Union-Find

Lecture 15

October 15, 2015

15.1: Union Find

15.2: Kruskal's algorithm – a quick reminder

Compute minimum spanning tree

1. **G**: Undirected graph with weights on edges.
2. Q: Compute MST (minimum spanning tree) of **G**.
3. Kruskal's Algorithm:
   1. Sort edges by increasing weight.
   2. Start with a copy of **G** with no edges.
   3. Add edges by increasing weight, and insert into graph $\iff$ do not form a cycle.
      (i.e., connect two different things together.)

Compute minimum spanning tree

1. **G**: Undirected graph with weights on edges.
2. Q: Compute MST (minimum spanning tree) of **G**.
3. Kruskal's Algorithm:
    1. Sort edges by increasing weight.
    2. Start with a copy of **G** with no edges.
    3. Add edges by increasing weight, and insert into graph $\iff$ do not form a cycle.
       (i.e., connect two different things together.)

Compute minimum spanning tree

1. **G**: Undirected graph with weights on edges.
2. Q: Compute MST (minimum spanning tree) of **G**.
3. Kruskal's Algorithm:
   1. Sort edges by increasing weight.
   2. Start with a copy of **G** with no edges.
   3. Add edges by increasing weight, and insert into graph $\iff$ do not form a cycle.
      (i.e., connect two different things together.)

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

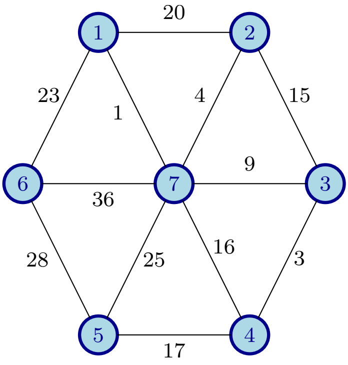


Figure: Graph G

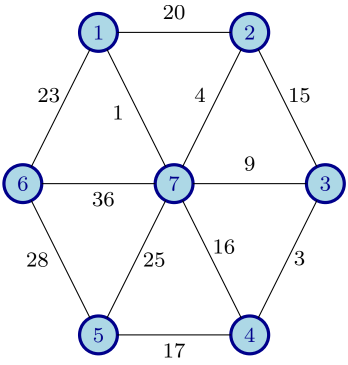


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

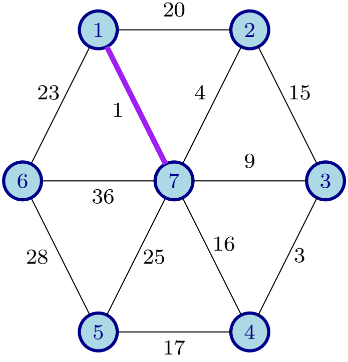


Figure: Graph G

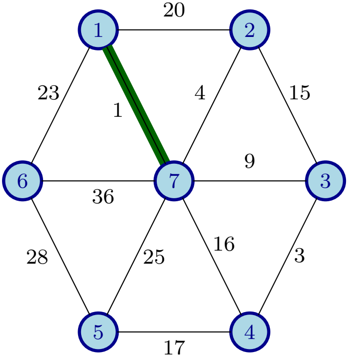


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

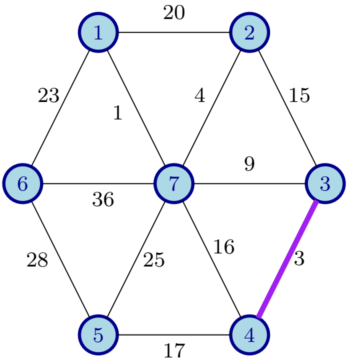


Figure: Graph G

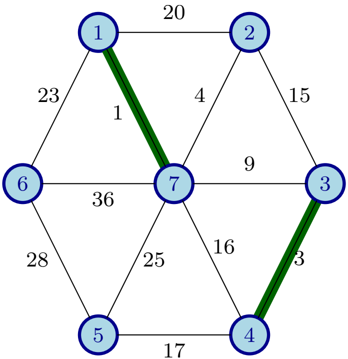


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

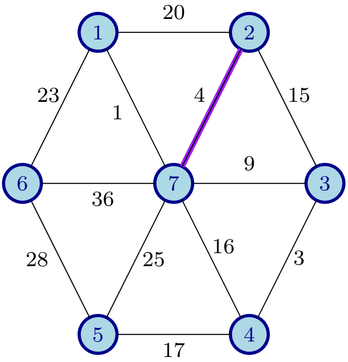


Figure: Graph G

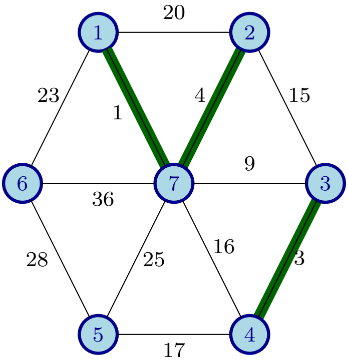


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

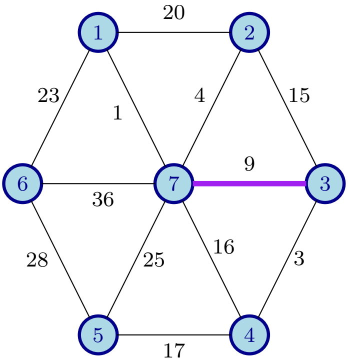


Figure: Graph G

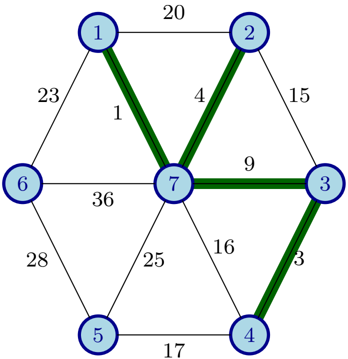


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

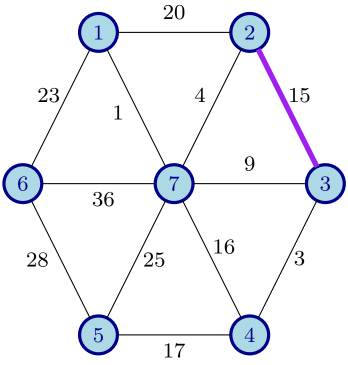


Figure: Graph G

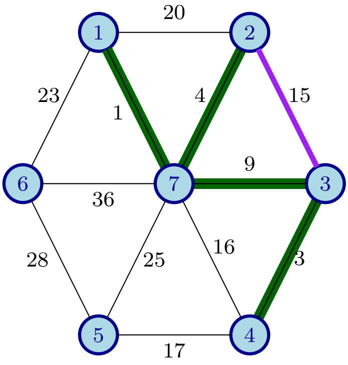


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

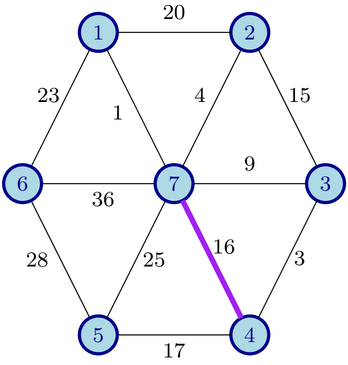


Figure: Graph G

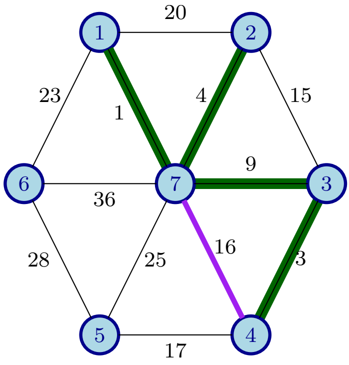


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

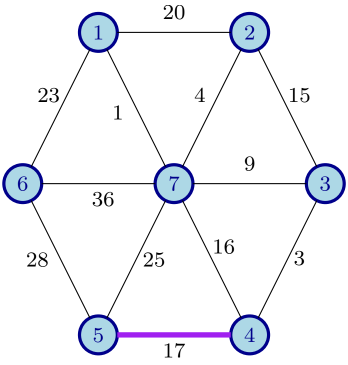


Figure: Graph G

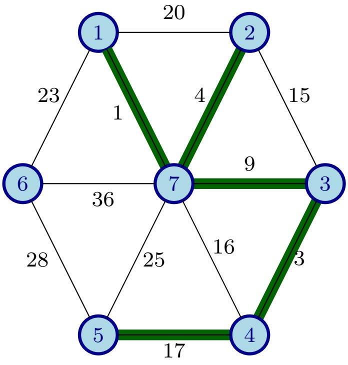


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

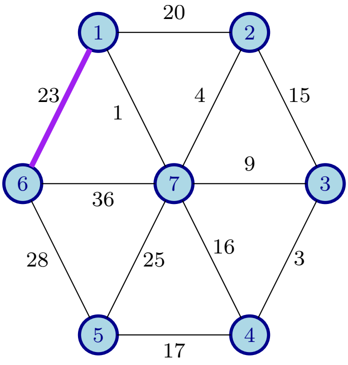


Figure: Graph G

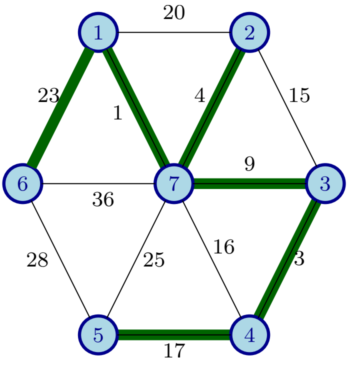


Figure: MST of G

15.2.1: Requirements from the data-structure

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.

15.2.2: Amortized analysis

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
   = average time to perform an operation on data-structure.
5. Amortized time per operation $= \dfrac{\text{overall running time}}{\text{number of operations}}$.

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
   = average time to perform an operation on data-structure.
5. Amortized time per operation $= \dfrac{\text{overall running time}}{\text{number of operations}}$.

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
   = average time to perform an operation on data-structure.
5. Amortized time per operation $= \frac{\text{overall running time}}{\text{number of operations}}$.

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
   $=$ average time to perform an operation on data-structure.
5. Amortized time per operation $= \dfrac{\text{overall running time}}{\text{number of operations}}$.

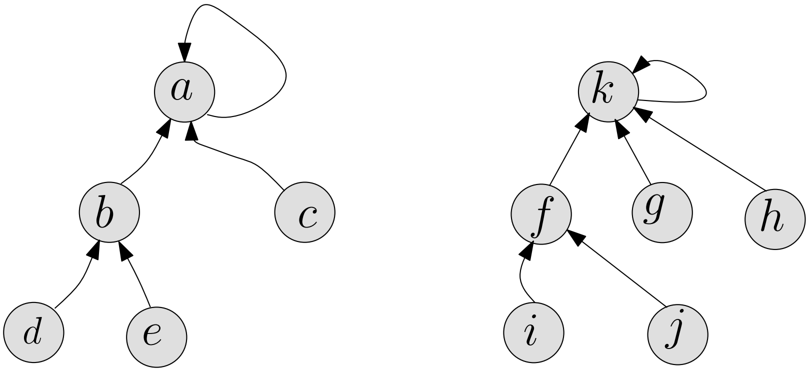
Amortized Analysis

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
   $=$ average time to perform an operation on data-structure.
5. Amortized time per operation $= \dfrac{\text{overall running time}}{\text{number of operations}}$.

15.2.3: The data-structure

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.

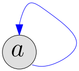
Reversed Trees

!esrever ni retteb si gnihtyreve esuaceB

1. Reversed Trees:
   1. Initially: Every element is its own node.
   2. Node v: $\overline{\mathbf{p}}(v)$ pointer to its parent.
   3. Set uniquely identified by root node/element.
2.
3.

Reversed Trees

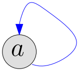
!esrever ni retteb si gnihtyreve esuaceB

1. Reversed Trees:
   1. Initially: Every element is its own node.
   2. Node v: $\overline{\mathbf{p}}(v)$ pointer to its parent.
   3. Set uniquely identified by root node/element.
2. **makeSet**: Create a singleton pointing to itself: 

Reversed Trees

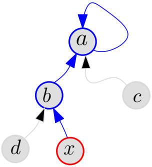
!esrever ni retteb si gnihtyreve esuaceB

1. Reversed Trees:
   1. Initially: Every element is its own node.
   2. Node v: $\overline{\mathbf{p}}(v)$ pointer to its parent.
   3. Set uniquely identified by root node/element.
2. **makeSet**: Create a singleton pointing to itself:



3. **find**(x):
   1. Start from node containing x, traverse up tree, till arriving to root.
   2. **find**(x):

      $x \rightarrow b \rightarrow a$
   3. a: returned as set.

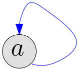


Reversed Trees

!esrever ni retteb si gnihtyreve esuaceB

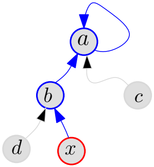
1. Reversed Trees:
   1. Initially: Every element is its own node.
   2. Node v: $\overline{\mathrm{p}}(v)$ pointer to its parent.
   3. Set uniquely identified by root node/element.

2. **makeSet**: Create a singleton pointing to itself:

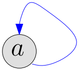


3. **find**(x):
   1. Start from node containing x, traverse up tree, till arriving to root.
   2. **find**(x):

      $x \rightarrow b \rightarrow a$
   3. a: returned as set.

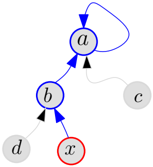


1. Reversed Trees:
   1. Initially: Every element is its own node.
   2. Node v: $\overline{\mathrm{p}}(v)$ pointer to its parent.
   3. Set uniquely identified by root node/element.
2. **makeSet**: Create a singleton pointing to itself:



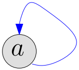
3. **find**(x):
   1. Start from node containing x, traverse up tree, till arriving to root.
   2. **find**(x):

      $x \rightarrow b \rightarrow a$
   3. a: returned as set.

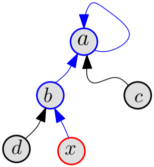


Reversed Trees

!esrever ni retteb si gnihtyreve esuaceB

1. Reversed Trees:
   1. Initially: Every element is its own node.
   2. Node v: $\overline{\mathrm{p}}(v)$ pointer to its parent.
   3. Set uniquely identified by root node/element.
2. **makeSet**: Create a singleton pointing to itself: 

3. **find**(x):
   1. Start from node containing x, traverse up tree, till arriving to root.
   2. **find**(x): $x \to b \to a$
   3. a: returned as set.

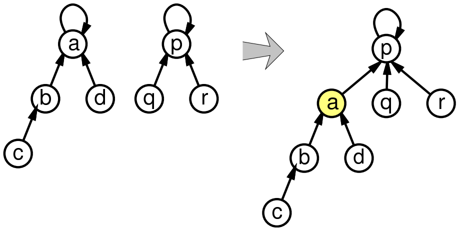


Union operation in reversed trees

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...

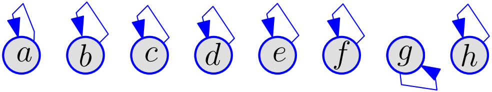
makeSet(x)
  $\overline{\mathrm{p}}(x) \leftarrow x$

find(x)
  if $x = \overline{\mathrm{p}}(x)$ **then**
    return x
  return **find**$(\overline{\mathrm{p}}(x))$

union(x, y)
  $A \leftarrow$ **find**(x)
  $B \leftarrow$ **find**(y)
  $\overline{\mathrm{p}}(B) \leftarrow A$

Example...

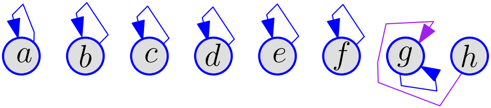
The long chain



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

Example...

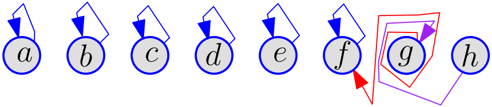
The long chain



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

Example...

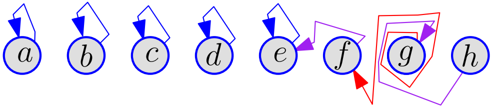
The long chain



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

Example...

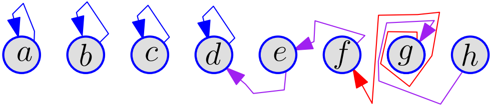
The long chain



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...

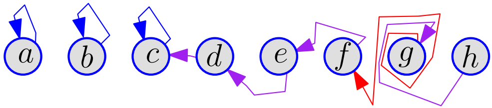
The long chain



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...

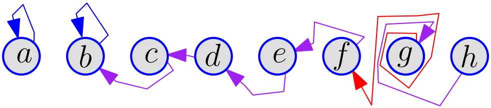
The long chain



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)

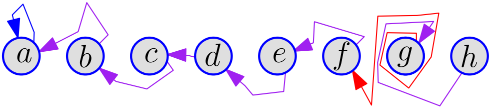
union(d, e)

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!

1. **find** might require $\Omega(n)$ time.
2. **Q**: How improve performance?
3. Two "hacks":
   (i) **Union by rank**:
   Maintain in root of tree , a bound on its depth (**rank**).
   Rule: Hang the smaller tree on the larger tree in **union**.
   (ii) **Path compression**:
   During find, make all pointers on path point to root.

Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.
2. **Q**: How improve performance?
3. Two "hacks":
   (i) **Union by rank**:
   Maintain in root of tree , a bound on its depth (**rank**).
   Rule: Hang the smaller tree on the larger tree in **union**.
   (ii) **Path compression**:
   During find, make all pointers on path point to root.

Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.
2. **Q**: How improve performance?
3. Two "hacks":
   (i) **Union by rank**:
   Maintain in root of tree , a bound on its depth (**rank**).
   Rule: Hang the smaller tree on the larger tree in **union**.
   (ii) **Path compression**:
   During find, make all pointers on path point to root.

Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.
2. **Q**: How improve performance?
3. Two "hacks":

   (i) **Union by rank**:
   Maintain in root of tree , a bound on its depth (**rank**).
   Rule: Hang the smaller tree on the larger tree in **union**.

   (ii) **Path compression**:
   During find, make all pointers on path point to root.

Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.
2. **Q**: How improve performance?
3. Two "hacks":

   (i) **Union by rank**:
   Maintain in root of tree , a bound on its depth (**rank**).
   Rule: Hang the smaller tree on the larger tree in **union**.

   (ii) **Path compression**:
   During find, make all pointers on path point to root.

Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.
2. **Q**: How improve performance?
3. Two "hacks":
   (i) **Union by rank**:
   Maintain in root of tree , a bound on its depth (**rank**).
   Rule: Hang the smaller tree on the larger tree in **union**.
   (ii) **Path compression**:
   During find, make all pointers on path point to root.

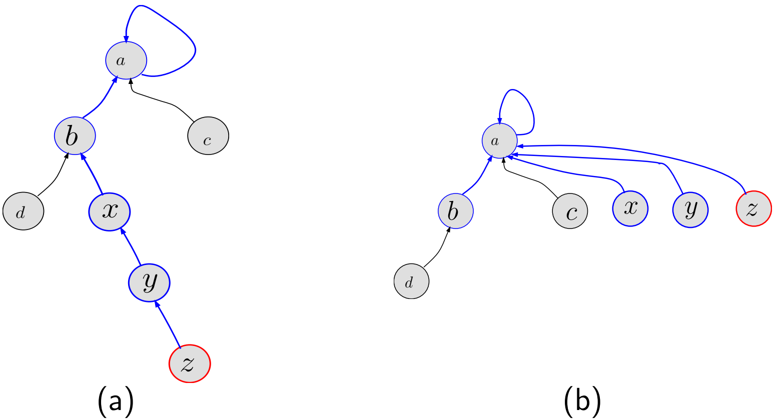
Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.
2. **Q**: How improve performance?
3. Two "hacks":

   (i) **Union by rank**:
   Maintain in root of tree , a bound on its depth (**rank**).
   Rule: Hang the smaller tree on the larger tree in **union**.

   (ii) **Path compression**:
   During find, make all pointers on path point to 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
```

15.3: Analyzing the Union-Find Data-Structure

Definition

Definition

v: Node **UnionFind** data-structure $\mathcal{D}$
v is **leader** $\iff$ v root of a (reversed) tree in $\mathcal{D}$.

"When you're not leader, you're little people."

"You know the score pal. If you're not cop, you're little people." - Blade Runner (movie).

Definition

Definition

v: Node **UnionFind** data-structure $\mathcal{D}$
v is **leader** $\iff$ v root of a (reversed) tree in $\mathcal{D}$.

"When you're not leader, you're little people."
"You know the score pal. If you're not cop, you're little people." - Blade Runner (movie).

Lemma

Lemma

Once node v stop being a leader, can never become leader again.

Proof.

1. x stopped being leader because **union** operation hanged x on y.
2. From this point on...
3. x might change only its parent pointer (**find**).
4. x parent pointer will never become equal to x again.
5. x never a leader again.

□

Lemma

Lemma

Once node v stop being a leader, can never become leader again.

Proof.

1. x stopped being leader because **union** operation hanged x on y.
2. From this point on...
3. x might change only its parent pointer (**find**).
4. x parent pointer will never become equal to x again.
5. x never a leader again.

□

Lemma

Lemma

Once node v stop being a leader, can never become leader again.

Proof.

1. x stopped being leader because **union** operation hanged x on y.
2. From this point on...
3. x might change only its parent pointer (**find**).
4. x parent pointer will never become equal to x again.
5. x never a leader again.

Lemma

Lemma

Once node v stop being a leader, can never become leader again.

Proof.

1. x stopped being leader because **union** operation hanged x on y.
2. From this point on...
3. x might change only its parent pointer (**find**).
4. x parent pointer will never become equal to x again.
5. x never a leader again.

□

Lemma

Lemma

Once node v stop being a leader, can never become leader again.

Proof.

1. x stopped being leader because **union** operation hanged x on y.
2. From this point on...
3. x might change only its parent pointer (**find**).
4. x parent pointer will never become equal to x again.
5. x never a leader again.

Lemma

Lemma

Once node v stop being a leader, can never become leader again.

Proof.

1. x stopped being leader because **union** operation hanged x on y.
2. From this point on...
3. x might change only its parent pointer (**find**).
4. x parent pointer will never become equal to x again.
5. x never a leader again.

□

Another Lemma

Lemma

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

Proof.

1. rank of element changes only by **union** operation.
2. **union** operation changes rank only for...
   the "new" leader of the new set.
3. if an element is no longer a leader, than its rank is fixed.

□

Another Lemma

Lemma

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

Proof.

1. rank of element changes only by **union** operation.
2. **union** operation changes rank only for...
   the "new" leader of the new set.
3. if an element is no longer a leader, than its rank is fixed.

□

Another Lemma

Lemma

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

Proof.

1. rank of element changes only by **union** operation.
2. **union** operation changes rank only for...
   the "new" leader of the new set.
3. if an element is no longer a leader, than its rank is fixed.

□

Another Lemma

Lemma

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

Proof.

1. rank of element changes only by **union** operation.
2. **union** operation changes rank only for...
   the "new" leader of the new set.
3. if an element is no longer a leader, than its rank is fixed.

□

Another Lemma

Lemma

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

Proof.

1. rank of element changes only by **union** operation.
2. **union** operation changes rank only for...
   the "new" leader of the new set.
3. 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...
...along a path from node to root of the tree.

Proof...

1. Claim: $\forall u \rightarrow v$ 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: $\forall u \rightarrow v$ 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 π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathrm{rank}(v) >$ 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 $\mathrm{root}(A)$ 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 π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathrm{rank}(v) >$ 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: $\mathrm{rank}(u) < \mathrm{rank}(v)$.
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 π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathrm{rank}(v) >$ 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: $\mathrm{rank}(u) < \mathrm{rank}(v)$.
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 π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathrm{rank}(v) >$ 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: $\forall u \rightarrow v$ 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 π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathrm{rank}(v) >$ 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: $\forall u \rightarrow v$ 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 π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathrm{rank}(v) >$ 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: $\forall u \rightarrow v$ 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 π, then all the nodes of π are made to point to the last node v of π.
   By induction, $\mathrm{rank}(v) >$ 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 node gets rank $k \implies$ *at least* $\geq 2^k$ *elements in its subtree.*

Proof.

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. node u gets rank k only if the merged two roots u, v has rank $k - 1$.
4. By induction, u and v have $\geq 2^{k-1}$ nodes before merge.
5. merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.

□

Trees grow exponentially in size with rank

Lemma

When node gets rank $k \implies$ at least $\geq 2^k$ elements in its subtree.

Proof.

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. node u gets rank k only if the merged two roots u, v has rank $k - 1$.
4. By induction, u and v have $\geq 2^{k-1}$ nodes before merge.
5. merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.

□

Trees grow exponentially in size with rank

Lemma

When node gets rank $k \implies$ at least $\geq 2^k$ elements in its subtree.

Proof.

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. node u gets rank k only if the merged two roots u, v has rank $k - 1$.
4. By induction, u and v have $\geq 2^{k-1}$ nodes before merge.
5. merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.

□

Trees grow exponentially in size with rank

Lemma

When node gets rank $k \implies$ *at least* $\geq 2^k$ *elements in its subtree.*

Proof.

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. node u gets rank k only if the merged two roots u, v has rank $k - 1$.
4. By induction, u and v have $\geq 2^{k-1}$ nodes before merge.
5. merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.

□

Trees grow exponentially in size with rank

Lemma

When node gets rank $k \implies$ at least $\geq 2^k$ elements in its subtree.

Proof.

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. node u gets rank k only if the merged two roots u, v has rank $k - 1$.
4. By induction, u and v have $\geq 2^{k-1}$ nodes before merge.
5. merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.

□

Trees grow exponentially in size with rank

Lemma

When node gets rank $k \implies$ *at least* $\geq 2^k$ *elements in its subtree.*

Proof.

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. node u gets rank k only if the merged two roots u, v has rank $k - 1$.
4. By induction, u and v have $\geq 2^{k-1}$ nodes before merge.
5. 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.

1. By induction. For $k = 0$ it is obvious.
2. when v become of rank k. Charge to roots merged: u and v.
3. Before union: u and v of rank $k - 1$
4. After merge: $\mathrm{rank}(v) = k$ and $\mathrm{rank}(u) = k - 1$.
5. u no longer leader. Its rank is now fixed.
6. u, v leave rank $k - 1 \implies v$ enters rank k.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created. $\implies$ # nodes rank k: $\leq (n/2^{k-1})/2 = n/2^k$.

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.

1. By induction. For $k = 0$ it is obvious.
2. when v become of rank k. Charge to roots merged: u and v.
3. Before union: u and v of rank $k - 1$
4. After merge: $\mathrm{rank}(v) = k$ and $\mathrm{rank}(u) = k - 1$.
5. u no longer leader. Its rank is now fixed.
6. u, v leave rank $k - 1 \implies v$ enters rank k.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created. $\implies$ # nodes rank k: $\leq (n/2^{k-1})/2 = n/2^k$.

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.

1. By induction. For $k = 0$ it is obvious.
2. when v become of rank k. Charge to roots merged: u and v.
3. Before union: u and v of rank $k - 1$
4. After merge: $\mathrm{rank}(v) = k$ and $\mathrm{rank}(u) = k - 1$.
5. u no longer leader. Its rank is now fixed.
6. u, v leave rank $k - 1$ $\implies$ v enters rank k.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created. $\implies$ # nodes rank k: $\leq (n/2^{k-1})/2 = n/2^k$.

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.

1. By induction. For $k = 0$ it is obvious.
2. when v become of rank k. Charge to roots merged: u and v.
3. Before union: u and v of rank $k - 1$
4. After merge: $\mathrm{rank}(v) = k$ and $\mathrm{rank}(u) = k - 1$.
5. u no longer leader. Its rank is now fixed.
6. u, v leave rank $k - 1 \implies v$ enters rank k.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created. $\implies \#$ nodes rank k: $\leq (n/2^{k-1})/2 = n/2^k$.

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.

1. By induction. For $k = 0$ it is obvious.
2. when v become of rank k. Charge to roots merged: u and v.
3. Before union: u and v of rank $k - 1$
4. After merge: $\mathbf{rank}(v) = k$ and $\mathbf{rank}(u) = k - 1$.
5. u no longer leader. Its rank is now fixed.
6. u, v leave rank $k - 1$ $\implies$ v enters rank k.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created.
   $\implies$ $\#$ nodes rank k: $\leq (n/2^{k-1})/2 = n/2^k$.

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.

1. By induction. For $k = 0$ it is obvious.
2. when v become of rank k. Charge to roots merged: u and v.
3. Before union: u and v of rank $k - 1$
4. After merge: $\mathbf{rank}(v) = k$ and $\mathbf{rank}(u) = k - 1$.
5. u no longer leader. Its rank is now fixed.
6. u, v leave rank $k - 1 \implies v$ enters rank k.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created. $\implies \#$ nodes rank k: $\leq (n/2^{k-1})/2 = n/2^k$.

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.

1. By induction. For $k = 0$ it is obvious.
2. when v become of rank k. Charge to roots merged: u and v.
3. Before union: u and v of rank $k - 1$
4. After merge: $\mathrm{rank}(v) = k$ and $\mathrm{rank}(u) = k - 1$.
5. u no longer leader. Its rank is now fixed.
6. u, v leave rank $k - 1 \implies v$ enters rank k.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created. $\implies$ # nodes rank k: $\leq \left(n/2^{k-1}\right)/2 = n/2^k$.

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.

1. rank of leader v of reversed tree T, bounds depth of T.
2. By previous lemma: $\max$ rank $\leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform **find** bounded by depth of tree.

□

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.

1. rank of leader v of reversed tree T, bounds depth of T.
2. By previous lemma: $\max$ rank $\leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform **find** bounded by depth of tree.

□

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.

1. rank of leader v of reversed tree T, bounds depth of T.
2. By previous lemma: $\max$ rank $\leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform **find** bounded by depth of tree.

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.

1. rank of leader v of reversed tree T, bounds depth of T.
2. By previous lemma: $\max$ rank $\leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform **find** bounded by depth of tree.

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.

1. rank of leader v of reversed tree T, bounds depth of T.
2. By previous lemma: $\max$ rank $\leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform **find** bounded by depth of tree.

□

$\log^*$ in detail

1. $\log^*(n)$: number of times to take $\lg$ of number to get number smaller than two.
2. $\log^* 2 = 1$
3. $\log^* 2^2 = 2$.
4. $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.
5. $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
6. $\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5.$
7. $\log^*$ is a monotone increasing function.
8. $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
   For practical purposes, $\log^*$ returns value ≤ 5.

$\log^*$ in detail

1. $\log^*(n)$: number of times to take $\lg$ of number to get number smaller than two.
2. $\log^* 2 = 1$
3. $\log^* 2^2 = 2$.
4. $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.
5. $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
6. $\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5$.
7. $\log^*$ is a monotone increasing function.
8. $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
   For practical purposes, $\log^*$ returns value ≤ 5.

$\log^*$ in detail

1. $\log^*(n)$: number of times to take $\lg$ of number to get number smaller than two.
2. $\log^* 2 = 1$
3. $\log^* 2^2 = 2$.
4. $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.
5. $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
6. $\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5$.
7. $\log^*$ is a monotone increasing function.
8. $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
   For practical purposes, $\log^*$ returns value ≤ 5.

$\log^*$ in detail

1. $\log^*(n)$: number of times to take $\lg$ of number to get number smaller than two.
2. $\log^* 2 = 1$
3. $\log^* 2^2 = 2$.
4. $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.
5. $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
6. $\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5$.
7. $\log^*$ is a monotone increasing function.
8. $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
   For practical purposes, $\log^*$ returns value ≤ 5.

$\log^*$ in detail

1. $\log^*(n)$: number of times to take $\lg$ of number to get number smaller than two.
2. $\log^* 2 = 1$
3. $\log^* 2^2 = 2$.
4. $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.
5. $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
6. $\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5.$
7. $\log^*$ is a monotone increasing function.
8. $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
   For practical purposes, $\log^*$ returns value ≤ 5.

$\log^*$ in detail

1. $\log^*(n)$: number of times to take $\lg$ of number to get number smaller than two.
2. $\log^* 2 = 1$
3. $\log^* 2^2 = 2$.
4. $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.
5. $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
6. $\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5$.
7. $\log^*$ is a monotone increasing function.
8. $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
   For practical purposes, $\log^*$ returns value ≤ 5.

$\log^*$ in detail

1. $\log^*(n)$: number of times to take $\lg$ of number to get number smaller than two.
2. $\log^* 2 = 1$
3. $\log^* 2^2 = 2$.
4. $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.
5. $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
6. $\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5.$
7. $\log^*$ is a monotone increasing function.
8. $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
   For practical purposes, $\log^*$ returns value ≤ 5.

$\log^*$ in detail

1. $\log^*(n)$: number of times to take $\lg$ of number to get number smaller than two.
2. $\log^* 2 = 1$
3. $\log^* 2^2 = 2$.
4. $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.
5. $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
6. $\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5.$
7. $\log^*$ is a monotone increasing function.
8. $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
   For practical purposes, $\log^*$ returns value ≤ 5.

Can do much better!

Theorem

For a sequence of m operations over n elements, the overall running time of the **UnionFind** *data-structure is $O((n + m) \log^* n)$.*

1. Intuitively: **UnionFind** 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

For a sequence of m operations over n elements, the overall running time of the **UnionFind** *data-structure is $O((n + m) \log^* n)$.*

1. Intuitively: **UnionFind** 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

For a sequence of m operations over n elements, the overall running time of the **UnionFind** *data-structure is $O((n+m)\log^* n)$.*

1. Intuitively: **UnionFind** 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

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

$\mathrm{Tower}(i)$: a tower of $2^{2^{2^{\cdot^{\cdot^{2}}}}}$ 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

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

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

The tower function...

Definition

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

$\mathrm{Tower}(i)$: a tower of $2^{2^{2^{\cdot^{\cdot^{2}}}}}$ 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

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

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

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

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

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

The tower function...

Definition

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

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

Definition

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

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

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

The tower function...

Definition

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

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

Definition

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

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

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

The tower function...

Definition

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

$\mathrm{Tower}(i)$: a tower of $2^{2^{2^{\cdot^{\cdot^{2}}}}}$ 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

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

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

The tower function...

Definition

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

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

Definition

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

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

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

The tower function...

Definition

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

$\mathrm{Tower}(i)$: a tower of $2^{2^{2^{\cdot^{\cdot^{2}}}}}$ 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

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

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

The tower function...

Definition

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

$\mathrm{Tower}(i)$: a tower of $2^{2^{2^{\cdot^{\cdot^{2}}}}}$ 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

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

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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{\mathrm{p}}(x_1)$, $x_3 = \overline{\mathrm{p}}(x_2)$, ..., $x_i = \overline{\mathrm{p}}(x_{i-1})$, $\ldots, x_m = \overline{\mathrm{p}}(x_{m-1})$ = 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 $\mathrm{rank}(x) \in \mathrm{Block}(i)$.

5. Looking for ways to pay for the **find** operation.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{p}(x_1)$, $x_3 = \overline{p}(x_2)$, ..., $x_i = \overline{p}(x_{i-1})$, $\ldots, x_m = \overline{p}(x_{m-1})$ = 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.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{p}(x_1)$, $x_3 = \overline{p}(x_2)$, ..., $x_i = \overline{p}(x_{i-1})$, $\ldots, x_m = \overline{p}(x_{m-1})$ = 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.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{p}(x_1)$, $x_3 = \overline{p}(x_2)$, $\ldots$, $x_i = \overline{p}(x_{i-1})$, $\ldots, x_m = \overline{p}(x_{m-1})$ = 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.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{p}(x_1)$, $x_3 = \overline{p}(x_2)$, ..., $x_i = \overline{p}(x_{i-1})$, $\ldots, x_m = \overline{p}(x_{m-1})$ = 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.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{\mathrm{p}}(x_1)$, $x_3 = \overline{\mathrm{p}}(x_2)$, ..., $x_i = \overline{\mathrm{p}}(x_{i-1})$, $\ldots, x_m = \overline{\mathrm{p}}(x_{m-1})$ = 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.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{\mathrm{p}}(x_1)$, $x_3 = \overline{\mathrm{p}}(x_2)$, $\ldots$, $x_i = \overline{\mathrm{p}}(x_{i-1})$, $\ldots, x_m = \overline{\mathrm{p}}(x_{m-1})$ = 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 $\mathrm{rank}(x) \in \mathrm{Block}(i)$.

5. Looking for ways to pay for the **find** operation.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{\mathrm{p}}(x_1)$, $x_3 = \overline{\mathrm{p}}(x_2)$, $\ldots$, $x_i = \overline{\mathrm{p}}(x_{i-1})$, $\ldots, x_m = \overline{\mathrm{p}}(x_{m-1})$ = 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.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{p}(x_1)$, $x_3 = \overline{p}(x_2)$, $\ldots$, $x_i = \overline{p}(x_{i-1})$, $\ldots, x_m = \overline{p}(x_{m-1})$ = 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.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{\mathrm{p}}(x_1)$, $x_3 = \overline{\mathrm{p}}(x_2)$, $\ldots$, $x_i = \overline{\mathrm{p}}(x_{i-1})$, $\ldots, x_m = \overline{\mathrm{p}}(x_{m-1}) =$ 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.
6. Since 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. ...start from x and we visit the sequence: $x_1 = x$, $x_2 = \overline{\mathrm{p}}(x_1)$, $x_3 = \overline{\mathrm{p}}(x_2)$, ..., $x_i = \overline{\mathrm{p}}(x_{i-1})$, $\ldots, x_m = \overline{\mathrm{p}}(x_{m-1})$ = 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.
6. Since 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, say 2).
3. **find** (x): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
6. node x: $\nu = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
7. $\mathrm{rank}(x) \in \mathrm{Block}\Big(\mathrm{index_B}(x)\Big)$.
8. $\mathrm{index_B}(x)$: **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, say 2).
3. **find** (x): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
6. node x: $\nu = \mathrm{index}_{\mathrm{B}}(x)$ index block containing $\mathrm{rank}(x)$.
7. $\mathrm{rank}(x) \in \mathrm{Block}\Big(\mathrm{index}_{\mathrm{B}}(x)\Big)$.
8. $\mathrm{index}_{\mathrm{B}}(x)$: **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, say 2).
3. **find** (x): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
6. node x: $\nu = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
7. $\mathrm{rank}(x) \in \mathrm{Block}\Big(\mathrm{index_B}(x)\Big)$.
8. $\mathrm{index_B}(x)$: **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, say 2).
3. **find** (x): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
6. node x: $\nu = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
7. $\mathrm{rank}(x) \in \mathrm{Block}\Big(\mathrm{index_B}(x)\Big)$.
8. $\mathrm{index_B}(x)$: **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, say 2).
3. **find** (x): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
6. node x: $\nu = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
7. $\mathrm{rank}(x) \in \mathrm{Block}\Big(\mathrm{index_B}(x)\Big)$.
8. $\mathrm{index_B}(x)$: **block of x**

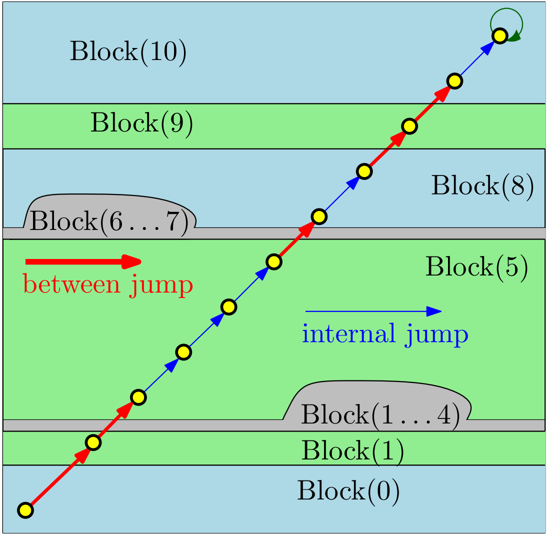
Blocks and jumping pointers

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, say 2).
3. **find** (x): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
6. node x: $\nu = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
7. $\mathrm{rank}(x) \in \mathrm{Block}\Big(\mathrm{index_B}(x)\Big)$.
8. $\mathrm{index_B}(x)$: **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, say 2).
3. **find** (x): π path used.
4. partition π into each by rank.
5. Price of **find** length π.
6. node x: $\nu = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
7. $\mathrm{rank}(x) \in \mathrm{Block}\Big(\mathrm{index_B}(x)\Big)$.
8. $\mathrm{index_B}(x)$: **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**.

1. If for $x \in \pi$, the node $\overline{\mathrm{p}}(x)$ is in a different block than x, then $x \to \overline{\mathrm{p}}(x)$ is a **jump between blocks**.
2. 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.

1. $\pi = x_1, \ldots, x_m$: path followed by **find**.
2. $x_i = \overline{\mathrm{p}}(x_{i-})$, for all i.
3. $0 \leq \mathrm{index_B}(x_1) \leq \mathrm{index_B}(x_2) \leq \ldots \leq \mathrm{index_B}(x_m)$.
4. $\mathrm{index_B}(x_m) = O(\log^* n)$.
5. Number of elements in π such that $\mathrm{index_B}(x) \neq \mathrm{index_B}(\overline{\mathrm{p}}(x))$...
6. ... at most $O(\log^* n)$.

□

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.

1. $\pi = x_1, \ldots, x_m$: path followed by **find**.
2. $x_i = \overline{\mathrm{p}}(x_{i-})$, for all i.
3. $0 \leq \mathrm{index_B}(x_1) \leq \mathrm{index_B}(x_2) \leq \ldots \leq \mathrm{index_B}(x_m)$.
4. $\mathrm{index_B}(x_m) = O(\log^* n)$.
5. Number of elements in π such that $\mathrm{index_B}(x) \neq \mathrm{index_B}(\overline{\mathrm{p}}(x))$...
6. ... at most $O(\log^* n)$.

□

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.

1. $\pi = x_1, \ldots, x_m$: path followed by **find**.
2. $x_i = \overline{\mathrm{p}}(x_{i-})$, for all i.
3. $0 \leq \mathrm{index_B}(x_1) \leq \mathrm{index_B}(x_2) \leq \ldots \leq \mathrm{index_B}(x_m)$.
4. $\mathrm{index_B}(x_m) = O(\log^* n)$.
5. Number of elements in π such that $\mathrm{index_B}(x) \neq \mathrm{index_B}(\overline{\mathrm{p}}(x))$...
6. ... at most $O(\log^* n)$.

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.

1. $\pi = x_1, \ldots, x_m$: path followed by **find**.
2. $x_i = \overline{\mathrm{p}}(x_{i-})$, for all i.
3. $0 \leq \mathrm{index_B}(x_1) \leq \mathrm{index_B}(x_2) \leq \ldots \leq \mathrm{index_B}(x_m)$.
4. $\mathrm{index_B}(x_m) = O(\log^* n)$.
5. Number of elements in π such that $\mathrm{index_B}(x) \neq \mathrm{index_B}(\overline{\mathrm{p}}(x))$...
6. ... at most $O(\log^* n)$.

□

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.

1. $\pi = x_1, \ldots, x_m$: path followed by **find**.
2. $x_i = \overline{\mathrm{p}}(x_{i-})$, for all i.
3. $0 \leq \mathrm{index_B}(x_1) \leq \mathrm{index_B}(x_2) \leq \ldots \leq \mathrm{index_B}(x_m)$.
4. $\mathrm{index_B}(x_m) = O(\log^* n)$.
5. Number of elements in π such that $\mathrm{index_B}(x) \neq \mathrm{index_B}(\overline{\mathrm{p}}(x))$...
6. ... at most $O(\log^* n)$.

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.

1. $\pi = x_1, \ldots, x_m$: path followed by **find**.
2. $x_i = \overline{\mathrm{p}}(x_{i-})$, for all i.
3. $0 \leq \mathrm{index_B}(x_1) \leq \mathrm{index_B}(x_2) \leq \ldots \leq \mathrm{index_B}(x_m)$.
4. $\mathrm{index_B}(x_m) = O(\log^* n)$.
5. Number of elements in π such that $\mathrm{index_B}(x) \neq \mathrm{index_B}(\overline{\mathrm{p}}(x))$...
6. ... at most $O(\log^* n)$.

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.

1. $\pi = x_1, \ldots, x_m$: path followed by **find**.
2. $x_i = \overline{\mathrm{p}}(x_{i-})$, for all i.
3. $0 \leq \mathrm{index_B}(x_1) \leq \mathrm{index_B}(x_2) \leq \ldots \leq \mathrm{index_B}(x_m)$.
4. $\mathrm{index_B}(x_m) = O(\log^* n)$.
5. Number of elements in π such that $\mathrm{index_B}(x) \neq \mathrm{index_B}(\overline{\mathrm{p}}(x))$...
6. ... at most $O(\log^* n)$.

□

Benefits of an internal jump

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

Benefits of an internal jump

1. x and $\overline{\mathrm{p}}(x)$ are in same block.
2. $\mathrm{index_B}(x) = \mathrm{index_B}(\overline{\mathrm{p}}(x))$.
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 x jumped forward...
8. ...and new parent has higher rank.
9. 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. $r_{\mathrm{bef}} = \mathbf{rank}(\overline{\mathrm{p}}(x))$ 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 x jumped forward...
8. ...and new parent has higher rank.
9. Charge internal block jumps to this "progress".

Changing parents...

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.,* $\mathrm{index_B}(x) = i$*) before* $\overline{\mathrm{p}}(x)$ *is no longer in the* i*th block. That is* $\mathrm{index_B}(\overline{\mathrm{p}}(x)) > i$*.*

Proof.

1. parent of x incr rank every-time internal jump goes through x.
2. At most $|\mathrm{Block}(i)|$ different values in the ith block.
3. $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1, \mathrm{Tower}(i)]$
4. Claim follows, as: $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$.

□

Changing parents...

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.

1. parent of x incr rank every-time internal jump goes through x.
2. At most $|\mathrm{Block}(i)|$ different values in the ith block.
3. $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1, \mathrm{Tower}(i)]$
4. Claim follows, as: $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$.

□

Changing parents...

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.

1. parent of x incr rank every-time internal jump goes through x.
2. At most $|\mathrm{Block}(i)|$ different values in the ith block.
3. $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1, \mathrm{Tower}(i)]$
4. Claim follows, as: $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$.

□

Changing parents...

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.

1. parent of x incr rank every-time internal jump goes through x.
2. At most $|\mathrm{Block}(i)|$ different values in the ith block.
3. $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1, \mathrm{Tower}(i)]$
4. Claim follows, as: $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$.

□

Changing parents...

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.,* $\mathrm{index}_B(x) = i$*) before* $\overline{\mathrm{p}}(x)$ *is no longer in the* i*th block. That is* $\mathrm{index}_B(\overline{\mathrm{p}}(x)) > i$.

Proof.

1. parent of x incr rank every-time internal jump goes through x.
2. At most $|\mathrm{Block}(i)|$ different values in the ith block.
3. $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1, \mathrm{Tower}(i)]$
4. Claim follows, as: $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$.

□

Few elements are in the bigger blocks

Lemma

At most $n/\mathrm{Tower}(i)$ nodes are assigned ranks in the ith block throughout the algorithm execution.

Proof.

By lemma, the number of elements with rank in the ith block

□

Few elements are in the bigger blocks

Lemma

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

Proof.

By lemma, the number of elements with rank in the ith block

$$\leq \sum_{k \in \mathrm{Block}(i)} \frac{n}{2^k}$$

□

Few elements are in the bigger blocks

Lemma

At most $n/\mathrm{Tower}(i)$ nodes are assigned ranks in the ith block throughout the algorithm execution.

Proof.

By lemma, the number of elements with rank in the ith block

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

□

Few elements are in the bigger blocks

Lemma

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

Proof.

By lemma, the number of elements with rank in the ith block

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

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

□

Few elements are in the bigger blocks

Lemma

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

Proof.

By lemma, the number of elements with rank in the ith block

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

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

□

Few elements are in the bigger blocks

Lemma

At most $n/\mathrm{Tower}(i)$ *nodes are assigned ranks in the* i*th 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 $O(n)$

Lemma

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

Proof.

1. x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
2. ... after that all jumps through x are between blocks, by lemma...
3. $\leq n/\mathrm{Tower}(i)$ elements assigned ranks in the ith block, throughout algorithm execution.
4. 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 $O(n)$

Lemma

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

Proof.

1. x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
2. ... after that all jumps through x are between blocks, by lemma...
3. $\leq n/\mathrm{Tower}(i)$ elements assigned ranks in the ith block, throughout algorithm execution.
4. 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 $O(n)$

Lemma

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

Proof.

1. x in ith block, have at most $|\text{Block}(i)|$ internal jumps...
2. ... after that all jumps through x are between blocks, by lemma...
3. $\leq n/\text{Tower}(i)$ elements assigned ranks in the ith block, throughout algorithm execution.
4. total number of internal jumps is $|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n.$

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.

1. x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
2. ... after that all jumps through x are between blocks, by lemma...
3. $\leq n/\mathrm{Tower}(i)$ elements assigned ranks in the ith block, throughout algorithm execution.
4. 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.

1. x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
2. ... after that all jumps through x are between blocks, by lemma...
3. $\leq n/\mathrm{Tower}(i)$ elements assigned ranks in the ith block, throughout algorithm execution.
4. 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 $O(n)$

Lemma

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

Proof.

1. x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
2. ... after that all jumps through x are between blocks, by lemma...
3. $\leq n/\mathrm{Tower}(i)$ elements assigned ranks in the ith block, throughout algorithm execution.
4. 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.

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. There are at most $O(n \log^* n)$ internal jumps.

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.

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. There are at most $O(n \log^* n)$ internal jumps.

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.

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. There are at most $O(n \log^* n)$ internal jumps.

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.

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. There are at most $O(n \log^* n)$ internal jumps.

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.

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. 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* $O((n+m)\log^* n)$.

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$

$f_2(x) = f_1(f_2(x-1)) = 2x$

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$

$f_3(x) = f_2(f_3(x-1)) = 2^x$

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$

$f_4(x) = f_3(f_4(x-1)) = \mathrm{Tower} x$

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$
$f_4(x) = \mathrm{Tower}(x)$	

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$
$f_4(x) = \mathrm{Tower}(x)$	$g_4(x) = \log^* x$

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$
$f_4(x) = \mathrm{Tower}(x)$	$g_4(x) = \log^* x$
$f_5(x) = \ldots$	

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$
$f_4(x) = \mathrm{Tower}(x)$	$g_4(x) = \log^* x$
$f_5(x) = \ldots$	

$f_i(x) = f_{i-1}^{(x)}(1)$

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$
$f_4(x) = \mathrm{Tower}(x)$	$g_4(x) = \log^* x$
$f_5(x) = \ldots$	

$f_i(x) = f_{i-1}^{(x)}(1)$
$g_i(x)$ = # of times one has to apply $g_{i-1}(\cdot)$ to x before we get number smaller than 2.

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$
$f_4(x) = \text{Tower}(x)$	$g_4(x) = \log^* x$
$f_5(x) = \ldots$	

$f_i(x) = f_{i-1}^{(x)}(1)$
$g_i(x) = \#$ of times one has to apply $g_{i-1}(\cdot)$ to x before we get number smaller than 2.
$A(n) = f_n(n)$: **Ackerman function**.

More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$
$f_4(x) = \text{Tower}(x)$	$g_4(x) = \log^* x$
$f_5(x) = \ldots$	

$f_i(x) = f_{i-1}^{(x)}(1)$
$g_i(x)$ = # of times one has to apply $g_{i-1}(\cdot)$ to x before we get number smaller than 2.
$A(n) = f_n(n)$: **Ackerman function.**
Inverse Ackerman function:
$\alpha(n) = A^{-1}(n) =$ min i s.t. $g_i(n) \leq i$.

Union-Find: Tarjan result

Theorem (Tarjan [1975])

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

(The above is not quite correct, but close enough.)

Notes

Notes

Notes

Notes

R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.*, 22:215–225, 1975.