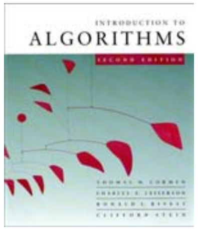
CS 5633 -- Spring 2006



Union-Find Data Structures

Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk

---

Disjoint-set data structure (Union-Find)

Problem:

- Maintain a dynamic collection of *pairwise-disjoint* sets $\mathsf{S} = \{S_1, S_2, \ldots, S_r\}$.
- Each set S_i has one element distinguished as the representative element, $rep[S_i]$.
- Must support 3 operations:
  - MAKE-SET(x): adds new set $\{x\}$ to 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 S (for any x, y in distinct sets S_x, S_y)
  - FIND-SET(x): returns representative $rep[S_x]$ of set S_x containing element x

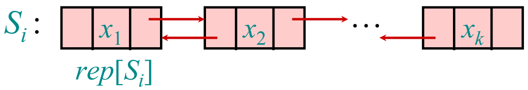
---

Disjoint-set data structure (Union-Find) II

- In all operations the elements x, y are given (as pointers or references for example)
- Hence, we do not need to first search for the element in the data structure.
- Let n denote the overall number of elements (equivalently, the number of MAKE-SET operations).

Simple linked-list solution

Store each 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.



$\Theta(1)$ • MAKE-SET(x) initializes x as a lone node.

$\Theta(n)$ • FIND-SET(x) walks left in the list containing x until it reaches the front of the list.

$\Theta(n)$ • UNION(x, y) calls FIND-SET on x and y and concatenates the lists containing x and y, leaving rep. as FIND-SET$[x]$.

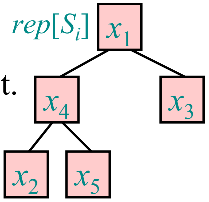
Simple balanced-tree solution

maintain how?

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.

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

- $\Theta(1)$ MAKE-SET(x) initializes x as a lone node.
- $\Theta(\log n)$ FIND-SET(x) walks up the tree containing x until reaching root.
- $\Theta(\log n)$ UNION(x, y) calls FIND-SET on x and y and concatenates the trees containing x and y, changing rep. of x or y

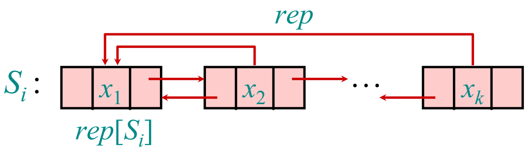


Plan of attack

- We will build a simple disjoint-union data structure that, in an **amortized sense**, performs significantly better than $\Theta(\log n)$ per op., even better than $\Theta(\log \log n)$, $\Theta(\log \log \log n)$, ..., 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 $\Theta(\log n)$ 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.

Augmented linked-list solution

Store $S_i = \{x_1, x_2, \ldots, x_k\}$ as unordered doubly linked list. **Augmentation:** Each element x_j also stores pointer $rep[x_j]$ to $rep[S_i]$ (which is the front of the list, x_1).

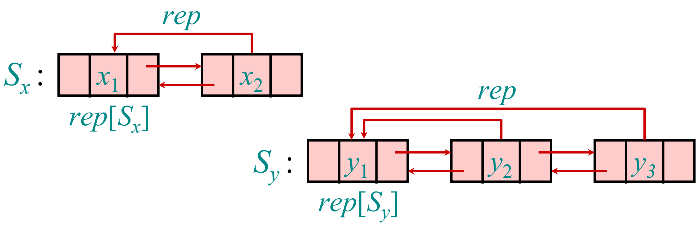


- FIND-SET(x) returns $rep[x]$. – $\Theta(1)$
- UNION(x, y) concatenates lists containing x and y and updates the rep pointers for all elements in the list containing y. – $\Theta(n)$

Example of augmented linked-list solution

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.

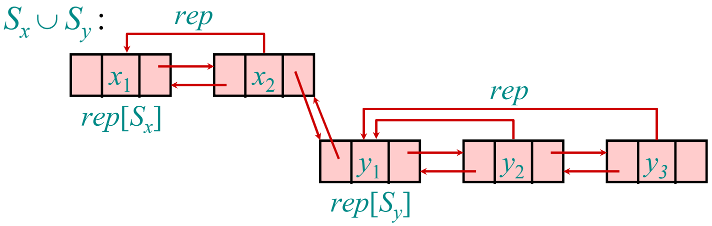


Example of augmented linked-list solution

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.

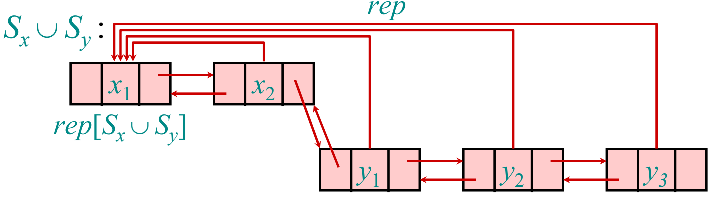


Example of augmented linked-list solution

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

UNION(x, y)

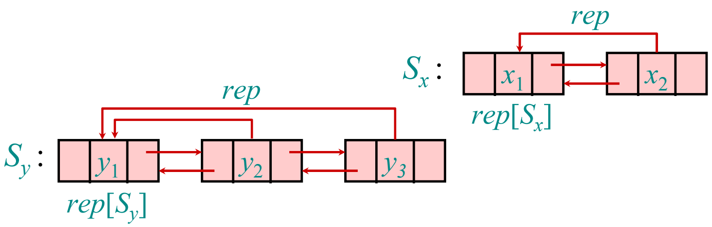
- concatenates the lists containing x and y, and
- updates the *rep* pointers for all elements in the list containing y.



Alternative concatenation

UNION(x, y) could instead

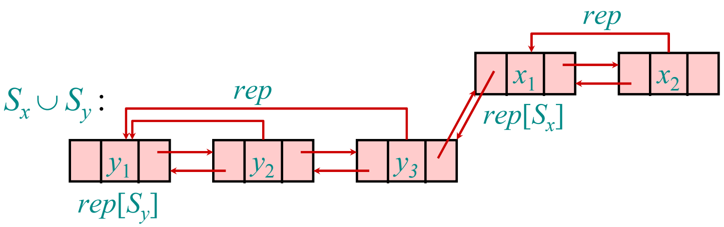
- concatenate the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing x.



Alternative concatenation

UNION(x, y) could instead

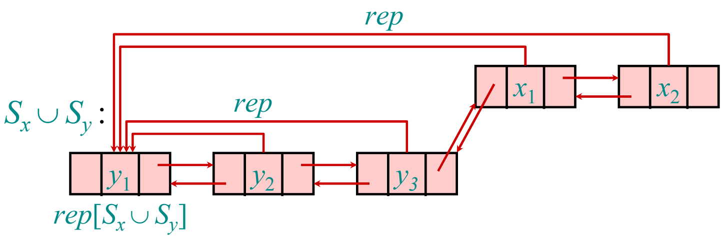
- concatenate the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing x.



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.



Trick 1: Smaller into larger

(weighted-union heuristic)

To save work, concatenate smaller list onto the end of the larger list. Cost = Θ(length of smaller list). Augment list to store its ***weight*** (# elements).

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

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

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

Analysis of Trick 1

(weighted-union heuristic)

Theorem: Total cost of UNION's is $O(n \log n)$.

Proof. • Monitor an element x and set S_x containing it.

- After initial MAKE-SET(x), $weight[S_x] = 1$.
- Each time S_x is united with S_y, $weight[S_y] \geq weight[S_x]$,
  - pay 1 to update $rep[x]$, and
  - $weight[S_x]$ at least doubles (increases by $weight[S_y]$).
- Each time S_x is united with smaller set S_y,
  - pay nothing, and
  - $weight[S_x]$ only increases.

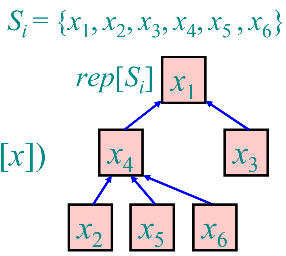
Thus pay $\leq \log n$ for x.

□

Disjoint set forest: Representing sets as trees

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an unordered, potentially unbalanced, not necessarily binary tree, storing only *parent* pointers. $rep[S_i]$ is the tree root.

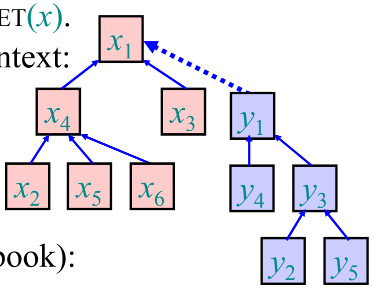
- MAKE-SET(x) initializes x as a lone node. – $\Theta(1)$
- FIND-SET(x) walks up the tree containing x until it reaches the root. – $\Theta(depth[x])$
- UNION(x, y) concatenates the trees containing x and y…



Trick 1 adapted to trees

- UNION(x, y) can use a simple concatenation strategy: Make root FIND-SET(y) a child of root FIND-SET(x). $\Rightarrow$ FIND-SET(y) = FIND-SET(x).
- Adapt Trick 1 to this context:

Union-by-weight:
Merge tree with smaller weight into tree with larger weight.



- Variant of Trick 1 (see book):

Union-by-rank:
rank of a tree = its height

Trick 1 adapted to trees
(union-by-weight)

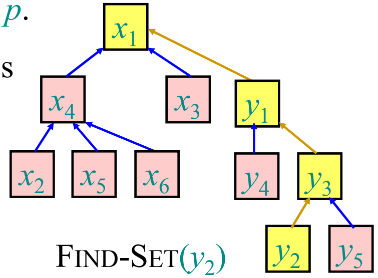
- Height of tree is logarithmic in weight, because:
  - Induction on the weight
  - Height of a tree T is determined by the two subtrees T_1, T_2 that T has been united from.
  - Inductively the heights of T_1, T_2 are the logs of their weights.
  - height(T) = max(height(T_1), height(T_2)) possibly +1, but only if T_1, T_2 have same height
- Thus total cost is $O(m \log n)$.

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 $\Theta(depth[x])$.



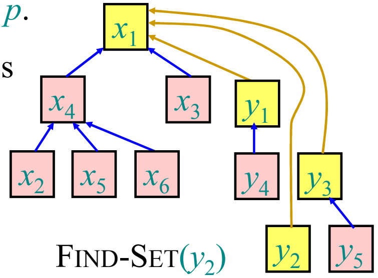
FIND-SET(y_2)

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 $\Theta(depth[x])$.



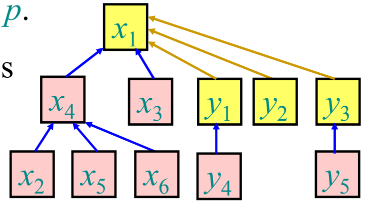
FIND-SET(y_2)

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)

Trick 2: Path compression

- Note that UNION(x,y) first calls FIND-SET(x) FIND-SET(y). Therefore path compression also affects UNION operations.

Analysis of Trick 2 alone

Theorem: Total cost of FIND-SET's is $O(m \log n)$.
Proof: By amortization. Omitted.

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 $O(m)$. ☐

Ackermann's function A, and it's "inverse" α

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 $j+1$ times

$A_0(j) = j + 1$
$A_1(j) \sim 2j$
$A_2(j) \sim 2j\,2^j > 2^j$
$A_3(j) > \left.2^{2^{2^{\cdot^{\cdot^{\cdot^{2^j}}}}}}\right\}j$
$A_4(j)$ is a lot bigger.

$A_0(1) = 2$
$A_1(1) = 3$
$A_2(1) = 7$
$A_3(1) = 2047$
$A_4(1) > \left.2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{2047}}}}}}}\right\}2048 \text{ times}$

Define $\alpha(n) = \min\,\{k : A_k(1) \geq n\} \leq 4$ for practical n.

Analysis of Tricks 1 + 2
for disjoint-set forests

Theorem: In general, total cost is $O(m\,\alpha(n))$.

(long, tricky proof – see Section 21.4 of CLRS)

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.