Disjoint Set Union via Compressed Trees

(with Randomized Linking)

A. Goel, S. Khanna, D. Larkin,& R. Tarjan

Disjoint set union (Union-Find)

Devise a data structure for an intermixed sequence of the following kinds of operations:

make-set(x) (x in no set): create a set $\{x\}$, with *root* x.

find(x): (x in a set): return the root of the set containing x.

unite(x, y): if x and y are in the same set, return **false**; otherwise combine the sets containing x and y into a single set, choose an element as the root of the new set, and return **true**.

Each element is in at most one set (sets are *disjoint*): *equivalence relation* on elements

The root of a set serves to identify it, can store information about the set (size, name, etc.)

Implementation can choose each set root

Applications

Global greedy MST algorithm (Kruskal)

FORTRAN compilers: COMMON and EQUIVALENCE statements

Incremental connected components

Percolation

Disjoint set implementation

Represent each set by a rooted tree, whose nodes are the elements of the set, with the set root the tree root, and each node x having a pointer to its parent $x.p$. Store information about set (such as name) in root.

The shape of the tree is *arbitrary*.

n = #elements, m = #finds, $n > 1$, $n = O(m)$

Set operations

make-set(*x*): make x the root of a new one-node tree: $x.p \leftarrow x$

find(*x*): follow parent pointers from *x* to the root:

if $x.p = x$ **then return** *x* **else return** *find*(*x.p*)

unite(*x*, *y*): $v \leftarrow$ *find*(*v*); $w \leftarrow$ *find*(*y*);

if $v = w$ **then return false**

else {*link*(*v*, *w*); **return true**}

link(*v*, *w*): $v.p \leftarrow w$ (*or* $w.p \leftarrow v$)

A bad sequence of unites can create a tree that is a path of n nodes, on which each *find* can take $\Omega(n)$ time, totaling $\Omega(mn)$ time for m *finds*

Goal: reduce the amortized time per *find*: reduce node depths

Improve *unites*: linking by *size* or by *rank*

Improve *finds*: *compact* the trees

Linking by size: maintain the number of nodes in each tree (store in root). Link root of smaller tree to larger. Break a tie arbitrarily.

make-set(x): $\{x.p \leftarrow x;\ x.s \leftarrow 1\}$

link(x, y):

if $x.s < y.s$ **then**$\{x.p \leftarrow y;\ y.s \leftarrow y.s + x.s\}$

else $\{y.p \leftarrow x;\ x.s \leftarrow x.s + y.s\}$

Linking by rank: Maintain an integer *rank* for each root, initially 0. Link root of smaller rank to root of larger rank. If tie, increase rank of new root by 1.

make-set(x): $\{x.p \leftarrow$ x; $x.r \leftarrow 0\}$

link(x, y): {**if** $x.r = y.r$ **then** $y.r \leftarrow y.r + 1$;

if $x.r < y.r$ **then** $x.p \leftarrow y$ **else** $y.p \leftarrow x$}

$x.r$ = height of x

Linking by index: Assume nodes are totally ordered. Link smaller root to larger.

make-set(x): {$x.p \leftarrow$ x}

link(x, y): {**if** $x < y$ **then** $x.p \leftarrow y$ **else** $y.p \leftarrow x$}

Linking by size and linking by rank have similar efficiency. Linking by rank needs fewer bits ($\lg\lg n$ for rank vs. $\lg n$ for size) and less time.

For any x, $x.r < x.p.r$

#(nodes of rank $\geq k$) $\leq n/2^k$

$\rightarrow$ $x.r \leq \lg n$, *find*(x) takes $O(\lg n)$ time

Path compression

During each *find*, make the root the parent of each node on the *find* path:

find(x): **if** $x.p.p \neq x.p$ **then** $x.p \leftarrow$ *find*($x.p$);
return $x.p$

Compression takes two passes over the *find* path

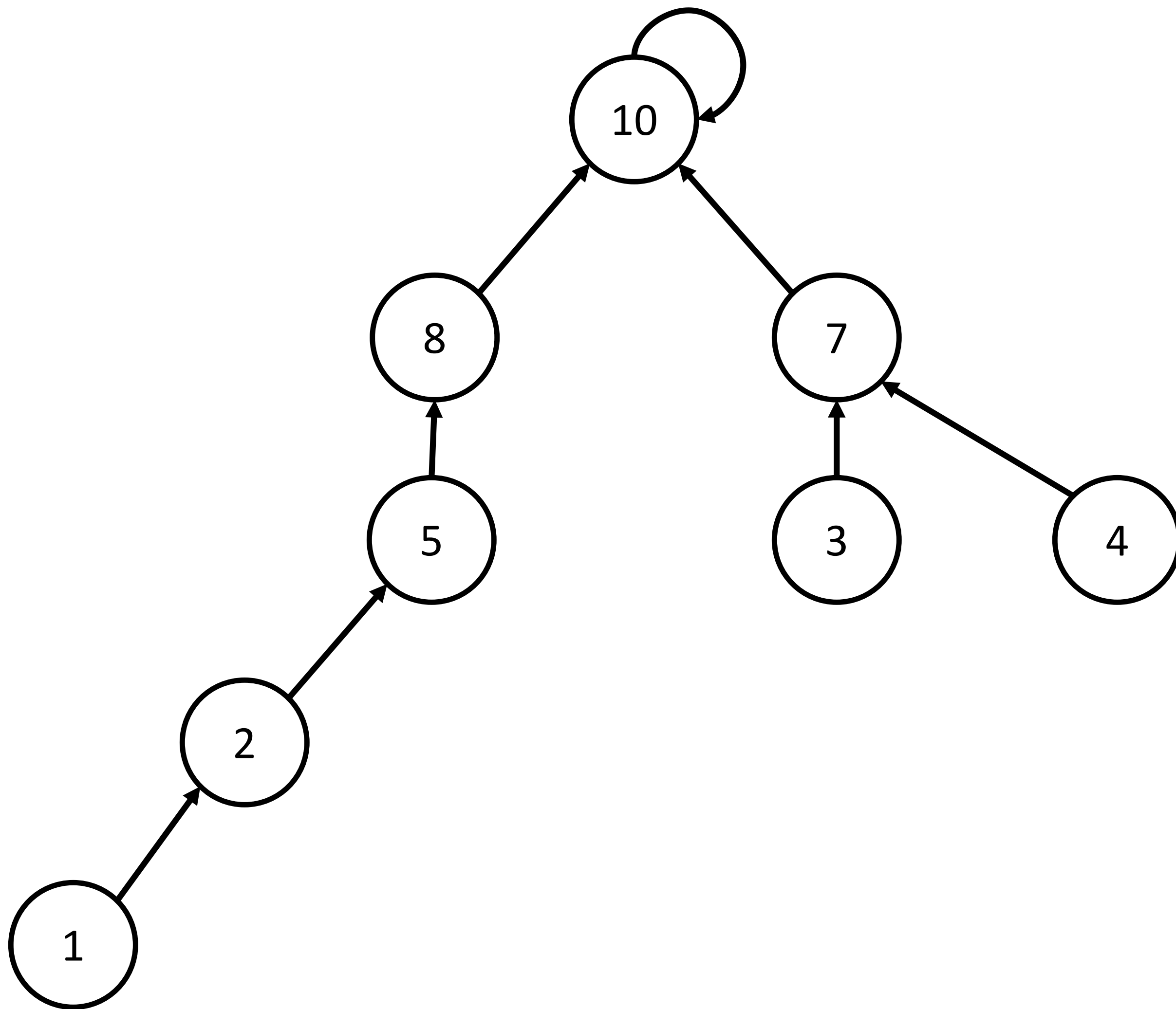
Path splitting

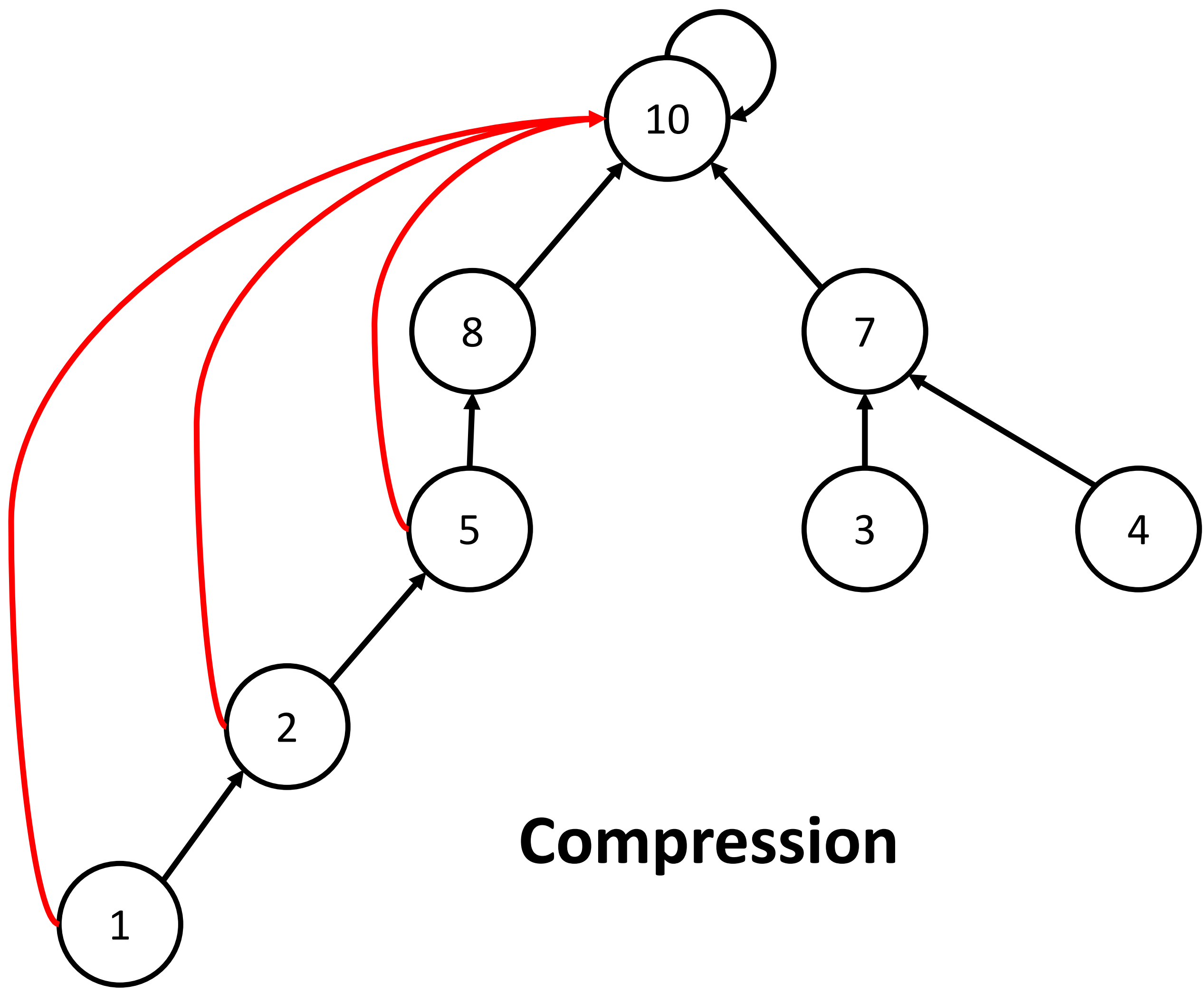
During each find, make each node point to its grandparent:

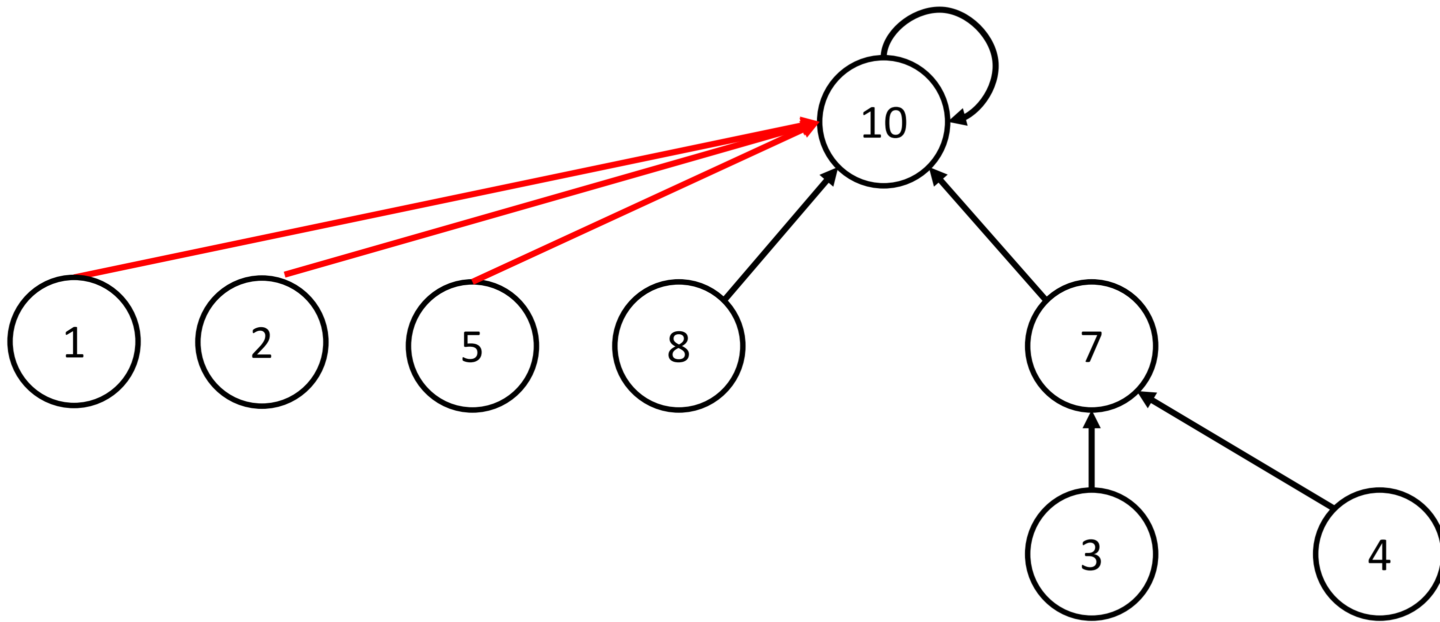
find(x): {$u \leftarrow x$;
  while $u.p.p \neq u.p$ **do**
    {$v \leftarrow u.p$; $u.p \leftarrow u.p.p$; $u \leftarrow v$};
  return $u.p$}

Only one pass over *find* path

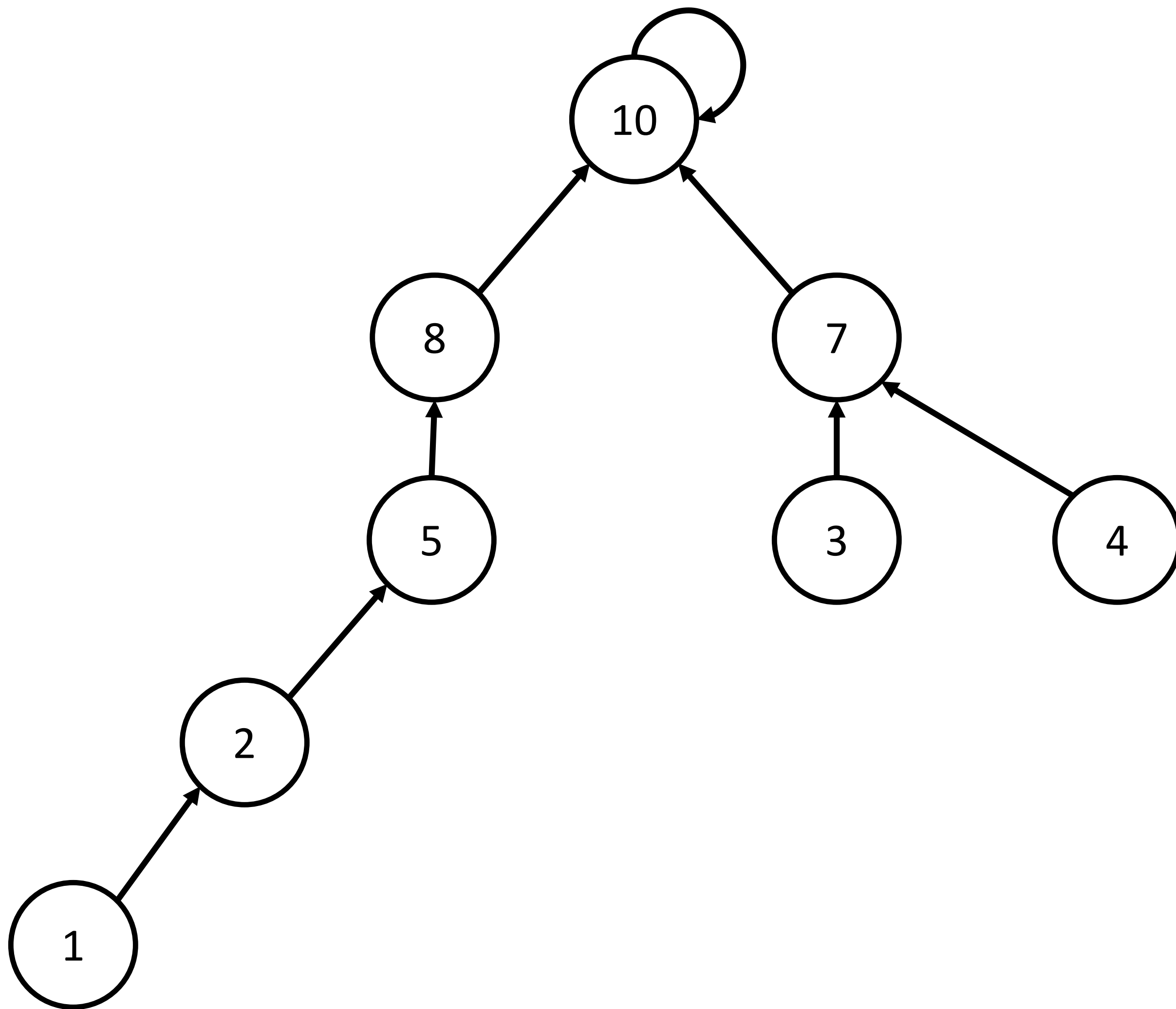
Alternative: make every other node point to its parent (**path halving**)

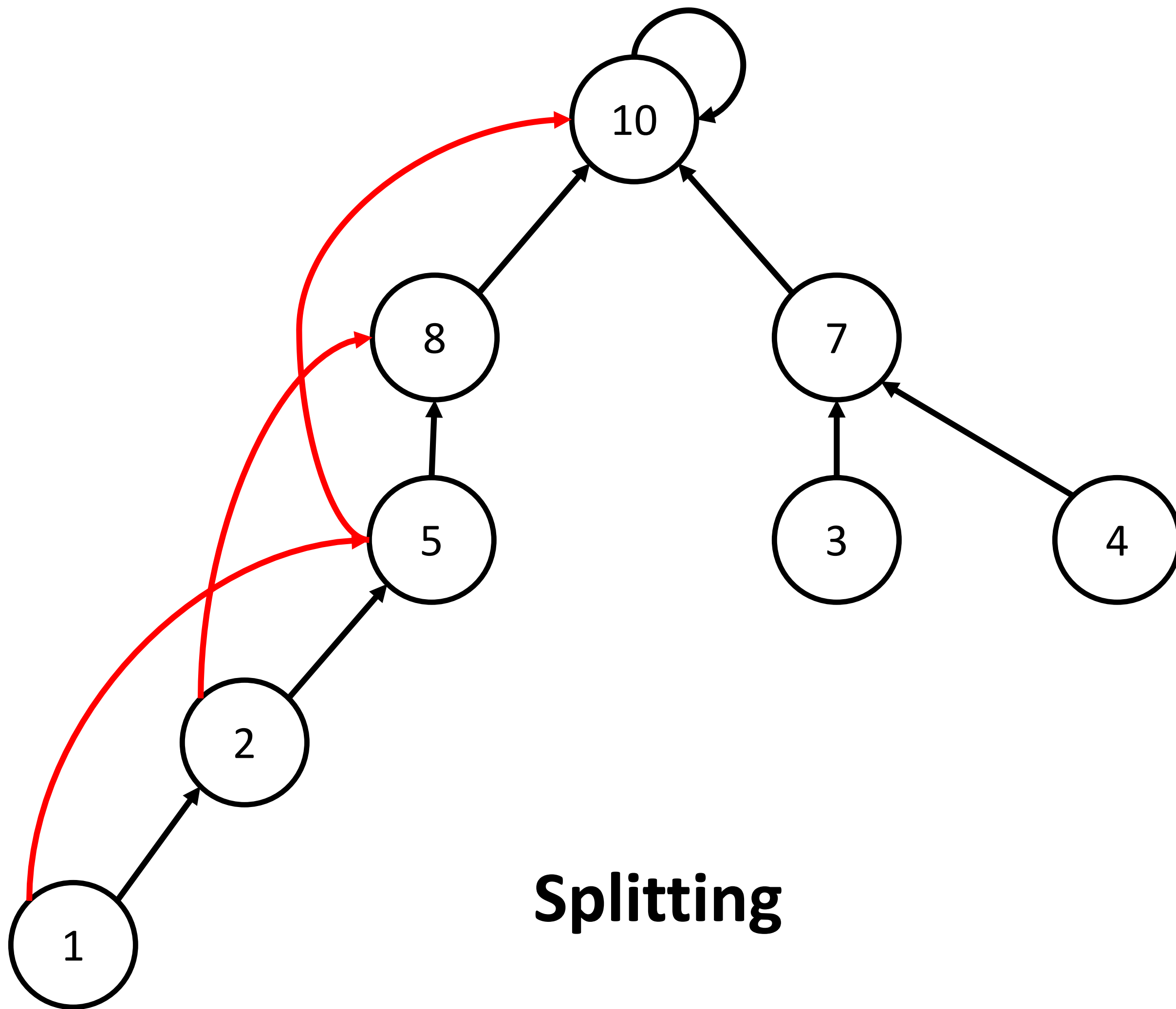

10
8
7
5
3
4
2
1

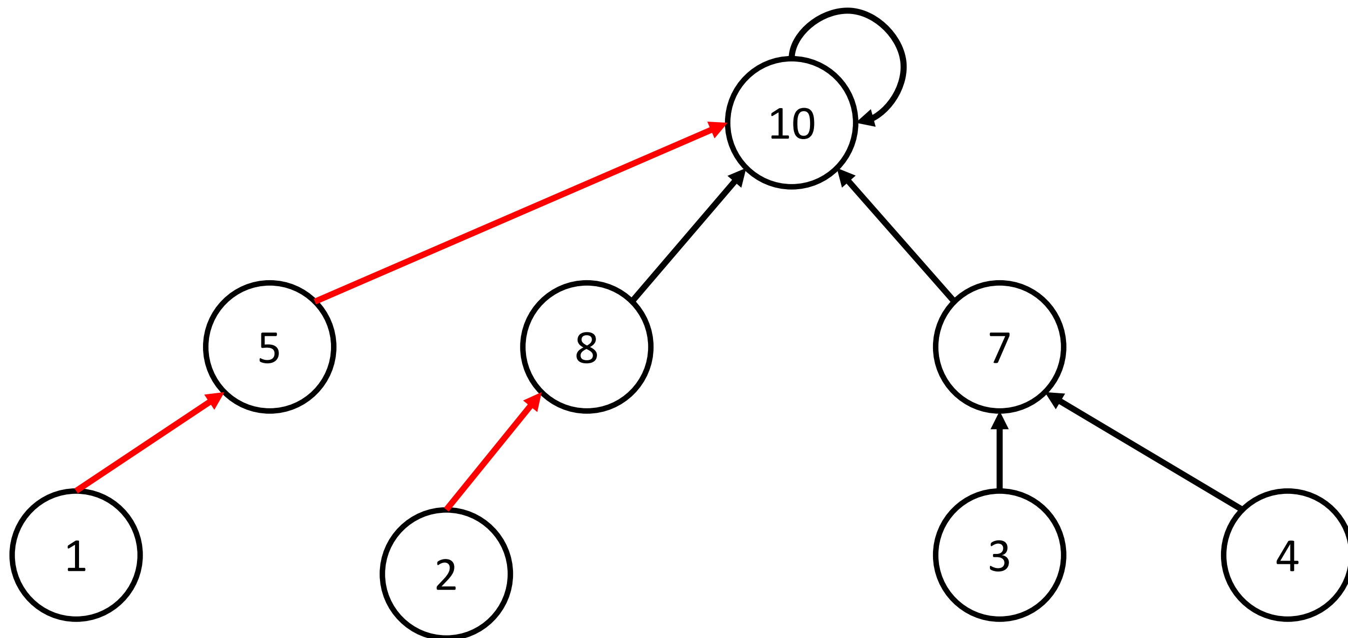

10
8
7
5
3
4
2
1
Compression



Compression


10
8
7
5
3
4
2
1


10
8
7
5
3
4
2
1
Splitting



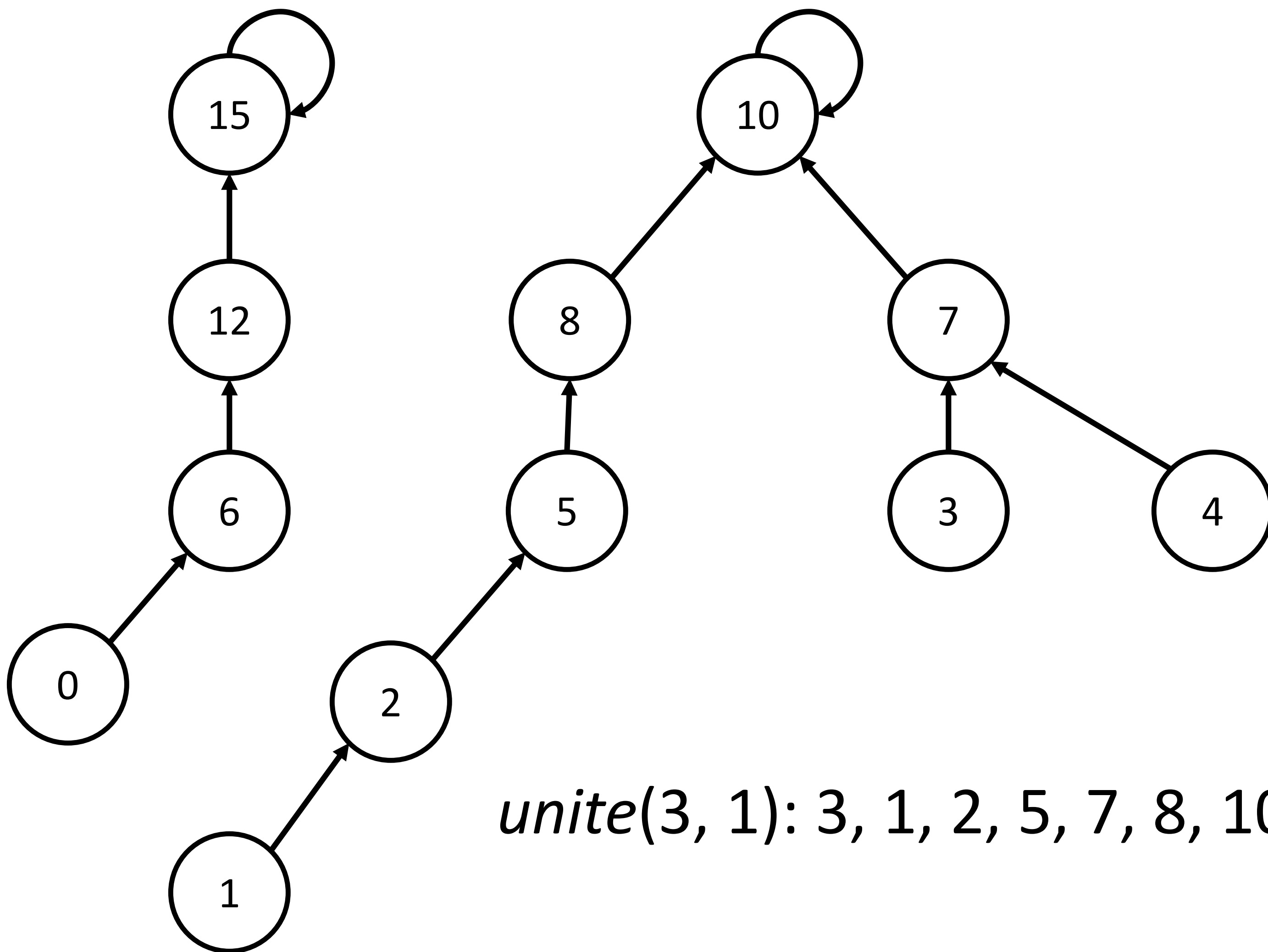
Splitting

Eager Linking

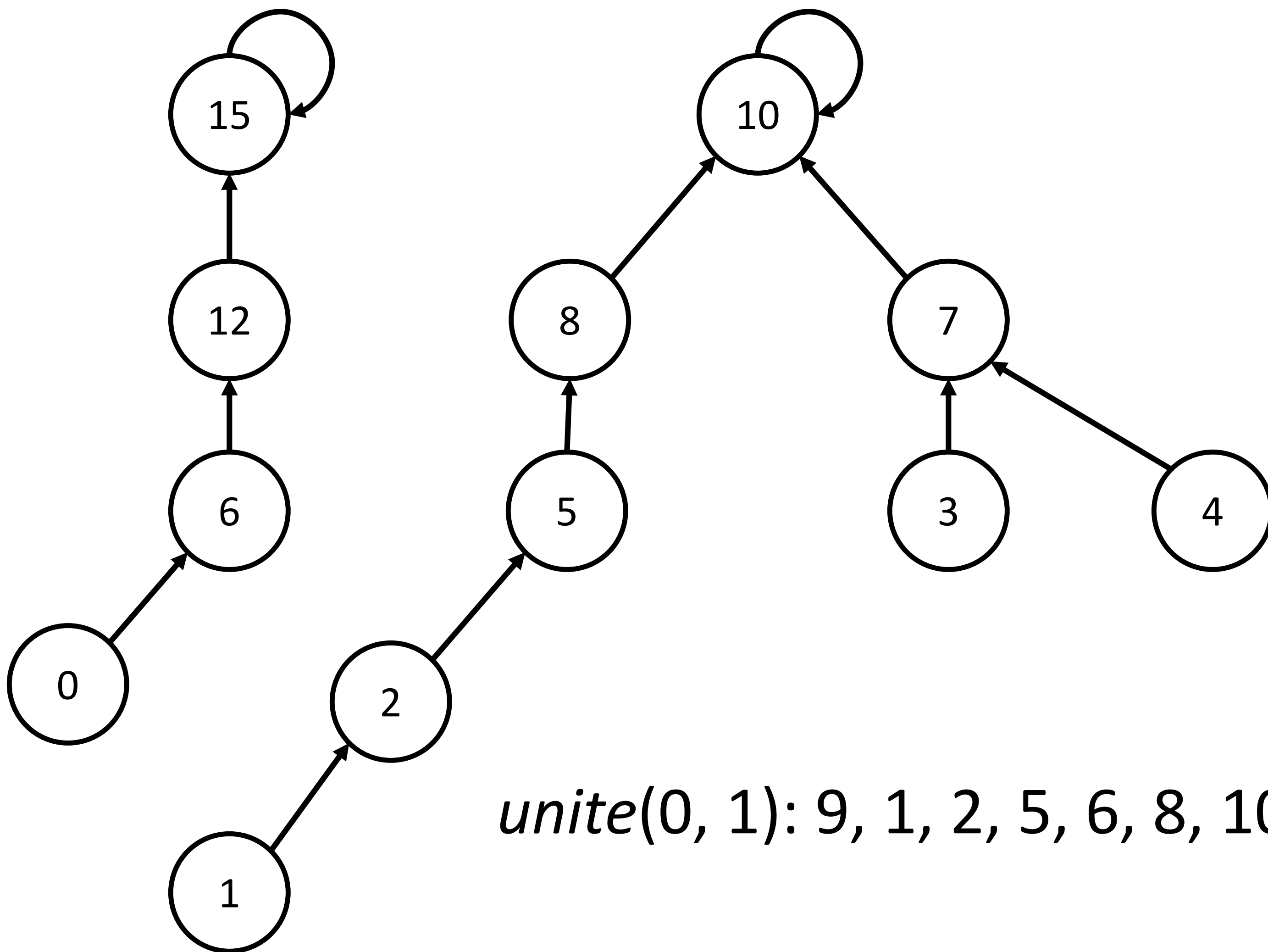
Do both *finds* during a *unite* concurrently, stop when one reaches a root (or both reach nearest common ancestor). Make the root a child of the current node on the other path.

How to interleave finds?

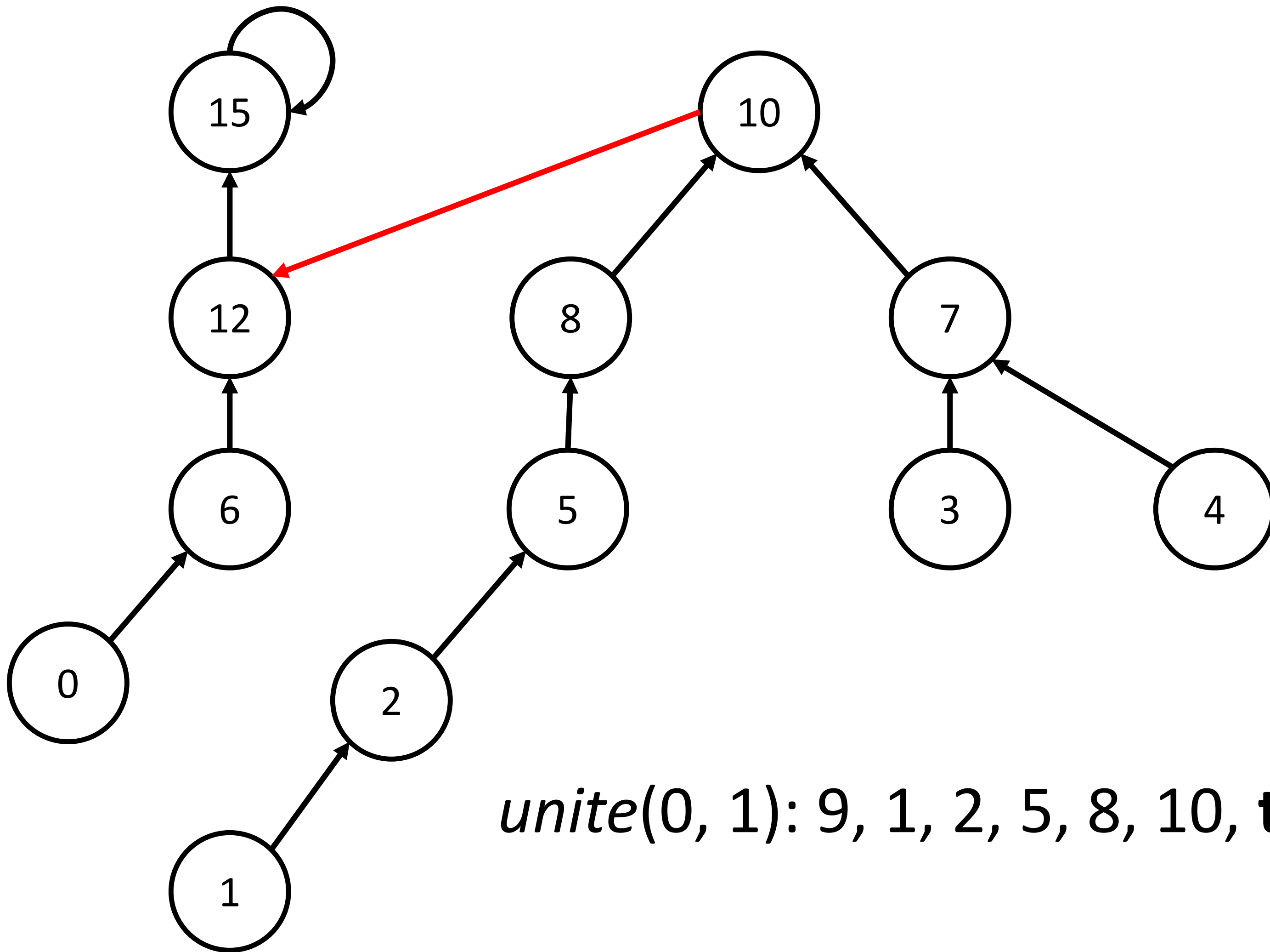
If nodes are totally ordered, always step to smaller node.



unite(3, 1): 3, 1, 2, 5, 7, 8, 10, **false**



unite(0, 1): 9, 1, 2, 5, 6, 8, 10, **true**



unite(0, 1): 9, 1, 2, 5, 8, 10, **true**

Rem’s algorithm: splicing

Compression or splitting with eager linking?

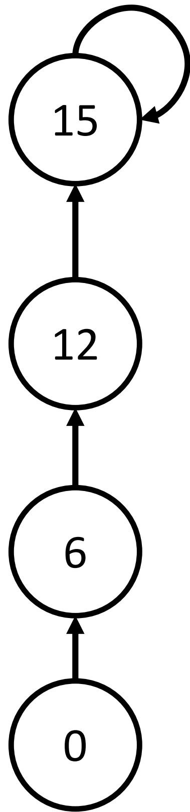
Yes

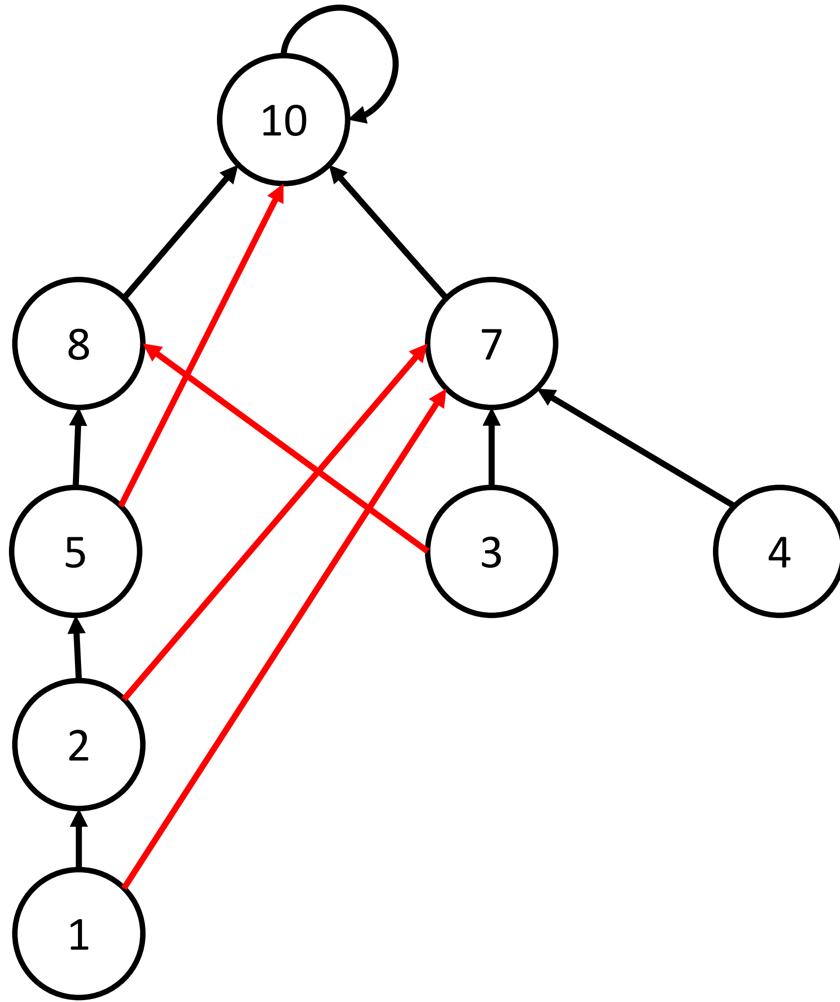
Or **splice***:* when taking a step from a node, change its parent to the current node on the *other* path

Can do eager linking (& splicing) by rank

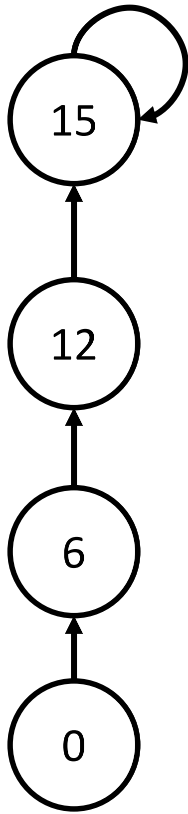
unite by index with splicing

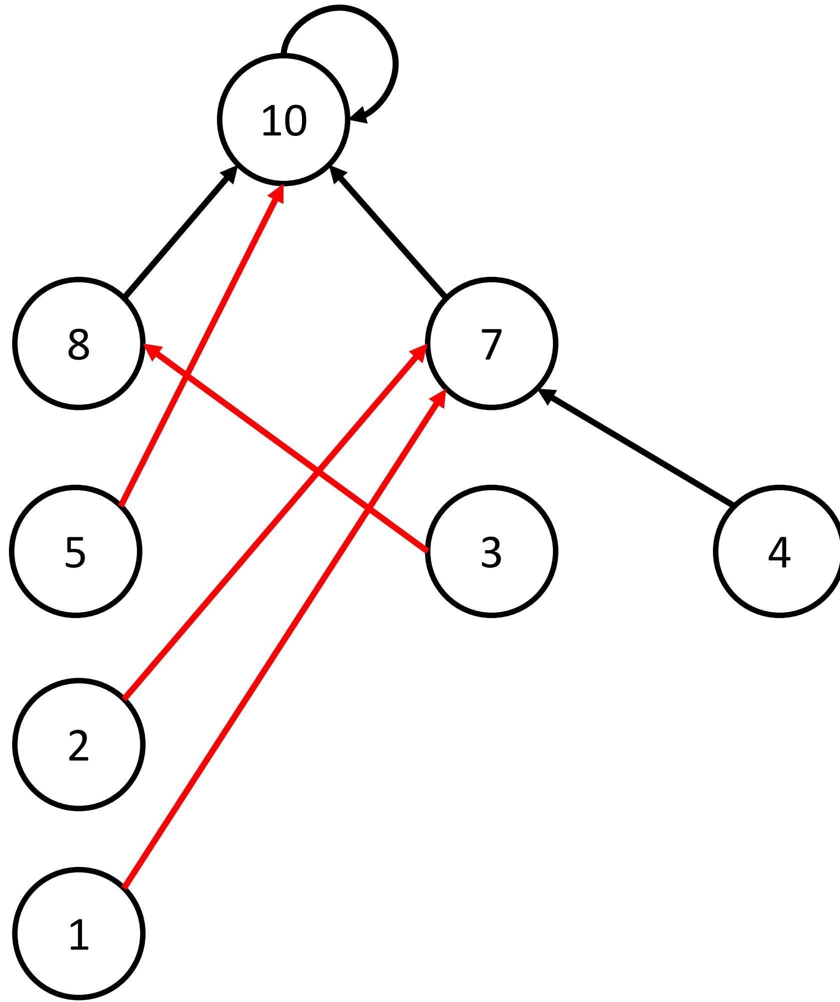
```
unite(x, y):
  {v ← x; w ← y;
   while v.p ≠ w.p do
      {if v.p > w.p then v ↔ w;
       u ← v.p;
       v.p ← w.p;
       if v = u then return true
       else v ← u};
   return false}
```



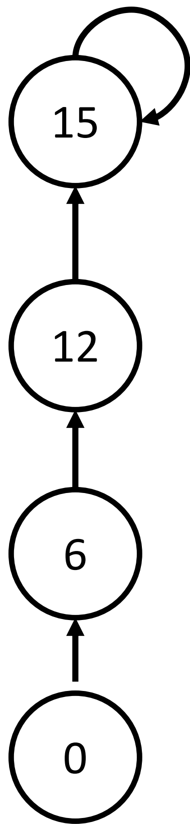


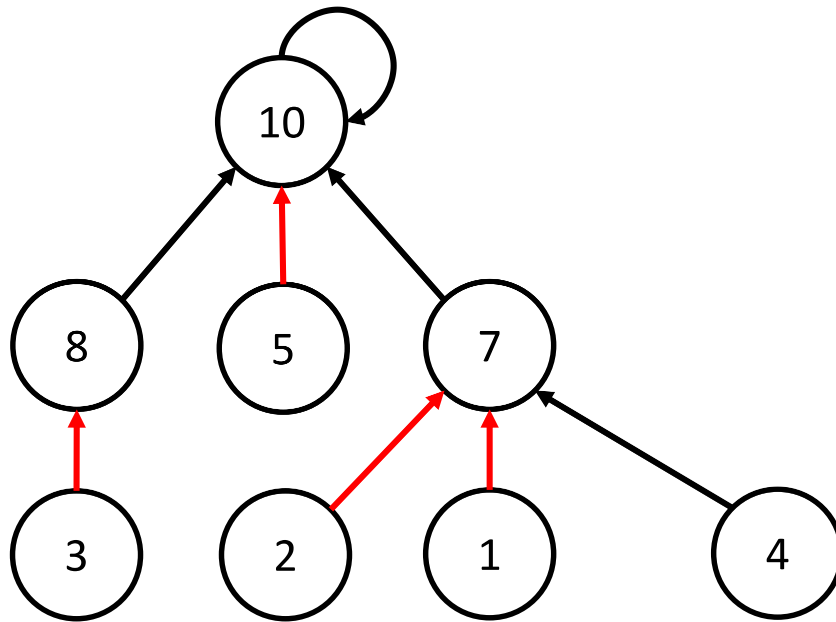
unite(3, 1): 3, 1, 2, 5, 7, 8, 10, **false**



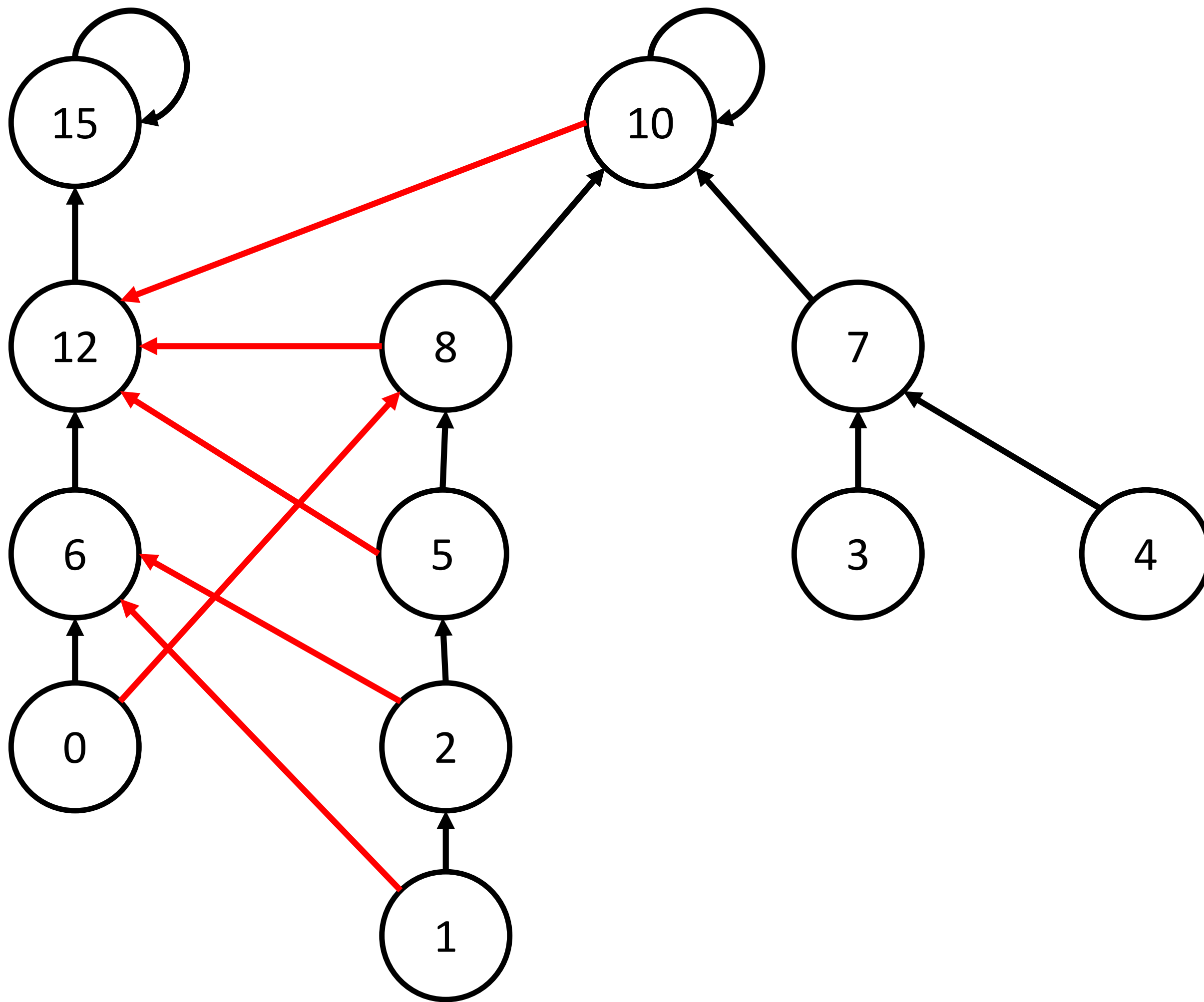


unite(3, 1): 3, 1, 2, 5, 7, 8, 10, **false**

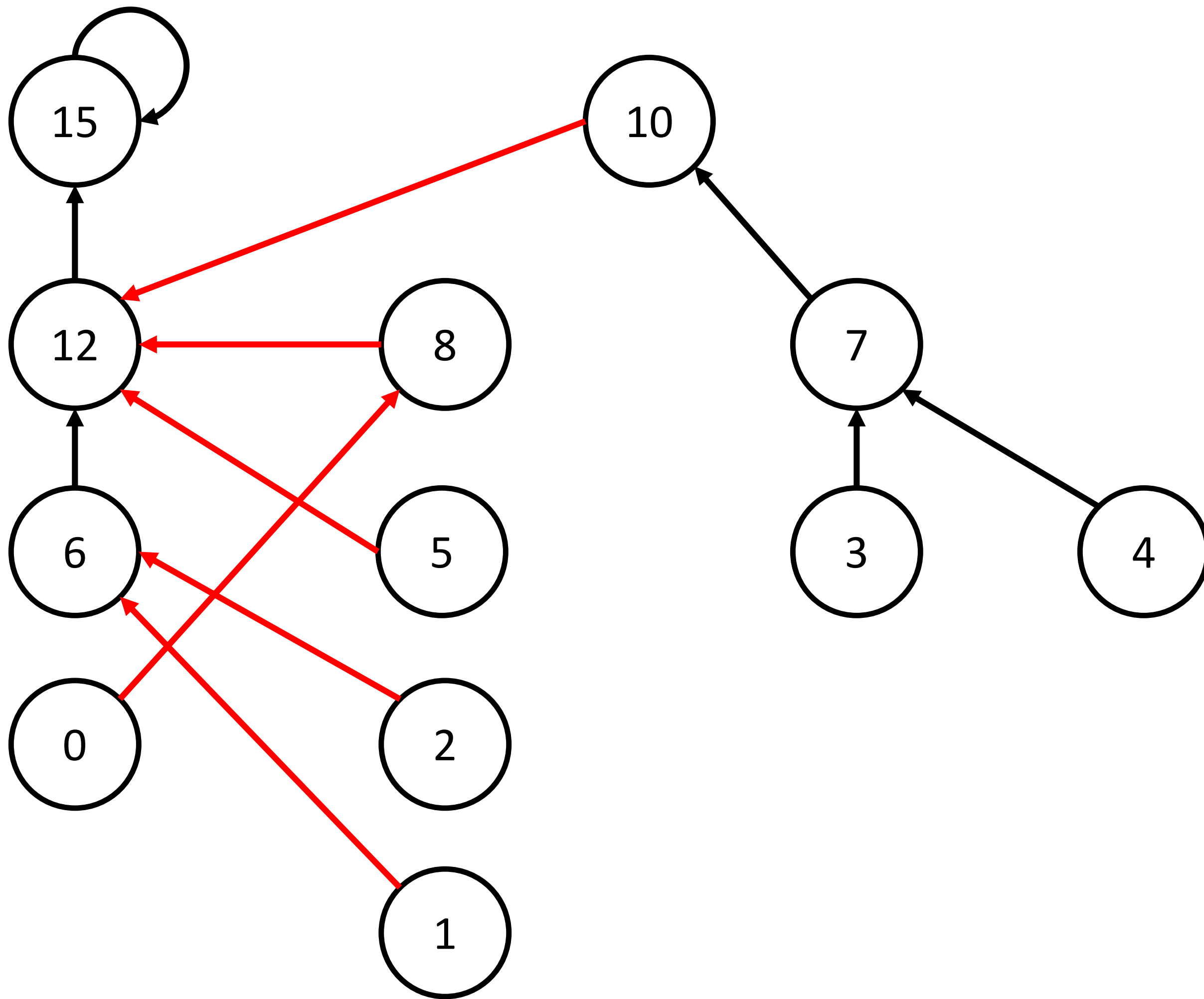




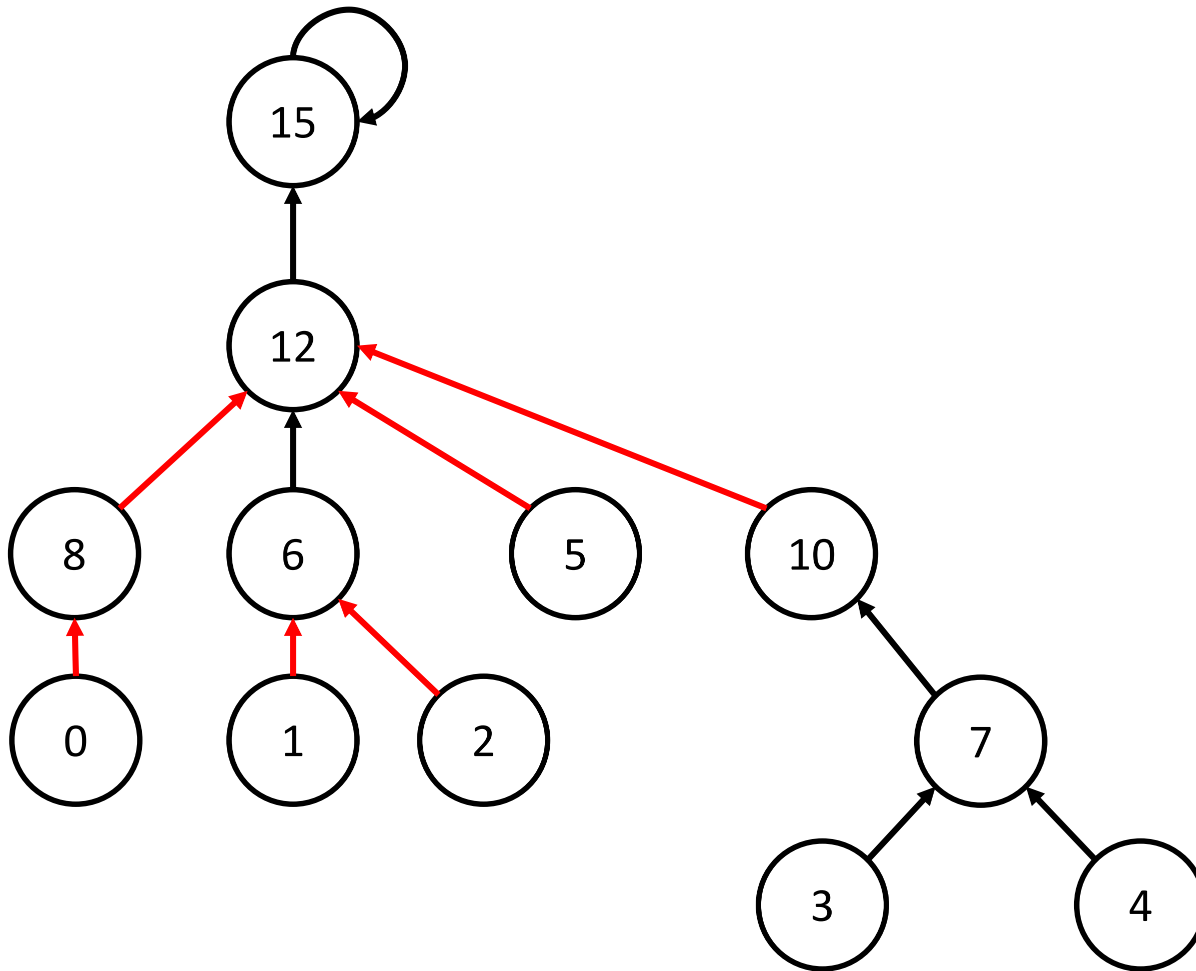
unite(3, 1): 3, 1, 2, 5, 7, 8, 10, **false**



unite(0, 1): 0, 1, 2, 5, 6, 8, 10, **true**



unite(0, 1): 0, 1, 2, 5, 6, 8, 10, **true**



unite(0, 1): 0, 1, 2, 5, 6, 8, 10, **true**

Path compression with linking by size (or rank)

History of bounds (amortized time per *find*)

1971 O(1) (false)

1972 O(lglg*n*) M. Fisher

1973 O(lg**n*) Hopcroft & Ullman

1975 $\Theta(\alpha(n, m/n))$ Tarjan

later Ω(lglg*n*) (false)

2005 top-down analysis Seidel & Sharir

Ackermann's function (Péter & Robinson)

$A(0, j) = j + 1$

$A(k,0) = A(k - 1, 1)$

$A(k, j) = A(k - 1, A(k, j - 1))$ if $k > 0, j > 0$

$A(1, j) = j + 2$, $A(2, j) = 2j + 3$, $A(3, j) > 2^j$, $A(4, j) >$ tower of j 2's, $A(4, 2)$ has 19,729 decimal digits

$A(k, j)$ is strictly increasing in both arguments

$\alpha(r, d) = \min\{k > 0 \mid A(k, \lceil d \rceil) > r\}$

Beautiful Theory… but Practice?

Experiments: Patwary, Blair, and Manne (2010)

Implementation of Kruskal’s algorithm: only *unites*

Two classes of random graphs, and “real-world” graphs

Results

Linking by index with splicing or splitting generally best, though by small amounts.

???

Why doesn't linking by rank (or size) help in practice?

Randomized linking

Number the elements from 1 to *n* uniformly at random, then do linking by index, or eager linking by index.

Hypothesis: This is what the experiments did, in effect, on at least the random instances.

May come for free, e. g. if the node names are hashed.

Equivalent to linking probabilistically by size, *not* by flipping a fair coin (50-50 linking).

(Cannot do either eagerly.)

Results

Randomized linking or randomized eager linking with compression, splicing, or halving, and randomized eager linking with splicing, take expected amortized $O(\alpha(n, m/n))$ time per *find*.

Tight by Fredman-Saks lower bound.

Not true for 50-50 linking. (We think we have a proof.)

Key Idea

How to define rank?

Obvious idea: rank = height in tree built by links with no compaction

Doesn’t seem to handle splicing (not monotonic in node numbers)

Harder to analyze than our solution

Want:

(i) ranks monotonic in node numbers

(ii) #(nodes of rank ≥ k) ≤ $n/2^k$

Solution: give nodes 1 through $n/2$ rank 0,

$n/2 + 1$ through $3n/4$ rank 1,

$3n/4 + 1$ through $7n/8$ rank 2...

(i) True → $x.r \leq x.p.r$ (vs. $x.r < x.p.r$)

(ii) True

(iii) For any node, at least half of higher nodes have strictly higher rank

Idea of Analysis

Apply an existing analysis of linking by rank to handle nodes x on find paths with $x.r < x.p.r$

Bound # nodes x on find paths with $x.r = x.p.r$

Lemma: For any x, expected # proper ancestors of x of same rank (in tree built by links with no compaction) ≤ 2

Proof of lemma: Given a sequence σ of *unites* and an element x, reorder σ into $\sigma(x)$: next *unite* adds an element to the set containing x, breaking ties in order by σ.

With linking by index, σ-ancestors of x are a subset of $\sigma(x)$-ancestors of x;

With early linking by index, ancestor sets are the same: use recursive characterization of $\sigma(x)$

Assume $\sigma = \pi \lozenge \rho$, $unite(v, w)$

where π builds S containing x, ρ builds R,

v in S, w in R, $\lozenge$ is arbitrary interleaving

Then $\sigma(x) = \pi(x)$, $unite(v, w)$, $\rho(w)$

(No unite in ρ has a node in S other than v as an input)

Let u be the root of the current tree containing x.

Next node u' added to tree has at least ½ chance of having rank greater than $u.x$, given that $u' > u$, by (iii)

→ expected #proper ancestors of same rank as x

$\leq 1/2 + 1/2 + 3/8 + 1/4 + \ldots \leq 2$

→ expected #nodes on find paths with parent of same rank $= O(n)$

Splicing

New parent of a node *not* necessarily an ancestor

Define *zero-level potential* of x = # proper ancestors of same rank: expected total over all nodes = O(n)

For splitting, need a window of 3 nodes to analyze *find* path. For splicing, window is 6

You too can learn/teach the α bound!

We define *x.a*, *x.b*, and x.c, the *level, index*, and *count of x*, as follows:

$$x.a = \max\{0, \max\{k \mid A(k, x.r) \leq x.p.r\}\}$$

$$x.b = \max\{0, \max\{j \mid A(x.a + 1, j) \leq x.p.r\}\}$$

$$x.c = x.a \times x.r + x.b$$

$$A(\alpha(n, 0), 0) > n \rightarrow x.a < \alpha(n, 0)$$

$$A(x.a + 1, x.r) > x.p.r \rightarrow x.b < x.r$$

$$\Sigma x.c = O(n\alpha(n, 0))$$

Lemma: If x is followed by y on a find path with $0 < x.r < x.p.r$ and $y.r < y.p.r$ and $x.a = y.a$, then compression of the find path increases $x.c$.

Proof: $A(x.a + 1, x.b + 1) = A(x.a, A(x.a + 1, x.b))$

$$\leq A(x.a, x.p.r) = A(y.a, x.p.r)$$

$$\leq A(y.a, y.r) \leq y.p.r$$

Hence the increase in $x.p.r$ to at least $y.p.r$ increases $x.a$ and hence $x.c$, or does not change $x.a$ but increases $x.b$, again increasing $x.c$

We are still learning…

Thanks!