9 Union Find

Union Find Data Structure $\mathcal{P}$: Maintains a partition of disjoint sets over elements.

9 Union Find

Union Find Data Structure $\mathcal{P}$: Maintains a partition of disjoint sets over elements.

- $\mathcal{P}.\mathbf{makeset}(x)$**:** Given an element x, adds x to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for x in the data-structure.

9 Union Find

Union Find Data Structure $\mathcal{P}$: Maintains a partition of disjoint sets over elements.

- **$\mathcal{P}.\mathbf{makeset}(x)$:** Given an element x, adds x to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for x in the data-structure.
- **$\mathcal{P}.\mathbf{find}(x)$:** Given a handle for an element x; find the set that contains x. Returns a representative/identifier for this set.

9 Union Find

Union Find Data Structure $\mathcal{P}$: Maintains a partition of disjoint sets over elements.

- **$\mathcal{P}$. makeset(x):** Given an element x, adds x to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for x in the data-structure.
- **$\mathcal{P}$. find(x):** Given a handle for an element x; find the set that contains x. Returns a representative/identifier for this set.
- **$\mathcal{P}$. union(x, y):** Given two elements x, and y that are currently in sets S_x and S_y, respectively, the function replaces S_x and S_y by $S_x \cup S_y$ and returns an identifier for the new set.

9 Union Find

Applications:

- Keep track of the connected components of a dynamic graph that changes due to insertion of nodes and edges.

9 Union Find

Applications:

- Keep track of the connected components of a dynamic graph that changes due to insertion of nodes and edges.
- Kruskals Minimum Spanning Tree Algorithm

9 Union Find

Algorithm 1 Kruskal-MST($G = (V, E), w$)

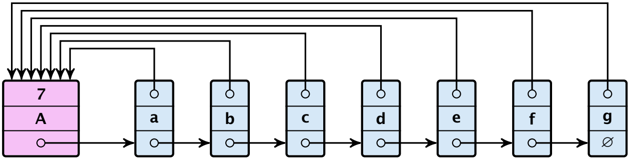
$A \leftarrow \emptyset$;
for all $v \in V$ **do**
    $v.\mathrm{set} \leftarrow \mathcal{P}.\mathrm{makeset}(v.\mathrm{label})$
sort edges in non-decreasing order of weight w
for all $(u, v) \in E$ in non-decreasing order **do**
    if $\mathcal{P}.\mathrm{find}(u.\mathrm{set}) \neq \mathcal{P}.\mathrm{find}(v.\mathrm{set})$ **then**
        $A \leftarrow A \cup \{(u, v)\}$
        $\mathcal{P}.\mathrm{union}(u.\mathrm{set}, v.\mathrm{set})$

List Implementation

- The elements of a set are stored in a list; each node has a backward pointer to the head.

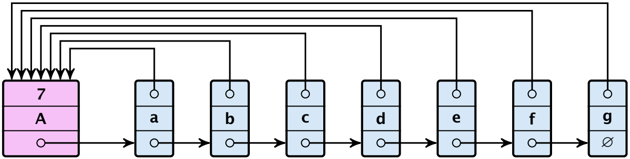
List Implementation

- The elements of a set are stored in a list; each node has a backward pointer to the head.
- The head of the list contains the identifier for the set and a field that stores the size of the set.



List Implementation

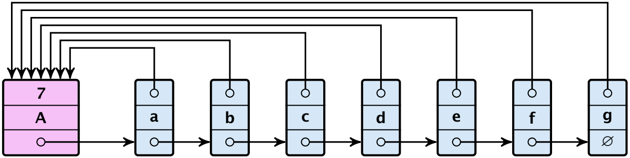
- The elements of a set are stored in a list; each node has a backward pointer to the head.
- The head of the list contains the identifier for the set and a field that stores the size of the set.



- $\text{makeset}(x)$ can be performed in constant time.

List Implementation

- The elements of a set are stored in a list; each node has a backward pointer to the head.
- The head of the list contains the identifier for the set and a field that stores the size of the set.



- makeset(x) can be performed in constant time.
- find(x) can be performed in constant time.

List Implementation

union(x, y)

- Determine sets S_x and S_y.

List Implementation

union(x, y)

- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_x.

List Implementation

union(x, y)

- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_x.
- Insert list S_y at the head of S_x.

List Implementation

union(x, y)

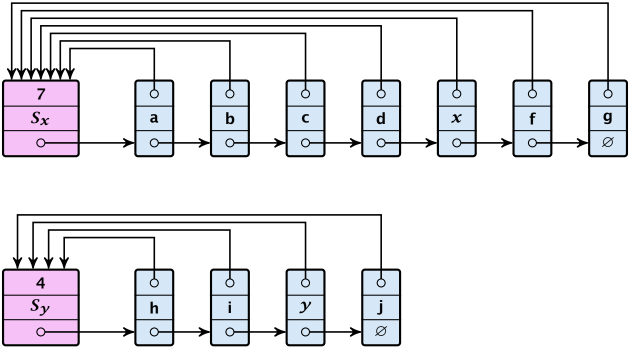
- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_x.
- Insert list S_y at the head of S_x.
- Adjust the size-field of list S_x.

List Implementation

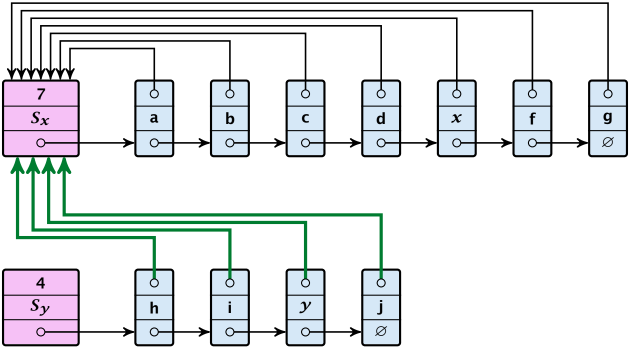
union(x, y)

- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_x.
- Insert list S_y at the head of S_x.
- Adjust the size-field of list S_x.
- Time: $\min\{|S_x|, |S_y|\}$.

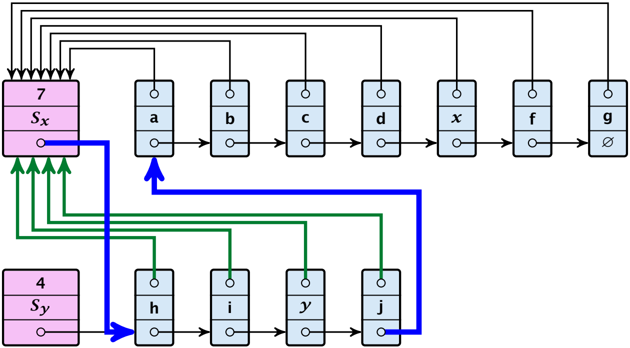
List Implementation



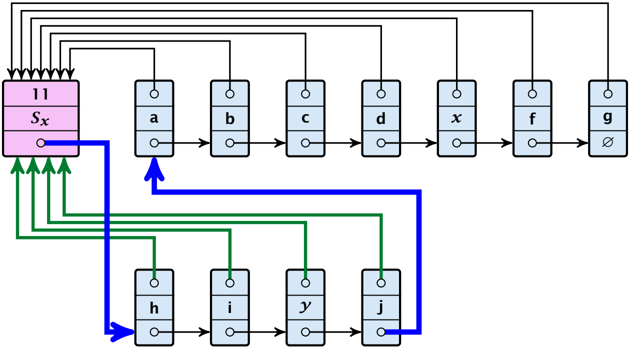
List Implementation



List Implementation



List Implementation



List Implementation

Running times:

- $\text{find}(x)$: constant
- $\text{makeset}(x)$: constant
- $\text{union}(x, y)$: $\mathcal{O}(n)$, where n denotes the number of elements contained in the set system.

List Implementation

Lemma 1

The list implementation for the ADT union find fulfills the following amortized time bounds:

- $\text{find}(x)$: $\mathcal{O}(1)$.
- $\text{makeset}(x)$: $\mathcal{O}(\log n)$.
- $\text{union}(x, y)$: $\mathcal{O}(1)$.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.
- If we can find a charging scheme that guarantees that balances always stay positive the amortized time bounds are proven.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).
- For each element a makeset operation occurs as the first operation involving this element.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).
- For each element a makeset operation occurs as the first operation involving this element.
- We inflate the amortized cost of the makeset-operation to $\Theta(\log n)$, i.e., at this point we fill the bank account of the element to $\Theta(\log n)$.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).
- For each element a makeset operation occurs as the first operation involving this element.
- We inflate the amortized cost of the makeset-operation to $\Theta(\log n)$, i.e., at this point we fill the bank account of the element to $\Theta(\log n)$.
- Later operations charge the account but the balance never drops below zero.

List Implementation

makeset(x): The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

List Implementation

makeset(x): The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x): For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

List Implementation

makeset(x): The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x): For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.

List Implementation

makeset(x): The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x): For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.

List Implementation

makeset(x): The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x): For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.
- Assume wlog. that S_x is the smaller set; let c denote the hidden constant, i.e., the actual cost is at most $c \cdot |S_x|$.

List Implementation

makeset(x): The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x): For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.
- Assume wlog. that S_x is the smaller set; let c denote the hidden constant, i.e., the actual cost is at most $c \cdot |S_x|$.
- Charge c to every element in set S_x.

List Implementation

Lemma 2
An element is charged at most $\lfloor \log_2 n \rfloor$ times, where n is the total number of elements in the set system.

List Implementation

Lemma 2

An element is charged at most $\lfloor \log_2 n \rfloor$ times, where n is the total number of elements in the set system.

Proof.

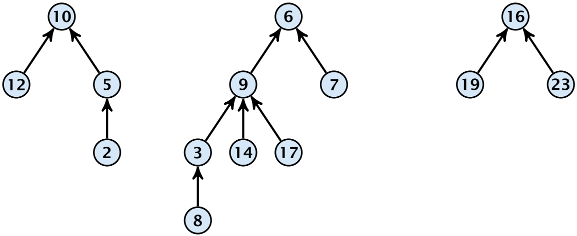
Whenever an element x is charged the number of elements in x's set doubles. This can happen at most $\lfloor \log n \rfloor$ times. $\square$

Implementation via Trees

- Maintain nodes of a set in a tree.
- The root of the tree is the label of the set.
- Only pointer to parent exists; we cannot list all elements of a given set.

Implementation via Trees

- Maintain nodes of a set in a tree.
- The root of the tree is the label of the set.
- Only pointer to parent exists; we cannot list all elements of a given set.
- Example:



Set system $\{2, 5, 10, 12\}$, $\{3, 6, 7, 8, 9, 14, 17\}$, $\{16, 19, 23\}$.

Implementation via Trees

makeset(x)

- Create a singleton tree. Return pointer to the root.

Implementation via Trees

makeset(x)

- Create a singleton tree. Return pointer to the root.
- Time: $\mathcal{O}(1)$.

Implementation via Trees

makeset(x)

- Create a singleton tree. Return pointer to the root.
- Time: $\mathcal{O}(1)$.

find(x)

- Start at element x in the tree. Go upwards until you reach the root.

Implementation via Trees

makeset(x)

- Create a singleton tree. Return pointer to the root.
- Time: $\mathcal{O}(1)$.

find(x)

- Start at element x in the tree. Go upwards until you reach the root.
- Time: $\mathcal{O}(\text{level}(x))$, where $\text{level}(x)$ is the distance of element x to the root in its tree. Not constant.

Implementation via Trees

To support union we store the size of a tree in its root.

Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \mathrm{find}(x)$; $b \leftarrow \mathrm{find}(y)$. Then: $\mathrm{link}(a, b)$.

Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \mathrm{find}(x)$; $b \leftarrow \mathrm{find}(y)$. Then: $\mathrm{link}(a, b)$.
- $\mathrm{link}(a, b)$ attaches the smaller tree as the child of the larger.

Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

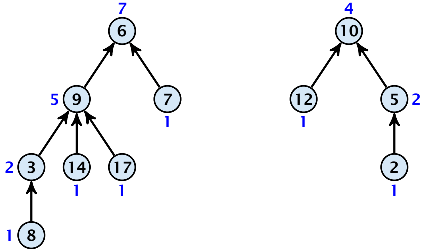
- Perform $a \leftarrow \mathrm{find}(x)$; $b \leftarrow \mathrm{find}(y)$. Then: $\mathrm{link}(a, b)$.
- $\mathrm{link}(a, b)$ attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.

Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \text{find}(x)$; $b \leftarrow \text{find}(y)$. Then: $\text{link}(a, b)$.
- $\text{link}(a, b)$ attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.

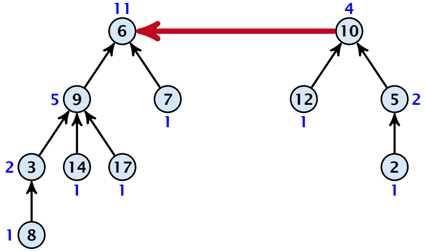


Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \text{find}(x)$; $b \leftarrow \text{find}(y)$. Then: $\text{link}(a, b)$.
- $\text{link}(a, b)$ attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.

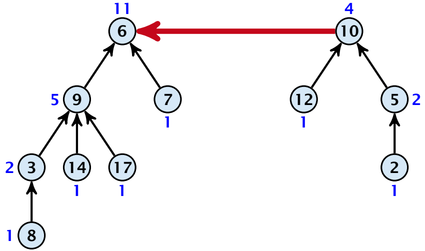


Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \text{find}(x)$; $b \leftarrow \text{find}(y)$. Then: $\text{link}(a, b)$.
- $\text{link}(a, b)$ attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.



- Time: constant for $\text{link}(a, b)$ plus two find-operations.

Implementation via Trees

Lemma 3
The running time (non-amortized!!!) for $\mathrm{find}(x)$ *is* $\mathcal{O}(\log n)$.

Implementation via Trees

Lemma 3

The running time (non-amortized!!!) for $\text{find}(x)$ *is* $\mathcal{O}(\log n)$.

Proof.

- When we attach a tree with root c to become a child of a tree with root p, then $\text{size}(p) \geq 2\,\text{size}(c)$, where size denotes the value of the size-field right after the operation.

Implementation via Trees

Lemma 3
The running time (non-amortized!!!) for $\mathrm{find}(x)$ *is* $\mathcal{O}(\log n)$.

Proof.

- When we attach a tree with root c to become a child of a tree with root p, then $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, where size denotes the value of the size-field right after the operation.
- After that the value of $\mathrm{size}(c)$ stays fixed, while the value of $\mathrm{size}(p)$ may still increase.

Implementation via Trees

Lemma 3
The running time (non-amortized!!!) for $\text{find}(x)$ *is* $\mathcal{O}(\log n)$.

Proof.

- When we attach a tree with root c to become a child of a tree with root p, then $\text{size}(p) \geq 2\,\text{size}(c)$, where size denotes the value of the size-field right after the operation.
- After that the value of $\text{size}(c)$ stays fixed, while the value of $\text{size}(p)$ may still increase.
- Hence, at any point in time a tree fulfills $\text{size}(p) \geq 2\,\text{size}(c)$, for any pair of nodes (p, c), where p is a parent of c.

Implementation via Trees

Lemma 3

The running time (non-amortized!!!) for $\text{find}(x)$ *is* $\mathcal{O}(\log n)$.

Proof.

- When we attach a tree with root c to become a child of a tree with root p, then $\text{size}(p) \geq 2\,\text{size}(c)$, where size denotes the value of the size-field right after the operation.
- After that the value of $\text{size}(c)$ stays fixed, while the value of $\text{size}(p)$ may still increase.
- Hence, at any point in time a tree fulfills $\text{size}(p) \geq 2\,\text{size}(c)$, for any pair of nodes (p, c), where p is a parent of c.

$\square$

Path Compression

find(x):

- Go upward until you find the root.

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.

Path Compression

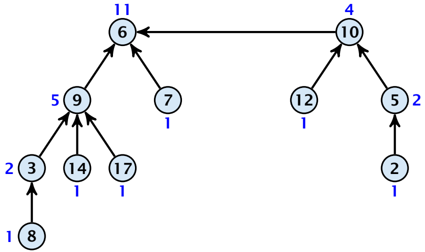
find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

Path Compression

find(x):

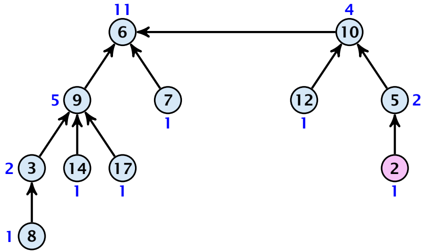
- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.



Path Compression

find(x):

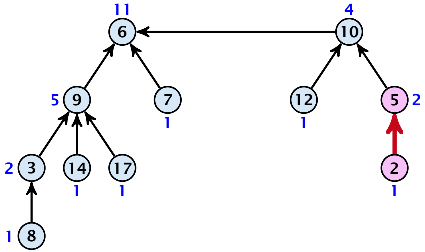
- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.



Path Compression

find(x):

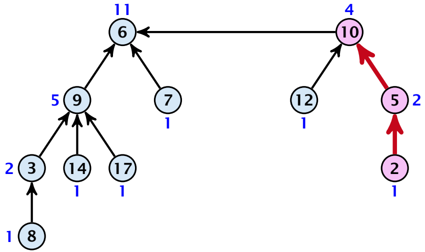
- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.



Path Compression

find(x):

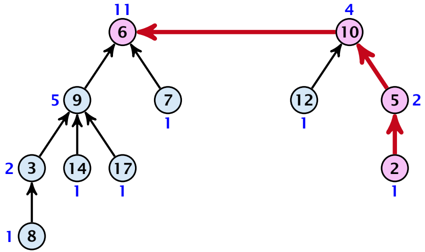
- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.



Path Compression

find(x):

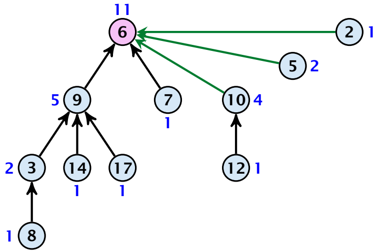
- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.



Path Compression

find(x):

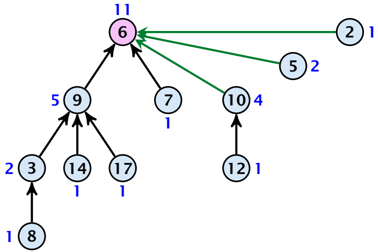
- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.



Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.



- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

Asymptotically the cost for a find-operation does not increase due to the path compression heuristic.

Path Compression

Asymptotically the cost for a find-operation does not increase due to the path compression heuristic.

However, for a worst-case analysis there is no improvement on the running time. It can still happen that a find-operation takes time $\mathcal{O}(\log n)$.

Amortized Analysis

Definitions:

Amortized Analysis

Definitions:

- $\text{size}(v) \coloneqq$ the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).

  Note that this is the same as the size of v's subtree in the case that there are no find-operations.

Amortized Analysis

Definitions:

- $\text{size}(v) :=$ the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).

  Note that this is the same as the size of v's subtree in the case that there are no find-operations.
- $\text{rank}(v) := \lfloor \log(\text{size}(v)) \rfloor$.

Amortized Analysis

Definitions:

- $\mathrm{size}(v) :=$ the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).

  Note that this is the same as the size of v's subtree in the case that there are no find-operations.
- $\mathrm{rank}(v) := \lfloor \log(\mathrm{size}(v)) \rfloor$.
- $\Longrightarrow \mathrm{size}(v) \geq 2^{\mathrm{rank}(v)}$.

Amortized Analysis

Definitions:

- $\mathrm{size}(v) :=$ the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).

  Note that this is the same as the size of v's subtree in the case that there are no find-operations.
- $\mathrm{rank}(v) := \lfloor \log(\mathrm{size}(v)) \rfloor$.
- $\Longrightarrow \mathrm{size}(v) \geq 2^{\mathrm{rank}(v)}$.

Lemma 4

The rank of a parent must be strictly larger than the rank of a child.

Amortized Analysis

Lemma 5
There are at most $n/2^s$ nodes of rank s.

Amortized Analysis

Lemma 5

There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees node x if v is in x's sub-tree at the time that x becomes a child.

$\square$

Amortized Analysis

Lemma 5

There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees node x if v is in x's sub-tree at the time that x becomes a child.
- A node v sees at most one node of rank s during the running time of the algorithm.

$\square$

Amortized Analysis

Lemma 5

There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees node x if v is in x's sub-tree at the time that x becomes a child.
- A node v sees at most one node of rank s during the running time of the algorithm.
- This holds because the rank-sequence of the roots of the different trees that contain v during the running time of the algorithm is a strictly increasing sequence.

□

Amortized Analysis

Lemma 5
There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees node x if v is in x's sub-tree at the time that x becomes a child.
- A node v sees at most one node of rank s during the running time of the algorithm.
- This holds because the rank-sequence of the roots of the different trees that contain v during the running time of the algorithm is a strictly increasing sequence.
- Hence, every node *sees* at most one rank s node, but every rank s node is seen by at least 2^s different nodes. $\square$

Amortized Analysis

We define

$$\mathrm{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\mathrm{tow}(i-1)} & \text{otw.} \end{cases}$$

Amortized Analysis

We define

$$\mathrm{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\mathrm{tow}(i-1)} & \text{otw.} \end{cases} \qquad \mathrm{tow}(i) = \left.2^{2^{2^{2^{2^{2}}}}}\right\} i \text{ times}$$

Amortized Analysis

We define

$$\mathrm{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\mathrm{tow}(i-1)} & \text{otw.} \end{cases} \qquad \mathrm{tow}(i) = \left.2^{2^{2^{2^{2}}}}\right\} i \text{ times}$$

and

$$\log^*(n) := \min\{i \mid \mathrm{tow}(i) \geq n\} \ .$$

Amortized Analysis

We define

$$\mathrm{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\mathrm{tow}(i-1)} & \text{otw.} \end{cases} \qquad \mathrm{tow}(i) = 2^{2^{2^{2^{2}}}} \Big\} i \text{ times}$$

and

$$\log^*(n) := \min\{i \mid \mathrm{tow}(i) \geq n\} \ .$$

Theorem 6
Union find with path compression fulfills the following amortized running times:

- $\mathrm{makeset}(x) : \mathcal{O}(\log^*(n))$
- $\mathrm{find}(x) : \mathcal{O}(\log^*(n))$
- $\mathrm{union}(x, y) : \mathcal{O}(\log^*(n))$

Amortized Analysis

In the following we assume $n \geq 2$.

Amortized Analysis

In the following we assume $n \geq 2$.

rank-group:

- A node with rank $\mathrm{rank}(v)$ is in rank group $\log^*(\mathrm{rank}(v))$.

Amortized Analysis

In the following we assume $n \geq 2$.

rank-group:

- A node with rank $\text{rank}(v)$ is in rank group $\log^*(\text{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.

Amortized Analysis

In the following we assume $n \geq 2$.

rank-group:

- A node with rank $\text{rank}(v)$ is in rank group $\log^*(\text{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\text{tow}(g-1)+1, \dots, \text{tow}(g)$.

Amortized Analysis

In the following we assume $n \geq 2$.

rank-group:

- A node with rank $\text{rank}(v)$ is in rank group $\log^*(\text{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\text{tow}(g-1)+1,\ldots,\text{tow}(g)$.
- The maximum non-empty rank group is $\log^*(\lfloor \log n \rfloor) \leq \log^*(n) - 1$ (which holds for $n \geq 2$).

Amortized Analysis

In the following we assume $n \geq 2$.

rank-group:

- A node with rank $\text{rank}(v)$ is in rank group $\log^*(\text{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\text{tow}(g-1)+1, \dots, \text{tow}(g)$.
- The maximum non-empty rank group is $\log^*(\lfloor \log n \rfloor) \leq \log^*(n) - 1$ (which holds for $n \geq 2$).
- Hence, the total number of rank-groups is at most $\log^* n$.

Amortized Analysis

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\mathrm{parent}[v]$ as follows:

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\text{parent}[v]$ as follows:

- If $\text{parent}[v]$ is the root we charge the cost to the find-account.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\text{parent}[v]$ as follows:

- If $\text{parent}[v]$ is the root we charge the cost to the find-account.
- If the group-number of $\text{rank}(v)$ is the same as that of $\text{rank}(\text{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\text{parent}[v]$ as follows:

- If $\text{parent}[v]$ is the root we charge the cost to the find-account.
- If the group-number of $\text{rank}(v)$ is the same as that of $\text{rank}(\text{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.
- Otherwise we charge the cost to the find-account.

Amortized Analysis

Observations:

Amortized Analysis

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).

Amortized Analysis

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node v is charged its parent-edge is re-assigned. The rank of the parent strictly increases.

Amortized Analysis

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node v is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- After some charges to v the parent will be in a larger rank-group. $\Longrightarrow v$ will never be charged again.

Amortized Analysis

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node v is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- After some charges to v the parent will be in a larger rank-group. $\Longrightarrow v$ will never be charged again.
- The total charge made to a node in rank-group g is at most $\mathrm{tow}(g) - \mathrm{tow}(g-1) - 1 \le \mathrm{tow}(g)$.

Amortized Analysis

What is the total charge made to nodes?

Amortized Analysis

What is the total charge made to nodes?

- The total charge is at most

$$\sum_g n(g) \cdot \operatorname{tow}(g) \ ,$$

where $n(g)$ is the number of nodes in group g.

Amortized Analysis

For $g \geq 1$ we have

$$n(g)$$

Amortized Analysis

For $g \geq 1$ we have

$$n(g) \leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s}$$

Amortized Analysis

For $g \geq 1$ we have

$$n(g) \leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\mathrm{tow}(g-1)+1}^{\infty} \frac{n}{2^s}$$

Amortized Analysis

For $g \geq 1$ we have

$$n(g) \leq \sum_{s=\text{tow}(g-1)+1}^{\text{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\text{tow}(g-1)+1}^{\infty} \frac{n}{2^s}$$

$$= \frac{n}{2^{\text{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s}$$

Amortized Analysis

For $g \geq 1$ we have

$$
\begin{aligned}
n(g) &\leq \sum_{s=\text{tow}(g-1)+1}^{\text{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\text{tow}(g-1)+1}^{\infty} \frac{n}{2^s} \\
&= \frac{n}{2^{\text{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} = \frac{n}{2^{\text{tow}(g-1)+1}} \cdot 2
\end{aligned}
$$

Amortized Analysis

For $g \geq 1$ we have

$$
\begin{aligned}
n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\mathrm{tow}(g-1)+1}^{\infty} \frac{n}{2^s} \\
&= \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \cdot 2 \\
&= \frac{n}{2^{\mathrm{tow}(g-1)}}
\end{aligned}
$$

Amortized Analysis

For $g \geq 1$ we have

$$
\begin{aligned}
n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\mathrm{tow}(g-1)+1}^{\infty} \frac{n}{2^s} \\
&= \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \cdot 2 \\
&= \frac{n}{2^{\mathrm{tow}(g-1)}} = \frac{n}{\mathrm{tow}(g)} \enspace .
\end{aligned}
$$

Amortized Analysis

For $g \geq 1$ we have

$$
\begin{aligned}
n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\mathrm{tow}(g-1)+1}^{\infty} \frac{n}{2^s} \\
&= \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \cdot 2 \\
&= \frac{n}{2^{\mathrm{tow}(g-1)}} = \frac{n}{\mathrm{tow}(g)} \enspace .
\end{aligned}
$$

Hence,

$$\sum_g n(g)\,\mathrm{tow}(g)$$

Amortized Analysis

For $g \geq 1$ we have

$$
\begin{aligned}
n(g) &\leq \sum_{s=\operatorname{tow}(g-1)+1}^{\operatorname{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\operatorname{tow}(g-1)+1}^{\infty} \frac{n}{2^s} \\
&= \frac{n}{2^{\operatorname{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} = \frac{n}{2^{\operatorname{tow}(g-1)+1}} \cdot 2 \\
&= \frac{n}{2^{\operatorname{tow}(g-1)}} = \frac{n}{\operatorname{tow}(g)} \ .
\end{aligned}
$$

Hence,

$$
\sum_g n(g) \operatorname{tow}(g) \leq n(0) \operatorname{tow}(0) + \sum_{g \geq 1} n(g) \operatorname{tow}(g)
$$

Amortized Analysis

For $g \geq 1$ we have

$$\begin{aligned} n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\mathrm{tow}(g-1)+1}^{\infty} \frac{n}{2^s} \\ &= \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \cdot 2 \\ &= \frac{n}{2^{\mathrm{tow}(g-1)}} = \frac{n}{\mathrm{tow}(g)} \ . \end{aligned}$$

Hence,

$$\sum_g n(g)\,\mathrm{tow}(g) \leq n(0)\,\mathrm{tow}(0) + \sum_{g \geq 1} n(g)\,\mathrm{tow}(g) \leq n \log^*(n)$$

Amortized Analysis

Without loss of generality we can assume that all makeset-operations occur at the start.

Amortized Analysis

Without loss of generality we can assume that all makeset-operations occur at the start.

This means if we inflate the cost of makeset to $\log^* n$ and add this to the node account of v then the balances of all node accounts will sum up to a positive value (this is sufficient to obtain an amortized bound).

Amortized Analysis

Amortized Analysis

The analysis is not tight. In fact it has been shown that the amortized time for the union-find data structure with path compression is $\mathcal{O}(\alpha(m, n))$, where $\alpha(m, n)$ is the inverse Ackermann function which grows a lot lot slower than $\log^* n$. (Here, we consider the average running time of m operations on at most n elements).

Amortized Analysis

The analysis is not tight. In fact it has been shown that the amortized time for the union-find data structure with path compression is $\mathcal{O}(\alpha(m,n))$, where $\alpha(m,n)$ is the inverse Ackermann function which grows a lot lot slower than $\log^* n$. (Here, we consider the average running time of m operations on at most n elements).

There is also a lower bound of $\Omega(\alpha(m,n))$.

Amortized Analysis

$$A(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1, A(x, y-1)) & \text{otw.} \end{cases}$$

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log n\}$$

Amortized Analysis

$$A(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1, A(x,y-1)) & \text{otw.} \end{cases}$$

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log n\}$$

- $A(0,y) = y+1$
- $A(1,y) = y+2$
- $A(2,y) = 2y+3$
- $A(3,y) = 2^{y+3} - 3$
- $A(4,y) = \underbrace{2^{2^{2^{2}}}}_{y+3 \text{ times}} - 3$