Disjoint-set data structure: Union-Find

Lecture 20

Disjoint-set data structure (Union-Find)

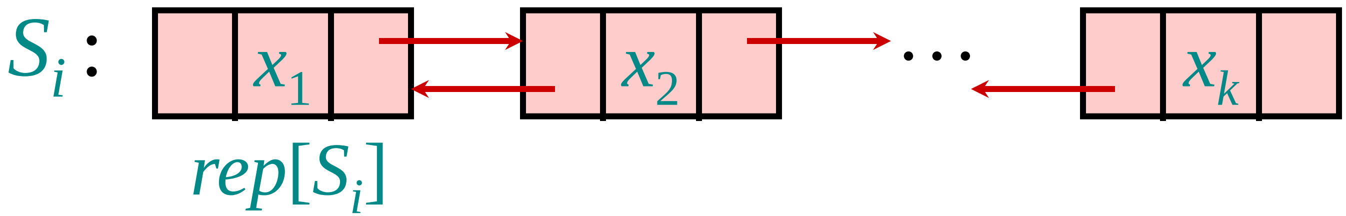
Problem: Maintain a dynamic collection of *pairwise-disjoint* sets $\mathsf{S} = \{S_1, S_2, \ldots, S_r\}$.
Each set S_i has one element distinguished as the representative element, $rep[S_\mathrm{i}]$.

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.

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.



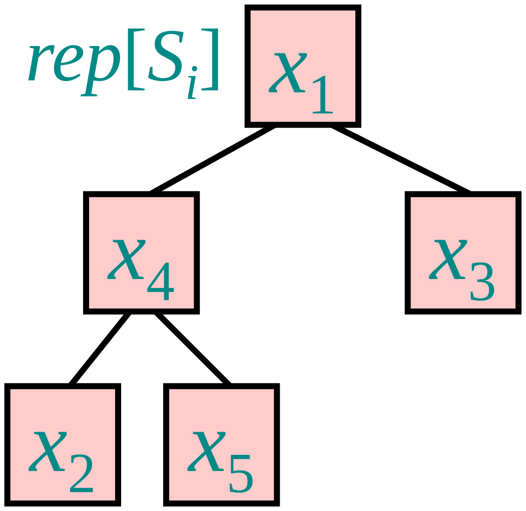
- MAKE-SET(x) initializes x as a lone node. – $\Theta(1)$
- FIND-SET(x) walks left in the list containing x until it reaches the front of the list. – $\Theta(n)$
- UNION(x, y) concatenates the lists containing x and y, leaving rep. as FIND-SET[x]. – $\Theta(n)$

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

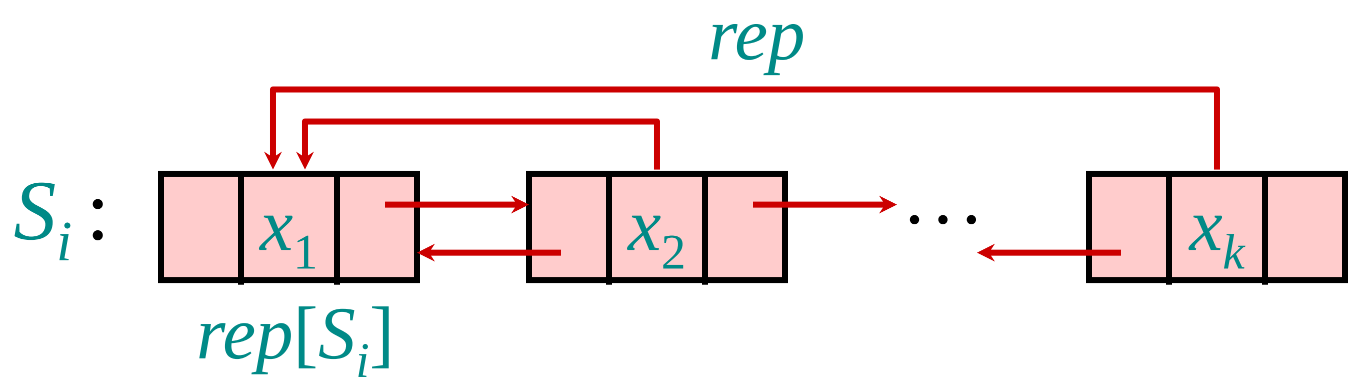
We will build a simple disjoint-union data structure that, in an amortized sense, performs significantly better than $\Theta(\lg n)$ per op., even better than $\Theta(\lg \lg n)$, $\Theta(\lg \lg \lg n)$, etc., but not quite $\Theta(1)$.

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.

Augmented linked-list solution

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



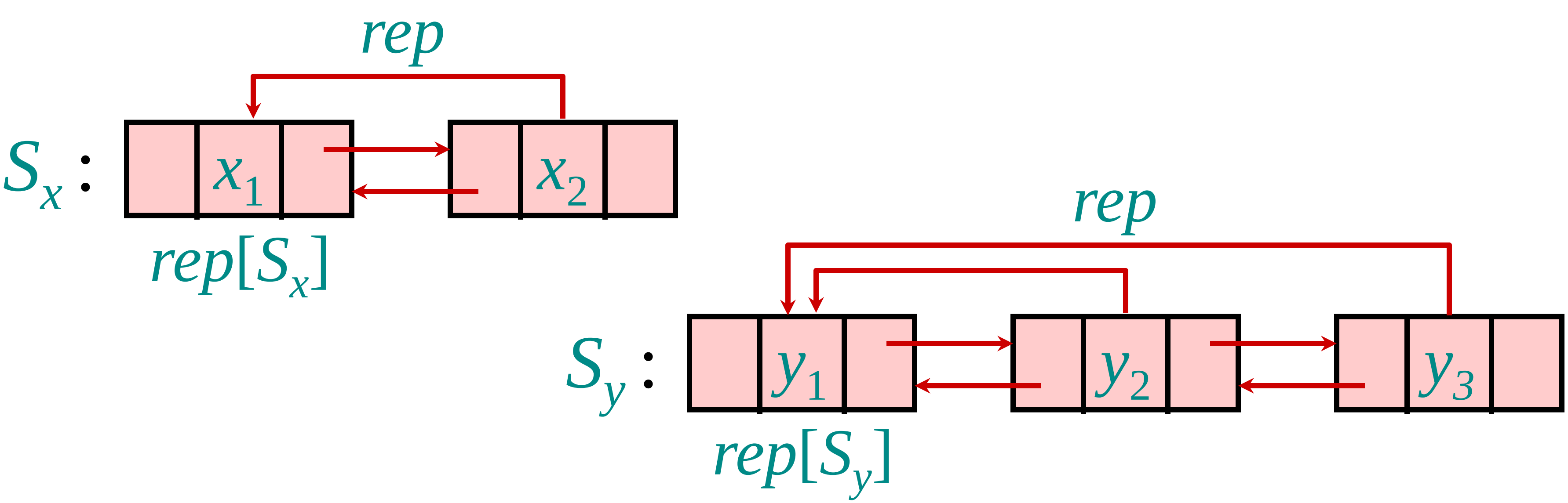
- FIND-SET(x) returns $rep[x]$. – $\Theta(1)$
- UNION(x, y) concatenates the lists containing x and y, and updates the rep pointers for all elements in the list containing y. – $\Theta(n)$

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.



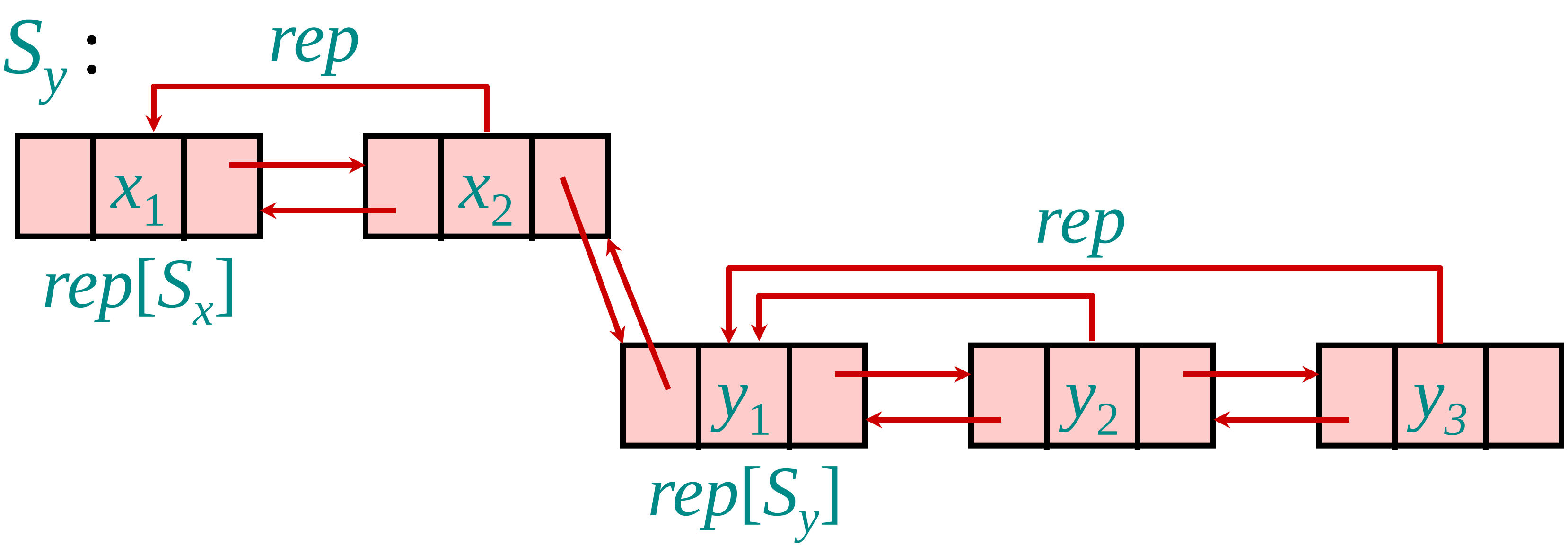
Example of augmented linked-list solution

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.

$S_x \cup S_y$:

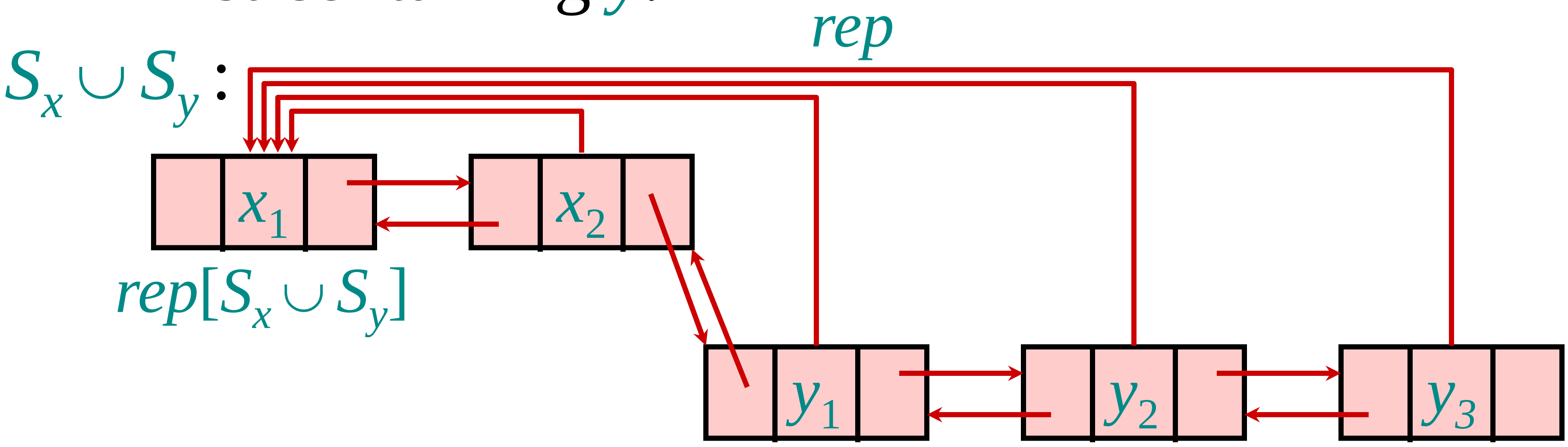


Example of augmented linked-list solution

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

UNION(x, y)

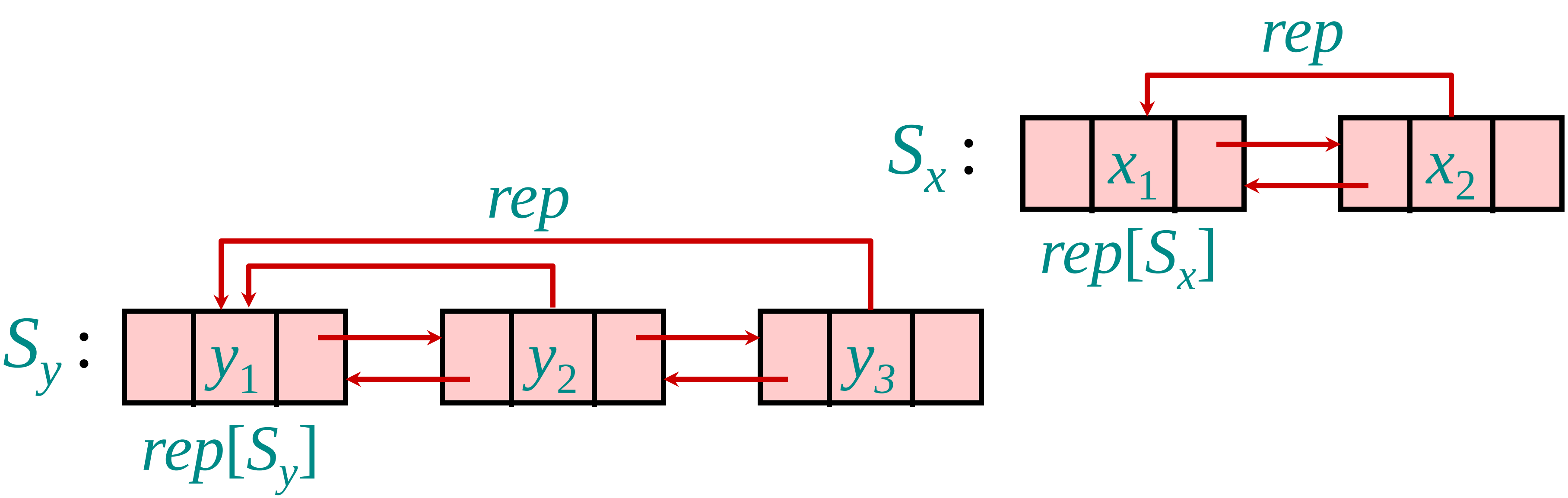
- concatenates the lists containing x and y, and
- updates the *rep* pointers for all elements in the list containing y.



Alternative concatenation

UNION(x, y) could instead

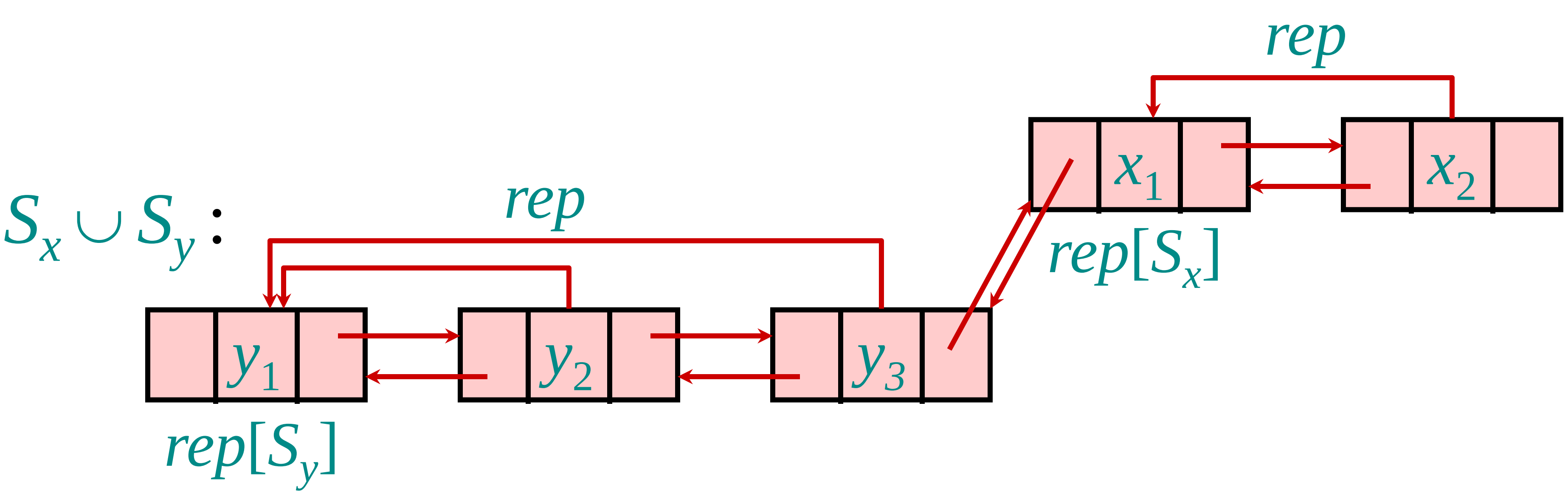
- concatenate the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing x.



Alternative concatenation

UNION(x, y) could instead

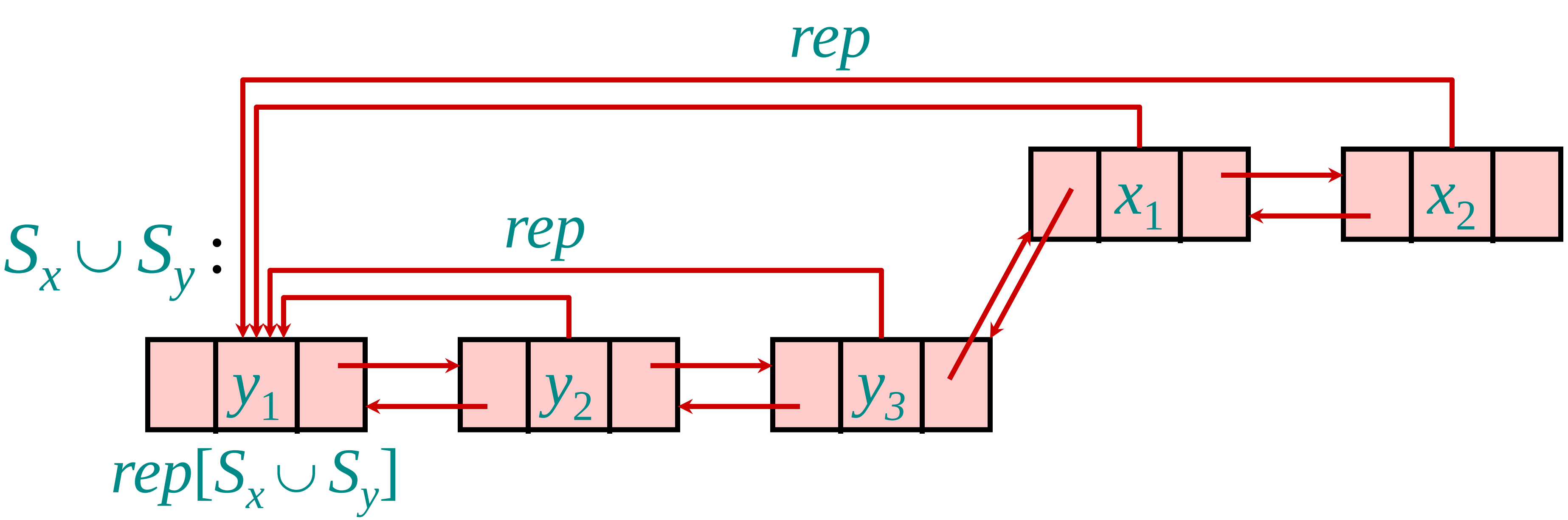
- concatenate the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing x.



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

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 $O(n \lg n)$.

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

Analysis of Trick 1

To save work, concatenate smaller list onto the end of the larger list. Cost $= \Theta(1 +$ length of smaller list$)$.

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

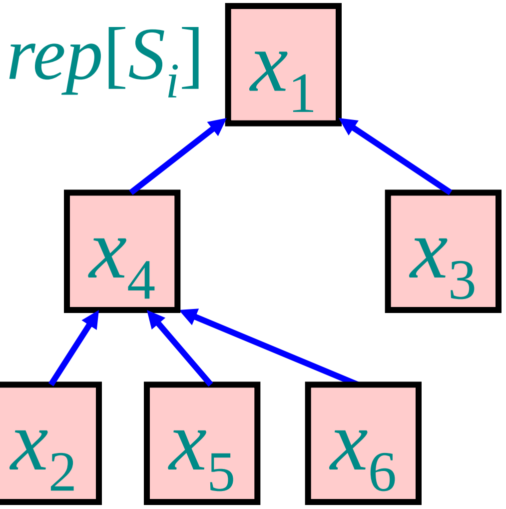
Proof. Monitor an element x and set S_x containing it. After initial MAKE-SET(x), $weight[S_x] = 1$. Each time S_x is united with set S_y, $weight[S_y] \geq weight[S_x]$, pay 1 to update $rep[x]$, and $weight[S_x]$ at least doubles (increasing by $weight[S_y]$). Each time S_y is united with smaller set S_y, pay nothing, and $weight[S_x]$ only increases. Thus pay $\leq \lg n$ for x. □

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.

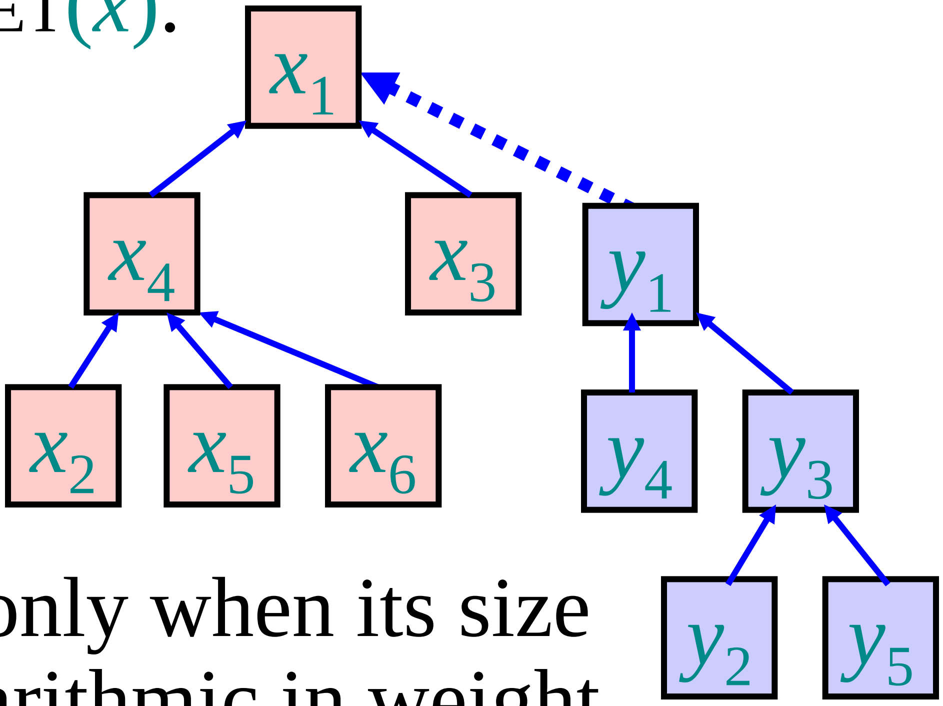
- MAKE-SET(x) initializes x as a lone node. – $\Theta(1)$
- FIND-SET(x) walks up the tree containing x until it reaches the root. – $\Theta(depth[x])$
- UNION(x, y) concatenates the trees containing x and y…

$S_i = \{x_1, x_2, x_3, x_4, x_5, x_6\}$



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).



We can adapt Trick 1 to this context also: Merge tree with smaller weight into tree with larger weight.

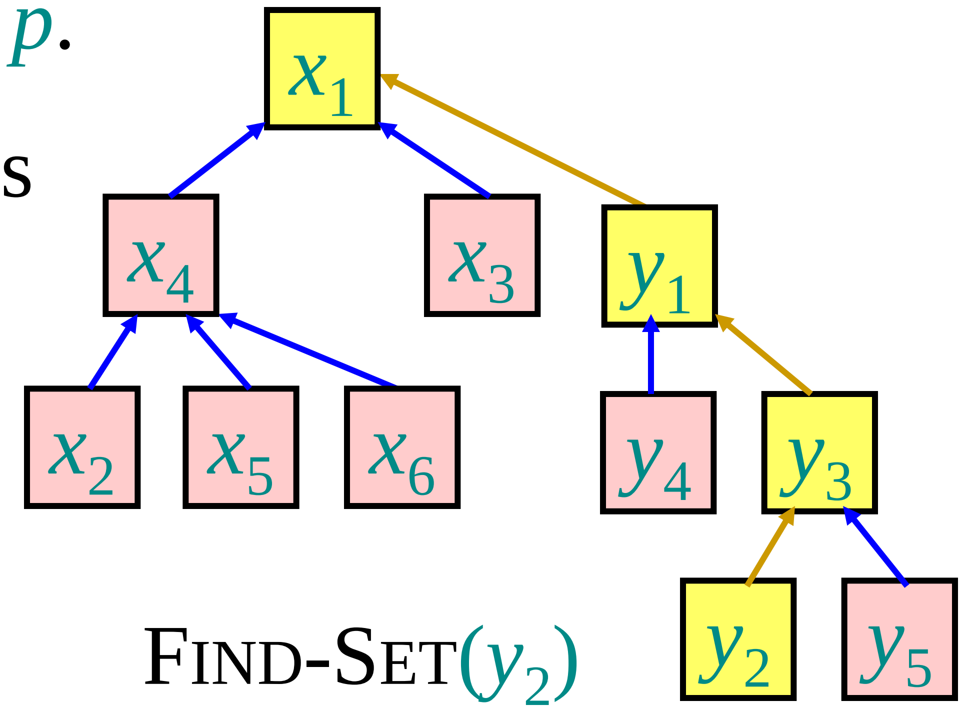
Height of tree increases only when its size doubles, so height is logarithmic in weight. Thus total cost is $O(m + f \lg n)$.

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 $\Theta(depth[x])$.

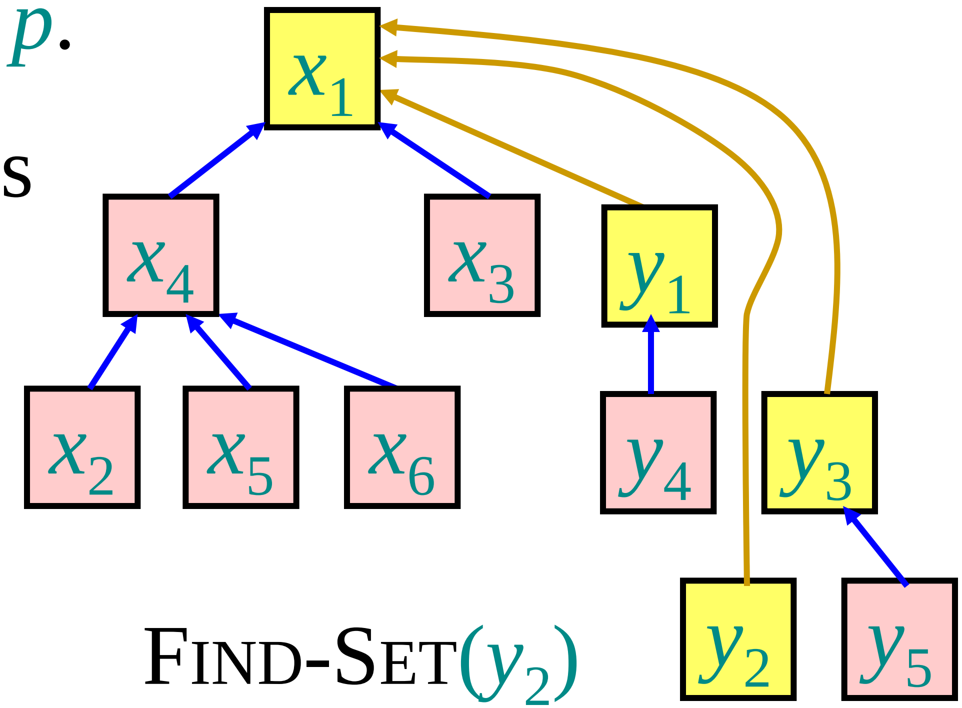


Trick 2: Path compression

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 $\Theta(depth[x])$.



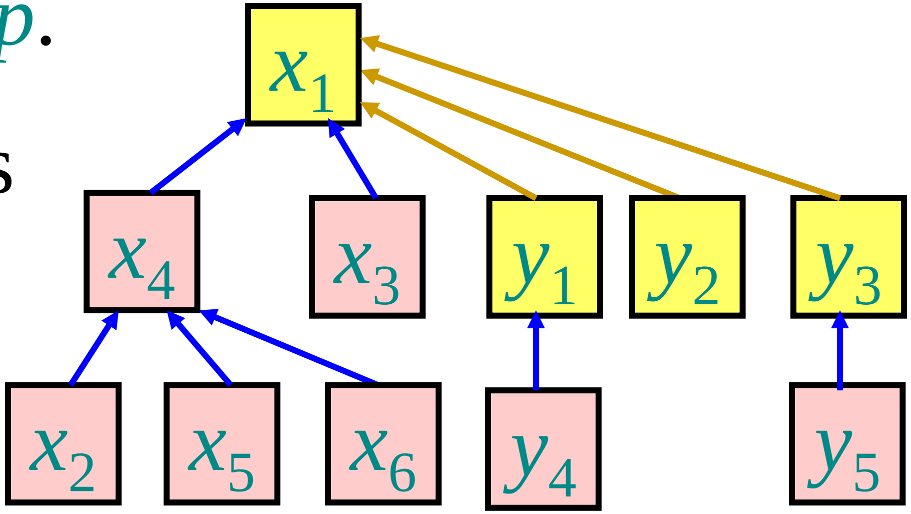
FIND-SET(y_2)

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)

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 $\lg weight[$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 $O(f + n)$. ◻

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 $j+1$ times

$A_0(j) = j + 1$ $\quad$ $A_0(1) = 2$

$A_1(j) \sim 2\,j$ $\quad$ $A_1(1) = 3$

$A_2(j) \sim 2j\,2^j > 2^j$ $\quad$ $A_2(1) = 7$

$A_3(1) = 2047$

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

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

$A_4(j)$ is a lot bigger.

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

Analysis of Tricks 1 + 2

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

(long, tricky proof – see the text book)

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.