfloor \geq n/4$ *(How large?)*
- So at least $n/4$ elements ≤ x
- Similarly: at least $n/4$ elements ≥ x

Worst-Case Linear-Time Selection

- Thus after partitioning around x, step 5 will call Select() on at most $3n/4$ elements
- The recurrence is therefore:

$$
\begin{aligned}
T(n) &\le T(\lfloor n/5 \rfloor) + T(3n/4) + \Theta(n) \\
&\le T(n/5) + T(3n/4) + \Theta(n) && \lfloor n/5 \rfloor \le n/5 \\
&\le cn/5 + 3cn/4 + \Theta(n) && \text{Substitute } T(n) = cn \\
&= 19cn/20 + \Theta(n) && \text{Combine fractions} \\
&= cn - (cn/20 - \Theta(n)) && \text{Express in desired form} \\
&\le cn \quad \text{if } c \text{ is big enough} && \text{What we set out to prove}
\end{aligned}
$$

Worst-Case Linear-Time Selection

- Intuitively:
  - Work at each level is a constant fraction (19/20) smaller
    - Geometric progression!
  - Thus the O(n) work at the root dominates

Linear-Time Median Selection

- Given a "black box" O(n) median algorithm, what can we do?
  - *i*th order statistic:
    - Find median *x*
    - Partition input around *x*
    - if ($i \leq (n+1)/2$) recursively find *i*th element of first half
    - else find ($i - (n+1)/2$)th element in second half
    - $T(n) = T(n/2) + O(n) = O(n)$

Linear-Time Median Selection

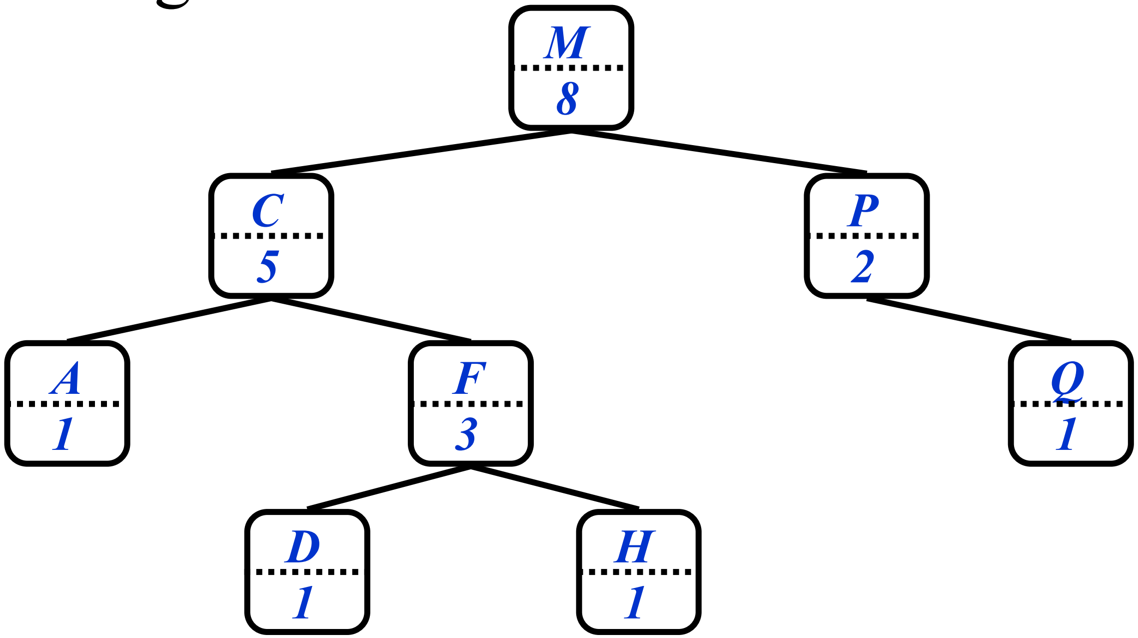
- Worst-case O(n lg n) quicksort
  - Find median *x* and partition around it
  - Recursively quicksort two halves
  - $T(n) = 2T(n/2) + O(n) = O(n \lg n)$

Dynamic Order Statistics

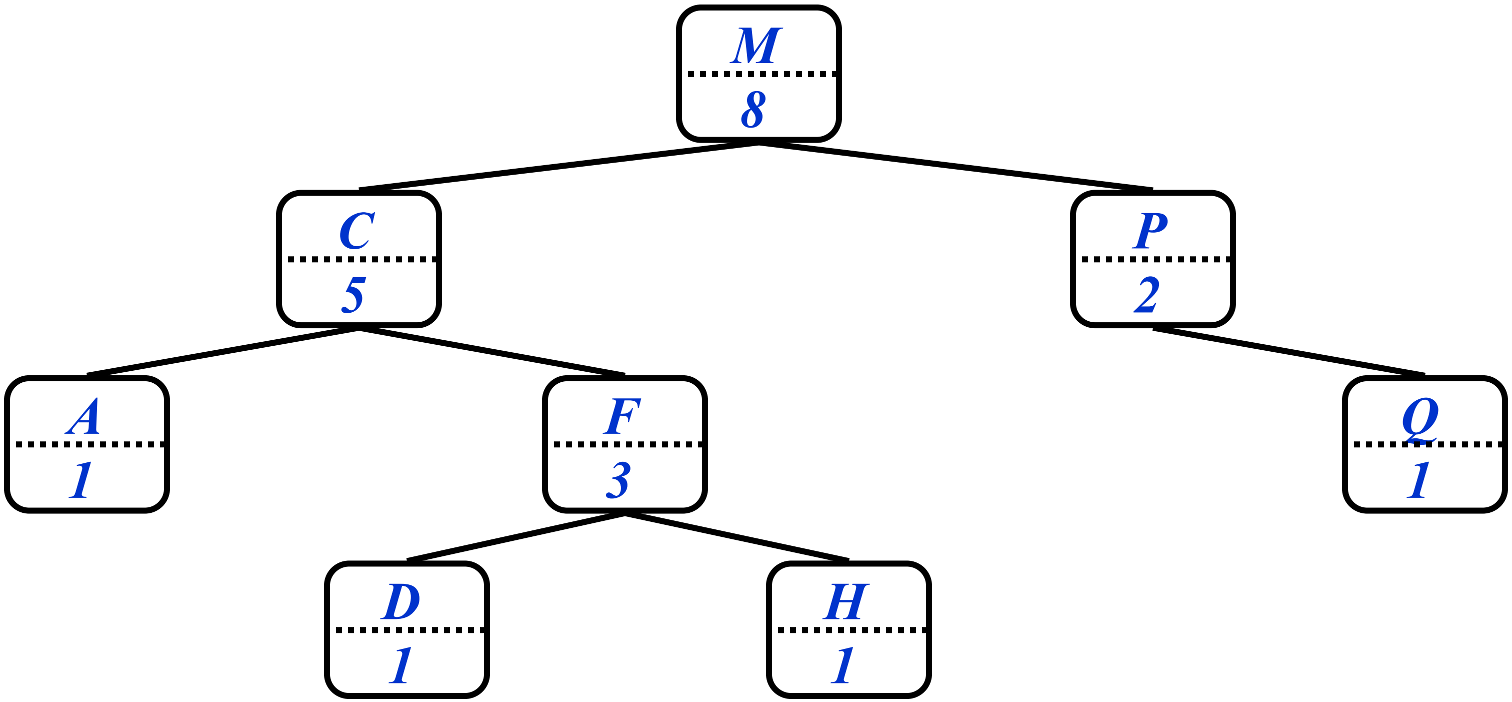
- We've seen algorithms for finding the *i*th element of an unordered set in O(*n*) time
- Next, a structure to support finding the *i*th element of a dynamic set in O(lg *n*) time
  - *What operations do dynamic sets usually support?*
  - *What structure works well for these?*
  - *How could we use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

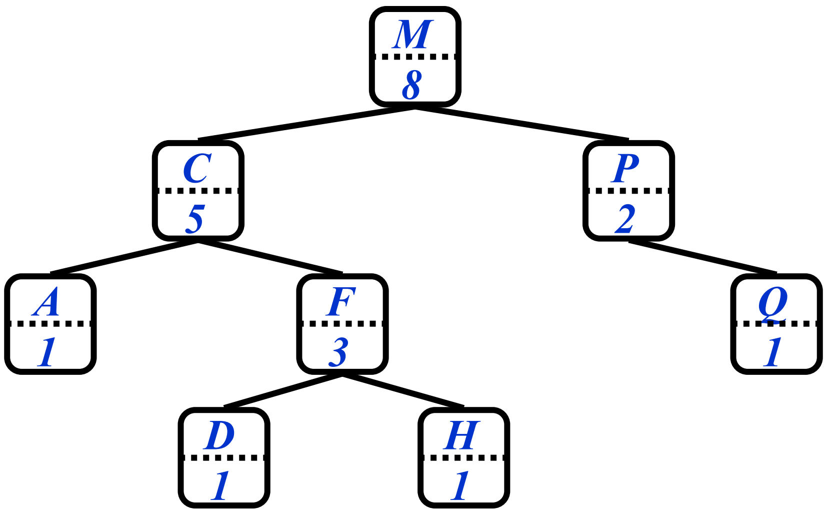
OS-Select

```
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):

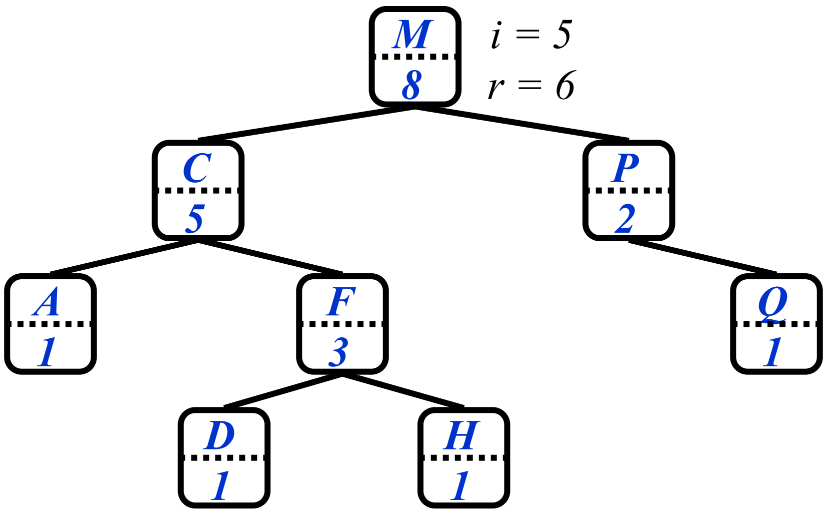
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



OS-Select Example

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

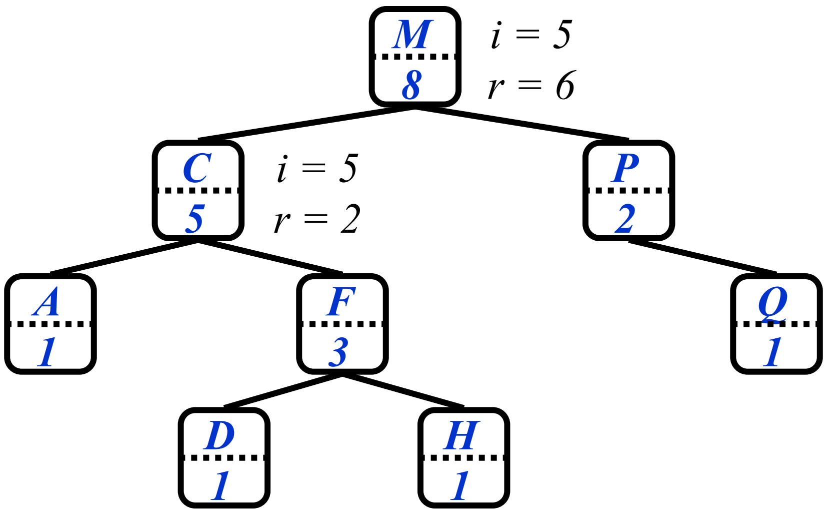
```
OS-Select(x, i)
{
   r = x->left->size + 1;
   if (i == r)
      return x;
   else if (i < r)
      return OS-Select(x->left, i);
   else
      return OS-Select(x->right, i-r);
}
```



OS-Select Example

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

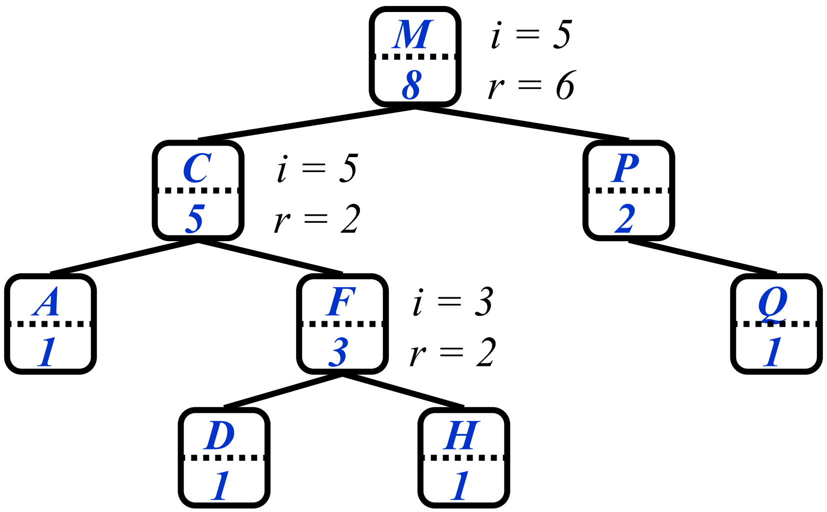
```
OS-Select(x, i)
{
   r = x->left->size + 1;
   if (i == r)
      return x;
   else if (i < r)
      return OS-Select(x->left, i);
   else
      return OS-Select(x->right, i-r);
}
```



OS-Select Example

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

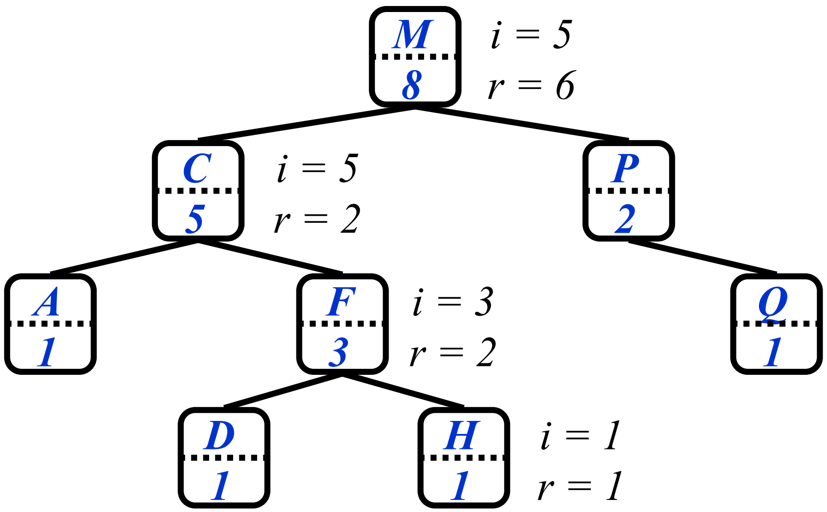
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



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);
}
```

Oops…

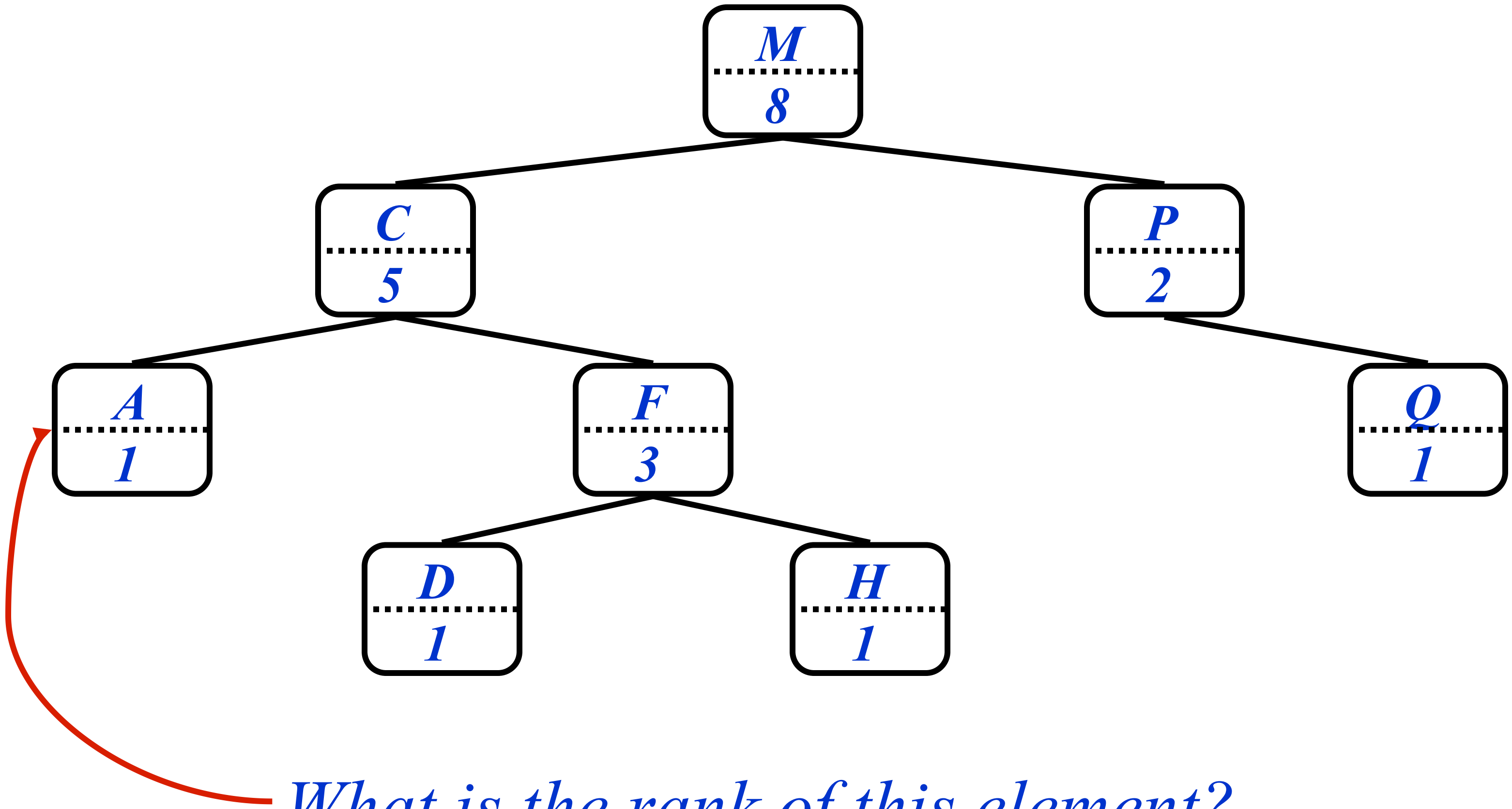
- *What happens at the leaves?*
- *How can we deal elegantly with this?*

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);
}
```

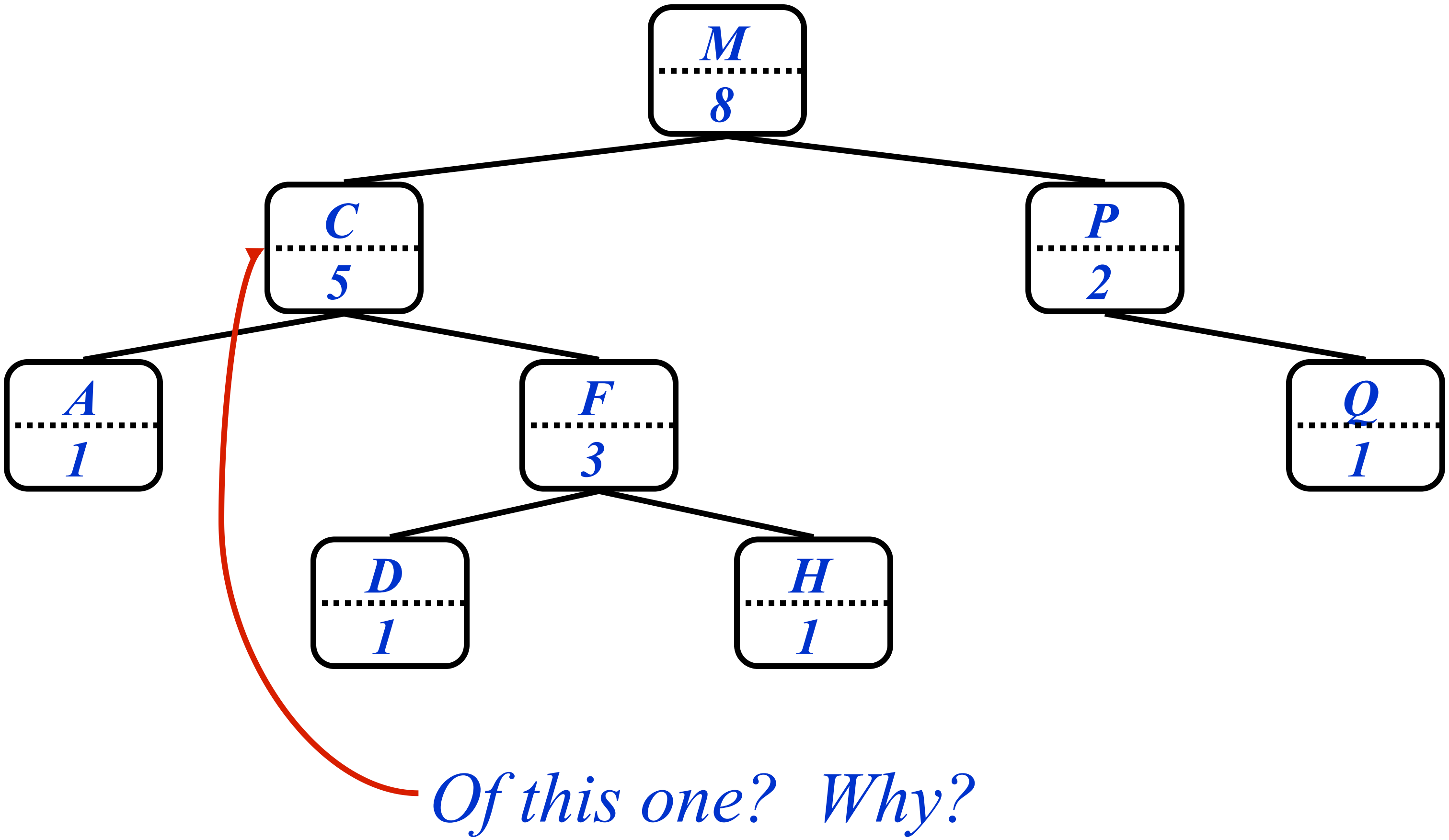
- *What will be the running time?*

Determining The Rank Of An Element

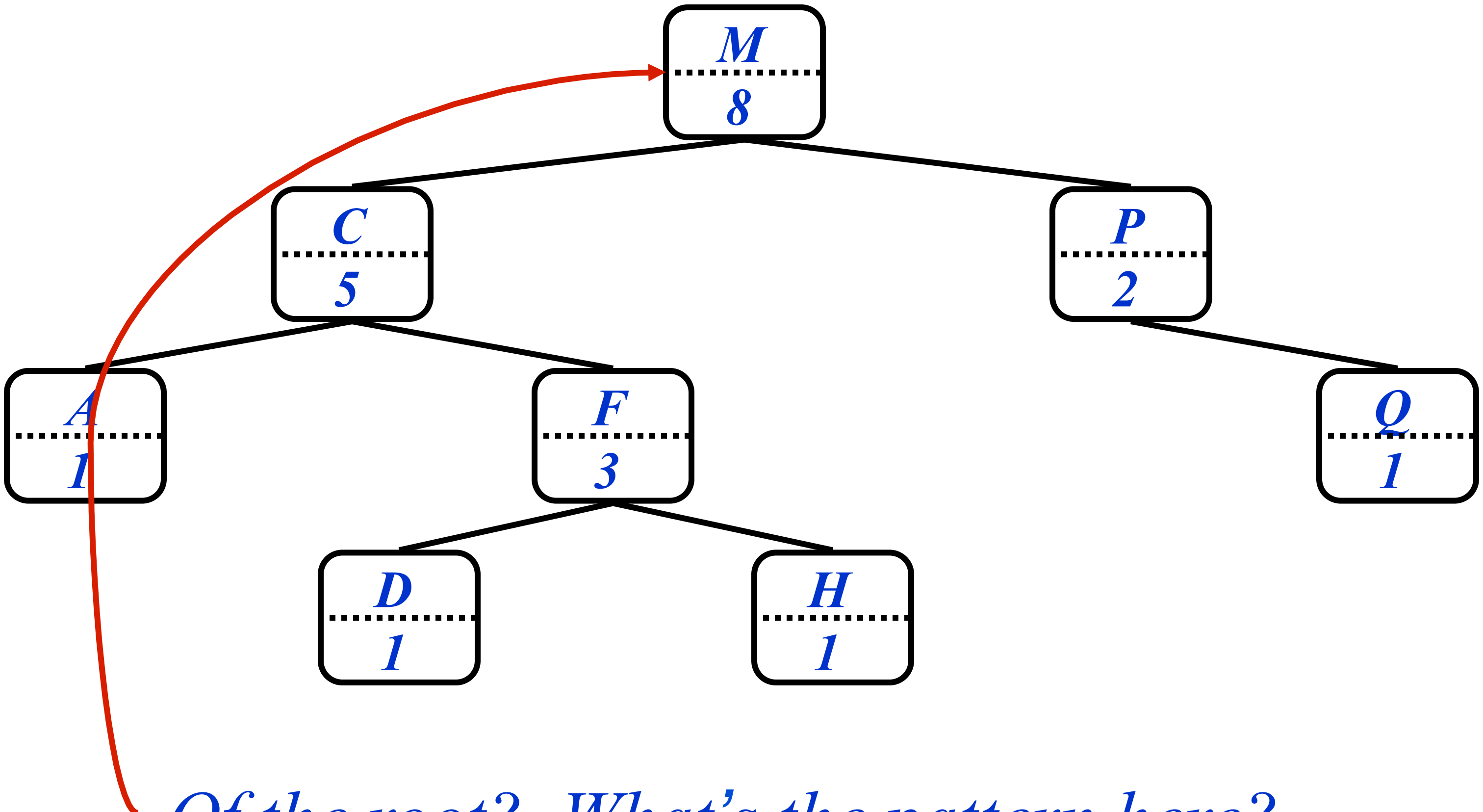


What is the rank of this element?

Determining The Rank Of An Element

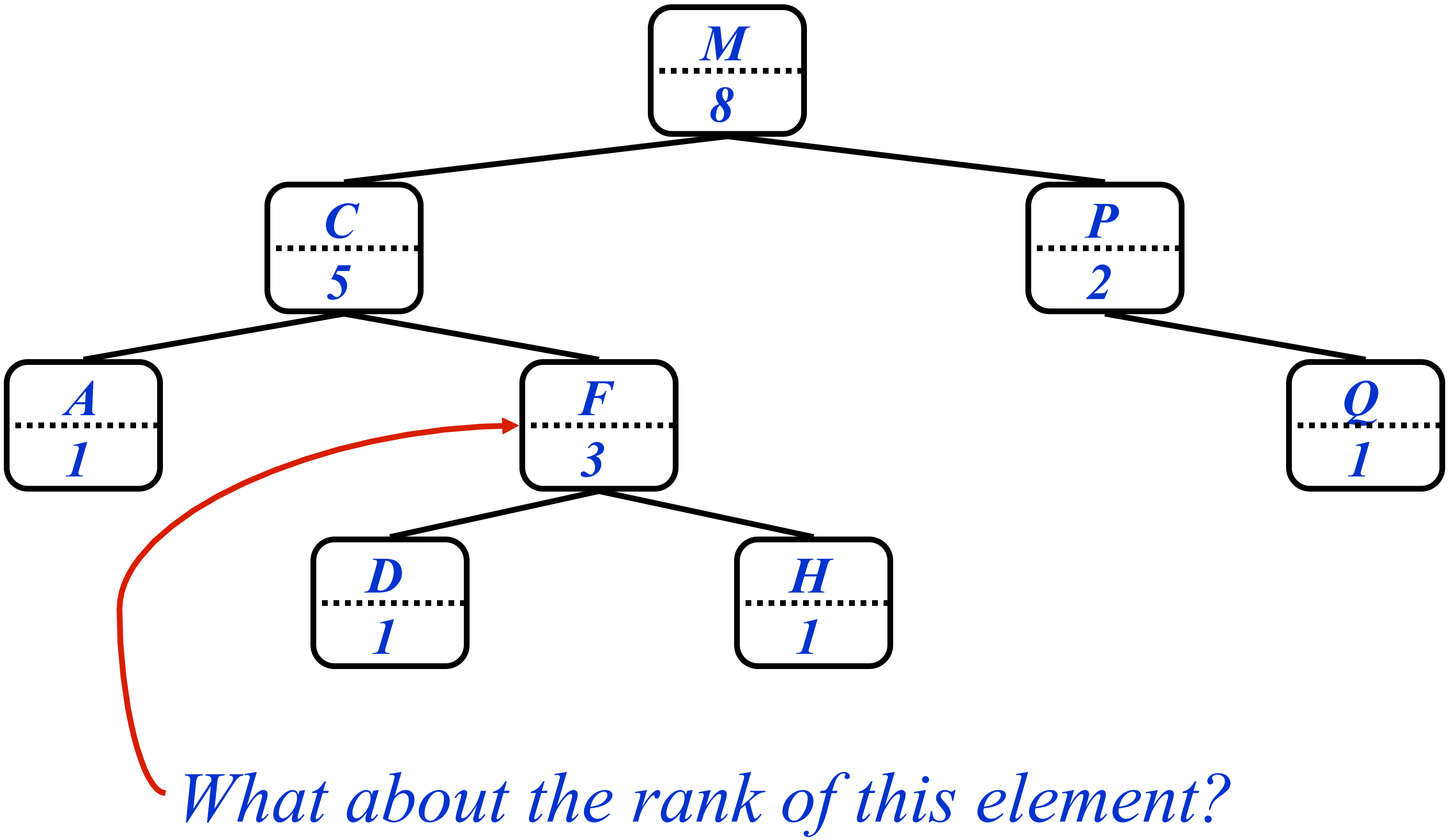


Determining The Rank Of An Element

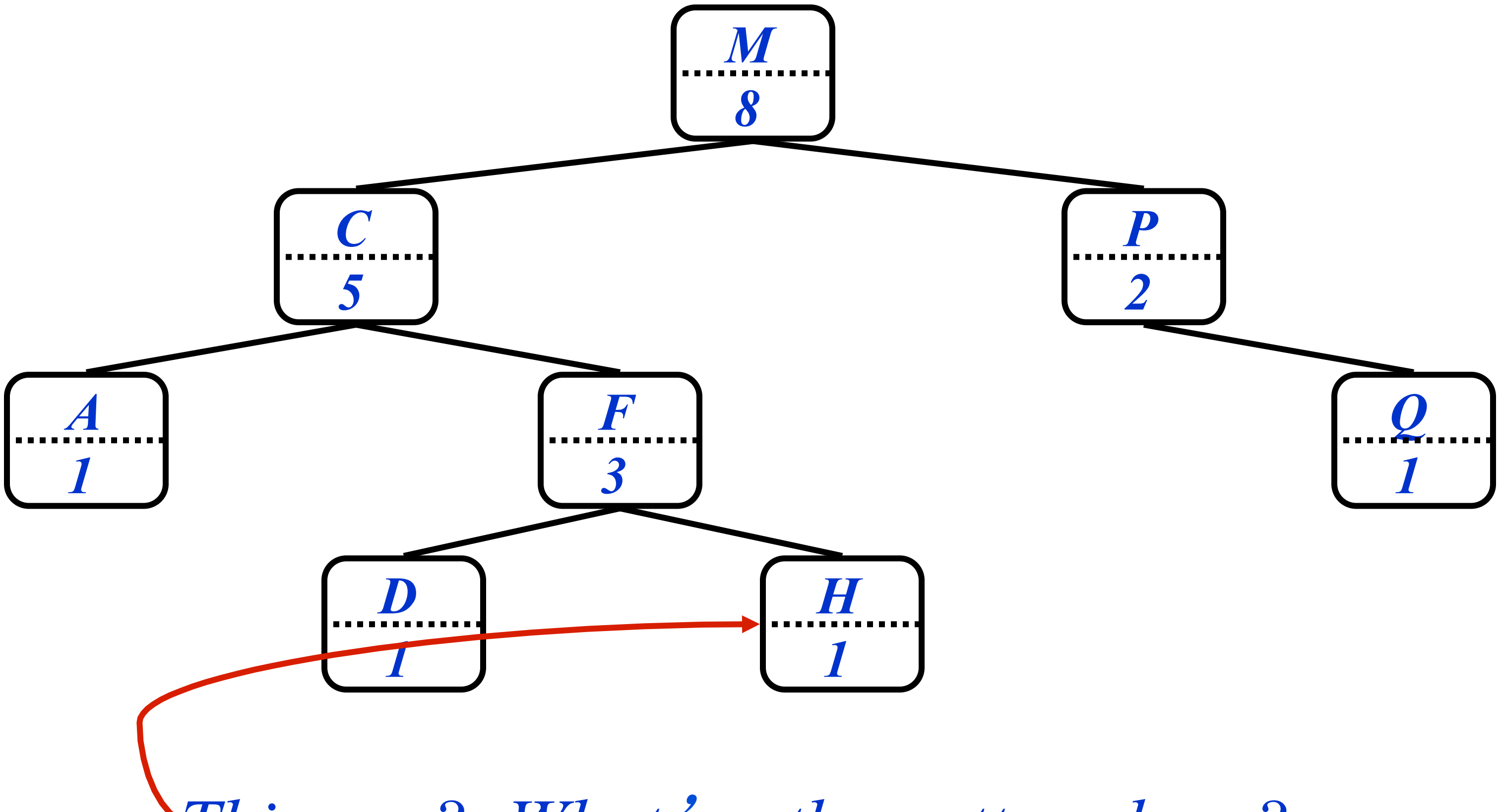


Of the root? What's the pattern here?

Determining The Rank Of An Element



Determining The Rank Of An Element



This one? What's the pattern here?

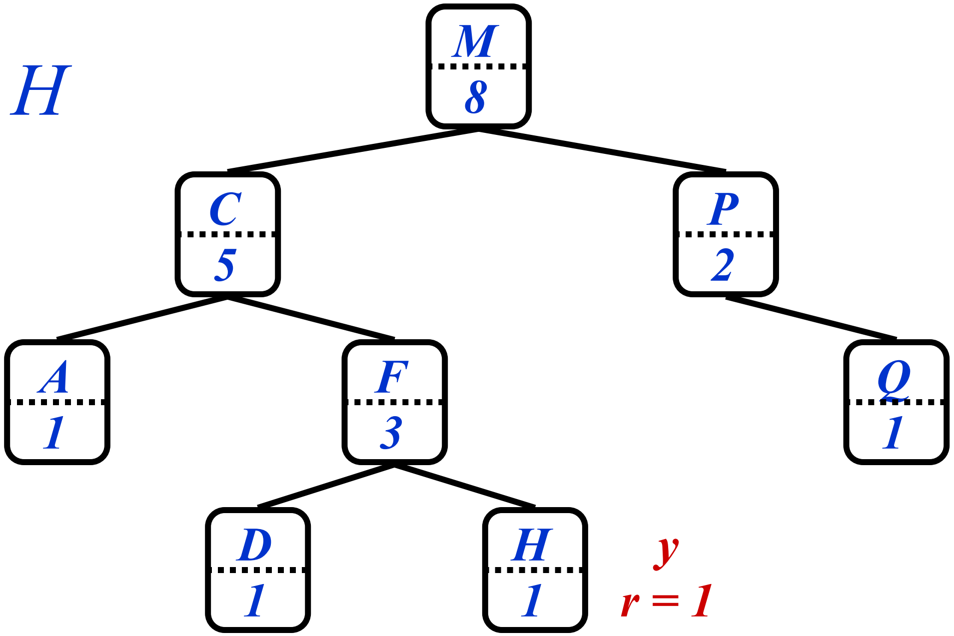
OS-Rank

```
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?*

Determining The Rank Of An Element

Example 1:
find rank of element with key H

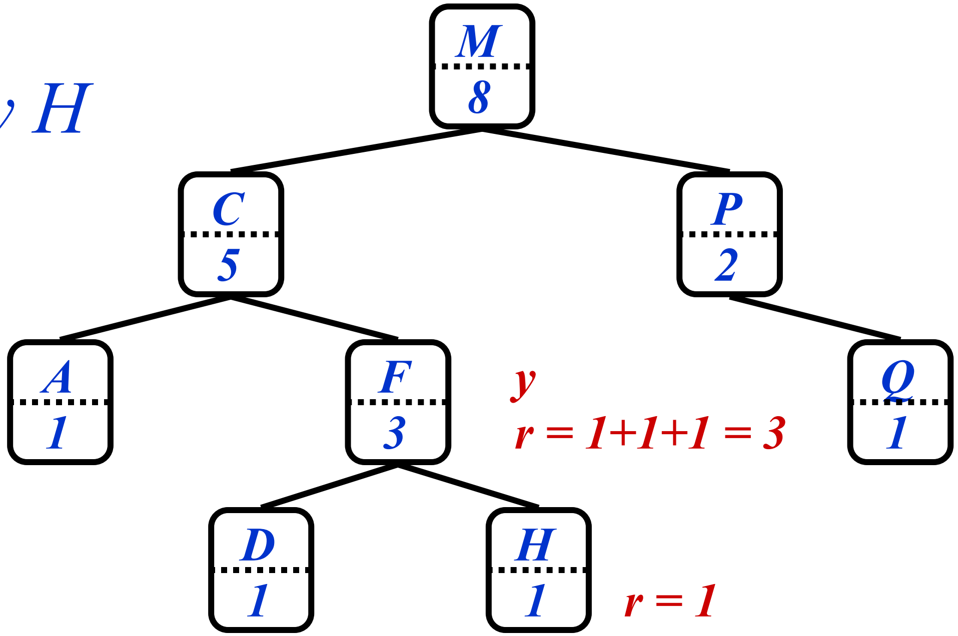



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

Determining The Rank Of An Element

Example 1:

find rank of element with key H

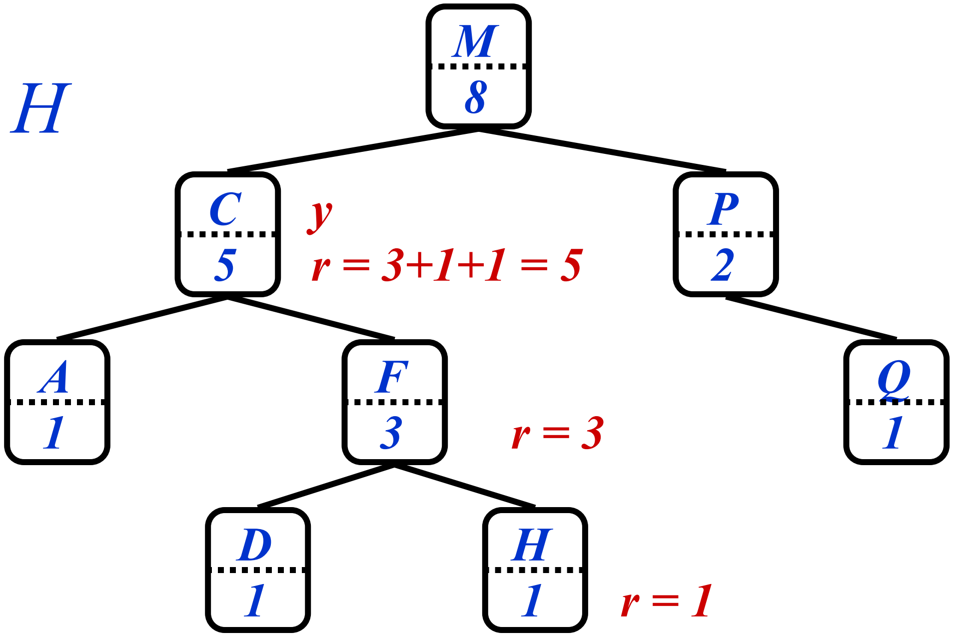



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

Determining The Rank Of An Element

Example 1:

find rank of element with key H

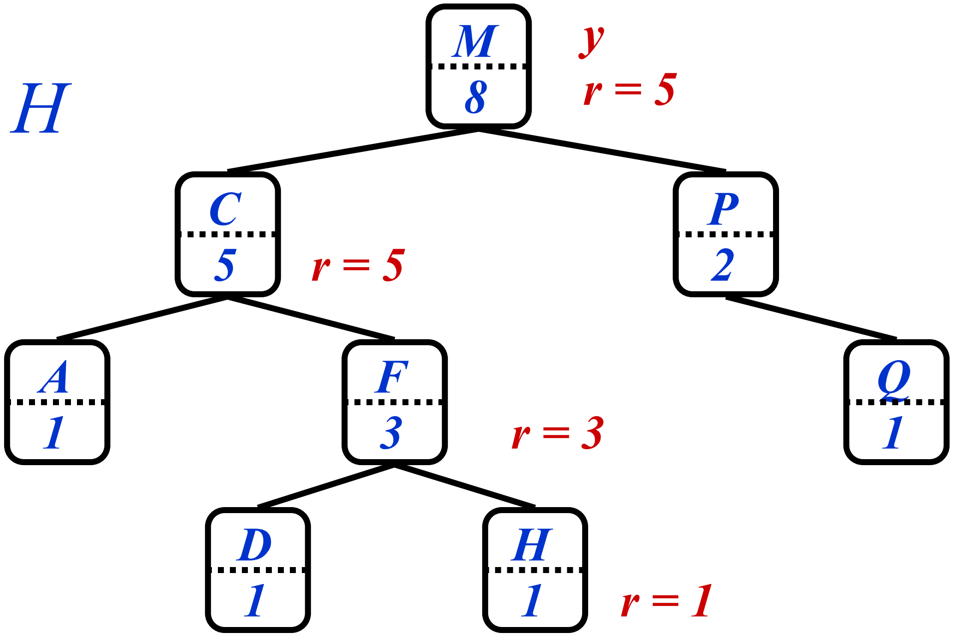



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

Determining The Rank Of An Element

Example 1:

find rank of element with key H

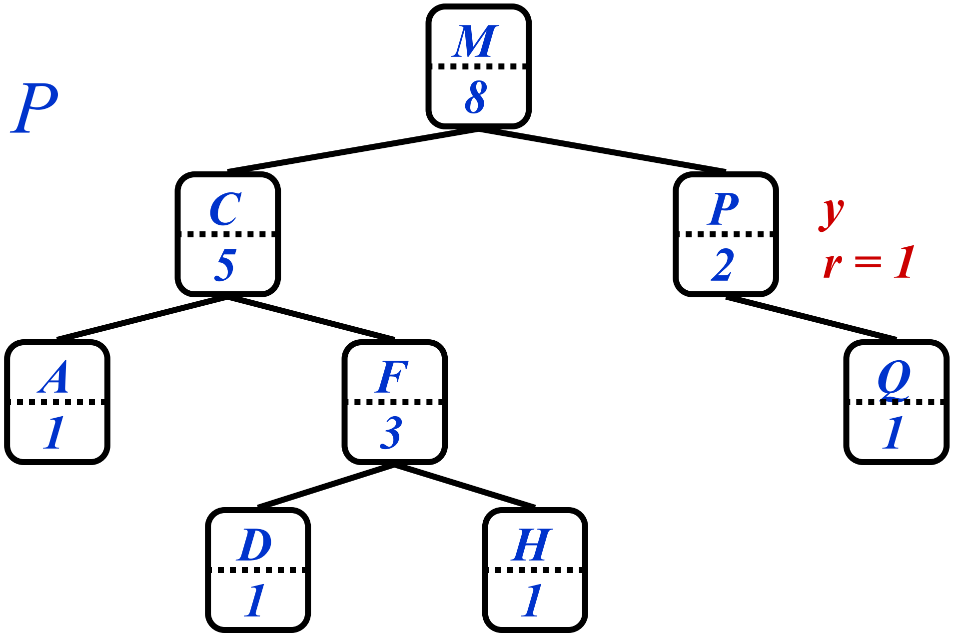



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

Determining The Rank Of An Element

Example 2:

find rank of element with key P

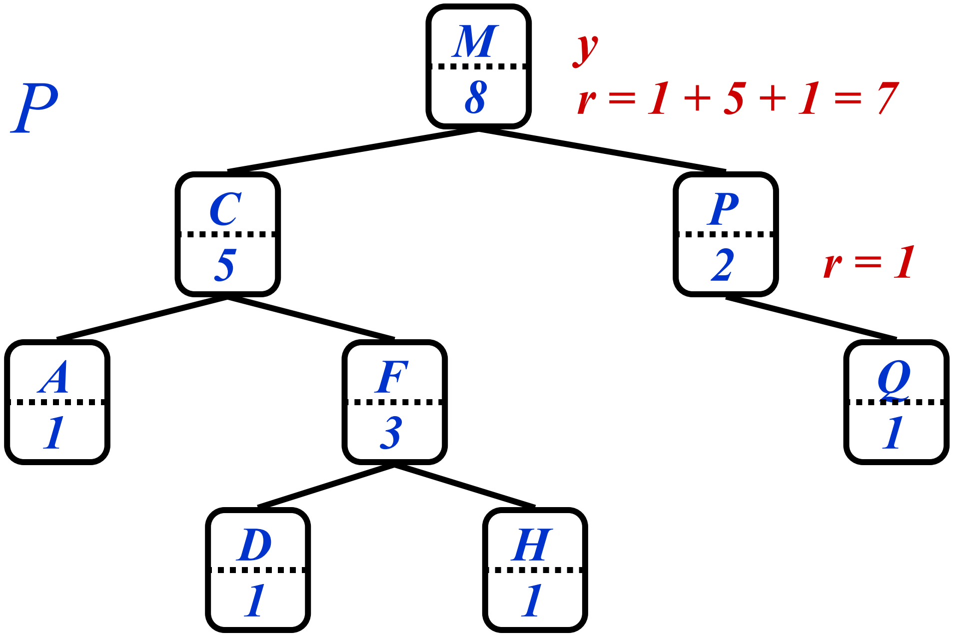



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

Determining The Rank Of An Element

Example 2:

find rank of element with key P

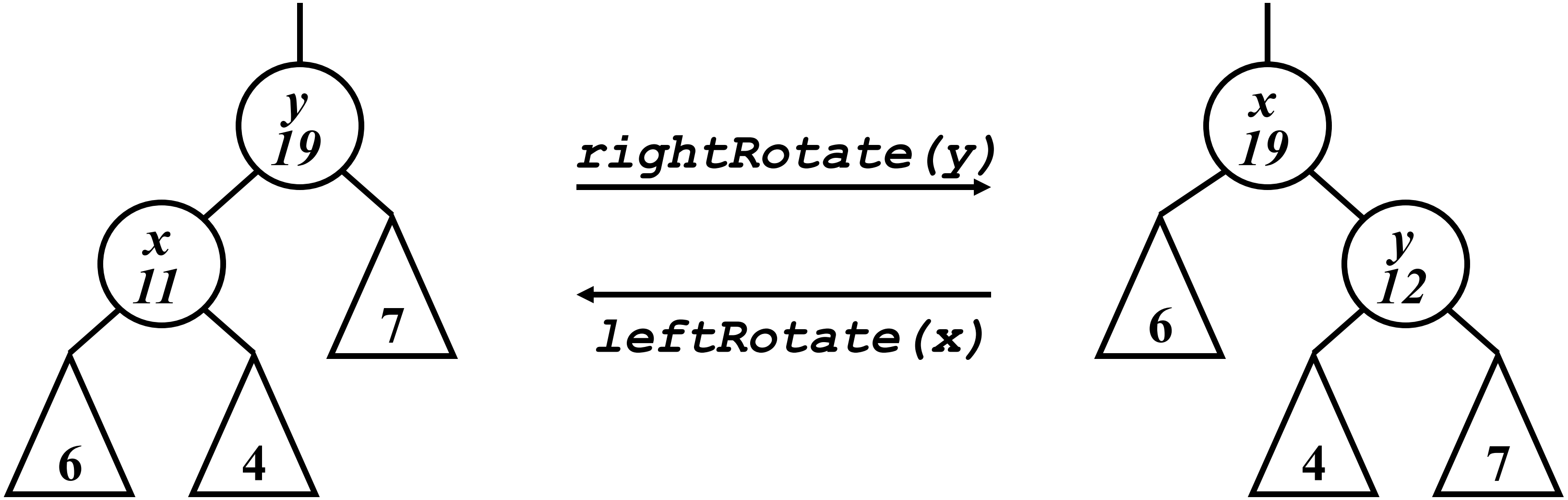



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

Maintaining Subtree Sizes

- So by keeping subtree sizes, order statistic operations can be done in O(lg n) time
- Maintain sizes during Insert() and Delete() operations
  - Insert(): Increment size fields of nodes traversed during search down the tree
  - Delete(): Decrement sizes along a path from the deleted node to the root
  - Both: Update sizes correctly during rotations

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()

Advanced Data Structures

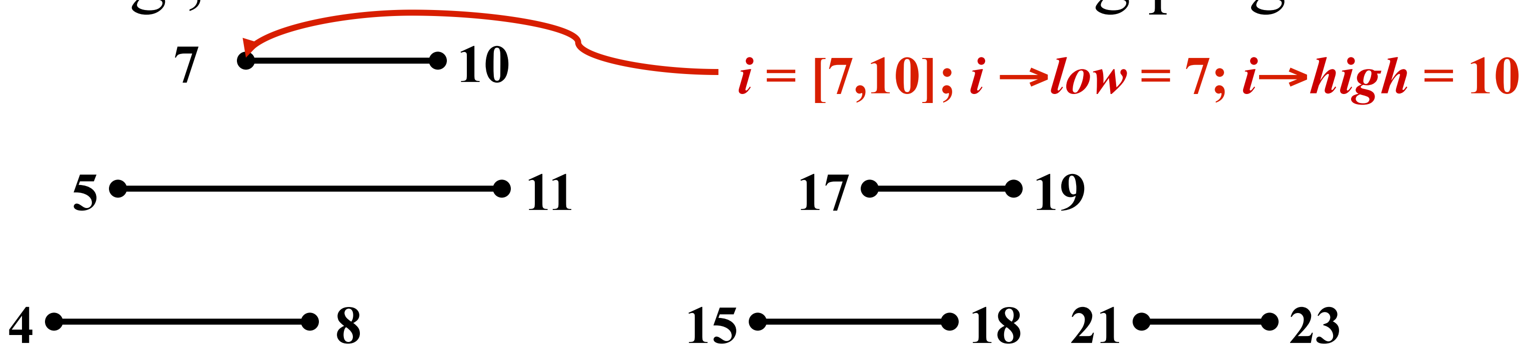
Augmenting Data Structures:
Interval Trees

Review: Methodology For Augmenting Data Structures

- Choose underlying data structure
- Determine additional information to maintain
- Verify that information can be maintained for operations that modify the structure
- Develop new operations

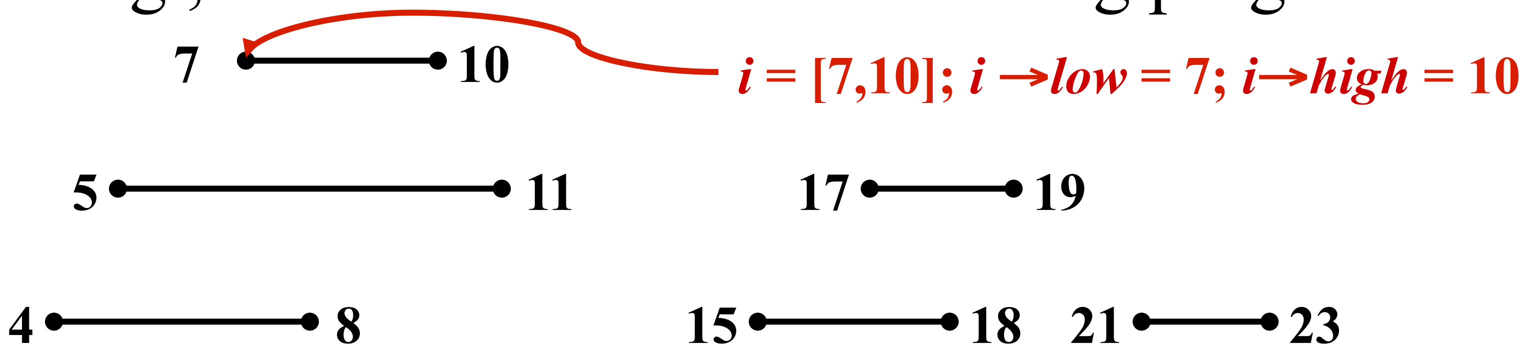
Interval Trees

- 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

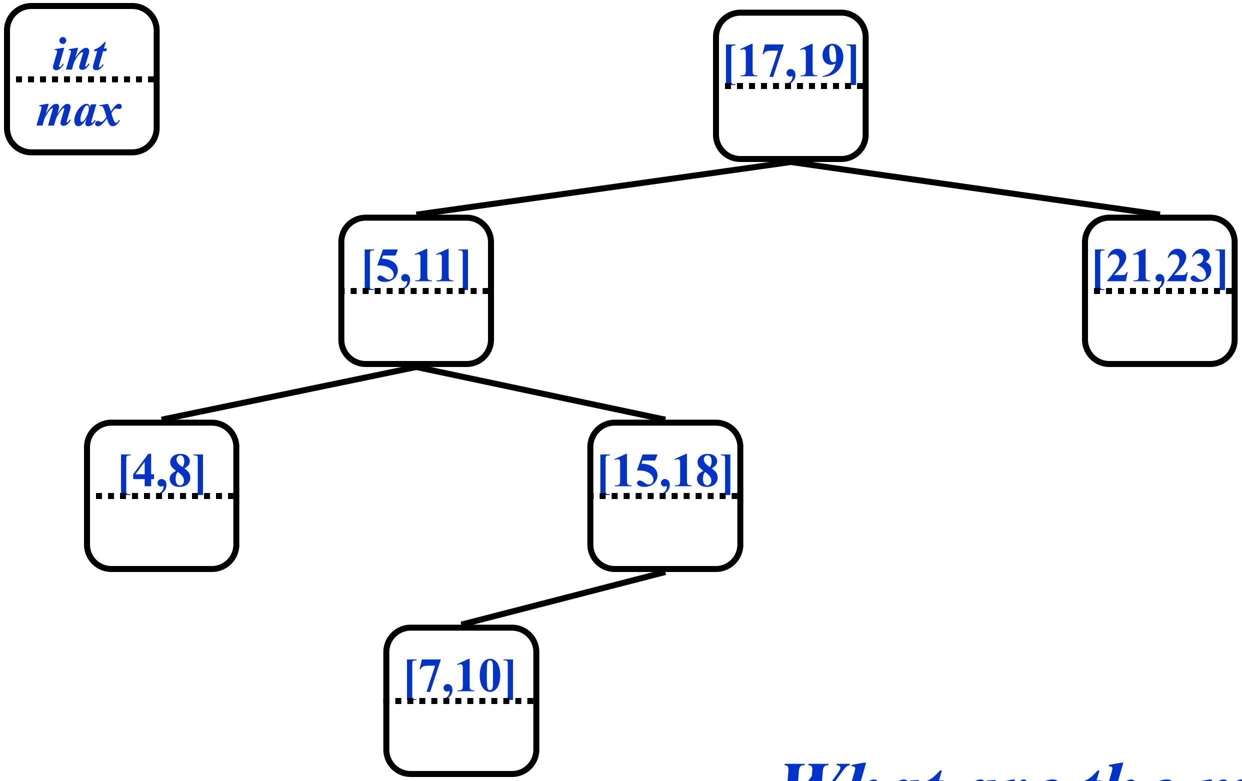
Interval Trees

- 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

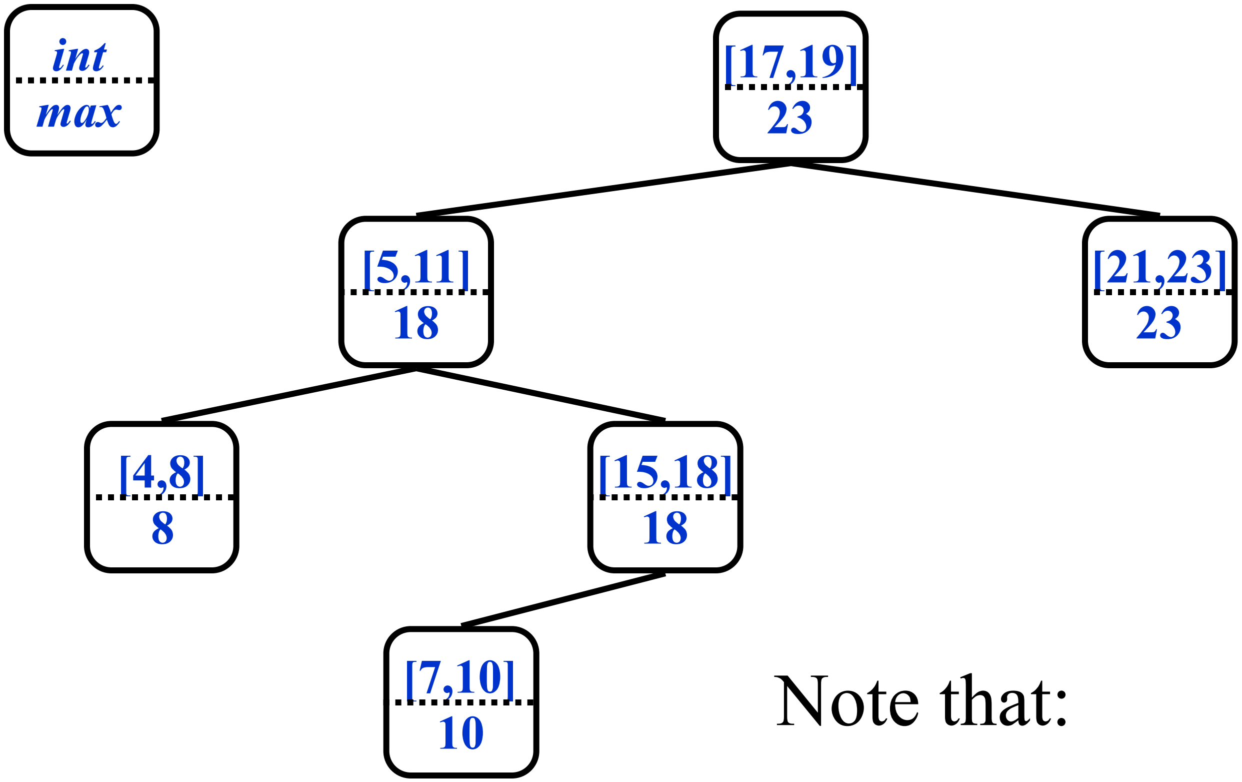
- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - *Decide what additional information to store*
    - We will store *max*, the maximum endpoint in the subtree rooted at *i*
  - Figure out how to maintain the information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

Interval Trees



What are the max fields?

Interval Trees



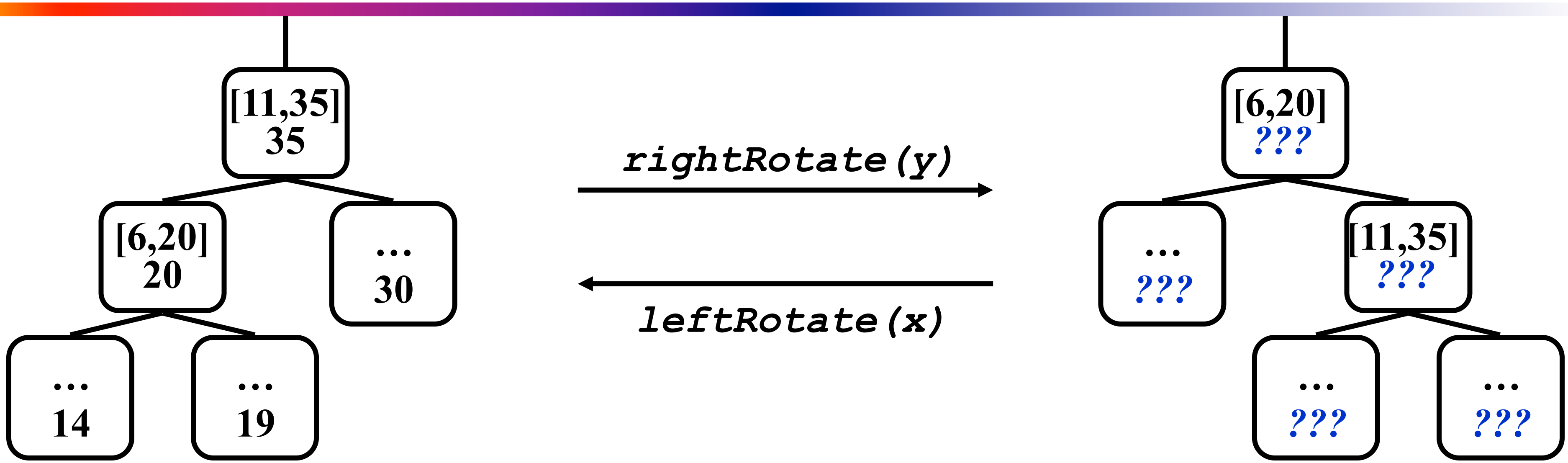
Note that:

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

Interval Trees

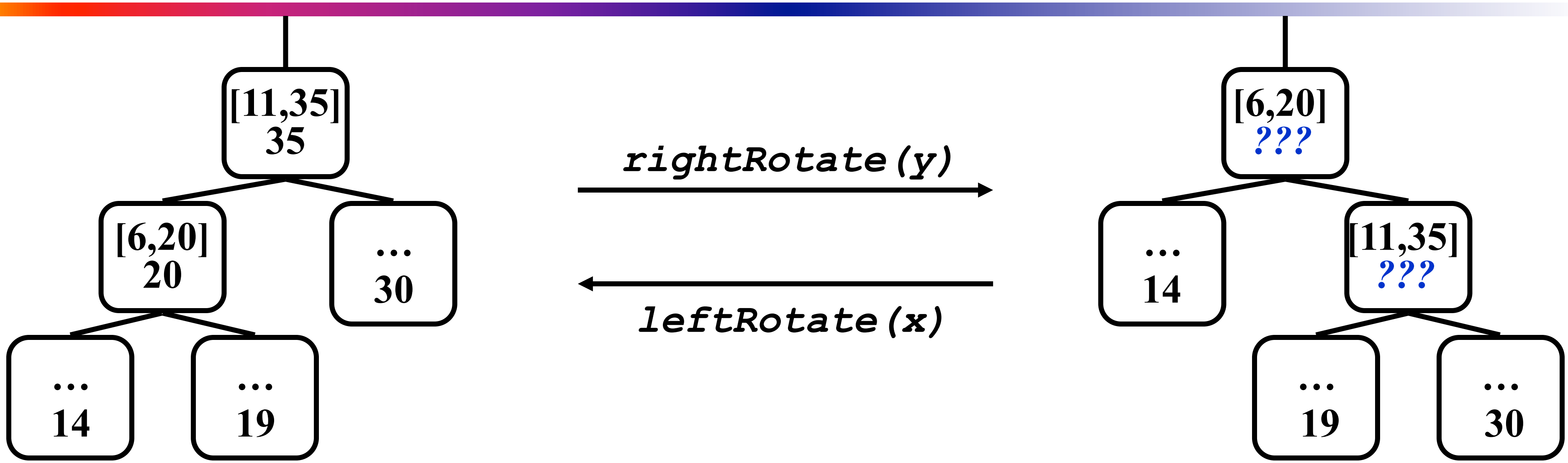
- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - Decide what additional information to store
    - Store the maximum endpoint in the subtree rooted at i
  - *Figure out how to maintain the information*
    - *How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

Interval Trees



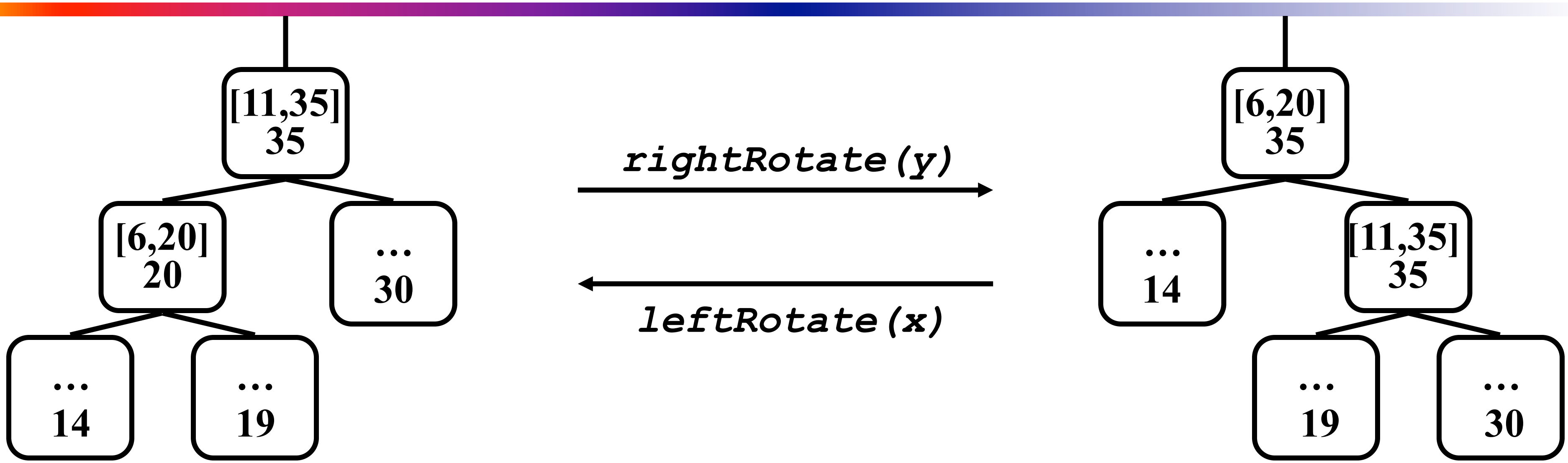
- *What are the new max values for the subtrees?*

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*

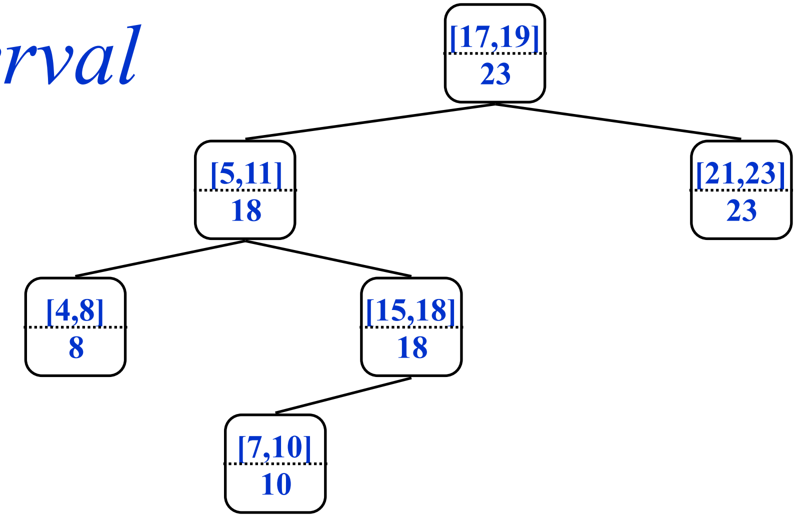
Searching Interval Trees

```
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

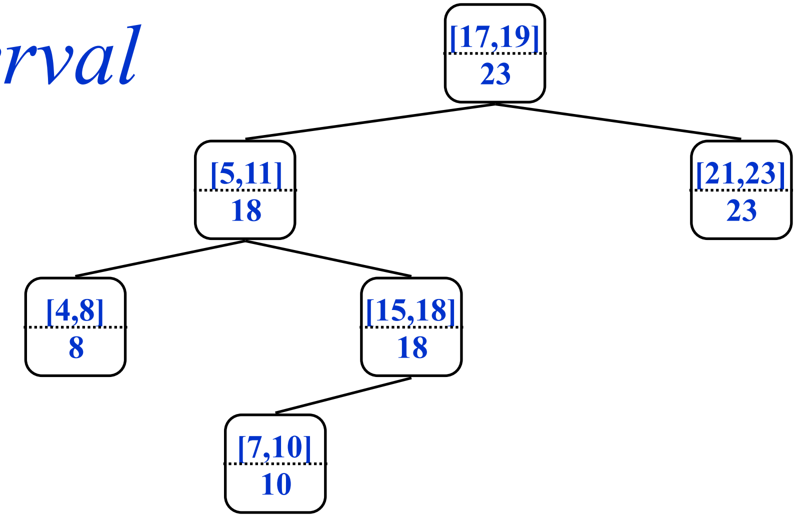
- *Example: search for interval overlapping [14,16]*




```
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;
```