CS473-Algorithms I

Lecture X

Disjoint Set Operations

Disjoint Set Operations

A disjoint-set data structure

- Maintains a collection $S = \{S_1, \ldots, S_k\}$ of disjoint dynamic sets
- Each set is identified by a representative which is some member of the set

In some applications,

- It doesn't matter which member is used as the representative
- We only care that,
  - ➢ if we ask for the representative of a set twice without modifying the set between the requests,
  - ✓ we get the same answer both times

Disjoint Set Operations

In other applications,
There may be a prescribed rule for choosing the representative

E.G. Choosing the smallest member in the set

Each element of a set is represented by an object "*x*"

MAKE-SET(*x*) creates a new set whose only member is *x*

- Object *x* is the representative of the set
- *x* is not already a member of any other set

UNION(*x*, *y*) unites the dynamic sets S_χ & S_y that contain *x* & y

- S_χ & S_y are assumed to be disjoint prior to the operation
- The new representative is some member of $S_\chi \cup S_y$

Disjoint Set Operations

– Usually, the representative of either S_χ OR S_y is chosen as the new representative

We destroy sets S_χ & S_y, removing them from the collection S since we require the sets in the collection to be disjoint

FIND-SET(x) returns a pointer to the representative of the unique set containing x

We will analyze the running times in terms of two parameters

- n : The number of MAKE-SET operations
- m : The total number of MAKE-SET, UNION and FIND-SET operations

Disjoint Set Operations

- Each union operation reduces the number of sets by one since the sets are disjoint
  - Therefore, only one set remains after $n - 1$ union operations
  - Thus, the number of union operations is $\leq n - 1$

- Also note that, $m \geq n$ always hold

  since MAKE-SET operations are included in the total number of operations

An Application of Disjoint-Set Data Structures

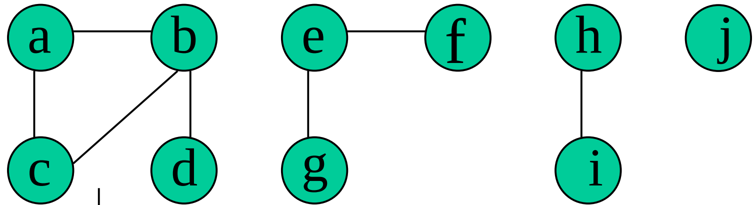
Determining the connected components of an undirected graph G=(V,E)

```
CONNECTED-COMPONENTS (G)
    for each vertex v ∈ V[G] do
        MAKE-SET(v)
    endfor
    for each edge (u, v) ∈ E[G] do
        if  FIND-SET(u) ≠ FIND-SET(v) then
            UNION(u, v)
        endif
    endfor
end
```

```
SAME-COMPONENT(u,v)
        if FIND-SET(u) = FIND-SET(v) then
                return TRUE
        else
                return FALSE
        endif
end
```

An Application of Disjoint-Set Data Structures

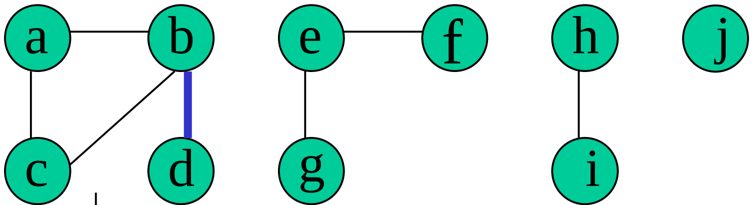
Determining the connected components of an undirected graph G=(V,E)



Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}

An Application of Disjoint-Set Data Structures

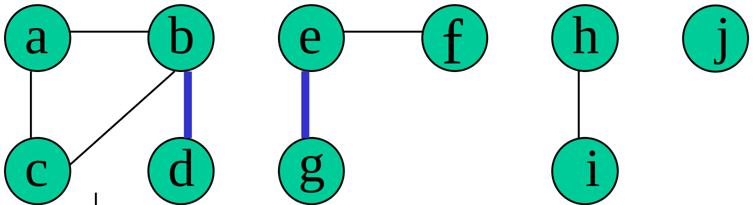
Determining the connected components of an undirected graph G=(V,E)



Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}

An Application of Disjoint-Set Data Structures

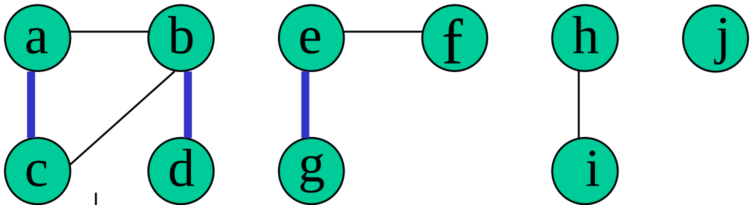
Determining the connected components of an undirected graph G=(V,E)



Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}

An Application of Disjoint-Set Data Structures

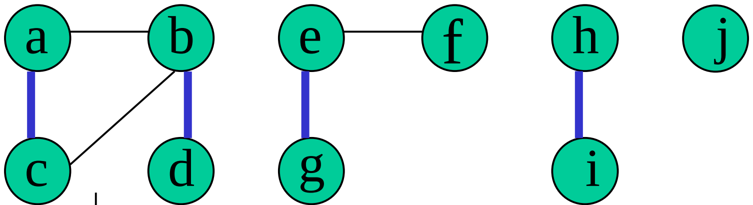
Determining the connected components of an undirected graph G=(V,E)



Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}
(a, c)	{a, c}	{b, d}			{e, g}	{f}		{h}	{i}	{j}

An Application of Disjoint-Set Data Structures

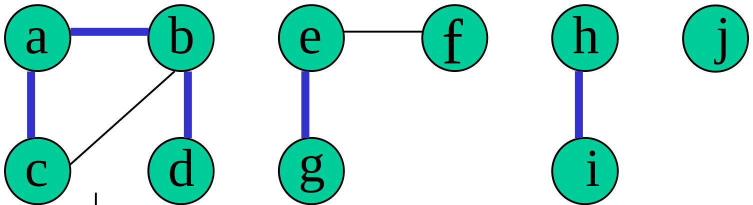
Determining the connected components of an undirected graph G=(V,E)



Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}
(a, c)	{a, c}	{b, d}			{e, g}	{f}		{h}	{i}	{j}
(h, i)	{a, c}	{b, d}			{e, g}	{f}		{h, i}		{j}

An Application of Disjoint-Set Data Structures

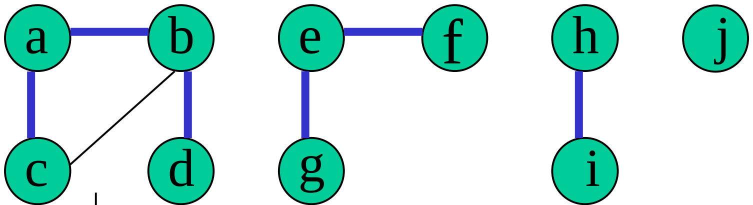
Determining the connected components of an undirected graph G=(V,E)



Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}
(a, c)	{a, c}	{b, d}			{e, g}	{f}		{h}	{i}	{j}
(h, i)	{a, c}	{b, d}			{e, g}	{f}		{h, i}		{j}
(a, b)	{a, b, c, d}				{e, g}	{f}		{h, i}		{j}

An Application of Disjoint-Set Data Structures

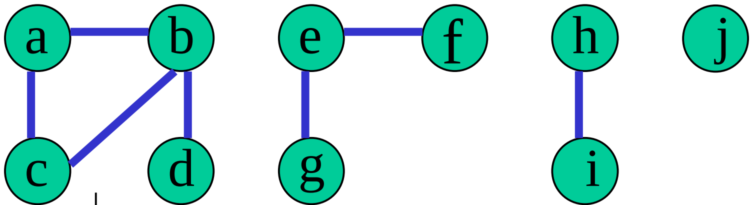
Determining the connected components of an undirected graph G=(V,E)



Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}
(a, c)	{a, c}	{b, d}			{e, g}	{f}		{h}	{i}	{j}
(h, i)	{a, c}	{b, d}			{e, g}	{f}		{h, i}		{j}
(a, b)	{a, b, c, d}				{e, g}	{f}		{h, i}		{j}
(e, f)	{a, b, c, d}				{e, f, g}			{h, i}		{j}

An Application of Disjoint-Set Data Structures

Determining the connected components of an undirected graph G=(V,E)

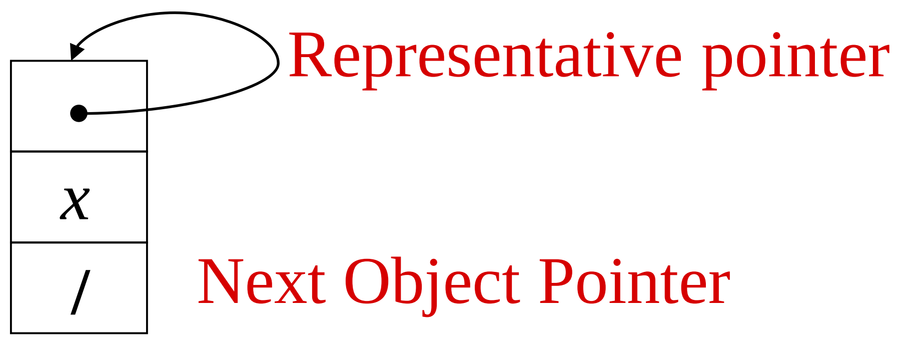


Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}
(a, c)	{a, c}	{b, d}			{e, g}	{f}		{h}	{i}	{j}
(h, i)	{a, c}	{b, d}			{e, g}	{f}		{h, i}		{j}
(a, b)	{a, b, c, d}				{e, g}	{f}		{h, i}		{j}
(e, f)	{a, b, c, d}				{e, f, g}			{h, i}		{j}
(b, c)	{a, b, c, d}				{e, f, g}			{h, i}		{j}

Linked-List Representation of Disjoint Sets

- Represent each set by a linked-list
- The first object in the linked-list serves as its set representative
- Each object in the linked-list contains
  1. A set member
  2. A pointer to the object containing the next set member
  3. A pointer back to the representative

MAKE-SET(x) : $O(1)$

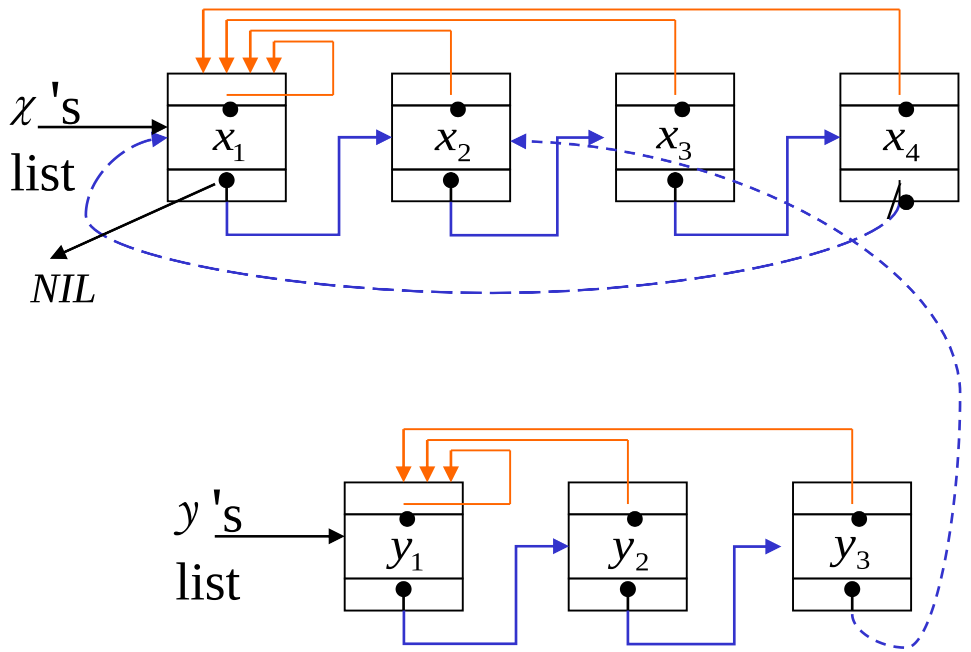


FIND-SET(x) : We return the representative pointer of x

Linked-List Representation of Disjoint Sets

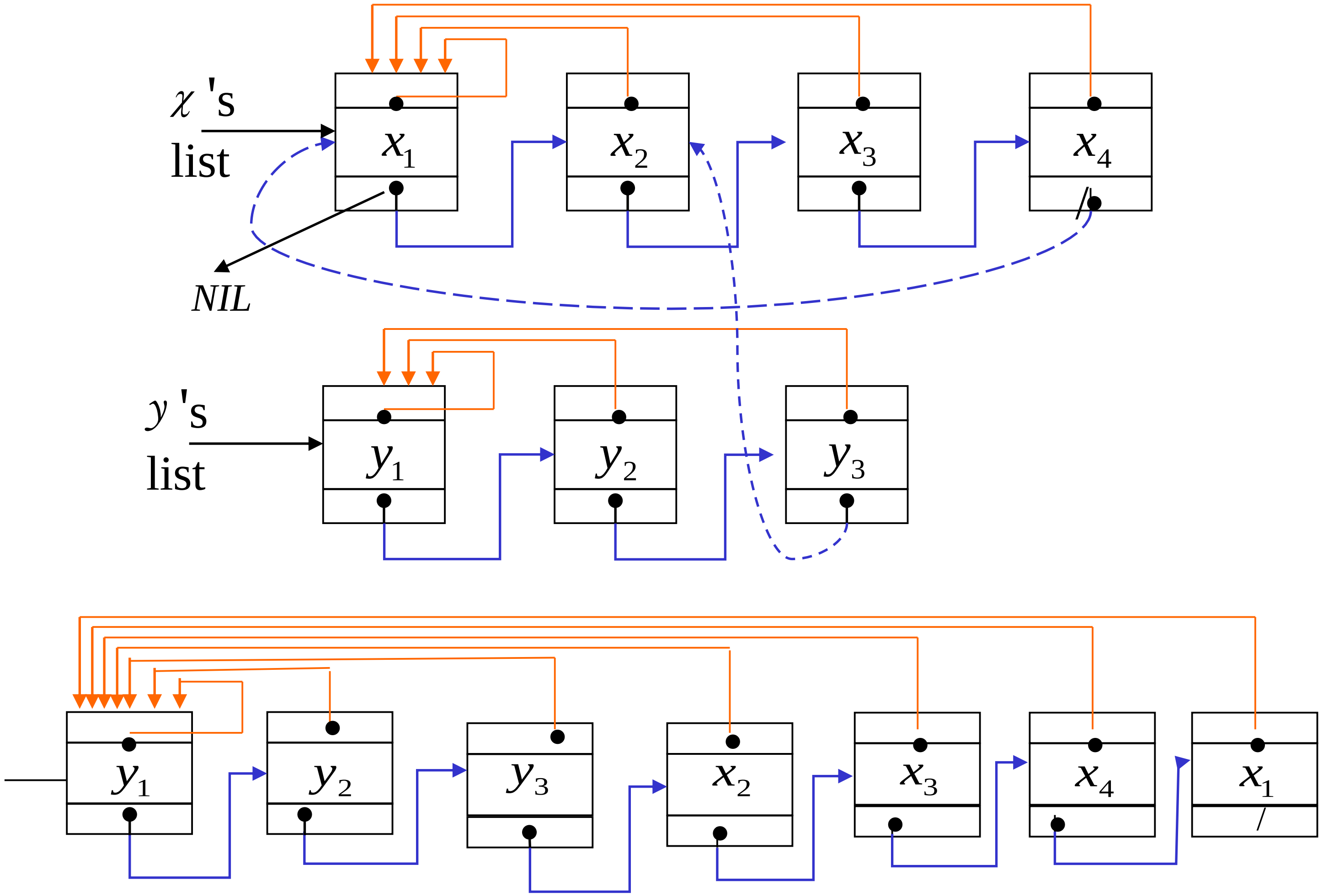
A Simple Implementation of Union : UNION(x, y)

- APPEND x's list to the end of y 's list
- The representative of y 's list becomes the new representative
- UPDATE the representative pointer of each object originally on x's list which takes time linear in the length of x's list



Linked-List Representation of Disjoint Sets

A Simple Implementation of Union : UNION(x, y)



Analysis of the Simple Union Implementation

- A sequence of m operations that requires $\Theta(m^2)$ time
- Suppose that we have n objects $x_1, x_2, \ldots, x_n$ and let $m = 2n - 1$

Operation	Number of Objects Updated	Updated Objects (Denoted By '✓')
MAKE-SET(χ_1)	1	$\{\overset{\checkmark}{\chi_1}\}$

Analysis of the Simple Union Implementation

Operation	Number of Objects Updated	Updated Objects (Denoted By '✓')
MAKE-SET(χ_1)	1	$\{\chi_1^{\checkmark}\}$
MAKE-SET(χ_2)	1	$\{\chi_2^{\checkmark}\}$

Analysis of the Simple Union Implementation

Operation	Number of Objects Updated	Updated Objects (Denoted By '✓')
MAKE-SET(χ_1)	1	$\{\chi_1^{✓}\}$
MAKE-SET(χ_2)	1	$\{\chi_2^{✓}\}$
.	.	
.	.	
.	.	

Analysis of the Simple Union Implementation

Operation	Number of Objects Updated	Updated Objects (Denoted By '✓')
MAKE-SET(χ_1)	1	$\{\chi_1^{\checkmark}\}$
MAKE-SET(χ_2)	1	$\{\chi_2^{\checkmark}\}$
.	.	
.	.	
.	.	
MAKE-SET(χ_n)	1	$\{\chi_n^{\checkmark}\}$

Analysis of the Simple Union Implementation

Operation	Number of Objects Updated	Updated Objects (Denoted By '✓')
MAKE-SET(χ_1)	1	$\{\chi_1\}$ ✓
MAKE-SET(χ_2)	1	$\{\chi_2\}$ ✓
.	.	
.	.	
.	.	
MAKE-SET(χ_n)	1	$\{\chi_n\}$ ✓
UNION($\chi_{1,}$ χ_2)	1	$\{\chi_1\} \cup \{\chi_2\} \rightarrow \{\chi_{1,}\ \chi_2\}$ ✓

Analysis of the Simple Union Implementation

Operation	Number of Objects Updated	Updated Objects (Denoted By '✓')
MAKE-SET(χ_1)	1	$\{\overset{\checkmark}{\chi_1}\}$
MAKE-SET(χ_2)	1	$\{\overset{\checkmark}{\chi_2}\}$
.	.	
.	.	
.	.	
MAKE-SET(χ_n)	1	$\{\overset{\checkmark}{\chi_n}\}$
UNION(χ_1, χ_2)	1	$\{\chi_1\} \cup \{\chi_2\} \rightarrow \{\overset{\checkmark}{\chi_1}, \chi_2\}$
UNION(χ_2, χ_3)	2	$\{\chi_1, \chi_2\} \cup \{\chi_3\} \longrightarrow \{\overset{\checkmark}{\chi_1}, \overset{\checkmark}{\chi_2}, \chi_3\}$

Analysis of the Simple Union Implementation

Operation	Number of Objects Updated	Updated Objects (Denoted By '✓')
MAKE-SET(χ_1)	1	$\{\overset{\checkmark}{\chi_1}\}$
MAKE-SET(χ_2)	1	$\{\overset{\checkmark}{\chi_2}\}$
.	.	
.	.	
.	.	
MAKE-SET(χ_n)	1	$\{\overset{\checkmark}{\chi_n}\}$
UNION(χ_1, χ_2)	1	$\{\chi_1\} \cup \{\chi_2\} \rightarrow \{\overset{\checkmark}{\chi_1}, \chi_2\}$
UNION(χ_2, χ_3)	2	$\{\chi_1, \chi_2\} \cup \{\chi_3\} \rightarrow \{\overset{\checkmark}{\chi_1}, \overset{\checkmark}{\chi_2}, \chi_3\}$
UNION(χ_3, χ_4)	3	$\{\chi_1, \chi_2, \chi_3\} \cup \{\chi_4\} \rightarrow \{\overset{\checkmark}{\chi_1}, \overset{\checkmark}{\chi_2}, \overset{\checkmark}{\chi_3}, \chi_4\}$

Analysis of the Simple Union Implementation

Operation	Number of Objects Updated	Updated Objects (Denoted By '✓')
MAKE-SET(χ_1)	1	$\{\overset{\checkmark}{\chi_1}\}$
MAKE-SET(χ_2)	1	$\{\overset{\checkmark}{\chi_2}\}$
.	.	
.	.	
.	.	
MAKE-SET(χ_n)	1	$\{\overset{\checkmark}{\chi_n}\}$
UNION($\chi_{1,}\ \chi_2$)	1	$\{\chi_1\} \cup \{\chi_2\} \rightarrow \{\overset{\checkmark}{\chi_1}, \chi_2\}$
UNION($\chi_{2,}\ \chi_3$)	2	$\{\chi_1, \chi_2\} \cup \{\chi_3\} \rightarrow \{\overset{\checkmark}{\chi_1}, \overset{\checkmark}{\chi_2}, \chi_3\}$
UNION($\chi_{3,}\ \chi_4$)	3	$\{\chi_1, \chi_2, \chi_3\} \cup \{\chi_4\} \rightarrow \{\overset{\checkmark}{\chi_1}, \overset{\checkmark}{\chi_2}, \overset{\checkmark}{\chi_3}, \chi_4\}$
.	.	
.	.	

Analysis of the Simple Union Implementation

Operation	Number of Objects Updated	Updated Objects (Denoted By '$\checkmark$')
MAKE-SET(χ_1)	1	$\{\overset{\checkmark}{\chi_1}\}$
MAKE-SET(χ_2)	1	$\{\overset{\checkmark}{\chi_2}\}$
$\vdots$	$\vdots$	
MAKE-SET(χ_n)	1	$\{\overset{\checkmark}{\chi_n}\}$
UNION(χ_1, χ_2)	1	$\{\chi_1\} \cup \{\chi_2\} \rightarrow \{\overset{\checkmark}{\chi_1}, \chi_2\}$
UNION(χ_2, χ_3)	2	$\{\chi_1, \chi_2\} \cup \{\chi_3\} \rightarrow \{\overset{\checkmark}{\chi_1}, \overset{\checkmark}{\chi_2}, \chi_3\}$
UNION(χ_3, χ_4)	3	$\{\chi_1, \chi_2, \chi_3\} \cup \{\chi_4\} \rightarrow \{\overset{\checkmark}{\chi_1}, \overset{\checkmark}{\chi_2}, \overset{\checkmark}{\chi_3}, \chi_4\}$
$\vdots$	$\vdots$	
UNION(χ_{n-1}, χ_n)	n - 1	$\{\chi_1, \chi_2, \ldots, \chi_{n-1}\} \cup \{\chi_n\} \rightarrow \{\overset{\checkmark}{\chi_1}, \overset{\checkmark}{\chi_2}, \ldots, \overset{\checkmark}{\chi_{n-1}}, \chi_n\}$

Analysis of the Simple Union Implementation

- The total number of representative pointer updates

$$= \underbrace{n}_{\text{MAKE-SET operations}} + \underbrace{\sum_{i=1}^{n-1} i}_{\text{UNION operations}} = n + \frac{1}{2}(n-1)n = \frac{1}{2}n^2 + \frac{1}{2}n = \Theta(n^2)$$

MAKE-SET operations

UNION operations

$$= \Theta(m^2) \quad \text{since} \quad n = \lceil m/2 \rceil$$

- Thus, on the average, each operation requires $\Theta(m)$ time
- That is, the amortized time of an operation is $\Theta(m)$

A Weighted-Union Heuristic

- The simple implementation is inefficient because
  - ➢ We may be appending a longer list to a shorter list during a UNION operation

    so that we must update the representative pointer of each member of the longer list

Weighted Union Heuristic

– Maintain the length of each list

– Always append the smaller list to the longer list

With ties broken arbitrarily

!! A single UNION can still take $\Omega(m)$ time if both sets have $\Omega(m)$ members

Weighted Union Heuristic

Theorem: A sequence of m MAKE-SET, UNION & FIND-SET operations, n of which are MAKE-SET operations, takes $O(m+n\lg n)$ time

Proof: Try to compute an upper bound on the number of representative pointer updates for each object in a set of size n

Consider a fixed object x

– Each time x's R-PTR was updated, x was a member of the smaller set

$\{x\} \cup \{v\} \rightarrow \{x, v\}$ 1-st update $|S_x| \geq 2$

$\{x, v\} \cup \{w_1, w_2\} \rightarrow \{x, v, w_1, w_2\}$ 2-nd update $|S_x| \geq 4$

$\{x, v, w_1, w_2\} \bigcup \{z_1, z_2, z_3, z_4\} \rightarrow \{x, v, w_1, w_2, z_1, z_2, z_3, z_4\}; |S_x| \geq 4$

3-rd update $|S| \geq 8$

Weighted Union Heuristic

- For any k $\leq$ n, after *x*'s R-PTR has been updated $\lceil lg\, k \rceil$ times the resulting set must have at least *k* members

➢ R-PTR of each object can be updated at most $\lceil lg\, n \rceil$ time over all UNION operations

Analysis of The Weighted-Union Heuristic

- The figure below illustrates a worst case sequence for a set with *n* = 16 objects
- The total number of R-PTR updates

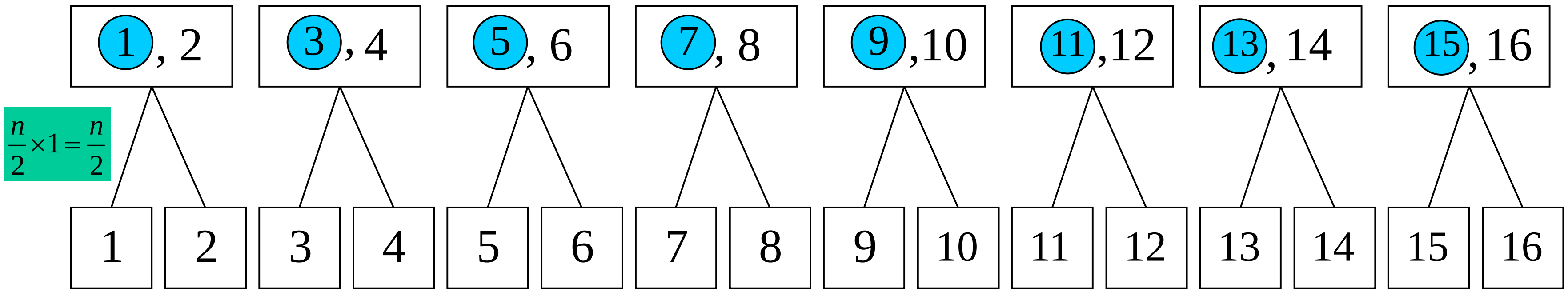
$$= \frac{16}{2} \times 1 + \frac{16}{4} \times 2 + \frac{16}{8} \times 4 + \frac{16}{16} \times 8 = 8 \times 1 + 4 \times 2 + 2 \times 4 + 1 \times 8 = 8 \times 4 = 32$$

$$= \underbrace{\frac{n}{2} + \frac{n}{2} + \ldots.. + \frac{n}{2}}_{\lg n} = \frac{n}{2} lg\, n = O(n\, lg\, n)$$

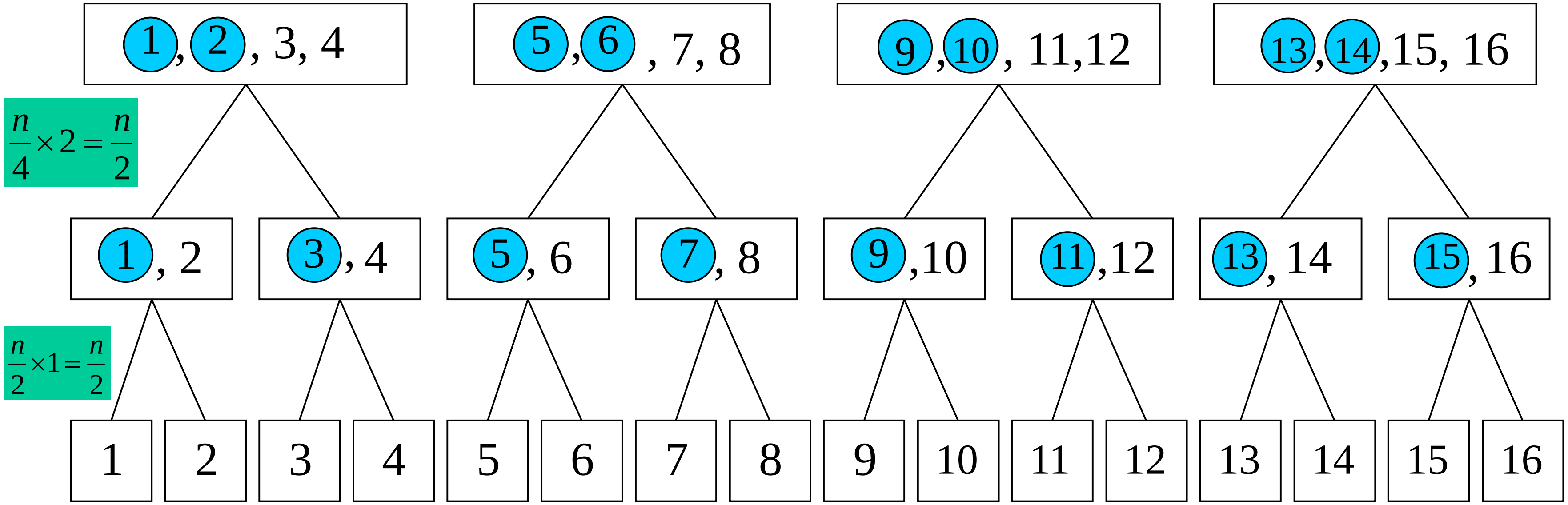
Analysis of The Weighted-Union Heuristic

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

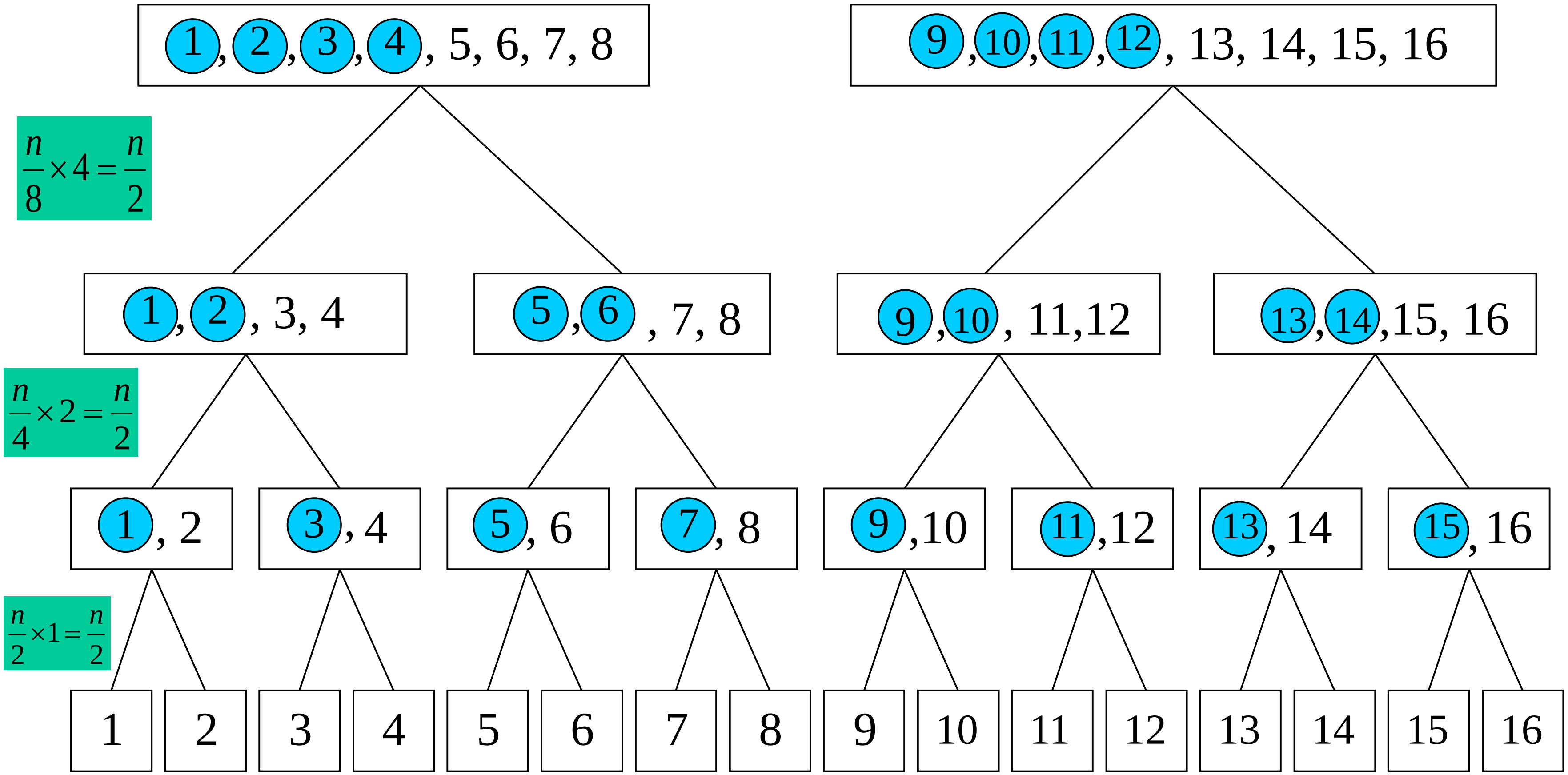
Analysis of The Weighted-Union Heuristic



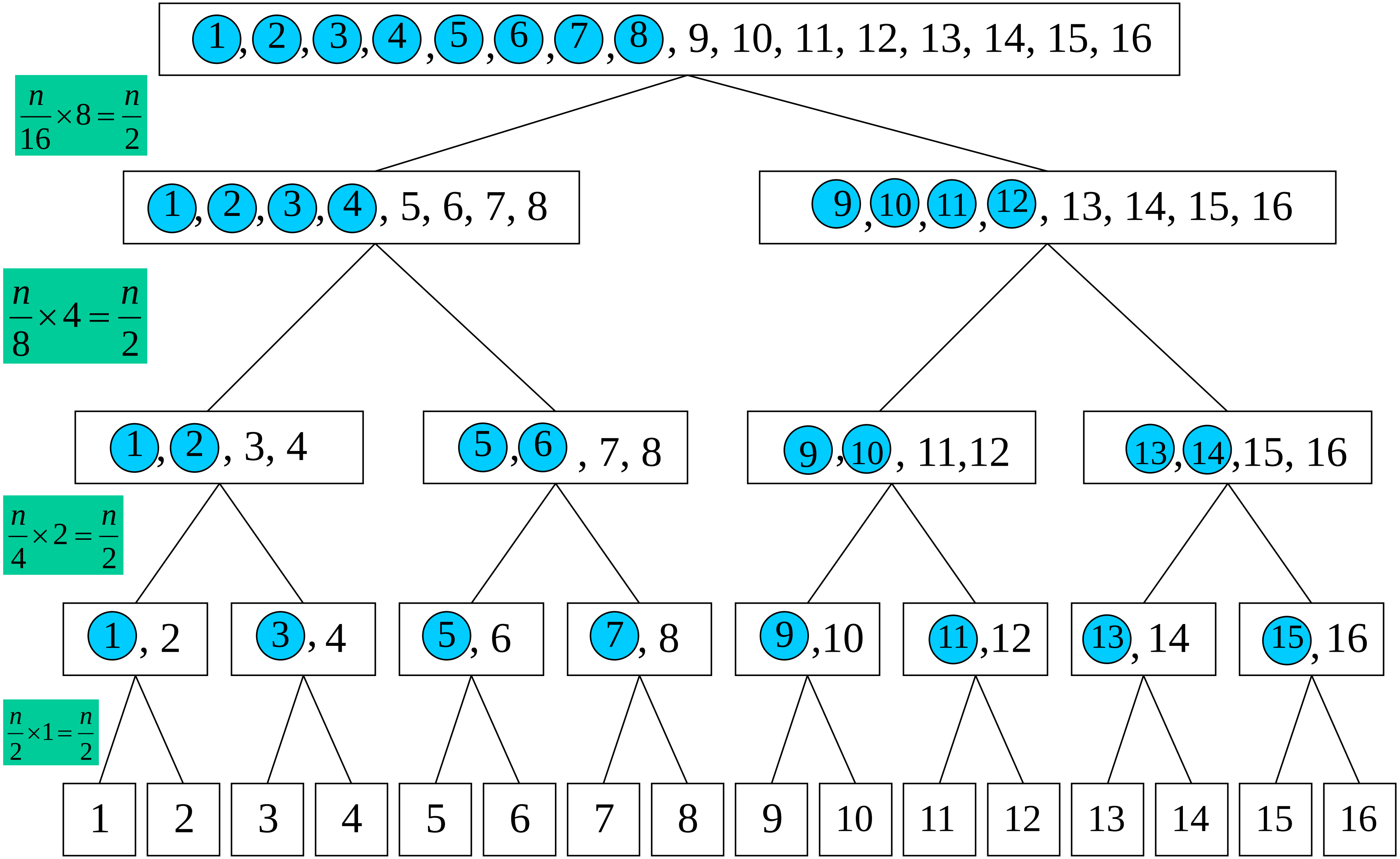
Analysis of The Weighted-Union Heuristic



Analysis of The Weighted-Union Heuristic



Analysis of The Weighted-Union Heuristic



Analysis of The Weighted-Union Heuristic

- Each MAKE-SET & FIND-SET operation takes $O(1)$ time, and there are $O(m)$ of them

➢ The total time for the entire sequence

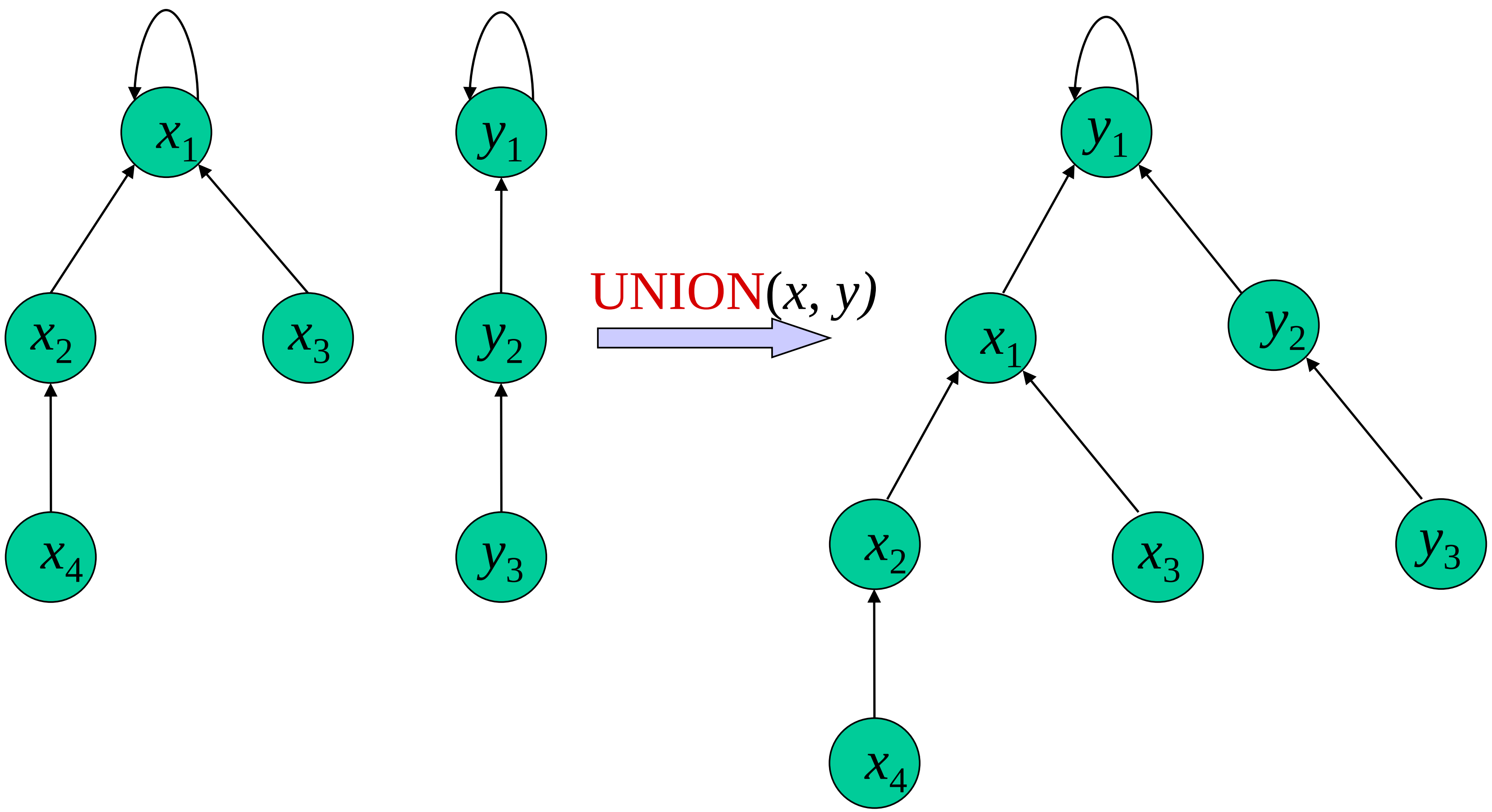
$= O(m + nlgn)$

Disjoint Set Forests

In a faster implementation, we represent sets by rooted trees

- Each node contains one member
- Each tree represents one set
- Each member points only to its parent
- The root of each tree contains the representative
- Each root is its own parent

Disjoint Set Forests



Disjoint Set Forests

Straightforward Implementation

MAKE-SET : Simply creates a tree with just one node : $O(1)$
FIND-SET : Follows parent pointers until the root node is found
The nodes visited on this path toward the root constitute the FIND-PATH

UNION : Makes the root of one tree to point to the other one

Heuristics To Improve the Running Time

- Straightforward implementation is no faster than ones that use the linked-list representation
- A sequence of $n - 1$ UNIONs, following a sequence of n MAKE-SETs, may create a tree, which is just a linear chain of n nodes

Heuristics To Improve the Running Time

First Heuristic : UNION by Rank

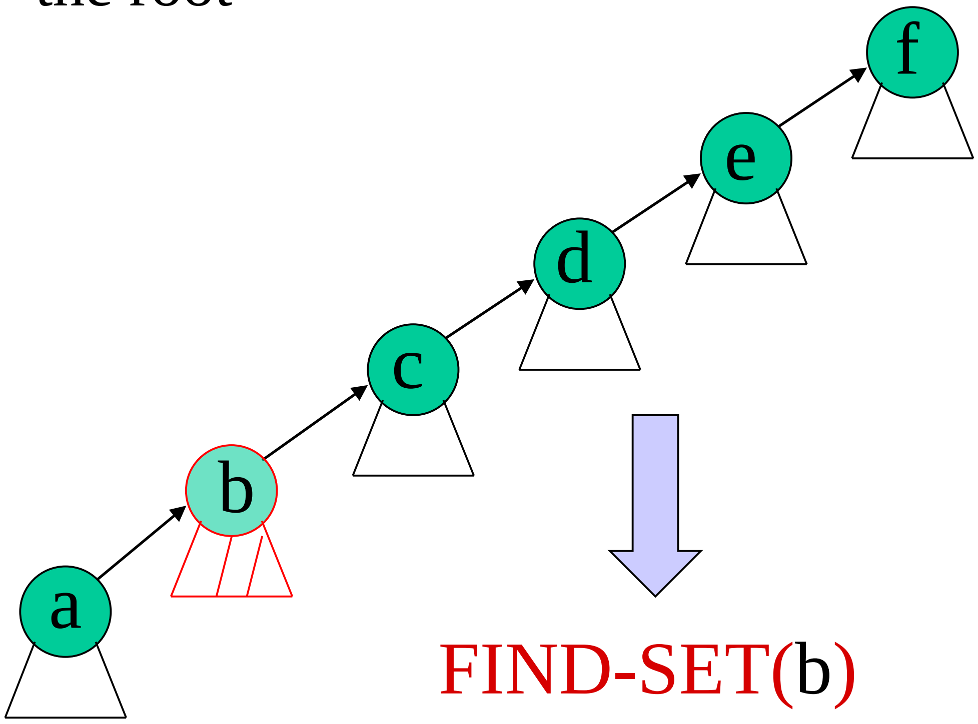
- Similar to the weighted-union used for the linked-list representation
- The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes
- Rather than explicitly keeping the size of the subtree rooted at each node

  We maintain a rank
  - that approximates the logarithm of the subtree size
  - and is also an upperbound on the height of the node
- During a UNION operation
  - make the root with smaller rank to point to the root with larger rank

Heuristics To Improve the Running Time

Second Heuristic : Path Compression

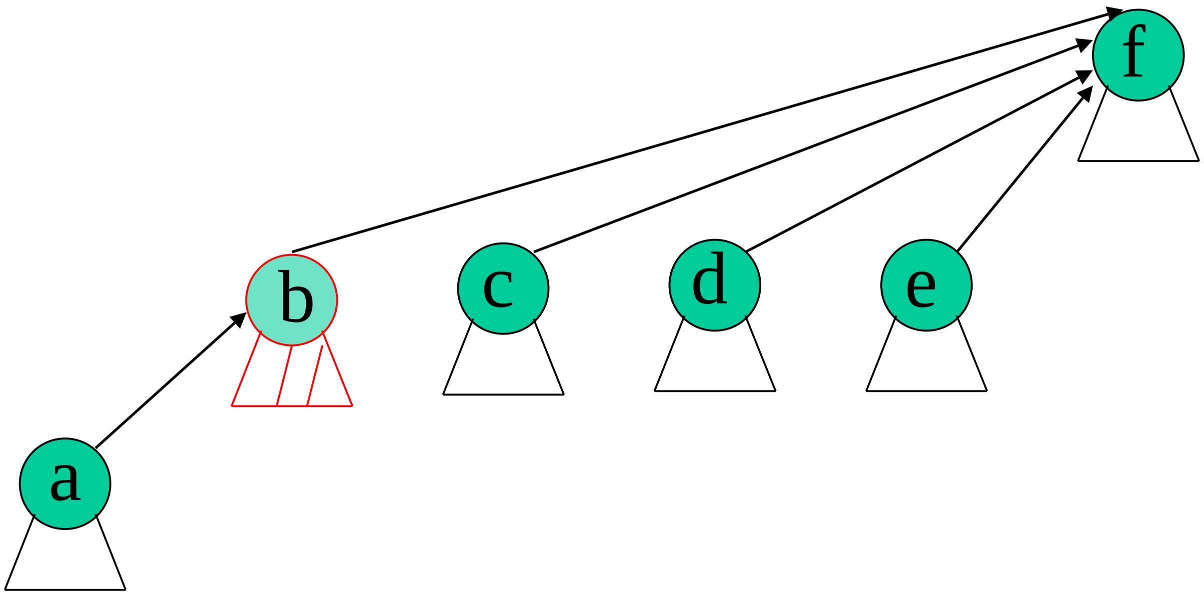
- Use it during the FIND-SET operations
- Make each node on the FIND-PATH to point directly to the root



FIND-SET(b)

Heuristics To Improve the Running Time

Path Compression During FIND-SET(b) Operation



Pseudocodes For the Heuristics

Implementation of UNION-BY-RANK Heuristic

p[x] : Pointer to the parent of the node x

rank[x] : An upperbound on the height of node x in the tree

```
MAKE-SET(x)
        p[x] ← x
        rank[x] ← 0
end
```

```
UNION(x,y)
    LINK(FIND-SET(x),FIND-SET(y))
end
```

```
LINK(x,y)
    if   rank[x] > rank[y] then
         p[y] ← x
    else
         p[x] ← y
         if   rank[x] = rank[y]  then
              rank[y] = rank[y] + 1
         endif
    endif
end
```

Implementation of UNION-BY-RANK Heuristic

– When a singleton set is created by a MAKE-SET
the initial rank of the single node in the tree is zero

– Each FIND-SET operation leaves all ranks unchanged

– When applying a UNION to two trees,
we make the root of tree with higher rank
the parent of the root of lower rank

Ties are broken arbitrarily

Implementation of the Path-Compression Heuristic

The FIND-SET procedure with Path-Compression

Iterative Version

```
FIND-SET(x)
        y ← x
        while   y ≠ p[y]  do
                y ← p[y]
        endwhile
        root ← y
        while   x ≠ p[x]  do
                parent ← p[x]
                p[x] ← root
                x ← parent
        endwhile
        return root
end
```

Recursive Version

```
FIND-SET(x)
        if   x ≠ p[x]  then
                p[x] ← FIND-SET(p[x])
        endif
        return p[x]
end
```