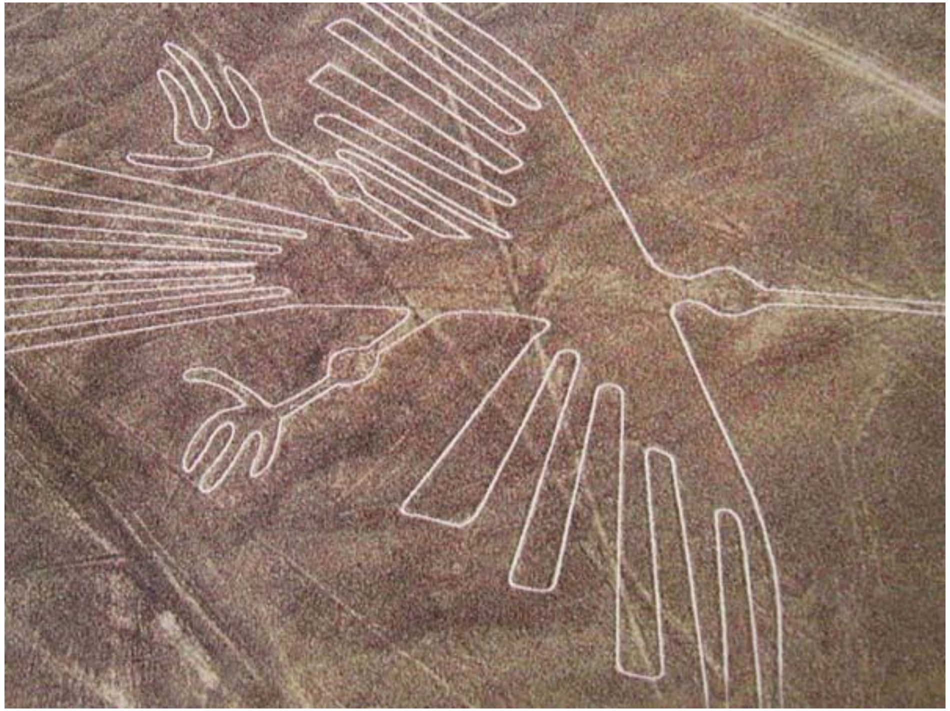
Lecture 14 (Data Structures 1)

Disjoint Sets



CS61B, Spring 2024 @ UC Berkeley

Slides credit: Josh Hug

Disjoint Sets API

Lecture 14, CS61B, Spring 2024

Introduction to Disjoint Sets

- **Disjoint Sets API**
- Tracking Connected Components

Implementations

- List of Sets
- Quick Find
- Quick Union
- Weighted Quick Union
- WQU with Path Compression

Meta-goals of the Coming Lectures: Data Structure Refinement

Next couple of weeks: Deriving classic solutions to interesting problems, with an emphasis on how sets, maps, and priority queues are implemented.

Today: Deriving the “Disjoint Sets” data structure for solving the “Dynamic Connectivity” problem. We will see:

- How a data structure design can evolve from basic to sophisticated.
- How our choice of underlying abstraction can affect asymptotic runtime (using our formal Big-Theta notation) and code complexity.

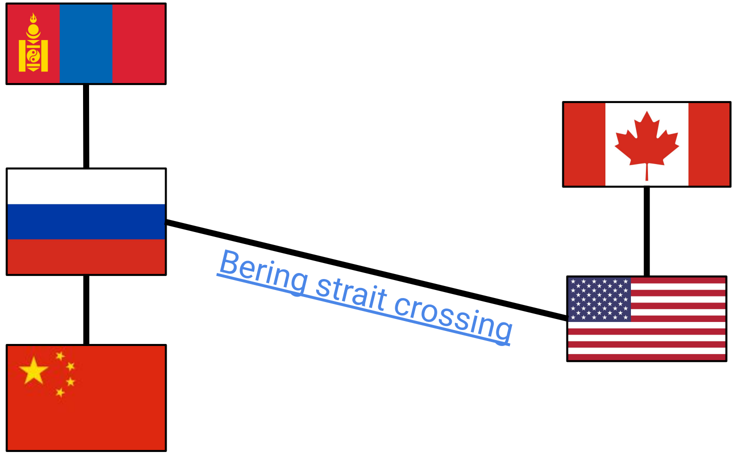
The Disjoint Sets Data Structure

The Disjoint Sets data structure has two operations:

- connect(x, y): Connects x and y.
- isConnected(x, y): Returns true if x and y are connected. Connections can be transitive, i.e. they don't need to be direct.

Example:

- connect(Russia, China)
- connect(Russia, Mongolia)
- isConnected(China, Mongolia)? `true`
- connect(USA, Canada)
- isConnected(USA, Mongolia)? `false`
- connect(Russia, USA)
- isConnected(USA, Mongolia)? `true`



CC BY NC SA

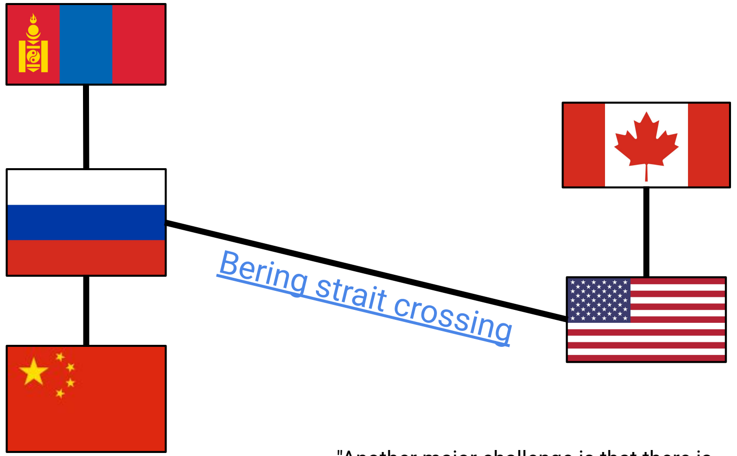
The Disjoint Sets Data Structure

The Disjoint Sets data structure has two operations:

- connect(x, y): Connects x and y.
- isConnected(x, y): Returns true if x and y are connected. Connections can be transitive, i.e. they don't need to be direct.

Useful for many purposes, e.g.:

- Percolation theory:
  - Computational chemistry.
- Implementation of other algorithms:
  - Kruskal's algorithm.



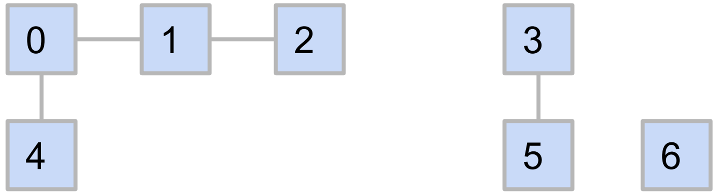
"Another major challenge is that there is nothing on either side of the Bering Strait to connect the bridge to."

Disjoint Sets on Integers

To keep things simple, we're going to:

- Force all items to be integers instead of arbitrary data (e.g. 8 instead of USA).
- Declare the number of items in advance, everything is disconnected at start.

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4): true
ds.isConnected(3, 0): false
```

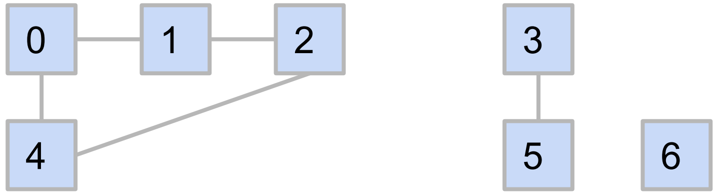


Disjoint Sets on Integers

To keep things simple, we're going to:

- Force all items to be integers instead of arbitrary data (e.g. 8 instead of USA).
- Declare the number of items in advance, everything is disconnected at start.

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4): true
ds.isConnected(3, 0): false
ds.connect(4, 2)
```



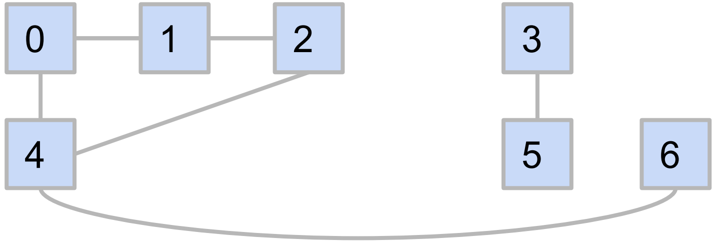
cc BY NC SA

Disjoint Sets on Integers

To keep things simple, we're going to:

- Force all items to be integers instead of arbitrary data (e.g. 8 instead of USA).
- Declare the number of items in advance, everything is disconnected at start.

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4): true
ds.isConnected(3, 0): false
ds.connect(4, 2)
ds.connect(4, 6)
```

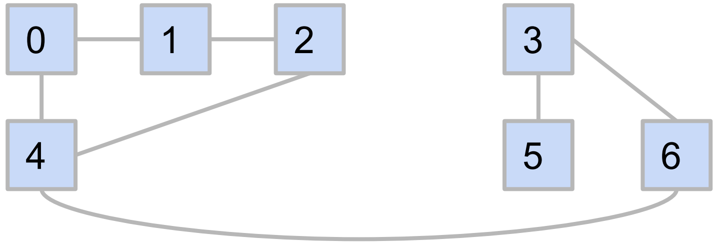


Disjoint Sets on Integers

To keep things simple, we're going to:

- Force all items to be integers instead of arbitrary data (e.g. 8 instead of USA).
- Declare the number of items in advance, everything is disconnected at start.

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4): true
ds.isConnected(3, 0): false
ds.connect(4, 2)
ds.connect(4, 6)
ds.connect(3, 6)
```

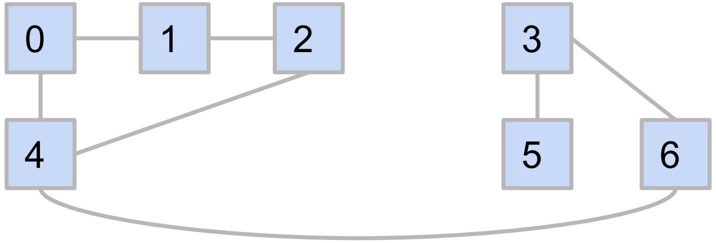


Disjoint Sets on Integers

To keep things simple, we're going to:

- Force all items to be integers instead of arbitrary data (e.g. 8 instead of USA).
- Declare the number of items in advance, everything is disconnected at start.

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4): true
ds.isConnected(3, 0): false
ds.connect(4, 2)
ds.connect(4, 6)
ds.connect(3, 6)
ds.isConnected(3, 0): true
```



The Disjoint Sets Interface

```
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);
}
```

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 it's only connect operations followed by only isConnected operations).

```
connect(int p, int q)
isConnected(int p, int q)
```

Tracking Connected Components

Lecture 14, CS61B, Spring 2024

Introduction to Disjoint Sets

- Disjoint Sets API
- **Tracking Connected Components**

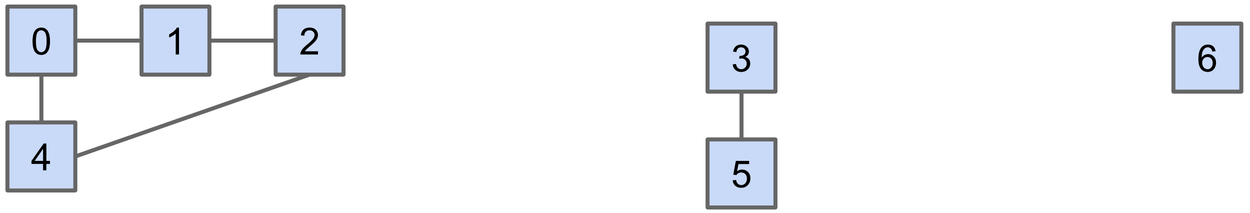
Implementations

- List of Sets
- Quick Find
- Quick Union
- Weighted Quick Union
- WQU with Path Compression

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

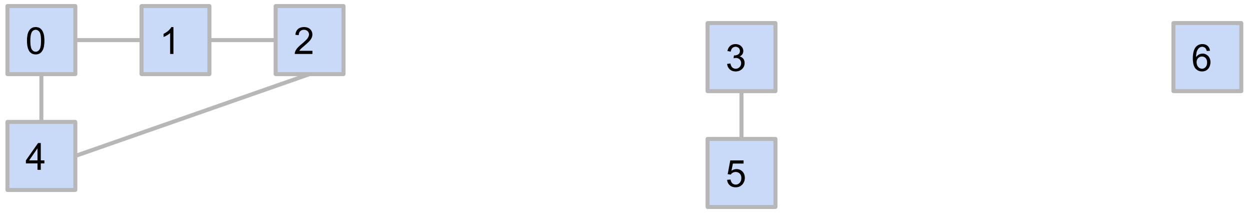
Rather than manually writing out every single connecting line, only record the sets that each item belongs to.

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

A Better Approach: Connected Components

For each item, its ***connected component*** is the set of all items that are connected to that item.

- 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.
  - Only need to keep track of which connected component each item belongs to.

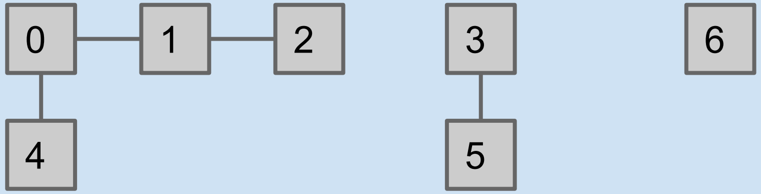


`{ 0, 1, 2, 4 }, {3, 5}, {6}`

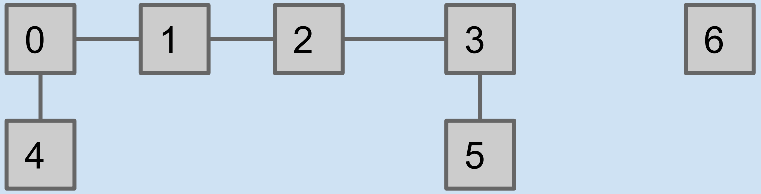
Up next: We'll consider how to do track set membership in Java.

Challenge: Pick Data Structures to Support Tracking of Sets

Before connect(2, 3) operation:



After connect(2, 3) operation:



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

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

Assume elements are numbered from 0 to N-1.

List of Sets

Lecture 14, CS61B, Spring 2024

Introduction to Disjoint Sets

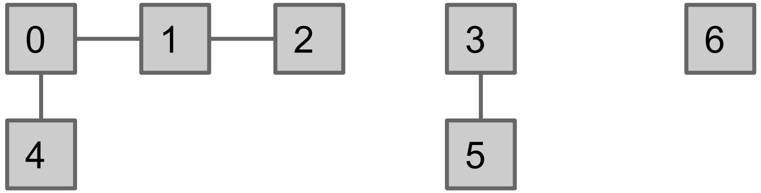
- Disjoint Sets API
- Tracking Connected Components

Implementations

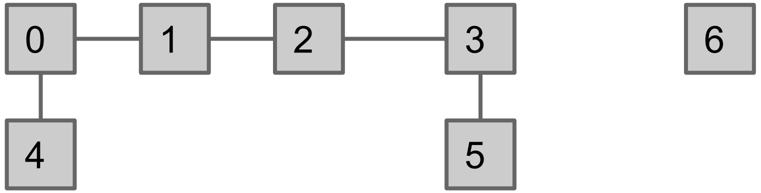
- **List of Sets**
- Quick Find
- Quick Union
- Weighted Quick Union
- WQU with Path Compression

Challenge: Pick Data Structures to Support Tracking of Sets

Before connect(2, 3) operation:



After connect(2, 3) operation:



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

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

Idea #1: List of sets of integers, e.g. [{0, 1, 2, 4}, {3, 5}, {6}]

- In Java: `List<Set<Integer>>`.
- Very intuitive idea, but actually **terrible**!

CC BY NC SA

Challenge: Pick Data Structures to Support Tracking of Sets

If nothing is connected:

Idea #1: List of sets of integers, e.g. [{0}, {1}, {2}, {3}, {4}, {5}, {6}]

- In Java: `List<Set<Integer>>`.
- Very intuitive idea, but actually **terrible**!
- Requires iterating through all the sets to find anything. Complicated and slow!
    - Worst case: If nothing is connected, then isConnected(5, 6) requires iterating through N-1 sets to find 5, then N sets to find 6. Overall runtime of Θ(N).

QuickFind

Lecture 14, CS61B, Spring 2024

Introduction to Disjoint Sets

- Disjoint Sets API
- Tracking Connected
- Components

Implementations

- List of Sets
- **Quick Find**
- Quick Union
- Weighted Quick Union
- WQU with Path Compression

Performance Summary

Implementation	constructor	connect	isConnected
ListOfSetsDS	Θ(N)	O(N)	O(N)

Constructor's runtime has order of growth N no matter what, so Θ(N).

Worst case is Θ(N), but other cases may be better. We'll say O(N) since O means "less than or equal".

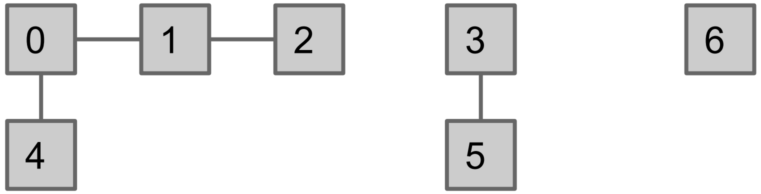
ListOfSetsDS is ***complicated*** and slow.

- Operations are linear when number of connections are small.
    - Have to iterate over all sets.
- Important point: By deciding to use a List of Sets, we have doomed ourselves to complexity and bad performance.

CC BY NC SA

Next Approach: Array of Integers

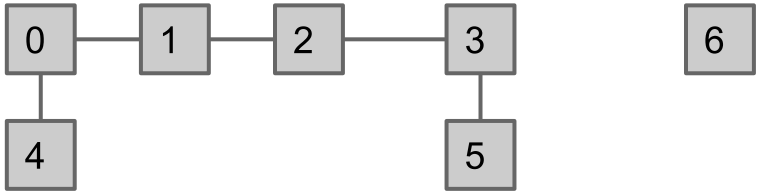
Before connect(2, 3) operation:



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

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

After connect(2, 3) operation:



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

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

Idea #2: list of integers where ith entry gives set number (a.k.a. “id”) of item i.

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

QuickFindDS

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

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

    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;
            }
        }...
```

Very fast: Two array accesses: Θ(1)

Relatively slow: N+2 to 2N+2 array accesses: Θ(N)

```java
public QuickFindDS(int N) {
    id = new int[N];
    for (int i = 0; i < N; i++)
        id[i] = -1;
    }
}
```

Performance Summary

Implementation	constructor	connect	isConnected
ListOfSetsDS	Θ(N)	O(N)	O(N)
QuickFindDS	Θ(N)	Θ(N)	Θ(1)

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

- Instead, let’s try something more radical.

Quick Union

Lecture 14, CS61B, Spring 2024

Introduction to Disjoint Sets

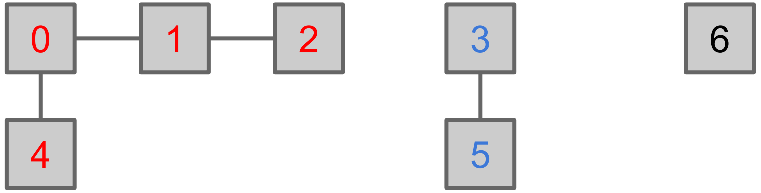
- Disjoint Sets API
- Tracking Connected Components

Implementations

- List of Sets
- Quick Find
- **Quick Union**
- Weighted Quick Union
- WQU with Path Compression

Improving the Connect Operation

Approach zero: Represent everything as boxes and lines. Overly complicated.



$\longrightarrow$??? in Java instance variables.

ListOfSets: Represent everything as connected components. Represented connected components as list of sets of integers.

`{ 0, 1, 2, 4 }, {3, 5}, {6}` $\longrightarrow$ [{0, 1, 2, 4}, {3, 5}, {6}]

`List<Set<Integer>>`

QuickFind: Represent everything as connected components. Represented connected components as a list of integers, where value = id.

`{ 0, 1, 2, 4 }, {3, 5}, {6}` $\longrightarrow$ [2, 2, 2, 3, 2, 3, 6]

`int[]`

Improving the Connect Operation

QuickFind: Represent everything as connected components. Represented connected components as a list of integers where value = id.

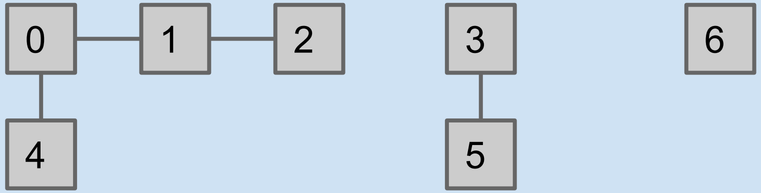
- Bad feature: Connecting two sets is slow!

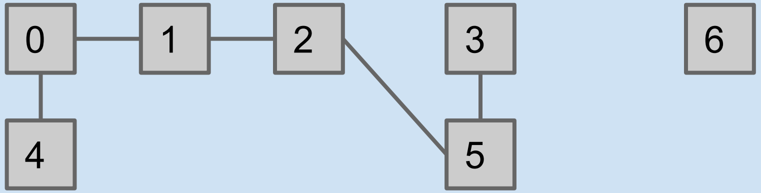
```
{ 0, 1, 2, 4 }, {3, 5}, {6}  ───────▶  [2, 2, 2, 3, 2, 3, 6]
                                              int[]
```

Next approach (QuickUnion): We will still represent everything as connected components, and we will still represent connected components as a list of integers. However, values will be chosen so that connect is fast.

Improving the Connect Operation

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}`

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

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

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

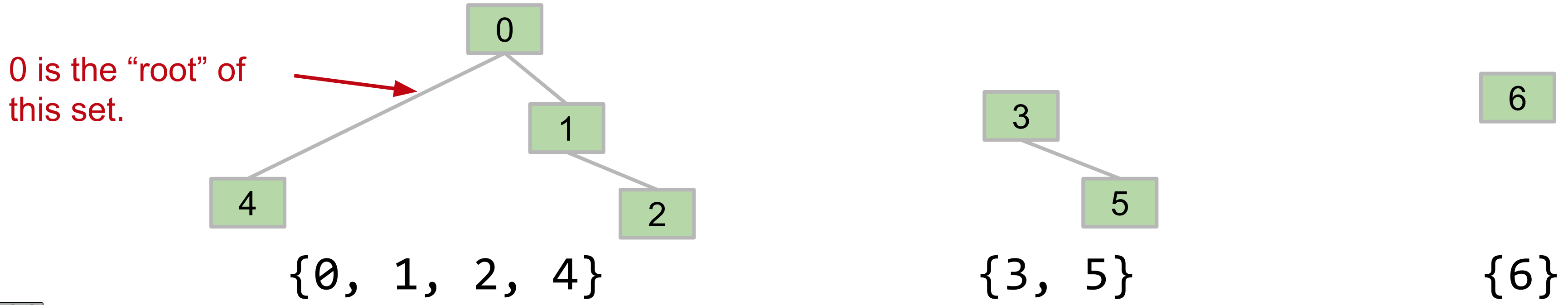
Improving the Connect Operation

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

- Idea: Assign each item a parent (instead of an id). Results in a tree-like shape.
  - An innocuous sounding, seemingly arbitrary solution.
  - Unlocks a pretty amazing universe of math that we won't discuss.

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

Note: The optional textbook has an item's parent as itself instead of -1 for root items.



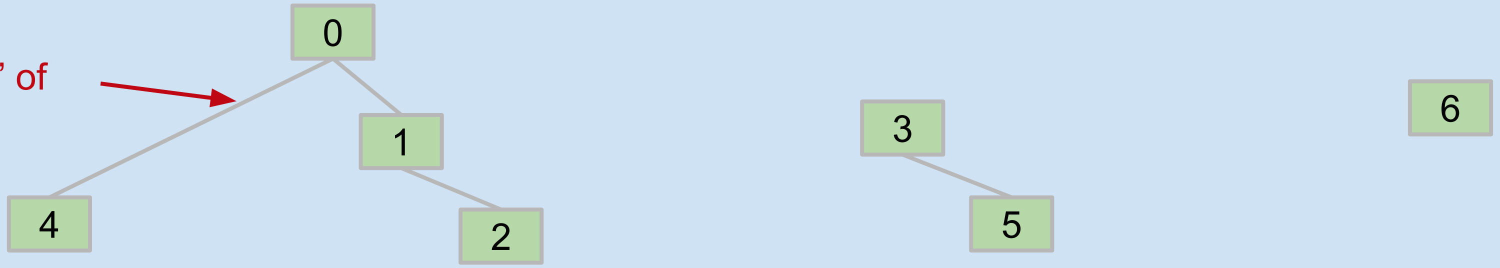
Improving the Connect Operation

connect(5, 2)

- How should we change the parent list to handle this connect operation?
  - If you’re not sure where to start, consider: why can’t we just set id[5] to 2?

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

0 is the “root” of this set.



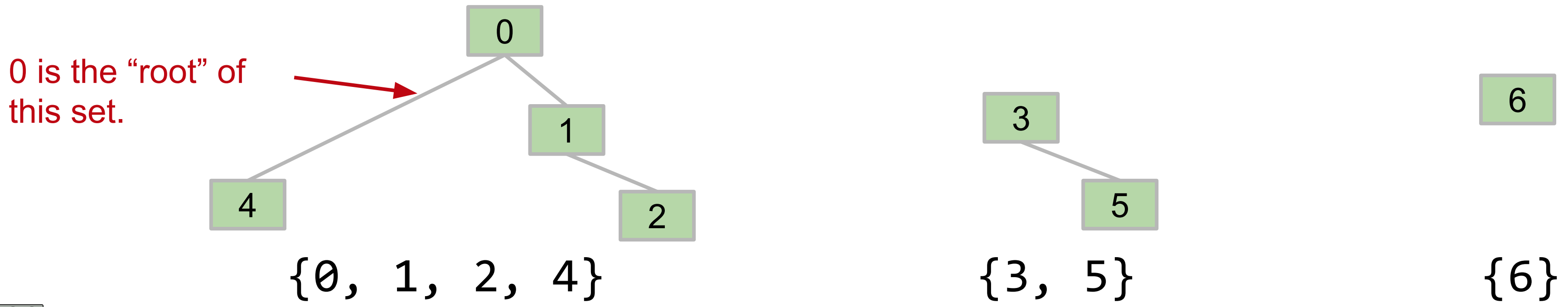
{0, 1, 2, 4}   {3, 5}   {6}

Improving the Connect Operation

connect(5, 2)

- Find root(5). // returns 3
- Find root(2). // returns 0
- Set root(5)'s value equal to root(2).

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

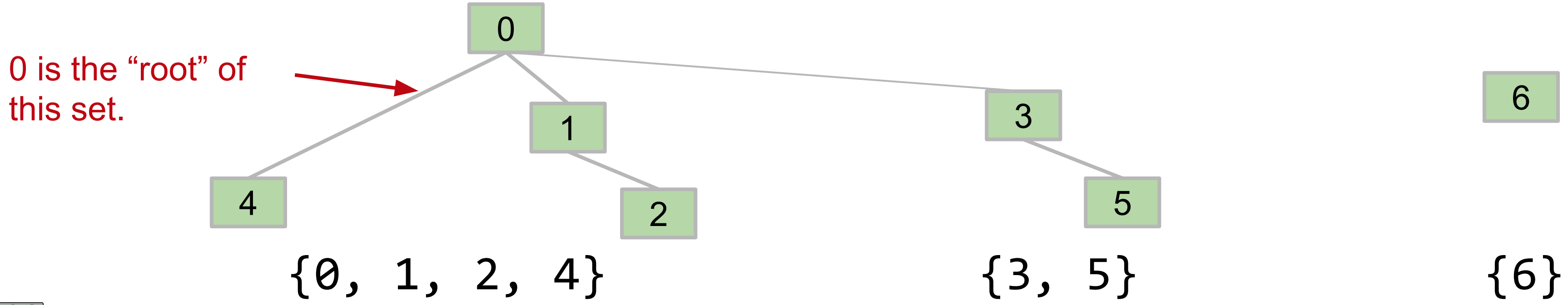


Improving the Connect Operation

connect(5, 2)

- Find root(5). // returns 3
- Find root(2). // returns 0
- Set root(5)'s value equal to root(2).

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



cc BY NC SA

Set Union Using Rooted-Tree Representation

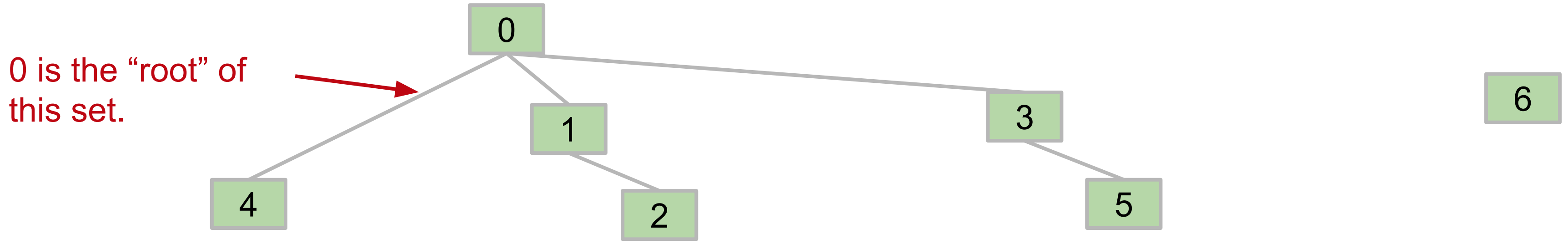
connect(5, 2)

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

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

What are the potential performance issues with this approach?

- Compared to QuickFind, we have to climb up a tree.

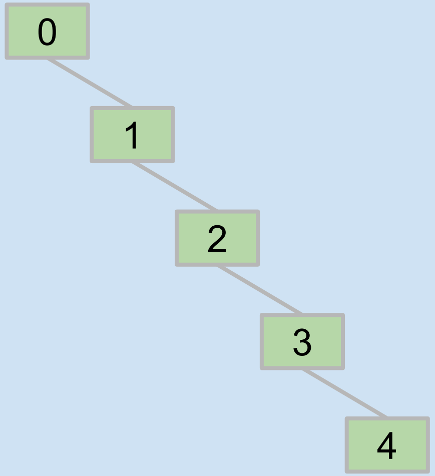


{0, 1, 2, 4, 3, 5} {6}

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)



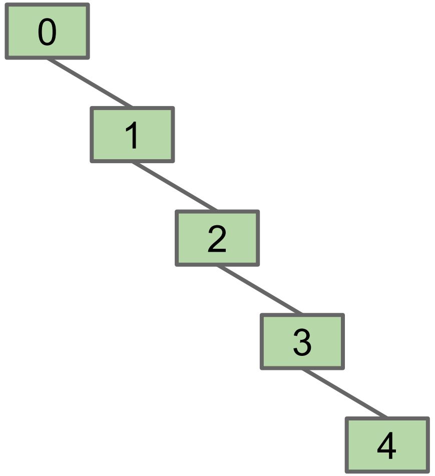
For N items, what's the worst case runtime…

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

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)? Θ(N)
- For isConnected(p, q)? Θ(N)

QuickUnionDS

```java
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] = -1; }
        }

    private int find(int p) {
        int r = p;
        while (parent[r] >= 0)
          { r = parent[r]; }
        return r;
    }
```

For N items, this means a worst case runtime of Θ(N).

```java
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;
}
```

Here the `find` operation is the same as the “root(x)” idea we had in earlier slides.

Performance Summary

Implementation	Constructor	connect	isConnected
ListOfSetsDS	Θ(N)	O(N)	O(N)
QuickFindDS	Θ(N)	Θ(N)	Θ(1)
QuickUnionDS	Θ(N)	O(N)	O(N)

Using O because runtime can be between constant and linear.

QuickFindDS defect: QuickFindDS is too slow: Connecting takes Θ(N) time.

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

- Observation: Things would be fine if we just kept our tree balanced.

CC BY NC SA

Weighted Quick Union

Lecture 14, CS61B, Spring 2024

Introduction to Disjoint Sets

- Disjoint Sets API
- Tracking Connected Components

Implementations

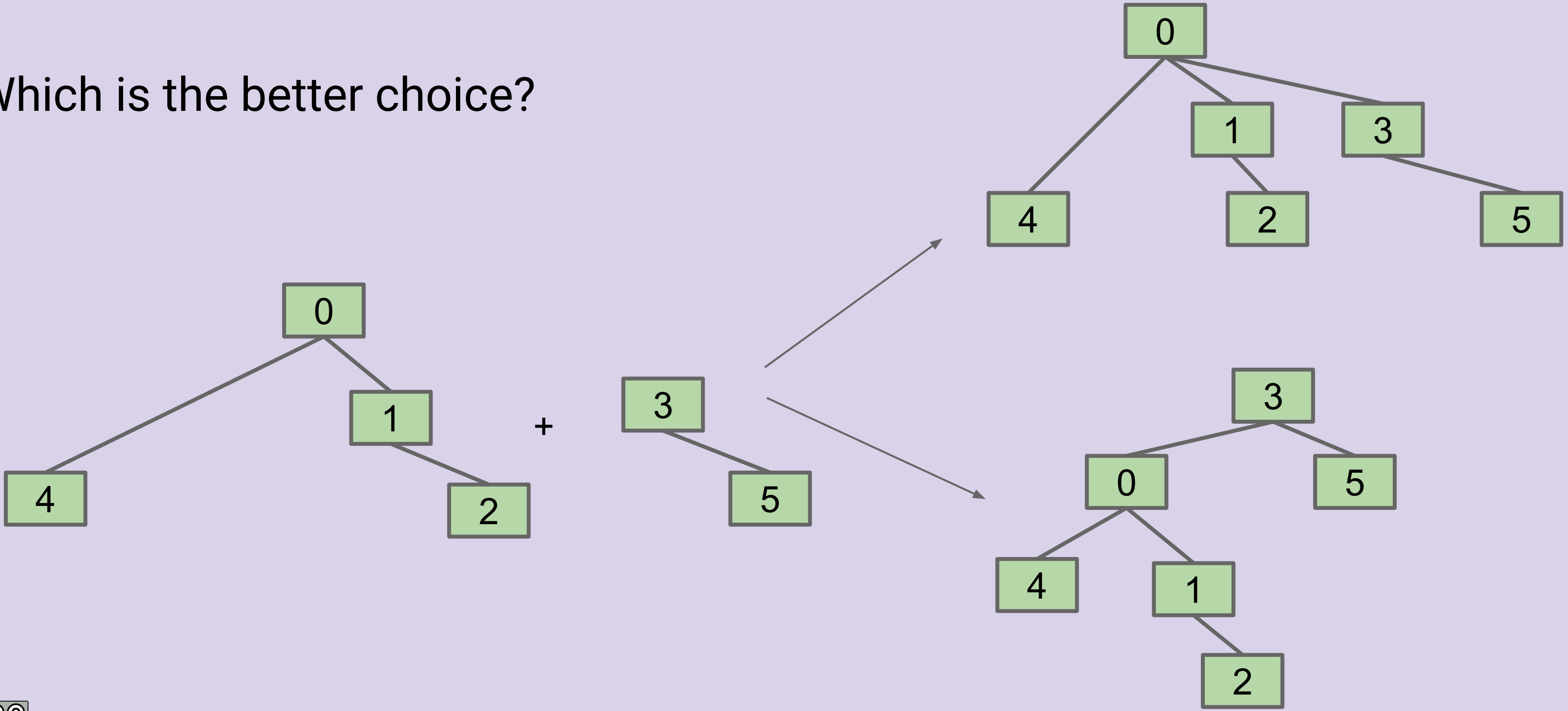
- List of Sets
- Quick Find
- Quick Union
- **Weighted Quick Union**
- WQU with Path Compression

A Choice of Two Roots: yellkey.com/detail

Suppose we are trying to connect(2, 5). We have two choices:

A. Make 5's root into a child of 2's root.
B. Make 2's root into a child of 5's root.

Which is the better choice?



CC BY NC SA

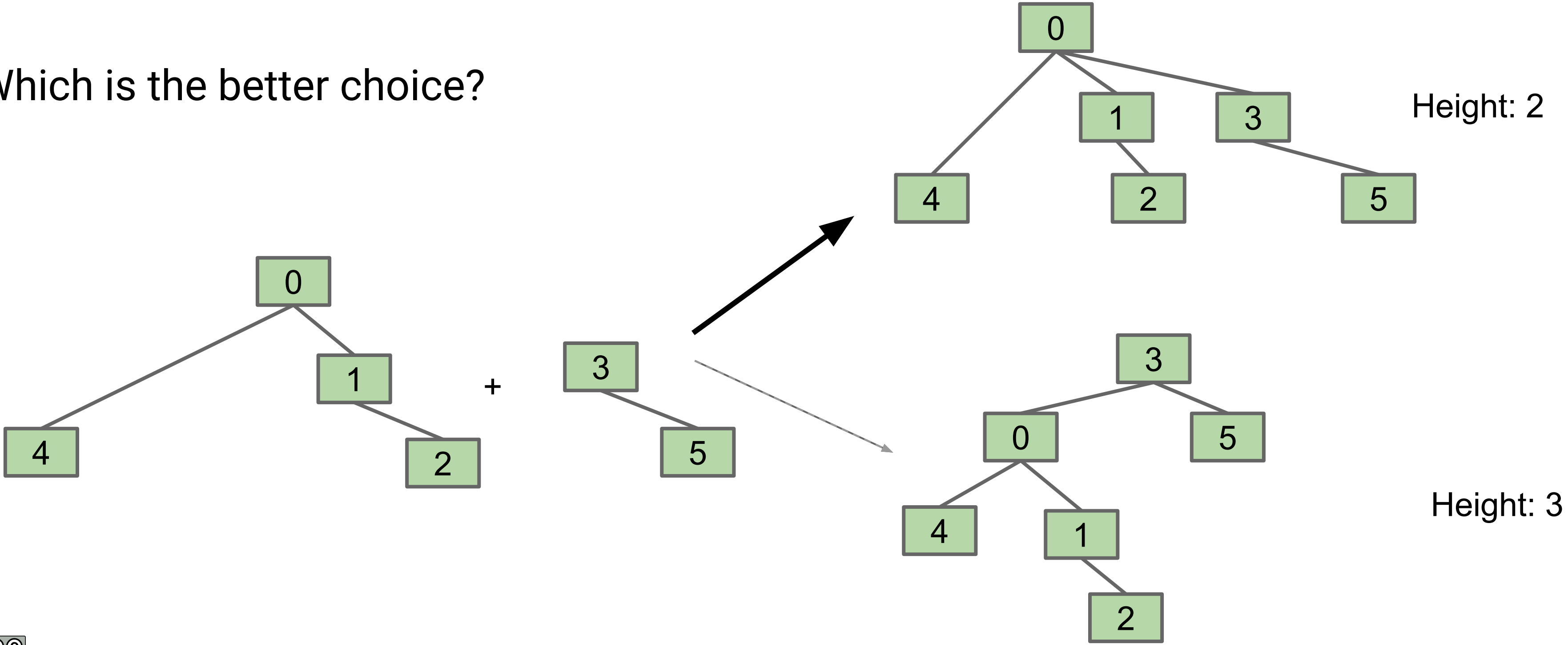
A Choice of Two Roots

Suppose we are trying to connect(2, 5). We have two choices:

A. Make 5's root into a child of 2's root.

B. Make 2's root into a child of 5's root.

Which is the better choice?



Possible Approach

One possible approach is to keep track of the height of every tree.

- Link up shorter tree below the larger tree.
- In case of a tie, break tie arbitrarily.

Unfortunately, tracking tree height is kind of annoying.

Interestingly, tracking the tree's "size", a.k.a. "weight" works just as well asymptotically.

- Size and weight both mean the total number of items in that tree.

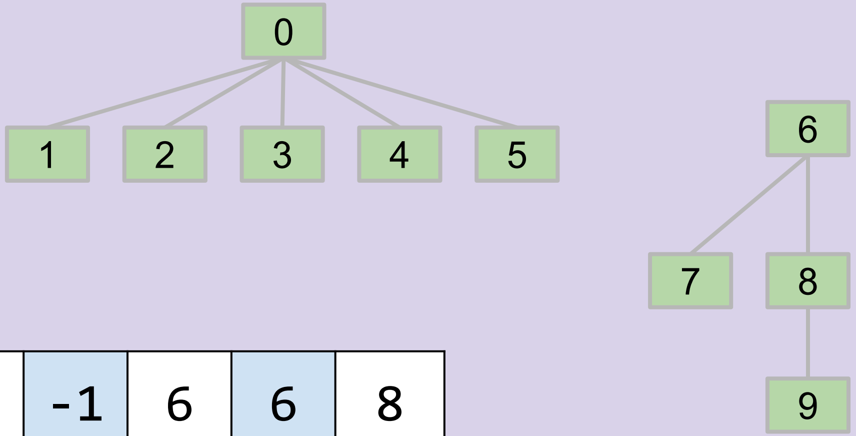
Weighted QuickUnion: http://yellkey.com/unit

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]



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

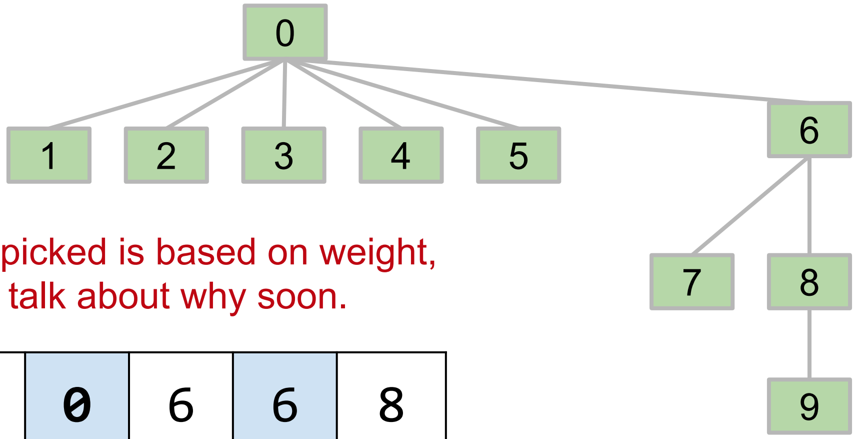
Improvement #1: Weighted QuickUnion

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]



Note: The rule I picked is based on weight, not height. We'll talk about why soon.

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

Implementing WeightedQuickUnion

Minimal changes needed:

- Use `parent[]` array as before.
- `isConnected(int p, int q)` requires no changes.
- `connect(int p, int q)` needs to somehow keep track of sizes.
  - See the old Disjoint Sets lab for the full details.
  - Two common approaches:
    - Replace -1 with -weight for roots (top approach).
    - Create a separate size array (bottom approach).

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

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

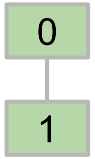
Weighted Quick Union Performance

Let’s consider the worst case where the tree height grows as fast as possible.

N	H
1	0

Weighted Quick Union Performance

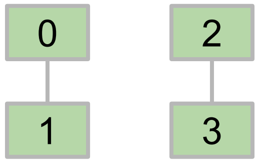
Let’s consider the worst case where the tree height grows as fast as possible.



N	H
1	0
2	1

Weighted Quick Union Performance

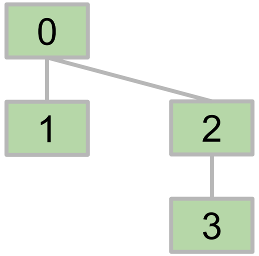
Let’s consider the worst case where the tree height grows as fast as possible.



N	H
1	0
2	1

Weighted Quick Union Performance

Let’s consider the worst case where the tree height grows as fast as possible.

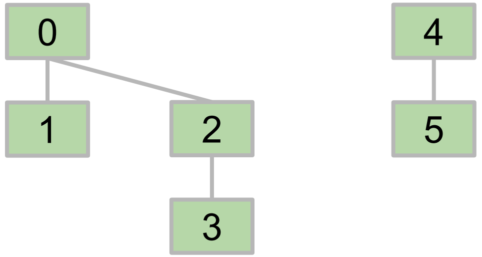


N	H
1	0
2	1
4	2

BY NC SA

Weighted Quick Union Performance

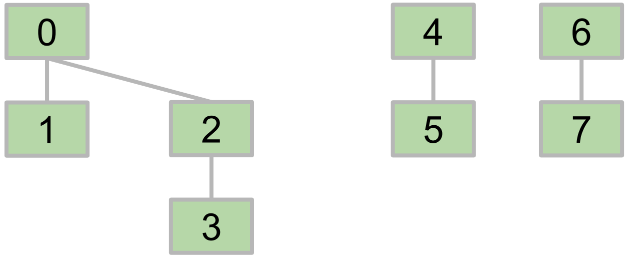
Let’s consider the worst case where the tree height grows as fast as possible.



N	H
1	0
2	1
4	2

Weighted Quick Union Performance

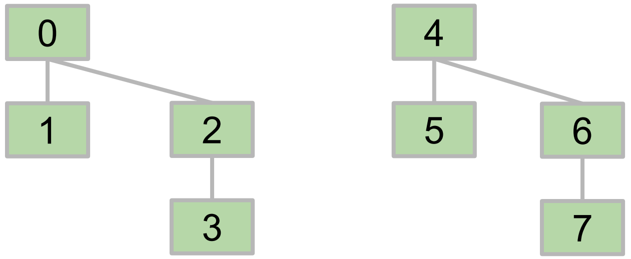
Let’s consider the worst case where the tree height grows as fast as possible.



N	H
1	0
2	1
4	2

Weighted Quick Union Performance

Let's consider the worst case where the tree height grows as fast as possible.

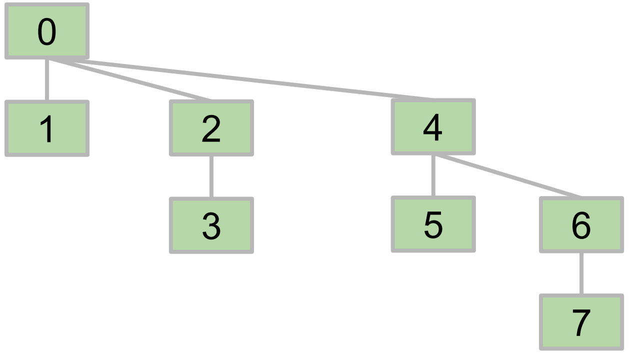


N	H
1	0
2	1
4	2

BY NC SA

Weighted Quick Union Performance

Let’s consider the worst case where the tree height grows as fast as possible.



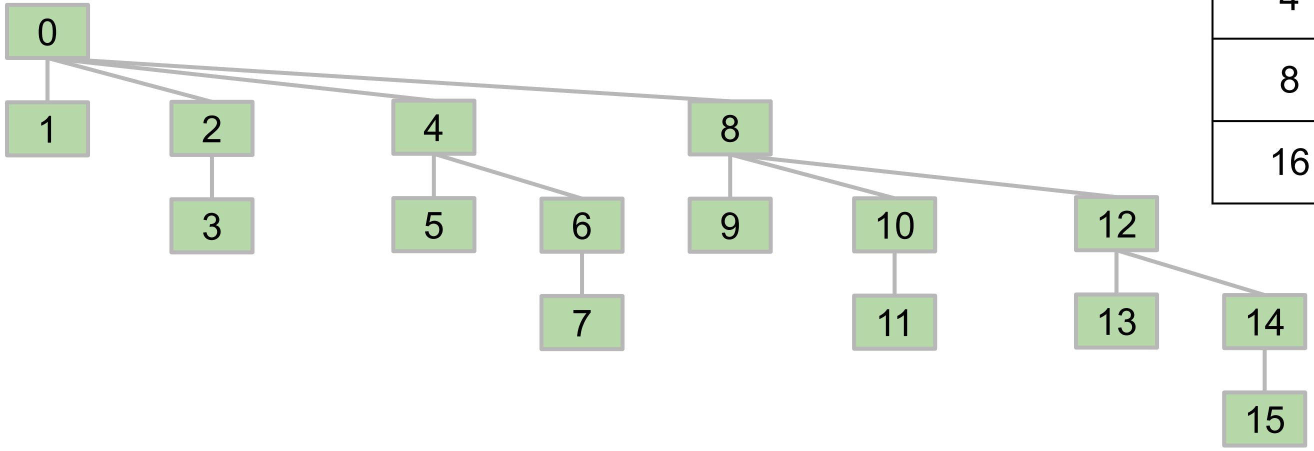
N	H
1	0
2	1
4	2
8	3

Weighted Quick Union Performance

Let’s consider the worst case where the tree height grows as fast as possible.

- Worst case tree height is Θ(log N).

N	H
1	0
2	1
4	2
8	3
16	4



Performance Summary

Implementation	Constructor	connect	isConnected
ListOfSetsDS	Θ(N)	O(N)	O(N)
QuickFindDS	Θ(N)	Θ(N)	Θ(1)
QuickUnionDS	Θ(N)	O(N)	O(N)
WeightedQuickUnionDS	Θ(N)	O(log N)	O(log N)

QuickUnion’s runtimes are O(H), and WeightedQuickUnionDS height is given by H = O(log N). Therefore connect and isConnected are both O(log N).

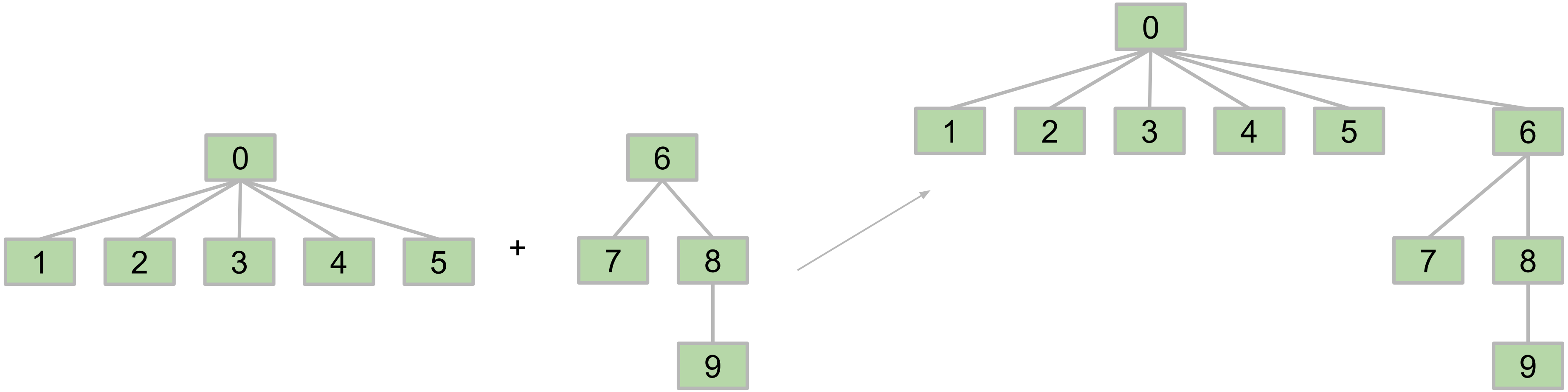
By tweaking QuickUnionDS we’ve achieved logarithmic time performance.

- Fast enough for all practical problems.

Why Weights Instead of Heights?

We used the number of items in a tree to decide upon the root.

- Why not use the height of the tree?
    - Worst case performance for HeightedQuickUnionDS is asymptotically the same! Both are Θ(log(N)).
    - Resulting code is more complicated with no performance gain.



cc BY NC SA

WQU with Path Compression

Lecture 14, CS61B, Spring 2024

Introduction to Disjoint Sets

- Disjoint Sets API
- Tracking Connected Components

Implementations

- List of Sets
- Quick Find
- Quick Union
- Weighted Quick Union
- **WQU with Path Compression**

What We’ve Achieved

Implementation	Constructor	connect	isConnected
ListOfSetsDS	Θ(N)	O(N)	O(N)
WeightedQuickUnionDS	Θ(N)	O(log N)	O(log N)

Performing M operations on a DisjointSet object with N elements:

O(N) cost for constructor.

- For our naive implementation, runtime is O(N + MN) or just O(MN).
- For our best implementation, runtime is O(N + M log N).
- For N = 10^9 and M = 10^9, difference is 30 years vs. 6 seconds.
    - Key point: Good data structure unlocks solutions to problems that could otherwise not be solved!
- Good enough for all practical uses, but could we theoretically do better?

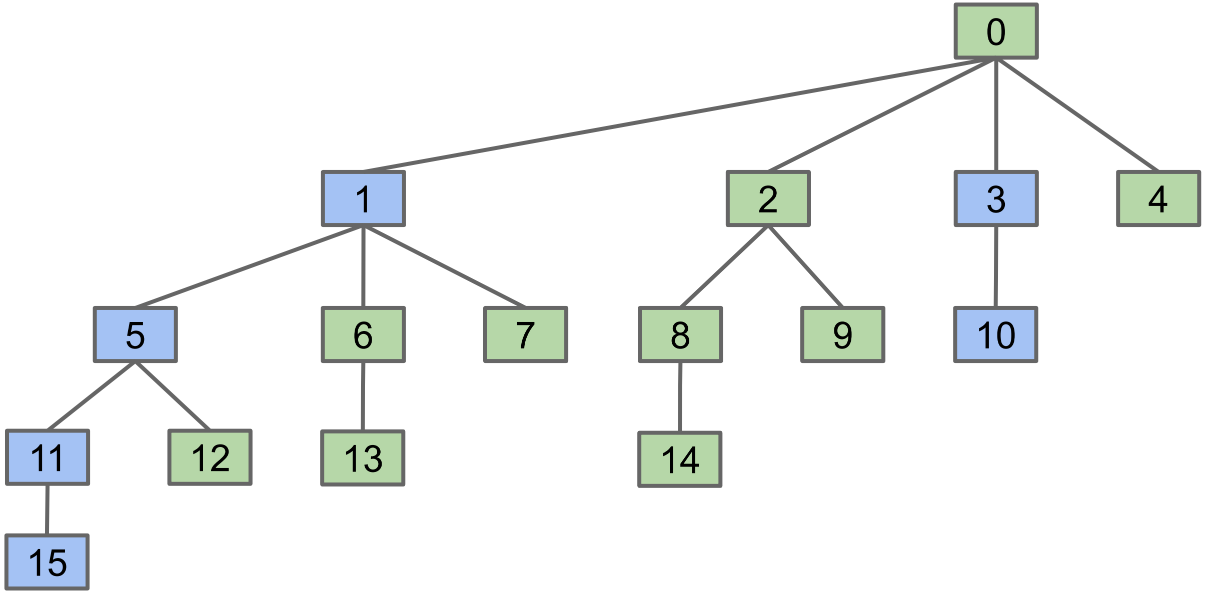
CC BY NC SA

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).



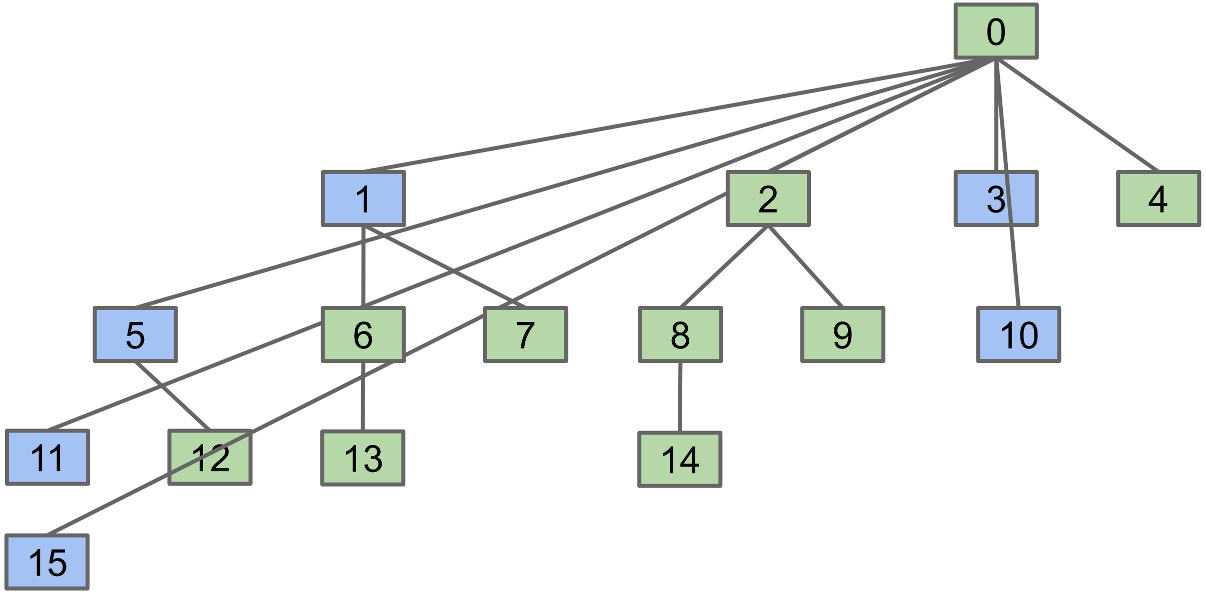
CC BY NC SA

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).



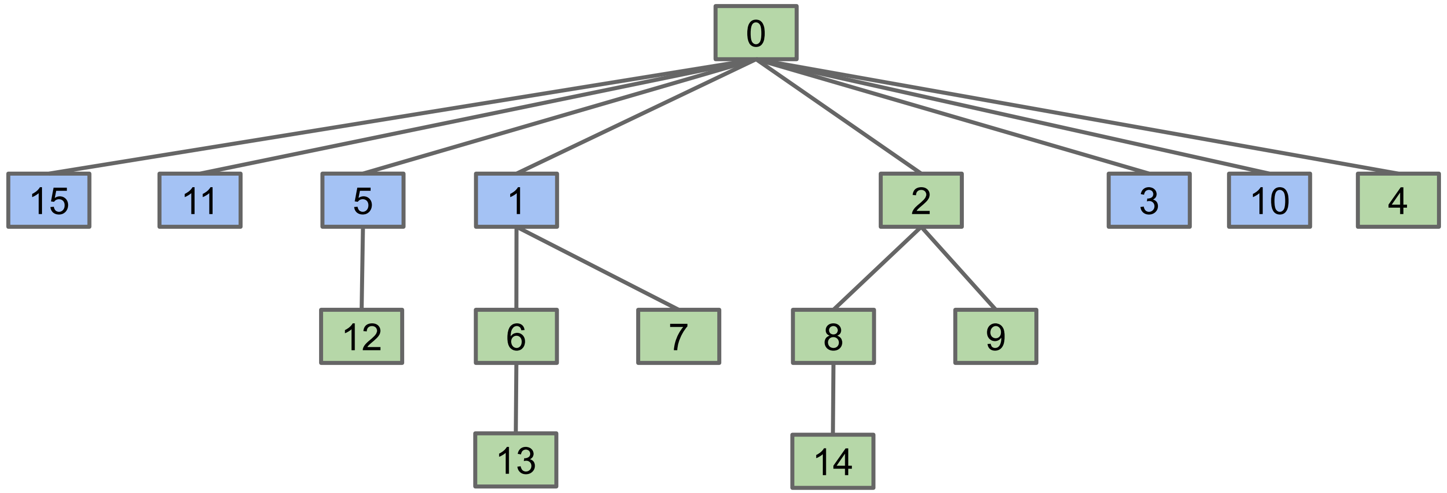
BY NC SA

170 Spoiler: Path Compression: A Clever Idea

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

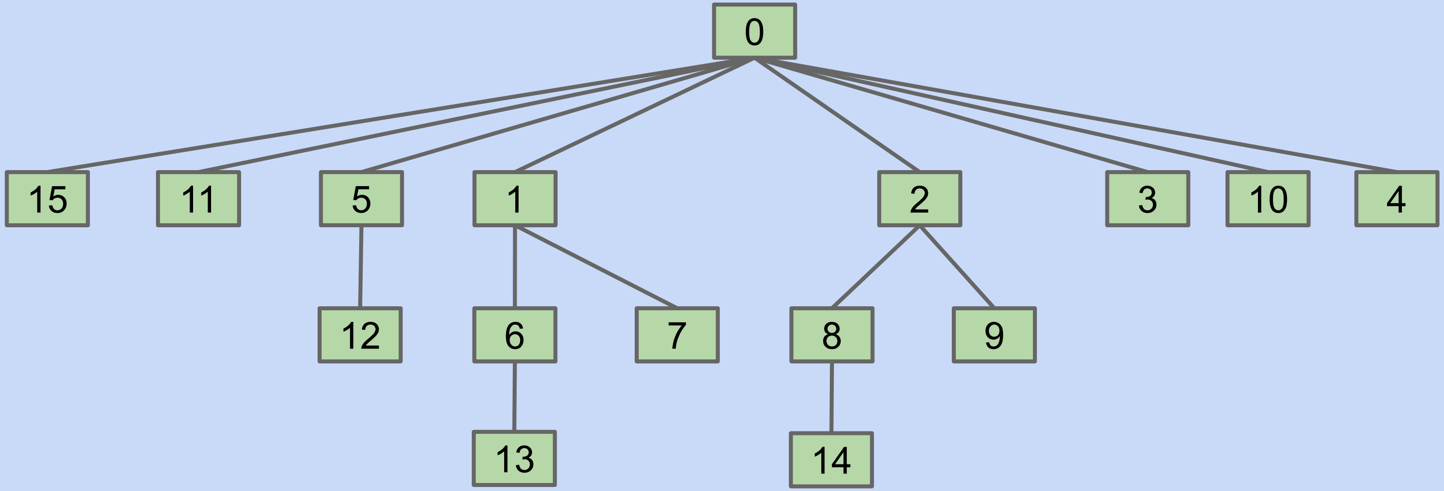
Clever idea: When we do isConnected(15, 10), tie all nodes seen to the root.

- Additional cost is insignificant (same order of growth).



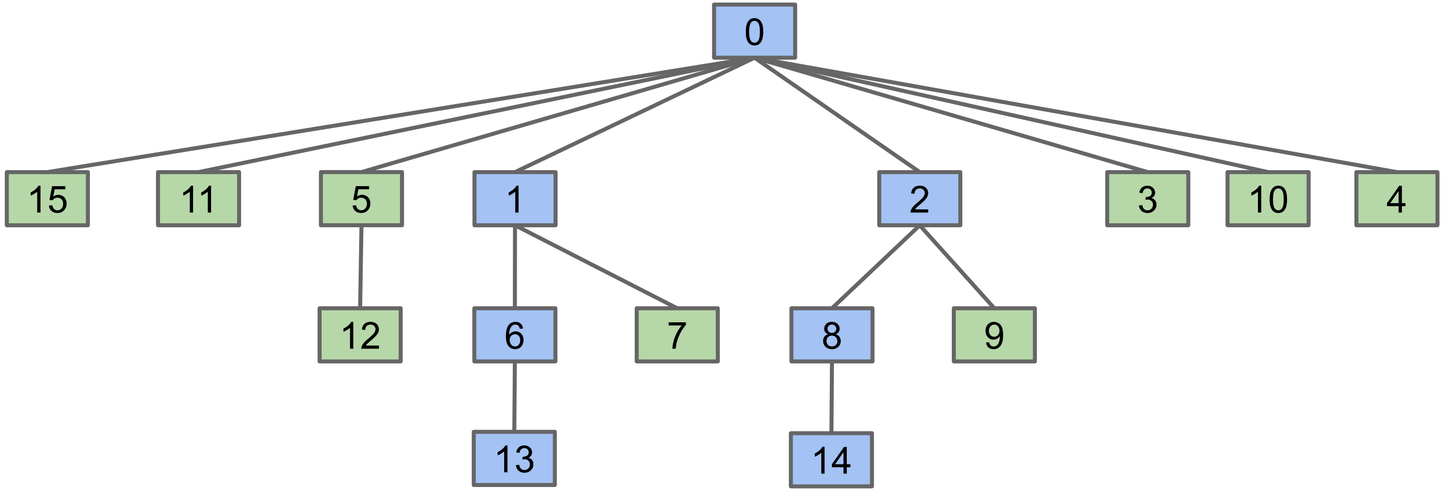
Path Compression: Another Clever Idea

Draw the tree after we call isConnected(14, 13).



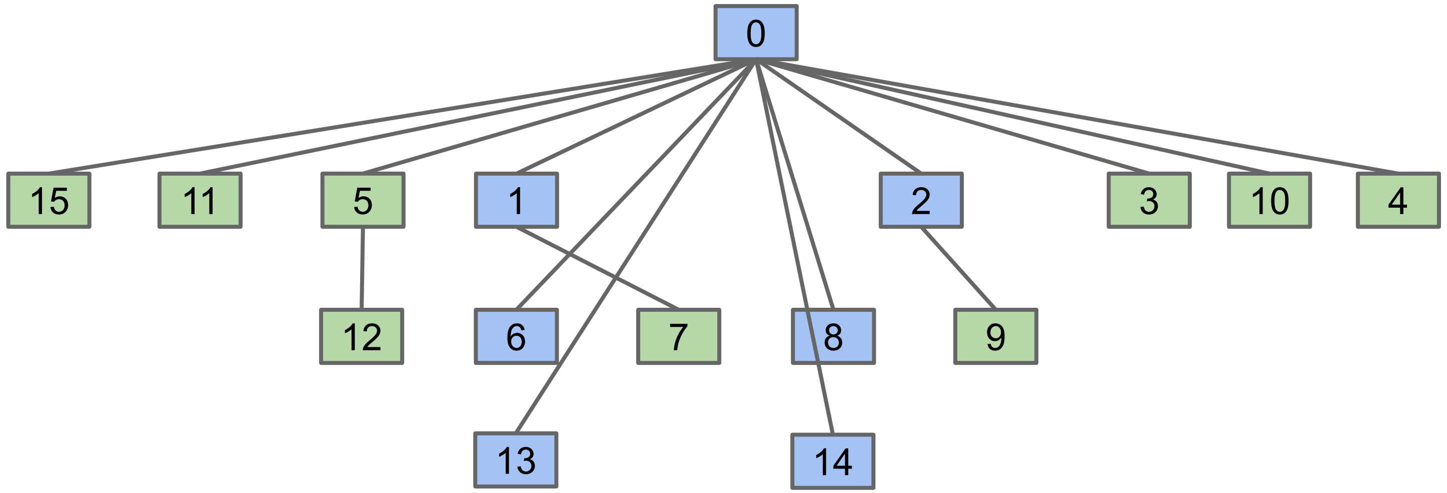
Path Compression: Another Clever Idea

Draw the tree after we call isConnected(14, 13).



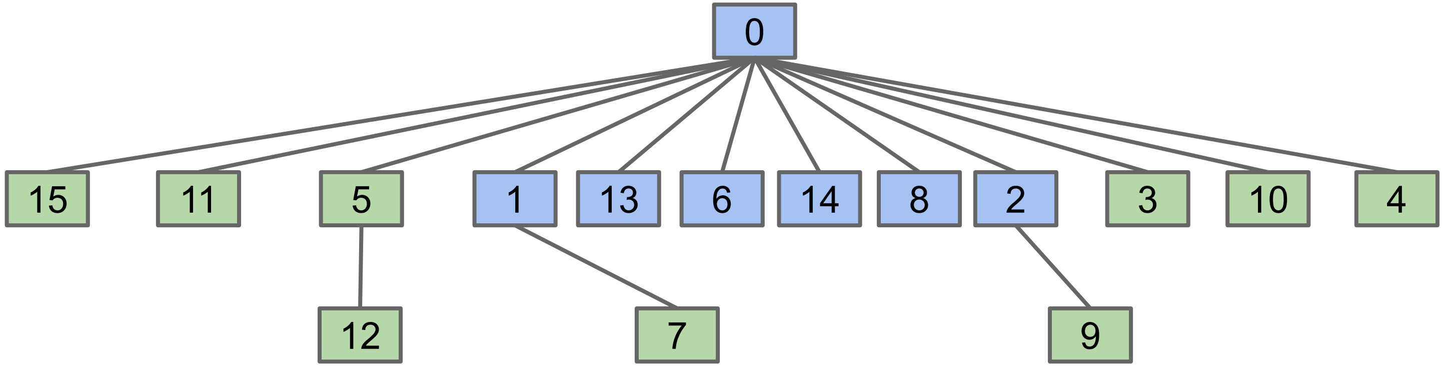
Path Compression: Another Clever Idea

Draw the tree after we call isConnected(14, 13).



Path Compression: Another Clever Idea

Draw the tree after we call isConnected(14, 13).



CC BY NC SA

Intuition

By compressing the tree with each union and isConnected call, we keep the tree nice and short.

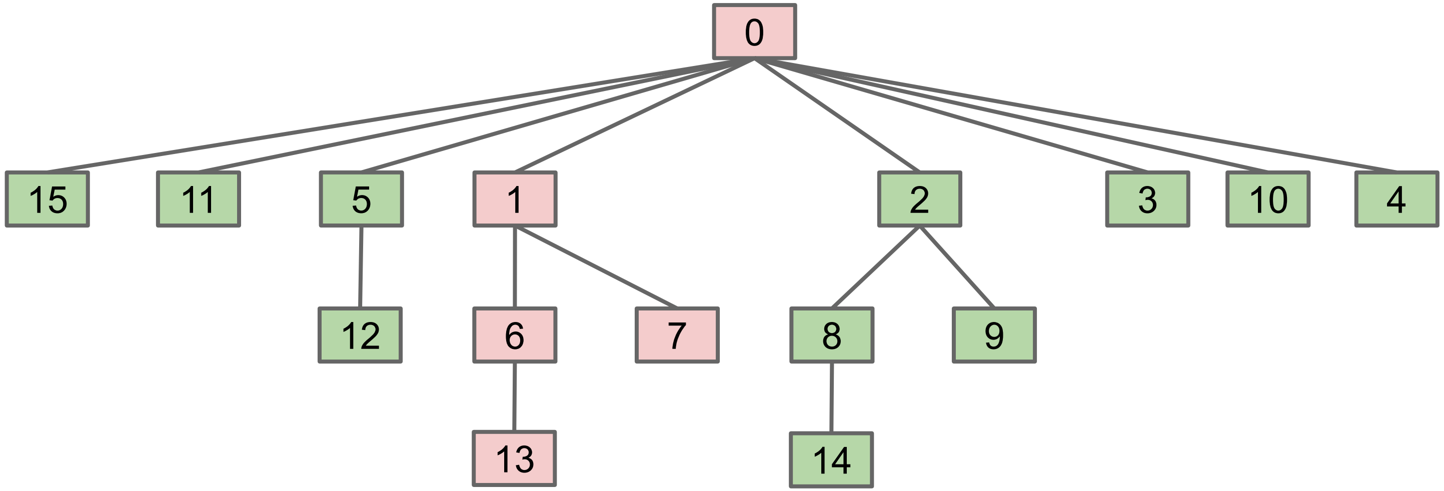
- As number of nodes N grows, our tree tends to get taller.
- As number of operations M grows, our tree tends to get shorter.
    - For enough operations tree height will shrink to 1.

Intuition

By compressing the tree with each union and isConnected call, we keep the tree nice and short.

Note: The tree we started with in the exercise you completed is impossible to generate if we're using path compression!

- The structure in red is impossible (try to convince yourself if you'd like).

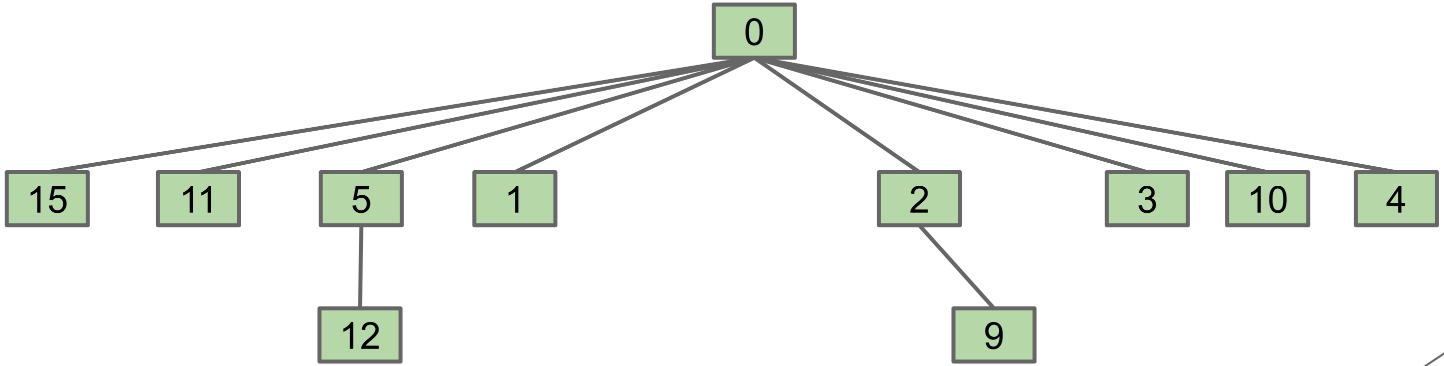


CC BY NC SA

CS 170 Analysis of Disjoint Sets

In CS170, you'll show that each `isConnected` or `connect` operation takes on average lg* N time because the tree is kept so compressed.

- lg*: How many times you need to press the log2 button on a calculator before you get to a number that is 1 or less. Example: http://joshh.ug/logstar/demo.html
- M operations on N nodes takes O(M lg* N) time for large M.
- lg* is less than or equal to 5 for any realistic input.



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

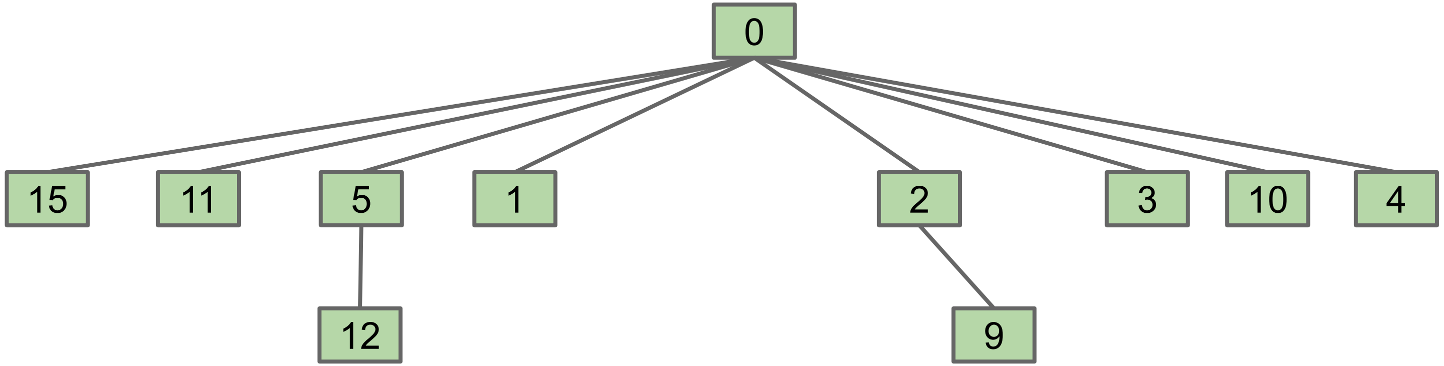
2^{16}

CC BY NC SA

Even More Careful Analysis

You can provide an even tighter bound, showing that each operation takes on average α(N) time.

- α is the inverse Ackermann function.
- See “Efficiency of a Good But Not Linear Set Union Algorithm.”
    - Written by Bob Tarjan while at UC Berkeley in 1975.



N	α(N)
1	0
...	1
...	2
...	3
...	4
	5

$$\underbrace{2^{2^{2^{\cdots 2}}}}_{65536}$$

CC BY NC SA

A Summary of Our Iterative Design Process

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 ideas that made our implementation efficient:

- Represent sets as connected components (don’t track individual connections).
    - **ListOfSetsDS**: Store connected components as a List of Sets (slow, complicated).
    - **QuickFindDS**: Store connected components as set ids.
    - **QuickUnionDS**: Store connected components as parent ids.
        - **WeightedQuickUnionDS**: Also track the size of each set, and use size to decide on new tree root.
            - **WeightedQuickUnionWithPathCompressionDS**: On calls to connect and isConnected, set parent id to the root for all items seen.

Performance Summary

Implementation	Runtime
ListOfSetsDS	O(NM)
QuickFindDS	Θ(NM)
QuickUnionDS	O(NM)
WeightedQuickUnionDS	O(M log N)
WeightedQuickUnionDSWithPathCompression	O(M α(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`.