# Lecture 14: April 2, 2002

Disjoint Set Data Structure

- Definition
- Linked-list implementation
- Forest implementation
- Speedups

## Disjoint Set Data Structure

Maintains a collection of disjoint sets $S_1 \dots S_n$, together with a *representative* from each set.

- $\textsc{Make-Set}(x)$: creates a new set $S = \{x\}$, the object $x$ becomes the representative of $S$
- $\textsc{Union}(x, y)$: replaces $S_x$,$S_y$ by $S_x \cup S_y$, where
  - $S_x$ is the set containing $x$
  - $S_y$ is the set containing $y$

  One of the elements in $S_x \cup S_y$ becomes the representative of $S_x \cup S_y$ (negotiable which one)
- $\textsc{Find}(x)$: returns the representative of the set containing $x$

## Applications

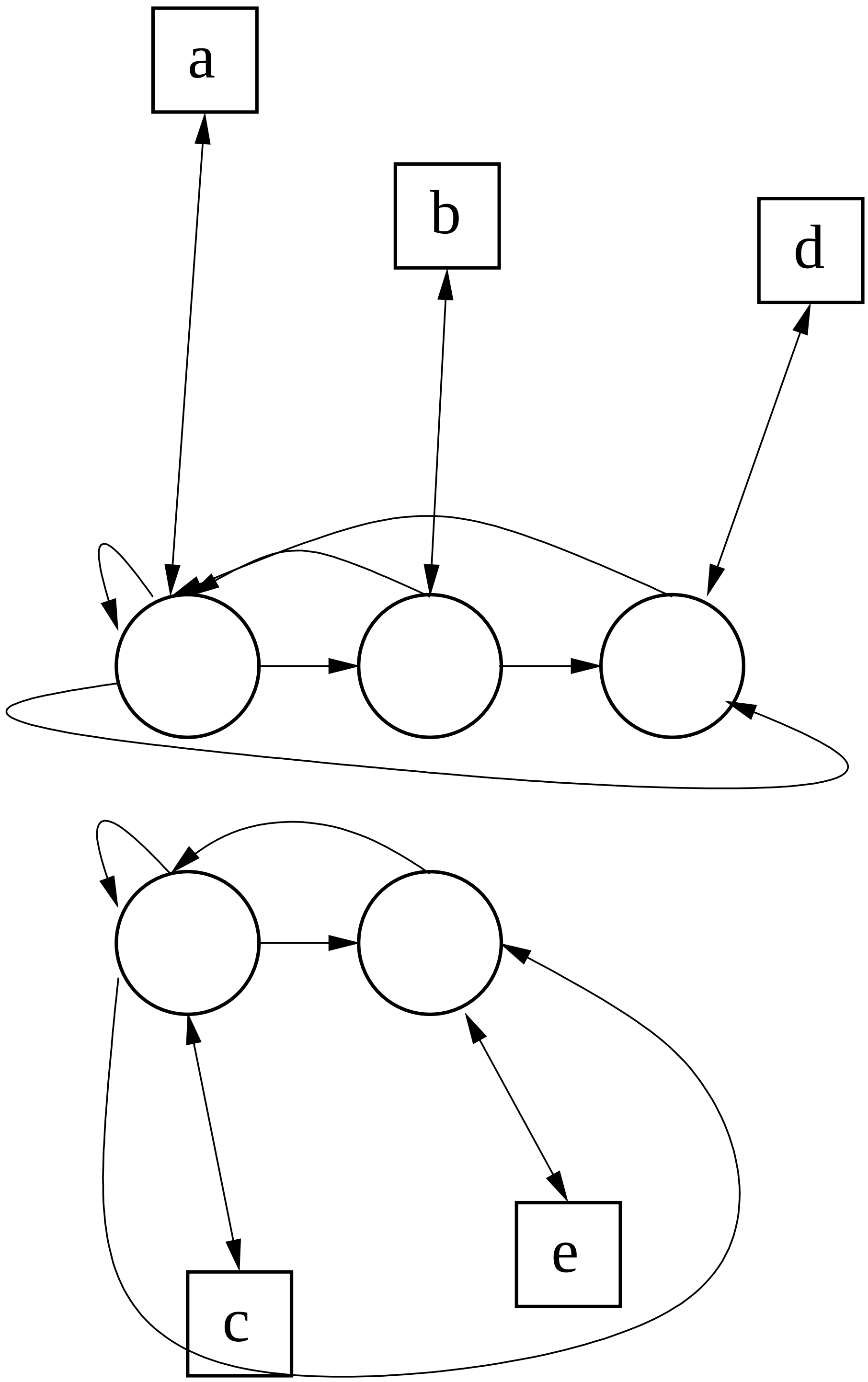
- Killer app: Minimum Spanning Tree (next lecture)
- In general, useful for maintaining *equivalence relations* (e.g. connected components of a graph)

# Implementation I: Linked-list

- We maintain a bunch of linked lists
- Each list corresponds to one set:
  - List contains nodes pointing to the objects from the set
  - Set elements point back to list nodes
  - All nodes in a list point to the head of the list, which is the representative of the set
  - Head also maintains a pointer to the tail of the list

# Linked-list: Example

Linked-list for sets $\{a, b, d\}$, $\{c, e\}$.



## Implementing the operations

- MAKE-SET($x$): create a new list containing pointer to $x$
- FIND($x$): lookup the head of the list which contains pointer to $x$
- UNION($x, y$):
  - Let $L_x$ (resp. $L_y$) denote the lists pointing to $x$ (resp. $y$)
  - Attach $L_y$ to the tail of $L_x$
  - Make all elements of $L_y$ point to the head of $L_x$
  - Update tails etc

## Analysis of Linked-Lists Implementation

- MAKE-SET($x$): $O(1)$ time
- FIND($x$): $O(1)$ time
- UNION($x, y$): $O$(length of $L_y$)

How much can UNION cost ?

- Worst-case: $\Theta(n)$,
  E.g., for sets $\{1 \ldots n/2\}, \{n/2 + 1 \ldots n\}$
- Amortized: total cost of $m$ UNION operations ?
- Unfortunately, it could be $\Theta(m^2)$:
  - Initial sets $\{1\}, \{2\}, \ldots, \{m-2\}, \{m-1\}, \{m\}$
  - Do UNION($m-1, m$), UNION($m-2, m-1$) $\ldots$
  - UNION($m-i, m-i+1$) modifies a list of length $i$
  - Total cost $\Sigma_{i=1}^{m-1} i = \Theta(m^2)$

## How to Improve UNION ?

- Always append the *shorter* list at the end of the *longer* one
- Worst-case cost unchanged
- What is the amortized cost ?
  - Need to bound the cost of $m$ UNION operations and $n$ MAKE-SET operations
  - For each list node $x$, we would like to bound the number of times its pointers to the representative have been updated
  - But each time $x$ is updated, the length of the list containing $x$ increases by a factor $\geq 2$
  - The update to $x$ can happen at most $\log n$ times
  - Apart from that, each UNION takes $O(1)$ time
- Total cost is $O(n \log n + m)$

## Can we do better ?

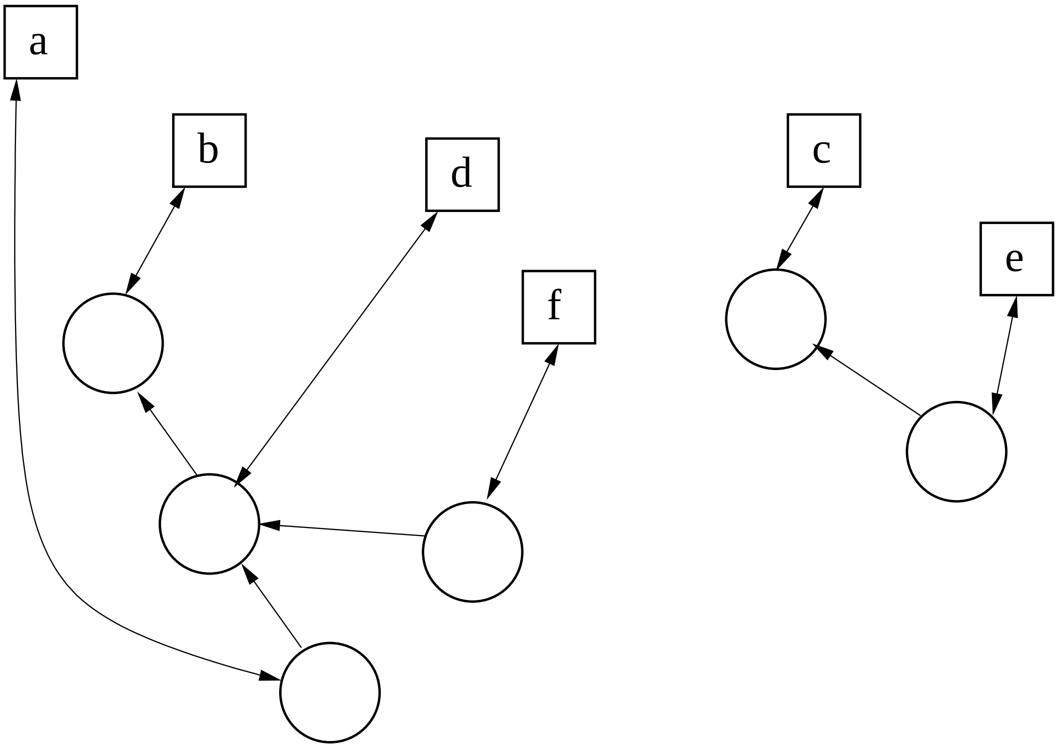
Will slow down FIND, speed up UNION

## Implementation II: Forest

- We maintain a bunch of *trees*
- Each tree corresponds to one set:
  - Tree contains nodes pointing to the objects from the set
  - Set elements point back to tree nodes
  - Each node points to its parent
  - The root of a tree is a representative for the set

## Forest: Example

Forest for sets $\{a, b, d, f\}$, $\{c, e\}$.



## Implementing the operations

- MAKE-SET($x$): create a new tree containing $x$
- FIND($x$): follow the tree up to the root, return the root
- UNION($x, y$):
  - Let $T_x$ (resp. $T_y$) denote the trees containing $x$ (resp. $y$)
  - Find the roots of $T_x$ and $T_y$ (using FIND)
  - Make the root of $T_y$ point to the root of $T_x$

## Analysis of Forest Implementation

- MAKE-SET($x$): $O(1)$ time
- FIND($x$): $O(\text{depth of } T_x)$
- UNION($x, y$): $O(\text{depth of } T_x + \text{depth of } T_y + 1)$ (or constant, if $x$, $y$ are representatives)
- How much can FIND cost ?
  - Worst-case $\Theta(n)$,
    Same example as before creates a tree with $n$ nodes and depth $n - 1$
  - Amortization does not help (can repeat FIND many times)
- What can we do ?

## Improvements

- Idea I: In UNION, attach smaller tree (with fewer nodes) to the larger tree (with more nodes)
  - Results in depth=$O(\log n)$ !
  - Proof as before: each time the depth of $x$ increases (by at most 1), the size of the tree grows by $\geq 2$
- Idea II: path compression (i.e., be nice to the future FIND operations)
  - After finding the root of $T_x$, make all traversed nodes (on the path from $x$ to root) point to the root
  - Extreme example: a very deep tree can become very shallow
  - Can prove that $n$ MAKE-SET ops and $m$ FIND ops cost

$$\Theta(n + m(1 + \log_{2+m/n} n))$$

## Big Bang

What happens if we use *both* ideas together ?

- Define $A_k(j)$ to be equal to
  - $j + 1$, if $k = 0$
  - $A_{k-1}(A_{k-1}(\ldots(A_{k-1}(j))))$ (repeated $j+1$ times), if $k \geq 1$
- $A_k(1)$ grows fast !
  E.g., $A_4(1) >> 2^{2048}$
- Define $\alpha(n) = \min\{k : A_k(1) \geq n\}$
- $\alpha(n) \leq 4$ for all conceivable values of $n$

**Theorem:** Any sequence of $n$ operations on the above data structure costs $\Theta(n\alpha(n))$.