CSE 326: Data Structures

Disjoint Sets

Neva Cherniavsky
Summer 2006

Equivalence Relations

Relation *R* :

- For every pair of elements *(a, b)* in a set S, a *R* b is either true or false.
- If a *R* b is true, then a is related to b.

An equivalence relation satisfies:

1. (Reflexive) a *R* a
2. (Symmetric) a *R* b iff b *R* a
3. (Transitive) a *R* b and b *R* c implies a *R* c

Examples of Equivalence Relations

- $\geq$: Is it reflexive, symmetric, and transitive?
- Electrical connectivity: Is it reflexive, symmetric, and transitive?
- Two cities in the same country: Is it reflexive, symmetric, and transitive?

Determining Equivalence Classes

- Divide set S into subsets containing items related to each other
  - {Paris, Lyon} , {Seattle, New York, Boston}, {London}, {Bombay, Calcutta}
- Given the set, how do we determine these classes?
  - {Paris} , {Lyon} , {Seattle} , {New York} , {Boston}, {London}, {Bombay} , {Calcutta}

Solution: Union/Find

Algorithm:

- Start with sets $S_0, S_1, S_2, \ldots, S_k$
- Check: is S_0 related to S_1? (Does find return the same value?)
- If so, perform union

Applications:

- Graph theory problems (project phase C)
- Compiler checking type relations

Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - {Paris} , {Lyon} , {Seattle, New York} , {Boston}, {London}, {Bombay, Calcutta}
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name, one of its members
  - {Paris} , {Lyon} , {Seattle, New York} , {Boston}, {London}, {Bombay, Calcutta}
  - {3,5,7} , {4,2,8}, {9}, {1,6}

Union

- Union(x,y) – take the union of two sets named x and y
  - {3,5,7} , {4,2,8}, {9}, {1,6}
  - Union(5,1)

    {3,5,7,1,6}, {4,2,8}, {9},

Find

- Find(x) – return the name of the set containing x.
  - {3,5,7,1,6}, {4,2,8}, {9},
  - Find(1) = 5
  - Find(4) = 8

Example

S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Find(8) = 7
Find(14) = 20

→ Union(7,20)

S
{1,2,7,8,9,13,19,14,20 26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Implementing the DS ADT

- *n* elements,
  Total Cost of: *m* finds, ≤ *n*-1 unions

can there be more unions?

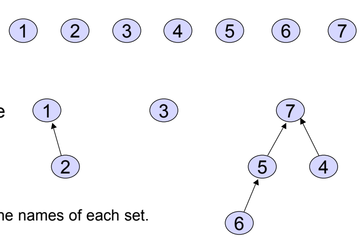
- Target complexity: O(*m*+*n*)
  i.e. O(1) amortized
- O(1) worst-case for find as well as union would be great, but…

  Known result: both find and union *cannot* be done in worst-case O(1) time

Up-Tree for DU/F

Initial state

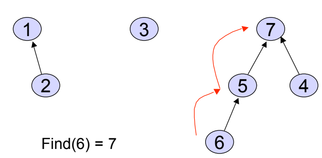
Intermediate state



Roots are the names of each set.

Find Operation

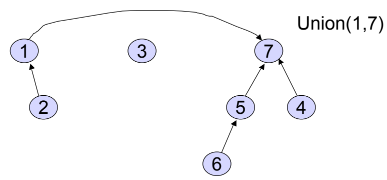
- Find(x) follow x to the root and return the root



Find(6) = 7

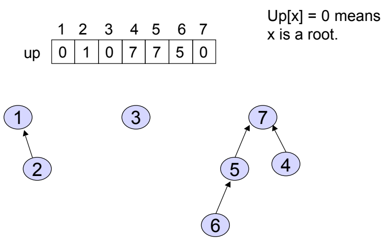
Union Operation

- Union(i,j) - assuming i and j roots, point i to j.



Simple Implementation

- Array of indices



Union

```
Union(up[] : integer array, x,y : integer) : {
//precondition: x and y are roots//
Up[x] := y
}
```

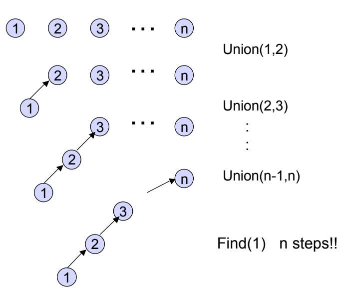
Constant Time!

Exercise

- Design Find operator
  - Recursive version
  - Iterative version

```
Find(up[] : integer array, x : integer) : integer {
//precondition: x is in the range 1 to size//
???
}
```

A Bad Case



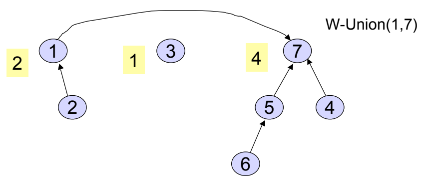
Now this doesn't look good ☹

Can we do better? *Yes!*

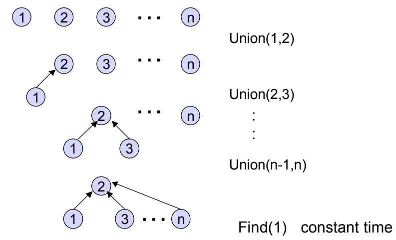
1. Improve union so that *find* only takes O(log *n*)
   - Union-by-size
   - Reduces complexity to O(*m* log *n* + *n*)
2. Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to almost O(*m* + *n*)

Weighted Union

- Weighted Union
  - Always point the smaller tree to the root of the larger tree

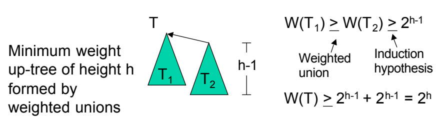


Example Again



Analysis of Weighted Union

- With weighted union an up-tree of height h has weight at least 2^h.
- Proof by induction
  - Basis: h = 0. The up-tree has one node, $2^0 = 1$
  - Inductive step: Assume true for all h' < h.

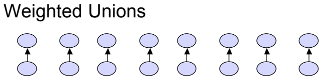


Analysis of Weighted Union

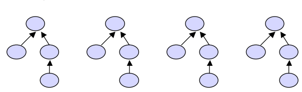
- Let T be an up-tree of weight n formed by weighted union. Let h be its height.
- $n \geq 2^h$
- $\log_2 n \geq h$
- Find(x) in tree T takes O(log n) time.
- Can we do better?

Worst Case for Weighted Union

n/2 Weighted Unions

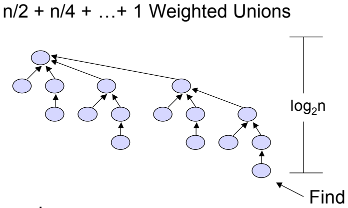


n/4 Weighted Unions



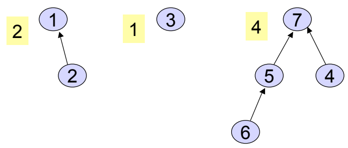
Example of Worst Cast (cont')

After n -1 = n/2 + n/4 + …+ 1 Weighted Unions



If there are $n = 2^k$ nodes then the longest path from leaf to root has length k.

Elegant Array Implementation



	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0
weight	2		1				4

Weighted Union

```
W-Union(i,j : index){
//i and j are roots//
  wi := weight[i];
  wj := weight[j];
  if wi < wj then
    up[i] := j;
    weight[j] := wi + wj;
  else
    up[j] :=i;
    weight[i] := wi +wj;
}
```

Union-by-size: Find Analysis

- Complexity of Find: O(max node depth)
- All nodes start at depth 0
- Node depth increases:
  - Only when it is part of smaller tree in a union
  - Only by one level at a time

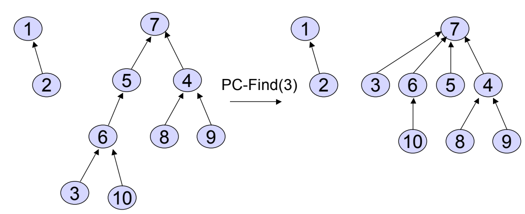
Result: **tree size doubles when node depth increases by 1**

Find runtime = O(node depth) =

runtime for m finds and n-1 unions =

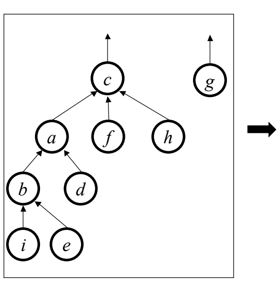
Path Compression

- On a Find operation point all the nodes on the search path directly to the root.

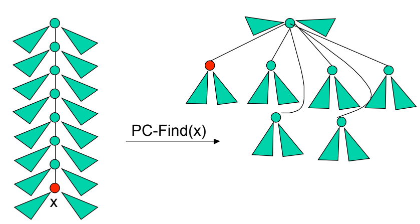


Student Activity

Draw the result of Find(e):



Self-Adjustment Works



Path Compression Find

```
PC-Find(i : index) {
  r := i;
  while up[r] ≠ 0 do //find root//
    r := up[r];
  if i ≠ r then  //compress path//
    k := up[i];
    while k ≠ r do
      up[i] := r;
      i := k;
      k := up[k]
  return(r)
}
```

Complex Complexity of Union-by-Size + Path Compression

Tarjan proved that, with these optimizations, p union and find operations on a set of n elements have worst case complexity of $O(p \cdot \alpha(p, n))$

For *all practical purposes* this is amortized constant time: $O(p \cdot 4)$ for p operations!

- Very complex analysis – worse than splay tree analysis etc. that we skipped!

Disjoint Union / Find with Weighted Union and PC

- Worst case time complexity for a W-Union is O(1) and for a PC-Find is O(log n).
- Time complexity for m ≥ n operations on n elements is O(m log* n) where log* n is a very slow growing function.
  - Log * n < 7 for all reasonable n. Essentially constant time per operation!
- Using "ranked union" gives an even better bound theoretically.

Amortized Complexity

- For disjoint union / find with weighted union and path compression.
  - average time per operation is essentially a constant.
  - worst case time for a PC-Find is O(log n).
- An individual operation can be costly, but over time the average cost per operation is not.

Find Solutions

Recursive

```
Find(up[] : integer array, x : integer) : integer {
//precondition: x is in the range 1 to size//
if up[x] = 0 then return x
else return Find(up,up[x]);
}
```

Iterative

```
Find(up[] : integer array, x : integer) : integer {
//precondition: x is in the range 1 to size//
while up[x] ≠ 0 do
  x := up[x];
return x;
}
```