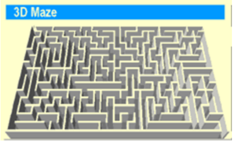
Disjoint Sets

Making a Good Maze

What's a Good Maze?

- Connected
- Just one path between any two rooms
- Random



The Maze Construction Problem

- Given:
    - collection of rooms: V
    - connections between rooms (initially all closed): E
- Construct a maze:
    - collection of rooms: $V' = V$
    - designated rooms in, $i \in V$, and out, $o \in V$
    - collection of connections to knock down: $E' \subseteq E$ such that one unique path connects every two rooms

Maze Construction Algorithm

While edges remain in E

1. Remove a random edge $e = (u, v)$ from E
   How can we do this efficiently?
2. If u and v have not yet been connected
   - add e to E
   - mark u and v as connected

   How to check connectedness efficiently?

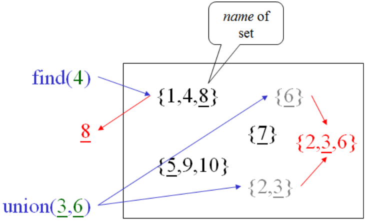
Equivalence Relations

- An equivalence relation $R(\subseteq A \times A)$ must have three properties
    - reflexive: $\forall x \in A, (x,x) \in R$
    - symmetric: $(x,y) \in R \Rightarrow (y,x) \in R$
    - transitive: $(x,y) \in R \wedge (y,z) \in R \Rightarrow (x,z) \in R$
- Connection between rooms is an equivalence relation
    - any room is connected to itself
    - if room a is connected to room b, then room b is connected to room a
    - if room a is connected to room b and room b is connected to room c, then room a is connected to room c

Disjoint Set Union/Find ADT

Union/Find operations

- create
- destroy
- union
- find



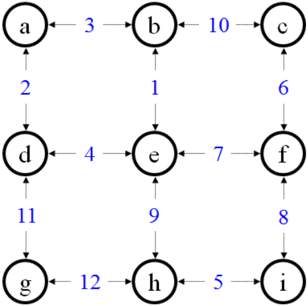
- *Disjoint set partition property:* element of a DS U/F structure belongs to *exactly one set* with a *unique name*
- *Dynamic equivalence property: Union*(a, b) creates a new set which is the union of the sets containing a and b

Example

Construct the maze on the right

Initial (the name of each set is in boldface): {**a**}{**b**}{**c**}{**d**}{**e**}{**f**}{**g**}{**h**}{**i**}

Randomly select edge 1

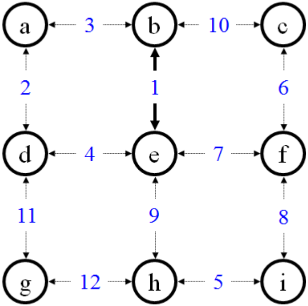


Order of edges in blue

Example, First Step

$\{\mathbf{a}\}\{\mathbf{b}\}\{\mathbf{c}\}\{\mathbf{d}\}\{\mathbf{e}\}\{\mathbf{f}\}\{\mathbf{g}\}\{\mathbf{h}\}\{\mathbf{i}\}$

find(b) $\Rightarrow$ **b**
find(e) $\Rightarrow$ **e**
find(b) $\neq$ find(e) so:
add 1 to E
union(b, e)



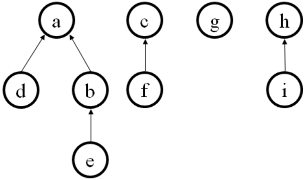
Order of edges in blue

$\{\mathbf{a}\}\{\mathbf{b}, e\}\{\mathbf{c}\}\{\mathbf{d}\}\{\mathbf{f}\}\{\mathbf{g}\}\{\mathbf{h}\}\{\mathbf{i}\}$

Up-Tree Intuition

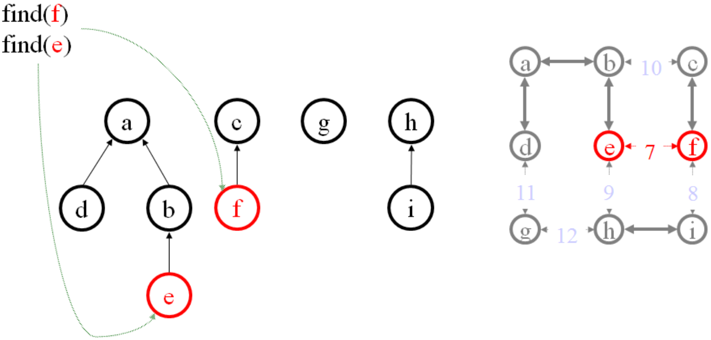
- Finding the representative member of a set is somewhat like the opposite of finding whether a given key exists in a set.
- So, instead of using trees with pointers from each node to its children; let's use trees with a pointer from each node to its parent.

  - Each subset is an up-tree with its root as its representative member
  - All members of a given set are nodes in that set's up-tree
  - Hash table maps input data to the node associated with that data



Up-trees are **not** necessarily binary!

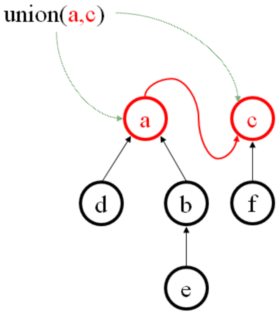
Find

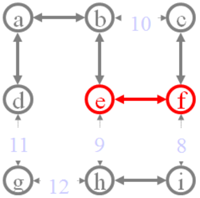


runtime:

Just traverse to the root!

Union

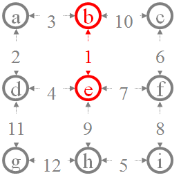




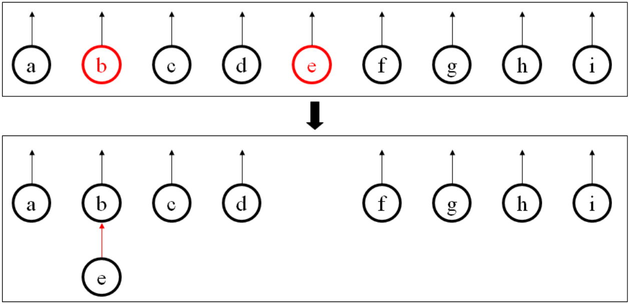
runtime:

Just hang one root from the other!

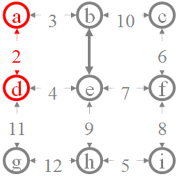
The Whole Example



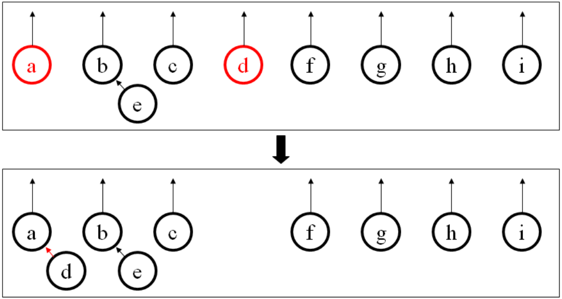
union(b,e)



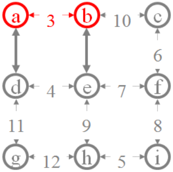
The Whole Example



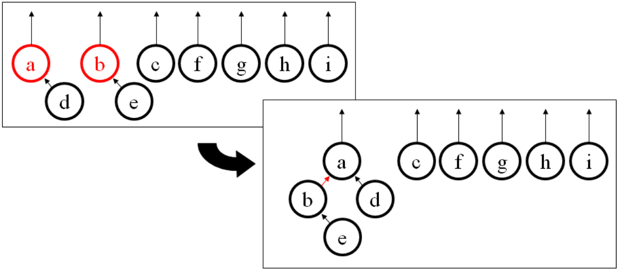
union(a,d)



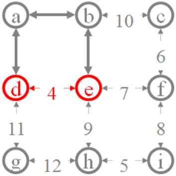
The Whole Example



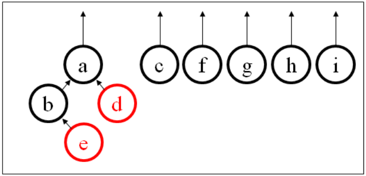
union(a,b)



The Whole Example

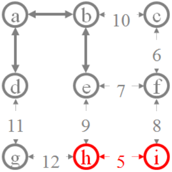


find(d) = find(e)
No union!

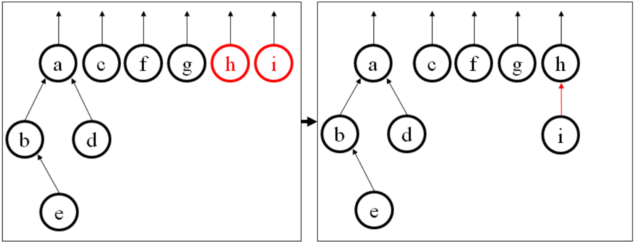


While we're finding *e*,
could we do anything else?

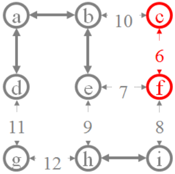
The Whole Example



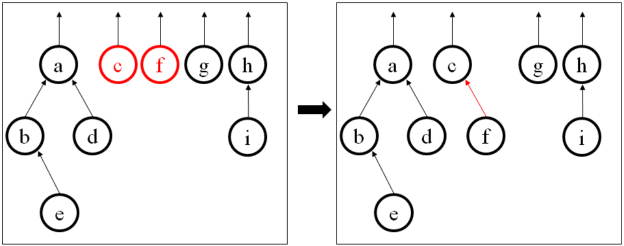
union(h,i)



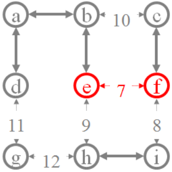
The Whole Example



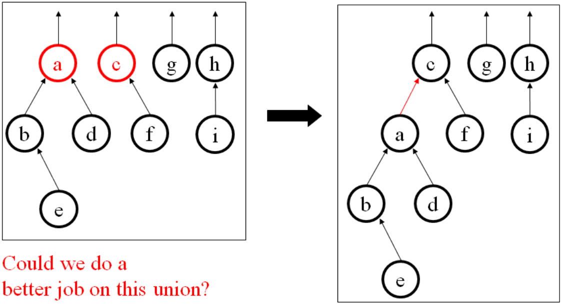
union(c,f)



The Whole Example

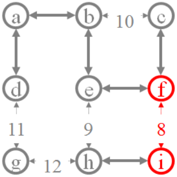


find(e)
find(f)
union(a,c)

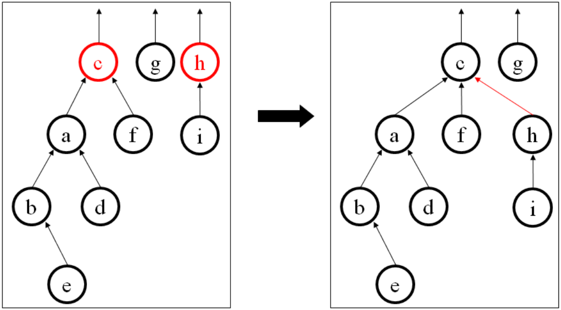


Could we do a
better job on this union?

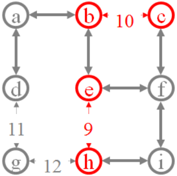
The Whole Example



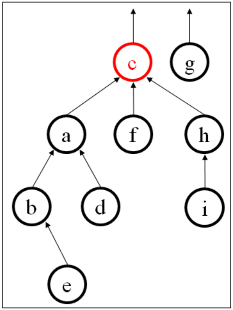
find(f)
find(i)
union(c,h)



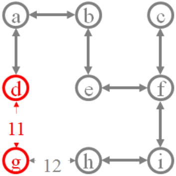
The Whole Example



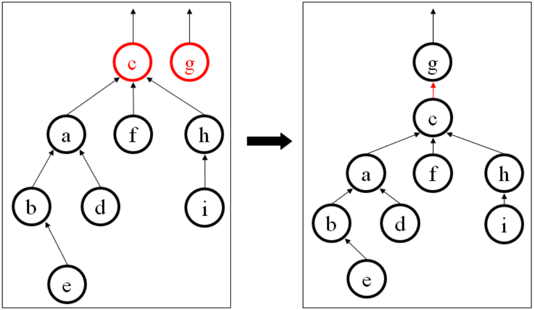
find(e) = find(h) and find(b) = find(c)
So, no unions for either of these.



The Whole Example

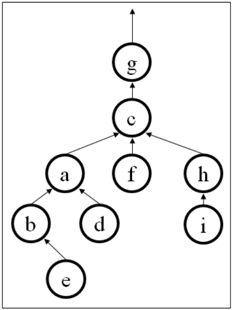


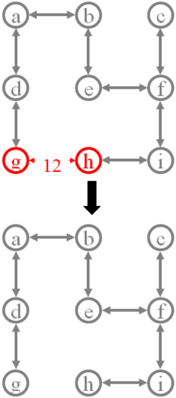
find(d)
find(g)
union(c, g)



The Whole Example

find(g) = find(h)
So, no union.
And, we're done!

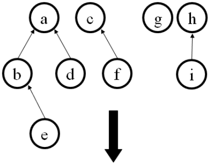




Ooh… scary!
Such a hard maze!

Nifty storage trick

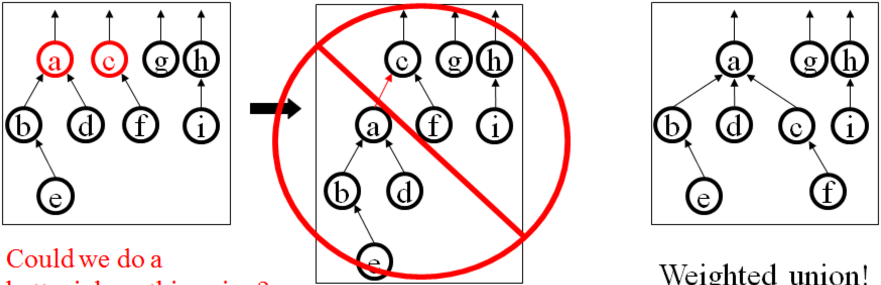
- A forest of up-trees can easily be stored in an array.
- Also, if the node names are integers or characters, we can use a very simple, perfect hash.



	0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)	6 (g)	7 (h)	8 (i)
up-index:	-1	0	-1	0	1	2	-1	-1	7

Room for Improvement: Weighted Union

- Always makes the root of the larger tree the new root
- Often cuts down on height of the new up-tree

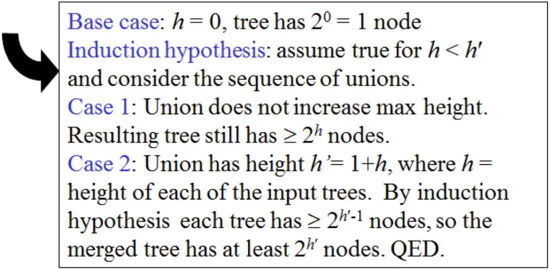


Could we do a better job on this union?

Weighted union!

Weighted Union Find Analysis

- Finds with weighted union are O(max up-tree height)
- But, an up-tree of height h with weighted union must have at least 2^h nodes



Base case: $h = 0$, tree has $2^0 = 1$ node
Induction hypothesis: assume true for $h < h'$ and consider the sequence of unions.
Case 1: Union does not increase max height. Resulting tree still has $\geq 2^h$ nodes.
Case 2: Union has height $h' = 1 + h$, where h = height of each of the input trees. By induction hypothesis each tree has $\geq 2^{h'-1}$ nodes, so the merged tree has at least $2^{h'}$ nodes. QED.

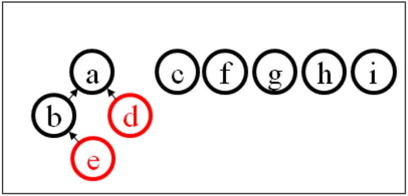
- Hence, $2^{max\ height} \leq n$ and max height $\leq \log n$
- So, find takes $O(\log n)$

Alternatives to Weighted Union

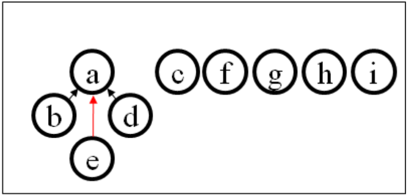
- Union by height
- Ranked union (cheaper approximation to union by height)

Room for Improvement: Path Compression

- Points everything along the path of a find to the root
- Reduces the height of the entire access path to 1



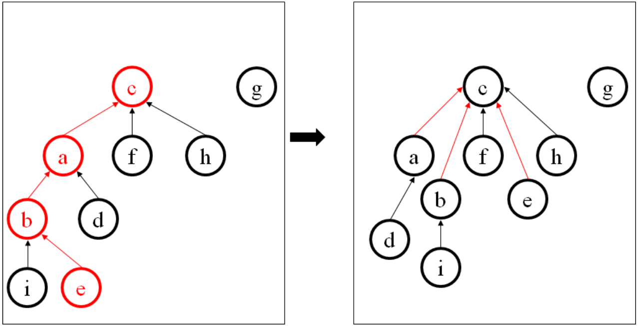
While we're finding ***e***,
could we do anything else?



Path compression!

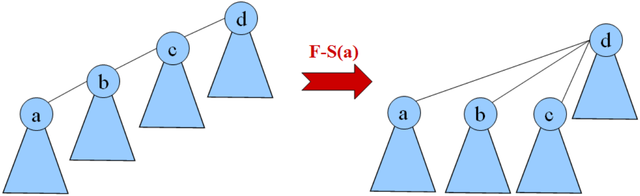
Path Compression Example

find(e)



Two Heuristics

1. Union by Rank
    - Store rank of tree in rep.
      Rank ≈ tree size.
    - Make root with smaller rank point to root with larger rank.
2. Path Compression
    - During Find-Set, "flatten" tree.



Operations

```
Make-Set(x)
    p[x] := x;
    rank[x] := 0
```

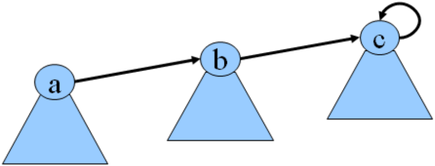
```
Find-Set(x)
    if x ≠ p[x] then
        p[x] := Find-Set(p[x])
    fi;
    return p[x]
```

```
Link(x, y)
    if rank[x] > rank[y] then
        p[y] := x
    else
        p[x] := y;
        if rank[x] = rank[y] then
            rank[y] := rank[y] + 1
        fi
    fi
```

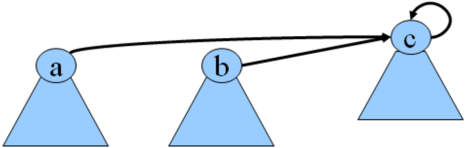
```
Union(x, y)
    Link(Find-Set(x), Find-Set(y))
```

rank = u.b. on height

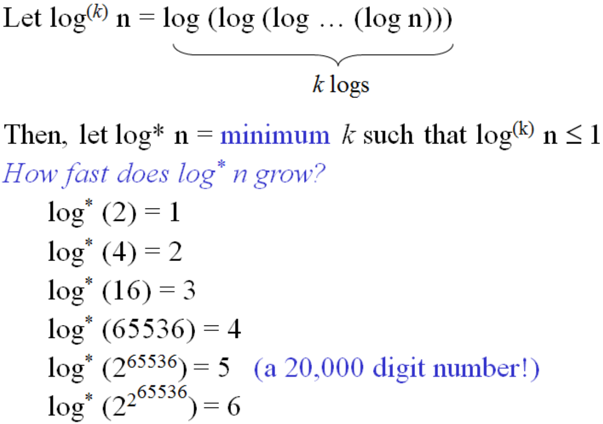
Find Set



```
F-S(a)
  p[a] := F-S(b)
              p[b] := F-S(c)
                          { return c
              return c
  return c
```



A Slow Growing Function



Let $\log^{(k)} n = \underbrace{\log(\log(\log \ldots (\log n)))}_{k \text{ logs}}$

Then, let $\log^* n$ = minimum k such that $\log^{(k)} n \le 1$

How fast does $\log^* n$ grow?

$\log^* (2) = 1$

$\log^* (4) = 2$

$\log^* (16) = 3$

$\log^* (65536) = 4$

$\log^* (2^{65536}) = 5$ (a 20,000 digit number!)

$\log^* (2^{2^{65536}}) = 6$

Complex Complexity of Weighted Union + Path Compression

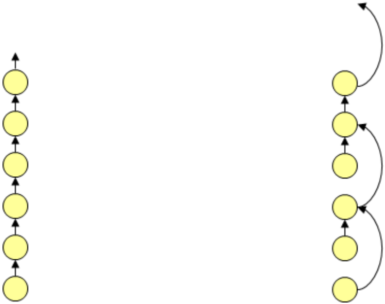
- Tarjan (1984) proved that m weighted union and find operations with path compression on a set of n elements have worst case complexity

  $$O(m \cdot \log^*(n))$$

  actually even a little better!
- For all practical purposes this is amortized constant time ($\log^* n \leq 5$)

One-pass path compression

- We can modify the structure of the tree as each edge is processed to attempt to save on the overall run-time of the algorithm.
- When we do a Find, we can do a bit of ”maintenance” work along the path traversed.



What is happening to the depth of the tree?

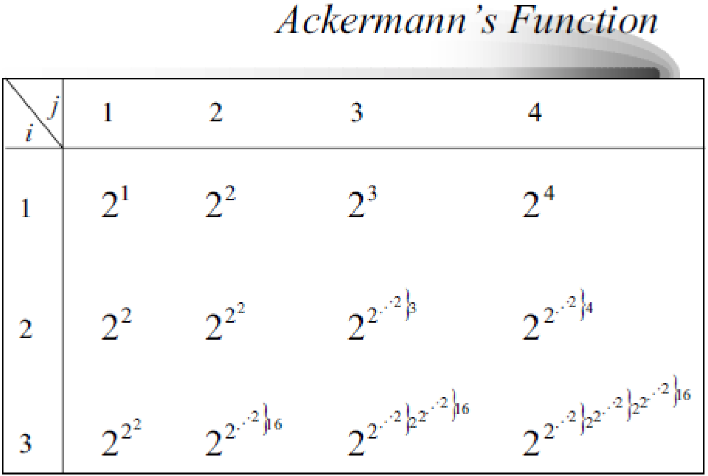
What impact does this work have on future *Find* operations?

Analysis

- n elements (n Make-Sets, at most $n-1$ Unions)
- m Make-Set, Union, and Find-Set operations
- $m = \Omega(n)$
- worst case = $O(m \cdot \alpha(m,n)) = O(m \log^* n)$
- $\alpha(m,n)$ is the inverse of Ackermann's function a very very very slow function
  - $A(1,j) = 2j$ for $j \geq 1$
  - $A(i,1) = A(i-1,2)$ for $i \geq 2$
  - $A(i,j) = A(i-1, A(i,j-1))$ for $i,j \geq 2$

  - $\alpha(m,n) = min\{i \geq 1 : A(i, m/n) > \log n\}$

Ackermann's Function



Ackermann's Function

$i \backslash j$	1	2	3	4
1	2^1	2^2	2^3	2^4
2	2^2	2^{2^2}	$\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}3$	$\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}4$
3	2^{2^2}	$\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}16$	$\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}16$	$\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}16$