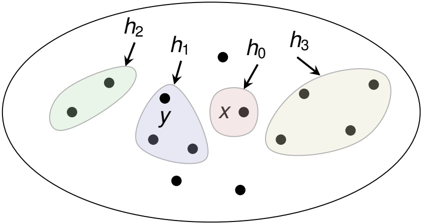


5.3: Disjoint Sets

Frank Stajano

Thomas Sauerwald

Lent 2015

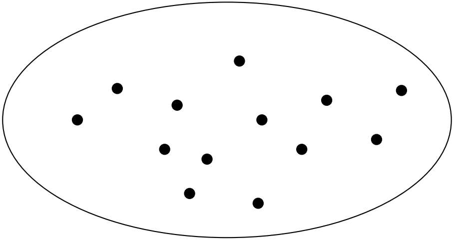
Outline

Disjoint Sets

Introduction to Graphs and Graph Searching

Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

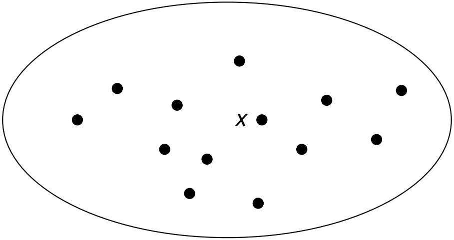


Disjoint Sets (aka Union Find)

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)`

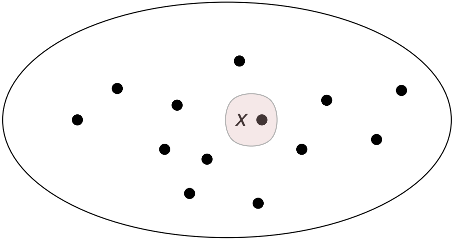


Disjoint Sets (aka Union Find)

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)`**

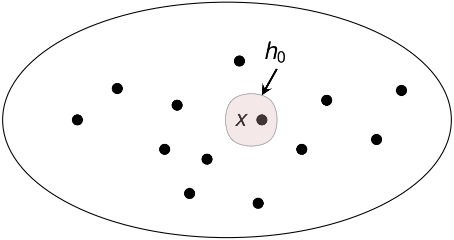


Disjoint Sets (aka Union Find)

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

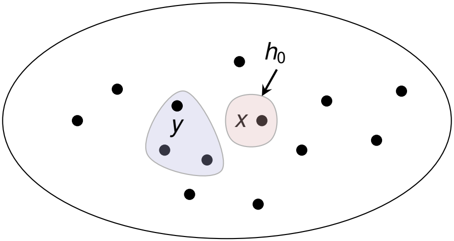
h_0=**`makeSet(x)`**



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

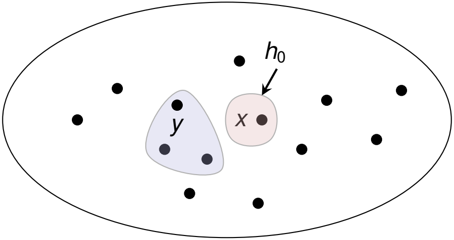


Disjoint Sets (aka Union Find)

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)`

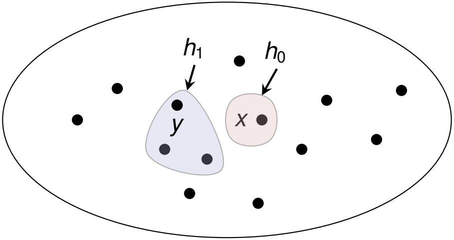


Disjoint Sets (aka Union Find)

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

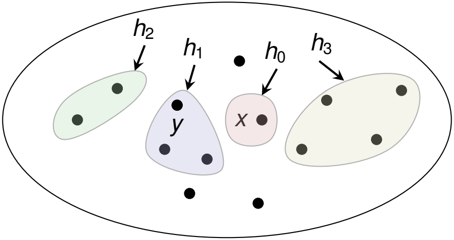
h_1=**`findSet(y)`**



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

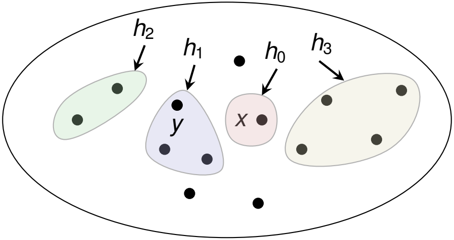


Disjoint Sets (aka Union Find)

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`)`

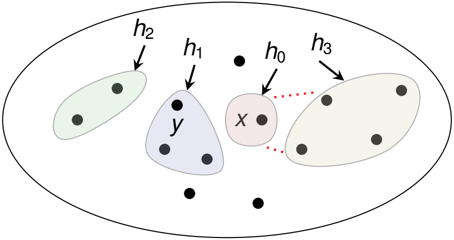


Disjoint Sets (aka Union Find)

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`)`

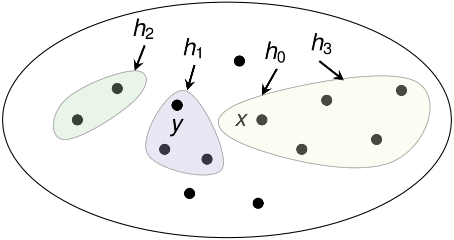


Disjoint Sets (aka Union Find)

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`)`

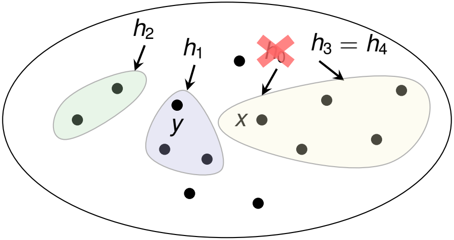


Disjoint Sets (aka Union Find)

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

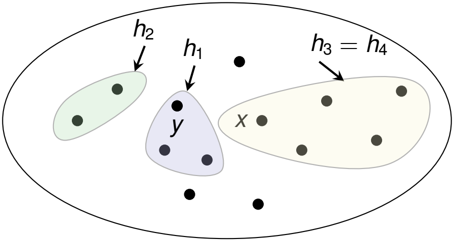
h_4`=Union(`h_0`,`h_3`)`



Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

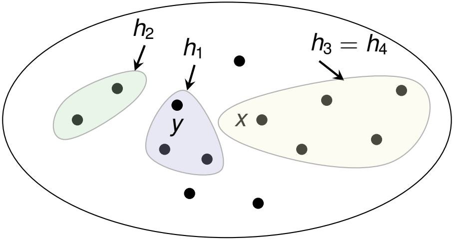
- `Handle MakeSet(Item x)`
  Precondition: none of the existing sets contains x
  Behaviour: create a new set $\{x\}$ and return its handle
- `Handle FindSet(Item x)`
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x
- `Handle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



Disjoint Sets 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`)`

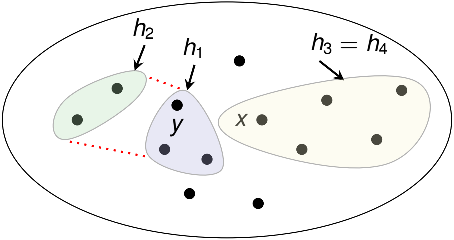


Disjoint Sets (aka Union Find)

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`)`

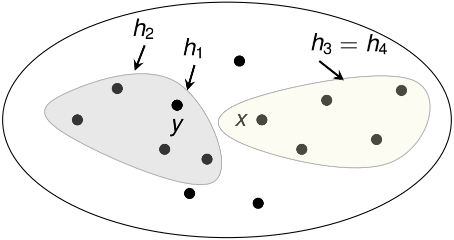


Disjoint Sets (aka Union Find)

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`)`

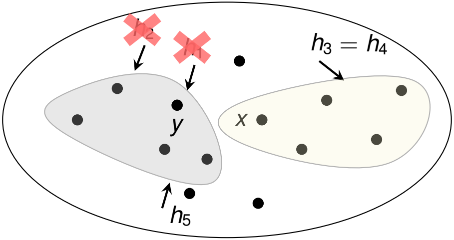


Disjoint Sets (aka Union Find)

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`)`

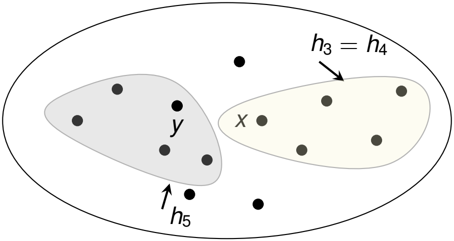


Disjoint Sets (aka Union Find)

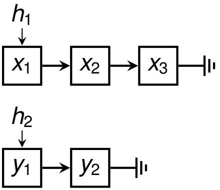
Disjoint Sets Data Structure

- **`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`)`



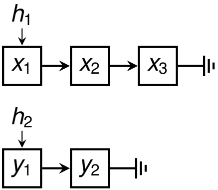
First Attempt: List Implementation



First Attempt: List Implementation

UNION-Operation

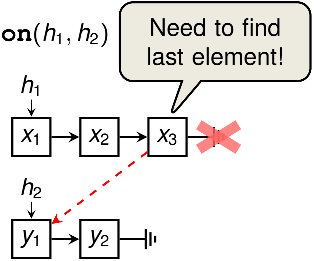
`Union`(h_1, h_2)



First Attempt: List Implementation

UNION-Operation

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

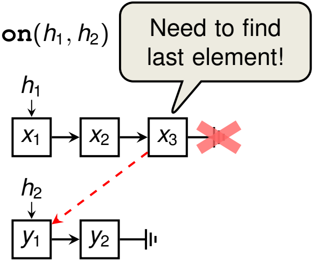


First Attempt: List Implementation

Union-Operation

- Add extra pointer to the last element in each list

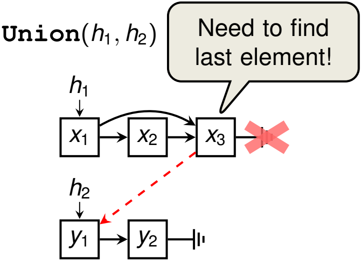
`Union`(h_1, h_2)



First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list



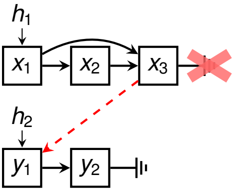
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

Union(h_1, h_2)



First Attempt: List Implementation

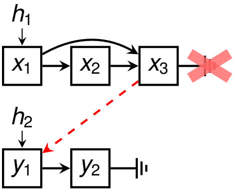
UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

FIND-Operation

Union(h_1, h_2)



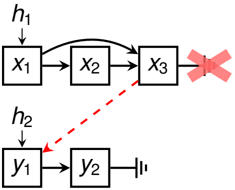
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

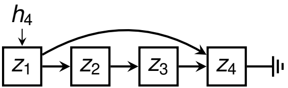
$\Rightarrow$ UNION takes constant time

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



FIND-Operation

$\texttt{FindSet}(z_3)$



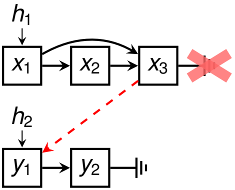
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

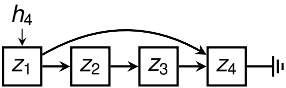
$\texttt{Union}(h_1, h_2)$



FIND-Operation

- Add backward pointer to the list head from everywhere

$\texttt{FindSet}(z_3)$



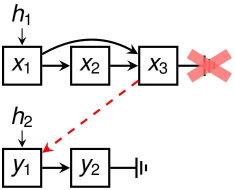
First Attempt: List Implementation

Union-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ Union takes constant time

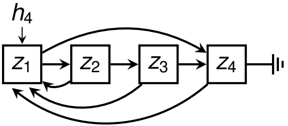
$\texttt{Union}(h_1, h_2)$



Find-Operation

- Add backward pointer to the list head from everywhere

$\texttt{FindSet}(z_3)$



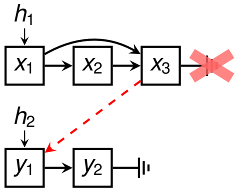
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

`Union`(h_1, h_2)

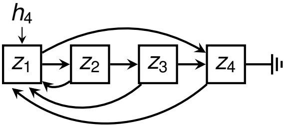


FIND-Operation

- Add backward pointer to the list head from everywhere

$\Rightarrow$ FIND takes constant time

`FindSet`(z_3)



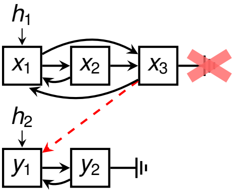
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

$\Rightarrow$ UNION takes constant time

Union(h_1, h_2)

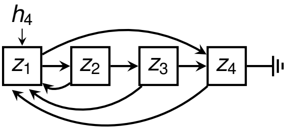


FIND-Operation

- Add backward pointer to the list head from everywhere

$\Rightarrow$ FIND takes constant time

FindSet(z_3)



First Attempt: List Implementation

Union-Operation

- Add extra pointer to the last element in each list

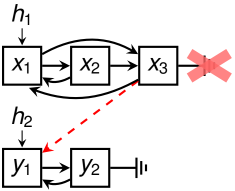
$\Rightarrow$ Union takes constant time

Find-Operation

- Add backward pointer to the list head from everywhere

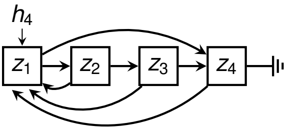
$\Rightarrow$ Find takes constant time

Union(h_1, h_2)



Need to update all backward pointers!

FindSet(z_3)



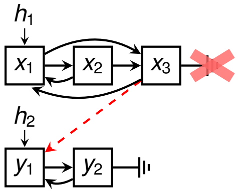
First Attempt: List Implementation

UNION-Operation

- Add extra pointer to the last element in each list

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

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



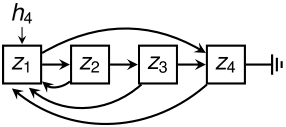
Need to update all backward pointers!

FIND-Operation

- Add backward pointer to the list head from everywhere

$\Rightarrow$ FIND takes constant time

$\texttt{FindSet}(z_3)$

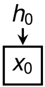


First Attempt: List Implementation (Analysis)

$d = \texttt{DisjointSet}()$

First Attempt: List Implementation (Analysis)

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



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)

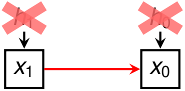
$d = \texttt{\textbf{DisjointSet}}()$
$h_0 = d.\texttt{\textbf{MakeSet}}(x_0)$

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

First Attempt: List Implementation (Analysis)

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

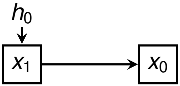
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

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

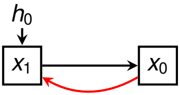
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

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

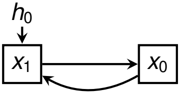
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

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

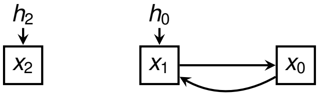
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$



First Attempt: List Implementation (Analysis)

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

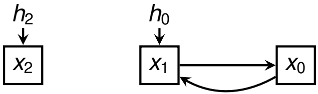
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$



First Attempt: List Implementation (Analysis)

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

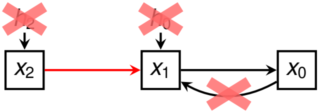
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

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

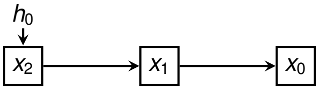
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

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

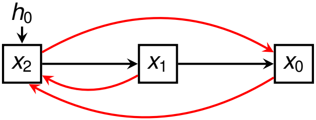
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

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

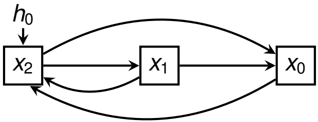
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

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

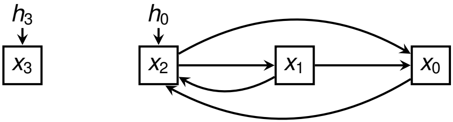
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$



First Attempt: List Implementation (Analysis)

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

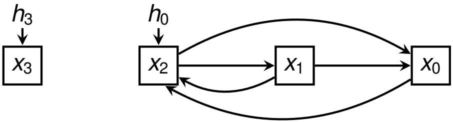
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$



First Attempt: List Implementation (Analysis)

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

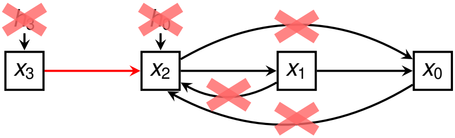
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

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

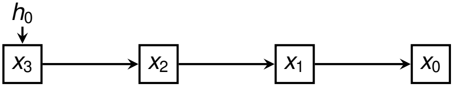
$h_1 = d.\textbf{MakeSet}(x_1)$
$h_0 = d.\textbf{Union}(h_1, h_0)$
$h_2 = d.\textbf{MakeSet}(x_2)$
$h_0 = d.\textbf{Union}(h_2, h_0)$
$h_3 = d.\textbf{MakeSet}(x_3)$
$h_0 = d.\textbf{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

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

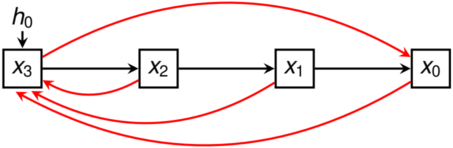
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

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

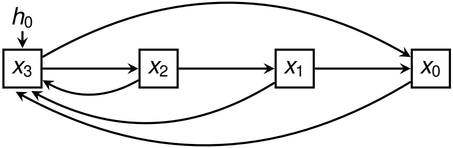
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

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

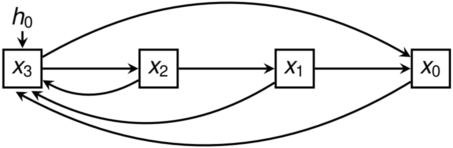
$h_1 = d.\texttt{MakeSet}(x_1)$
$h_0 = d.\texttt{Union}(h_1, h_0)$
$h_2 = d.\texttt{MakeSet}(x_2)$
$h_0 = d.\texttt{Union}(h_2, h_0)$
$h_3 = d.\texttt{MakeSet}(x_3)$
$h_0 = d.\texttt{Union}(h_3, h_0)$



First Attempt: List Implementation (Analysis)

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

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

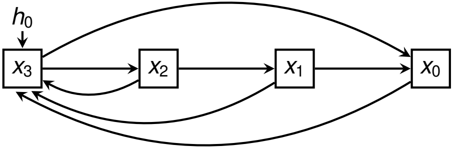


Cost for n UNION operations: $\sum_{i=1}^{n} i = \Theta(n^2)$

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 MAKE-SET 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 MAKE-SET 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 MAKE-SET 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 MAKE-SET 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 MAKE-SET 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

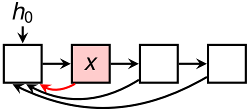
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKE-SET 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 the backward pointer

Analysis of Weighted-Union Heuristic



Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKE-SET 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 the backward pointer

Analysis of Weighted-Union Heuristic

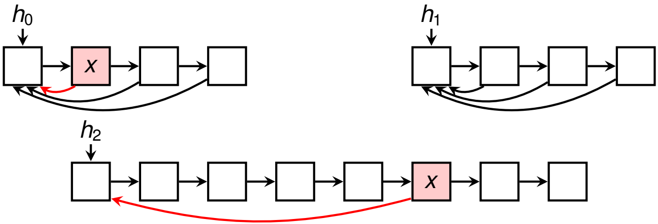
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKE-SET 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 the backward pointer

Analysis of Weighted-Union Heuristic



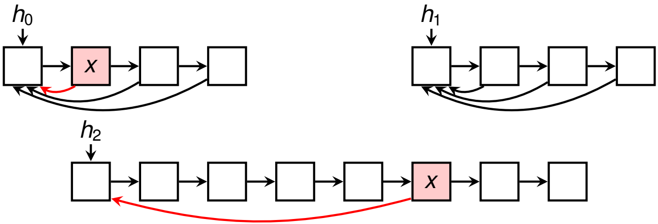
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKE-SET 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 the backward pointer

Analysis of Weighted-Union Heuristic



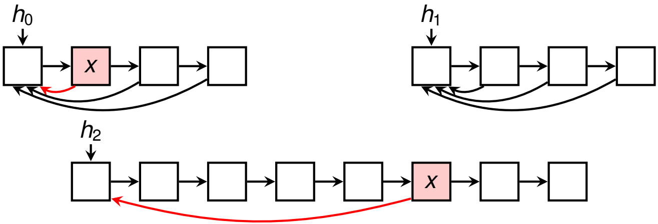
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKE-SET 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 the 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

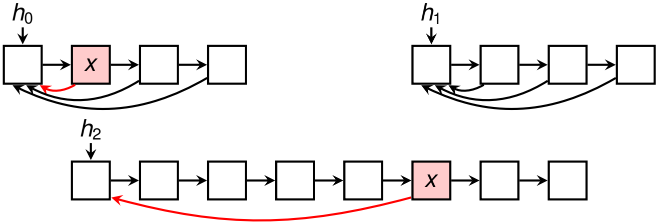
Using the Weighted-Union heuristic, any sequence of m operations, n of which are MAKE-SET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n-1$ UNION operations
- Consider element x and the number of updates of the 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 MAKE-SET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

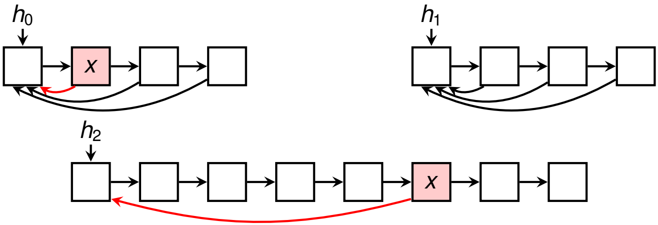
Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of the 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 MAKE-SET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

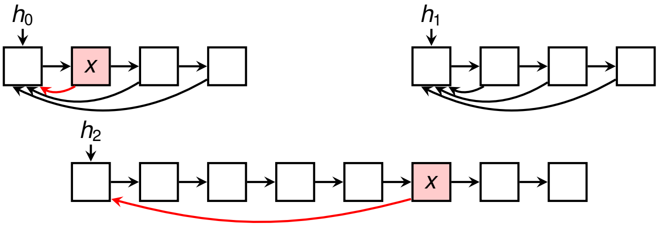
Proof:

- n MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element x and the number of updates of the 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 MAKE-SET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Can we improve on this further?

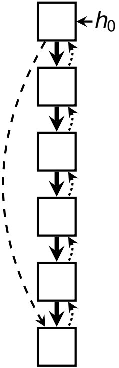
Proof:

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

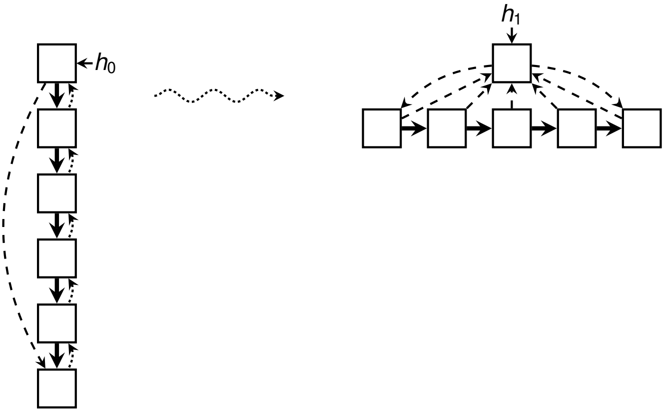
How to Improve?



Doubly-Linked List

- MAKE-SET: $\mathcal{O}(1)$
- FIND-SET: $\mathcal{O}(n)$
- UNION: $\mathcal{O}(1)$

How to Improve?



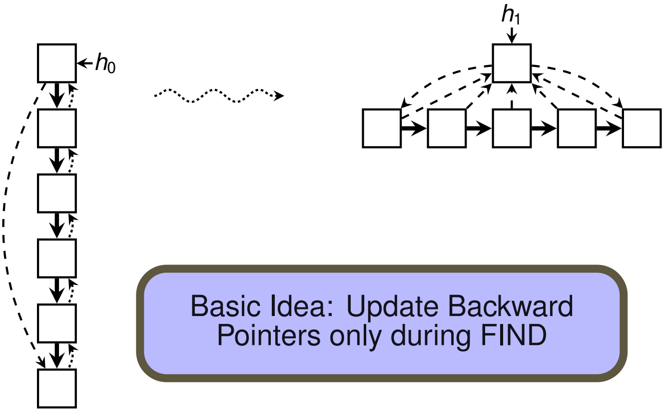
Doubly-Linked List

- MAKE-SET: $\mathcal{O}(1)$
- FIND-SET: $\mathcal{O}(n)$
- UNION: $\mathcal{O}(1)$

Weighted-Union Heuristic

- MAKE-SET: $\mathcal{O}(1)$
- FIND-SET: $\mathcal{O}(1)$
- UNION: $\mathcal{O}(\log n)$ (amortized)

How to Improve?



Basic Idea: Update Backward Pointers only during FIND

Doubly-Linked List

- MAKE-SET: $\mathcal{O}(1)$
- FIND-SET: $\mathcal{O}(n)$
- UNION: $\mathcal{O}(1)$

Weighted-Union Heuristic

- MAKE-SET: $\mathcal{O}(1)$
- FIND-SET: $\mathcal{O}(1)$
- UNION: $\mathcal{O}(\log n)$ (amortized)

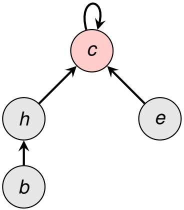
Disjoint Sets via Forests

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

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

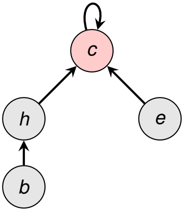


Forest Structure

- 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

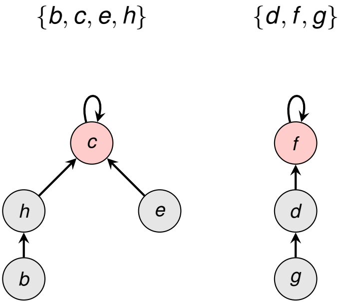
$\{b, c, e, h\}$



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

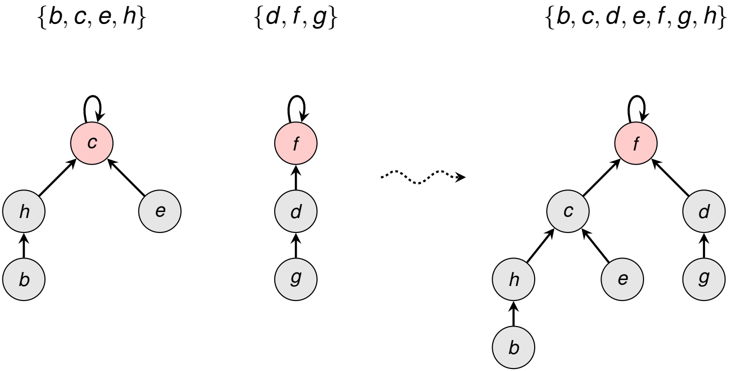
Disjoint Sets via Forests



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

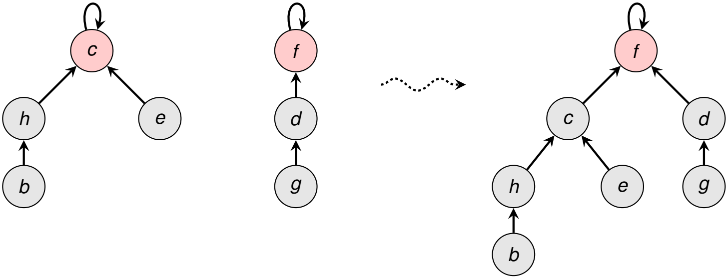


Forest Structure

- 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

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

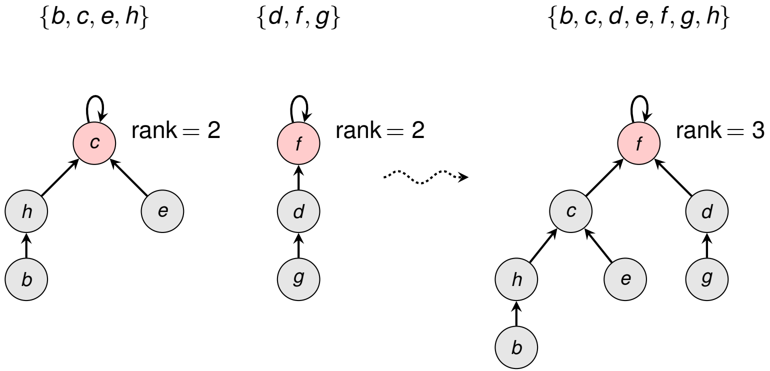


Forest Structure

- 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

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

Disjoint Sets via Forests

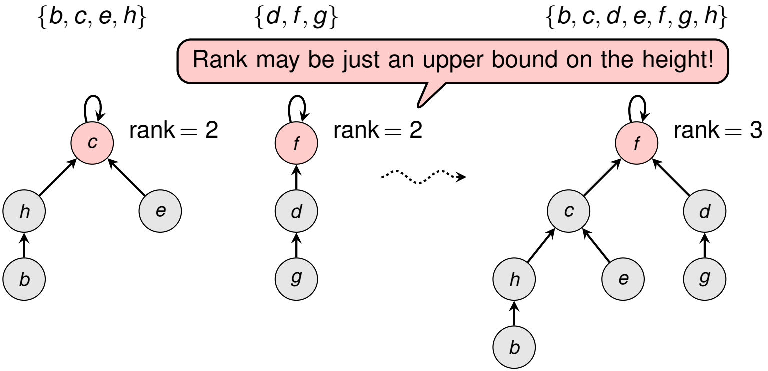


Forest Structure

- 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

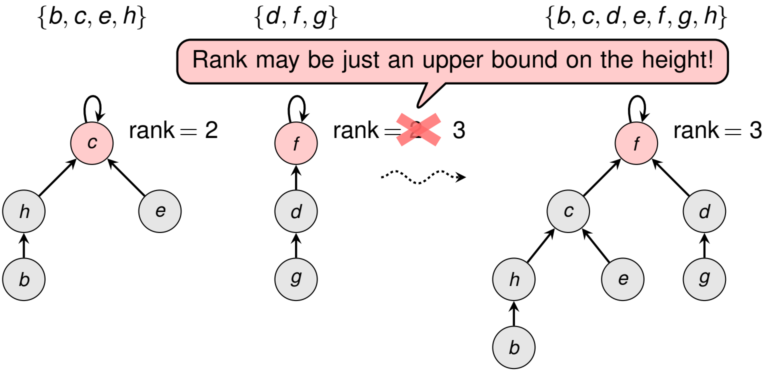


Forest Structure

- 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

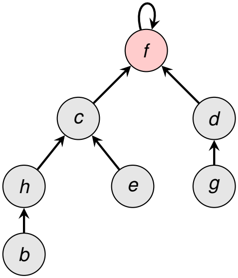


Forest Structure

- 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 $\rightsquigarrow$ Union by Rank

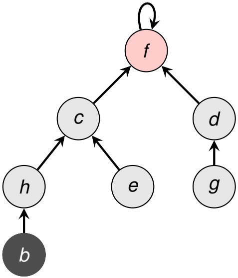
Path Compression during FIND-SET




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

Path Compression during FIND-SET

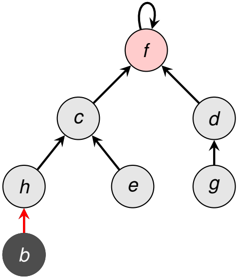
$\boxed{b}$




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

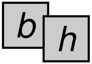
Path Compression during FIND-SET

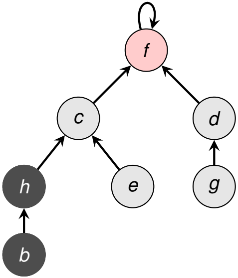
$\boxed{b}$




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

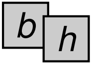
Path Compression during FIND-SET

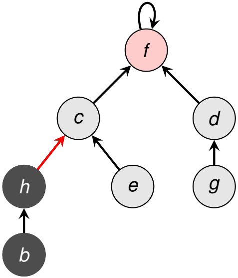





```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

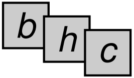
Path Compression during FIND-SET

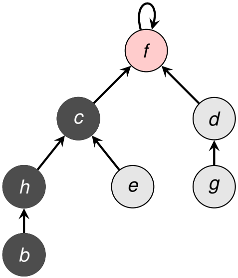





```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

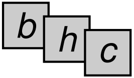
Path Compression during FIND-SET

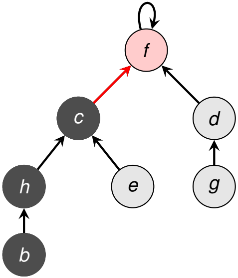





```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

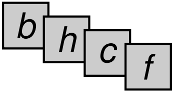
Path Compression during FIND-SET

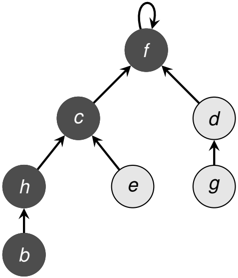





```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

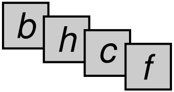
Path Compression during FIND-SET

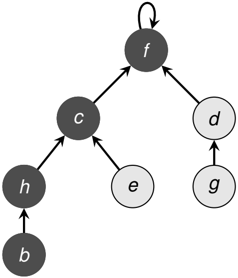





```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

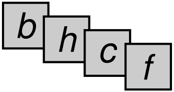
Path Compression during FIND-SET

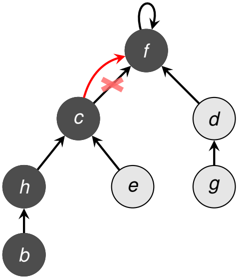





```
FindSet(x)
    if x ≠ x.p
        x.p = FindSet(x.p)
    return x.p
```

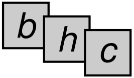
Path Compression during FIND-SET

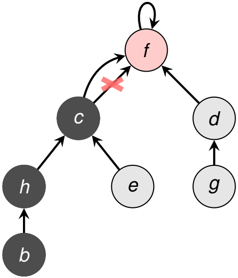





```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

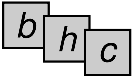
Path Compression during FIND-SET

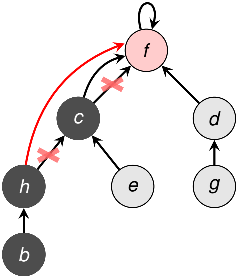





```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

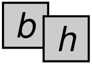
Path Compression during FIND-SET

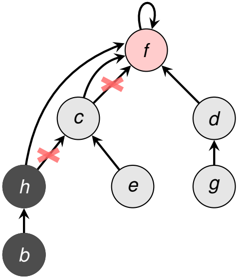





```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

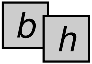
Path Compression during FIND-SET

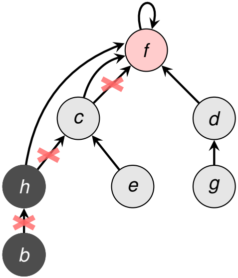





```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FIND-SET

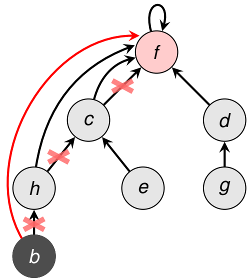





```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FIND-SET

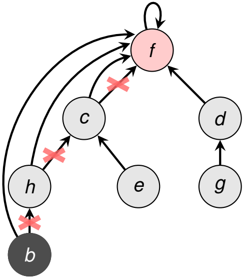
$\boxed{b}$




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

Path Compression during FIND-SET

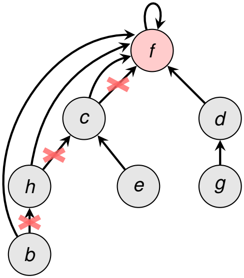
$\boxed{b}$




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

Path Compression during FIND-SET

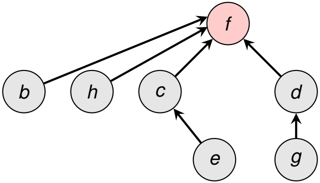
$\boxed{b}$




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

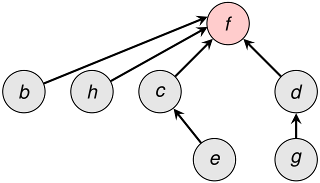
Path Compression during FIND-SET

$\boxed{b}$



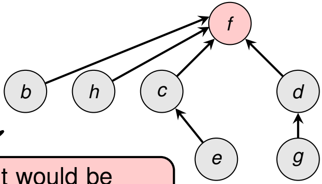

```
FindSet(x)
   if x ≠ x.p
      x.p = FindSet(x.p)
   return x.p
```

Path Compression during FIND-SET




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

Path Compression during FIND-SET



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

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

Combining Union by Rank and Path Compression

Theorem 21.14

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

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

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

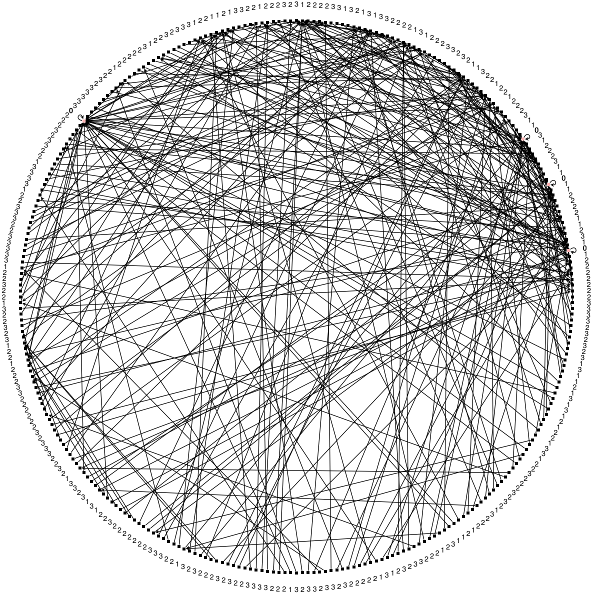
Simulating the Effects of Union by Rank and Path Compression

Simulating the Effects of Union by Rank and Path Compression

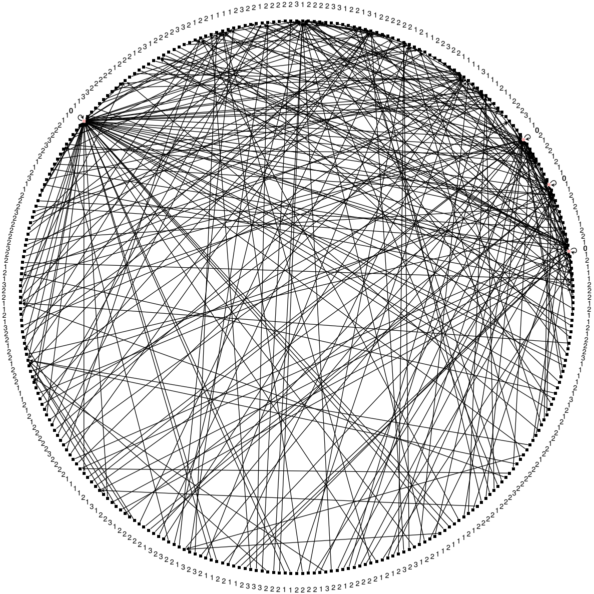
Experimental Setup

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 $\textsc{Union}(\textsc{Find}(i), \textsc{Find}(r))$
3. Perform $j \in \{0, 100, 200, 300, 600, 900\}$ many $\textsc{Find}(r)$, where $1 \leq r \leq 300$ is random

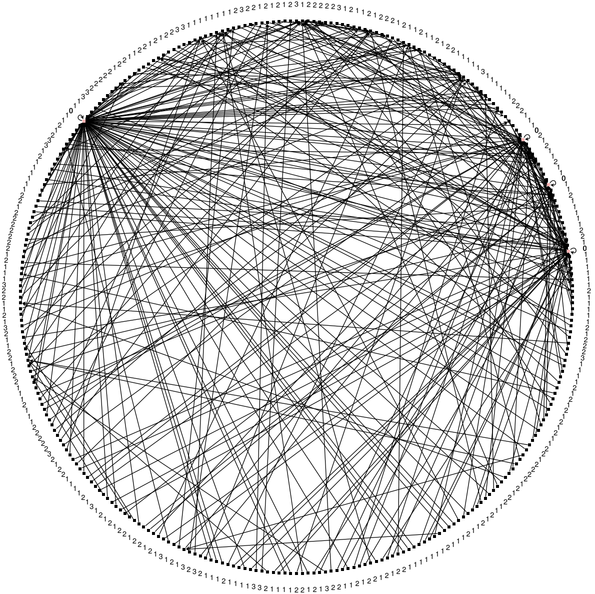
Union by Rank without Path Compression



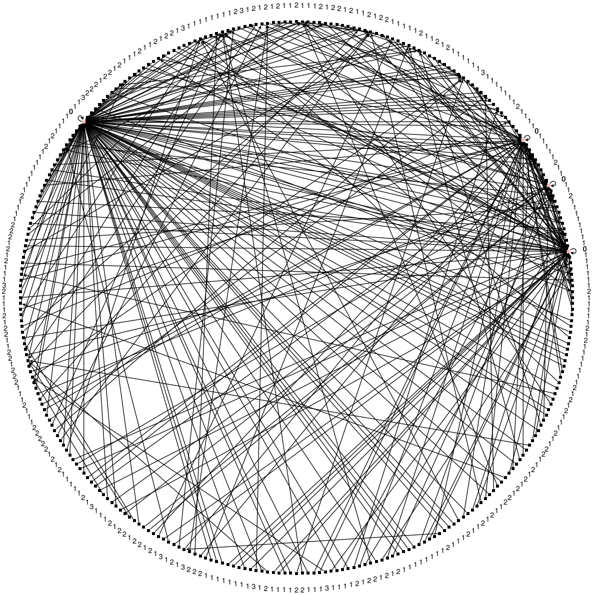
Union by Rank with Path Compression



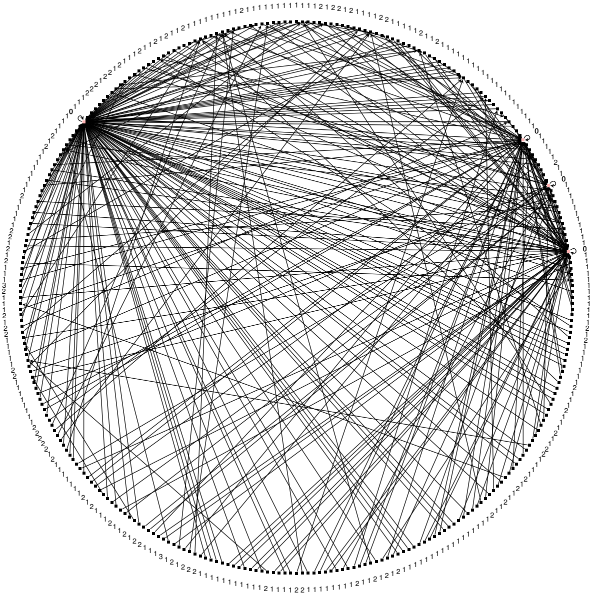
After 100 additional FINDs



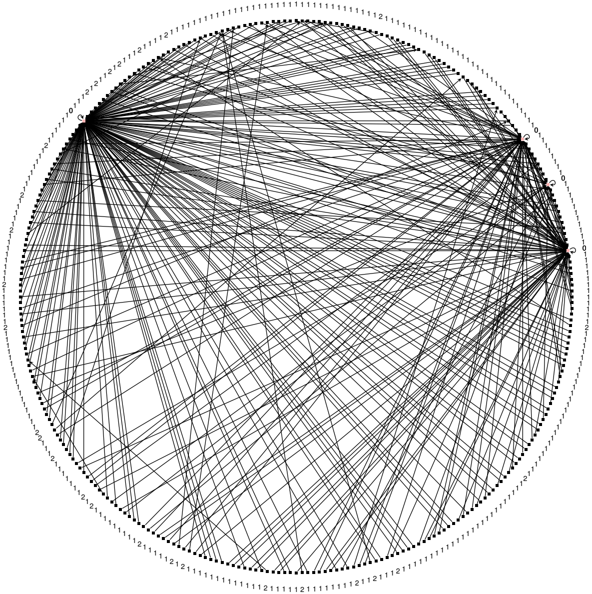
After 200 additional FINDs



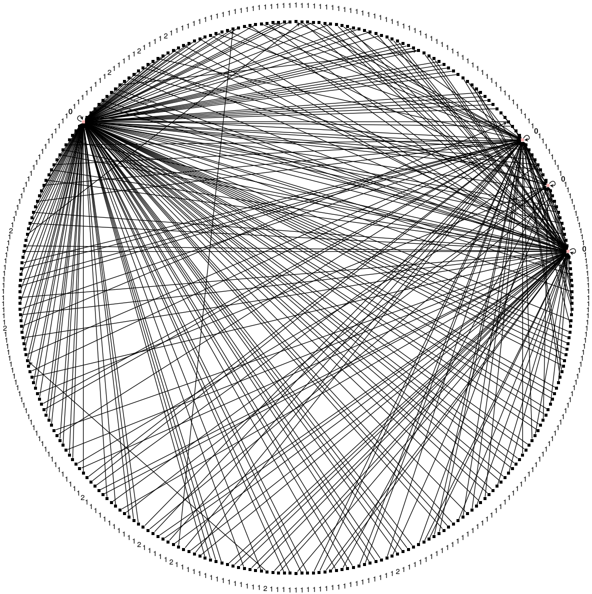
After 300 additional FINDs



After 600 additional FINDs



After 900 additional FINDs

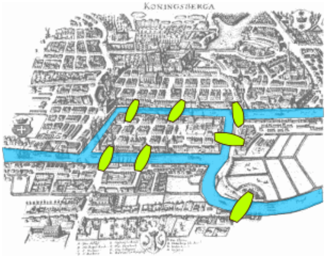


Outline

Disjoint Sets

Introduction to Graphs and Graph Searching

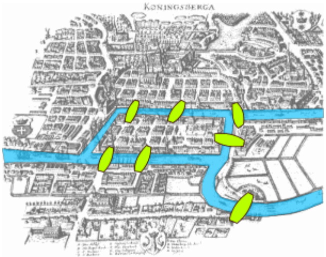
Origin of Graph Theory



Source: Wikipedia

Seven Bridges at Königsberg 1737

Origin of Graph Theory



Source: Wikipedia

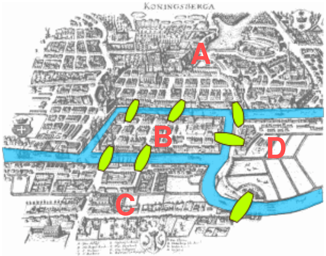
Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?

Origin of Graph Theory



Source: Wikipedia

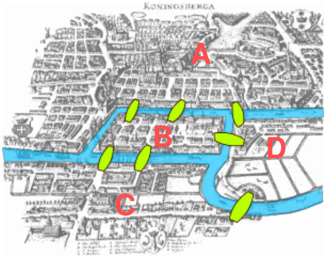
Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?

Origin of Graph Theory



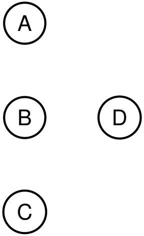
Source: Wikipedia

Seven Bridges at Königsberg 1737

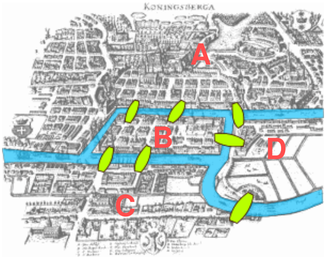
Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?



Origin of Graph Theory



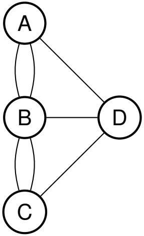
Source: Wikipedia

Seven Bridges at Königsberg 1737

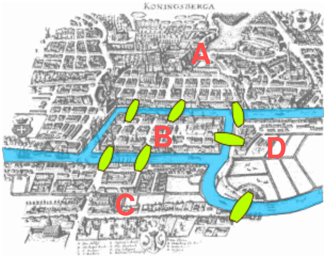
Source: Wikipedia

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge **exactly once**?



Origin of Graph Theory

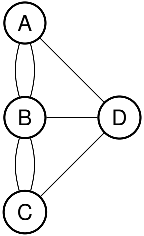


Source: Wikipedia

Seven Bridges at Königsberg 1737

Source: Wikipedia

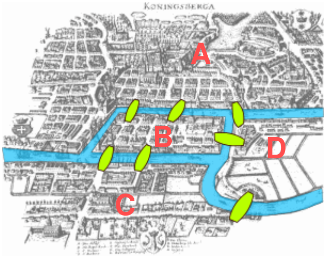
Leonhard Euler (1707-1783)



Is there a tour which crosses each bridge **exactly once**?

Is there a tour which visits every island **exactly once**?

Origin of Graph Theory

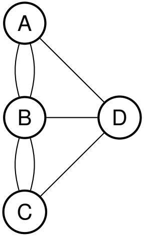


Source: Wikipedia

Seven Bridges at Königsberg 1737

Source: Wikipedia

Leonhard Euler (1707-1783)



Is there a tour which crosses each bridge **exactly once**?

Is there a tour which visits every island **exactly once**?
$\rightsquigarrow$ 1B course: Complexity Theory

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

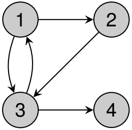
- V: the set of vertices
- E: the set of edges (arcs)

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

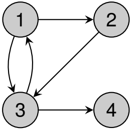


What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)



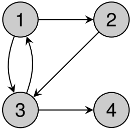
$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

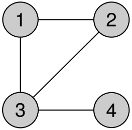


Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

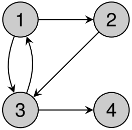


What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

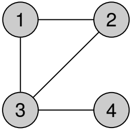


$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



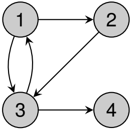
$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

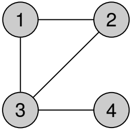


$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

Paths and Connectivity

- A sequence of edges between two vertices forms a path

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

Undirected Graph

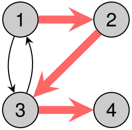
A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

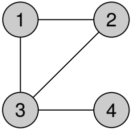
Paths and Connectivity

- A sequence of edges between two vertices forms a path

Path $p = (1, 2, 3, 4)$



$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$



$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

Undirected Graph

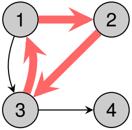
A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

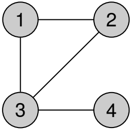
Paths and Connectivity

- A sequence of edges between two vertices forms a path

Path $p = (1, 2, 3, 1)$, which is a cycle



$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$



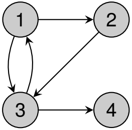
$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

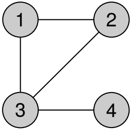


$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

Paths and Connectivity

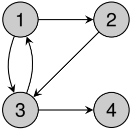
- A sequence of edges between two vertices forms a path

What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

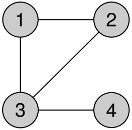


$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

Paths and Connectivity

- A sequence of edges between two vertices forms a path

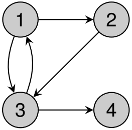
What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

G is not a DAG

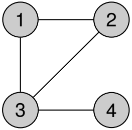


$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

What is a Graph?

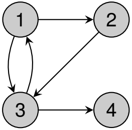
Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

G is not a DAG

G is not (strongly) connected

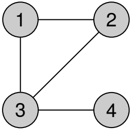


$$V = \{1, 2, 3, 4\}$$
$$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges



G is connected

$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$

Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

What is a Graph?

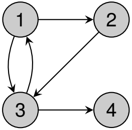
Directed Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of edges (arcs)

G is not a DAG

G is not (strongly) connected



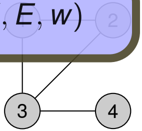
$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$

Undirected Graph

A graph $G = (V, E)$ consists of:

- V: the set of vertices
- E: the set of (undirected) edges

Later: edge-weighted graphs $G = (V, E, w)$



G is connected

$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then G is connected

Representations of Directed and Undirected Graphs

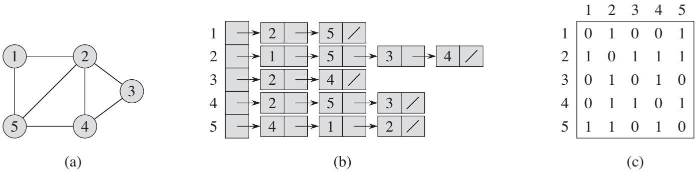
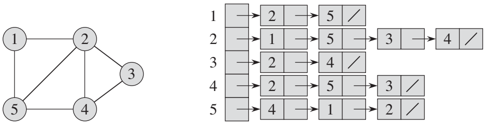


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Representations of Directed and Undirected Graphs

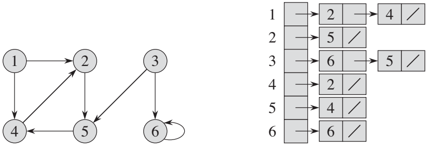


(a) (b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.



(a) (b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Representations of Directed and Undirected Graphs

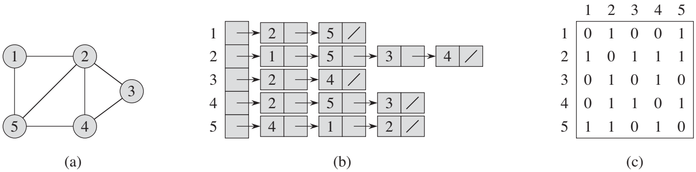


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

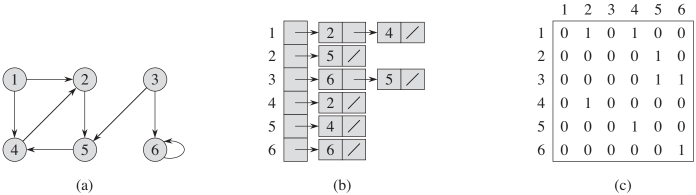


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.