Medians and Order Statistics

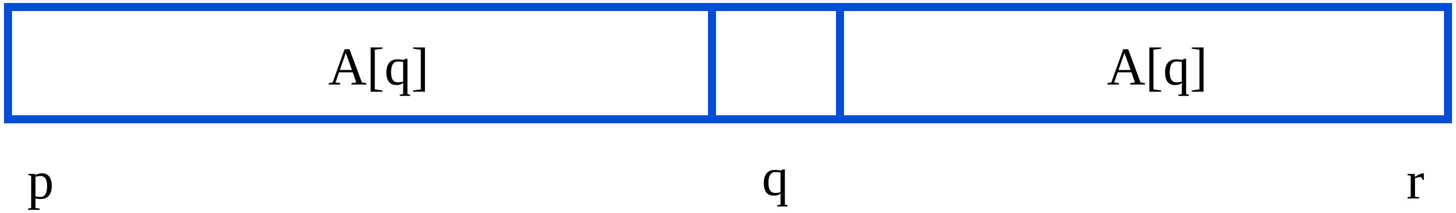
- The ith *order statistic* in a set of n elements is the ith smallest element
- The *minimum* is thus the 1st order statistic
- The *ma imum* is the nth order statistic
- The *median* is the $n/2$ order statistic
  - If n is even, there are 2 medians
- *Ho can e calculate order statistics?*
- *What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

- *Ho man comparisons are needed to find the minimum element in a set? The ma imum?*
- *Can e find the minimum and ma imum ith less than t ice 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)

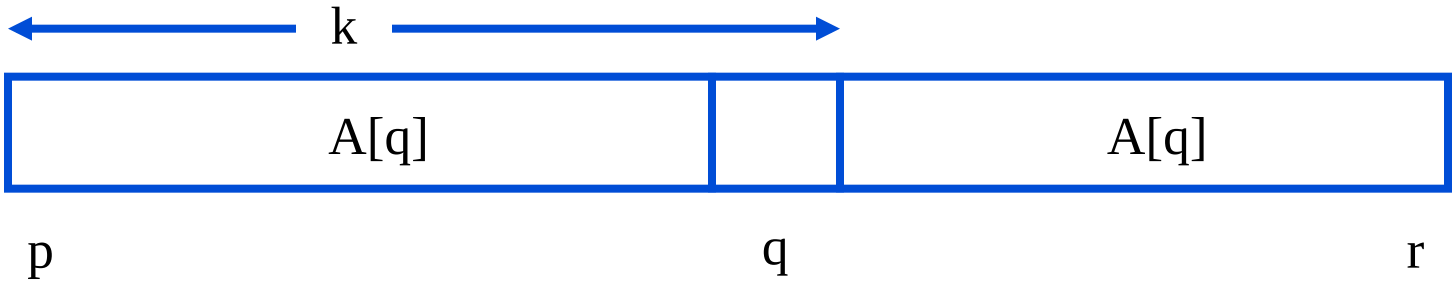
- A more interesting problem is *selection*: finding the *i*th smallest element of a set
- We will show:
  - A practical randomized algorithm with O(n) expected running time
  - A cool algorithm of theoretical interest only with O(n) worst-case running time

- 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)



```
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);
```



- 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?*

- Average case
  - For upper bound, assume *i*

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

$$(\quad) \le \frac{}{} \sum_{= \ /}^{-1} (\quad) + \Theta(\quad)$$

$$\le \frac{}{} \sum_{= \ /}^{-1} \quad + \Theta(\quad)$$ ***W*** ***?***

$$= \frac{}{}\left(\sum_{=1}^{-1} \quad - \sum_{=1}^{/ \ -1} \quad \right) + \Theta(\quad)$$ ***“S ”***

$$= \frac{}{}\left(\frac{1}{}(\quad -1) \quad - \frac{1}{}\left(\frac{}{} - 1 \right) \frac{}{} \right) + \Theta(\quad)$$ ***W*** ***?***

$$= \quad (\quad -1) - \frac{}{}\left(\frac{}{} - 1 \right) + \Theta(\quad)$$ ***W*** ***?***

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

$$T\ \ n \le c\left(n - \ \right) - \frac{c}{\ }\left(\frac{n}{\ } - \ \right) + \Theta(n)$$ ***T***

$$= cn - c - \frac{cn}{\ } + \frac{c}{\ } + \Theta(n)$$ ***M***

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

$$= cn - \left(\frac{cn}{\ } + \frac{c}{\ } - \Theta(n)\right)$$ ***R***

$$\le cn$$ ***W***

- 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

- The algorithm in words:

1. Divide *n* elements into groups of 5
2. Find median of each group (*Ho ? Ho long?*)
3. Use Select() recursively to find median of the ⌊n/5⌋ medians
4. Partition the *n* elements around . Let *k* = rank( )
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

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

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

$$T(n) \quad T\left(\lfloor n/5 \rfloor\right) \quad T\left(3n/4\right) \qquad \left(n\right)$$

$$T\left(n/5\right) \quad T\left(3n/4\right) \qquad \left(n\right)$$

n

$$cn/5 \quad 3cn/4 \quad \Theta(n)$$

$$= 19cn/20 + \Theta(n)$$

$$= cn - \left(cn/20 - \Theta\left(n\right)\right)$$

$$\leq cn \quad \text{if } c \text{ is big enough}$$

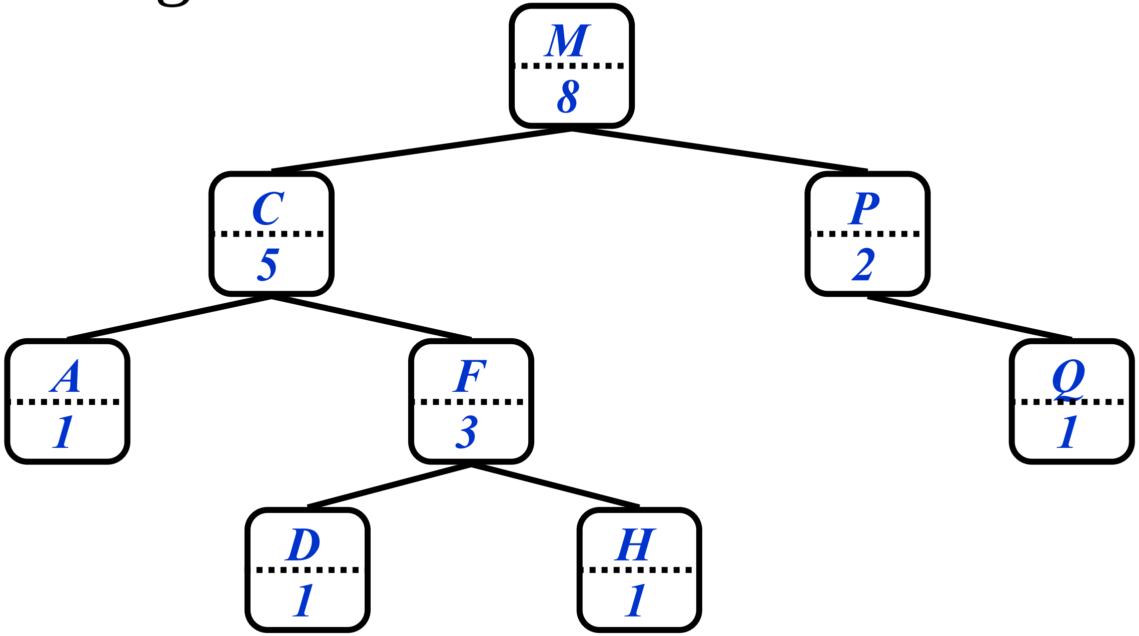
- Intuitively:
  - Work at each level is a constant fraction (19/20) smaller
    - Geometric progression!
  - Thus the O(n) work at the root dominates

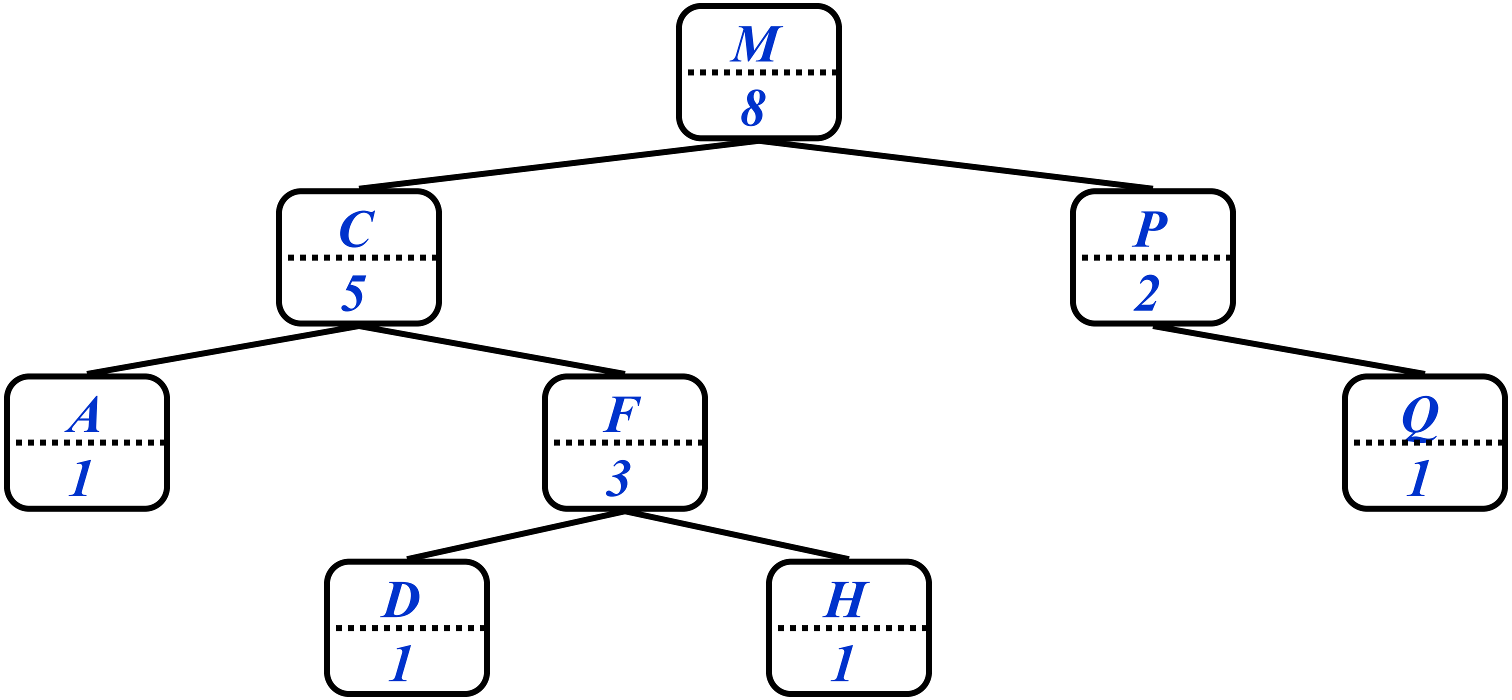
- Given a “black box” O(n) median algorithm, what can we do?
  - *i*th order statistic:
    - Find median
    - Partition input around
    - if (*i*   (n+1)/2) recursively find *i*th element of first half
    - else find (*i* - (n+1)/2)th element in second half
    - T(n) = T(n/2) + O(n) = O(n)

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

- 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 d namic sets usuall support?*
  - *What structure orks ell for these?*
  - *Ho could e use this str -2□(t) 2□(r)-2□(uc) □□255349cm*

- 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:



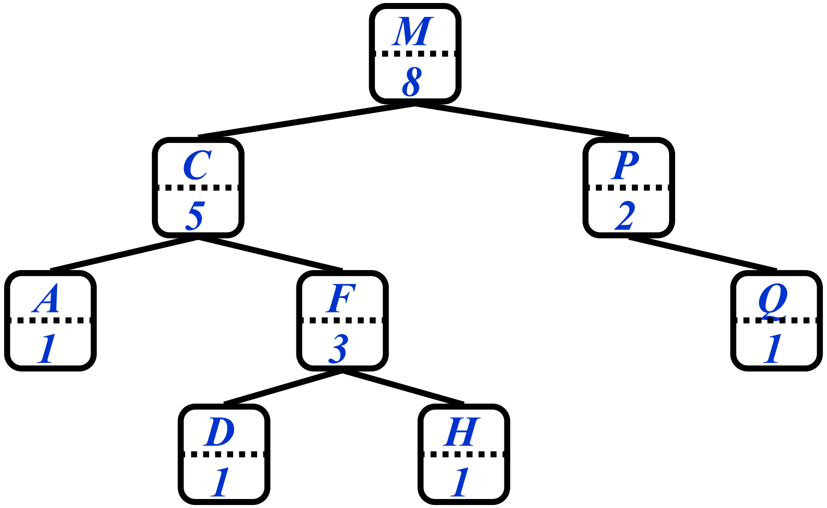


Ho can e use this propert
to select the i*th element of the set?*

```
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);
}
```

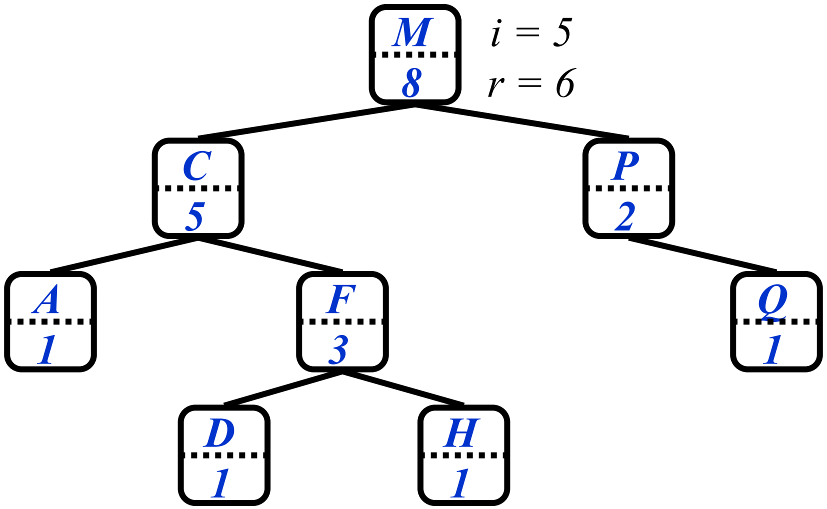
- Example: show OS-Select(*root*, 5):

```
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);
}
```



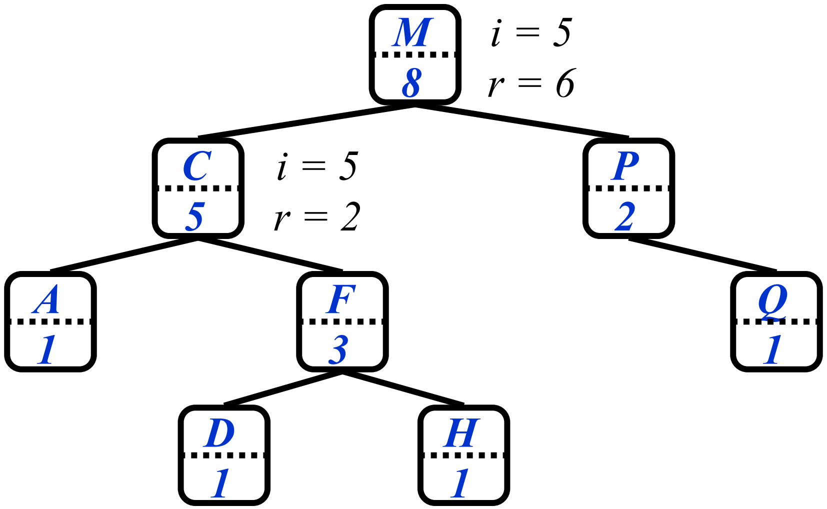
- Example: show OS-Select(*root*, 5):

```
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);
}
```



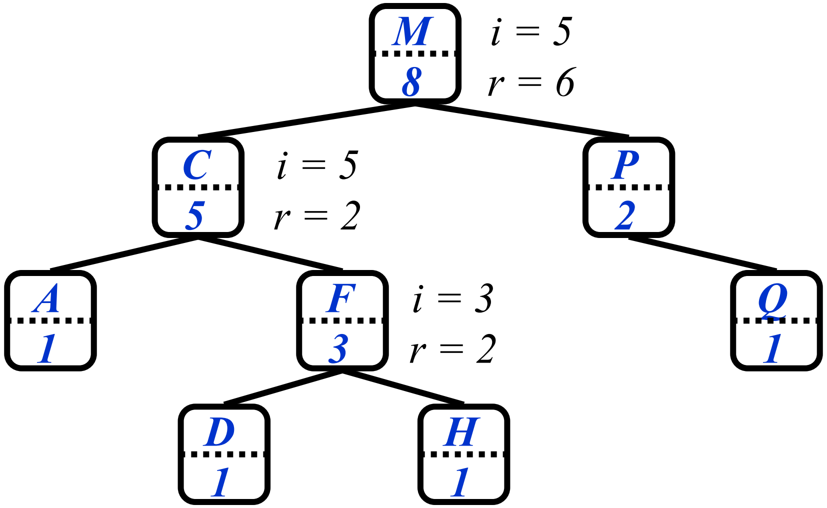
- Example: show OS-Select(*root*, 5):

```
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);
}
```



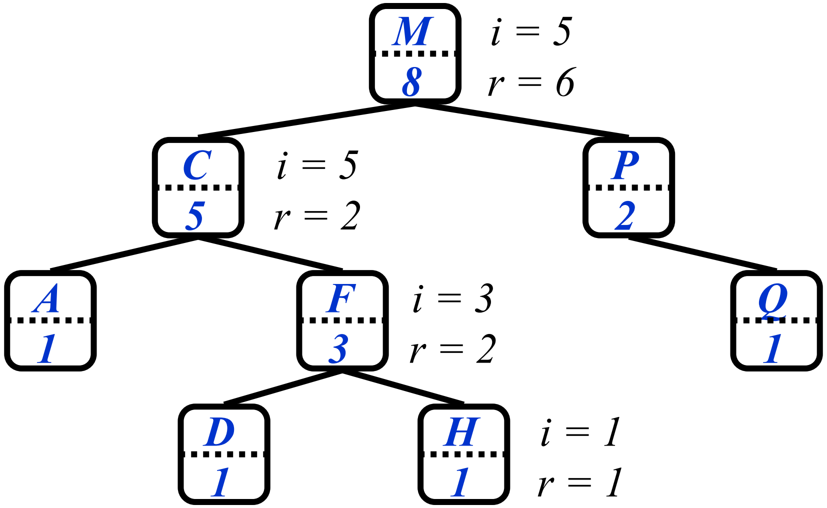
- Example: show OS-Select(*root*, 5):

```
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);
}
```



- 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);
}
```



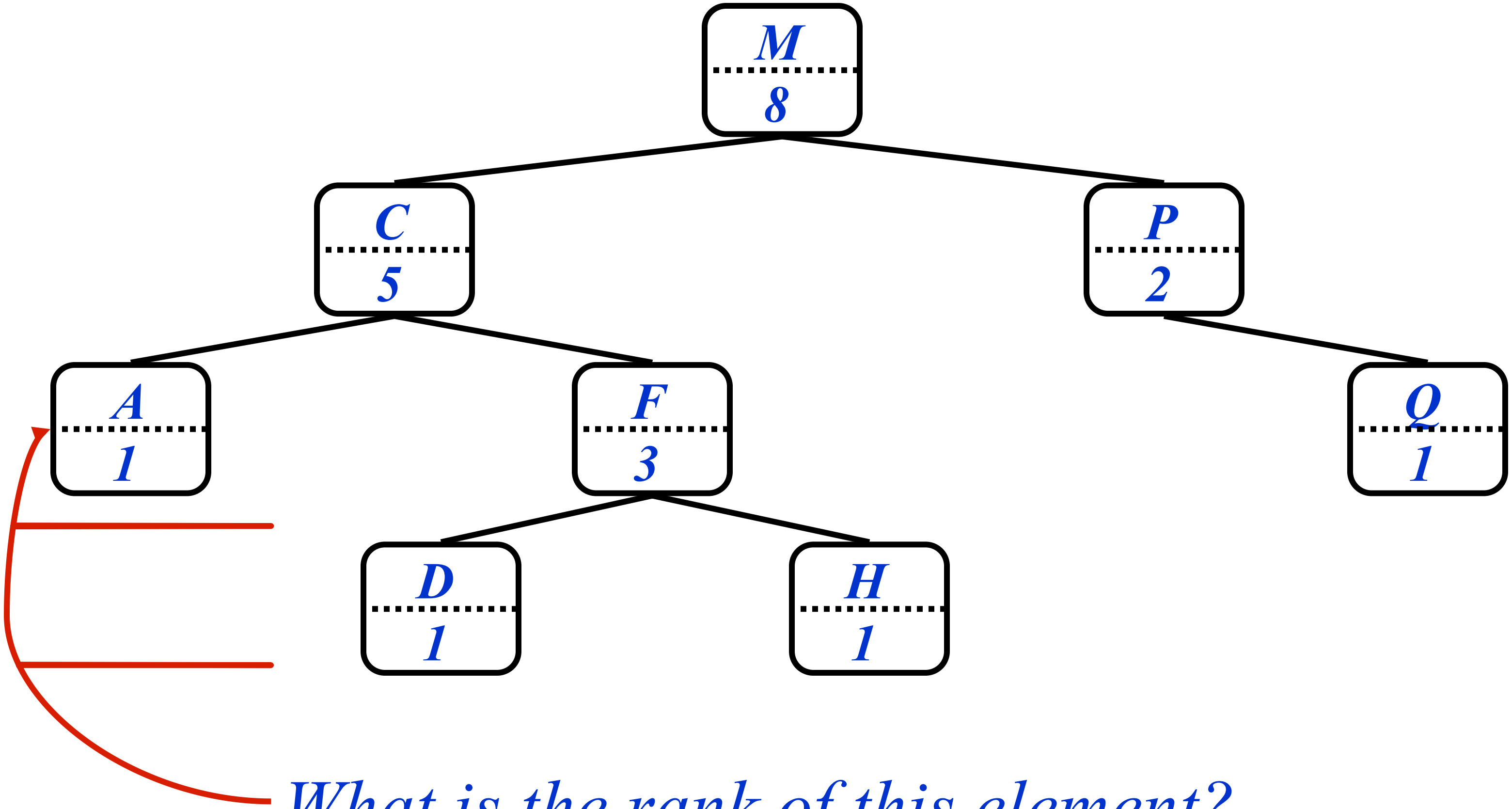
```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```

Oops…

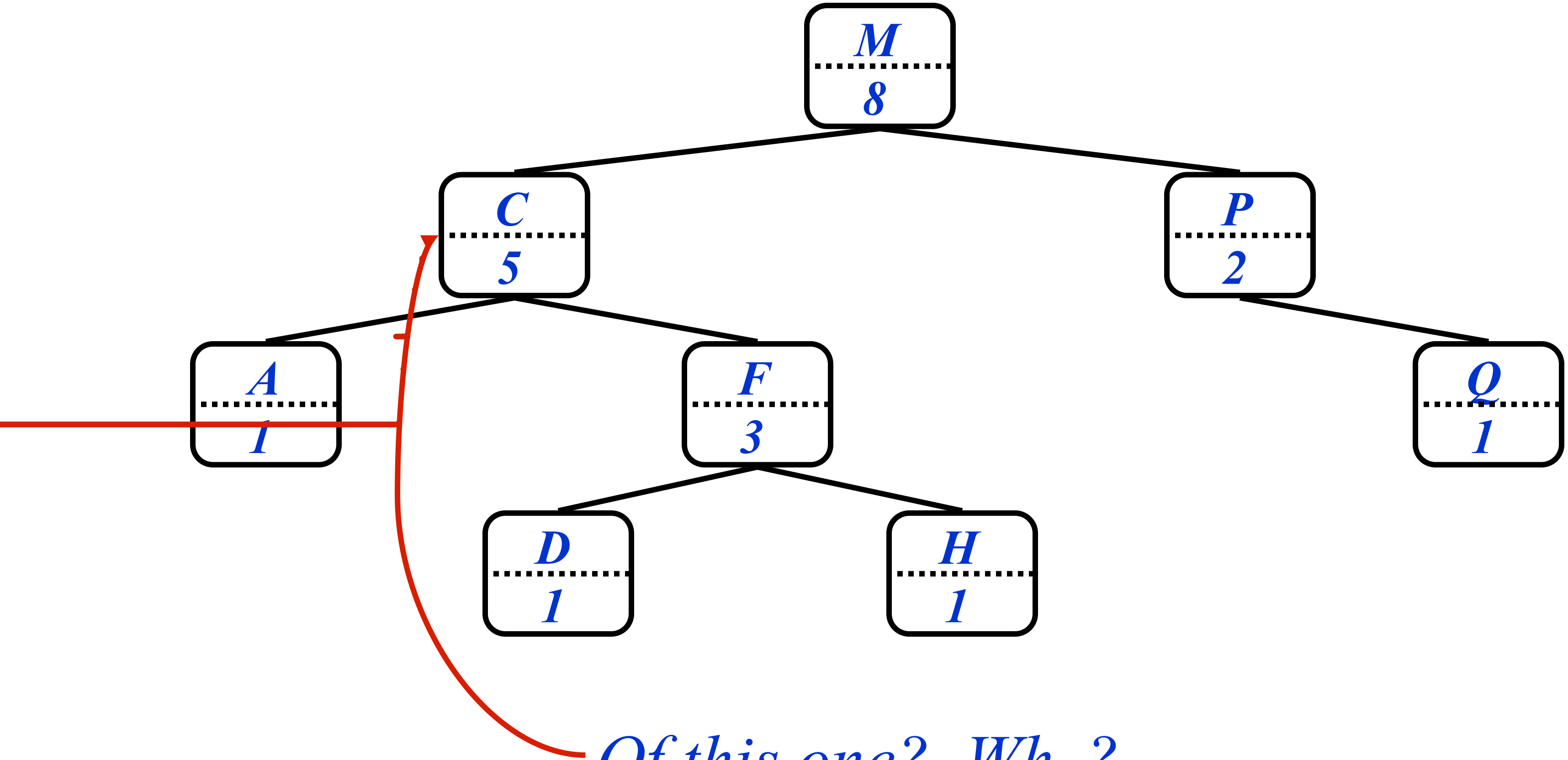
- *What happens at the leaves?*
- *Ho can e deal elegantl ith this?*

```
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 ill be the running time?*

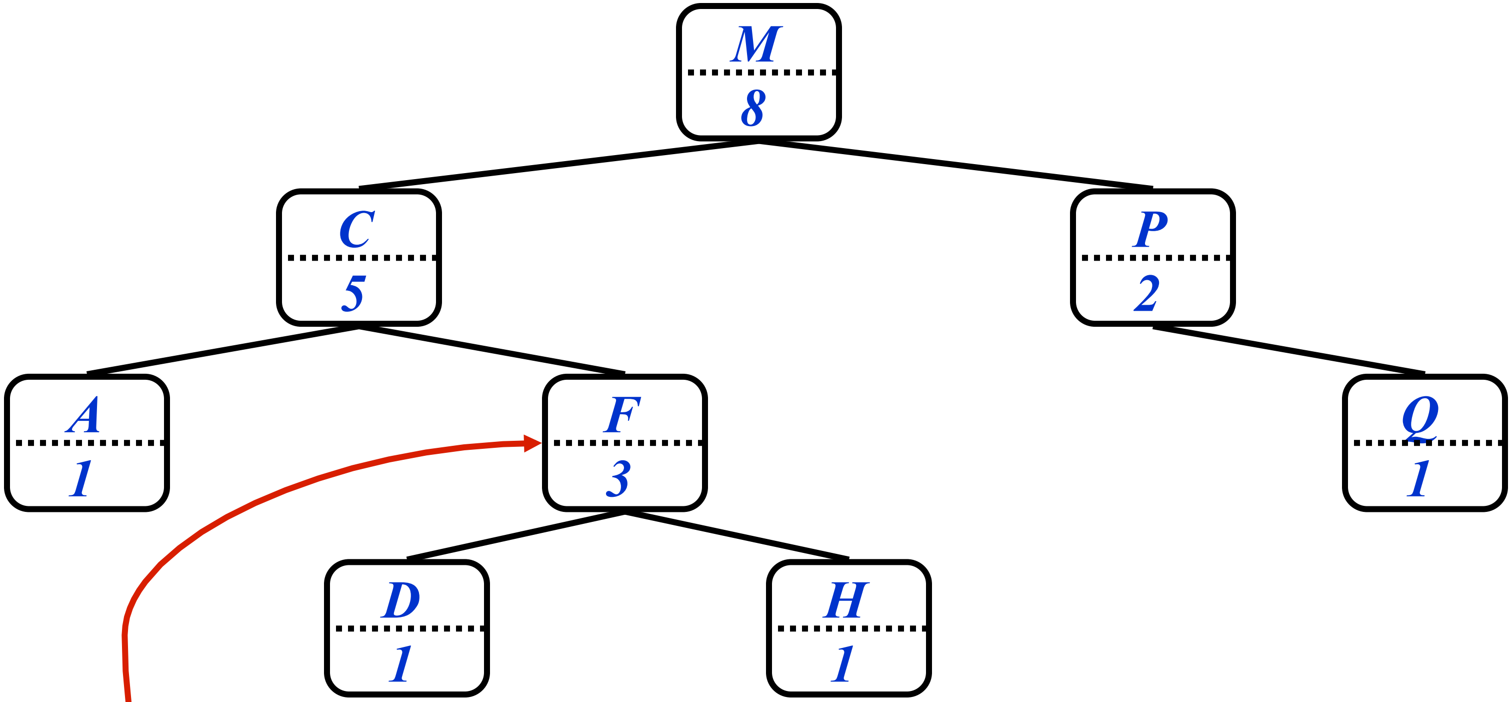


What is the rank of this element?

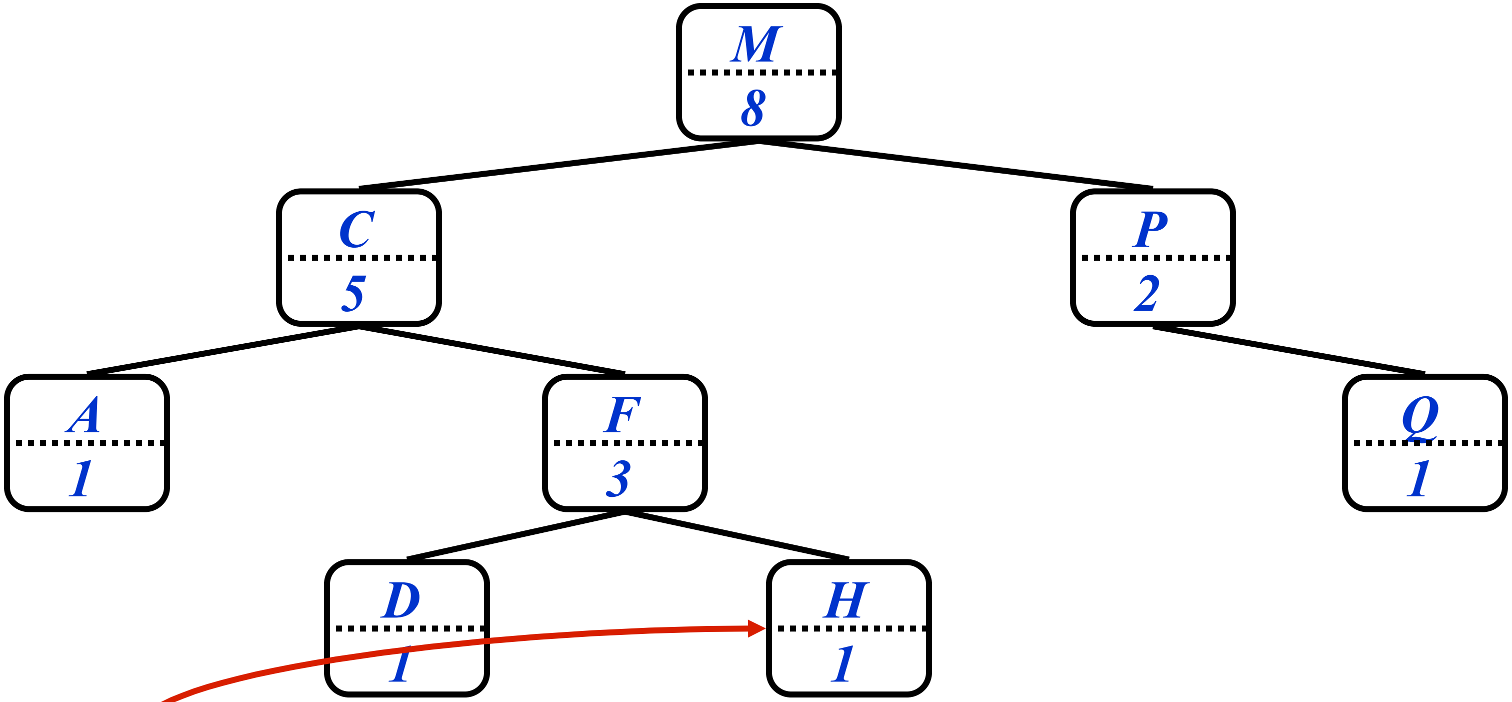


Of this one? Wh ?

M



What about the rank of this element?

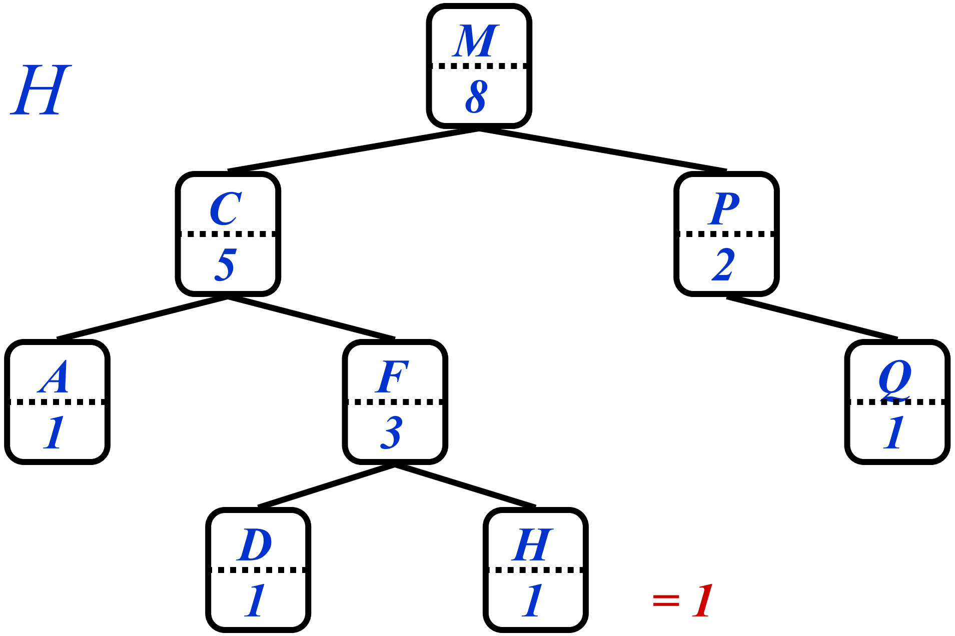


This one? What’ s the pattern here?

```
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 ill be the running time?*

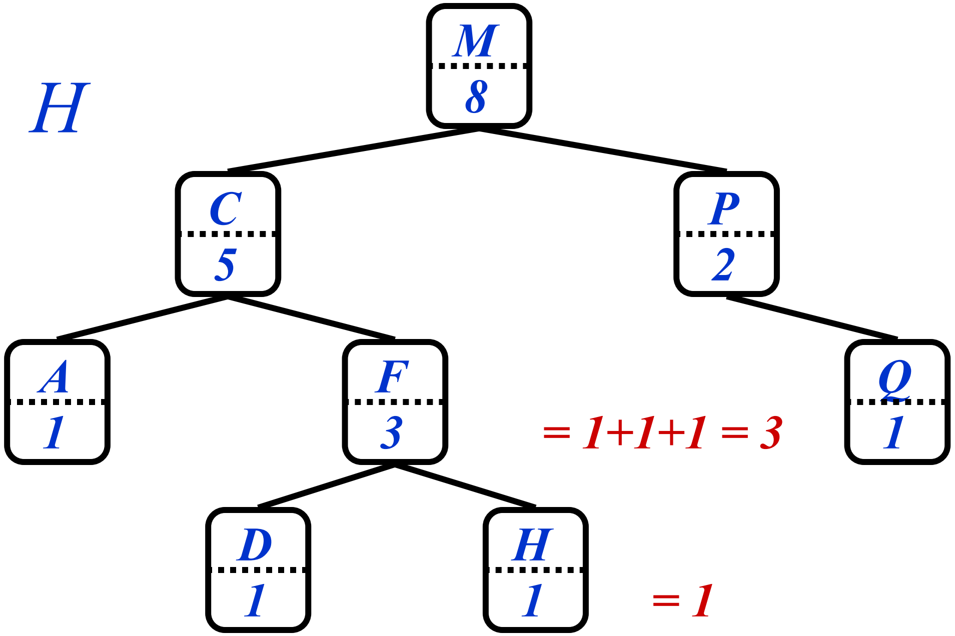
E ample 1:
find rank of element ith ke H



```
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;
}
```

E ample 1:

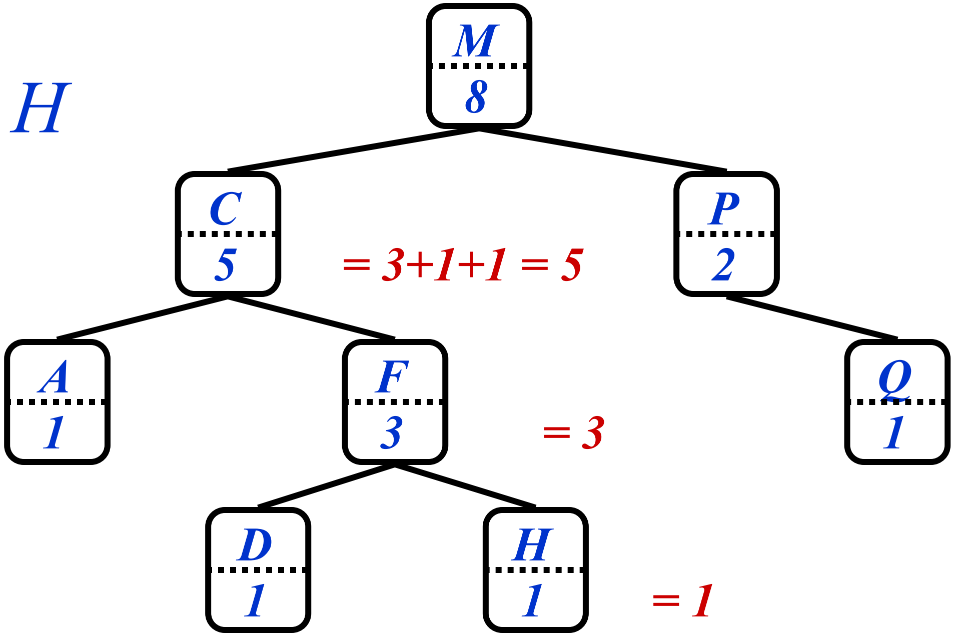
find rank of element ith ke H



```
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;
}
```

E ample 1:

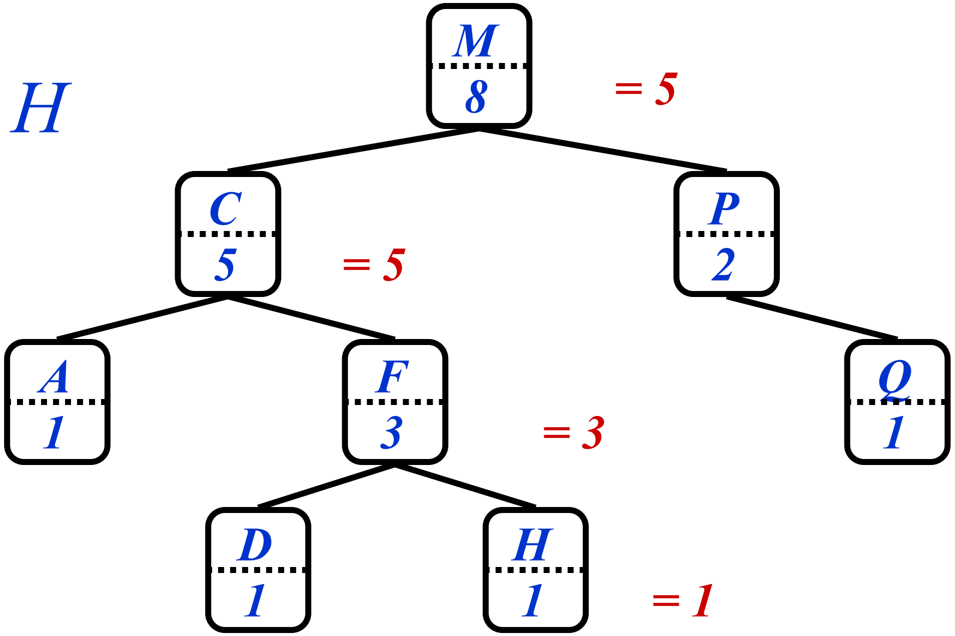
find rank of element ith ke H



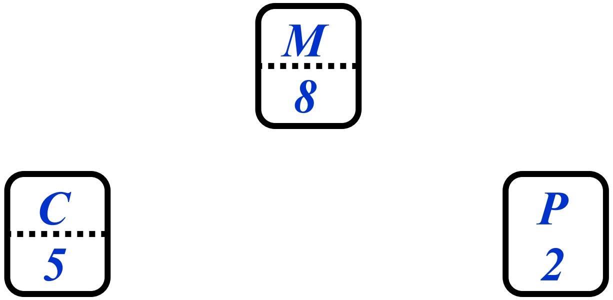
```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

E ample 1:

find rank of element ith ke H

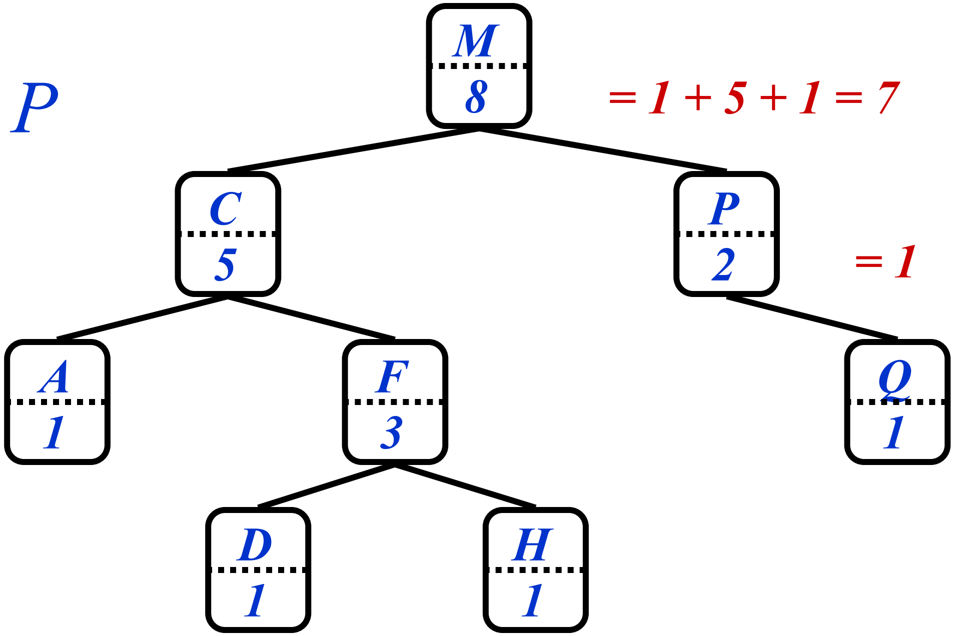


```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```



M
8
C
5
P
2

E ample 2:
find rank of element ith ke P

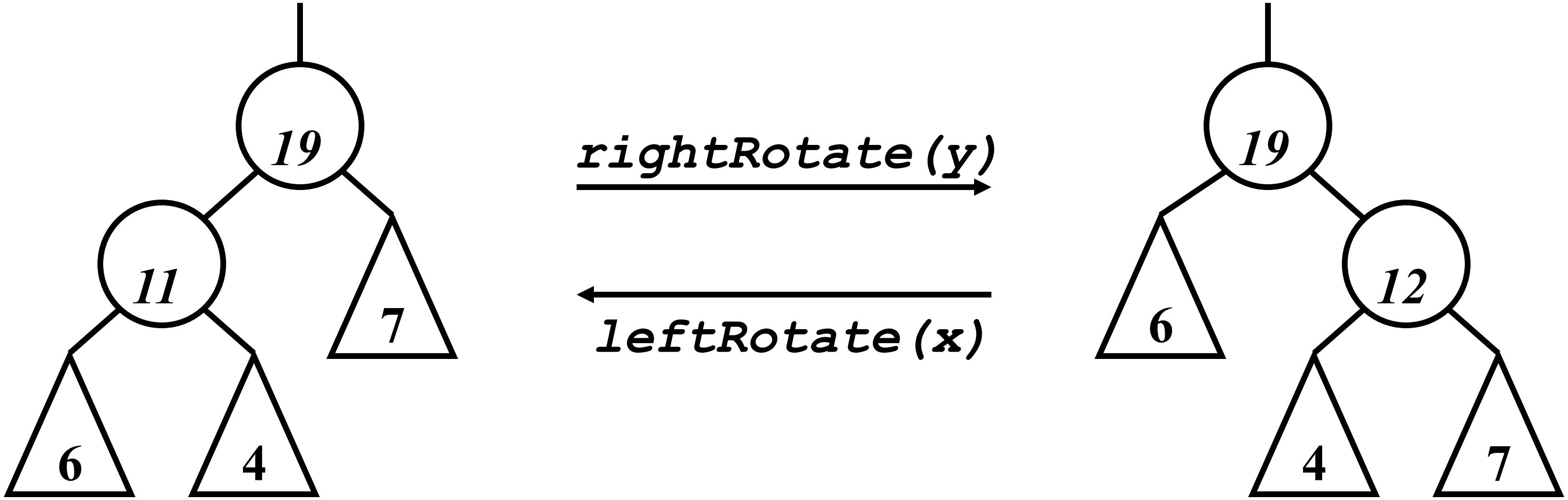


= 1 + 5 + 1 = 7

= 1

```
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;
}
```

- 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



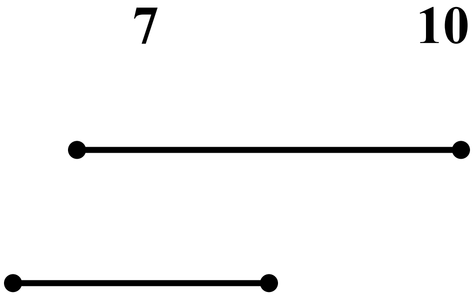
- Salient point: rotation invalidates only   and
- Can recalculate their sizes in constant time
  - *Wh ?*

- 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 Structures: Interval Trees

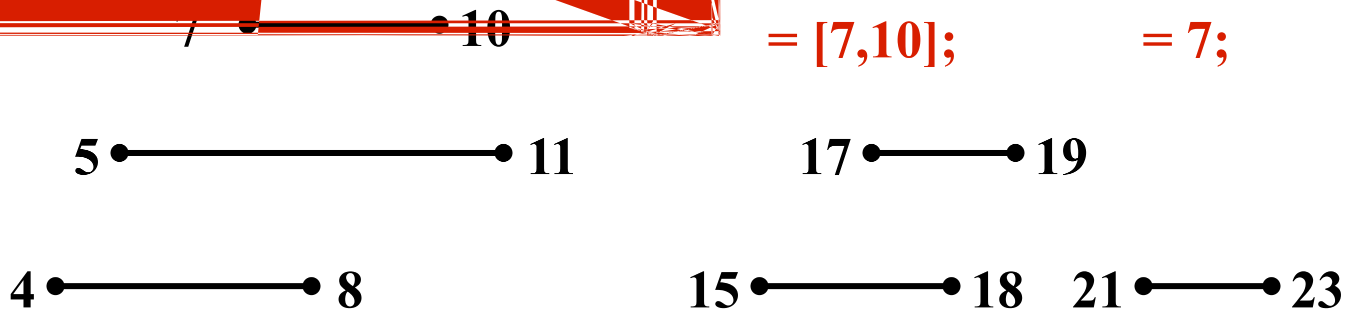
- Choose underlying data structure
- Determine additional information to maintain
- Verify that information can be maintained for operations that modify the structure
- Develop new operations

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



The problem: maintain a set of intervals

- time intervals for a scheduling program:



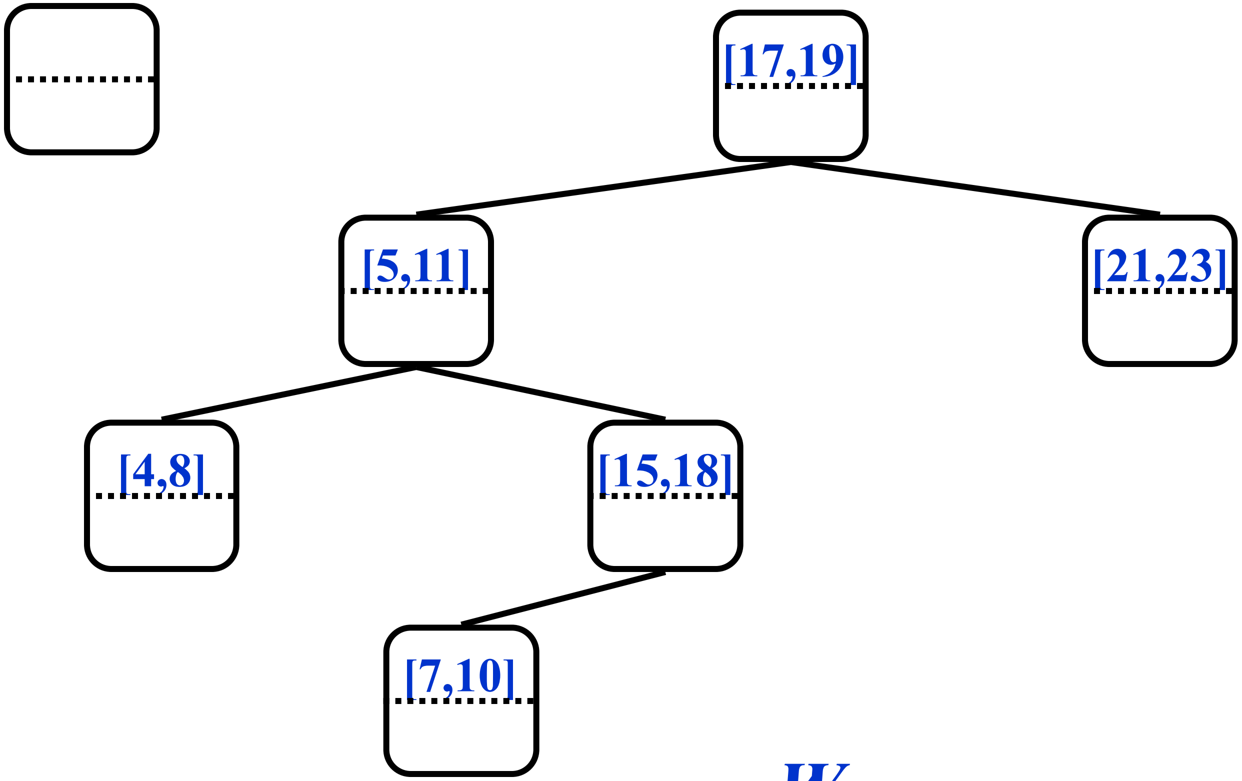
= [7,10]; = 7; = 10

- 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

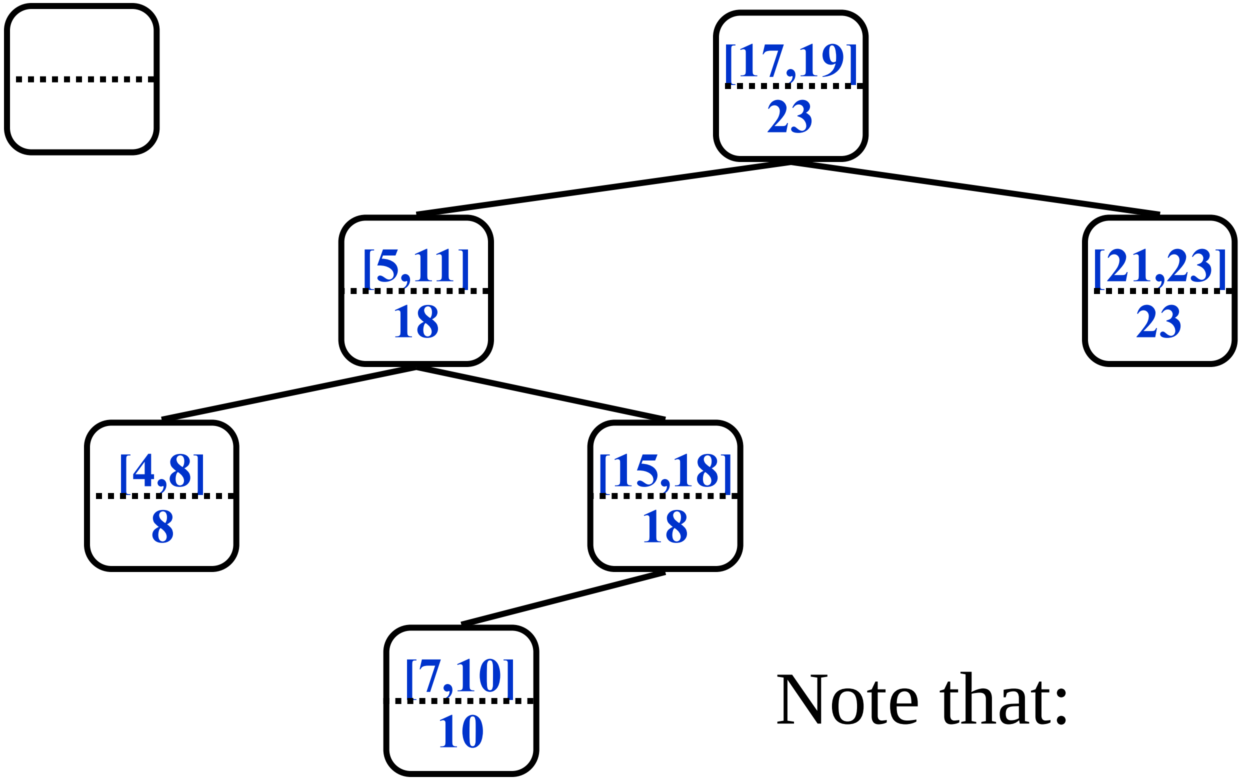
- Following the methodology:
    - Pick underlying data structure
    - Decide what additional information to store
    - Figure out how to maintain the information
    - Develop the desired new operations

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

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



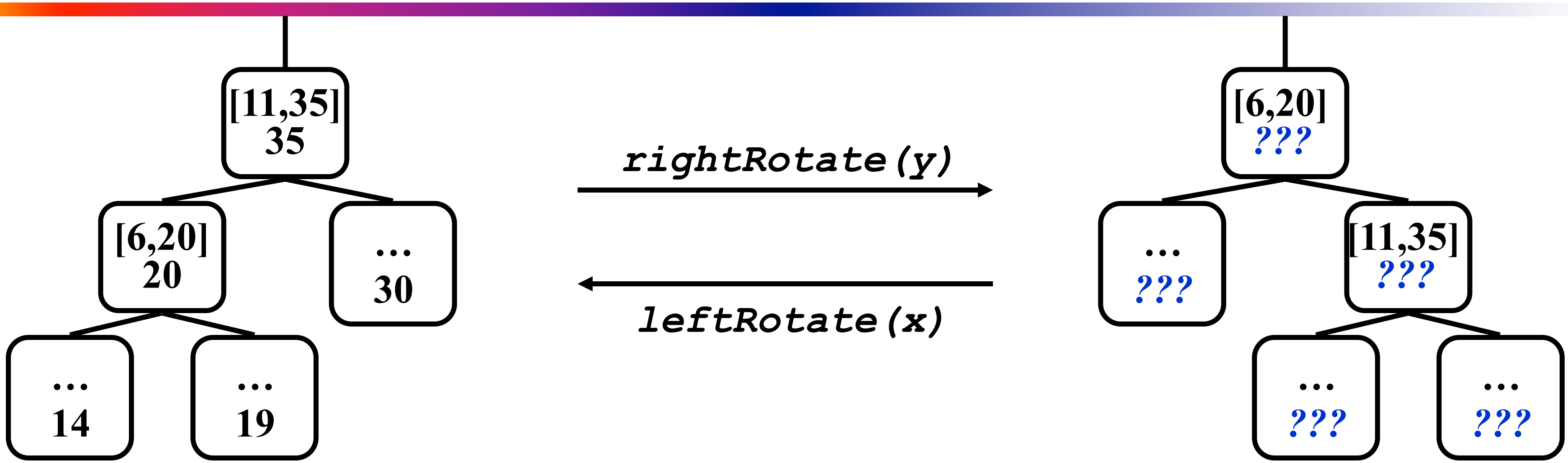
W ?



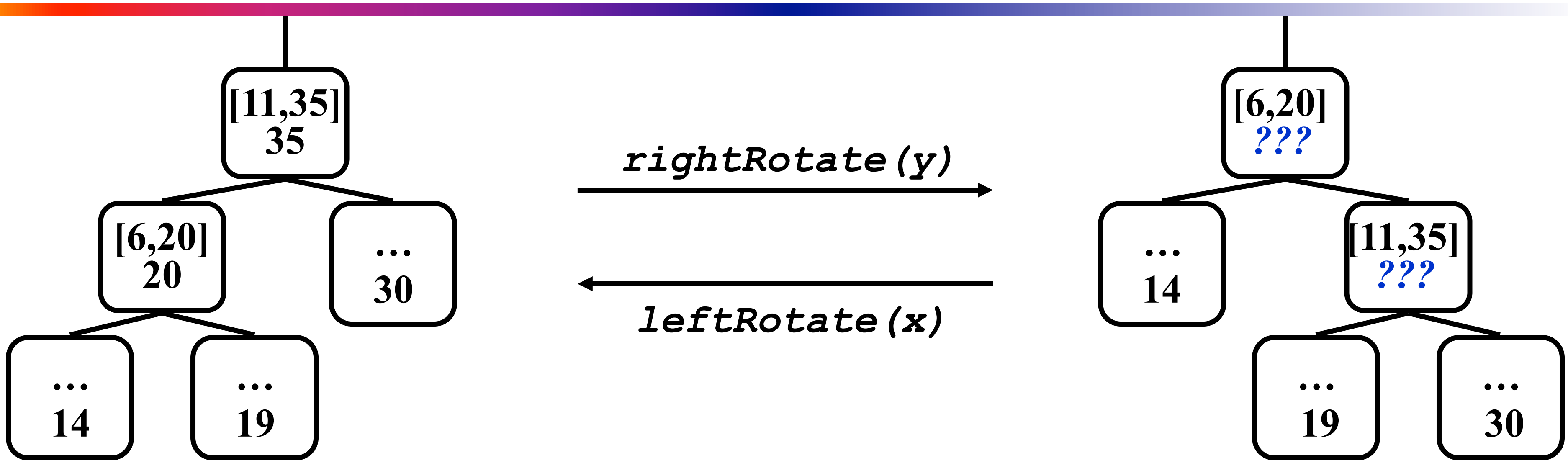
Note that:

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

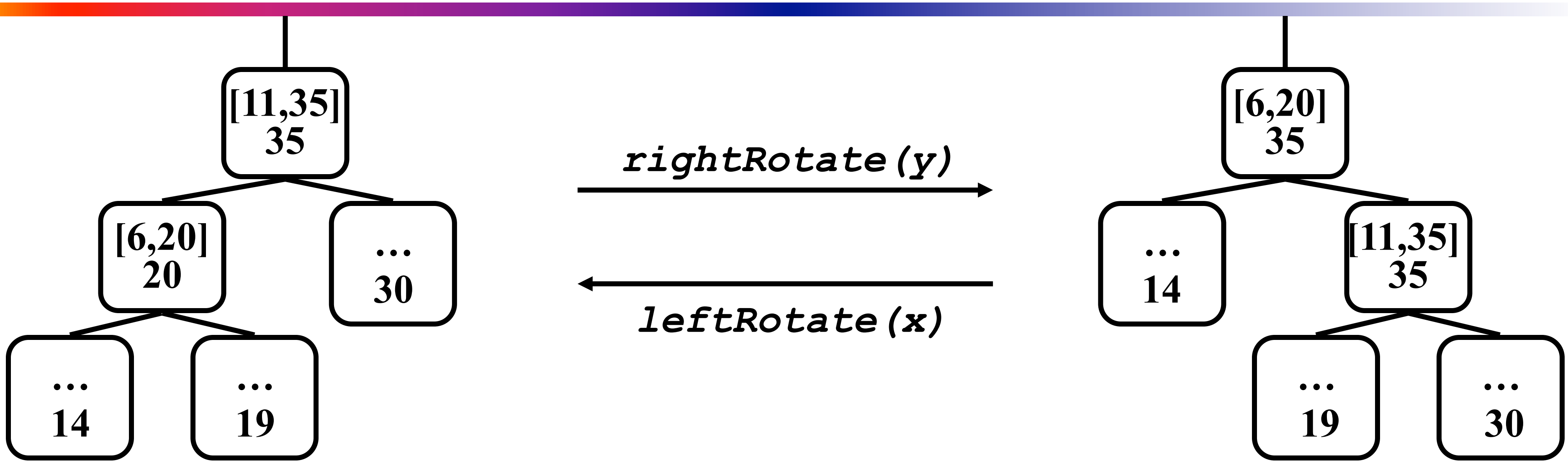
- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on i lo
  - Decide what additional information to store
    - Store the maximum endpoint in the subtree rooted at i
  - *Figure out ho to maintain the information*
    - *Ho ould e maintain ma field for a BST?*
    - *What's different?*
  - Develop the desired new operations



- *What are the ne   ma   values for the subtrees?*



- *What are the ne ma values for the subtrees?*
- A: Unchanged
- *What are the ne ma values for and ?*



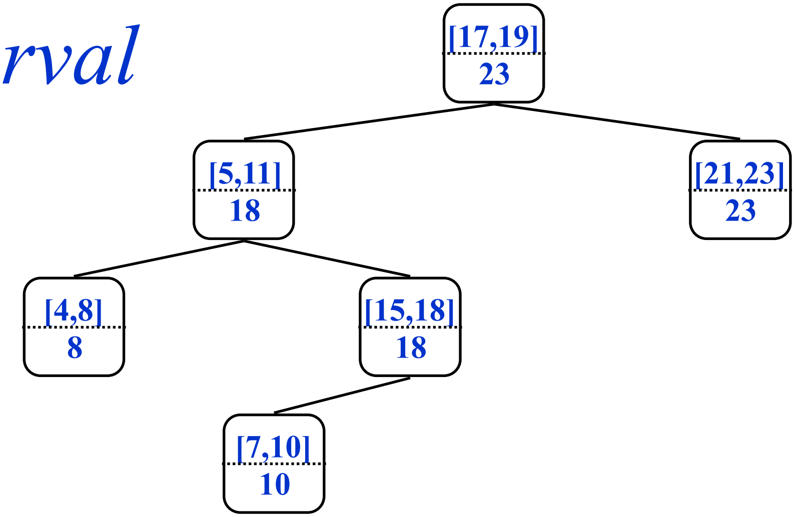
- *What are the ne   ma   values for the subtrees?*
- A: Unchanged
- *What are the ne   ma   values for   and   ?*
- A: root value unchanged, recompute other

- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on *i*   *lo*
  - 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 ne   operations*

```
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
}
```

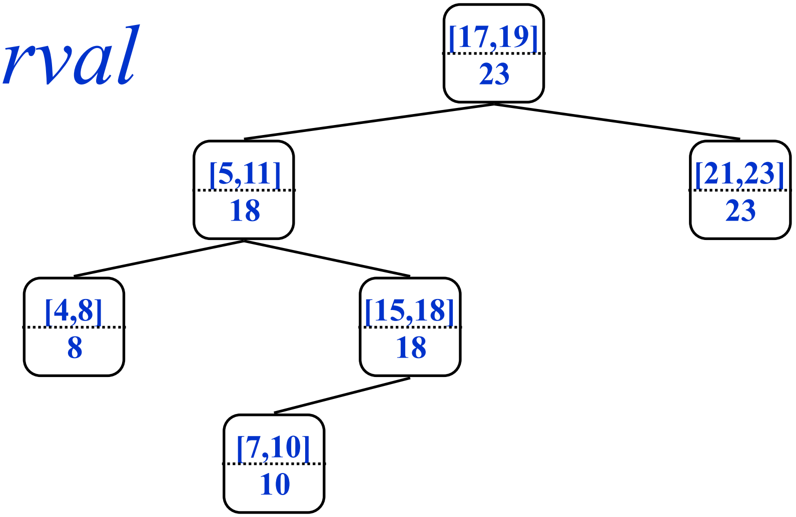
- *What   ill be the running time?*

- *E ample: 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
}
```

- *E ample: 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
}
```

- 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

- 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 < i low (*Wh ?*)
    - 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;
```

- Case 2: if search goes left, ∃ overlap in the left subtree or no overlap in either subtree
  - If ∃ overlap in left subtree, we