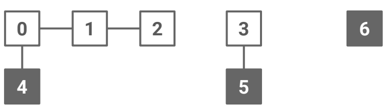
Array Representation with **Quick Find** Invariants

Before connect(2, 3) operation:

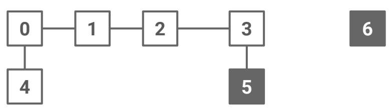
`{0, 1, 2, 4}, {3, 5}, {6}`



	0	1	2	3	4	5	6
id	4	4	4	5	4	5	6

After connect(2, 3) operation:

`{0, 1, 2, 4, 3, 5}, {6}`



	0	1	2	3	4	5	6
id	5	5	5	5	5	5	6


```
private int[] id;
boolean isConnected(int p, int q) {
  return id[p] == id[q];
}
void connect(int p, int q) {
  int setP = id[p];
  int setQ = id[q];
  for (int i=0; i<id.length; i++) {
    if (id[i] == setP)
      id[i] = setQ;
  }
}
```

Quick Find Analysis

If we have **V** vertices…

- **E** isConnected calls, each O(1).
- **V** connect calls, each O(**V**).

Simple graph: **E** < **V**^{2}.

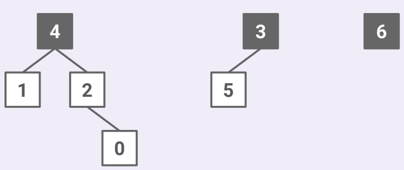
Kruskal's: O(E log V + **E** + **V**^{2})
= O(**E** log **V** + **V**^{2})

Both operations need to be O(log **V**)!

Q Improving the connect Operation

Quick Union invariant. For each **v**, parent[**v**] is the parent of **v**.

Show the result after calling connect(5, 0).

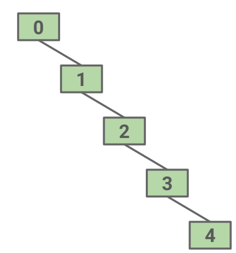


	0	1	2	3	4	5	6
parent	2	4	4	3	4	3	6

Worst-Case Height Trees

Spindly tree: repeatedly connect the first item's tree below the second item's tree.

- connect(4, 3)
- connect(3, 2)
- connect(2, 1)
- connect(1, 0)



Worst-case runtime for **both** connect and isConnected is Θ(N).

```java
private int find(int p) {
  while (p != parent[p])
    p = parent[p];
  return p;
}
boolean isConnected(int p, int q) {
  return find(p) == find(q);
}
void connect(int p, int q) {
  int i = find(p);
  int j = find(q);
  parent[i] = j;
}
```

Naive Quick Union Analysis

If we have **V** vertices…

- **E** isConnected calls, each O(**V**).
- **V** connect calls, each O(**V**).

Kruskal's: O(E log V + **EV** + **V**^{2})
= O(**E** log **V** + **EV** + **V**^{2})
= O(**EV** + **V**^{2})

Worst case is slower than Quick Find!


```java
private int find(int p) {
  while (p != parent[p])
    p = parent[p];
  return p;
}
boolean isConnected(int p, int q) {
  return find(p) == find(q);
}
void connect(int p, int q) {
  int i = find(p);
  int j = find(q);
  parent[i] = j;
}
```

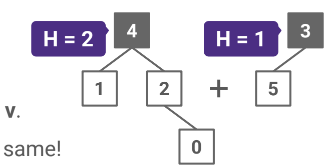
Naive Quick Union Analysis

Hypothesis (from B-Trees). Unbalanced growth leads to worst-case height trees.

Identify (different due to parent pointers). When connecting, the second item's tree always becomes the new root.

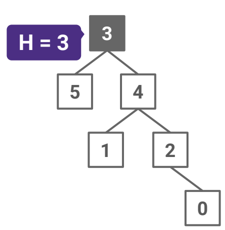
Plan. Choose the new root based on a metric such as tree height.

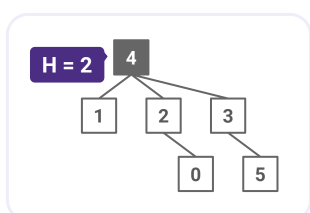
Weighted Quick Union by Height



Quick Union invariant. For each **v**, parent[**v**] is the parent of **v**.

The result of connect(5, 0) and connect(0, 5) should be the same!





Describe how to construct a worst-case height tree given a weighted quick union by height.

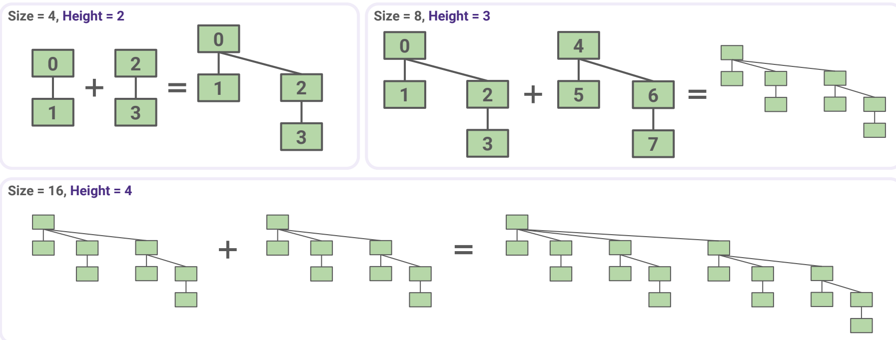
Join by Web

1. Go to **PollEv.com**
2. Enter **KEVINL**
3. Respond to activity

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

A WQUByHeight: Worst-Case Height Tree




```
void connect(int p, int q) {
  int i = find(p);
  int j = find(q);
  if (i == j) return;
  if (height[i] < height[j])
    parent[i] = j;
  else if (height[i] > height[j])
    parent[j] = i;
  else { // heights are equal
    parent[j] = i;
    height[i] += 1;
  }
}
```

WQUByHeight Analysis

Keep track of heights with an extra array.

Worst-case height is log(**V**)!

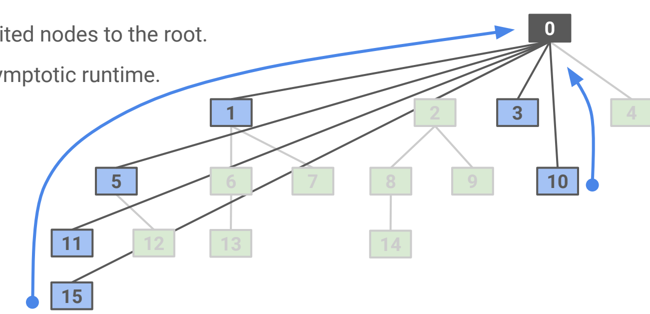
- **E** isConnected calls, each O(log **V**).
- **V** connect calls, each O(log **V**).

Kruskal's: O(E log V + **E** log **V** + **V** log **V**)
= O(**E** log **V** + **V** log **V**)
= O(**E** log **V**) if **E** > **V**

Weighted Quick Union with Path Compression

Tie all visited nodes to the root.

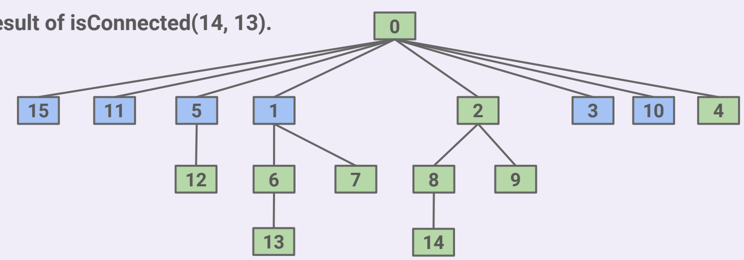
Same asymptotic runtime.



Q Weighted Quick Union with Path Compression

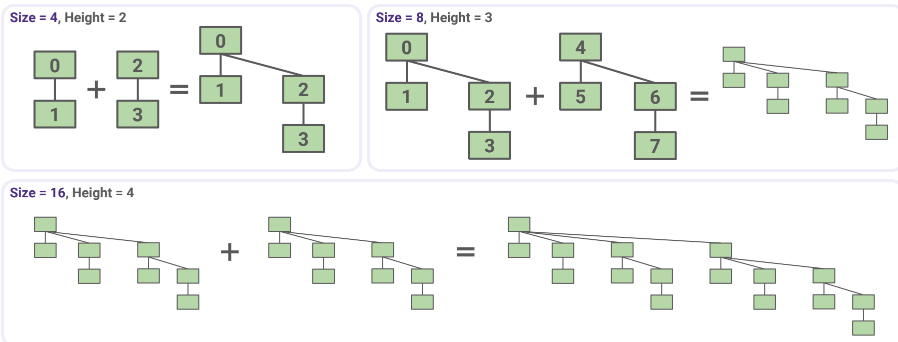
Tie all visited nodes to the root.

Draw result of isConnected(14, 13).



WQUBySize: Worst-Case Height Tree

Worst-case analysis still works when we track subtree **size**, rather than subtree height!




```
void connect(int p, int q) {
  int i = find(p);
  int j = find(q);
  if (i == j) return;
  if (size[i] < size[j]) {
    parent[i] = j;
    size[j] += size[i];
  } else {
    parent[j] = i;
    size[i] += size[j];
  }
}
```

WQUBySize Analysis

Keep track of **sizes** with an extra array.

Worst-case height is $\log(V)$!

- **E** isConnected calls, each $O(\log V)$.
- **V** connect calls, each $O(\log V)$.

Kruskal's: $O(E \log V + E \log V + V \log V)$
$= O(E \log V + V \log V)$
$= O(E \log V)$ if $E > V$


```
private int find(int p) {
  int root = p;
  while (root != parent[root])
    root = parent[root];
  while (p != root) {
    int newP = parent[p];
    parent[p] = root;
    p = newP;
  }
  return root;
}
```

WQUPathCompression

WQUBySize with Path Compression.

Worst-case height is **log***(**V**), where log* is the **iterated logarithm**–nearly constant.

- **E** isConnected calls, each $O(\log^* V)$.
- **V** connect calls, each $O(\log^* V)$.

e.g. $\log^*(2^{65536}) = 5$.

Analysis is out of scope.

Kruskal's: $O(E \log V + E \log^* V + V \log^* V)$
$= O(E \log V)$ if $E > V$

Summary

Disjoint Sets ADT is used to track connected components in Kruskal's algorithm.

Graph algorithm runtime can depend on efficient data structure implementations.

Quick Find: Array representation with no tree structure. Fast isConnected, slow connect.

Quick Union: Array representation with tree structure. Worst-case linear-height trees.

Weighted Quick Union: Choose the new root strategically based on a metric.

- **WQUByHeight**: Use subtree height as a metric. Results in log **V** height.
- **WQUBySize**: Use subtree size as a metric. Results in log **V** height.
- **WQUPathCompression**:
  Use subtree size as a metric. Results in log* **V** height–nearly constant.