2IL50 Data Structures

2023-24 Q3

Lecture 10: Data Structures for Disjoint Sets

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Abstract data type

Abstract Data Type (ADT)

A set of data values and associated operations that are precisely specified independent of any particular implementation.

Dictionary, stack, queue, priority queue, set, bag ...

Dynamic sets

Dynamic sets

Sets that can grow, shrink, or otherwise change over time.

Two types of operations:

- queries — return information about the set
- modifying operations — change the set

Common queries

Search, Minimum, Maximum, Successor, Predecessor

Common modifying operations

Insert, Delete

Union-find structure

Union-Find Structure

Stores a collection of disjoint dynamic sets.

Operations

Make-Set(x): creates a new set whose only member is x

Union(x, y): unites the dynamic sets that contain x and y

Find-Set(x): finds the set that contains x

Union-find structure

Union-Find Structure

Stores a collection of disjoint dynamic sets.

every set S_i is identified by a representative

(It doesn't matter which element is the representative, but if we ask for it twice, without modifying the set, we need to get the same answer both times.)

Operations

Make-Set(x): creates a new set whose only member is x

(x is the representative.)

Union(x, y): unites the dynamic sets S_x and S_y that contain x and y

(Representative of new set is any member of S_x or S_y, often one of their representatives. Destroys S_x and S_y since sets must be disjoint.)

Find-Set(x): finds the set that contains x

(Returns the representative of the set containing x, assumes that x is an element of one of the sets.)

Analysis of union-find structures

Union-find structures are often used as an auxiliary data structure by algorithms

➡ total running time over all operations is more important than worst case running time for each operation

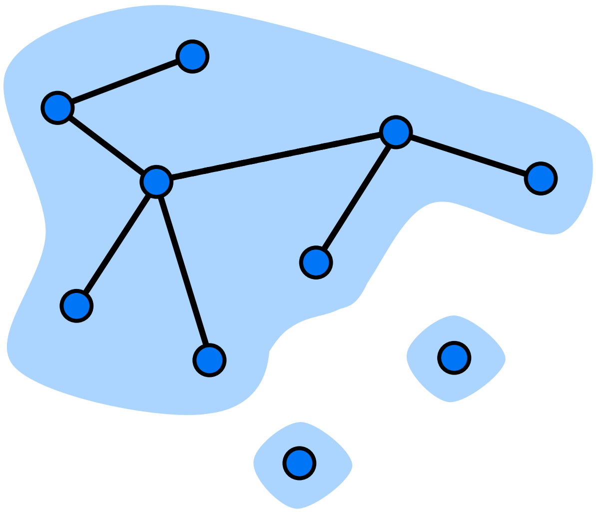
Analysis in terms of

n = # of elements = # Make-Set operations

m = total # of operations (incl. Make-Set)

Example application: connected components

Maintain the connected components of a graph $G = (V, E)$ under edge insertions.



Same-Component(u, v)

if Find-Set(u) == Find-Set(v)
  return true
else
  return false

Connected-Components(V, E)

for each vertex $v \in V$
  Make-Set(v)
for each edge $(u, v) \in E$
  Insert-Edge(u, v)

Insert-Edge(u, v)

if Find-Set$(u) \neq$ Find-Set(v)
  Union(u, v)

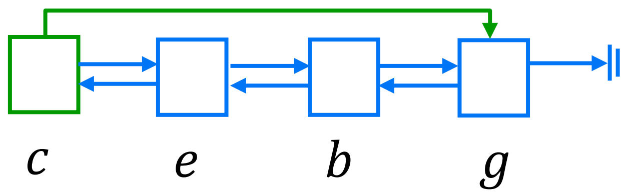
Data Structures for union-find: Solution 1

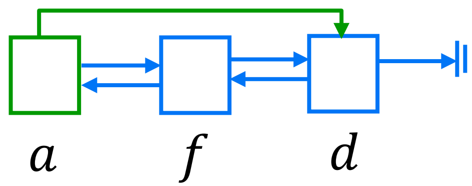
Store every set S_i in a doubly-linked lists

Representative: first element of the list

The prev-pointer of the first element points to the last element

$S_1 = \{\, c, e, b, g \,\}$ $\qquad$ $S_2 = \{\, a, f, d \,\}$





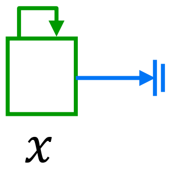
x is the representative if $x.\mathrm{prev}.\mathrm{next} = NIL$

Disclaimer: This is not quite the same solution as in Chapter 19 of the textbook ...

Solution 1: Make-Set and Find-Set

Make-Set(x)

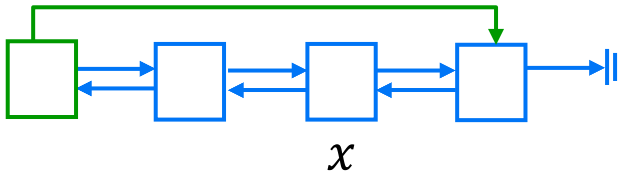
$x.\text{prev} = x$
$x.\text{next} = NIL$



Note: x is a pointer to an element in the list and hence we do not need to search.

Find-Set(x)

if $x.\text{prev}.\text{next} \neq NIL$
    return Find-Set($x.\text{prev}$)
return x



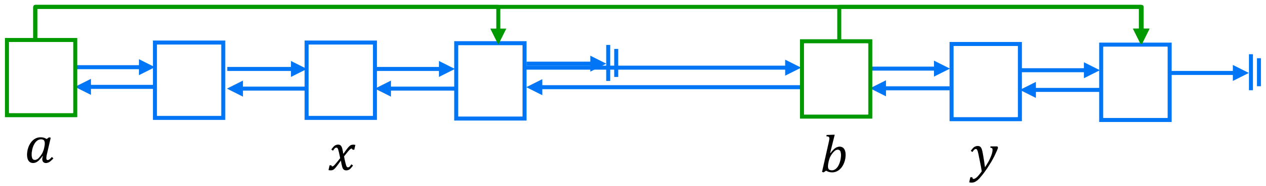
Solution 1: Union

Union(x, y)

// assumes x and y are elements of different sets

1 $a =$ Find-Set(x); $b =$ Find-Set(y)

2 append the list of b onto the end of the list of a



Analysis Solution 1

Make-Set(x): $O(1)$
Find-Set(x): $O(\text{size of set that contains } x)$
Union(x, y): 2 Find-Set + $O(1)$ $= O(\text{size of both sets})$

Total running time for m operations, of which n are Make-Set:

Each set has size $\leq n$ ➡ total running time $O(mn)$

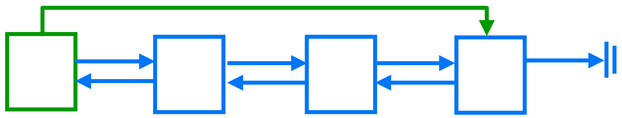
Is this possible at all?!?

Yes

$$\underbrace{\begin{array}{l}\text{Make-Set}(x_1), \ldots, \text{Make-Set}(x_n) \\ \text{Union}(x_2, x_1), \text{Union}(x_3, x_1), \ldots, \text{Union}(x_n, x_1) \\ \text{Find-Set}(x_1), \text{Find-Set}(x_1), \text{Find-Set}(x_1), \ldots\end{array}}_{(m - 2n + 1)}$$

Problems with Solution 1

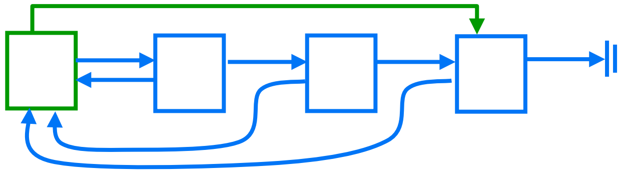
Problem: Find-Set takes too long



Solution 2

Replace $x.\mathrm{prev}$ pointer with a $x.\mathrm{rep}$ pointer to the representative

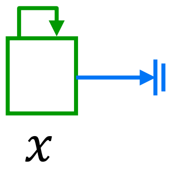
The rep-pointer of the representative points to the last element



Solution 2: Make-Set and Find-Set

Make-Set(x)

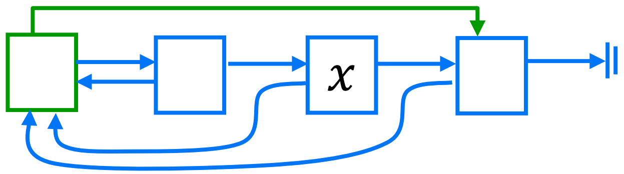
$x.\mathrm{rep} = x$
$x.\mathrm{next} = NIL$



Find-Set can now be executed in $O(1)$ time:

Find-Set(x)

if $x.\mathrm{rep.next} == NIL$
        return x
return $x.\mathrm{rep}$



Solution 2: Union

Union(x, y)

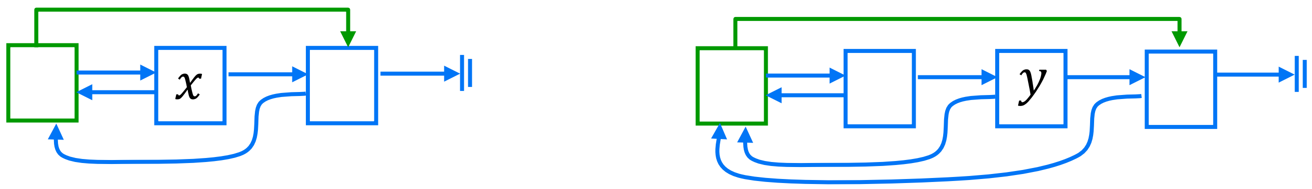
// assumes x and y are elements of different sets

a = Find-Set(x); b = Find-Set(y)

append the list of b onto the end of the list of a

update all rep-pointers

Running time? O(size of set that contains y)



Analysis Solution 2

Make-Set(x): $O(1)$
Find-Set(x): $O(1)$
Union(x, y): $O(\text{size of set that contains } y)$

Total running time for m operations, of which n are Make-Set:

Let's check the worst case example for Solution 1 ...

Worst case for Solution 1

Make-Set(x_1), … , Make-Set(x_n)

Union(x_2, x_1), Union(x_3, x_1), … , Union(x_n, x_1)

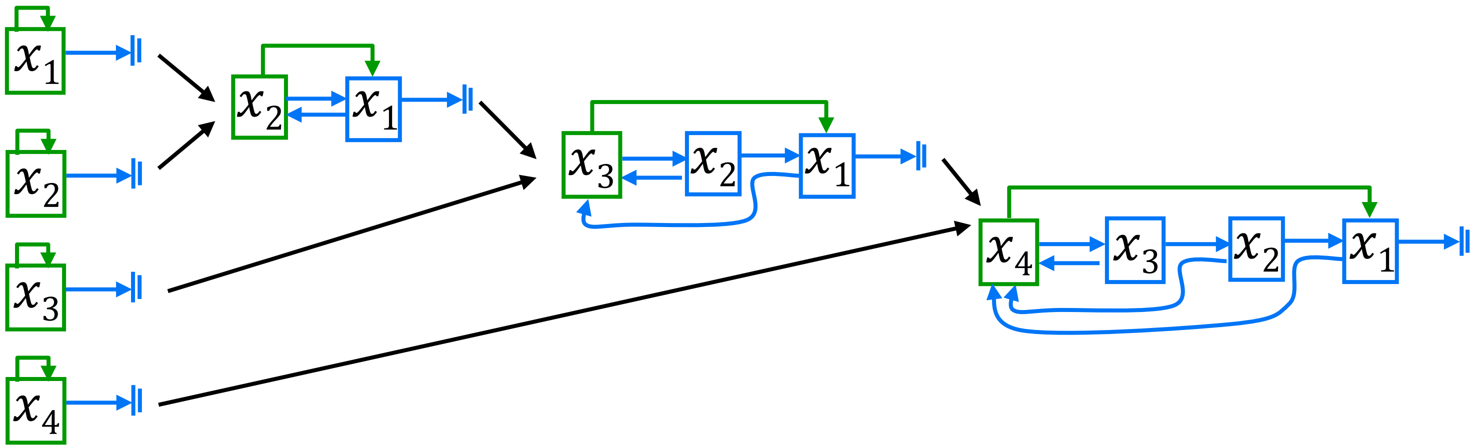
$\underbrace{\text{Find-Set}(x_1),\ \text{Find-Set}(x_1),\ \text{Find-Set}(x_1),\ \ldots}_{m-2n+1}$

$\Theta(n)$

$\sum_{2\le i\le n}\Theta(i) = \Theta(n^2)$

$\Theta(m-2n)$

Total: $\Theta(m+n^2)$

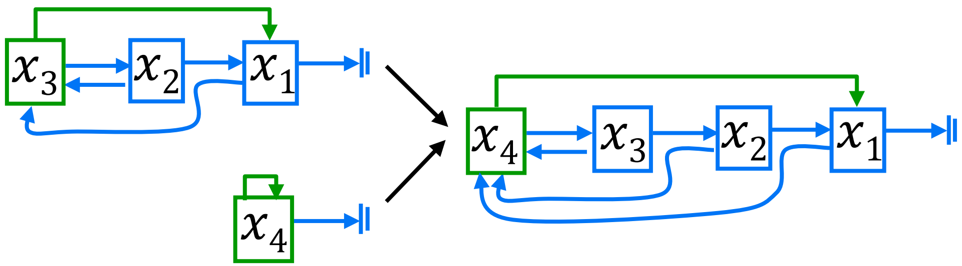


Make-Set(x) and Find-Set(x): $O(1)$

Union(x, y): O(size of set that contains y)

Problems with Solution 2

What is the problem?



Appending $\{x_3, x_2, x_1\}$ onto $\{x_4\}$ was not a great idea ...

Solution 3 — Always append the shorter list onto the longer list
Less rep-pointers need to be updated

Union-by-size

Solution 3

Solution 3 The same as Solution 2, but

Store with each list its length (*this can be easily maintained*)

Union(x, y) always appends the shorter onto the longer list

Theorem

A sequence of m operations, of which n are Make-Set, takes $\Theta(m + n \log n)$ time in the worst case.

We can do even better ...

Proof Make-Set and Find-Set cost $\Theta(1)$ per operation $O(m)$ in total.

Time for all Union operations

$= O$(total number of times that a rep-pointer was moved)

$= \sum_x$(number of times that $x.\mathrm{rep}$ was moved)

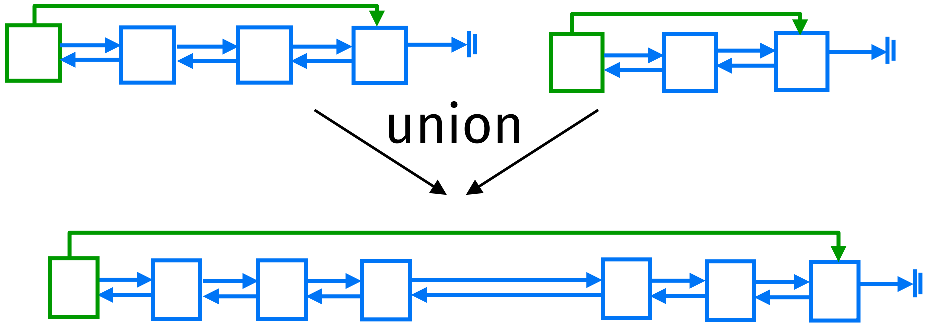
$= \sum_x O(\log n) = O(n \log n)$ ∎

Can it really be $\Omega(m + n \log n)$? Yes.

Solution 4

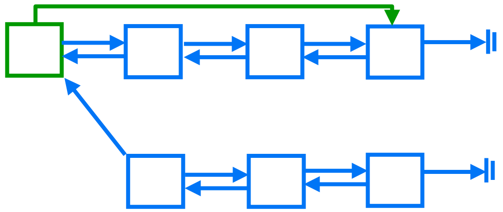
Solution 1

append one list onto the other



New idea

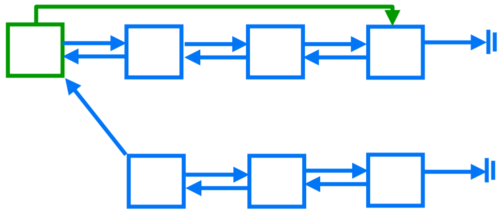
append one list directly onto (under) the representative of the other



Solution 4

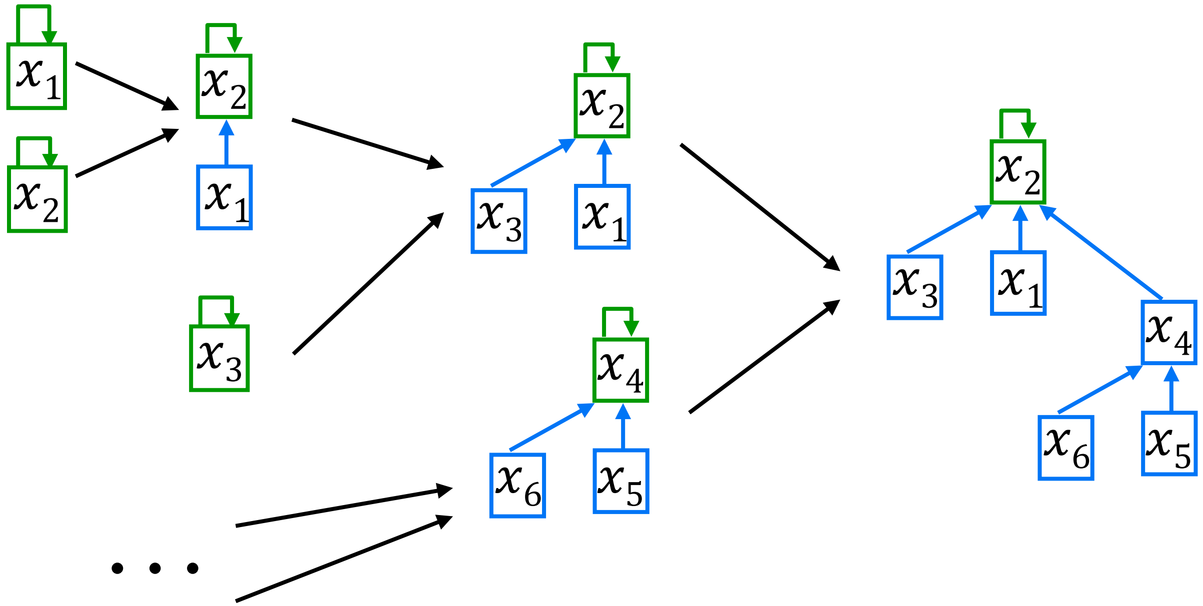
New idea

append one list directly onto (under) the representative of the other



next-pointers are not needed anymore

the rep-pointer of the representative points to the representative



a sort of tree structure …

disjoint-set forest

Disjoint-set forest: The data structure

Each set is stored in a tree;
nodes have only a pointer $x.p$ that points to their parent.

The root is the representative of the set;
the parent-pointer $x.p$ of the root points to the root.

We need to know the height of each tree to attach the smaller tree to the larger

Each node x has a field $x.\mathrm{rank}$, which is an upper bound for the height of x.

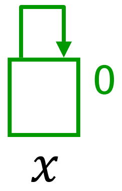
height of x = the number of edges in the longest path between x and a descendant leaf

Union-by-rank

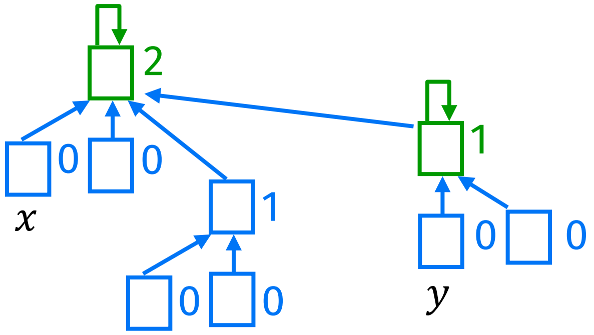
Disjoint-set forest: Make-Set

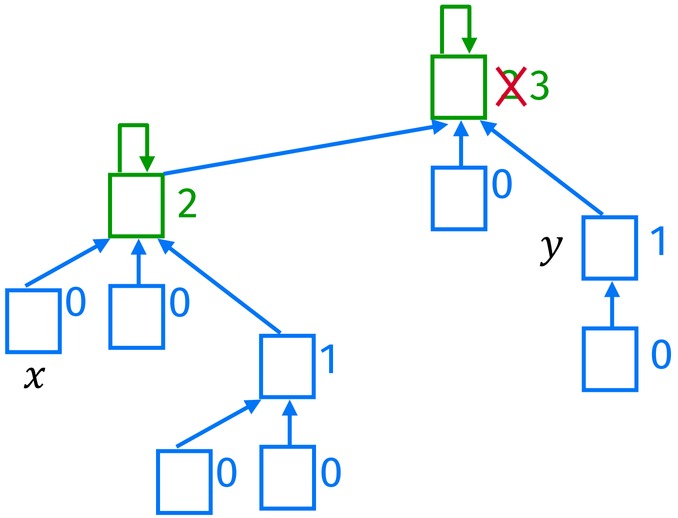
Make-Set(x)

$x.p = x$
$x.\text{rank} = 0$



Disjoint-set forest: Union



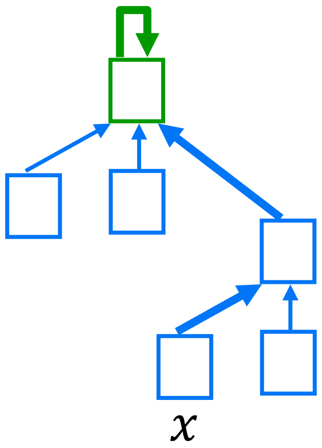


Union(x, y)

$a = $ Find-Set(x); $b = $ Find-Set(y)
if $a.\text{rank} > b.\text{rank}$
    $b.p = a$
else
    $a.p = b$
    if $a.\text{rank} == b.\text{rank}$
        $b.\text{rank} = b.\text{rank} + 1$

Disjoint-set forest: Find-Set

Find-Set(x)
```
1  if x ≠ x.p
2      return Find-Set(x.p)
3  return x
```



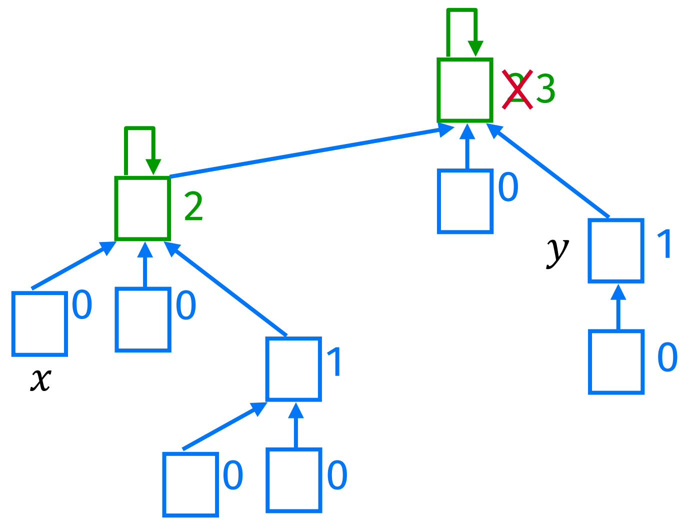
Analysis disjoint-set forest

Lemma (# elements in the tree rooted at x) $\geq 2^{x.\text{rank}}$

Proof Induction on $r = x.\text{rank}$

Base case: $r = 0$

elements $\geq 1 = 2^0$ ✓



Inductive step: $r > 0$

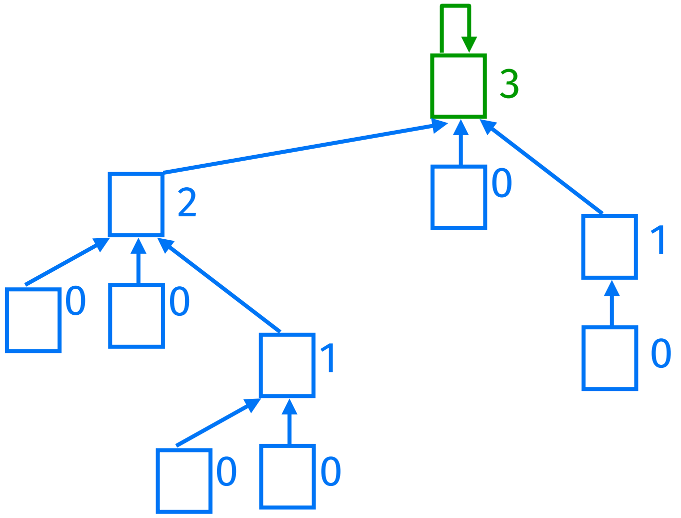
a node x with rank r is created by joining two trees with roots of rank $r - 1$

➡ (# elements in new subtree rooted at x) $\geq 2 \cdot 2^{r-1} = 2^r$ ∎

This immediately implies $x.\text{rank} \leq \log n$

Analysis disjoint-set forest

Theorem A sequence of m operations, of which n are Make-Set, takes $O(m \log n)$ time in the worst case.



Proof

$x.\mathrm{rank} \leq \log n$

the rank of nodes on the find path increases by at least one in every step

➡ maximal length of find path = maximal rank $\leq \log n$

➡ Find-Set takes $O(\log n)$ time

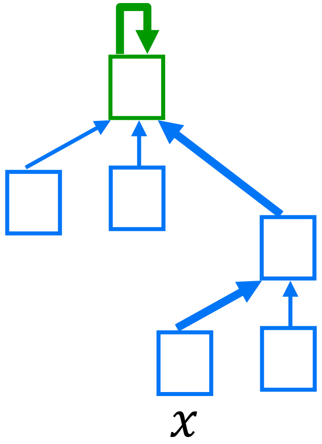
Make-Set and Union (excl. Find-Set) both take $O(1)$ time ■

But Solution 3 works in $O(m + n \log n)$ … ?!?

Disjoint-set forest: Find-Set (again)

Find-Set(x)

if $x \neq x.p$
  return Find-Set$(x.p)$
return x

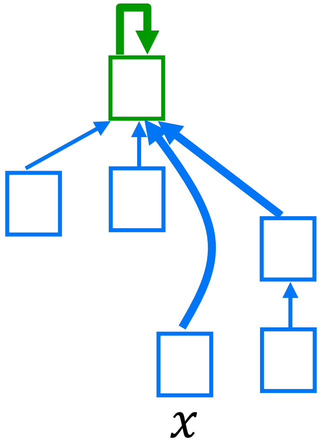


Path compression

Find path = nodes visited during Find-Set on the path to the root.
Make all nodes on the find path direct children of the root.

Find-Set(x)

if $x \neq x.p$
  $x.p =$ Find-Set$(x.p)$
return $x.p$



Analysis disjoint-set forest

Theorem A sequence of m operations, of which n are Make-Set, takes $O(m\,\alpha(n))$ time in the worst case.

$\alpha(n)$ is a function that grows extremely slow

$\alpha(n) \leq \log^* n$

Number of times that one has to take a $\log$ before getting to 1 or below:

$\log^* 2 = 1$ $\quad$ $\log^* 2^2 = 2$ $\quad$ $\log^* 2^4 = 3$

$\log^* 2^{16} = 4$ $\quad$ $\log^* 2^{65536} = 5$

Proof *is somewhat complicated …*

we will prove $O(m \log^* n)$

Analysis disjoint-set forest

Theorem A sequence of m operations, of which n are Make-Set, takes $O(m \log^* n)$ time in the worst case.

Proof

Make-Set and Union (excl. Find-Set) both take $O(1)$ time

There are n Make-Set and at most $n - 1$ Union operations

➡ in total $O(n)$ time for all Make-Set and Union (excl. Find-Set) operations

remains to show:

m Find-Set operations can be executed in $O(m \log^* n)$ time

The $\log^* n$ function

Define function $t\colon \mathbb{N} \to \mathbb{N}$ as

$$t(i) = \begin{cases} 1 & \text{if } i = 0 \\ 2^{t(i-1)} & \text{if } i > 0 \end{cases}$$

i	0	1	2	3	4	5
$t(i)$	1	2	4	16	65,536	$2^{65,536}$

$$\log^* n = \min\{i\colon\ t(i) \geq n\}$$

Note $\quad \log^* t(i) = i \quad$ and $\quad \log^* t(0) = 0$

Rank groups

Divide nodes into rank groups: node x is in rank group g if $g = \log^*(x.\mathrm{rank})$

➡ $t(g-1) < x.\mathrm{rank} \leq t(g)$ for $x.\mathrm{rank} > 1$ *rank group 0 contains ranks 0 and 1*

Lemma (# nodes in rank group g) $\leq n/t(g)$ *obvious for $g = 0$, proof holds for $g > 0$*

Proof (# nodes in rank group g)

$$\leq \sum_{t(g-1)+1 \leq r \leq t(g)} (\text{\# nodes with rank } r)$$

$$\leq \sum_{t(g-1)+1 \leq r \leq t(g)} n/2^r$$

$$= n/2^{t(g-1)+1} \cdot \sum_{0 \leq r \leq t(g)-t(g-1)-1} 1/2^r$$

$$< n/2^{t(g-1)+1} \cdot 2$$

$$= n/2^{t(g-1)}$$

$$= n/t(g)$$

■

Lemma

(# elements in the tree rooted at x) $\geq 2^{x.\mathrm{rank}}$

➡ (# nodes with rank r) $< n/2^r$

Analysis disjoint-set forest: Find-Set

Lemma m Find-Set operations can be executed in $O(m \log^* n)$ time.

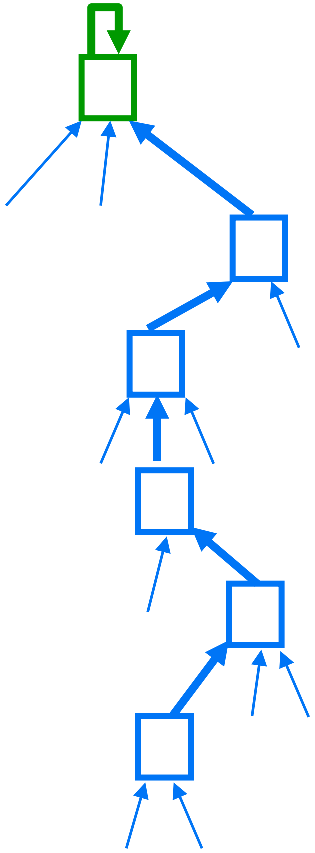
Proof Idea: bound # parent pointers on all find paths
Note: applies to $n > 1$

Three cases:

(i) pointer to root
2 per find path $O(m)$ in total ✓

(ii) pointer from node y to $y.p$ with $\mathrm{group}(y.p) > \mathrm{group}(y)$
highest rank is $\leq \log n$
groups $\leq \log^*(\log n) + 1 = \log^* n - 1 + 1 = \log^* n$
at most $\log^* n$ per find path $O(m \log^* n)$ in total ✓

(iii) pointer from node y to $y.p$ with $\mathrm{group}(y.p) = \mathrm{group}(y)$



Analysis disjoint-set forest: Find-Set

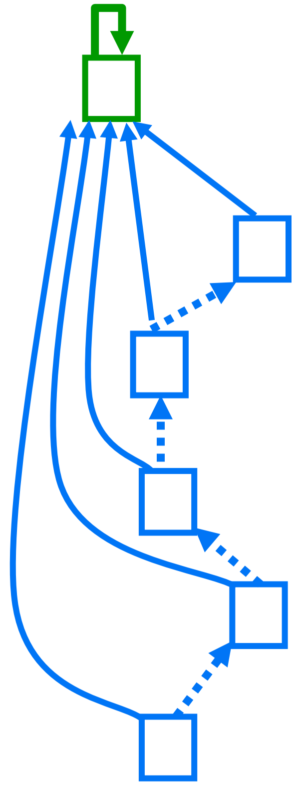
(iii) pointer from node y to $y.p$ with $\mathrm{group}(y.p) = \mathrm{group}(y)$

after following the pointer $y.p$, y will get a new parent because of path compression

ranks are monotonically increasing

➡ (new parent). rank > (previous parent). rank

if the new parent is in a higher group, y will never be in Case (iii) again (*the rank of a node that is not a root never changes*)



Q How often can Case (iii) occur for one node y?

A At most # different ranks in y's rank group

Total for Case (iii)

$$\sum_{\mathrm{nodes}\ y} (\text{\# ranks in rank group of } y)$$

$$= \sum_{1 \le g \le \log^* n - 1} \sum_{y \text{ in rank group } g} (\text{\# ranks in group } g)$$

Analysis disjoint-set forest: Find-Set

Q How often can Case (iii) occur for one node y?

A At most # different ranks in y's rank group

Total for Case (iii)

$$\sum_{\text{nodes } y} (\text{\# ranks in rank group of } y)$$

$$= \sum_{1 \le g \le \log^* n - 1} \sum_{y \text{ in rank group } g} (\text{\# ranks in group } g)$$

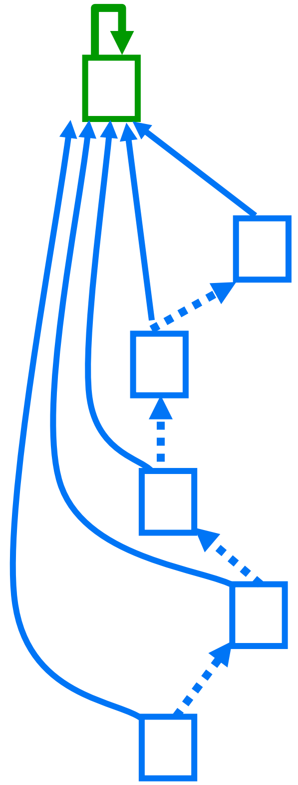
$$\le \sum_{1 \le g \le \log^* n - 1} (n / t(g)) \cdot (t(g) - t(g-1))$$

$$= n \cdot \sum_{1 \le g \le \log^* n - 1} \left(1 - \frac{t(g-1)}{t(g)}\right)$$

$$= n \log^* n - n \cdot \sum_{1 \le g \le \log^* n - 1} t(g-1) \cdot \left(\frac{1}{2}\right)^{t(g-1)}$$

$$\le n \log^* n - n \cdot 2$$

$$= O(n \log^* n)$$



Analysis disjoint-set forest

Theorem

If we implement a union-find data structure with a collection of trees, using the union-by-rank heuristic and the path-compression heuristic, then a sequence of m operations, of which n are Make-Set, takes $O(m \log^* n)$ time in the worst case.