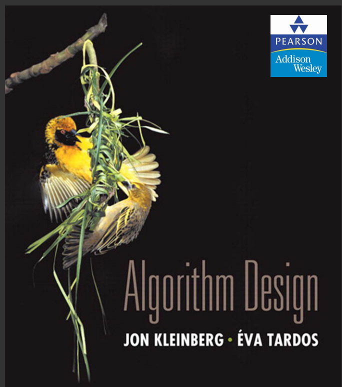


UNION–FIND

- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 1/15/20 6:59 AM

Disjoint-sets data type

Goal. Support three operations on a collection of disjoint sets.

- MAKE-SET(x): create a new set containing only element x.
- FIND(x): return a canonical element in the set containing x.
- UNION(x, y): replace the sets containing x and y with their union.

Performance parameters.

- m = number of calls to MAKE-SET, FIND, and UNION.
- n = number of elements = number of calls to MAKE-SET.

Dynamic connectivity. Given an initially empty graph G, support three operations. ← disjoint sets = connected components

- ADD-NODE(u): add node u. ← 1 MAKE-SET operation
- ADD-EDGE(u, v): add an edge between nodes u and v. ← 1 UNION operation
- IS-CONNECTED(u, v): is there a path between u and v? ← 2 FIND operations

Disjoint-sets data type: applications

Original motivation. Compiling EQUIVALENCE, DIMENSION, and COMMON statements in Fortran.

An Improved Equivalence Algorithm

BERNARD A. GALLER AND MICHAEL J. FISHER
University of Michigan, Ann Arbor, Michigan

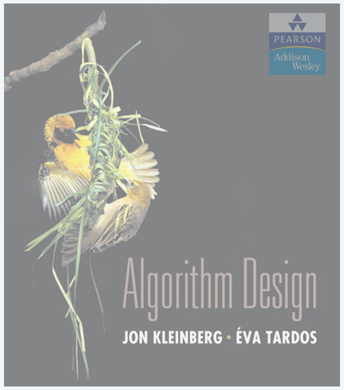
An algorithm for assigning storage on the basis of EQUIVALENCE, DIMENSION and COMMON declarations is presented. The algorithm is based on a tree structure, and has reduced computation time by 40 percent over a previously published algorithm by identifying all equivalence classes with one scan of the EQUIVALENCE declarations. The method is applicable in any problem in which it is necessary to identify equivalence classes, given the element pairs defining the equivalence relation.

Note. This 1964 paper also introduced key data structure for problem.

Disjoint-sets data type: applications

Applications.

- Percolation.
- Kruskal's algorithm.
- Connected components.
- Computing LCAs in trees.
- Computing dominators in digraphs.
- Equivalence of finite state automata.
- Checking flow graphs for reducibility.
- Hoshen–Kopelman algorithm in physics.
- Hinley–Milner polymorphic type inference.
- Morphological attribute openings and closings.
- Matlab's BW-LABEL function for image processing.
- Compiling EQUIVALENCE, DIMENSION and COMMON statements in Fortran.
- …



UNION–FIND

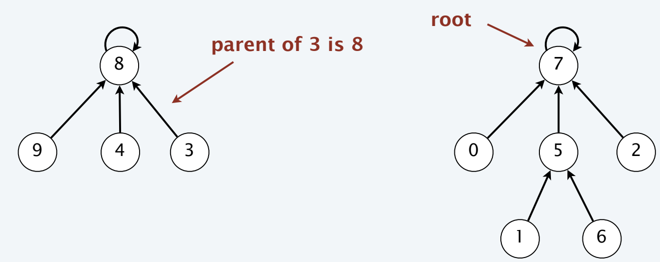
- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

Disjoint-sets data structure

Parent-link representation. Represent each set as a tree of elements.

- Each element has an explicit parent pointer in the tree.
- The root serves as the canonical element (and points to itself).
- FIND(x): find the root of the tree containing x.
- UNION(x, y): merge trees containing x and y (by making one root point to the other root).

UNION(3, 5)

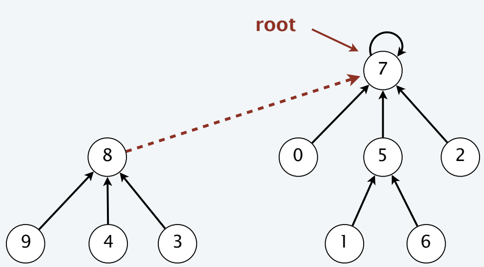


Disjoint-sets data structure

Parent-link representation. Represent each set as a tree of elements.

- Each element has an explicit parent pointer in the tree.
- The root serves as the canonical element (and points to itself).
- FIND(x): find the root of the tree containing x.
- UNION(x, y): merge trees containing x and y (by making one root point to the other root).

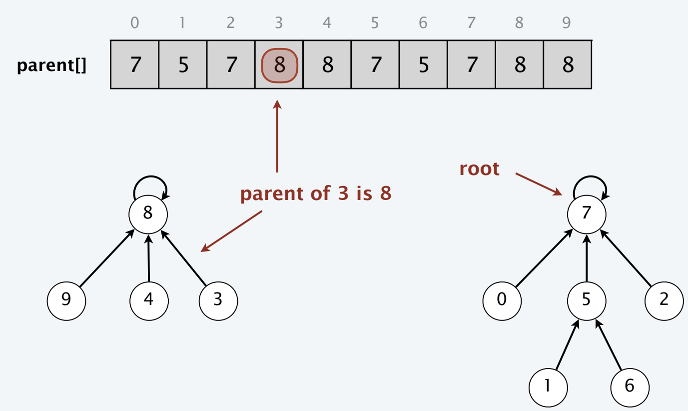
UNION(3, 5)



Disjoint-sets data structure

Array representation. Represent each set as a tree of elements.

- Allocate an array $parent[]$ of length n. ← must know number of elements n a priori
- $parent[i] = j$ means parent of element i is element j.

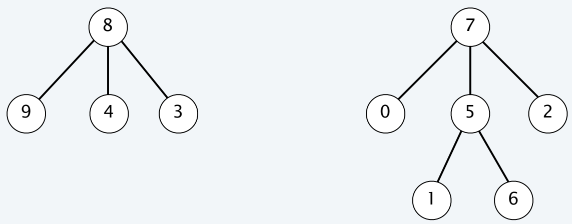


Note. For brevity, we suppress arrows and self-loops in figures.

Naïve linking

Naïve linking. Link root of first tree to root of second tree.

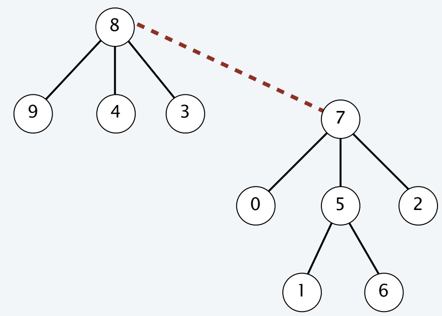
UNION(5, 3)



Naïve linking

Naïve linking. Link root of first tree to root of second tree.

UNION(5, 3)



Naïve linking

Naïve linking. Link root of first tree to root of second tree.

MAKE-SET(x)

$parent[x] \leftarrow x$.

FIND(x)

WHILE $(x \neq parent[x])$

$\quad x \leftarrow parent[x]$.

RETURN x.

UNION(x, y)

$r \leftarrow$ FIND(x).

$s \leftarrow$ FIND(y).

$parent[r] \leftarrow s$.

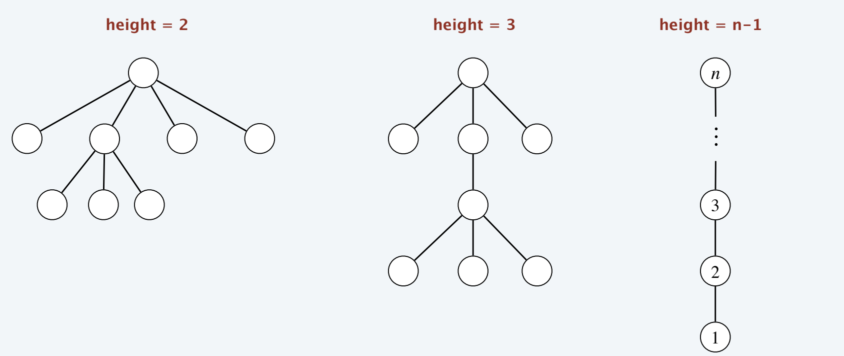
Naïve linking: analysis

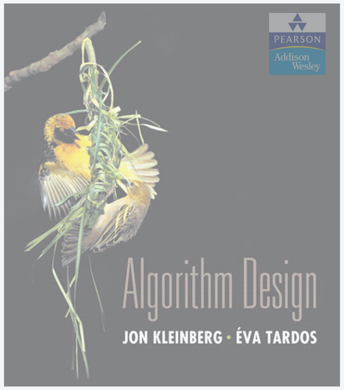
Theorem. Using naïve linking, a UNION or FIND operation can take $\Theta(n)$ time in the worst case, where n is the number of elements.

max number of links on any path from root to leaf node

Pf.

- In the worst case, FIND takes time proportional to the height of the tree.
- Height of the tree is $n - 1$ after the sequence of union operations: UNION$(1, 2)$, UNION$(2, 3)$, …, UNION$(n - 1, n)$.



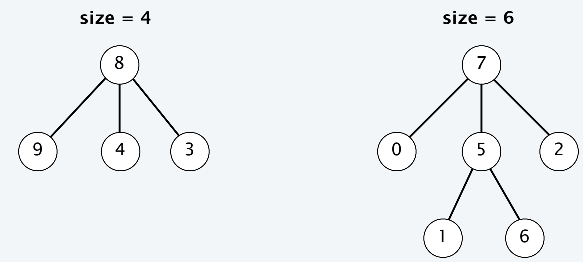


UNION–FIND

Link-by-size

Link-by-size. Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

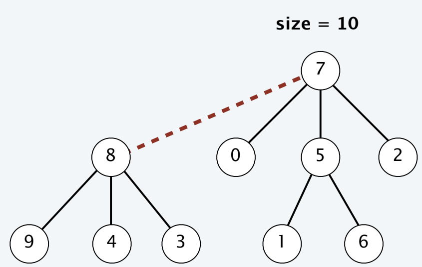
UNION(5, 3)



Link-by-size

Link-by-size. Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

UNION(5, 3)



Link-by-size

Link-by-size. Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

```
MAKE-SET(x)
parent[x] ← x.
size[x] ← 1.
```

```
FIND(x)
WHILE (x ≠ parent[x])
   x ← parent[x].
RETURN x.
```

```
UNION(x, y)
r ← FIND(x).
s ← FIND(y).
IF (r = s) RETURN.
ELSE IF (size[r] > size[s])
   parent[s] ← r.
   size[r] ← size[r] + size[s].
ELSE
   parent[r] ← s.
   size[s] ← size[r] + size[s].
```

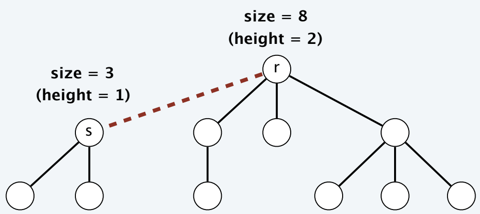
link-by-size

Link-by-size: analysis

Property. Using link-by-size, for every root node r: $size[r] \geq 2^{height(r)}$.
Pf. [by induction on number of links]

- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first i links.
- Tree rooted at r changes only when a smaller (or equal) size tree rooted at s is linked into r.
- Case 1. [$height(r) > height(s)$]

$$\begin{aligned} size'[r] &> size[r] \\ &\geq 2^{height(r)} \quad \longleftarrow \text{inductive hypothesis} \\ &= 2^{height'(r)}. \end{aligned}$$

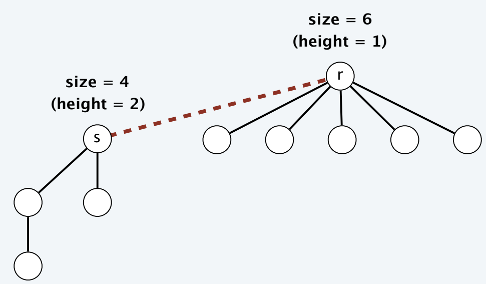


Link-by-size: analysis

Property. Using link-by-size, for every root node r: $size[r] \geq 2^{height(r)}$.
Pf. [by induction on number of links]

- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first i links.
- Tree rooted at r changes only when a smaller (or equal) size tree rooted at s is linked into r.
- Case 2. [$height(r) \leq height(s)$]

$$\begin{aligned} size'[r] &= size[r] + size[s] \\ &\geq 2\ size[s] \quad \longleftarrow \text{link-by-size} \\ &\geq 2 \cdot 2^{height(s)} \quad \longleftarrow \text{inductive hypothesis} \\ &= 2^{height(s)+1} \\ &= 2^{height'(r)}. \quad \blacksquare \end{aligned}$$



Link-by-size: analysis

Theorem. Using link-by-size, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where n is the number of elements.
Pf.

- The running time of each operation is bounded by the tree height.
- By the previous property, the height is $\leq \lfloor \lg n \rfloor$. ▪

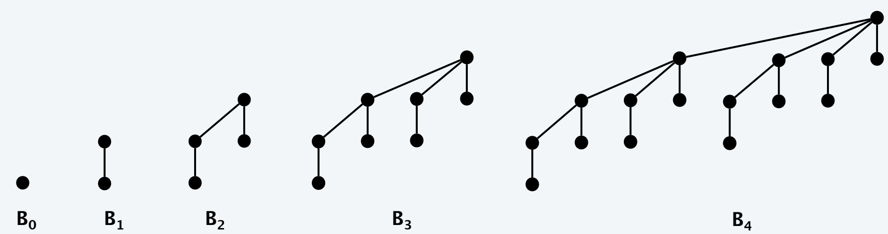
$\lg n = \log_2 n$

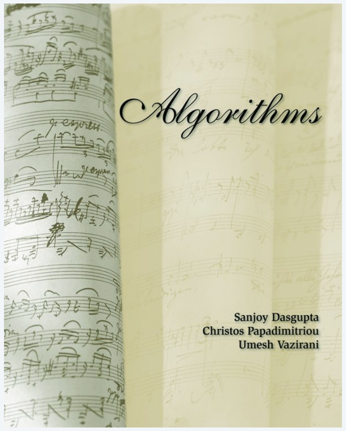
Note. The UNION operation takes $O(1)$ time except for its two calls to FIND.

A tight upper bound

Theorem. Using link-by-size, a tree with n nodes can have height $= \lg n$.
Pf.

- Arrange $2^k - 1$ calls to UNION to form a binomial tree of order k.
- An order-k binomial tree has 2^k nodes and height k. ▪





SECTION 5.1.4

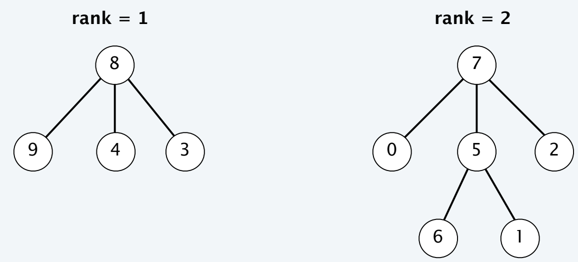
UNION–FIND

- naïve linking
- link-by-size
- **link-by-rank**
- path compression
- link-by-rank with path compression
- context

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.

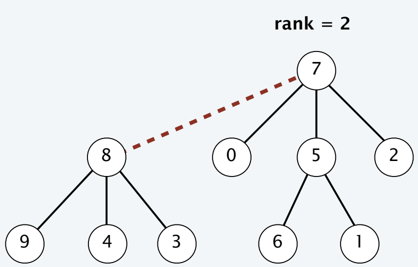
UNION(5, 3)



Note. For now, rank = height.

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.



Note. For now, rank = height.

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.

```
MAKE-SET(x)
parent[x] ← x.
rank[x] ← 0.
```

```
FIND(x)
WHILE (x ≠ parent[x])
   x ← parent[x].
RETURN x.
```

```
UNION(x, y)
r ← FIND(x).
s ← FIND(y).
IF (r = s) RETURN.
ELSE IF (rank[r] > rank[s])
   parent[s] ← r.
ELSE IF (rank[r] < rank[s])
   parent[r] ← s.
ELSE
   parent[r] ← s.
   rank[s] ← rank[s] + 1.
```

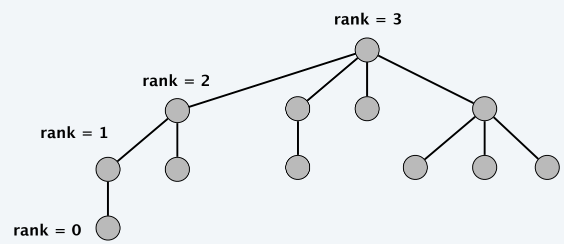
link-by-rank

Link-by-rank: properties

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.
Pf. A node of rank k is created only by linking two roots of rank $k - 1$. ▪

PROPERTY 2. If x is not a root node, then $rank[x]$ will never change again.
Pf. Rank changes only for roots; a nonroot never becomes a root. ▪

PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.
Pf. The parent can change only for a root, so before linking $parent[x] = x$. After x is linked-by-rank to new root r we have $rank[r] > rank[x]$. ▪

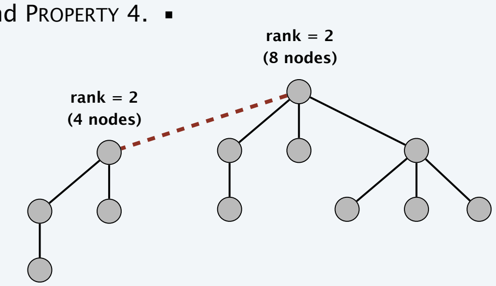


Link-by-rank: properties

PROPERTY 4. Any root node of rank k has $\geq 2^k$ nodes in its tree.
Pf. [by induction on k]

- Base case: true for $k = 0$.
- Inductive hypothesis: assume true for $k - 1$.
- A node of rank k is created only by linking two roots of rank $k - 1$.
- By inductive hypothesis, each subtree has $\geq 2^{k-1}$ nodes
  $\Rightarrow$ resulting tree has $\geq 2^k$ nodes. ▪

PROPERTY 5. The highest rank of a node is $\leq \lfloor \lg n \rfloor$.
Pf. Immediate from PROPERTY 1 and PROPERTY 4. ▪

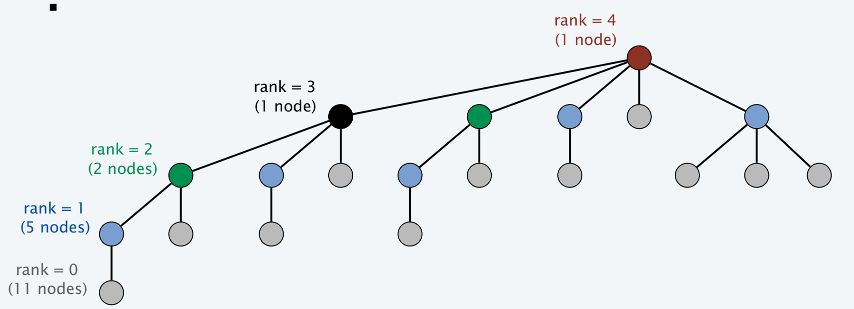


Link-by-rank: properties

PROPERTY 6. For any integer $k \geq 0$, there are $\leq n / 2^k$ nodes with rank k.
Pf.

- Any root node of rank k has $\geq 2^k$ descendants. [PROPERTY 4]
- Any nonroot node of rank k has $\geq 2^k$ descendants because:
  - it had this property just before it became a nonroot [PROPERTY 4]
  - its rank doesn't change once it became a nonroot [PROPERTY 2]
  - its set of descendants doesn't change once it became a nonroot
- Different nodes of rank k can't have common descendants. [PROPERTY 1]

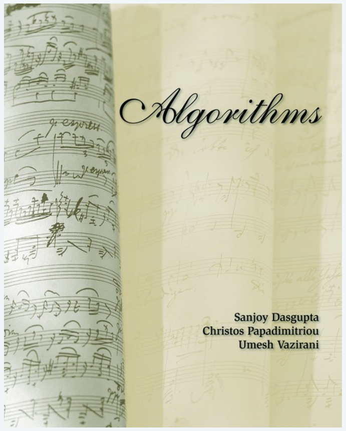
▪



Link-by-rank: analysis

Theorem. Using link-by-rank, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where n is the number of elements.
Pf.

- The running time of UNION and FIND is bounded by the tree height.
- By PROPERTY 5, the height is $\leq \lfloor \lg n \rfloor$. ▪



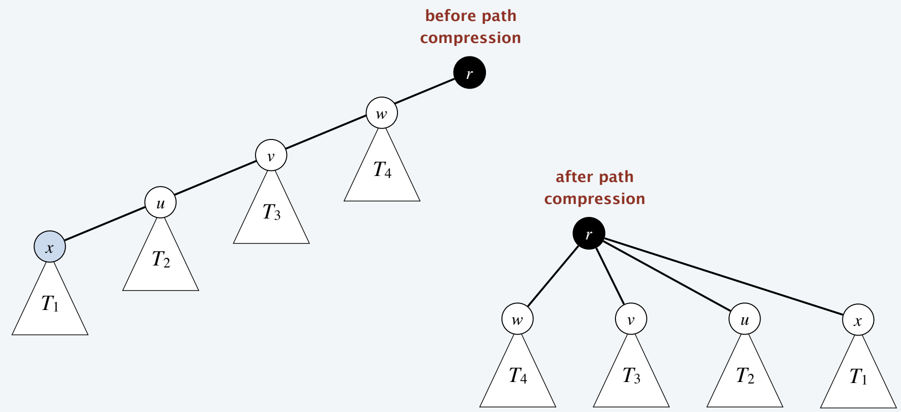
SECTION 5.1.4

UNION–FIND

- naïve linking
- link-by-size
- link-by-rank
- **path compression**
- link-by-rank with path compression
- context

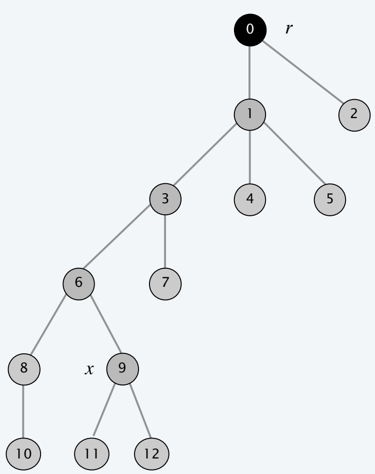
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



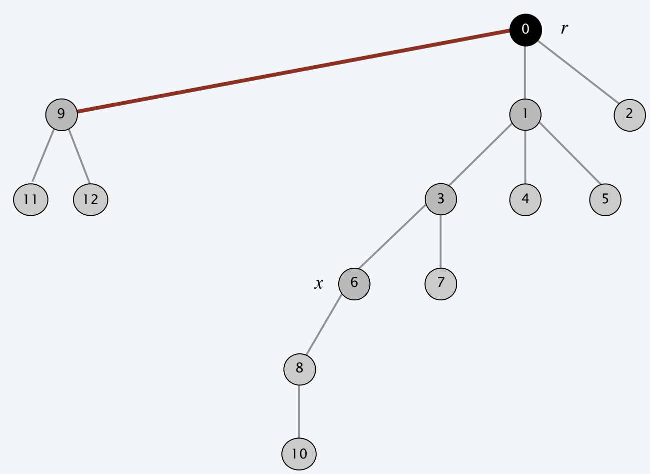
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



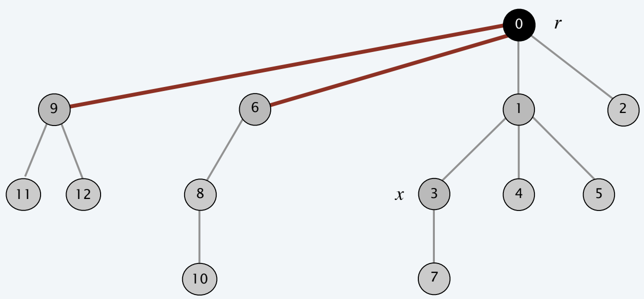
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



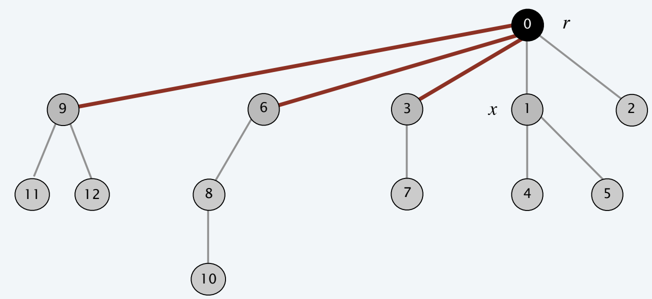
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



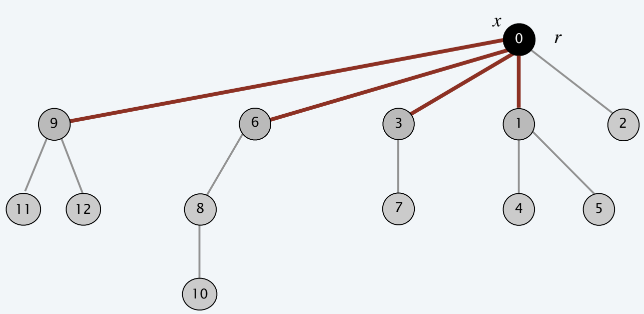
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.

FIND(x)

IF ($x \neq parent[x]$)

$parent[x] \leftarrow$ FIND($parent[x]$).

RETURN $parent[x]$.

this FIND implementation changes the tree structure (!)

Note. Path compression does not change the rank of a node; so $height(x) \leq rank[x]$ but they are not necessarily equal.

Path compression

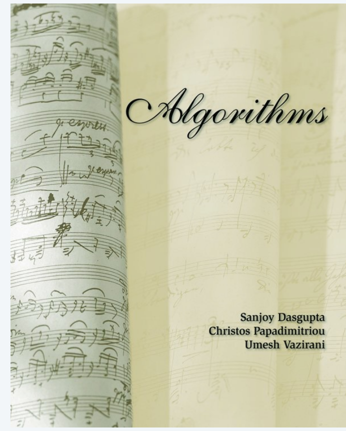
Fact. Path compression with naïve linking can require $\Omega(n)$ time to perform a single UNION or FIND operation, where n is the number of elements.

Pf. The height of the tree is $n-1$ after the sequence of union operations: UNION$(1, 2)$, UNION$(2, 3)$, …, UNION$(n-1, n)$. ▪

naïve linking: link root of first tree to root of second tree

Theorem. [Tarjan–van Leeuwen 1984] Starting from an empty data structure, path compression with naïve linking performs any intermixed sequence of $m \ge n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log n)$ time.

Pf. Nontrivial (but omitted).



SECTION 5.1.4

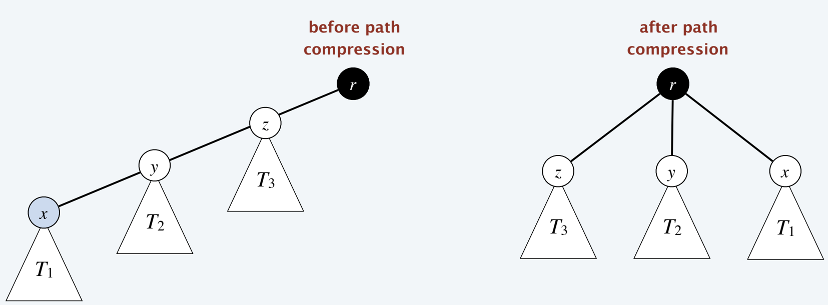
UNION–FIND

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- context

Link-by-rank with path compression: properties

PROPERTY. The tree roots, node ranks, and elements within a tree are the same with or without path compression.

Pf. Path compression does not create new roots, change ranks, or move elements from one tree to another. ▪



Link-by-rank with path compression: properties

PROPERTY. The tree roots, node ranks, and elements within a tree are the same with or without path compression.

COROLLARY. PROPERTY 2, 4–6 hold for link-by-rank with path compression.

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.
PROPERTY 2. If x is not a root node, then $rank[x]$ will never change again.
PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.
PROPERTY 4. Any root node of rank k has $\ge 2^k$ nodes in its tree.
PROPERTY 5. The highest rank of a node is $\le \lfloor \lg n \rfloor$.
PROPERTY 6. For any integer $k \ge 0$, there are $\le n / 2^k$ nodes with rank k.

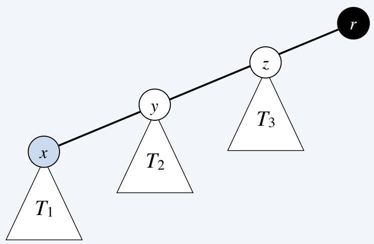
Bottom line. PROPERTY 1–6 hold for link-by-rank with path compression.
(but we need to recheck PROPERTY 1 and PROPERTY 3)

Link-by-rank with path compression: properties

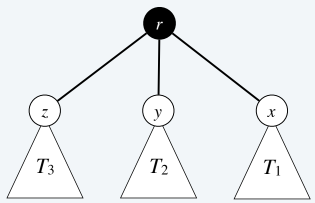
PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.
Pf. Path compression can make x point to only an ancestor of $parent[x]$.

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.
Pf. Path compression doesn't change any ranks, but it can change parents. If $parent[x]$ doesn't change during a path compression, the inequality continues to hold; if $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.

before path compression



after path compression



Iterated logarithm function

Def. The iterated logarithm function is defined by:

$$\lg^* n = \begin{cases} 0 & \text{if } n \le 1 \\ 1 + \lg^*(\lg n) & \text{otherwise} \end{cases}$$

n	$\lg^* n$
1	0
2	1
[3, 4]	2
[5, 16]	3
[17, 65536]	4
[65537, 2^{65536}]	5

iterated lg function

Note. We have $\lg^* n \le 5$ unless n exceeds the # atoms in the universe.

Analysis

Divide nonzero ranks into the following groups:

- $\{ 1 \}$
- $\{ 2 \}$
- $\{ 3, 4 \}$
- $\{ 5, 6, \ldots, 16 \}$
- $\{ 17, 18, \ldots, 2^{16} \}$
- $\{ 65537, 65538, \ldots, 2^{65536} \}$
- ...

Property 7. Every nonzero rank falls within one of the first $\lg^* n$ groups.
Pf. The rank is between 0 and $\lfloor \lg n \rfloor$. [PROPERTY 5]

Creative accounting

Credits. A node receives credits as soon as it ceases to be a root.
If its rank is in the interval $\underbrace{\{ k+1, k+2, \ldots, 2^k \}}_{\text{group } k}$, we give it 2^k credits.

Proposition. Number of credits disbursed to all nodes is $\le n \lg^* n$.
Pf.

- All nodes in group k have rank $\ge k+1$.
- By PROPERTY 6, the number of nodes with rank $\ge k+1$ is at most

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \le \frac{n}{2^k}$$

- Thus, nodes in group k need at most n credits in total.
- There are $\le \lg^* n$ groups. [PROPERTY 7] ▪

Running time of FIND

Running time of FIND. Bounded by number of parent pointers followed.

- Recall: the rank strictly increases as you go up a tree. [PROPERTY 1]
- Case 0: $parent[x]$ is a root $\Rightarrow$ only happens for one link per FIND.
- Case 1: $rank[parent[x]]$ is in a higher group than $rank[x]$.
- Case 2: $rank[parent[x]]$ is in the same group as $rank[x]$.

Case 1. At most $\lg^* n$ nodes on path can be in a higher group. [PROPERTY 7]

Case 2. These nodes are charged 1 credit to follow parent pointer.

- Each time x pays 1 credit, $rank[parent[x]]$ strictly increases. [PROPERTY 1]
- Therefore, if $rank[x]$ is in the group $\{ k+1, \ldots, 2^k \}$, the rank of its parent will be in a higher group before x pays 2^k credits.
- Once $rank[parent[x]]$ is in a higher group than $rank[x]$, it remains so because:
  - $rank[x]$ does not change once it ceases to be a root. [PROPERTY 2]
  - $rank[parent[x]]$ does not decrease. [PROPERTY 3]
  - thus, x has enough credits to pay until it becomes a Case 1 node. ▪

Link-by-rank with path compression

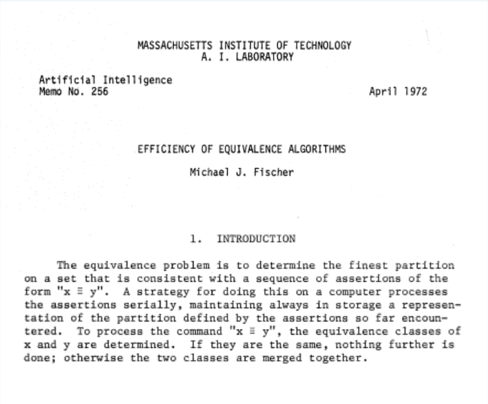
Theorem. Starting from an empty data structure, link-by-rank with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log^* n)$ time.

UNION–FIND

- naïve linking
- link-by-size
- link-by-rank
- path compression
- link-by-rank with path compression
- *context*

Link-by-size with path compression

Theorem. [Fischer 1972] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log \log n)$ time.



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 256

April 1972

EFFICIENCY OF EQUIVALENCE ALGORITHMS

Michael J. Fischer

1. INTRODUCTION

The equivalence problem is to determine the finest partition on a set that is consistent with a sequence of assertions of the form "x ≡ y". A strategy for doing this on a computer processes the assertions serially, maintaining always in storage a representation of the partition defined by the assertions so far encountered. To process the command "x ≡ y", the equivalence classes of x and y are determined. If they are the same, nothing further is done; otherwise the two classes are merged together.

Link-by-size with path compression

Theorem. [Hopcroft–Ullman 1973] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log^* n)$ time.

SIAM J. COMPUT.
Vol. 2, No. 4, December 1973

SET MERGING ALGORITHMS*

J. E. HOPCROFT† AND J. D. ULLMAN‡

Abstract. This paper considers the problem of merging sets formed from a total of n items in such a way that at any time, the name of a set containing a given item can be ascertained. Two algorithms using different data structures are discussed. The execution times of both algorithms are bounded by a constant times $nG(n)$, where $G(n)$ is a function whose asymptotic growth rate is less than that of any finite number of logarithms of n.

Key words. algorithm, algorithmic analysis, computational complexity, data structure, equivalence algorithm, merging, property grammar, set, spanning tree

Link-by-size with path compression

Theorem. [Tarjan 1975] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m\, \alpha(m, n))$ time, where $\alpha(m, n)$ is a functional inverse of the Ackermann function.

Efficiency of a Good But Not Linear Set Union Algorithm

ROBERT ENDRE TARJAN

University of California, Berkeley, California

ABSTRACT. Two types of instructions for manipulating a family of disjoint sets which partition a universe of n elements are considered $FIND(x)$ computes the name of the (unique) set containing element x $UNION(A, B, C)$ combines sets A and B into a new set named C. A known algorithm for implementing sequences of these instructions is examined It is shown that, if $t(m, n)$ is the maximum time required by a sequence of $m \geq n$ $FIND$s and $n - 1$ intermixed $UNION$s, then $k_1 m\alpha(m, n) \leq t(m, n) \leq k_2 m\alpha(m, n)$ for some positive constants k_1 and k_2, where $\alpha(m, n)$ is related to a functional inverse of Ackermann's function and is *very* slow-growing.

Ackermann function

Ackermann function. [Ackermann 1928] A computable function that is not primitive recursive.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Zum Hilbertschen Aufbau der reellen Zahlen.

Von

Wilhelm Ackermann in Göttingen.

Um den Beweis für die von Cantor aufgestellte Vermutung zu erbringen, daß sich die Menge der reellen Zahlen, d. h. der zahlentheoretischen Funktionen, mit Hilfe der Zahlen der zweiten Zahlklasse auszählen läßt, benutzt Hilbert einen speziellen Aufbau der zahlentheoretischen Funktionen. Wesentlich bei diesem Aufbau ist der Begriff des Typs einer Funktion. Eine Funktion vom Typ 1 ist eine solche, deren Argumente und Werte ganze Zahlen sind, also eine gewöhnliche zahlentheoretische Funktion. Die Funktionen vom Typ 2 sind die Funktionenfunktionen. Eine derartige Funktion ordnet jeder zahlentheoretischen Funktion eine Zahl zu. Eine Funktion vom Typ 3 ordnet wieder den Funktionenfunktionen Zahlen zu, usw. Die Definition der Typen läßt sich auch ins Transfinite fortsetzen, für den Gegenstand dieser Arbeit ist das aber nicht von Belang[1]).

Note. There are many inequivalent definitions.

Ackermann function

Ackermann function. [Ackermann 1928] A computable function that is not primitive recursive.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Inverse Ackermann function.

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

" I am not smart enough to understand this easily. "

— Raymond Seidel

Inverse Ackermann function

Definition.

$$\alpha_k(n) = \begin{cases} \lceil n/2 \rceil & \text{if } k = 1 \\ 0 & \text{if } n = 1 \text{ and } k \geq 2 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases}$$

Ex.

- $\alpha_1(n) = \lceil n / 2 \rceil$.
- $\alpha_2(n) = \lceil \lg n \rceil$ = # of times we divide n by 2, until we reach 1.
- $\alpha_3(n) = \lg^* n$ = # of times we apply the lg function to n, until we reach 1.
- $\alpha_4(n)$ = # of times we apply the iterated lg function to n, until we reach 1.

$2 \uparrow 65536 \; = \; \underbrace{2^{2^{2^{\cdot^{\cdot^{2}}}}}}_{65536 \text{ times}}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…	2^{16}	…	2^{65536}	…	$2 \uparrow 65536$
$\alpha_1(n)$	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	…	2^{15}	…	2^{65535}	…	*huge*
$\alpha_2(n)$	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	…	16	…	65536	…	$2 \uparrow 65535$
$\alpha_3(n)$	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	…	4	…	5	…	65536
$\alpha_4(n)$	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	…	3	…	3	…	4

Inverse Ackermann function

Definition.

$$\alpha_k(n) = \begin{cases} \lceil n/2 \rceil & \text{if } k = 1 \\ 0 & \text{if } n = 1 \text{ and } k \geq 2 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases}$$

Property. For every $n \geq 5$, the sequence $\alpha_1(n), \alpha_2(n), \alpha_3(n), \ldots$ converges to 3.

Ex. $[n = 9876!]$ $\alpha_1(n) \geq 10^{35163}$, $\alpha_2(n) = 116812$, $\alpha_3(n) = 6$, $\alpha_4(n) = 4$, $\alpha_5(n) = 3$.

One-parameter inverse Ackermann. $\alpha(n) = \min \{ k : \alpha_k(n) \leq 3 \}$.

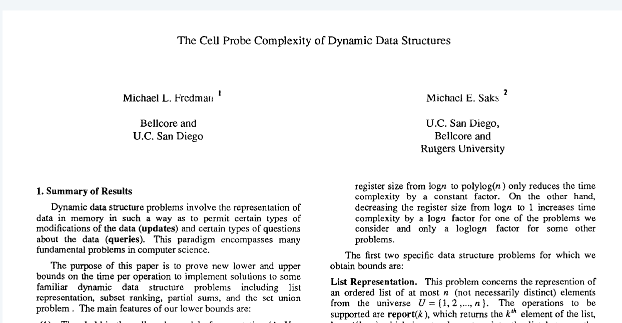
Ex. $\alpha(9876!) = 5$.

Two-parameter inverse Ackermann. $\alpha(m, n) = \min \{ k : \alpha_k(n) \leq 3 + m / n \}$.

A tight lower bound

Theorem. [Fredman–Saks 1989] In the worst case, any CELL-PROBE($\log n$) algorithm requires $\Omega(m \, \alpha(m, n))$ time to perform an intermixed sequence of m MAKE-SET, UNION, and FIND operations on a set of n elements.

Cell-probe model. [Yao 1981] Count only number of words of memory accessed; all other operations are free.



The Cell Probe Complexity of Dynamic Data Structures

Michael L. Fredman [1]

Bellcore and
U.C. San Diego

Michael E. Saks [2]

U.C. San Diego,
Bellcore and
Rutgers University

1. Summary of Results

Dynamic data structure problems involve the representation of data in memory in such a way as to permit certain types of modifications of the data (**updates**) and certain types of questions about the data (**queries**). This paradigm encompasses many fundamental problems in computer science.

The purpose of this paper is to prove new lower and upper bounds on the time per operation to implement solutions to some familiar dynamic data structure problems including list representation, subset ranking, partial sums, and the set union problem . The main features of our lower bounds are:

(1) They hold in the *cell probe* model of computation (A. Yao

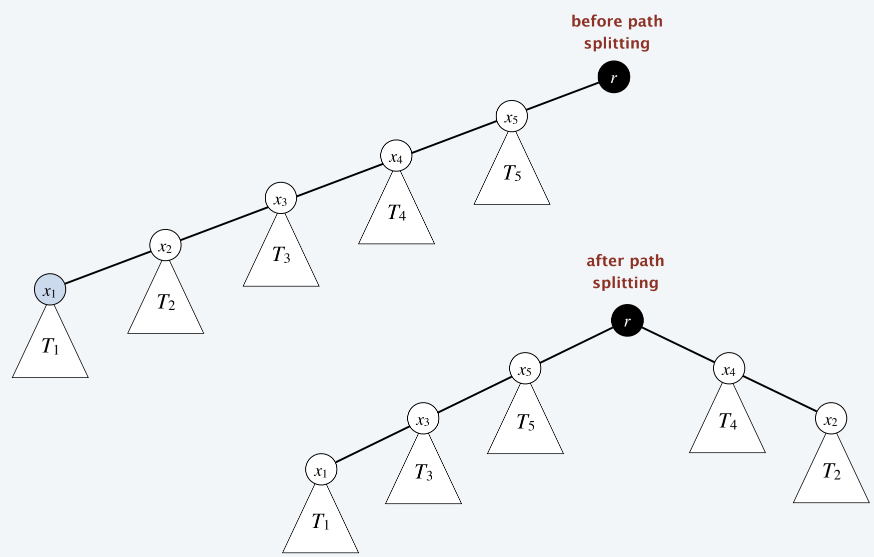
register size from log$n$ to polylog(n) only reduces the time complexity by a constant factor. On the other hand, decreasing the register size from log$n$ to 1 increases time complexity by a log$n$ factor for one of the problems we consider and only a loglog$n$ factor for some other problems.

The first two specific data structure problems for which we obtain bounds are:

List Representation. This problem concerns the represention of an ordered list of at most n (not necessarily distinct) elements from the universe $U = \{1, 2, \ldots, n\}$. The operations to be supported are **report**(k), which returns the k^{th} element of the list, **insert**(k, u) which inserts element u into the list between the

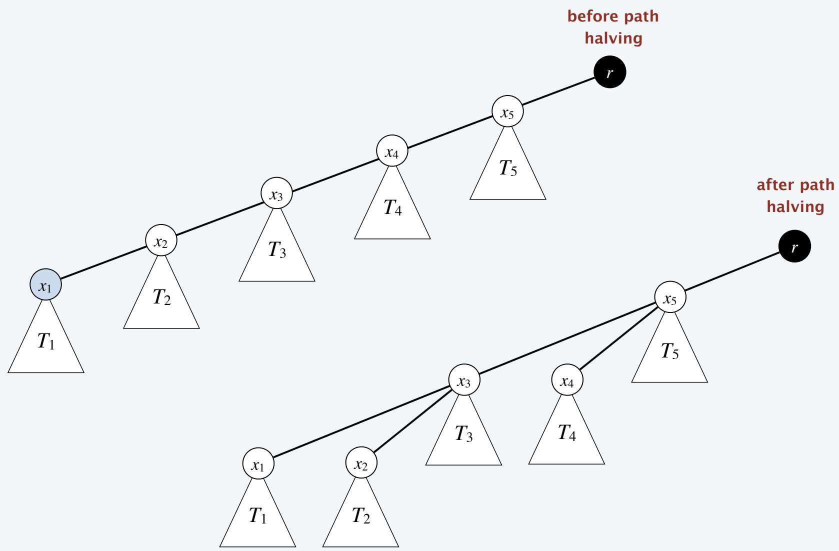
Path compaction variants

Path splitting. Make every node on path point to its grandparent.



Path compaction variants

Path halving. Make every other node on path point to its grandparent.



Linking variants

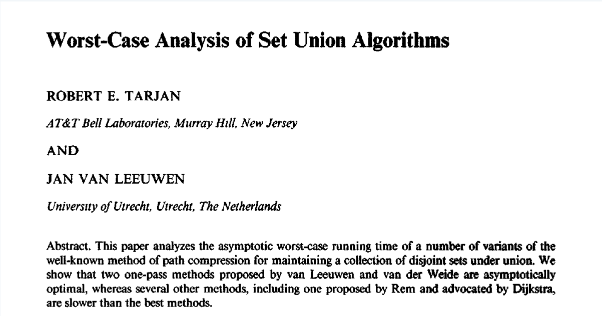
Link-by-size. Number of nodes in tree.

Link-by-rank. Rank of tree.

Link-by-random. Label each element with a random real number between 0.0 and 1.0. Link root with smaller label into root with larger label.

Disjoint-sets data structures

Theorem. [Tarjan–van Leeuwen 1984] Starting from an empty data structure, link-by- { size, rank } combined with { path compression, path splitting, path halving } performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m\ \alpha(m, n))$ time.



Worst-Case Analysis of Set Union Algorithms

ROBERT E. TARJAN

AT&T Bell Laboratories, Murray Hill, New Jersey

AND

JAN VAN LEEUWEN

University of Utrecht, Utrecht, The Netherlands

Abstract. This paper analyzes the asymptotic worst-case running time of a number of variants of the well-known method of path compression for maintaining a collection of disjoint sets under union. We show that two one-pass methods proposed by van Leeuwen and van der Weide are asymptotically optimal, whereas several other methods, including one proposed by Rem and advocated by Dijkstra, are slower than the best methods.