Algorithms

Dynamic Order Statistics

Review: Choosing A Hash Function

- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- We discussed three methods:
  - Division method
  - Multiplication method
  - Universal hashing

Review: Universal Hashing

- When attempting to foil an malicious adversary, randomize the algorithm
- *Universal hashing*: pick a hash function randomly when the algorithm begins (*not* upon every insert!)
  - Guarantees good performance on average, no matter what keys adversary chooses
  - Need a family of hash functions to choose from

Review: Universal Hashing

- A family of hash functions ς is said to be *universal* if:
  - With a random hash function from ς, the chance of a collision between x and y is exactly $1/m$ $(x \neq y)$
- We can use this to get good expected performance:
  - Choose h from a universal family of hash functions
  - Hash n keys into a table of m slots, $n \leq m$
  - Then the expected number of collisions involving a particular key x is less than 1

Review: A Universal Hash Function

- Choose table size m to be prime
- Decompose key x into r+1 bytes, so that $x = \{x_0, x_1, \ldots, x_r\}$
  - Only requirement is that max value of byte < m
  - Let $a = \{a_0, a_1, \ldots, a_r\}$ denote a sequence of r+1 elements chosen randomly from $\{0, 1, \ldots, m - 1\}$
  - Define corresponding hash function $h_a \in \varsigma$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

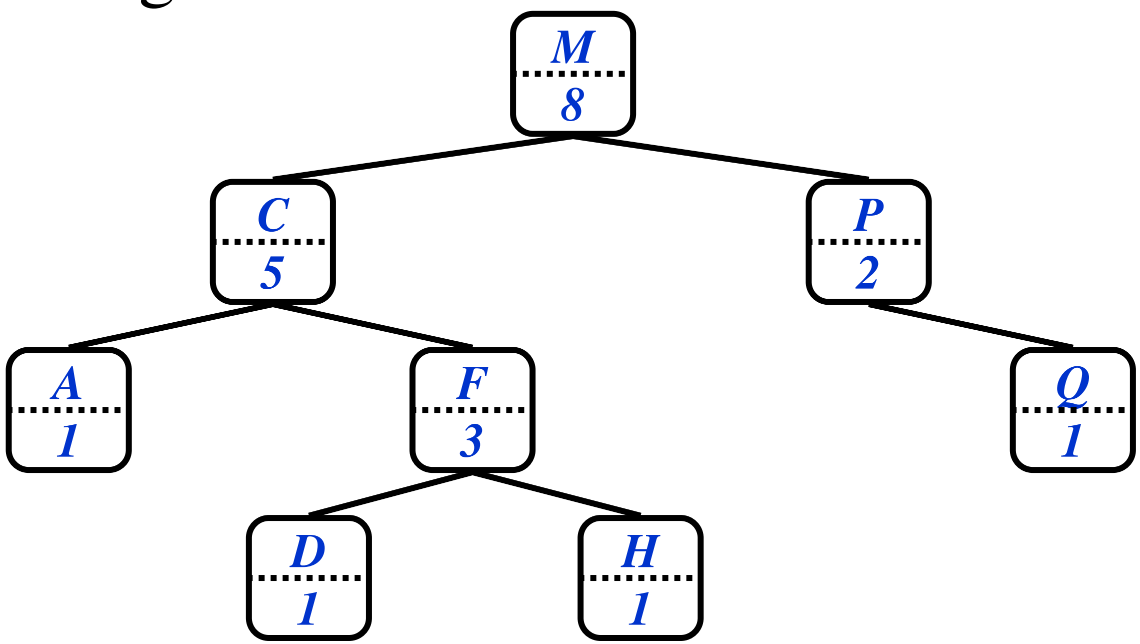
  - With this definition, ς has m^{r+1} members

Review: A Universal Hash Function

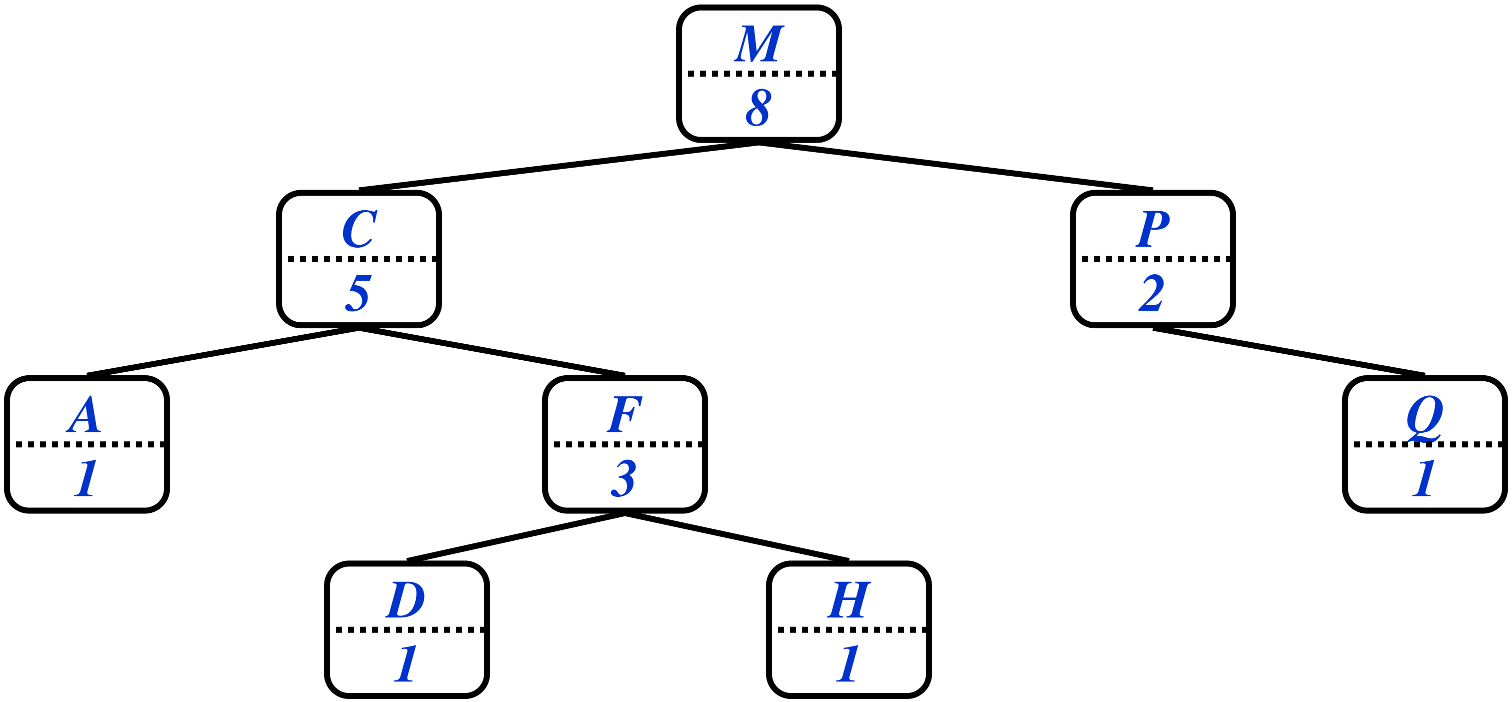
- ς is a universal collection of hash functions (Theorem 12.4)
- How to use:
  - Pick r based on m and the range of keys in U
  - Pick a hash function by (randomly) picking the a's
  - Use that hash function on all keys

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:



Selection On OS Trees



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

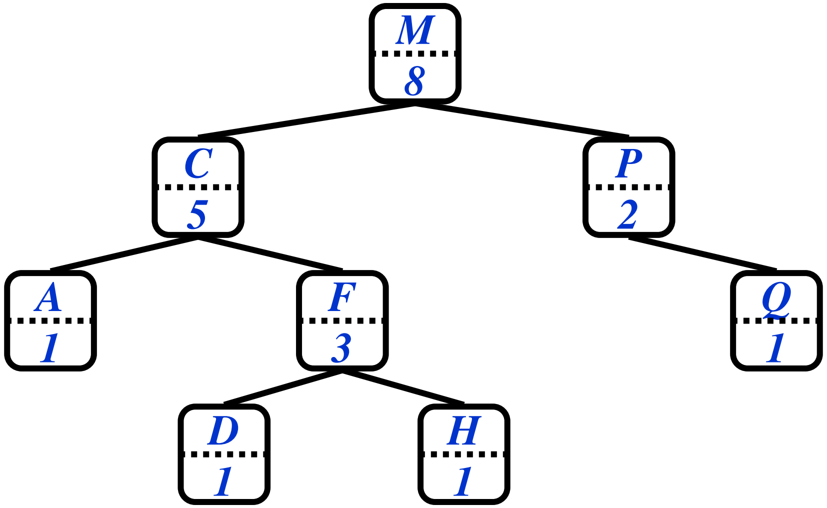
OS-Select

```
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):

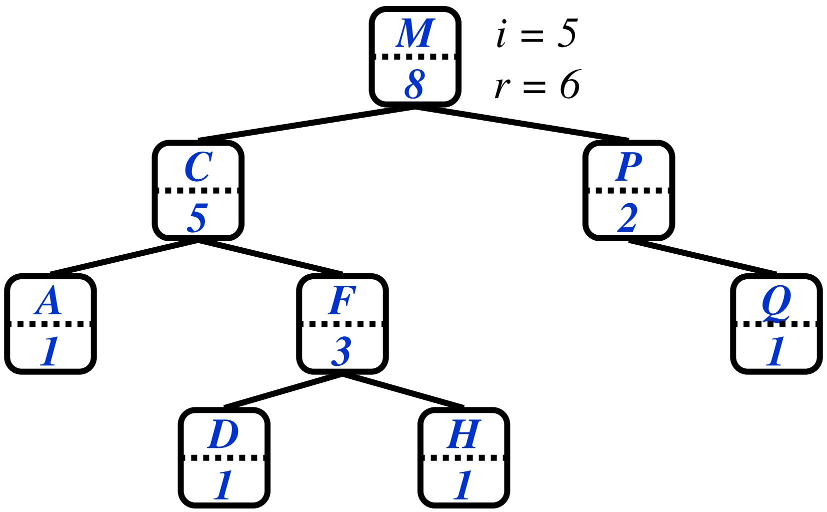
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



OS-Select Example

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

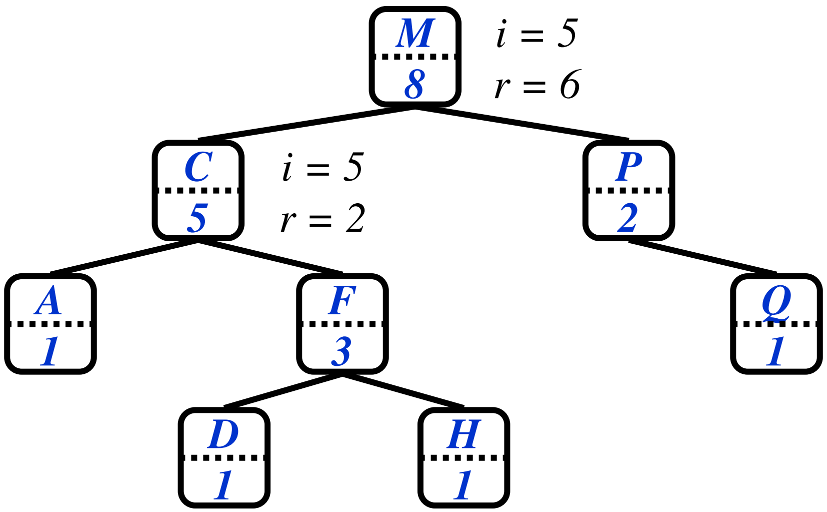
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



OS-Select Example

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

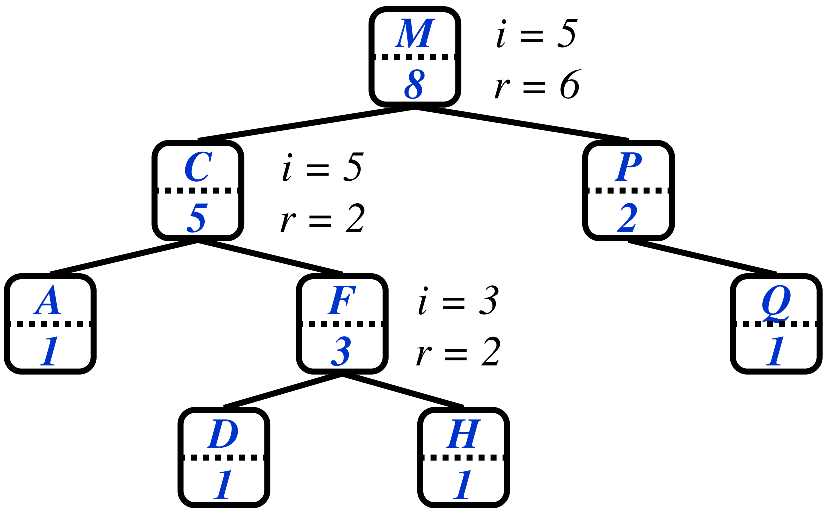
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



OS-Select Example

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

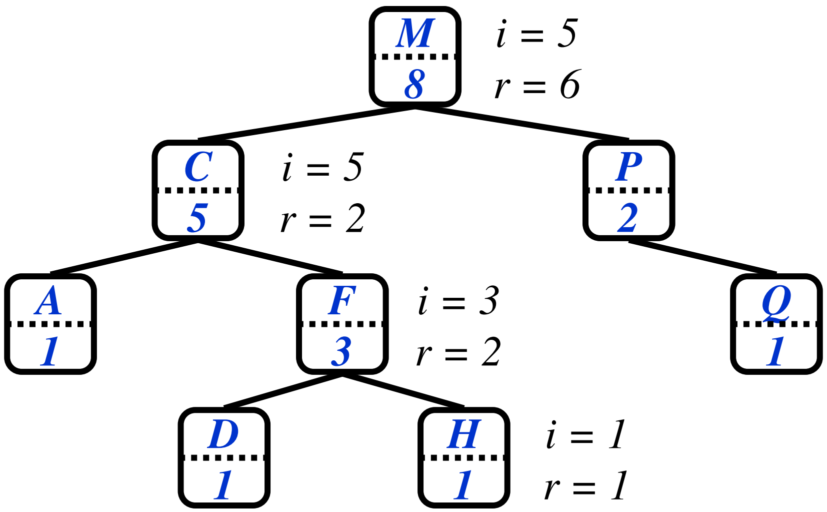
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



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);
}
```

Oops…

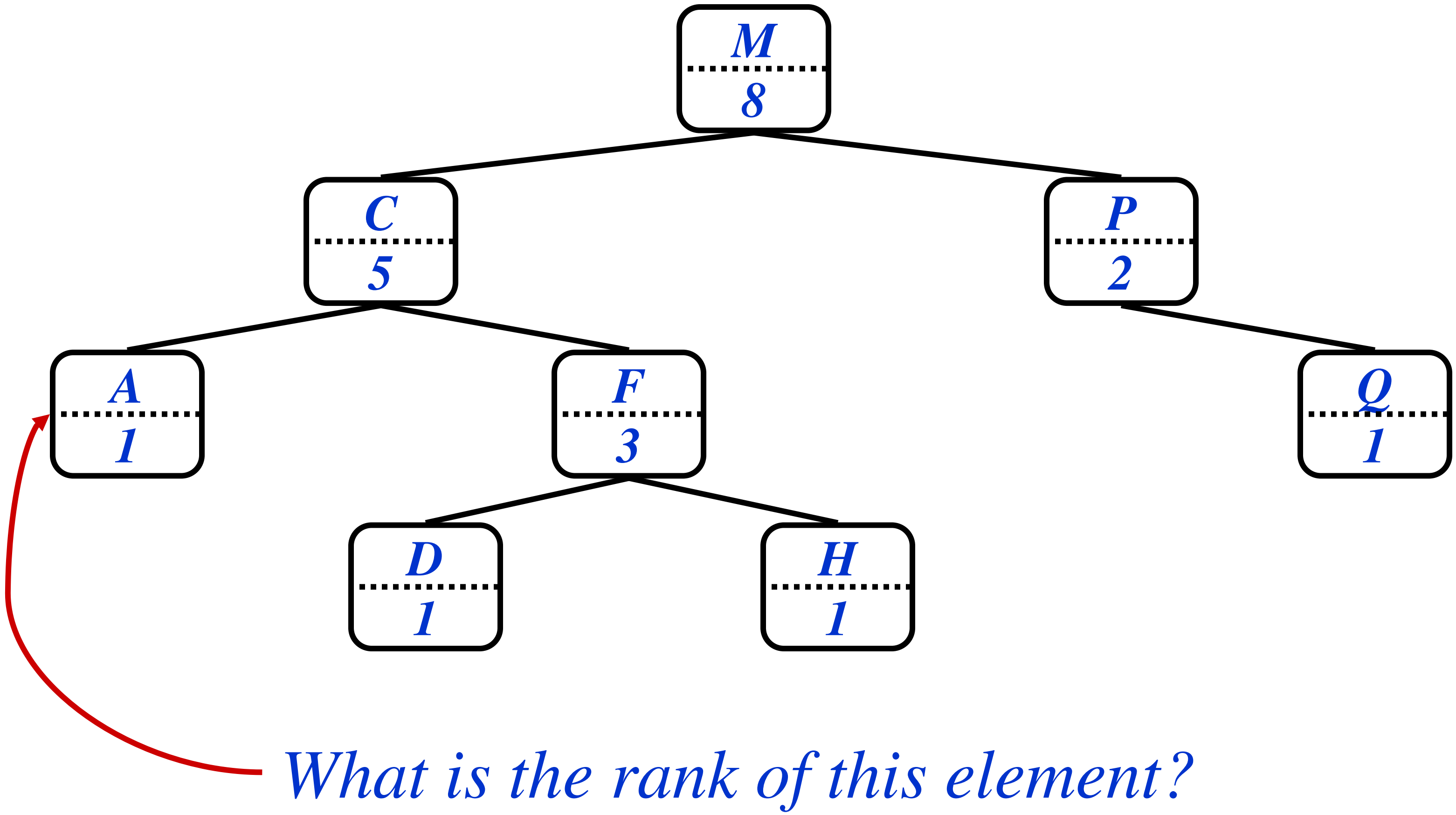
- *What happens at the leaves?*
- *How can we deal elegantly with this?*

OS-Select

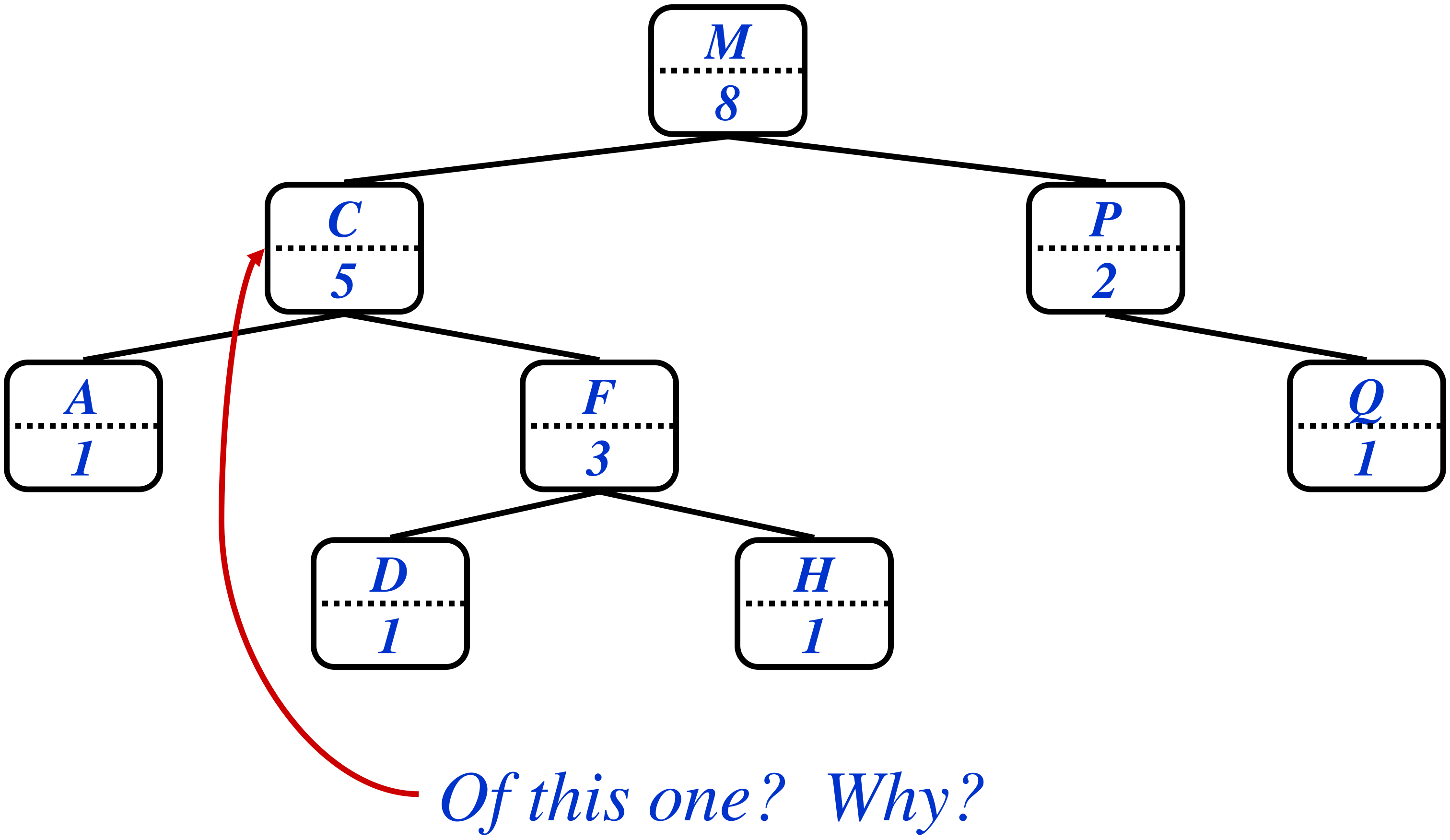
```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```

- *What will be the running time?*

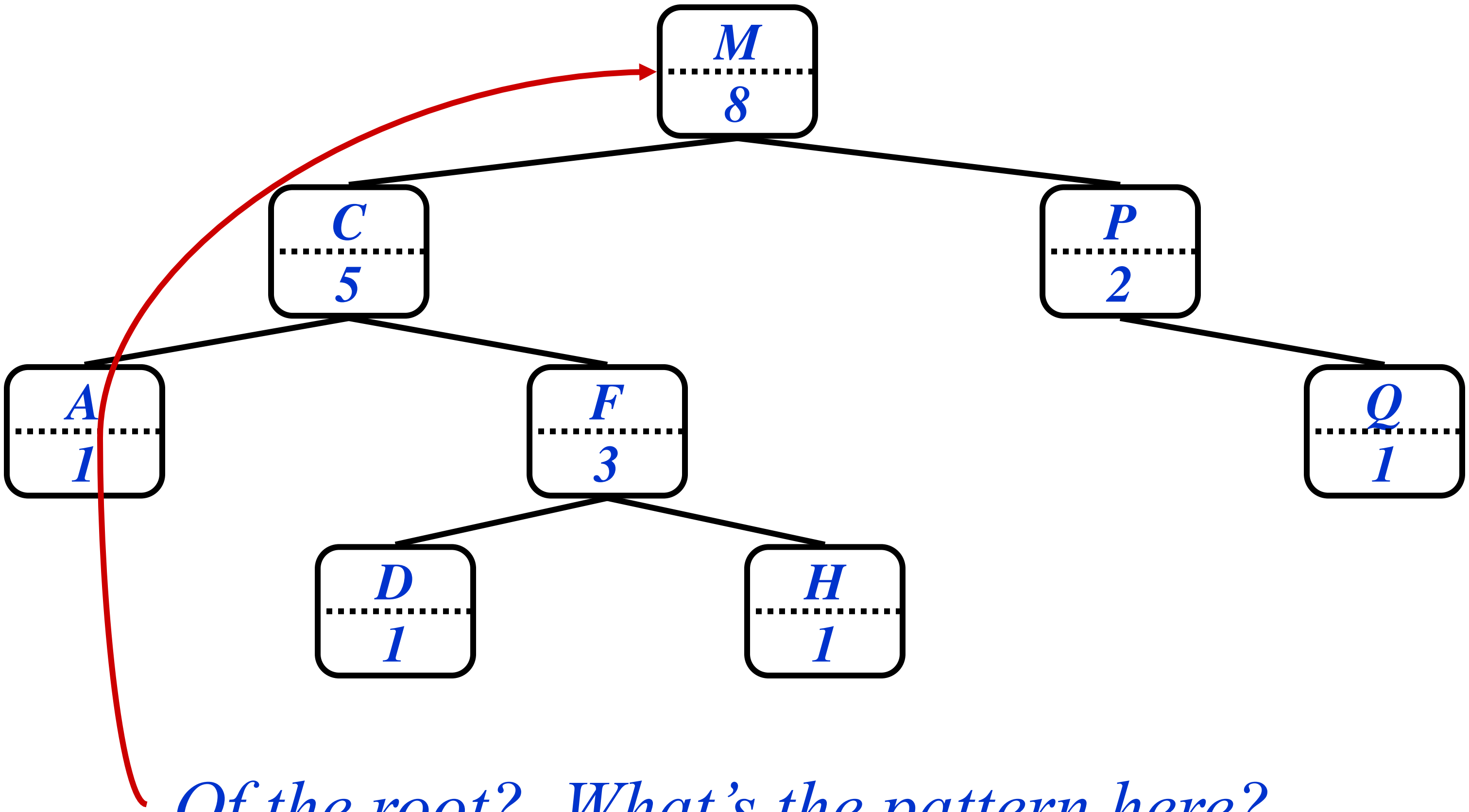
Determining The Rank Of An Element



Determining The Rank Of An Element

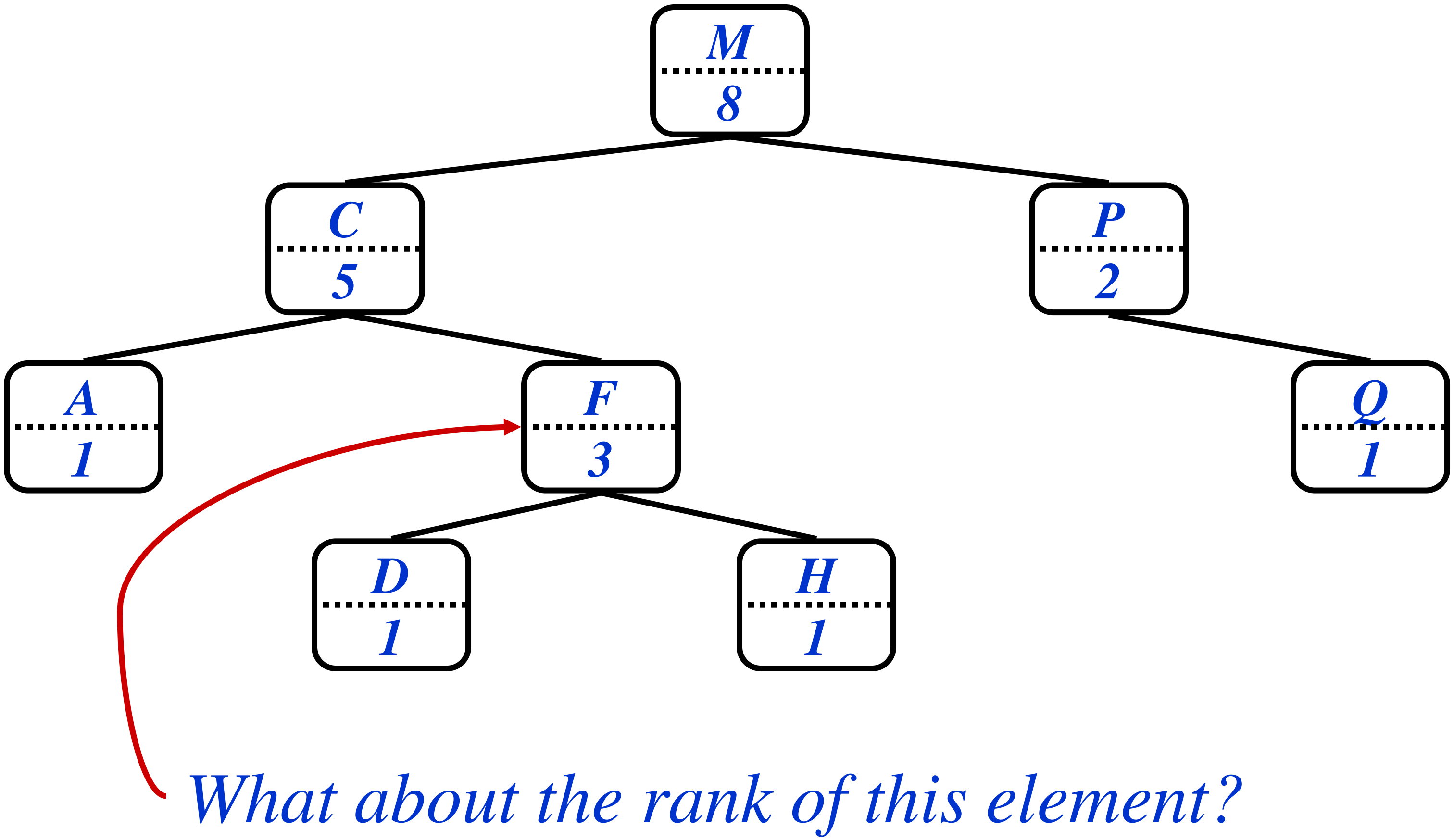


Determining The Rank Of An Element

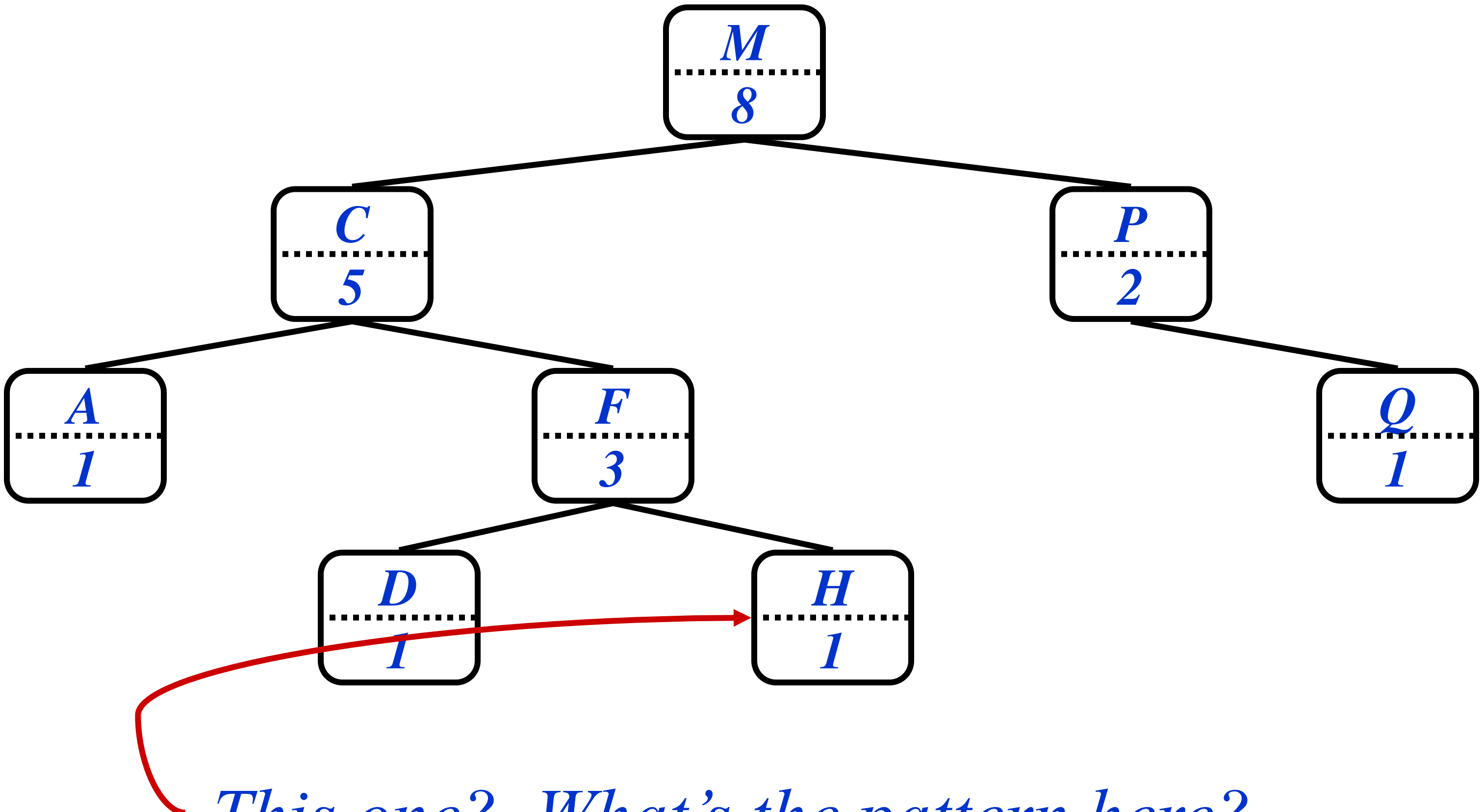


Of the root? What's the pattern here?

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?*

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?*

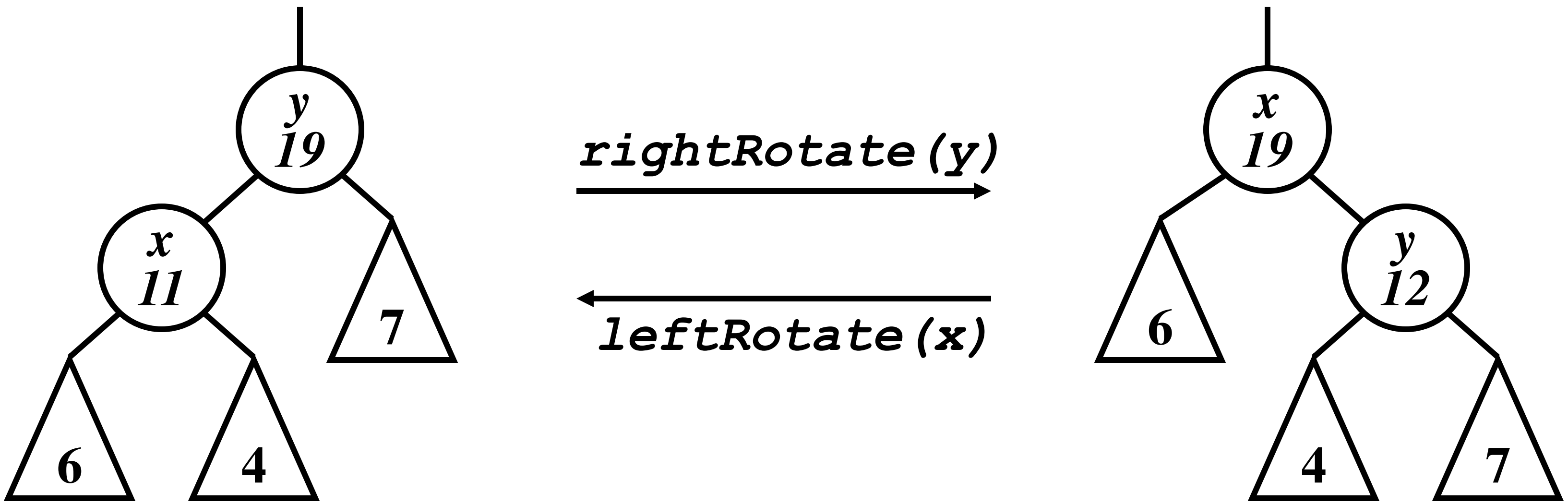
OS-Trees: Maintaining Sizes

- 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()