chang@umbc.edu

Disjoint Set Union

Problem: Set union is slow because eliminating duplicates requires us to find the intersection.

Solution: Disallow duplicates

$$S_i \cap S_j = \emptyset$$

Example: Assassin Game

Each player issued a target card with name of another player. ← MakeSet

If you "kill" your target, you take over his cards. ← Union

Last player has all of the cards.

Question: Who has my card? ← Find

Union-Find operations

Make-Set (x): create set $\{x\}$

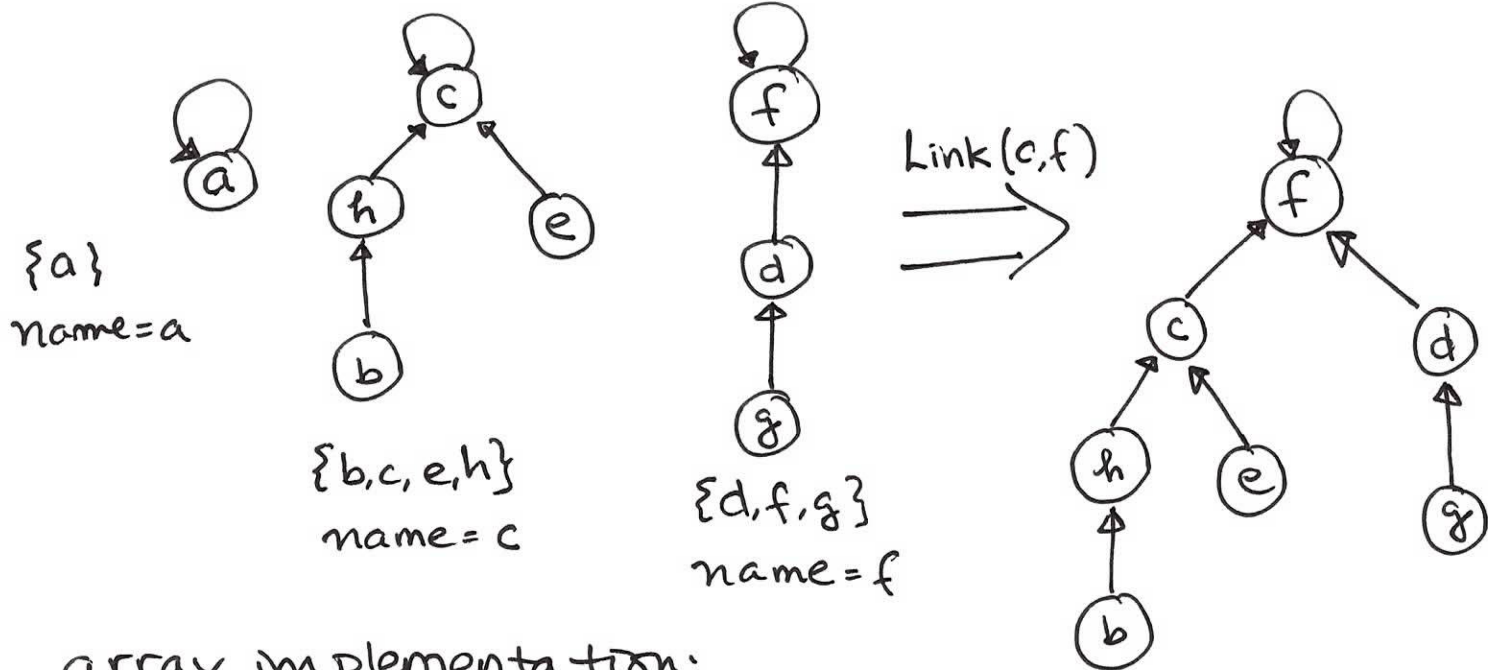
changes over time

Find-Set (x): return "name" of set that contains x.

Link (name1, name2): union of sets named name1 and name2

Union (x, y): join sets containing x & y.

Representation: parent pointers

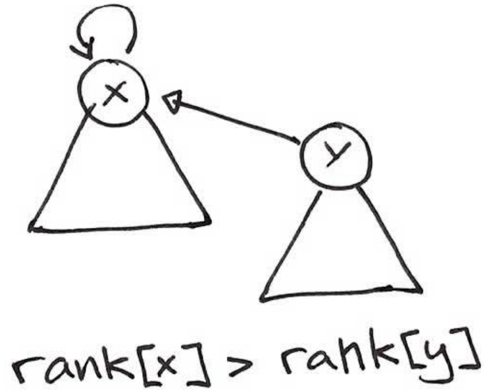


array implementation:

a	b	c	d	e	f	g	h
a	h	c	f	c	f	d	c

Problem: Link(x,y) can create long chains

Solution: Union-by-Rank = higher ranked node is root.




```
Link(x,y) {
    if rank[x] > rank[y]
        p[y] = x
    else {
        p[x] = y
        if rank[x] == rank[y]
            rank[y]++
    }
}
```

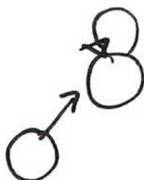
initial rank = 0

rank increases by 1 when
2 equal ranked nodes are linked.

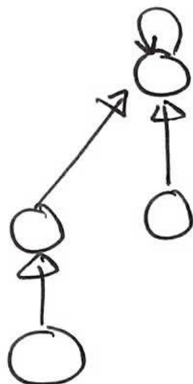
Smallest trees with rank r.

r = 0

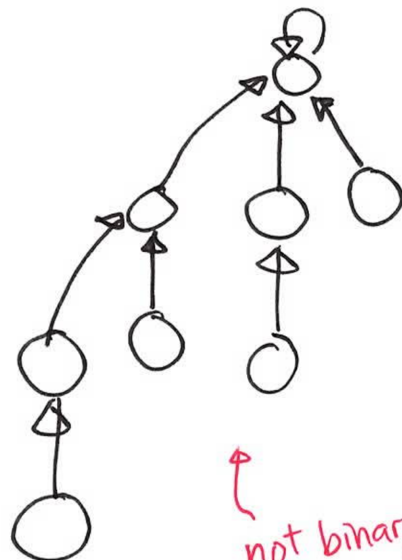
r = 1



r = 2



r = 3



not binary trees

Claim: A node with rank r must have $\geq 2^r$ descendants (incl. itself).

Pf: (induction on r)

true for $r=0$.

The only way to get a node with rank $r+1$ is to link 2 nodes with rank r. By induction, these have $\geq 2^r$ descendants each. New root has $2 \cdot 2^r = 2^{r+1}$ descendants.

Claim: A node with rank r has height $\leq r$.

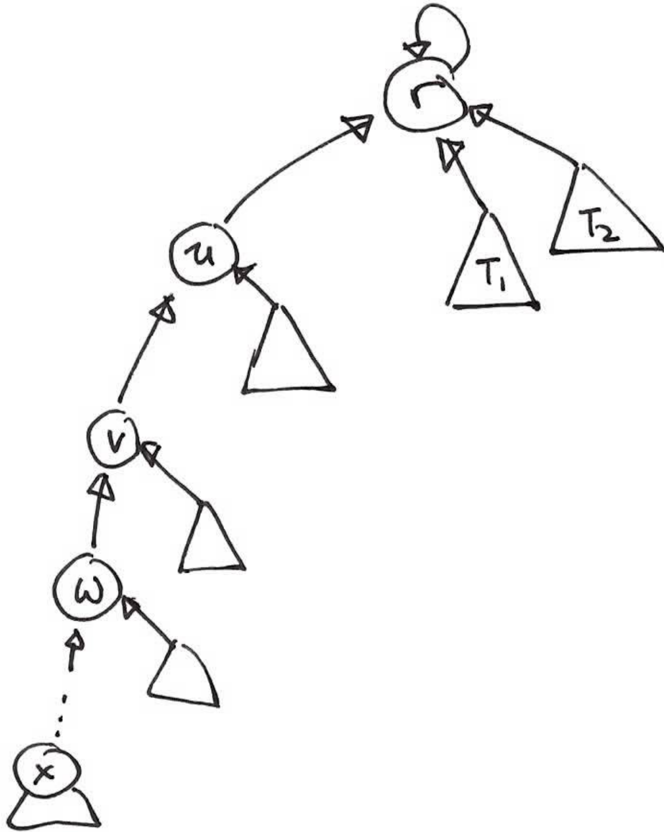
Pf: Easy induction.

Running time of Union-by-Rank:

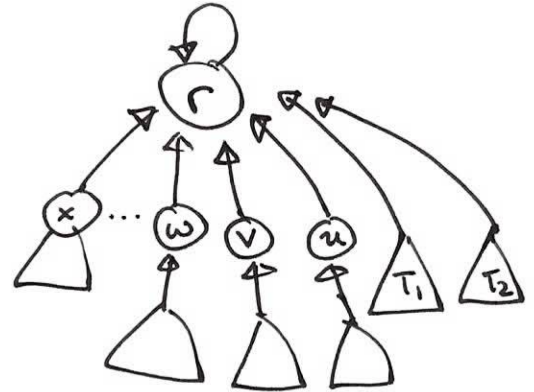
Make-Set	$O(1)$
Find-Set	$O(\log n)$
Link	$O(1)$
Union	$O(\log n)$

← We can do even better (Good enough for Kruskal's)

Path Compression



Find-Set(x)



```
Find-Set (x) {
    r ← x
    while (r ≠ p[r])      // p[r]: parent of
        r ← p[r]

    /* r is root */

    while (x ≠ r) {
        temp ← p[x]
        p[x] ← r          // everybody point to root r
        x ← temp
    }
    return(r)
}
```

Path compression:

- doubles time for Find-Set
- future finds much faster

Amortized Analysis:

Suppose we have m operations including n Make-Sets.

If we use Union-Find in another algorithm (e.g. Kruskal's), we don't care how long each operation takes. We just want to know the total running time for all m operations.

divide by m gives per operation "amortized" running time

With Path Compression <u>and</u> Union-by-Rank, m disjoint set union operations (including n Make-Set operations) takes time $O(m \lg^* n)$.

very slowly growing function.

$$\lg^* n = \min \{ i \geq 0 : \underbrace{\log_2 (\log_2 (\cdots (\log_2 n) \cdots))}_{i \text{ times}} \leq 1 \}$$

$$\lg^* 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} n = n+1$$

$\lg^* 2 = 1$

$\lg^* 4 = 2$

$\lg^* 16 = 3$

$\lg^* 64k = 4$

$\lg^* 2^{64k} = 5$

$\lg^* 10^{80} < 5$

# of atoms in observable universe

Ackerman's function

$$A(1,j) = 2^j \quad , j \geq 1$$

$$A(i,1) = A(i-1,2) \quad i \geq 2$$

$$A(i,j) = A(i-1, A(i,j-1)) \quad i,j \geq 2$$

A	j = 1	2	3	4
i = 1	2	2^2	2^3	2^4
i = 2	2^2	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
i = 3	2^{2^2}	$2^{2^{\cdot^{\cdot^{2}}}}\}16$	$2^{2^{\cdot^{\cdot^{2}}}}\}$ $2^{2^{\cdot^{\cdot^{2}}}}\}16$	

← stack of j 2's

Ackerman's function (cont'd)

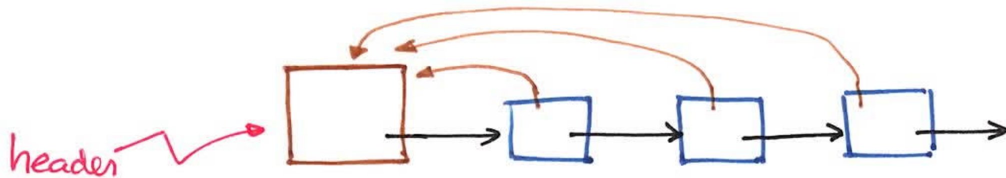
$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) > \lg n\}$$

$A(4,1) > 10^{80}$ = # of atoms in universe

$\alpha(m,n) < 4$ for all practical purposes

Example of Amortized Analysis of Union-Find.

Linked list with root pointer.

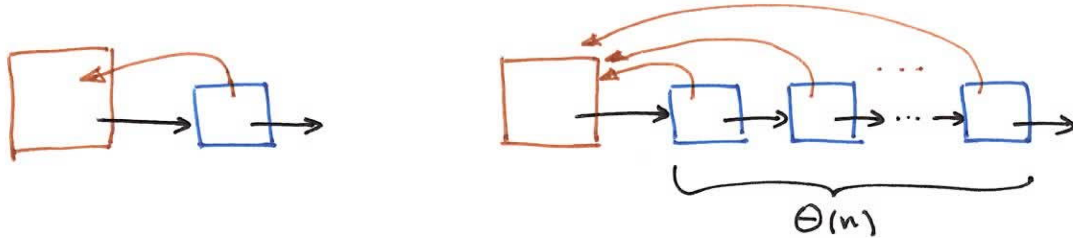


Make-Set	$\Theta(1)$	worst case running time
Find-Set	$\Theta(1)$	
Link	$\Theta(n)$	
Union	$\Theta(n)$	

Link & Union worst case: update root pointers of $\Theta(n)$ nodes

Each Union might take $\Theta(n)$ time.

If we had m unions, how much time does that take?



Idea: make nodes in shorter list point to root of longer list.

1. Fewer pointer updates

2. When a node's root pointer is updated, the size of its list at least doubles

3. Each node's pointer can be updated at most $\log n$ times. Size of list cannot exceed n.

4. Worst case Union cannot happen very often.

Amortized Analysis

- Estimate the running time of a sequence of instructions (rather than a single one).

- Often "object-oriented":
  Estimate the time spent on each piece of data rather than time spent on each line of code

- Credits for time.

Back to linked lists with root pointers:

- Time for Union is proportional to number of pointer updates.

- m operations including n Make-Sets: total time for all Unions is $O(n \log n)$.

  There are n nodes, each root pointer can be updated at most $\log n$ times.

- Amortized time for Union: $\log n$.

$\frac{O(n \log n)}{m} \leq \log n$ since $m \geq n$

Credits:

Alternate method for amortized analysis

1. Think prepaid cellphones: use "units" to pay for running time.

2. Use 1 credit for each pointer update.

3. Prepay for $\log n$ credits at each node.

4. Can "charge" the credits to other operations. (think parents & cellphones)

5. Charge credits to Make-Set operations:

Make-Set	$\Theta(\log n)$	Link	$\Theta(1)$
Find-set	$\Theta(1)$	Union	$\Theta(1)$

Link and Union: pay for pointer updates using prepaid credits

Getting $\log \log n$ amortized time...

- Use disjoint set forests, union-by-rank, path compression
- Link is $\Theta(1)$
- Must pay for path compression
  - Charge some credits to Find-Set
  - Charge some credits to nodes
  - Don't have to charge nodes equally

} depends on rank of node & rank of parent

Rank(x) - not exactly what you think it is.

- Rank(x) used to be height of x, but path compression messes that up.
- Once x becomes non-root, rank(x) does not change
- Let size(x) = # of items in subtree rooted at x including x.
- Claim: $size(x) \geq 2^{rank(x)}$ if x is a root.

Easy induction. Rank increases only during union-by-rank. Size must at least double.

when equal ranked roots are linked.

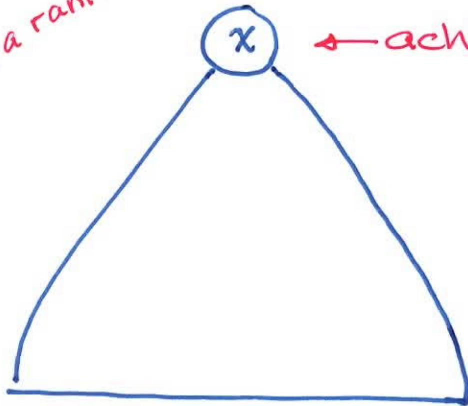
- For all x, $\text{rank}(x) \le \lfloor \log n \rfloor$.

  Otherwise, $\text{size}(x) > n$.

- At most $n/2^r$ items ever achieve rank r.

  during entire run of disjoint set union.

Fix a rank r.



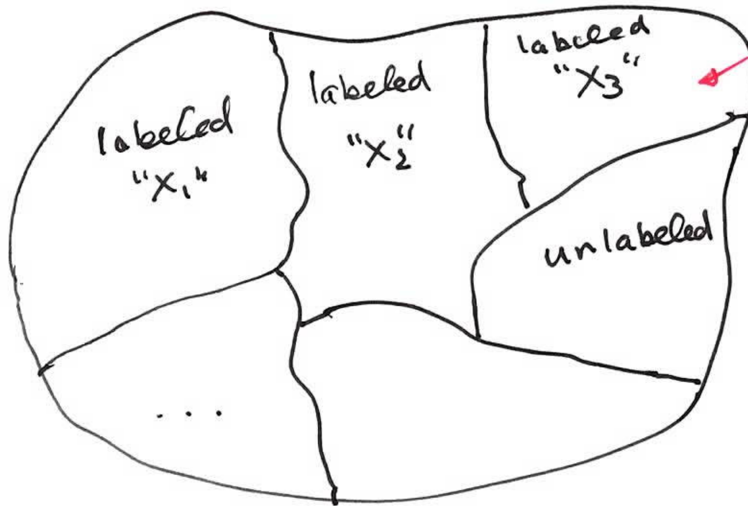
label all descendants "x"

Each item can only ever have 1 ancestor w/ rank r

At least 2^r nodes in subtree

n nodes either unlabeled or labeled by unique "x".

Can't have more than $n/2^r$ labels.

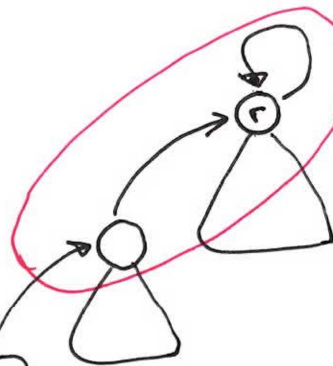


each partition has at least 2^r items

Can't have more than $n/2^r$ partitions.

Path Compression

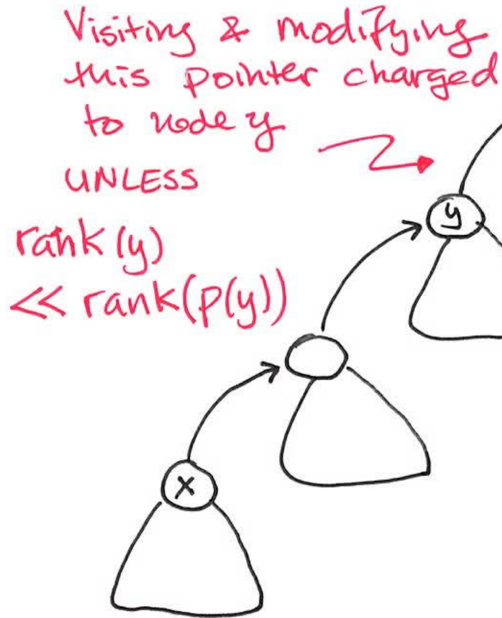
Find-Set(x)
must make pointers
point to root r.



Visit to root & child of root charged to Find-Set

parent of y

p(y)



Visiting & modifying this pointer charged to node y
UNLESS
rank(y) $\ll$ rank(p(y))

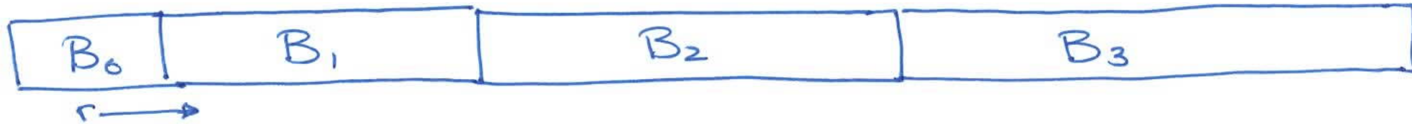
What does "$\ll$" mean?

note: ① rank(y) < rank(p(y)) is always true.

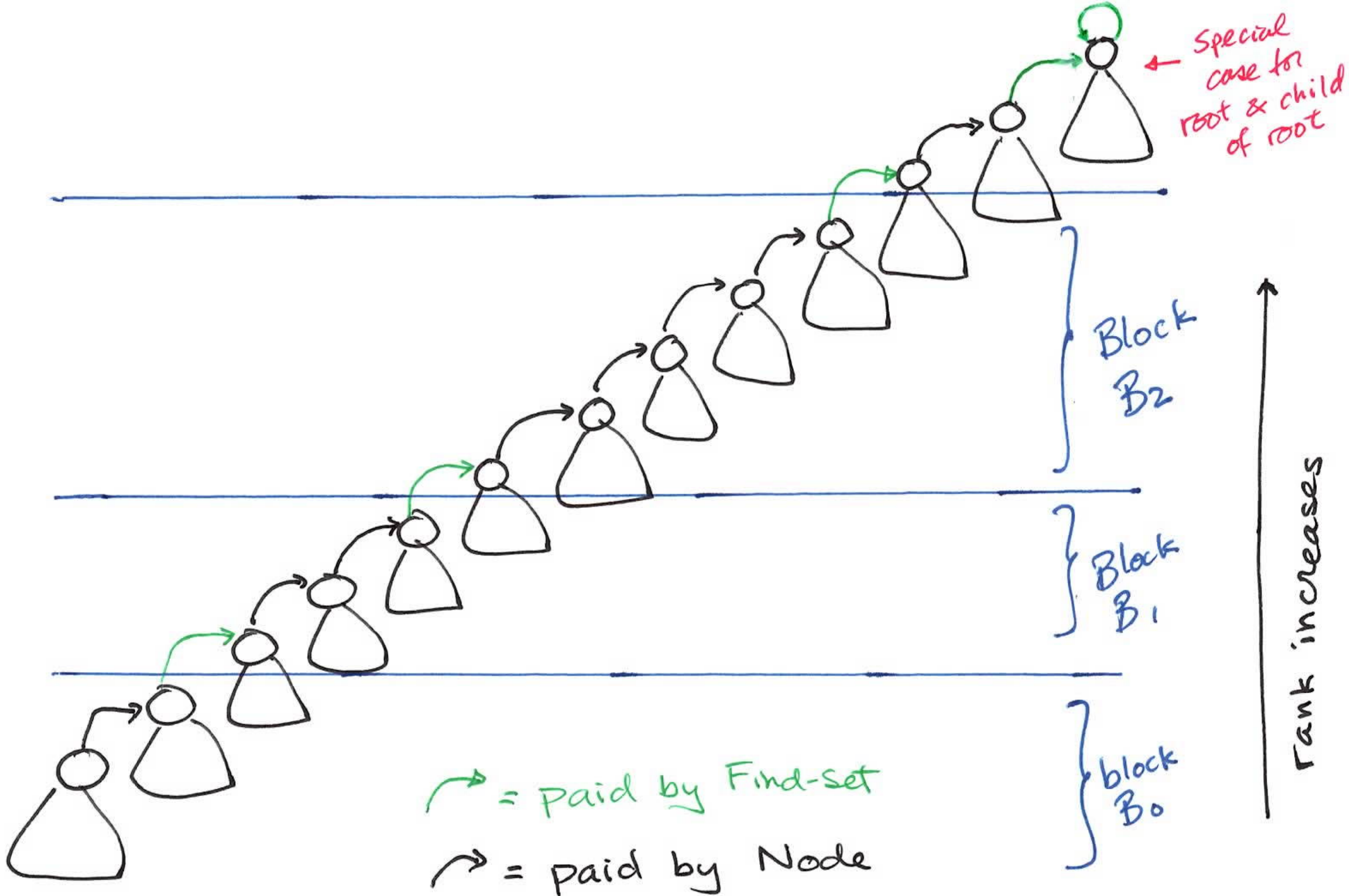
② rank(p(y)) can increase because p(y) changes after path compression.

Idea: Node y pays for modifying its parent pointer until rank(p(y)) gets too large. Then, Find-Set pays.

Divide ranks into "blocks"



if rank(y) & rank(p(y)) are in different blocks, then we think rank(y) $\ll$ rank(p(y)) and Find-Set pays.



Special case for root & child of root
Block B_2
Block B_1
block B_0
rank increases
= paid by Find-set
= paid by Node

We want Find-Set to run in $O(\log\log n)$ amortized time, so we can't have too many blocks.

- Limit # of blocks to $\log\log n$.
- Have exponentially increasing block sizes. ← useful later

$B_0 = \{0\}$
$B_1 = \{1\}$
$B_2 = \{2, 3\}$
$B_3 = \{4, 5, 6, 7\}$
$B_4 = \{8, 9, 10, \ldots, 15\}$
$B_5 = \{16, 17, 18, \ldots, 31\}$
$B_6 = \{32, 33, 34, \ldots, 63\}$
$B_i = \{2^{i-1}, \ldots, 2^i - 1\}$

$|B_i| = 2^{i-1}$ for $i > 0$

# of blocks $\leq \log\log n$. ← max rank $\leq \log n$

Items with higher rank are charged more often until rank of their parents move to next block.

(Each time y is involved in path compression, the rank of its parent must increase by at least 1.)

Example: $B_3 = \{4, 5, 6, 7\}$

if rank(y) = 4, then after 3 path compressions rank(p(y)) must be in B_4 or higher.

Example: $B_6 = \{32, 33, 34, \ldots, 63\}$

if rank(y) = 32, then it might take 31 path compressions before rank(p(y)) moves to block B_7.

THIS IS OK because there are very few items with high rank.

Recap: n items, m operations w/ n Makesets

Link $\Theta(1)$

Union = 2 Find sets + 1 Link

Find Set $\Theta(\log \log n)$ amortized

Make Set $\Theta(1)$ + node charges $= \Theta(\log \log n)$ amortized

(node charges) depends on rank, but we don't know the rank, yet.

Solution: pool together all node credits.

Total # of node charges:

$$\sum_{i=1}^{\log\log n} \sum_{r \in B_i} \left(\frac{n}{2^r}\right) 2^{i-1} = n \sum_{i=1}^{\log\log n} \sum_{r=2^{i-1}}^{2^i - 1} \frac{2^{i-1}}{2^r}$$

max # of nodecharges for block Bi

sum over blocks

each rank r in blocks

# of items w/ rank r

$$\leq n \sum_{i=1}^{\log\log n} \left(2^{i-1} \cdot \frac{2^{i-1}}{2^{2^{i-1}}} \right)$$

# of r's

smallest r in Bi

$$= n \sum_{i=1}^{\log\log n} \frac{2^{2i-2}}{2^{2^i - 1}}$$

$$< n \sum_{i=1}^{\log\log n} 1 = n \log\log n$$

If we charge each of the n MakeSets log log n credits, then there are enough credits for all node charges. Some nodes charged more than others.

Even better bounds

$m \log\log\log n$: Make block sizes double exponential

$$B_i = \{2^{2^{i-1}}, \ldots, 2^{2^i} - 1\}$$

$m \log^* n$: block size = stack of 2's

$$B_i = \{b_i, \ldots, b_{i+1} - 1\}$$

where $b_i = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ } $i-1$ times.

easy

See notes online for calculations.