13 Union-Find Data Structure for Disjoint Sets

Here, we consider a data structure for representing sets of elements. Let $\mathcal{U} = \{x_1, x_2, \ldots, x_n\}$ denote the universe of objects that we would like to consider. We want to maintain sets $S_1, S_2, \ldots \subseteq \mathcal{U}$ that are disjoint, i.e. $\forall i, j, i \neq j, S_i \cap S_j = \emptyset$. In other words, an element of $\mathcal{U}$ can belong to at most one set.

13.1 Motivation

A disjoint-set data structure has a lot of applications. For instance, Kruskal's minimum spanning tree algorithm (MST-Kruskal, CLR pp. 504–505) relies on such a data structure to maintain the components of a partially constructed MST. One can also imagine an on-line application involving a graph that is changing dynamically as vertices and/or edges are added, and we would like to maintain the connected components of the graph. In both these applications, we can make use of a disjoint-set data structure to represent the set of vertices in a component of the graph.

13.2 Design Choices

Not surprisingly, we represent an object in the universe by a node identified by a label (such as x_1). A set is also identified by a unique name; w.l.o.g.[1], we identify a set by an arbitrarily chosen member of the set which is its "representative" or "leader" (note that this makes the set identifier unique since an element can belong to at most one set). The only criterion for selecting a "leader" is that it should not change unless the set itself is changed (i.e. new elements are added).

13.3 Set Operations

The operations that we wish to support are:

1. Make-Set(x) — given object x, create a new set consisting of the element x alone. We assume that x does not already belong to some set. x is the "leader" of the singleton set created.

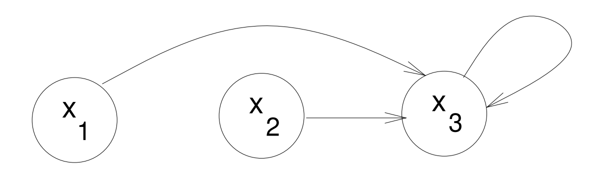
2. Union(x, y) — given objects x and y, create a new set that is the union of the two (different) sets to which x and y belong and destroy the original two sets in the process. Again w.l.o.g., we choose the leader of either of the two original sets to be the leader of the new set.

3. Find-Set(x) — given object x, find the set to which it belongs; i.e. return the identity of the "leader" of that set.

13.4 Linked-List Implementation

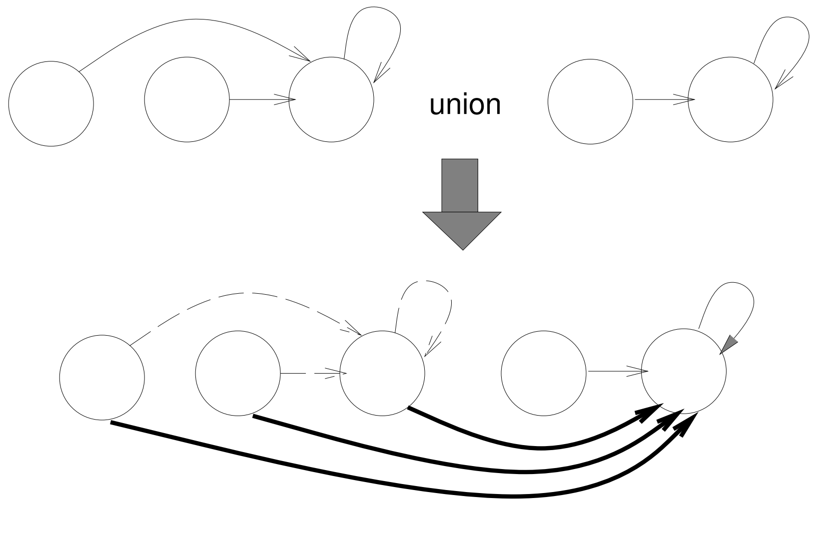
Suppose we choose to represent a set by a linked list — a member of the set corresponds to a node in the list; the first node (at the head of the list) is the "leader"; every node has a pointer to the leader at the head of the list.

[1] "without loss of generality"



The set { x_1 , x_2 , x_3 } with x_3 as the leader.

To take the union of two sets S_1 and S_2, we combine the two linked lists together. w.l.o.g. let the leader of the new set be the same as the leader of S_1; then, the pointers for all elements of set S_2 have to be changed to point to the new leader. The time to do this is proportional to $|S_2|$, the number of elements in S_2.



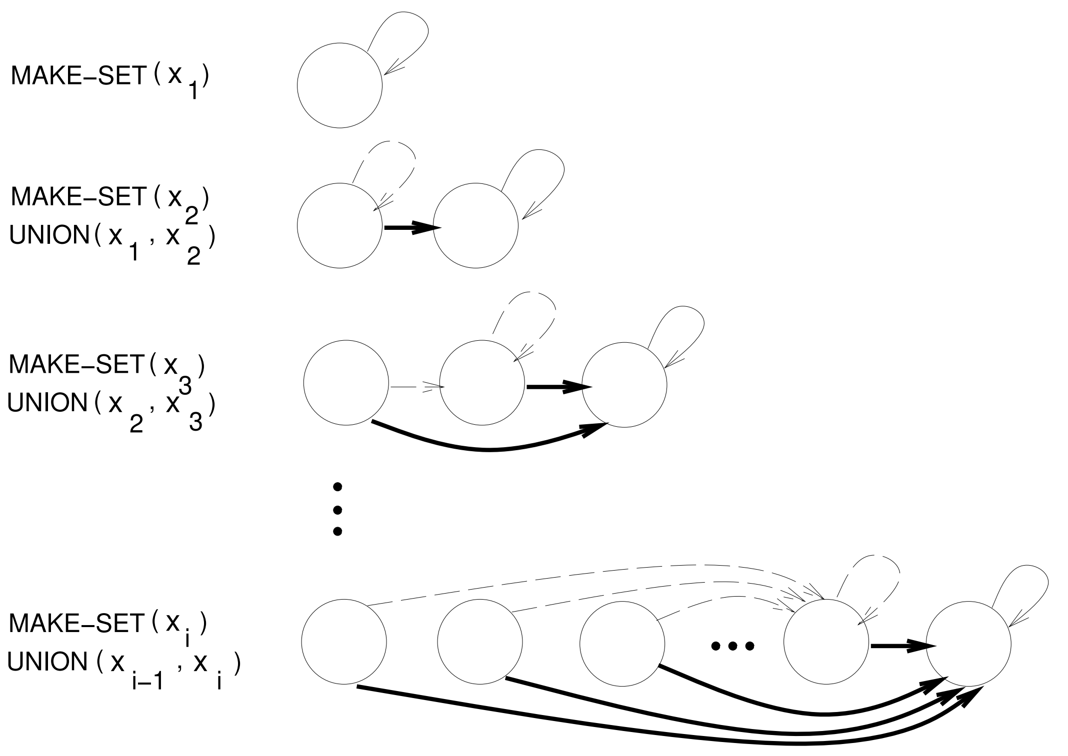
Implementation of the UNION operation.

13.4.1 Analysis

With this implementation, MAKE-SET takes $\Theta(1)$ time to create a linked-list with a single node, and FIND-SET(x) takes only $\Theta(1)$ time to follow the pointer from x to the head of the list and return the identity of the leader. However, the UNION operation can be very slow.

Let n denote the number of MAKE-SET operations, which is the same as the number of objects in our universe. Let m be the total number of MAKE-SET, FIND-SET and UNION operations taken together. Clearly, $m \geq n$. Since every UNION operation reduces the number of sets by one, and we start with n sets and end with at least 1, the number of UNION operations u is at most $n - 1$.

What is the worst-case sequence of operations?



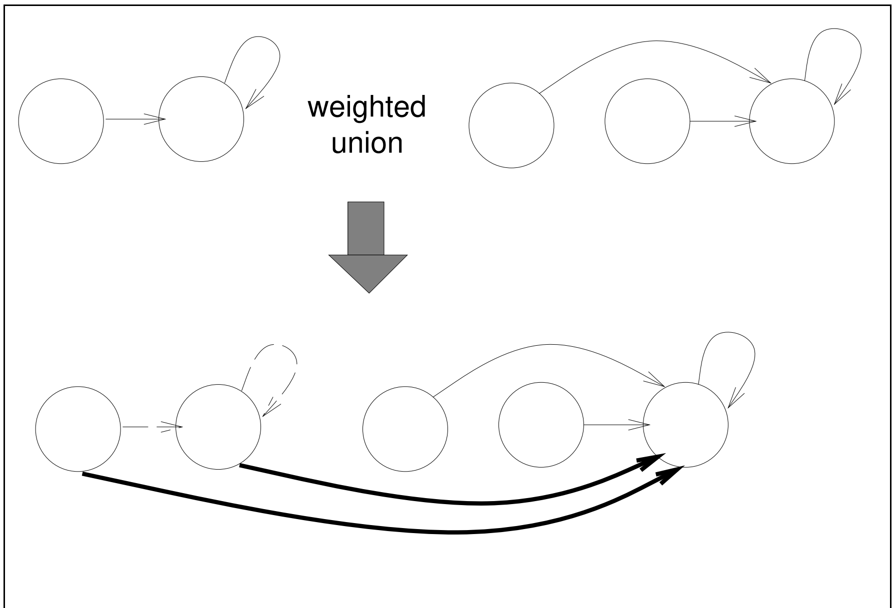
Worst−case sequence of MAKE−SET and UNION operations.

The i^{th} UNION operation takes the union of the sets $\{x_1, x_2, \ldots, x_{i-1}\}$ and $\{x_i\}$. Since x_i could be chosen as the leader of the new set, the operation may need to modify $i - 1$ pointers, and take time $\Theta(i)$. Therefore, the total cost of the sequence of m operations is $\sum_{i=1}^{m} \Theta(i) = \Theta(m^2)$, and the amortized cost of a single UNION operation is $\Theta(m^2)/m = \Theta(m)$.

13.4.2 Weighted Union Heuristic

This is a simple heuristic that can improve the running time. When taking the union of two sets, combine the two lists by modifying the pointers in the smaller list and keep the longer list unchanged.

Thus, the leader of the larger set becomes the leader of the new set as well. Since the elements in the larger set retain the same leader, their leader pointers do not need to be updated; only the leader pointers for the nodes in the smaller list need to be changed to point to the new leader.



Weighted union heuristic.

Clearly, we also need to maintain the size of the list, for instance in the header node, in order to implement this heuristic, with only an additional $\Theta(1)$ time overhead. MAKE-SET and FIND-SET operations still take constant time.

13.4.3 Analysis

Theorem 13.1. *With the weighted-union heuristic, a sequence of* m *operations takes time* $O(m + n \log n)$.

Proof: Consider a fixed object x. How often does its leader pointer change? Every time the leader of x changes because of a UNION operation, x must be in a set that is at least as large as the other set.

Initially, x is in a set all by itself. The first time x acquires a new leader, the size of the set containing x at least doubles in size after the UNION operation. In general, after the leader pointer of x has changed k times, the resulting set has at least 2^k members.

Since the largest possible set has at most n members, if k_{max} is the maximum number of times that the leader pointer of x is changed, then

$$2^{k_{max}} \leq n$$

$$\therefore k_{max} \leq \lceil \lg n \rceil.$$

Over all the n objects, the total amount of work done in updating leader pointers is $O(n \log n)$. In addition, there are at most m MAKE-SET and FIND-SET operations with a total cost of $O(m)$.

Therefore, the total worst-case cost is $O(m + n \log n)$. □

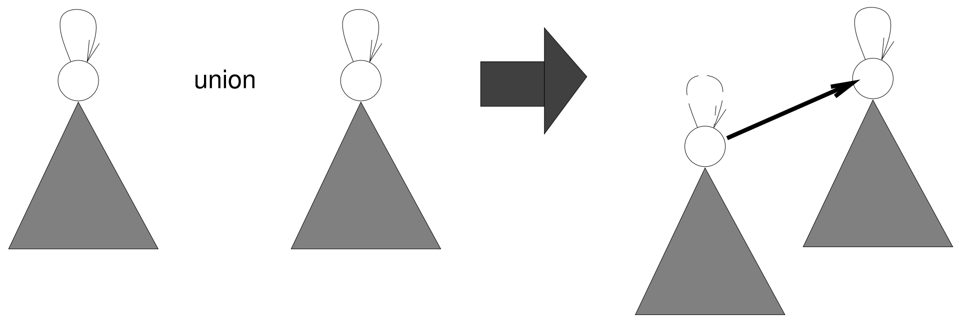
Corollary 13.1.1. *The amortized cost of a* UNION *operation with the weighted-union heuristic is* $O(\log n)$.

Proof: CLR Ex. 22.2-3 pp. 446. □

13.5 Rooted-Tree Implementation

An alternative implementation would be to represent a set by a rooted tree. Every member of the set corresponds to a node in the tree. A node x has a pointer $p[x]$ to its parent; the parent of the root node is the node itself. The root of the tree is the representative or leader of the set. Our collection of disjoint sets therefore forms a disjoint forest of rooted trees.

The naive implementation is no better than the linked-list data structure. MAKE-SET still takes $\Theta(1)$ time; the UNION of two sets can be done, in $\Theta(1)$ time, by making the root of one tree a (new) child of the root of the other; FIND-SET(x) involves following parent pointers from the node x to the root of the tree containing x. It is not hard to come up with a sequence of operations that results in a tree that is a long chain of nodes, so that FIND-SET takes as much as $\Theta(n)$ time.



Implementation of the UNION operation.

However, we can utilize two heuristics to achieve optimal running time in the amortized sense.

13.5.1 Heuristic 1: Union By Rank

This is analogous to the weighted-union heuristic for linked lists: we make the root of the smaller tree a child of the root of the larger one.

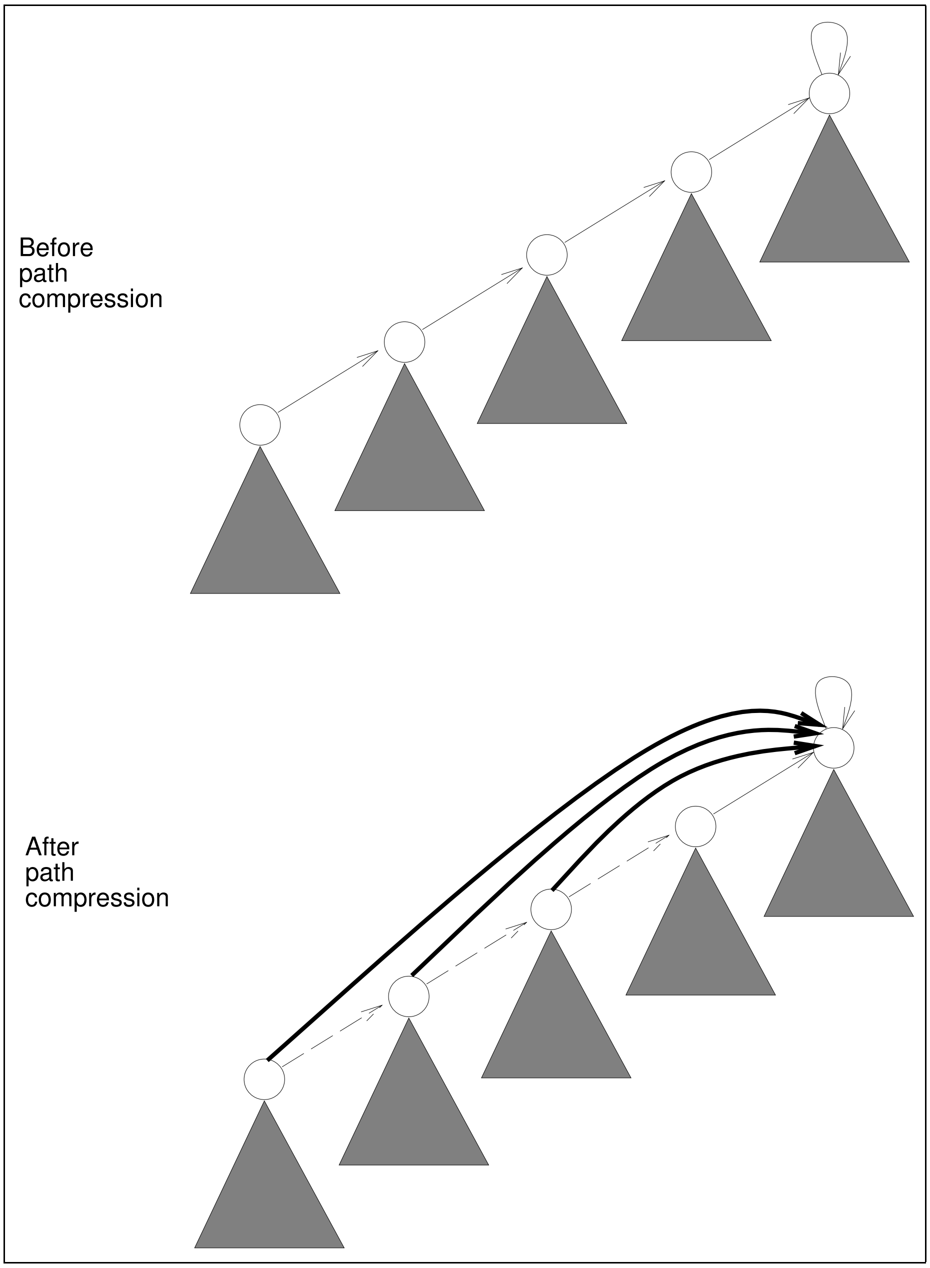
However, we don't maintain the size of every subtree explicitly; instead, we use the **rank** of a node as an upper bound on the height of the subtree rooted at that node. When taking the union of two trees (sets), we make the root of the tree with smaller rank a child of the root of the tree with larger rank, breaking ties arbitrarily if the ranks happen to be equal.

If only the union by rank heuristic is used, then we achieve the same running time as with the weighted-union heuristic for the linked-list representation (CLR Ex. 22.4-3 pp. 458).

13.5.2 Heuristic 2: Path Compression

Define the **find path** of a node x as the sequence of nodes encountered along the path from x to the root of the tree containing x. Thus, the find path of x is $\langle x, x_1, x_2, \ldots, x_r \rangle$ where $x_1 = p[x]$, $x_{i+1} = p[x_i]$, and x_r is the root node. These are the nodes encountered while performing the operation FIND-SET(x).

The path compression heuristic makes an extra pass over the find path of x and updates each node on the path so that it points directly to the root.



13.5.3 Psuedo-code

```
MAKE-SET(x)
    p[x] ← x;
    rank[x] ← 0;
```

```
UNION(x', y')
    LINK(FIND-SET(x'), FIND-SET(y'));
```

```
LINK(x, y)
    if rank[x] > rank[y] {Comment: Union by rank heuristic.}
        p[y] ← x;
    else
        p[x] ← y;
        if rank[x] = rank[y]
            rank[y] ← rank[y] + 1;
```

```
FIND-SET(x)
    if x ≠ p[x]
        {Comment: Recursively compute FIND-SET(p[x]),
        and then update p[x] to point directly to the root.}
        p[x] ← FIND-SET(p[x]);
    return p[x];
```

13.5.4 Analysis

Theorem 13.2. *Using both union by rank and path compression heuristics, the worst-case running time of a sequence of* m *operations,* n *of which are* MAKE-SET *operations, is* $O(m \cdot \alpha(m, n))$*, where* $\alpha(m, n)$ *is the functional inverse of Ackermann's function.*

Ackermann's function: It is defined as

$$\begin{aligned} A[1, j] &= 2^j, \text{ for all } j \geq 1 \\ A[i, 1] &= A[i-1, 2], \text{ for all } i \geq 2 \\ A[i, j] &= A[i-1, A[i, j-1]], \text{ for all } i, j \geq 2 \end{aligned}$$

The inverse of Ackermann's function is defined as

$$\alpha(m, n) = \min\left\{i \geq 1 : A\left[i, \left\lfloor \frac{m}{n} \right\rfloor\right] > \lg n\right\}$$

Define **tower**(i) as a tower of exponentials of 2's of height i. In other words,

$$\begin{aligned} \text{tower}(0) &= 1 \\ \text{tower}(i) &= 2^{\text{tower}(i-1)}, \text{ when } i > 0 \end{aligned}$$

Note the following properties of Ackermann's function and its inverse:

1. $A[i, j]$ grows monotonically in both coordinates.
2. For a fixed value of n, as m increases, $\lfloor m/n \rfloor$ increases. Therefore, the smallest value of i required to bring $A[i, \lfloor m/n \rfloor]$ down to $\lg n$ decreases. This coincides with the intuition that as m increases the average length of the find path decreases due to path compression.
3. $A[4, 1] = \text{tower}(17)$ which is greater than the estimated number of atoms in the observable universe. Therefore, for any practical values of m and n, $\alpha(m, n) \leq 4$.
4. $A[2, j] = \text{tower}(j + 1)$.

5. Recall the definition of the $\lg^*$ function:

$$\mathrm{tower}(i-1) < n \leq \mathrm{tower}(i) \iff \lg^* n = i$$

   Again for all practical purposes, $\lg^* n \leq 5$.

The $O(m \cdot \alpha(m,n))$ time bound is tight. Here, we prove the weaker bound of $O(m \log^* n)$. First, we need the following properties of ranks:

1. For all nodes x,
   (a) The rank of x is initially 0.
   (b) $\mathrm{rank}[x]$ increases with time until x is no longer the root (i.e. $x \neq p[x]$); from then onwards, $\mathrm{rank}[x]$ does not change.
   (c) If x is the root, then $x = p[x]$ so that $\mathrm{rank}[x] = \mathrm{rank}[p[x]]$; otherwise, $\mathrm{rank}[x] < \mathrm{rank}[p[x]]$.
   (d) Whenever x acquires a new parent, the rank of the new parent is strictly bigger than the rank of the old parent; $\mathrm{rank}[p[x]]$ is a monotonically increasing function of time.

2. For all tree roots x,

$$\mathrm{size}(x) \geq 2^{\mathrm{rank}[x]}.$$

   The proof follows by a simple induction on the number of LINK operations.

3. For all integers $r \geq 0$, there are at most $n/2^r$ nodes of rank r; hence, it follows that the rank of a node is at most $\lfloor \lg n \rfloor$.

Proof: We use the aggregate method of analysis to prove that the worst-case cost of a sequence of m operations is $O(m \log^* n)$. We charge each operation proportional to the actual cost; the total number of charges accumulated over the entire sequence of operations gives us the amortized cost.

Since MAKE-SET and LINK do only a constant amount of work, we charge them unit cost.

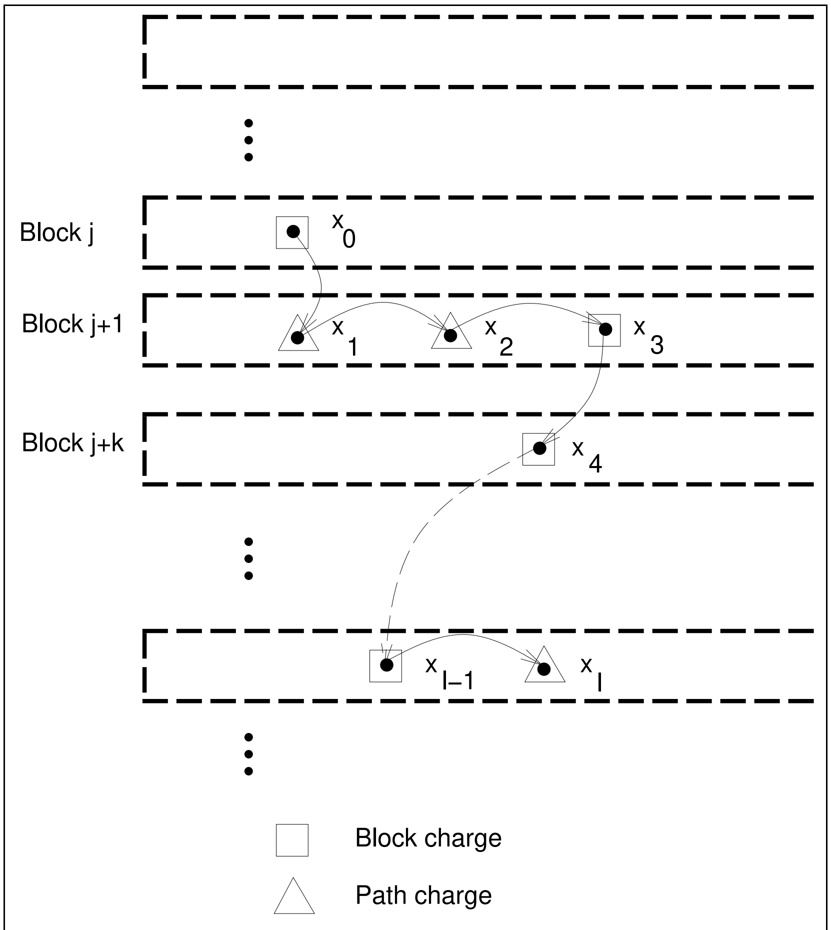
The cost of a FIND-SET operation is proportional to the number of nodes visited on the find path. We assess two types of charges to the nodes on the find path — a **path charge** and a **block charge**. The nodes are partitioned into blocks depending on their ranks.

Define

$$\begin{aligned} B(j) &= -1, \text{ if } j = -1 \\ &= \mathrm{tower}(j), \text{ otherwise} \end{aligned}$$

The function $B(j)$ is used to partition the ranks into blocks — the j^{th} block consists of ranks in the set $\{B(j-1)+1, B(j-1)+2, \ldots, B(j)-1, B(j)\}$. A node with rank r is assigned to the block numbered $\lg^* r$. Recall that the ranks of the nodes range from 0 through $\lfloor \lg n \rfloor$; therefore, the block numbers range from 0 through $\lg^* \lfloor \lg n \rfloor = \lg^* n - 1$. There are a total of $\lg^* n$ blocks.

Consider the cost of the operation FIND-SET(x_0). Let the find path be $\langle x_0, x_1, x_2, \ldots, x_l \rangle$. For every node on the find path that is the last node on the find path with rank in some block, we assess one block charge. For every other node on the find path, we assess one path charge. In other words, we assess the node x_i one unit block charge iff its parent has a rank in another (higher-numbered) block, i.e. iff $\lg^* \mathrm{rank}[x_i] < \lg^* \mathrm{rank}[x_{i+1}]$. In addition, we charge the child of the root an extra block charge.



How block and path charges are assessed for nodes on the find path.

Once a node (other than the root or its child) is assessed a block charge, it will never again be assessed a path charge. This is true because the node already had a parent in a different block and any new parent it acquires over time will have even higher rank and will also be in a different block from the node itself.

Since there are $\lg^* n$ blocks and we have an additional block charge for the child of the root, the total number of block charges over the sequence of m operations is at most $m(\lg^* n + 1)$.

Observe that if a node x_i is assessed a path charge, then x_i cannot be a child of the root (because we assess a block charge to the child of the root). After path compression, x_i will acquire a new parent: the root node x_l. Also, this new parent will have rank higher than the old parent.

Suppose x_i has rank in block j. How many times can it happen that x_i acquires a parent of successively higher rank in the same block (block j), and is thus assessed a path charge, until x_i eventually acquires a parent in a different block, after which x_i will never again be assessed a path charge? In the worst-case, x_i might have the lowest rank in its block, i.e. $B(j-1)+1$, and its' parents' ranks successively take on the values $B(j-1)+2, B(j-1)+3, \ldots, B(j)$. Therefore, x_i can be assessed at most $B(j) - B(j-1) - 1$ path charges while its rank is in block j.

How many nodes have ranks in block j? For all integers $j \geq 0$, let $N(j)$ be the number of nodes with ranks in block j. Then, from property 3,

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

For $j = 0$,

$$N(0) = \frac{3n}{2} = \frac{3n}{2B(0)}.$$

For $j \geq 1$, we have

$$\begin{aligned} N(j) &\leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{r=B(j)-B(j-1)-1} \frac{1}{2^r} \\ &< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\ &= \frac{n}{2^{B(j-1)}} \\ &= \frac{n}{B(j)} \end{aligned}$$

Thus,

$$N(j) \leq \frac{3n}{2B(j)}, \text{ for all } j \geq 0.$$

The overall number of path charges, $P(n)$, is bounded by the sum over all blocks of the product of the maximum number of nodes in a given block and the maximum number of path charges per node of that block.

$$\begin{aligned} P(n) &\leq \sum_{j=0}^{\lg^* n-1} \frac{3n}{2B(j)}(B(j) - B(j-1) - 1) \\ &\leq \sum_{j=0}^{\lg^* n-1} \frac{3n}{2B(j)} B(j) \\ &= \frac{3}{2} n \lg^* n \end{aligned}$$

Therefore, the total number of block and path charges over the sequence of m operations is at most $m(\lg^* n+1)+(3/2)n\lg^* n = O(m\lg^* n)$, since $m \geq n$. In addition, there are $O(n)$ MAKE-SET and LINK operations, with unit cost each. Thus, the total running time is $O(m\lg^* n)$. □