

Announcements

Project 2, Phase 2 due tonight (with 24 hour grace period).

Project 2 demos this are this week:

- See [demo link in project 2 spec](#) for the exact script we'll use for demos.
- You will let us know which commit to use from github.
 - Submissions from March 7th or later will incur a penalty on your 80 demo points. Will be done on a per day basis, e.g. submissions from March 7th will be 10% off, from March 8th will be 20% off, etc.
- Gold points are not part of the lab demo.

Gold points:

- No late penalty for submitting by March 9th.

CS61B



Lecture 20: Disjoint Sets

- Dynamic Connectivity and the Disjoint Sets Problem
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression (CS170 Preview)

Meta-goals of the Coming Lectures: Data Structure Refinement

Project 2: A chance to see how a design evolves.

Next couple of weeks: Deriving classic solutions to interesting problems, with an emphasis on how set, map, and priority queue ADTs are implemented.

- Today: A chance to see how an implementation of an ADT can evolve and how different underlying data structures affect asymptotic runtime (using our formal notation).

Today's Goal: Dynamic Connectivity

Today: A case study in ADT implementation.

Goal: Given a series of pairwise connectedness declarations, determine if two items are connected transitively. Only care about yes vs. no.

- Example: We have Mexico, USA, Canada, Ukraine, and Estonia.
 - USA is connected to Mexico.
 - USA is connected to Canada.
 - Is Mexico connected to Canada? **Yes**.
 - Ukraine is connected to Estonia.
 - Is Ukraine connected to USA? **No**.

Solvable using a simple ADT with cool implementation details.

The Dynamic Connectivity Problem

Goal: Given a series of pairwise integers connectedness declarations, determine if two integers (or items) are connected. Two operations:

- `connect(p, q)`: Connect items `p` and `q`.
- `isConnected(p, q)`: Are `p` and `q` connected?

`connect(0, 1)`

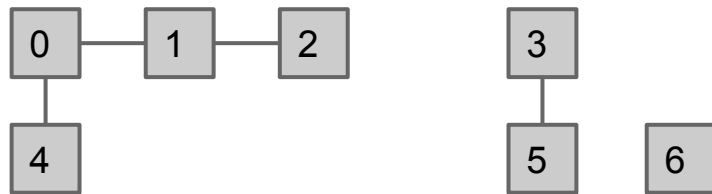
`connect(1, 2)`

`connect(0, 4)`

`connect(3, 5)`

`isConnected(2, 4)`: **true**

`isConnected(3, 0)`: **false**

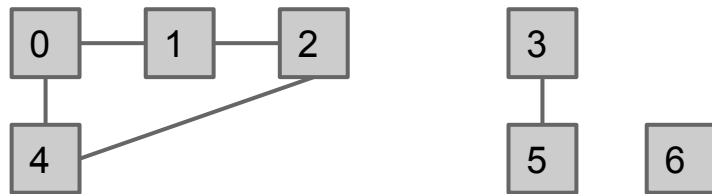


The Dynamic Connectivity Problem

Goal: Given a series of pairwise integers connectedness declarations, determine if two integers (or items) are connected. Two operations:

- `connect(p, q)`: Connect items `p` and `q`.
- `isConnected(p, q)`: Are `p` and `q` connected?

```
connect(0, 1)
connect(1, 2)
connect(0, 4)
connect(3, 5)
isConnected(2, 4): true
isConnected(3, 0): false
connect(4, 2)
```

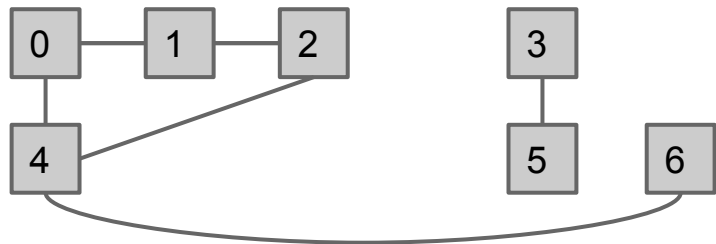


The Dynamic Connectivity Problem

Goal: Given a series of pairwise integers connectedness declarations, determine if two integers (or items) are connected. Two operations:

- `connect(p, q)`: Connect items `p` and `q`.
- `isConnected(p, q)`: Are `p` and `q` connected?

```
connect(0, 1)
connect(1, 2)
connect(0, 4)
connect(3, 5)
isConnected(2, 4): true
isConnected(3, 0): false
connect(4, 2)
connect(4, 6)
```

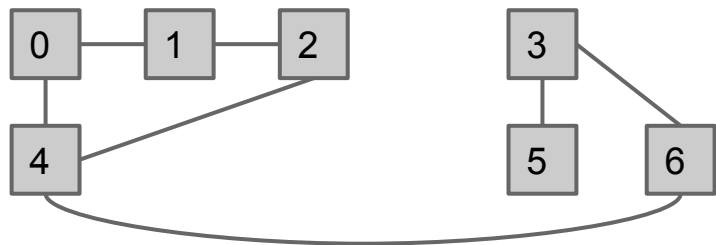


The Dynamic Connectivity Problem

Goal: Given a series of pairwise integers connectedness declarations, determine if two integers (or items) are connected. Two operations:

- `connect(p, q)`: Connect items `p` and `q`.
- `isConnected(p, q)`: Are `p` and `q` connected?

```
connect(0, 1)
connect(1, 2)
connect(0, 4)
connect(3, 5)
isConnected(2, 4): true
isConnected(3, 0): false
connect(4, 2)
connect(4, 6)
connect(3, 6)
```

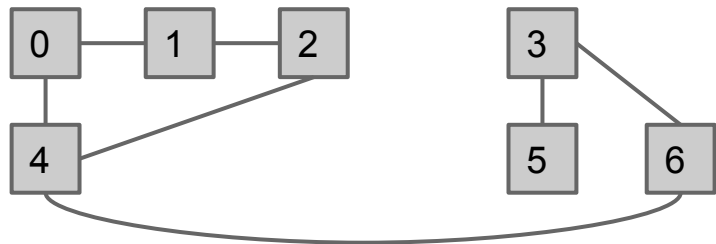


The Dynamic Connectivity Problem

Goal: Given a series of pairwise integers connectedness declarations, determine if two integers (or items) are connected. Two operations:

- `connect(p, q)`: Connect items `p` and `q`.
- `isConnected(p, q)`: Are `p` and `q` connected?

```
connect(0, 1)
connect(1, 2)
connect(0, 4)
connect(3, 5)
isConnected(2, 4): true
isConnected(3, 0): false
connect(4, 2)
connect(4, 6)
connect(3, 6)
isConnected(3, 0): true
```



The Disjoint Sets ADT

```
public interface DisjointSets {  
    /** Connects two items P and Q. */  
    void connect(int p, int q);  
  
    /** Checks to see if two items are connected. */  
    boolean isConnected(int p, int q);  
}
```

connect(int p, int q)

isConnected(int p, int q)

Goal: Design an efficient DisjointSets implementation.

- Number of elements N can be huge.
- Number of method calls M can be huge.
- Calls to methods may be interspersed (e.g. can't assume that we stop getting connect calls after some point).

The Naive Approach

Naive approach:

- Connecting two things: Record every single connecting line in some data structure.
- Checking connectedness: Do some sort of (??) iteration over the lines to see if one thing can be reached from the other.



A Better Approach: Connected Components

Rather than manually writing out every single connecting line, record the sets that something belongs to.

	$\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
<code>connect(0, 1)</code>	$\{0, 1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
<code>connect(1, 2)</code>	$\{0, 1, 2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
<code>connect(0, 4)</code>	$\{0, 1, 2, 4\}, \{3\}, \{5\}, \{6\}, \{7\}$
<code>connect(3, 5)</code>	$\{0, 1, 2, 4\}, \{3, 5\}, \{6\}, \{7\}$
<code>isConnected(2, 4):</code>	<code>true</code>
<code>isConnected(3, 0):</code>	<code>false</code>
<code>connect(4, 2)</code>	$\{0, 1, 2, 4\}, \{3, 5\}, \{6\}, \{7\}$
<code>connect(4, 6)</code>	$\{0, 1, 2, 4, 6\}, \{3, 5\}, \{7\}$
<code>connect(3, 6)</code>	$\{0, 1, 2, 3, 4, 5, 6\}, \{7\}$
<code>isConnected(3, 0):</code>	<code>true</code>

A Better Approach: Connected Components

A ***connected component*** is a maximal set of items that are mutually connected.

- Naive approach: Record every single connecting line somehow.
- Better approach: Model connectedness in terms of sets.
 - How things are connected isn't something we need to know.

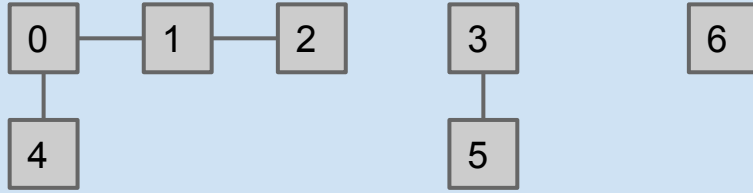


$\{ 0, 1, 2, 4 \}, \{ 3, 5 \}, \{ 6 \}$

Quick Find

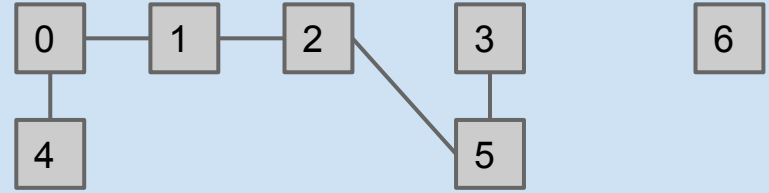
Challenge: Pick Data Structures to Support Tracking of Sets

Before connect(2, 5) operation:



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

After connect(2, 5) operation:

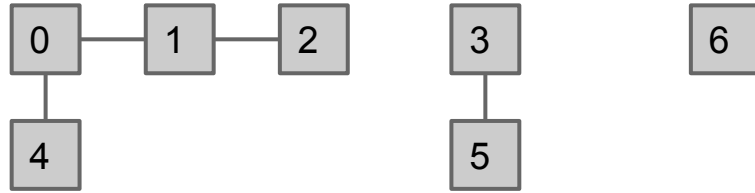


{ 0, 1, 2, 4, 3, 5 }, {6}

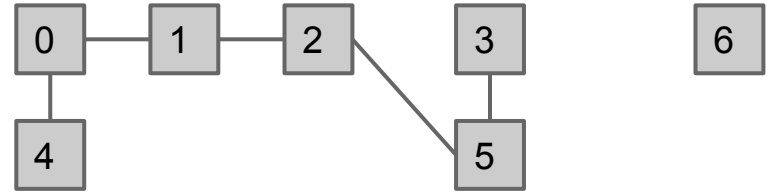
Assume elements are numbered from 0 to N-1.

Challenge: Pick Data Structures to Support Tracking of Sets

Before connect(2, 5) operation:



After connect(2, 5) operation:



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

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

Idea #1:

A map from integers is also an array.

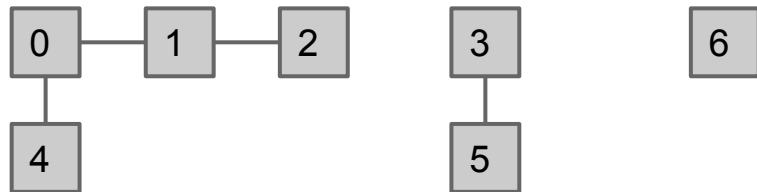
- Map<Integer, SetGuy>, where the Integer is the item in question.
 - map.get(1) ← return a reference to {0, 1, 2, 4}

Idea #2: List<HashSet>, where we move things around between sets

- Requires iterating through all the hashsets to find something.

Using an Array

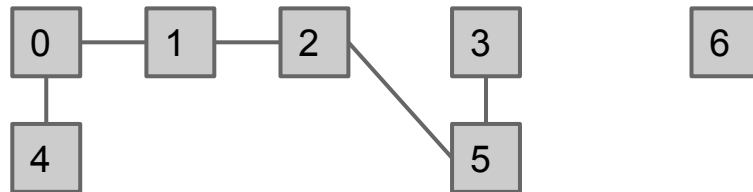
Before connect(2, 5) operation:



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

int[] id	0	0	0	3	0	3	6
	0	1	2	3	4	5	6

After connect(2, 5) operation:



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

int[] id	3	3	3	3	3	3	6
	0	1	2	3	4	5	6

One natural choice: `int[]` where `ith` entry gives set number of item `i`.

- `connect(p, q)`: Change entries that equal `id[p]` to `id[q]`

QuickFindDS

```
public class QuickFindDS implements DisjointSets {  
    private int[] id;
```

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

Very fast: Two array accesses.

```
    public void connect(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;  
            }  
        }  
    }...}
```

Relatively slow: $N+2$ to $2N+2$ array accesses.

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

Performance Summary

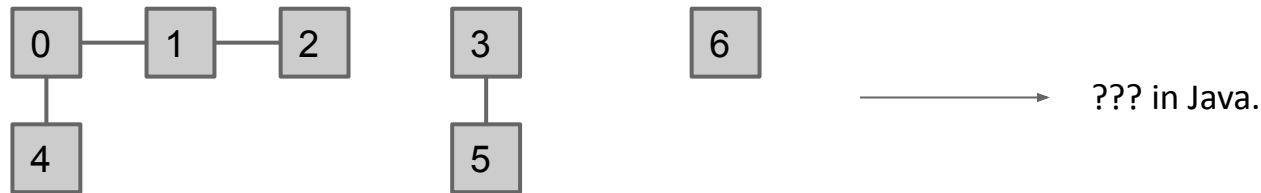
Implementation	constructor	connect	isConnected
QuickFindDS	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$

QuickFindDS is too slow: Connecting two items takes N time.

Quick Union

Improving the Connect Operation

Approach zero: Represent everything as boxes and lines. This was overkill.



Approach one: Represent everything as connected components. Represented connected components as an array.

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

int[] id

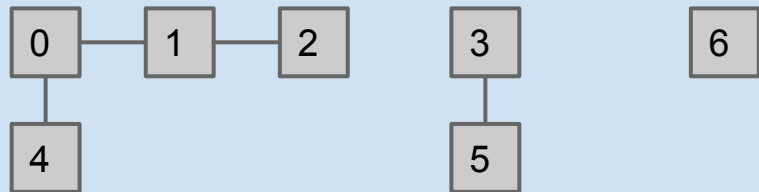
0	0	0	3	0	3	6
0	1	2	3	4	5	6

Approach two: We're still going to stick with connected components, but will represent connected components differently.

{ 0, 1, 2, 4 }, {3, 5}, {6} → ??? in Java.

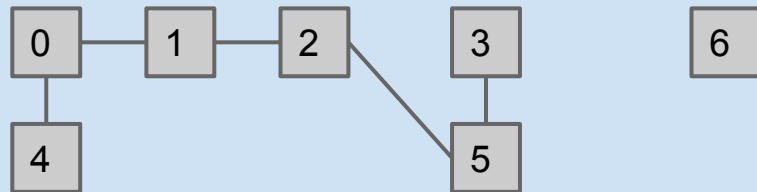
Improving the Connect Operation

A hard question: How could we change our set representation so that combining two sets into their union requires changing **one** value?



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

int[] id	0	0	0	3	0	3	6
	0	1	2	3	4	5	6

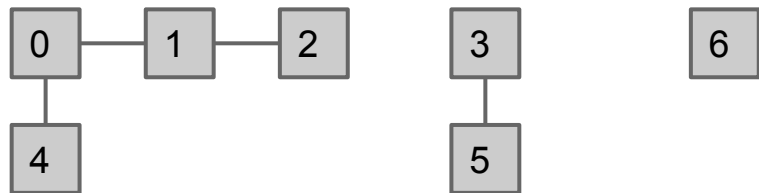


{ 0, 1, 2, 4, 3, 5 }, {6}

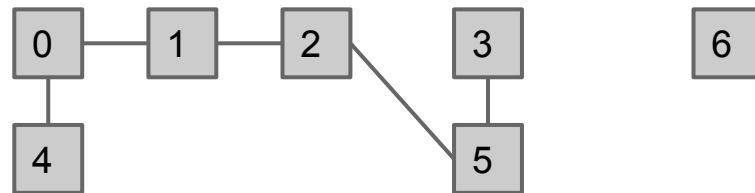
int[] id	3	3	3	3	3	3	6
	0	1	2	3	4	5	6

Improving the Connect Operation

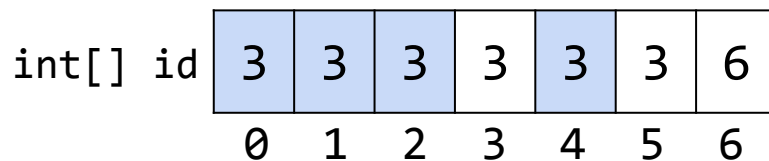
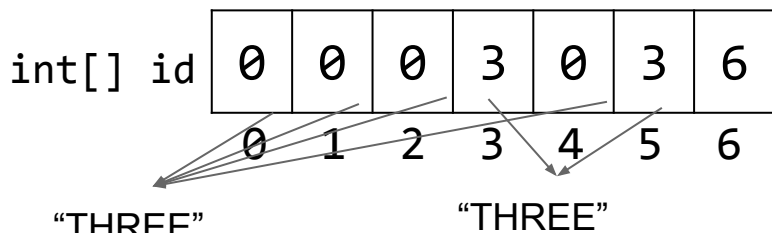
A hard question: How could we change our set representation so that combining two sets into their union requires changing **one** value?



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



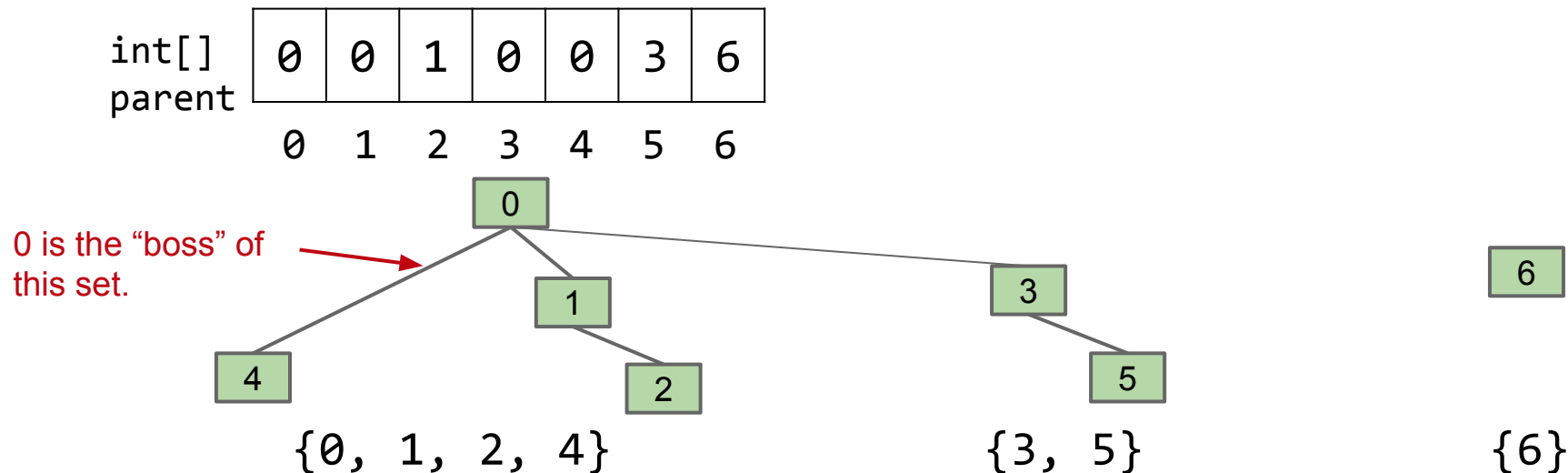
{ 0, 1, 2, 4, 3, 5 }, {6}



Improving the Connect Operation

Possibly harder question:

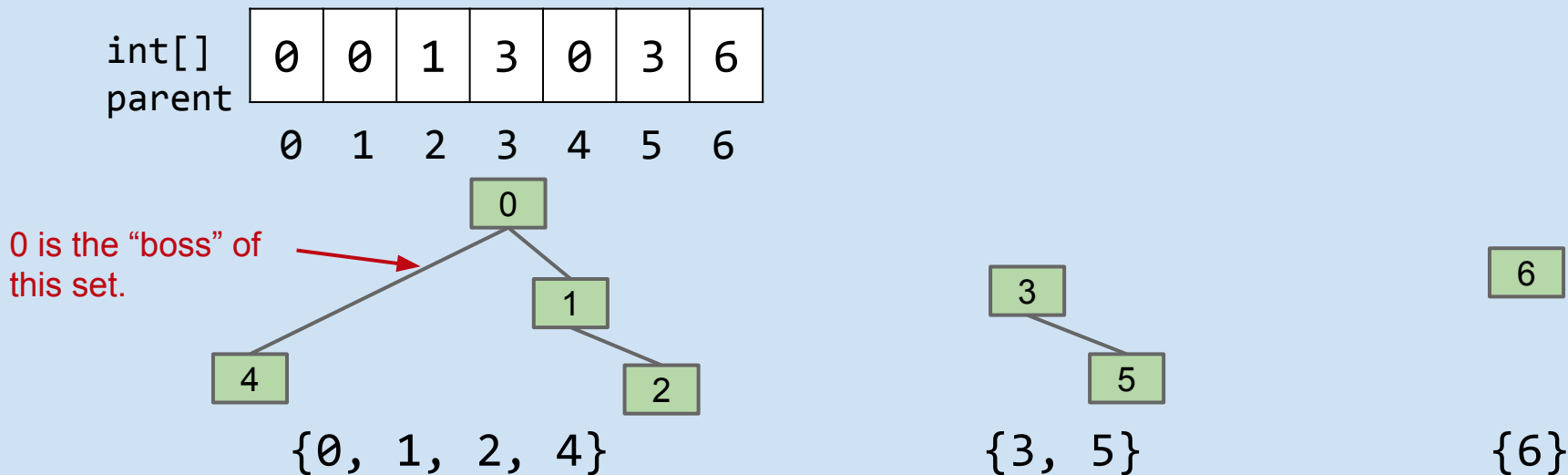
- Knowing that we only need to support the union/belongs operations, how can we represent a set such that the set union operation is very fast?
- Idea: Assign each node a parent (instead of an id).
 - An innocuous sounding, seemingly arbitrary solution.
 - Unlocks a pretty amazing universe of math that we won't discuss.



Improving the Connect Operation

connect(5, 2)

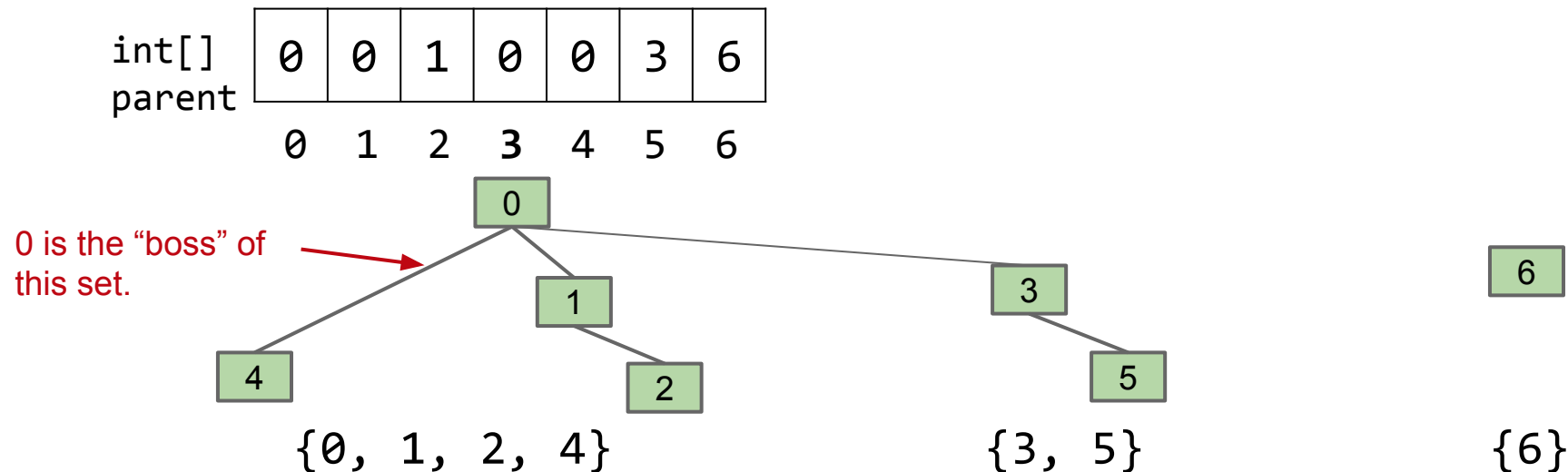
- How do we do this?
 - If you're not sure where to start, consider: why can't we just set id[5] to 2?



Improving the Connect Operation

connect(5, 2)

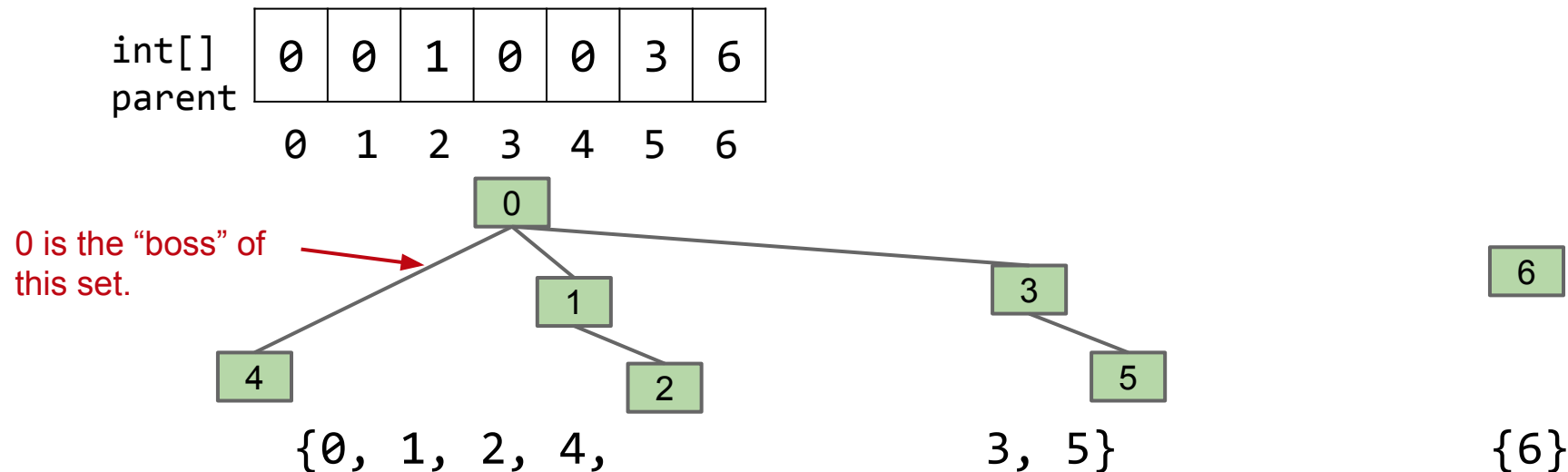
- How do we do this?
 - Find the boss of 5. ← this isn't free!
 - Find the boss of 2.
 - Change the value of the boss of 5 to boss of 2?



Improving the Connect Operation

connect(5, 2)

- How do we do this?
 - If you're not sure where to start, consider: why can't we just set `id[5]` to 2?
 - make `root(5)` into a child of `root(2)`.



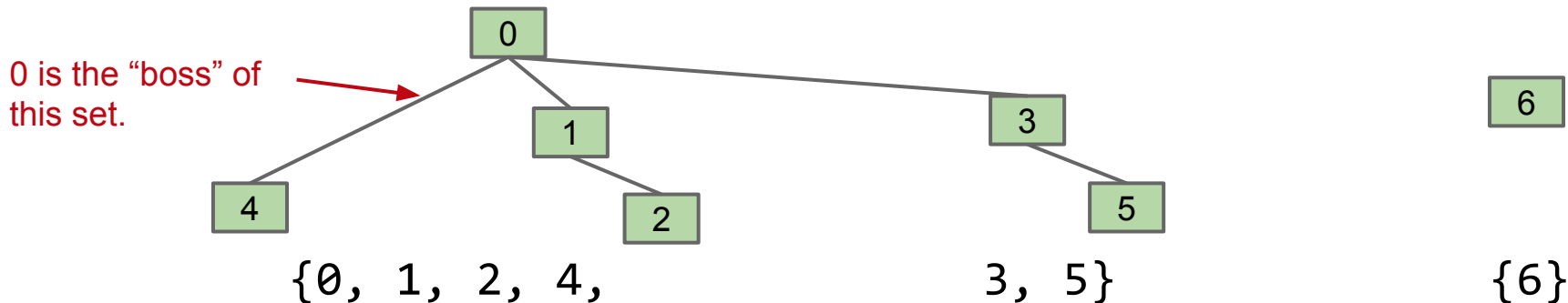
Set Union Using Rooted-Tree Representation

connect(5, 2)

- Make root(5) into a child of root(2).

int[]	0	0	1	0	0	3	6
parent	0	1	2	3	4	5	6

What are the potential performance issues with this approach?



Set Union Using Rooted-Tree Representation

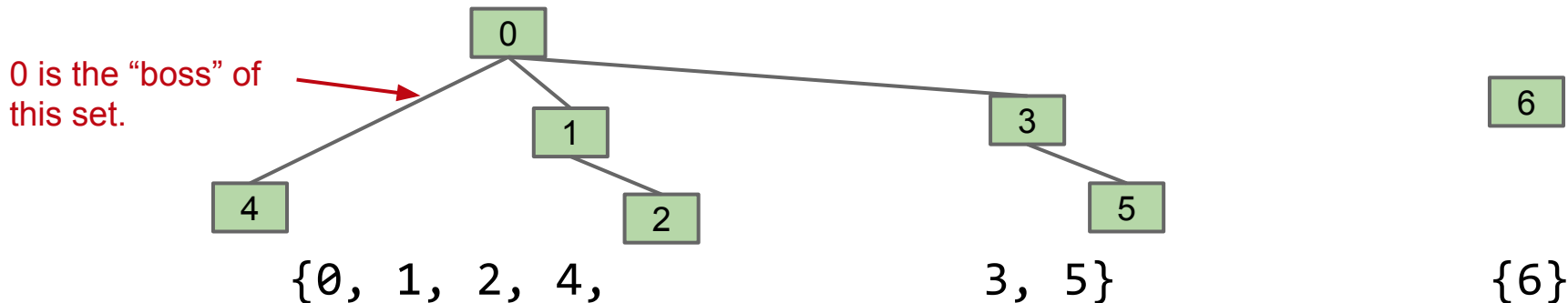
connect(5, 2)

- Make root(5) into a child of root(2).

int[]	0	0	1	0	0	3	6
parent	0	1	2	3	4	5	6

What are the potential performance issues with this approach?

- Tree can get too tall!



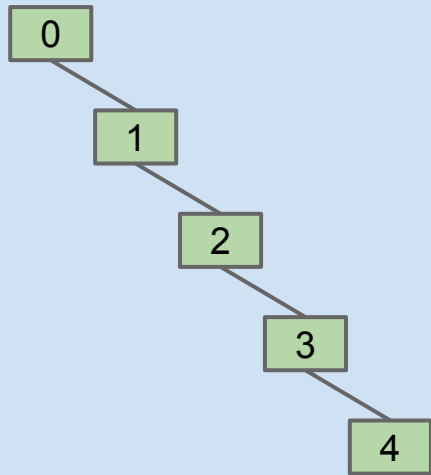
The Worst Case

If we always connect the first item's tree below the second item's tree, we can end up with a tree of height M after M operations:

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

For N items, what's the worst case runtime...

- For `connect(p, q)`?
- For `isConnected(p, q)`?



QuickUnionDS

```
public class QuickUnionDS implements DisjointSets {  
    private int[] parent;  
    public QuickUnionDS(int N) {  
        parent = new int[N];  
        for (int i = 0; i < N; i++)  
            parent[i] = i;  
    }
```

For N items, this means a worst case runtime of $\Theta(N)$.



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

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

```
public void connect(int p, int q) {  
    int i = find(p);  
    int j = find(q);  
    parent[i] = j;  
}
```

Performance Summary

Implementation	Constructor	connect	isConnected
QuickFindDS	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$
QuickUnionDS	$\Theta(N)$	$O(N)$	$O(N)$

QuickFindDS defect: QuickFindDS is too slow: Connecting two items takes N time in the worst case.

QuickUnion defect: Trees can get tall. Results in potentially even worse performance than QuickFind if tree is imbalanced.

Weighted Quick Union

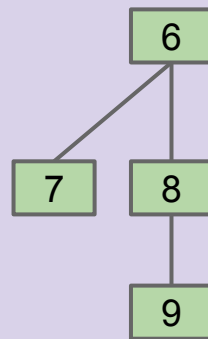
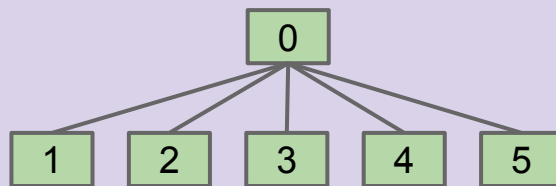
Weighted QuickUnion: <http://yellkey.com/analysis>

Modify quick-union to avoid tall trees.

- Track tree size (**number** of elements).
- New rule: Always link root of *smaller* tree *to larger* tree.

New rule: If we call `connect(3, 8)`, which entry (or entries) of `parent[]` changes?

- A. `parent[3]`
- B. `parent[0]`
- C. `parent[8]`
- D. `parent[6]`



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

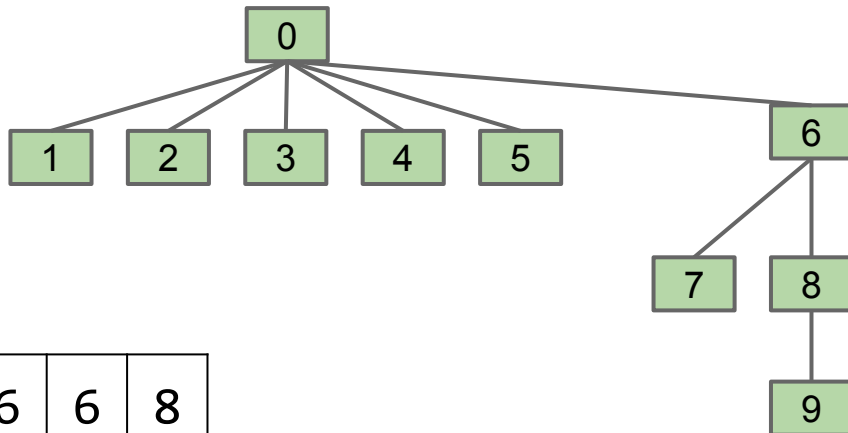
Improvement #1: Weighted QuickUnion

Modify quick-union to avoid tall trees.

- Track tree size (**number** of elements).
- New rule: Always link root of *smaller* tree *to larger* tree.

New rule: If we call `connect(3, 8)`, which entry (or entries) of `parent[]` changes?

- A. `parent[3]`
- B. `parent[0]`
- C. `parent[8]`
- D. **`parent[6]`**



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

Implementing WeightedQuickUnion

Minimal changes needed:

- Use `parent[]` array as before, but also add `size[]` array.
- `isConnected(int p, int q)` requires no changes.
- `connect(int p, int q)` needs two changes:
 - Link root of smaller tree to larger tree.
 - Update `size[]` array.

Now the connect method looks like:

```
public 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]; }  
}
```

int[]
parent

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

size

10	1	1	1	1	1	4	1	2	1
0	1	2	3	4	5	6	7	8	9

WeightedQuickUnion Performance

As before, connect and isConnected require time proportional to depth of the items involved.

Max depth of any item: $\log N$

Very brief proof for the curious (not covered in lecture):

- Depth of an element x increases only when tree T_1 that contains x is linked below some other tree T_2 .
 - The size of the tree at least doubles since $\text{weight}(T_2) \geq \text{weight}(T_1)$.
 - Tree containing x doubles at most $\log N$ times.

Performance Summary

Implementation	Constructor	connect	isConnected
QuickFindDS	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$
QuickUnionDS	$\Theta(N)$	$O(N)$	$O(N)$
WeightedQuickUnionDS	$\Theta(N)$	$O(\log N)$	$O(\log N)$

By tweaking QuickUnionDS we've achieved logarithmic time performance.

- Fast enough for most (all?) practical problems.

Performance Summary

Implementation	Constructor	connect	isConnected
QuickFindDS	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$
QuickUnionDS	$\Theta(N)$	$O(N)$	$O(N)$
WeightedQuickUnionDS	$\Theta(N)$	$O(\log N)$	$O(\log N)$

Performing M operations on a DisjointSet object with N elements:

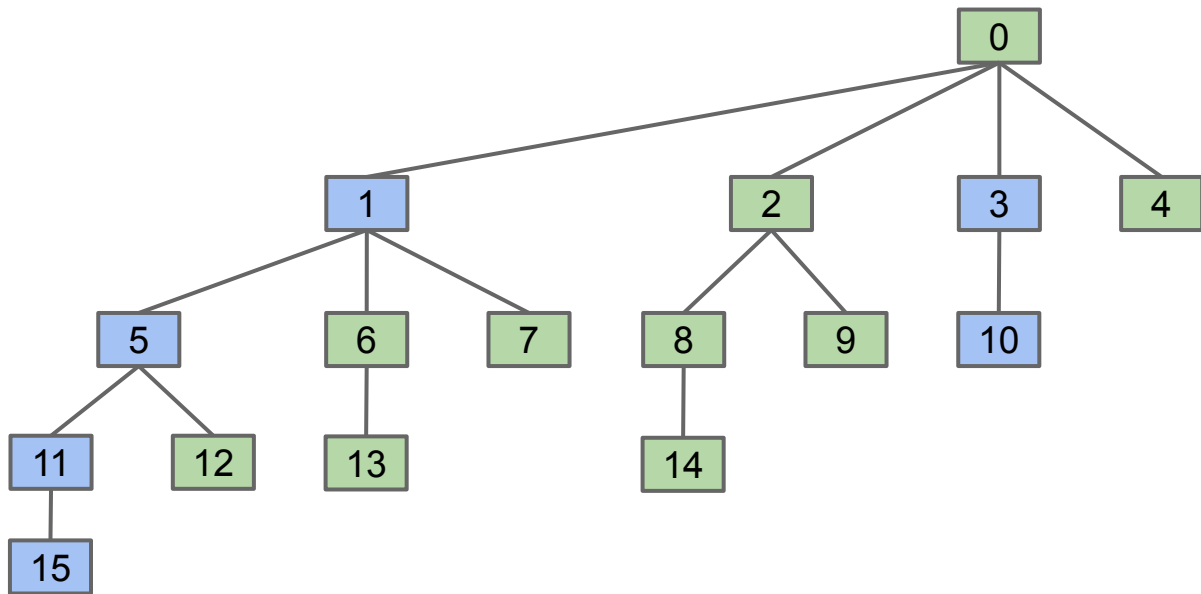
- Runtime goes from $O(MN)$ to $O(N + M \log N)$
- For $N = 10^9$ and $M = 10^9$, time to run goes from 30 years to 6 seconds.
 - Key point: Good data structure unlocks solutions to problems that could otherwise not be solved!
- ... pretty good for most problems, but could we do better?

Path Compression (CS170 Spoiler)

170 Spoiler: Path Compression: A Clever Idea

Below is the topology of the worst case if we use WeightedQuickUnion.

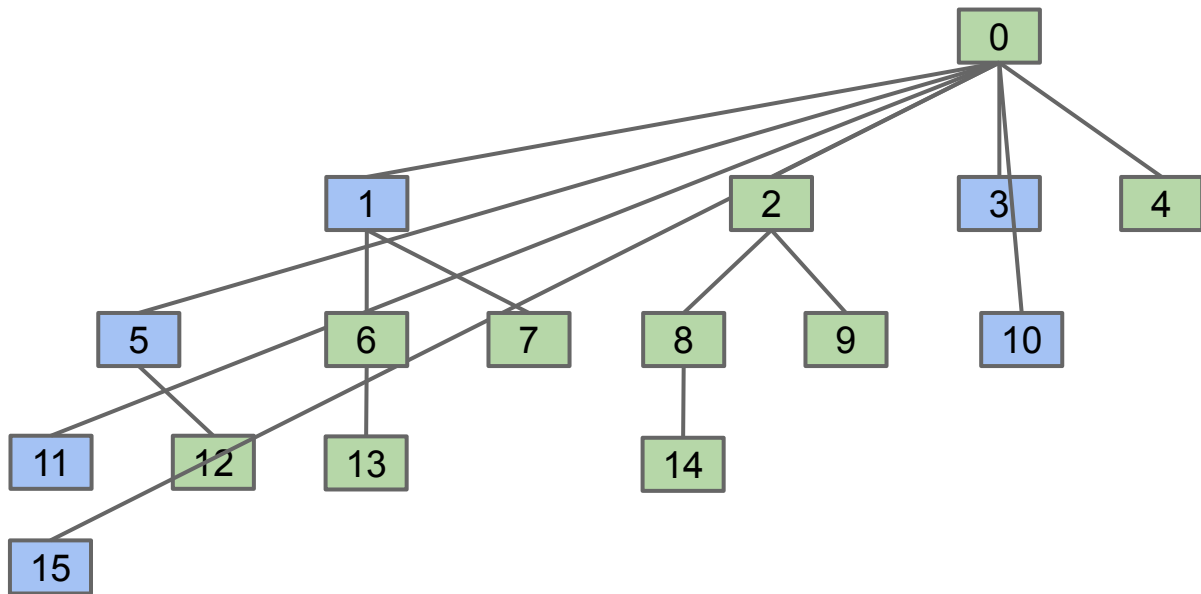
- Clever idea: When we do `isConnected(15, 10)`, tie all nodes seen to the root.
 - Additional cost is insignificant (same order of growth).



170 Spoiler: Path Compression: A Clever Idea

Below is the topology of the worst case if we use WeightedQuickUnion

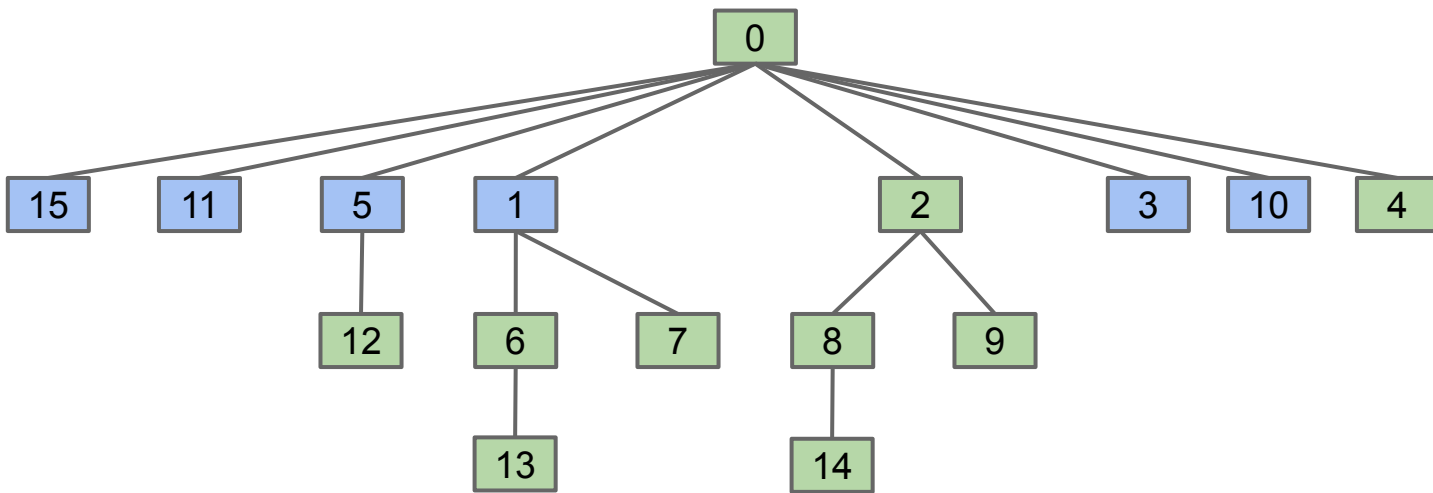
- Clever idea: When we do `isConnected(15, 10)`, tie all nodes seen to the root.
 - Additional cost is insignificant (same order of growth).



170 Spoiler: Path Compression: A Clever Idea

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

- M operations on N nodes is $O(N + M \lg^* N)$ - you will see this in CS170.
- \lg^* is less than 5 for any realistic input.

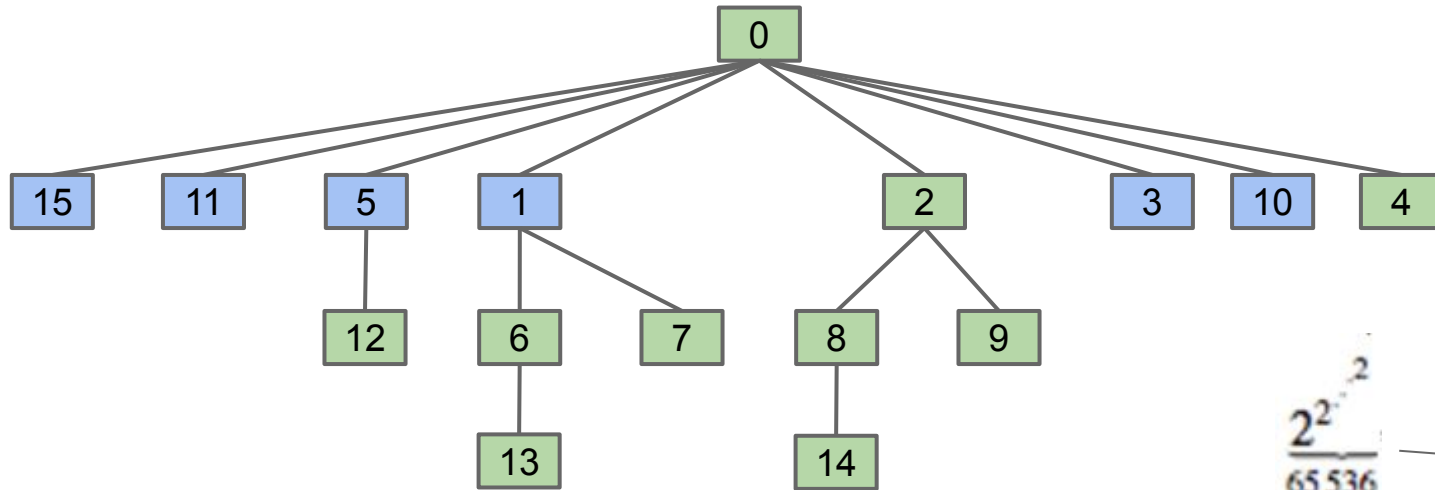


N	$\lg^* N$
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

170 Spoiler: Path Compression: A Clever Idea

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

- M operations on N nodes is $O(N + M \lg^* N)$ - you will see this in CS170.
- A tighter bound: $O(N + M \alpha(N))$, where α is the inverse Ackermann function.
- The inverse ackermann function is less than 5 for all practical inputs!



N	$\alpha(N)$
1	0
...	1
...	2
...	3
...	4
...	5

$2^{2^{2^2}}$
65536

170 Spoiler: Path Compression: A Clever Idea

- And we're done! The end result of our iterative design process is the standard way disjoint sets are implemented today - quick union and path compression.
- The resulting code for find() is simple:

```
private int find(int p) {  
    if (p == parent[p]) {  
        return p;  
    } else {  
        parent[p] = find(parent[p]);  
        return parent[p];  
    }  
}
```

Everything All Together...

```
public class WeightedQuickUnionDSWithPathCompression implements DisjointSets {
    private int[] parent; private int[] size;
    public WeightedQuickUnionDSWithPathCompression(int N) {
        parent = new int[N]; size = new int[N];
        for (int i = 0; i < N; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    private int find(int p) {
        if (p == parent[p]) {
            return p;
        } else {
            parent[p] = find(parent[p]);
            return parent[p];
        }
    }
}
```

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

    public 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];
        }
    }
}
```

Performance Summary

Implementation	Runtime
QuickFindDS	$\Theta(NM)$
QuickUnionDS	$O(NM)$
WeightedQuickUnionDS	$O(N + M \log N)$
WeightedQuickUnionDSWithPathCompression	$O(N + M \alpha(N))$

Runtimes are given assuming:

- We have a DisjointSets object of size N .
- We perform M operations, where an operation is defined as either a call to `connected` or `isConnected`.

Citations

Nazca Lines:

http://redicecreations.com/ul_img/24592nazca_bird.jpg

Implementation code adapted from Algorithms, 4th edition and Professor Jonathan Shewchuk's lecture notes on disjoint sets, where he presents a faster one-array solution. I would recommend taking a look.

(<http://www.cs.berkeley.edu/~jrs/61b/lec/33>)

The proof of the inverse ackermann runtime for disjoint sets is given here:

http://www.uni-trier.de/fileadmin/fb4/prof/INF/DEA/Uebungen_LVA-Ankuendigungen/ws07/KAuD/effi.pdf

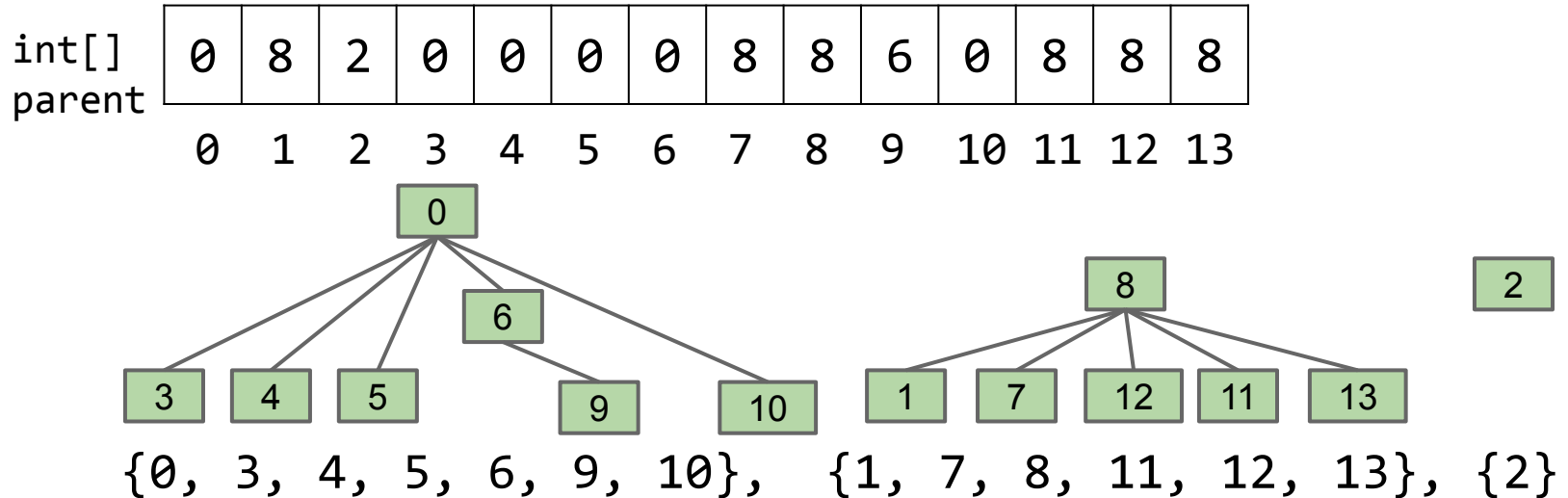
as originally proved by Tarjan here at UC Berkeley in 1975.

extra

Set Union Using Parent Representation

connect(11, 3)

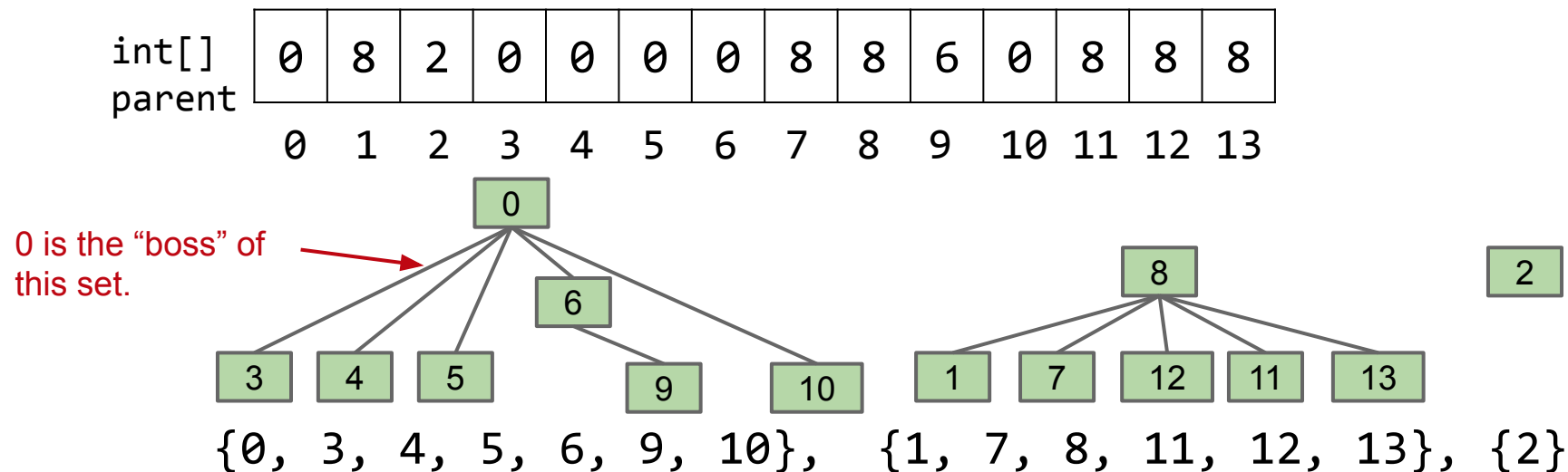
- Any ideas how to do this quickly?



Improving the Connect Operation

Possibly harder question:

- Knowing that we only need to support the union/belongs operations, how can we represent a set such that the set union operation is very fast?
- Idea: Assign each node a parent (instead of an id).
 - An innocuous sounding, seemingly arbitrary solution.
 - Unlocks a pretty amazing universe of math that we won't discuss. D:



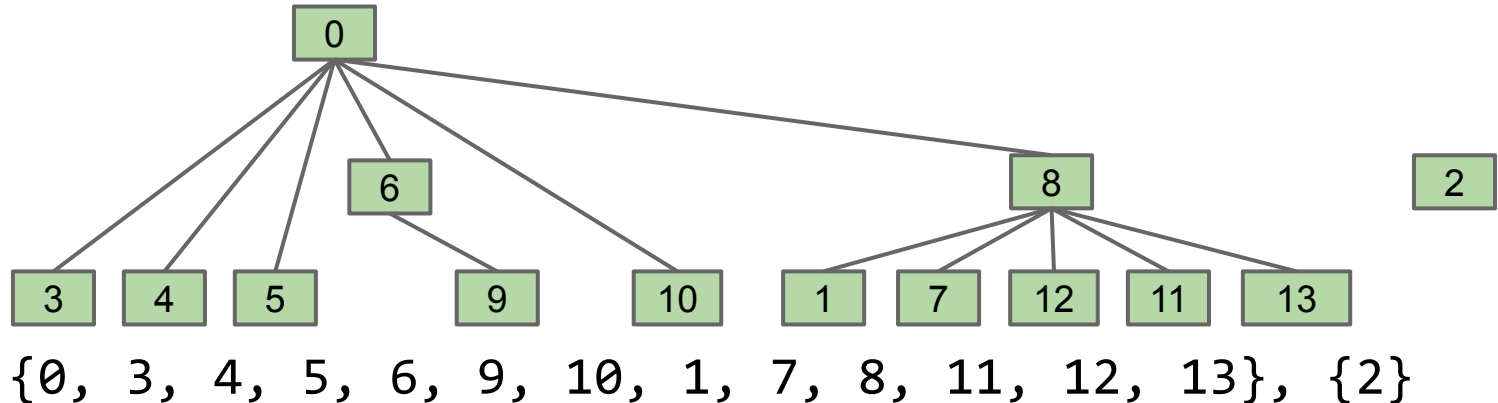
Set Union Using Rooted-Tree Representation

connect(11, 3)

- Make root(11) into a child of root(3).

int[] id	0	8	2	0	0	0	0	8	0	6	0	8	8	8
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Anybody see any issues with this?



Set Union Using Rooted-Tree Representation

connect(11, 3)

- Make root(11) into a child of root(3).

int[] id	0	8	2	0	0	0	0	8	0	6	0	8	8	8
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Anybody see any issues with this? Tree can get too tall!

