Introduction to

Algorithm Design and Analysis

[10] Union-Find

Jingwei Xu
https://ics.nju.edu.cn/~xjw/
Institute of Computer Software
Nanjing University

In the last class …

- **Hashing**
  - Basic idea
- **Collision handling for hashing**
  - Closed address
  - Open address
- **Amortized analysis**
  - Array doubling
  - Stack operations
  - Binary counter

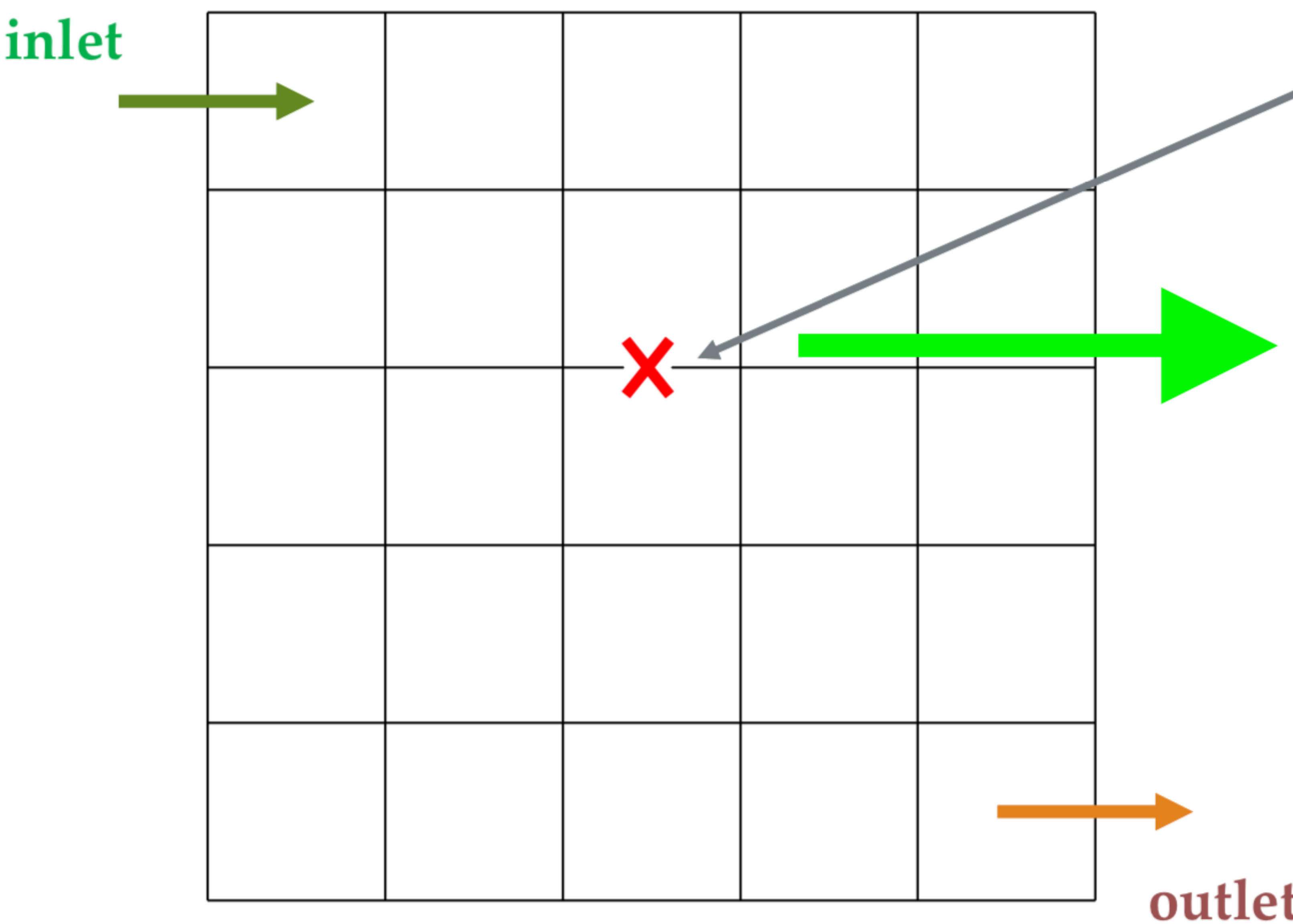
Union-Find

- **Dynamic Equivalence Relation**
  - Examples
  - Definitions
  - Brute force implementations
- **Disjoint Set**
  - Straightforward Union-Find
  - Weighted Union + Straightforward Find
  - Weighted Union + Path-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"**?

Maze Generation



Select a wall to pull down randomly

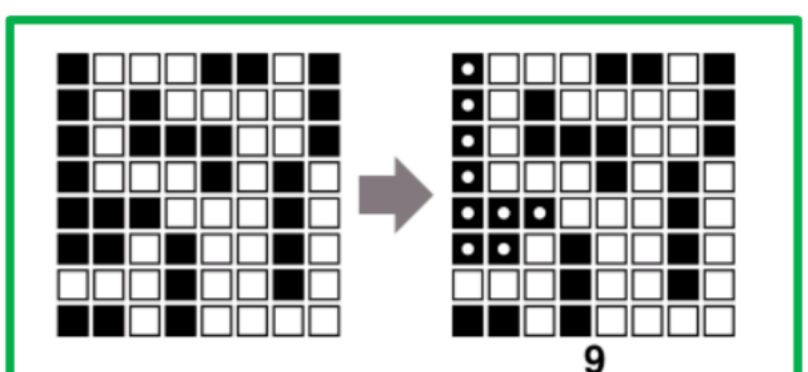
If i and j are in same *equivalence class*, then select another wall to pull down.

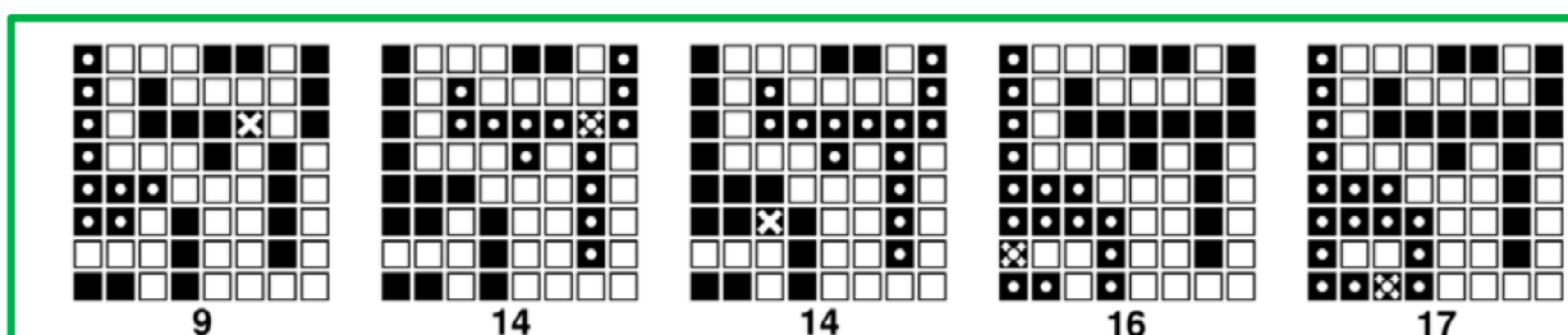
Otherwise, joint the two classes into one.

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

Black Pixels

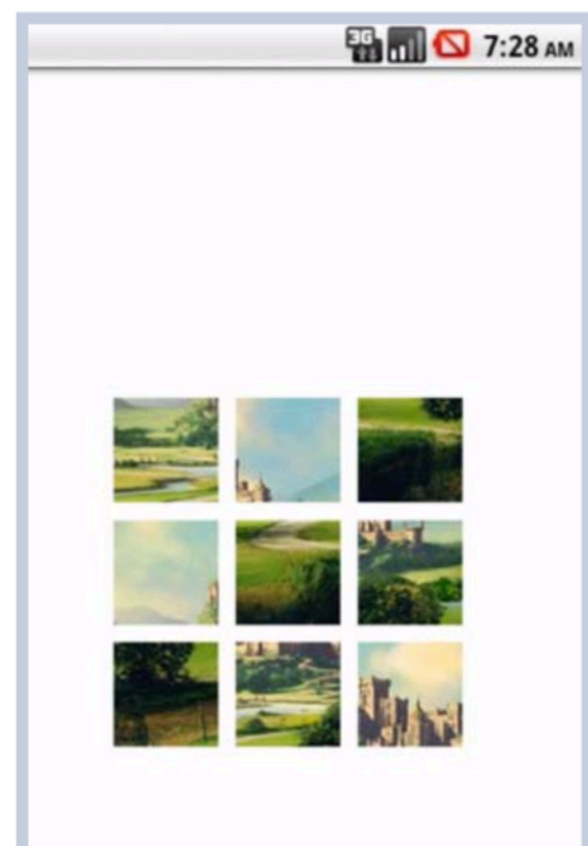
- **Maximum black pixel component**
  - Let α be the size of the component
- **Color one pixel black**
  - How α changes?
  - How to choose the pixel, to accelerate the change in α

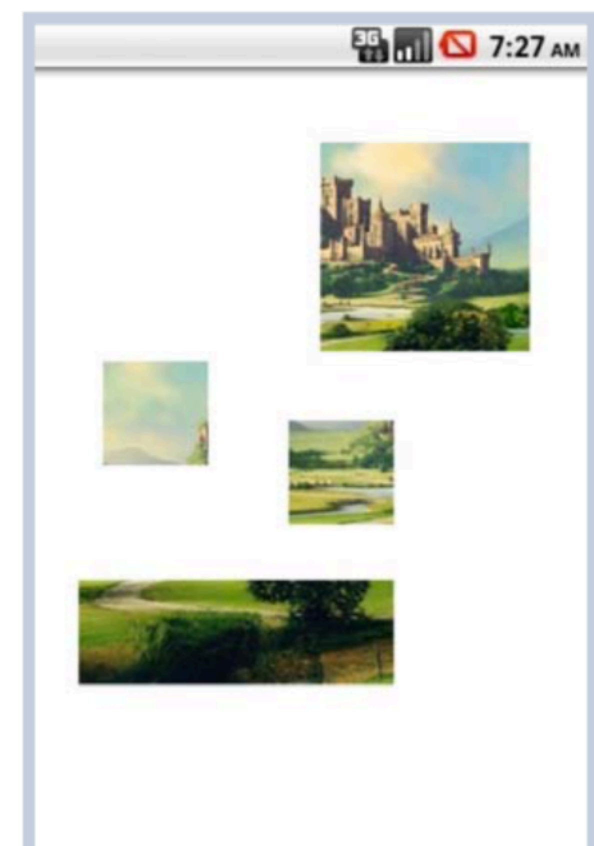




Jigsaw Puzzle

- **Multiple pieces may be glued together**
- **From “one player” to “two players”**
  - Each group can only be moved in 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 maze problem**
  - Randomly delete a wall and union two cells
  - Loop until you find the inlet and outlet are in one equivalent class
- **The Kruskal algorithm**
  - Find whether u and v are in the same equivalent class
  - If not, add the edge and union the two nodes
- **The black pixels problem**
  - Find two black pixels not in the same group
  - How the union will increase α

Implementation: Choices

- **Matrix (relation matrix)**
  - Space in $\Theta(n^2)$, and worst-case cost in $\Omega(mn)$ (mainly for row copying for MAKE/union)
- **Array (for equivalence class ID)**
  - Space in $\Theta(n)$, and worst-case cost in $\Omega(mn)$ (mainly for search and change for MAKE/union)
- **Forest of rooted trees**
  - A collection of disjoint sets, supporting Union and Find operations
  - Not necessary to traverse all the elements in one set

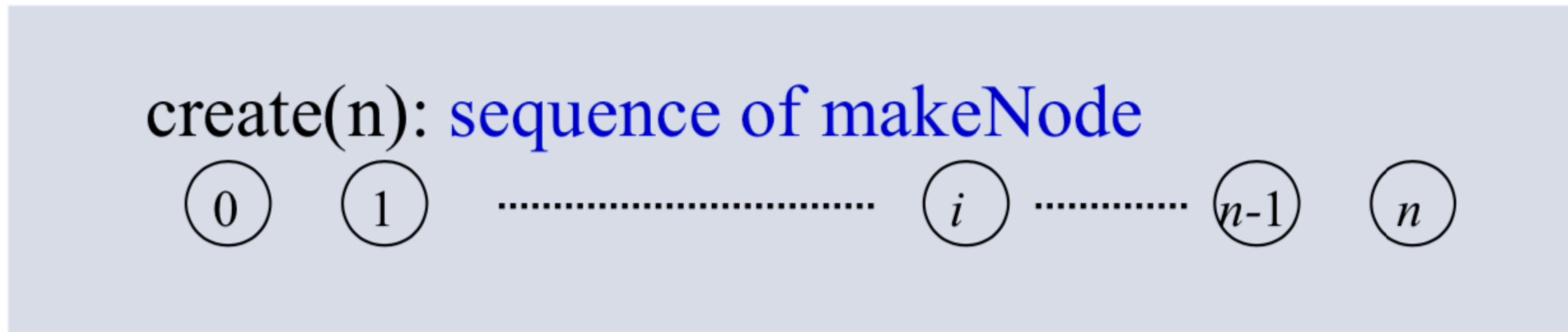
Union-Find ADT

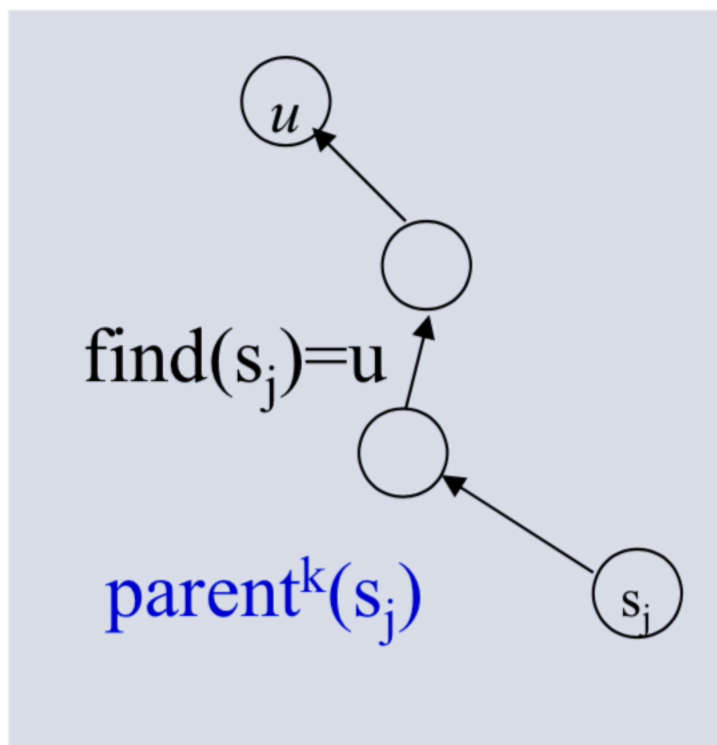
- **Constructor: Union-Find create(int n)**
  - sets = create(n) refers to a newly created group of sets {1}, {2}, …, {n} (n singletons)
- **Access Function: int find(UnionFind sets, e)**
  - find(sets, e) = <e>
- **Manipulation Procedures**
  - **void** makeSet(UnionFind sets, **int** e)
  - **void** union(UnionFind sets, **int** s, **int** t)

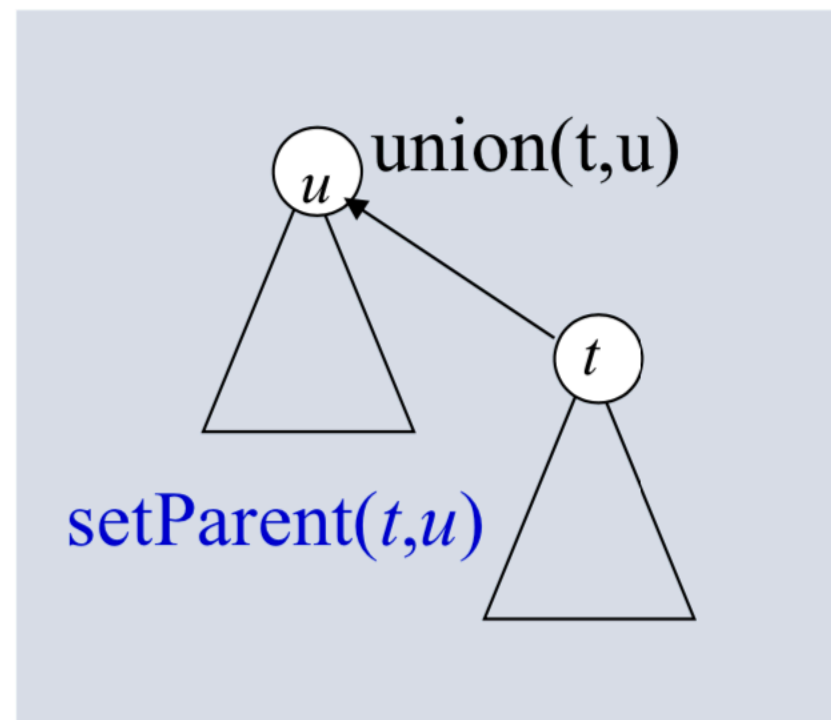
Using Rooted Tree

- **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







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 in any order
- **A union-find program is considered an input**
  - The object on 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/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 (wUnion)**
  - always have the tree with fewer nodes as subtree

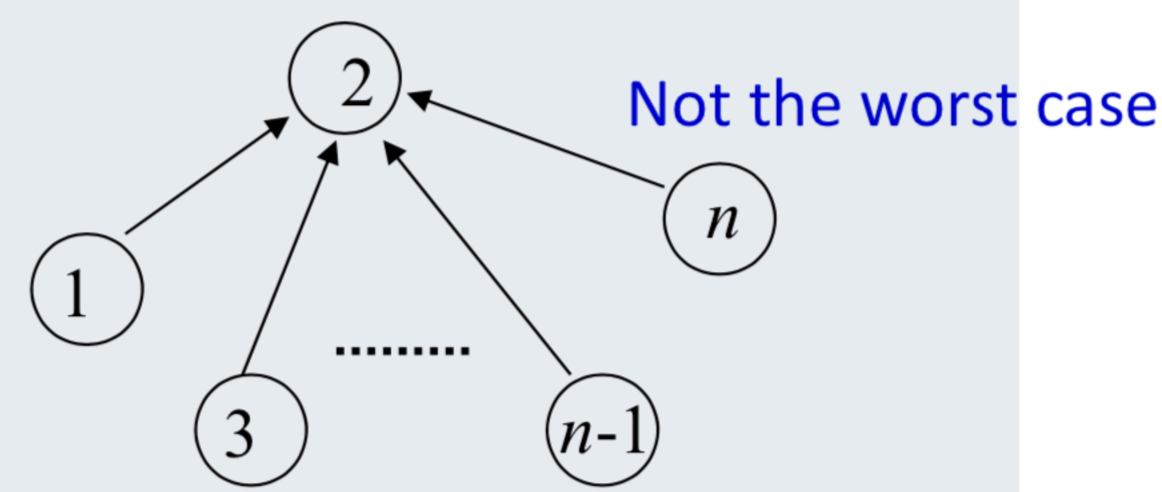
To keep the *Union* valid, each *Union* operation is replaced by:

t=find(i);

u=find(j);

union(t,u)

The order of (t,u) satisfying the requirement



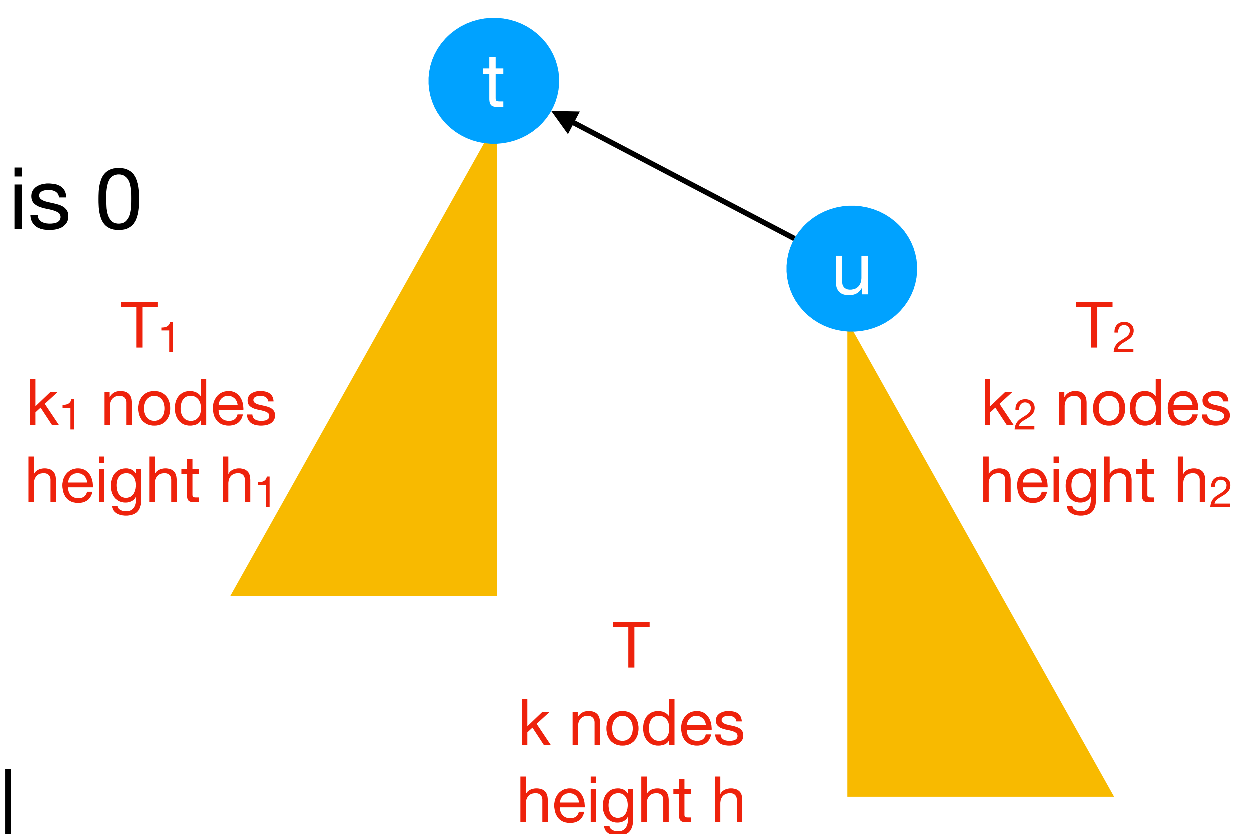
Tree made by wUnion

Cost for the program:

$n+3(n-1)+2(m-n+1)$

Upper Bound of Tree Height

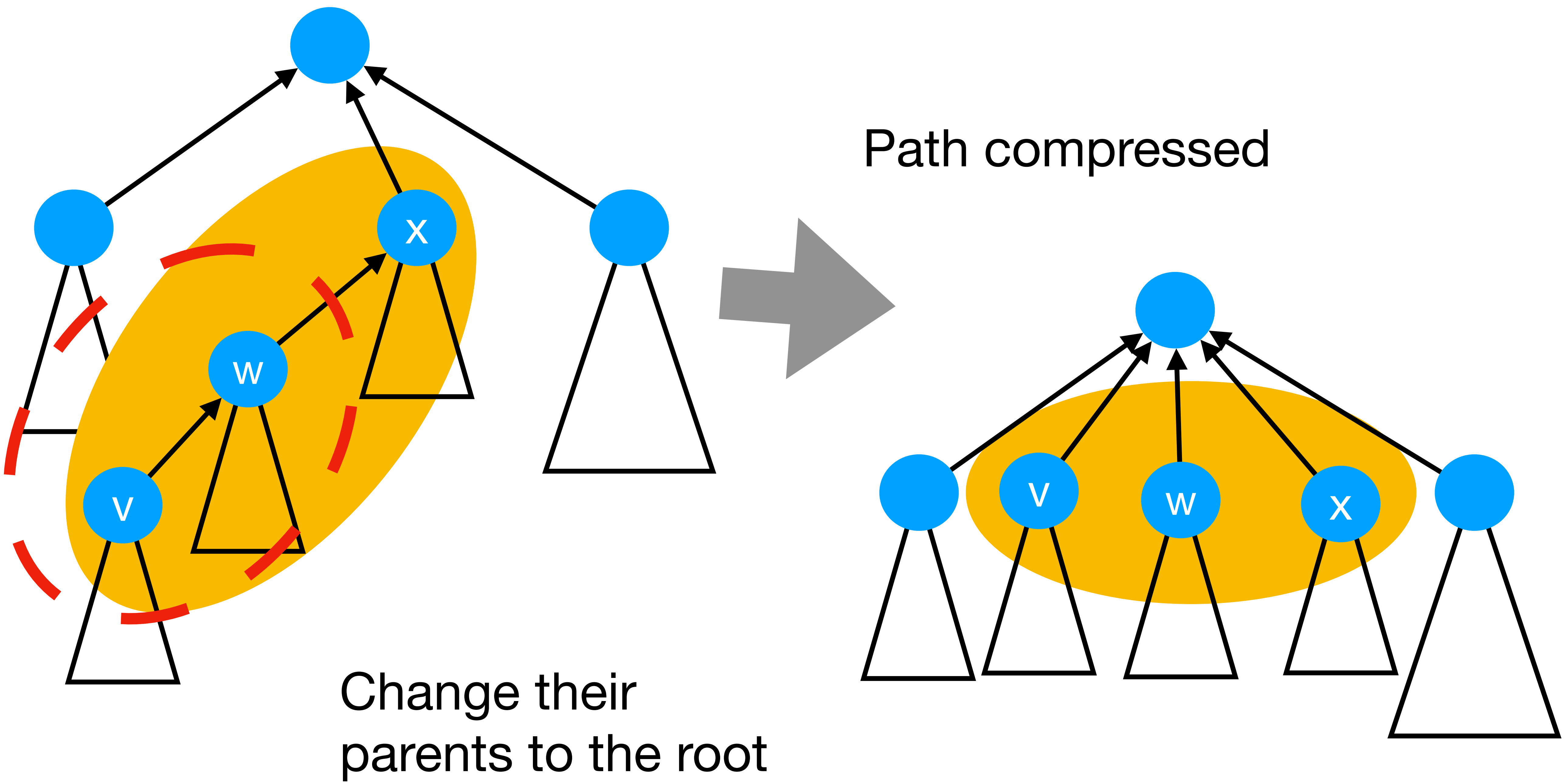
- **After any sequence of Union instructions, implemented by wUnion, any tree that has k nodes will have height at most $\lfloor \log k \rfloor$**
- **Proof by induction on k:**
  - base case: k=1, the height is 0
  - by inductive hypothesis:
    - $h_1 \leq \lfloor \lg k_1 \rfloor$, $h_2 \leq \lfloor \lg k_2 \rfloor$
  - $h = \max(h_1, h_2 + 1)$ $k = k_1 + k_2$
  - if $h = h_1$, $h_1 \leq \lfloor \lg k_1 \rfloor \leq \lfloor \lg k \rfloor$
  - if $h = h_2 + 1$, note: $k_2 \leq k/2$, so $h_2 + 1 \leq \lfloor \lg k_2 \rfloor + 1 \leq \lfloor \lg k \rfloor$



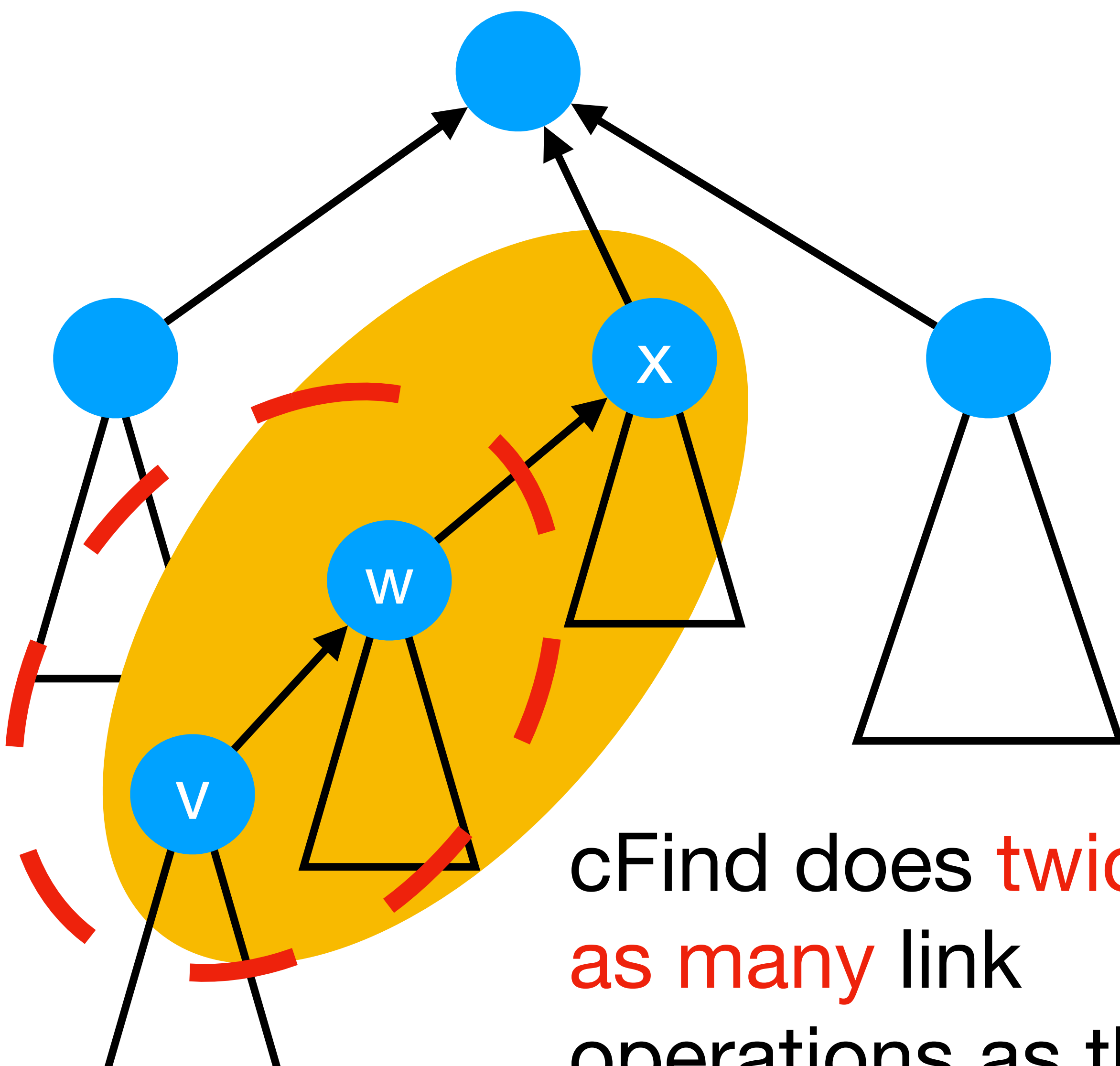
Upper Bound for Union-Find Program

- **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 *union* 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 $m(\lfloor \log n \rfloor +1) \in O(n+m\log n)$
  - There do exist programs requiring $\Omega(n+(m-n)\log n)$ steps.

Path Compression

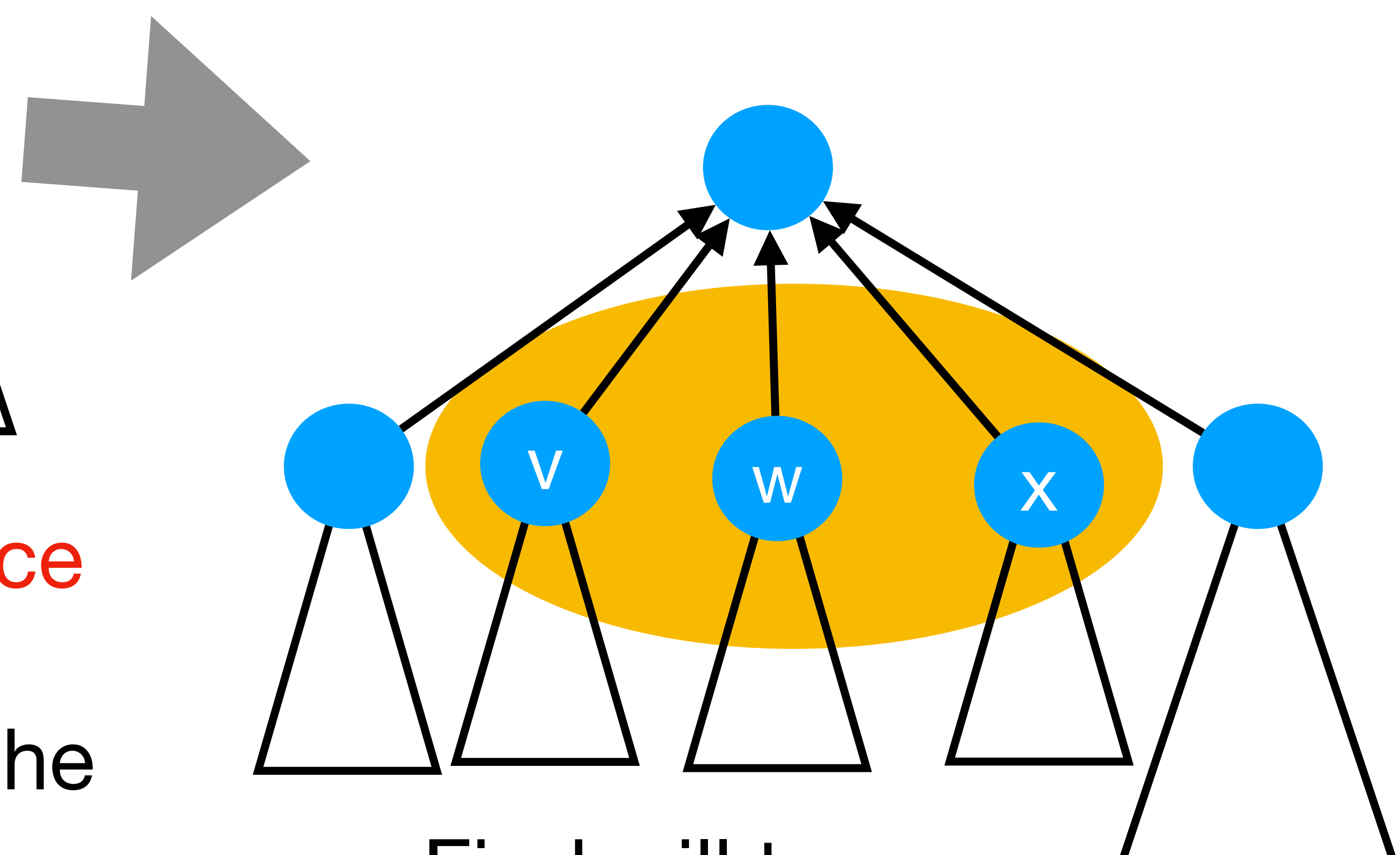


Challenges for the Analysis



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

Path compressed



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**
  - Path compressions depends on previous unions
- **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: (Ackermann)

$$H(0) = 1$$

$$H(i) = 2^{H(i-1)}$$

  - Then, log*(j) for j≥1 is defined as:

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

A function Growing Extremely Slowly

- **Function H:**

$$H(0) = 1$$

$$H(i) = 2^{H(i-1)}$$

That is: $H(k) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ k 2'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

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 finds 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 $r(r \geq 0)$ is $n/2^r$**
  - Remember that the height of the tree built by wUnion is at most $\lfloor \log n \rfloor$, 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.**
  - Note: the new parent was an ancestor of the previous parent.

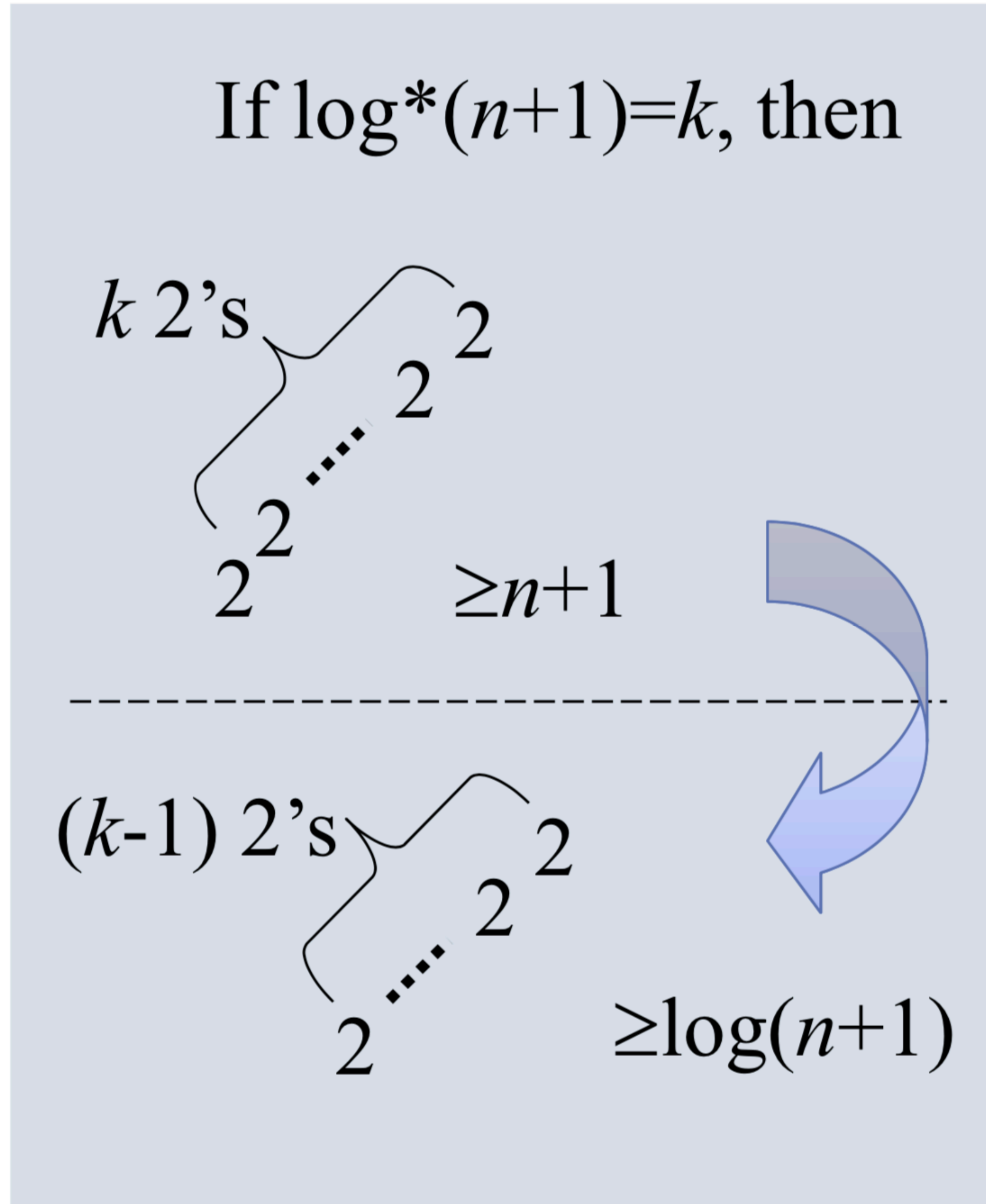
Grouping Nodes by Ranks

- **Node $v \in s_i$ ($i \geq 0$) iff. $\log^*(1+\text{rank of } v)=i$**
  - 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 [65535, ???]

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

Very Few Groups

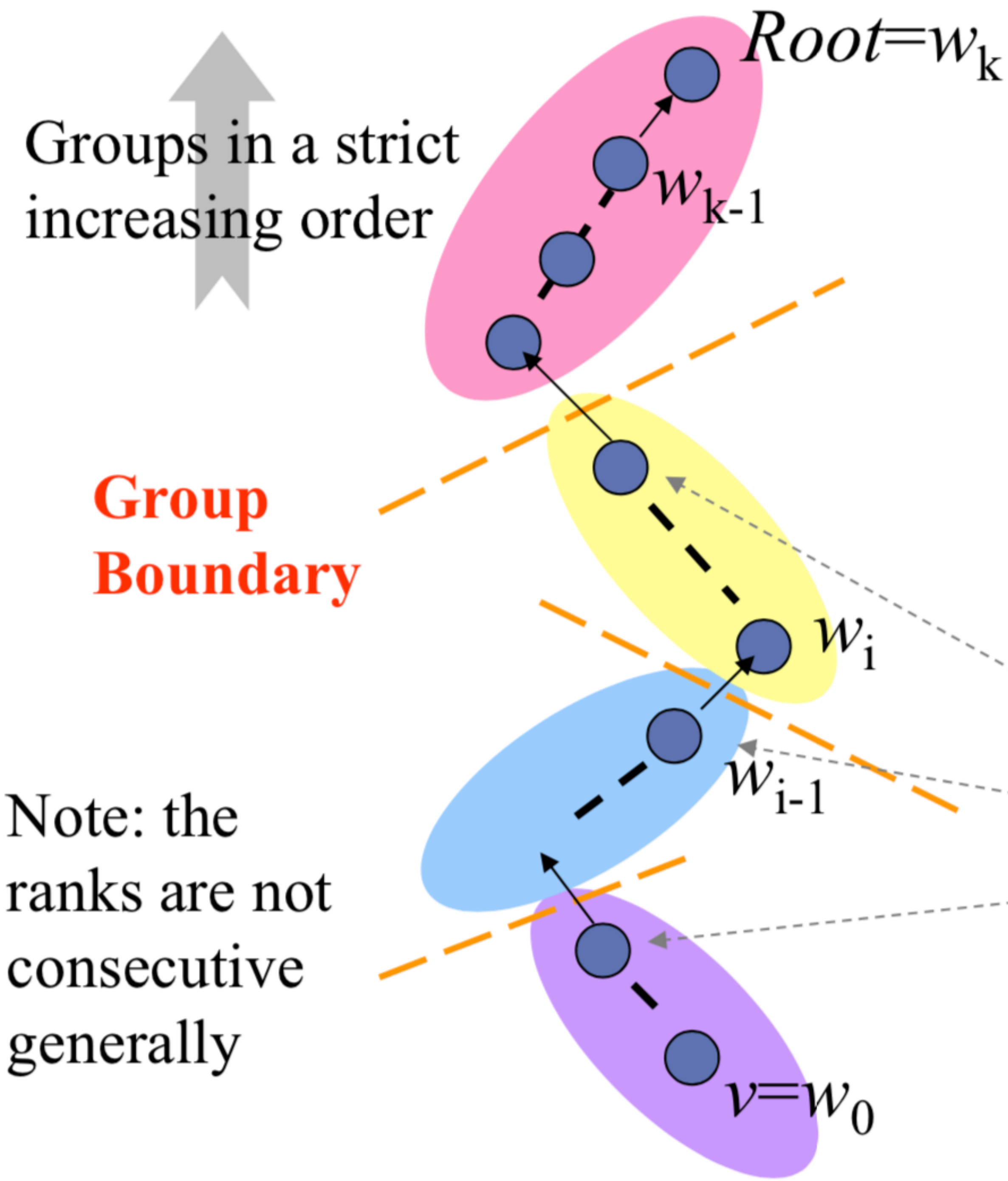
- **Node $v \in S_i$ ($i \geq 0$) iff.**
  log*(1 + rank of v)=i
- **Upper bound of the number of distinct node groups is log*(n+1)**
  - The rank of any node in F is at most $\lfloor \log n \rfloor$, so the largest group index is $\log^*(1+\lfloor \log n \rfloor)=\log^*(\lceil \log n+1 \rceil) = \log^*(n+1)-1$



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)



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 4log*(n+1)
  - So, the amortized cost is 1+4log*(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 ≤ 2k-2((k-1)-(log*(n+1)-1))=2log*(n+1)
    (Compare with the worst case cost of *cFind,* 2log$n$)

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:

a loose upper bound of ranks in a group

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

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.**
  - 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: $(n+m)(1+4\log^*(n+1))$

.

Thank you!
Q & A