Introduction to

Algorithm Design and Analysis

[10] Union-Find

Yu Huang

http://cs.nju.edu.cn/yuhuang
Institute of Computer Software
Nanjing University

introduction to The Design & Analysis of Algorithms

In the last class…

- **Hashing**
  - Basic idea

- **Collision handling for hashing**
  - Closed address
  - Open address

- **Amortized analysis**
  - Array doubling
  - Stack operations
  - Binary counter

Union-Find

- **Dynamic Equivalence Relation**
  - Examples
  - Definitions
  - Brute force implementations

- **Union-Find Set**
  - Straight Union-Find
  - Making Shorter Tree by Weighted Union
  - Compressing Path by Compressing-Find

Minimum Spanning Tree

- **Kruskal's algorithm, greedy strategy:**
  - Select one edge
    - With the minimum weight
    - Not in the tree
  - Evaluate this edge
    - This edge will **NOT** result in a cycle
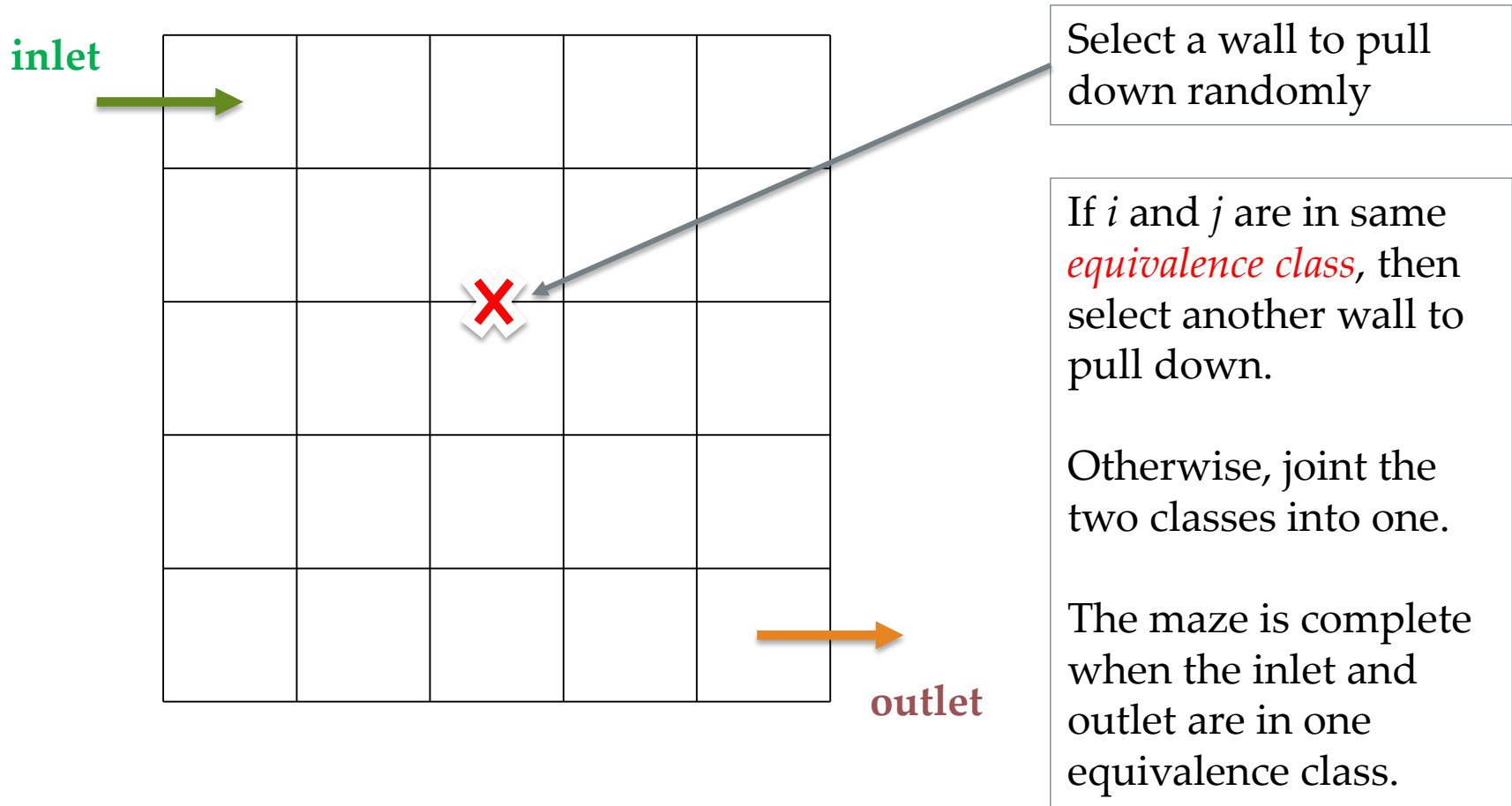- **Critical issue:**
  - How to know "NO CYCLE"?

Guess My Number – an Example

- **I pick a secret positive integer x**

- **In each round, you write a list of numbers**
  - You list at most x numbers
  - I told you the position of x

- **You must guarantee to win**
 - What is the least number of rounds in the worst case?

You	Me
1	It's bigger than 1.
4, 42	It's between 4 and 42.
8, 15, 16, 23, 30	It's between 8 and 15.
9, 10, 11, 12, 13, 14	It's 11; you win!

Maze Creation – an Example

inlet

outlet

Select a wall to pull down randomly

If i and j are in same *equivalence class*, then select another wall to pull down.

Otherwise, joint the two classes into one.

The maze is complete when the inlet and outlet are in one equivalence class.

Black pixels – an Example

- **Maximum black pixel component**
  - Let α be the size of the component

- **Color one pixel black**
  - How α changes?
  - How to choose the pixel, to accelerate the change in α
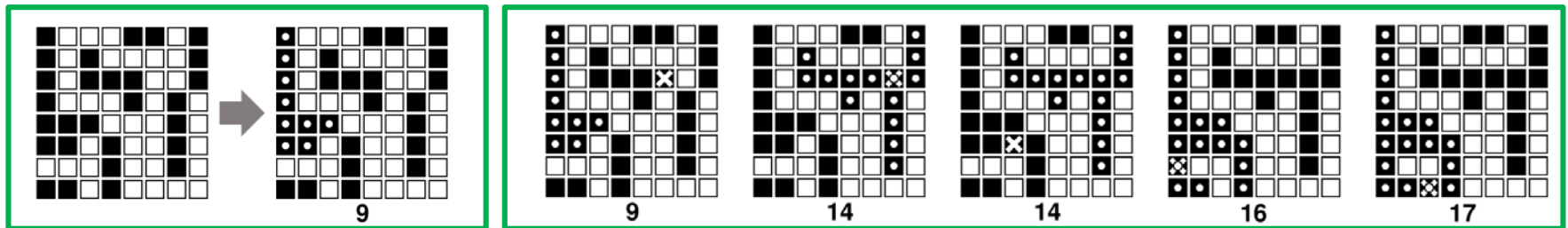
Jigsaw Puzzle – an Example

- **Multiple pieces may be glued together**
- **From "one player" to "two players"**
  - Each group can only be moved in a mutual exclusive way
  - How to decide the relation of "in the same group"
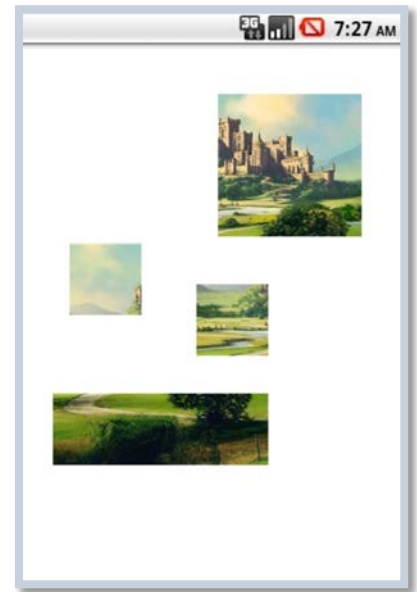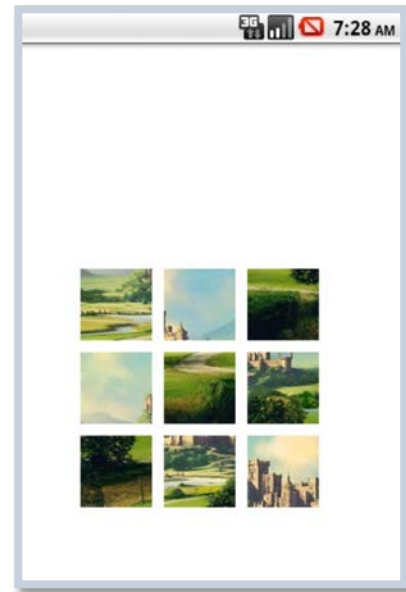
Dynamic Equivalence Relations

- **Equivalence**
  - Reflexive, symmetric, transitive
  - Equivalent classes forming a partition
- **Dynamic equivalence relation**
  - Changing in the process of computation
  - **IS** instruction: *yes* or *no* (in the same equivalence class)
  - **MAKE** instruction: combining two equivalent classes, by relating two unrelated elements, and influencing the results of subsequent IS instructions.
  - Starting as equality relation

Implementation: How to Measure

- **The number of basic operations for processing a sequence of m MAKE and/or IS instructions on a set S with n elements.**

- **An Example: S={1,2,3,4,5}**

  - 0. [create] {{1}, {2}, {3}, {4}, {5}}

  - 1. IS 2≡4?   No
  - 2. IS 3≡5?   No
  - 3. MAKE 3≡5.   {{1}, {2}, {3,5}, {4}}
  - 4. MAKE 2≡5.   {{1}, {2,3,5}, {4}}
  - 5. IS 2≡3?   Yes
  - 6. MAKE 4≡1.   {{1,4}, {2,3,5}}
  - 7. IS 2≡4?   No

Union-Find based Implementation

- **The matrix problem**
  - Each cell as a set
  - Randomly delete a wall and union two cells
  - Loop until you find the inlet and outlet are in one equivalent class

- **The Kruskal algorithm**
  - Each node as a set
  - Choose the least weight edge (u,v)
  - Find whether u and v are in the same equivalent class
  - If not, add the edge and union the two nodes

Implementation: Choices

- **Matrix** (relation matrix)
  - Space in $\Theta(n^2)$, and worst-case cost in $\Theta(mn)$ (mainly for row copying for MAKE).
- **Array** (for equivalence class id)
  - Space in $\Theta(n)$, and worst-case cost in $\Theta(mn)$ (mainly for search and change for MAKE).
- **Union-Find**
  - A object of type Union-Find is a collection of disjoint sets
  - There is no way to traverse through all the elements in one set.

Union-Find ADT

- **Constructor: Union-Find** create(int n)

  o sets=create(n) refers to a newly created group of sets {1}, {2}, ..., {n} (n singletons)

- **Access Function: int** find(UnionFind sets, e)

  o find(sets, e)=<e>

- **Manipulation Procedures**

  o **void** makeSet(UnionFind sets, **int** e)

  o **void** union(UnionFind sets, **int** s, **int** t)

Using Union-Find (as inTree)

- **IS** $s_i \equiv s_j$:
  - t=find(s_i);
  - u=find(s_j);
  - (t==u)?

- **MAKE** $s_i \equiv s_j$:
  - t=find(s_i);
  - u=find(s_j);
  - union(t,u);

implementation by inTree

create(n): sequence of makeNode

$\boxed{0}$ $\boxed{1}$ .... $\boxed{i}$ .... $\boxed{n\text{-}1}$ $\boxed{n}$

find(s_j)=u

parent^k(s_j)

union(t,u)

setParent(t,u)
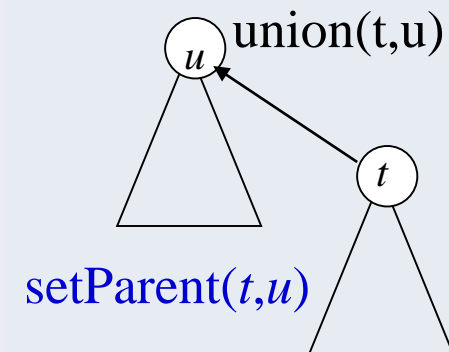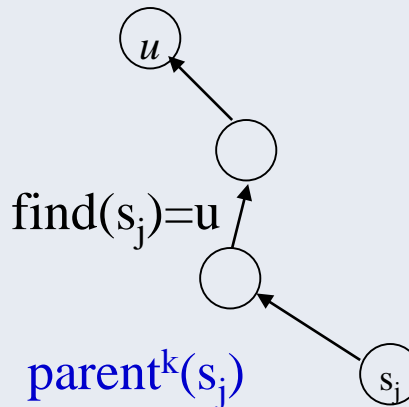
Union-Find Program

- A **union-find program of length m**
  - is (a *create*(n) operation followed by) a sequence of m union and/or find operations interspersed in any order.

- **A union-find program is considered an input, the object for which the analysis is conducted.**

- **The measure: number of accesses to the *parent***
  - assignments: for union operations
  - lookups: for find operations
  } **link operation**

Worst-case Analysis for Union-Find Program

- **Assuming each lookup/assignment take O(1).**
- **Each makeSet or union does one assignment, and each find does d+1 lookups, where d is the depth of the node.**

1. Union(1,2)
2. Union(2,3)

    ⋮

n-1. Union(n-1,n)
n. Find(1)

    ⋮

m. Find(1)

Example

The sequence of *Union* makes a chain of length n-1, which is the tree with the largest height

operations done:
$n+(n-1)+(m-n+1)n$

$\Theta\ (mn)$

Find(1) needs n array lookups

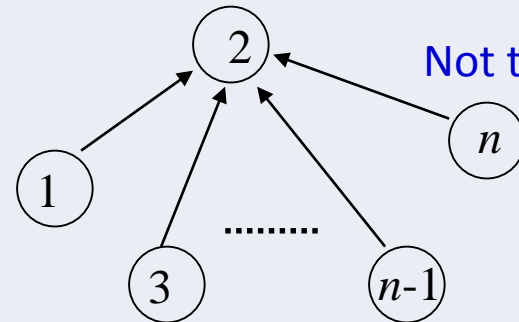# Weighted Union: for Short Trees

- **Weighted union: always have the tree with fewer nodes as subtree. (*wUnion*)**

To keep the *Union* valid, each *Union* operation is replaced by:

t=find(i);
u=find(j);
union(t,u)

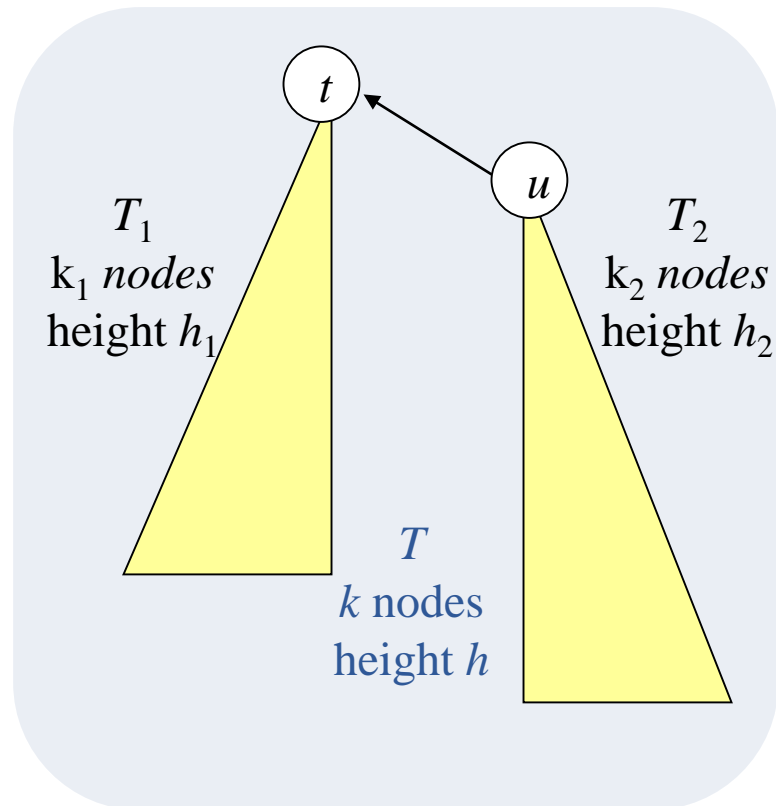The order of (t,u) satisfying the requirement

Not the worst case!

Tree made by wUnion

Cost for the program:
$n+3(n-1)+2(m-n+1)$

Upper Bound of Tree Height

- **After any sequence of *Union* instructions, implemented by *wUnion*, any tree that has k nodes will have height at most $\lfloor \lg k \rfloor$**

- **Proof by induction on k:**
  - base case: $k=1$, the height is 0.
  - by inductive hypothesis:
    - $h_1 \leq \lfloor \lg k_1 \rfloor$, $h_2 \leq \lfloor \lg k_2 \rfloor$
  - h=max(h1, h2+1), k=k1+k2
    - if $h=h_1$, $h \leq \lfloor \lg k_1 \rfloor \leq \lfloor \lg k \rfloor$
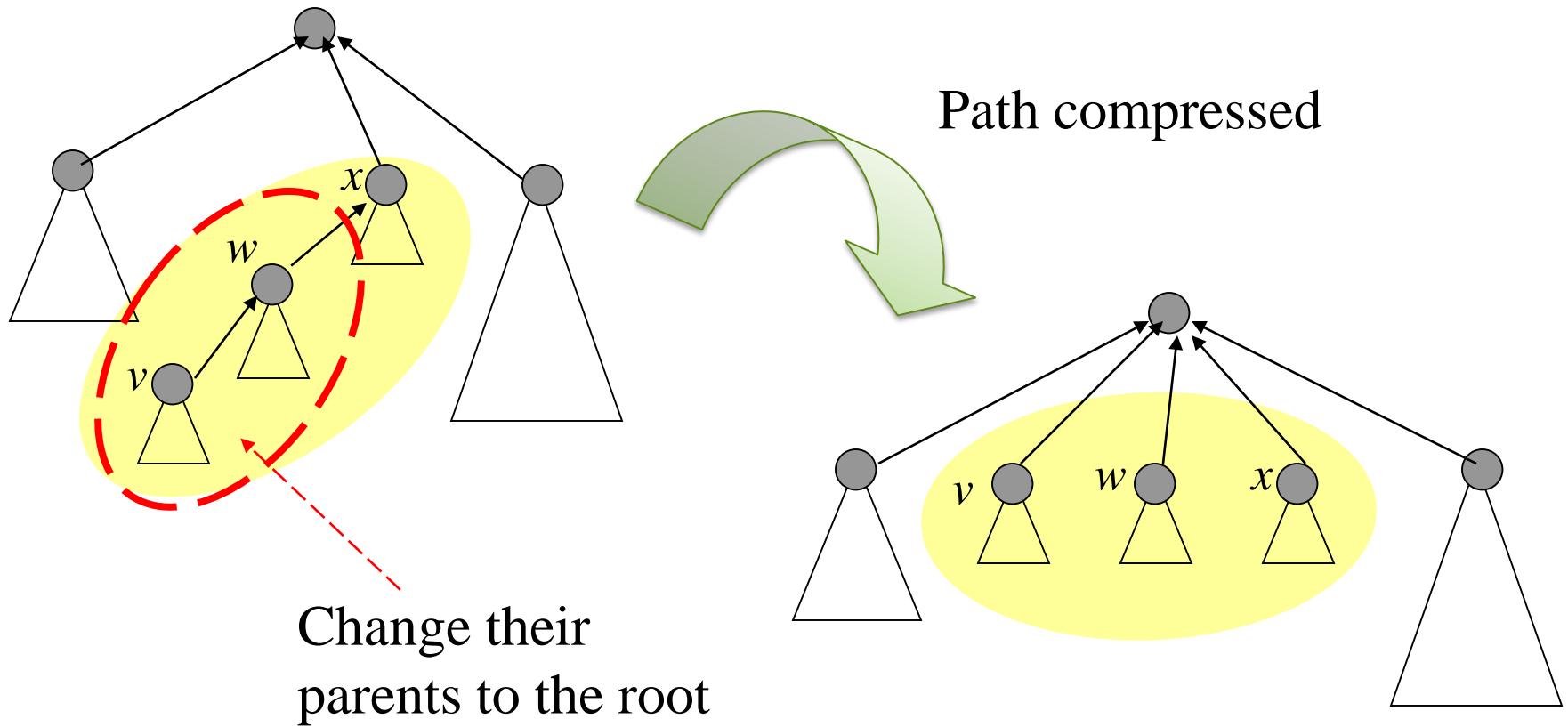   - if $h=h_2+1$, note: $k_2 \leq k/2$ so, $h_2+1 \leq \lfloor \lg k_2 \rfloor + 1 \leq \lfloor \lg k \rfloor$

T_1
k_1 *nodes*
height h_1

T_2
k_2 *nodes*
height h_2

T
k nodes
height h

Upper Bound for Union-Find Program

- **A Union-Find program of size *m*, on a set of *n* elements, performs O($n+m$log$n$) link operations in the worst case if *wUnion* and straight *find* are used.**

- **Proof:**

  - At most n-1 *wUnion* can be done, building a tree with height at most $\lfloor$log$n\rfloor$,

  - Then, each *find* costs at most $\lfloor$log$n\rfloor$+1.

  - Each *wUnion* costs in O(1), so, the upper bound on the cost of any combination of *m wUnion/find* operations is the cost of *m find* operations, that is $m(\lfloor$log$n\rfloor$+1)\in O$($n+mlogn$)
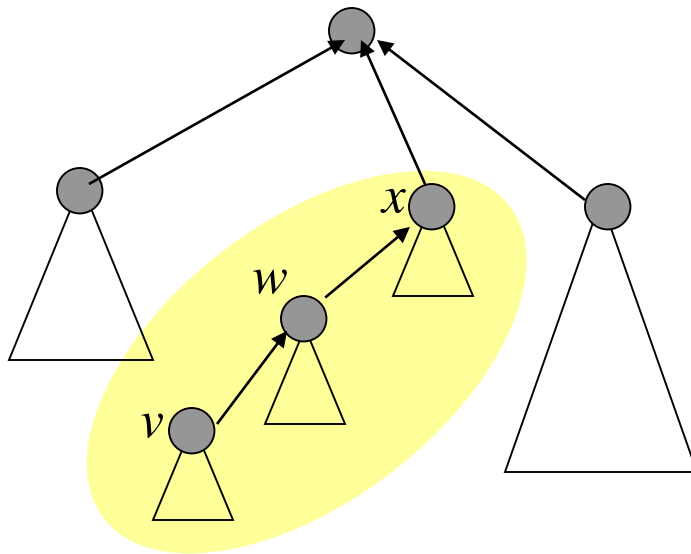
  There do exist programs requiring $\Omega(n+(m-n)$log$n)$ *steps.*

Path Compression
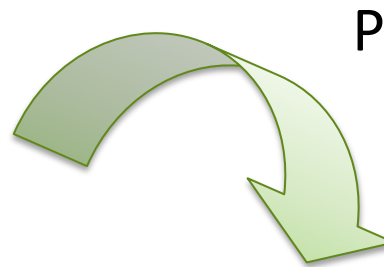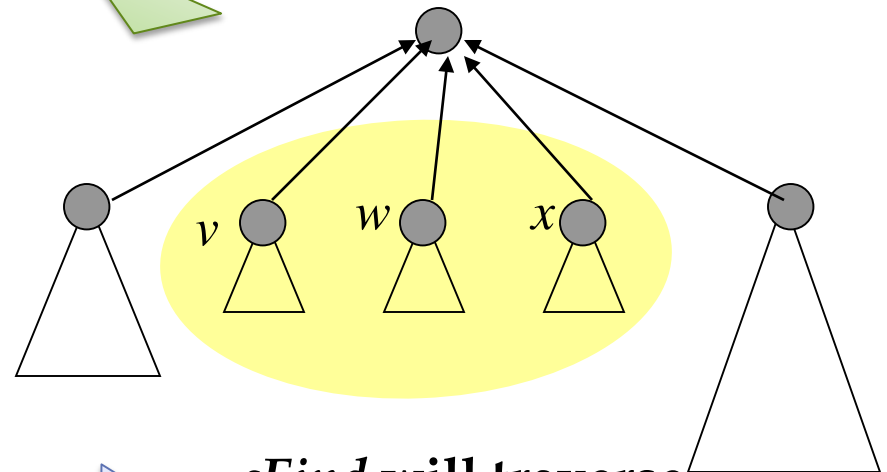

Path compressed

Change their parents to the root

Challenges for the Analysis

Path compressed

cFind does **twice as many** link operations as the *find* does for a given node in a given tree.

but…

cFind will traverse **shorter** paths

Analysis: the Basic Idea

- **cFind may be an expensive operation**
  - in the case that find(i) is executed and the node i has great depth.

- **However, such cFind can be executed only for limited times**
  - in contrast to other operations of lower cost.

- **So, *amortized analysis* applies**

Co-Strength of *wUnion* and *cFind*

- ## $O((n+m)\log^*(n))$
  - Link operations for a *Union-Find* program of length m on a set of n elements is in the worst case.
  - Implemented with *wUnion* and *cFind*

What's log*(n)?
- Define the function H as following:

$$\begin{cases} H(0) = 1 \\ H(i) = 2^{H(i-1)} \ \ for \ i > 0 \end{cases}$$

- Then, $\log^*(j)$ for $j \geq 1$ is defined as:

$$\log^*(j) = \min\{ \, k \, | \, H(k) \geq j \, \}$$

Definitions with a *Union-Find* Program *P*

- **Forest *F*:** the forest constructed by the sequence of *union* instructions in *P*, assuming:
  - *wUnion* is used;
  - the *find*s in the *P* are ignored
- **Height** of a node v in any tree: the height of the subtree rooted at v
- **Rank** of v: the height of v *in F*

> Note: *cFind* changes the height of a node, but the rank for any node is invariable.

Constraints on Ranks in *F*

- **The upper bound of the number of nodes with rank *r* (*r*≥0) is $\dfrac{n}{2^r}$**

  - Remember that the height of the tree built by *wUnion* is at most $\lfloor \lg n \rfloor$, which means the subtree of height *r* has at least 2^r nodes.

  - The subtrees with root at rank *r* are disjoint.

- **There are at most $\lfloor \log n \rfloor$ different ranks.**

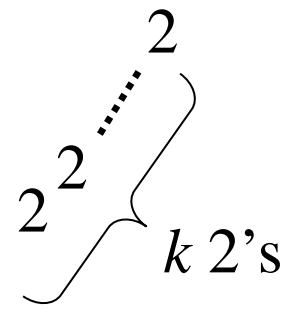 - There are altogether n elements in S, that is, n nodes in F.

Increasing Sequence of Ranks

- **The ranks of the nodes on a path from a leaf to a root of a tree in *F* form a strictly increasing sequence.**

- **When a *cFind* operation changes the parent of a node, the new parent has higher rank than the old parent of that node.**

  o Note: the new parent was an ancestor of the previous parent.

A Function Growing Extremely Slowly

- **Function H:**

  $\left\{\begin{array}{l} H(0)=1 \\ H(i+1)=2^{H(i)} \end{array}\right.$

  that is: $H(k)=2^{2^{2^{\cdot^{\cdot^{2}}}}}\left.\begin{array}{l}\\\\\end{array}\right\} k\ 2\text{'s}$

  Note:

  H grows extremely fast:

  $H(4)=2^{16}=65536$

  $H(5)=2^{65536}$

- **Function Log-star**

  log*(j) is defined as the least i such that:

  $H(i) \geq j$ for $j>0$

- **Log-star grows extremely slowly**

  $$\lim_{n\to\infty}\frac{log*(n)}{log^{(p)}n}=0$$

  ***p* is any fixed nonnegative constant**

  For any x: $2^{16} \leq x \leq 2^{65536}-1$, log*($x$)=5 !

Grouping Nodes by Ranks

- **Node $v \in s_i$ ($i \geq 0$) iff. log*(1+rank of v)=i**
  - which means that: if node v is in group i, then

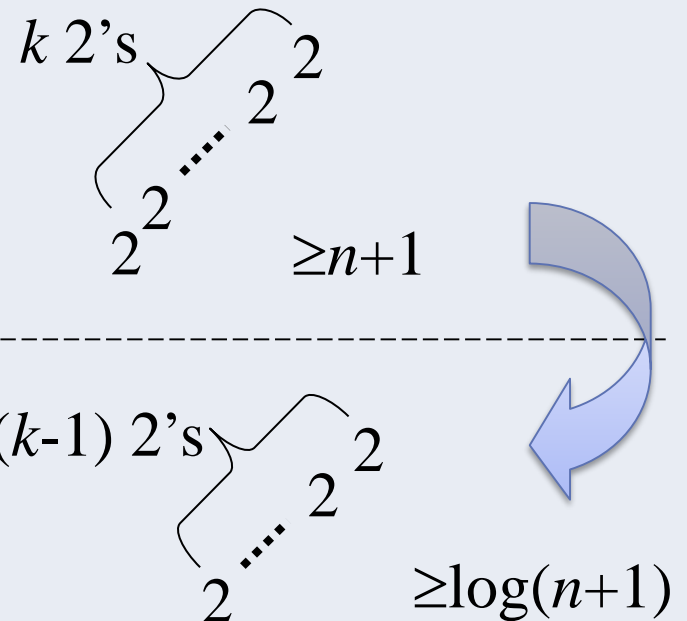 $r_v \leq H(i)-1$, but not in group with smaller labels

- **So,**
  - Group 0: all nodes with rank 0
  - Group 1: all nodes with rank 1
  - Group 2: all nodes with rank 2 or 3
  - Group 3: all nodes with its rank in [4,15]
  - Group 4: all nodes with its rank in [16, 65535]
  - Group 5: all nodes with its rank in [65536, ???]

Group 5 exists only when n is at least 2^{65536}. What is that?

Very Few Groups

- **Node $v \in S_i$ ($i \geq 0$) iff.**

  log*(1+rank of v)=i

- **Upper bound of the number of distinct node groups is log*(n+1)**

  - The rank of any node in F is at most $\lfloor \log n \rfloor$, so the largest group index is $\log^*(1+\lfloor \log n \rfloor)=\log^*(\lceil \log n+1 \rceil) = \log^*(n+1)-1$

If $\log^*(n+1)=k$, then

k 2's $\left.\begin{array}{l} \\ \\ \\ \end{array}\right\}$ $2^{2^{\cdots^{2^2}}}$ $\geq n+1$

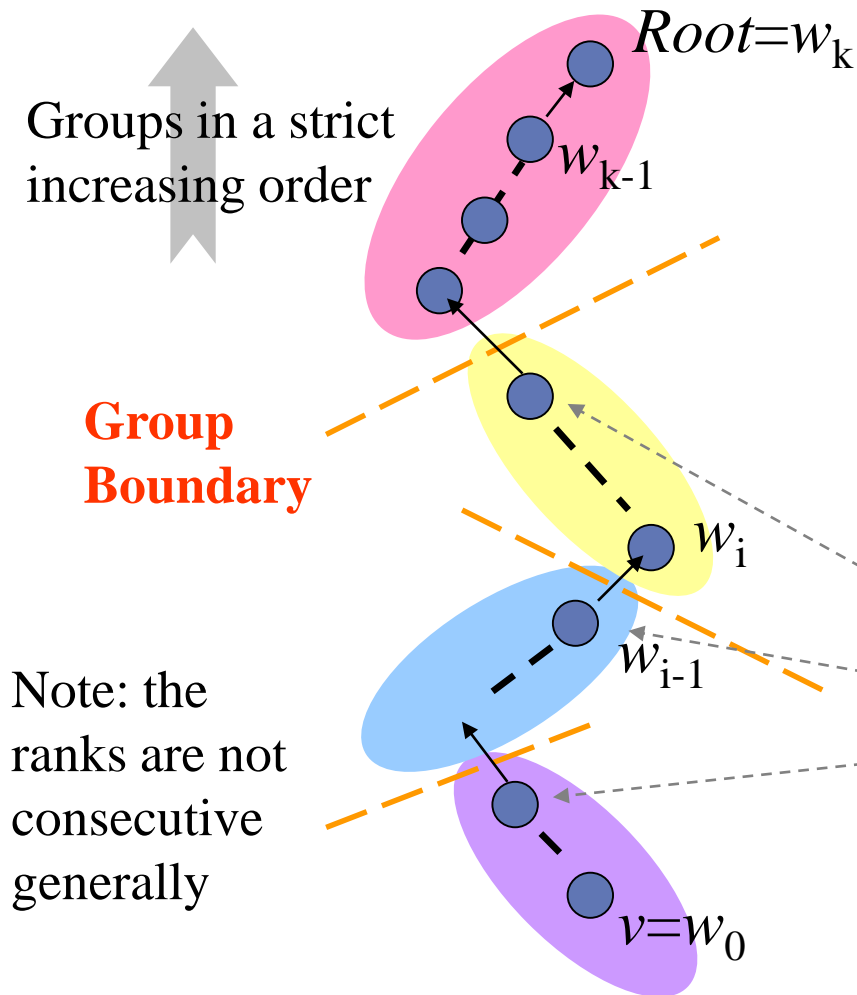$(k-1)$ 2's $\left.\begin{array}{l} \\ \\ \end{array}\right\}$ $2^{2^{\cdots^{2^2}}}$ $\geq \log(n+1)$

Amortized Cost of *Union-Find*

- **Amortized Equation Recalled**

  amortized cost = actual cost + accounting cost

- **The operations to be considered:**
  - n makeSets
  - m union & find (with at most n-1 unions)

One Execution of *cFind*(w_0)

Root=w_k

Groups in a strict increasing order

w_{k-1}

Group Boundary

w_i

w_{i-1}

Note: the ranks are not consecutive generally

$v=w_0$

Only when k=0,1, there is no parent change

For one *cFind* operation, the actual cost is $2k$ Not $2(k+1)$

Accounting cost is -2 for each pair of (w_{i-1}, w_i) for the the 2 nodes in ths **same group only**, which we call a **withdrawal**.

Amortizing Scheme for *wUnion-cFind*

- **makeSet**
  - Accounting cost is $4\log^*(n+1)$
  - So, the amortized cost is $1+4\log^*(n+1)$
- *wUnion*
  - Accounting cost is 0
  - So the amortized cost is 1
- *cFind*
  - Accounting cost is describes as in the previous page.
  - Amortized cost $\leq 2k-2((k-1)-(\log^*(n+1)-1))=2\log^*(n+1)$
    (Compare with the worst case cost of *cFind,* $2\log n$)

Number of withdrawal

Validation of the Amortizing Scheme

- **We must be assure that the sum of the accounting costs is never negative.**

- **The sum of the negative charges, incurred by *cFind*, does not exceed $4n\log^*(n+1)$**
  - We prove this by showing that at most $2n\log^*(n+1)$ withdrawals on nodes occur during all the executions of *c*Find.

Key Idea in the Derivation

- **For any node, the number of withdrawal will be less than the number of different ranks in the group it belongs to**

  - When a *cFind* changes the parent of a node, the new parent is always has higher rank than the old parent.
  - Once a node is assigned a new parent in a higher group, no more negative amortized cost will incurred for it again.

- **The number of different ranks is limited within a group.**

Derivation

- ## Bounding the number of withdrawals

The number of withdrawals from all $w \in S$ is:

$$\sum_{i=0}^{log^*(n+1)-1} H(i)(\text{number of nodes in group } i)$$

a loose upper bound of ranks in a group

The number of nodes in group i is at most:

$$\sum_{r=H(i-1)}^{H(i)-1} \frac{n}{2^r} \leq \frac{n}{2^{H(i-1)}} \sum_{j=0}^{\infty} \frac{1}{2^j} = \frac{2n}{2^{H(i-1)}} = \frac{2n}{H(i)}$$

So,

$$\sum_{i=0}^{log^*(n+1)-1} H(i)\frac{2n}{H(i)} = 2n \log^*(n+1)$$

Conclusion

- **The number of link operations done by a *Union-Find* program implemented with *wUnion* and *cFind*, of length *m* on a set of *n* elements is in $O((n+m)log*(n))$ in the worst case.**

  o Note: since the sum of accounting cost is never negative, the actual cost is always not less than amortized cost. The upper bound of amortized cost is: $(n+m)(1+4log*(n+1))$

Thank you!

Q & A

Yu Huang
http://cs.nju.edu.cn/yuhuang