CSCI-UA.0480-004
Algorithmic Problem Solving

Brett Bernstein and Sean McIntyre

Lecture 3: Data Structures

Analyzing runtime

- Nested for loops runtime
  - How many `myOperation()` calls?

```
public static void main(String args[]) {
    for (int i = 0; i < 300; i++) {
        for (int j = 0; j < 600; j++) {
            for (int k = 0; k < 200; k++) {
                myOperation(i, j, k);
            }
        }
    }
}
```

Testing exercise

- You receive a time limit exceeded response for an your O(N^3) solution. (1 ≤ N ≤ 100)
    - Abandon the problem
    - Improve the performance of your solution
    - Create tricky test cases and find the bug

Testing exercise

- You receive a time limit exceeded response for an your O(*N*^3) solution. (1 ≤ *N* ≤ **1,000,000**)
  - Abandon the problem
  - Improve the performance of your solution
  - Create tricky test cases and find the bug

Testing exercise

- You receive a runtime error response. Your code runs OK in your machine. What should you do?
    - Abandon the problem
    - Improve the performance of your solution
    - Create tricky test cases and find the bug

Handout exercises

- 5 minutes to read through exercises 1-3

Exercise 1

- Integer radix

```
public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;

    while ((line = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line);

        // Parse
        int x = Integer.parseInt(st.nextToken());
        int y = Integer.parseInt(st.nextToken());
        String baseXIntStr = st.nextToken();

        // Format
        int theInt = Integer.parseInt(baseXIntStr, x);
        String baseYIntStr = Integer.toString(theInt, y);
        System.out.println(baseYIntStr);
    }
}
```

Exercise 2

- Pad with zeros

```
public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;

    while ((line = in.readLine()) != null) {
        // Parse
        int x = Integer.parseInt(line);

        // Format
        System.out.printf("%09d\n", x);

        /* Also valid: */
        // String outputString = String.format("%09d", x);
        // System.out.println(outputString);
    }
}
```

Exercise 3

- Printing decimals (reference)

```
public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;

    while ((line = in.readLine()) != null) {
        // Parse
        double x = Double.parseDouble(line);

        // Format
        System.out.printf("%.3f\n", x);

        /* Also valid: */
        // String outputString = String.format("%.3f", x);
        // System.out.println(outputString);
    }
}
```

Exercise 4

- Set intersection

```
public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;

    LinkedHashSet<Integer> A = new LinkedHashSet<Integer>();
    LinkedHashSet<Integer> B = new LinkedHashSet<Integer>();

    while ((line = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line);

        while (st.hasMoreTokens())
            A.add(Integer.parseInt(st.nextToken()));

        st = new StringTokenizer(in.readLine());

        while (st.hasMoreTokens())
            B.add(Integer.parseInt(st.nextToken()));

        A.retainAll(B); // Set intersection
        System.out.println(A.size());
    }
}
```

Exercise 5

- Greatest Euclidean distance, small
    - Brute force is fine!

Exercise 6

- Greatest Euclidean distance, large
  - Brute force is much too slow
  - Turns out convex hull works

Data Structures Lecture

Linear data structures

1) Static arrays

- `int myArray[] = new int[10];`
- Accessing and setting: O(1) operations
- Don't forget to clear between test cases
  - `Arrays.fill(myArray, 0);`

Linear data structures

2) ArrayLists (convenient resizable arrays)

- `ArrayList myList = new ArrayList();`
- Constructor has one parameter, an integer
  - e.g., `new ArrayList(1000)` instantiates a new ArrayList with initial capacity of 1000 items
  - Default (no param): initial capacity is 10 items
- Unbounded growth (within memory limit of program)

Linear data structures

2) ArrayLists

- Appending to list: amortized O(1) operation
  - When a resize occurs, all elements are copied to a new array, which is O(n) operations
- Inserting to list: O(n) operations
  - Elements are shifted over to accommodate
- If you reuse one between test cases, run `list.clear()` between runs!!
- Reference

Linear data structures

- Example pitfall:
  - Problem description:

    Write a program that finds if an integer is in a list of integers.
  - Sample input:

    1 2 4 7 5 9

    5
  - Sample output:

    yes

Linear data structures

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String line;

// Let's just use one ArrayList for this problem
ArrayList<Integer> myList = new ArrayList<Integer>();

while ((line = in.readLine()) != null) {
     StringTokenizer st = new StringTokenizer(line);

     while (st.hasMoreTokens()) {
          myList.add(Integer.parseInt(st.nextToken()));
     }

     line = in.readLine();

     int x = Integer.parseInt(line);

     if (myList.contains(x)) {
          System.out.println("yes");
     } else {
          System.out.println("no");
     }

     myList.clear(); // Don't forget to clear!
}
```

Linear data structures

- Common operations on arrays and ArrayLists
  - Sorting
    - `Arrays.sort(myArray)` – quicksort, O(n log n)
    - `Collections.sort(myList)` – merge sort, O(n log n)
  - Searching
    - Unsorted list: exhaustive search, O(n)
    - Sorted list: binary search, O(log n)
      - `Arrays.binarySearch()` and `Collections.binarySearch()` – more later

Linear data structures

3) Bitmask

- Treat a primitive int or long as a set of booleans
- Further discussion next class

Linear data structures

4) LinkedList

- O(n) time to access an indexed element
- O(n) to search for an element
- O(n) to insert (or O(1) with a ListIterator)
- Just use an ArrayList

Linear data structures

5) Stack

- LIFO operations: Push, pop
- Useful when a stack could be useful

6) Queue

- FIFO operations: Push, pop
- In Java, implemented as an interface
  - Has a LinkedList data structure backend, not good to search through / insert
  - `Queue<X> myQueue = new LinkedList<X>();`
  - Will be used later in this class

Non-linear data structures

1) Binary search tree

- Java's TreeSet and TreeMap implement a Red-Black tree
  - Self-balancing binary tree
- Cost:
  - Insertion: `myTree.put(x)` – O(log n)
  - Membership: `myTree.containsKey(x)` – O(log n)
  - Remove: `myTree.remove(x)` – O(log n)
  - Fetch (TreeMap): `myTree.get(x)` – O(log n)
- Reference

Non-linear data structures

2) Hash table

- Java implements a standard hash table
  - Buckets (an array) of key-value objects called “Entries”
  - Keys with the same hash codes are stored in the same bucket using a linked list
    - Not LinkedList
  - Collision time/space trade-off regulated by the *load factor* (default 0.75)
    - How full the table can become before growing
  - Also can be given an initial capacity (default 16)

Non-linear data structures

2) Hash table

- Cost:
  - Insertion, fetch, removal, membership: expected O(1)
  - Depends on a good hash function
    - If you make a custom class, ensure you override the hashCode() so collisions are minimized
    - Eclipse is your friend: Source → Generate hashCode() and equals()
- HashMap and HashSet
- Reference

Non-linear data structures

3) Linked hash table

- Convenience class for efficiently traversing hash table keys
  - `for (Entry<K, V> e : myHashTable.getEntries())`
- Java: LinkedHashMap, LinkedHashSet
- Iteration order:
  - Order in which elements were added
- Cost of iterating:
  - Linear in size

Non-linear data structures

4) Heap

- Tree structure
- Each element:
  - Is larger than its parent
  - Is smaller than its children
- Java: PriorityQueue, a binary heap
- Operations:
  - Add: Put the element in the tree – O(log n)
  - Poll: Remove and return top element from the heap tree, i.e., the smallest element – O(log n)

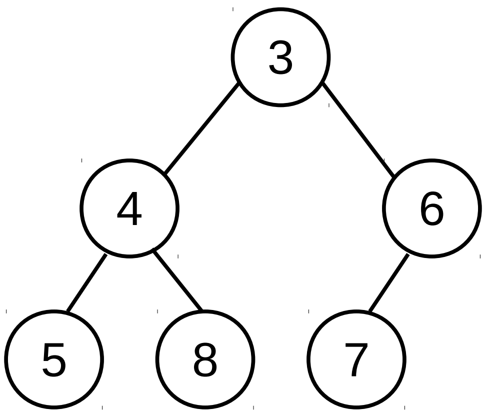
Testing exercise

- You receive a wrong answer response for a very easy problem. What should you do?
  - Abandon the problem
  - Improve the performance of your solution
  - Create tricky test cases and find the bug

Non-linear data structures

4) Heap

- Stored in contiguous memory for fast lookup



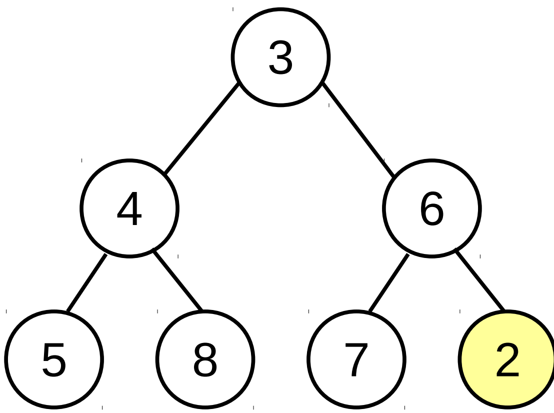
idx	0	1	2	3	4	5	6	7
val	3	4	6	5	8	7		

Parent: `(idx – 1) >>> 1`

Non-linear data structures

4) Heap

- Add 2 – sift up



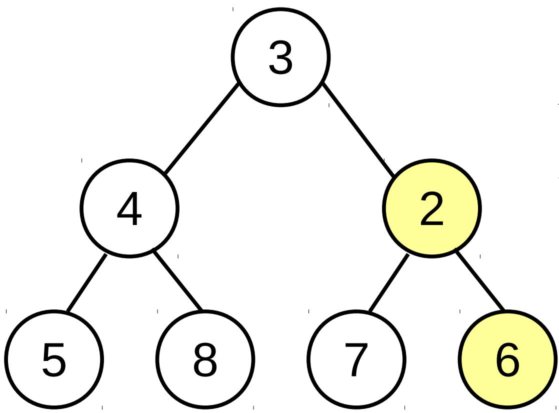
idx	0	1	2	3	4	5	6	7
val	3	4	6	5	8	7	2	

Parent: `(idx – 1) >>> 1`

Non-linear data structures

4) Heap

- Add 2 – swap 2 and 6



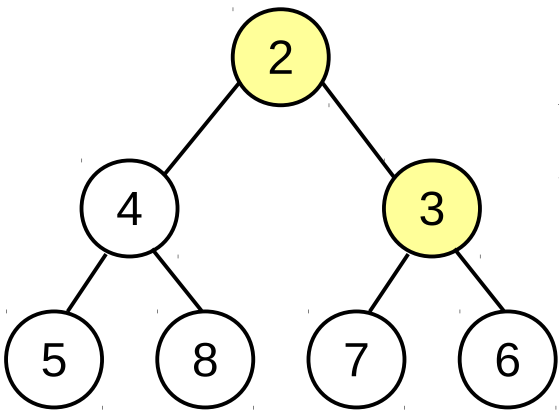
idx	0	1	2	3	4	5	6	7
val	3	4	**2**	5	8	7	**6**	

Parent: `(idx – 1) >>> 1`

Non-linear data structures

4) Heap

- Add 2 – swap 2 and 3



idx	0	1	2	3	4	5	6	7
val	**2**	4	**3**	5	8	7	6	

Parent: `(idx – 1) >>> 1`

Graphs

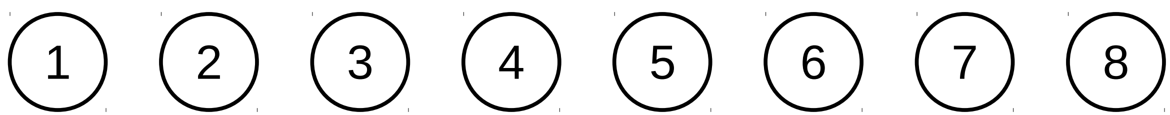
- A set of nodes connected by edges
  - Directed, undirected
  - Cyclic, acyclic
- Represented by:
  1) Adjacency Matrix
  2) Adjacency List
  3) Edge List
- Much more later

Union-find disjoint sets

- Motivation:
  - You want a data structure to quickly union two or more disjoint sets
  - You want to quickly find what set an element belongs to
- How to do this efficiently
  - Make a forest of trees for each element
  - The root of the tree is the set identifier

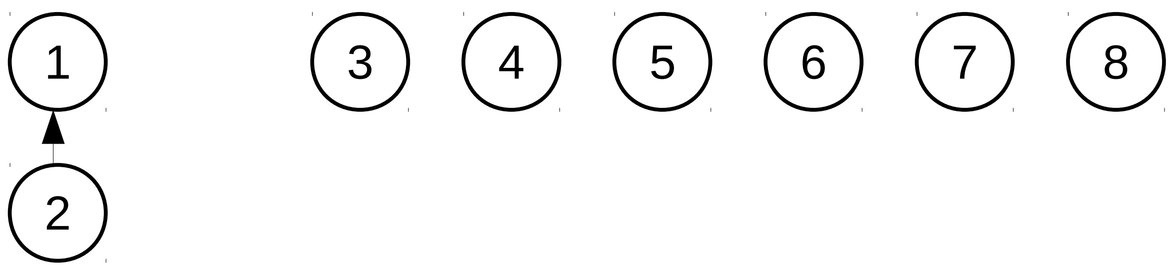
Union-find disjoint sets

- Starting with 8 disjoint sets / trees:



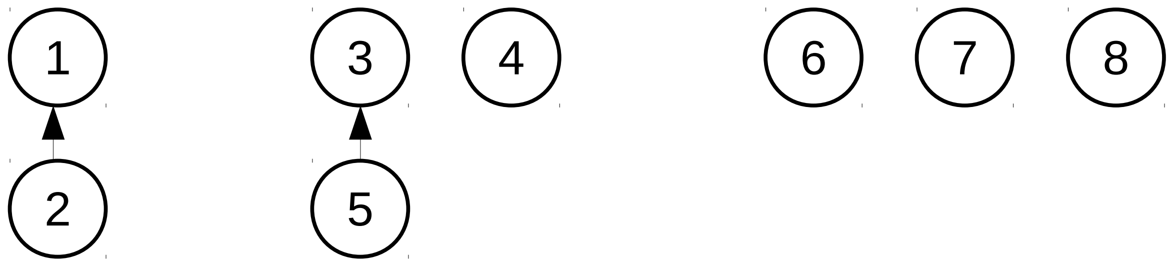
Union-find disjoint sets

- Union the sets containing 1 and 2



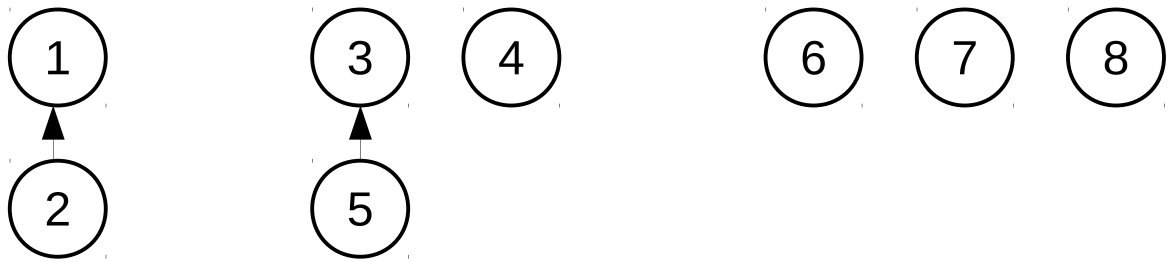
Union-find disjoint sets

- Union the sets containing 3 and 5



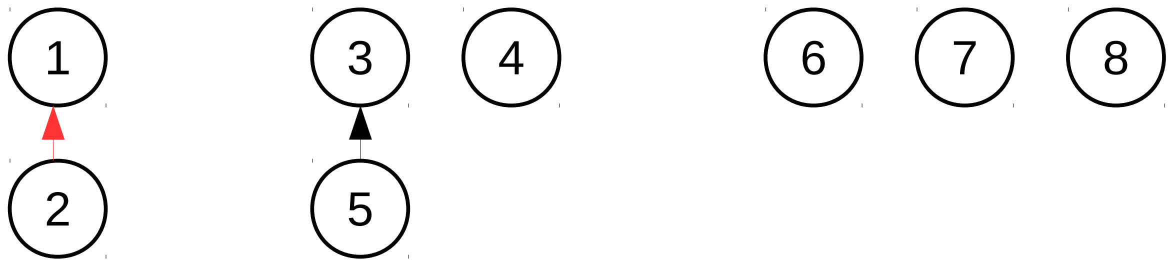
Union-find disjoint sets

- Are 2 and 6 in the same set?



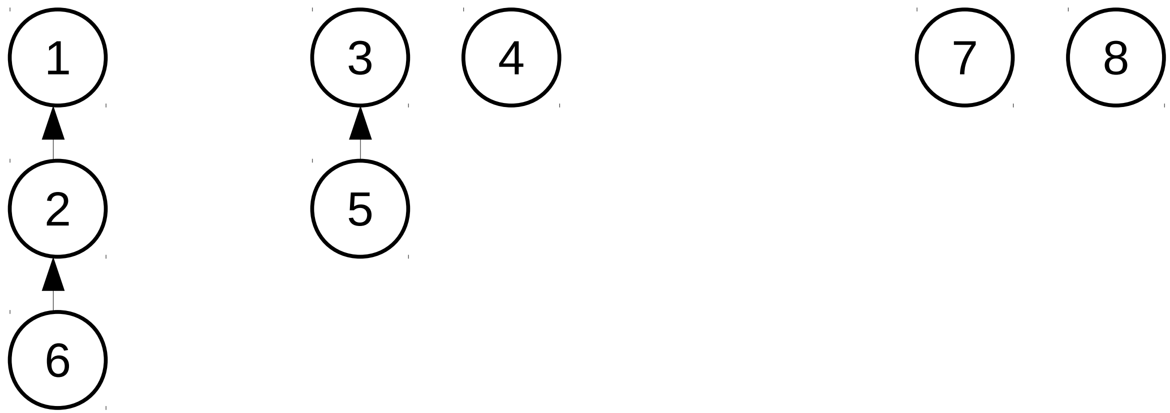
Union-find disjoint sets

- Are 2 and 6 in the same set? No.



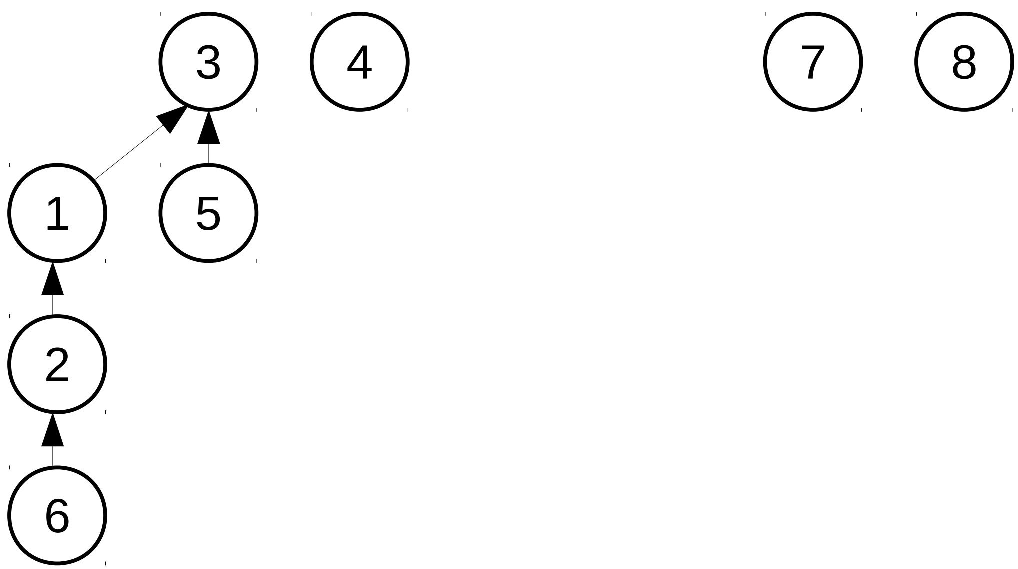
Union-find disjoint sets

- Union the sets containing 2 and 6



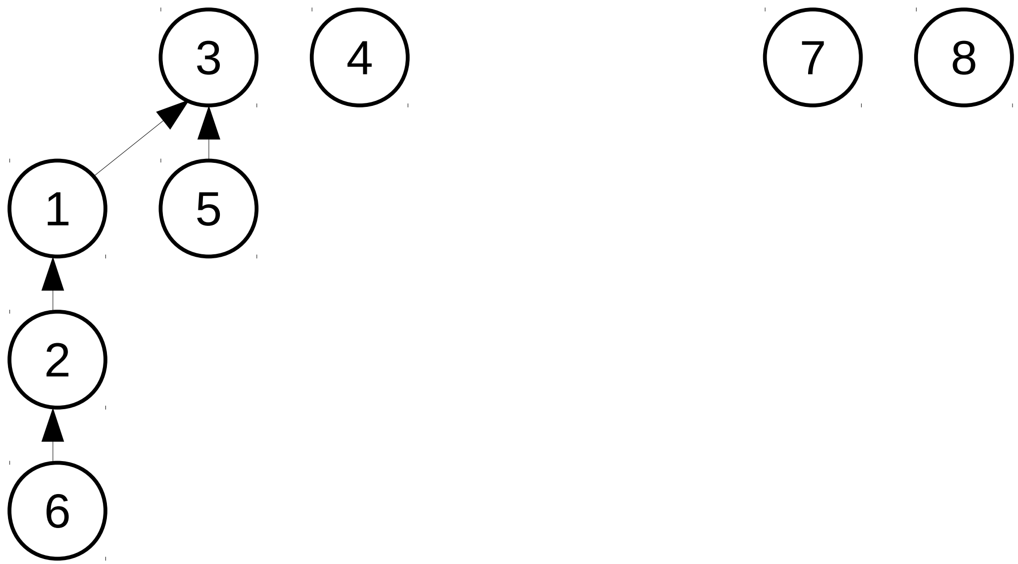
Union-find disjoint sets

- Union the sets containing 5 and 6



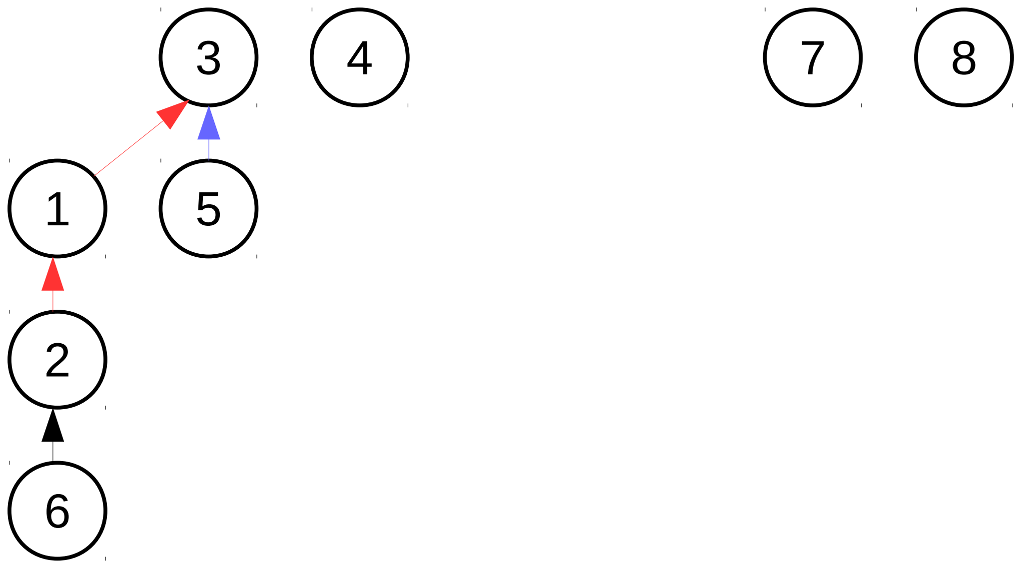
Union-find disjoint sets

- Are 2 and 5 in the same set?



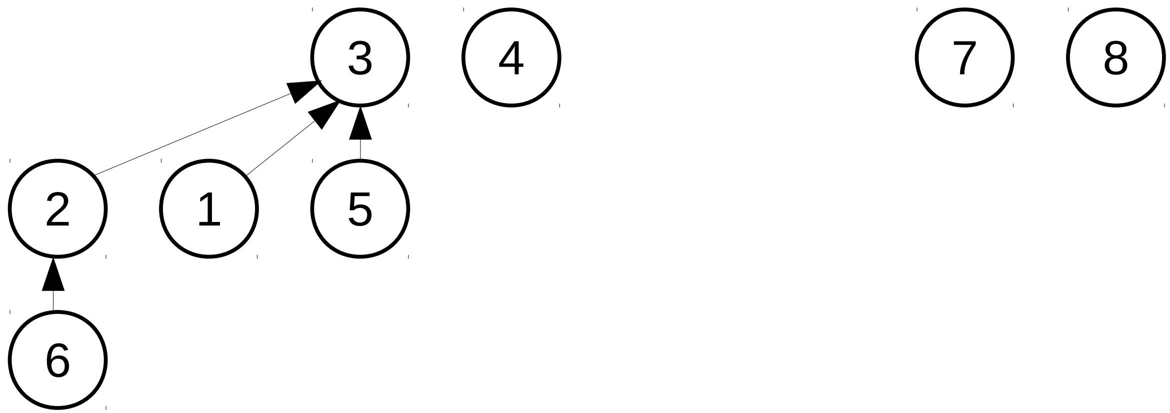
Union-find disjoint sets

- Are 2 and 5 in the same set? Yes.



Union-find disjoint sets

- Path compression after the find



Union-find disjoint sets

- Runtime:
  - Consider N union-find operations to take about O(N) time

Union-find disjoint sets

- Union pseudo-code

```
function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    xRoot.parent := yRoot
```

- Find pseudo-code

```
function Find(x)
    if x.parent != x
       x.parent := Find(x.parent)
    return x.parent
```

Data structure problems

- Read exercises 7-9

Data structure problems

- If we have time…
  - Hardwood Species
  - Minesweeper
  - List of Conquests

Readings

- Book reference:
  - Programming Challenges 2.1 to 2.4.2