CS 573: Algorithms, Fall 2013

# Union-Find

Lecture 23

November 14, 2013



## Part I

## Union Find



## Requirements from the data-structure

1. Maintain a collection of sets.
2. **makeSet**($\boldsymbol{x}$) - creates a set that contains the single element $\boldsymbol{x}$.
3. **find**(x) - returns the set that contains $\boldsymbol{x}$.
4. **union**($\boldsymbol{A}, \boldsymbol{B}$) - returns set = union of $\boldsymbol{A}$ and $\boldsymbol{B}$. That is $\boldsymbol{A} \cup \boldsymbol{B}$.
   ... merges the two sets $\boldsymbol{A}$ and $\boldsymbol{B}$ and return the merged set.

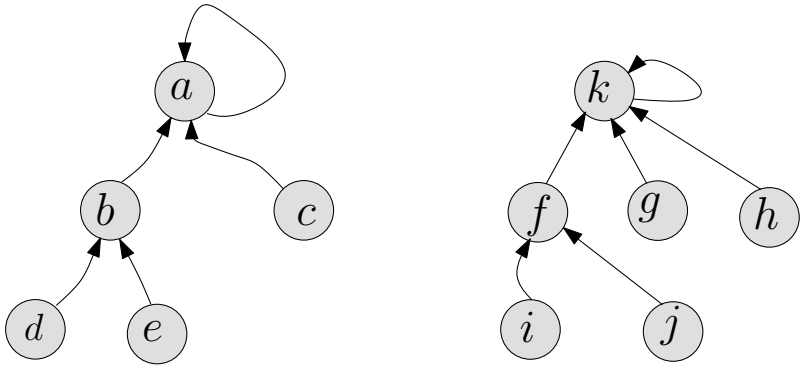


## Amortized Analysis

1. Use data-structure as a black-box inside algorithm.
   ... Union-Find in Kruskal algorithm for computing MST.
2. Bounded worst case time per operation.
3. Care: *overall* running time spend in data-structure.
4. ***amortized running-time*** of operation is the average time it takes to perform an operation on the data-structure.
5. Amortized time of an operation is $\dfrac{\text{overall running time}}{\text{number of operations}}$.

# Reversed Trees

Representing sets in the Union-Find DS




The Union-Find representation of the sets $\boldsymbol{A} = \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}\}$ and $\boldsymbol{B} = \{\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}\}$. The set $\boldsymbol{A}$ is uniquely identified by a pointer to the root of $\boldsymbol{A}$, which is the node containing $\boldsymbol{a}$.

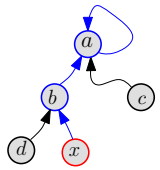


# Reversed Trees

!esrever ni retteb si gnihtyreve esuaceB

1. Reversed Trees:
   1. Every element is stored in its own node.
   2. A node has one pointer to its parent.
   3. A set is uniquely identified with the element stored in the root.
2. **makeSet**: Create a singleton pointing to itself: 
3. **find**($\boldsymbol{x}$): Start from node containing $\boldsymbol{x}$, traverse up the tree (using parent pointers), till arriving to root.

   Thus, doing a **find**($\boldsymbol{x}$) operation in the reversed tree shown on the right, involve going up the tree from $\boldsymbol{x} \rightarrow \boldsymbol{b} \rightarrow \boldsymbol{a}$, and returning $\boldsymbol{a}$ as the set.



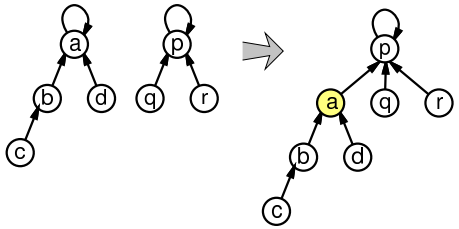



# Union operation in reversed trees

Just hang them on each other.

**union**($\boldsymbol{a}, \boldsymbol{p}$): Merge two sets.

1. Hanging the root of one tree, on the root of the other.
2. A destructive operation, and the two original sets no longer exist.






# Pseudo-code of naive version...

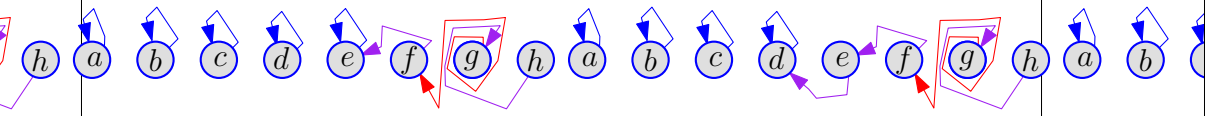
```
makeSet(x)
  p̄(x) ← x
```

```
find(x)
  if x = p̄(x) then
    return x
  return find(p̄(x))
```

```
union( x, y )
  A ← find(x)
  B ← find(y)
  p̄(B) ← A
```

# Example...

The long chain




After: **makeSet**($a$), **makeSet**($b$), **makeSet**($c$), **makeSet**($d$), **makeSet**($e$), **makeSet**($f$), **makeSet**($g$), **makeSet**($h$)
**union**($g, h$)
**union**($f, g$)
**union**($e, f$)
**union**($d, e$)
**union**($c, d$)
**union**($b, c$)
**union**($a, b$)



# Find is slow, hack it!

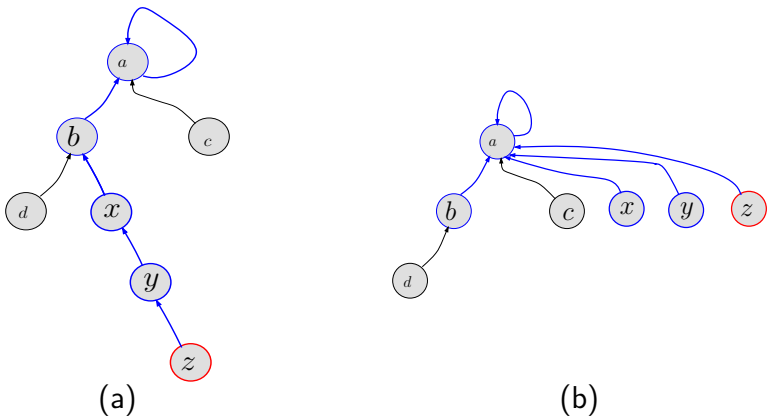
**find** might require $\Omega(n)$ time.
So, the question is how to further improve the performance of this data-structure. We are going to do this, by using two "hacks":

(i) ***Union by rank***: Maintain for every tree, in the root, a bound on its depth (called ***rank***). Always hang the smaller tree on the larger tree.

(ii) ***Path compression***: Since, anyway, we travel the path to the root during a **find** operation, we might as well hang all the nodes on the path directly on the root.



# Path compression in action...




(a) The tree before performing **find**($z$), and (b) The reversed tree after performing **find**($z$) that uses path compression.



# Pseudo-code of improved version...

```
makeSet(x)
  p̄(x) ← x
  rank(x) ← 0
```

```
find(x)
  if x ≠ p̄(x) then
    p̄(x) ← find(p̄(x))
  return p̄(x)
```

```
union(x, y )
  A ← find(x)
  B ← find(y)
  if rank(A) > rank(B) then
    p̄(B) ← A
  else
    p̄(A) ← B
    if rank(A) = rank(B) then
      rank(B) ← rank(B) + 1
```

# Part II

## Analyzing the Union-Find Data-Structure



## Definition

### Definition

A node in the union-find data-structure is a ***leader*** if it is the root of a (reversed) tree.



## Lemma

### Lemma

*Once a node stop being a leader (i.e., the node in top of a tree), it can never become a leader again.*

### Proof.

Note, that an element $x$ can stop being a leader only because of a **union** operation that hanged $x$ on an element $y$. From this point on, the only operation that might change $x$ parent pointer, is a **find** operation that traverses through $x$. Since path-compression can only change the parent pointer of $x$ to point to some other element $y$, it follows that $x$ parent pointer will never become equal to $x$ again. Namely, once $x$ stop being a leader, it can never be a leader again. □



## Another Lemma

### Lemma

*Once a node stop being a leader then its rank is fixed.*

### Proof.

The rank of an element changes only by the **union** operation. However, the **union** operation changes the rank, only for elements that are leader after the operation is done. As such, if an element is no longer a leader, than its rank is fixed. □

## Ranks are strictly monotonically increasing

**Lemma**

*Ranks are monotonically increasing in the reversed trees, as we travel from a node to the root of the tree.*



## Proof...

1. Claim: $\forall u \to v$ in DS: $\mathrm{rank}(u) < \mathrm{rank}(v)$.
2. Proof by induction. Base: all singletons. Holds.
3. Assume claim holds at time $t$, before an operation.
4. If operation is **union** $(A, B)$, and assume that we hanged $\mathrm{root}(A)$ on $\mathrm{root}(B)$.
   Must be that $\mathrm{rank}(\mathrm{root}(B))$ is now larger than $\mathrm{rank}(\mathrm{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path $\pi$, then all the nodes of $\pi$ are made to point to the last node $v$ of $\pi$.
   By induction, $\mathrm{rank}(v) >$ rank of all other nodes of $\pi$.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ■



## Trees grow exponentially in size with rank

**Lemma**

*When a node gets rank $k$ than there are at least $\geq 2^k$ elements in its subtree.*

**Proof.**

The proof is by induction. For $k = 0$ it is obvious since a singleton has a rank zero, and a single element in the set. Next observe that a node gets rank $k$ only if the merged two roots has rank $k - 1$. By induction, they have $2^{k-1}$ nodes (each one of them), and thus the merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes. □



## Having higher rank is rare

**Lemma**

*# nodes that get assigned rank $k$ throughout execution of Union-Find DS is at most $n/2^k$.*

**Proof.**

Again, by induction. For $k = 0$ it is obvious.
Charge a node $v$ of rank $k$ to two elements $u$ and $v$ of rank $k - 1$ that were leaders used to create new larger set.
After the merge $v$ is of rank $k$ and $u$ is of rank $k - 1$ and it is no longer a leader. Thus, we can charge this event to the two (no longer active) nodes of degree $k - 1$. Namely, $u$ and $v$.
By induction: algorithm created at most $n/2^{k-1}$ nodes of rank $k - 1 \implies$ # nodes of rank $k$ created by algorithm is $\leq \left(n/2^{k-1}\right)/2 = n/2^k$. □

## Find takes logarithmic time

### Lemma

*The time to perform a single* **find** *operation when we perform union by rank and path compression is* $O(\log n)$ *time.*

### Proof.

The rank of the leader $v$ of a reversed tree $T$, bounds the depth of a tree $T$ in the Union-Find data-structure. By the above lemma, if we have $n$ elements, the maximum rank is $\lg n$ and thus the depth of a tree is at most $O(\log n)$. □



## $\log^*$ in detail

$\log^*(n)$: number of times one has to take $\lg$ of a number to get a number smaller than two.
Thus, $\log^* 2 = 1$ and $\log^* 2^2 = 2$. Similarly,
$\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$. Similarly,
$\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
Things get really exciting, when one considers

$$\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5.$$

However, $\log^*$ is a monotone increasing function. And $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$ is a huge number (considerably larger than the number of atoms in the universe). Thus, for all practical purposes, $\log^*$ returns a value which is smaller than $5$.



## Can do much better!

### Theorem

*If we perform a sequence of* $m$ *operations over* $n$ *elements, the overall running time of the Union-Find data-structure is* $O((n+m)\log^* n)$.

1. Intuitively: (in the amortized sense) Union-Find data-structure takes constant time per operation (unless $n$ is larger than $\beta$ which is unlikely).
2. Not quite correct if $n$ sufficiently large...



## The tower function...

### Definition

$\mathrm{Tower}(b) = 2^{\mathrm{Tower}(b-1)}$ and $\mathrm{Tower}(0) = 1$.

$\mathrm{Tower}(i)$: a tower of $2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ of height $i$.
Observe that $\log^*(\mathrm{Tower}(i)) = i$.

### Definition

For $i \geq 0$, let $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1, \mathrm{Tower}(i)]$; that is

$$\mathrm{Block}(i) = \left[z, 2^{z-1}\right] \quad \text{for} \quad z = \mathrm{Tower}(i-1)+1.$$

Also $\mathrm{Block}(0) = [0,1]$. As such,
$\mathrm{Block}(0) = \left[0,1\right]$, $\mathrm{Block}(1) = \left[2,2\right]$, $\mathrm{Block}(2) = \left[3,4\right]$,
$\mathrm{Block}(3) = \left[5,16\right]$, $\mathrm{Block}(4) = \left[17,65536\right]$,
$\mathrm{Block}(5) = \left[65537, 2^{65536}\right] \ldots$

## Running time of find...

1. RT of **find**(x) proportional to length of the path from $x$ to the root of its tree.
2. ...since start from $x$ and we visit the sequence:
   $x_1 = x, x_2 = \overline{\mathrm{p}}(x) = \overline{\mathrm{p}}(x_1), \ldots,$
   $x_i = \overline{\mathrm{p}}(x_{i-1}), \ldots, x_m =$ root of tree.
3. $\mathrm{rank}(x_1) < \mathrm{rank}(x_2) < \mathrm{rank}(x_3) < \ldots < \mathrm{rank}(x_m)$.
4. RT of **find**($x$) is $O(m)$.

**Definition**

A node $x$ is in the $i$th block if $\mathrm{rank}(x) \in \mathrm{Block}(i)$.

5. Looking for ways to pay for the **find** operation.
   (other two operations take constant time).

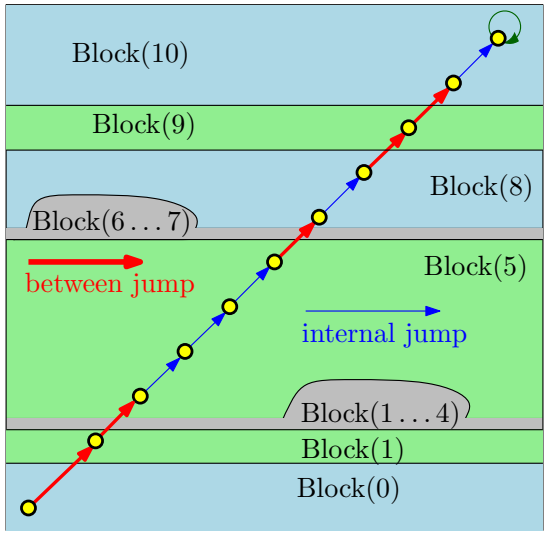


## Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.
2. # of blocks is $O(\log^* n)$, as $O(\log n) \in \mathrm{Block}(c \log^* n)$, ($c$: constant).
3. **find** ($x$): $\pi$ path used.
4. partition $\pi$ into each by rank.
5. Price of **find** length $\pi$.
6. For node $x$: $\nu = \mathrm{index}_{\mathrm{B}}(x)$ index of block containing $\mathrm{rank}(x)$. $\mathrm{rank}(x) \in \mathrm{Block}\big(\mathrm{index}_{\mathrm{B}}(x)\big)$.
7. $\mathrm{index}_{\mathrm{B}}(x)$: ***block of x***



## The path of find operation, and its pointers






## The pointers between blocks...

1. During a **find** operation...
2. $\pi$: path traversed.
3. Ranks of the nodes visited in $\pi$ monotone increasing.
4. Once leave block $i$th, never go back!
5. charge visit to nodes in $\pi$ next to element in a different block...
6. to total number of blocks $\leq O(\log^* n)$.

## Jumping pointers

### Definition

$\pi$: path traversed by **find**.
$x \in \pi$, $\overline{\mathrm{p}}(x)$ is in a different block, is a ***jump between blocks***.
jump inside a block is an ***internal jump*** (i.e., $x$ and $\overline{\mathrm{p}}(x)$ are in same block).



## Not too many jumps between blocks

### Lemma

*During a single* **find**$(x)$ *operation, the number of jumps between blocks along the search path is* $O(\log^* n)$.

### Proof.

Consider the search path $\pi = x_1, \ldots, x_m$, and consider the list of numbers $0 \leq \mathrm{index}_{\mathrm{B}}(x_1) \leq \mathrm{index}_{\mathrm{B}}(x_2) \leq \ldots \leq \mathrm{index}_{\mathrm{B}}(x_m)$. We have that $\mathrm{index}_{\mathrm{B}}(x_m) = O(\log^* n)$. As such, the number of elements $x$ in $\pi$ such that $\mathrm{index}_{\mathrm{B}}(x) \neq \mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x))$ is at most $O(\log^* n)$. □



## Benefits of an internal jump

1. $x$ and $\overline{\mathrm{p}}(x)$ are in same block.
2. $\mathrm{index}_{\mathrm{B}}(x) = \mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x))$.
3. **find** passes through $x$.
4. $r_{\mathrm{bef}} = \mathrm{rank}(\overline{\mathrm{p}}(x))$ before **find** operation.
5. $r_{\mathrm{aft}} = \mathrm{rank}(\overline{\mathrm{p}}(x))$ after **find** operation.
6. By path compression: $r_{\mathrm{aft}} > r_{\mathrm{bef}}$.
7. $\implies$ parent pointer of $x$ jump forward and the new parent has higher rank.
8. Charge internal block jumps to this "progress".



## Your parent can be promoted only a few times before leaving block

### Lemma

*At most* $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$ **find** *operations can pass through an element* $x$*, which is in the* $i$*th block (i.e.,* $\mathrm{index}_{\mathrm{B}}(x) = i$*) before* $\overline{\mathrm{p}}(x)$ *is no longer in the* $i$*th block. That is* $\mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x)) > i$.

### Proof.

By above discussion, the parent of $x$ increases its rank every-time an internal jump goes through $x$. Since there at most $|\mathrm{Block}(i)|$ different values in the $i$th block, the claim follows. The inequality $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$ holds by definition:

$$\mathrm{Block}(i) = [\mathrm{Tower}(i-1) + 1, \mathrm{Tower}(i)]$$

□

## Few elements are in the bigger blocks

### Lemma

*There are at most $n/\mathrm{Tower}(i)$ nodes that have ranks in the $i$th block throughout the algorithm execution.*

### Proof.

By lemma, the number of elements with rank in the $i$th block

$$\leq \sum_{k \in \mathrm{Block}(i)} \frac{n}{2^k} = \sum_{k=\mathrm{Tower}(i-1)+1}^{\mathrm{Tower}(i)} \frac{n}{2^k}$$

$$= n \cdot \sum_{k=\mathrm{Tower}(i-1)+1}^{\mathrm{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\mathrm{Tower}(i-1)}} = \frac{n}{\mathrm{Tower}(i)}.$$

□



## Total number of internal jumps is O(n)

### Lemma

*The number of internal jumps performed, inside the $i$th block, during the lifetime of the union-find data-structure is $O(n)$.*

### Proof.

An element $x$ in the $i$th block, can have at most $|\mathrm{Block}(i)|$ internal jumps, before all jumps through $x$ are jumps between blocks, by lemma... There are at most $n/\mathrm{Tower}(i)$ elements with ranks in the $i$th block, throughout the algorithm execution, by other Lemma. Thus, the total number of internal jumps is

$$|\mathrm{Block}(i)| \cdot \frac{n}{\mathrm{Tower}(i)} \leq \mathrm{Tower}(i) \cdot \frac{n}{\mathrm{Tower}(i)} = n.$$

□



## Total number of internal jumps

### Lemma

*The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.*

### Proof.

Every internal jump can be associated with the block it is being performed in. Every block contributes $O(n)$ internal jumps throughout the execution of the union-find data-structures, by Lemma 23.3.16. There are $O(\log^* n)$ blocks. As such there are at most $O(n \log^* n)$ internal jumps. □



## Result...

### Lemma

*The overall time spent on $m$ **find** operations, throughout the lifetime of a union-find data-structure defined over $n$ elements, is $O((m+n) \log^* n)$.*

### Theorem

*If we perform a sequence of $m$ operations over $n$ elements, the overall running time of the Union-Find data-structure is $O((n+m) \log^* n)$.*