Analysis of Algorithm

Disjoint Set Representation

Andres Mendez-Vazquez

March 11, 2018

Outline

Outline

Disjoint Set Representation

Problem

1. Items are drawn from the finite universe $U = 1, 2, ..., n$ for some fixed n.
2. We want to maintain a partition of U as a collection of disjoint sets.
3. In addition, we want to uniquely name each set by one of its items called its representative item.

Disjoint Set Representation

Problem

1. Items are drawn from the finite universe $U = 1, 2, ..., n$ for some fixed n.
2. We want to maintain a partition of U as a collection of disjoint sets.
3. In addition, we want to uniquely name each set by one of its items called its representative item.

These disjoint sets are maintained under the following operations

1. MakeSet(x)
2. Union(A,B)
3. Find(x)

Disjoint Set Representation

Problem

1. Items are drawn from the finite universe $U = 1, 2, ..., n$ for some fixed n.
2. We want to maintain a partition of U as a collection of disjoint sets.
3. In addition, we want to uniquely name each set by one of its items called its representative item.

These disjoint sets are maintained under the following operations

1. MakeSet(x)
2. Union(A,B)
3. Find(x)

Disjoint Set Representation

Problem

1. Items are drawn from the finite universe $U = 1, 2, ..., n$ for some fixed n.
2. We want to maintain a partition of U as a collection of disjoint sets.
3. In addition, we want to uniquely name each set by one of its items called its representative item.

These disjoint sets are maintained under the following operations

1. MakeSet(x)
2. Union(A,B)
3. Find(x)

Disjoint Set Representation

Problem

1. Items are drawn from the finite universe $U = 1, 2, ..., n$ for some fixed n.
2. We want to maintain a partition of U as a collection of disjoint sets.
3. In addition, we want to uniquely name each set by one of its items called its representative item.

These disjoint sets are maintained under the following operations

1. MakeSet(x)
2. Union(A,B)
3. Find(x)

Disjoint Set Representation

Problem

1. Items are drawn from the finite universe $U = 1, 2, ..., n$ for some fixed n.
2. We want to maintain a partition of U as a collection of disjoint sets.
3. In addition, we want to uniquely name each set by one of its items called its representative item.

These disjoint sets are maintained under the following operations

1. MakeSet(x)
2. Union(A,B)
3. Find(x)

Outline

Operations

MakeSet(x)

- Given $x \in U$ currently not belonging to any set in the collection, create a new singleton set $\{x\}$ and name it x.
    - This is usually done at start, once per item, to create the initial trivial partition.

Union(A,B)

- It changes the current partition by replacing its sets A and B with $A \cup B$. Name the set A or B.
    - The operation may choose either one of the two representatives as the new representatives.

Find(x)

It returns the name of the set that currently contains item x.

Operations

MakeSet(x)

- Given $x \in U$ currently not belonging to any set in the collection, create a new singleton set $\{x\}$ and name it x.
  - This is usually done at start, once per item, to create the initial trivial partition.

Union(A,B)

- It changes the current partition by replacing its sets A and B with $A \cup B$. Name the set A or B.
  - The operation may choose either one of the two representatives as the new representatives.

Find(x)

It returns the name of the set that currently contains item x.

Operations

MakeSet(x)

- Given $x \in U$ currently not belonging to any set in the collection, create a new singleton set $\{x\}$ and name it x.
    - This is usually done at start, once per item, to create the initial trivial partition.

Union(A,B)

- It changes the current partition by replacing its sets A and B with $A \cup B$. Name the set A or B.
    - The operation may choose either one of the two representatives as the new representatives.

Find(x)

It returns the name of the set that currently contains item x.

Example

for $x = 1$ to 9 do MakeSet(x)

Then, you do a Union$(1, 2)$

Now, Union$(3, 4)$; Union$(5, 8)$; Union$(6, 9)$

Example

for $x = 1$ to 9 do MakeSet(x)

Then, you do a Union$(1, 2)$

Now, Union$(3, 4)$; Union$(5, 8)$; Union$(6, 9)$

Example

for $x = 1$ to 9 do MakeSet(x)

Then, you do a Union$(1, 2)$

Now, Union$(3, 4)$; Union$(5, 8)$; Union$(6, 9)$

Example

Now, Union$(1, 5)$; Union$(7, 4)$

Then, if we do the following operations

- Find(1) returns 5
- Find(9) returns 9

Finally, Union$(5, 9)$

Then Find(9) returns 5

Example

Now, Union$(1, 5)$; Union$(7, 4)$

Then, if we do the following operations

- Find(1) returns 5
- Find(9) returns 9

Finally, Union$(5, 9)$

Then Find(9) returns 5

Example

Now, Union$(1, 5)$; Union$(7, 4)$

Then, if we do the following operations

- Find(1) returns 5
- Find(9) returns 9

Finally, Union$(5, 9)$

Then Find(9) returns 5

Outline

Union-Find Problem

Problem

- $S =$ be a sequence of $m = |S|$ MakeSet, Union and Find operations (intermixed in arbitrary order):
    - n of which are MakeSet.
    - At most $n - 1$ are Union.
    - The rest are Finds.
- Cost(S) = total computational time to execute sequence s.
- Goal: Find an implementation that, for every m and n, minimizes the amortized cost per operation:

$$\frac{Cost\,(S)}{|S|} \tag{1}$$

for any arbitrary sequence S.

Union-Find Problem

Problem

- $S =$ be a sequence of $m = |S|$ MakeSet, Union and Find operations (intermixed in arbitrary order):
  - n of which are MakeSet.
  - At most $n - 1$ are Union.
  - The rest are Finds.
- Cost(S) $=$ total computational time to execute sequence s.
- Goal: Find an implementation that, for every m and n, minimizes the amortized cost per operation:

$$\frac{Cost\,(S)}{|S|} \tag{1}$$

for any arbitrary sequence S.

Union-Find Problem

Problem

- $S =$ be a sequence of $m = |S|$ MakeSet, Union and Find operations (intermixed in arbitrary order):
  - n of which are MakeSet.
  - At most $n - 1$ are Union.
  - The rest are Finds.
- Cost(S) = total computational time to execute sequence s.
- Goal: Find an implementation that, for every m and n, minimizes the amortized cost per operation:

$$\frac{Cost\left(S\right)}{|S|} \tag{1}$$

for any arbitrary sequence S.

Union-Find Problem

Problem

- $S =$ be a sequence of $m = |S|$ MakeSet, Union and Find operations (intermixed in arbitrary order):
  - n of which are MakeSet.
  - At most $n-1$ are Union.
  - The rest are Finds.
- Cost(S) = total computational time to execute sequence s.
- Goal: Find an implementation that, for every m and n, minimizes the amortized cost per operation:

$$\frac{Cost\,(S)}{|S|} \tag{1}$$

for any arbitrary sequence S.

Union-Find Problem

Problem

- $S =$ be a sequence of $m = |S|$ MakeSet, Union and Find operations (intermixed in arbitrary order):
  - n of which are MakeSet.
  - At most $n-1$ are Union.
  - The rest are Finds.
- Cost(S) $=$ total computational time to execute sequence s.
- Goal: Find an implementation that, for every m and n, minimizes the amortized cost per operation:

$$\frac{Cost\,(S)}{|S|} \tag{1}$$

for any arbitrary sequence S.

Union-Find Problem

Problem

- $S =$ be a sequence of $m = |S|$ MakeSet, Union and Find operations (intermixed in arbitrary order):
  - n of which are MakeSet.
  - At most $n-1$ are Union.
  - The rest are Finds.
- Cost$(S) =$ total computational time to execute sequence s.
- Goal: Find an implementation that, for every m and n, minimizes the amortized cost per operation:

$$\frac{Cost\,(S)}{|S|} \tag{1}$$

for any arbitrary sequence S.

Outline

Applications

Examples

1. Maintaining partitions and equivalence classes.
2. Graph connectivity under edge insertion.
3. Minimum spanning trees (e.g. Kruskal's algorithm).
4. Random maze construction.

Applications

Examples

1. Maintaining partitions and equivalence classes.
2. Graph connectivity under edge insertion.
3. Minimum spanning trees (e.g. Kruskal's algorithm).
4. Random maze construction.

Applications

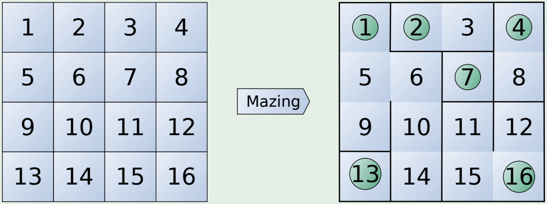
Examples

1. Maintaining partitions and equivalence classes.
2. Graph connectivity under edge insertion.
3. Minimum spanning trees (e.g. Kruskal's algorithm).
4. Random maze construction.

Applications

Examples

1. Maintaining partitions and equivalence classes.
2. Graph connectivity under edge insertion.
3. Minimum spanning trees (e.g. Kruskal's algorithm).
4. Random maze construction.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

①	②	3	④
5	6	⑦	8
9	10	11	12
⑬	14	15	⑯

Outline

Circular lists

We use the following structures

Data structure: Two arrays $Set[1..n]$ and $next[1..n]$.

- $Set[x]$ returns the name of the set that contains item x.
- A is a set **if and only if** $Set[A] = A$
- $next[x]$ returns the next item on the list of the set that contains item x.

Circular lists

We use the following structures

Data structure: Two arrays $Set[1..n]$ and $next[1..n]$.

- $Set[x]$ returns the name of the set that contains item x.
- A is a set **if and only if** $Set[A] = A$
- $next[x]$ returns the next item on the list of the set that contains item x.

Circular lists

We use the following structures

Data structure: Two arrays $Set[1..n]$ and $next[1..n]$.

- $Set[x]$ returns the name of the set that contains item x.
- A is a set **if and only if** $Set[A] = A$
- $next[x]$ returns the next item on the list of the set that contains item x.

Circular lists

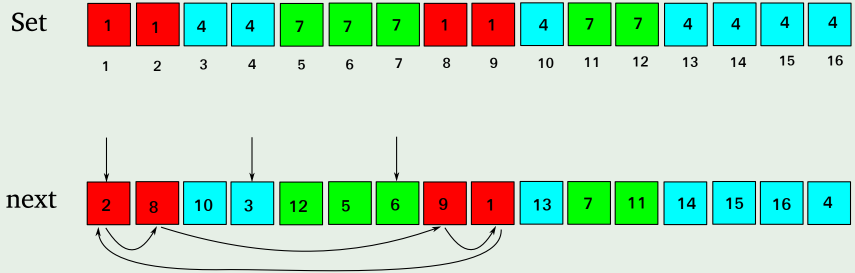
We use the following structures

Data structure: Two arrays $Set[1..n]$ and $next[1..n]$.

- $Set[x]$ returns the name of the set that contains item x.
- A is a set **if and only if** $Set[A] = A$
- $next[x]$ returns the next item on the list of the set that contains item x.

Circular lists

Example: $n = 16,$
Partition: $\{\{1, 2, 8, 9\}, \ \{4, 3, 10, 13, 14, 15, 16\}, \ \{7, 6, 5, 11, 1$



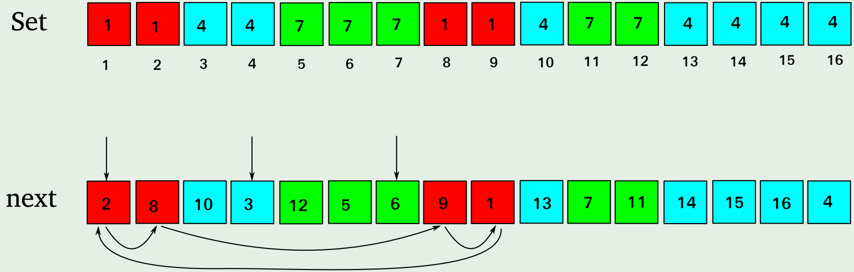
Set	1	1	4	4	7	7	7	1	1	4	7	7	4	4	4	4
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
next	2	8	10	3	12	5	6	9	1	13	7	11	14	15	16	4

Set Position 1

Circular lists

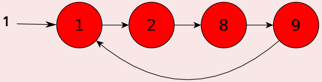
Example: $n = 16$,

Partition: $\{\{1, 2, 8, 9\}, \{4, 3, 10, 13, 14, 15, 16\}, \{7, 6, 5, 11, 1$



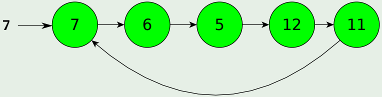
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Set	1	1	4	4	7	7	7	1	1	4	7	7	4	4	4	4
next	2	8	10	3	12	5	6	9	1	13	7	11	14	15	16	4

Set Position 1



Circular lists

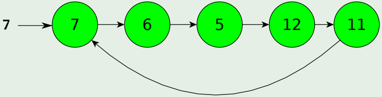
Set Position 7



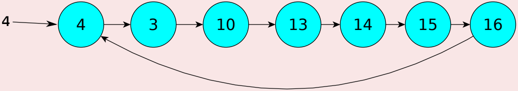
Set Position 4

Circular lists

Set Position 7



Set Position 4



Operations and Cost

Make(x)

1. $\text{Set}[x] = x$
2. $\text{next}[x] = x$

Complexity

- $O(1)$ Time

Find(x)

1. return $\text{Set}[x]$

Complexity

- $O(1)$ Time

Operations and Cost

Make(x)

1. $\text{Set}[x] = x$
2. $\text{next}[x] = x$

Complexity

- $O(1)$ Time

Find(x)

1. return $\text{Set}[x]$

Complexity

- $O(1)$ Time

Operations and Cost

Make(x)

1. $\text{Set}[x] = x$
2. $\text{next}[x] = x$

Complexity

- $O(1)$ Time

Find(x)

1. return $\text{Set}[x]$

Complexity

- $O(1)$ Time

Operations and Cost

Make(x)

1. $\text{Set}[x] = x$
2. $\text{next}[x] = x$

Complexity

- $O(1)$ Time

Find(x)

1. return $\text{Set}[x]$

Complexity

- $O(1)$ Time

Operations and Cost

For the union

We are assuming Set$[A] = A \neq$Set$[B] = B$

Union1(A, B)

1. Set$[B] = A$
2. $x =$next$[B]$

Operations and Cost

For the union

We are assuming Set$[A] = A \neq$Set$[B] = B$

Union1(A, B)

```
Set$[B] = A$
$x =$next$[B]$
while $(x \neq B)$
    Set$[x] = A$ /* Rename Set B to A*/
    $x =$next$[x]$
$x =$next$[B]$ /* Splice list A and B */
next$[B]$ =next$[A]$
next$[A] = x$
```

Operations and Cost

For the union

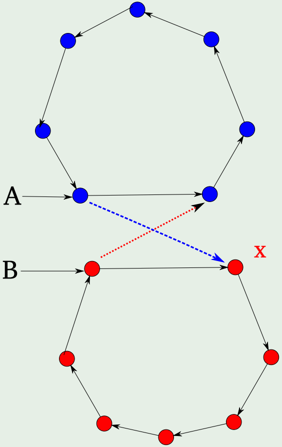
We are assuming $\text{Set}[A] = A \neq \text{Set}[B] = B$

Union1(A, B)

```
Set$[B] = A$
$x$ =next$[B]$
while $(x \neq B)$
        Set$[x] = A$ /* Rename Set B to A*/
        $x$ =next$[x]$
$x$ =next$[B]$ /* Splice list A and B */
next$[B]$ =next$[A]$
next$[A] = x$
```

Operations an Cost

Thus, we have in the Splice part



We have a Problem

Complexity

$O(|B|)$ Time

Not only that, if we have the following sequence of operations

```
for $x = 1$ to $n$
    MakeSet($x$)
for $x = 1$ to $n - 1$
    Union1($x + 1, x$)
```

We have a Problem

Complexity

$O\left(|B|\right)$ Time

Not only that, if we have the following sequence of operations

```
for $x = 1$ to $n$
    MakeSet$(x)$
for $x = 1$ to $n - 1$
    Union1$(x + 1, x)$
```

Thus

Thus, we have the following number of aggregated steps

$$
\begin{aligned}
n + \sum_{i=1}^{n-1} i &= n + \frac{n\left(n-1\right)}{2} \\
&= n + \frac{n^2 - n}{2} \\
&= \frac{n^2}{2} + \frac{n}{2} \\
&= \Theta\left(n^2\right)
\end{aligned}
$$

Thus

Thus, we have the following number of aggregated steps

$$
\begin{aligned}
n + \sum_{i=1}^{n-1} i &= n + \frac{n(n-1)}{2} \\
&= n + \frac{n^2 - n}{2} \\
&= \frac{n^2}{2} + \frac{n}{2} \\
&= \Theta\left(n^2\right)
\end{aligned}
$$

Thus

Thus, we have the following number of aggregated steps

$$\begin{aligned}
n + \sum_{i=1}^{n-1} i &= n + \frac{n\left(n-1\right)}{2} \\
&= n + \frac{n^2 - n}{2} \\
&= \frac{n^2}{2} + \frac{n}{2} \\
&= \Theta\left(n^2\right)
\end{aligned}$$

Thus

Thus, we have the following number of aggregated steps

$$
\begin{aligned}
n + \sum_{i=1}^{n-1} i &= n + \frac{n(n-1)}{2} \\
&= n + \frac{n^2 - n}{2} \\
&= \frac{n^2}{2} + \frac{n}{2} \\
&= \Theta\left(n^2\right)
\end{aligned}
$$

Thus

Thus, we have the following number of aggregated steps

$$
\begin{aligned}
n+\sum_{i=1}^{n-1} i &= n+\frac{n\left(n-1\right)}{2}\\
&= n+\frac{n^2-n}{2}\\
&= \frac{n^2}{2}+\frac{n}{2}\\
&= \Theta\left(n^2\right)
\end{aligned}
$$

Aggregate Time

Thus, the aggregate time is as follow

Aggregate Time $= \Theta\left(n^2\right)$

Therefore

Amortized Time per operation $= \Theta\left(n\right)$

Aggregate Time

Thus, the aggregate time is as follow

Aggregate Time $= \Theta\left(n^2\right)$

Therefore

Amortized Time per operation $= \Theta\left(n\right)$

This is not exactly good

Thus, we need to have something better

We will try now the Weighted-Union Heuristic!!!

Outline

Implementation 2: Weighted-Union Heuristic Lists

We extend the previous data structure

Data structure: Three arrays $Set[1..n]$, $next[1..n]$, $size[1..n]$.

- $size[A]$ returns the number of items in set A if $A == Set[A]$ (Otherwise, we do not care).

Operations

MakeSet(x)

1. Set$[x] = x$
2. next$[x] = x$
3. size$[x] = 1$

Complexity

$O(1)$ time

Find(x)

1. return Set$[x]$

Complexity

$O(1)$ time

Operations

MakeSet(x)

1. Set$[x] = x$
2. next$[x] = x$
3. size$[x] = 1$

Complexity

$O(1)$ time

Find(x)

1. return Set$[x]$

Complexity

$O(1)$ time

Operations

MakeSet(x)

1. Set$[x] = x$
2. next$[x] = x$
3. size$[x] = 1$

Complexity

$O(1)$ time

Find(x)

1. return Set$[x]$

Complexity

$O(1)$ time

Operations

MakeSet(x)

1. Set$[x] = x$
2. next$[x] = x$
3. size$[x] = 1$

Complexity

$O(1)$ time

Find(x)

1. return Set$[x]$

Complexity

$O(1)$ time

Operations

Union2(A, B)

```
if size[$set\ [A]$] > size[$set\ [B]$]
    size[$set\ [A]$] = size[$set\ [A]$] + size[$set\ [B]$]
    Union1$(A, B)$
else
    size[$set\ [B]$] = size[$set\ [A]$] + size[$set\ [B]$]
    Union1$(B, A)$
```

Note: Weight Balanced Union: Merge smaller set into large set

Complexity

$O\left(\min\left\{|A|, |B|\right\}\right)$ time.

Operations

Union2(A, B)

```
if size[$set[A]$] > size[$set[B]$]
    size[$set[A]$] = size[$set[A]$] + size[$set[B]$]
    Union1$(A, B)$
else
    size[$set[B]$] = size[$set[A]$] + size[$set[B]$]
    Union1$(B, A)$
```

Note: Weight Balanced Union: Merge smaller set into large set

Complexity

$O\left(\min\left\{|A|, |B|\right\}\right)$ time.

What about the operations eliciting the worst behavior

Remember

```
for $x = 1$ to $n$
        MakeSet$(x)$
for $x = 1$ to $n - 1$
        Union2$(x + 1, x)$
```

We have then

$$
\begin{aligned}
n + \sum_{i=1}^{n-1} 1 &= n + n - 1\\
&= 2n - 1\\
&= \Theta(n)
\end{aligned}
$$

IMPORTANT: This is not the worst sequence!!!

What about the operations eliciting the worst behavior

Remember

```
for $x = 1$ to $n$
    MakeSet($x$)
for $x = 1$ to $n - 1$
    Union2($x + 1, x$)
```

We have then

$$
\begin{aligned}
n + \sum_{i=1}^{n-1} 1 &= n + n - 1 \\
&= 2n - 1 \\
&= \Theta(n)
\end{aligned}
$$

IMPORTANT: This is not the worst sequence!!!

For this, notice the following worst sequence

Worst Sequence s

MakeSet(x), for $x = 1, .., n$. Then do $n - 1$ Unions in round-robin manner.

- Within each round, the sets have roughly equal size.
    - Starting round: Each round has size 1.
    - Next round: Each round has size 2.
    - Next: ... size 4.
    - ...

For this, notice the following worst sequence

Worst Sequence s

MakeSet(x), for $x = 1, .., n$. Then do $n - 1$ Unions in round-robin manner.

- Within each round, the sets have roughly equal size.
  - Starting round: Each round has size 1.
  - Next round: Each round has size 2.
  - Next: ... size 4.
  - ...

We claim the following

- Aggregate time $= \Theta(n \log n)$
- Amortized time per operation $= \Theta(\log n)$

For this, notice the following worst sequence

Worst Sequence s

MakeSet(x), for $x = 1, .., n$. Then do $n-1$ Unions in round-robin manner.

- Within each round, the sets have roughly equal size.
  - Starting round: Each round has size 1.
  - Next round: Each round has size 2.
  - Next: ... size 4.
  - ...

We claim the following

- Aggregate time $= \Theta(n \log n)$
- Amortized time per operation $= \Theta(\log n)$

For this, notice the following worst sequence

Worst Sequence s

MakeSet(x), for $x = 1, .., n$. Then do $n - 1$ Unions in round-robin manner.

- Within each round, the sets have roughly equal size.
  - Starting round: Each round has size 1.
  - Next round: Each round has size 2.
  - Next: ... size 4.
  - ...

We claim the following

- Aggregate time $= \Theta(n \log n)$
- Amortized time per operation $= \Theta(\log n)$

For this, notice the following worst sequence

Worst Sequence s

MakeSet(x), for $x = 1, .., n$. Then do $n - 1$ Unions in round-robin manner.

- Within each round, the sets have roughly equal size.
    - Starting round: Each round has size 1.
    - Next round: Each round has size 2.
    - Next: ... size 4.
    - ...

We claim the following

- Aggregate time $= \Theta(n \log n)$
- Amortized time per operation $= \Theta(\log n)$

For this, notice the following worst sequence

Worst Sequence s

MakeSet(x), for $x = 1, .., n$. Then do $n - 1$ Unions in round-robin manner.

- Within each round, the sets have roughly equal size.
  - Starting round: Each round has size 1.
  - Next round: Each round has size 2.
  - Next: ... size 4.
  - ...

We claim the following

- Aggregate time $= \Theta(n \log n)$
- Amortized time per operation $= \Theta(\log n)$

For this, notice the following worst sequence

Worst Sequence s

MakeSet(x), for $x = 1, .., n$. Then do $n - 1$ Unions in round-robin manner.

- Within each round, the sets have roughly equal size.
    - Starting round: Each round has size 1.
    - Next round: Each round has size 2.
    - Next: ... size 4.
    - ...

We claim the following

- Aggregate time $= \Theta(n \log n)$
- Amortized time per operation $= \Theta(\log n)$

For this, notice the following worst sequence

Worst Sequence s

MakeSet(x), for $x = 1, .., n$. Then do $n - 1$ Unions in round-robin manner.

- Within each round, the sets have roughly equal size.
  - Starting round: Each round has size 1.
  - Next round: Each round has size 2.
  - Next: ... size 4.
  - ...

We claim the following

- Aggregate time $= \Theta(n \log n)$
- Amortized time per operation $= \Theta(\log n)$

For this, notice the following worst sequence

Example $n = 16$

- Round 0: {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16}
- Round 1: {1, 2} {3, 4} {5, 6} {7, 8} {9, 10} {11, 12} {13, 14} {15, 16}
- Round 2: {1, 2, 3, 4} {5, 6, 7, 8} {9, 10, 11, 12} {13, 14, 15, 16}
- Round 3: {1, 2, 3, 4, 5, 6, 7, 8} {9, 10, 11, 12, 13, 14, 15, 16}
- Round 4: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

For this, notice the following worst sequence

Example $n = 16$

- Round 0: {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16}
- Round 1: {1, 2} {3, 4} {5, 6} {7, 8} {9, 10} {11, 12} {13, 14} {15, 16}
- Round 2: {1, 2, 3, 4} {5, 6, 7, 8} {9, 10, 11, 12} {13, 14, 15, 16}
- Round 3: {1, 2, 3, 4, 5, 6, 7, 8} {9, 10, 11, 12, 13, 14, 15, 16}
- Round 4: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

For this, notice the following worst sequence

Example $n = 16$

- Round 0: {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16}
- Round 1: {1, 2} {3, 4} {5, 6} {7, 8} {9, 10} {11, 12} {13, 14} {15, 16}
- Round 2: {1, 2, 3, 4} {5, 6, 7, 8} {9, 10, 11, 12} {13, 14, 15, 16}
- Round 3: {1, 2, 3, 4, 5, 6, 7, 8} {9, 10, 11, 12, 13, 14, 15, 16}
- Round 4: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

For this, notice the following worst sequence

Example $n = 16$

- Round 0: {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16}
- Round 1: {1, 2} {3, 4} {5, 6} {7, 8} {9, 10} {11, 12} {13, 14} {15, 16}
- Round 2: {1, 2, 3, 4} {5, 6, 7, 8} {9, 10, 11, 12} {13, 14, 15, 16}
- Round 3: {1, 2, 3, 4, 5, 6, 7, 8} {9, 10, 11, 12, 13, 14, 15, 16}
- Round 4: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

For this, notice the following worst sequence

Example $n = 16$

- Round 0: {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16}
- Round 1: {1, 2} {3, 4} {5, 6} {7, 8} {9, 10} {11, 12} {13, 14} {15, 16}
- Round 2: {1, 2, 3, 4} {5, 6, 7, 8} {9, 10, 11, 12} {13, 14, 15, 16}
- Round 3: {1, 2, 3, 4, 5, 6, 7, 8} {9, 10, 11, 12, 13, 14, 15, 16}
- Round 4: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

Now

Given the previous worst case

What is the complexity of this implementation?

Now, the Amortized Costs of this implementation

Claim 1: Amortized time per operation is $O(\log n)$

For this, we have the following theorem!!!

Theorem

Theorem 21.1

Using the linked-list representation of disjoint sets and the weighted-Union heuristic, a sequence of m MakeSet, Union, and FindSet operations, n of which are MakeSet operations, takes $O(m + n \log n)$ time.

Proof

Because each Union operation unites two disjoint sets

We perform at most $n-1$ Union operations over all.

Proof

Because each Union operation unites two disjoint sets

We perform at most $n-1$ Union operations over all.

We now bound the total time taken by these Union operations

- We start by determining, for each object,
  - an upper bound on the number of times the **object's pointer back to its set object** is updated.

Proof

Because each Union operation unites two disjoint sets

We perform at most $n-1$ Union operations over all.

We now bound the total time taken by these Union operations

- We start by determining, for each object,
  - an upper bound on the number of times the **object's pointer back to its set object** is updated.

Proof

Consider a particular object x.

- We know that each time x's pointer was updated, x **must have started in the smaller set.**

The first time x's pointer was updated

- The resulting set must have had at least 2 members.

Similarly

- Similarly, the next time x's pointer was updated, the resulting set must have had at least 4 members.

Proof

Consider a particular object x.

- We know that each time x's pointer was updated, x **must have started in the smaller set.**

The first time x's pointer was updated

- The resulting set must have had at least 2 members.

Similarly

- Similarly, the next time x's pointer was updated, the resulting set must have had at least 4 members.

Proof

Consider a particular object x.

- We know that each time x's pointer was updated, x **must have started in the smaller set.**

The first time x's pointer was updated

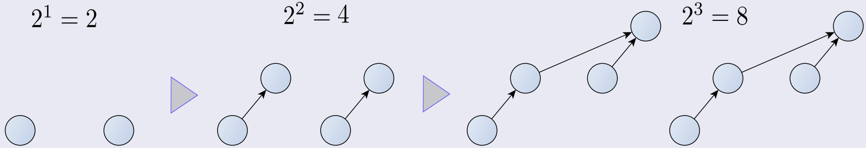
- The resulting set must have had at least 2 members.

Similarly

- Similarly, the next time x's pointer was updated, the resulting set must have had at least 4 members.

Proof

Example



$$n = 2^{\log n}$$

Proof

Continuing on

We observe that for any $k \leq n$, after x's pointer has been updated $\lceil \log n \rceil$ times!!!

- The resulting set must have at least k members.

Proof

Continuing on

We observe that for any $k \leq n$, after x's pointer has been updated $\lceil \log n \rceil$ times!!!

- The resulting set must have at least k members.

Thus

Since the largest set has at most n members, each object's pointer is updated at most $\lceil \log n \rceil$ times over all the Union operations.

Proof

Continuing on

We observe that for any $k \leq n$, after x's pointer has been updated $\lceil \log n \rceil$ times!!!

- The resulting set must have at least k members.

Thus

Since the largest set has at most n members, each object's pointer is updated at most $\lceil \log n \rceil$ times over all the Union operations.

Then

The total time spent updating object pointers over all Union operations is $O(n \log n)$.

Proof

Continuing on

We observe that for any $k \leq n$, after x's pointer has been updated $\lceil \log n \rceil$ times!!!

- The resulting set must have at least k members.

Thus

Since the largest set has at most n members, each object's pointer is updated at most $\lceil \log n \rceil$ times over all the Union operations.

Then

The total time spent updating object pointers over all Union operations is $O\left(n \log n\right)$.

Proof

We must also account for updating the tail pointers and the list lengths

It takes only $O(1)$ time per Union operation

Therefore

The total time spent in all Union operations is thus $O(n \log n)$.

The time for the entire sequence of m operations follows easily

Each MakeSet and FindSet operation takes $O(1)$ time, and there are $O(m)$ of them.

Proof

We must also account for updating the **tail pointers** and the list lengths

It takes only $O(1)$ time per Union operation

Therefore

The total time spent in all Union operations is thus $O(n \log n)$.

The time for the entire sequence of m operations follows easily

Each MakeSet and FindSet operation takes $O(1)$ time, and there are $O(m)$ of them.

Proof

We must also account for updating the tail pointers and the list lengths

It takes only $O\left(1\right)$ time per Union operation

Therefore

The total time spent in all Union operations is thus $O\left(n \log n\right)$.

The time for the entire sequence of m operations follows easily

Each MakeSet and FindSet operation takes $O\left(1\right)$ time, and there are $O\left(m\right)$ of them.

Proof

Therefore

The total time for the entire sequence is thus $O(m + n \log n)$.

Amortized Cost: Aggregate Analysis

Aggregate cost $O(m + n\log n)$. Amortized cost per operation $O(\log n)$.

$$\frac{O(m + n\log n)}{m} = O\left(1 + \log n\right) = O\left(\log n\right) \tag{2}$$

There are other ways of analyzing the amortized cost

It is possible to use

1. Accounting Method.
2. Potential Method.

Amortized Costs: Accounting Method

Accounting method

- MakeSet(x): Charge $(1 + \log n)$. 1 to do the operation, $\log n$ stored as credit with item x.
- Find(x): Charge 1, and use it to do the operation.
- Union(A, B): Charge 0 and use 1 stored credit from each item in the smaller set to move it.

Amortized Costs: Accounting Method

Accounting method

- MakeSet(x): Charge $(1+\log n)$. 1 to do the operation, $\log n$ stored as credit with item x.
- Find(x): Charge 1, and use it to do the operation.
- Union(A, B): Charge 0 and use 1 stored credit from each item in the smaller set to move it.

Amortized Costs: Accounting Method

Accounting method

- MakeSet(x): Charge $(1 + \log n)$. 1 to do the operation, $\log n$ stored as credit with item x.
- Find(x): Charge 1, and use it to do the operation.
- Union(A, B): Charge 0 and use 1 stored credit from each item in the smaller set to move it.

Amortized Costs: Accounting Method

Credit invariant

Total stored credit is $\sum_{S} |S| \log\left(\frac{n}{|S|}\right)$, where the summation is taken over the collection S of all disjoint sets of the current partition.

Amortized Costs: Potential Method

Potential function method

Exercise:

- Define a regular potential function and use it to do the amortized analysis.
- Can you make the Union amortized cost $O(\log n)$, MakeSet and Find costs $O(1)$?

Amortized Costs: Potential Method

Potential function method

Exercise:

- Define a regular potential function and use it to do the amortized analysis.
- Can you make the Union amortized cost $O(\log n)$, MakeSet and Find costs $O(1)$?

Amortized Costs: Potential Method

Potential function method

Exercise:

- Define a regular potential function and use it to do the amortized analysis.
- Can you make the Union amortized cost $O(\log n)$, MakeSet and Find costs $O(1)$?

Outline

Improving over the heuristic using union by rank

Union by Rank

Instead of using the number of nodes in each tree to make a decision, we maintain a **rank, a upper bound on the height of the tree.**

- A is a set if and only if $A = p[A]$ (a tree root).
- $x \in A$ if and only if x is in the tree rooted at A.

Improving over the heuristic using union by rank

Union by Rank

Instead of using the number of nodes in each tree to make a decision, we maintain a **rank, a upper bound on the height of the tree.**

We have the following data structure to support this:

We maintain a parent array p[1..n].

- A is a set **if and only if** $A = p[A]$ (a tree root).
- $x \in A$ **if and only if** x is in the tree rooted at A.

Improving over the heuristic using union by rank

Union by Rank

Instead of using the number of nodes in each tree to make a decision, we maintain a **rank, a upper bound on the height of the tree.**

We have the following data structure to support this:

We maintain a parent array p[1..n].

- A is a set **if and only if** $A = p[A]$ (a tree root).
- $x \in A$ **if and only if** x is in the tree rooted at A.

Improving over the heuristic using union by rank

Union by Rank

Instead of using the number of nodes in each tree to make a decision, we maintain a **rank, a upper bound on the height of the tree.**

We have the following data structure to support this:

We maintain a parent array p[1..n].

- A is a set **if and only if** $A = p[A]$ (a tree root).
- $x \in A$ **if and only if** x is in the tree rooted at A.

Improving over the heuristic using union by rank

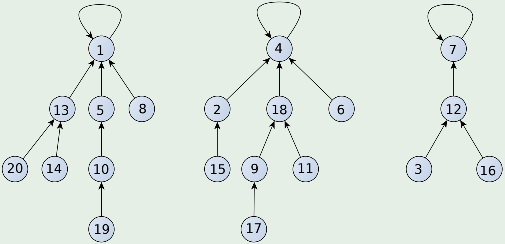
Union by Rank

Instead of using the number of nodes in each tree to make a decision, we maintain a **rank, a upper bound on the height of the tree.**

We have the following data structure to support this:

We maintain a parent array p[1..n].

- A is a set **if and only if** $A = p[A]$ (a tree root).
- $x \in A$ **if and only if** x is in the tree rooted at A.



Forest of Up-Trees: Operations without union by rank or weight

MakeSet(x)

1. p$[x] = x$

Complexity

$O(1)$ time

Union(A, B)

1. p$[B] = A$

   Note: We are assuming that p$[A] == A \neq$p$[B] == B$. This is the reason we need a find operation!!!

Forest of Up-Trees: Operations without union by rank or weight

MakeSet(x)

1. p$[x] = x$

Complexity

$O(1)$ time

Union(A, B)

1. p$[B] = A$

   Note: We are assuming that p$[A] == A \neq$ p$[B] == B$. This is the reason we need a find operation!!!

Forest of Up-Trees: Operations without union by rank or weight

MakeSet(x)

1. p$[x] = x$

Complexity

$O(1)$ time

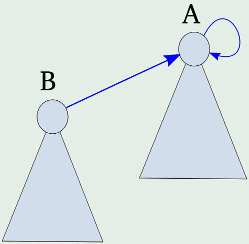
Union(A, B)

1. p$[B] = A$

Note: We are assuming that p$[A] == A \neq$p$[B] == B$. This is the reason we need a find operation!!!

Example

Remember we are doing the joins without caring about getting the worst case



Forest of Up-Trees: Operations without union by rank or weight

Find(x)

```
if $x ==$p$[x]$
        return $x$
return Find(p$[x]$)
```

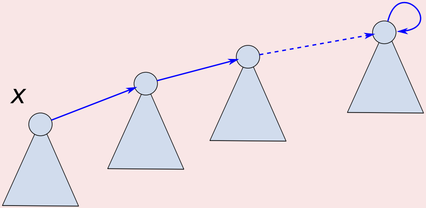
Example

Forest of Up-Trees: Operations without union by rank or weight

Find(x)

```
if $x$ ==p[$x$]
        return $x$
return Find(p [$x$])
```

Example



Forest of Up-Trees: Operations without union by rank or weight

Still I can give you a horrible case

Sequence of operations

```
for $x = 1$ to $n$
        MakeSet($x$)
for $x = 1$ to $n - 1$
        Union($x$)
for $x = 1$ to $n - 1$
        Find($1$)
```

Forest of Up-Trees: Operations without union by rank or weight

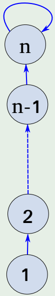
Still I can give you a horrible case

Sequence of operations

```
for $x = 1$ to $n$
        MakeSet$(x)$
for $x = 1$ to $n - 1$
        Union$(x)$
for $x = 1$ to $n - 1$
        Find$(1)$
```

Forest of Up-Trees: Operations without union by rank or weight

We finish with this data structure

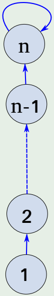


Thus the last part of the sequence give us a total time of

- Aggregate Time $\Theta\left(n^2\right)$
- Amortized Analysis per operation $\Theta\left(n\right)$

Forest of Up-Trees: Operations without union by rank or weight

We finish with this data structure



Thus the last part of the sequence give us a total time of

- Aggregate Time $\Theta\left(n^2\right)$
- Amortized Analysis per operation $\Theta\left(n\right)$

Self-Adjusting forest of Up-Trees

How, we avoid this problem

Use together the following heuristics!!!

1. Balanced Union.
    - By tree weight (i.e., size)
    - By tree rank (i.e., height)
2. Find with path compression

Self-Adjusting forest of Up-Trees

How, we avoid this problem

Use together the following heuristics!!!

1. Balanced Union.
   - By tree weight (i.e., size)
   - By tree rank (i.e., height)
2. Find with path compression

Observations

- Each single improvement (1 or 2) by itself will result in logarithmic amortized cost per operation.
- The two improvements combined will result in amortized cost per operation approaching very close to $O(1)$.

Self-Adjusting forest of Up-Trees

How, we avoid this problem

Use together the following heuristics!!!

1. Balanced Union.
    - By tree weight (i.e., size)
    - By tree rank (i.e., height)
2. Find with path compression

Observations

- Each single improvement (1 or 2) by itself will result in logarithmic amortized cost per operation.
- The two improvements combined will result in amortized cost per operation approaching very close to $O(1)$.

Self-Adjusting forest of Up-Trees

How, we avoid this problem

Use together the following heuristics!!!

1. Balanced Union.
   - By tree weight (i.e., size)
   - By tree rank (i.e., height)
2. Find with path compression

Observations

- Each single improvement (1 or 2) by itself will result in logarithmic amortized cost per operation.
- The two improvements combined will result in amortized cost per operation approaching very close to $O(1)$.

Self-Adjusting forest of Up-Trees

How, we avoid this problem

Use together the following heuristics!!!

1. Balanced Union.
   - By tree weight (i.e., size)
   - By tree rank (i.e., height)
2. Find with path compression

Observations

- Each single improvement (1 or 2) by itself will result in logarithmic amortized cost per operation.
- The two improvements combined will result in amortized cost per operation approaching very close to $O(1)$.

Self-Adjusting forest of Up-Trees

How, we avoid this problem

Use together the following heuristics!!!

1. Balanced Union.
    - By tree weight (i.e., size)
    - By tree rank (i.e., height)
2. Find with path compression

Observations

- Each single improvement (1 or 2) by itself will result in logarithmic amortized cost per operation.
- The two improvements combined will result in amortized cost per operation approaching very close to $O(1)$.

Self-Adjusting forest of Up-Trees

How, we avoid this problem

Use together the following heuristics!!!

1. Balanced Union.
    - By tree weight (i.e., size)
    - By tree rank (i.e., height)
2. Find with path compression

Observations

- Each single improvement (1 or 2) by itself will result in logarithmic amortized cost per operation.
- The two improvements combined will result in amortized cost per operation approaching very close to $O(1)$.

Balanced Union by Size

Using size for Balanced Union

We can use the size of each set to obtain what we want

We have then

MakeSet(x)

```
p$[x] = x$
size$[x] = 1$
```

Note: Complexity $O(1)$ time

Union(A, B)

Input: assume that p[A]=A≠p[B]=B

```
if size$[A]$ >size$[B]$
    size$[A]$ =size$[A]$+size$[B]$
    p$[B] = A$
else
    size$[B]$ =size$[A]$+size$[B]$
    p$[A] = B$
```

Note: Complexity $O(1)$ time

We have then

MakeSet(x)

```
p$[x] = x$
size$[x] = 1$
```

Note: Complexity $O(1)$ time

Union(A, B)

Input: assume that p[A]=A≠p[B]=B

```
if size$[A]$ $>$size$[B]$
    size$[A]$ =size$[A]$+size$[B]$
    p$[B] = A$
else
    size$[B]$ =size$[A]$+size$[B]$
    p$[A] = B$
```

Note: Complexity $O(1)$ time

We have then

MakeSet(x)

```
p$[x] = x$
size$[x] = 1$
```

Note: Complexity $O(1)$ time

Union(A, B)

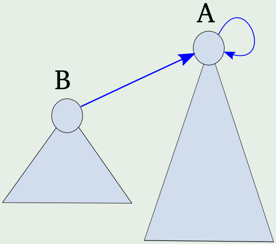
Input: assume that p[A]=A≠p[B]=B

```
if size$[A]$ >size$[B]$
    size$[A]$ =size$[A]$+size$[B]$
    p$[B] = A$
else
    size$[B]$ =size$[A]$+size$[B]$
    p$[A] = B$
```

Note: Complexity $O(1)$ time

Example

Now, we use the size for the union



size[A]>size[B]

Nevertheless

Union by size can make the analysis too complex

People would rather use the rank

Rank

It is defined as the height of the tree

Because

The use of the rank simplify the amortized analysis for the data structure!!!

Nevertheless

Union by size can make the analysis too complex

People would rather use the rank

Rank

It is defined as the height of the tree

Because

The use of the rank simplify the amortized analysis for the data structure!!!

Nevertheless

Union by size can make the analysis too complex

People would rather use the rank

Rank

It is defined as the height of the tree

Because

The use of the rank simplify the amortized analysis for the data structure!!!

Thus, we use the balanced union by rank

MakeSet(x)

1. p$[x] = x$
2. rank$[x] = 0$

Note: Complexity $O(1)$ time

Union(A, B)

Input: assume that p[A]=A≠p[B]=B

```
if rank$[A]$ >rank$[B]$
    p$[B] = A$
else
    p$[A] = B$
    if rank$[A]$ ==rank$[B]$
        rank$[B]$=rank$[B]$+1
```

Note: Complexity $O(1)$ time

Thus, we use the balanced union by rank

MakeSet(x)

```
p$[x] = x$
rank$[x] = 0$
```

Note: Complexity $O(1)$ time

Union(A, B)

Input: assume that p[A]=A≠p[B]=B

```
if rank$[A]$ >rank$[B]$
        p$[B] = A$
else
        p$[A] = B$
        if rank$[A]$ ==rank$[B]$
                rank$[B]$=rank$[B]$+1
```

Note: Complexity $O(1)$ time

Thus, we use the balanced union by rank

MakeSet(x)

```
p$[x] = x$
rank$[x] = 0$
```

Note: Complexity $O(1)$ time

Union(A, B)

Input: assume that p[A]=A≠p[B]=B

```
if rank$[A]$ >rank$[B]$
        p$[B] = A$
else
        p$[A] = B$
        if rank$[A]$ ==rank$[B]$
                rank$[B]$=rank$[B]$+1
```

Note: Complexity $O(1)$ time

Example

Now

We use the rank for the union

Case I

The rank of A is larger than B

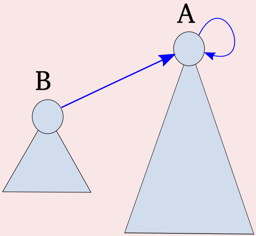
Example

Now

We use the rank for the union

Case I

The rank of A is larger than B



rank[A]>rank[B]

Example

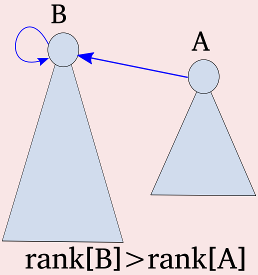
Case II

The rank of B is larger than A

Example

Case II

The rank of B is larger than A



Outline

Here is the new heuristic to improve overall performance:

Path Compression

```
Find$(x)$
if x$\neq$p$[x]$
        p$[x]$=Find(p $[x]$)
return p$[x]$
```

Complexity

$O\left(depth\left(x\right)\right)$ time

Here is the new heuristic to improve overall performance:

Path Compression

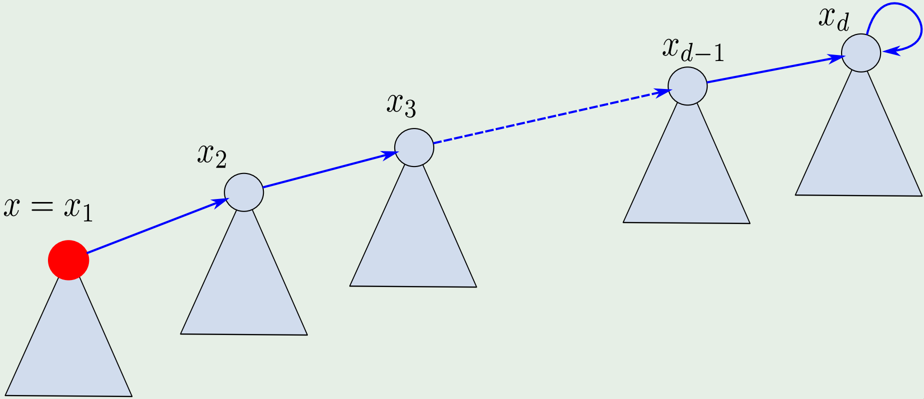
```
Find($x$)
 1. if x≠p[$x$]
 2.         p[$x$]=Find(p[$x$])
 3. return p[$x$]
```

Complexity

$O\left(depth\left(x\right)\right)$ time

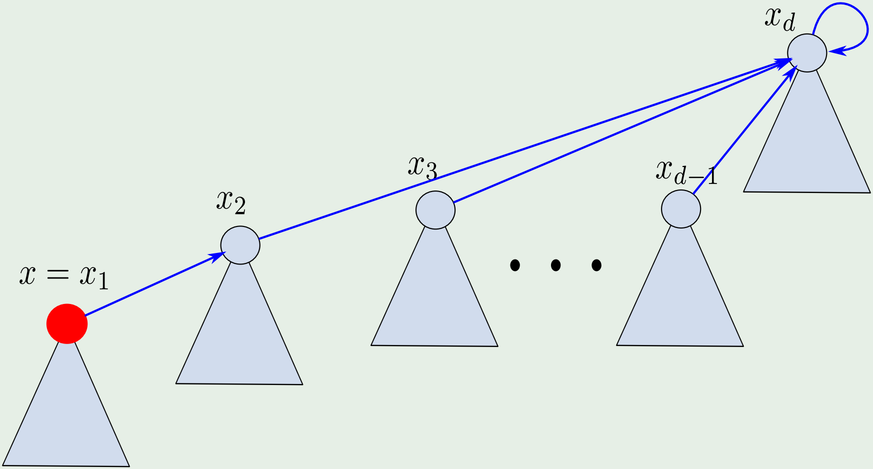
Example

We have the following structure



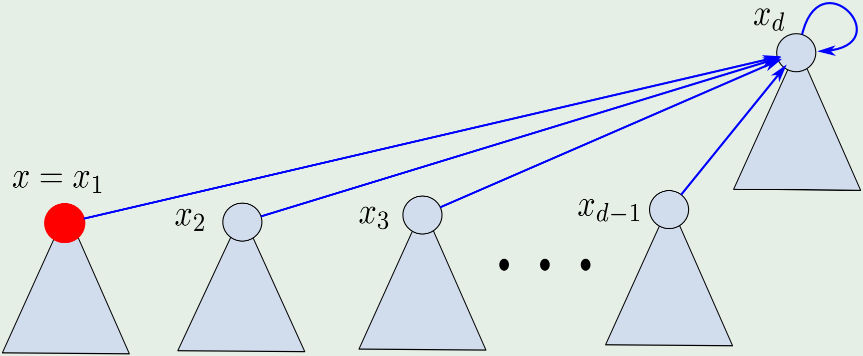
Example

The recursive Find(p $[x]$)



Example

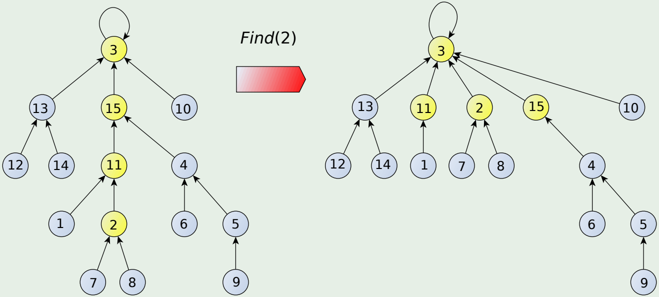
The recursive Find(p $[x]$)



Path compression

Find(x) should traverse the path from x up to its root.

This might as well create shortcuts along the way to improve the efficiency of the future operations.



Outline

Time complexity

Tight upper bound on time complexity

- An amortized time of $O(m\alpha(m,n))$ for m operations.
- Where $\alpha(m,n)$ is the inverse of the Ackermann's function (almost a constant).
- This bound, for a slightly different definition of α than that given here is shown in Cormen's book.

Time complexity

Tight upper bound on time complexity

- An amortized time of $O(m\alpha(m,n))$ for m operations.
- Where $\alpha(m,n)$ is the inverse of the Ackermann's function (almost a constant).
- This bound, for a slightly different definition of α than that given here is shown in Cormen's book.

Time complexity

Tight upper bound on time complexity

- An amortized time of $O(m\alpha(m,n))$ for m operations.
- Where $\alpha(m,n)$ is the inverse of the Ackermann's function (almost a constant).
- This bound, for a slightly different definition of α than that given here is shown in Cormen's book.

Ackermann's Function

Definition

- $A(1, j) = 2^j$ where $j \geq 1$
- $A(i, 1) = A(i - 1, 2)$ where $i \geq 2$
- $A(i, j) = A(i - 1, A(i, j - 1))$ where $i, j \geq 2$

Note:

- This is one of several in-equivalent but similar definitions of Ackermann's function found in the literature.
- Cormen's book authors give a different definition, although they never really call theirs Ackermann's function.

Ackermann's Function

Definition

- $A(1, j) = 2^j$ where $j \geq 1$
- $A(i, 1) = A(i-1, 2)$ where $i \geq 2$
- $A(i, j) = A(i-1, A(i, j-1))$ where $i, j \geq 2$

Note:
- This is one of several in-equivalent but similar definitions of Ackermann's function found in the literature.
- Cormen's book authors give a different definition, although they never really call theirs Ackermann's function.

Property

Ackermann's function grows very fast, thus it's inverse grows very slow.

Ackermann's Function

Definition

- $A(1, j) = 2^j$ where $j \geq 1$
- $A(i, 1) = A(i-1, 2)$ where $i \geq 2$
- $A(i, j) = A(i-1, A(i, j-1))$ where $i, j \geq 2$

Note:

- This is one of several in-equivalent but similar definitions of Ackermann's function found in the literature.
- Cormen's book authors give a different definition, although they never really call theirs Ackermann's function.

Property

Ackermann's function grows very fast, thus it's inverse grows very slow.

Ackermann's Function

Definition

- $A(1, j) = 2^j$ where $j \geq 1$
- $A(i, 1) = A(i-1, 2)$ where $i \geq 2$
- $A(i, j) = A(i-1, A(i, j-1))$ where $i, j \geq 2$

Note:
- This is one of several in-equivalent but similar definitions of Ackermann's function found in the literature.
- Cormen's book authors give a different definition, although they never really call theirs Ackermann's function.

Property

Ackermann's function grows very fast, thus it's inverse grows very slow.

Ackermann's Function

Definition

- $A(1, j) = 2^j$ where $j \geq 1$
- $A(i, 1) = A(i-1, 2)$ where $i \geq 2$
- $A(i, j) = A(i-1, A(i, j-1))$ where $i, j \geq 2$

Note:
- This is one of several in-equivalent but similar definitions of Ackermann's function found in the literature.
- Cormen's book authors give a different definition, although they never really call theirs Ackermann's function.

Property

Ackermann's function grows very fast, thus it's inverse grows very slow.

Ackermann's Function

Definition

- $A(1, j) = 2^j$ where $j \geq 1$
- $A(i, 1) = A(i-1, 2)$ where $i \geq 2$
- $A(i, j) = A(i-1, A(i, j-1))$ where $i, j \geq 2$

Note:

- This is one of several in-equivalent but similar definitions of Ackermann's function found in the literature.
- Cormen's book authors give a different definition, although they never really call theirs Ackermann's function.

Property

Ackermann's function grows very fast, thus it's inverse grows very slow.

Ackermann's Function

Example $A(3,4)$

$$A\left(3,4\right)=2^{\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}^{\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}^{\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}^{16}}}}$$

Notation: $2^{\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}^{10}}$ means $2^{2^{2^{2^{2^{2^{2^{2^{2^{2^{2}}}}}}}}}}$

Inverse of Ackermann's function

Definition

$$\alpha(m,n)=\min\left\{i\geq 1 | A\left(i,\left\lfloor\frac{m}{n}\right\rfloor\right)>\log n\right\} \tag{3}$$

Note: This is not a true mathematical inverse.

Intuition: Grows about as slowly as Ackermann's function does fast.

Inverse of Ackermann's function

Definition

$$\alpha(m,n)=\min\left\{i\geq 1 | A\left(i,\left\lfloor\frac{m}{n}\right\rfloor\right)>\log n\right\} \tag{3}$$

Note: This is not a true mathematical inverse.

Intuition: Grows about as slowly as Ackermann's function does fast.

How slowly?

Let $\left\lfloor\frac{m}{n}\right\rfloor=k$, then $m\geq n\rightarrow k\geq 1$

Inverse of Ackermann's function

Definition

$$\alpha(m,n) = \min\left\{ i \geq 1 | A\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) > \log n \right\} \tag{3}$$

Note: This is not a true mathematical inverse.

Intuition: Grows about as slowly as Ackermann's function does fast.

How slowly?

Let $\left\lfloor \frac{m}{n} \right\rfloor = k$, then $m \geq n \rightarrow k \geq 1$

Inverse of Ackermann's function

Definition

$$\alpha(m,n) = \min\left\{ i \geq 1 | A\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) > \log n \right\} \tag{3}$$

Note: This is not a true mathematical inverse.

Intuition: Grows about as slowly as Ackermann's function does fast.

How slowly?

Let $\lfloor \frac{m}{n} \rfloor = k$, then $m \geq n \to k \geq 1$

Thus

First

We can show that $A(i,k) \geq A(i,1)$ for all $i \geq 1$.

- This is left to you...

Thus

First

We can show that $A(i,k) \geq A(i,1)$ for all $i \geq 1$.

- This is left to you...

For Example

Consider $i = 4$, then $A(i,k) \geq A(4,1) = 2^{\left.2^{\cdot^{\cdot^{2}}}\right\}10} \approx 10^{80}$.

Thus

First

We can show that $A(i,k) \geq A(i,1)$ for all $i \geq 1$.

- This is left to you...

For Example

Consider $i = 4$, then $A(i,k) \geq A(4,1) = 2^{2^{\cdot^{\cdot^{2}}}}\Big\}10 \approx 10^{80}$.

Finally

if $\log n < 10^{80}$. i.e., if $n < 2^{10^{80}} \implies \alpha(m,n) \leq 4$

Thus

First

We can show that $A(i,k) \geq A(i,1)$ for all $i \geq 1$.

- This is left to you...

For Example

Consider $i = 4$, then $A(i,k) \geq A(4,1) = 2^{\left.2^{\cdot^{\cdot^{2}}}\right\}10} \approx 10^{80}$.

Finally

if $\log n < 10^{80}$. i.e., if $n < 2^{10^{80}} \implies \alpha(m,n) \leq 4$

Thus

First

We can show that $A(i,k) \geq A(i,1)$ for all $i \geq 1$.

- This is left to you...

For Example

Consider $i = 4$, then $A(i,k) \geq A(4,1) = 2^{\left.2^{\cdot^{\cdot^{2}}}\right\}10} \approx 10^{80}$.

Finally

if $\log n < 10^{80}$. i.e., if $n < 2^{10^{80}} \implies \alpha(m,n) \leq 4$

Instead of Using the Ackermann Inverse

We define the following function

$$\log^* n = \min\left\{ i \geq 0 | \log^{(i)} n \leq 1 \right\} \tag{4}$$

- The i means $\log \cdots \log n$ i times

Then

We will establish $O\left(m \log^* n\right)$ as upper bound.

Instead of Using the Ackermann Inverse

We define the following function

$$\log^* n = \min\left\{i \geq 0 | \log^{(i)} n \leq 1\right\} \tag{4}$$

- The i means $\log \cdots \log n$ i times

Then

We will establish $O\left(m \log^* n\right)$ as upper bound.

In particular

Something Notable

In particular, we have that $\log^* 2^{\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}k} = k+1$

For Example

$$\log^* 2^{65536} = 2^{\left.2^{2^{2}}\right\}4} = 5 \tag{5}$$

Therefore

We have that $\log^* n \leq 5$ for all practical purposes.

In particular

Something Notable

In particular, we have that $\log^* 2^{\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}k} = k+1$

For Example

$$\log^* 2^{65536} = 2^{\left.2^{2^{2^{2}}}\right\}4} = 5 \tag{5}$$

Therefore

We have that $\log^* n \leq 5$ for all practical purposes.

In particular

Something Notable

In particular, we have that $\log^* 2^{\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}k} = k+1$

For Example

$$\log^* 2^{65536} = 2^{\left.2^{2^{2^{2}}}\right\}4} = 5 \tag{5}$$

Therefore

We have that $\log^* n \leq 5$ for all practical purposes.

The Rank Observation

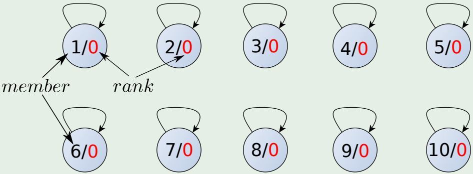
Something Notable

It is that once somebody becomes a child of another node their rank does not change given any posterior operation.

For Example

The number in the right is the height

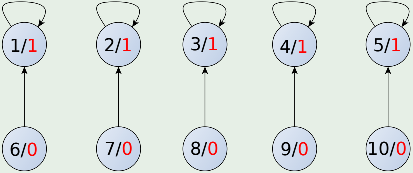
MakeSet(1),MakeSet(2),MakeSet(3), ..., MakeSet(10)



Example

Now, we do

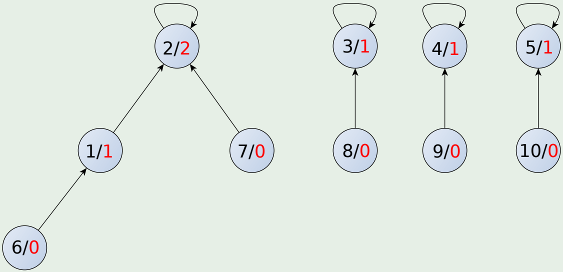
Union$(6,1)$,Union$(7,2)$, ..., Union$(10,1)$



Example

Next - Assuming that you are using a FindSet to get the name set

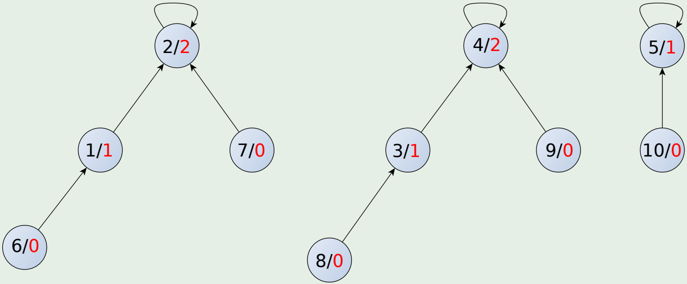
Union$(1, 2)$



Example

Next

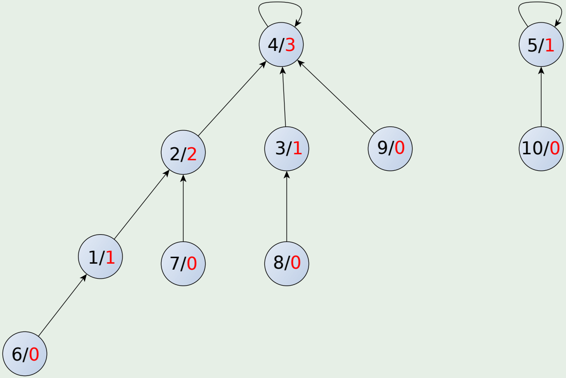
Union$(3,4)$



Example

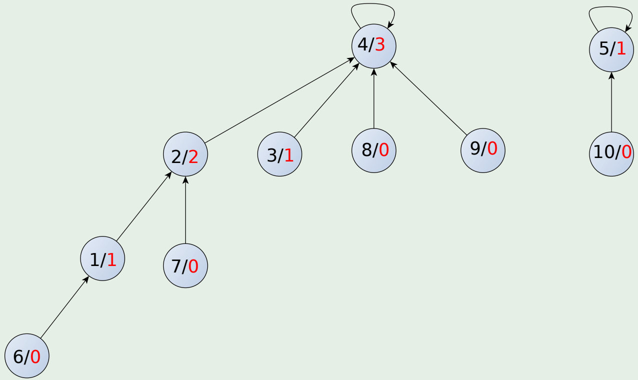
Next

$\text{Union}(2, 4)$



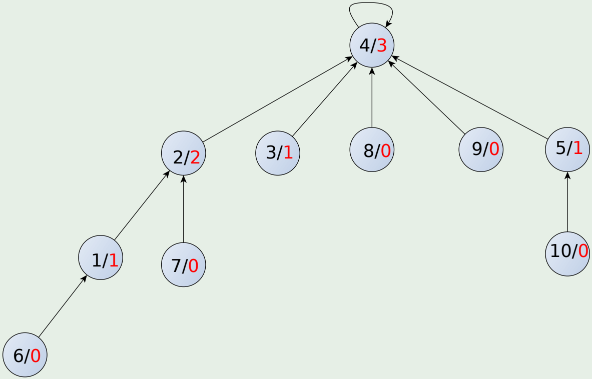
Example

Now you give a FindSet(8)



Example

Now you give a Union$(4, 5)$



Properties of ranks

Lemma 1 (About the Rank Properties)

1. $\forall x$, $rank[x] \leq rank[p[x]]$.
2. $\forall x$ and $x \neq p[x]$, then $rank[x] < rank[p[x]]$.
3. $rank[x]$ is initially 0.
4. $rank[x]$ does not decrease.
5. Once $x \neq p[x]$ holds $rank[x]$ does not change.
6. $rank[p[x]]$ is a monotonically increasing function of time.

Properties of ranks

Lemma 1 (About the Rank Properties)

1. $\forall x$, $rank[x] \leq rank[p[x]]$.
2. $\forall x$ and $x \neq p[x]$, then $rank[x] < rank[p[x]]$.
3. $rank[x]$ is initially 0.
4. $rank[x]$ does not decrease.
5. Once $x \neq p[x]$ holds $rank[x]$ does not change.
6. $rank[p[x]]$ is a monotonically increasing function of time.

Proof

By induction on the number of operations...

Properties of ranks

Lemma 1 (About the Rank Properties)

1. $\forall x$, $rank[x] \leq rank[p[x]]$.
2. $\forall x$ and $x \neq p[x]$, then $rank[x] < rank[p[x]]$.
3. $rank[x]$ is initially 0.
4. $rank[x]$ does not decrease.
5. Once $x \neq p[x]$ holds $rank[x]$ does not change.
6. $rank[p[x]]$ is a monotonically increasing function of time.

Proof

By induction on the number of operations...

Properties of ranks

Lemma 1 (About the Rank Properties)

1. $\forall x$, $rank[x] \leq rank[p[x]]$.
2. $\forall x$ and $x \neq p[x]$, then $rank[x] < rank[p[x]]$.
3. $rank[x]$ is initially 0.
4. $rank[x]$ does not decrease.
5. Once $x \neq p[x]$ holds $rank[x]$ does not change.
6. $rank[p[x]]$ is a monotonically increasing function of time.

Proof

By induction on the number of operations...

Properties of ranks

Lemma 1 (About the Rank Properties)

1. $\forall x$, $rank[x] \leq rank[p[x]]$.
2. $\forall x$ and $x \neq p[x]$, then $rank[x] < rank[p[x]]$.
3. $rank[x]$ is initially 0.
4. $rank[x]$ does not decrease.
5. Once $x \neq p[x]$ holds $rank[x]$ does not change.
6. $rank[p[x]]$ is a monotonically increasing function of time.

Proof

By induction on the number of operations...

Properties of ranks

Lemma 1 (About the Rank Properties)

1. $\forall x$, $rank[x] \leq rank[p[x]]$.
2. $\forall x$ and $x \neq p[x]$, then $rank[x] < rank[p[x]]$.
3. $rank[x]$ is initially 0.
4. $rank[x]$ does not decrease.
5. Once $x \neq p[x]$ holds $rank[x]$ does not change.
6. $rank[p[x]]$ is a monotonically increasing function of time.

Proof

By induction on the number of operations...

Properties of ranks

Lemma 1 (About the Rank Properties)

1. $\forall x$, $rank[x] \leq rank[p[x]]$.
2. $\forall x$ and $x \neq p[x]$, then $rank[x] < rank[p[x]]$.
3. $rank[x]$ is initially 0.
4. $rank[x]$ does not decrease.
5. Once $x \neq p[x]$ holds $rank[x]$ does not change.
6. $rank[p[x]]$ is a monotonically increasing function of time.

Proof

By induction on the number of operations...

For Example

Imagine a MakeSet(x)

- Then, $rank\left[x\right] \leq rank\left[p\left[x\right]\right]$
- Thus, it is true after n operations.
- The we get the $n+1$ operations that can be:
    - Case I - FindSet.
    - Case II - Union.

For Example

Imagine a MakeSet(x)

- Then, $rank\left[x\right] \leq rank\left[p\left[x\right]\right]$
- Thus, it is true after n operations.
- The we get the $n+1$ operations that can be:
  - Case I - FindSet.
  - Case II - Union.

The rest are for you to prove

It is a good mental exercise!!!

For Example

Imagine a MakeSet(x)

- Then, $rank\left[x\right] \leq rank\left[p\left[x\right]\right]$
- Thus, it is true after n operations.
- The we get the $n+1$ operations that can be:
    - Case I - FindSet.
    - Case II - Union.

The rest are for you to prove

It is a good mental exercise!!!

For Example

Imagine a MakeSet(x)

- Then, $rank\left[x\right] \leq rank\left[p\left[x\right]\right]$
- Thus, it is true after n operations.
- The we get the $n+1$ operations that can be:
    - Case I - FindSet.
    - Case II - Union.

The rest are for you to prove

It is a good mental exercise!!!

For Example

Imagine a MakeSet(x)

- Then, $rank\left[x\right] \leq rank\left[p\left[x\right]\right]$
- Thus, it is true after n operations.
- The we get the $n+1$ operations that can be:
  - Case I - FindSet.
  - Case II - Union.

The rest are for you to prove

It is a good mental exercise!!!

The Number of Nodes in a Tree

Lemma 2

For all tree roots x, $size(x) \geq 2^{rank[x]}$

Note: $size(x)$= Number of nodes in tree rooted at x

The Number of Nodes in a Tree

Lemma 2

For all tree roots x, $size(x) \geq 2^{rank[x]}$

Note $size\,(x)$= Number of nodes in tree rooted at x

Proof

By induction on the number of link operations:

- Basis Step
  - Before first link, all ranks are 0 and each tree contains one node.
- Inductive Step
  - Consider linking x and y $(Link\,(x, y))$
  - Assume lemma holds before this operation; we show that it will holds after.

The Number of Nodes in a Tree

Lemma 2

For all tree roots x, $size(x) \geq 2^{rank[x]}$

Note $size\,(x)$= Number of nodes in tree rooted at x

Proof

By induction on the number of link operations:

- Basis Step
  - Before first link, all ranks are 0 and each tree contains one node.
- Inductive Step
  - Consider linking x and y $(Link\,(x, y))$
  - Assume lemma holds before this operation; we show that it will holds after.

The Number of Nodes in a Tree

Lemma 2

For all tree roots x, $size(x) \geq 2^{rank[x]}$

Note $size\,(x)$= Number of nodes in tree rooted at x

Proof

By induction on the number of link operations:

- Basis Step
    - Before first link, all ranks are 0 and each tree contains one node.
- Inductive Step
    - Consider linking x and y $(Link\,(x, y))$
    - Assume lemma holds before this operation; we show that it will holds after.

The Number of Nodes in a Tree

Lemma 2

For all tree roots x, $size(x) \geq 2^{rank[x]}$

Note $size(x)=$ Number of nodes in tree rooted at x

Proof

By induction on the number of link operations:

- Basis Step
  - Before first link, all ranks are 0 and each tree contains one node.
- Inductive Step
  - Consider linking x and y $(Link(x,y))$
  - Assume lemma holds before this operation; we show that it will holds after.

The Number of Nodes in a Tree

Lemma 2

For all tree roots x, $size(x) \geq 2^{rank[x]}$

Note $size\,(x)$= Number of nodes in tree rooted at x

Proof

By induction on the number of link operations:

- Basis Step
  - Before first link, all ranks are 0 and each tree contains one node.
- Inductive Step
  - Consider linking x and y $(Link\,(x,y))$
  - Assume lemma holds before this operation; we show that it will holds after.

The Number of Nodes in a Tree

Lemma 2

For all tree roots x, $size(x) \geq 2^{rank[x]}$

Note $size\,(x)$= Number of nodes in tree rooted at x

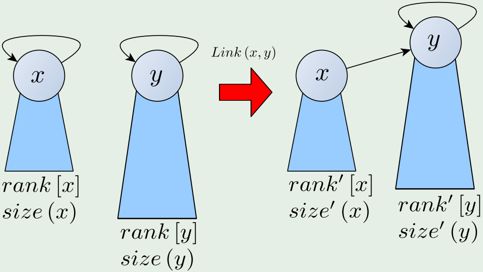
Proof

By induction on the number of link operations:

- Basis Step
  - Before first link, all ranks are 0 and each tree contains one node.
- Inductive Step
  - Consider linking x and y $(Link\,(x, y))$
  - Assume lemma holds before this operation; we show that it will holds after.

Case 1: $rank[x] \neq rank[y]$

Assume $rank\left[x\right] < rank\left[y\right]$



Note:

- $rank'\left[x\right] == rank\left[x\right]$ and $rank'\left[y\right] == rank\left[y\right]$

Therefore

We have that

$$
\begin{aligned}
size'\left(y\right) &= size\left(x\right) + size\left(y\right)\\
&\geq 2^{rank[x]} + 2^{rank[y]}\\
&\geq 2^{rank[y]}\\
&= 2^{rank'[y]}
\end{aligned}
$$

Therefore

We have that

$$
\begin{aligned}
size'(y) &= size(x) + size(y) \\
&\geq 2^{rank[x]} + 2^{rank[y]} \\
&\geq 2^{rank[y]} \\
&= 2^{rank'[y]}
\end{aligned}
$$

Therefore

We have that

$$
\begin{aligned}
size'(y) &= size(x) + size(y)\\
&\geq 2^{rank[x]} + 2^{rank[y]}\\
&\geq 2^{rank[y]}\\
&= 2^{rank'[y]}
\end{aligned}
$$

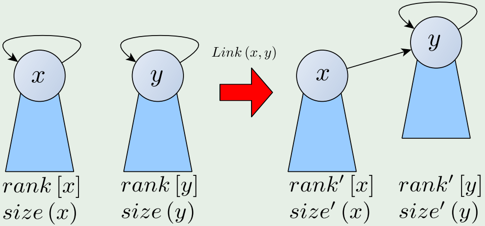
Therefore

We have that

$$\begin{aligned} size'(y) &= size(x) + size(y) \\ &\geq 2^{rank[x]} + 2^{rank[y]} \\ &\geq 2^{rank[y]} \\ &= 2^{rank'[y]} \end{aligned}$$

Case 2: $rank[x] == rank[y]$

Assume $rank\left[x\right] == rank\left[y\right]$



Note:

- $rank'\left[x\right] == rank\left[x\right]$ and $rank'\left[y\right] == rank\left[y\right] + 1$

Therefore

We have that

$$\begin{aligned} size'(y) &= size(x) + size(y) \\ &\geq 2^{rank[x]} + 2^{rank[y]} \\ &\geq 2^{rank[y]+1} \\ &= 2^{rank'[y]} \end{aligned}$$

Note: In the worst case $rank[x] == rank[y] == 0$

Therefore

We have that

$$
\begin{aligned}
size'(y) &= size(x) + size(y) \\
&\geq 2^{rank[x]} + 2^{rank[y]} \\
&\geq 2^{rank[y]+1} \\
&= 2^{rank'[y]}
\end{aligned}
$$

Note: In the worst case $rank[x] == rank[y] == 0$

Therefore

We have that

$$
\begin{aligned}
size'(y) &= size(x) + size(y) \\
&\geq 2^{rank[x]} + 2^{rank[y]} \\
&\geq 2^{rank[y]+1} \\
&= 2^{rank'[y]}
\end{aligned}
$$

Note: In the worst case $rank[x] == rank[y] == 0$

Therefore

We have that

$$
\begin{aligned}
size'(y) &= size(x) + size(y)\\
&\geq 2^{rank[x]} + 2^{rank[y]}\\
&\geq 2^{rank[y]+1}\\
&= 2^{rank'[y]}
\end{aligned}
$$

Note: In the worst case $rank[x] == rank[y] == 0$

Therefore

We have that

$$\begin{aligned} size'(y) &= size(x) + size(y) \\ &\geq 2^{rank[x]} + 2^{rank[y]} \\ &\geq 2^{rank[y]+1} \\ &= 2^{rank'[y]} \end{aligned}$$

Note: In the worst case $rank[x] == rank[y] == 0$

The number of nodes at certain rank

Lemma 3

For any integer $r \geq 0$, there are an most $\frac{n}{2^r}$ nodes of rank r.

The number of nodes at certain rank

Lemma 3

For any integer $r \geq 0$, there are an most $\frac{n}{2^r}$ nodes of rank r.

Proof

- First fix r.
- When rank r is assigned to some node x, then imagine that you label each node in the tree rooted at x by "x."
- By lemma 21.3, 2^r or more nodes are labeled each time when executing a union.
- By lemma 21.2, each node is labeled at most once, when its root is first assigned rank r.
- **If there were more than** $\frac{n}{2^r}$ nodes of rank r.
- Then, we will have that more than $2^r \cdot \left(\frac{n}{2^r}\right) = n$ nodes would be labeled by a node of rank r, a contradiction.

The number of nodes at certain rank

Lemma 3

For any integer $r \geq 0$, there are an most $\frac{n}{2^r}$ nodes of rank r.

Proof

- First fix r.
- When rank r is assigned to some node x, then imagine that you label each node in the tree rooted at x by "x."
- By lemma 21.3, 2^r or more nodes are labeled each time when executing a union.
- By lemma 21.2, each node is labeled at most once, when its root is first assigned rank r.
- **If there were more than** $\frac{n}{2^r}$ nodes of rank r.
- Then, we will have that more than $2^r \cdot \left(\frac{n}{2^r}\right) = n$ nodes would be labeled by a node of rank r, a contradiction.

The number of nodes at certain rank

Lemma 3

For any integer $r \geq 0$, there are an most $\frac{n}{2^r}$ nodes of rank r.

Proof

- First fix r.
- When rank r is assigned to some node x, then imagine that you label each node in the tree rooted at x by "x."
- By lemma 21.3, 2^r or more nodes are labeled each time when executing a union.
- By lemma 21.2, each node is labeled at most once, when its root is first assigned rank r.
- **If there were more than** $\frac{n}{2^r}$ nodes of rank r.
- Then, we will have that more than $2^r \cdot \left(\frac{n}{2^r}\right) = n$ nodes would be labeled by a node of rank r, a contradiction.

The number of nodes at certain rank

Lemma 3

For any integer $r \geq 0$, there are an most $\frac{n}{2^r}$ nodes of rank r.

Proof

- First fix r.
- When rank r is assigned to some node x, then imagine that you label each node in the tree rooted at x by "x."
- By lemma 21.3, 2^r or more nodes are labeled each time when executing a union.
- By lemma 21.2, each node is labeled at most once, when its root is first assigned rank r.
- **If there were more than** $\frac{n}{2^r}$ nodes of rank r.
- Then, we will have that more than $2^r \cdot \left(\frac{n}{2^r}\right) = n$ nodes would be labeled by a node of rank r, a contradiction.

The number of nodes at certain rank

Lemma 3

For any integer $r \geq 0$, there are an most $\frac{n}{2^r}$ nodes of rank r.

Proof

- First fix r.
- When rank r is assigned to some node x, then imagine that you label each node in the tree rooted at x by "x."
- By lemma 21.3, 2^r or more nodes are labeled each time when executing a union.
- By lemma 21.2, each node is labeled at most once, when its root is first assigned rank r.
- **If there were more than** $\frac{n}{2^r}$ nodes of rank r.
- Then, we will have that more than $2^r \cdot \left(\frac{n}{2^r}\right) = n$ nodes would be labeled by a node of rank r, a contradiction.

The number of nodes at certain rank

Lemma 3

For any integer $r \geq 0$, there are an most $\frac{n}{2^r}$ nodes of rank r.

Proof

- First fix r.
- When rank r is assigned to some node x, then imagine that you label each node in the tree rooted at x by "x."
- By lemma 21.3, 2^r or more nodes are labeled each time when executing a union.
- By lemma 21.2, each node is labeled at most once, when its root is first assigned rank r.
- **If there were more than** $\frac{n}{2^r}$ nodes of rank r.
- Then, we will have that more than $2^r \cdot \left(\frac{n}{2^r}\right) = n$ nodes would be labeled by a node of rank r, a contradiction.

Corollary 1

Corollary 1

Every node has rank at most $\lfloor \log n \rfloor$.

Proof

if there is a rank r such that $r > \log n \rightarrow \frac{n}{2^r} < 1$ nodes of rank r a contradiction.

Corollary 1

Corollary 1

Every node has rank at most $\lfloor \log n \rfloor$.

Proof

if there is a rank r such that $r > \log n \rightarrow \frac{n}{2^r} < 1$ nodes of rank r a contradiction.

Providing the time bound

Lemma 4 (Lemma 21.7)

Suppose we convert a sequence S' of m' MakeSet, Union and FindSet operations into a sequence S of m MakeSet, Link, and FindSet operations by turning each Union into two FindSet operations followed by a Link. Then, if sequence S runs in $O(m \log^* n)$ time, sequence S' runs in $O(m' \log^* n)$ time.

Proof:

The proof is quite easy

1. Since each UNION operation in sequence S' is converted into three operations in S.

$$m' \leq m \leq 3m' \tag{6}$$

2. We have that $m = O(m')$
3. Then, if the new sequence S runs in $O(m \log^* n)$ this implies that the old sequence S' runs in $O(m' \log^* n)$

Proof:

The proof is quite easy

1. Since each UNION operation in sequence S' is converted into three operations in S.

$$m' \leq m \leq 3m' \tag{6}$$

2. We have that $m = O\left(m'\right)$
3. Then, if the new sequence S runs in $O\left(m \log^* n\right)$ this implies that the old sequence S' runs in $O\left(m' \log^* n\right)$

Proof:

The proof is quite easy

1. Since each UNION operation in sequence S' is converted into three operations in S.

$$m' \leq m \leq 3m' \tag{6}$$

2. We have that $m = O(m')$
3. Then, if the new sequence S runs in $O(m \log^* n)$ this implies that the old sequence S' runs in $O(m' \log^* n)$

Theorem for Union by Rank and Path Compression

Theorem

Any sequence of m MakeSet, Link, and FindSet operations, n of which are MakeSet operations, is performed in worst-case time $O(m \log^* n)$.

Theorem for Union by Rank and Path Compression

Theorem

Any sequence of m MakeSet, Link, and FindSet operations, n of which are MakeSet operations, is performed in worst-case time $O(m \log^* n)$.

Proof

- First, MakeSet and Link take $O(1)$ time.
- The Key of the Analysis is to **Accurately Charging FindSet.**

Theorem for Union by Rank and Path Compression

Theorem

Any sequence of m MakeSet, Link, and FindSet operations, n of which are MakeSet operations, is performed in worst-case time $O(m \log^* n)$.

Proof

- First, MakeSet and Link take $O(1)$ time.
- The Key of the Analysis is to **Accurately Charging FindSet**.

For this, we have the following

We can do the following

- Partition ranks into blocks.
- Put each rank j into block $\log^* r$ for $r = 0, 1, ..., \lfloor \log n \rfloor$ (Corollary 1).
- Highest-numbered block is $\log^*(\log n) = (\log^* n) - 1$.

For this, we have the following

We can do the following

- Partition ranks into blocks.
- Put each rank j into block $\log^* r$ for $r = 0, 1, ..., \lfloor \log n \rfloor$ (Corollary 1).
- Highest-numbered block is $\log^*(\log n) = (\log^* n) - 1$.

In addition, the cost of FindSet pays for the foollowing situations

1. The FindSet pays for the cost of the root and its child.
2. A bill is given to every node whose rank parent changes in the path compression!!!

For this, we have the following

We can do the following

- Partition ranks into blocks.
- Put each rank j into block $\log^* r$ for $r = 0, 1, \ldots, \lfloor \log n \rfloor$ (Corollary 1).
- Highest-numbered block is $\log^*(\log n) = (\log^* n) - 1$.

In addition, the cost of FindSet pays for the foollowing situations

1. The FindSet pays for the cost of the root and its child.
2. A bill is given to every node whose rank parent changes in the path compression!!!

For this, we have the following

We can do the following

- Partition ranks into blocks.
- Put each rank j into block $\log^* r$ for $r = 0, 1, ..., \lfloor \log n \rfloor$ (Corollary 1).
- Highest-numbered block is $\log^*(\log n) = (\log^* n) - 1$.

In addition, the cost of FindSet pays for the foollowing situations

1. The FindSet pays for the cost of the root and its child.
2. A bill is given to every node whose rank parent changes in the path compression!!!

For this, we have the following

We can do the following

- Partition ranks into blocks.
- Put each rank j into block $\log^* r$ for $r = 0, 1, ..., \lfloor \log n \rfloor$ (Corollary 1).
- Highest-numbered block is $\log^*(\log n) = (\log^* n) - 1$.

In addition, the cost of FindSet pays for the foollowing situations

1. The FindSet pays for the cost of the root and its child.
2. A bill is given to every node whose rank parent changes in the path compression!!!

Now, define the Block function

Define the following Upper Bound Function

$$B(j) \equiv \begin{cases} -1 & \text{if } j = -1 \\ 1 & \text{if } j = 0 \\ 2 & \text{if } j = 1 \\ 2^{\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}j-1} & \text{if } j \geq 2 \end{cases}$$

First

Something Notable

These are going to be the upper bounds for blocks in the ranks

Where

For $j = 0, 1, ..., \log^* n - 1$, block j consist of the set of ranks:

$$\underbrace{B(j-1)+1, B(j-1)+2, ..., B(j)}_{\text{Elements in Block } j} \tag{7}$$

First

Something Notable

These are going to be the upper bounds for blocks in the ranks

Where

For $j = 0, 1, ..., \log^* n - 1$, block j consist of the set of ranks:

$$\underbrace{B(j-1)+1, B(j-1)+2, ..., B(j)}_{\text{Elements in Block } j} \tag{7}$$

For Example

We have that

$$B(-1) = 1$$

$$B(0) = 0$$

$$B(1) = 2$$

$$B(2) = 2^2 = 4$$

$$B(3) = 2^{2^2} = 2^4 = 16$$

$$B(4) = 2^{2^{2^2}} = 2^{16} = 65536$$

For Example

We have that

$$
\begin{aligned}
B(-1) &= 1\\
B(0) &= 0\\
B(1) &= 2\\
B(2) &= 2^2 = 4\\
B(3) &= 2^{2^2} = 2^4 = 16\\
B(4) &= 2^{2^{2^2}} = 2^{16} = 65536
\end{aligned}
$$

For Example

We have that

$$
\begin{aligned}
B(-1) &= 1\\
B(0) &= 0\\
B(1) &= 2\\
B(2) &= 2^2 = 4\\
B(3) &= 2^{2^2} = 2^4 = 16\\
B(4) &= 2^{2^{2^2}} = 2^{16} = 65536
\end{aligned}
$$

For Example

We have that

$$
\begin{aligned}
B(-1) &= 1 \\
B(0) &= 0 \\
B(1) &= 2 \\
B(2) &= 2^2 = 4 \\
B(3) &= 2^{2^2} = 2^4 = 16 \\
B(4) &= 2^{2^{2^2}} = 2^{16} = 65536
\end{aligned}
$$

For Example

We have that

$$
\begin{aligned}
B(-1) &= 1\\
B(0) &= 0\\
B(1) &= 2\\
B(2) &= 2^2 = 4\\
B(3) &= 2^{2^2} = 2^4 = 16\\
B(4) &= 2^{2^{2^2}} = 2^{16} = 65536
\end{aligned}
$$

For Example

We have that

$$
\begin{aligned}
B(-1) &= 1 \\
B(0) &= 0 \\
B(1) &= 2 \\
B(2) &= 2^2 = 4 \\
B(3) &= 2^{2^2} = 2^4 = 16 \\
B(4) &= 2^{2^{2^2}} = 2^{16} = 65536
\end{aligned}
$$

For Example

Thus, we have

Block j	Set of Ranks
0	0,1
1	2
2	3,4
3	5,...,16
4	17,...,65536
$\vdots$	$\vdots$

Note $B(j) = 2^{B(j-1)}$ for $j > 0$.

For Example

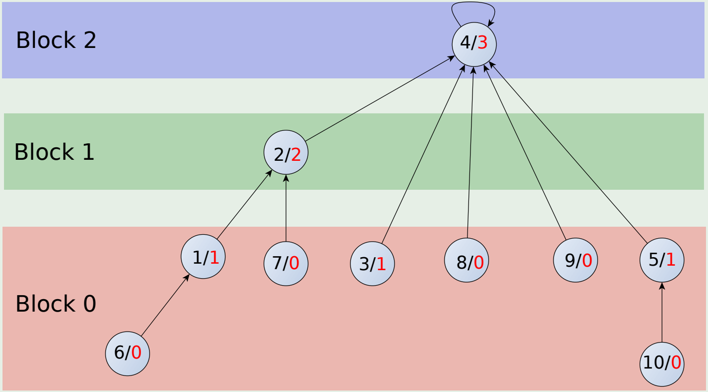
Thus, we have

Block j	Set of Ranks
0	0,1
1	2
2	3,4
3	5,...,16
4	17,...,65536
$\vdots$	$\vdots$

Note $B(j) = 2^{B(j-1)}$ for $j > 0$.

Example

Now you give a Union$(4, 5)$



Finally

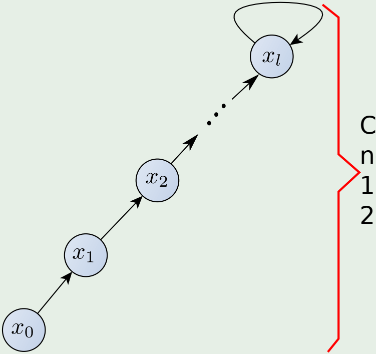
Given our Bound in the Ranks

Thus, all the blocks from $B\left(0\right)$ to $B\left(\log^* n - 1\right)$ will be used for storing the ranking elements

Charging for FindSets

Two types of charges for FindSet(x_0)

Block charges and Path charges.



Charge each node as either:
1) Block Charge
2) Path Charge

Charging for FindSets

Thus, for find sets

- The find operation pays for the work done for the root and its immediate child.
- It also pays for all the nodes which are not in the **same block as their parents**.

Charging for FindSets

Thus, for find sets

- The find operation pays for the work done for the root and its immediate child.
- It also pays for all the nodes which are not in the **same block as their parents**.

Then

First

1. All these nodes are children of some other nodes, so their ranks will not change and they are bound to stay in the same block until the end of the computation.
2. If a node is in the same block as its parent, it will be charged for the work done in the FindSet Operation!!!

Then

First

1. All these nodes are children of some other nodes, so their ranks will not change and they are bound to stay in the same block until the end of the computation.
2. If a node is in the same block as its parent, it will be charged for the work done in the FindSet Operation!!!

Thus

We have the following charges

- Block Charge :
  - For $j = 0, 1, ..., \log^* n - 1$, give one block charge to the **last node** with rank in block j on the path $x_0, x_1, ..., x_l$.
  - Also give one block charge to the child of the root, i.e., x_{l-1}, and the root itself, i.e., x_{l-1}.
- Path Charge :
  - Give nodes in $x_0, ..., x_l$ a path charge until they are moved to point to a name element with a rank different from the child's block

Thus

We have the following charges

- Block Charge :
  - For $j = 0, 1, ..., \log^* n - 1$, give one block charge to the **last node** with rank in block j on the path $x_0, x_1, ..., x_l$.
  - Also give one block charge to the child of the root, i.e., x_{l-1}, and the root itself, i.e., x_{l-1}.
- Path Charge :
  - Give nodes in $x_0, ..., x_l$ a path charge until they are moved to point to a name element with a rank different from the child's block

Thus

We have the following charges

- Block Charge :
  - For $j = 0, 1, ..., \log^* n - 1$, give one block charge to the **last node** with rank in block j on the path $x_0, x_1, ..., x_l$.
  - Also give one block charge to the child of the root, i.e., x_{l-1}, and the root itself, i.e., x_{l-1}.
- Path Charge :
  - Give nodes in $x_0, ..., x_l$ a path charge until they are moved to point to a name element with a rank different from the child's block

Thus

We have the following charges

- Block Charge :
  - For $j = 0, 1, ..., \log^* n - 1$, give one block charge to the **last node** with rank in block j on the path $x_0, x_1, ..., x_l$.
  - Also give one block charge to the child of the root, i.e., x_{l-1}, and the root itself, i.e., x_{l-1}.
- Path Charge :
  - Give nodes in $x_0, ..., x_l$ a path charge until they are moved to point to a name element with a rank different from the child's block

Thus

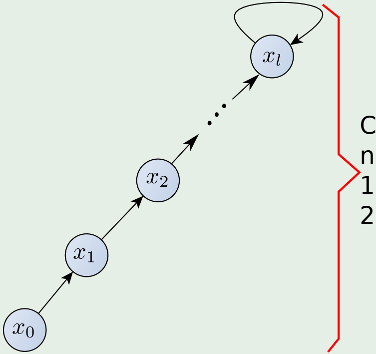
We have the following charges

- Block Charge :
    - For $j = 0, 1, ..., \log^* n - 1$, give one block charge to the **last node** with rank in block j on the path $x_0, x_1, ..., x_l$.
    - Also give one block charge to the child of the root, i.e., x_{l-1}, and the root itself, i.e., x_{l-1}.
- Path Charge :
    - Give nodes in $x_0, ..., x_l$ a path charge until they are moved to point to a name element with a rank different from the child's block

Charging for FindSets

Two types of charges for FindSet(x_0)

Block charges and Path charges.

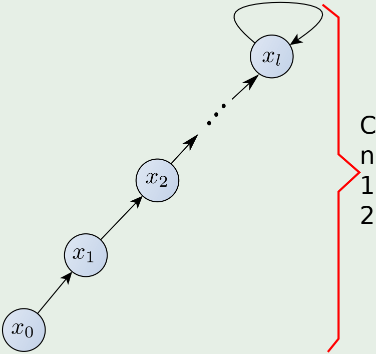


Charge each
node as either:
1) Block Charge
2) Path Charge

Charging for FindSets

Two types of charges for FindSet(x_0)

Block charges and Path charges.



Charge each node as either:
1) Block Charge
2) Path Charge

Next

Something Notable

- Number of nodes whose parents are in different blocks is limited by $(\log^* n) - 1$.
  - Making it an upper bound for the charges when changing the **last node** with rank in block j.
- 2 charges for the root and its child.

Next

Something Notable

- Number of nodes whose parents are in different blocks is limited by $(\log^* n) - 1$.
  - Making it an upper bound for the charges when changing the **last node** with rank in block j.
- 2 charges for the root and its child.

Thus

The cost of the Block Charges for the FindSet operation is upper bounded by:

$$\log^* n - 1 + 2 = \log^* n + 1. \tag{8}$$

Next

Something Notable

- Number of nodes whose parents are in different blocks is limited by $(\log^* n) - 1$.
    - Making it an upper bound for the charges when changing the **last node** with rank in block j.
- 2 charges for the root and its child.

Thus

The cost of the Block Charges for the FindSet operation is upper bounded by:

$$\log^* n - 1 + 2 = \log^* n + 1. \tag{8}$$

Next

Something Notable

- Number of nodes whose parents are in different blocks is limited by $(\log^* n) - 1$.
  - Making it an upper bound for the charges when changing the **last node** with rank in block j.
- 2 charges for the root and its child.

Thus

The cost of the Block Charges for the FindSet operation is upper bounded by:

$$\log^* n - 1 + 2 = \log^* n + 1. \tag{8}$$

Next

Something Notable

- Number of nodes whose parents are in different blocks is limited by $(\log^* n) - 1$.
  - Making it an upper bound for the charges when changing the **last node** with rank in block j.
- 2 charges for the root and its child.

Thus

The cost of the Block Charges for the FindSet operation is upper bounded by:

$$\log^* n - 1 + 2 = \log^* n + 1. \tag{8}$$

Claim

Claim

Once a node other than a root or its child is given a Block Charge (B.C.), it will never be given a Path Charge (P.C.)

Proof

Proof

Given a node x, we know that:

- $rank\left[p\left[x\right]\right] - rank\left[x\right]$ is monotonically increasing $\Rightarrow$ $\log^* rank\left[p\left[x\right]\right] - \log^* rank\left[x\right]$ is monotonically increasing.
- Thus, Once x and $p[x]$ are in different blocks, they will always be in different blocks because:
  - The rank of the parent can only increases.
  - And the child's rank stays the same
- Thus, the node x will be billed in the first FindSet operation a patch charge and block charge if necessary.
- Thus, the node x will never be charged again a path charge because is already pointing to the member set name.

Proof

Proof

Given a node x, we know that:

- $rank\left[p\left[x\right]\right] - rank\left[x\right]$ is monotonically increasing $\Rightarrow$ $\log^{*} rank\left[p\left[x\right]\right] - \log^{*} rank\left[x\right]$ is monotonically increasing.
- Thus, Once x and $p[x]$ are in different blocks, they will always be in different blocks because:
  - The rank of the parent can only increases.
  - And the child's rank stays the same
- Thus, the node x will be billed in the first FindSet operation a patch charge and block charge if necessary.
- Thus, the node x will never be charged again a path charge because is already pointing to the member set name.

Proof

Proof

Given a node x, we know that:

- $rank\left[p\left[x\right]\right] - rank\left[x\right]$ is monotonically increasing $\Rightarrow$ $\log^{*} rank\left[p\left[x\right]\right] - \log^{*} rank\left[x\right]$ is monotonically increasing.
- Thus, Once x and $p[x]$ are in different blocks, they will always be in different blocks because:
  - The rank of the parent can only increases.
  - And the child's rank stays the same
- Thus, the node x will be billed in the first FindSet operation a patch charge and block charge if necessary.
- Thus, the node x will never be charged again a path charge because is already pointing to the member set name.

Proof

Proof

Given a node x, we know that:

- $rank\left[p\left[x\right]\right] - rank\left[x\right]$ is monotonically increasing $\Rightarrow$ $\log^{*} rank\left[p\left[x\right]\right] - \log^{*} rank\left[x\right]$ is monotonically increasing.
- Thus, Once x and $p[x]$ are in different blocks, they will always be in different blocks because:
  - The rank of the parent can only increases.
  - And the child's rank stays the same
- Thus, the node x will be billed in the first FindSet operation a patch charge and block charge if necessary.
- Thus, the node x will never be charged again a path charge because is already pointing to the member set name.

Proof

Proof

Given a node x, we know that:

- $rank\left[p\left[x\right]\right] - rank\left[x\right]$ is monotonically increasing $\Rightarrow$ $\log^{*} rank\left[p\left[x\right]\right] - \log^{*} rank\left[x\right]$ is monotonically increasing.
- Thus, Once x and $p[x]$ are in different blocks, they will always be in different blocks because:
  - The rank of the parent can only increases.
  - And the child's rank stays the same
- Thus, the node x will be billed in the first FindSet operation a patch charge and block charge if necessary.
- Thus, the node x will never be charged again a path charge because is already pointing to the member set name.

Proof

Proof

Given a node x, we know that:

- $rank\left[p\left[x\right]\right] - rank\left[x\right]$ is monotonically increasing $\Rightarrow$ $\log^{*} rank\left[p\left[x\right]\right] - \log^{*} rank\left[x\right]$ is monotonically increasing.
- Thus, Once x and $p[x]$ are in different blocks, they will always be in different blocks because:
  - The rank of the parent can only increases.
  - And the child's rank stays the same
- Thus, the node x will be billed in the first FindSet operation a patch charge and block charge if necessary.
- Thus, the node x will never be charged again a path charge because is already pointing to the member set name.

Proof

Proof

Given a node x, we know that:

- $rank\left[p\left[x\right]\right] - rank\left[x\right]$ is monotonically increasing $\Rightarrow$ $\log^* rank\left[p\left[x\right]\right] - \log^* rank\left[x\right]$ is monotonically increasing.
- Thus, Once x and $p[x]$ are in different blocks, they will always be in different blocks because:
    - The rank of the parent can only increases.
    - And the child's rank stays the same
- Thus, the node x will be billed in the first FindSet operation a patch charge and block charge if necessary.
- Thus, the node x will never be charged again a path charge because is already pointing to the member set name.

Remaining Goal

The Total cost of the FindSet's Operations

Total cost of FindSet's = **Total Block Charges** + **Total Path Charges**.

We want to show

Total Block Charges + **Total Path Charges**$= O(m \log^* n)$

Remaining Goal

The Total cost of the FindSet's Operations

Total cost of FindSet's = **Total Block Charges** + **Total Path Charges**.

We want to show

Total Block Charges + **Total Path Charges**$= O(m \log^* n)$

Bounding Block Charges

This part is easy

Block numbers range over $0, ..., \log^* n - 1$.

- The number of Block Charges per FindSet is $\leq \log^* n + 1$.
- The total number of FindSet's is $\leq m$
- The total number of Block Charges is $\leq m(\log^* n + 1)$.

Bounding Block Charges

This part is easy

Block numbers range over $0, ..., \log^* n - 1$.

- The number of Block Charges per FindSet is $\leq \log^* n + 1$.
- The total number of FindSet's is $\leq m$
- The total number of Block Charges is $\leq m(\log^* n + 1)$.

Bounding Block Charges

This part is easy

Block numbers range over $0, ..., \log^* n - 1$.

- The number of Block Charges per FindSet is $\leq \log^* n + 1$.
- The total number of FindSet's is $\leq m$
- The total number of Block Charges is $\leq m(\log^* n + 1)$.

Bounding Block Charges

This part is easy

Block numbers range over $0, ..., \log^* n - 1$.

- The number of Block Charges per FindSet is $\leq \log^* n + 1$.
- The total number of FindSet's is $\leq m$
- The total number of Block Charges is $\leq m(\log^* n + 1)$.

Bounding Path Charges

Claim

Let $N(j)$ be the number of nodes whose ranks are in block j. Then, for all $j \geq 0$, $N(j) \leq \frac{3n}{2B(j)}$

Bounding Path Charges

Claim

Let $N(j)$ be the number of nodes whose ranks are in block j. Then, for all $j \geq 0$, $N(j) \leq \frac{3n}{2B(j)}$

Proof

- By Lemma 3, $N(j) \leq \sum\limits_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r}$ summing over all possible ranks
- For j = 0:

$$\begin{aligned} N(0) &\leq \frac{n}{2^0} + \frac{n}{2} \\ &= \frac{3n}{2} \\ &= \frac{3n}{2B(0)} \end{aligned}$$

Bounding Path Charges

Claim

Let $N(j)$ be the number of nodes whose ranks are in block j. Then, for all $j \geq 0$, $N(j) \leq \frac{3n}{2B(j)}$

Proof

- By Lemma 3, $N(j) \leq \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r}$ summing over all possible ranks
- For j = 0:

$$
\begin{aligned}
N(0) &\leq \frac{n}{2^0} + \frac{n}{2} \\
&= \frac{3n}{2} \\
&= \frac{3n}{2B(0)}
\end{aligned}
$$

Bounding Path Charges

Claim

Let $N(j)$ be the number of nodes whose ranks are in block j. Then, for all $j \geq 0$, $N(j) \leq \frac{3n}{2B(j)}$

Proof

- By Lemma 3, $N(j) \leq \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r}$ summing over all possible ranks
- For j = 0:

$$\begin{aligned} N(0) &\leq \frac{n}{2^0} + \frac{n}{2} \\ &= \frac{3n}{2} \\ &= \frac{3n}{2B(0)} \end{aligned}$$

Bounding Path Charges

Claim

Let $N(j)$ be the number of nodes whose ranks are in block j. Then, for all $j \geq 0$, $N(j) \leq \frac{3n}{2B(j)}$

Proof

- By Lemma 3, $N(j) \leq \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r}$ summing over all possible ranks
- For j = 0:

$$\begin{aligned} N(0) &\leq \frac{n}{2^0} + \frac{n}{2} \\ &= \frac{3n}{2} \\ &= \frac{3n}{2B(0)} \end{aligned}$$

Proof of claim

For $j \geq 1$

$$N(j) \leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r}$$

$$< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r}$$

$$= \frac{n}{2^{B(j-1)}}$$ This is where the fact that $B(j) = 2^{B(j-1)}$ is used.

$$= \frac{n}{B(j)}$$

$$< \frac{3n}{2B(j)}$$

Proof of claim

For $j \geq 1$

$$
\begin{aligned}
N(j) &\leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r} \\
&< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\
&= \frac{n}{2^{B(j-1)}} \quad \text{This is where the fact that } B(j) = 2^{B(j-1)} \text{ is used.} \\
&= \frac{n}{B(j)} \\
&< \frac{3n}{2B(j)}
\end{aligned}
$$

Proof of claim

For $j \geq 1$

$$
\begin{aligned}
N(j) &\leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r} \\
&< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\
&= \frac{n}{2^{B(j-1)}} \text{ This is where the fact that } B(j) = 2^{B(j-1)} \text{is used.} \\
&= \frac{n}{B(j)} \\
&< \frac{3n}{2B(j)}
\end{aligned}
$$

Proof of claim

For $j \geq 1$

$$\begin{aligned}
N(j) &\leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r} \\
&< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\
&= \frac{n}{2^{B(j-1)}} \text{ This is where the fact that } B(j) = 2^{B(j-1)} \text{is used.} \\
&= \frac{n}{B(j)} \\
&< \frac{3n}{2B(j)}
\end{aligned}$$

Proof of claim

For $j \geq 1$

$$\begin{aligned} N(j) &\leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r} \\ &< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\ &= \frac{n}{2^{B(j-1)}} \text{ This is where the fact that } B\left(j\right) = 2^{B(j-1)} \text{is used.} \\ &= \frac{n}{B(j)} \\ &< \frac{3n}{2B(j)} \end{aligned}$$

Bounding Path Charges

We have the following

- Let $P(n)$ denote the overall number of path charges. Then:

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \alpha_j \cdot \beta_j \tag{9}$$

  - α_j is the max number of nodes with ranks in Block j
  - β_j is the max number of path charges per node of Block j.

Bounding Path Charges

We have the following

- Let $P(n)$ denote the overall number of path charges. Then:

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \alpha_j \cdot \beta_j \tag{9}$$

  - α_j is the max number of nodes with ranks in Block j
  - β_j is the max number of path charges per node of Block j.

Bounding Path Charges

We have the following

- Let $P(n)$ denote the overall number of path charges. Then:

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \alpha_j \cdot \beta_j \tag{9}$$

  - α_j is the max number of nodes with ranks in Block j
  - β_j is the max number of path charges per node of Block j.

Then, we have the following

Upper Bounds

- By claim, α_j upper-bounded by $\frac{3n}{2B(j)}$,
- In addition, we need to bound β_j that represents the maximum number of path charges for nodes x at block j.

  Note: Any node in Block j that is given a P.C. will be in Block j after all m operations.

Then, we have the following

Upper Bounds

- By claim, α_j upper-bounded by $\frac{3n}{2B(j)}$,
- In addition, we need to bound β_j that represents the maximum number of path charges for nodes x at block j.

Note: Any node in Block j that is given a P.C. will be in Block j after all m operations.

Then, we have the following

Upper Bounds

- By claim, α_j upper-bounded by $\frac{3n}{2B(j)}$,
- In addition, we need to bound β_j that represents the maximum number of path charges for nodes x at block j.

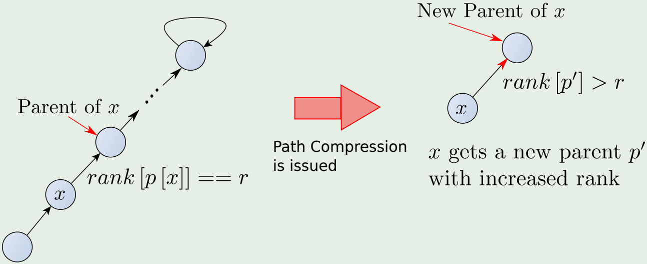
Note: Any node in Block j that is given a P.C. will be in Block j after all m operations.

Then, we have the following

Upper Bounds

- By claim, α_j upper-bounded by $\frac{3n}{2B(j)}$,
- In addition, we need to bound β_j that represents the maximum number of path charges for nodes x at block j.

Note: Any node in Block j that is given a P.C. will be in Block j after all m operations.



Now, we bound β_j

So, every time x is assessed a Path Charges, it gets a new parent with increased rank.

Note: x's rank **is not changed** by path compression

Now, we bound β_j

So, every time x is assessed a Path Charges, it gets a new parent with increased rank.

Note: x's rank **is not changed** by path compression

Suppose x has a rank in Block j

- Repeated Path Charges to x will ultimately result in x's parent having a rank in a Block higher than j.
- From that point onward, x is given Block Charges, not Path Charges.

Now, we bound β_j

So, every time x is assessed a Path Charges, it gets a new parent with increased rank.

Note: x's rank **is not changed** by path compression

Suppose x has a rank in Block j

- Repeated Path Charges to x will ultimately result in x's parent having a rank in a Block higher than j.
- From that point onward, x is given Block Charges, not Path Charges.

Therefore, the Worst Case

- x has the lowest rank in Block j, i.e., $B(j-1)+1$, and x's parents ranks successively take on the values.

$$B(j-1)+2, B(j-1)+3, ..., B(j)$$

Now, we bound β_j

So, every time x is assessed a Path Charges, it gets a new parent with increased rank.

Note: x's rank **is not changed** by path compression

Suppose x has a rank in Block j

- Repeated Path Charges to x will ultimately result in x's parent having a rank in a Block higher than j.
- From that point onward, x is given Block Charges, not Path Charges.

Therefore, the Worst Case

- x has the lowest rank in Block j, i.e., $B\left(j-1\right)+1$, and x's parents ranks successively take on the values.

$$B(j-1)+2, B(j-1)+3, \ldots, B(j)$$

Now, we bound β_j

So, every time x is assessed a Path Charges, it gets a new parent with increased rank.

Note: x's rank **is not changed** by path compression

Suppose x has a rank in Block j

- Repeated Path Charges to x will ultimately result in x's parent having a rank in a Block higher than j.
- From that point onward, x is given Block Charges, not Path Charges.

Therefore, the Worst Case

- x has the lowest rank in Block j, i.e., $B\left(j-1\right)+1$, and x's parents ranks successively take on the values.

$$B(j-1)+2, B(j-1)+3, ..., B(j)$$

Finally

- Hence, x can be given at most $B(j) - B(j-1) - 1$ Path Charges.
- Therefore:

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} \left(B(j) - B(j-1) - 1\right)$$

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} B(j)$$

$$P(n) = \frac{3}{2} n \log^* n$$

Finally

- Hence, x can be given at most $B(j) - B(j-1) - 1$ Path Charges.
- Therefore:

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} (B(j) - B(j-1) - 1)$$

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} B(j)$$

$$P(n) = \frac{3}{2} n \log^* n$$

Finally

- Hence, x can be given at most $B(j) - B(j-1) - 1$ Path Charges.
- Therefore:

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} \left(B(j) - B(j-1) - 1\right)$$

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} B(j)$$

$$P(n) = \frac{3}{2} n \log^* n$$

Finally

- Hence, x can be given at most $B(j) - B(j-1) - 1$ Path Charges.
- Therefore:

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} (B(j) - B(j-1) - 1)$$

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} B(j)$$

$$P(n) = \frac{3}{2} n \log^* n$$

Thus

FindSet operations contribute

$$O(m(\log^* n + 1) + n \log^* n) = O(m \log^* n) \quad (10)$$

MakeSet and Link contribute $O(n)$

Entire sequence takes $O\left(m \log^* n\right)$.

Thus

FindSet operations contribute

$$O(m(\log^* n + 1) + n \log^* n) = O(m \log^* n) \tag{10}$$

MakeSet and Link contribute $O(n)$

Entire sequence takes $O\left(m \log^* n\right)$**.**