Union-Find

Lecture 23
November 14, 2013
Part I

Union Find
Requirements from the data-structure

1. Maintain a collection of sets.
2. `makeSet(x)` - creates a set that contains the single element `x`.
3. `find(x)` - returns the set that contains `x`.
4. `union(A, B)` - returns set = union of `A` and `B`. That is `A \cup B`. ... merges the two sets `A` and `B` and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.
2. \texttt{makeSet}(x) - creates a set that contains the single element $x$.
3. \texttt{find}(x) - returns the set that contains $x$.
4. \texttt{union}(A, B) - returns set = union of $A$ and $B$. That is $A \cup B$.
   ... merges the two sets $A$ and $B$ and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.
2. **makeSet(x)** - creates a set that contains the single element \(x\).
3. **find(x)** - returns the set that contains \(x\).
4. **union(A, B)** - returns set = union of \(A\) and \(B\). That is \(A \cup B\). ... merges the two sets \(A\) and \(B\) and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.

2. `makeSet(x)` - creates a set that contains the single element $x$.

3. `find(x)` - returns the set that contains $x$.

4. `union(A, B)` - returns set $= \text{union of } A \text{ and } B$. That is $A \cup B$. ... merges the two sets $A$ and $B$ and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.
2. `makeSet(x)` - creates a set that contains the single element \( x \).
3. `find(x)` - returns the set that contains \( x \).
4. `union(A, B)` - returns set = union of \( A \) and \( B \). That is \( A \cup B \).

... merges the two sets \( A \) and \( B \) and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.
2. \texttt{makeSet}(x) - creates a set that contains the single element \( x \).
3. \texttt{find}(x) - returns the set that contains \( x \).
4. \texttt{union}(A, B) - returns set = union of \( A \) and \( B \). That is \( A \cup B \). ... merges the two sets \( A \) and \( B \) and return the merged set.
Amortized Analysis

1. Use data-structure as a black-box inside algorithm.
   ... Union-Find in Kruskal algorithm for computing MST.

2. Bounded worst case time per operation.

3. Care: overall running time spend in data-structure.

4. *Amortized running-time* of operation is the average time it takes to perform an operation on the data-structure.

5. Amortized time of an operation is \( \frac{\text{overall running time}}{\text{number of operations}} \).
Amortized Analysis

1. Use data-structure as a black-box inside algorithm. ... Union-Find in Kruskal algorithm for computing MST.
2. Bounded worst case time per operation.
3. Care: overall running time spend in data-structure.
4. *Amortized running-time* of operation is the average time it takes to perform an operation on the data-structure.
5. Amortized time of an operation is \( \frac{\text{overall running time}}{\text{number of operations}} \).
Amortized Analysis

1. Use data-structure as a black-box inside algorithm. ... Union-Find in Kruskal algorithm for computing MST.

2. Bounded worst case time per operation.

3. Care: overall running time spend in data-structure.

4. *Amortized running-time* of operation is the average time it takes to perform an operation on the data-structure.

5. Amortized time of an operation is \( \frac{\text{overall running time}}{\text{number of operations}} \).
Amortized Analysis

1. Use data-structure as a black-box inside algorithm. ... Union-Find in Kruskal algorithm for computing MST.

2. Bounded worst case time per operation.

3. Care: overall running time spend in data-structure.

4. *amortized running-time* of operation is the average time it takes to perform an operation on the data-structure.

5. Amortized time of an operation is \( \frac{\text{overall running time}}{\text{number of operations}} \).
Amortized Analysis

1. Use data-structure as a black-box inside algorithm. ... Union-Find in Kruskal algorithm for computing MST.

2. Bounded worst case time per operation.

3. Care: overall running time spend in data-structure.

4. **Amortized running-time** of operation is the average time it takes to perform an operation on the data-structure.

5. Amortized time of an operation is \( \frac{\text{overall running time}}{\text{number of operations}} \).
The Union-Find representation of the sets $A = \{a, b, c, d, e\}$ and $B = \{f, g, h, i, j, k\}$. The set $A$ is uniquely identified by a pointer to the root of $A$, which is the node containing $a$. 
Reversed Trees:

1. Every element is stored in its own node.
2. A node has one pointer to its parent.
3. A set is uniquely identified with the element stored in the root.
Reversed Trees:

1. Every element is stored in its own node.
2. A node has one pointer to its parent.
3. A set is uniquely identified with the element stored in the root.

**makeSet**: Create a singleton pointing to itself: $a$
Reversed Trees

Every element is stored in its own node.
A node has one pointer to its parent.
A set is uniquely identified with the element stored in the root.

makeSet: Create a singleton pointing to itself.

find(x): Start from node containing x, traverse up the tree (using parent pointers), till arriving to root.
Thus, doing a find(x) operation in the reversed tree shown on the right, involve going up the tree from x → b → a, and returning a as the set.
**Union operation in reversed trees**

*Just hang them on each other.*

---

**union** \((a, p)\): Merge two sets.

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

---

![Diagram showing the union operation on reversed trees](image)
Pseudo-code of naive version...

\[
\text{makeSet}(x) \\
\quad \overline{p}(x) \leftarrow x
\]

\[
\text{find}(x) \\
\quad \text{if } x = \overline{p}(x) \text{ then} \\
\quad \quad \text{return } x \\
\quad \text{return } \text{find}(\overline{p}(x))
\]

\[
\text{union}(x, y) \\
\quad A \leftarrow \text{find}(x) \\
\quad B \leftarrow \text{find}(y) \\
\quad \overline{p}(B) \leftarrow A
\]
Example...
The long chain

After: \( \text{makeSet}(a), \text{makeSet}(b), \text{makeSet}(c), \text{makeSet}(d), \text{makeSet}(e), \text{makeSet}(f), \text{makeSet}(g), \text{makeSet}(h) \)
Example...

The long chain

After: \texttt{makeSet}(a), \texttt{makeSet}(b), \texttt{makeSet}(c), \texttt{makeSet}(d), \texttt{makeSet}(e), \texttt{makeSet}(f), \texttt{makeSet}(g), \texttt{makeSet}(h) \\
\texttt{union}(g, h)
Example...

The long chain

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

\( \text{union}(g, h) \)

\( \text{union}(f, g) \)
Example...
The long chain

After:  \texttt{makeSet}(a), \texttt{makeSet}(b), \texttt{makeSet}(c), \texttt{makeSet}(d), \texttt{makeSet}(e), \texttt{makeSet}(f), \texttt{makeSet}(g), \texttt{makeSet}(h)  
\texttt{union}(g, h)  
\texttt{union}(f, g)  
\texttt{union}(e, f)
Example...
The long chain

After: \( \text{makeSet}(a), \text{makeSet}(b), \text{makeSet}(c), \text{makeSet}(d), \text{makeSet}(e), \text{makeSet}(f), \text{makeSet}(g), \text{makeSet}(h) \)
\( \text{union}(g, h) \)
\( \text{union}(f, g) \)
\( \text{union}(e, f) \)
\( \text{union}(d, e) \)
Example…

The long chain

After: \texttt{makeSet}(a), \texttt{makeSet}(b), \texttt{makeSet}(c), \texttt{makeSet}(d), \texttt{makeSet}(e), \texttt{makeSet}(f), \texttt{makeSet}(g), \texttt{makeSet}(h) \texttt{union}(g, h) \texttt{union}(f, g) \texttt{union}(e, f) \texttt{union}(d, e) \texttt{union}(c, d)
Example...

The long chain

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

union(g, h)
union(f, g)
union(e, f)
union(d, e)
union(c, d)
union(b, c)
Example...
The long chain

After: \textit{makeSet}(a), \textit{makeSet}(b), \textit{makeSet}(c), \textit{makeSet}(d), \textit{makeSet}(e), \textit{makeSet}(f), \textit{makeSet}(g), \textit{makeSet}(h)
union(g, h)
union(f, g)
union(e, f)
union(d, e)
union(c, d)
union(b, c)
union(a, b)
Find is slow, hack it!

\textbf{find} might require $\Omega(n)$ time. So, the question is how to further improve the performance of this data-structure. We are going to do this, by using two “hacks”:

(i) \textit{Union by rank}: Maintain for every tree, in the root, a bound on its depth (called \textit{rank}). Always hang the smaller tree on the larger tree.

(ii) \textit{Path compression}: Since, anyway, we travel the path to the root during a \textit{find} operation, we might as well hang all the nodes on the path directly on the root.
(a) The tree before performing $\text{find}(z)$, and (b) The reversed tree after performing $\text{find}(z)$ that uses path compression.
Pseudo-code of improved version...

**makeSet**(x)
- \( \overline{p}(x) \leftarrow x \)
- \( \text{rank}(x) \leftarrow 0 \)

**find**(x)
- if \( x \neq \overline{p}(x) \) then
  - \( \overline{p}(x) \leftarrow \text{find}(\overline{p}(x)) \)
- return \( \overline{p}(x) \)

**union**(x, y)
- \( A \leftarrow \text{find}(x) \)
- \( B \leftarrow \text{find}(y) \)
- if \( \text{rank}(A) > \text{rank}(B) \) then
  - \( \overline{p}(B) \leftarrow A \)
- else
  - \( \overline{p}(A) \leftarrow B \)
- if \( \text{rank}(A) = \text{rank}(B) \) then
  - \( \text{rank}(B) \leftarrow \text{rank}(B) + 1 \)
Part II

Analyzing the Union-Find Data-Structure
A node in the union-find data-structure is a \textit{leader} if it is the root of a (reversed) tree.
Lemma

Once a node stop being a leader (i.e., the node in top of a tree), it can never become a leader again.

Proof.

Note, that an element $x$ can stop being a leader only because of a union operation that hanged $x$ on an element $y$. From this point on, the only operation that might change $x$ parent pointer, is a find operation that traverses through $x$. Since path-compression can only change the parent pointer of $x$ to point to some other element $y$, it follows that $x$ parent pointer will never become equal to $x$ again. Namely, once $x$ stop being a leader, it can never be a leader again.
Lemma

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

Proof.

The rank of an element changes only by the \texttt{union} operation. However, the \texttt{union} operation changes the rank, only for elements that are leader after the operation is done. As such, if an element is no longer a leader, than its rank is fixed.
Ranks are strictly monotonically increasing

Lemma

Ranks are monotonically increasing in the reversed trees, as we travel from a node to the root of the tree.
Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).


Assume claim holds at time \( t \), before an operation.

If operation is \textbf{union} \((A, B)\), and assume that we hanged \text{root}(A) on \text{root}(B).

Must be that \( \text{rank(root}(B)) \) is now larger than \( \text{rank(root}(A)) \) (verify!).

Claim true after operation!

If operation \textbf{find}: traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).

By induction, \( \text{rank}(v) \geq \text{rank of all other nodes of } \pi \).

All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
1. Claim: $\forall u \rightarrow v$ in DS: $\text{rank}(u) < \text{rank}(v)$.


3. Assume claim holds at time $t$, before an operation.

4. If operation is $\text{union}(A, B)$, and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$.
   Must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$ (verify!).
   Claim true after operation!

5. If operation $\text{find}$: traverse path $\pi$, then all the nodes of $\pi$ are made to point to the last node $v$ of $\pi$.
   By induction, $\text{rank}(v) >$ rank of all other nodes of $\pi$.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).


Assume claim holds at time \( t \), before an operation.

If operation is \texttt{union}(A, B), and assume that we hanged \( \text{root}(A) \) on \( \text{root}(B) \). Must be that \( \text{rank}(
\text{root}(B)) \) is now larger than \( \text{rank}(
\text{root}(A)) \) (verify!). Claim true after operation!

If operation \texttt{find}: traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \). By induction, \( \text{rank}(v) > \text{rank} \) of all other nodes of \( \pi \). All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Claim: $\forall u \rightarrow v$ in DS: $\text{rank}(u) < \text{rank}(v)$.


Assume claim holds at time $t$, before an operation.

If operation is $\text{union} (A, B)$, and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$.

Must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$ (verify!).
Claim true after operation!

If operation $\text{find}$: traverse path $\pi$, then all the nodes of $\pi$ are made to point to the last node $v$ of $\pi$.
By induction, $\text{rank}(v) > \text{rank}$ of all other nodes of $\pi$.
All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Claim: $\forall u \rightarrow v$ in DS: $\text{rank}(u) < \text{rank}(v)$.


Assume claim holds at time $t$, before an operation.

If operation is $\text{union}(A, B)$, and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$. Must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$ (verify!).

Claim true after operation!

If operation $\text{find}$: traverse path $\pi$, then all the nodes of $\pi$ are made to point to the last node $v$ of $\pi$. By induction, $\text{rank}(v) > \text{rank}$ of all other nodes of $\pi$. All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Claim: $\forall u \rightarrow v \text{ in DS: } \text{rank}(u) < \text{rank}(v)$.


Assume claim holds at time $t$, before an operation.

If operation is $\text{union } (A, B)$, and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$.
Must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$ (verify!).
Claim true after operation!

If operation $\text{find}$: traverse path $\pi$, then all the nodes of $\pi$ are made to point to the last node $v$ of $\pi$.
By induction, $\text{rank}(v) > \text{rank}$ of all other nodes of $\pi$.
All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).


Assume claim holds at time \( t \), before an operation.

If operation is \textbf{union} \((A, B)\), and assume that we hanged \( \text{root}(A) \) on \( \text{root}(B) \).
Must be that \( \text{rank}(\text{root}(B)) \) is now larger than \( \text{rank}(\text{root}(A)) \) (verify!).
Claim true after operation!

If operation \textbf{find}: traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).
By induction, \( \text{rank}(v) > \text{rank} \) of all other nodes of \( \pi \).
All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Proof...

1. Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).


3. Assume claim holds at time \( t \), before an operation.

4. If operation is \textbf{union} \((A, B)\), and assume that we hanged \( \text{root}(A) \) on \( \text{root}(B) \).
Must be that \( \text{rank}([\text{root}(B)]) \) is now larger than \( \text{rank}(\text{root}(A)) \) (verify!).
Claim true after operation!

5. If operation \textbf{find}: traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).
By induction, \( \text{rank}(v) > \) rank of all other nodes of \( \pi \).
All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Proof...

1. Claim: \( \forall u \rightarrow v \text{ in DS}: \text{rank}(u) < \text{rank}(v) \).
3. Assume claim holds at time \( t \), before an operation.
4. If operation is union \((A, B)\), and assume that we hanged root\((A)\) on root\((B)\).
   Must be that \( \text{rank}(\text{root}(B)) \) is now larger than \( \text{rank}(\text{root}(A)) \) (verify!).
   Claim true after operation!
5. If operation find: traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).
   By induction, \( \text{rank}(v) > \text{rank} \) of all other nodes of \( \pi \).
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Lemma

When a node gets rank $k$ than there are at least $\geq 2^k$ elements in its subtree.

Proof.

The proof is by induction. For $k = 0$ it is obvious since a singleton has a rank zero, and a single element in the set. Next observe that a node gets rank $k$ only if the merged two roots has rank $k - 1$. By induction, they have $2^{k-1}$ nodes (each one of them), and thus the merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.
Having higher rank is rare

Lemma

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

Proof.

Again, by induction. For \( k = 0 \) it is obvious.

Charge a node \( v \) of rank \( k \) to two elements \( u \) and \( v \) of rank \( k - 1 \) that were leaders used to create new larger set.

After the merge \( v \) is of rank \( k \) and \( u \) is of rank \( k - 1 \) and it is no longer a leader. Thus, we can charge this event to the two (no longer active) nodes of degree \( k - 1 \). Namely, \( u \) and \( v \).

By induction: algorithm created at most \( n/2^{k-1} \) nodes of rank \( k - 1 \) \( \implies \) \# nodes of rank \( k \) created by algorithm is \( \leq \left( \frac{n}{2^{k-1}} \right) / 2 = \frac{n}{2^k} \).
Having higher rank is rare

**Lemma**

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

**Proof.**

Again, by induction. For $k = 0$ it is obvious.

Charge a node $v$ of rank $k$ to two elements $u$ and $v$ of rank $k - 1$ that were leaders used to create new larger set.

After the merge $v$ is of rank $k$ and $u$ is of rank $k - 1$ and it is no longer a leader. Thus, we can charge this event to the two (no longer active) nodes of degree $k - 1$. Namely, $u$ and $v$.

By induction: algorithm created at most $n/2^{k-1}$ nodes of rank $k - 1$ $\implies$ # nodes of rank $k$ created by algorithm is $\leq (n/2^{k-1}) / 2 = n/2^k$. 

$\square$
Having higher rank is rare

Lemma

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

Proof.

Again, by induction. For $k = 0$ it is obvious. Charge a node $v$ of rank $k$ to two elements $u$ and $v$ of rank $k - 1$ that were leaders used to create new larger set. After the merge $v$ is of rank $k$ and $u$ is of rank $k - 1$ and it is no longer a leader. Thus, we can charge this event to the two (no longer active) nodes of degree $k - 1$. Namely, $u$ and $v$.

By induction: algorithm created at most $n/2^{k-1}$ nodes of rank $k - 1 \implies \# nodes of rank $k$ created by algorithm is $\leq \left(\frac{n}{2^{k-1}}\right)/2 = n/2^k$. \qed
Find takes logarithmic time

Lemma

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

Proof.

The rank of the leader $v$ of a reversed tree $T$, bounds the depth of a tree $T$ in the Union-Find data-structure. By the above lemma, if we have $n$ elements, the maximum rank is $\log n$ and thus the depth of a tree is at most $O(\log n)$. □
**log**∗ in detail

**log**∗**(n)**: number of times one has to take **lg** of a number to get a number smaller than two.

Thus, **log**∗ 2 = 1 and **log**∗ 2^2 = 2. Similarly, **log**∗ 2^{2^2} = 1 + **log**∗(2^2) = 2 + **log**∗ 2 = 3. Similarly, **log**∗ 2^{2^{2^2}} = **log**∗(65536) = 4.

Things get really exciting, when one considers

\[ \text{log}^* 2^{2^{2^2}} = \text{log}^* 2^{65536} = 5. \]

However, **log**∗ is a monotone increasing function. And

\[ \beta = 2^{2^{2^2}} = 2^{65536} \] is a huge number (considerably larger than the number of atoms in the universe). Thus, for all practical purposes, **log**∗ returns a value which is smaller than 5.
Can do much better!

**Theorem**

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

1. Intuitively: (in the amortized sense) Union-Find data-structure takes constant time per operation (unless $n$ is larger than $\beta$ which is unlikely).

2. Not quite correct if $n$ sufficiently large...
Theorem

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

1. Intuitively: (in the amortized sense) Union-Find data-structure takes constant time per operation (unless \( n \) is larger than \( \beta \) which is unlikely).

2. Not quite correct if \( n \) sufficiently large...
Can do much better!

**Theorem**

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

1. **Intuitively:** (in the amortized sense) Union-Find data-structure takes constant time per operation (unless \( n \) is larger than \( \beta \) which is unlikely).

2. **Not quite correct if** \( n \) **sufficiently large...**
The tower function...

**Definition**

\[
\text{Tower}(b) = 2^{\text{Tower}(b-1)} \quad \text{and} \quad \text{Tower}(0) = 1.
\]

\(\text{Tower}(i)\): a tower of \(2^{2^{2^{\ldots^2}}}\) of height \(i\).

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

**Definition**

For \(i \geq 0\), let \(\text{Block}(i) = [\text{Tower}(i-1) + 1, \text{Tower}(i)]\); that is

\[
\text{Block}(i) = [z, 2^{z-1}]
\]

for \(z = \text{Tower}(i-1) + 1\).

Also \(\text{Block}(0) = [0, 1]\). As such,

\[
\begin{align*}
\text{Block}(0) &= [0, 1], \\
\text{Block}(1) &= [2, 2], \\
\text{Block}(2) &= [3, 4], \\
\text{Block}(3) &= [5, 16], \\
\text{Block}(4) &= [17, 65536], \\
\text{Block}(5) &= [65537, 2^{65536}] \ldots
\end{align*}
\]
The tower function...

Definition

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

**Tower**\((i)\): a tower of \(2^{2^{2^{\ldots^2}}}\) of height \(i\).

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

Definition

For \(i \geq 0\), let \(\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]\); that is

\[ \text{Block}(i) = [z, 2^{z-1}] \quad \text{for} \quad z = \text{Tower}(i - 1) + 1. \]

Also \(\text{Block}(0) = [0, 1]\). As such,

- \(\text{Block}(0) = [0, 1]\), \(\text{Block}(1) = [2, 2]\), \(\text{Block}(2) = [3, 4]\),
- \(\text{Block}(3) = [5, 16]\), \(\text{Block}(4) = [17, 65536]\),
- \(\text{Block}(5) = [65537, 2^{65536}]\) ...
The tower function...

**Definition**

\[
\text{Tower}(b) = 2^{\text{Tower}(b-1)} \quad \text{and} \quad \text{Tower}(0) = 1.
\]

\textbf{Tower}(i): a tower of } 2^{2^{2^{\ldots^2}}} \text{ of height } i.

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

**Definition**

For \( i \geq 0 \), let \( \text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)] \); that is

\[
\text{Block}(i) = [z, 2^{z-1}] \quad \text{for} \quad z = \text{Tower}(i - 1) + 1.
\]

Also \( \text{Block}(0) = [0, 1] \). As such,

\[
\begin{align*}
\text{Block}(0) &= [0, 1], \\
\text{Block}(1) &= [2, 2], \\
\text{Block}(2) &= [3, 4], \\
\text{Block}(3) &= [5, 16], \\
\text{Block}(4) &= [17, 65536], \\
\text{Block}(5) &= [65537, 2^{65536}] \ldots
\end{align*}
\]
The tower function...

**Definition**

\[
\text{Tower}(b) = 2^{\text{Tower}(b-1)} \quad \text{and} \quad \text{Tower}(0) = 1.
\]

\text{Tower}(i): a tower of \( 2^{2^{2\ldots^2}} \) of height \( i \).

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

**Definition**

For \( i \geq 0 \), let \( \text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)] \); that is

\[
\text{Block}(i) = [z, 2^{z-1}]
\]

for \( z = \text{Tower}(i - 1) + 1 \).

Also \( \text{Block}(0) = [0, 1] \). As such,

\[
\begin{align*}
\text{Block}(0) &= [0, 1], & \text{Block}(1) &= [2, 2], & \text{Block}(2) &= [3, 4], \\
\text{Block}(3) &= [5, 16], & \text{Block}(4) &= [17, 65536], \\
\text{Block}(5) &= [65537, 2^{65536}] \ldots
\end{align*}
\]
The tower function...

**Definition**

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

**Tower**\((i)\): a tower of \(2^{2^{\ldots^{2}}} \) of height \(i\).

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

**Definition**

For \(i \geq 0\), let \(\text{Block}(i) = [\text{Tower}(i-1) + 1, \text{Tower}(i)]\); that is \(\text{Block}(i) = [z, 2^{z-1}] \) for \(z = \text{Tower}(i-1) + 1\).

Also \(\text{Block}(0) = [0, 1]\). As such, \(\text{Block}(0) = [0, 1], \text{Block}(1) = [2, 2], \text{Block}(2) = [3, 4], \text{Block}(3) = [5, 16], \text{Block}(4) = [17, 65536], \text{Block}(5) = [65537, 2^{65536}] \ldots\)
The tower function...

Definition

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

\[ \text{Tower}(i) \]: a tower of \( 2^{2^{\cdot^{\cdot^{2}}}} \) of height \( i \).

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

Definition

For \( i \geq 0 \), let \( \text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)] \); that is

\[ \text{Block}(i) = [z, 2^{z-1}] \]

for \( z = \text{Tower}(i - 1) + 1 \).

Also \( \text{Block}(0) = [0, 1] \). As such,

\[ \text{Block}(0) = [0, 1], \text{ Block}(1) = [2, 2], \text{ Block}(2) = [3, 4], \]

\[ \text{Block}(3) = [5, 16], \text{ Block}(4) = [17, 65536], \]

\[ \text{Block}(5) = [65537, 2^{65536}] \ldots \]
The tower function...

**Definition**

\[
\text{Tower}(b) = 2^{\text{Tower}(b-1)} \quad \text{and} \quad \text{Tower}(0) = 1.
\]

**Tower** \((i)\): a tower of \(2^{2^{\ldots^2}}\) of height \(i\).

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

**Definition**

For \(i \geq 0\), let \(\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]\); that is

\[
\text{Block}(i) = \left[z, 2^{z-1}\right] \quad \text{for} \quad z = \text{Tower}(i - 1) + 1.
\]

Also \(\text{Block}(0) = [0, 1]\). As such,

\[
\begin{align*}
\text{Block}(0) &= [0, 1], \\
\text{Block}(1) &= [2, 2], \\
\text{Block}(2) &= [3, 4], \\
\text{Block}(3) &= [5, 16], \\
\text{Block}(4) &= [17, 65536], \\
\text{Block}(5) &= [65537, 2^{65536}] 
\end{align*}
\]
The tower function...

**Definition**

\[
\text{Tower}(b) = 2^{\text{Tower}(b-1)} \quad \text{and} \quad \text{Tower}(0) = 1.
\]

**Tower**\((i)\): a tower of \(2^{2^{\cdots 2}}\) of height \(i\).

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

**Definition**

For \(i \geq 0\), let \(\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]\); that is \(\text{Block}(i) = [z, 2^{z-1}]\) for \(z = \text{Tower}(i - 1) + 1\).

Also \(\text{Block}(0) = [0, 1]\). As such,

\(\text{Block}(0) = [0, 1], \text{Block}(1) = [2, 2], \text{Block}(2) = [3, 4], \text{Block}(3) = [5, 16], \text{Block}(4) = [17, 65536], \text{Block}(5) = [65537, 2^{65536}] \ldots\)
The tower function...

**Definition**

\[
\text{Tower}(b) = 2^{\text{Tower}(b-1)} \quad \text{and} \quad \text{Tower}(0) = 1.
\]

\[
\text{Tower}(i): \text{ a tower of } 2^{2^{2^{\ldots 2}}} \text{ of height } i.
\]

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

**Definition**

For \( i \geq 0 \), let \( \text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)] \); that is

\[
\text{Block}(i) = [z, 2^{z-1}] \quad \text{for} \quad z = \text{Tower}(i - 1) + 1.
\]

Also \( \text{Block}(0) = [0, 1] \). As such,

- \( \text{Block}(0) = [0, 1] \), \( \text{Block}(1) = [2, 2] \), \( \text{Block}(2) = [3, 4] \),
- \( \text{Block}(3) = [5, 16] \), \( \text{Block}(4) = [17, 65536] \),
- \( \text{Block}(5) = [65537, 2^{65536}] \) ...
Running time of find...

1. RT of \textbf{find}(x) proportional to length of the path from \( x \) to the root of its tree.

2. ...since start from \( x \) and we visit the sequence:
   
   \[
   x_1 = x, \ x_2 = \overline{p}(x) = \overline{p}(x_1), \ldots,
   \]
   
   \[
   x_i = \overline{p}(x_{i-1}), \ldots, \ x_m = \text{root of tree}.
   \]

3. \( \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m) \).

4. RT of \textbf{find}(x) is \( O(m) \).

**Definition**

A node \( x \) is in the \( i \)th block if \( \text{rank}(x) \in \text{Block}(i) \).

5. Looking for ways to pay for the \textbf{find} operation.
   (other two operations take constant time).
Running time of find...

1. RT of \textbf{find}(x) proportional to length of the path from \textit{x} to the root of its tree.

2. ...since start from \textit{x} and we visit the sequence:
   \[
   x_1 = x, \quad x_2 = \overline{p}(x) = \overline{p}(x_1), \ldots, \\
   x_i = \overline{p}(x_{i-1}), \ldots, x_m = \text{root of tree}.
   \]

3. \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m).

4. RT of \textbf{find}(x) is \(O(m)\).

Definition

A node \(x\) is in the \(i\)th block if \(\text{rank}(x) \in \text{Block}(i)\).

5. Looking for ways to pay for the \textbf{find} operation. (other two operations take constant time).
Running time of find...

1. RT of \textbf{find}(x) proportional to length of the path from \( x \) to the root of its tree.

2. \ldots since start from \( x \) and we visit the sequence:
   \[ x_1 = x, x_2 = \overline{p}(x) = \overline{p}(x_1), \ldots, \]
   \[ x_i = \overline{p}(x_{i-1}), \ldots, x_m = \text{root of tree}. \]

3. \( \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m) \).

4. RT of \textbf{find}(x) is \( O(m) \).

\bigskip

\textbf{Definition}

A node \( x \) is in the \( i \)th block if \( \text{rank}(x) \in \text{Block}(i) \).

5. Looking for ways to pay for the \textbf{find} operation. (other two operations take constant time).
Running time of find...

1. RT of \texttt{find}(x) proportional to length of the path from \(x\) to the root of its tree.

2. ...since start from \(x\) and we visit the sequence:
   \[
   x_1 = x, \ x_2 = \mathcal{P}(x) = \mathcal{P}(x_1), \ldots , \ x_i = \mathcal{P}(x_{i-1}), \ldots , \ x_m = \text{root of tree}.
   \]

3. \(\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)\).

4. RT of \texttt{find}(x) is \(O(m)\).

Definition

A node \(x\) is in the \(i\)th block if \(\text{rank}(x) \in \text{Block}(i)\).

5. Looking for ways to pay for the \texttt{find} operation. (other two operations take constant time).
1. RT of \texttt{find}(x) proportional to length of the path from \(x\) to the root of its tree.

2. ...since start from \(x\) and we visit the sequence:
   \[ x_1 = x, x_2 = \overline{p}(x) = \overline{p}(x_1), \ldots, \]
   \[ x_i = \overline{p}(x_{i-1}), \ldots, x_m = \text{root of tree}. \]

3. \(\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)\).

4. RT of \texttt{find}(x) is \(O(m)\).

**Definition**

A node \(x\) is in the \(i\)th block if \(\text{rank}(x) \in \text{Block}(i)\).

5. Looking for ways to pay for the \texttt{find} operation. (other two operations take constant time).
Running time of find...

1. RT of \textit{find}(x) proportional to length of the path from \textit{x} to the root of its tree.

2. ...since start from \textit{x} and we visit the sequence:
   \begin{align*}
   x_1 &= x, x_2 = \overline{p}(x) = \overline{p}(x_1), \
   x_i &= \overline{p}(x_{i-1}), \ldots, x_m &= \text{root of tree}.
   \end{align*}

3. \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m).

4. RT of \textit{find}(x) is $O(m)$.

Definition

A node \textit{x} is in the \textit{i}th block if \text{rank}(x) $\in$ \text{Block}(i).

5. Looking for ways to pay for the \textit{find} operation. (other two operations take constant time).
Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.

2. # of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant).

3. find ($x$): $\pi$ path used.

4. partition $\pi$ into each by rank.

5. Price of find length $\pi$.

6. For node $x$: $\nu = \text{index}_B(x)$ index of block containing $\text{rank}(x)$. $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$.

7. $\text{index}_B(x)$: block of $x$
Blocks and jumping pointers

1. Maximum rank of node $v$ is $O(\log n)$.
2. Number of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant).
3. $\text{find}(x)$: $\pi$ path used.
4. Partition $\pi$ into each by rank.
5. Price of $\text{find}$ length $\pi$.
6. For node $x$: $\nu = \text{index}_B(x)$ index of block containing $\text{rank}(x)$. $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$.
7. $\text{index}_B(x)$: block of $x$
Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.
2. # of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant).
3. **find** ($x$): $\pi$ path used.
4. partition $\pi$ into each by rank.
5. Price of **find** length $\pi$.
6. For node $x$: $\nu = \text{index}_B(x)$ index of block containing $\text{rank}(x)$. $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$.
7. $\text{index}_B(x)$: block of $x$
Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.
2. \# of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant).
3. \textbf{find} ($x$): $\pi$ path used.
4. partition $\pi$ into each by rank.
5. Price of \textbf{find} length $\pi$.
6. For node $x$: $\nu = \text{index}_B(x)$ index of block containing rank($x$). rank($x$) $\in$ Block(index$_B(x)$).
7. index$_B(x)$: \textit{block of} $x$
Blocks and jumping pointers

1. Maximum rank of node $v$ is $O(\log n)$.
2. Number of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant).
3. \texttt{find}(x): $\pi$ path used.
4. Partition $\pi$ into each by rank.
5. Price of \texttt{find} length $\pi$.
6. For node $x$: $\nu = \text{index}_B(x)$ index of block containing $\text{rank}(x)$. $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$.
7. $\text{index}_B(x)$: block of $x$
Blocks and jumping pointers

1. Maximum rank of node $v$ is $O(\log n)$.
2. Number of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant).
3. **find** ($x$): $\pi$ path used.
4. Partition $\pi$ into each by rank.
5. Price of **find** length $\pi$.
6. For node $x$: $\nu = \text{index}_B(x)$ index of block containing $\text{rank}(x)$. $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$.
7. $\text{index}_B(x)$: *block of* $x$
Blocks and jumping pointers

1. maximum rank of node \( v \) is \( \mathcal{O}(\log n) \).
2. \# of blocks is \( \mathcal{O}(\log^* n) \), as \( \mathcal{O}(\log n) \in \text{Block}(c \log^* n) \), (\( c \): constant).
3. \textbf{find} \( (x) \): \( \pi \) path used.
4. partition \( \pi \) into each by rank.
5. Price of \textbf{find} length \( \pi \).
6. For node \( x \): \( \nu = \text{index}_B(x) \) index of block containing \( \text{rank}(x) \). \( \text{rank}(x) \in \text{Block}(\text{index}_B(x)) \).
7. \text{index}_B(x): block of \( x \)
The path of find operation, and its pointers
The pointers between blocks...

1. During a **find** operation...
2. $\pi$: path traversed.
3. Ranks of the nodes visited in $\pi$ monotone increasing.
4. Once leave block $i$th, never go back!
5. charge visit to nodes in $\pi$ next to element in a different block...
6. to total number of blocks $\leq O(\log^* n)$. 
The pointers between blocks...

1. During a **find** operation...
2. $\pi$: path traversed.
3. Ranks of the nodes visited in $\pi$ monotone increasing.
4. Once leave block $i$th, never go back!
5. charge visit to nodes in $\pi$ next to element in a different block...
6. to total number of blocks $\leq O(\log^* n)$. 
The pointers between blocks...

1. During a **find** operation...

2. $\pi$: path traversed.

3. Ranks of the nodes visited in $\pi$ monotone increasing.

4. Once leave block $i$th, never go back!

5. charge visit to nodes in $\pi$ next to element in a different block...

6. to total number of blocks $\leq O(\log^* n)$. 
The pointers between blocks...

1. During a **find** operation...
2. $\pi$: path traversed.
3. Ranks of the nodes visited in $\pi$ monotone increasing.
4. Once leave block $i$th, never go back!
5. charge visit to nodes in $\pi$ next to element in a different block...
6. to total number of blocks $\leq O(\log^* n)$. 
Jumping pointers

**Definition**

\( \pi \): path traversed by \texttt{find}.

\( x \in \pi, \overline{p(x)} \) is in a different block, is a **jump between blocks**. Jump inside a block is an **internal jump** (i.e., \( x \) and \( \overline{p(x)} \) are in same block).
Not too many jumps between blocks

Lemma

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

Proof.

Consider the search path $\pi = x_1, \ldots, x_m$, and consider the list of numbers $0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m)$. We have that $\text{index}_B(x_m) = O(\log^* n)$. As such, the number of elements $x$ in $\pi$ such that $\text{index}_B(x) \neq \text{index}_B(\bar{p}(x))$ is at most $O(\log^* n)$. 

\hfill $\square$
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.

2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.

3. `find` passes through $x$.

4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before `find` operation.

5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after `find` operation.

6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.

7. $\quad \quad \Rightarrow$ parent pointer of $x$ jump forward and the new parent has higher rank.

8. Charge internal block jumps to this “progress”.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.

2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.

3. **find** passes through $x$.

4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before **find** operation.

5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after **find** operation.

6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.

7. $\iff$ parent pointer of $x$ jump forward and the new parent has higher rank.

8. Charge internal block jumps to this “progress”.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. **find** passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before **find** operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after **find** operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
8. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
9. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
10. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
11. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
12. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
13. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
14. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
15. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
16. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
17. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
18. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
19. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
20. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
21. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
22. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
23. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
24. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
25. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
26. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
27. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
28. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
29. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
30. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
31. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
32. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
33. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
34. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
35. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
36. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. \texttt{find} passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before \texttt{find} operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after \texttt{find} operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\Rightarrow$ parent pointer of $x$ jump forward and the new parent has higher rank.
8. Charge internal block jumps to this “progress”.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. \textbf{find} passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before \textbf{find} operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after \textbf{find} operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\implies$ parent pointer of $x$ jump forward and the new parent has higher rank.
8. Charge internal block jumps to this “progress”.

Charge internal block jumps to this “progress”.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in the same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. `find` passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before `find` operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after `find` operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\Longrightarrow$ parent pointer of $x$ jumps forward and the new parent has a higher rank.
8. Charge internal block jumps to this “progress”.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. find passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before find operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after find operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\implies$ parent pointer of $x$ jump forward and the new parent has higher rank.
8. Charge internal block jumps to this “progress”.
Your parent can be promoted only a few times before leaving block

**Lemma**

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

**Proof.**

By above discussion, the parent of \(x\) increases its rank every-time an internal jump goes through \(x\). Since there at most \(|\text{Block}(i)|\) different values in the \(i\)th block, the claim follows. The inequality \(|\text{Block}(i)| \leq \text{Tower}(i)|\) holds by definition:

\[
\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]
\]
Few elements are in the bigger blocks

Lemma

There are at most $\frac{n}{\text{Tower}(i)}$ nodes that have ranks in the $i$th block throughout the algorithm execution.

Proof.

By lemma, the number of elements with rank in the $i$th block

\[
\leq \sum_{k \in \text{Block}(i)} \frac{n}{2^k} = \sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{n}{2^k} = n \cdot \sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\text{Tower}(i-1)}} = \frac{n}{\text{Tower}(i)}.
\]
The total number of internal jumps is $O(n)$.

**Lemma**

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

**Proof.**

An element $x$ in the $i$th block, can have at most $|\text{Block}(i)|$ internal jumps, before all jumps through $x$ are jumps between blocks, by lemma... There are at most $n/Tower(i)$ elements with ranks in the $i$th block, throughout the algorithm execution, by other Lemma. Thus, the total number of internal jumps is

$$|\text{Block}(i)| \cdot \frac{n}{Tower(i)} \leq Tower(i) \cdot \frac{n}{Tower(i)} = n.$$
Total number of internal jumps is $O(n)$

**Lemma**

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

**Proof.**

An element $x$ in the $i$th block, can have at most $|\text{Block}(i)|$ internal jumps, before all jumps through $x$ are jumps between blocks, by lemma... There are at most $n/Tower(i)$ elements with ranks in the $i$th block, throughout the algorithm execution, by other Lemma. Thus, the total number of internal jumps is

$$|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n.$$
Total number of internal jumps is $O(n)$

**Lemma**

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

**Proof.**

An element $x$ in the $i$th block, can have at most $|\text{Block}(i)|$ internal jumps, before all jumps through $x$ are jumps between blocks, by lemma... There are at most $n / \text{Tower}(i)$ elements with ranks in the $i$th block, throughout the algorithm execution, by other Lemma. Thus, the total number of internal jumps is

\[
|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n.
\]
Total number of internal jumps

**Lemma**

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

**Proof.**

Every internal jump can be associated with the block it is being performed in. Every block contributes $O(n)$ internal jumps throughout the execution of the union-find data-structures, by Lemma 16. There are $O(\log^* n)$ blocks. As such there are at most $O(n \log^* n)$ internal jumps.
Lemma

The overall time spent on $m$ find operations, throughout the lifetime of a union-find data-structure defined over $n$ elements, is $O((m + n) \log^* n)$.

Theorem

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

The overall time spent on $m$ find operations, throughout the lifetime of a union-find data-structure defined over $n$ elements, is $O((m + n) \log^* n)$.

**Theorem**

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