Chap 21 – Data Structures for Disjoint Sets

21.1 Disjoint-set operations

21.2 Linked-list representation of disjoint sets

21.3 Disjoint-set forests

*21.4 Analysis of union by rank with path compression

21.1 Disjoint-set operations

- Disjoint-set data structures
  - Also known as "union find"
  - Maintain a collection
    $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$
    of disjoint dynamic (changing over time) sets.
  - Each set is identified by a *representative*, which is some member of the set.

21.1 Disjoint-set operations

- Disjoint-set operations
  - MAKE-SET(x)
    Make a new set $S_x = \{x\}$, and add S_x to $\mathcal{S}$.
  - UNION(x, y)
    If $x \in S_x$, $y \in S_y$ then $\mathcal{S} = \mathcal{S} - S_x - S_y \cup (S_x \cup S_y)$
    - Representative of new set is any member of $S_x \cup S_y$, often the representative of one of S_x and S_y.
    - Destroy S_x and S_y (since sets must be disjoint).
  - FIND-SET(x)
    Return representative of set containing x.

21.1 Disjoint-set operations

- Disjoint-set application
  - Compute connected components

    CONNECTED-COMPONENTS(G)

    for each vertex $v \in G.V$

    MAKE-SET(v)

    for each edge $(u, v) \in G.E$

    if FIND-SET$(u) \neq$ FIND-SET(v) **then** UNION(u, v)
  - Check if two vertices are in the same component, once CONNECTED-COMPONENTS has preprocessed the graph

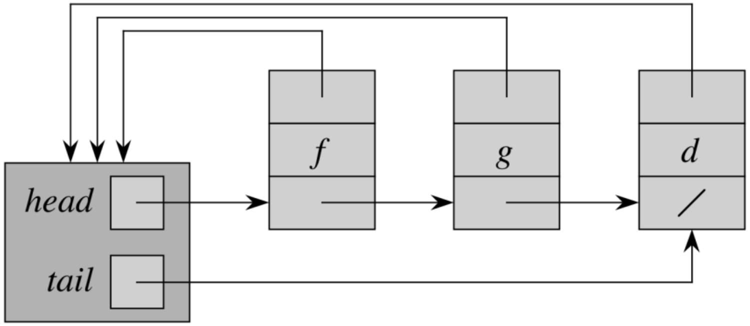
    SAME-COMPONENT(u, v)

    if FIND-SET$(u) ==$ FIND-SET(v) **then return** true

    else return false

21.2 Linked-list representation of disjoint sets

- Linked-list representation
  - Each set is a singly linked list represented by a set object.
  - Each set object has a head (pointer to the representative) and a tail.
  - Each node contains
    - a set member
    - a pointer to the set object
    - a list pointer



set object

21.2 Linked-list representation of disjoint sets

- Implementations of disjoint-set operations
  - MAKE-SET(x)
    - Create a singleton list
  - FIND-SET(x)
    - Follow the pointer back to the set object
    - Then, follow the head pointer to the representative
  - UNION(x, y)

    Simple implementation
    - Always append y's list onto the end of x's list (use x's tail pointer to find the end)

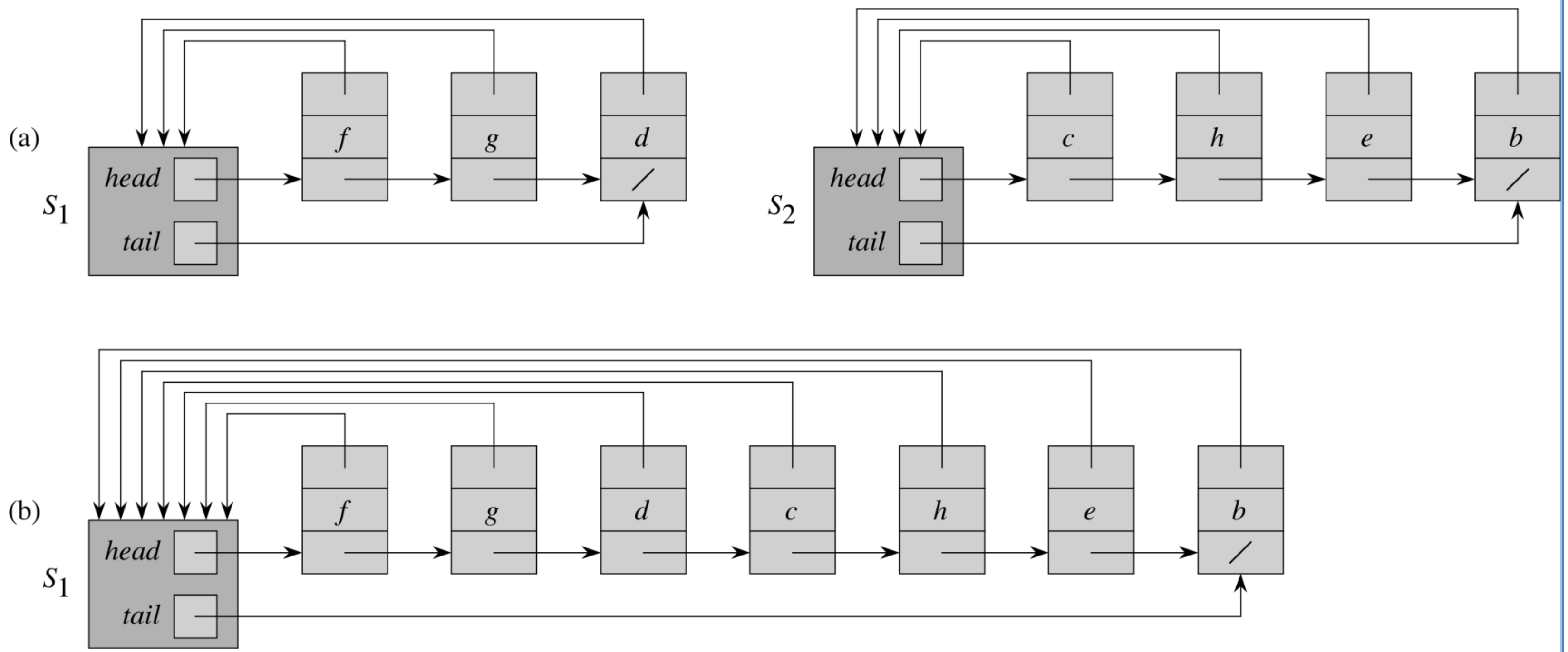
21.2 Linked-list representation of disjoint sets

- Implementations of disjoint-set operations
  - UNION(x, y)
    - The representative of x's list becomes the representative of the resulting set.
    - Need to update the pointer back to the set object for every node on y's list.

    Weighted-union heuristic
    - Always append the smaller list to the larger list.
    - Break ties arbitrarily.
    - Faster than simple implementation, if y's list is longer than x's list.

21.2 Linked-list representation of disjoint sets

- Implementations of disjoint-set operations
  - Example on simple implementation



UNION(g, e)

21.2 Linked-list representation of disjoint sets

- Implementations of disjoint-set operations

  Simple implementation

  - Consider a sequence of m $(\geq n)$ operations on n elements
  - Worst case: $\Theta(n^2)$

  $\textsc{Make-Set}(x_1)$ … $\textsc{Make-Set}(x_n)$ $\cdots \Theta(n), n$ elements

  $\textsc{Union}(x_2, x_1)$ $\textsc{Union}(x_3, x_2)$ … $\textsc{Union}(x_n, x_{n-1})$

  $$\cdots \sum_{k=1}^{n-1} \text{\# of objects updated} = \sum_{k=1}^{n-1} k = \Theta(n^2)$$

  Total time $= \Theta(n) + \Theta(n^2) = \Theta(n^2)$

  Amortized cost per operation

  $= \Theta(n^2)/m = \Theta(n) \quad \because m = 2n - 1$

21.2 Linked-list representation of disjoint sets

- Implementations of disjoint-set operations

  Weighted-union heuristic

  - **Theorem** 21.1

    With weighted union, a sequence of m ($\geq n$) operations on n elements takes $O(m + n \lg n)$ time $\Rightarrow$ amortized cost per operation $= O(1 + \frac{n}{m} \lg n) = O(1 + \lg n)$

    Proof

    Each Make-Set and Find-Set still takes $O(1)$ time, and there are $O(m)$ of them.

    Claim If an object is updated k times, the resulting set has $\geq 2^k$ objects.

21.2 Linked-list representation of disjoint sets

- Implementations of disjoint-set operations
  - **Theorem** 21.1 (Cont'd)

    Basis: $k = 1$

    the object's smaller set has ≥ 1 object

    $\Rightarrow$ the resulting set has $\geq 2^1$ objects

    Induction step

    the object has already been updated k times

    $\Rightarrow$ the object's smaller set has $\geq 2^k$ objects by IH

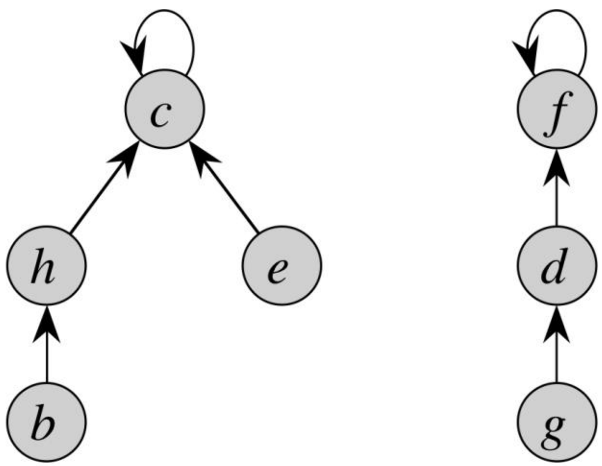
    $\Rightarrow$ the resulting set has $\geq 2^{k+1}$ objects $\blacksquare$

    Now, $n \geq$ the size of the resulting set $\geq 2^k \Rightarrow \lg n \geq k$

    Thus, total number of updates for n objects $\leq n \lg n$.

21.3 Disjoint-set forests

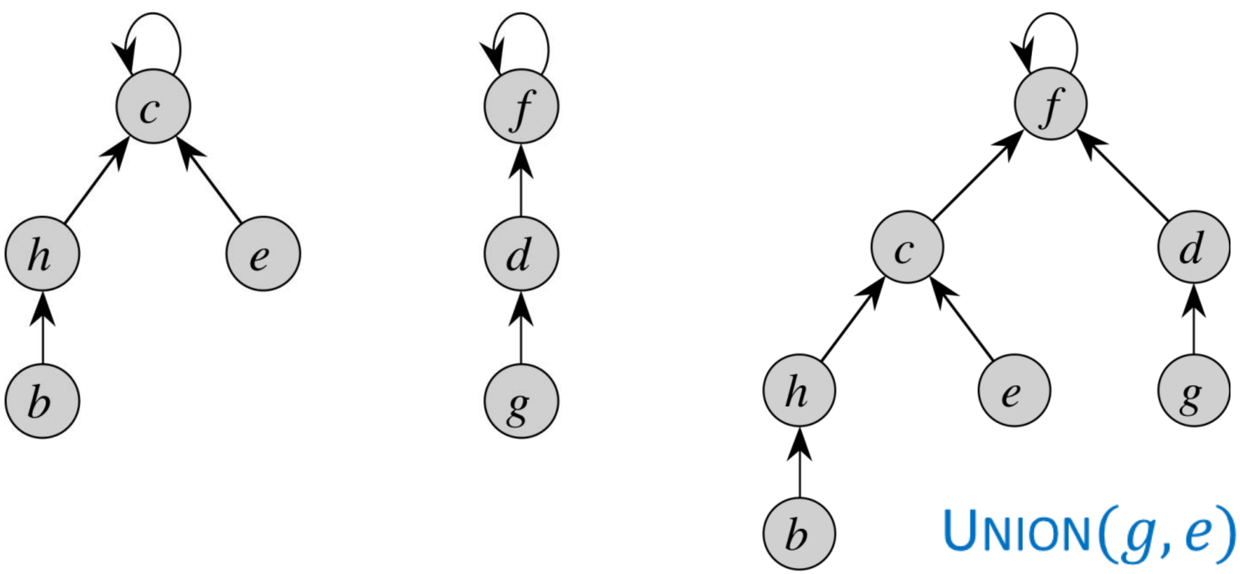
- Forest-of-trees representation
  - Each set is a rooted tree.
  - The root is the representative.
  - Each node points to its parent (the root is its own parent).



21.3 Disjoint-set forests

- Implementation of disjoint-set operations
  - MAKE-SET(x): Create a single-node tree
  - FIND-SET(x): Follow pointers to the root
  - UNION(x, y): Make one root a child of the other root.

    Not so good – Could get a linear chain of nodes



UNION(g, e)

21.3 Disjoint-set forests

- Implementation of disjoint-set operations

  1st heuristic: Union by rank

  - Idea: Make the root of the smaller tree into a child of the root of the larger tree
  - Don't actually use the ***size*** of a tree. Use ***rank***.
  - For each node, maintain a ***rank*** that is an upper bound on the height of the node.
  - Make the root with the smaller rank into a child of the root with the larger rank
  - Alone, union by rank yields a running time of $\Theta(m \lg n)$. (See Ex. 21.4-4 and 21.3-3)

21.3 Disjoint-set forests

- Implementation of disjoint-set operations

1st heuristic: Union by rank

- $\text{MAKE-SET}(x)$

  $x.p = x$

  $x.rank = 0$

- $\text{LINK}(x, y)$

  if $x.rank > y.rank$ **then** $y.p = x$

  else $x.p = y$

  // If equal, choose y as parent and increment its rank

  if $x.rank == y.rank$ **then**

  $y.rank = y.rank + 1$

21.3 Disjoint-set forests

- Implementation of disjoint-set operations

  1st heuristic: Union by rank

  - UNION(x, y)

    LINK(FIND-SET(x), FIND-SET(y))

  - FIND-SET(x)

    if $x.p \neq x$ **then return** FIND-SET$(x.p)$

    return $x.p$ // or, x

  - Comment

    So far, with the union-by-rank heuristic alone,

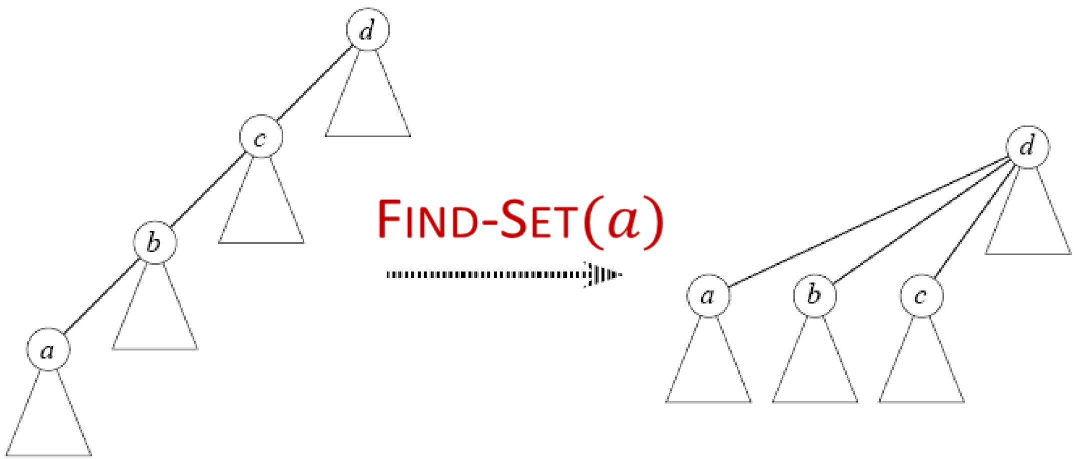
    the ***rank*** of a node = the height of the node.

21.3 Disjoint-set forests

- Implementation of disjoint-set operations

  2nd heuristic: Path compression

  - Make all nodes visited during FIND-SET on the trip to the root direct children of the root



  - Alone, path compression gives a worst-case running time of $\Theta(n + f(1 + \log_{2+f/n} n))$, if there are n MAKE-SETs and f FIND-SETs.

21.3 Disjoint-set forests

- Implementation of disjoint-set operations

  Both heuristics: Union by rank + Path compression

  - Modify FIND-SET as follows.

    The other operations remain unchanged.
  - FIND-SET(x)

    if $x.p \neq x$ **then** $x.p = $ FIND-SET$(x.p)$

    return $x.p$
  - Path compression doesn't change any ranks

    $\Rightarrow$ the ***rank*** of a node $\geq$ the height of the node

21.3 Disjoint-set forests

- Implementation of disjoint-set operations

  Both heuristics: Union by rank + Path compression

  - With both heuristics, the worst-case running time is $O(m\alpha(n))$, where $\alpha(n)$ grows very slowly. (Sec. 21.4)

    $$\alpha(n) = \begin{cases} 0, & n = 0,1,2 \\ 1, & n = 3 \\ 2, & n = 4,5,6,7 \\ 3, & 8 \leq n \leq 2047 \\ 4, & 2048 \leq n \leq A_4(1) \end{cases}$$

    where $A_4(1) \gg 10^{80} =$ the estimated # of atoms in the observable universe
  - Thus, $\alpha(n) \leq 4$ for all practice purposes.

21.4 Analysis of union by rank with path compression

- A very quickly growing function
  - For $k \geq 0, j \geq 1$, define

    $$A_k(j) = \begin{cases} j+1, & k=0 \\ A_{k-1}^{(j+1)}(j), & k \geq 1 \end{cases}$$

    where

    $$A_{k-1}^{(j+1)}(j) = \underbrace{(A_{k-1} \circ \cdots \circ A_{k-1})}_{j+1 \text{ times}}(j)$$
  - **Lemma** $A_1(j) = 2j+1$
  - **Lemma** $A_2(j) = 2^{j+1}(j+1) - 1$
  - Example

    $A_0(1) = 2, A_1(1) = 3, A_2(1) = 2^2 \cdot 2 - 1 = 7$

21.4 Analysis of union by rank with path compression

- A very quickly growing function
  - Example

$$A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7)$$
$$= 2^8 \cdot 8 - 1 = 2^{11} - 1$$
$$= 2047$$
$$A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047)$$
$$= A_2^{(2048)}(2047)$$
$$\gg A_2(2047)$$
$$= 2^{2048} \cdot 2048 - 1 = 2^{2059} - 1$$
$$> (2^{10})^{205}$$
$$> (10^3)^{205} = 10^{615} \gg 10^{80}$$

21.4 Analysis of union by rank with path compression

- A very slowly growing function
  - Define the inverse of $A_k(n)$ by

    $\alpha(n) = \min\{k : A_k(1) \geq n\}$
  - From the above values of $A_k(1)$, we see that

$$
\begin{aligned}
&A_0(1) = 2 \geq n && \Rightarrow \alpha(n) = 0, && n = 0,1,2 \\
&A_1(1) = 3 \geq n && \Rightarrow \alpha(n) = 1, && n = 3 \\
&A_2(1) = 7 \geq n && \Rightarrow \alpha(n) = 2, && n = 4,5,6,7 \\
&A_3(1) = 2047 \geq n && \Rightarrow \alpha(n) = 3, && 8 \leq n \leq 2047 \\
&A_4(1) = x \geq n && \Rightarrow \alpha(n) = 4, && 2048 \leq n \leq x = A_4(1)
\end{aligned}
$$