CS 573: Algorithms, Fall 2014

Union-Find

Lecture 21 November 6, 2014

Part I

Union Find

- Maintain a collection of sets.
- \bigcirc find(x) returns the set that contains x
- 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.

- Maintain a collection of sets.
- 2 makeSet(x) creates a set that contains the single element x.
- \bigcirc find(x) returns the set that contains x.
- **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.

- Maintain a collection of sets.
- \bigcirc find(x) returns the set that contains x
- 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.

- Maintain a collection of sets.
- **Implies** \mathbf{o} find(x) returns the set that contains x.
- ullet 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.

- Maintain a collection of sets.
- **o** find(x) returns the set that contains x.
- lacktriangledown union (A,B) returns set = union of A and B. That is $A\cup B$.

 \dots merges the two sets A and B and return the merged set.

- Maintain a collection of sets.
- ullet makeSet(x) creates a set that contains the single element x.
- **o** find(x) returns the set that contains x.
- **1** 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.

- Use data-structure as a black-box inside algorithm.... Union-Find in Kruskal algorithm for computing MST.
- 2 Bounded worst case time per operation.
- Care: overall running time spend in data-structure.
- amortized running-time of operation
 average time to perform an operation on data-structure

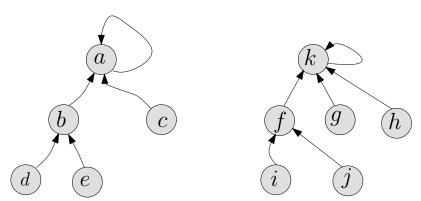
- Use data-structure as a black-box inside algorithm.... Union-Find in Kruskal algorithm for computing MST.
- Bounded worst case time per operation.
- Care: overall running time spend in data-structure.
- amortized running-time of operation
 average time to perform an operation on data-structure

- Use data-structure as a black-box inside algorithm.... Union-Find in Kruskal algorithm for computing MST.
- Bounded worst case time per operation.
- Care: overall running time spend in data-structure.
- amortized running-time of operation
 average time to perform an operation on data-structure

- Use data-structure as a black-box inside algorithm.Union-Find in Kruskal algorithm for computing MST.
- 2 Bounded worst case time per operation.
- Care: overall running time spend in data-structure.
- amortized running-time of operation
 average time to perform an operation on data-structure.

- Use data-structure as a black-box inside algorithm.
 Union-Find in Kruskal algorithm for computing MST.
- Bounded worst case time per operation.
- Care: overall running time spend in data-structure.
- amortized running-time of operation
 average time to perform an operation on data-structure.

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.

!esrever ni retteb si gnihtyreve esuaceB

- Reversed Trees:
 - 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.

!esrever ni retteb si gnihtyreve esuaceB

- Reversed Trees:
 - 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.



@ makeSet: Create a singleton pointing to itself:

!esrever ni retteb si gnihtyreve esuaceB

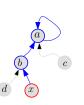
- Reversed Trees:
 - 1 Initially: Every element is its own node.
 - 2 Node $v: \overline{\mathbf{p}}(v)$ pointer to its parent.
 - 3 Set uniquely identified by root node/element.



- makeSet: Create a singleton pointing to itself:
- \odot find(x):
 - Start from node containing x, traverse up tree, till arriving to root.





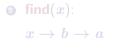


!esrever ni retteb si gnihtyreve esuaceB

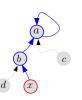
- Reversed Trees:
 - 1 Initially: Every element is its own node.
 - **2** Node $v: \overline{\mathbf{p}}(v)$ pointer to its parent.
 - Set uniquely identified by root node/element.



- \odot find(x):
 - Start from node containing x, traverse up tree, till arriving to root.

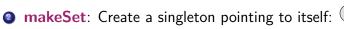






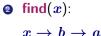
!esrever ni retteb si gnihtyreve esuaceB

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





- \odot find(x):
 - Start from node containing x, traverse up tree, till arriving to root.

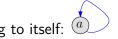




a: returned as set

!esrever ni retteb si gnihtyreve esuaceB

- 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.



- makeSet: Create a singleton pointing to itself:
- - Start from node containing x, traverse up tree, till arriving to root.



 \odot find(x):

$$x \rightarrow b \rightarrow a$$

a: returned as set.

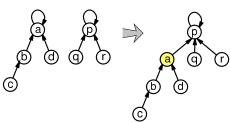
6

Union operation in reversed trees

Just hang them on each other.

union(a, p): Merge two sets.

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



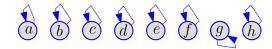
Pseudo-code of naive version...

$$\frac{\mathsf{makeSet}(\mathsf{x})}{\overline{\mathsf{p}}(x) \leftarrow x}$$

$$\begin{array}{c} \operatorname{find}(\mathbf{x}) \\ \text{if } x = \overline{\mathbf{p}}(x) \text{ then} \\ \text{return } x \\ \\ \text{return} \\ \operatorname{find}(\overline{\mathbf{p}}(x)) \end{array}$$

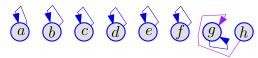
```
egin{aligned} \mathsf{union}(\ x,\ y\ ) \ A &\leftarrow \mathsf{find}(x) \ B &\leftarrow \mathsf{find}(y) \ \overline{\mathrm{p}}(B) &\leftarrow A \end{aligned}
```

The long chain



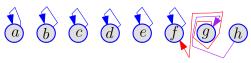
After: makeSet(a), makeSet(b), makeSet(c), makeSet(d), makeSet(e), makeSet(f), makeSet(g), makeSet(h)

The long chain

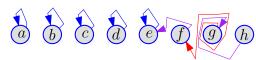


After: $\mathsf{makeSet}(a)$, $\mathsf{makeSet}(b)$, $\mathsf{makeSet}(c)$, $\mathsf{makeSet}(d)$, $\mathsf{makeSet}(e)$, $\mathsf{makeSet}(f)$, $\mathsf{makeSet}(g)$, $\mathsf{makeSet}(h)$ $\mathsf{union}(g,h)$

The long chain

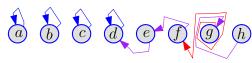


After: $\mathsf{makeSet}(a)$, $\mathsf{makeSet}(b)$, $\mathsf{makeSet}(c)$, $\mathsf{makeSet}(d)$, $\mathsf{makeSet}(e)$, $\mathsf{makeSet}(f)$, $\mathsf{makeSet}(g)$, $\mathsf{makeSet}(h)$ $\mathsf{union}(g,h)$ $\mathsf{union}(f,g)$

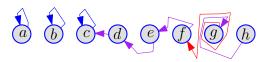


```
After: \mathsf{makeSet}(a), \mathsf{makeSet}(b), \mathsf{makeSet}(c), \mathsf{makeSet}(d), \mathsf{makeSet}(e), \mathsf{makeSet}(f), \mathsf{makeSet}(g), \mathsf{makeSet}(h) \mathsf{union}(g,h) \mathsf{union}(f,g) \mathsf{union}(e,f)
```

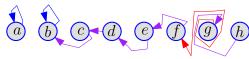
The long chain



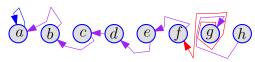
```
After: \mathsf{makeSet}(a), \mathsf{makeSet}(b), \mathsf{makeSet}(c), \mathsf{makeSet}(d), \mathsf{makeSet}(e), \mathsf{makeSet}(f), \mathsf{makeSet}(g), \mathsf{makeSet}(h) \mathsf{union}(g,h) \mathsf{union}(f,g) \mathsf{union}(e,f) \mathsf{union}(d,e)
```



```
After: \mathsf{makeSet}(a), \mathsf{makeSet}(b), \mathsf{makeSet}(c), \mathsf{makeSet}(d), \mathsf{makeSet}(e), \mathsf{makeSet}(f), \mathsf{makeSet}(g), \mathsf{makeSet}(h) \mathsf{union}(g,h) \mathsf{union}(f,g) \mathsf{union}(e,f) \mathsf{union}(d,e) \mathsf{union}(c,d)
```



```
After: \mathsf{makeSet}(a), \mathsf{makeSet}(b), \mathsf{makeSet}(c), \mathsf{makeSet}(d), \mathsf{makeSet}(e), \mathsf{makeSet}(f), \mathsf{makeSet}(g), \mathsf{makeSet}(h) \mathsf{union}(g,h) \mathsf{union}(f,g) \mathsf{union}(e,f) \mathsf{union}(d,e) \mathsf{union}(c,d) \mathsf{union}(c,d) \mathsf{union}(b,c)
```



```
After: \mathsf{makeSet}(a), \mathsf{makeSet}(b), \mathsf{makeSet}(c), \mathsf{makeSet}(d), \mathsf{makeSet}(e), \mathsf{makeSet}(f), \mathsf{makeSet}(g), \mathsf{makeSet}(h) \mathsf{union}(g,h) \mathsf{union}(f,g) \mathsf{union}(e,f) \mathsf{union}(d,e) \mathsf{union}(c,d) \mathsf{union}(c,d) \mathsf{union}(b,c) \mathsf{union}(b,c) \mathsf{union}(a,b)
```

- find might require $\Omega(n)$ time.
- Q: How improve performance?
- 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.

- **1 find** might require $\Omega(n)$ time.
- Q: How improve performance?
- 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.

- **1 find** might require $\Omega(n)$ time.
- Q: How improve performance?
- 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.

- **1 find** might require $\Omega(n)$ time.
- Q: How improve performance?
- 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.

- **1 find** might require $\Omega(n)$ time.
- Q: How improve performance?
- 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

- **1 find** might require $\Omega(n)$ time.
- Q: How improve performance?
- 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:

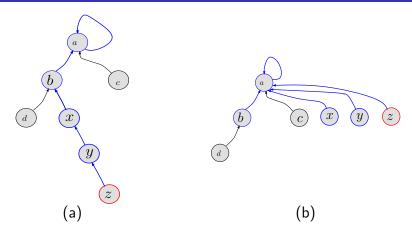
During find, make all pointers on path point to root.

Find is slow, hack it!

- **1 find** might require $\Omega(n)$ time.
- Q: How improve performance?
- 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.

Sariel (UIUC) CS573 Fall 2014 10 / 38

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...

```
\begin{array}{c} \operatorname{find}(\mathsf{x}) \\ \text{if } x \neq \overline{\mathrm{p}}(x) \text{ then} \\ \overline{\mathrm{p}}(x) \leftarrow \operatorname{find}(\overline{\mathrm{p}}(x)) \\ \text{return } \overline{\mathrm{p}}(x) \end{array}
```

```
union(x, y)
   A \leftarrow \mathsf{find}(x)
   B \leftarrow \mathsf{find}(y)
   if rank(A) > rank(B) then
      \overline{\mathbf{p}}(B) \leftarrow A
   else
      \overline{p}(A) \leftarrow B
      if rank(A) = rank(B) then
          rank(B) \leftarrow rank(B) + 1
```

Part II

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 a leader, you're little people."

Lemma

Once node $oldsymbol{v}$ stop being a leader, can never become leader again.

- lacktriangledown lacktriangledown lacktriangledown stopped being leader because **union** operation hanged $oldsymbol{x}$ on $oldsymbol{y}$.
- 2 From this point on...
- ullet x parent pointer will never become equal to x again.
- $oldsymbol{\circ}$ x never a leader again.



Lemma

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

- $oldsymbol{0}$ x stopped being leader because **union** operation hanged x on y.
- From this point on...
- @~x parent pointer will never become equal to x again.
- ullet x never a leader again.



Lemma

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

- $oldsymbol{0}$ x stopped being leader because union operation hanged x on y.
- 2 From this point on...
- @~x parent pointer will never become equal to x again.
- lacktriangledown x never a leader again.



Lemma

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

- $oldsymbol{0}$ x stopped being leader because **union** operation hanged x on y.
- From this point on...
- @~x parent pointer will never become equal to x again.
- ullet x never a leader again.



Lemma

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

- $oldsymbol{0}$ x stopped being leader because union operation hanged x on y.
- From this point on...
- $oldsymbol{\circ}$ $oldsymbol{x}$ might change only its parent pointer (find).
- $oldsymbol{0}$ $oldsymbol{x}$ parent pointer will never become equal to $oldsymbol{x}$ again.
- lacksquare x never a leader again.



emma

_emma

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

- x stopped being leader because union operation hanged x on y.
- From this point on...
- *x* might change only its parent pointer (find).
- \boldsymbol{x} parent pointer will never become equal to \boldsymbol{x} again.
- x never a leader again.



Lemma

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

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

Lemma

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

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

Lemma

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

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

Lemma

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

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



Lemma

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

Proof.

- rank of element changes only by union operation.
- union operation changes rank only for... the "new" leader of the new set.
- o 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.

- Claim: $\forall u \to v \text{ in DS: } \operatorname{rank}(u) < \operatorname{rank}(v)$.
- Proof by induction. Base: all singletons. Holds.
- \odot Assume claim holds at time t, before an operation.
- If operation is union(A, B), and assume that we hanged root(A) on root(B).
 Must be that rank(root(B)) is now larger than rank(root(A)) (verify!).
 Claim true after operation!
- 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, 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.

- Claim: $\forall u \to v \text{ in DS: } \operatorname{rank}(u) < \operatorname{rank}(v)$.
- Proof by induction. Base: all singletons. Holds.
- \odot Assume claim holds at time t, before an operation.
- If operation is union(A, B), and assume that we hanged root(A) on root(B).
 Must be that rank(root(B)) is now larger than rank(root(A)) (verify!).
 Claim true after operation!
- 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, 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.

- ① Claim: $\forall u \to v \text{ in DS: } \operatorname{rank}(u) < \operatorname{rank}(v)$.
- Proof by induction. Base: all singletons. Holds.
- **3** Assume claim holds at time t, before an operation.
- If operation is union(A, B), and assume that we hanged root(A) on root(B).
 Must be that rank(root(B)) is now larger than rank(root(A)) (verify!).
 Claim true after operation!
- 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, 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.

- lacksquare Claim: $\forall u o v$ in DS: $\mathrm{rank}(u) < \mathrm{rank}(v)$.
- Proof by induction. Base: all singletons. Holds.
- **3** Assume claim holds at time t, before an operation.
- ① If operation is union(A, B), and assume that we hanged root(A) on root(B).
 - Must be that rank(root(B)) is now larger than rank(root(A)) (verify!).
- 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, 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.

- ① Claim: $\forall u \to v \text{ in DS: } \operatorname{rank}(u) < \operatorname{rank}(v).$
- Proof by induction. Base: all singletons. Holds.
- **3** Assume claim holds at time t, before an operation.
- If operation is union(A, B), and assume that we hanged root(A) on root(B).
 Must be that rank(root(B)) is now larger than rank(root(A)) (verify!).
 - Claim true after operation!
- 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, 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.

- ① Claim: $\forall u \to v \text{ in DS: } \operatorname{rank}(u) < \operatorname{rank}(v)$.
- Proof by induction. Base: all singletons. Holds.
- \odot Assume claim holds at time t, before an operation.
- If operation is union(A, B), and assume that we hanged root(A) on root(B).
 Must be that rank(root(B)) is now larger than rank(root(A)) (verify!).
 - Claim true after operation!
- 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, 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

- ① Claim: $\forall u \to v \text{ in DS: } \operatorname{rank}(u) < \operatorname{rank}(v)$.
- Proof by induction. Base: all singletons. Holds.
- **3** Assume claim holds at time t, before an operation.
- If operation is union(A, B), and assume that we hanged root(A) on root(B).
 Must be that rank(root(B)) is now larger than rank(root(A)) (verify!).
 - Claim true after operation!
- **1** 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, $\operatorname{rank}(v) > \operatorname{rank}$ of all other nodes of π . All the nodes that get compressed, the rank of their new parent is larger than their own rank.

- Claim: $\forall u \to v \text{ in DS: } \operatorname{rank}(u) < \operatorname{rank}(v).$
- Proof by induction. Base: all singletons. Holds.
- **3** Assume claim holds at time t, before an operation.
- If operation is union(A, B), and assume that we hanged root(A) on root(B).
 - Must be that rank(root(B)) is now larger than rank(root(A)) (verify!). Claim true after operation!
- **1** 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, $\operatorname{rank}(v) > \operatorname{rank}$ of all other nodes of π .

All the nodes that get compressed, the rank of their new parent is larger than their own rank.

- Claim: $\forall u \to v \text{ in DS: } \operatorname{rank}(u) < \operatorname{rank}(v).$
- Proof by induction. Base: all singletons. Holds.
- **3** Assume claim holds at time t, before an operation.
- If operation is union(A, B), and assume that we hanged root(A) on root(B).
 Must be that rank(root(B)) is now larger than rank(root(A)) (verify!).
 - Claim true after operation!
- **1** 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, $\operatorname{rank}(v) > \operatorname{rank}$ of all other nodes of π .

All the nodes that get compressed, the rank of their new parent, is larger than their own rank.

Lemma

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

Proof.

Fall 2014 19 / 38

Lemma

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

Proof.

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

Lemma

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

Proof.

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

Lemma

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

Proof.

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

Lemma

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

Proof.

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

Lemma

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

Proof.

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



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.
- lacksquare Before union: u and v of rank k-1
- After merge: rank(v) = k and rank(u) = k 1.
- u no longer leader. Its rank is now fixed.
- ullet u,v leave rank $k-1\implies v$ enters rank k
- ② By induction: at most $n/2^{k-1}$ nodes of rank k-1 created. \implies # nodes rank k: $\le \left(n/2^{k-1}\right)/2 = n/2^k$.

Sariel (UIUC) CS573 20 Fall 2014 20 / 38

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
- After merge: rank(v) = k and rank(u) = k 1.
- u no longer leader. Its rank is now fixed.
- 0 u, v leave rank $k-1 \implies v$ enters rank k.
- ② By induction: at most $n/2^{k-1}$ nodes of rank k-1 created. \implies # nodes rank k: $\le \left(n/2^{k-1}\right)/2 = n/2^k$.

Sariel (UIUC) CS573 20 Fall 2014 20 / 38

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 $oldsymbol{v}$ become of rank $oldsymbol{k}$. Charge to roots merged: $oldsymbol{u}$ and $oldsymbol{v}$.
- **3** Before union: u and v of rank k-1
- ① After merge: rank(v) = k and rank(u) = k 1.
- u no longer leader. Its rank is now fixed.
- ullet u,v leave rank $k-1\implies v$ enters rank k
- \bigcirc By induction: at most $n/2^{k-1}$ nodes of rank k-1 created. \implies # nodes rank k: $<\left(n/2^{k-1}\right)/2=n/2^k$.

Sariel (UIUC) CS573 20 Fall 2014 20 / 38

Lemma

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

- **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
- After merge: rank(v) = k and rank(u) = k 1.
- ullet u no longer leader. Its rank is now fixed.
- ullet u,v leave rank $k-1\implies v$ enters rank k
- ② By induction: at most $n/2^{k-1}$ nodes of rank k-1 created. \implies # nodes rank k: $<\left(n/2^{k-1}\right)/2=n/2^k$.

Having higher rank is rare

Lemma

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

- **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
- After merge: rank(v) = k and rank(u) = k 1.
- ullet u,v leave rank $k-1\implies v$ enters rank k
- ② By induction: at most $n/2^{k-1}$ nodes of rank k-1 created. \implies # nodes rank k: $<\left(n/2^{k-1}\right)/2=n/2^k$.

Having higher rank is rare

Lemma

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

- **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
- After merge: rank(v) = k and rank(u) = k 1.
- ullet u no longer leader. Its rank is now fixed.
- 0 u, v leave rank $k-1 \implies v$ enters rank k.
- ② By induction: at most $n/2^{k-1}$ nodes of rank k-1 created. \implies # nodes rank k: $\le \left(n/2^{k-1}\right)/2 = n/2^k$.

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.

- **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
- After merge: rank(v) = k and rank(u) = k 1.
- 0 u, v leave rank $k-1 \implies v$ enters rank k.
- lacktriangledown By induction: at most $n/2^{k-1}$ nodes of rank k-1 created. $\Longrightarrow \#$ nodes rank k: $\le \left(n/2^{k-1}\right)/2 = n/2^k$.

Lemma

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

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



Lemma

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

- lacksquare 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 $O(\log n)$.
- Time to perform find bounded by depth of tree.



Lemma

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

- lacksquare 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 $O(\log n)$.
- Time to perform find bounded by depth of tree.



Lemma

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

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



Lemma

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

- lacktriangle rank of leader $oldsymbol{v}$ of reversed tree $oldsymbol{T}$, bounds depth of $oldsymbol{T}$.
- By previous lemma: \max rank $\leq \lg n$.
- Depth of tree is $O(\log n)$.
- Time to perform find bounded by depth of tree.



- $\log^*(n)$: number of times to take \lg of number to get number smaller than two.
- $\log^* 2 = 1$
- $\log^* 2^2 = 2.$

- log* is a monotone increasing function.
- $eta=2^{2^{2^{2^2}}}=2^{65536}$: huge number For practical purposes, \log^* returns value ≤ 5

- $\log^*(n)$: number of times to take \lg of number to get number smaller than two.
- $\log^* 2 = 1$
- $\log^* 2^2 = 2.$

- log* is a monotone increasing function.
- $\beta=2^{2^{2^{2^2}}}=2^{65536}$: huge number For practical purposes, \log^* returns value ≤ 5

log[∗] in detail

- $\log^*(n)$: number of times to take \lg of number to get number smaller than two.
- $\log^* 2 = 1$
- $\log^* 2^2 = 2.$

- log* is a monotone increasing function.
- $eta=2^{2^{2^{2^2}}}=2^{65536}$: huge number For practical purposes, \log^* returns value ≤ 5

- $\log^*(n)$: number of times to take \lg of number to get number smaller than two.
- $\log^* 2 = 1$
- $\log^* 2^2 = 2.$

- O log* is a monotone increasing function.
- $eta=2^{2^{2^{2^2}}}=2^{65536}$: huge number For practical purposes, \log^* returns value ≤ 5

- $\log^*(n)$: number of times to take \lg of number to get number smaller than two.
- $\log^* 2 = 1$
- $\log^* 2^2 = 2.$

- O log* is a monotone increasing function.
- $\beta=2^{2^{2^{2^2}}}=2^{65536}$: huge number For practical purposes, \log^* returns value ≤ 5

- $\log^*(n)$: number of times to take \lg of number to get number smaller than two.
- $\log^* 2 = 1$
- $\log^* 2^2 = 2.$

- O log* is a monotone increasing function.
- $eta=2^{2^{2^{2^2}}}=2^{65536}$: huge number For practical purposes, \log^* returns value ≤ 5

- $\log^*(n)$: number of times to take \lg of number to get number smaller than two.
- $\log^* 2 = 1$
- $\log^* 2^2 = 2.$

- **◊** log* is a monotone increasing function.
- $eta=2^{2^{2^2}}=2^{65536}$: huge number For practical purposes, \log^* returns value ≤ 5

- $\log^*(n)$: number of times to take \lg of number to get number smaller than two.
- $\log^* 2 = 1$
- $\log^* 2^2 = 2.$

- **◊** log* is a monotone increasing function.
- $\beta=2^{2^{2^{2^2}}}=2^{65536}$: 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 $O((n+m)\log^* n)$.

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

- 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...

Definition

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

Tower(i): a tower of $2^{2^{2^{-1}}}$ of height i. Observe that $\log^*(\operatorname{Tower}(i)) = i$.

Definition

For
$$i \geq 0$$
, let $\operatorname{Block}(i) = [\operatorname{Tower}(i-1) + 1, \operatorname{Tower}(i)]$; that is $\operatorname{Block}(i) = [z, 2^{z-1}]$ for $z = \operatorname{Tower}(i-1) + 1$. Also $\operatorname{Block}(0) = [0, 1]$. As such, $\operatorname{Block}(0) = [0, 1]$, $\operatorname{Block}(1) = [2, 2]$, $\operatorname{Block}(2) = [3, 4]$, $\operatorname{Block}(3) = [5, 16]$, $\operatorname{Block}(4) = [17, 65536]$, $\operatorname{Block}(5) = [65537, 2^{65536}]\dots$

Definition

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

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

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

For
$$i \geq 0$$
, let $\operatorname{Block}(i) = [\operatorname{Tower}(i-1) + 1, \operatorname{Tower}(i)];$ that i $\operatorname{Block}(i) = [z, 2^{z-1}]$ for $z = \operatorname{Tower}(i-1) + 1$. Also $\operatorname{Block}(0) = [0, 1]$. As such, $\operatorname{Block}(0) = [0, 1]$, $\operatorname{Block}(1) = [2, 2]$, $\operatorname{Block}(2) = [3, 4]$, $\operatorname{Block}(3) = [5, 16]$, $\operatorname{Block}(4) = [17, 65536]$, $\operatorname{Block}(5) = [65537, 2^{65536}]$...

Definition

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

Tower(i): a tower of $2^{2^{2^{i-1}}}$ of height i. Observe that $\log^*(\text{Tower}(i)) = i$.

For
$$i \geq 0$$
, let $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1,\mathrm{Tower}(i)];$ that if $\mathrm{Block}(i) = \begin{bmatrix} z,2^{z-1} \end{bmatrix}$ for $z = \mathrm{Tower}(i-1)+1.$ Also $\mathrm{Block}(0) = [0,1].$ As such, $\mathrm{Block}(0) = \begin{bmatrix} 0,1 \end{bmatrix}, \, \mathrm{Block}(1) = \begin{bmatrix} 2,2 \end{bmatrix}, \, \mathrm{Block}(2) = \begin{bmatrix} 3,4 \end{bmatrix},$ $\mathrm{Block}(3) = \begin{bmatrix} 5,16 \end{bmatrix}, \, \mathrm{Block}(4) = \begin{bmatrix} 17,65536 \end{bmatrix},$ $\mathrm{Block}(5) = \begin{bmatrix} 65537,2^{65536} \end{bmatrix}\dots$

Definition

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

Tower(i): a tower of $2^{2^{2^{-1/2}}}$ of height i.

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

For
$$i \geq 0$$
, let $\operatorname{Block}(i) = [\operatorname{Tower}(i-1)+1,\operatorname{Tower}(i)];$ that is $\operatorname{Block}(i) = \begin{bmatrix} z,2^{z-1} \end{bmatrix}$ for $z = \operatorname{Tower}(i-1)+1.$ Also $\operatorname{Block}(0) = [0,1].$ As such,

Block(0) =
$$\begin{bmatrix} 0, 1 \end{bmatrix}$$
, Block(1) = $\begin{bmatrix} 2, 2 \end{bmatrix}$, Block(2) = $\begin{bmatrix} 3, 4 \end{bmatrix}$, Block(3) = $\begin{bmatrix} 5, 16 \end{bmatrix}$, Block(4) = $\begin{bmatrix} 17, 65536 \end{bmatrix}$,

Definition

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

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

For
$$i \geq 0$$
, let $\operatorname{Block}(i) = [\operatorname{Tower}(i-1) + 1, \operatorname{Tower}(i)];$ that is $\operatorname{Block}(i) = [z, 2^{z-1}]$ for $z = \operatorname{Tower}(i-1) + 1$. Also $\operatorname{Block}(0) = [0, 1]$. As such, $\operatorname{Block}(0) = [0, 1]$, $\operatorname{Block}(1) = [2, 2]$, $\operatorname{Block}(2) = [3, 4]$, $\operatorname{Block}(3) = [5, 16]$, $\operatorname{Block}(4) = [17, 65536]$, $\operatorname{Block}(5) = [65537, 2^{65536}] \dots$

Definition

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

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

For
$$i \geq 0$$
, let $\operatorname{Block}(i) = [\operatorname{Tower}(i-1) + 1, \operatorname{Tower}(i)]$; that is $\operatorname{Block}(i) = [z, 2^{z-1}]$ for $z = \operatorname{Tower}(i-1) + 1$. Also $\operatorname{Block}(0) = [0, 1]$. As such, $\operatorname{Block}(0) = [0, 1]$, $\operatorname{Block}(1) = [2, 2]$, $\operatorname{Block}(2) = [3, 4]$, $\operatorname{Block}(3) = [5, 16]$, $\operatorname{Block}(4) = [17, 65536]$, $\operatorname{Block}(5) = [65537, 2^{65536}]$...

Definition

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

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

For
$$i \geq 0$$
, let $\operatorname{Block}(i) = [\operatorname{Tower}(i-1) + 1, \operatorname{Tower}(i)];$ that is $\operatorname{Block}(i) = \begin{bmatrix} z, 2^{z-1} \end{bmatrix}$ for $z = \operatorname{Tower}(i-1) + 1.$ Also $\operatorname{Block}(0) = [0, 1]$. As such, $\operatorname{Block}(0) = \begin{bmatrix} 0, 1 \end{bmatrix}$, $\operatorname{Block}(1) = \begin{bmatrix} 2, 2 \end{bmatrix}$, $\operatorname{Block}(2) = \begin{bmatrix} 3, 4 \end{bmatrix}$, $\operatorname{Block}(3) = \begin{bmatrix} 5, 16 \end{bmatrix}$, $\operatorname{Block}(4) = \begin{bmatrix} 17, 65536 \end{bmatrix}$, $\operatorname{Block}(5) = \begin{bmatrix} 65537, 2^{65536} \end{bmatrix}$...

Definition

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

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

For
$$i \geq 0$$
, let $\operatorname{Block}(i) = [\operatorname{Tower}(i-1) + 1, \operatorname{Tower}(i)];$ that is $\operatorname{Block}(i) = \begin{bmatrix} z, 2^{z-1} \end{bmatrix}$ for $z = \operatorname{Tower}(i-1) + 1.$ Also $\operatorname{Block}(0) = \begin{bmatrix} 0, 1 \end{bmatrix}$. As such, $\operatorname{Block}(0) = \begin{bmatrix} 0, 1 \end{bmatrix}$, $\operatorname{Block}(1) = \begin{bmatrix} 2, 2 \end{bmatrix}$, $\operatorname{Block}(2) = \begin{bmatrix} 3, 4 \end{bmatrix}$, $\operatorname{Block}(3) = \begin{bmatrix} 5, 16 \end{bmatrix}$, $\operatorname{Block}(4) = \begin{bmatrix} 17, 65536 \end{bmatrix}$, $\operatorname{Block}(5) = \begin{bmatrix} 65537, 2^{65536} \end{bmatrix}$...

Definition

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

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

For
$$i \geq 0$$
, let $\operatorname{Block}(i) = [\operatorname{Tower}(i-1) + 1, \operatorname{Tower}(i)]$; that is $\operatorname{Block}(i) = \begin{bmatrix} z, 2^{z-1} \end{bmatrix}$ for $z = \operatorname{Tower}(i-1) + 1$. Also $\operatorname{Block}(0) = [0, 1]$. As such, $\operatorname{Block}(0) = \begin{bmatrix} 0, 1 \end{bmatrix}$, $\operatorname{Block}(1) = \begin{bmatrix} 2, 2 \end{bmatrix}$, $\operatorname{Block}(2) = \begin{bmatrix} 3, 4 \end{bmatrix}$, $\operatorname{Block}(3) = \begin{bmatrix} 5, 16 \end{bmatrix}$, $\operatorname{Block}(4) = \begin{bmatrix} 17, 65536 \end{bmatrix}$, $\operatorname{Block}(5) = \begin{bmatrix} 65537, 2^{65536} \end{bmatrix}$...

- **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: $x_1=x$, $x_2=\overline{\mathrm{p}}(x_1)$, $x_3=\overline{\mathrm{p}}(x_2)$, ..., $x_i=\overline{\mathrm{p}}(x_{i-1})$, ..., $x_m=\overline{\mathrm{p}}(x_{m-1})=\mathrm{root}$ of tree.
- $\operatorname{rank}(x_1) < \operatorname{rank}(x_2) < \operatorname{rank}(x_3) < \ldots < \operatorname{rank}(x_m).$
- lacksquare RT of $\mathbf{find}(x)$ is O(m).

Definition

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

- 5 Looking for ways to pay for the find operation.
- Since other two operations take constant time...

- **1** RT of find(x) proportional to length of the path from x to the root of its tree.
- $oldsymbol{2}$...start from $oldsymbol{x}$ 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}), ..., x_m = \overline{p}(x_{m-1}) = \text{root of tree.}
```

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

Definition

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

- 5 Looking for ways to pay for the find operation.
- Since other two operations take constant time...

- **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: $x_1 = x$, $x_2 = \overline{x}(x_1)$, $x_3 = \overline{x}(x_2)$

$$x_1 = x$$
, $x_2 = \overline{p}(x_1)$, $x_3 = \overline{p}(x_2)$, ..., $x_i = \overline{p}(x_{i-1})$, ..., $x_m = \overline{p}(x_{m-1}) = \text{root of tree}$.

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

Definition

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

- Solution Looking for ways to pay for the find operation.
- Since other two operations take constant time...

- **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: $x_1=x,\ x_2=\overline{\mathrm{p}}(x_1),\ x_3=\overline{\mathrm{p}}(x_2),\ \ldots,\ x_i=\overline{\mathrm{p}}(x_{i-1}),\ \ldots,x_m=\overline{\mathrm{p}}(x_{m-1})=\mathrm{root}\ \mathrm{of}\ \mathrm{tree}.$
- $\operatorname{rank}(x_1) < \operatorname{rank}(x_2) < \operatorname{rank}(x_3) < \ldots < \operatorname{rank}(x_m).$
- \bigcirc RT of **find**(x) is O(m).

Definition

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

- Output
 Looking for ways to pay for the find operation.
- Since other two operations take constant time...

- **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: $x_1=x, \ x_2=\overline{\mathrm{p}}(x_1), \ x_3=\overline{\mathrm{p}}(x_2), \ \ldots, \ x_i=\overline{\mathrm{p}}(x_{i-1}), \ \ldots, \ x_m=\overline{\mathrm{p}}(x_{m-1})=\mathrm{root} \ \mathrm{of} \ \mathrm{tree}.$
- $\operatorname{rank}(x_1) < \operatorname{rank}(x_2) < \operatorname{rank}(x_3) < \ldots < \operatorname{rank}(x_m).$
- lacksquare RT of $\operatorname{find}(x)$ is O(m).

Definition

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

- Output
 Looking for ways to pay for the find operation.
- Since other two operations take constant time...

- **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: $x_1=x,\ x_2=\overline{\mathrm{p}}(x_1),\ x_3=\overline{\mathrm{p}}(x_2),\ \ldots,\ x_i=\overline{\mathrm{p}}(x_{i-1}),\ \ldots,x_m=\overline{\mathrm{p}}(x_{m-1})=\text{root of tree}.$
- $\operatorname{rank}(x_1) < \operatorname{rank}(x_2) < \operatorname{rank}(x_3) < \ldots < \operatorname{rank}(x_m).$
- 9 RT of $\mathbf{find}(x)$ is O(m).

Definition

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

- 5 Looking for ways to pay for the find operation.
- Since other two operations take constant time...

- **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: $x_1=x,\ x_2=\overline{\mathrm{p}}(x_1),\ x_3=\overline{\mathrm{p}}(x_2),\ \ldots,\ x_i=\overline{\mathrm{p}}(x_{i-1}),\ \ldots,x_m=\overline{\mathrm{p}}(x_{m-1})=\text{root of tree}.$
- \bigcirc RT of **find**(x) is O(m).

Definition

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

- Output
 Looking for ways to pay for the find operation.
- Since other two operations take constant time...

- **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: $x_1=x,\ x_2=\overline{\mathrm{p}}(x_1),\ x_3=\overline{\mathrm{p}}(x_2),\ \ldots,\ x_i=\overline{\mathrm{p}}(x_{i-1}),\ \ldots,x_m=\overline{\mathrm{p}}(x_{m-1})=\text{root of tree}.$
- lacktriangledown RT of $\mathbf{find}(x)$ is O(m).

Definition

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

- Solution Looking for ways to pay for the find operation.
- Since other two operations take constant time...

Running time of find...

- **9** 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: $x_1=x,\ x_2=\overline{\mathrm{p}}(x_1),\ x_3=\overline{\mathrm{p}}(x_2),\ \ldots,\ x_i=\overline{\mathrm{p}}(x_{i-1}),\ \ldots,x_m=\overline{\mathrm{p}}(x_{m-1})=\text{root of tree}.$
- lacktriangledown RT of $\mathbf{find}(x)$ is O(m).

Definition

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

- Solution Looking for ways to pay for the find operation.
- Since other two operations take constant time...

Sariel (UIUC) CS573 25 Fall 2014 25 / 38

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: $x_1=x,\ x_2=\overline{\mathrm{p}}(x_1),\ x_3=\overline{\mathrm{p}}(x_2),\ \ldots,\ x_i=\overline{\mathrm{p}}(x_{i-1}),\ \ldots,x_m=\overline{\mathrm{p}}(x_{m-1})=\text{root of tree}.$
- lacktriangledown RT of $\mathbf{find}(x)$ is O(m).

Definition

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

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

Sariel (UIUC) CS573 25 Fall 2014 25 / 38

Running time of find...

- **9** 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: $x_1=x,\ x_2=\overline{\mathrm{p}}(x_1),\ x_3=\overline{\mathrm{p}}(x_2),\ \ldots,\ x_i=\overline{\mathrm{p}}(x_{i-1}),\ \ldots,x_m=\overline{\mathrm{p}}(x_{m-1})=\text{root of tree}.$
- lacktriangledown RT of $\mathbf{find}(x)$ is O(m).

Definition

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

- Looking for ways to pay for the find operation.
- Since other two operations take constant time...

Sariel (UIUC) CS573 25 Fall 2014 25 / 38

- **1** maximum rank of node v is $O(\log n)$.
- $ext{@} \# ext{ of blocks is } O(\log^* n), ext{ as } O(\log n) \in \operatorname{Block}(c \log^* n), \ (c : \operatorname{constant}, \operatorname{say } 2).$
- \odot find (x): π path used.
- $ext{ } ext{ } ext$
- **6** Price of **find** length π .
- lacktriangledown node x: $u = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
- \bullet index_B(x): **block of** x

- **1** maximum rank of node v is $O(\log n)$.
- $ext{@} ext{$\#$ of blocks is } O(\log^* n), ext{ as } O(\log n) \in \operatorname{Block}(c \log^* n), \ (c : \operatorname{constant}, \operatorname{say } 2).$
- 3 find (x): π path used.
- $ext{ } ext{ } ext$
- **o** Price of **find** length π .
- lack 0 node x: $u = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
- index_B(x): **block of** x

- maximum rank of node v is $O(\log n)$.
- $ext{@} ext{\# of blocks is } O(\log^* n), ext{ as } O(\log n) \in \operatorname{Block}(c \log^* n), \ (c : \operatorname{constant}, \operatorname{say } 2).$
- **3** find (x): π path used.
- $ext{ } ext{ } ext$
- **6** Price of **find** length π .
- lack 0 node x: $u = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
- rank $(x) \in \operatorname{Block}(\operatorname{index}_{\operatorname{B}}(x)).$
- \bullet index_B(x): **block of** x

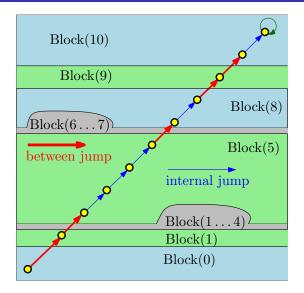
- maximum rank of node v is $O(\log n)$.
- $ext{@} ext{\# of blocks is } O(\log^* n), ext{ as } O(\log n) \in \operatorname{Block}(c \log^* n), \ (c : \operatorname{constant}, \operatorname{say } 2).$
- **3** find (x): π path used.
- lacktriangle partition π into each by rank.
- **6** Price of **find** length π
- lack 0 node x: $u = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
- \bullet index_B(x): **block of** x

- **1** maximum rank of node v is $O(\log n)$.
- $ext{@} ext{\# of blocks is } O(\log^* n), ext{ as } O(\log n) \in \operatorname{Block}(c \log^* n), \ (c : \operatorname{constant}, \operatorname{say } 2).$
- **3** find (x): π path used.
- lacktriangle partition π into each by rank.
- **5** Price of **find** length π .
- node x: $\nu = \mathrm{index}_{\mathrm{B}}(x)$ index block containing $\mathrm{rank}(x)$.
- rank $(x) \in \operatorname{Block} (\operatorname{index}_{\operatorname{B}}(x)).$
- index_B(x): **block of** x

- **1** maximum rank of node v is $O(\log n)$.
- $ext{@} ext{\# of blocks is } O(\log^* n)$, as $O(\log n) \in \operatorname{Block}(c \log^* n)$, $(c: \operatorname{constant}, \operatorname{say } 2)$.
- **3** find (x): π path used.
- lacktriangle partition π into each by rank.
- **5** Price of **find** length π .
- node x: $\nu = \mathrm{index}_{\mathrm{B}}(x)$ index block containing $\mathrm{rank}(x)$.
- \mathbf{O} rank $(x) \in \operatorname{Block}(\operatorname{index}_{\mathbf{B}}(x)).$
- index_B(x): **block of** x

- **1** maximum rank of node v is $O(\log n)$.
- $ext{@} ext{\# of blocks is } O(\log^* n)$, as $O(\log n) \in \operatorname{Block}(c \log^* n)$, $(c: \operatorname{constant}, \operatorname{say } 2)$.
- **3** find (x): π path used.
- lacktriangle partition π into each by rank.
- **5** Price of **find** length π .
- **o** node x: $\nu = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
- $ightharpoonup \operatorname{rank}(x) \in \operatorname{Block}(\operatorname{index}_{\mathbf{B}}(x)).$
- \bullet index_B(x): **block of** x

The path of find operation, and its pointers



- During a find operation...
- **3** Ranks of the nodes visited in π monotone increasing.
- Once leave block ith, never go back
- ullet charge visit to nodes in π next to element in a different block...
- \bullet to total number of blocks $\leq O(\log^* n)$.

- During a find operation...
- **3** Ranks of the nodes visited in π monotone increasing.
- Once leave block ith, never go back!
- \odot charge visit to nodes in π next to element in a different block...
- \bullet to total number of blocks $\leq O(\log^* n)$.

- During a find operation...
- **3** Ranks of the nodes visited in π monotone increasing.
- Once leave block ith, never go back!
- lacktriangle charge visit to nodes in π next to element in a different block...
- 0 to total number of blocks $\leq O(\log^* n)$.

- During a find operation...
- lacksquare Ranks of the nodes visited in π monotone increasing.
- Once leave block ith, never go back!
- lacktriangle charge visit to nodes in π next to element in a different block...
- \bullet to total number of blocks $\leq O(\log^* n)$.

Jumping pointers

Definition

 π : path traversed by **find**.

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

Lemma

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

Proof.

- \bullet $\pi = x_1, \ldots, x_m$: path followed by **find**.

Lemma

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

Proof.

- \bullet $\pi = x_1, \ldots, x_m$: path followed by find.

Lemma

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

Proof.

- \bullet $\pi = x_1, \ldots, x_m$: path followed by find.
- $\mathbf{2} \ x_i = \overline{\mathbf{p}}(x_{i-}), \text{ for all } i.$

Lemma

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

Proof.

- \bullet $\pi = x_1, \ldots, x_m$: path followed by find.
- $\mathbf{2} \ x_i = \overline{\mathbf{p}}(x_{i-}), \text{ for all } i.$
- $0 \le \operatorname{index}_{\mathrm{B}}(x_1) \le \operatorname{index}_{\mathrm{B}}(x_2) \le \ldots \le \operatorname{index}_{\mathrm{B}}(x_m).$

Lemma

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

Proof.

- \bullet $\pi = x_1, \ldots, x_m$: path followed by find.
- $\mathbf{2} \ x_i = \overline{\mathbf{p}}(x_{i-}), \text{ for all } i.$
- $0 < \operatorname{index}_{B}(x_{1}) < \operatorname{index}_{B}(x_{2}) < \ldots < \operatorname{index}_{B}(x_{m}).$

Lemma

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

Proof.

- \bullet $\pi = x_1, \ldots, x_m$: path followed by find.
- $\mathbf{2} \ x_i = \overline{\mathbf{p}}(x_{i-})$, for all i.
- $0 \le \mathrm{index}_{\mathrm{B}}(x_1) \le \mathrm{index}_{\mathrm{B}}(x_2) \le \ldots \le \mathrm{index}_{\mathrm{B}}(x_m).$
- Number of elements in π such that $index_B(x) \neq index_B(\overline{p}(x))...$
- 0 ... at most $O(\log^* n)$.

Lemma

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

Proof.

- \bullet $\pi = x_1, \ldots, x_m$: path followed by find.
- $\mathbf{2} \ x_i = \overline{\mathbf{p}}(x_{i-})$, for all i.
- $0 \le \operatorname{index}_{B}(x_1) \le \operatorname{index}_{B}(x_2) \le \ldots \le \operatorname{index}_{B}(x_m).$
- Number of elements in π such that $index_B(x) \neq index_B(\overline{p}(x))...$
- \bullet ... at most $O(\log^* n)$.

- $lackbox{1}{x}$ and $\overline{\mathbf{p}}(x)$ are in same block.
- \odot find passes through x.
- $\mathbf{0} \ \ r_{
 m aft} = {
 m rank}(\overline{
 m p}(x)) \ {
 m after} \ {
 m find} \ {
 m operation}.$
- ullet By path compression: $r_{
 m aft} > r_{
 m bef}$.
- \bigcirc \Longrightarrow parent pointer x jumped forward...
- ...and new parent has higher rank.
- Oharge internal block jumps to this "progress".

- **1** x and $\overline{p}(x)$ are in same block.
- **3** find passes through x.
- $\mathbf{0} \ \ r_{
 m aft} = {
 m rank}(\overline{
 m p}(x)) \ {
 m after} \ {
 m find} \ {
 m operation}.$
- ullet By path compression: $r_{
 m aft} > r_{
 m bef}$.
- \bigcirc \Longrightarrow parent pointer x jumped forward...
- ...and new parent has higher rank.
- Oharge internal block jumps to this "progress".

- **1** x and $\overline{p}(x)$ are in same block.
- **3** find passes through x.
- $lack r_{
 m bef} = {
 m rank}(\overline{
 m p}(x))$ before find operation.
- $\mathbf{0} \ \ r_{
 m aft} = {
 m rank}(\overline{
 m p}(x)) \ {
 m after} \ {
 m find} \ {
 m operation}.$
- **6** By path compression: $r_{
 m aft} > r_{
 m bef}$.
- \bigcirc \Longrightarrow parent pointer x jumped forward...
- ...and new parent has higher rank.
- Oharge internal block jumps to this "progress".

- $oldsymbol{0}$ $oldsymbol{x}$ and $\overline{\mathbf{p}}(oldsymbol{x})$ are in same block.
- $index_B(x) = index_B(\overline{p}(x)).$
- \odot find passes through x.
- $lack r_{
 m bef} = {
 m rank}(\overline{
 m p}(x))$ before find operation.
- $oldsymbol{0} r_{
 m aft} = {
 m rank}(\overline{
 m p}(x))$ after find operation.
- 6 By path compression: $r_{
 m aft} > r_{
 m bef}$.
- \bigcirc \Longrightarrow parent pointer x jumped forward...
- ...and new parent has higher rank.
- Oharge internal block jumps to this "progress".

- **1** x and $\overline{p}(x)$ are in same block.
- $index_B(x) = index_B(\overline{p}(x)).$
- **3** find passes through x.
- $lack r_{
 m bef} = {
 m rank}(\overline{
 m p}(x))$ before find operation.
- $oldsymbol{0} r_{
 m aft} = {
 m rank}(\overline{
 m p}(x))$ after find operation.
- **1** By path compression: $r_{
 m aft} > r_{
 m bef}$.
- \bigcirc \Longrightarrow parent pointer x jumped forward...
- ...and new parent has higher rank.
- Oharge internal block jumps to this "progress".

- $oldsymbol{0}$ $oldsymbol{x}$ and $\overline{\mathbf{p}}(oldsymbol{x})$ are in same block.
- $index_B(x) = index_B(\overline{p}(x)).$
- **3** find passes through x.
- $lack r_{
 m bef} = {
 m rank}(\overline{
 m p}(x))$ before find operation.
- $oldsymbol{0} r_{
 m aft} = {
 m rank}(\overline{
 m p}(x))$ after find operation.
- **1** By path compression: $r_{
 m aft} > r_{
 m bef}$.
- \bigcirc \Longrightarrow parent pointer x jumped forward...
- ...and new parent has higher rank.
- Oharge internal block jumps to this "progress".

- **1** x and $\overline{p}(x)$ are in same block.
- $index_B(x) = index_B(\overline{p}(x)).$
- **3** find passes through x.
- $lacktriangledown r_{
 m bef} = {
 m rank}(\overline{
 m p}(x))$ before find operation.
- $oldsymbol{0} \ r_{
 m aft} = {
 m rank}(\overline{
 m p}(x))$ after find operation.
- **1** By path compression: $r_{
 m aft} > r_{
 m bef}$.
- \bigcirc \Longrightarrow parent pointer x jumped forward...
- ...and new parent has higher rank.
- Oharge internal block jumps to this "progress".

- **1** x and $\overline{p}(x)$ are in same block.
- **3** find passes through x.
- $oldsymbol{0} \ r_{
 m aft} = {
 m rank}(\overline{
 m p}(x))$ after find operation.
- **1** By path compression: $r_{
 m aft} > r_{
 m bef}$.
- \bigcirc \Longrightarrow parent pointer x jumped forward...
- 3 ...and new parent has higher rank.
- Oharge internal block jumps to this "progress".

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

Lemma

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

Proof.

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

Proof.

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

Sariel (UIUC) CS573 32 Fall 2014 32 / 38

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

Lemma

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

Proof.

- **1** parent of x incr rank every-time internal jump goes through x.
- \bigcirc At most |Block(i)| different values in the *i*th block.

Fall 2014 32 / 38

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

Proof.

- lacktriangle parent of $m{x}$ incr rank every-time internal jump goes through $m{x}$.
- ② At most |Block(i)| different values in the *i*th block.
- ① Claim follows, as: $|Block(i)| \leq Tower(i)$.

Sariel (UIUC) CS573 32 Fall 2014 32 / 38

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

Proof.

- lacktriangle parent of x incr rank every-time internal jump goes through x.
- ② At most |Block(i)| different values in the *i*th block.
- Claim follows, as: $|Block(i)| \leq Tower(i)$.



Sariel (UIUC) CS573 32 Fall 2014 32 / 38

Lemma

At most n/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

Sariel (UIUC) CS573 Fall 2014 33 / 38

Lemma

At most n/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

$$\leq \sum_{k \in \operatorname{Block}(i)} rac{n}{2^k}$$



Lemma

At most n/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

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



Sariel (UIUC) CS573 33 Fall 2014 33 / 38

Lemma

At most n/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

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

Sariel (UIUC) CS573 33 Fall 2014 33 / 38

_emma

At most n/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

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

Sariel (UIUC) CS573 Fall 2014 33 / 38

Lemma

At most n/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

$$egin{aligned} & \leq \sum \limits_{k \in \operatorname{Block}(i)} rac{n}{2^k} = \sum \limits_{k = \operatorname{Tower}(i-1)+1}^{\operatorname{Tower}(i)} rac{n}{2^k} \ & = n \cdot \sum \limits_{k = \operatorname{Tower}(i-1)+1}^{\operatorname{Tower}(i)} rac{1}{2^k} \leq rac{n}{2^{\operatorname{Tower}(i-1)}} = rac{n}{\operatorname{Tower}(i)}. \end{aligned}$$

Sariel (UIUC) CS573 33 Fall 2014 33 / 38

Lemma

At most n/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

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

Sariel (UIUC) CS573 33 Fall 2014 33 / 38

Lemma

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

Proof.

- $@ \dots$ all jumps through x are between blocks, by lemma...
- ullet total number of internal jumps is $|\mathrm{Block}(i)| \cdot rac{n}{\mathrm{Tower}(i)} \leq \mathrm{Tower}(i) \cdot rac{n}{\mathrm{Tower}(i)} = n$.

Sariel (UIUC) CS573 34 Fall 2014 34 / 38

Lemma

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

Proof.

- $oldsymbol{0}$ x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
- $@ \dots$ all jumps through x are between blocks, by lemma...
- \odot total number of internal jumps $|\operatorname{Block}(i)| \cdot \frac{n}{\operatorname{Tower}(i)} \leq \operatorname{Tower}(i) \cdot \frac{n}{\operatorname{Tower}(i)} = n.$

Sariel (UIUC) CS573 34 Fall 2014 34 / 38

Lemma

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

- $oldsymbol{0}$ x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
- $oldsymbol{2}$... all jumps through $oldsymbol{x}$ are between blocks, by lemma...
- \odot total number of internal jumps $|\mathbf{Block}(i)| \cdot rac{n}{\mathrm{Tower}(i)} \leq \mathrm{Tower}(i) \cdot rac{n}{\mathrm{Tower}(i)} = n.$

Lemma

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

Proof.

- $oldsymbol{0}$ x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
- $oldsymbol{2}$... all jumps through $oldsymbol{x}$ are between blocks, by lemma...
- ① total number of internal jumps $|\operatorname{Block}(i)| \cdot \tfrac{n}{\operatorname{Tower}(i)} \leq \operatorname{Tower}(i) \cdot \tfrac{n}{\operatorname{Tower}(i)} = n.$

Sariel (UIUC) CS573 34 Fall 2014 34 / 38

Lemma

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

- $oldsymbol{0}$ x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
- $oldsymbol{2}$... all jumps through $oldsymbol{x}$ are between blocks, by lemma...
- **③** total number of internal jumps is $|\operatorname{Block}(i)| \cdot \frac{n}{\operatorname{Tower}(i)} \le \operatorname{Tower}(i) \cdot \frac{n}{\operatorname{Tower}(i)} = n$.

Lemma

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

Proof.

- $oldsymbol{0}$ x in ith block, have at most $|\mathrm{Block}(i)|$ internal jumps...
- $oldsymbol{2}$... all jumps through $oldsymbol{x}$ are between blocks, by lemma...
- total number of internal jumps is $|\mathrm{Block}(i)| \cdot \frac{n}{\mathrm{Tower}(i)} \leq \mathrm{Tower}(i) \cdot \frac{n}{\mathrm{Tower}(i)} = n$.

Sariel (UIUC) CS573 34 Fall 2014 34 / 38

Lemma

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

- Every internal jump associated with block it is in.
- @ Every block contributes O(n) internal jumps throughout time. (By previous lemma.)
- $exttt{3}$ There are $O(\log^* n)$ blocks.
- \odot There are at most $\mathit{O}(n\log^* n)$ internal jumps.

Lemma

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

- Every internal jump associated with block it is in.
- ullet Every block contributes O(n) internal jumps throughout time. (By previous lemma.)
- $exttt{3}$ There are $O(\log^* n)$ blocks.
- \odot There are at most $\mathit{O}(n\log^* n)$ internal jumps.



Lemma

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

- Every internal jump associated with block it is in.
- ullet Every block contributes O(n) internal jumps throughout time. (By previous lemma.)
- $exttt{3}$ There are $O(\log^* n)$ blocks.
- \odot There are at most $\mathit{O}(n\log^* n)$ internal jumps.

Lemma

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

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

Lemma

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

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

Sariel (UIUC) CS573 36 Fall 2014 36 / 38

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

	<u> </u>
Function	Inverse function

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

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

$$\overline{f_2(x) = f_1(f_2(x-1)) = 2x}$$

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

$$\overline{f_3(x)} = f_2(f_3(x-1)) = 2^x$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

 $g_i(x) = \#$ 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.

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

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

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

 $g_i(x) = \#$ 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:

$$lpha(n) = A^{-1}(n) = \min \, i \; ext{s.t.} \; g_i(n) \leq i.$$

Union-Find: Tarjan result

Theorem (?)

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

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

Sariel (UIUC) CS573 39 Fall 2014 39 / 38

Sariel (UIUC) CS573 40 Fall 2014 40 / 38

Sariel (UIUC) CS573 41 Fall 2014 41 / 38

Sariel (UIUC) CS573 42 Fall 2014 42 / 38