Asymptotics, Disjoint Sets

Discussion 05

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
		2/20 Lab 4 Due Project 1C Due				
	2/26 Lab 5 Due Homework 2 Due					

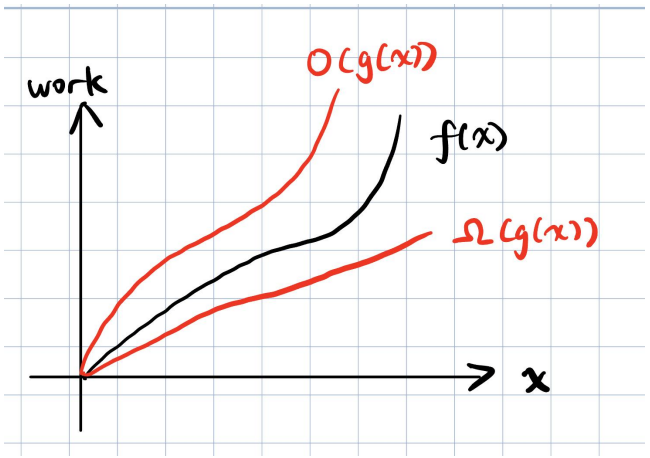
Content Review

Asymptotics

Asymptotics allow us to evaluate the performance of programs using math.

We ignore all constants and only care about the total work done when the input is very large.

Big O - If a function f(x) has big O in g(x), it grows at most as fast as g(x).
Big Ω - f(x) grows at least as fast as g(x),
Big Θ - When a function is both O(g(x)) and Ω(g(x)), it is Θ(g(x))



Common Orders of Growth

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(c^n)$
- Alternatively: constant < logarithmic < linear < nlogn < quadratic < exponential
- Desmos example here
    - Constants don't matter in the long run!
- Fun sums:

$$1 + 2 + 3 + \ldots + N = \Theta(N^2)$$

$$1 + 2 + 4 + 8 + \ldots + N = \Theta(N)$$

Tightest Bound?

- Sometimes it's easier to bound the runtime than to calculate the runtime.
- When you bound, always provide the tightest bound
  - i.e, the bound that provides the most information about the runtime.
- Ex. Given $f(n) = 2n + 5$, we could say that $f(n) \in O(n^n)$, but that doesn't tell us very much
  - a lot of functions are upper bounded by $O(n^n)$ (grows really fast!)
  - A better, tighter bound would be $f(n) \in \Theta(n)$

Best vs. Worst Case

- In best-worst case analysis, we still assume the input is very large.
- Therefore, you cannot make assumptions such as N == 1 or N <= 10 in these analyses.
- **The best case is not when the input is 1.**
- **The best case is not when the input is 1.**
- **Seriously, the best case is not when the input is 1.**

Best vs. Worst Case

- Represented with tight bound Θ because they should be consistent (always run in the same time)
- Look out for branching statements, loop conditions, breaks
- Check: What is the best/worst case runtime of the function below?
- **Remember: The best case is not when the input is 1.**

```
public static void example(int N) {
     while (N > 0) {
          if (func(N)) {
               break;
          }
          N -= 1;
     }
}
```

Best case: Θ(1), where N = some int for which func(N) is immediately true

I'm not assuming N is 1 or N is small. func(N) could be return whether N is an even number, and when N is very large but even number this function runs in constant time

Worst case: Θ(N), where N = some int for which func(N), func(N - 1), …, func(1) are all false

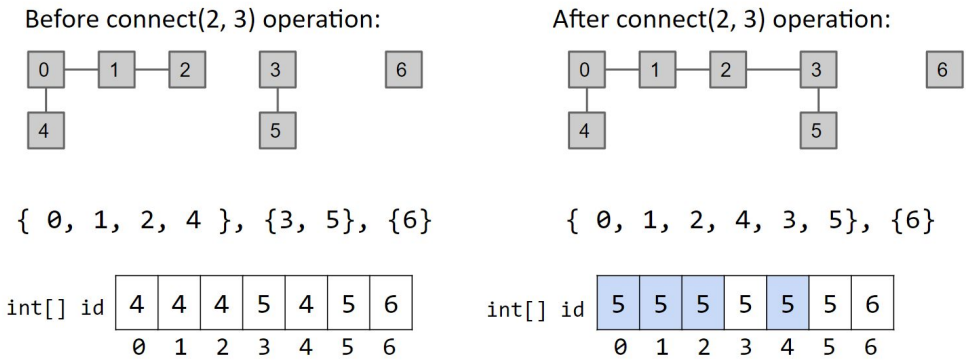
Best vs. Worst Case

- Best/worst case vs. lower/upper bound analogy: how much does it cost to eat at a restaurant?
  - Best/worst-case: “the cheapest thing on the menu is $5 and the most expensive is $50”
  - Lower/upper bound: “every item is at least $5 and at most $50” (credit: Alex Schedel)
- Which one is more informative?
  - The first one: the best/worst-case are the tightest lower/upper-bounds you can give.

Disjoint Sets, also known as Union Find

```
public interface DisjointSet {
     void connect (x, y); // Connects nodes x and y (you may also see union)
     boolean isConnected(x, y); // Returns true if x and y are connected
}
```

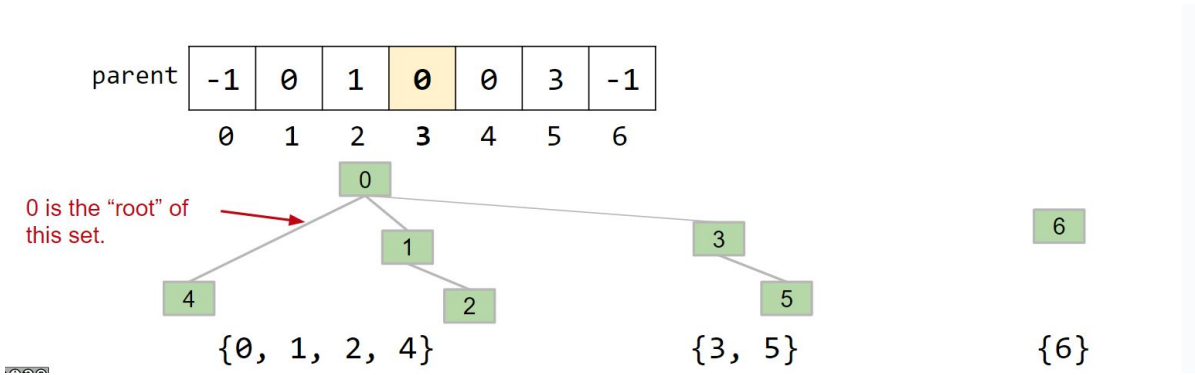
QuickFind uses an array of integers to track which set each element belongs to.



Disjoint Sets, also known as Union Find

```
public interface DisjointSet {
     void connect (x, y); // Connects nodes x and y (you may also see union)
     boolean isConnected(x, y); // Returns true if x and y are connected
}
```

QuickUnion stores the parent of each node rather than the set to which it belongs and merges sets by setting the parent of one root to the other.



Disjoint Sets, also known as Union Find

```
public interface DisjointSet {
     void connect (x, y); // Connects nodes x and y (you may also see union)
     boolean isConnected(x, y); // Returns true if x and y are connected
}
```

WeightedQuickUnion does the same as QuickUnion except it decides which set is merged into which by size (merge smaller into larger), reducing stringiness.

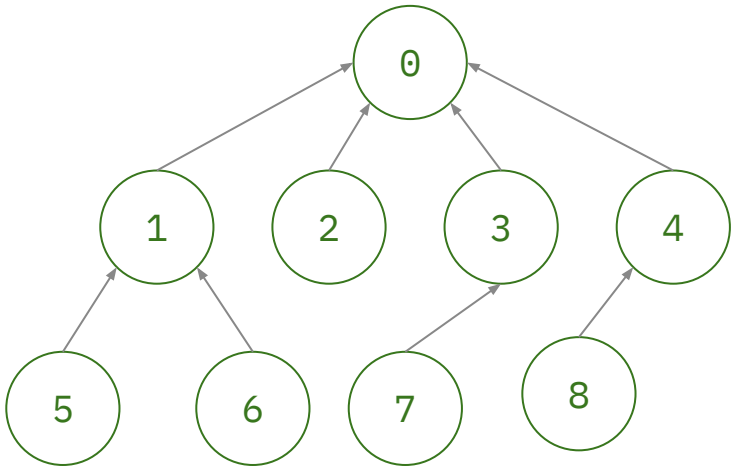
WeightedQuickUnion with Path Compression sets the parent of each node to the set's root whenever find() is called on it.

Disjoint Sets Representation

- We can use a single array to represent our disjoint set when implementing `connect()` optimally (ie. `WeightedQuickUnion`)

- `arr[i]` contains the parent of element i in the set; the index of a root contains - (`# elements in set rooted at that index`)

```
[-9, 0, 0, 0, 0, 1, 1, 3, 4]
```



Disjoint Sets Asymptotics

```
public interface DisjointSet {
     void connect (x, y); // Connects nodes x and y (you may also see union)
     boolean isConnected(x, y); // Returns true if x and y are connected
}
```

Implementation	Constructor	connect()	isConnected()
QuickUnion	Θ(N)	O(N)	O(N)
QuickFind	Θ(N)	O(N)	O(1)
Weighted Quick Union	Θ(N)	O(log N)	O(log N)
WQU with Path Compression	Θ(N)	O(log N) Θ(1)-ish amortized	O(log N) Θ(1)-ish amortized

* we don't really talk about QU/QF in application, more to show the asymptotic motivation for WQU

Worksheet

1A Asymptotics

Say we have a function `findMax` that iterates through an unsorted int array one time and returns the maximum element in that array.

Give the tightest lower and upper bounds ($\Omega(\cdot)$ and $O(\cdot)$) of findMax in terms of N, the length of the array.

Is it possible to define a $\Theta(\cdot)$ bound for findMax?

1A Asymptotics

Say we have a function `findMax` that iterates through an unsorted int array one time and returns the maximum element in that array.

Give the tightest lower and upper bounds (Ω(·) and O(·)) of findMax in terms of N, the length of the array.

Is it possible to define a Θ(·) bound for findMax?

Lower bound: Ω(N), Upper bound: O(N)

Theta bound: Θ(N)

1B Asymptotics What are the best and worst case runtimes?

```
for (int i = N; i > 0; i--) {
     for (int j = 0; j <= M; j++) {
          if (ping(i, j) > 64) {
               break;
          }
     }
}
```


1B Asymptotics What are the best and worst case runtimes?

```
for (int i = N; i > 0; i--) {
     for (int j = 0; j <= M; j++) {
          if (ping(i, j) > 64) {
               break;
          }
     }
}
```

Best: $\Theta(N)$

Worst: $\Theta(MN)$

i	N	N - 1	...	2	1
Best case work per i	1	1	...	1	1
Worst case work per i	M + 1	M + 1	M + 1	M + 1	M + 1

1C Asymptotics

What is the best case and worst case runtime?

```
public static boolean noUniques(int[] array) {
      array = sort(array);
      int N = array.length;
      for (int i = 0; i < N; i+= 1) {
            boolean hasDuplicate = false;
            for (int j = 0; j < N; j += 1) {
                  if (i != j && array[i] == array[j]) {
                        hasDuplicate = true;
                  }
            }
            if (!hasDuplicate) {
                  return false;
            }
      }
      return true;
}
```


1C Asymptotics

What is the best case and worst case runtime?

```
public static boolean noUniques(int[] array) {
      array = sort(array);
      int N = array.length;
      for (int i = 0; i < N; i+= 1) {
            boolean hasDuplicate = false;
            for (int j = 0; j < N; j += 1) {
                  if (i != j && array[i] == array[j]) {
                        hasDuplicate = true;
                  }
            }
            if (!hasDuplicate) {
                  return false;
            }
      }
      return true;
}
```

Don't forget the sort!

Best: $\Theta(N \log N)$

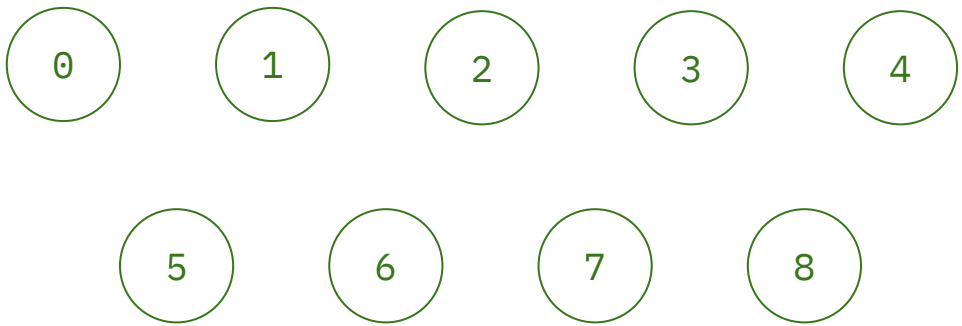
Worst: $\Theta(N^2)$

i	0	1	…	N - 1
Best case work per i	N	0 (already returned)	…	0 (already returned)
Worst case work per i	N	N	…	N

2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

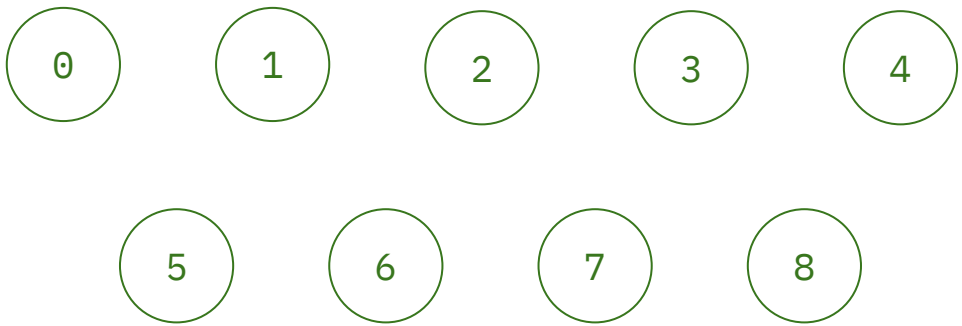
```
[-1, -1, -1, -1, -1, -1, -1, -1, -1]
```



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

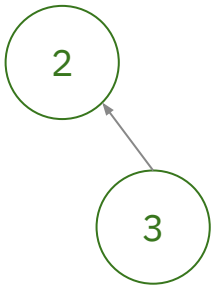
```
[-1, -1, -1, -1, -1, -1, -1, -1, -1]
```



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

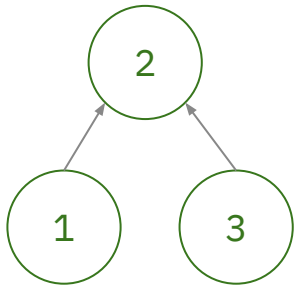
```
[-1, -1, -2, 2, -1, -1, -1, -1, -1]
```



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

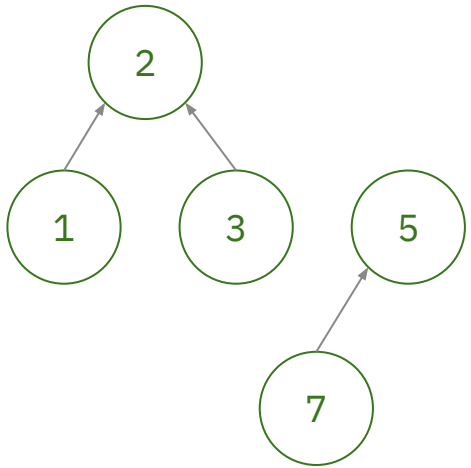
```
[-1, 2, -3, 2, -1, -1, -1, -1, -1]
```



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

```
[-1, 2, -3, 2, -1, -2, -1, 5, -1]
```



0

4

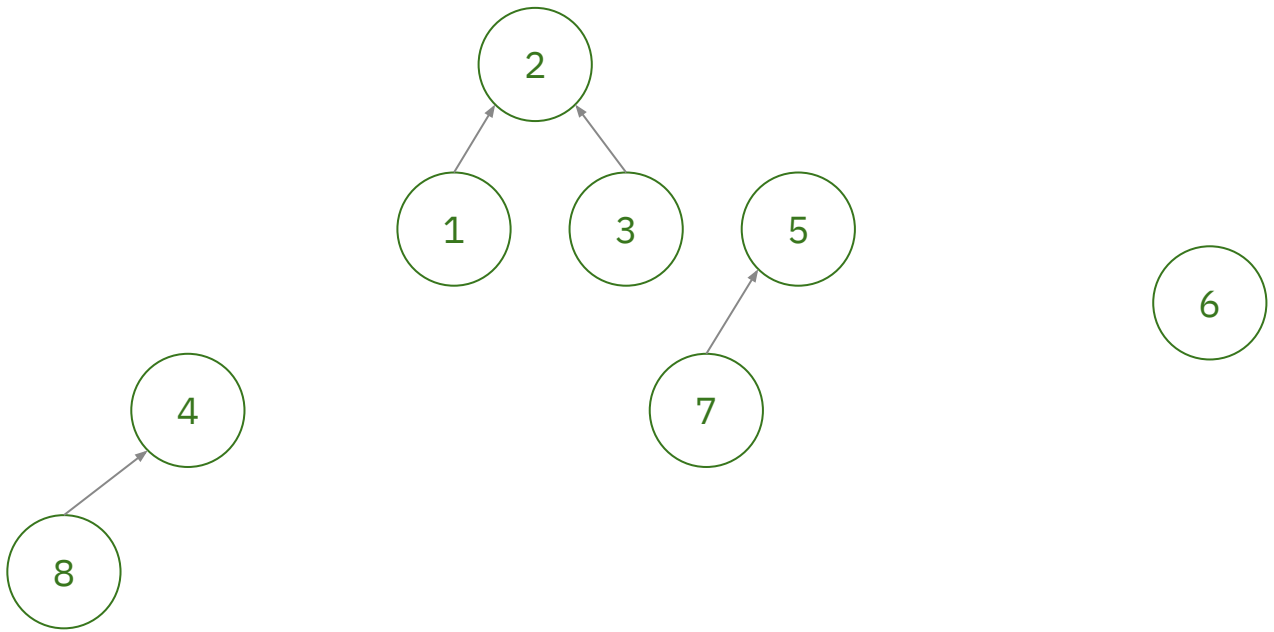
6

8

2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

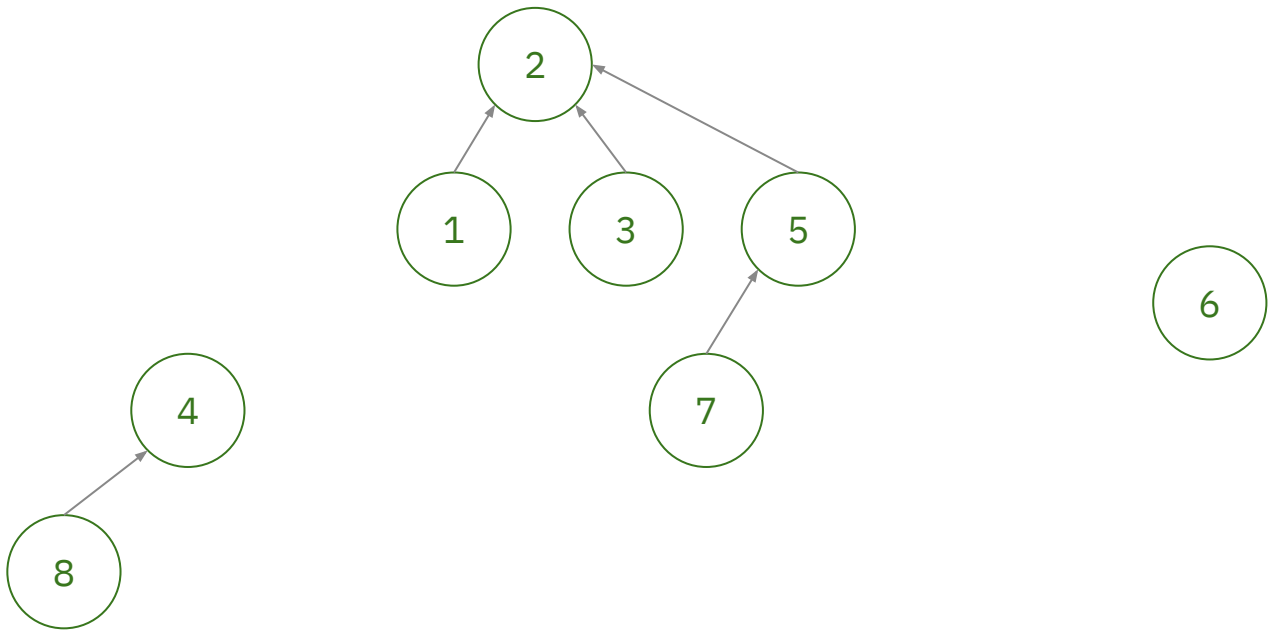
[-1, 2, -3, 2, -2, -2, -1, 5, 4]



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

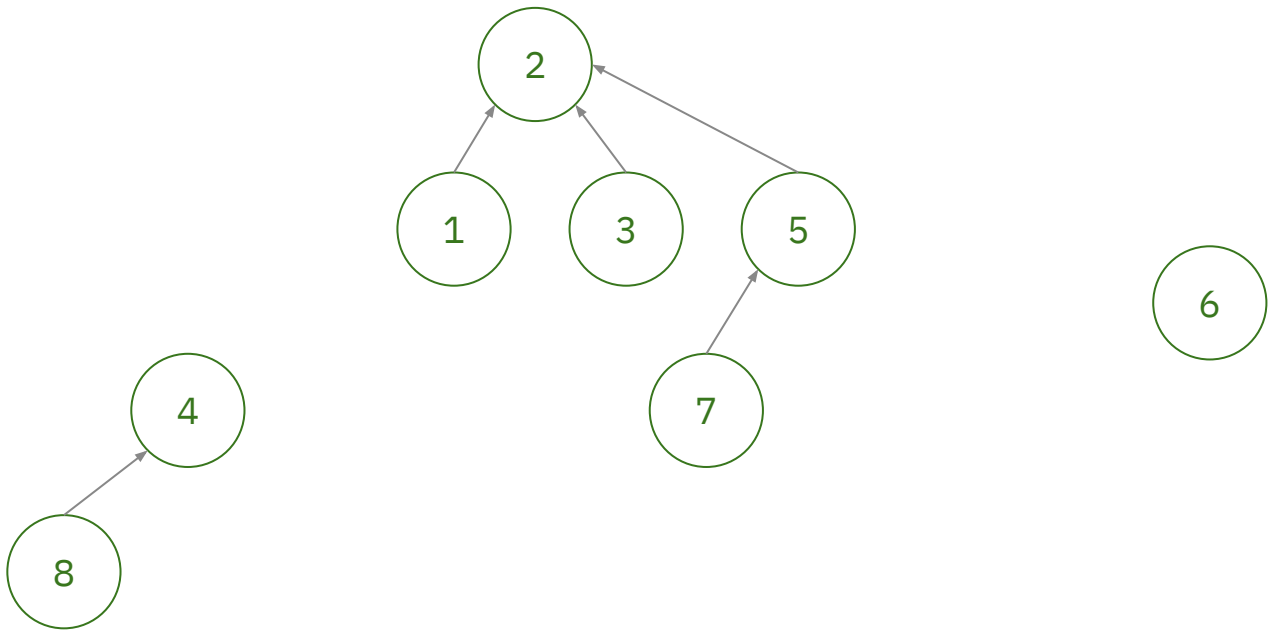
`[-1, 2, -5, 2, -2, 2, -1, 5, 4]`



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3); // 2
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

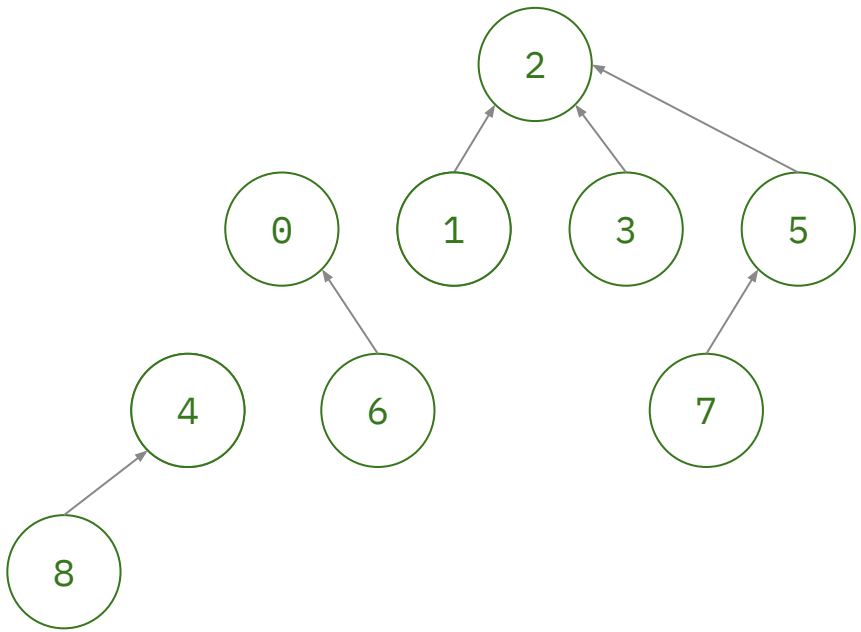
`[-1, 2, -5, 2, -2, 2, -1, 5, 4]`



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3); // 2
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

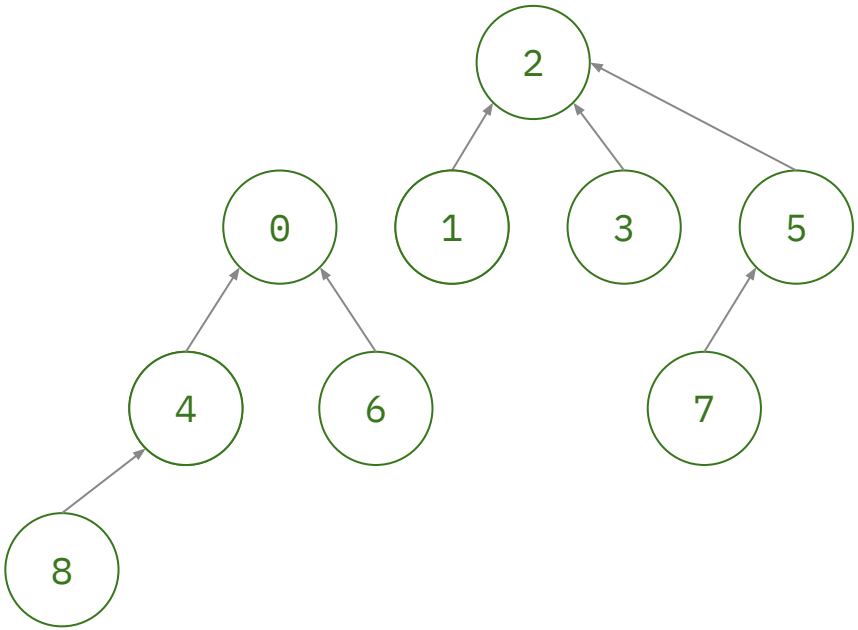
```
[-2, 2, -5, 2, -2, 2, 0, 5, 4]
```



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3); // 2
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

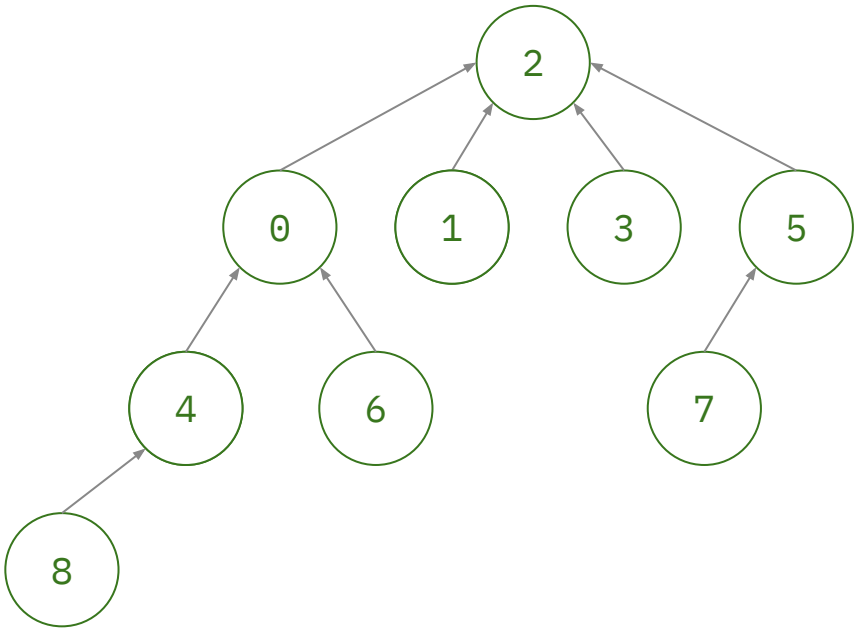
```
[-4, 2, -5, 2, 0, 2, 0, 5, 4]
```



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3); // 2
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

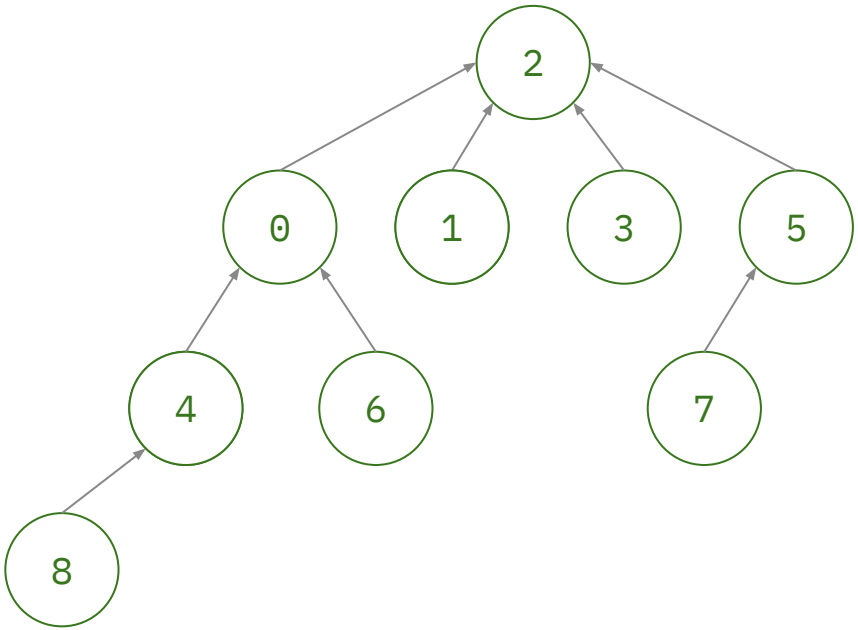
```
[2, 2, -9, 2, 0, 2, 0, 5, 4]
```



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3); // 2
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8); // 2
find(6);
```

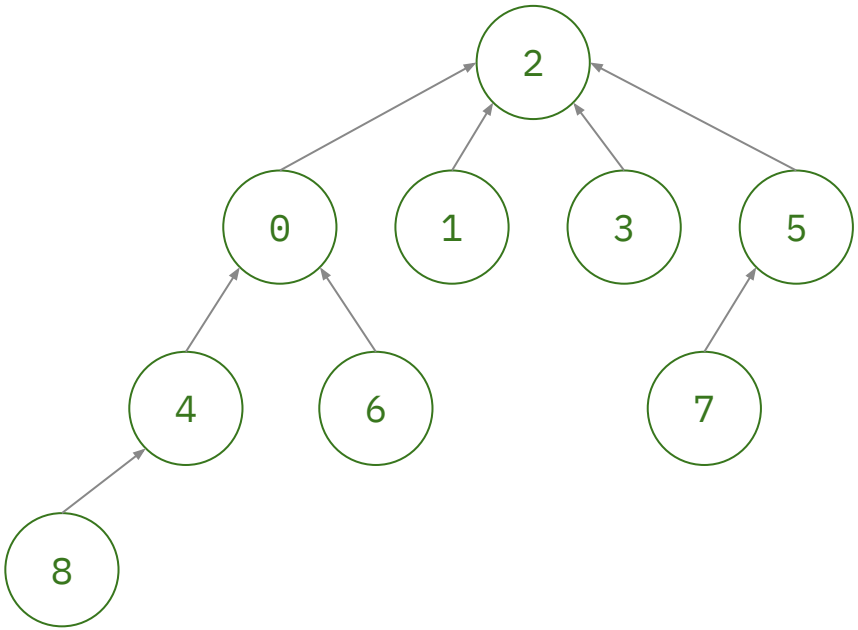
```
[2, 2, -9, 2, 0, 2, 0, 5, 4]
```



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3); // 2
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8); // 2
find(6); // 2
```

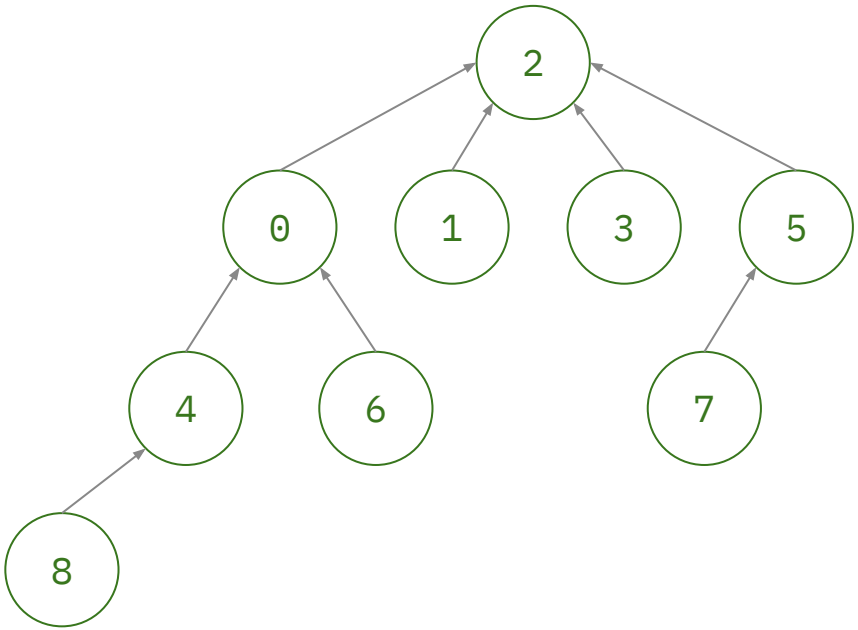
```
[2, 2, -9, 2, 0, 2, 0, 5, 4]
```



2A Disjoint Sets Draw the union find tree and underlying array.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3); // 2
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8); // 2
find(6); // 2
```

```
[2, 2, -9, 2, 0, 2, 0, 5, 4]
```

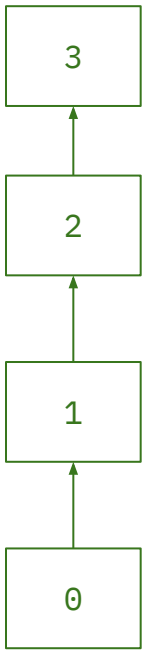


2B Disjoint Sets Draw the worst-case structure and runtime for find (assume this implementation uses QuickUnion).

```
public int find(int val) {
    int p = parent(val);
    if (p == -1) {
        return val;
    } else {
        int root = find(p);
        return root;
    }
}
```


2B Disjoint Sets Draw the worst-case structure and runtime for find (assume this implementation uses QuickUnion).

```
public int find(int val) {
    int p = parent(val);
    if (p == -1) {
        return val;
    } else {
        int root = find(p);
        return root;
    }
}
```



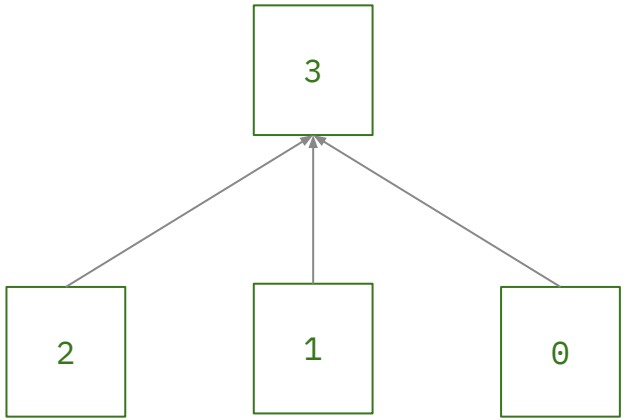
Θ(N): when the set is linear or close to linear

2C Disjoint Sets How do we make find faster with setParent?

```
public int find(int val) {
    int p = parent(val);
    if (p == -1) {
        return val;
    } else {
        int root = find(p);
        return root;
    }
}
```


2C Disjoint Sets How do we make find faster with setParent?

```
public int find(int val) {
    int p = parent(val);
    if (p == val) {
        return val;
    } else {
        int root = find(p);
        setParent(val, root);
        return root;
    }
}
```

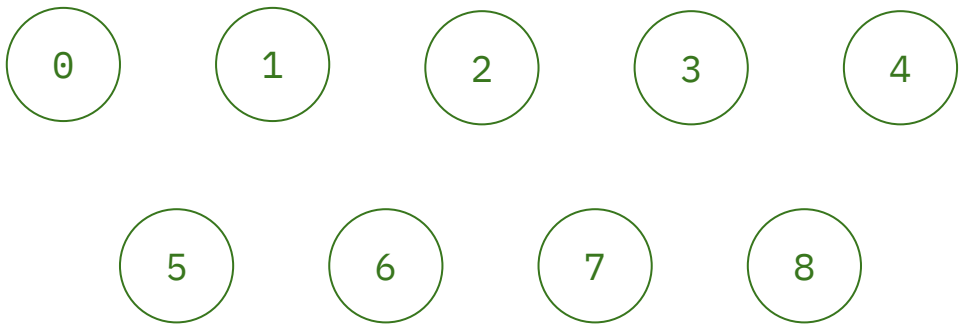


subsequent calls to find() would be completed in amortized O(log*N)!

2D Disjoint Sets Extra: Draw the tree and array of this WQU with PC

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(7, 4);
connect(6, 3);
find(8);
find(6);
```

```
[-1, -1, -1, -1, -1, -1, -1, -1, -1]
```



2D Disjoint Sets Extra: Draw the tree and array of this WQU with PC

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3); // 2
connect(0, 6);
connect(7, 4);
connect(6, 3);
find(8); // 2
find(6); // 2
```

```
[2, 2, -9, 2, 2, 2, 2, 2, 2]
```