CS 255, Spring 2014, SJSU

Data Structures for Disjoint Sets

Fernando Lobo

Data Structures for Disjoint Sets

- Also known as UNION-FIND.
- Goal: Maintain a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint sets, that change through time.
- As change we only allow set union (removing elements or breaking a set into two sets is not allowed.)

Data Structures for Disjoint Sets

- Each set is identified by a representative, which is a member of the set.
- The choice of the representative is irrelevant. But if we ask for the representative of a given set we should always get the same answer, assuming the set didn't change between queries.
- Such a data structure has several applications, as we shall see later.

Operations

- $\textsc{Make-Set}(x)$: creats a set $S_i = \{x\}$ e adds it to S.
- $\textsc{Find-Set}(x)$: returns the identifier (a pointer to the representative) of the set that contains x.
- $\textsc{Union}(x, y)$: if $x \in S_i$ and $y \in S_j$, then $S = S - S_i - S_j \cup \{S_i \cup S_j\}$

Analysis

We shall make an analysis in terms of:

- n = total number of elements = number of MAKE-SETS.
- m = total number of operations.
  - $m \geq n$ because MAKE-SETS are included in the total number of operations.
  - There can only be a maximum of $n-1$ UNIONS (after that we are left with a single set.)

Application example: Connected components of a graph

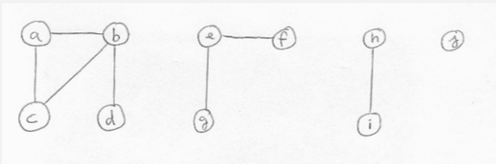
```
CONNECTED-COMPONENTS(G)
    for each v ∈ G.V
        MAKE-SET(v)
    for each (u, v) ∈ G.E
        if FIND-SET(u) ≠ FIND-SET(v)
            UNION(u, v)
```


Application example: Connected components of a graph

Once we find the connected components, the function SAME-COMPONENT allows us to check if two nodes are in the same component.

```
SAME-COMPONENT(u, v)
    if FIND-SET(u) == FIND-SET(v)
        return TRUE
    else
        return FALSE
```


Example

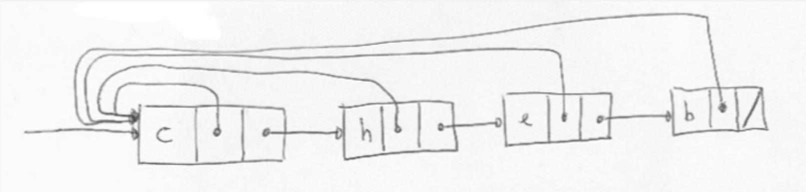


	Collection of disjoint sets
	{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}
(b,d)	{a} {b,d} {c} {e} {f} {g} {h} {i} {j}
(e,g)	{a} {b,d} {c} {e,g} {f} {h} {i} {j}
(a,c)	{a,c} {b,d} {e,g} {f} {h} {i} {j}
(h,i)	{a,c} {b,d} {e,g} {f} {h,i} {j}
(a,b)	{a,b,c,d} {e,g} {f} {h,i} {j}
(e,f)	{a,b,c,d} {e,f,g} {h,i} {j}
(b,c)	{a,b,c,d} {e,f,g} {h,i} {j}

How to implement these operations efficiently?

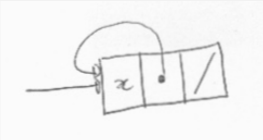
A first approach: use a linked list for each set.

- Each element of the list has:
  - an element of the set.
  - a pointer to the representative of the set, which we can choose to be the first element in the list.
  - a pointer to the next element of the list
- The list has pointers to head and tail.



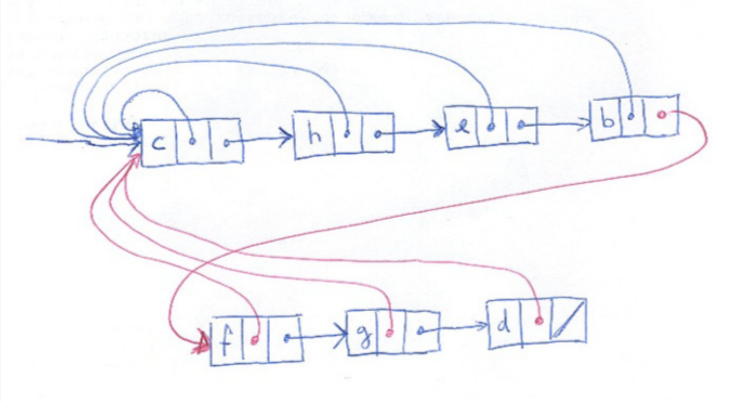
1st approach (cont.)

- MAKE-SET(x) $\rightarrow$ create a new list with a single element $\rightarrow$ O(1).



- FIND-SET(x) $\rightarrow$ return the element pointed by the representative (i.e, the first element of the list) $\rightarrow$ O(1).

- UNION(X, Y) $\rightarrow$ need to concatenate the list that contains x with the list that contains y. The pointer to the tail allows us to do it in O(1) time. But we need O(n) time to update the pointers to the set representative. So the running time of this operation is O(n).



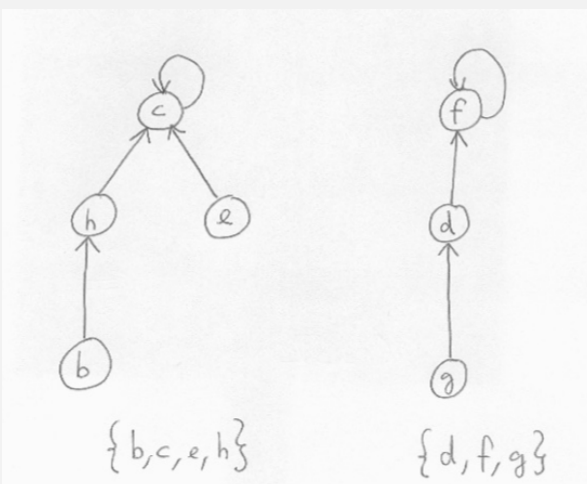
Heuristic: concatenate the smaller list at the end of the larger list

- Concatenating the larger list at the end of the smaller list should be avoided.
- Can improve performance by always concatenating the smaller list at the end of the larger list.
  - Easy, just need to keep an attribute for each list that maintains the list size.
  - Still, union takes $\Omega(n)$ if both sets have $\Omega(n)$ elements.
  - It can be shown that a sequence of m operations over n elements takes $O(m + n \lg n)$ time. (Proof in textbook)
  - $O(m)$ for MAKE-SET and FIND-SET operations.
    $O(n \lg n)$ for the UNION operations.

2nd approach

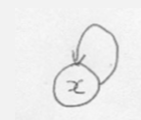
We can do better.

- Instead of using a linked list, use an inverted tree to represent a set.
- Each tree node points to its parent. The root points to itself.
- The element at the root is the set representative.



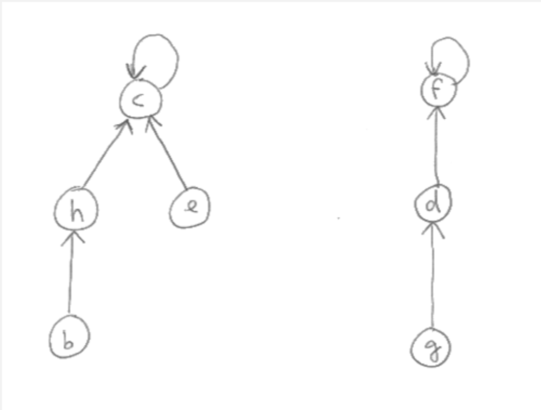
2nd approach

- MAKE-SET(x) $\rightarrow$ Creates a tree with a single node.

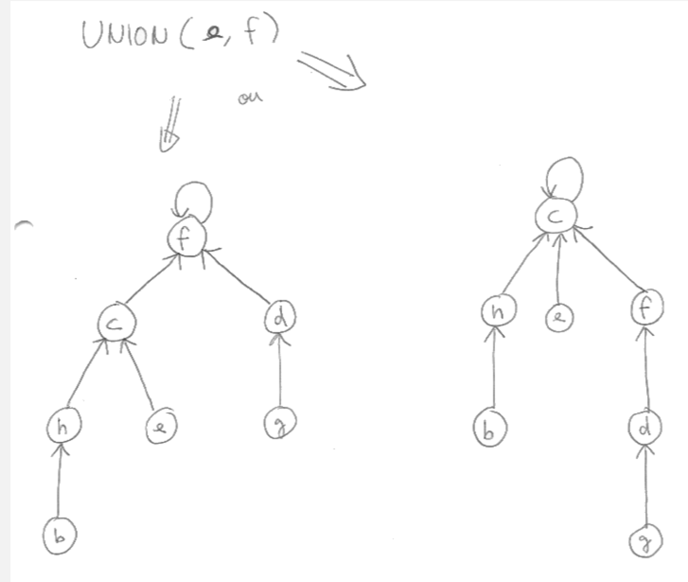


- FIND-SET(x) $\rightarrow$ Follow the parent points until reaching the root, then return the element at the root.
- UNION(x, y) $\rightarrow$ Let the root of one tree become child of the root the other tree.

Example: UNION(e, f)



Example (cont.)



Improving performance

- With this data structure it is still possible to get a degenerated tree that ends up being like a linked-list.
- Can avoid that by using two heuristics:
  1. Union by rank $\rightarrow$ make the shorter tree the child of the taller tree.
  2. Path compression $\rightarrow$ while executing FIND-SET(x) rearrange the tree in such a way that all the nodes on the path from x to the root, have their parent become the root.

Union by rank

- We use the *rank* of the root $\rightarrow$ an upper bound for the tree height. (For now think of rank and being height.)
- When we do union, we compare the rank of the roots. The one with smaller rank become child of the one with larger rank. Break ties arbitrarily.
- The idea is to avoid growth of tree height.

Implementation

- The implementation is very simple.
- Each node only needs to know its parent and its rank.
  $\Longrightarrow$ a single array is enough to represent the forest.

Pseudocode for union by rank

```
MAKE-SET(x)
    parent[x] = x
    rank[x] = 0

FIND-SET(x)
    while x ≠ parent[x]
        x = parent[x]
    return x
```


Pseudocode for union by rank

```
UNION(x, y)
    rx = FIND-SET(x)
    ry = FIND-SET(y)
    if rx == ry
        return
    if rank[rx] > rank[ry]
        parent[ry] = rx
    else
        parent[rx] = ry
        if rank[rx] == rank[ry]
            rank[ry] = rank[ry] + 1
```

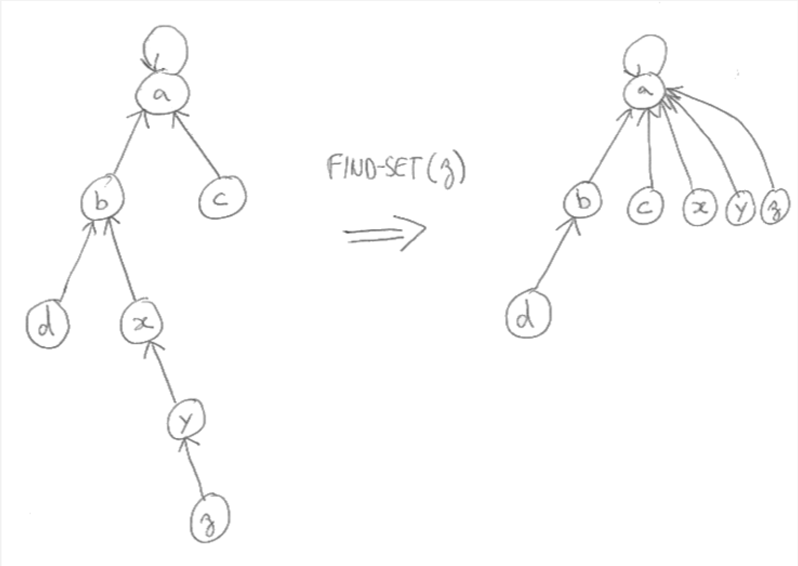

Observations on union by rank

- The *rank* of a node is the height of the subtree rooted at that node.
- **Property 1:** For any node x except the root, $rank[x] < rank[parent[x]]$ (because a root node of rank k is created by merging two trees with roots of rank $k-1$)
- **Property 2:** Any root node of rank k has at least 2^k nodes in its tree. (can be easily shown by induction.)
- **Property 3:** If there are n elements, there can be at most $n/2^k$ nodes of rank k. ($\Longrightarrow$ maximum rank is $\lg n$ $\Longrightarrow$ all trees have height $\leq \lg n$.)

Path compression

- While executing FIND-SET(x) we need to traverse the path from node x to the root of the tree.
- We might as well take the opportunity and make all those nodes become children of the root.
- We can do it spending only a constant time per element along the path from x to the root.
- All subsequent find operations on elements that were on the path from x to the root, will be done faster.
- Trees become less deep and bushier.

Example of path compression



Pseudocode for path compression

FIND-SET(x)
    if $x \neq parent[x]$
        $parent[x] =$ FIND-SET($parent[x]$)
    return $parent[x]$

- MAKE-SET(x) and UNION(x, y) stay the same.

Observations on path compression

- Node ranks are untouched by path compression.
- But the rank can no longer be interpreted as tree height.
- The rank becomes an upper bound on tree height.

Analysis

- It can be shown that the running time of a sequence of m operations over n elements takes $O(m \lg^* n)$ time. $\Longrightarrow$ The amortized cost per operation is $O(\lg^* n)$
  - m is the total number of operations (Make-Sets, Find-Sets and Unions).
  - n is the number of Make-Sets.
- $\lg^* n$ is the iterated log function, the number of consecutive iterations of the logarithm function that are necessary to reach a number less or equal to 1.

$\lg^*$ grows very slowly

$$
\begin{aligned}
\lg^* 2 &= 1 \\
\lg^* 4 &= \lg^* 2^2 &= 2 \\
\lg^* 16 &= \lg^* 2^{2^2} &= 3 \\
\lg^* 65536 &= \lg^* 2^{2^{2^2}} &= 4 \\
\lg^* 2^{65536} &= \lg^* 2^{2^{2^{2^2}}} &= 5
\end{aligned}
$$

- $2^{65536} = 2^{2^{2^{2^2}}}$ is HUGE, much larger than the total number of atoms in the universe! $\Longrightarrow$ For all practical purposes, $\lg^* n \leq 5$.
- A sequence of m operations takes barely over linear time (for all practical purposes it's linear.)