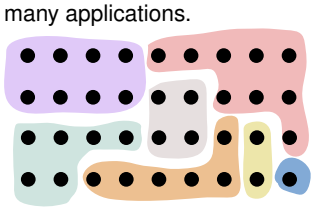
14 Disjoint Sets

Disjoint Sets. A family of disjoint sets is nothing more than just sets without overlapping elements. In computer science, this abstraction, which can be supported efficiently, is surprisingly rich and has found many applications.



A data structure for disjoint sets represents a dynamic (i.e., continually changing) collection of disjoint (i.e., nonoverlapping) sets. More precisely, it maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint sets, where X is a base set, every set $S_i \subseteq X$, and $S_i \cap S_j = \emptyset$ for all $i \neq j$. The collection is dynamic as sets can be united together to form larger sets.

Most often, the collection starts out with each member of the base set X being in its own set. For example, if $X = \{1, 2, 3\}$, the sets are $S_1 = \{1\}$, $S_2 = \{2\}$, and $S_3 = \{3\}$. We intend to to support the following two main operations:

- Given two elements $x, y \in X$, determine whether they belong in the same set.
- Given two elements $x, y \in X$, unite the set containing x and the set containing y.

Our goal is to design and implement an efficient disjoint sets data structure. There are a few considerations:

- The number of elements in the base set X—denoted by n—is large.
- The number of method calls (operations)—denoted by m—can be large.
- Calls to these methods can be interleaving (i.e., an arbitrary combination of queries and uniting).

Sample Applications. This mathematical abstraction has various applications, ranging from cycle detection to being a key component in a popular algorithm for minimum spanning trees. It can also be used in other contexts:

- Support the base set lists computers in a network. Can machine A reach machine B?
- Suppose the base set lists variable names in a computer program. Are these variables the same? This is used in program optimization, register allocation, and plagiarism detection.

This chapter begins with a look at a simple community model and a strawman solution. This forms the basis for more efficient disjoint-set data structures. Following that, we will look at several other representations for the problem, each attempting to fix a issue identified in prior representations. This includes implicit membership, lazy linking, and height control.

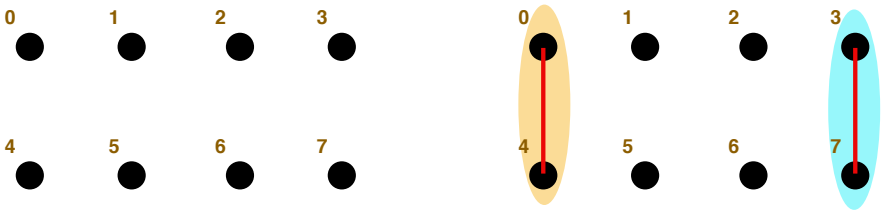


Figure 14.1: Isolated communities become less isolated after linking `link(0, 4)` and `link(3, 7)`.

14.1 Storing Connected Worlds

Consider a town with n members $0, 1, 2, \ldots, n-1$. Somewhat strangely but initially no one knew anyone else, so we started off with n isolated "communities," as shown in Figure 14.1 (left). Not long after, bonding would take place. For example, linkages between 0 and 4, and between 3 and 7 were formed, creating larger communities in this town.

With more bonding, larger communities were formed. The rule is simple: when x and y bond together, the community in which x belongs and the community in which y belongs unite into a larger community.

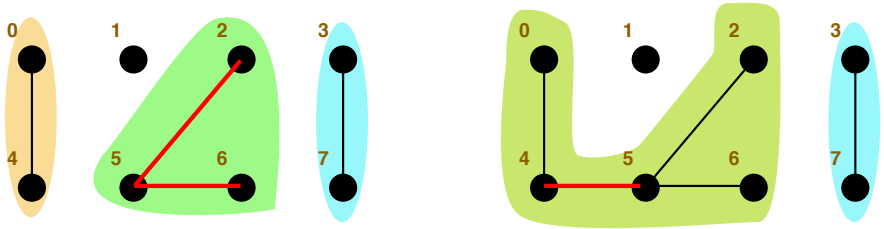


Figure 14.2: Larger communities are formed with more linking (thick lines).

It is easy to see that communities in this town are disjoint sets and we expect to support the following concrete operations:

- `isConnected(x, y)` returns `true` if `x` and `y` belong in the same set (community) at the moment (i.e., given all the linkages so far).
- `link(x, y)` unites the set containing `x` with the set containing `y`.

How should we represent such a collection of disjoint sets so that the operations as listed can be efficiently supported?

Explicit Sets and Back References

To store a system of disjoint sets, we may consider storing each set using an existing `Set` data type. This can be implemented via the `HashSet` or `TreeSet` classes in Java. For example, the system in Figure 14.2 (left) will be kept as 4 separate sets:

```
A = {0, 4}
B = {1}
C = {2, 5, 6}
D = {3, 7}
```

These sets do not need to be named; we only name them so we can refer back to them more easily. In this representation, we do not have an efficient way to support `isConnected`: given two elements `x` and `y`, we cannot tell which sets they belong in, so we are unable to quickly verify whether they belong in the same set.

This is, however, not difficult to fix—keep a map from each element to the set that contains it. For instance, we would keep the following back-reference map:

```
S = { 0: A, 1: B, 2: C, 3: D, 4: A, 5: C, 6: C, 7: D}
```

If such a map is kept as a `HashMap`, the operation `isConnected` can be supported in constant time. This is as fast as one could hope for.

Yet there appears to be no way to efficiently support `link`. For example, to unite `link(4, 5)`, we appear to need two things:

- Form a new set that is the union of $\{0, 4\}$ and $\{2, 4, 6\}$. Because this is a set, it would be hard to avoid spending time proportional to the size of the set if we hope to store it explicitly as Java's sets.
- Redirect the back-reference map so that the elements of the union point to the new set. At the very least, we need to redirect elements of the smaller of the two sets.

These two obstacles motivate new a design, without which `link` would often take $O(n)$ time, where n is the size of the base set.

Implicit Membership With Eager Linking

Without loss of generality, the elements of X can be named $0, 1, \ldots, n-1$, as was the case in our example. One thing becomes quickly clear: to support our data type, we do *not* need to keep the different sets explicitly. Being able to do so means we do not need to form a new set every time two sets are united.

With this in mind, we are motivated to implicitly represent membership in a set as follows:

> Keep a map `find` from each member of X to a number such that `find(x)` is the same as `find(y)` if and only if they belong in the same set.

In other words, `find(x)` gives the "name" of the set that contains `x`.

We will store this map in the simplest possible way, as an array

```
int[] p;
```

of length `n`, where `p[x]` is the value we want `find(x)` to return. This is possible because the members are named $0, 1, \ldots, n-1$.

Initialization. We make `p[i] = i` for `i = 0, 1, ..., n-1`. This makes each element belong in its own set, satisfying our initial condition.

Operations. Supporting `isConnected(x, y)` is simple.

```
boolean isConnected(int x, int y) {
    return p[x] == p[y];
}
```

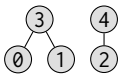
This works because we have arranged for the map encoded by p to give the same value if and only if they belong in the same set. Also, it is easy to see that `isConnected` takes $O(1)$ time to answer.

Supporting `link(x, y)` is more involved, yet still conceptually simple: Change all elements whose p are the same as `p[x]` to `p[y]`. In code, we have:

```
void link(int x, int y) {
    int xName = p[x], yName = p[y];
    for (int i=0;i<p.length;i++) {
        if (p[i] == xName)
            p[i] = yName;
    }
}
```

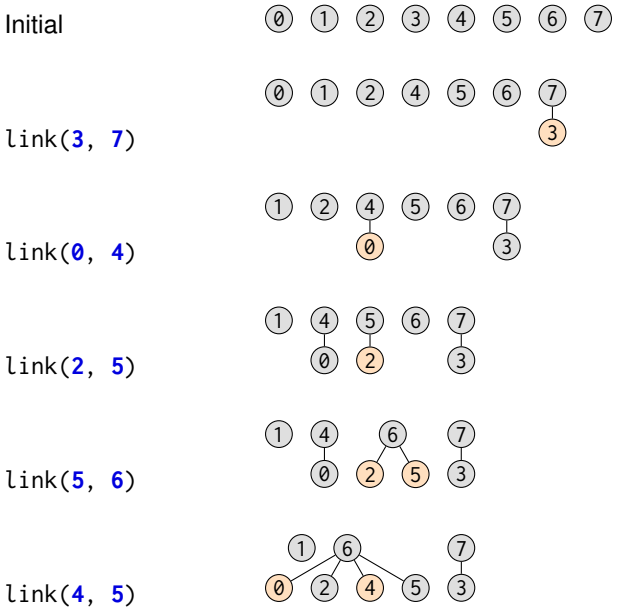
This code has inside it two implicit steps: it looks for the members of the set that contains x. These are precisely everyone whose number was the same as x. Once identified, they are updated to have the same number as the number that y has.

Using this interpretation, the array `p = {3, 3, 4, 3, 4}` can be depicted as follows (self-pointing is omitted):



Visualization. To visualize our data structure, it helps to interpret the array `p[]` as "pointing to." More precisely, `p[x]` is what element x points to. This can be viewed as family trees. In tree terminology, `p[x]` is the parent of x and if `p[x]` is x itself, it is the root of that tree. That is, p can be seen as a forest.

Example. Suppose we play out the example above step-by-step. Specifically, we use $n = 8$ and apply `link` in the following order:



As it stands, while we have avoided storing the sets explicitly, we have not managed to avoid redirecting a whole host of references. The core issue is that when a set is united with another set, the entire set of elements is moved to join a new set. In other words, our `link` operation appears too eager.

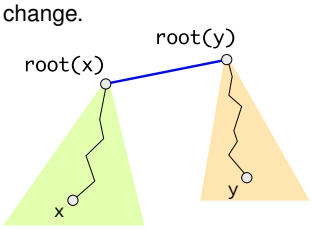
14.2 Lazy Linking

When we link x and y, the eager scheme would redirect every member of the set containing x to the "name" of the set that contains y. This strategy is too eager—that is, it carries out too much upfront work. How can we make `link` less eager? This turns out to be simple: more indirection.

Using the "point-to" view we took in the visualization above, each element x can be followed until the element there points to itself. We term this element the root of x, denoted by `root(x)`. This root element is special: we can make it represent the set S whose elements have it as the root—that is, every $y \in S$ has the same root $r = \texttt{root}(y)$.

Once the root of x and the root of y have been identified, `link` will simply make `root(x)` point to `root(y)`. This linking step clearly takes constant time.

Lazy linking will only direct the roots. Once the roots are found, nothing else has to change.



Implementation of Lazy Linking. To implement this idea, we will begin by writing a method that identifies the root of a given element, like so:

```
int root(int x) {
    while (p[x] != x) {
        x = p[x];
    }
    return x;
}
```

The `root` method does nothing more than following the "point-to" relation, until the element points to itself. When this happens, we know we have reached the root.

To support `isConnected(x, y)`, we only need to figure out whether x and y have the same root, hence writing:

```
boolean isConnected(int x, int y) {
    return root(x) == root(y);
}
```

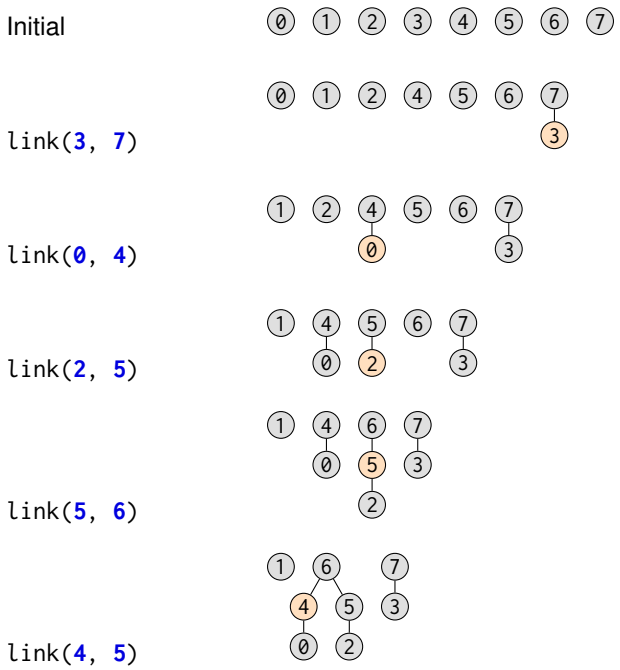
To support `link(x, y)`, we will make the root of x point to the root of y:

```
boolean link(int x, int y) {
    p[root(x)] = root(y);
}
```

Notice that the workhorse of both operations turns out to be the `root` function. We will discuss their running time soon. For the time being, we will look at a concrete example of this idea in action.

Visualization. By interpreting the `p[]` array like before, we can visualize our lazy linking data structure as the following example shows.

Example. Suppose we play out the example above step-by-step. Specifically, we use $n = 8$ and apply `link` in the following order:



Running Time Analysis. The running time of both `isConnected(x, y)` and `link(x, y)` is the same as the running time of running `root(x)` and `root(y)`.

But how long does `root` take? By inspecting the code, it is clear that the running time of `root` is the number of times it has to follow the "point-to" relation until the code reaches its root.

As it turns out, this quantity is variable. For some element, the element itself is already a root. For some other element, the element is just one or two steps away from its root. Yet it is possible that for some element, it has to go through essentially the whole roster of base elements before hitting its root. Below is a concrete example:

Example. Consider a system with n elements. Suppose `link` is issued in the following order:

```
link(0, 1)    link(1, 2)    link(2, 3)    ...
link(n-2, n-1)
```

The "point-to" relation is a long chain $0 \to 1 \to 2 \to \cdots \to n-1$. Hence, if `root(0)` is called, it will take $\Theta(n)$ time in this case.

This leads to the conclusion that lazy linking, as it stands, requires $O(n)$ for both `isConnected` and `link` in the worst case. But we should remember that the bottleneck here is the potentially long chain that `root` has to follow before either the real linking or checking can be carried out. In what follows, we will look at a technique that helps control the length of such a chain.

Height Control

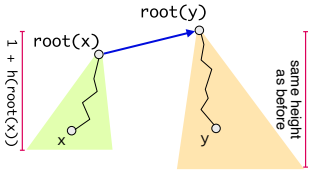
Lazy linking turns out to be an important ingredient in efficient linking. The challenge? We need to avoid deep structures. This can be accomplished via simple modifications. Remember that lazy linking points the root of a set to the root of another set. We will keep track of the size of each root so that when it is time to unite two sets, we will **link the smaller root into the larger one**.

The larger root intuitively houses a deeper structure than the smaller root. Hence, we expect that linking the smaller root into the larger one will not make the structure any deeper whereas the opposite of linking the larger root into the smaller one will inevitably deepen the resulting structure. This can be formalized as follows.

Let $h(e)$ be the *height* of an element e, defined to be the length of the longest chain of point-to chasing that reaches e. In tree terminology, the height of e is the height of the tree rooted at e. We can guarantee the following:

Lemma 14.1. For every root element r, the height $h(r) \leqslant \log_2 n(r)$, where $n(r)$ is the number of elements for which r is the root.

Proof. The proof will be invariant-style, showing that the lemma holds at the beginning and after every operation that affects heights.



Initially, every element is the root of itself, so at the start, the relation $h(r) \leqslant \log_2 1 = 0$ is true for all $r \in X$. Now the only operation that can affect the heights is `link`. Consider a `link(x, y)` call. Let $r_x = \texttt{root}(x)$ and $r_y = \texttt{root}(y)$. Prior to this `link` operation, it is an invariant that $h(r_x) \leqslant \log_2 n(r_x)$ and $h(r_y) \leqslant \log_2 n(r_y)$. We will assume without loss of generality that $n(r_x) \leqslant n(r_y)$; otherwise, we can just swap the roles of x and y. This means that r_x will be made to point to r_y. Now that r_x is pointing to r_y, the new size of r_y is $n' = n(r_x) + n(r_y)$. At the same time, the new height of r_y is $h' = \max(1 + h(r_x), h(r_y))$—see the illustration (right).

We claim that $h' \leqslant \log_2(n')$. Because it is clear that $h(r_y) \leqslant \log_2 n(r_y) \leqslant \log_2 n'$, we are left to show that $1 + h(r_x) \leqslant \log_2 n'$. Now we know that

$$1 + h(r_x) \leqslant 1 + \log_2 n(r_x) = \log_2 (2 \cdot n(r_x)),$$

but then, we have $n(r_x) \leqslant n(r_y)$, so

$$2n(r_x) = n(r_x) + n(r_x) \leqslant n(r_x) + n(r_y) = n'.$$

This means $1 + h(r_x) \leqslant \log_2 n'$. Altogether, we have that the invariant holds after `link` and hence the lemma holds throughout. □

Consequently, this lemma guarantees that calling `root` from anywhere will take at most $\log_2 n$ steps, making `root` a $O(\log n)$-time operation.

Implementation. We will implement this in a class called `LazyWithHeightControl`. The only modifications here will be related

to keeping track of the size of each set and appropriately pointing the roots based on their sizes. But how can know the size of a set? It suffices to keep an array that indicates for each root the size of the set it is responsible for. Hence, we keep the following member variables:

```
private int[] p;
private int[] sz;
private int n;
```

Specifically, if `r` is a root, we keep in `sz[r]` the size of the set that `r` is the root. The other member variables are as before. Initially, each set contains exactly one member, so at the start, `sz` is an array of all `1`s. Therefore, the constructor looks as follows:

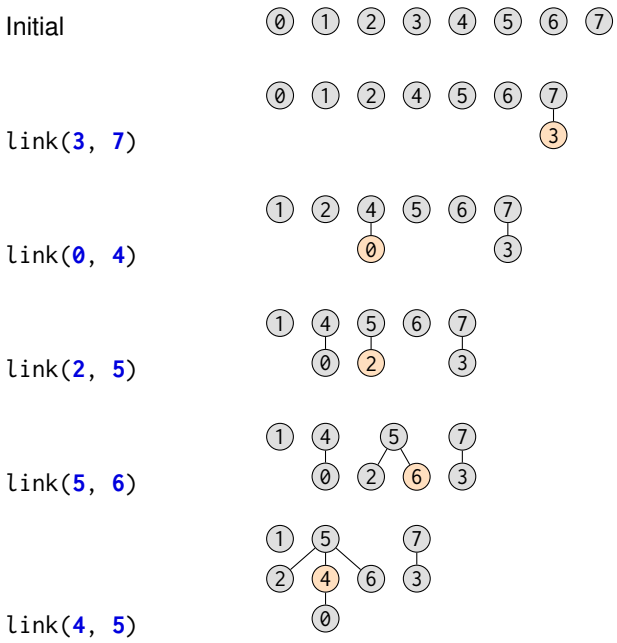
```
public LazyWithHeightControl(int n) {
    this.n = n;
    this.p = new int[n];
    this.sz = new int[n];
    for (int i=0;i<n;i++) {
        p[i] = i; // points to itself
        sz[i] = 1; // size 1
    }
}
```

The only other method that has to change is `link`. But this is straightforward to write now that we know the size of each root:

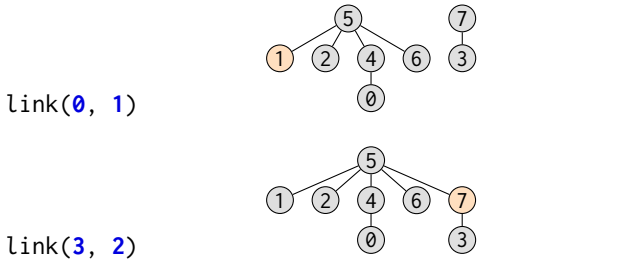
```
public void link(int x, int y) {
    int rootX = root(x), rootY = root(y);  // O(log n) each
    if (sz[rootX] <= sz[rootY]) {
        // join x into y
        p[rootX] = rootY;
        sz[rootY] += sz[rootX];
    } else {
        // vice versa
        p[rootY] = rootX;
        sz[rootX] += sz[rootY];
    }
}
```

Visualization. We will now visualize the same sequence of linking when this height-control policy is used.

Example. Suppose we play out the example above step-by-step. Specifically, we use $n = 8$ and apply `link` in the following order:



Let us extend this example a bit further so that we can observe how height control plays out when the sets become larger.



Running Time Analysis. The constructor that makes n sets takes $O(n)$ as before. It follows directly from Lemma 14.1 that each of `isConnected` and `link`, which internally call `root` twice, takes at most $O(\log n)$ time.

Exercises

Exercise 14.1. Let `a--b` denote calling `link(a, b)`. Suppose eager linking is used. Starting from $n = 7$ disjoint sets, what is the resulting structure after applying the following sequence of operations?

```
0--4, 1--3, 6--4, 5--2, 1--5, 0--1
```

Exercise 14.2. For the same set up as Exercise 14.1., suppose instead that lazy linking is used but this time instead of linking a smaller root into a larger root,

we link a larger root into a smaller root. What does the resulting structure look like?

Exercise 14.3. For the same set up as Exercise 14.1., suppose instead that lazy linking is used and we are linking a smaller root into a larger root as we did earlier for height control. What does the resulting structure look like?

Exercise 14.4. Draw a visualization similar to the examples in this chapter corresponding to the following `p[]` array:

i	0	1	2	3	4	5	6	7	8	9
p[i]	1	1	2	1	2	2	2	5	7	4

Exercise 14.5. Johnny keeps a disjoint-sets data structure that implements lazy linking with size-based height control as discussed earlier. He claims that the following `p[]` array was obtained by applying a series of `link` operations to a starting point that involves n disjoint sets $\{0\}, \{1\}, \ldots, \{n-1\}$ with $n = 7$.

i	0	1	2	3	4	5	6
p[i]	4	3	2	2	2	2	6

First, draw a visualization corresponding to this array. Then, give a sequence of `link(x, y)` operations that results in that `p[]` array, or argue that Johnny couldn't have obtained such an array (by, e.g., proving the array violates some critical property).

Exercise 14.6. Another effective height-control strategy is linking by rank (aka. union by rank). The idea is to maintain the rank of each root. Initially, every element is a root and it has rank `0`. When uniting two sets, we link the root of a lower-ranked set into the root of a higher-ranked set (the set whose rank is a larger number). When root r_x is made to point to root r_y, the rank of r_y stays the same unless the rank r_x and the rank r_y are the same, in which case the rank r_y is increased by 1.

Prove that if `e` is an element and r_e is the root of `e`, then `root(e)` takes time $O(\text{rank}(r_e))$, where $\text{rank}(r_e)$ denotes the rank of r_e.

Exercise 14.7. For the same set up as Exercise 14.1., suppose linking by rank from the previous exercise is used instead. What does the resulting structure look like?

Exercise 14.8. Write a Java class that implements disjoint sets using linking by rank as discussed in Exercise 14.6. What is the running time of each of your operations?

Exercise 14.9. Remember that an undirected tree T is a simple, connected graph that has no cycle. Here, connectedness means any vertex in the graph can reach all the other vertices in the graph. A graph that can be partitioned into one or more trees that do not have common vertices are called a forest.

You will be working with an undirected, simple graph $G = (V, E)$. What is a little unusual is that the edges are presented to you one by one as an `Iterable`, in an arbitrary order. You are to implement a method

```
int countTrees(int n, Iterable<Pair<Integer, Integer>> edges)
```

with the following specifications:

- The input is a number n—the number of vertices in G (vertices are called $0, 1, \ldots, n-1$)—and an `Iterable` whose length is m, where each element is a pair representing an undirected edge. Therefore, such a pair `e` simply indicates that there is an edge between `e.first` and `e.second`.
- The method is to return the number of trees in the forest if G is a forest—or otherwise return 0 if G is not a forest.

Your algorithm must run in $O(m \log n + n)$ time or faster, and use at most $O(n)$ space. That is, it is not possible to store all the edges in your algorithm.

Exercise 14.10. Further optimization to height control can be made by introducing what is known as path compression: when `root(x)` is called to identify the root of `x`, it redirects everyone it visits to the root, like so:

```
int root(int x) {
    if (p[x] != x)
        p[x] = root(p[x]);
    return p[x];
}
```

This helps reduce the height of the tree while doing the same work that `root` has to do nonetheless. There are many interesting theoretical properties that path compression has. For now, we'll study it empirically by looking at how they perform on the following sequence of operations: the notation `a--b` means `link(a, b)`.

```
3--7, 0--4, 2--5, 5--6, 4--5, 0--1, 3--2
```

How does the structure compare to the structure you would otherwise obtain without path compression?

Chapter Notes

Galler and Fischer [GF64] are believed to be the first to study disjoint-set data structures. The simple $O(\log n)$-time data structures discussed in this chapter were known from early on. Linking strategies (e.g., by rank and by size) were extensively studied. Path compression was introduced to reduce the height of the forest storing the sets. Hopcroft and Ullman [HU73] show that $O(\log^* n)$ is possible, where $\log^*$ denotes iterated logarithm. Many of the important results for disjoint-set data structures are due to Robert E. Tarjan, who gives, among other results, a $O(\alpha(m, n))$-time data structure [Tar75] and shows that it is not possible to do it faster in the amortized sense [Tar79]. Here, $\alpha(\cdot, \cdot)$ denotes the inverse of the Ackermann function. The function α grows extremely slowly. It is believed that α is less than 5 for any practical input size. For further study, more advanced algorithms textbooks such as Cormen et al. [Cor+09], Dasgupta et al. [DPV08], and Kleinberg and Tardos [KT06] discuss this topic at length.