Union-Find

Lecture 23

November 14, 2013

Part I

Union Find

Requirements from the data-structure

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

Requirements from the data-structure

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

Requirements from the data-structure

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

Requirements from the data-structure

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

Requirements from the data-structure

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

Requirements from the data-structure

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

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 $\frac{\text{overall running time}}{\text{number of operations}}$.

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 $\frac{\text{overall running time}}{\text{number of operations}}$.

Amortized Analysis

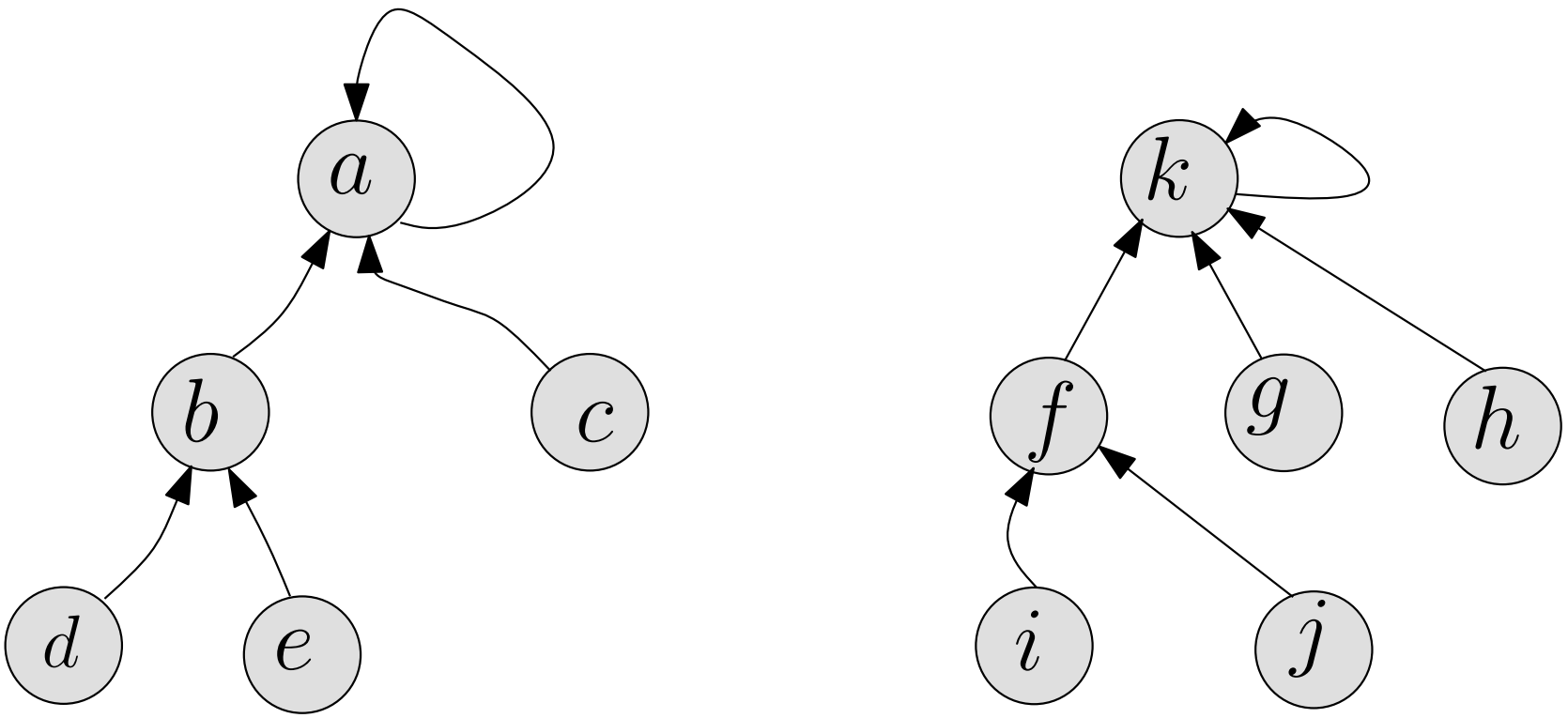
1. Use data-structure as a black-box inside algorithm.
   ... Union-Find in Kruskal algorithm for computing MST.
2. Bounded worst case time per operation.
3. Care: *overall* running time spend in data-structure.
4. ***amortized running-time*** of operation is the average time it takes to perform an operation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

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 $\frac{\text{overall running time}}{\text{number of operations}}$.

Reversed Trees

Representing sets in the Union-Find DS



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

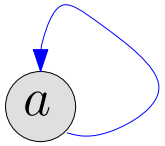
Reversed Trees

!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.
3.

Reversed Trees

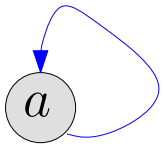
!esrever ni retteb si gnihtyreve esuaceB

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.

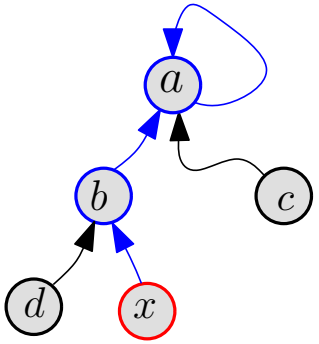
Reversed Trees

!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**(x): Start from node containing x, traverse up the tree (using parent pointers), till arriving to root.

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

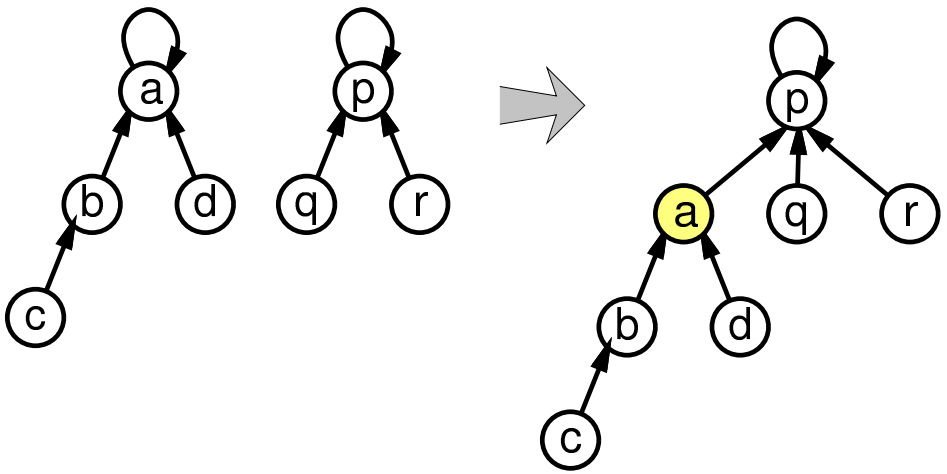


Union operation in reversed trees

Just hang them on each other.

union(a, 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...

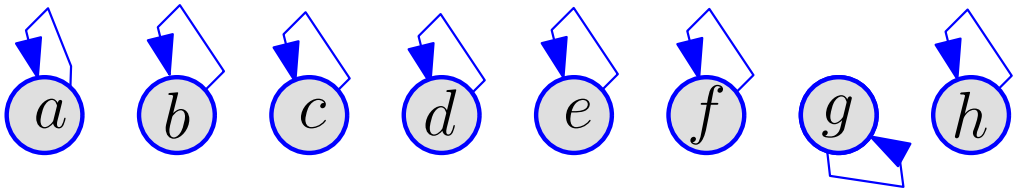
```
makeSet(x)
  p̄(x) ← x
```

```
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...

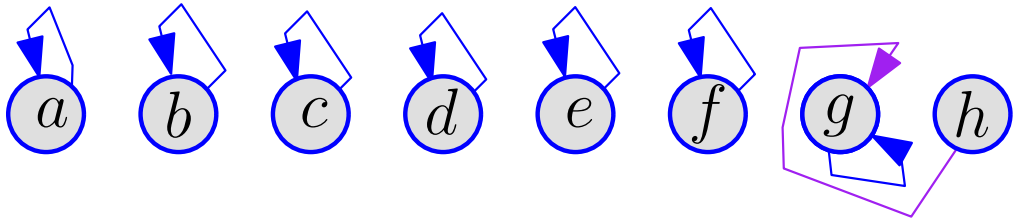
The long chain



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

Example...

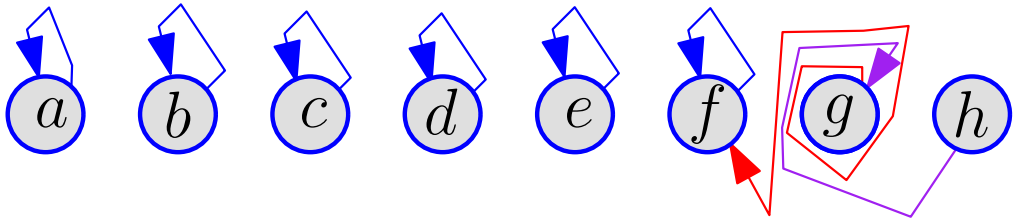
The long chain



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

Example...

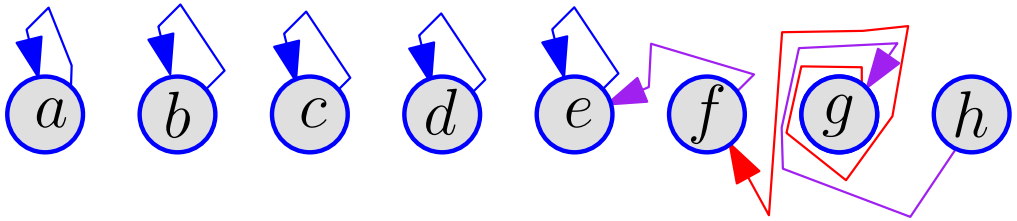
The long chain



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

Example...

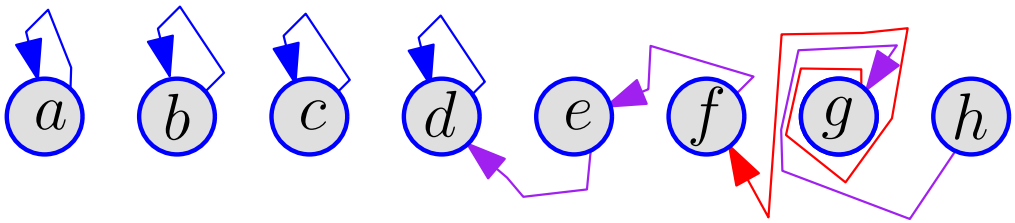
The long chain



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)

Example...

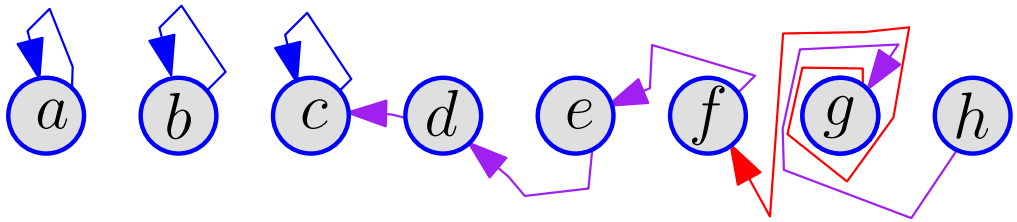
The long chain



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)

Example...

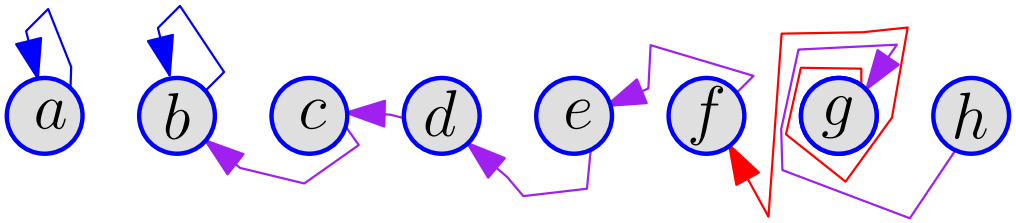
The long chain



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)

Example...

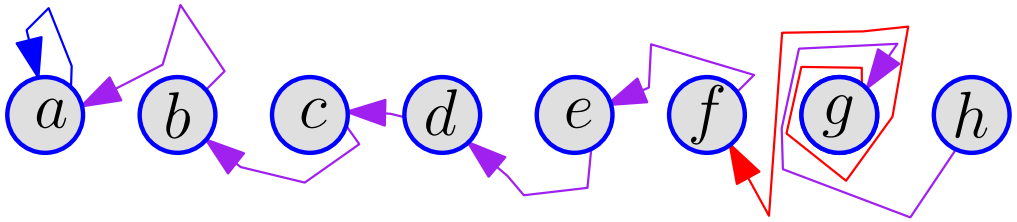
The long chain



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)

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)

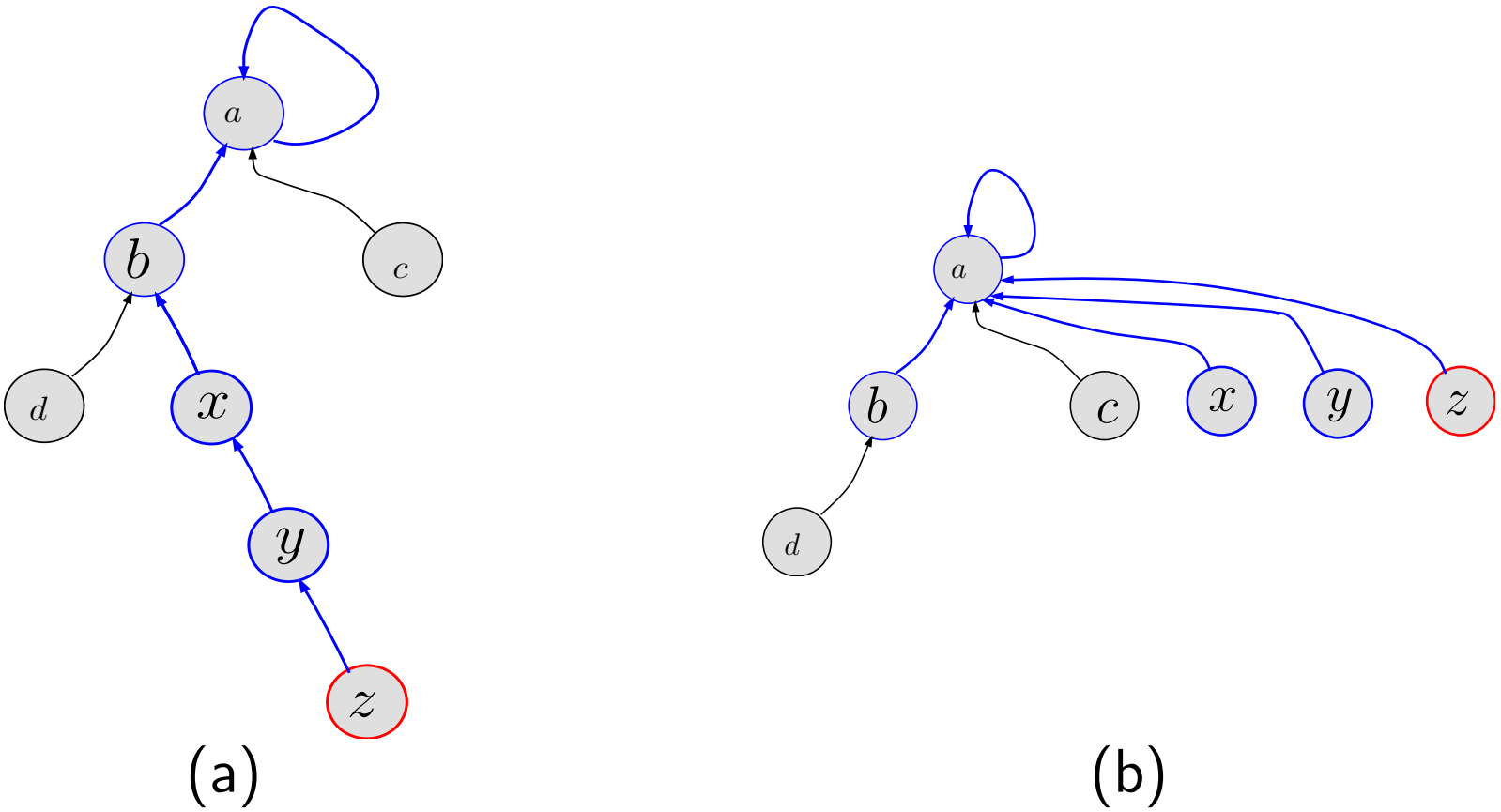
Find is slow, hack it!

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: $\mathbf{rank}(u) < \mathbf{rank}(v)$.
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 $\mathbf{root}(A)$ on $\mathbf{root}(B)$.
   Must be that $\mathbf{rank}(\mathbf{root}(B))$ is now larger than $\mathbf{rank}(\mathbf{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathbf{rank}(v) >$ rank of all other nodes of π.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ∎

Proof...

1. Claim: $\forall u \to v$ in DS: $\mathbf{rank}(u) < \mathbf{rank}(v)$.
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 $\mathbf{root}(A)$ on $\mathbf{root}(B)$.
   Must be that $\mathbf{rank}(\mathbf{root}(B))$ is now larger than $\mathbf{rank}(\mathbf{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathbf{rank}(v) >$ rank of all other nodes of π.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ∎

Proof...

1. Claim: $\forall u \to v$ in DS: $\mathbf{rank}(u) < \mathbf{rank}(v)$.
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 $\mathbf{root}(A)$ on $\mathbf{root}(B)$.
   Must be that $\mathbf{rank}(\mathbf{root}(B))$ is now larger than $\mathbf{rank}(\mathbf{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathbf{rank}(v) >$ rank of all other nodes of π.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ∎

Proof...

1. Claim: $\forall u \to v$ in DS: $\mathbf{rank}(u) < \mathbf{rank}(v)$.
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 $\mathbf{root}(A)$ on $\mathbf{root}(B)$.
   Must be that $\mathbf{rank}(\mathbf{root}(B))$ is now larger than $\mathbf{rank}(\mathbf{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathbf{rank}(v) >$ rank of all other nodes of π.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ∎

Proof...

1. Claim: $\forall u \to v$ in DS: $\mathbf{rank}(u) < \mathbf{rank}(v)$.
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 $\mathbf{root}(A)$ on $\mathbf{root}(B)$.
   Must be that $\mathbf{rank}(\mathbf{root}(B))$ is now larger than $\mathbf{rank}(\mathbf{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathbf{rank}(v) >$ rank of all other nodes of π.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ∎

Proof...

1. Claim: $\forall u \to v$ in DS: $\mathbf{rank}(u) < \mathbf{rank}(v)$.
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 $\mathbf{root}(A)$ on $\mathbf{root}(B)$.
   Must be that $\mathbf{rank}(\mathbf{root}(B))$ is now larger than $\mathbf{rank}(\mathbf{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathbf{rank}(v) >$ rank of all other nodes of π.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ∎

Proof...

1. Claim: $\forall u \to v$ in DS: $\mathbf{rank}(u) < \mathbf{rank}(v)$.
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 $\mathbf{root}(A)$ on $\mathbf{root}(B)$.
   Must be that $\mathbf{rank}(\mathbf{root}(B))$ is now larger than $\mathbf{rank}(\mathbf{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathbf{rank}(v) >$ rank of all other nodes of π.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ∎

Proof...

1. Claim: $\forall u \to v$ in DS: $\mathbf{rank}(u) < \mathbf{rank}(v)$.
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 $\mathbf{root}(A)$ on $\mathbf{root}(B)$.
   Must be that $\mathbf{rank}(\mathbf{root}(B))$ is now larger than $\mathbf{rank}(\mathbf{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathbf{rank}(v) >$ rank of all other nodes of π.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ∎

Proof...

1. Claim: $\forall u \to v$ in DS: $\mathbf{rank}(u) < \mathbf{rank}(v)$.
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 $\mathbf{root}(A)$ on $\mathbf{root}(B)$.
   Must be that $\mathbf{rank}(\mathbf{root}(B))$ is now larger than $\mathbf{rank}(\mathbf{root}(A))$ (verify!).
   Claim true after operation!
5. If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathbf{rank}(v) >$ rank of all other nodes of π.
   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. $\square$

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 $\leq \left(n/2^{k-1}\right)/2 = n/2^k$. □

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 $\leq \left(n/2^{k-1}\right)/2 = n/2^k$. □

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 $\leq \left(n/2^{k-1}\right)/2 = n/2^k$. $\square$

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 β which is unlikely).
2. Not quite correct if n sufficiently large...

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 β which is unlikely).
2. Not quite correct if n sufficiently large...

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 β which is unlikely).
2. Not quite correct if n sufficiently large...

The tower function...

Definition

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

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

Definition

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

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

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

The tower function...

Definition

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

$\mathbf{Tower}(i)$: a tower of $2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ of height i.

Observe that $\log^*(\mathbf{Tower}(i)) = i$.

Definition

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

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

Also $\mathbf{Block}(0) = [0, 1]$. As such,

$\mathbf{Block}(0) = \left[0, 1\right]$, $\mathbf{Block}(1) = \left[2, 2\right]$, $\mathbf{Block}(2) = \left[3, 4\right]$,

$\mathbf{Block}(3) = \left[5, 16\right]$, $\mathbf{Block}(4) = \left[17, 65536\right]$,

$\mathbf{Block}(5) = \left[65537, 2^{65536}\right] \ldots$

The tower function...

Definition

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

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

Definition

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

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

Also $\mathbf{Block}(0) = [0, 1]$. As such,

$\mathbf{Block}(0) = \left[0, 1\right]$, $\mathbf{Block}(1) = \left[2, 2\right]$, $\mathbf{Block}(2) = \left[3, 4\right]$,

$\mathbf{Block}(3) = \left[5, 16\right]$, $\mathbf{Block}(4) = \left[17, 65536\right]$,

$\mathbf{Block}(5) = \left[65537, 2^{65536}\right] \ldots$

The tower function...

Definition

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

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

Definition

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

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

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

The tower function...

Definition

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

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

Definition

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

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

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

The tower function...

Definition

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

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

Definition

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

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

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

The tower function...

Definition

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

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

Definition

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

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

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

The tower function...

Definition

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

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

Definition

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

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

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

The tower function...

Definition

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

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

Definition

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

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

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

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 ith 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).

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 ith 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).

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. $\mathbf{rank}(x_1) < \mathbf{rank}(x_2) < \mathbf{rank}(x_3) < \ldots < \mathbf{rank}(x_m)$.
4. RT of **find**(x) is $O(m)$.

Definition

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

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

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 ith 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).

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 ith 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).

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:
   $x_1 = x, x_2 = \overline{\mathrm{p}}(x) = \overline{\mathrm{p}}(x_1), ...,$
   $x_i = \overline{\mathrm{p}}(x_{i-1}), \ldots, x_m =$ 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 ith 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): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
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

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): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
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

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): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
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

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): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
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

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): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
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

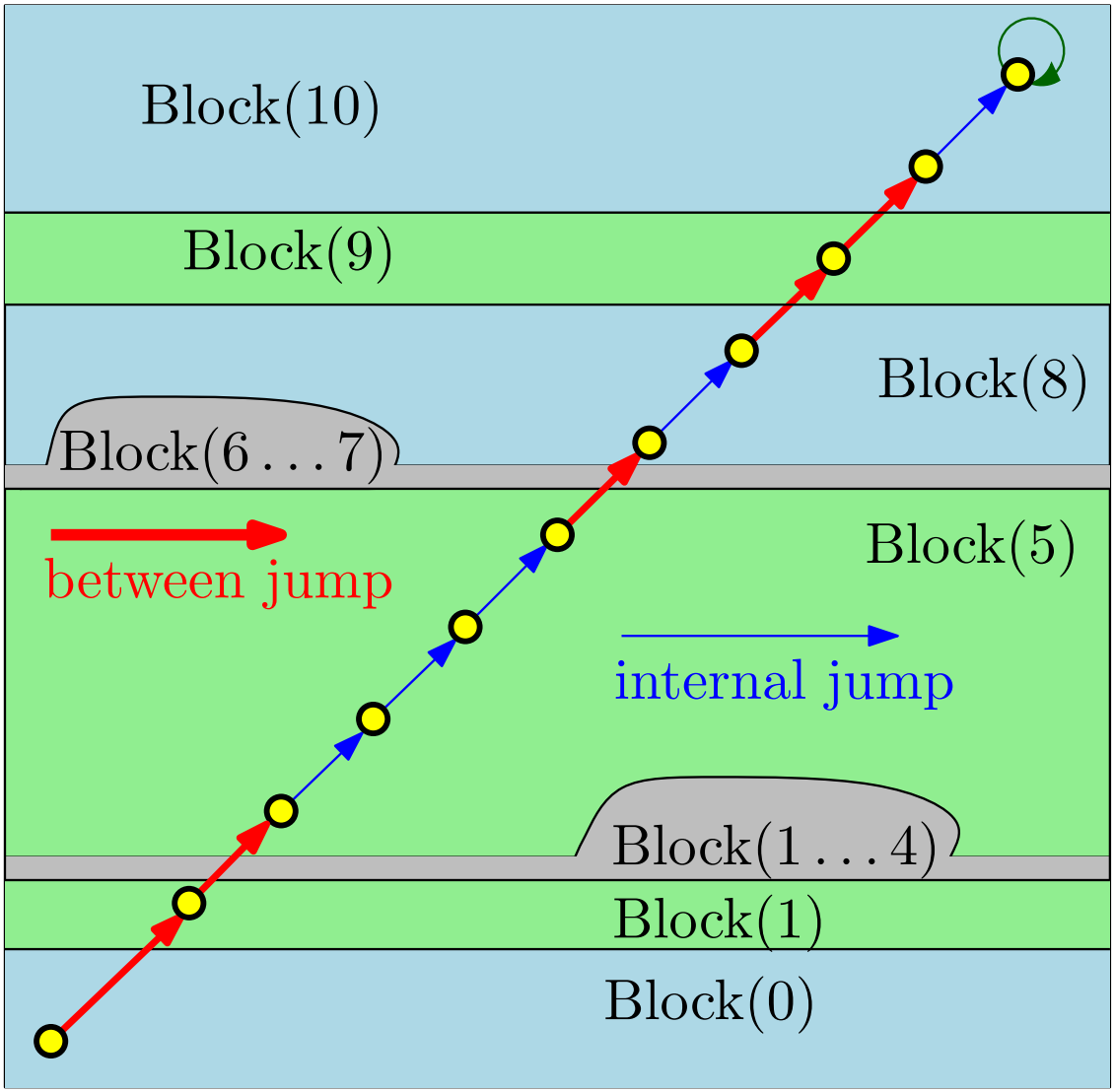
Blocks and jumping pointers

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): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
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

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): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
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. π: path traversed.
3. Ranks of the nodes visited in π monotone increasing.
4. Once leave block ith, never go back!
5. charge visit to nodes in π next to element in a different block...
6. to total number of blocks $\leq O(\log^* n)$.

The pointers between blocks...

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

The pointers between blocks...

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

The pointers between blocks...

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

Jumping pointers

Definition

π: 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 π such that $\mathrm{index}_{\mathrm{B}}(x) \neq \mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x))$ is at most $O(\log^* n)$. $\square$

Benefits of an internal jump

1. x and $\overline{\mathrm{p}}(x)$ are in same block.
2. $\mathbf{index_B}(x) = \mathbf{index_B}(\overline{\mathrm{p}}(x))$.
3. **find** passes through x.
4. $r_{\mathrm{bef}} = \mathbf{rank}(\overline{\mathrm{p}}(x))$ before **find** operation.
5. $r_{\mathrm{aft}} = \mathbf{rank}(\overline{\mathrm{p}}(x))$ 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".

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".

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".

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".

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. $\Longrightarrow$ parent pointer of x jump forward and the new parent has higher rank.
8. Charge internal block jumps to this "progress".

Benefits of an internal jump

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

Benefits of an internal jump

1. x and $\overline{\mathrm{p}}(x)$ are in same block.
2. $\mathbf{index_B}(x) = \mathbf{index_B}(\overline{\mathrm{p}}(x))$.
3. **find** passes through x.
4. $r_{\mathbf{bef}} = \mathbf{rank}(\overline{\mathrm{p}}(x))$ before **find** operation.
5. $r_{\mathbf{aft}} = \mathbf{rank}(\overline{\mathrm{p}}(x))$ after **find** operation.
6. By path compression: $r_{\mathbf{aft}} > r_{\mathbf{bef}}$.
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 $|\mathbf{Block}(i)| \leq \mathbf{Tower}(i)$ **find** *operations can pass through an element* x*, which is in the* i*th block (i.e.,* $\mathbf{index_B}(x) = i$*) before* $\overline{\mathrm{p}}(x)$ *is no longer in the* i*th block. That is* $\mathbf{index_B}(\overline{\mathrm{p}}(x)) > i$*.*

Proof.

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

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

□

Few elements are in the bigger blocks

Lemma

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

Proof.

By lemma, the number of elements with rank in the ith 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 $\mathbf{O(n)}$

Lemma

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

Proof.

An element x in the ith 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 ith 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 is $\mathbf{O(n)}$

Lemma

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

Proof.

An element x in the ith 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 ith 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 is $\mathbf{O(n)}$

Lemma

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

Proof.

An element x in the ith 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 ith 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 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 $O((n+m)\log^* n)$.

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* $O((n+m)\log^* n)$.

Notes

Notes

Notes

Notes