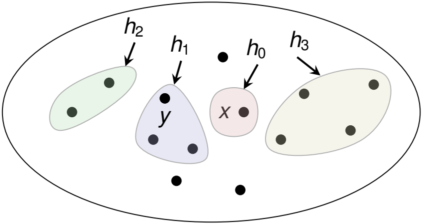


5.3: Disjoint Sets

Frank Stajano

Thomas Sauerwald

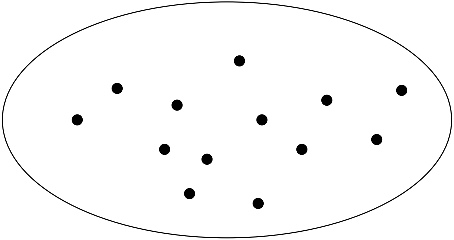
Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

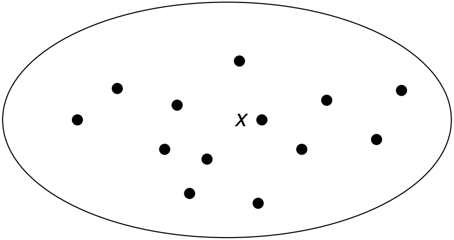


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle

h_0=**`MakeSet(x)`**

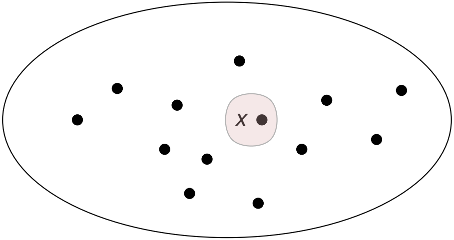


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle

h_0=**`MakeSet(x)`**

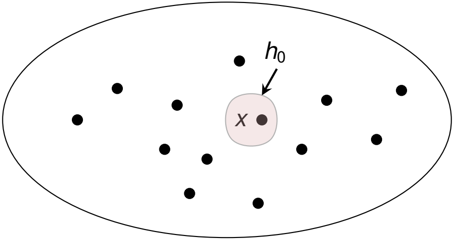


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle

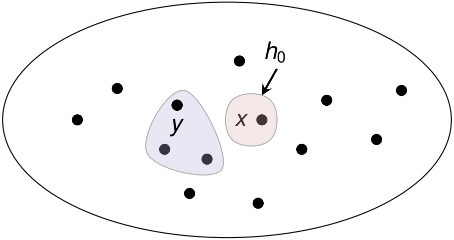
h_0=`MakeSet(x)`



Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

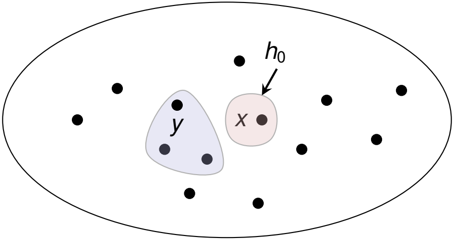


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

h_1=**`FindSet(y)`**

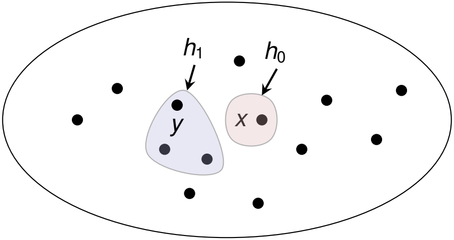


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

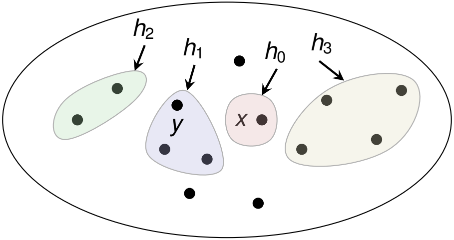
h_1=**`FindSet(y)`**



Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

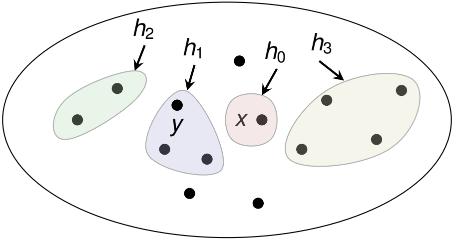


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

h_4=`Union(`h_0`,`h_3`)`

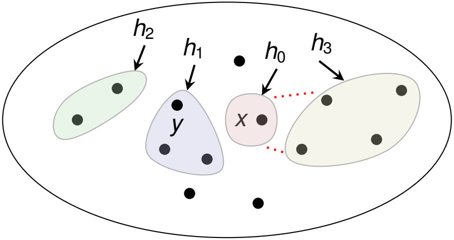


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- `Handle MakeSet(Item x)`
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- `Handle FindSet(Item x)`
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- `Handle Union(Handle h, Handle g)`
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_4=`Union(`h_0`,`h_3`)`

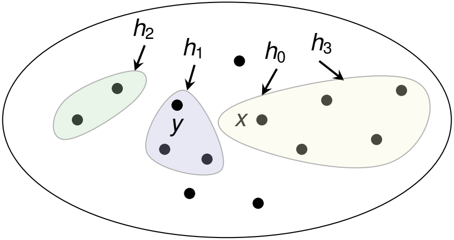


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

h_4=`Union(`h_0`,`h_3`)`

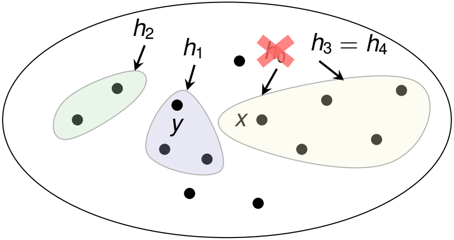


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

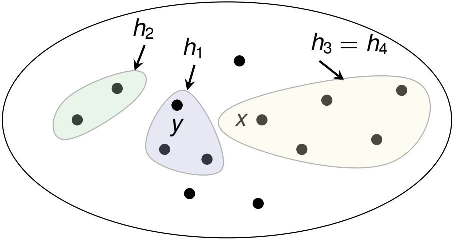
h_4=**Union**(h_0, h_3)



Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

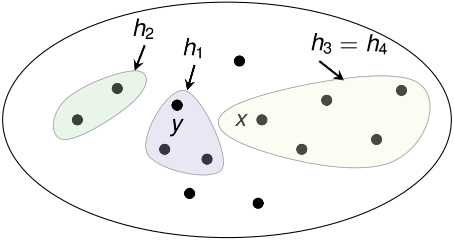


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union(`h_1,h_2`)`

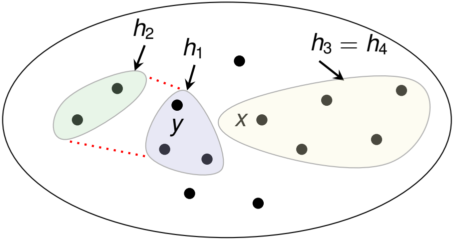


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: $h \neq g$
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=**Union**(h_1, h_2)

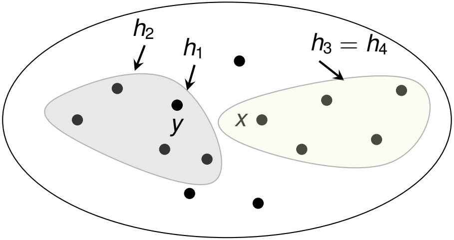


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union(`h_1`,`h_2`)`

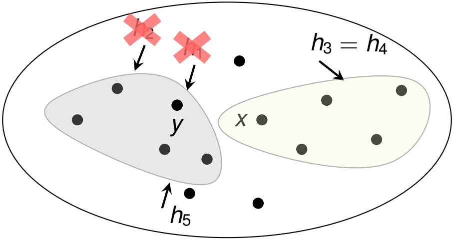


Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- `Handle MakeSet(Item x)`
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- `Handle FindSet(Item x)`
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- `Handle Union(Handle h, Handle g)`
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union(`h_1`,`h_2`)`

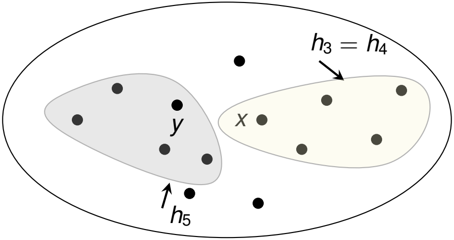


Disjoint Sets (aka Union Find)

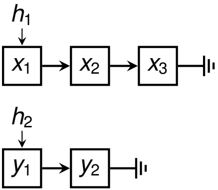
Disjoint Sets Data Structure

- **`Handle MakeSet(Item x)`**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- **`Handle FindSet(Item x)`**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- **`Handle Union(Handle h, Handle g)`**
  Precondition: h $\neq$ g
  Behaviour: merge two **disjoint** sets and return handle of new set

h_5=`Union(`h_1`,`h_2`)`



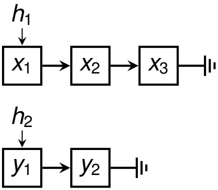
First Attempt: List Implementation



First Attempt: List Implementation

UNION-Operation

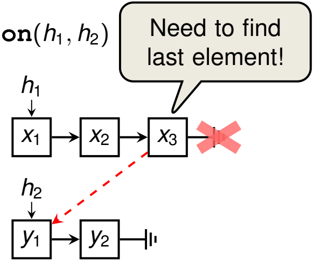
$\texttt{Union}(h_1, h_2)$



First Attempt: List Implementation

UNION-Operation

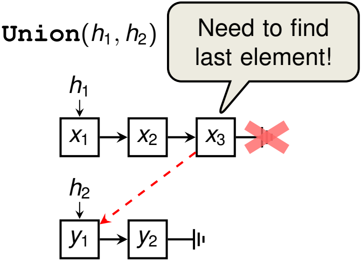
$\mathtt{Union}(h_1, h_2)$



First Attempt: List Implementation

UNION-Operation

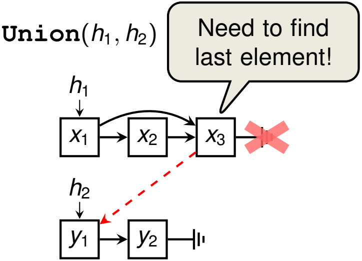
- Add extra pointer to the last element in each list



First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list



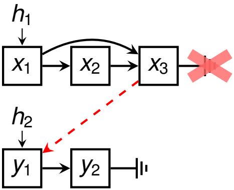
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

Union(h_1, h_2)



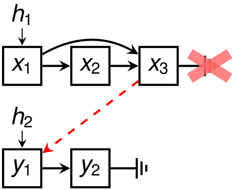
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

`Union`(h_1, h_2)



FINDSET-Operation

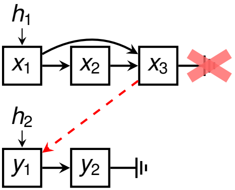
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

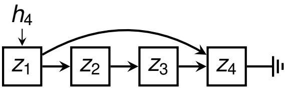
$\Rightarrow$ UNION takes constant time

$\texttt{Union}(h_1, h_2)$



FINDSET-Operation

$\texttt{FindSet}(z_3)$



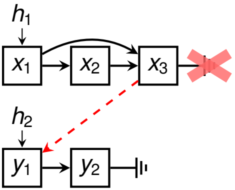
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

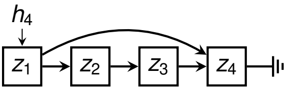
Union(h_1, h_2)



FINDSET-Operation

- Add backward pointer to the list head from everywhere

FindSet(z_3)



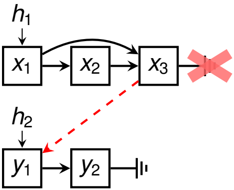
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

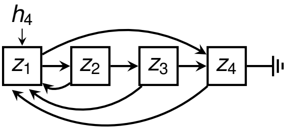
Union(h_1, h_2)



FINDSET-Operation

- Add backward pointer to the list head from everywhere

FindSet(z_3)



First Attempt: List Implementation

Union-Operation

- Add extra pointer to the last element in each list

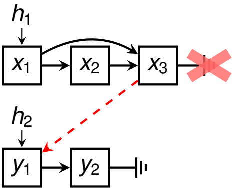
$\Rightarrow$ Union takes constant time

FindSet-Operation

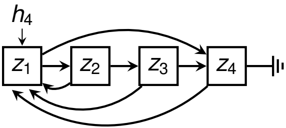
- Add backward pointer to the list head from everywhere

$\Rightarrow$ FindSet takes constant time

`Union`(h_1, h_2)



`FindSet`(z_3)



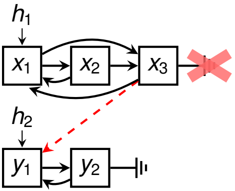
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

Union(h_1, h_2)

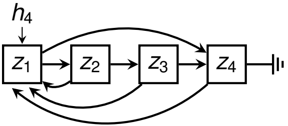


FINDSET-Operation

- Add backward pointer to the list head from everywhere

$\Rightarrow$ FINDSET takes constant time

FindSet(z_3)



First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

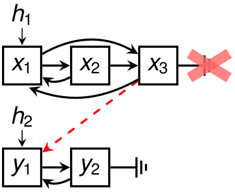
$\Rightarrow$ UNION takes constant time

FINDSET-Operation

- Add backward pointer to the list head from everywhere

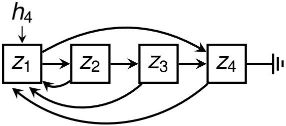
$\Rightarrow$ FINDSET takes constant time

`Union`(h_1, h_2)



Need to update all backward pointers!

`FindSet`(z_3)



First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

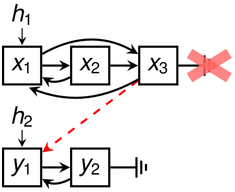
$\Rightarrow$ ~~UNION takes constant time~~

FINDSET-Operation

- Add backward pointer to the list head from everywhere

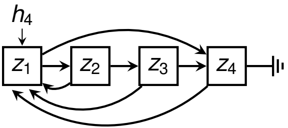
$\Rightarrow$ FINDSET takes constant time

`Union`(h_1, h_2)



Need to update all backward pointers!

`FindSet`(z_3)

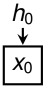


First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$

First Attempt: List Implementation (Analysis)

$d = \mathbf{\texttt{DisjointSet}}()$
$h_0 = d.\mathbf{\texttt{MakeSet}}(x_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

$h_1 = d.\texttt{MakeSet}(x_1)$

First Attempt: List Implementation (Analysis)

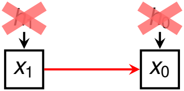
$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$

First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

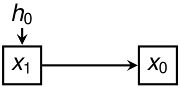
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

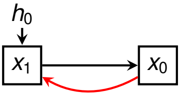
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

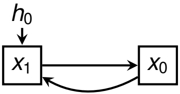
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

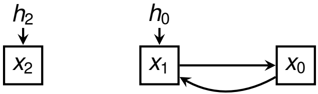
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

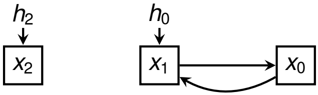
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

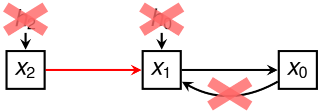
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

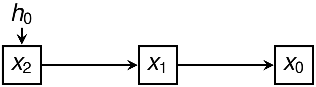
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

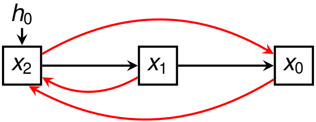
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

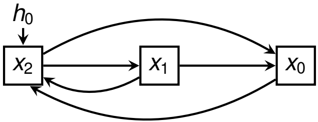
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

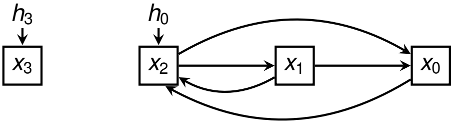
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

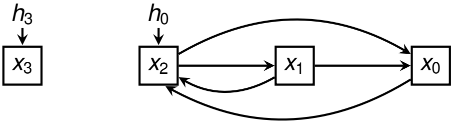
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

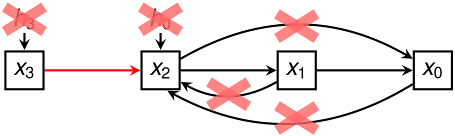
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

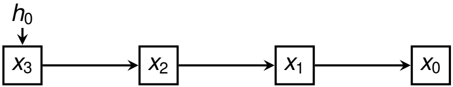
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

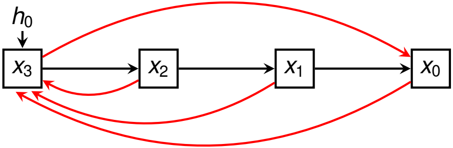
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

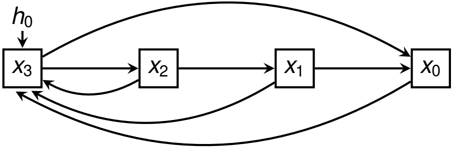
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

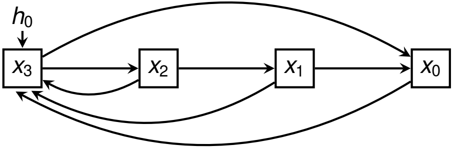
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$

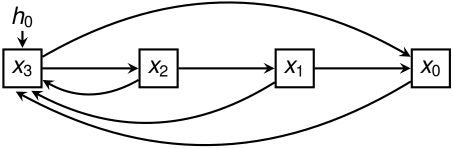


Cost for n UNION operations: $\sum_{i=1}^{n} i = \Theta(n^2)$

First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$
$h_0 = d.\texttt{MakeSet}(x_0)$

$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



better to append shorter list to longer $\rightsquigarrow$ Weighted-Union Heuristic

Cost for n UNION operations: $\sum_{i=1}^{n} i = \Theta(n^2)$

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Weighted-Union Heuristic

Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Amortized Analysis: Every operation has amortized cost $\mathcal{O}(\log n)$, but there may be operations with total cost $\Theta(n)$.

Analysis of Weighted-Union Heuristic

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Analysis of Weighted-Union Heuristic

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

Analysis of Weighted-Union Heuristic

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations

Analysis of Weighted-Union Heuristic

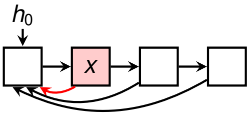
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer

Analysis of Weighted-Union Heuristic



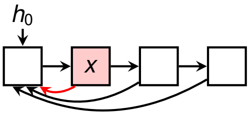
Theorem 21.1

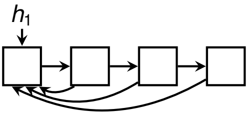
Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer

Analysis of Weighted-Union Heuristic





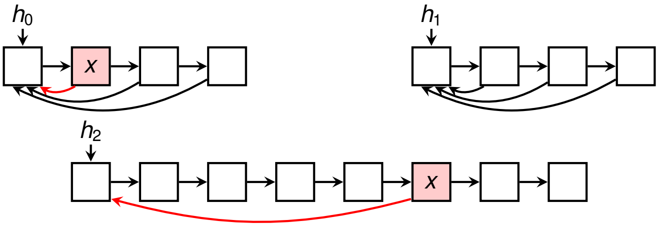
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer

Analysis of Weighted-Union Heuristic



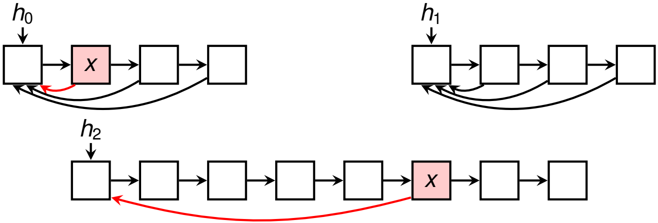
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer

Analysis of Weighted-Union Heuristic



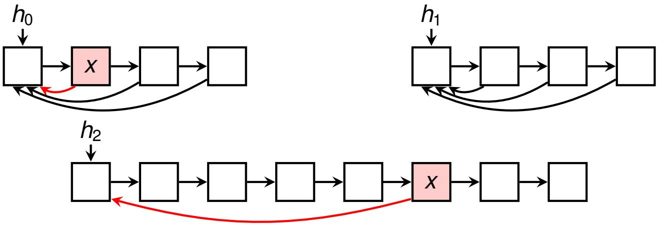
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

Analysis of Weighted-Union Heuristic



Theorem 21.1

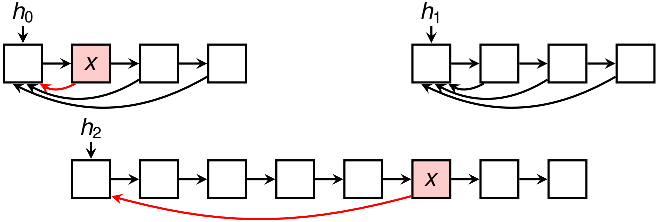
Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n-1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

$\Rightarrow$ Backward pointer of x is updated at most $\log_2 n$ times

Analysis of Weighted-Union Heuristic



Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

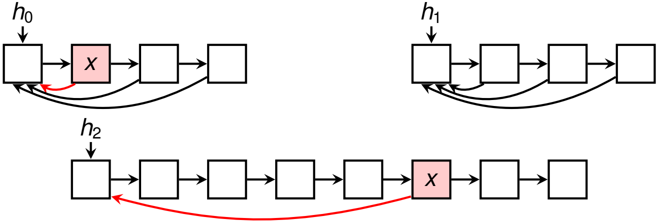
Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

$\Rightarrow$ Backward pointer of x is updated at most $\log_2 n$ times

- Other updates for UNION, MAKE-SET & FIND-SET take $\mathcal{O}(1)$ time per operation

Analysis of Weighted-Union Heuristic



Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

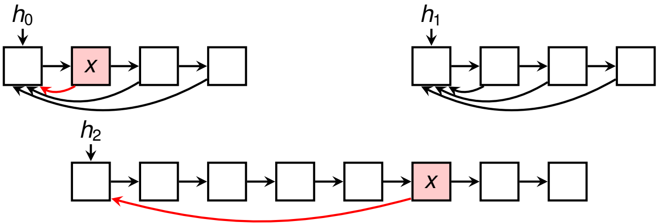
Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

$\Rightarrow$ Backward pointer of x is updated at most $\log_2 n$ times

- Other updates for UNION, MAKE-SET & FIND-SET take $\mathcal{O}(1)$ time per operation

Analysis of Weighted-Union Heuristic



Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKESET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Can we improve on this further?

Proof:

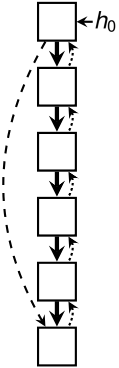
- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of its backward pointer
- After each update of x, its set increases by a factor of at least 2

$\Rightarrow$ Backward pointer of x is updated at most $\log_2 n$ times

- Other updates for UNION, MAKE-SET & FIND-SET take $\mathcal{O}(1)$ time per operation

□

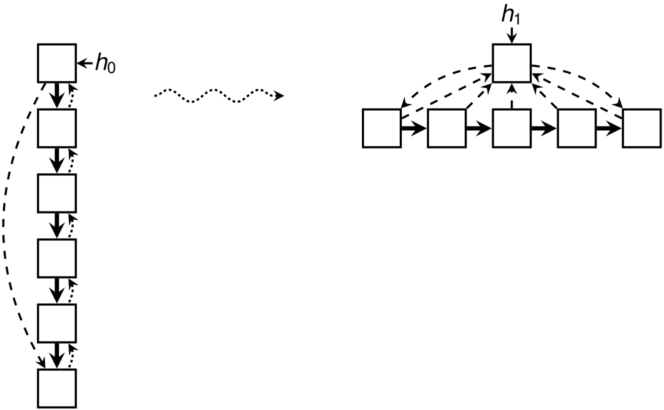
How to Improve?



Doubly-Linked List

- MAKESET: $\mathcal{O}(1)$
- FINDSET: $\mathcal{O}(n)$
- UNION: $\mathcal{O}(1)$

How to Improve?



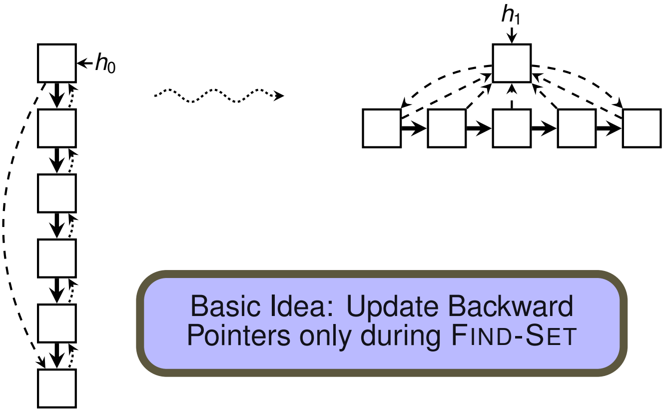
Doubly-Linked List

- MAKESET: $\mathcal{O}(1)$
- FINDSET: $\mathcal{O}(n)$
- UNION: $\mathcal{O}(1)$

Weighted-Union Heuristic

- MAKESET: $\mathcal{O}(1)$
- FINDSET: $\mathcal{O}(1)$
- UNION: $\mathcal{O}(\log n)$ (amortized)

How to Improve?



Basic Idea: Update Backward Pointers only during FIND-SET

Doubly-Linked List

- MAKESET: $\mathcal{O}(1)$
- FINDSET: $\mathcal{O}(n)$
- UNION: $\mathcal{O}(1)$

Weighted-Union Heuristic

- MAKESET: $\mathcal{O}(1)$
- FINDSET: $\mathcal{O}(1)$
- UNION: $\mathcal{O}(\log n)$ (amortized)

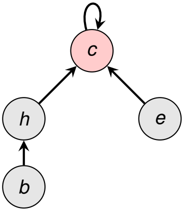
Disjoint Sets via Forests

Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)

Disjoint Sets via Forests

$\{b, c, e, h\}$

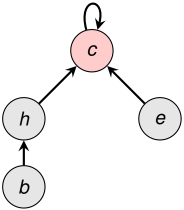


Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)

Disjoint Sets via Forests

$\{b, c, e, h\}$

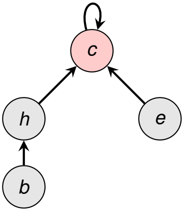


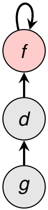
Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Disjoint Sets via Forests

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

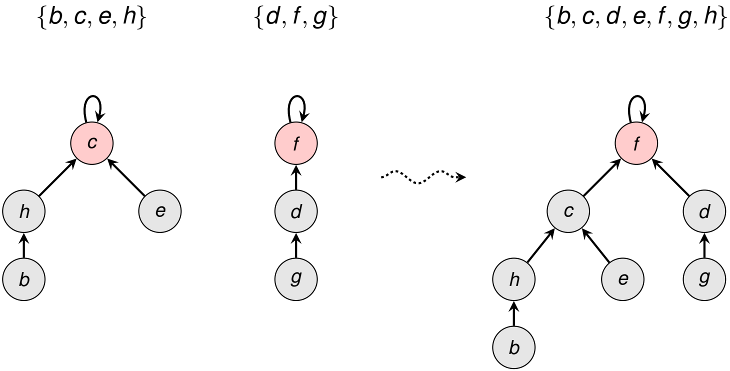




Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

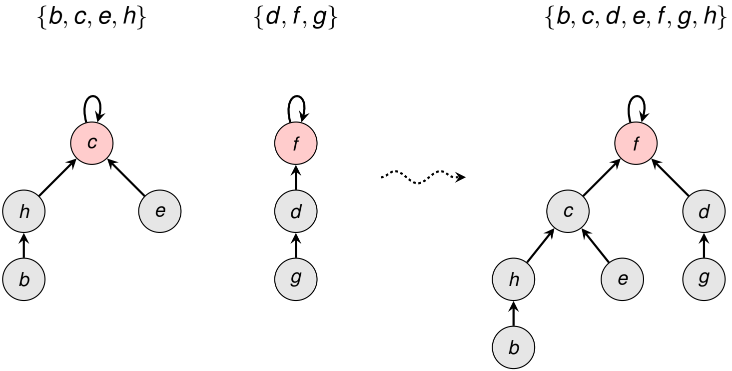
Disjoint Sets via Forests



Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Disjoint Sets via Forests

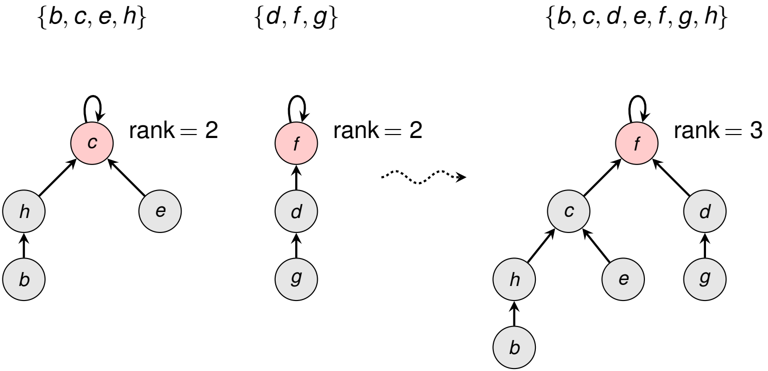


Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Append tree of smaller height $\rightsquigarrow$ Union by Rank

Disjoint Sets via Forests

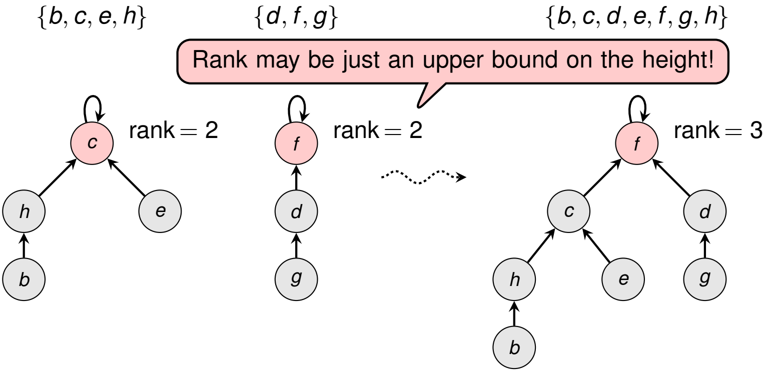


Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Append tree of smaller height $\rightsquigarrow$ Union by Rank

Disjoint Sets via Forests

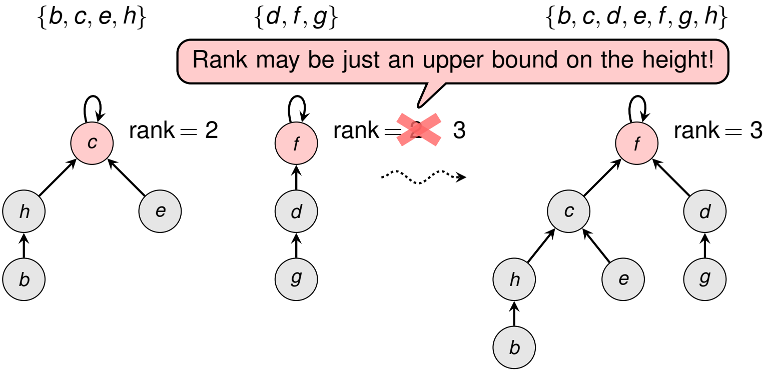


Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Append tree of smaller height $\rightsquigarrow$ Union by Rank

Disjoint Sets via Forests



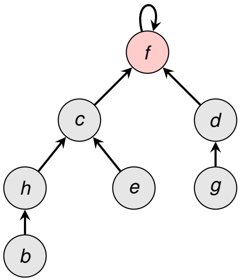
Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $.p$ to its parent (for root x, $x.p = x$)
- UNION: Merge the two trees

Append tree of smaller height ⇝ Union by Rank

Path Compression during FindSet

FindSet(b):

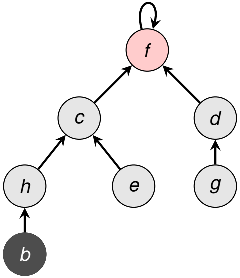



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet(b):

$\boxed{b}$

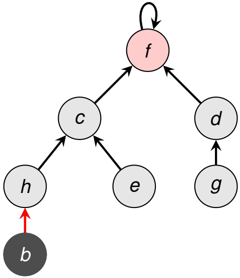



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FindSet

FindSet(b):

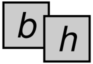
$\boxed{b}$

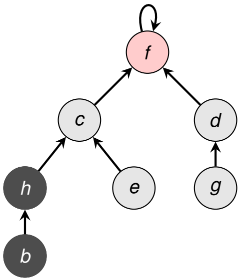



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

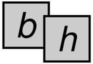
FindSet(b):

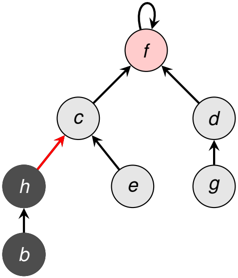





```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

FindSet (b):

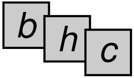


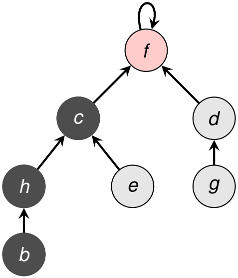



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet(b):

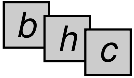


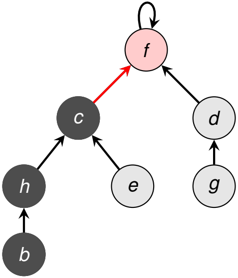



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FindSet

FindSet(b):

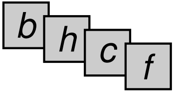


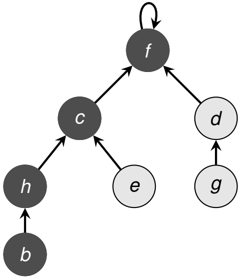



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet(b):

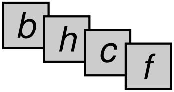


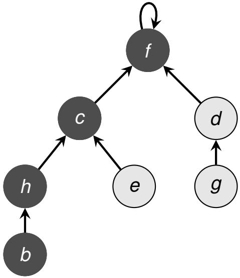



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet(b):

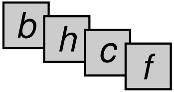


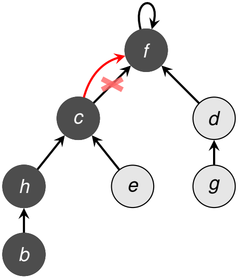



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet(b):

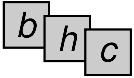


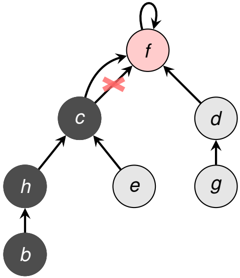



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FINDSET

FindSet(b):

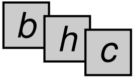


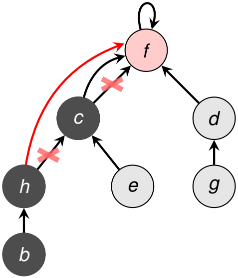



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

FindSet(b):

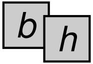


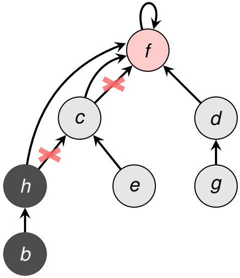



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FindSet

FindSet(b):

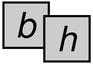


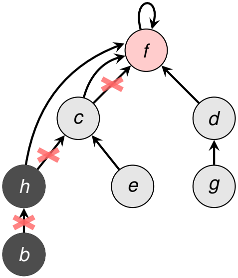



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FindSet

FindSet (b):



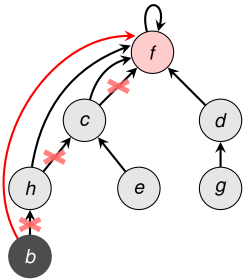



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

FindSet(b):

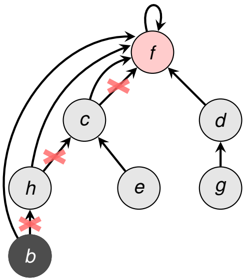
$\boxed{b}$




```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

FindSet(*b*):

$\boxed{b}$

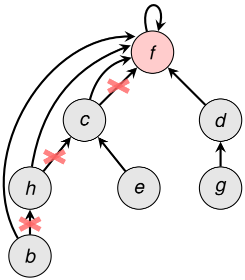



```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

Path Compression during FindSet

FindSet(b):

$\boxed{b}$

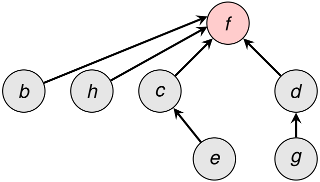



```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

FindSet(b):

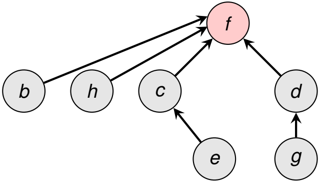
$\boxed{b}$




```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FINDSET

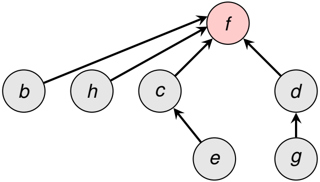
FindSet(*b*):




```
FindSet(x)
  if x ≠ x.p
    x.p = FindSet(x.p)
  return x.p
```

Path Compression during FINDSET

`FindSet`(b):



Maintaining the exact height would be costly, hence rank is only an **upper bound**!

```
0: FindSet(x)
1:   if x ≠ x.p
2:     x.p = FindSet(x.p)
3:   return x.p
```

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

More than the number of atoms in the universe!

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

$\log^*(n)$, **the iterated logarithm**, satifies $\alpha(n) \leq \log^*(n)$, but still $\log^*(10^{80}) = 5$.

Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

In practice, $\alpha(n)$ is a small constant

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

Combining Union by Rank and Path Compression

Data Structure is essentially optimal! (for more details see CLRS)

Theorem 21.14

Any sequence of m MAKESET, UNION, FINDSET operations, n of which are MAKESET operations, can be performed in $\mathcal{O}(m \cdot \alpha(n))$ time.

In practice, $\alpha(n)$ is a small constant

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \leq n \leq 7, \\ 3 & \text{for } 8 \leq n \leq 2047, \\ 4 & \text{for } 2048 \leq n \leq 10^{80} \end{cases}$$

Simulating the Effects of Union by Rank and Path Compression

Simulating the Effects of Union by Rank and Path Compression

Experimental Setup

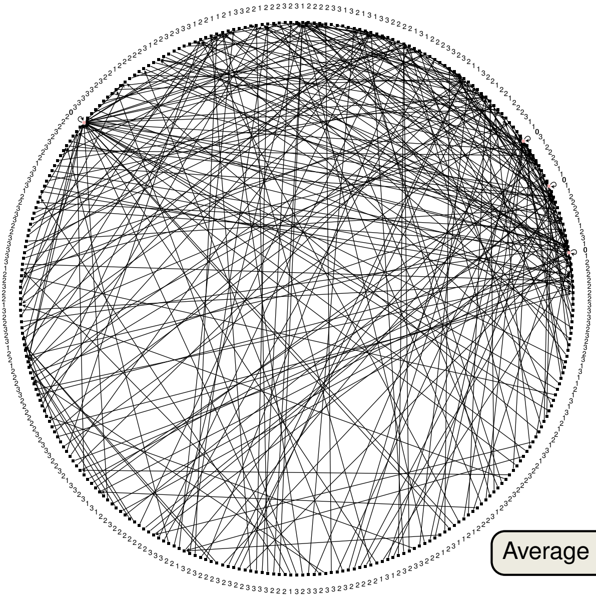
1. Initialise singletons $1, 2, \ldots, 300$
2. For every $1 \leq i \leq 300$, pick a random $1 \leq r \leq 300$, $r \neq i$ and perform $\textsc{Union}(\textsc{FindSet}(i), \textsc{FindSet}(r))$

Simulating the Effects of Union by Rank and Path Compression

Experimental Setup

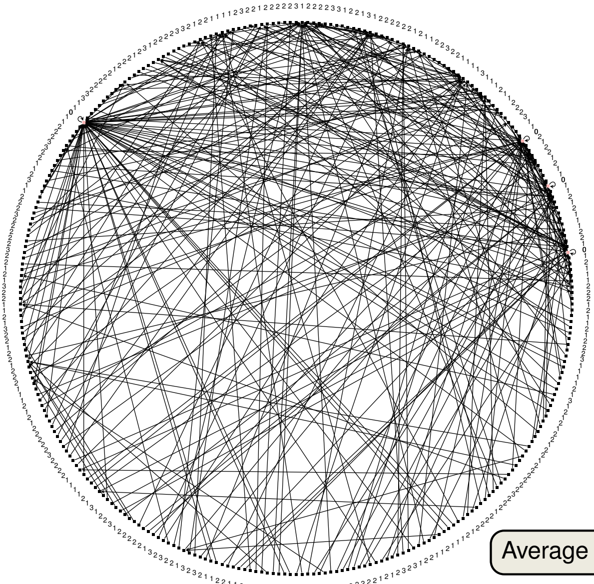
1. Initialise singletons $1, 2, \ldots, 300$
2. For every $1 \leq i \leq 300$, pick a random $1 \leq r \leq 300$, $r \neq i$ and perform UNION(FINDSET(i), FINDSET(r))
3. Perform $j \in \{0, 100, 200, 300, 600, 900, 1200, 1500, 1800\}$ many additional FINDSET(r), where $1 \leq r \leq 300$ is random

Union by Rank without Path Compression



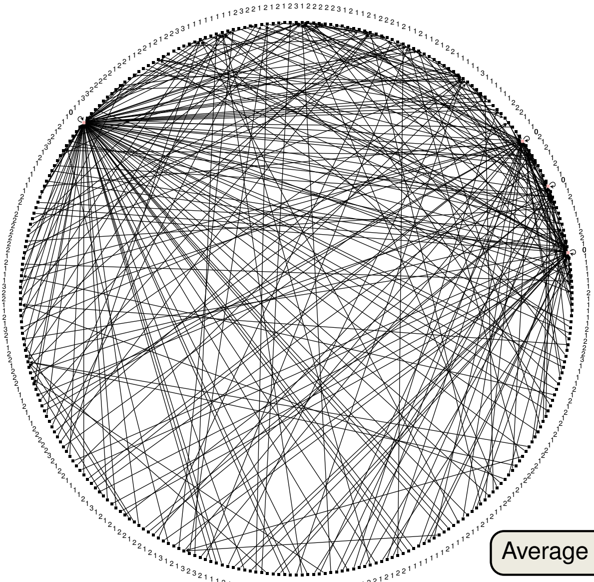
Average Height: 2.12

Union by Rank with Path Compression



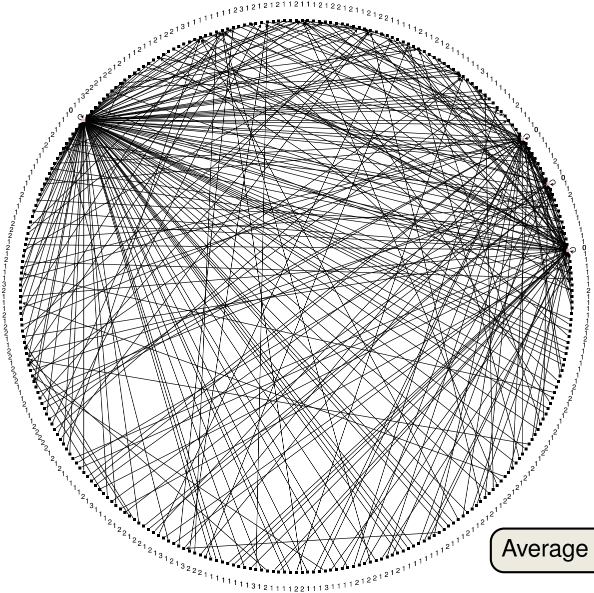
Average Height: 1.75

Union by Rank with Path Compression (100 additional FINDSET)



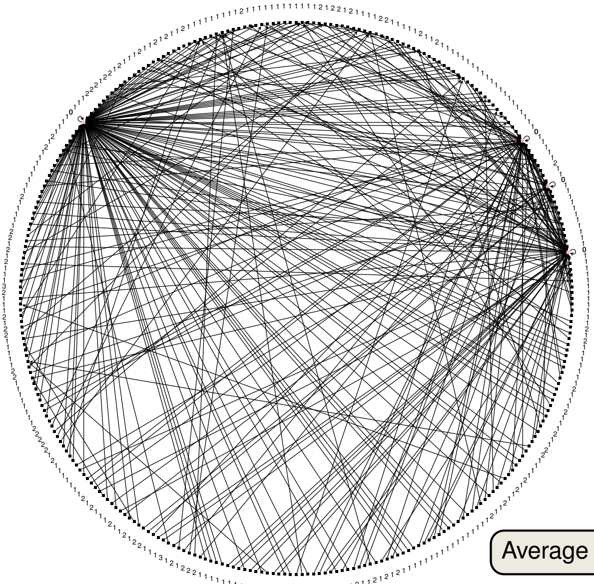
Average Height: 1.53

Union by Rank with Path Compression (200 additional FINDSET)



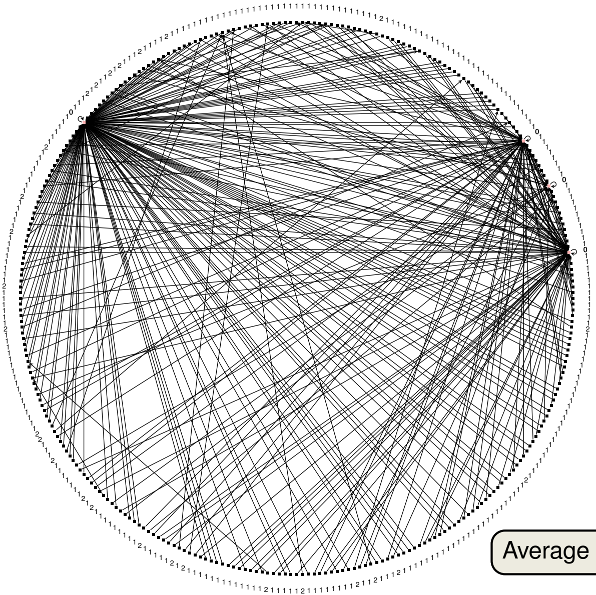
Average Height: 1.35

Union by Rank with Path Compression (300 additional FINDSET)



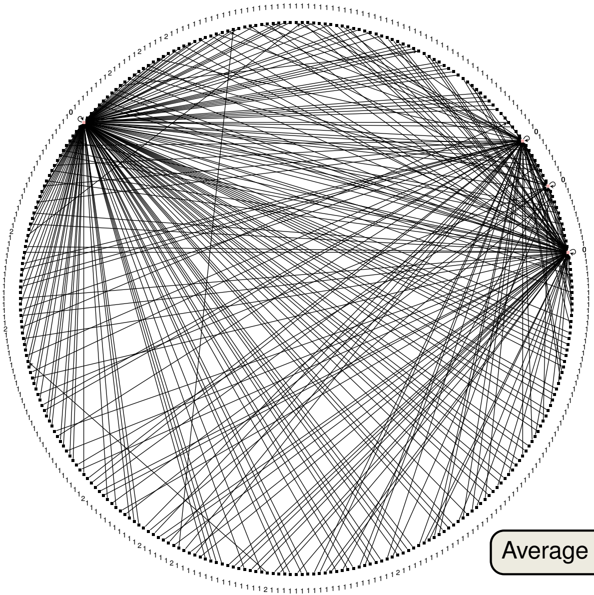
Average Height: 1.22

Union by Rank with Path Compression (600 additional FINDSET)



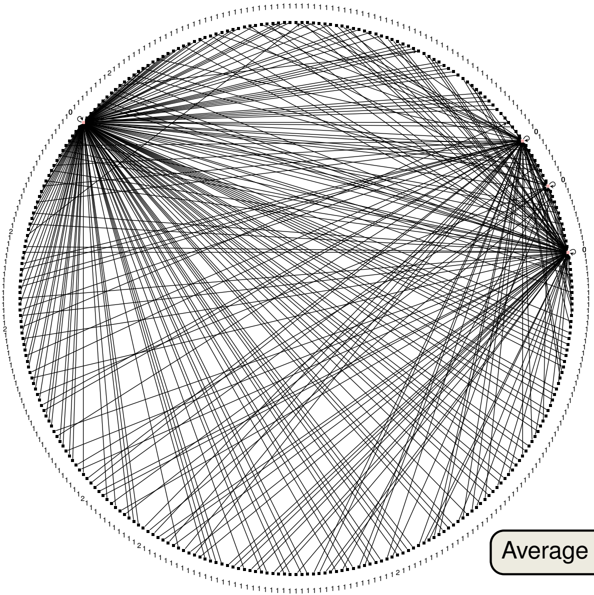
Average Height: 1.08

Union by Rank with Path Compression (900 additional FINDSET)



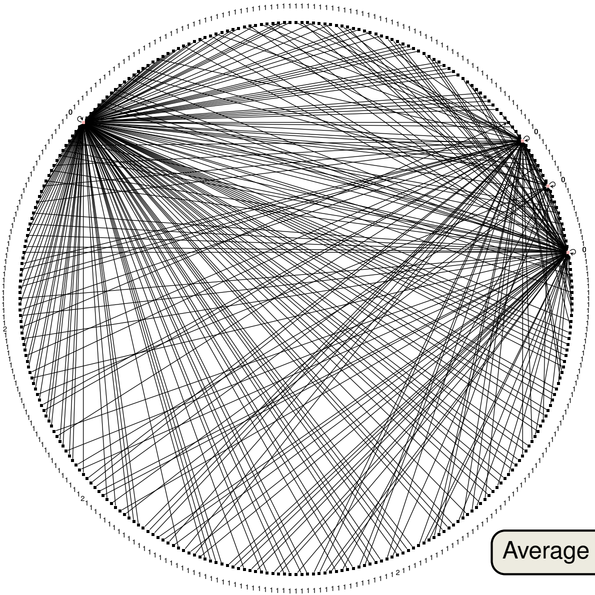
Average Height: 1.02

Union by Rank with Path Compression (1200 additional FINDSET)



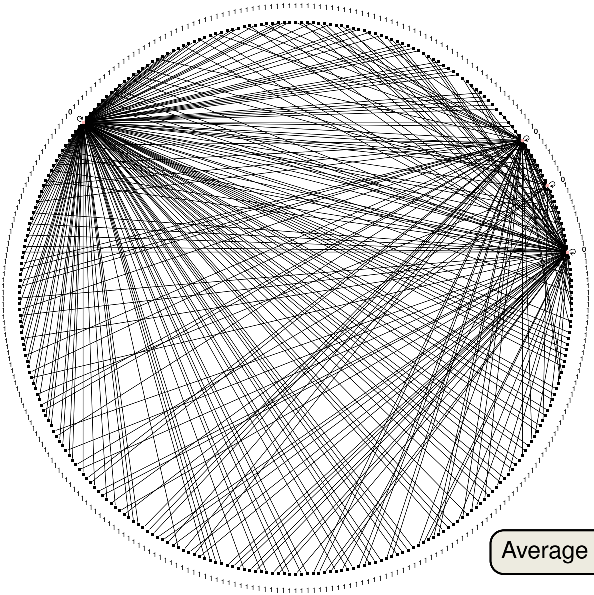
Average Height: 1.01

Union by Rank with Path Compression (1500 additional FINDSET)



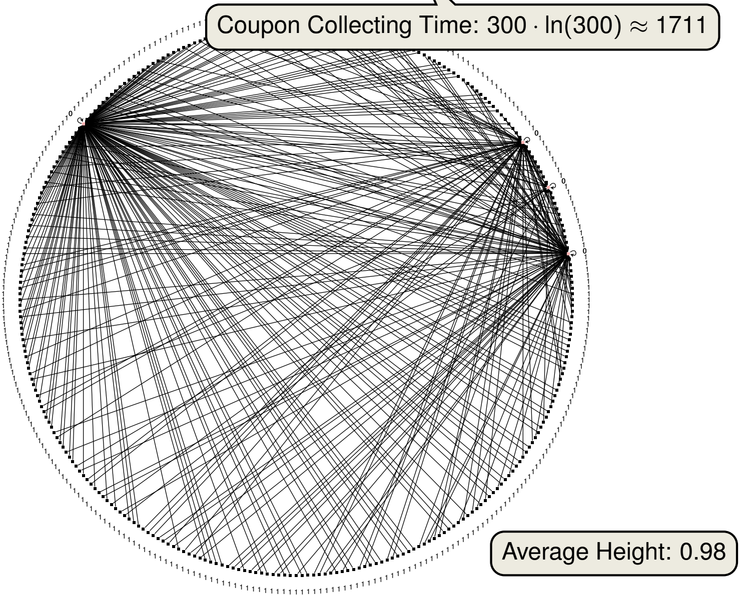
Average Height: 1.00

Union by Rank with Path Compression (1800 additional FINDSET)



Average Height: 0.98

Union by Rank with Path Compression (1800 additional FINDSET)



Overview

	Union by Rank	Union by Rank & Path Compression
300 MAKESET & 300 UNION	2.12	1.75
100 extra FINDSET	2.12	1.53
200 extra FINDSET	2.12	1.35
300 extra FINDSET	2.12	1.22
600 extra FINDSET	2.12	1.08
900 extra FINDSET	2.12	1.02
1200 extra FINDSET	2.12	1.01
1500 extra FINDSET	2.12	1.00
1800 extra FINDSET	2.12	0.98