EECS 4101/5101

Prof. Andy Mirzaian

Disjoint Set Union

References:

- [CLRS] chapter 21
- Lecture Note 6

Disjoint Set Union

- Items are drawn from the finite universe U = {1, 2, …, n} for some fixed n. Maintain a partition of (a subset of) U, as a collection of disjoint sets. Uniquely name each set by one of its items called its representative item.

- These disjoint sets are maintained under the following operations:

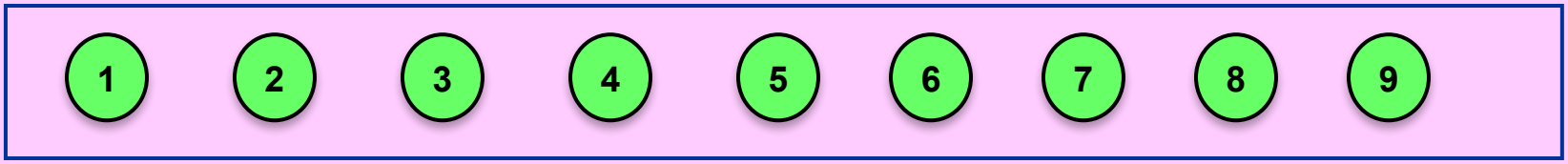
MakeSet(x):
Given item x∈U currently not belonging to any set in the collection, create a new singleton set {x}. Name this set x.
[This is usually done at start, once per item, to create the initial trivial partition.]

Union(A,B):
Change the current partition by replacing its sets A and B with A∪B. Name the new set A or B.
[The operation may choose either one of the two reps as the new rep.]

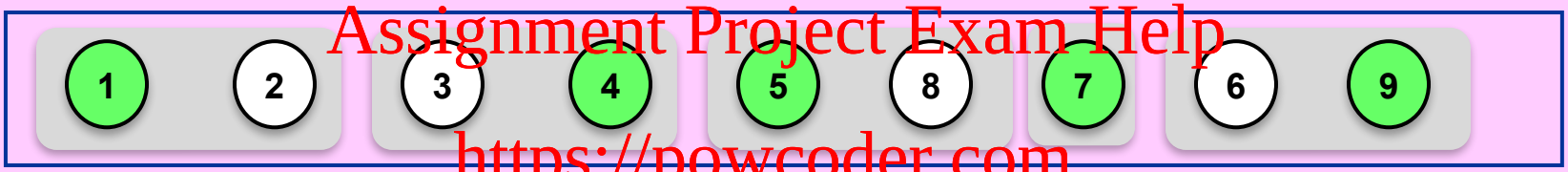
Find(x):
Return the name of the set that currently contains item x.

Example

for x ←1..9 **do** MakeSet(x)

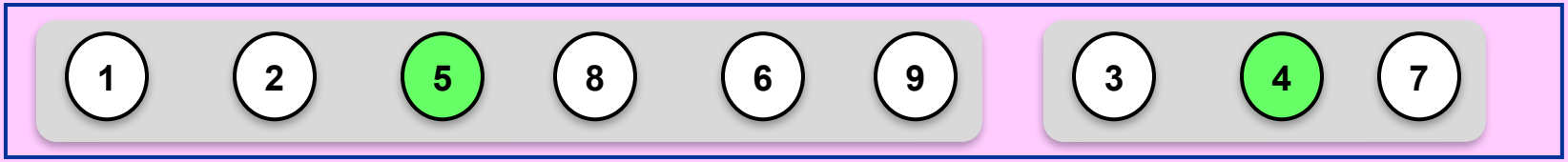


Union(1,2); Union(3,4); Union(5,8); Union(6,9)



Union(1,5); Union(7,4)

Find(1): returns 5; Find(9): returns 9; Union(5,9)



Find(9): retruns 5.

Union-Find Problem

PROBLEM:

s = an on-line sequence of m = |s| MakeSet, Union, and Find operations (intermixed in arbitrary order), n of which are MakeSet, at most n–1 are Union, and the rest are Finds.

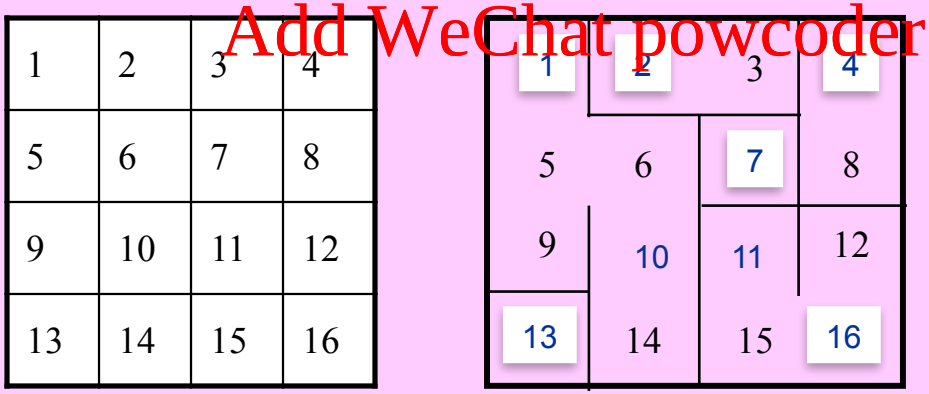
Cost(s) = total computation time to execute sequence s.

Goal: find an implementation that, for every m and n, minimizes the amortized cost per operation:

$\max_s$ Cost(s)/|s|.

Applications

1. Maintaining partitions and equivalence classes.
2. Graph connectivity under edge insertion.
3. Minimum Spanning Trees (e.g., Kruskal's algorithm).
4. Random maze construction.
5. More applications as exercises at the end of this Slide.

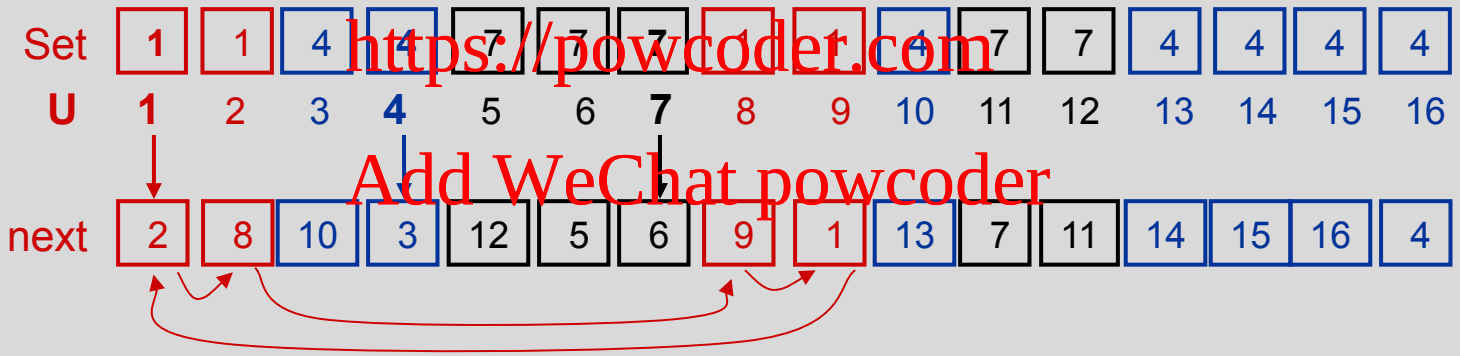


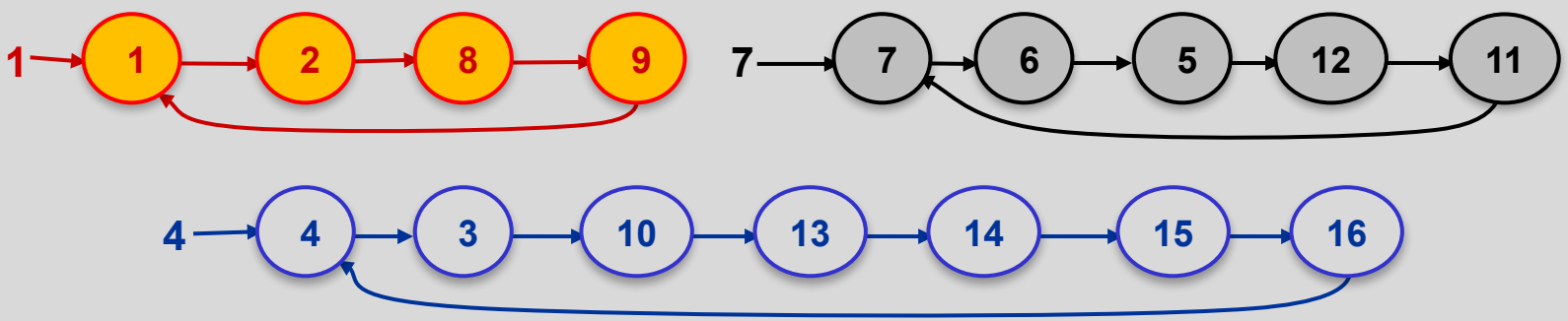
Implementation1: Circular Lists

Data Structure: 2 arrays Set[1..n], next[1..n] (each maps to 1..n)

Set[x] = name of the set that contains item x.
A is a set ⇔ Set[A] = A
next[x] = next item on the list of the set that contains item x.

n = 16, Partition { {1, 2, 8, 9}, {4, 3, 10, 13, 14, 15, 16}, {7, 6, 5, 11, 12} }





Implementation1: Operations & Cost

MakeSet (x) **O(1) time**

```
    Set[x] ← x
    next[x] ← x
end
```

Find (x) **O(1) time**

```
    return Set[x]
end
```

Sequence s:

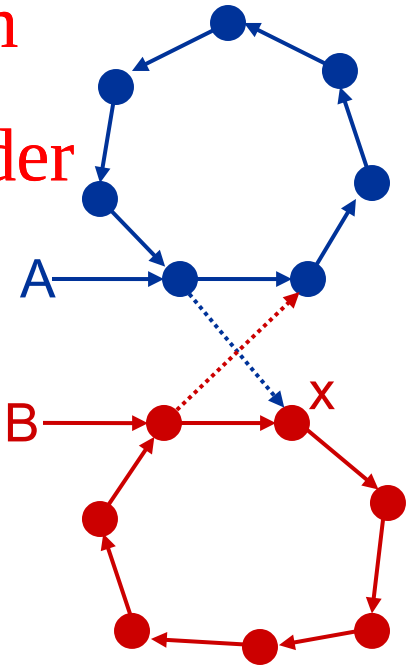
```
for x ← 1 .. n    do  MakeSet(x)
for x ← 1 .. n-1  do  Union1(x+1,x)
```

Aggregate Time = Θ(n²)

Amortized Time per operation = Θ(n)

Union1 (A,B) **O(|B|) time**

```
    (* Set[A]=A ≠ Set[B]=B *)
    (* move all items from set B into set A *)
    Set[B] ← A
    x ← next[B]
    while  x ≠ B  do
          Set[x] ← A          (*rename set B to A*)
          x ← next[x]
    end-while
    x ← next[B]               (*splice lists A & B*)
    next[B] ← next[A]
    next[A] ← x
end
```



Implementation2: Weighted Lists

Data Structure: 3 arrays Set [1..n], next[1..n], size[1..n].

size[A] = # items in set A if A=Set[A] (Otherwise, don't care.)

MakeSet (x) **O(1) time**

```
Set[x] ← x
next[x] ← x
size[x] ← 1
end
```

Find (x) **O(1) time**

```
return Set[x]
end
```

Sequence s:

```
for x ← 1 .. n   do MakeSet (x)
for x ← 1 .. n-1 do Union(x+1,x)
```

Aggregate Time = Θ(n)

This is not the worst sequence! See next page.

Union (A,B) **O(min{|A|, |B|}) time**

```
(* Set[A]=A ≠ Set[B]=B *)
(* Weight-Balanced Union:  merge smaller set into larger set *)
if  size[A] > size[B]
    then do  size[A] ← size[A] + size[B]  ;  Union1 (A,B)   end-then
    else do  size[B] ← size[A] + size[B]  ;  Union1 (B,A)   end-else
end
```

Implementation2: Worst sequence

Sequence s: MakeSet(x), for x=1..n. Then do n-1 Unions in round-robin manner.

Within each round, the sets have roughly equal size.
Starting round: each set has size 1.
Next round: each size 2.
Next: … size 4,
…

Aggregate Time = Θ(n log n)

Amortized time per operation = Θ(log n). **We claim this is the worst.**

Example: n = 16.

Round 0: {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16}

Round 1: {1, 2} {3, 4} {5, 6} {7, 8} {9, 10} {11, 12} {13, 14} {15, 16}

Round 2: {1, 2, 3, 4} {5, 6, 7, 8} {9, 10, 11, 12} {13, 14, 15, 16}

Round 3: {1, 2, 3, 4, 5, 6, 7, 8} {9, 10, 11, 12, 13, 14, 15, 16}

Round 4: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

Implementation2: Amortized Costs

CLAIM 1: Amortized time per operation is O(log n).

Aggregate Method:

- O(1) per MakeSet and Find. O(m) aggregate on MakeSets and Finds.
- O(min{ |A|, |B| }) for Union(A,B). O(n log n) aggregate on Unions. Why?
  - Each item is born in a singleton.
  - O(min{ |A|, |B| }) cost of Union(A,B) is charged O(1) per item moved.
  - Each time an item moves, its set size at least doubles ($|A| \le |B| \Rightarrow 2|A| \le |A \cup B|$).
  - So, an item can move at most log n times (by all Unions).
  - Total charge over the n items is O(n log n).
- Aggregate cost O(m+ n log n). Amortized cost per op = O(log n).

Accounting Method:

Credit Invariant: Total stored credit is $\$ \sum_S |S| \log (n / |S|)$, where the summation is taken over all disjoint sets S of the current partition.

MakeSet(x): Charge $(1 + log n). $1 to do the op, $(log n) stored as credit with item x.

Find(x): Charge $1, and use it to do the op.

Union(A,B): Charge $0, use $1 stored credit from each item in the smaller set to move it.

Potential Function Method:

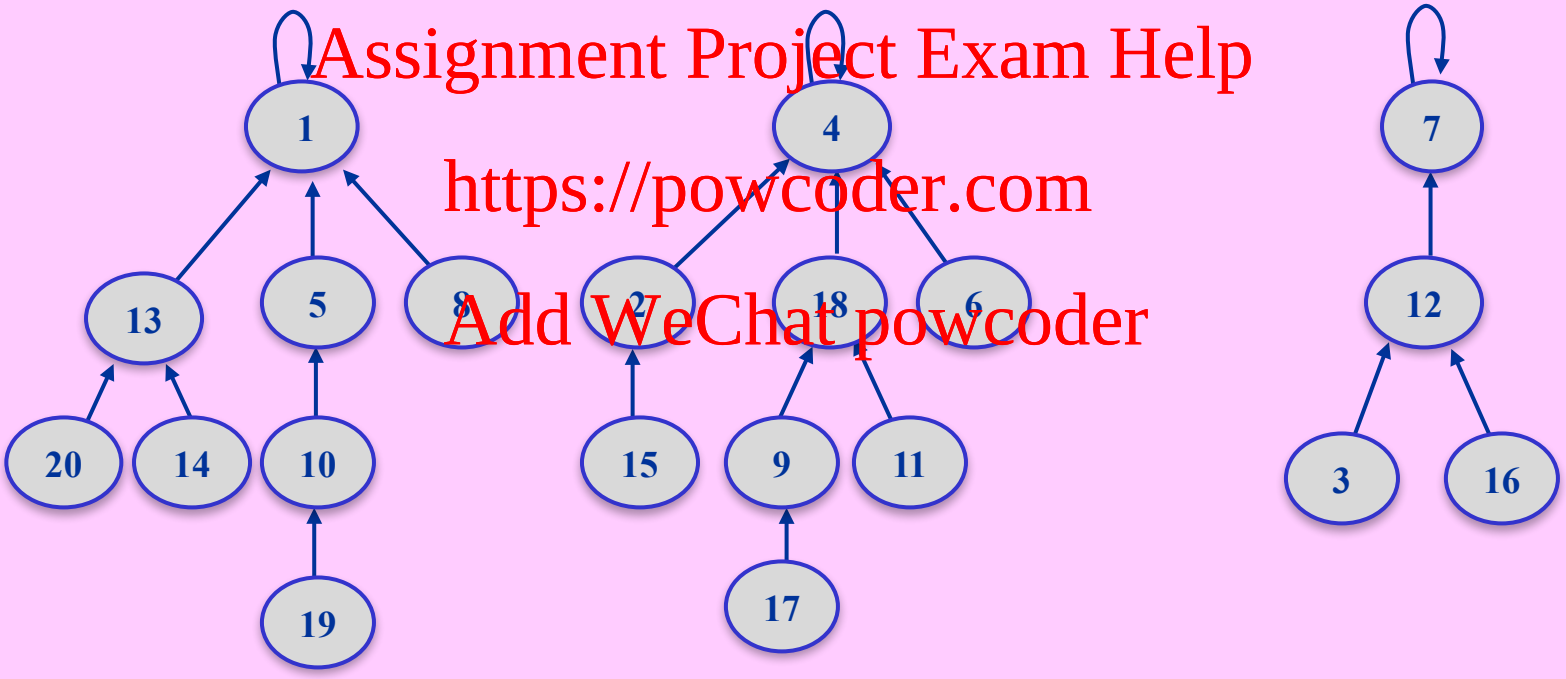
Exercise: Define a regular potential function and use it to do the amortized analysis. Can you make the Union amortized cost O(log n), MakeSet & Find costs O(1)?

Implementation3: Forest of Up-Trees

Data Structure: "parent" array p[1..n]

A is a set $\Leftrightarrow$ A = p[A] (a tree root)

$x \in A \Leftrightarrow$ x is in the tree rooted at A.



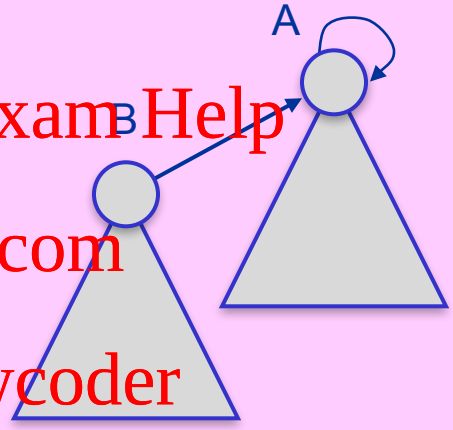
Forest of Up-Trees: Operations

MakeSet (x) **O(1) time**

```
    p[x] ← x
end
```

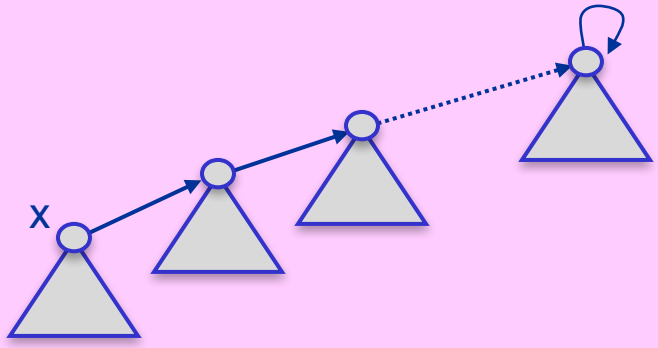
Union (A,B) **O(1) time**

```
    (* p[A]=A ≠ p[B]=B *)
    (* Link B under A *)
    p[B] ← A
end
```



Find (x) **O(depth(x)) time**

```
    if x = p[x]  then return x
    return Find(p[x])
end
```



Forest of Up-Trees: Amortized Cost

MakeSet (x) **O(1) time**

```
    p[x] ← x
end
```

Union (A,B) **O(1) time**

```
    (* p[A]=A ≠ p[B]=B *)
    (* Link B under A *)
    p[B] ← A
end
```

Find (x) **O(depth(x)) time**

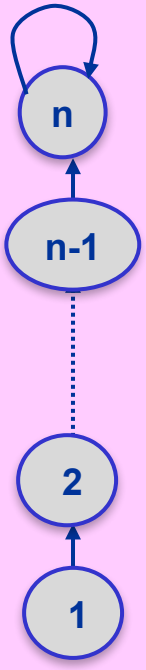
```
    if x = p[x]  then return x
    return Find(p[x])
end
```

Sequence s:

```
for x ← 1 .. n    do  MakeSet (x)
for x ← 1 .. n-1  do  Union (x+1,x)
for x ← 1 .. n    do  Find(1)
```

Aggregate Time = $\Theta(n^2)$

Amortized Time per operation = $\Theta(n)$



Self-Adjusting Forest of Up-Trees

Two self-adjusting improvements:

1. **Balanced Union:**
   a) by tree weight (i.e., size), or
   b) by tree rank (i.e., height)
2. **Find with Path Compression.**

- Each single improvement (1 or 2) by itself will result in logarithmic amortized cost per operation.
- The two improvements combined will result in amortized cost per operation approaching very close to O(1).

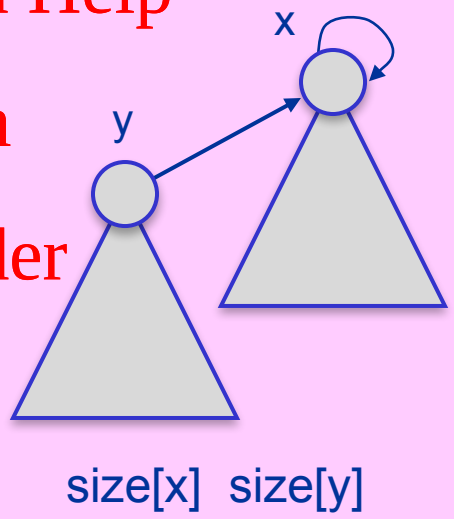
Balanced Union by Weight

MakeSet (x) **O(1) time**

```
    p[x] ← x
    size[x] ← 1
end
```


Union (A,B) **O(1) time**

```
    (* p[A]=A ≠ p[B]=B *)
    (* Link smaller under larger size tree *)
    if   size[A] > size[B]
         then size[A] ← size[A] + size[B]
                   p[B] ← A
         else  size[B]  ← size[A] + size[B]
                   p[A] ← B
end
```

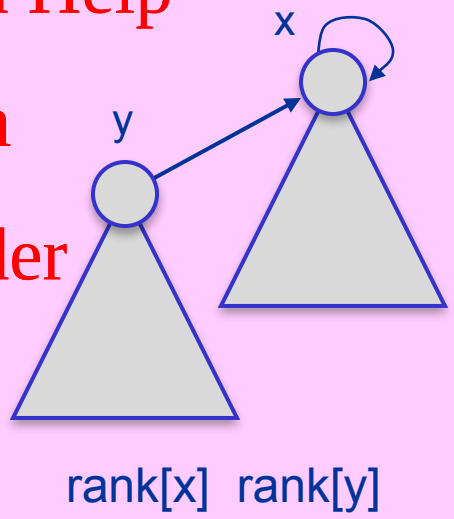
Balanced Union by Rank

MakeSet (x) **O(1) time**

```
    p[x] ← x
    rank[x] ← 0
end
```


Union (A,B) **O(1) time**

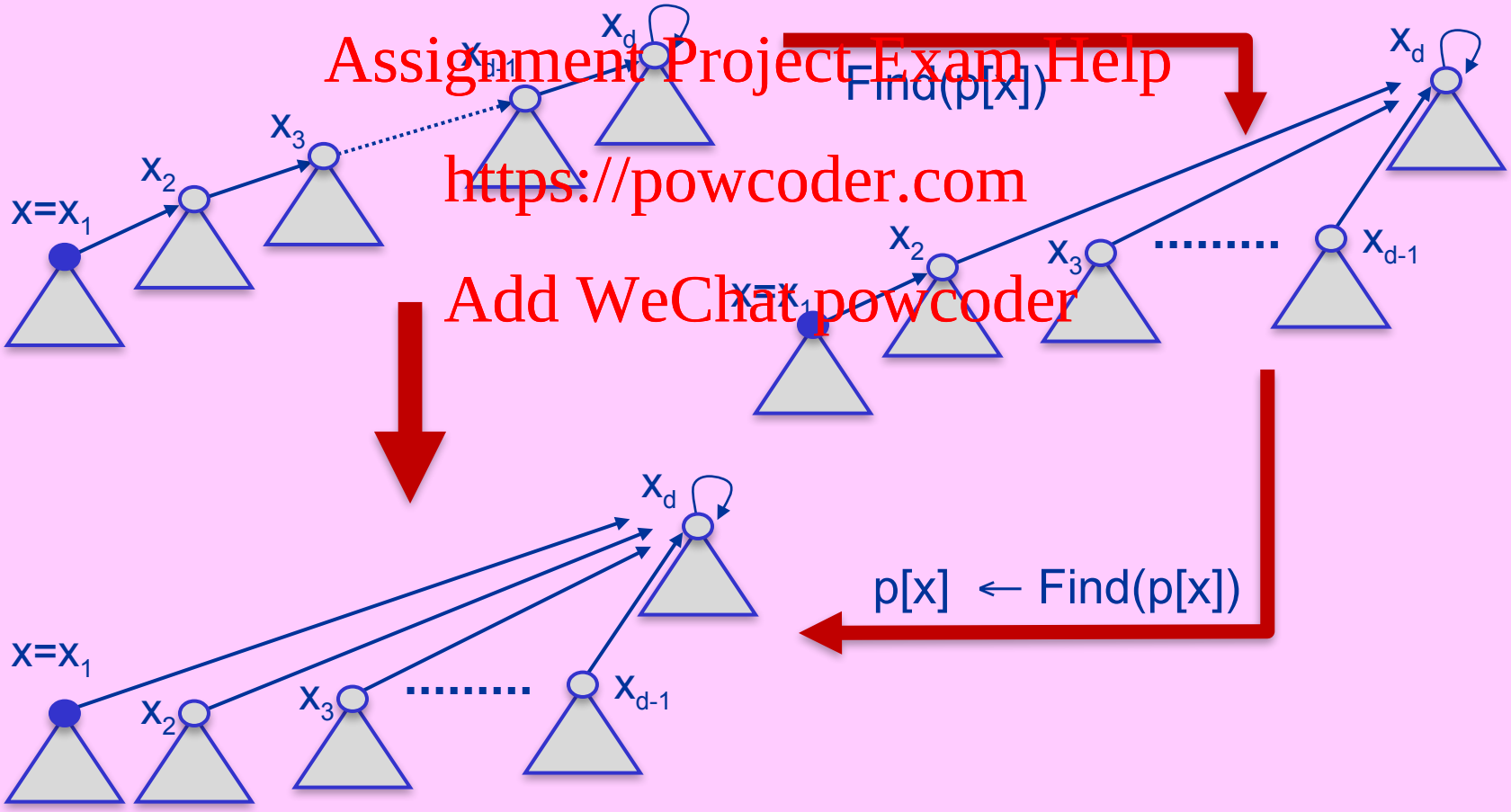
```
    (* p[A]=A ≠ p[B]=B *)
    (* Link lower under heigher rank tree *)
    if   rank[A] > rank[B]
         then  p[B] ← A
         else   p[A] ← B
                if  rank[A] = rank[B]
                    then rank[B] ← rank[B] + 1
         end-else
end
```

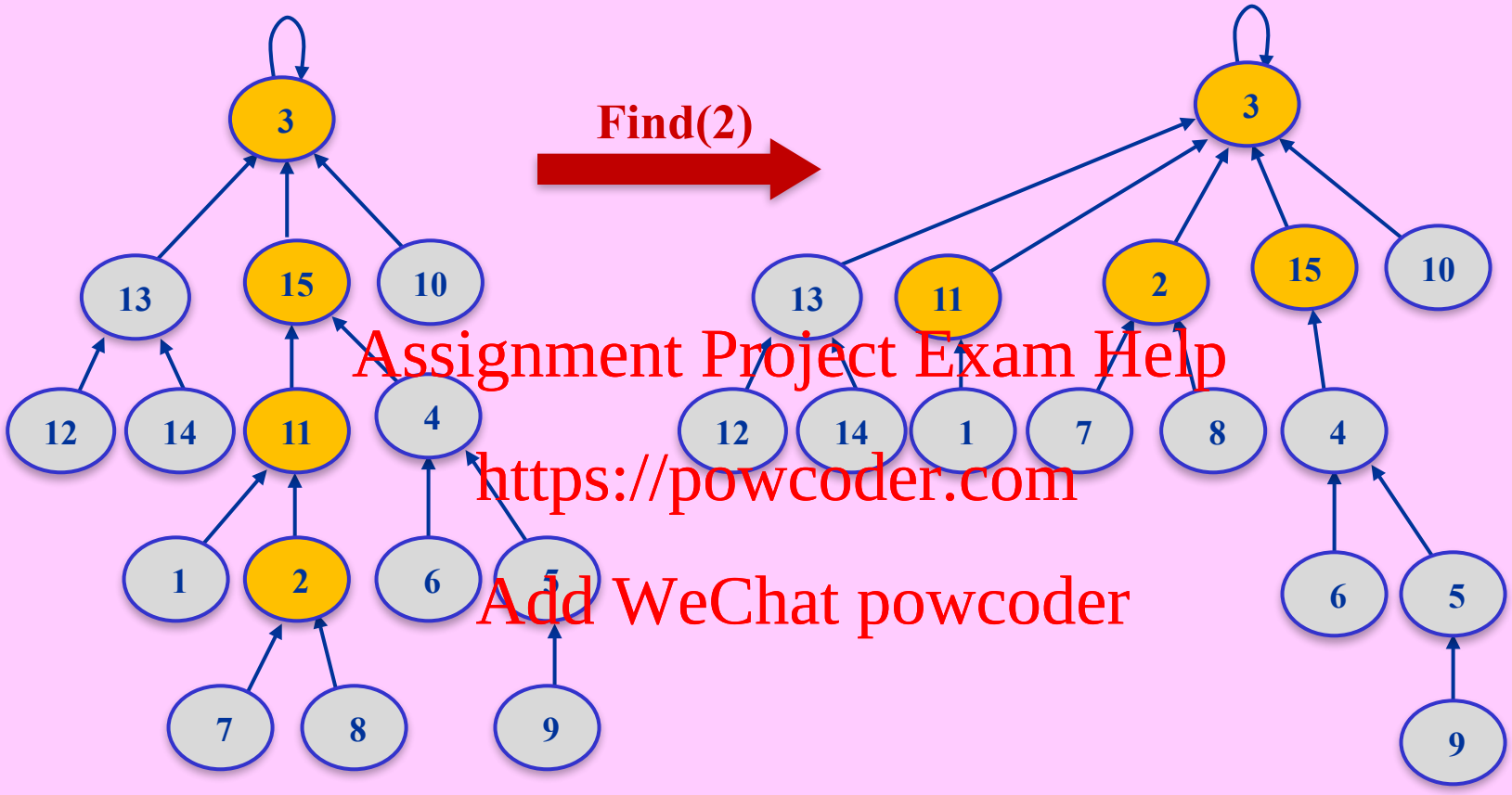
Path Compression

Find (x) **O(depth(x)) time**

```
    if x ≠ p[x]  then  p[x] ← Find(p[x])
    return p[x]
end
```



Example: Path Compression



Self-Adjustment: Find(x) should traverse the path from x up to its root.

It might as well create shortcuts along the way to improve the efficiency of the future operations.

Balanced Union FACT 0

FACT 0: With Bal.U. by rank but no Path Compression,
$\forall x$: rank[x] = height(x) (height of (sub-)tree rooted at x).

Proof: By induction on the # of operations. Relevant op's are:

MakeSet(x): rank[x] = 0 = height(x).

Union(x,y): rank[x] rank[y]:

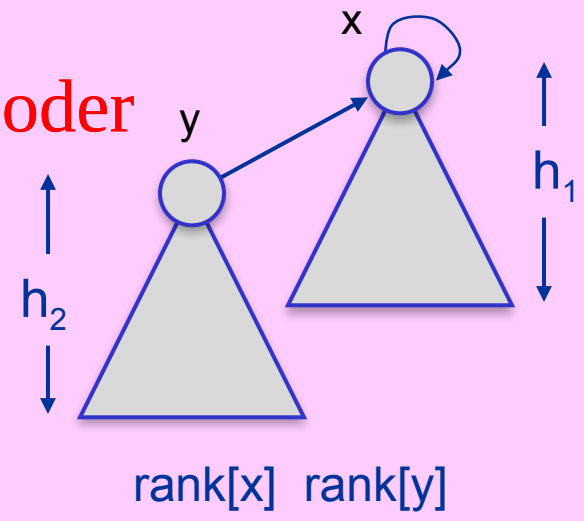
New height(x)

= max { h_1 , 1 + h_2 }

= max { rank[x], 1 + rank[y] }

$$= \begin{cases} \text{rank[x]} & \text{if rank[x] > rank[y]} \\ 1+\text{rank[x]} & \text{if rank[x] = rank[y]} \end{cases}$$

= new rank[x].



Balanced Union FACT 1

FACT 1: With Bal.U. by rank but no Path Compression,

$\forall x$: size of (sub-)tree rooted at x, $\text{size}(x) \geq 2^{\text{rank}[x]}$.

For a root x, this applies even with Path Compression (& Bal.U.).

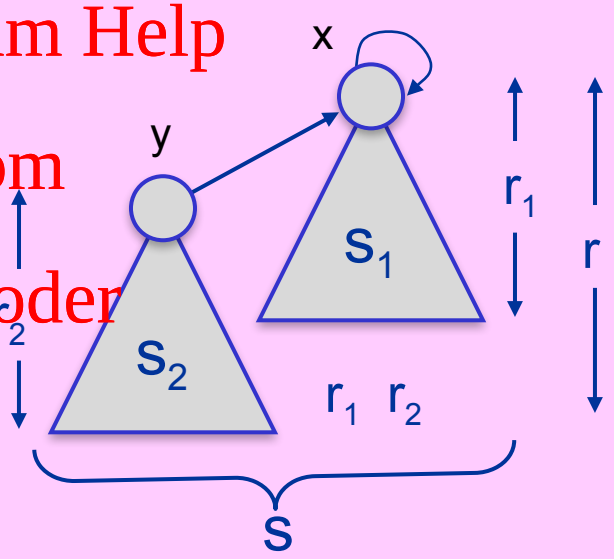
Proof: By induction on the # of operations. Relevant op's are:

MakeSet(x): size[x] = 1, rank(x) = 0.

Union(x,y): rank[x] ≠ rank[y].

$s_1 \geq 2^{r_1}$, $s_2 \geq 2^{r_2}$, $r_1 \neq r_2$

$s = s_1 + s_2$, $r = \max\{ r_1, 1+r_2 \}$



If $r_1 > r_2$: $s = s_1 + s_2 \geq s_1 \geq 2^{r_1} = 2^r$

If $r_1 = r_2$: $s = s_1 + s_2 \geq 2^{r_2} + 2^{r_2} = 2^{1+r_2} = 2^r$

$\therefore s \geq 2^r$

Balanced Union FACT 2

FACT 2: With Bal.U. by size but no Path Compression,

$\forall x$: size[x] $2^{rank(x)}$, where rank(x) = height(x).

For a root x, this applies even with Path Compression (& Bal.U.).

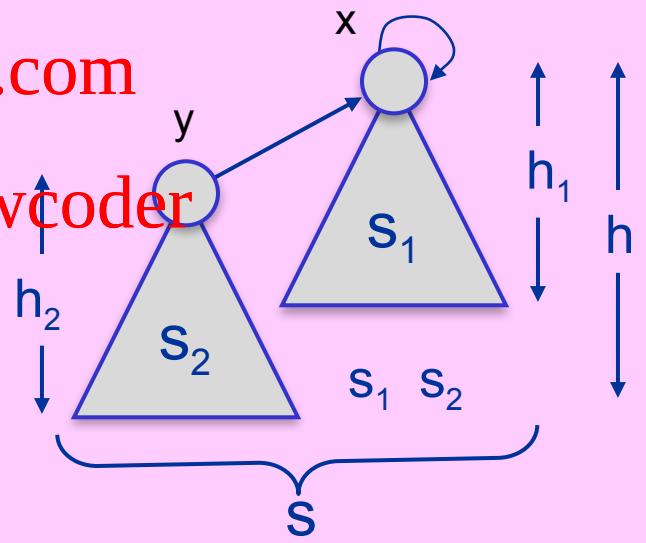
Proof: By induction on the # of operations. Relevant op's are:

MakeSet(x): size[x] = 1, rank(x) = height(x) = 0.

Union(x,y), size[x] ≥ size[y]:

s_1 2^{h_1}, s_2 2^{h_2}, $s_1 \geq s_2$

$s = s_1 + s_2$, $h = \max\{ h_1, 1+h_2 \}$



$s = s_1 + s_2$ s_1 2^{h_1}

$s = s_1 + s_2$ $2s_2$ $2 \cdot 2^{h_2} = 2^{1+h_2}$

$\therefore s$ $\max\{2^{h_1}, 2^{1+h_2}\} = 2^h$.

Balanced Union FACTS 3 & 4

FACT 3: With Balanced Union by size or by rank but without Path Compression,

$\forall x$: size(x) $\geq 2^{rank(x)}$, where rank(x)=height(x).

FACT 4: With Balanced Union by size or by rank (with or without Path Compression), height of each root is at most log n.

Balanced Union Amortized Time

FACT 5: With Balanced Union by size or by rank, Total time over a sequence of m MakeSet, Union, Find operations over a forest of size n is O(n + m log n). Amortized time per operation is O(log n).

Proof: Each MakeSet & Union take O(1) time.
By Fact 4, each Find(x) takes O(depth(x)) = O(log n) time.

Analysis of Balanced Union + Path Compression

Ackermann's Function

DEFINITION: Ackermann's function $A: Z^+ \times Z^+ \rightarrow Z^+$ (Z^+ = positive integers):

$$
\begin{aligned}
A(1, j) &= 2^j && \text{for } j \quad 1 \\
A(i, 1) &= A(i-1, 2) && \text{for } i \quad 2 \\
A(i, j) &= A(i-1, A(i, j-1)) && \text{for } i, j \quad 2
\end{aligned}
$$

A(i,j) is monotonically increasing in both i and j.

Example:

$A(2,2) = A(1, A(2,1)) = A(1, A(1,2)) = A(1,4) = 2^4 = 16.$

$A(2,3) = A(1, A(2,2)) = 2^{A(2,2)} = 2^{16} = 65536.$

$A(2,4) = A(1, A(2,3)) = 2^{A(2,3)} = 2^{65536}$.

Ackermann's Function

$$
\begin{aligned}
A(1, j) &= 2^j && \text{for } j \geq 1 \\
A(i, 1) &= A(i-1, 2) && \text{for } i \geq 2 \\
A(i, j) &= A(i-1, A(i, j-1)) && \text{for } i, j \geq 2
\end{aligned}
$$

$A(2,j) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (*j+1 two's*)

For j=1: $A(2,1) = A(1,2) = 2^2$

For j>1: $A(2,j) = A(1,A(2,j-1)) = 2^{A(2,j-1)} = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (*j two's*)

$A(3,j) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (*A(3, j–1) two's*)

$A(4,j) = !!!$

Inverse Ackermann's Function

DEFINITION: Inverse Ackermann's function:

$$\forall m \quad n \quad 1 :$$

$$\alpha(m,n) = \min\{ i \quad 1 : A(i, \lceil \tfrac{m}{n} \rceil) \quad n \}.$$

FACTS:

- For fixed m: $\alpha(m,n)$ is monotonically increasing in n.
- For fixed n: $\alpha(m,n)$ is monotonically decreasing in m.

- $\forall m \; n \; 1$: $\alpha(m,n) \leq \alpha(n,n)$.
- $\forall m \; n \log n$: $\alpha(m,n) = 1$.
- $\forall m \; n \log^* n$: $\alpha(m,n) \leq 2$.

log* n = super log (defined 2 slides ahead).

Amortized Cost for BU + PC

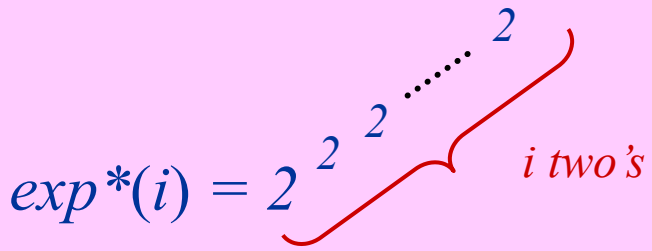
THEOREM 1:
With Balanced Union (by size or by rank) and Find with Path Compression, the amortized time per operation, over any sequence of m MakeSet, Union, and Find operations on a forest of size n, is $O(\alpha(n,m))$.

Super LOG & Super EXP

$\exp^*(0) = 1$

$\exp^*(i) = 2^{\exp^*(i-1)}$ for $i \geq 1$

$[\exp^*(-1) = -1]$



$\exp^*(5) = 2^{65536}$

$\log^*(j) = \min \{ i \geq 0 : \exp^*(i) \geq j \}$

$\log^*(2^{65536}) = 5$

Iterated Log: $\log^{(0)} j = j$, $\log^{(i)} j = \log (\log^{(i-1)} j)$

$\log^{(i)} j = \log (\log (\ldots (\log j)))$ (i logs)

That is, $\log^{(i)} j = \log (\log (\ldots (\log j)))$

Then, $\log^*(j) = \min \{ i \geq 0 : \log^{(i)} j \leq 1 \}$

Super LOG & Super EXP

$\exp^*(0) = 1$

$\exp^*(i) = 2^{\exp^*(i-1)}$ for $i \geq 1$.

$\log^*(j) = \min \{ i \geq 0 : \exp^*(i) \geq j \}$

$= \min \{ i \geq 0 : \log^{(i)} j \leq 1 \}$.

FACT:

- $\forall i \geq 0: \log^*(j) = i \Leftrightarrow \exp^*(i-1) < j \leq \exp^*(i)$

- $\log^*(2^x) = 1 + \log^*(x)$

- $\log^*(\log n) = \log^*(n) - 1$
- $\log^*(n) \geq \alpha(n,n) \geq \alpha(m,n)$.

Super LOG & Super EXP

$\exp^*(0) = 1$

$\exp^*(i) = 2^{\exp^*(i-1)}$ for $i \geq 1$.

$\log^*(j) = \min \{ i \geq 0 : \exp^*(i) \geq j \}$

$= \min \{ i \geq 0 : \log^{(i)} j \leq 1 \}$.

$$\log^*(j) = i \iff \exp^*(i-1) < j \leq \exp^*(i).$$

i:	0	1	2	3	4	5	…
$\exp^*(i)$:	1	2	$2^2=4$	$2^4=16$	$2^{16}=65536$	2^{65536}	…

j:	0	1	2	3	4	5 .. 16	17 .. 65536	65537 .. 2^{65536}	…
$\log^*(j)$:	0	1	2	3	4			5	…

Theorem: Amortized Cost for BU + PC

THEOREM 1:
With Balanced Union (by size or by rank) and Find with Path Compression, the amortized time per operation, over any sequence of m MakeSet, Union, and Find operations on a forest of size n, is $O(\alpha(m,n))$.

$\alpha(m,n) \quad \log^*(n)$

We will show a proof of the following somewhat weaker result:

THEOREM 2:
With Balanced Union (by size or by rank) and Find with Path Compression, the amortized time per operation, over any sequence of m MakeSet, Union, and Find operations on a forest of size n, is $O(\log^*(n))$.

FUCF & Node Rank

DEFINITION:

- Final UnCompressed Forest (FUCF) = the final state of the data structure after the entire sequence of m operations are performed with Balanced Union but Finds without Path Compression.

- Note: The actual data structure is dynamically changing with each operation, and Finds are done with Path Compression.

- We will compare the fixed FUCF with our Dynamically changing balanced Compressed Forest (DCF).

- rank(x) (the fixed) height of x in the FUCF.

BU + PC Facts 6-9

FACT 6: In FUCF: $\forall x$: size(x) $2^{rank(x)}$.

FACT 7: # rank r nodes $\leq \frac{n}{2^r}$.

Max rank = $\lfloor \log n \rfloor$.

FACT 8: If at some time in DCF, node y is a proper descendant of x, then, in FUCF y is a proper descendant of x, thus, rank(y) < rank(x).

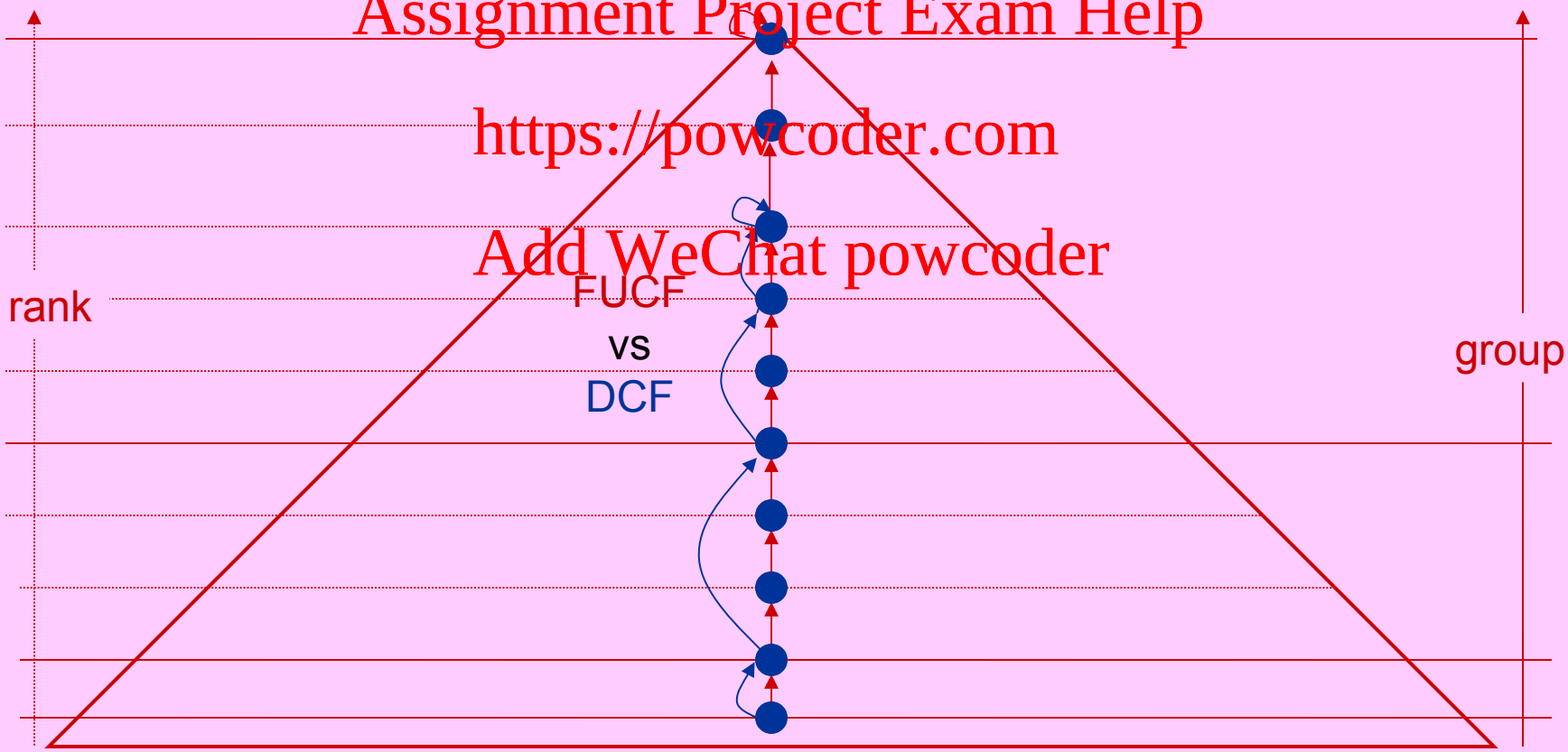
FACT 9: If at some time in DCF, p[x] changes from y to z, then rank(x) ≤ rank(y) < rank(z).

Node Groups

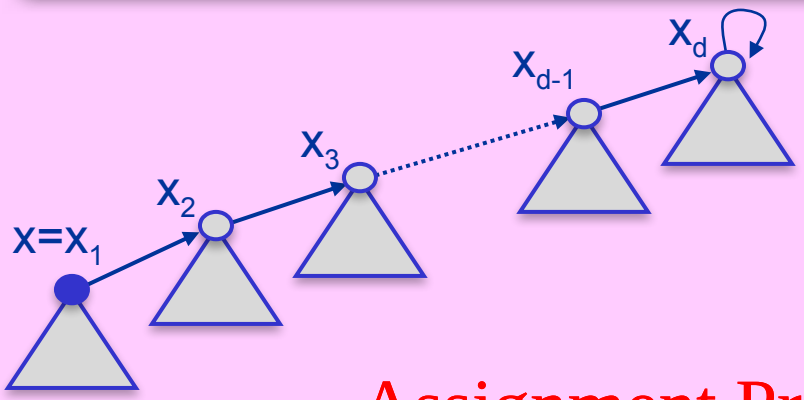
DEFINITION:

$$\text{group}(x) = \log^*(\text{rank}(x)) \quad \text{(fixed for x)}$$

$$\leq \log^*(\log n) = \log^*(n) - 1.$$



Find(x) cost accounting: 1/3



Find(x) costs
O(1) per node x_i, i=1..d.

Charge O(1) cost of node x_i to:

(1) x_i if $x_i \neq p[x_i]$ & $group(x_i) = group(p[x_i])$

(2) Find(x) if $x_i = p[x_i]$ or $group(x_i) \neq group(p[x_i])$

(2): Find(x) is charged O(log* n) .
Aggregate charge to all Finds is O(m log* n).

(1): Consider a node x charged this way.
Let g = group(x) = group(p[x]).
So, exp*(g-1) < rank(x) < rank(p[x]) ≤ exp*(g),
Fact 9: rank(p[x]) < rank(new p[x]). So, over all Find operations,
x can receive an aggregate charge of O(exp*(g) – exp*(g-1)).

Find(x) cost accounting: 2/3

$\text{group}(x) = g \iff \exp^*(g-1) < \text{rank}(x) \le \exp^*(g)$

Define: *N(g) = # nodes in group g.*

By Fact 7:

$$N(g) \le \sum_{r=1+\exp^*(g-1)}^{\exp^*(g)} \frac{n}{2^r}$$

$$\le \frac{n}{2^{1+\exp^*(g-1)}}\left(1+\tfrac{1}{2}+\tfrac{1}{4}+\cdots\right)$$

$$\le \frac{n}{2^{\exp^*(g-1)}} = \frac{n}{\exp^*(g)}.$$

Find(x) cost accounting: 3/3

Aggregate charge to all nodes in group g

$$= O\left(N(g) \left(\exp^*(g) - \exp^*(g-1) \right) \right)$$

$$= O\left(n \frac{\exp^*(g) - \exp^*(g-1)}{\exp^*(g)} \right)$$

$$= O(n).$$

(1) Aggregate charge to all nodes = O(n log* n).
(2) Aggregate charge to all Finds = O(m log* n).

∴ Total cost of all Find operations = O((m+n) log* n) = O(m log* n).

∴ Total cost of m MakeSet, Union, Finds = O(n + m log* n).

∴ Amortized cost per operation = O(log* n).

QED: Theorem 2.

Bibliography:

R.E. Tarjan, "Class notes: Disjoint set union," COS 423, Princeton University, 1999.
[shows the O(α(m,n)) amortized bound proof that appears in [CLRS]]

H.N. Gabow, R.E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," J. Computer & System Sciences 30(2), pp:209-221, 1985.

R.E. Tarjan, J. van Leeuwen "Worst-case analysis of set union algorithms," JACM 31(2), pp:245-281, 1984. [one-pass variants w. path-compression, e.g., path halving]

R.E. Tarjan, "Data Structures and Network Algorithms," CBMS-NSF, SIAM Monograph, 1983.
[original author to show the O(α(m,n)) amortized bound]

R.E. Tarjan, "A unified approach to path problems," JACM 28, pp:57-593, 1981.

R.E. Tarjan, "Applications of path compression on balanced trees," JACM 26(4), pp:690-715, 1979.

R.E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," J. Computer & System Sciences 18, pp:110-127, 1979.
[shows Ω(α(m,n)) amortized bound is required by any algorithm on a pointer machine]

R.E. Tarjan, "Efficiency of a good but not linear set union algorithm," JACM 22, pp:215-225, 1975.
[shows the O(α(m,n)) amortized bound on Bal.U. + P.C. is tight in the worst-case]

A.V. Aho, J.E. Hopcroft, J.D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, 1974. [original authors to show the O(log* n) amortized bound]

Exercises

1. **[CLRS, Exercise 21.2-2, pages 567-568]** Consider the following program.
   for i ← 1..16 **do** MakeSet(i)
   for i ← 1..8 **do** Union(Find(2i–1), Find(2i))
   for i ← 1..4 **do** Union(Find(4i–3), Find(4i–1))
   Union(Find(1), Find(5)); Union(Find(11), Find(13)); Union(Find(1), Find(10))
   Find(2); Find(9).
   Show the data structure that results and the answers returned by the Find operations:
   (a) on Implementation2 (weighted cyclic lists).
   (b) on Forest of Up-trees with balanced union by rank but without path compression.
   (c) on Forest of Up-trees with balanced union by size and path compression.

2. Amortized analysis of Implementation2 (weighted cyclic lists).
   (a) Define a regular potential function for the amortized analysis.
   (b) Can you make the Union amortized cost O(log n), MakeSet & Find costs O(1)?

3. **[CLRS, Exercise 21.3-5, page 572]** Consider any sequence s of m MakeSet, Union, and Find operations (of which n are MakeSets) applied to an initially empty data structure. Furthermore, all Union operations in s appear before any of the Find operations. Show that execution of s takes only O(m) time if path compression is used with or without balanced union. (Note: what [CLRS] calls "Link", we call "Union". So, no path compression takes place during Union calls.)
   [Hint: redefine the node groups.]

4. **Path halving:** Suppose we implement partial path compression on Find(x) by making every other node on the path from x to the root point to its grandparent. Parents of the other nodes on the path do not change. Let us call this path halving.
   (a) Write a one-pass bottom-up procedure for Find with path halving.
   (b) Prove that if Find is done with path halving, and balanced union by rank or by size is used, the worst-case running time for m MakeSet, Union, and Find operations (of which n are MakeSets) applied to an initially empty data structure, still takes time O(m log* n) (actually O(m α(m,n)).

5. **Augmented Union-Find I:** Suppose in addition to operations MakeSet, Union and Find, we have the extra operation **Remove(x)** , which removes element x from its current set and places it in its own singleton set. Show in detail how to modify the forest of up-trees data structure and its procedures so that a sequence of m MakeSet, Union, Find, and Remove operations (of which n are MakeSets) applied to an initially empty data structure, still takes time O(m log* n) (actually O(m α(m,n)).

6. **Augmented Union-Find II:** Suppose in addition to operations MakeSet, Union and Find operations, we have the extra operation **Deunion**, which undoes the last Union operation that has not been already undone.

(a) Consider a sequence of MakeSet, Union, Find, and Deunion operations. Describe a simple auxiliary data structure that can be used to quickly determine, for each Deunion in the sequence, what is the matching previous Union that it has to undo (and point to the location in the forest where change needs to take place).

(b) Show that if we do balanced union by rank and Finds without path-compression, then Deunion is easy and a sequence of m MakeSet, Union, Find, and Deunion operations on a forest of size n takes O(m log n) time.

(c) Why does path compression make Deunion harder to implement? Can you improve the O(m log n) time bound in part (b)?

7. **Graph edge removal sequence:** We are given a connected undirected graph G = (V,E) with n = |V| vertices and m = |E| edges, and an ordered list of its edges $\langle e_1, e_2, \ldots, e_m \rangle$ off-line . Consider removing edges of G, one by one, in the given order. Let G(i) be the graph after the first i edges on the list are removed from G. (Eventually, when all edges are removed, every connected component of G will have only one vertex in it.) The problem is to determine the smallest i such that the number of vertices in each connected component of G(i) is at most n/2. Describe an efficient algorithm to solve the problem and analyze its time complexity.

8. **Minimum Spanning Tree:** We are given a connected, undirected graph G = (V,E) with an edge weight function w: E → ℜ. We call w(u,v) the weight of edge (u,v). We wish to find an MST(G): an acyclic subset T⊆ E that connects all the vertices in V and whose total weight $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized.

Answer parts (a) – (d), assuming the edges E of G are given in **sorted** order of their weights.

(a) Design and analyze an efficient variant of Kruskal's algorithm to compute MST(G).
(b) Design and analyze an efficient variant of Prim's algorithm to compute MST(G).
(c) Is there an O(|E|) time algorithm that computes MST(G)?
(d) Is there an O(|E|) time algorithm that verifies whether a given subset T⊆ E is an MST(G)?

9. [CLRS, Problem 21-1, pages 582-583] Off-line minimum. The off-line minimum problem asks us to maintain a dynamic set T of items from the universe U = {1, 2, …, n} under the operations Insert and DeleteMin. We are given a sequence s of n Insert and m DeleteMin calls, where each key in U is inserted exactly once. We wish to determine which key is returned by each DeleteMin call. Specifically, we wish to fill in an array ***deleted[1..m]***, where for i = 1..m, *deleted*[i] is the key returned by the i^{th} DeleteMin call. The problem is "off-line", i.e., we are allowed to process the entire sequence s before determining any of the returned keys.

(a) Fill in the correct values in the *deleted* array for the following instance of the off-line minimum problem, where each Insert is represented by a number and each DeleteMin is represented by the letter D: ⟨4, 8, D, 3, D, 9, 2, 6, D, D, D, 1, 7, D, 5⟩.

To develop an algorithm for this problem, we break the sequence s into homogeneous subsequences. That is, we represent s by ⟨I_1, D, I_2, D, . . . , I_m, D, I_{m+1}⟩, where each D represents a single DeleteMin call and each I_j represents a (possibly empty) sequence of zero or more Insert calls. For each subsequence I_j, we initially place the keys inserted by these operations into a set K_j, which is empty if I_j is empty. We then do the following:


```
OffLineMinimum(m,n)
    for i ← 1..n   do
        determine j such that i ∈ K_j
        if   j ≠ m+1  then do
                    deleted[j] ← i
                    let t be the smallest value greater than j for which set K_t exists
                    K_t ← K_j ∪ K_t, destroying K_j
    return deleted
    end
```

(b) Argue that the array deleted returned by OffLineMinimum is correct.

(c) Describe how to implement OffLineMinimum efficiently with forest of up-trees. Give a tight bound on the worst-case running time of your implementation.

10. [CLRS, Problem 21-2, pages 583-584] Depth Determination. In the depth determination problem, we maintain a forest $\mathcal{F} = \{T_i\}$ of rooted trees under three operations:
MakeTree(v) creates a tree whose only node is v.
FindDepth(v) returns the depth of node v within its tree.
Graft(r, v) makes node r, which is assumed to be the root of a tree, become the child of node v, which is assumed to be in a different tree than r but may or may not itself be a root.

(a) Suppose that we use a tree representation similar to forest of up-trees: p[v] is the parent of node v (p[v] =v if v is a root). Suppose we implement Graft(r, v) by setting p[r] ← v and FindDepth(v) by following the find path up to its root, returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m MakeTree, FindDepth, and Graft operations for this implementation is $\Theta(m^2)$.

By using balanced union and path compression, we can reduce the worst-case running time. We use the forest of up-trees $\mathcal{S} = \{S_i\}$, where each set S_i (which is itself a tree) corresponds to a tree T_i in the forest $\mathcal{F}$. The tree structure within a set S_i, however, is not necessarily identical to that of T_i. In fact, the implementation of S_i does not record the exact parent-child relationships but nevertheless allows us to determine any node's depth in T_i.

The key idea is to maintain in each node v a "pseudo-distance" d[v], which is defined so that the sum of the pseudo-distances along the path from v to the root of its set S_i equals the depth of v in T_i. That is, if the path from v to its root in S_i is $\langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v$ and v_k is S_i's root, then the depth of v in T_i is $d[v_0] + d[v_1] + \ldots + d[v_k]$.

(b) Give an implementation of MakeTree.
(c) Show how to modify Find to implement FindDepth. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates pseudo-distances correctly.
(d) Show how to implement Graft(r, v), which combines the sets containing r and v, by modifying the Union procedure. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i.
(e) Give a tight bound on the worst-case running time of a sequence of m MakeTree, FindDepth, and Graft operations, n of which are MakeTree operations.

11. [CLRS, Problem 21-3, pages 584-585] Off-line Least Common Ancestors. The least common ancestor of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T with the above property. In the off-line least common ancestor problem, we are given a rooted tree T and an arbitrary set $P = \{ \{u_i, v_i\} \mid i=1..m\}$ of unordered pairs of nodes in T, and we wish to determine the least common ancestor of each pair $\{u_i, v_i\}$ in P.

To solve the off-line least common ancestors problem, the following procedure performs a tree walk of T with the initial call LCA(root[T]). Each node is assumed to be coloured WHITE prior to the walk.

```
LCA(u)
1     MakeSet(u)
2     ancestor[Find(u)] ← u
3     for each child v of u in T do
4         LCA(v)
5         Union(Find(u), Find(v))
6         ancestor[Find(u)] ← u
7     end-for
8     colour[u] ← BLACK
9     for each node v such that {u,v} ∈ P do
10        if colour[v] = BLACK
11        then print "The least common ancestor of" u "and" v "is" ancestor[Find(v)]
end
```


(a) Argue that line 11 is executed exactly once for each pair $\{u,v\} \in P$.
(b) Argue that at the time of the call LCA(u), the number of sets in the forest of up-trees data structure is equal to the depth of u in T.
(c) Prove that LCA correctly prints the least common ancestor for each pair $\{u,v\} \in P$.
(d) Analyze the running time of LCA, assuming that we use the implementation with forest of up trees with balanced union and path compression.

END