CMP461: Algorithms

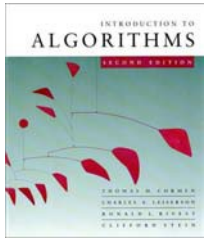
Lecture 11: Amortized Analysis and Disjoint Sets

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Fall 2013

Agenda

- Amortized Analysis
  - Aggregate Method
  - Accounting Method
- Disjoint Sets Data Structures (Union-Find)
  - Applications
  - Simple Linked List
  - Augmented Linked List

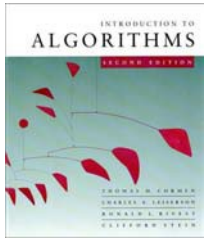
Acknowledgment
A lot of slides adapted from the slides of Erik Demaine, Charles Leiserson, and Piotr Indyk.



How large should a hash table be?

Goal: Make the table as small as possible, but large enough so that it won't overflow (or otherwise become inefficient).

Problem: What if we don't know the proper size in advance?



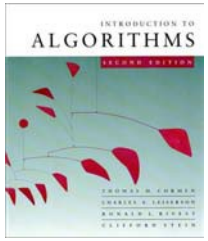
How large should a hash table be?

Goal: Make the table as small as possible, but large enough so that it won't overflow (or otherwise become inefficient).

Problem: What if we don't know the proper size in advance?

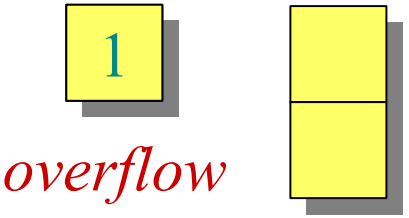
Solution: *Dynamic tables.*

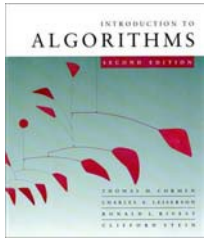
IDEA: Whenever the table overflows, "grow" it by allocating (via **`malloc`** or **`new`**) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.



Example of a dynamic table

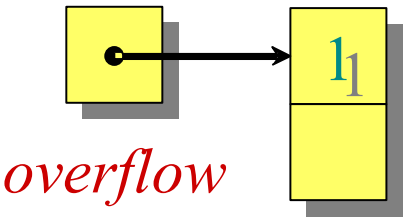
1. INSERT
2. INSERT

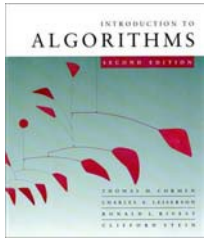




Example of a dynamic table

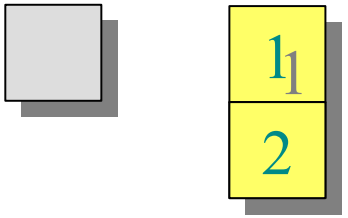
1. INSERT
2. INSERT

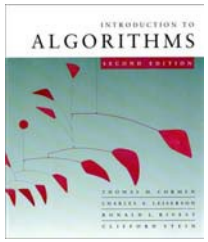




Example of a dynamic table

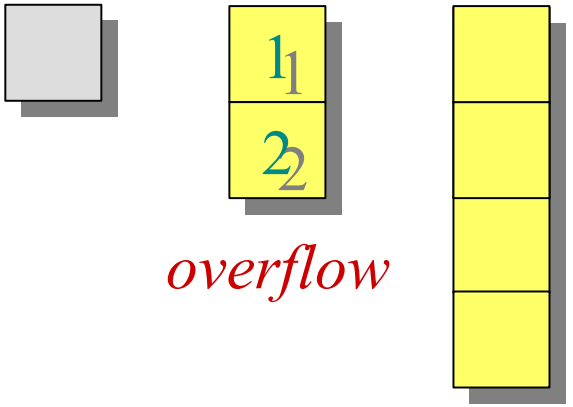
1. INSERT
2. INSERT

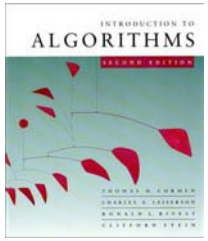




Example of a dynamic table

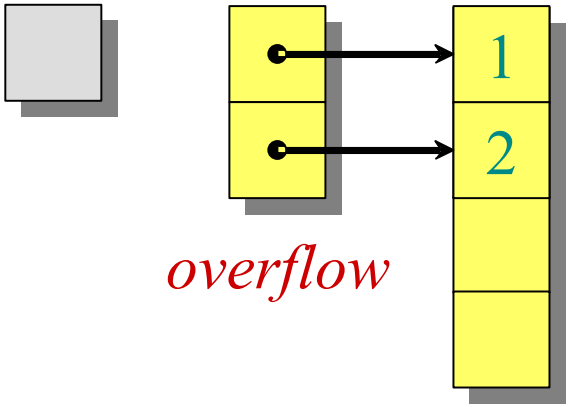
1. INSERT
2. INSERT
3. INSERT

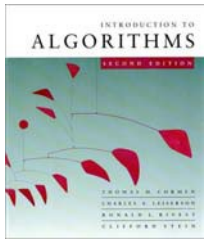




Example of a dynamic table

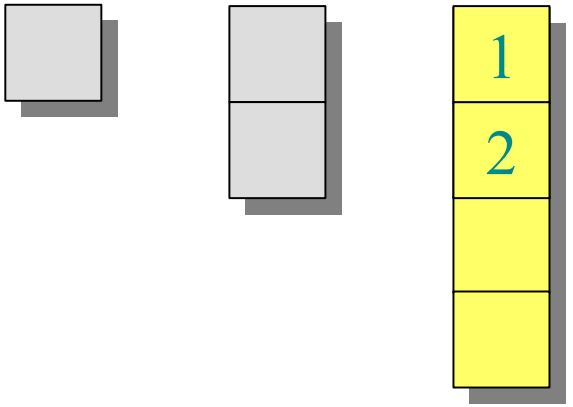
1. INSERT
2. INSERT
3. INSERT

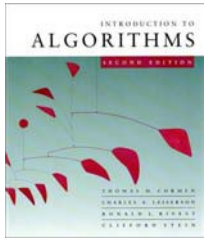




Example of a dynamic table

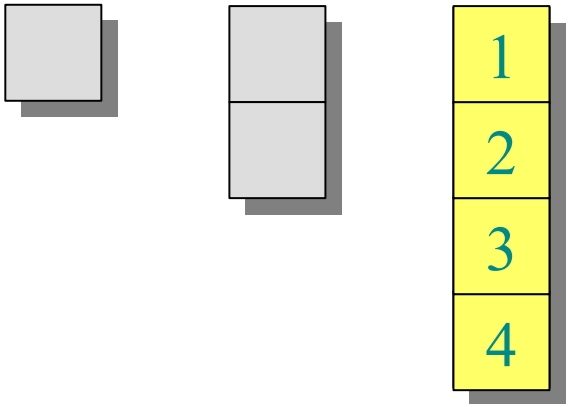
1. INSERT
2. INSERT
3. INSERT

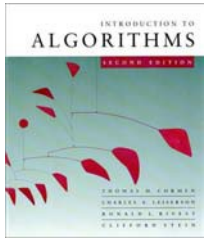




Example of a dynamic table

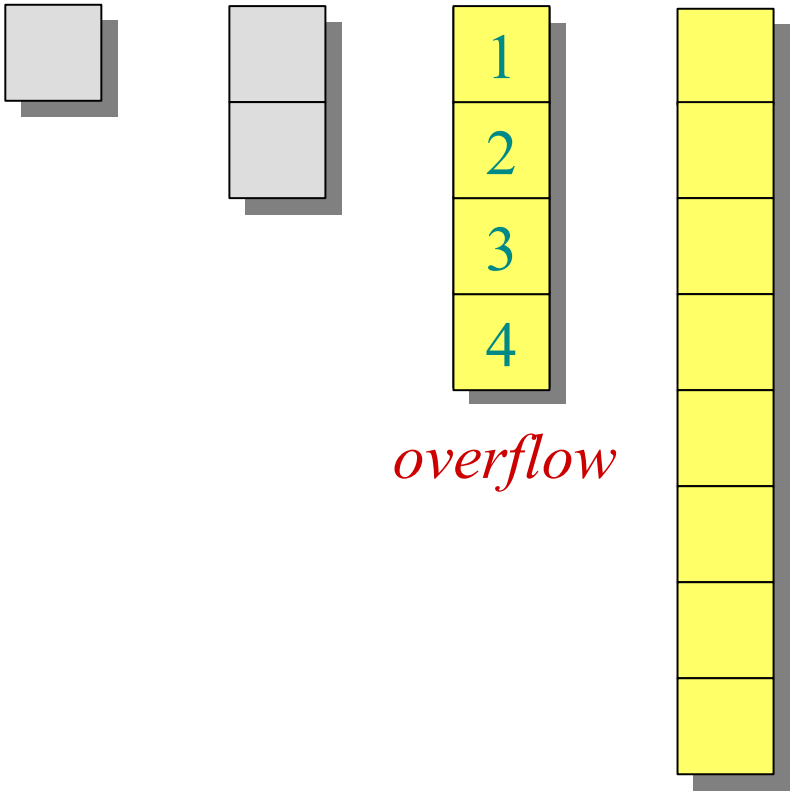
1. INSERT
2. INSERT
3. INSERT
4. INSERT

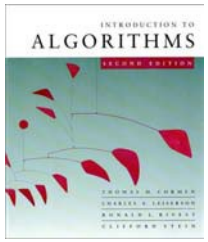




Example of a dynamic table

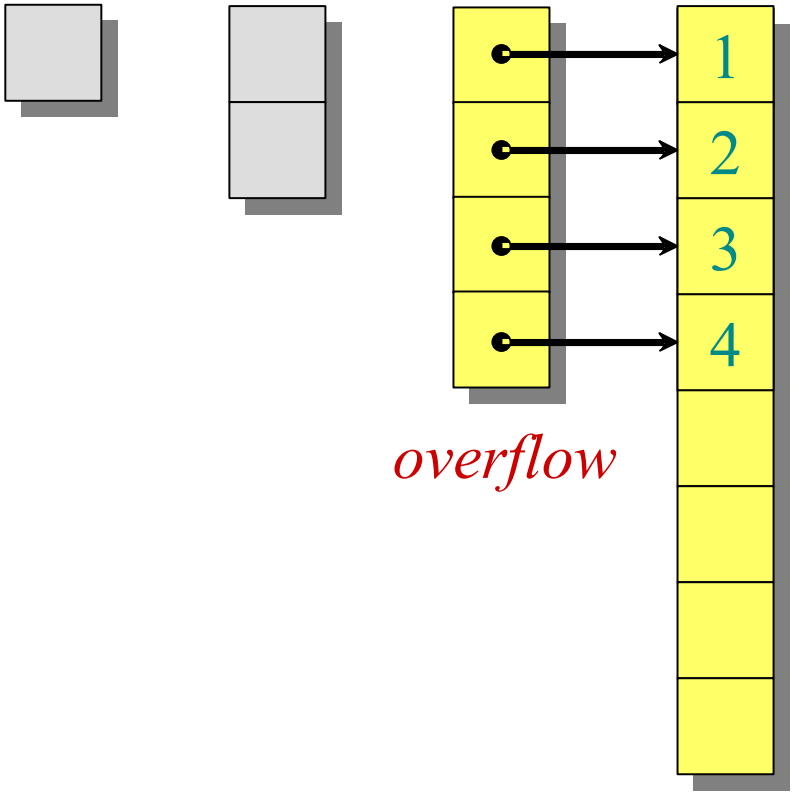
1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

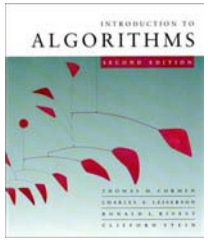




Example of a dynamic table

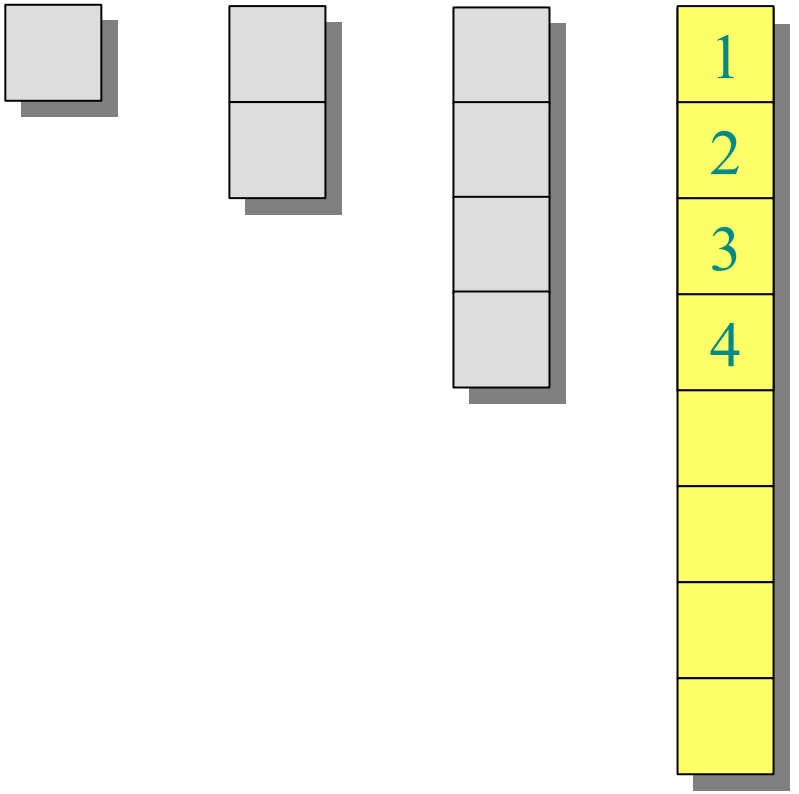
1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

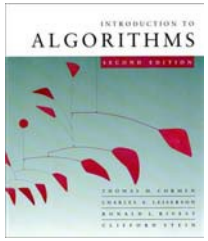




Example of a dynamic table

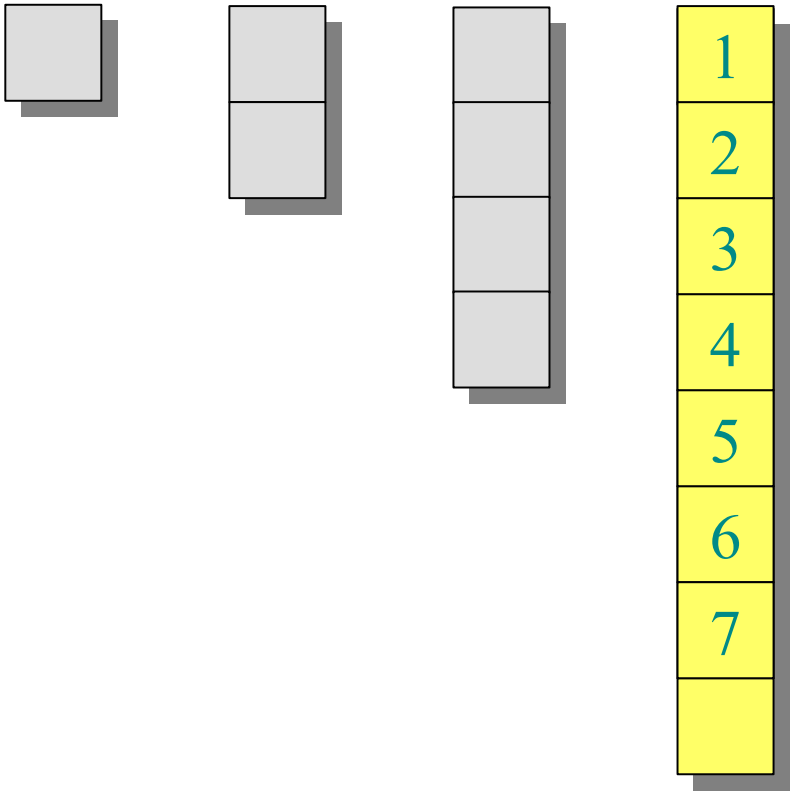
1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

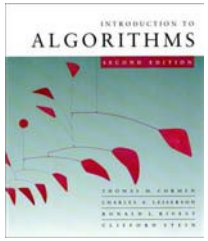




Example of a dynamic table

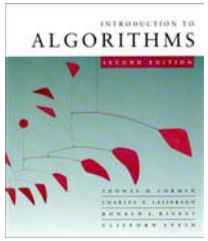
1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT





Worst-case analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

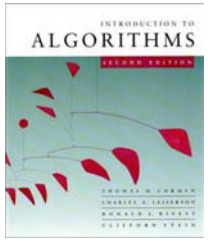


Worst-case analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

WRONG! In fact, the worst-case cost for n insertions is only $\Theta(n) < \Theta(n^2)$.

Let's see why.

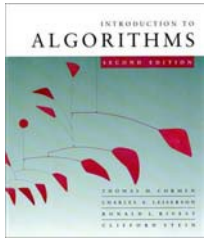


Tighter analysis

Let $c_i =$ the cost of the ith insertion

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16

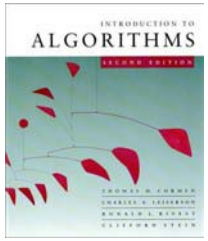


Tighter analysis

Let $c_i =$ the cost of the ith insertion

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

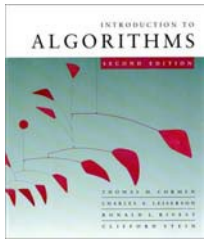


Tighter analysis

Let $c_i =$ the cost of the ith insertion

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
	1	1	1	1	1	1	1	1	1	1
c_i		1	2		4				8	



Tighter analysis (continued)

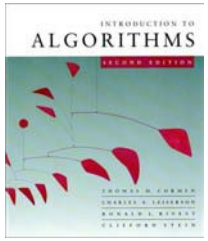
$$\text{Cost of } n \text{ insertions} = \sum_{i=1}^{n} c_i$$

$$\leq n + \sum_{j=0}^{\lg(n-1)} 2^j$$

$$\leq 3n$$

$$= \Theta(n) .$$

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

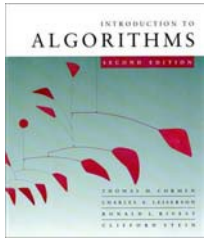


Amortized analysis

An *amortized analysis* is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

Even though we're taking averages, however, probability is not involved!

- An amortized analysis guarantees the average performance of each operation in the *worst case*.



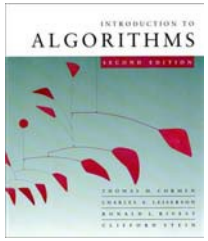
Types of amortized analyses

Three common amortization arguments:

- the *aggregate* method,
- the *accounting* method,
- the *potential* method.

We've just seen an aggregate analysis.

We will only discuss the *accounting* method next, and will not cover the *potential* method here.

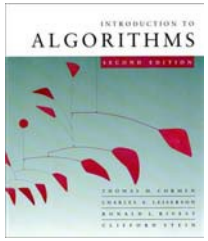


Accounting method

- Charge ith operation a fictitious *amortized cost* $\hat{c}_i$ where \$1 pays for 1 unit of work (*i.e.*, time).
- 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 must not go negative! We must ensure that

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

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

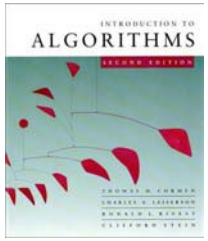


Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

- $\$1$ pays for the immediate insertion.
- $\$2$ is stored for later table doubling.

When the table doubles, $\$1$ pays to move a recent item, and $\$1$ pays to move an old item.



Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

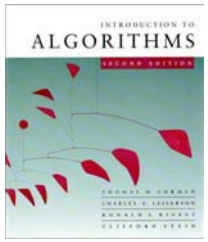
- $1 pays for the immediate insertion.
- $2 is stored for later table doubling.

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:

overflow

Situation just after inserting 8 elements in the table



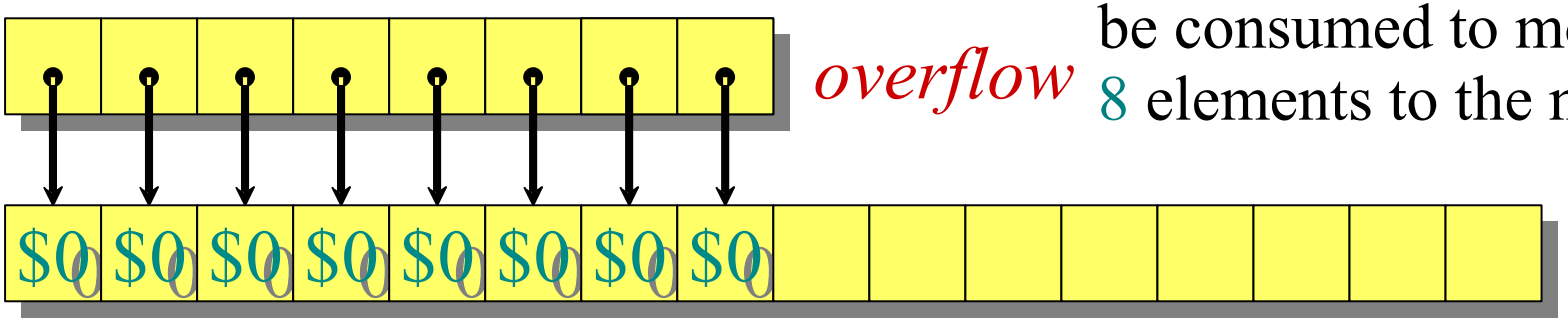
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

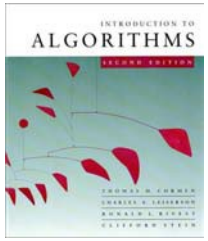
- $\$1$ pays for the immediate insertion.
- $\$2$ is stored for later table doubling.

When the table doubles, $\$1$ pays to move a recent item, and $\$1$ pays to move an old item.

Example:



The money in the bank will be consumed to move the 8 elements to the new table



Accounting analysis of dynamic tables

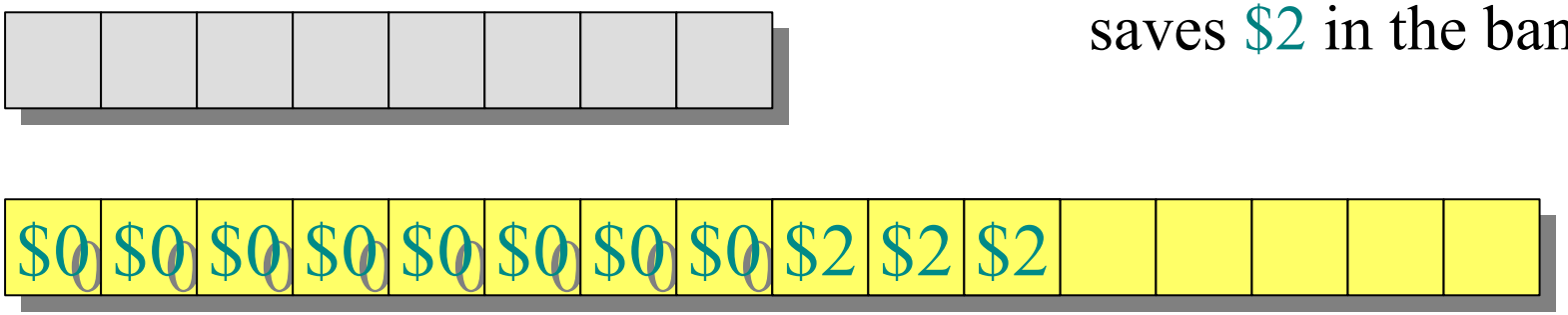
Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

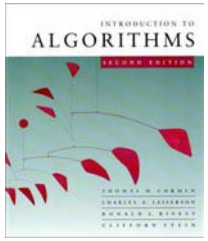
- $1 pays for the immediate insertion.
- $2 is stored for later table doubling.

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:

Each new element inserted saves $2 in the bank





Accounting analysis (continued)

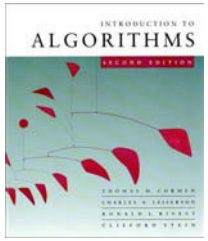
Key invariant: Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2*	3	3	3	3	3	3	3	3	3
$bank_i$	1	2	2	4	2	4	6	8	2	4

*Okay, so I lied. The first operation costs only \$2, not \$3.

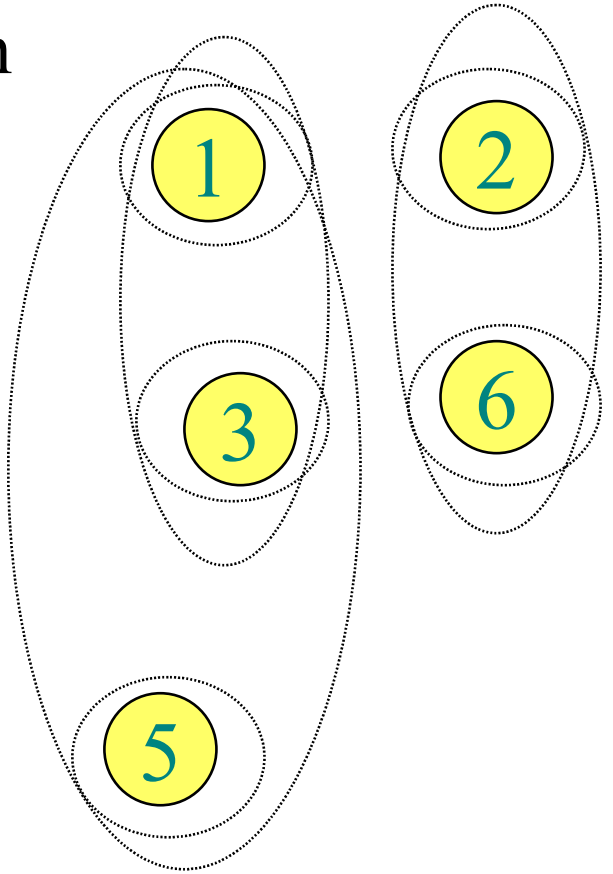
Amortized Analysis Summary

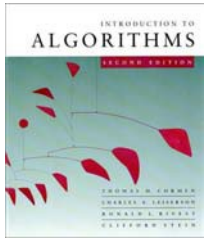
- Computes the total cost of any sequence of *n* operations to find the *worst-case* average cost per operation
- Two methods:
  - *Aggregate* method: computes the total worst-case cost of a sequence of operations
  - *Accounting* method: charge amortized cost per operation and store extra in the bank, making sure it never goes bust
- See **CLRS** Ch. 17 for more examples



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

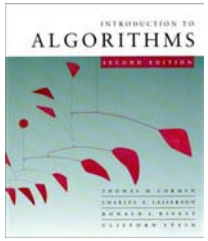




Disjoint-set data structure (Union-Find)

Problem:

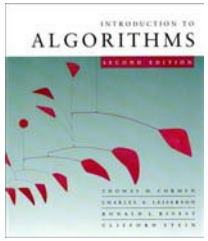
- Maintain a collection of *pairwise-disjoint* sets $\mathcal{S} = \{S_1, S_2, \ldots, S_r\}$.
- Each S_i has one representative element $x = rep[S_i]$.
- Must support three operations:
  - MAKE-SET(x): adds new set $\{x\}$ to $\mathcal{S}$ with $rep[\{x\}] = x$ (for any $x \notin S_i$ for all i).
 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



Disjoint-set data structure (Union-Find)

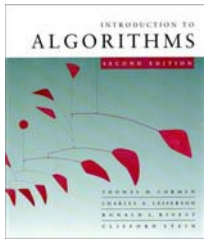
Problem:

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



Quiz

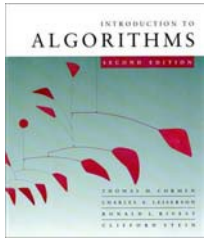
- If we have a WEAKUNION(x, y) that works only if x, y are representatives, how can we implement UNION that works for *any* x, y ?



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))



Application: Dynamic connectivity

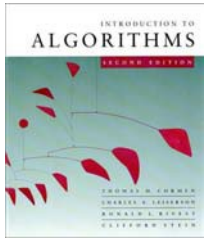
Suppose a graph is given to us *incrementally* by

- 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

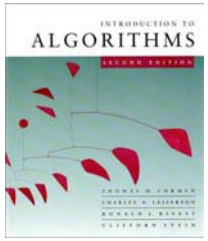
Sets of vertices represent *connected components*.
Suppose a graph is given to us *incrementally* by

- ADD-VERTEX(v) – MAKE-SET(v)
- ADD-EDGE(u, v) – if not CONNECTED(u, v)
  then UNION(u, v)

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?

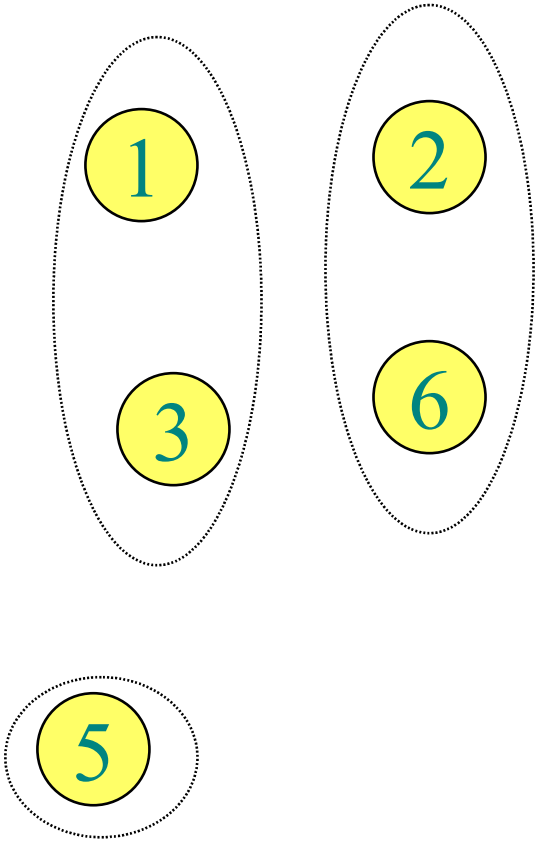
For example, we want to maintain a spanning forest, so we check whether each new edge connects a previously disconnected pair of vertices.

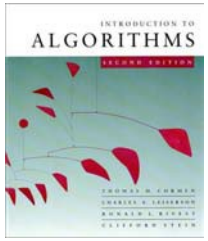


Applications

- Data clustering
- Killer App: Minimum Spanning Tree

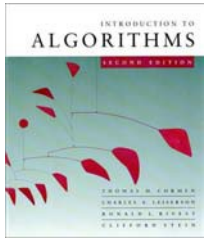
  Kruskal's Algorithm
- Amortized analysis





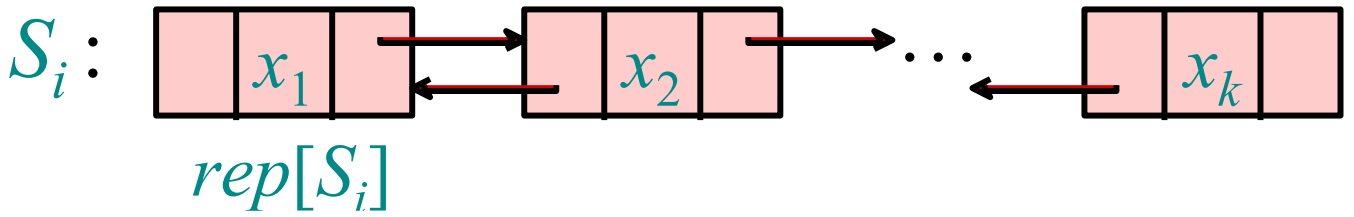
Ideas ?

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



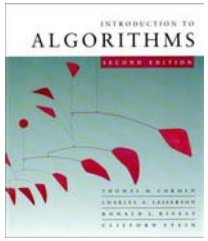
Simple linked-list solution

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.



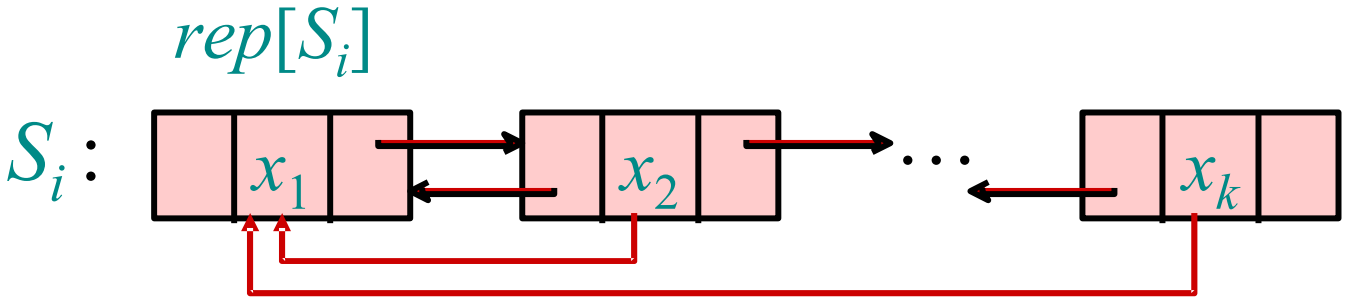
- Make-Set(x): initialize x as a lone node — $\Theta(1)$
- Find-Set(x): walks left in the list containing x until it reaches the head — $\Theta(n)$
- Union(x, y): concatenates the lists containing x and y leaving *rep.* as Find-Set(x) — $\Theta(n)$

Improvements?

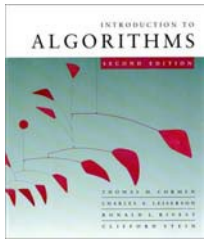


Augmented linked-list solution

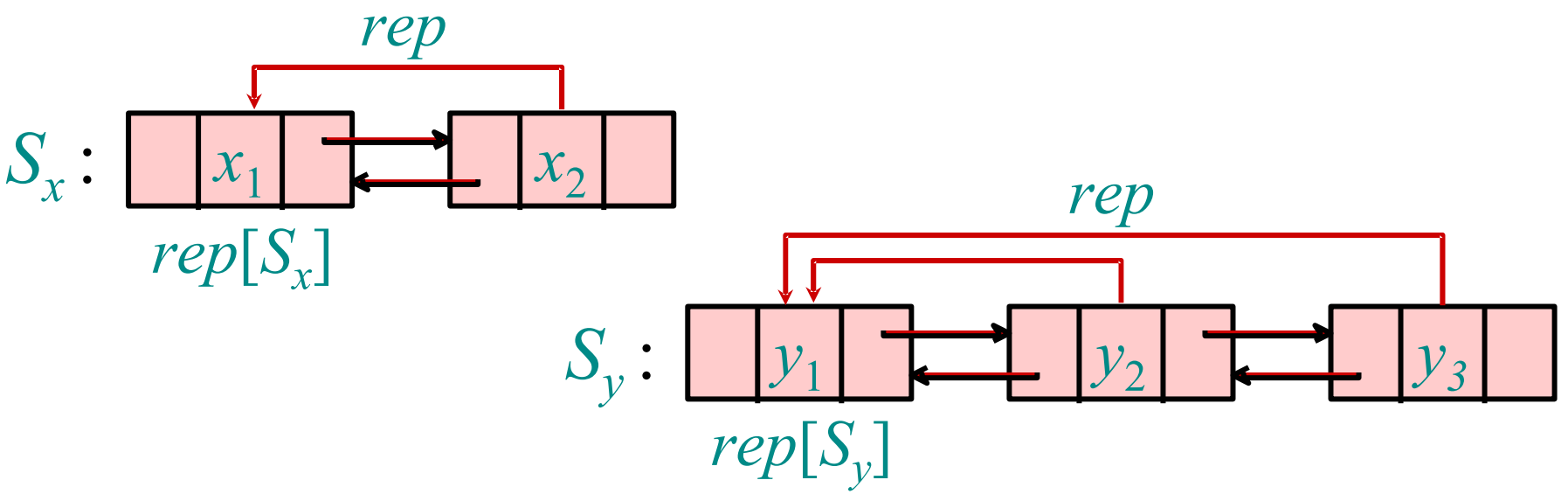
Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as unordered doubly linked list. Each x_j also stores pointer $rep[x_j]$ to head.

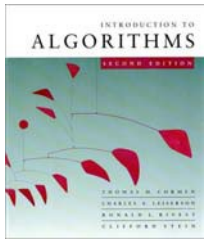


- FIND-SET(x) returns $rep[x]$.
- UNION(x, y) concatenates the lists containing x and y, and updates the rep pointers for all elements in the list containing y.



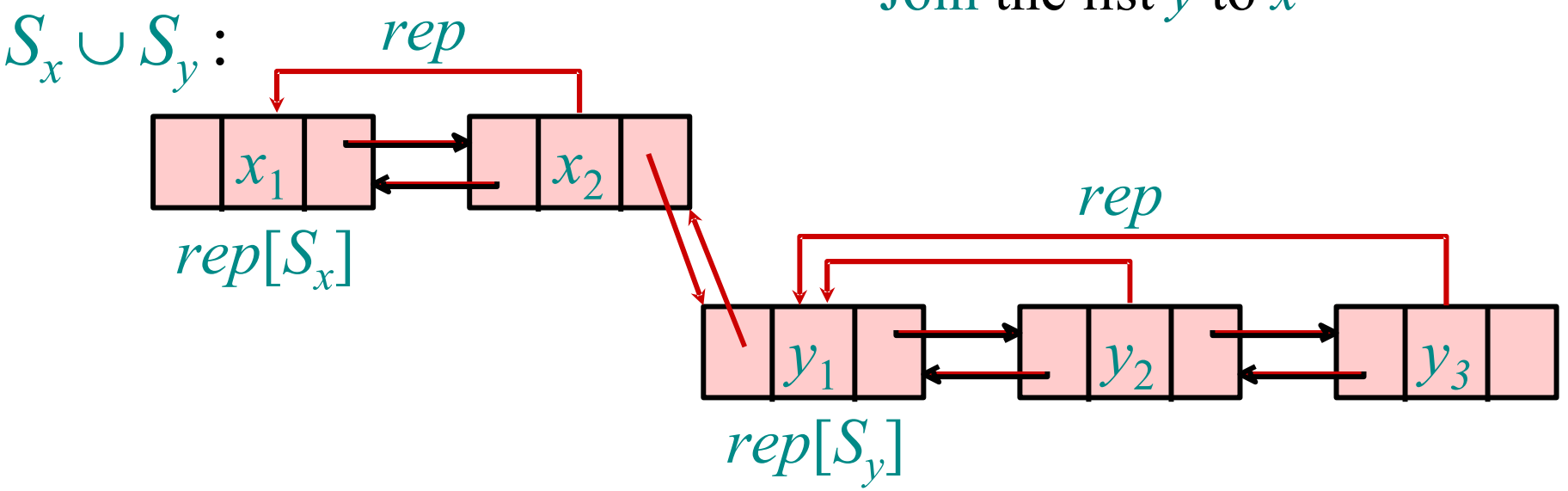
Example of augmented linked-list solution

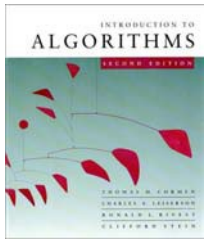




Example of augmented linked-list solution

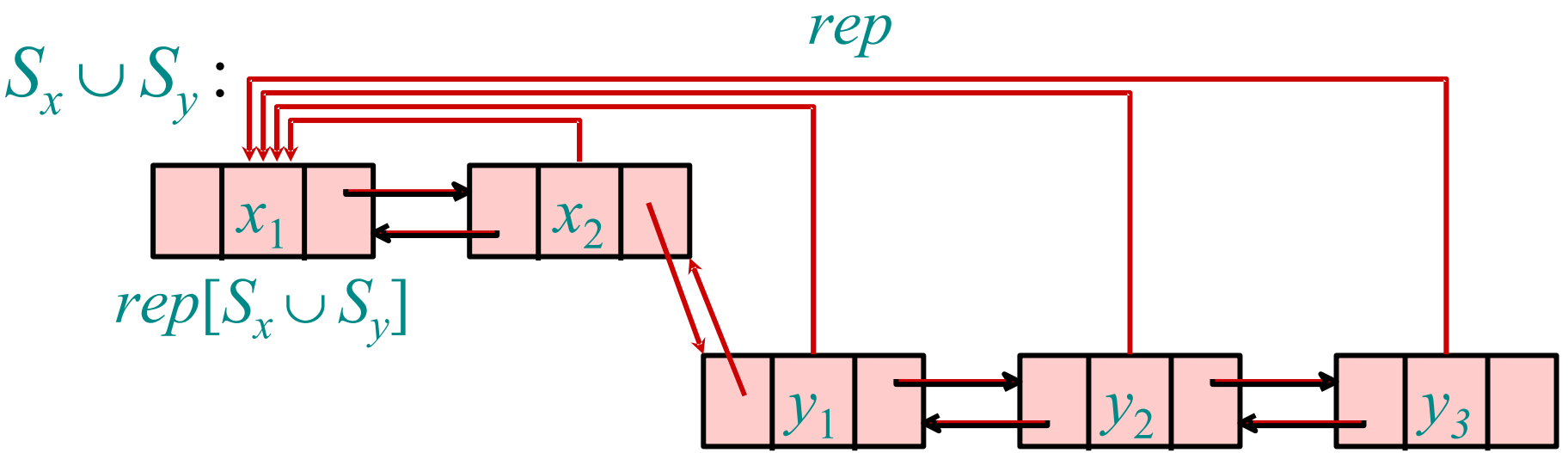
Join the list y to x

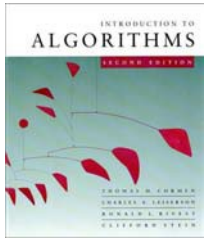




Example of augmented linked-list solution

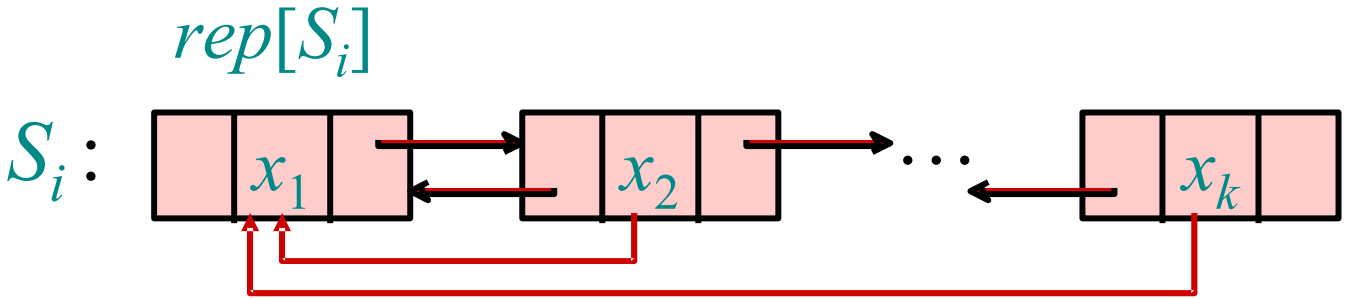
Update *rep* of the elements of y



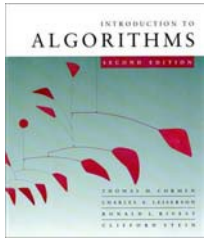


Augmented linked-list solution

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.

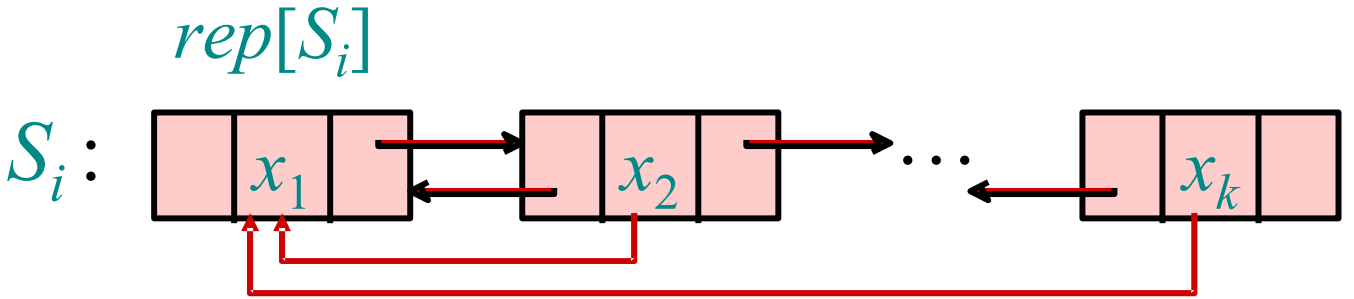


- FIND-SET(x) returns $rep[x]$. $\Theta(1)$
- UNION(x, y) concatenates the lists containing x and y, and updates the rep pointers for all elements in the list containing y. $\Theta(n)$

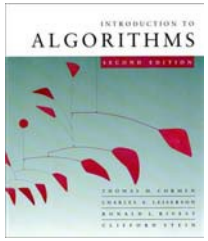


Augmented linked-list solution

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. $\Theta(n)$?

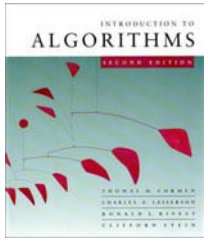


Amortized analysis

- 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 $1,1,1,\ldots,1,n,1,\ldots,1$
- 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?

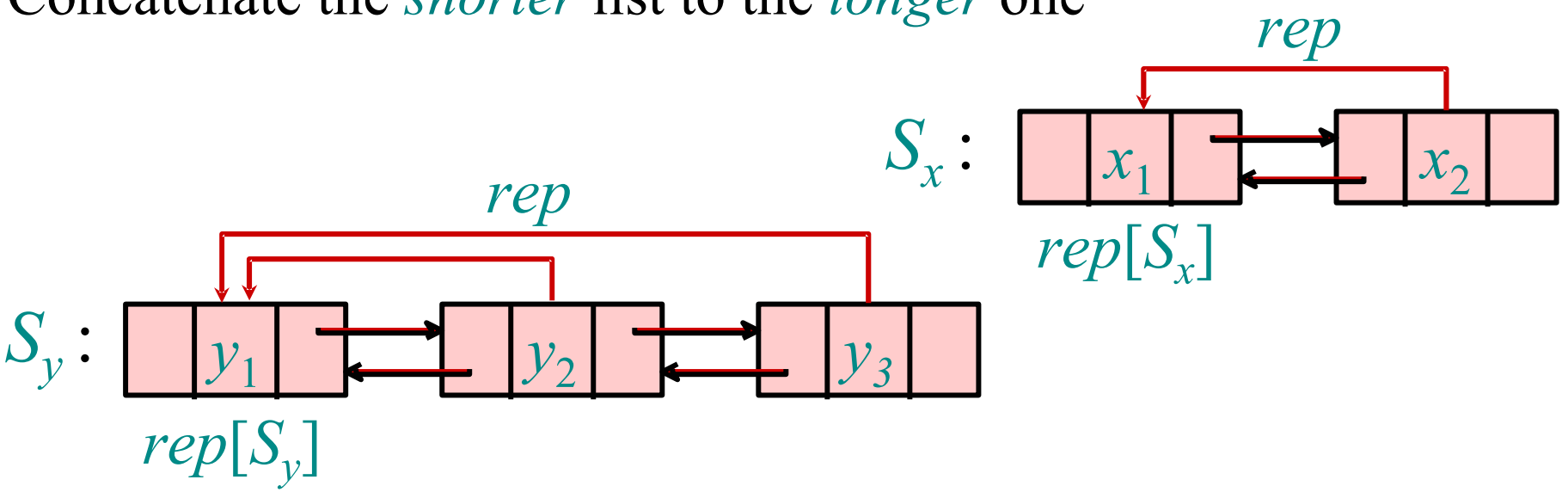


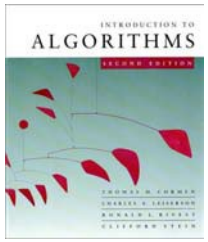
Alternative

UNION(x, y) :

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

Concatenate the *shorter* list to the *longer* one



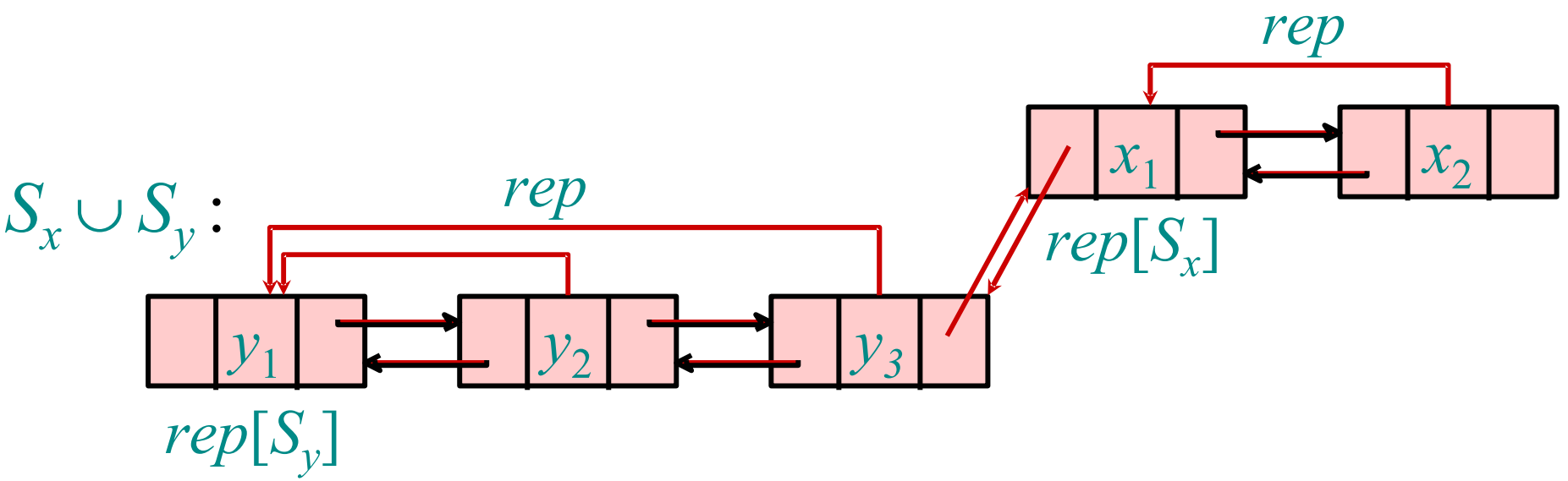


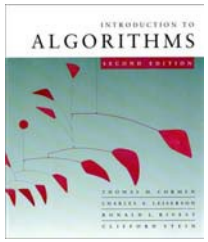
Alternative concatenation

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.

Join the list x to y

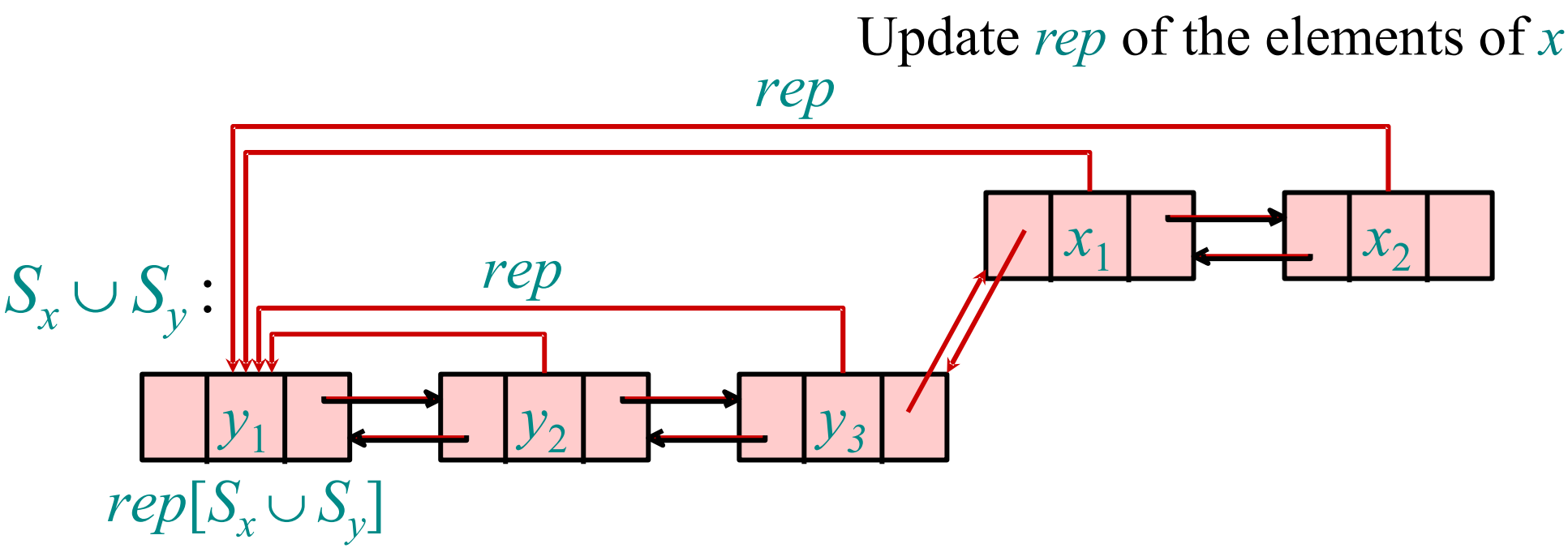


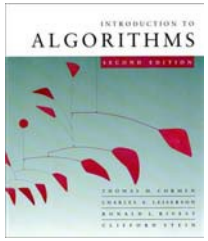


Alternative concatenation

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

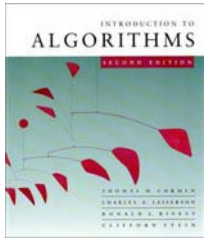
- Concatenate smaller list onto the end of the larger list (each list stores its *weight* = # elements)

- 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 $O(n \lg n)$.

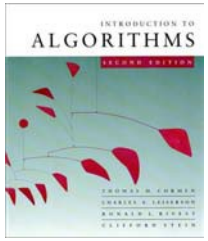
Corollary: Total cost is $O(m + n \lg n)$.



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

Proof:

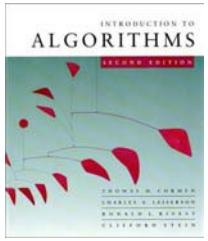
- Monitor an element x and set S_x containing it
- After initial MAKE-SET(x), $weight[S_x] = 1$
- Consider any time when S_x is merged with set S_y
  - If $weight[S_y] \geq weight[S_x]$
    - pay 1 to update $rep[x]$
    - $weight[S_x]$ at least doubles (increasing by $weight[S_y]$)
  - Otherwise
    - 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)

Disjoint Sets Summary

- Maintain a dynamic data structure of disjoint sets
- Applications
  - Clustering
  - MST
  - Connected Components
- Linked list implementation
  - $O(n)$ amortized cost with simple linked list
  - $O(\lg n)$ amortized cost with minimum weight heuristic
- See **CLRS** Ch. 21 for more information, and a faster implementation using *trees*

Recap

- Amortized Analysis
  - Aggregate Method
  - Accounting Method
- Disjoint Sets Data Structures (Union-Find)
  - Applications
  - Simple Linked List
  - Augmented Linked List