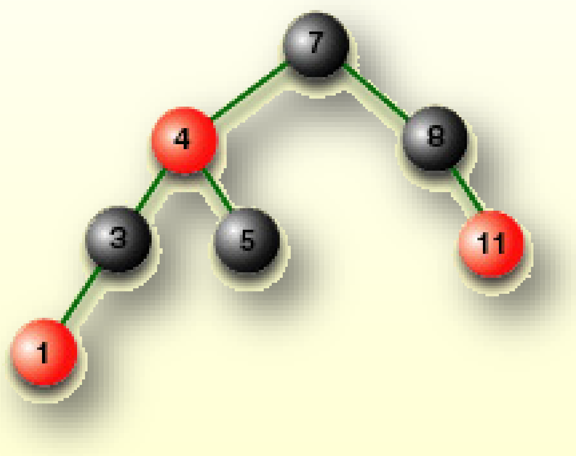
CS161: Design and Analysis of Algorithms



Lecture 12
Leonidas Guibas

Outline

- Review of last lecture: Dynamic Programming
- Today: Augmented data structures
  - dynamic order statistics
  - Interval trees
- Amortized analysis
  - the accounting method
  - the potential method

Slides modified from

- http://www.cs.virginia.edu/~luebke/cs332/
- http://profmsaeed.org/wp-content/uploads/2009/.../AOAAmortizedAnalysis.ppt

Augmenting Data Structures

Dynamic Order Statistics

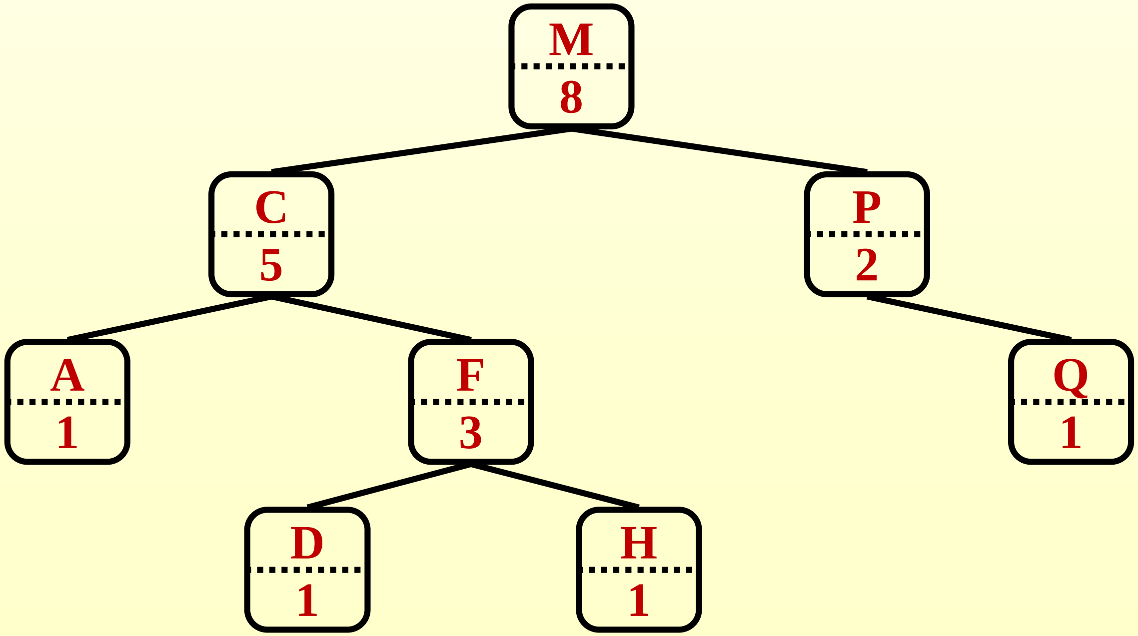
- We have covered algorithms for finding the *i*-th element of a static unordered set in O(*n*) time
- Of course, if a set is ordered, we can find the *i*-th element in O(1) time

- *OS-Trees*: a structure to support finding the *i*-th 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:

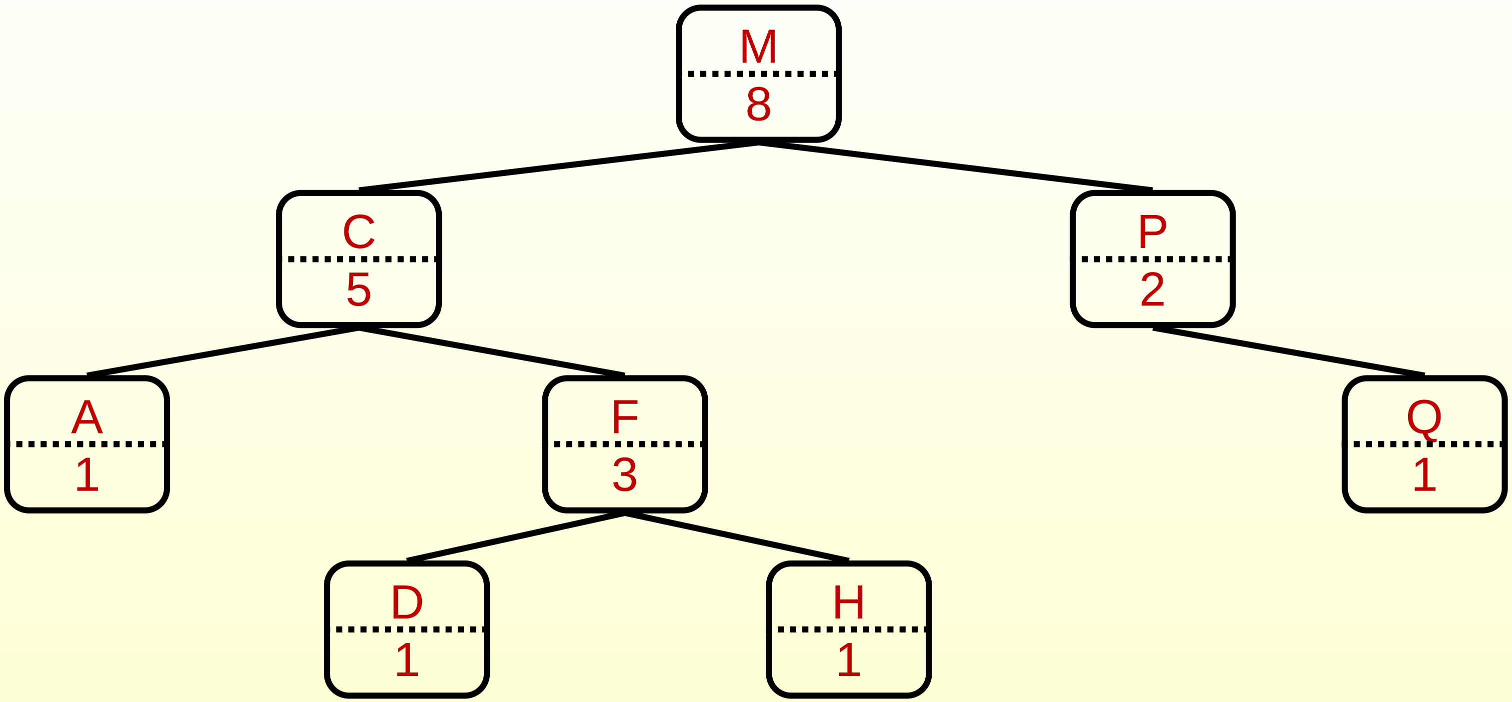
```
void OS-Select(root, i);
int OS-Rank(x);
```

Order Statistics Trees

- OS Trees: augment red-black trees
  - Associate a new size field with each node in the tree
  - **`x->size`** records the size of subtree rooted at **`x`**, including **`x`** itself:



Selection in 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);
}
```

compute rank r of the root

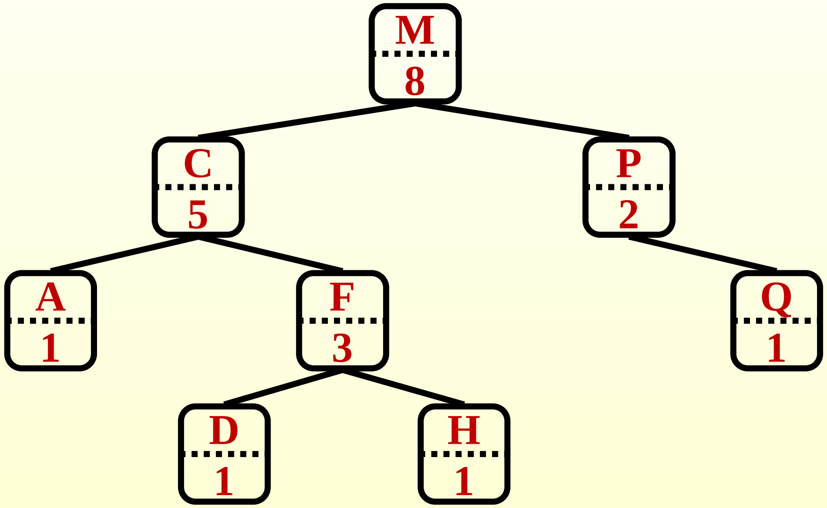
go left

go right

OS-Select Example

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

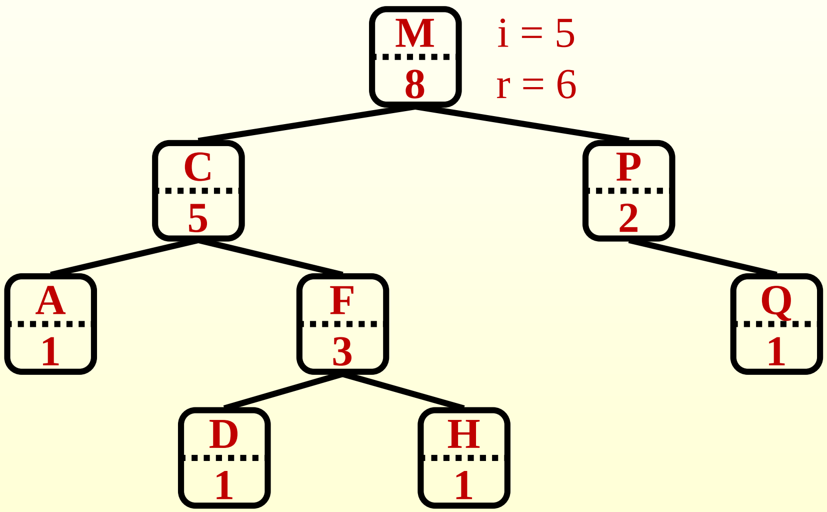
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



OS-Select Example

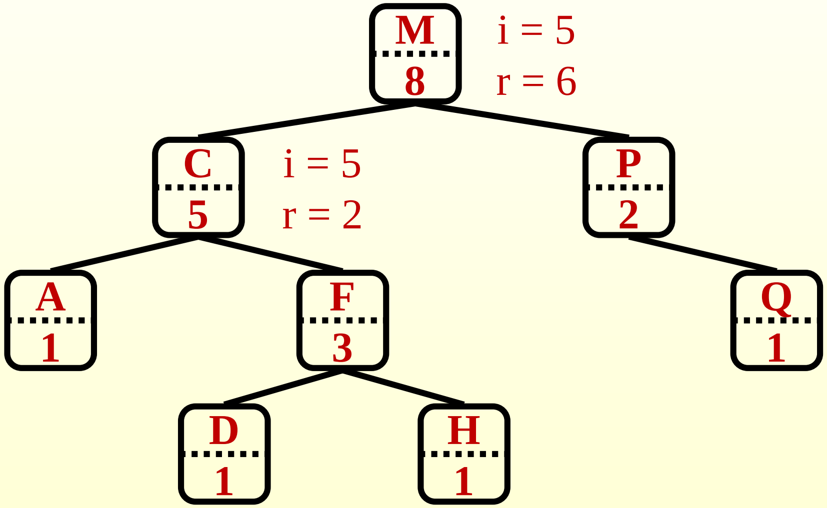
- Example: show OS-Select(*root*, 5):

```
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

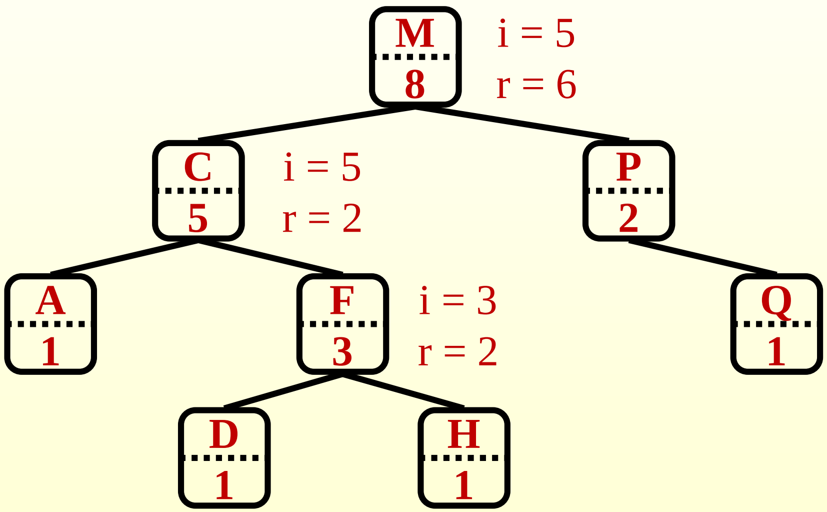
- Example: show OS-Select(*root*, 5):




```
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):

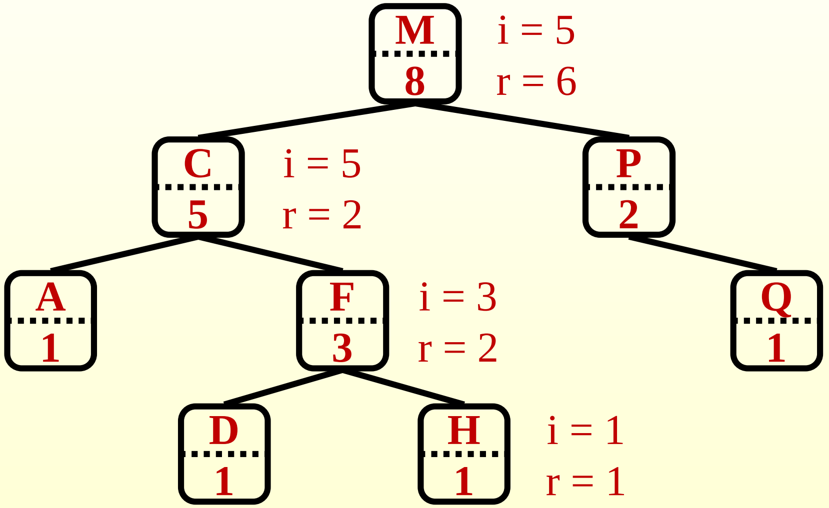



```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```

OS-Select Example

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

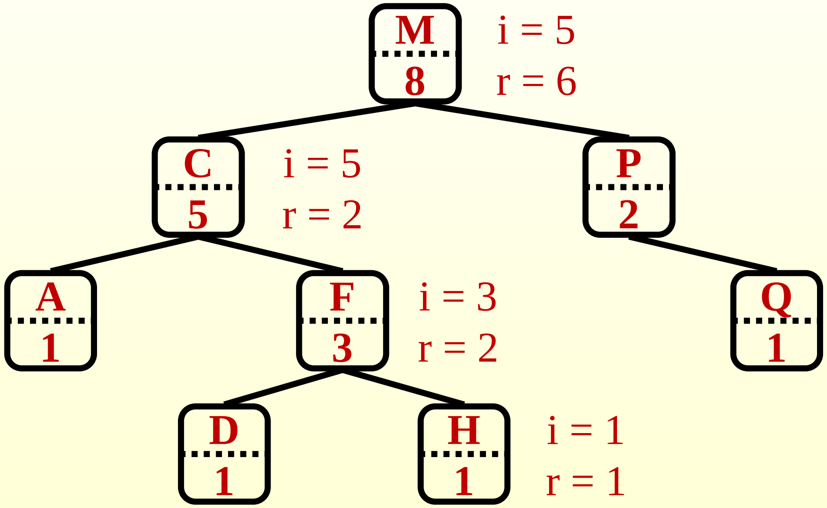
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



OS-Select 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);
}
```



Note: use a sentinel NIL element at the leaves with size = 0 to simplify code, avoid testing for NULL

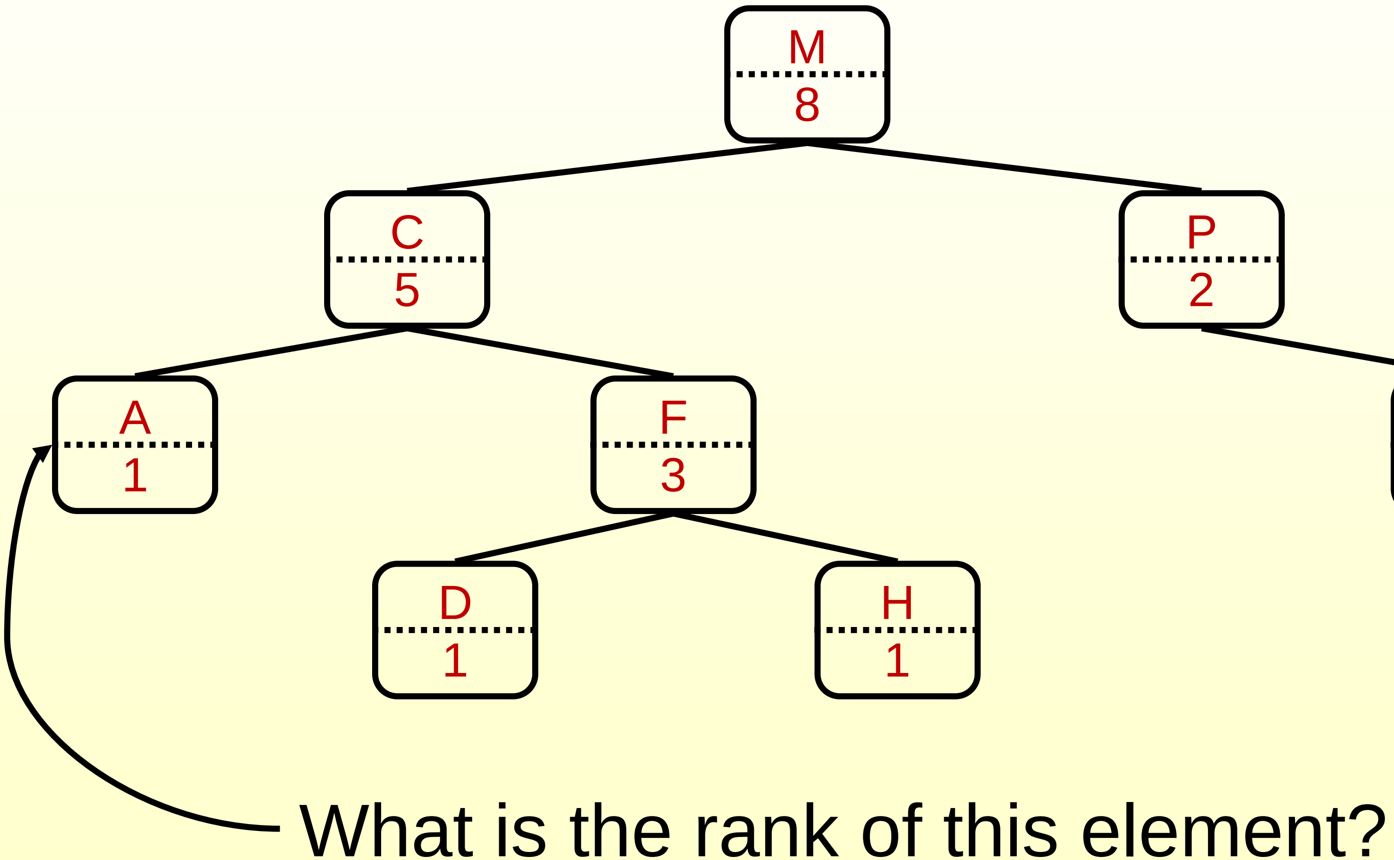
OS-Select

```
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);
}
```

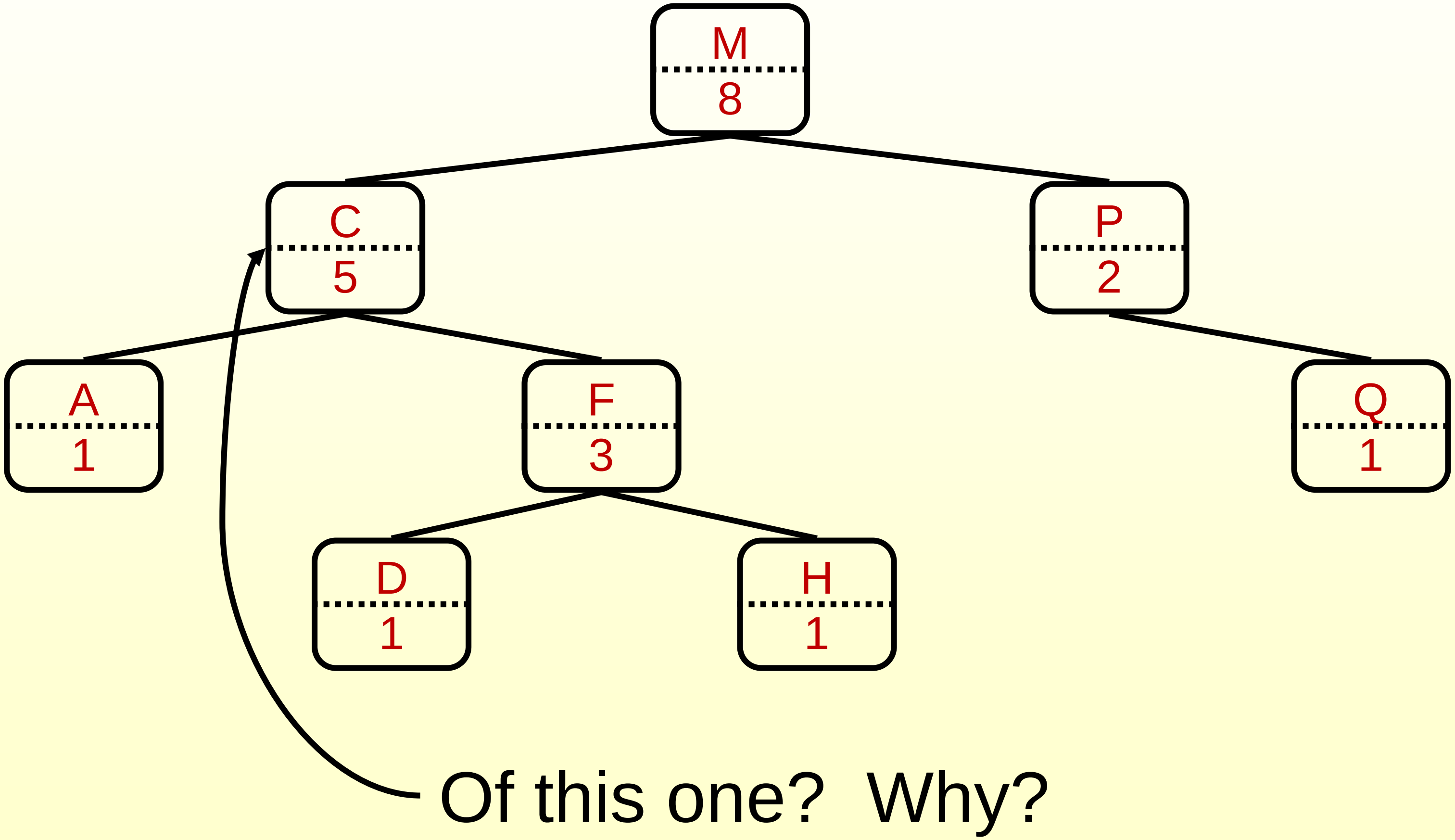
- *What is the running time?*

O(log n)

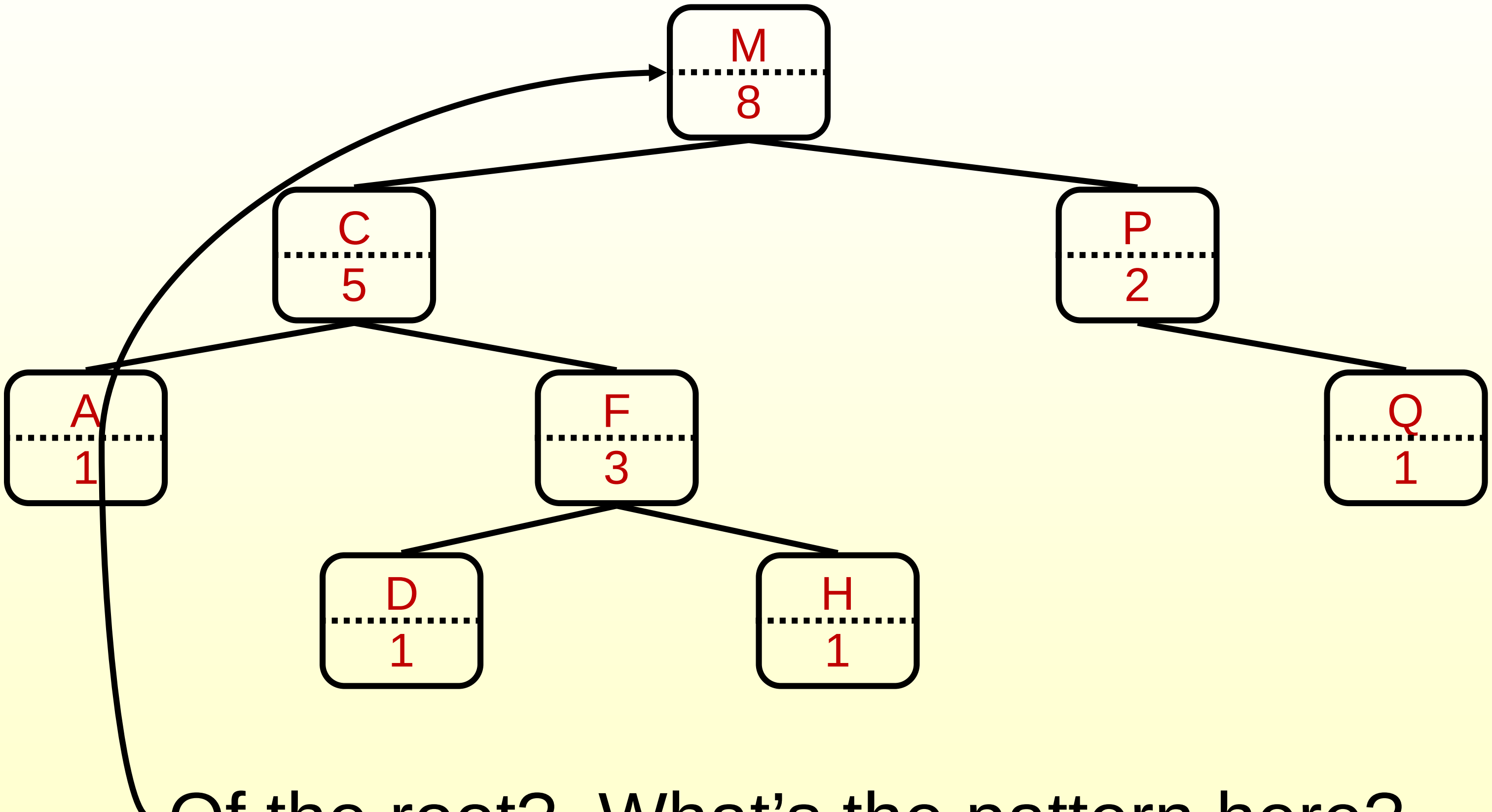
Determining The Rank of an Element



Determining The Rank of an Element

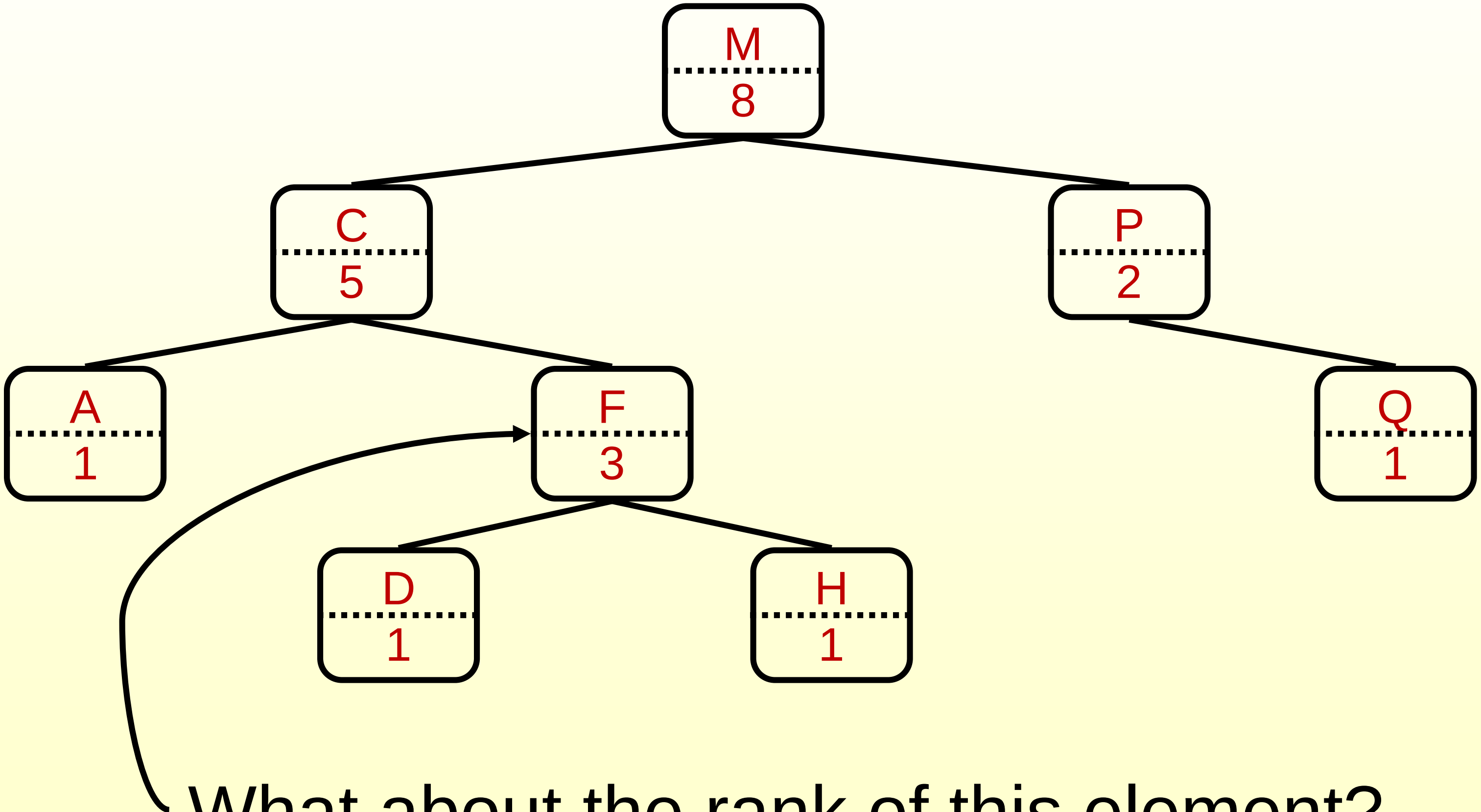


Determining The Rank of an Element



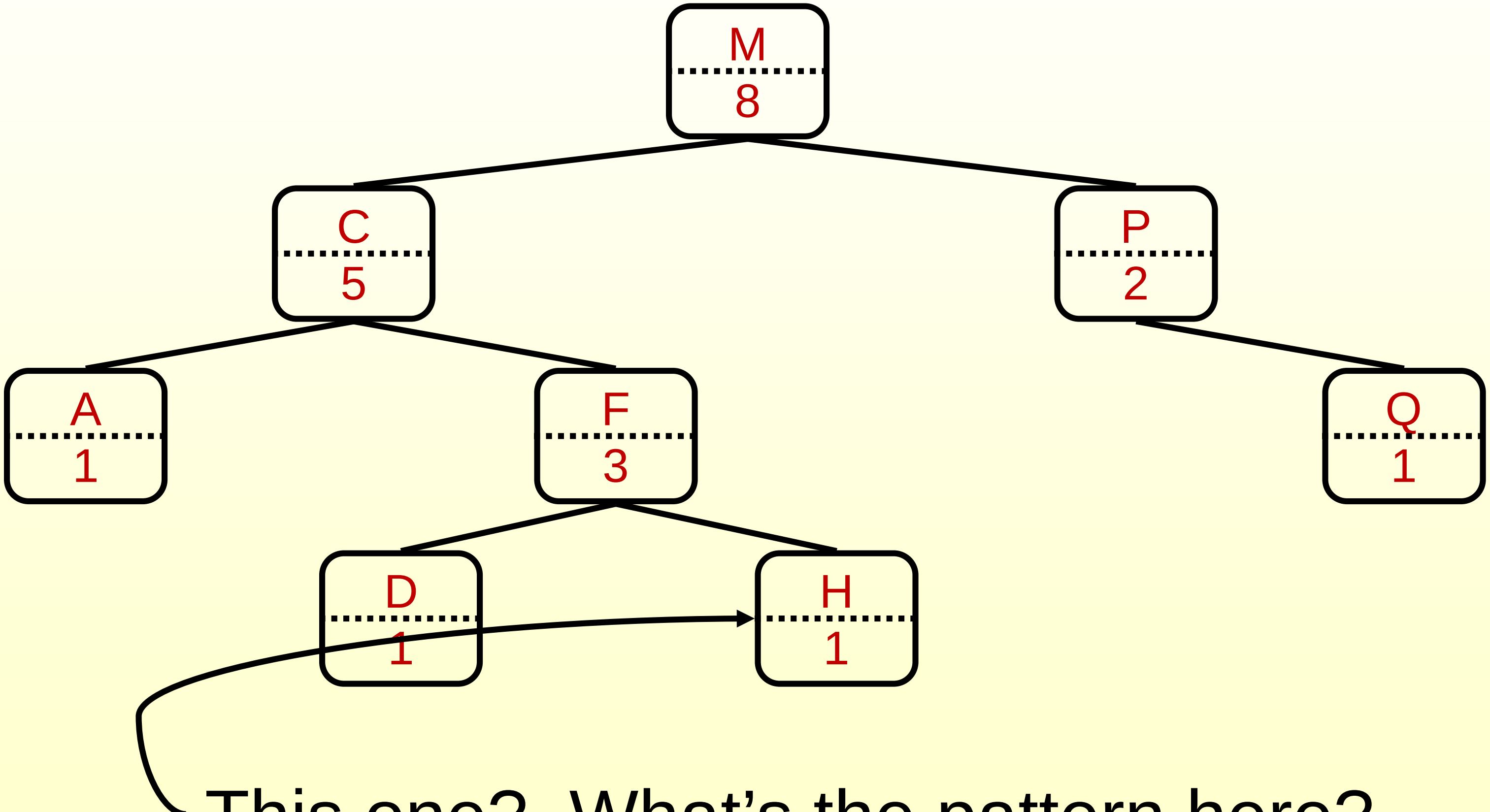
Of the root? What's the pattern here?

Determining The Rank of an Element



What about the rank of this element?

Determining The Rank of an Element



This one? What's the pattern here?

OS-Rank

Idea: rank of right child x is one more than its parent's rank, plus the size of x's left subtree

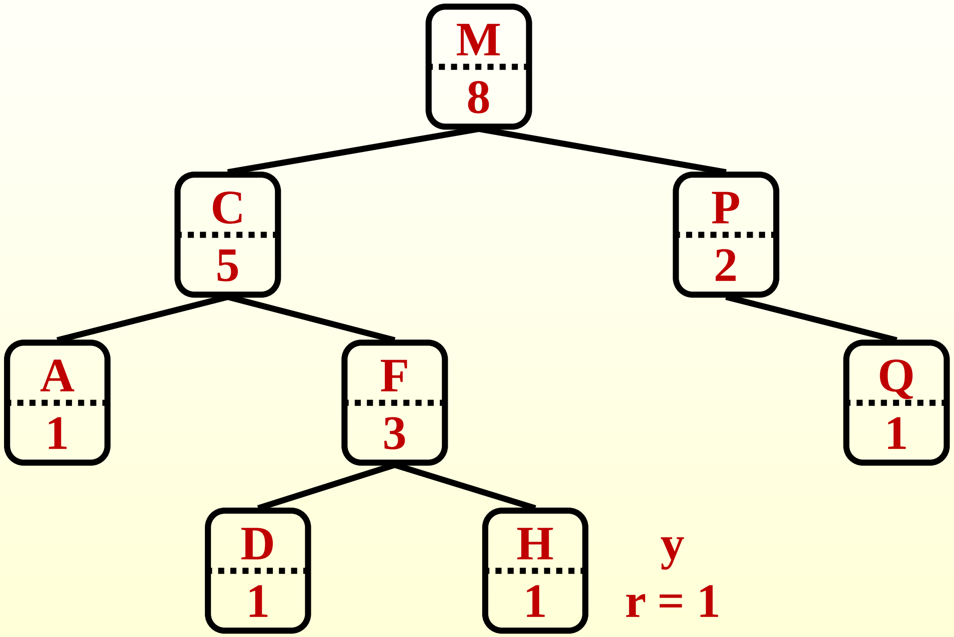
```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

- *What is the running time?*

O(log n)

Determining The Rank of an Element

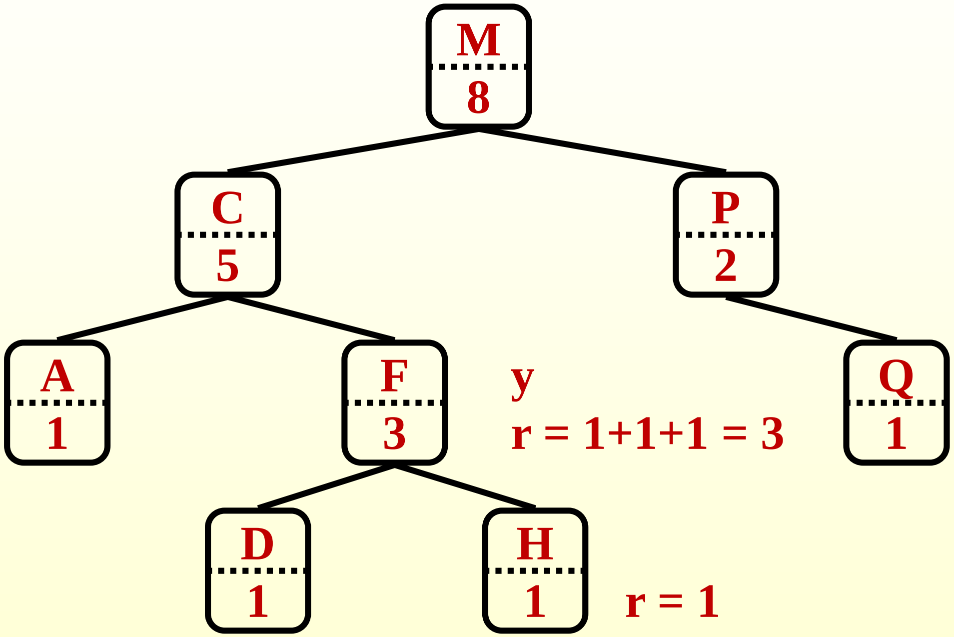
Example 1:
find rank of element with key H




```
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

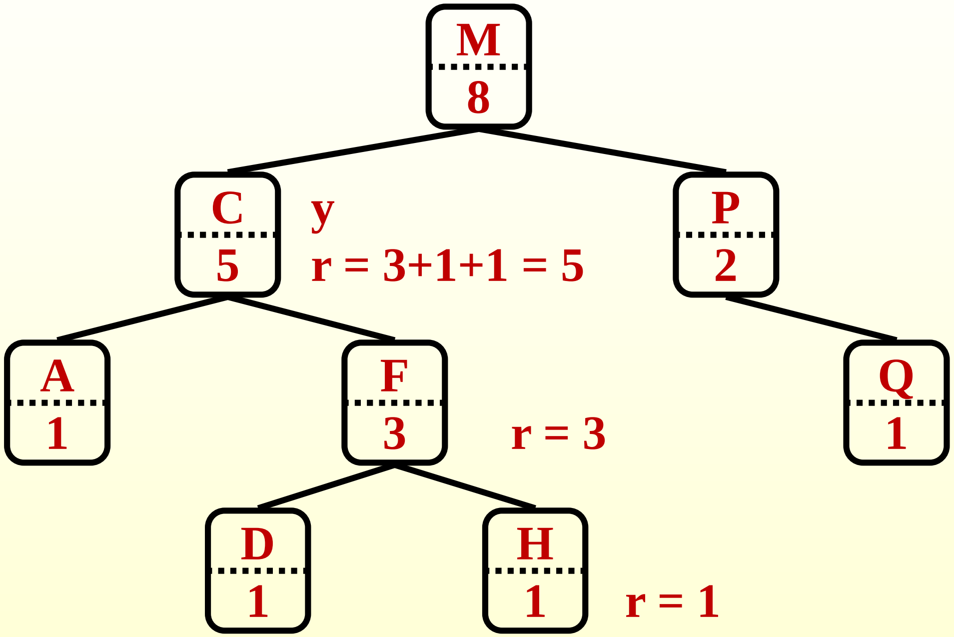
Example 1:
find rank of element with key H




```
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

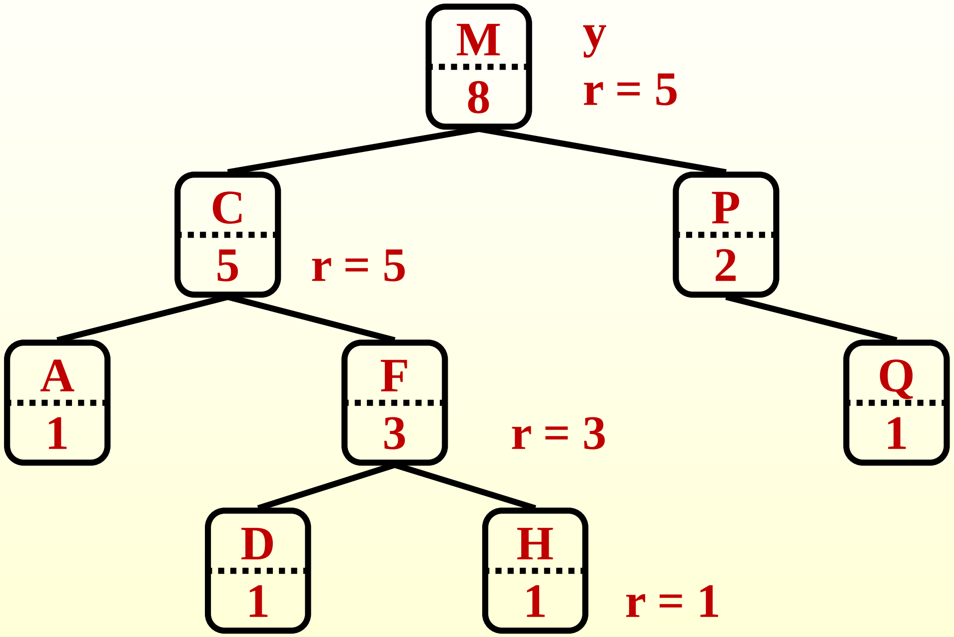
Example 1:
find rank of element with key H




```
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

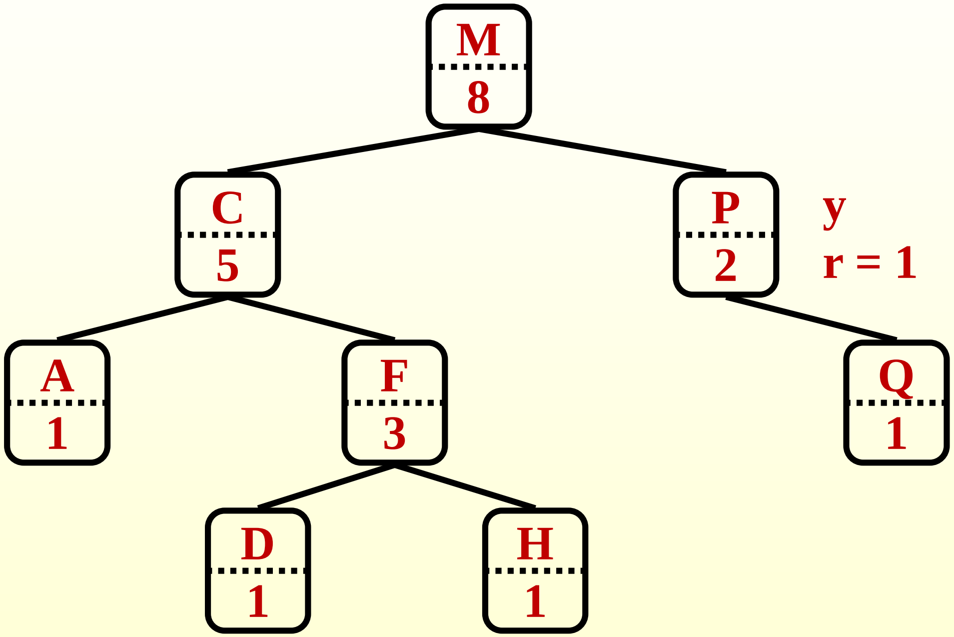



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

Review: Determining The Rank of an Element

Example 2:

find rank of element with key P

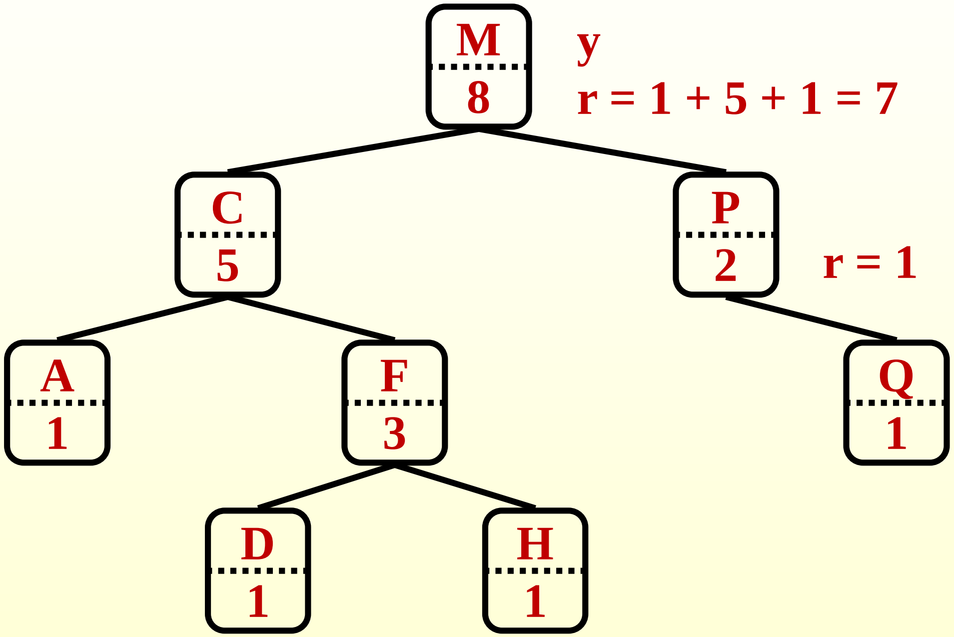



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

Review: 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;
}
```

OS-Trees: Maintaining Sizes

- We have shown that, with subtree sizes, order statistic operations can be done in O(log n) time
- Next step: maintain sizes during Insert() and Delete() operations
  - *How should we adjust the size fields during insertion on a plain binary search tree?*

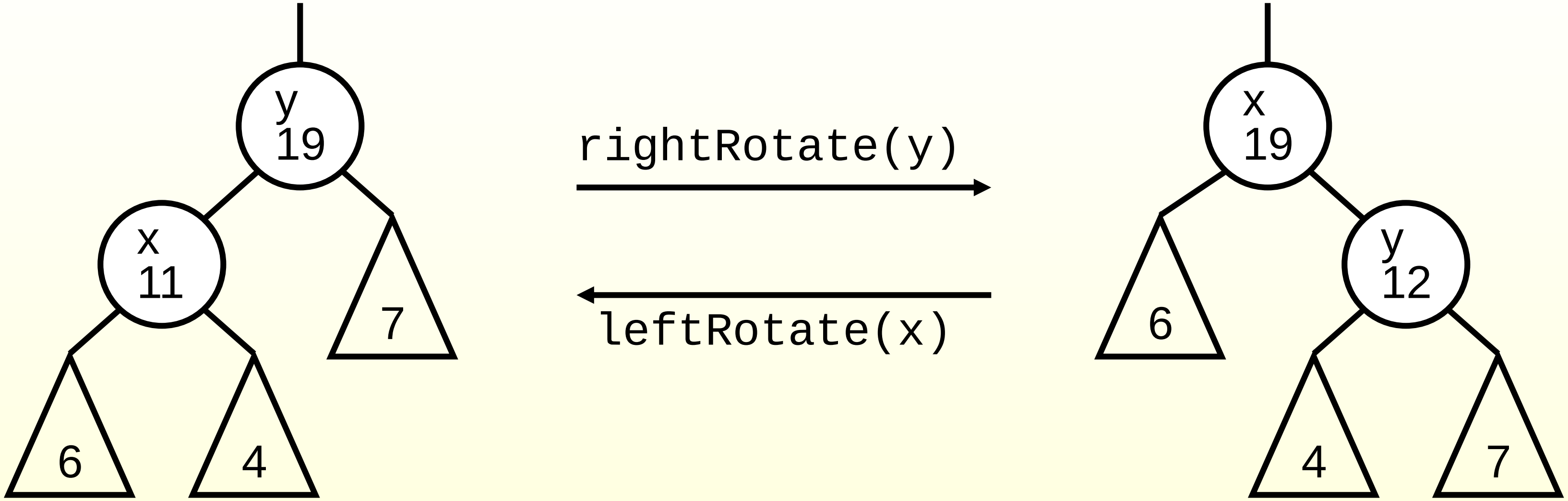
OS-Trees: Maintaining Sizes

- We have shown that, with subtree sizes, order statistic operations can be done in O(log 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: in insertion, increment sizes of nodes traversed during unsuccessful search

OS-Trees: Maintaining Sizes

- We have shown that, with subtree sizes, order statistic operations can be done in O(log 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: in insertion, increment sizes of nodes traversed during unsuccessful search
  - *Why won't this work on red-black trees?*

Maintaining Sizes Through Rotations



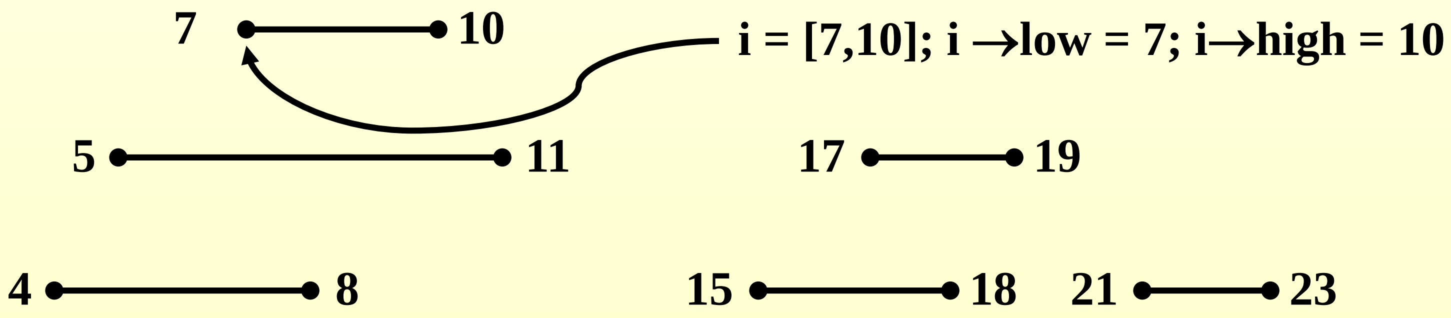
- Salient point: rotation invalidates only *x* and *y*
- Can recalculate their sizes in constant time
  - *Why?*

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()

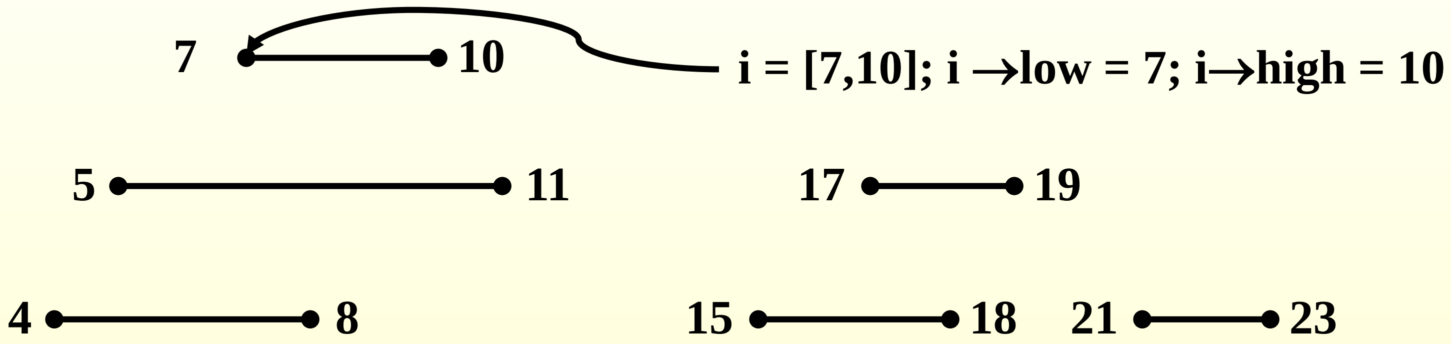
Interval Trees

- 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 (conflict detection)
    - [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

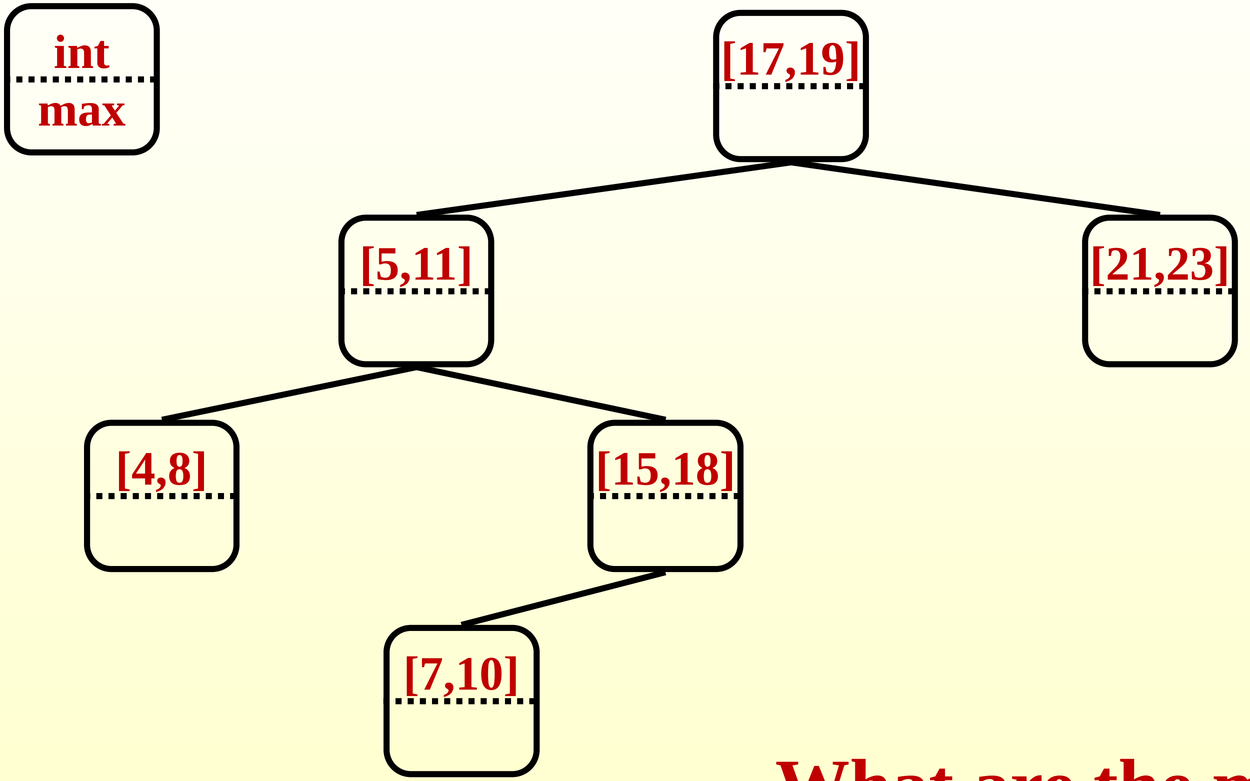
Interval Trees

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

Interval Trees

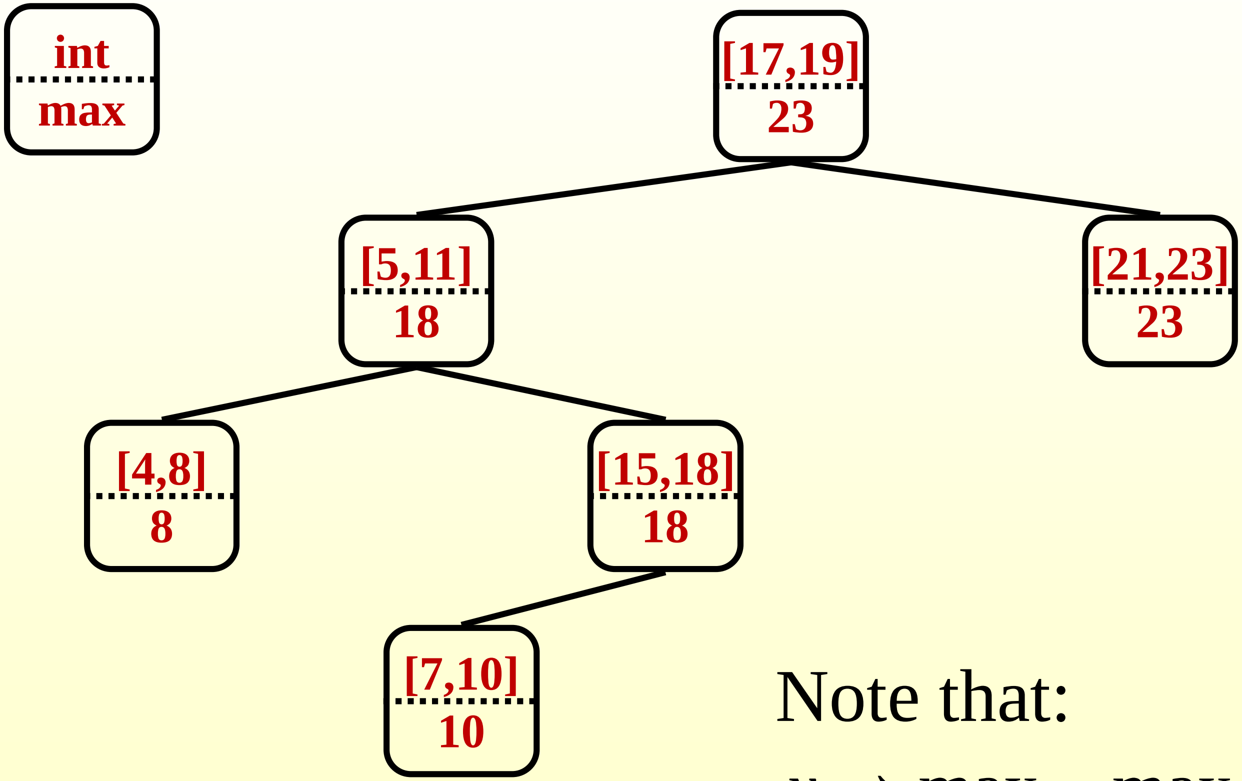
- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$ (the left endpoint)
  - *Decide what additional information to store*
    - We will store *max*, the maximum right endpoint in the subtree rooted at each node
  - Figure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

Interval Trees



What are the max fields?

Interval Trees



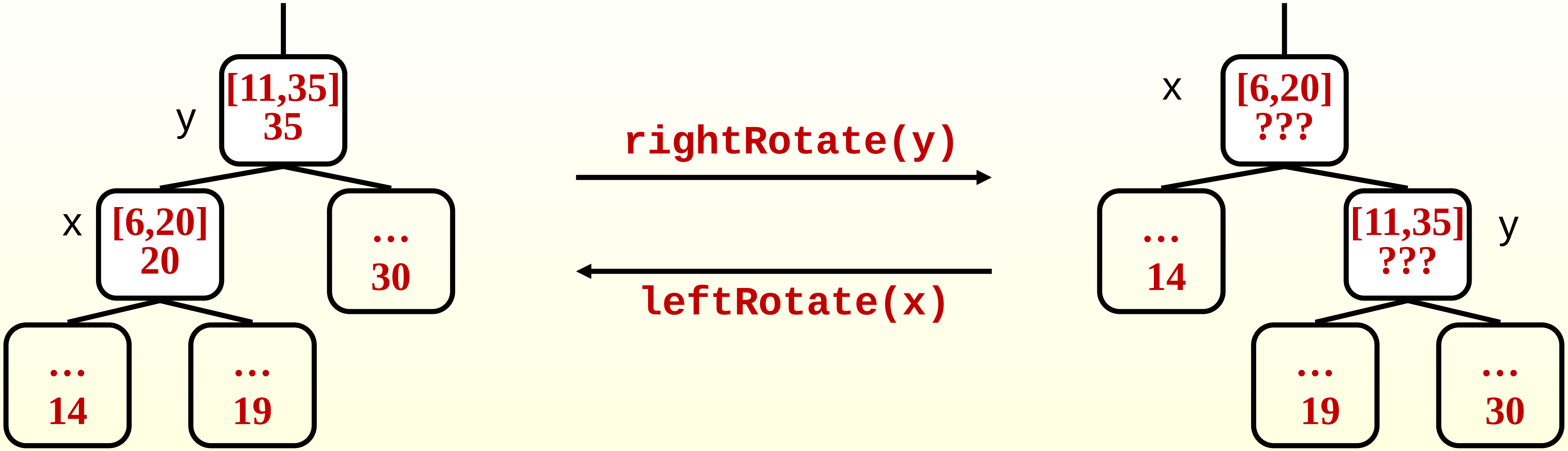
Note that:

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

Interval Trees

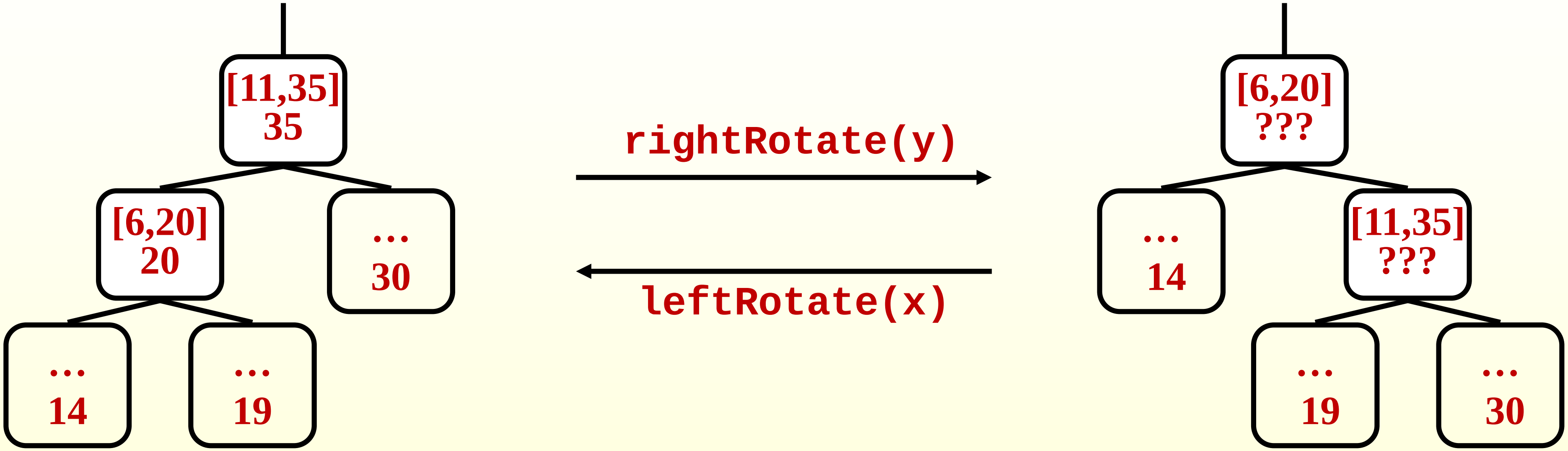
- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$ (the left endpoint)
  - Decide what additional information to store
    - Store the maximum right endpoint in the subtree rooted at *i*
  - *Figure out how to maintain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

Interval Trees



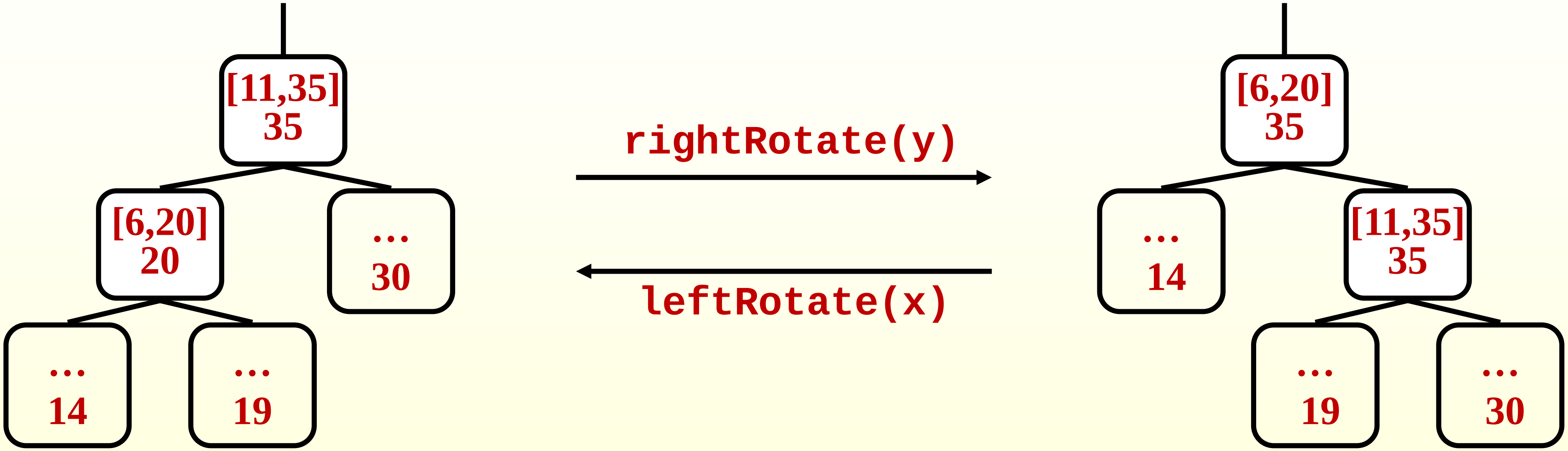
- *What are the new max values for the subtrees below x and y?*

Interval Trees



- *What are the new max values for the subtrees below x and y?*
- A: Unchanged
- *What are the new max values for x and y?*

Interval Trees



- *What are the new max values for the subtrees below x and y?*
- A: Unchanged
- *What are the new max values for x and y?*
- A: root value unchanged, recompute the other

Interval Trees

- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$ (the left endpoint)
  - Decide what additional information to store
    - Store the maximum right 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

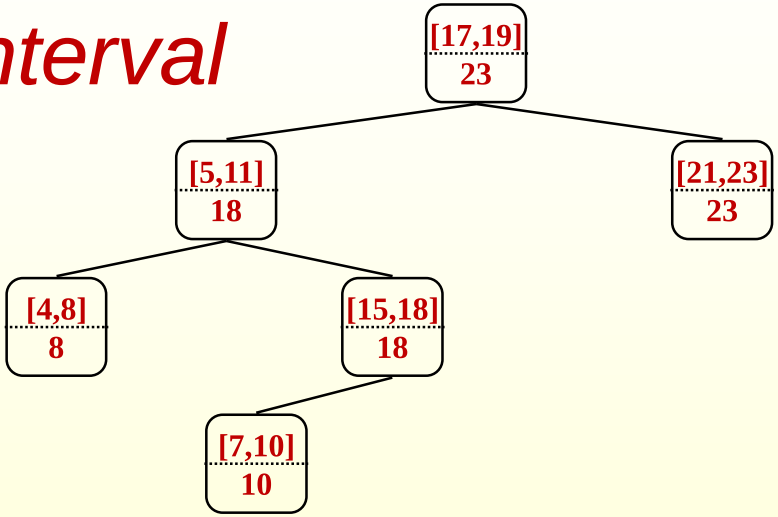
```
IntervalSearch(T, i)
{
    x = T->root;
    while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max ≥ i->low)
            x = x->left;
        else
            x = x->right;
    return x
}
```

- *What is the running time?*

O(log n)

IntervalSearch() Example

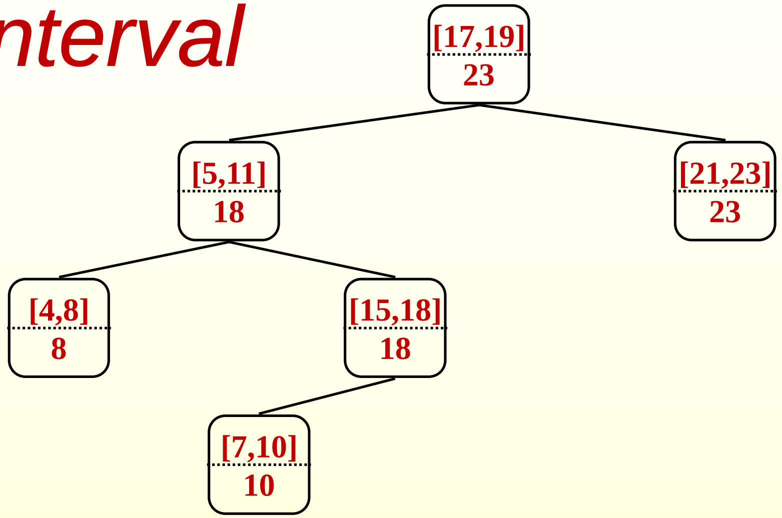
- *Example1: search for interval overlapping [20,22]*




```
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

- *Example2: search for interval overlapping [16,20]*




```
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 one of a node's two children
  - Case 1: search goes right
    - Show that $\exists$ overlap in right subtree, or no overlap at all
  - Case 2: search goes left
    - Show that $\exists$ 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 $x \rightarrow left \rightarrow max < x \rightarrow low$ (*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:
    - $i \rightarrow low \leq x \rightarrow left \rightarrow max$, by branch condition
    - $x \rightarrow left \rightarrow max = y \rightarrow high$ for some y in left subtree
    - Since i and y don't overlap and $i \rightarrow low \leq y \rightarrow high$, $i \rightarrow high < y \rightarrow low$
    - Since tree is sorted by low's, $i \rightarrow high$ < 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;
```

Amortized Analysis

Amortized Analysis

Key point: The time required to perform a sequence of data structure operations is averaged over all operations performed

- Amortized analysis can be used to show that
  - The average cost of an operation is small
    - If one averages over a sequence of operations

  even though a single operation might be expensive

Amortized Analysis vs Average Case Analysis

- Amortized analysis does not use any *probabilistic reasoning*
- Amortized analysis guarantees

  the average performance of each operation in the worst case

  but now we average over a sequence of operations

Methods of Amortized Analysis

- **Aggregate Method**: we determine an upper bound T(n) on the total sequence of n operations. The cost of each will then be T(n)/n.
- **Accounting Method:** we overcharge some operations early and use them to as prepaid charge later.
- **Potential Method:** we maintain credit as potential energy associated with the data structure as a whole.

Amortized Analysis

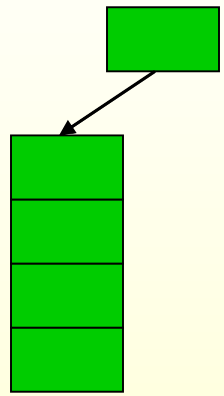
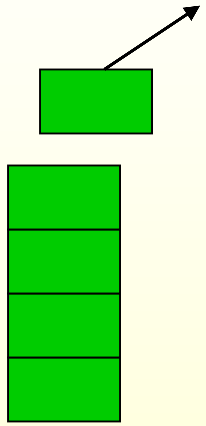
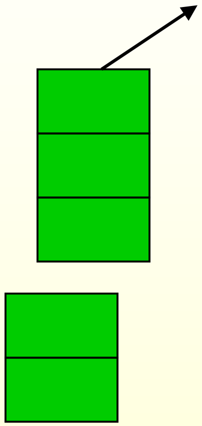
1. Aggregate Method

- Show that for all n, a sequence of n operations take worst-case time T(n) in total

- In the worst case, the average cost, or amortized cost , per operation is T(n)/n.

- The amortized cost applies to each operation, even when there are several types of operations in the sequence.

Aggregate Analysis: Stack Example

<table>
<tr><td rowspan="2">3 ops:</td><td></td><td></td><td></td></tr>
<tr><td>Push(S,x)</td><td>Pop(S)</td><td>Multi-pop(S,k)</td></tr>
<tr><td>Worst-case cost:</td><td>1</td><td>1</td><td>min(|S|,k)
[= O(n)]</td></tr>
</table>

Amortized cost: O(1) per operation

……. Aggregate Analysis: Stack Example

- Sequence of n *push*, *pop*, *Multipop* operations
  - Worst-case cost of Multipop is O(n)
  - Have n operations
  - Therefore, worst-case cost of sequence is $O(n^2)$
- Observations
  - Each object can be popped only once per time that it's pushed
  - Have at most n pushes => at most n pops, including those in Multipop
  - Therefore total cost = O(n)
  - Average over n operations => O(1) per operation on average
- Notice that no probability is involved

Amortized Analysis

2. Accounting Method

- Charge i-th operation a fictitious amortized cost $\hat{c}_i$, where \$1 pays for 1 unit of work (i.e., time).
  - Assign different charges (amortized costs) to different operations
    - Some are charged more than actual cost
    - Some are charged less
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is "stored in the bank" for use by subsequent operations.
- The bank balance (the credit) must not go negative!

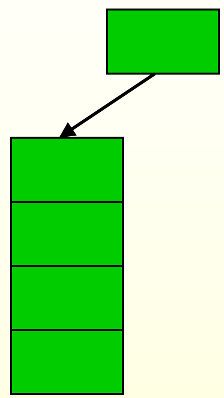
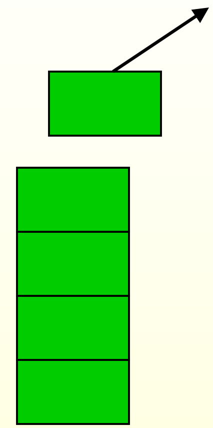
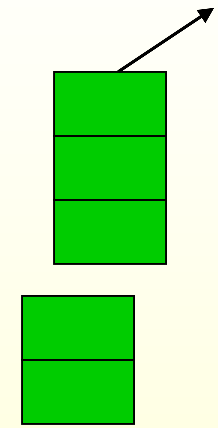
We must ensure that for all n.

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

- Thus, the total amortized costs provide an upper bound on the total true costs.

Amortized Analysis

….. Accounting Method: Stack Example

3 ops:			
	Push(S,x)	Pop(S)	Multi-pop(S,k)
•Assigned cost:	2	0	0
•Actual cost:	1	1	min($\lvert S\rvert$,k)

Push(S,x) pays for possible later pop of x.

….. Accounting Method: Stack Example

- When pushing an object, pay $2
  - $1 pays for the push
  - $1 is prepayment for it being popped by either pop or Multipop
  - Since each object has $1, which is credit, the credit can never go negative
  - Therefore, total amortized cost = O(n), is an upper bound on total actual cost

….. Accounting Method: k-bit Binary Counter

Introduction

- k-bit Binary Counter: A[0..k–1]

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

```
INCREMENT(A)
1.  i ← 0
2.  while i < length[A] and A[i] = 1
3.       do  A[i] ← 0          ▹ reset a bit (carry propagation)
4.            i ← i + 1
5.  if  i < length[A]
6.       then  A[i] ← 1        ▹ set a bit
```

Amortized Analysis

….. Accounting Method: k-bit Binary Counter

Consider a sequence of n increments. The worst-case time to execute one increment is $\Theta(k)$. Therefore, the worst-case time for n increments is $n \cdot \Theta(k) = \Theta(n \cdot k)$.

WRONG! In fact, the worst-case cost for n increments is only $\Theta(n) \ll \Theta(n \cdot k)$.

Let's see why.

Note: You'd be correct
if you'd said $O(n \cdot k)$.
But, it's an overestimate.

Amortized Analysis

….. Accounting Method: k-bit Binary Counter

Total cost of *n* operations

A[0] flipped every op n

A[1] flipped every 2 ops $n/2$

A[2] flipped every 4 ops $n/2^2$

A[3] flipped every 8 ops $n/2^3$

… … … … …

A[*i*] flipped every 2^i ops $n/2^i$

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]	*Cost*
0	0	0	0	0	**0**	*0*
1	0	0	0	**0**	**1**	*1*
2	0	0	0	1	**0**	*3*
3	0	0	**0**	**1**	**1**	*4*
4	0	0	1	0	**0**	*7*
5	0	0	1	**0**	**1**	*8*
6	0	0	1	1	**0**	*10*
7	0	**0**	**1**	**1**	**1**	*11*
8	0	1	0	0	**0**	*15*
9	0	1	0	**0**	**1**	*16*
10	0	1	0	1	**0**	*18*
11	0	1	**0**	**1**	**1**	*19*

Amortized Analysis

….. Accounting Method: k-bit Binary Counter

Cost of n increments

$$= \sum_{i=1}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor$$

$$< n \sum_{i=1}^{\infty} \frac{1}{2^i} = 2n$$

$$= \Theta(n)$$

Thus, the average cost of each increment operation is $\Theta(n)/n = \Theta(1)$.

….. Accounting Method: k-bit Binary Counter

Charge an amortized cost of \$2 every time a bit is set from 0 to 1

- \$1 pays for the actual bit setting.
- \$1 is stored for later re-setting (from 1 to 0).

At any point, every 1 bit in the counter has \$1 on it… that pays for resetting it. (reset is "*free*")

Example:

0 0 0 1[\$1] 0 1[\$1] 0

0 0 0 1[\$1] 0 1[\$1] 1[\$1] **Cost = \$2**

0 0 0 1[\$1] 1[\$1] 0 0 **Cost = \$2**

Note that in the accounting method we bank credits with individual data items

Amortized Analysis

….. Accounting Method: k-bit Binary Counter

```
INCREMENT(A)
1.  i ← 0
2.  while i < length[A] and A[i] = 1
3.      do  A[i] ← 0          ▻ reset a bit
4.          i ← i + 1
5.  if  i < length[A]
6.      then  A[i] ← 1        ▻ set a bit
```

- When Incrementing,
  - Amortized cost for line 3 = $0
  - Amortized cost for line 6 = $2

- Amortized cost for INCREMENT(A) = $2
- Amortized cost for n INCREMENT(A) = $2n =O(n)

Amortized Analysis

3. Potential Method

IDEA: View the bank account as the potential energy (as in physics) of the dynamic set.

FRAMEWORK:

- Start with an initial data structure D_0.
- Operation i transforms D_{i-1} to D_i.
- The cost of operation i is c_i.
- Define a ***potential function*** $\Phi : \{D_i\} \rightarrow \mathrm{R}$, such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i.
- The ***amortized cost*** $\hat{c}_i$ with respect to Φ is defined to be $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Amortized Analysis

….. Potential Method

- Like the accounting method, but think of the credit as ***potential*** stored with the ***entire data structure***.
  - Accounting method stores credit with specific objects while potential method stores potential in the data structure as a whole.
  - Can release potential to pay for future operations
- Most flexible of the amortized analysis methods.

Amortized Analysis

….. Potential Method

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textit{potential difference } \Delta\Phi_i}$$

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation *i* stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation *i*.

Amortized Analysis

….. Potential Method

The total amortized cost of *n* operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

The RHS telescopes to give:

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i$$ since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$.

The Potential Method

If we can ensure that $\phi(D_i) \geq \phi(D_0)$ then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ is an upper bound on the total actual cost

However, $\phi(D_n) \geq \phi(D_0)$ should hold for all possible n since, in practice, we do not always know n in advance

Hence, if we require that $\phi(D_i) \geq \phi(D_0)$, for all i, then we ensure that we pay in advance (as in the accounting method)

Amortized Analysis

….. Potential Method: Stack Example

Define: $\phi(D_i)$ = # items in stack Thus, $\phi(D_0)=0$.

Plug in for operations to get amortized costs: (j = actual stack size)

Push: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= 1 + j - (j-1)$

$= 2$

Pop: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= 1 + (j-1) - j$

$= 0$

Multi-pop: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= k' + (j-k') - j$ $k'=\min(|S|,k)$

$= 0$

….. Potential Method: k-bit Binary Counter

Define the potential of the counter after the i^{th} operation by $\Phi(D_i) = b_i$, the number of 1's in the counter after the i^{th} operation.

Note:

- $\Phi(D_0) = 0$,
- $\Phi(D_i) \geq 0$ for all i.

Example:

0 0 0 1 0 1 0

(**0 0 0 1$^{\$1}$ 0 1$^{\$1}$ 0** Accounting method)

….. Potential Method

Assume ith INCREMENT resets t_i bits (in line 3).
Actual cost $c_i = (t_i + 1)$
Number of 1's after ith operation: $b_i = b_{i-1} - t_i + 1$
The amortized cost of the i th INCREMENT is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (t_i + 1) + (1 - t_i) \\ &= 2\end{aligned}$$

Therefore, n INCREMENTs cost $\Theta(n)$ in the worst case.

Amortized Analysis

- Takes some experience to properly define amortized costs
- It is very common in data structures that an individual operation can be expensive, but not all operations in a sequence can