CS3383 Unit 2.2: Union Find / Disjoint Set

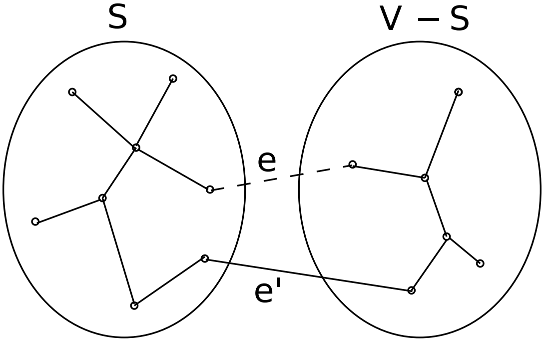
David Bremner

February 23, 2018

Outline

Contents

Minimum Spanning Trees: Cut Property

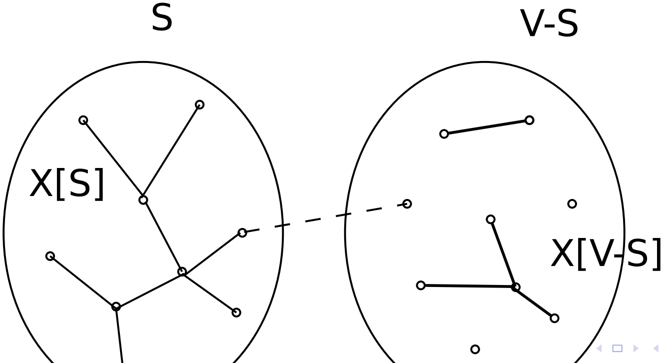


Lemma

Let T be a minimum spanning tree, $X \subset T$ s.t. X does not connect $(S, V - S)$. Let e be the lightest edge from S to $V - S$. $X \cup e$ is part of some MST.

Generic MST

```
$X \leftarrow \{\}$
while $|X| < |V| - 1$ do
    Choose $S$ s.t. $X$ does not connect $(S, V - S)$
    Add the lightest crossing edge to $X$
end while
```



Disjoint set operations

makeset(key) create a singleton set containing key

Disjoint set operations

makeset(key)	create a singleton set containing key
find(key)	find the set containing key

Disjoint set operations

makeset(key)	create a singleton set containing key
find(key)	find the set containing key
union(p,q)	merge the sets of p and q

$$\begin{tikzcd}
\boxed{\{a,b,p,r\},\{z,t,q\}} \arrow[d] \\
\boxed{\{a,b,p,r,z,t,q\}}
\end{tikzcd}$$

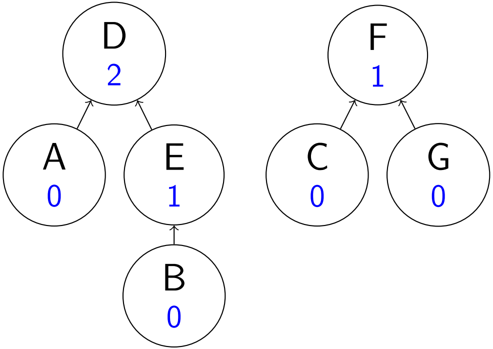
Kruskal's MST algorithm

```
$\forall u \in V$ makeset(u)
$X \leftarrow \{\}$
sort edges by weight
for $(u, v) \in E$ do
    if $\mathrm{find}(u) \neq find(v)$ then
        $X \leftarrow X \cup \{(u, v)\}$
        union(u,v)
    end if
end for
```

- what is S?

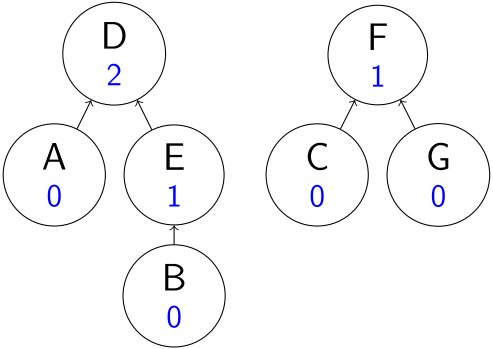
Contents

Forest representations



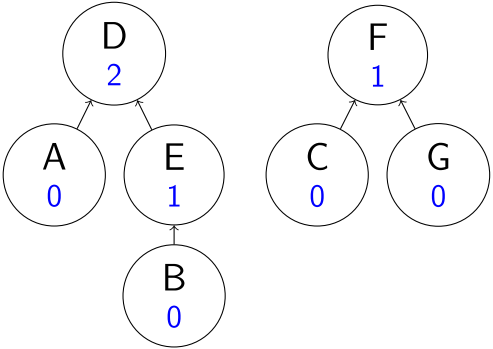
- each set is a tree.

Forest representations



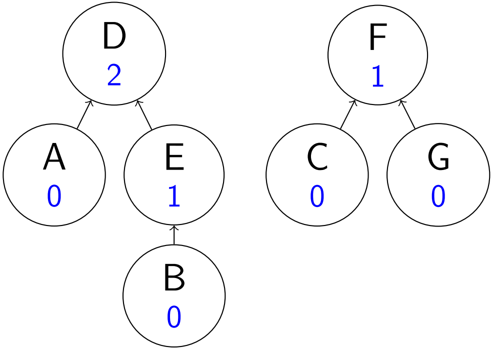
- each set is a tree.
- Each tree is represented by its root

Forest representations



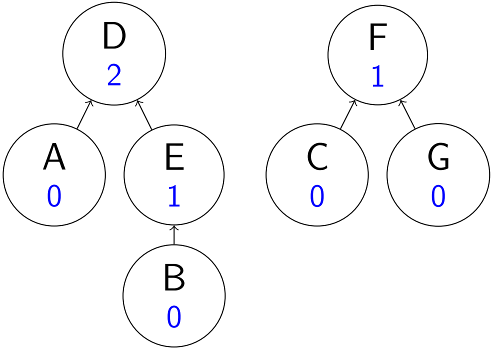
- each set is a tree.
- Each tree is represented by its root
- find(B) returns D

Forest representations



- each set is a tree.
- Each tree is represented by its root
- find(B) returns D
- makeset(x) just creates a single tree node.

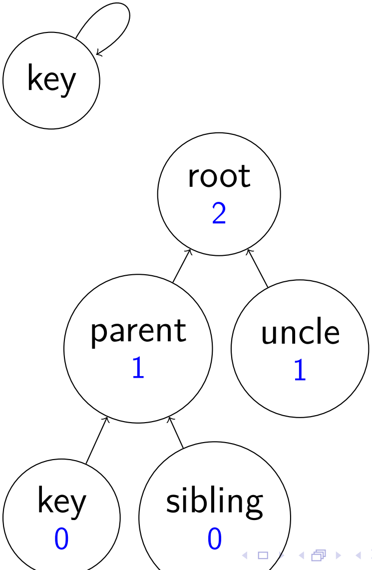
Forest representations



- each set is a tree.
- Each tree is represented by its root
- find(B) returns D
- makeset(x) just creates a single tree node.
- union points the root of one tree to another node.

Makeset and Find

```
function MAKESET(key)
    parent[key] $\leftarrow$ key
    rank[key]=0
end function
function FIND(key)
    while parent[key]$\neq$key do
        key $\leftarrow$ parent[key]
    end while
    return key
end function
```

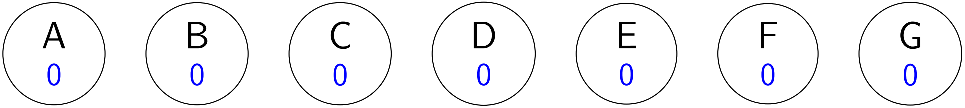


Union operation

```
function UNION($x, y$)
    $r_x \leftarrow \mathrm{find}(x)$
    $r_y \leftarrow \mathrm{find}(y)$
    if $r_y \neq r_x$ then
        if $\mathrm{rank}[r_x] > \mathrm{rank}[r_y]$ then
            $\mathrm{parent}[r_y] \leftarrow r_x$
        else
            $\mathrm{parent}[r_x] \leftarrow r_y$
            if $\mathrm{rank}[r_x] = \mathrm{rank}[r_y]$ then
                $\mathrm{rank}[r_y] + +$
            end if
        end if
    end if
```

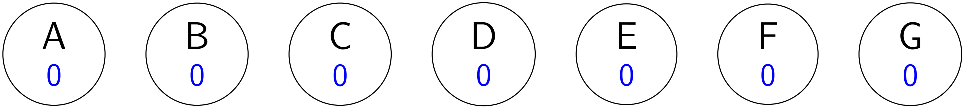
Union Find Example 1/3

- after makeset(A) ... makeset(G)

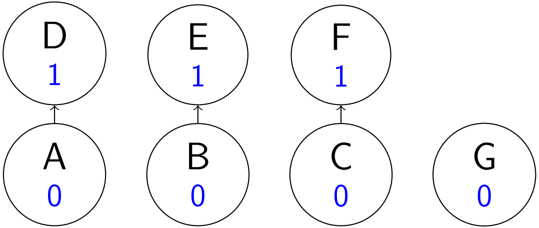


Union Find Example 1/3

- after makeset(A) ... makeset(G)

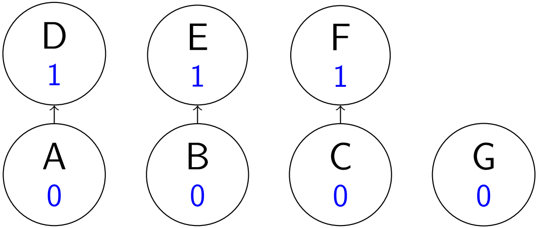


- after union(A,D), union(B,E), union(C,F)

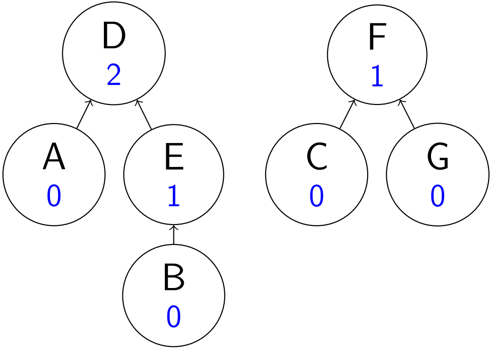


Union Find Example 2/3

after union(A,D), union(B,E), union(C,F)

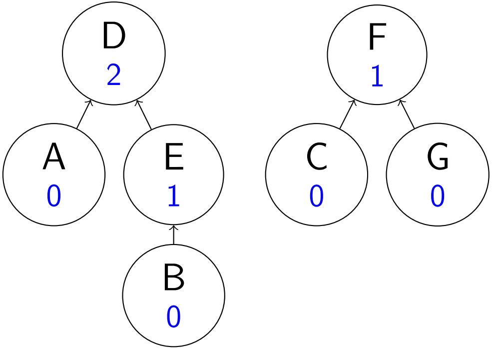


after union(C,G), union(E,A)

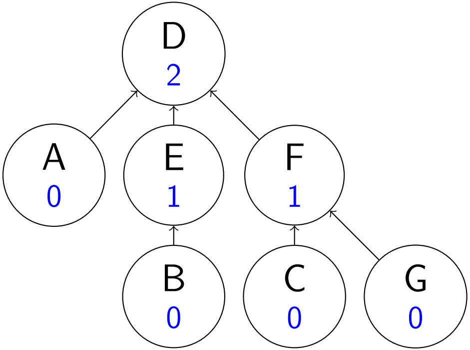


Union Find Example 3/3

after union(C,G), union(E,A)



after union(B,G)



Contents

Properties of Union Find trees

Property 1

For any x such that $\mathrm{parent}(x) \neq x$, $\mathrm{rank}(x) < \mathrm{rank}(\mathrm{parent}(x))$

$\therefore$ Trees are height at most $\log_2 n$

Properties of Union Find trees

Property 1

For any x such that $\mathrm{parent}(x) \neq x$, $\mathrm{rank}(x) < \mathrm{rank}(\mathrm{parent}(x))$

Property 2

Any node of rank k has at least 2^k nodes in its subtree.

$\therefore$ Trees are height at most $\log_2 n$

Properties of Union Find trees

Property 1

For any x such that $\mathrm{parent}(x) \neq x$, $\mathrm{rank}(x) < \mathrm{rank}(\mathrm{parent}(x))$

Property 2

Any node of rank k has at least 2^k nodes in its subtree.

Property 3

If there are n elements, there are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

$\therefore$ Trees are height at most $\log_2 n$

Proof of Property 1

Property 1

For any x such that $\mathrm{parent}(x) \neq x$, $\mathrm{rank}(x) < \mathrm{rank}(\mathrm{parent}(x))$

Proof of Property 1

Property 1

For any x such that $\mathrm{parent}(x) \neq x$, $\mathrm{rank}(x) < \mathrm{rank}(\mathrm{parent}(x))$

Proof.

- Initially every node has $\mathrm{parent}(x) = x$.
- Updating parent in union preserves this property.

```
**if** $\mathrm{rank}[r_x] > \mathrm{rank}[r_y]$ **then**
    $\mathrm{parent}[r_y] \leftarrow r_x$
**else**
    $\mathrm{parent}[r_x] \leftarrow r_y$
    **if** $\mathrm{rank}[r_x] = \mathrm{rank}[r_y]$ **then**
        $\mathrm{rank}[r_y]++$
    **end if**
**end if**
```

Proof of Property 2

Property 2

Any node of rank k has at least 2^k nodes in its subtree.

Proof of Property 2

Property 2

Any node of rank k has at least 2^k nodes in its subtree.

Proof.

- Base case: true for $k = 0$.
- Rank $k + 1$ is created only when joining two trees of rank k.

```
...
if $\mathrm{rank}[r_x] = \mathrm{rank}[r_y]$ then
    $\mathrm{rank}[r_y] + +$
end if
...
```

Proof of property 3

Property 3

If there are n elements, are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

Proof of property 3

Property 3

If there are n elements, are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

Proof.

- By Property 1 any element has at most one ancestor of rank k.
- Therefore the children of two rank k nodes are distinct.
- Apply property 2.

☐

Contents

Motivation

Using union-find in Kruskal's Algorithm

- For unbounded edge weights, the sorting costs $\Omega(|E| \log |E|) = \Omega(|E| \log |V|)$

Motivation

Using union-find in Kruskal's Algorithm

- For unbounded edge weights, the sorting costs $\Omega(|E| \log |E|) = \Omega(|E| \log |V|)$
    - Naive union-find is fast enough.

Motivation

Using union-find in Kruskal's Algorithm

- For unbounded edge weights, the sorting costs $\Omega(|E| \log |E|) = \Omega(|E| \log |V|)$
  - Naive union-find is fast enough.
- For small edge weights (e.g. weights bounded by $|E|$), sorting is no longer the bottleneck.

Amortized analysis

- It's not easy to make union faster than $O(\log n)$ in the worst case

Amortized analysis

- It's not easy to make union faster than $O(\log n)$ in the worst case
- What we can do easily is make the *average* cost of all union operations in one run of a program almost constant

Amortized analysis

- It's not easy to make union faster than $O(\log n)$ in the worst case
- What we can do easily is make the *average* cost of all union operations in one run of a program almost constant
- This kind of average cost analysis is called amortized analysis

Amortized analysis

- It's not easy to make union faster than $O(\log n)$ in the worst case
- What we can do easily is make the *average* cost of all union operations in one run of a program almost constant
- This kind of average cost analysis is called amortized analysis
- Like with randomized algorithms, the algorithms are simple, but the analysis is a bit subtle.

"Memoizing" the find routine

Old

```
function FIND(key)
    while parent[key]$\neq$key do
        key $\leftarrow$ parent[key]
    end while
    return key
end function
```

"Memoizing" the find routine

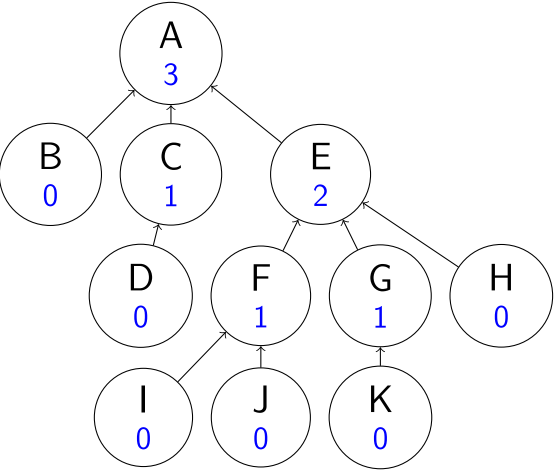
Old

```
function FIND(key)
    while parent[key]$\neq$key do
        key $\leftarrow$ parent[key]
    end while
    return key
end function
```

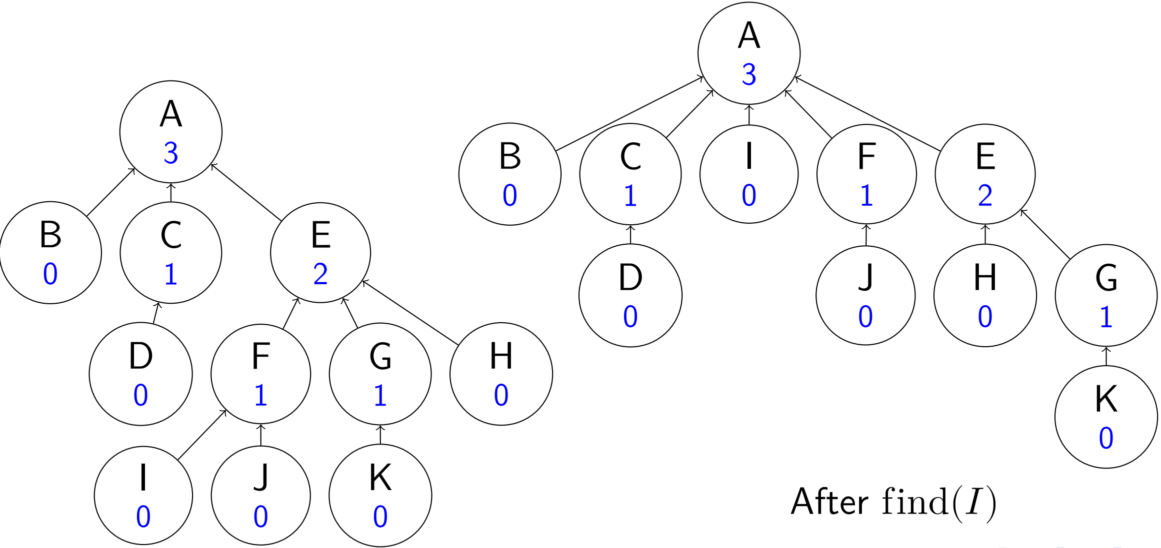
New

```
function FIND(key)
    if parent[key]$\neq$key then
        parent [key] $\leftarrow$ find(parent [key])
    end if
    return parent[key]
end function
```

Example of new find, find(I)

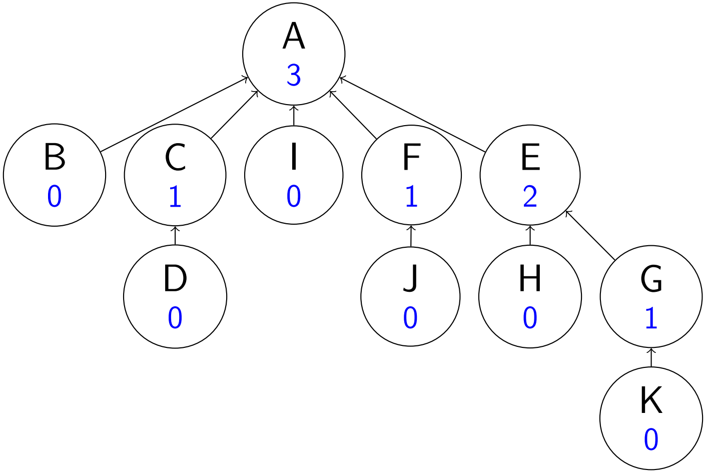


Example of new find, find(I)

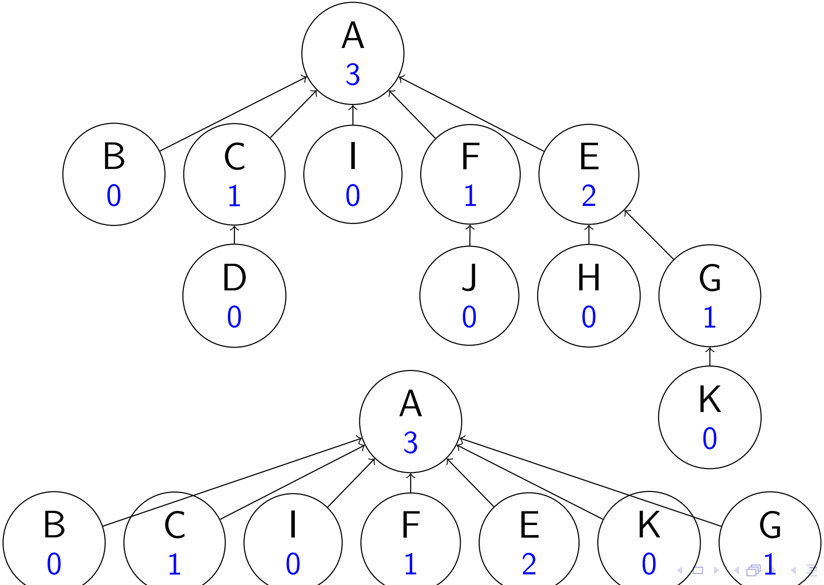


After find(I)

find(I), find(K)

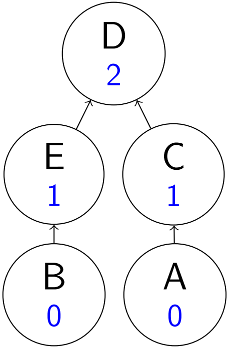


find(I), find(K)



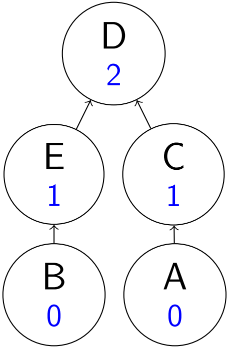
Strong Memoization

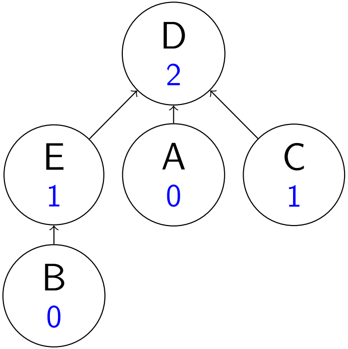
- not only only repeating the same query will be fast, but also any node on the path to the root.



Strong Memoization

- not only only repeating the same query will be fast, but also any node on the path to the root.





- After $\mathrm{find}(A)$

Rank ordering is maintained

Property 1

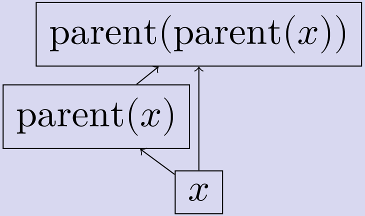
For any x such that $\mathrm{parent}(x) \neq x$, $\mathrm{rank}(x) < \mathrm{rank}(\mathrm{parent}(x))$

Rank ordering is maintained

Property 1

For any x such that $\mathrm{parent}(x) \neq x$, $\mathrm{rank}(x) < \mathrm{rank}(\mathrm{parent}(x))$

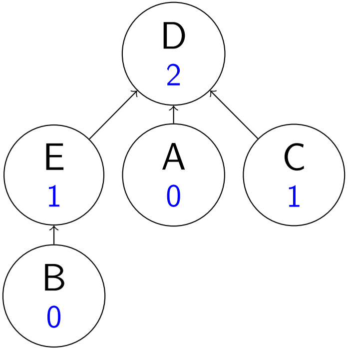
Shortcuts preserve order



Size of subtrees is preserved, but not subtrees.

Property 2

Any node of rank k has at least 2^k nodes in its subtree.



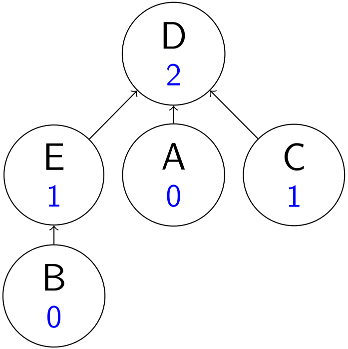
Size of subtrees is preserved, but not subtrees.

Property 2

Any node of rank k has at least 2^k nodes in its subtree.

Property 2'

Any root node of rank k has at least 2^k nodes in its subtree.



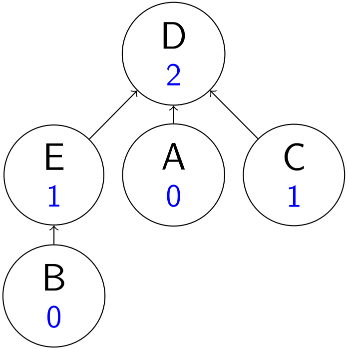
Size of subtrees is preserved, but not subtrees.

Property 2

Any node of rank k has at least 2^k nodes in its subtree.

Property 2'

Any root node of rank k has at least 2^k nodes in its subtree.



Proof of property 2'.

Same induction as before; note that path compression never moves nodes between trees ☐

Not too many nodes of rank k

Property 3

If there are n elements, there are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

Not too many nodes of rank k

Property 3

If there are n elements, there are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

proof

Not too many nodes of rank k

Property 3

If there are n elements, there are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

proof

- From property 1, every node has ≤ 1 rank k ancestor

Not too many nodes of rank k

Property 3

If there are n elements, there are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

proof

- From property 1, every node has ≤ 1 rank k ancestor
    - So descendents of a given rank k node are distinct.

Not too many nodes of rank k

Property 3

If there are n elements, there are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

proof

- From property 1, every node has ≤ 1 rank k ancestor
  - So descendents of a given rank k node are distinct.
- We charge each rank k node for all of its descendents at moment of becoming rank k.

Not too many nodes of rank k

Property 3

If there are n elements, there are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

proof

- From property 1, every node has ≤ 1 rank k ancestor
    - So descendents of a given rank k node are distinct.
- We charge each rank k node for all of its descendents at moment of becoming rank k.
- if path compression moves nodes from underneath a node, it moves to a node of higher rank

Not too many nodes of rank k

Property 3

If there are n elements, there are at most $\lfloor n/2^k \rfloor$ nodes of rank k.

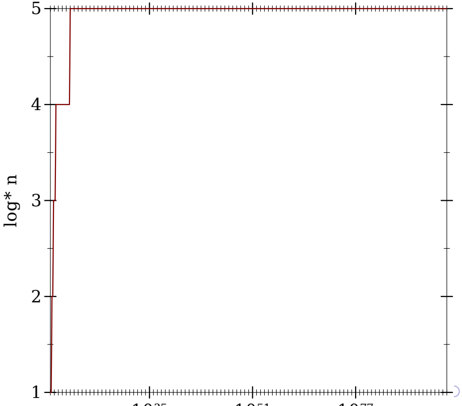
proof

- From property 1, every node has ≤ 1 rank k ancestor
    - So descendents of a given rank k node are distinct.
- We charge each rank k node for all of its descendents at moment of becoming rank k.
- if path compression moves nodes from underneath a node, it moves to a node of higher rank
- no node is ever charged towards more than one node of rank k

Contents

$\log^* n$

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



Amortization

- We will keep track of (some) operations by counting them locally at every node.

Amortization

- We will keep track of (some) operations by counting them locally at every node.
- After a node ceases to be a root node, its rank never changes.

Amortization

- We will keep track of (some) operations by counting them locally at every node.
- After a node ceases to be a root node, its rank never changes.
- In order to ”pay” for future operations, we give every node 2^k “dollars” if its max rank is in

  $$[k+1, \ldots 2^k]$$

  for some $k = 2^j$.

Amortization

- We will keep track of (some) operations by counting them locally at every node.
- After a node ceases to be a root node, its rank never changes.
- In order to ”pay” for future operations, we give every node 2^k “dollars” if its max rank is in

  $$[k+1, \dots 2^k]$$

  for some $k = 2^j$.
- We will count the total amount of money passed out

Amortization

- We will keep track of (some) operations by counting them locally at every node.
- After a node ceases to be a root node, its rank never changes.
- In order to ”pay” for future operations, we give every node 2^k “dollars” if its max rank is in

  $$[k+1, \dots 2^k]$$

  for some $k = 2^j$.
- We will count the total amount of money passed out
- And argue that no node runs out of money.

Rank Intervals

- We divide the numbers $[1, n]$ into $[k+1, 2^k]$

$$[1,1],[2,2],[3,4],[5,16],\dots,[k+1,2^k]$$

Rank Intervals

- We divide the numbers $[1, n]$ into $[k+1, 2^k]$

$$[1,1],[2,2],[3,4],[5,16],\dots,[k+1,2^k]$$

- The first p intervals cover

$$2^{2^{2^{\cdots 2}}}\Big\}p-1 \text{ times}$$

Rank Intervals

- We divide the numbers $[1, n]$ into $[k+1, 2^k]$

$$[1,1], [2,2], [3,4], [5,16], \ldots, [k+1, 2^k]$$

- The first p intervals cover

$$2^{2^{2^{\cdots 2}}} \Big\} p-1 \text{ times}$$

- It follows $\log^*(n) + 1$ intervals cover n, and $\log^*(n)$ intervals cover $\log n$.

Bounding disbursements 1/2

- Recall that each node in an interval ending in 2^k gets 2^k dollars.

Bounding disbursements 1/2

- Recall that each node in an interval ending in 2^k gets 2^k dollars.
- By property 3, the total number of nodes in such an interval is at most
$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \frac{n}{2^{k+3}} + \dots \frac{n}{2^{2^k}}$$

Bounding disbursements 1/2

- Recall that each node in an interval ending in 2^k gets 2^k dollars.
- By property 3, the total number of nodes in such an interval is at most

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \frac{n}{2^{k+3}} + \dots \frac{n}{2^{2^k}}$$

- We need to bound

$$\sum_{i=k+1}^{2^k} 2^{-i}$$

Bounding disbursements 1/2

- Recall that each node in an interval ending in 2^k gets 2^k dollars.
- By property 3, the total number of nodes in such an interval is at most
$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \frac{n}{2^{k+3}} + \dots \frac{n}{2^{2^k}}$$
- We need to bound
$$\sum_{i=k+1}^{2^k} 2^{-i} = \frac{1}{2^{k+1}} \sum_{i=0}^{2^k-k-1} 2^{-i}$$

Bounding disbursements 1/2

- Recall that each node in an interval ending in 2^k gets 2^k dollars.
- By property 3, the total number of nodes in such an interval is at most
$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \frac{n}{2^{k+3}} + \dots \frac{n}{2^{2^k}}$$
- We need to bound
$$\sum_{i=k+1}^{2^k} 2^{-i} = \frac{1}{2^{k+1}} \sum_{i=0}^{2^k-k-1} 2^{-i}$$
$$\leq \frac{1}{2^{k+1}} \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i$$

Bounding disbursements 2/2

- We need to bound

$$\sum_{i=k+1}^{2^k} 2^{-i} = \frac{1}{2^{k+1}} \sum_{i=0}^{2^k-k-1} 2^{-i}$$
$$\leq \frac{1}{2^{k+1}} \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i$$

- nodes in each interval get at most n dollars in total ($n \log^* n$ dollars over all intervals).

Bounding disbursements 2/2

- We need to bound

$$\begin{aligned}\sum_{i=k+1}^{2^k} 2^{-i} &= \frac{1}{2^{k+1}} \sum_{i=0}^{2^k-k-1} 2^{-i} \\ &\leq \frac{1}{2^{k+1}} \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \\ &= \frac{1}{2^{k+1}} \frac{1}{1-1/2} \qquad \text{Geometric Series}\end{aligned}$$

- nodes in each interval get at most n dollars in total ($n \log^* n$ dollars over all intervals).

Paying for find operations 1/2

```
function FIND(key)
    if parent[key]≠key then
        $\text{parent}[\text{key}] \leftarrow \text{find}(\text{parent}[\text{key}])$
    end if
    return parent[key]
end function
```

- Either $\text{rank}(\text{parent}[\text{key}])$ is in a later interval than $\text{rank}(\text{key})$ or not.

Paying for find operations 1/2

```
function FIND(key)
    if parent[key]≠key then
        $\text{parent}[\text{key}] \leftarrow \text{find}(\text{parent}[\text{key}])$
    end if
    return parent[key]
end function
```

- Either $\text{rank}(\text{parent}[\text{key}])$ is in a later interval than $\text{rank}(\text{key})$ or not.
- Increasing intervals can happen at most $\log^* n$ times.

Paying for find operations 1/2

```
function FIND(key)
    if parent[key]$\neq$key then
        $\text{parent}\,[\text{key}] \leftarrow \text{find}(\text{parent}\,[\text{key}])$
    end if
    return parent[key]
end function
```

- Either $\text{rank}(\text{parent}\,[\text{key}])$ is in a later interval than $\text{rank}(\text{key})$ or not.
- Increasing intervals can happen at most $\log^* n$ times.
- If in the same interval, we say key pays a dollar back.

Paying for find operations 2/2

- Either $\mathrm{rank}(\mathrm{parent}\,[\text{key}])$ is in a later interval than $\mathrm{rank}(\text{key})$ or not.

No node goes broke

Paying for find operations 2/2

- Either $\mathrm{rank}(\mathrm{parent}\,[\mathsf{key}])$ is in a later interval than $\mathrm{rank}(\mathsf{key})$ or not.
- Increasing intervals can happen at most $\log^* n$ times.

No node goes broke

Paying for find operations 2/2

- Either $\mathrm{rank}(\mathrm{parent}\,[\text{key}])$ is in a later interval than $\mathrm{rank}(\text{key})$ or not.
- Increasing intervals can happen at most $\log^* n$ times.
- If in the same interval, we say key pays a dollar back.

No node goes broke

Paying for find operations 2/2

- Either $\mathrm{rank}(\mathrm{parent}\,[\text{key}])$ is in a later interval than $\mathrm{rank}(\text{key})$ or not.
- Increasing intervals can happen at most $\log^* n$ times.
- If in the same interval, we say key pays a dollar back.

No node goes broke

- Each time x pays a dollar, it increases the rank of its parent.

Paying for find operations 2/2

- Either $\mathrm{rank}(\mathrm{parent}\,[\text{key}])$ is in a later interval than $\mathrm{rank}(\text{key})$ or not.
- Increasing intervals can happen at most $\log^* n$ times.
- If in the same interval, we say key pays a dollar back.

No node goes broke

- Each time x pays a dollar, it increases the rank of its parent.
- If $\mathrm{rank}(x) \in [k+1 \ldots 2^k]$, that can repeat less than 2^k times before its parent is in a higher interval.

Paying for find operations 2/2

- Either $\mathrm{rank}(\mathrm{parent}\,[\text{key}])$ is in a later interval than $\mathrm{rank}(\text{key})$ or not.
- Increasing intervals can happen at most $\log^* n$ times.
- If in the same interval, we say key pays a dollar back.

No node goes broke

- Each time x pays a dollar, it increases the rank of its parent.
- If $\mathrm{rank}(x) \in [k+1 \ldots 2^k]$, that can repeat less than 2^k times before its parent is in a higher interval.
- Once that happens, payments stop.

Summing up

- Total cost for n operations
    - $\leq n \log^* n$ total steps where parent is in next interval
    - $\leq n \log^* n$ total steps where parent is in same interval
- Amortized cost in $O(\log^* n)$ per operation.