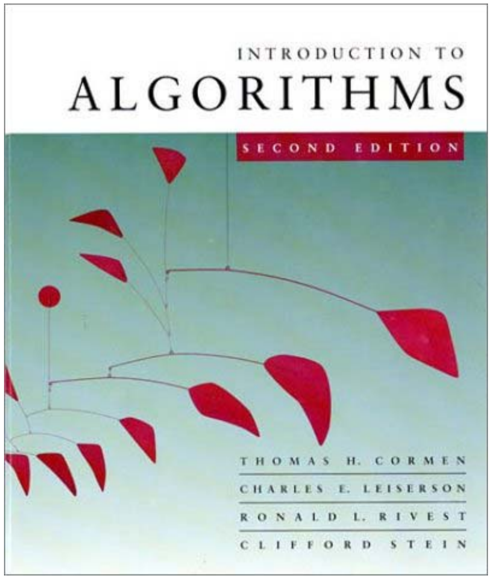
Introduction to Algorithms

6.046J/18.401J/SMA5503

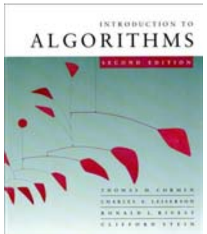


Lecture 10

Prof. Piotr Indyk

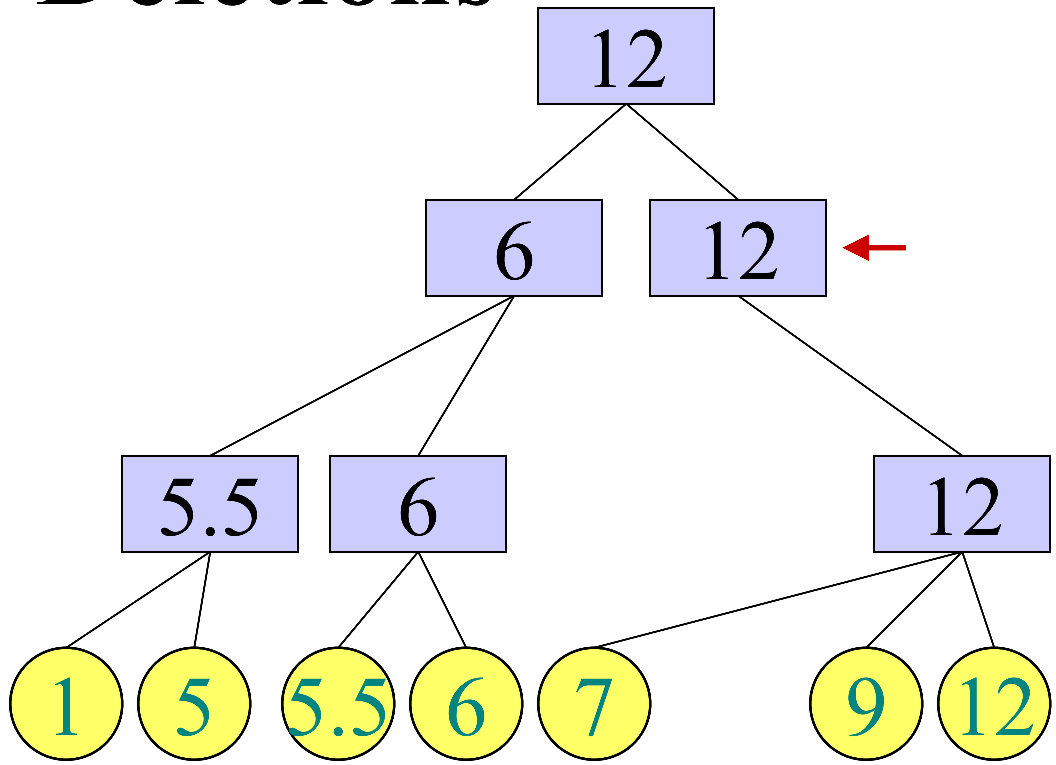
Today

- A data structure for a new problem
- Amortized analysis

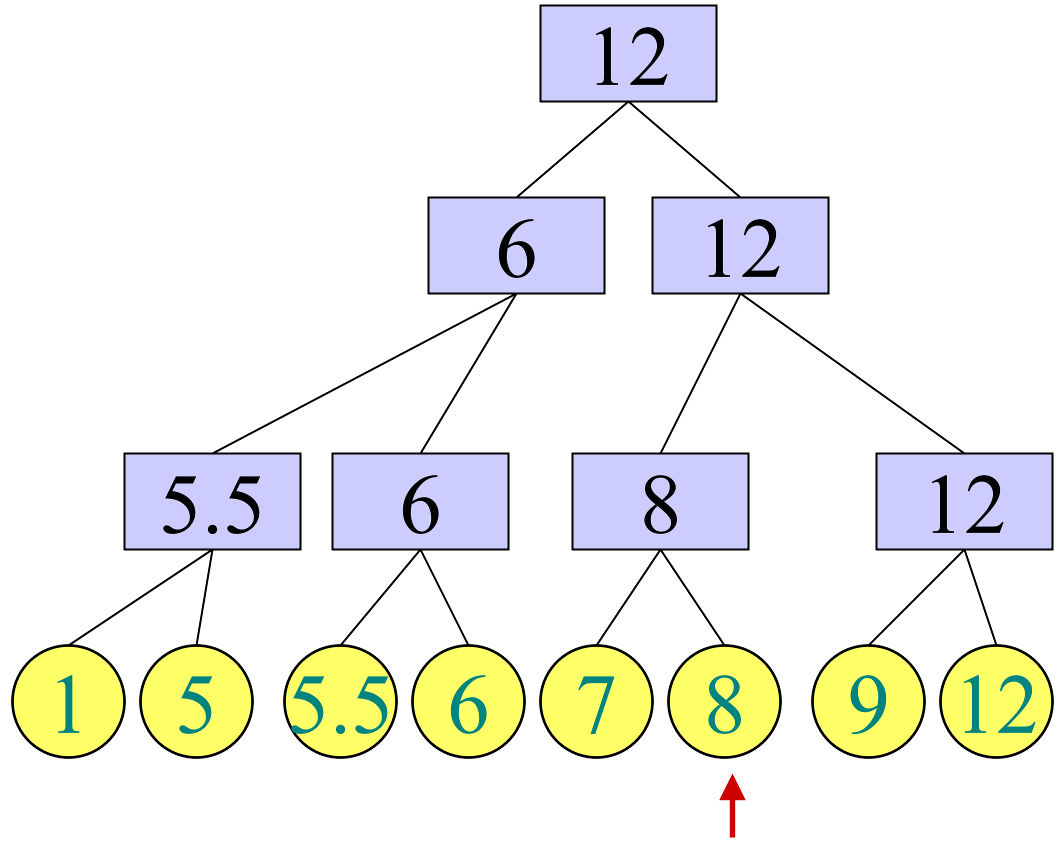


2-3 Trees: Deletions

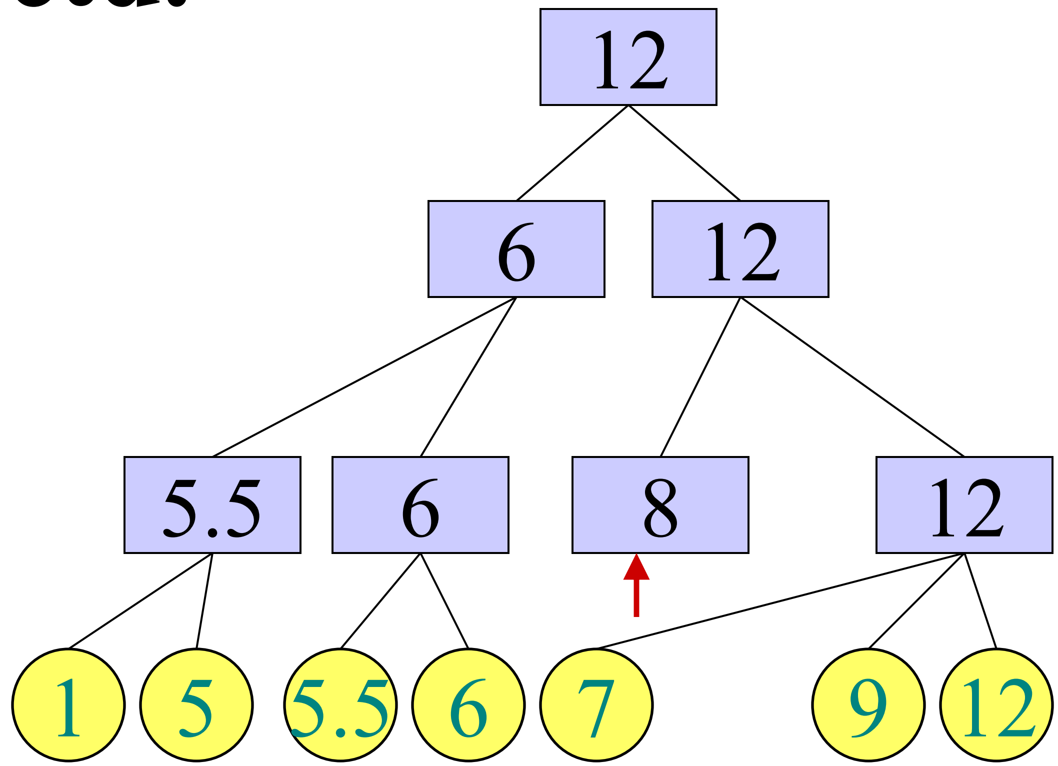
- Problem: there is an internal node that has only 1 child
- Solution: delete recursively



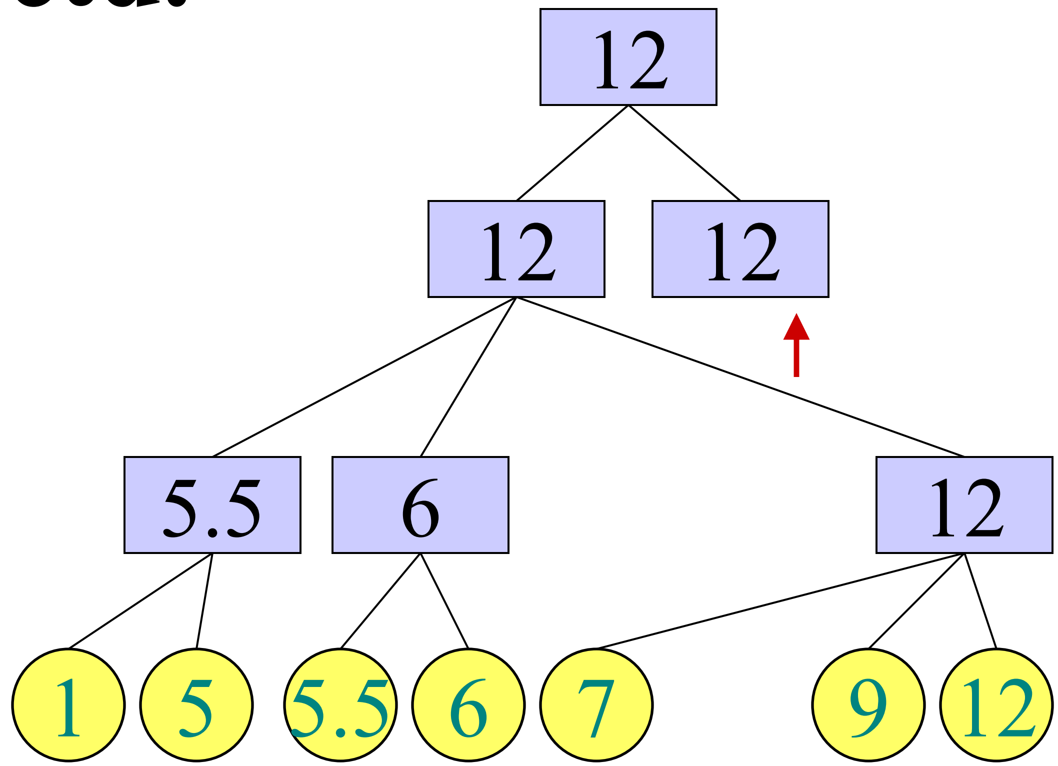
Example



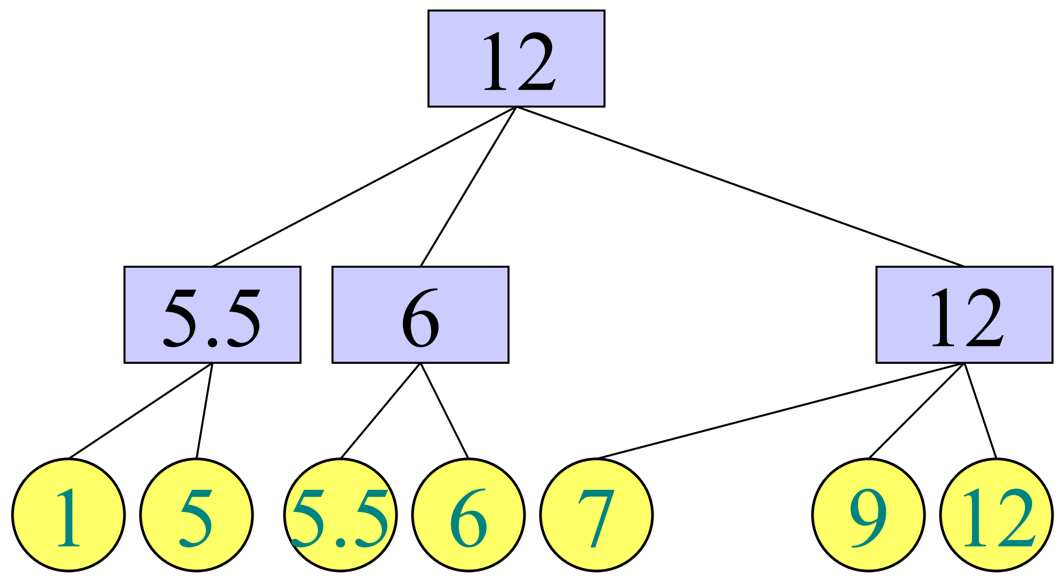
Example, ctd.

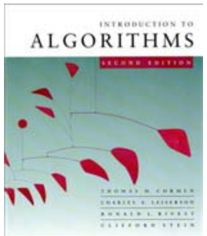


Example, ctd.

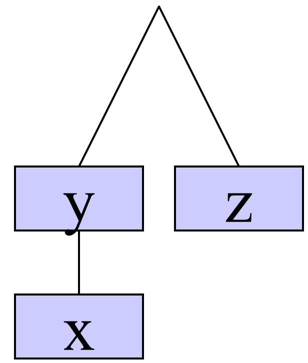


Example, ctd.





Procedure for Delete(x)



- Let y=p(x)
- Remove x
- If y≠root then
  - Let z be the sibling of y.
  - Assume z is the right sibling of y, otherwise the code is symmetric.
  - If y has only 1 child w left

    Case 1: z has 3 children
    - Attach left[z] as the rightmost child of y
    - Update y.max and z.max

    Case 2: z has 2 children:
    - Attach the child w of y as the leftmost child of z
    - Update z.max
    - Delete(y) (recursively*)
  - Else
    - Update max of y, p(y), p(p(y)) and so on until root
- Else
  - If root has only one child u
    - Remove root
    - Make u the new root

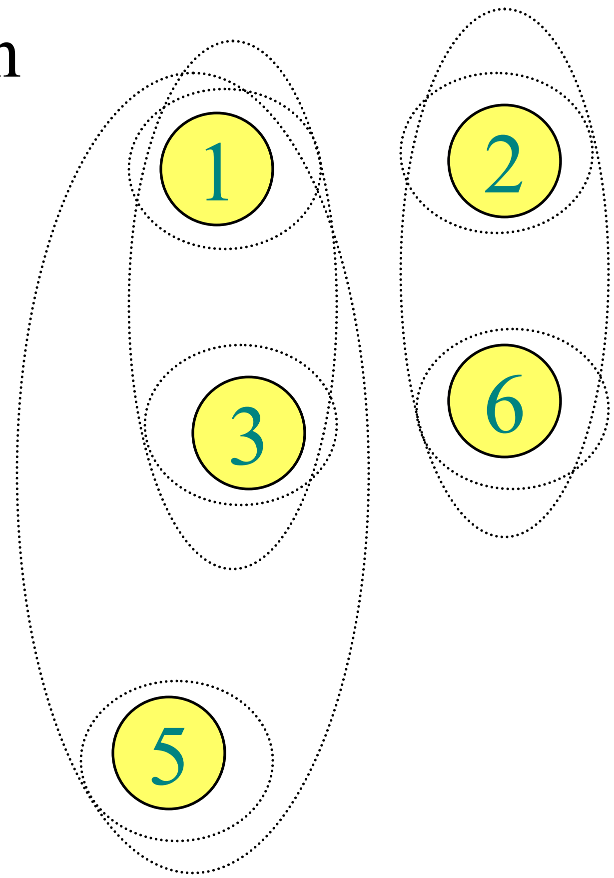
*Note that the input of Delete does not have to be a leaf

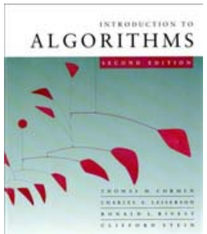
2-3 Trees

- The simplest balanced trees on the planet!
  (but, nevertheless, not completely trivial)

Dynamic Maintenance of Sets

- Assume, we have a collection of elements
- The elements are clustered
- Initially, each element forms its own cluster/set
- We want to enable two operations:
  - FIND-SET(x): report the cluster containing x
  - UNION(C_1, C_2): merges the clusters C_1, C_2

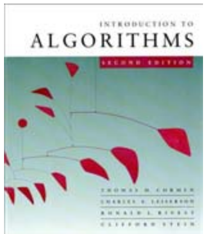




Disjoint-set data structure (Union-Find)

Problem:

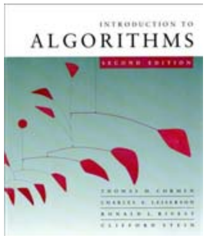
- Maintain a collection of *pairwise-disjoint* sets $\mathsf{S} = \{S_1, S_2, \ldots, S_r\}$.
- Each S_i has one representative element $x = rep[S_i]$.
- Must support three operations:
  - MAKE-SET(x): adds new set $\{x\}$ to S with $rep[\{x\}] = x$ (for any $x \notin S_i$ for all i).
  - WEAK • UNION(x, y): replaces sets S_x, S_y with $S_x \cup S_y$ in S for any *rep.* x, y in distinct sets S_x, S_y.
  - FIND-SET(x): returns representative $rep[S_x]$ of set S_x containing element x.



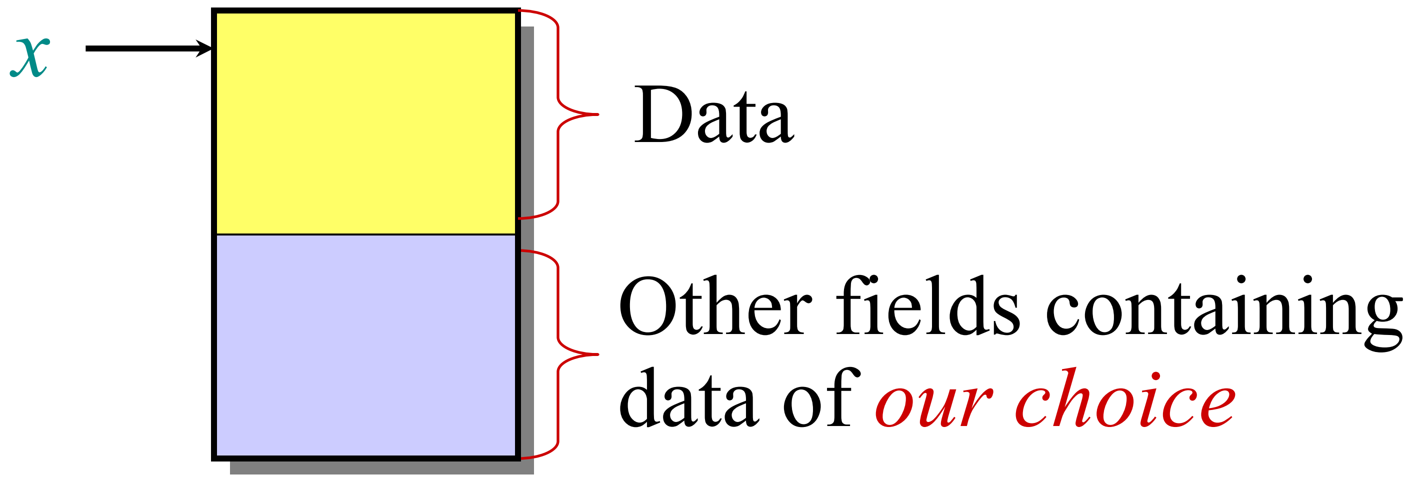
Quiz

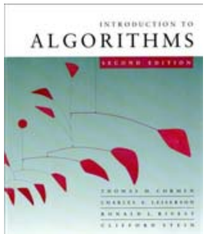
- If we have a WEAKUNION(x, y) that works only if x, y are representatives, how can we implement UNION that works for *any* x, y ?
- UNION(x, y)

  =WEAKUNION(FIND-SET(x) , FIND-SET(y))



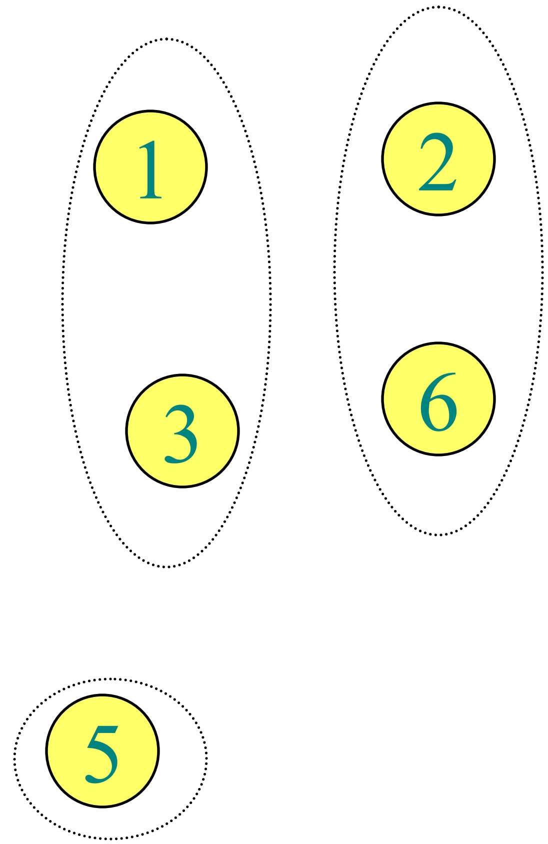
Representation





Applications

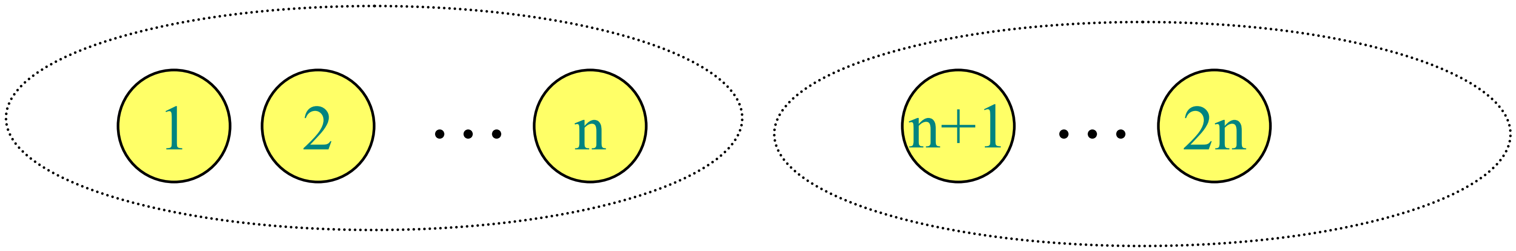
- Data clustering
- Killer App: Minimum Spanning Tree (Lecture 13)
- Amortized analysis

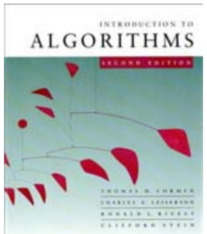


Ideas ?

- How can we implement this data structure efficiently ?
  - MAKE-SET
  - UNION
  - FIND-SET

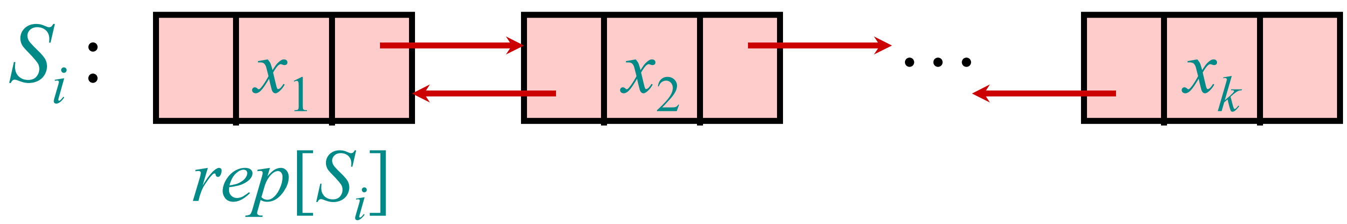
Bad case for UNION or FIND





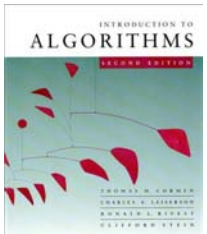
Simple linked-list solution

Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an (unordered) doubly linked list. Define representative element $rep[S_i]$ to be the front of the list, x_1.



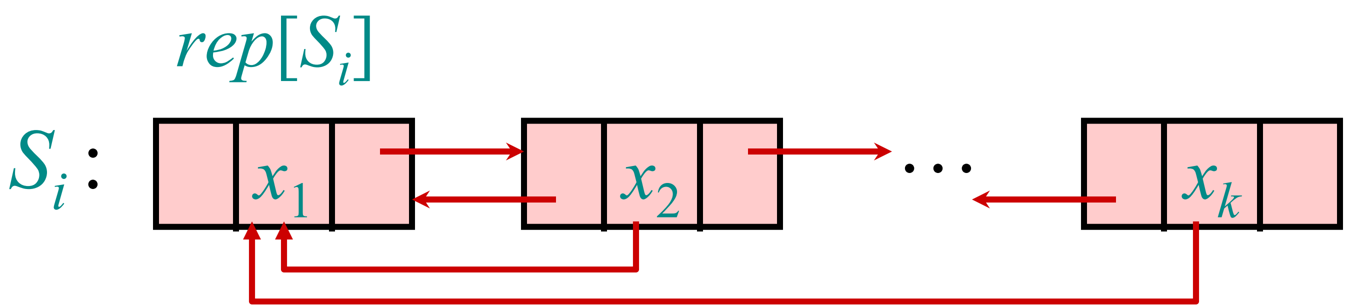
How can we improve it ?

- MAKE-SET(x) initializes x as a lone node. $\Theta(1)$
- FIND-SET(x) walks left in the list containing x until it reaches the front of the list. $\Theta(n)$
- UNION(x, y) concatenates the lists containing x and y, leaving rep. as FIND-SET$[x]$. $\Theta(n)$



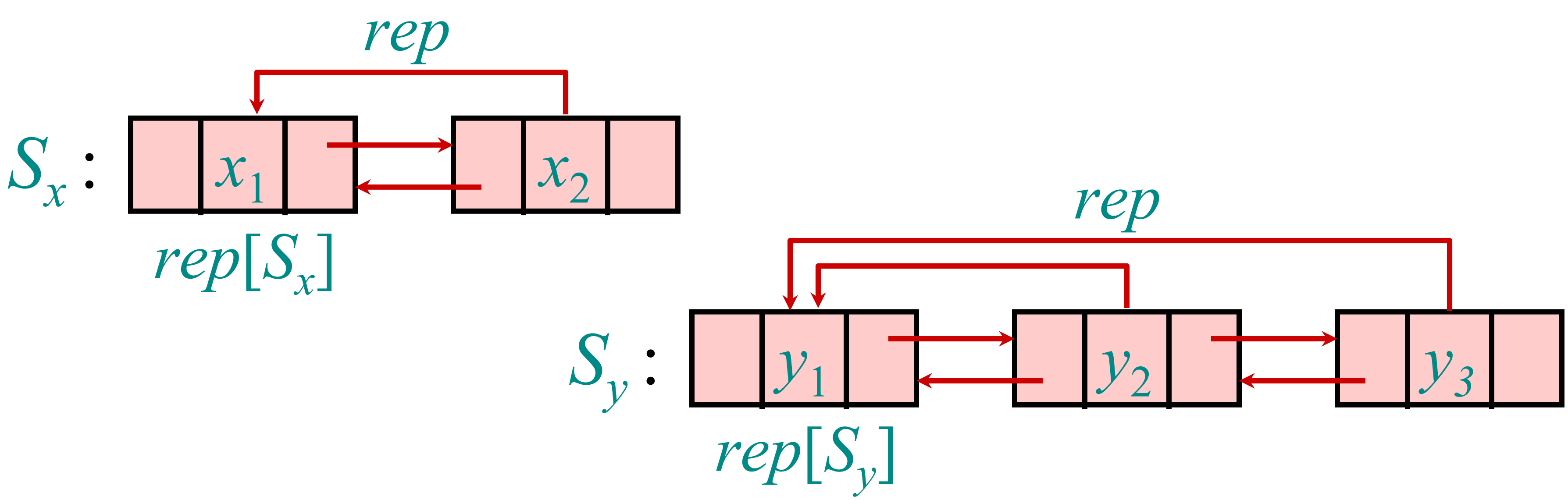
Augmented linked-list solution

Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as unordered doubly linked list. Each x_j also stores pointer $rep[x_j]$ to head.

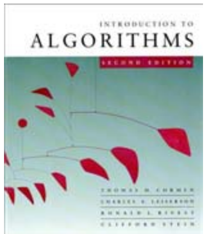


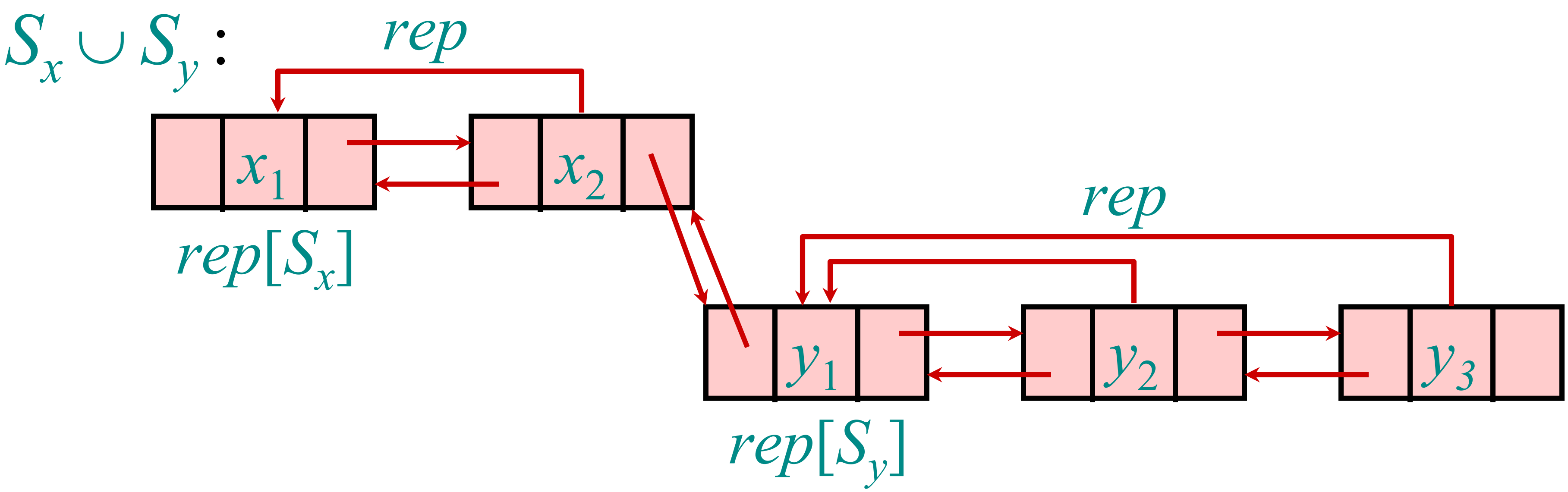
- FIND-SET(x) returns $rep[x]$.
- UNION(x, y) concatenates the lists containing x and y, and updates the rep pointers for all elements in the list containing y.

Example of augmented linked-list solution

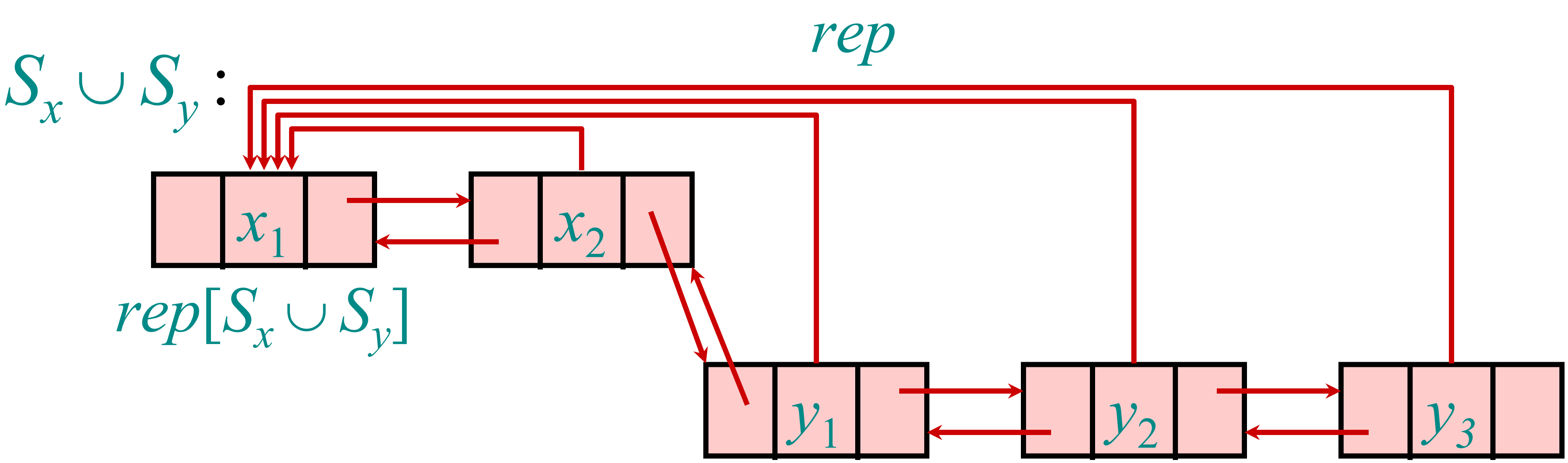


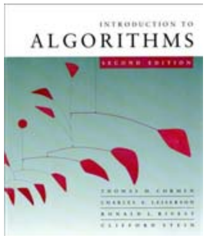
Example of augmented linked-list solution





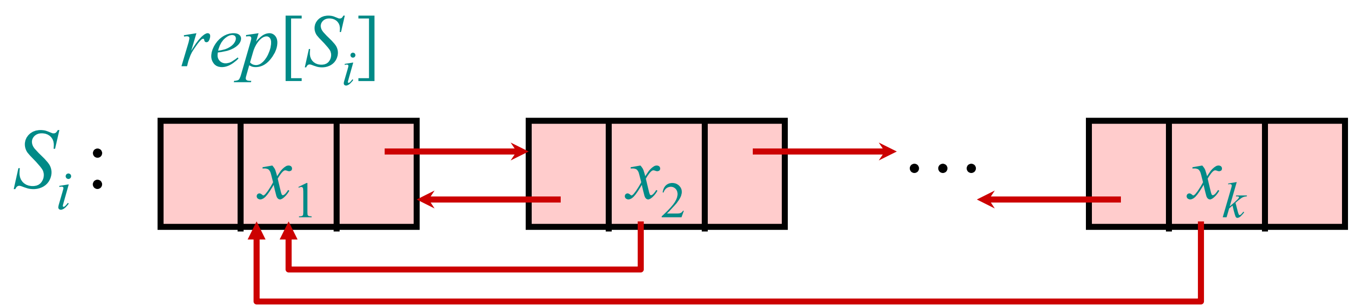
Example of augmented linked-list solution



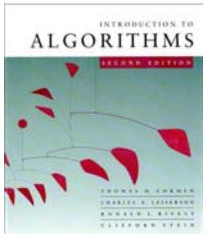


Augmented linked-list solution

Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as unordered doubly linked list. Each x_j also stores pointer $rep[x_j]$ to head.



- FIND-SET(x) returns $rep[x]$. $\Theta(1)$
- UNION(x, y) concatenates the lists containing x and y, and updates the *rep* pointers for all elements in the list containing y. $\Theta(n)$ ↑ ?



Amortized analysis

- So far, we focused on worst-case time of *each* operation.
  - E.g., UNION takes $\Theta(n)$ time for *some* operations
- Amortized analysis: count the *total* time spent by any sequence of operations
- Total time is always at most

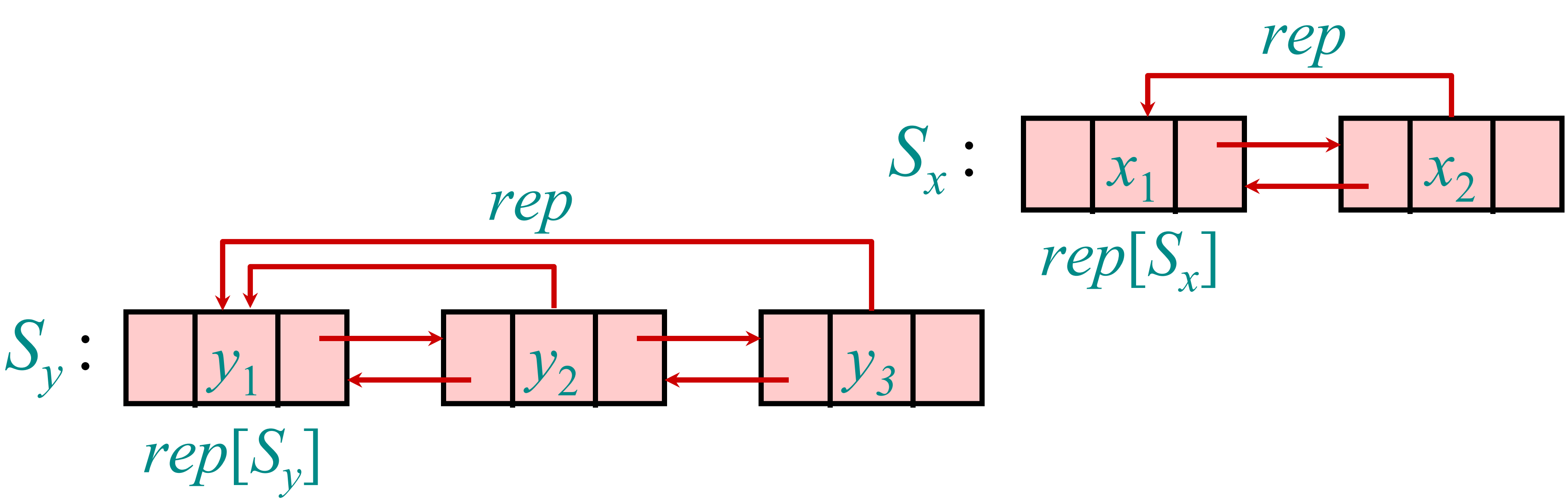
  worst-case-time-per-operation * #operations

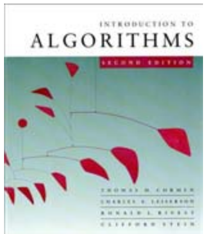
  but it can be much better!
- E.g., if times are *1,1,1,…,1,n,1,…,1*
- Can we modify the linked-list data structure so that any sequence of m MAKE-SET, FIND-SET, UNION operations cost less than m*$\Theta(n)$ time?

Alternative

UNION(x, y) :

- concatenates the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing ~~y~~ x

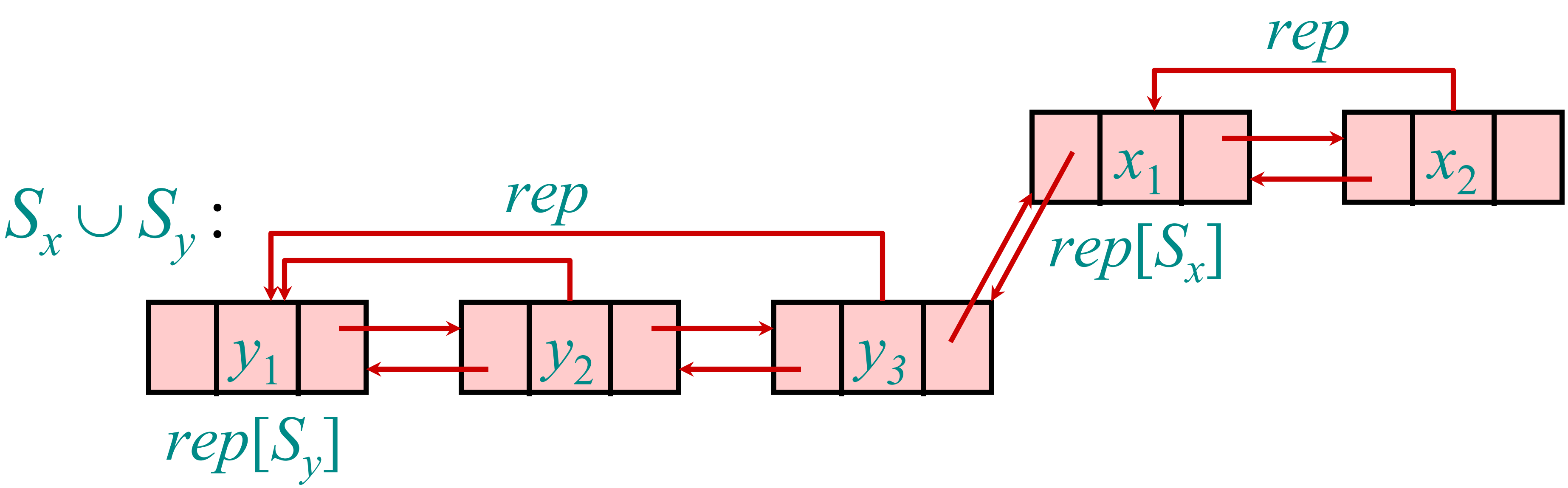


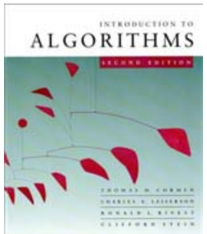


Alternative concatenation

UNION(x, y) could instead

- concatenate the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing x.

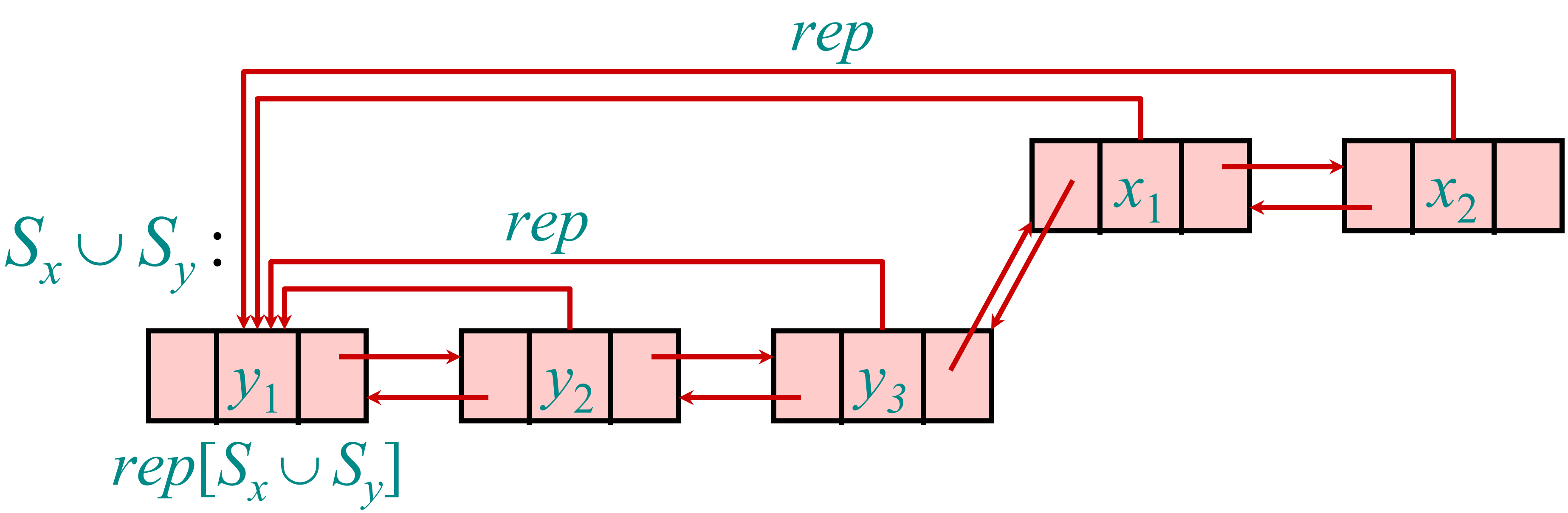




Alternative concatenation

UNION(x, y) could instead

- concatenate the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing x.



Smaller into larger

- Concatenate smaller list onto the end of the larger list (each list stores its *weight* = # elements)

- Cost = Θ(length of smaller list).

Let n denote the overall number of elements (equivalently, the number of MAKE-SET operations). Let m denote the total number of operations.

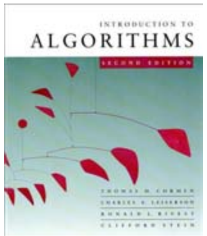
Theorem: Cost of all UNION's is $O(n \lg n)$.

Corollary: Total cost is $O(m + n \lg n)$.

Total UNION cost is $O(n \lg n)$

Proof:

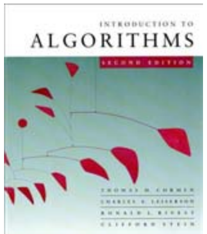
- Monitor an element x and set S_x containing it
- After initial MAKE-SET(x), $weight[S_x] = 1$
- Consider any time when S_x is merged with set S_y
  - If $weight[S_y] \geq weight[S_x]$
    - pay 1 to update $rep[x]$
    - $weight[S_x]$ at least doubles (increasing by $weight[S_y]$)
  - Otherwise
    - pay nothing
    - $weight[S_x]$ only increases
- Thus:
  - Each time we pay 1, the weight doubles
  - Maximum possible weight is n
  - Maximum pay $\leq \lg n$ for x , or $O(n \log n)$ overall



Final Result

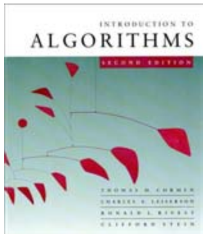
- We have a data structure for dynamic sets which supports:
  - MAKE-SET: O(1) worst case
  - FIND-SET: O(1) worst case
  - UNION:
    - Any sequence of any m operations* takes $O(m \log n)$ time, or
    - … the *amortized complexity* of the operations* is $O(\log n)$

* I.e., MAKE-SET, FIND-SET or UNION



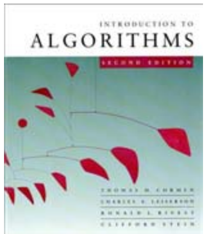
Amortized vs Average

- What is the difference between average case complexity and amortized complexity ?
  - "Average case" assumes *random distribution* over the input (e.g., random sequence of operations)
  - "Amortized" means we count the *total* time taken by *any* sequence of m operations (and divide it by m)



Can we do better ?

- One can do:
  - MAKE-SET: O(1) worst case
  - FIND-SET: O(lg n) worst case
  - WEAKUNION: O(1) worst case
  - Thus, UNION: O(lg n) worst case

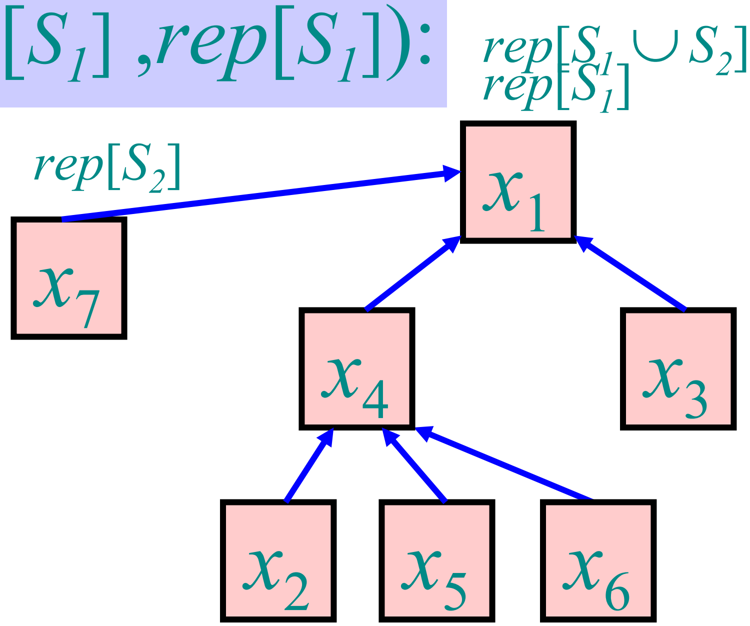


Representing sets as trees

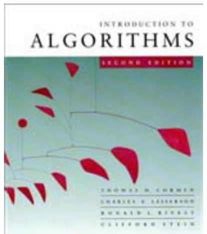
- Each set $S_i = \{x_1, x_2, \ldots, x_k\}$ stored as a tree
- $rep[S_i]$ is the tree root.

UNION($rep[S_1]$,$rep[S_1]$):

- MAKE-SET(x) initializes x as a lone node.
- FIND-SET(x) walks up the tree containing x until it reaches the root.
- UNION(x, y) concatenates the trees containing x and y

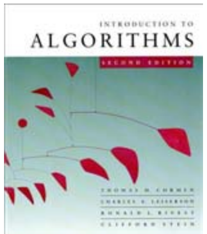


$S_1 = \{x_1, x_2, x_3, x_4, x_5 , x_6\}$
$S_2 = \{x_7\}$



Time Analysis

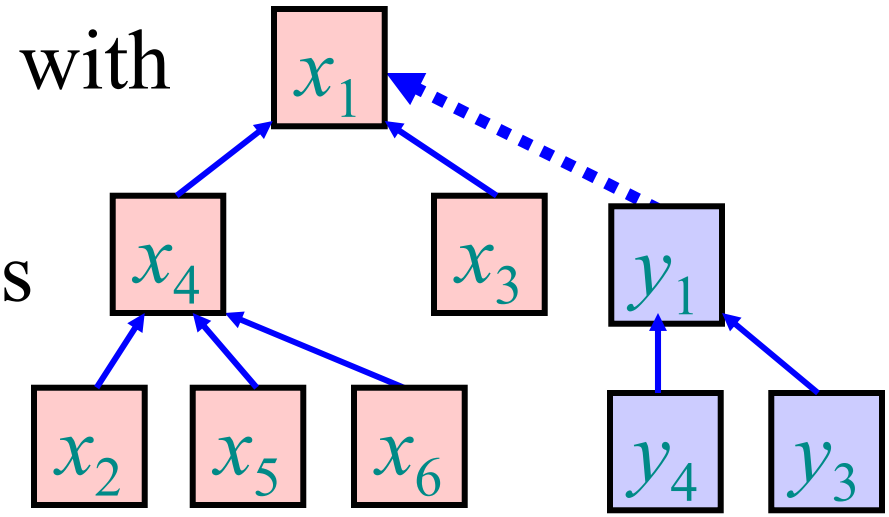
- MAKE-SET(x) initializes x as a lone node. — O(1)
- FIND-SET(x) walks up the tree containing x until it reaches the root. — O(depth) = ?
- WEAKUNION(x, y) concatenates the trees containing x and y — O(1)



"Smaller into Larger" in trees

Algorithm: Merge tree with smaller weight into tree with larger weight.

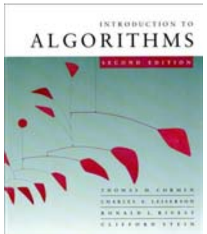
- Height of tree increases only when its size doubles
- Height logarithmic in weight



“Smaller into Larger” in trees

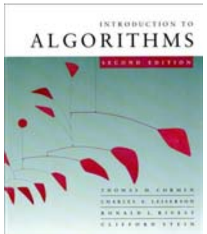
Proof:

- Monitor the height of an element z
- Each time the height of z increases, the weight of its tree doubles
- Maximum weight is n
- Thus, height of z is $\leq \log n$



Tree implementation

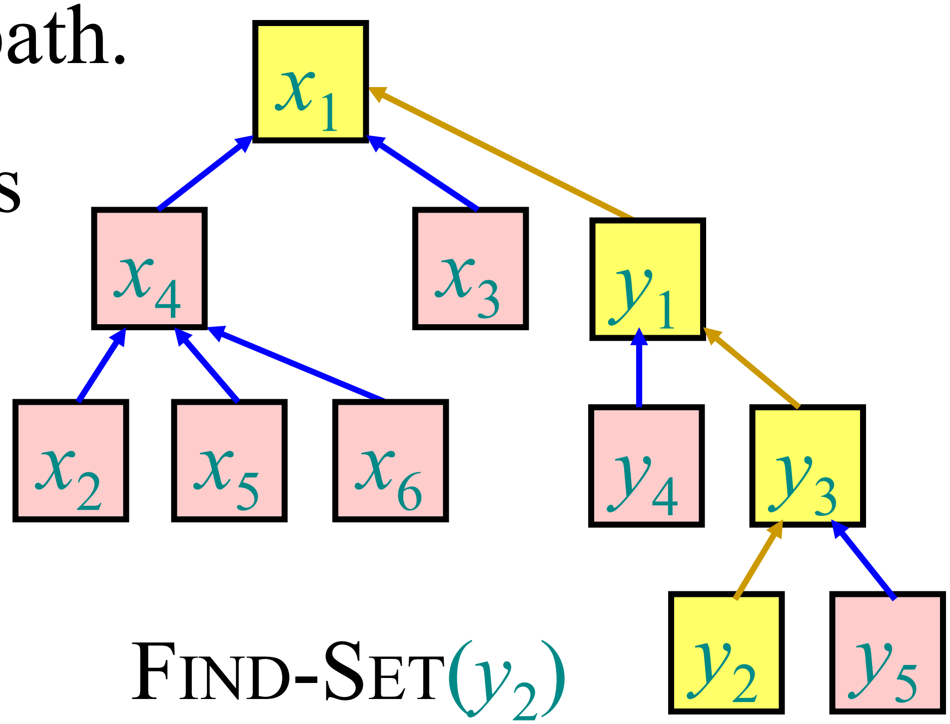
- We have:
  - Make-Set: O(1) worst case
  - Find-Set: O(depth) = O(lg n) worst case
  - WeakUnion: O(1) worst case
- Can amortized analysis buy us anything ?
- Need another trick…

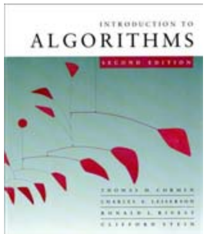


Trick 2: Path compression

When we execute a FIND-SET operation and walk up a path to the root, we *know* the representative for *all* the nodes on the path.

Path compression makes all of those nodes direct children of the root.

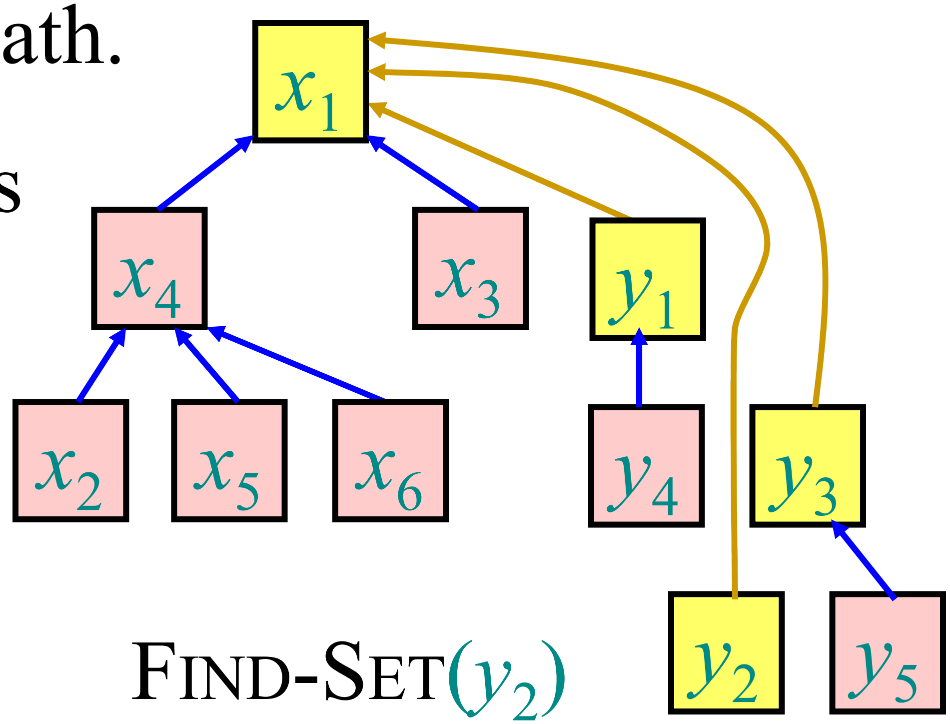


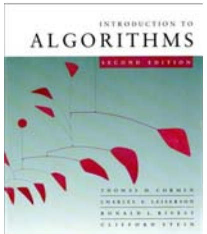


Trick 2: Path compression

When we execute a FIND-SET operation and walk up a path to the root, we know the representative for all the nodes on the path.

Path compression makes all of those nodes direct children of the root.



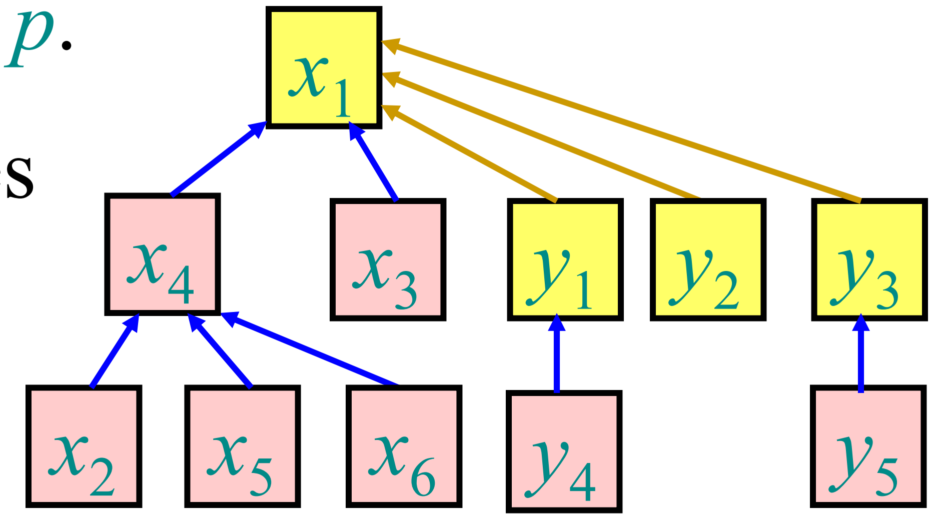


Trick 2: Path compression

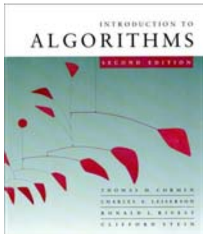
When we execute a FIND-SET operation and walk up a path p to the root, we know the representative for all the nodes on path p.

Path compression makes all of those nodes direct children of the root.

Cost of FIND-SET(x) is still $\Theta(depth[x])$.

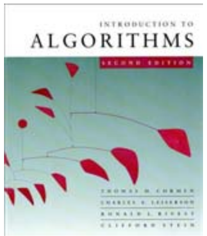


FIND-SET(y_2)



The Theorem

Theorem: In general, amortized cost is $O(\alpha(n))$, where $\alpha(n)$ grows really, really, really slow.



Ackermann's function A

Define $A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$ -iterate $A_{k-1}()$ $j+1$ times

$A_0(j) = j + 1$

$A_1(j) = A_0(\ldots(A_0(j)\ldots) \sim 2j$

$A_2(j) = A_1(\ldots A_1(j)\ldots) \sim 2j\ 2^j$

$A_3(j) > 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^j}}}}}} \Big\} j$

$A_4(j)$ is a lot bigger.

$A_0(1) = 2$

$A_1(1) = 3$

$A_2(1) = 7$

$A_3(1) = 2047$

$A_4(1) > 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{2047}}}}}}} \Big\} 2048$

Define $\alpha(n) = \min \{k : A_k(1) \geq n\}$.

The Theorem

Theorem: In general, amortized cost is $O(\alpha(n))$, where $\alpha(n)$ grows really, really, really slow.

Proof: Really, really, really long (CLRS, p. 509)

Application: Dynamic connectivity

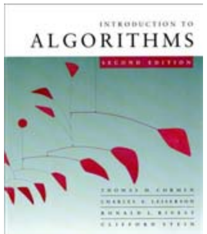
Suppose a graph is given to us ***incrementally*** by

- Add-Vertex(v)
- Add-Edge(u, v)

and we want to support ***connectivity*** queries:

- Connected(u, v):
  Are u and v in the same connected component?

For example, we want to maintain a spanning forest, so we check whether each new edge connects a previously disconnected pair of vertices.



Application: Dynamic connectivity

Sets of vertices represent *connected components*.
Suppose a graph is given to us ***incrementally*** by

- Add-Vertex(v) – Make-Set(v)
- Add-Edge(u, v) – **if** not Connected(u, v)
  then Union(v, w)

and we want to support ***connectivity*** queries:

- Connected(u, v): – Find-Set(u) = Find-Set(v)
  Are u and v in the same connected component?

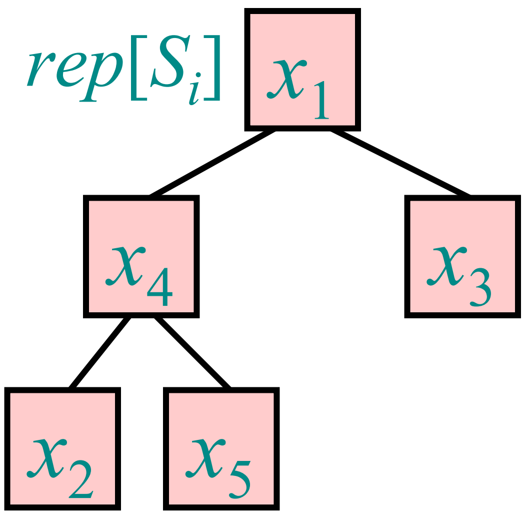
For example, we want to maintain a spanning forest, so we check whether each new edge connects a previously disconnected pair of vertices.

Simple balanced-tree solution

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as a balanced tree (ignoring keys). Define representative element $rep[S_i]$ to be the root of the tree.

- MAKE-SET(x) initializes x as a lone node. – $\Theta(1)$
- FIND-SET(x) walks up the tree containing x until it reaches the root. – $\Theta(\lg n)$
- UNION(x, y) concatenates the trees containing x and y, changing rep. – $\Theta(\lg n)$

$S_i = \{x_1, x_2, x_3, x_4, x_5\}$

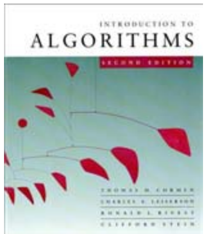


Plan of attack

We will build a simple disjoint-union data structure that, in an amortized sense, performs significantly better than $\Theta(\lg n)$ per op., even better than $\Theta(\lg \lg n)$, $\Theta(\lg \lg \lg n)$, etc., but not quite $\Theta(1)$.

To reach this goal, we will introduce two key ***tricks***. Each trick converts a trivial $\Theta(n)$ solution into a simple $\Theta(\lg n)$ amortized solution. Together, the two tricks yield a much better solution.

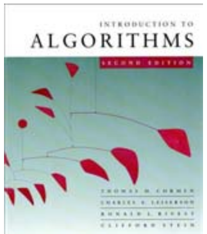
First trick arises in an augmented linked list. Second trick arises in a tree structure.



Each element x_j stores pointer $rep[x_j]$ to $rep[S_i]$.

UNION(x, y)

- concatenates the lists containing x and y, and
- updates the *rep* pointers for all elements in the list containing y.



Analysis of Trick 2 alone

Theorem: Total cost of FIND-SET's is $O(m \lg n)$.

Proof: Amortization by potential function.

The ***weight*** of a node x is # nodes in its subtree.

Define $\phi(x_1, \ldots, x_n) = \Sigma_i \lg weight[x_i]$.

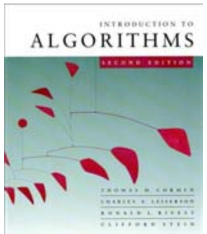
UNION(x_i, x_j) increases potential of root FIND-SET(x_i) by at most $\lg weight[\text{root FIND-SET}(x_j)] \le \lg n$.

Each step down $p \to c$ made by FIND-SET(x_i), except the first, moves c's subtree out of p's subtree. Thus if $weight[c] \ge ½\, weight[p]$, ϕ decreases by ≥ 1, paying for the step down. There can be at most $\lg n$ steps $p \to c$ for which $weight[c] < ½\, weight[p]$. ∎

Analysis of Trick 2 alone

Theorem: If all UNION operations occur before all FIND-SET operations, then total cost is O(m).

Proof: If a FIND-SET operation traverses a path with k nodes, costing O(k) time, then $k - 2$ nodes are made new children of the root. This change can happen only once for each of the n elements, so the total cost of FIND-SET is O($f + n$). □



UNION(x, y)

- Every tree has a *rank*
- Rank is an upper bound for height
- When we take UNION(x, y):
  - If rank[x] >rank[y] then link y to x
  - If rank[x] <rank[y] then link x to y
  - If rank[x]=rank[y] then
    - link x to y
    - rank[y]=rank[y]+1

- Can show that $2^{\text{rank}(x)} \leq$ #elements in x (Exercise 21.4-2)
- Therefore, height is O(log n)