Disjoint Sets: Efficient Implementations

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Data Structures
Data Structures and Algorithms

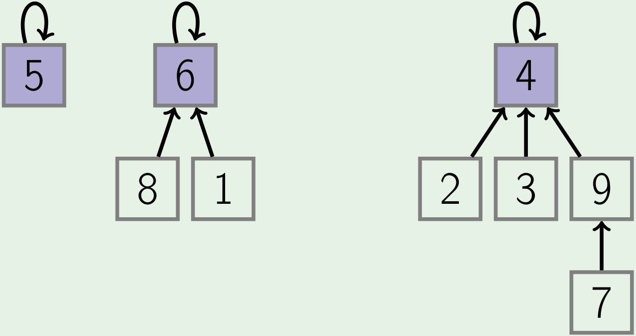
Outline

- Represent each set as a rooted tree

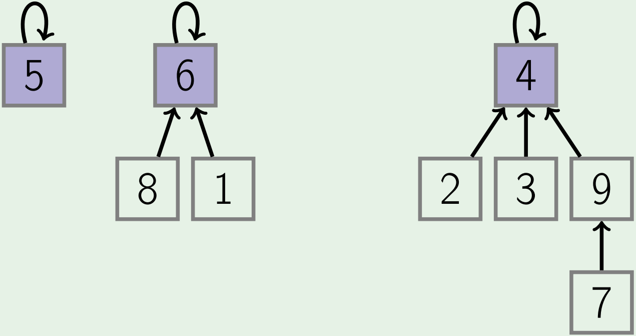
- Represent each set as a rooted tree
- ID of a set is the root of the tree

- Represent each set as a rooted tree
- ID of a set is the root of the tree
- Use array `parent`$[1 \ldots n]$: `parent`$[i]$ is the parent of i, or i if it is the root

Example



Example



	1	2	3	4	5	6	7	8	9
parent	6	4	4	4	5	6	9	6	4

`MakeSet`(i)

`parent`$[i] \leftarrow i$

`MakeSet`(i)

`parent`$[i] \leftarrow i$

Running time: $O(1)$

MakeSet(i)

```
parent[i] ← i
```

Running time: $O(1)$

Find(i)

```
while i ≠ parent[i]:
    i ← parent[i]
return i
```

`MakeSet(`i`)`

```
parent[i] ← i
```

Running time: $O(1)$

`Find(`i`)`

```
while i ≠ parent[i]:
   i ← parent[i]
return i
```

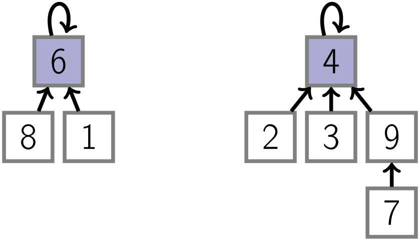
Running time: $O(\text{tree height})$

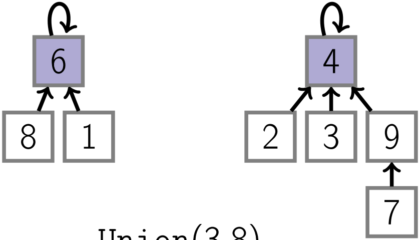
- How to merge two trees?

- How to merge two trees?
- Hang one of the trees under the root of the other one

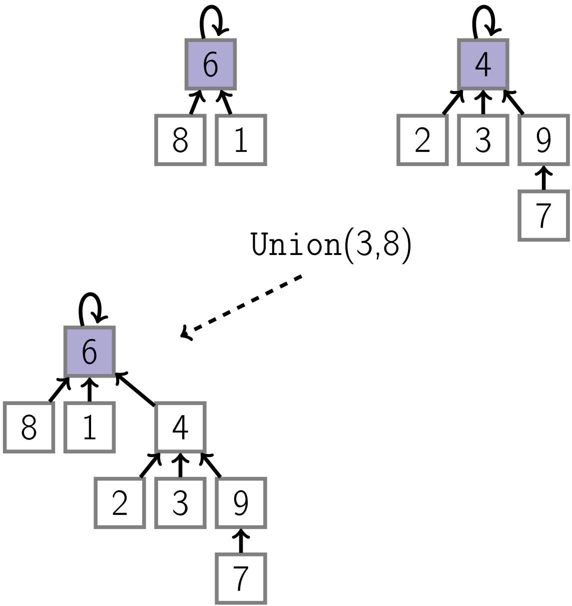
- How to merge two trees?
- Hang one of the trees under the root of the other one
- Which one to hang?

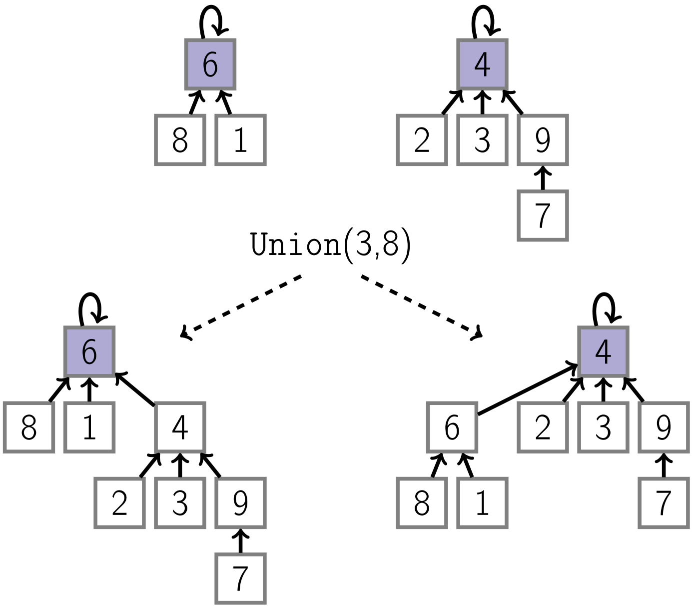
- How to merge two trees?
- Hang one of the trees under the root of the other one
- Which one to hang?
- A shorter one, since we would like to keep the trees shallow

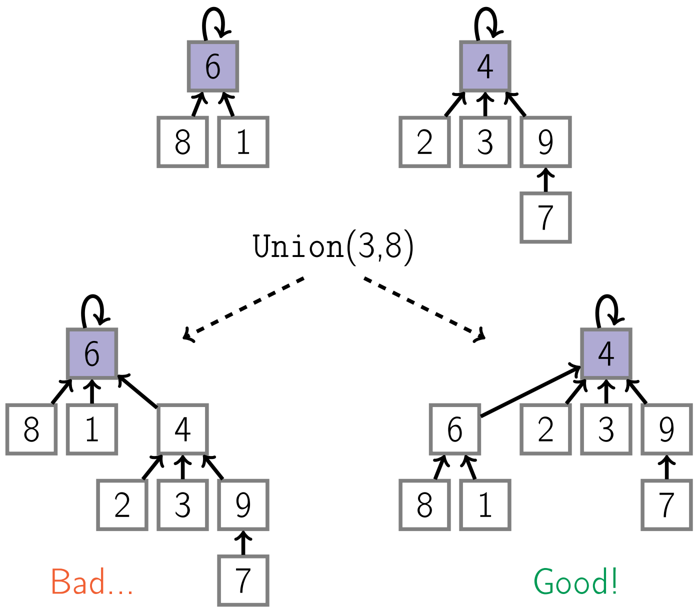

6
8
1
4
2
3
9
7



`Union(3,8)`


6
8
1
4
2
3
9
7
Union(3,8)
6
8
1
4
2
3
9
7


6
8
1
4
2
3
9
7
Union(3,8)
6
8
1
4
2
3
9
7
4
6
2
3
9
8
1
7



6
8
1
4
2
3
9
7
Union(3,8)
6
8
1
4
2
3
9
7
Bad...
4
6
2
3
9
8
1
7
Good!

Outline

1. Trees
2. Union by Rank
3. Path Compression
4. Analysis

- When merging two trees we hang a shorter one under the root of a taller one

- When merging two trees we hang a shorter one under the root of a taller one
- To quickly find a height of a tree, we will keep the height of each subtree in an array $\texttt{rank}[1 \dots n]$: $\texttt{rank}[i]$ is the height of the subtree whose root is i

- When merging two trees we hang a shorter one under the root of a taller one
- To quickly find a height of a tree, we will keep the height of each subtree in an array $\texttt{rank}[1 \dots n]$: $\texttt{rank}[i]$ is the height of the subtree whose root is i
- (The reason we call it `rank`, but not `height` will become clear later)

- When merging two trees we hang a shorter one under the root of a taller one
- To quickly find a height of a tree, we will keep the height of each subtree in an array $\texttt{rank}[1 \dots n]$: $\texttt{rank}[i]$ is the height of the subtree whose root is i
- (The reason we call it `rank`, but not `height` will become clear later)
- Hanging a shorter tree under a taller one is called a union by rank heuristic

```
MakeSet(i)

parent[i] ← i
rank[i] ← 0
```

```
Find(i)

while i ≠ parent[i]:
    i ← parent[i]
return i
```

Union(i, j)

```
i_id ← Find(i)
j_id ← Find(j)
if i_id = j_id:
  return
if rank[i_id] > rank[j_id]:
  parent[j_id] ← i_id
else:
  parent[i_id] ← j_id
  if rank[i_id] = rank[j_id]:
    rank[j_id] ← rank[j_id] + 1
```

Example

Query:

	1	2	3	4	5	6
`parent`						
`rank`						

Example

Query:

`MakeSet(1)`

`MakeSet(2)`

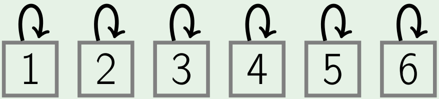
`...`

`MakeSet(6)`

	1	2	3	4	5	6
parent						
rank						

Example

Query:

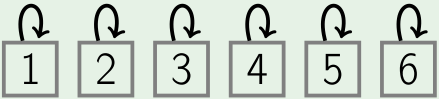


	1	2	3	4	5	6
`parent`	1	2	3	4	5	6
`rank`	0	0	0	0	0	0

Example

Query:

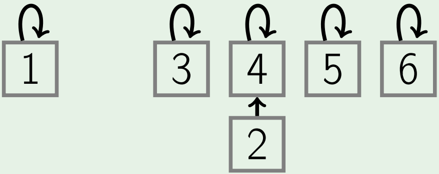
`Union(2, 4)`



	1	2	3	4	5	6
parent	1	2	3	4	5	6
rank	0	0	0	0	0	0

Example

Query:

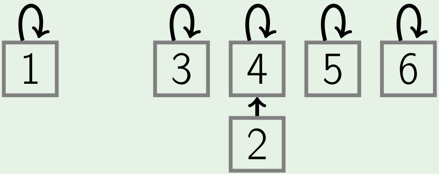


	1	2	3	4	5	6
`parent`	1	4	3	4	5	6
`rank`	0	0	0	1	0	0

Example

Query:

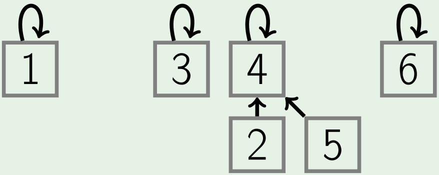
`Union(5, 2)`



	1	2	3	4	5	6
parent	1	4	3	4	5	6
rank	0	0	0	1	0	0

Example

Query:

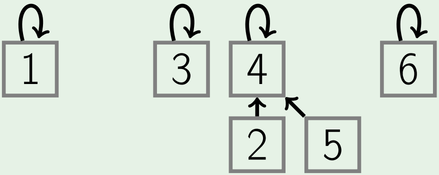


	1	2	3	4	5	6
`parent`	1	4	3	4	4	6
`rank`	0	0	0	1	0	0

Example

Query:

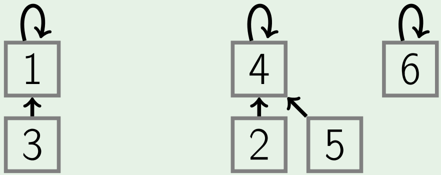
`Union(3, 1)`



	1	2	3	4	5	6
parent	1	4	3	4	4	6
rank	0	0	0	1	0	0

Example

Query:

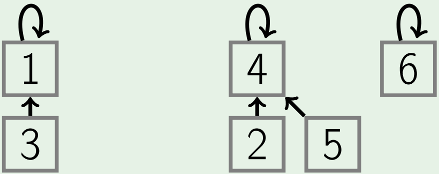


	1	2	3	4	5	6
`parent`	1	4	1	4	4	6
`rank`	1	0	0	1	0	0

Example

Query:

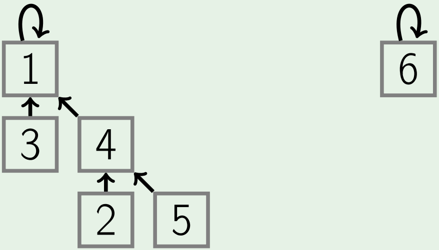
`Union`$(2, 3)$



	1	2	3	4	5	6
`parent`	1	4	1	4	4	6
`rank`	1	0	0	1	0	0

Example

Query:

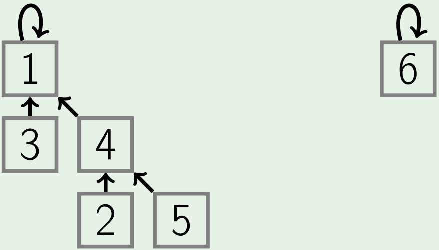


	1	2	3	4	5	6
`parent`	1	4	1	1	4	6
`rank`	2	0	0	1	0	0

Example

Query:

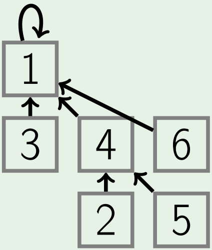
$\texttt{Union}(2, 6)$



	1	2	3	4	5	6
parent	1	4	1	1	4	6
rank	2	0	0	1	0	0

Example

Query:



	1	2	3	4	5	6
`parent`	1	4	1	1	4	1
`rank`	2	0	0	1	0	0

Important property: for any node i, rank[i] is equal to the height of the tree rooted at i

Lemma

The height of any tree in the forest is at most $\log_2 n$.

Lemma

The height of any tree in the forest is at most $\log_2 n$.

Follows from the following lemma.

Lemma

Any tree of height k in the forest has at least 2^k nodes.

Proof

Induction on k.

- Base: initially, a tree has height 0 and one node: $2^0 = 1$.
- Step: a tree of height k results from merging two trees of height $k - 1$. By induction hypothesis, each of two trees has at least 2^{k-1} nodes, hence the resulting tree contains at least 2^k nodes. $\square$

Summary

The union by rank heuristic guarantees that `Union` and `Find` work in time $O(\log n)$.

Summary

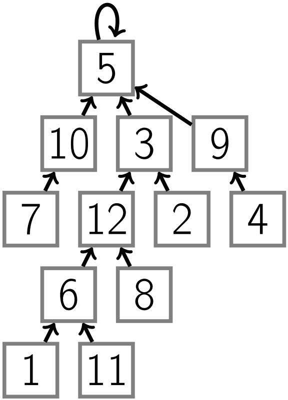
The union by rank heuristic guarantees that `Union` and `Find` work in time $O(\log n)$.

Next part

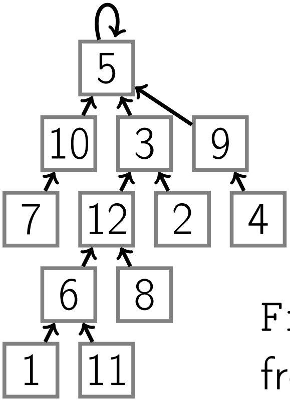
We'll discover another heuristic that improves the running time to nearly constant!

Outline

Path Compression: Intuition

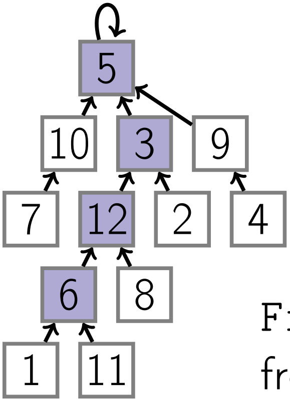


Path Compression: Intuition



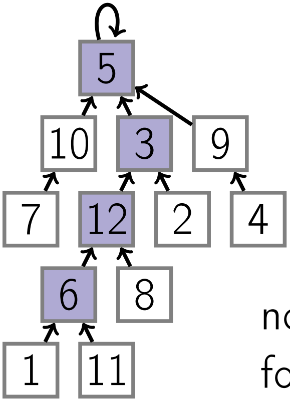
`Find`(6) traverses the path from 6 to the root

Path Compression: Intuition



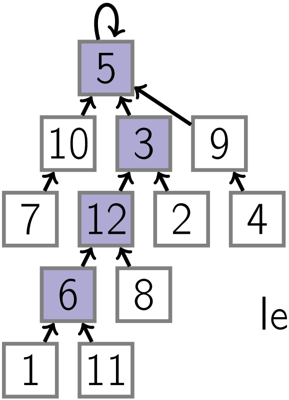
`Find`(6) traverses the path from 6 to the root

Path Compression: Intuition



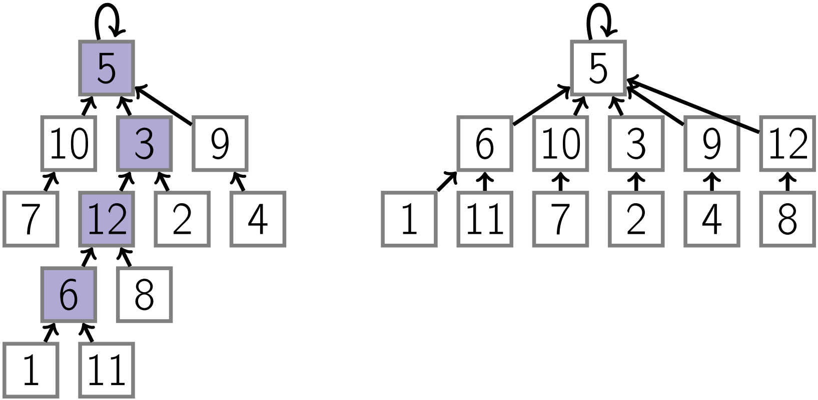
not only it finds the root for 6, it does so for all the nodes on this path

Path Compression: Intuition

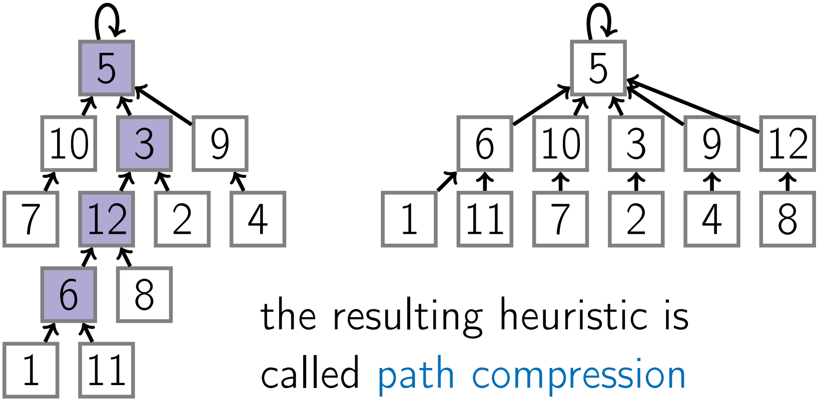


let's not lose this useful info

Path Compression: Intuition



Path Compression: Intuition



the resulting heuristic is called path compression

`Find`(i)

```
if  i ≠ parent[i]:
    parent[i] ← Find(parent[i])
return parent[i]
```

Definition

The iterated logarithm of n, $\log^* n$, is the number of times the logarithm function needs to be applied to n before the result is less or equal than 1:

$$\log^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Example

n	$\log^* n$
$n = 1$	0
$n = 2$	1
$n \in \{3, 4\}$	2
$n \in \{5, 6, \ldots, 16\}$	3
$n \in \{17, \ldots, 65536\}$	4
$n \in \{65537, \ldots, 2^{65536}\}$	5

Lemma

Assume that initially the data structure is empty. We make a sequence of m operations including n calls to `MakeSet`. Then the total running time is $O(m \log^* n)$.

In other words

The amortized time of a single operation is $O(\log^* n)$.

In other words

The amortized time of a single operation is $O(\log^* n)$.

Nearly constant!

For practical values of n, $\log^* n \leq 5$.

Outline

1. Trees
2. Union by Rank
3. Path Compression
4. Analysis

Goal

Prove that when both union by rank heuristic and path compression heuristic are used, the average running time of each operation is nearly constant.

Height $\leq$ Rank

- When using path compression, `rank`[i] is no longer equal to the height of the subtree rooted at i

Height $\leq$ Rank

- When using path compression, $\texttt{rank}[i]$ is no longer equal to the height of the subtree rooted at i
- Still, the height of the subtree rooted at i is at most $\texttt{rank}[i]$

Height $\leq$ Rank

- When using path compression, $\texttt{rank}[i]$ is no longer equal to the height of the subtree rooted at i
- Still, the height of the subtree rooted at i is at most $\texttt{rank}[i]$
- And it is still true that a root node of rank k has at least 2^k nodes in its subtree: a root node is not affected by path compression

Important Properties

1. There are at most $\frac{n}{2^k}$ nodes of rank k

Important Properties

1. There are at most $\frac{n}{2^k}$ nodes of rank k
2. For any node i,
   $\texttt{rank}[i] < \texttt{rank}[\texttt{parent}[i]]$

Important Properties

1. There are at most $\frac{n}{2^k}$ nodes of rank k
2. For any node i, $\texttt{rank}[i] < \texttt{rank}[\texttt{parent}[i]]$
3. Once an internal node, always an internal node

$T(\text{all calls to } \texttt{Find}) =$

$\#(i \to j) =$

$\#(i \to j : j \text{ is a root}) +$

$\#(i \to j :\ \log^*(\texttt{rank}[i]) < \log^*(\texttt{rank}[j])) +$

$\#(i \to j :\ \log^*(\texttt{rank}[i]) = \log^*(\texttt{rank}[j]))$

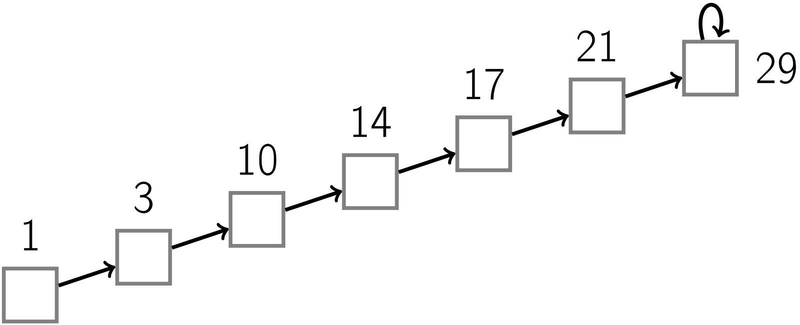
$T(\text{all calls to } \texttt{Find}) =$

$\#(i \to j) =$

$\#(i \to j : j \text{ is a root}) +$

$\#(i \to j : \log^*(\texttt{rank}[i]) < \log^*(\texttt{rank}[j])) +$

$\#(i \to j : \log^*(\texttt{rank}[i]) = \log^*(\texttt{rank}[j]))$



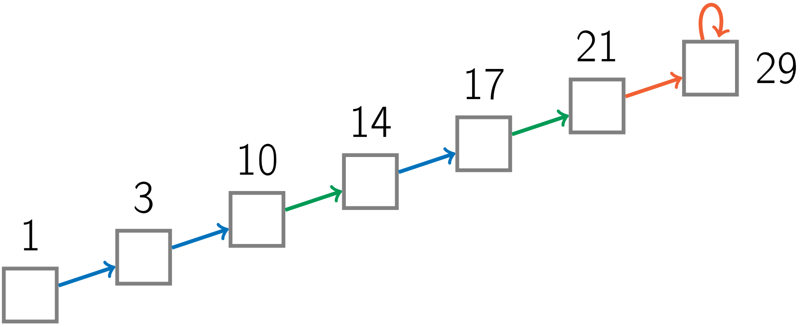
$$T(\text{all calls to } \texttt{Find}) =$$

$$\#(i \to j) =$$

$$\#(i \to j : j \text{ is a root}) +$$

$$\#(i \to j : \log^*(\texttt{rank}[i]) < \log^*(\texttt{rank}[j])) +$$

$$\#(i \to j : \log^*(\texttt{rank}[i]) = \log^*(\texttt{rank}[j]))$$



Claim

$\#(i \to j : j \text{ is a root}) \leq O(m)$

Claim

$\#(i \to j : j \text{ is a root}) \leq O(m)$

Proof

There are at most m calls to `Find`. $\square$

Claim

$$\#(i \to j:\ \log^*(\mathtt{rank}[i]) < \log^*(\mathtt{rank}[j])) \leq O(m \log^* n)$$

Claim

$$\#(i \to j : \ \log^*(\mathtt{rank}[i]) < \log^*(\mathtt{rank}[j])) \\ \le O(m \log^* n)$$

Proof

There are at most $\log^* n$ different values for $\log^*(\mathtt{rank})$. $\square$

Claim

$$\#(i \to j\colon \log^*(\texttt{rank}[i]) = \log^*(\texttt{rank}[j])) \leq O(n \log^* n)$$

Proof

- assume $\texttt{rank}[i] \in \{k+1, \ldots, 2^k\}$

Proof

- assume $\texttt{rank}[i] \in \{k+1, \dots, 2^k\}$
- the number of nodes with rank lying in this interval is at most

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}$$

Proof

- assume $\texttt{rank}[i] \in \{k+1, \ldots, 2^k\}$
- the number of nodes with rank lying in this interval is at most
$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}$$
- after a call to $\texttt{Find}(i)$, the node i is adopted by a new parent of strictly larger rank

Proof

- assume $\texttt{rank}[i] \in \{k+1, \ldots, 2^k\}$
- the number of nodes with rank lying in this interval is at most

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}$$

- after a call to $\texttt{Find}(i)$, the node i is adopted by a new parent of strictly larger rank
- after at most 2^k calls to $\texttt{Find}(i)$, the parent of i will have rank from a different interval

Proof (Continued)

- there are at most $\frac{n}{2^k}$ nodes with rank in $\{k+1, \ldots, 2^k\}$

Proof (Continued)

- there are at most $\frac{n}{2^k}$ nodes with rank in $\{k+1, \ldots, 2^k\}$
- each of them contributes at most 2^k

Proof (Continued)

- there are at most $\frac{n}{2^k}$ nodes with rank in $\{k+1, \ldots, 2^k\}$
- each of them contributes at most 2^k
- the contribution of all the nodes with rank from this interval is at most $O(n)$

Proof (Continued)

- there are at most $\frac{n}{2^k}$ nodes with rank in $\{k+1, \ldots, 2^k\}$
- each of them contributes at most 2^k
- the contribution of all the nodes with rank from this interval is at most $O(n)$
- the number of different intervals is $\log^* n$

Proof (Continued)

- there are at most $\frac{n}{2^k}$ nodes with rank in $\{k+1, \ldots, 2^k\}$
- each of them contributes at most 2^k
- the contribution of all the nodes with rank from this interval is at most $O(n)$
- the number of different intervals is $\log^* n$
- thus, the contribution of all nodes is $O(n \log^* n)$ $\square$

Summary

- Represent each set as a rooted tree

Summary

- Represent each set as a rooted tree
- Use the root of the set as its ID

Summary

- Represent each set as a rooted tree
- Use the root of the set as its ID
- Union by rank heuristic: hang a shorter tree under the root of a taller one

Summary

- Represent each set as a rooted tree
- Use the root of the set as its ID
- Union by rank heuristic: hang a shorter tree under the root of a taller one
- Path compression heuristic: when finding the root of a tree for a particular node, reattach each node from the traversed path to the root

Summary

- Represent each set as a rooted tree
- Use the root of the set as its ID
- Union by rank heuristic: hang a shorter tree under the root of a taller one
- Path compression heuristic: when finding the root of a tree for a particular node, reattach each node from the traversed path to the root
- Amortized running time: $O(\log^* n)$ (constant for practical values of n)