COMP6902

COMPLEXITY ALGORITHM

PROJECT

Anne Odeh - 201890105

[redacted]

[redacted]

Case Study : Union Find

Introduction

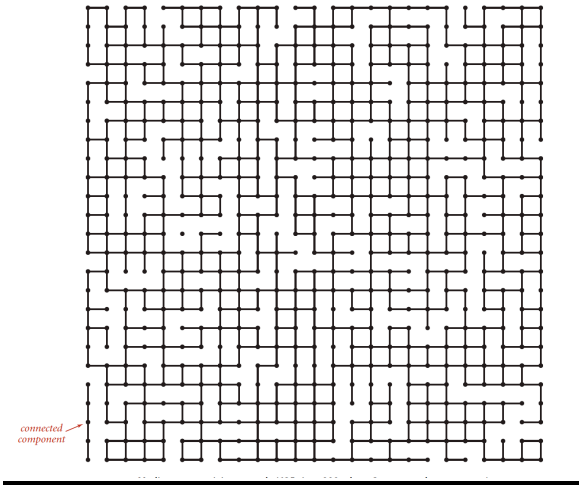
In Union find case study, the writer is explaining fundamental real-life computation problems and also proposing different possible solutions to solve those problems, i.e. Quick Find, Quick Union and Weighted Quick Union. In example, we take a problem of a complex computer network where all the nodes can be considered as an individual computer. In this problem, we need to identify if two different computers (i.e. v and u) want to interact, do we need to establish a new link between those computers? Or do we need to identify if they are already connected with some direct or indirect links? This case study is referring to the reflexive property, where if p is connected to q, and q is connected to r, then p is connected to r. Similarly, let's take an example of a large social network where pairs might reflect the friendship between different people on that network. In this case study, we will go through the possible solutions the writer is proposing to solve a kind of challenging problem and how we can compute the running cost for each possible solution against very large data sets.

Terminologies

Moving forward, we will use network terminology and refer to each object as sites, the pairs as connections, and the equivalence classes as components. Components are sets – if two sites are connected, we can say that they belong to the same component. For simplicity, we assume that we have N sites with integer names, from 0 to N-1.

Problem

Here is an example of a complex network where you can easily identify that in bottom left corner, there are five sites which are connected, and all five sites are in same component. However, with rest of the sites it is very difficult to identify if they are connected or not. Thus, here will try to analyze these very complex connections.



To specify the problem, we are choosing an array as data structure and developing an API, which will encapsulate the basic operations that we need such as initialize the sites, add connection between two sites, identify the component that contains a site, and determine whether two sites are connected or not. Does it mean they are in same component? Lastly, we need a count function to count the number of components.

Union Find Class and Basic Methods:

Public Class UF

UF(int N)	<	initialize N sites with integer names (0 to N-1)
void union(int p, int q)	<	add connection between p and q
int find(int p)	<	component identifier for p (0 to N-1)
boolean connected(int p, int q)	<	return true if p and q are in the same component
int count()	<	number of components

Implementation of API

Declaration of two instance variables, the count of components and array of sites:

```
private int[] id;        // access to component id (site indexed)
private int count;       // number of components
```

Constructor to initialize the number of components based on user input N:

```
public UF(int N)
{  // Initialize component id array.
   count = N;
   id = new int[N];
   for (int i = 0; i < N; i++)
      id[i] = i;
}
```

Return count of components:

```
public int count()
{  return count;  }
```

Connected take sites p and q and decided whether they are in same component (class) or not, the implementation of connected function is based on find function.

```
public boolean connected(int p, int q)
{  return find(p) == find(q);  }
```

Find and Union method implementation is based on each solution quick-find, quick-union and weighted that will discuss further with each solution explanation.

```
  public int  find(int p)
  public void union(int p, int q)
 // See page 222 (quick-find),page 224 (quick-union) andpage 228 (weighted)
```

Main program:

The main function in which we are initially taking the number of components, N to initialize the array of components, then further keep taking input p and q as two sites and decide whether they are in same component or not. The program will output only if p and q are not connected, and will connect them otherwise, it will simply move forward and output nothing. In the following program, union function is the process of moving p and q into same component and once they moved to same component, we can consider them as connected.

```
public static void main(String[] args)
{  // Solve dynamic connectivity problem on StdIn.
   int N = StdIn.readInt();              // Read number of sites.
   UF uf = new UF(N);                    // Initialize N components.
   while (!StdIn.isEmpty())
   {
      int p = StdIn.readInt();
      int q = StdIn.readInt();           // Read pair to connect.
      if (uf.connected(p, q)) continue;  // Ignore if connected.
      uf.union(p, q);                    // Combine components
      StdOut.println(p + " " + q);       //   and print connection.
   }
   StdOut.println(uf.count() + " components");
}
```

Algorithm Union-Find

Dynamic connectivity

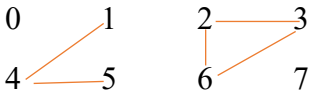
The input is a sequence of pairs of integers, where each integer represents an object of some type, and interpret the pair p q as meaning p is connected to q. We assume that "is connected to" is an equivalence relation:

- Symmetric: If p is connected to q, then q is connected to p.
- Transitive: If p is connected to q and q is connected to r, then p is connected to r.
- Reflexive: p is connected to p.

An equivalence relation partitions the objects into equivalence classes or connected components.

Our goal is to write a program to filter out extraneous pairs from the sequence: When the program reads a pair p q from the input, it should write the pair to the output only if the pairs it has seen to that point do not imply that p is connected to q. If the previous pairs do imply that p is connected to q, then the program should ignore the pair p q and proceed to read in the next pair.

Connected Components: Maximal set of objects that mutually connected

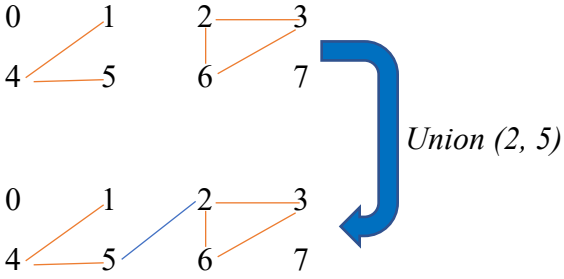


- 3 connected components: {0} {1 4 5} {2 3 6 7}

Implementing the operations

Find query: Check if two objects are in the same component

Use the union command to replace components containing two objects.



- 2 connected components: {0} {1 2 3 4 5 6 7}

Union-Find data type (API)

Goal: To design an efficient data structure for union-find

- Number of objects N can be huge

- Number of operations M can be huge
- Find queries and Union commands maybe intermixed

Dynamic-connectivity client

- Read in number of objects N from standard input
- Repeat:
  - Read in pair of integers from standard input
  - If the pair of integers is not yet connected, connect them and print out the pair
  - If the pair of integers are connected, then skip them.

```
public static void main(String[] args)
{
   int N = StdIn.readInt();
   UF uf = new UF(N);
   while (!StdIn.isEmpty())
   {
      int p = StdIn.readInt();
      int q = StdIn.readInt();
      if (!uf.connected(p, q))
      {
         uf.union(p, q);
         StdOut.println(p + " " + q);
      }
   }
}
```

```
% more tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
```

To test the utility of the API and to provide a basis for development,

- We include a client in **main ()** that uses it to solve the dynamic connectivity problem
- It reads the value of the N followed by a sequence of pairs of integers (each in the range 0 to N-1), calling **find ()** for each pair:
  - If the two sites in the pair are already connected, it moves onto the next pair
  - If they are not, it calls **Union ()** and prints the pair.

The goal is to be able to handle large inputs, for instance, 2 million connections among 1 million sites in a reasonable amount of time.

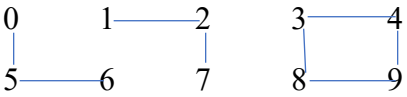
Algorithm Quick-Find

Quick-Find maintains the invariant that p and q are connected if and only if id[p] is equal to id[q]. In other words, all sites in a component must have the same value in id[].

	1	2	3	4	5	6	7	8	9	10
Id[]	0	1	1	8	8	0	0	1	8	8

- 0, 5 and 6 are connected at entry 0

- 1, 2 and 7 are connected at entry 1
- 3, 4, 8 and 9 are connected at entry 8



Find: To check if p and q have the id

Id[6] = 0; id[1] = 1

6 and 1 are not connected.

Union: To merge components containing p and q, change all entries whose id equals id[p] to id[q]

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

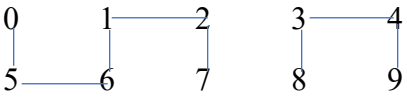
The Union is more difficult to merge the components containing two given objects, and we have to change all the entries whose ID is equal to one of them to the other one end arbitrarily. We choose to change the ones that are the same as p to the ones that are the same as q.

So, if we are going to union 6 and 1, then we have to change entries 0, 5, and 6.

	0	1	2	3	4	**5**	**6**	7	8	9
Id[]	1	1	1	8	8	1	1	1	8	8

This is a problem when we have a huge number of values that can change, but it is easy to implement, and that will be the starting point.

How Quick-Find works:



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

Set up the id[] array with each entry equal to its index, i.e., the objects are independent and are in their own connected components.

- When a union operation is used, for instance, if 4 is supposed to Union with 3, then change all entries whose ID is equal to the first ID to the second one. So, in this case, change 4 to 3.

	0	1	2	3	4	5	6	7	8	9
Id[]	0	1	2	**3**	**3**	5	6	7	8	9

- Union (3, 8)
  3 and 4 must be connected to 8; therefore both of those entries have to change to 8

	0	1	2	3	4	5	6	7	8	9
Id[]	0	1	2	**8**	**8**	5	6	7	**8**	9

- Union (6, 5) – change 6 to 5

	0	1	2	3	4	5	6	7	8	9
Id[]	0	1	2	8	8	5	**5**	7	8	9

- Union (9, 4) – change 9 to 8

	0	1	2	3	4	5	6	7	8	9
Id[]	0	1	2	8	8	5	5	7	8	**8**

- Union (2, 1) – change 2 to 1

	0	1	2	3	4	5	6	7	8	9
Id[]	0	1	**1**	8	8	5	5	7	8	8

- Union (5, 0) – change 5 to 0 and connect the entry corresponding to both 5 and 6 to 0

	0	1	2	3	4	5	6	7	8	9
Id[]	0	1	1	8	8	**0**	**0**	7	8	8

- Union (7, 2) – change 7 to 1

	0	1	2	3	4	5	6	7	8	9
Id[]	0	1	1	8	8	0	0	**1**	8	8

- Union (6, 1) – change 6 to 1 and connect the entries corresponding to 0, 5, and 6 to 1.

	0	1	2	3	4	5	6	7	8	9

Id[]	1	1	1	8	8	1	1	1	8	8

Coding the above algorithm (Quick-Find)

```
public class QuickFindUF
{
   private int[] id;

   public QuickFindUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++)
         id[i] = i;
   }

   public boolean connected(int p, int q)
   {  return id[p] == id[q];  }

   public void union(int p, int q)
   {
      int pid = id[p];
      int qid = id[q];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = qid;
   }
}
```

Set id of each object to itself
(N array accesses)

Check whether p and q are in the same
Component (2 array accesses)

Change all entries with id[p] to id[q]
(at most 2N + 2 array accesses)

The most complicated operation is **Union,** and we must find the first ID corresponding with the first argument and then the ID corresponding to the second argument, and we go through the whole array looking for the entries whose ID is equals to the ID of the first argument and set those to the ID of the second argument.

Quick-Find Analysis

The **find()** operation is certainly quick, as it only accesses the id[] array once to complete the operation. But quick-find is typically not useful for large problems because **Union()** needs to scan through the whole id[] array for each input pair.

Quick-Find is too slow

Cost Model: Number of array accesses (for read and write)

Algorithm	Initialize	Union	Find
Quick-Find	N	N	1

Order of growth of the number of array accesses

Quick-Find defect: Union is too expensive and takes N^2 array accesses to process sequence of N **Union** commands on N objects.

This is problematic because the **union** operation is too expensive. If you have N **union** commands on N objects, which is not reasonable either way, they are either connected or not. As a result, that will take quadratic time N^2 time. This is because quadratic is much too slow, and unacceptable for large problems. The reason for this is the quadratic does not scale and gets slower as computers get bigger and faster.

Algorithm Quick-Union

Below is the implementation of Quick Union, both find and union function:

```
private int find(int p)
{  // Find component name.
   while (p != id[p]) p = id[p];
   return p;
}

public void union(int p, int q)
{  // Give p and q the same root.
   int pRoot = find(p);
   int qRoot = find(q);
   if (pRoot == qRoot) return;

   id[pRoot] = qRoot;

   count--;
}
```

Quick-union

The Quick Union function is speedier in comparison to Quick find. It is based on the same data structure that we discussed previously. For the implementation of find will take a site and follow its link to another site and keep moving until we find the root where a site has been linked to itself.

For further explanation, we are considering a small data set as an input.

For example: N = 10, means there are 10 components and each component include one of its members as its identity. Thus, in example, when we initialize the UF with N = 10, then each array index represents a separate component, as you can see below:

Id[] = 0 => 0
1 => 1
2 => 2
3 => 3
4 => 4
5 => 5
6 => 6
7 => 7
8 => 8
9 => 9

Let's input p = 5 and q = 9.
As per the main program implementation that we discussed earlier, the connected function will run UF connected(int p , int q) – the output of the connected function will be false as per the above implementation of find function, p == id[p] and q == id[q] also find(p) != find(q). As a result, out will be false.

```
public boolean connected(int p, int q)
{  return find(p) == find(q);  }
```

Now, the program will print p and q values (5 and 9) on the screen and apply union function as both p and q because they are pointing to themselves (in other words, keeping the same value on the same index, for instance, id[5] = 5 and id[9] = 9) meaning that they are root components. As per the union function implementation:

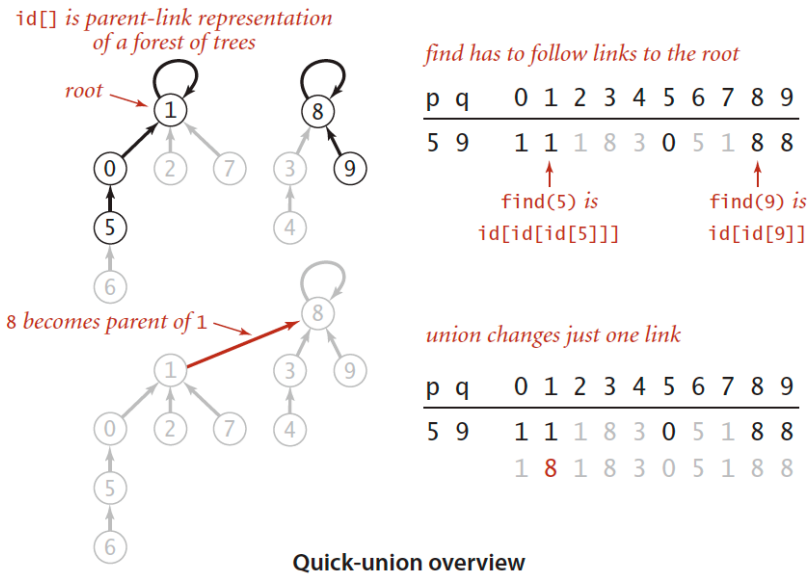
```
public void union(int p, int q)
{  // Give p and q the same root.
   int pRoot = find(p);
   int qRoot = find(q);
   if (pRoot == qRoot) return;

   id[pRoot] = qRoot;

   count--;
}
```

pRoot = 5 and qRoot = 9 as both are root components so pRoot != qRoot and now array will look like the following index 5, which now have the value of 9, which means 5 and 9 are now in the same component (group) and that is why we are reducing the component count by doing count:

Id[] = 0 => 0
1 => 1
2 => 2
3 => 3
4 => 4
5 => 9
6 => 6
7 => 7
8 => 8
9 => 9

Now, if you will again run the same algorithm on the same input p = 5 and q = 9, it will output nothing, as p and q are connected and in same component (class).

Forest of Tree Representation



Quick-union overview

Representing sites as nodes (labeled circle) and the link as an arrow from one node to another, as shown in the above image, gives a graphical representation of the data structure. We call it Forest Tree Representation. It makes it relatively easy to understand the operation of the algorithm.

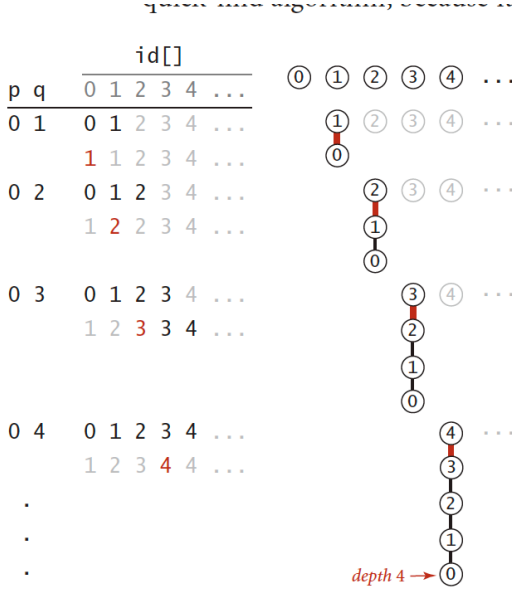
Further, as shown in the above screenshot, 1 and 8 are root components as they are pointing to themselves. Similarly, 0 is pointing to 1, and 5 is pointing to 0, which makes a good tree structure. If the program will run union find as per the above diagram, we will get find (p) which

will return 1 and find (q), which will then further return 8 their root nodes. Both are different than the root nodes, thus, now 8 will be placed on 1 index. Because of this, the left tree will now become a child component of 8 and 1, which is now no longer be a root node. So, they are interconnected with quick union.

Quick Union Analysis

Quick Union seems to be faster than Quick find algorithm because it does not have to go through the entire array of each input pair. However, let us see how much faster it is – analyzing the cost of quick union is more complex then quick find algorithm. Here, the cost is more dependent on the nature of the input. In the best case find method, we just need one array access to find the identifier associate with a site. This means that in the case of root node, the worst-case scenario would be to have it iterate 2N + 1 array access.

As for 0 in the example on the left (this count is conservative since the compiled code will typically not do an array access for the second reference to id[p] in the while loop). Accordingly, it is not difficult to construct a best-case input for which the running time of our dynamic connectivity client is linear; on the other hand, it is also not difficult to construct a worst-case input for which the running time is quadratic.



Quick-union worst case

Fortunately, we do not need to face the problem of analyzing quick union and we will not dwell on comparative performance of quick-find and quick-union because we will next examine another variant that is far more efficient than either. For the instance, you can regard quick-union as an improvement over quick-find because it removes quick-find's main liability (that union() always takes linear time). This difference certainly represents an improvement for typical data,

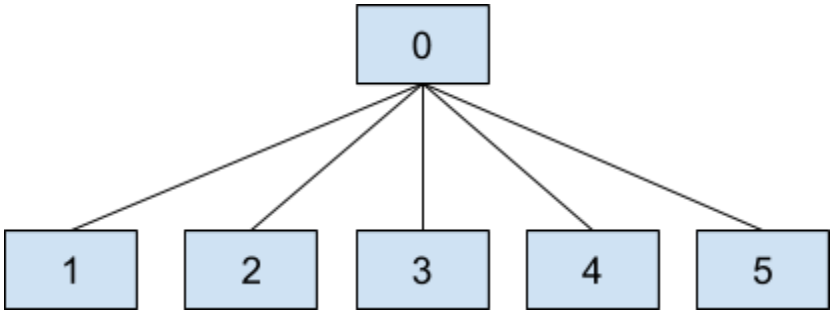
however, quick-union still has the liability that we cannot guarantee it to be substantially faster than quick-find in every case (for certain input data, quick-union is no faster than quick-find).

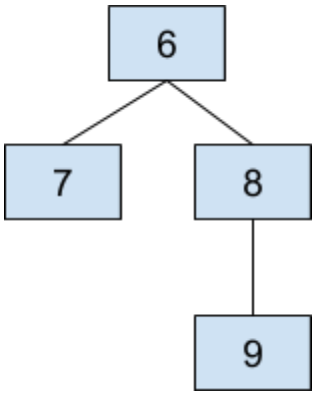
Weighted Quick Union

Weighted Quick Union is the modification of the Quick Union algorithm that avoids tall trees. Rather than linking the first tree with the second tree, we maintain the height of each tree and always link the root of the **smaller** tree to the **larger** tree.

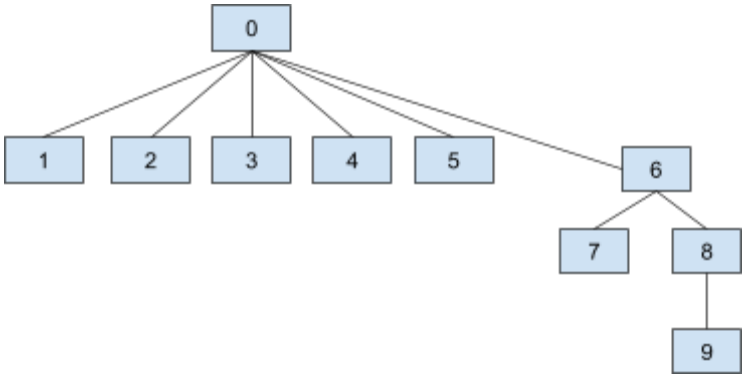
Example: Given the universe

-1	0	0	0	0	0	0	6	6	8
0	1	2	3	4	5	6	7	8	9



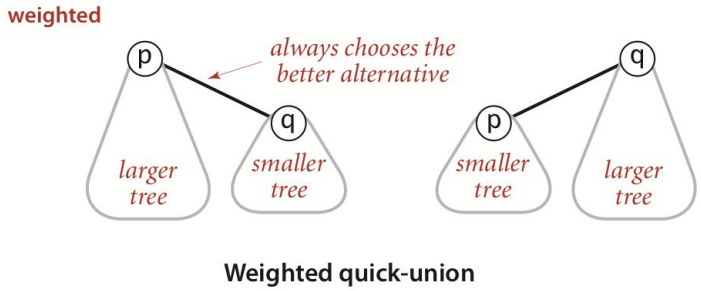


If we call connect(3 8), the second tree with height three will be linked to the first tree.



How to Implement Weighted Quick Union

1. User parent[] to store the parent of each item in the universe/set
2. isConnected(int p, int q) remain the same as in Quick Union
3. The connect(int p, int q) needs to keep track of tree or set sizes either
    a. Use values other than -1 in the parent array for root nodes to track size
    b. Create a separate size array.



Weighted quick-union

Fig: Form Algorithm 4th Edition by Robert Sedgewick, Kevin Wayne

Weighted Quick Union Performance

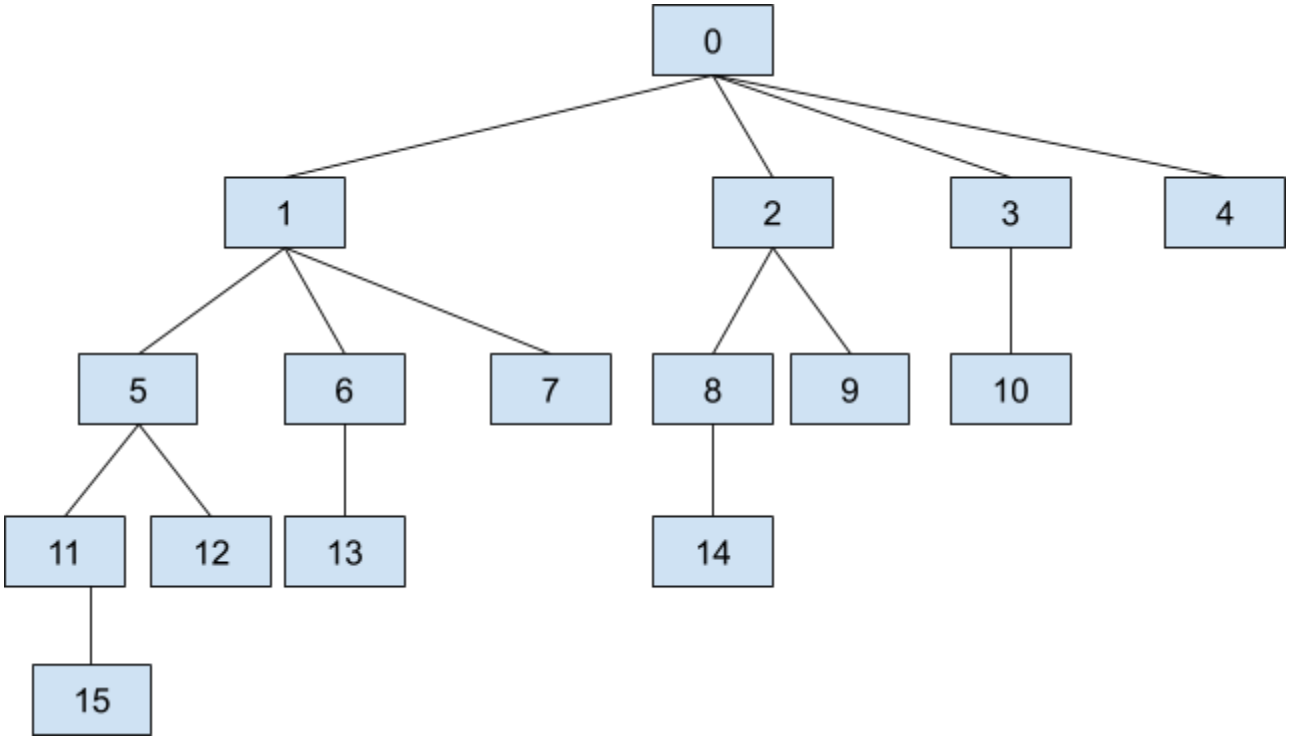
Considering the worst case where the height of the trees to be merged are always equal.

Number of Nodes	Heights
1	0
2	1
4	2
8	3
16	4

The observation above generalizes to provide proof that the **Weighted Quick Union** algorithm can guarantee **Logarithmic** performance

Improving the Weighted Quick Union

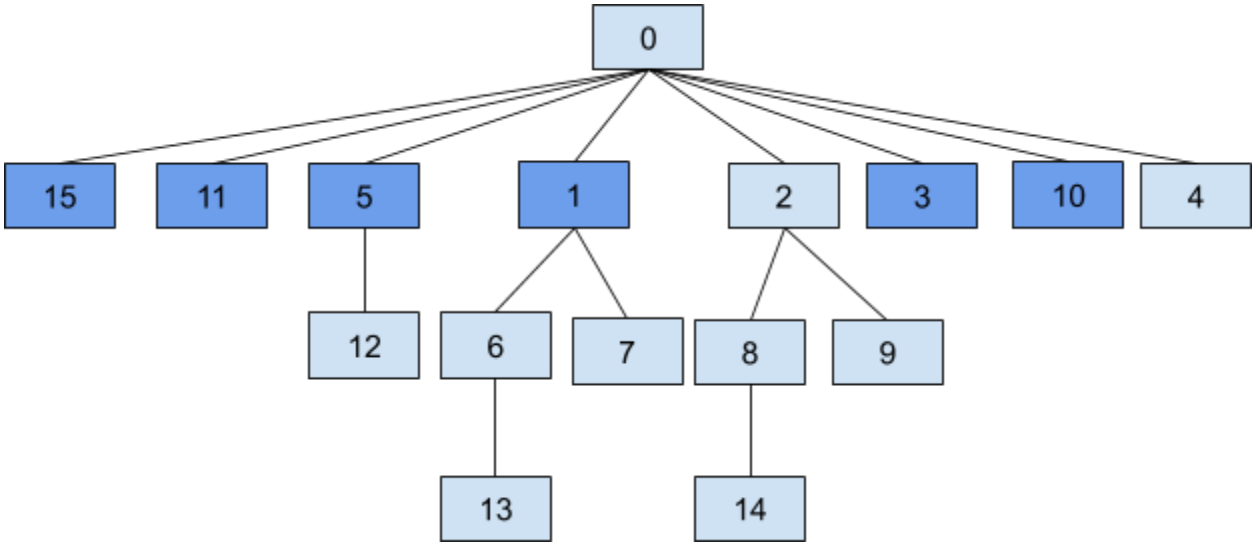
Let's take an example:



Assume we want to find **isConnected(15,10):**

1- Check parent of 15 = 11
2- Check parent of 11 = 5
3- Check parent of 5 = 1
4- Check parent of 1 = 0
5- Check parent of 0 = -1
6- Check parent of 10 = 3
7- Check parent of 3 = 0

Well what if we change all the parents of these nodes to 0?



The additional cost of doing this is insignificant, however, future calls to **isConnected**, will be faster.

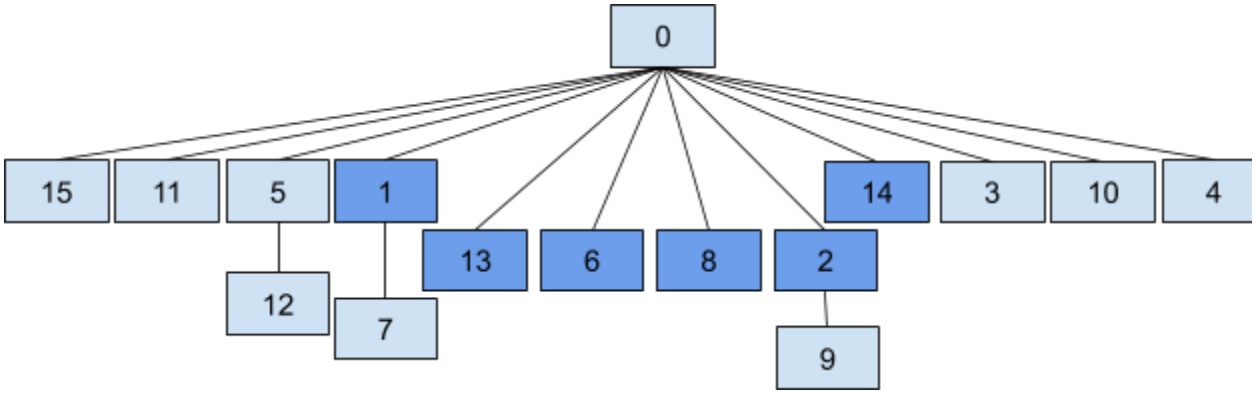
If, for example, we want to call **isConnected(11, 10)**, then instead of going up several parents to find out, we only check the parent of each value once, as follows:

1- Check parent of 11 = 0
2- Check parent of 10 = 0

And done!

Assume we call **isConnected(14, 13):**

We will go through several iterations, however we will be changing our tree to a much simpler tree, and it will look as follows:



If we keep calling **isConnected** over and over, then all the items will become direct descendants of the root.

We call this approach: **Path Compression**

Path Compression results in a union/connected operations that are very close to amortized constant time.

Doing **M** operations on **N** nodes, will improve from **O(N+M*log(N)),** to **O(N+M* lg*N)**
lg*N is less than 5 for any realistic input

N	lg*N
1	0
2	1
4	2
16	3
65536	4
2^65536	5

A tighter bound: **O(N + M ⍺(N))**, where ⍺ is the inverse Ackermann function

$\lim_{\to}$N	⍺(N)
1	0
...	1
...	2
...	3
...	4
...	5

$2^{2^{\cdot^{\cdot}}}$

65536

To summarize

Implementation	**Runtime**
WeightedQuickUnion	O(N + M log N)
WeightedQuickUnion With Path Compression	O(N + M ⍺ (N))