CMPT 225: Data Structures & Programming
– Unit 29 –
Sets and Union-Find Structures

Dr. Jack Thomas

Simon Fraser University

Spring 2021

Today's Topics

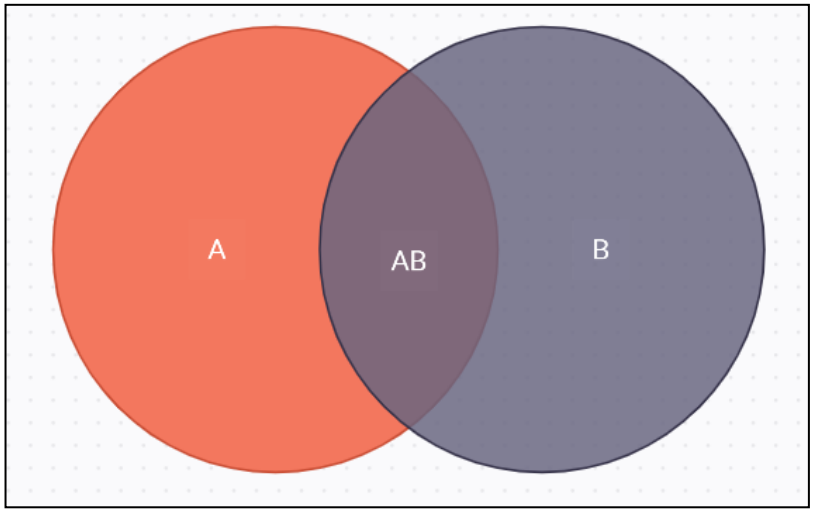
- Sets
- Ordered Sets
- Set Theory Operations
- Mergable Sets
- Union-Find

Switching Back to Data Structures

- Now that we've explored sorting, let's talk about a type of data structure that doesn't need sorting at all – the **Set**.
- A Set is a very high-level data structure that represents **any collection of unique entries**, and that's it.
- Sets are extremely general, being found **high in the inheritance hierarchy** for several other structures and classes we've seen so far.

Mathematical Sets

- The role Sets fulfill for Computer Science is found in mathematics and **Set Theory**.



- They allow us to **apply the algorithms and rules** laid down by that theory on collections of data entries.

The Set ADT

- A collection of distinct objects.
- Extremely general – no explicit notion of keys or even an order, elements are just part of the set or not part of the set.
- Include the following standard methods:
  - **Add**: Adds a given element to the set.
  - **Remove**: Removes (but does not return) a given element from the set.
  - **Contains**: Tells you whether a given element is in a set or not.
  - **Iterator**: Returns a collection of the elements in the set.

Does Java Have a Set?

- Good news, it does!
- There's the **Set interface**, which includes the standard methods from the ADT.

```
String firstElement = "I'm first";
String secondElement = "I'm second";
String thirdElement = "I'm third";
Set exampleSet = new HashSet();
exampleSet.add(firstElement);
exampleSet.add(secondElement);
exampleSet.add(thirdElement);
System.out.println(exampleSet.contains(firstElement));
exampleSet.remove(firstElement);
System.out.println(exampleSet.contains(firstElement));
```

```
true
false
```

Java's Set

- This interface is inherited by **multiple other interfaces** and implemented by a large number of different classes using different underlying data structures.
- This includes **HashSet**, which uses an underlying **Hashtable** and is otherwise a perfectly standard Set, or **AbstractSet**, which provides a barebones abstract class version of Set for you to fill in.

Implementing, Analyzing, and Using Sets

- Being very high level, abstract, and tied to theory means that Sets **don't say much about how they're implemented**.
- As such, it's not very meaningful to analyze the performance of Set as an implemented class, because it **mostly exists in practice to be extended or implemented** by other data structures.
- Its most useful practical application is providing a **common ancestor to different Set-based data structures**, so that methods that target or expect Sets will work for all of them.

Re-Introducing Order

- The first specialization of Sets we'll consider is reintroducing order and comparisons, unsurprisingly called **Ordered Sets**.
- Ordered Sets are still a very high-level structure that doesn't explain how exactly this order is to be implemented or maintained, it just **asserts that the elements in the Set can now be ordered** and gives you methods that assume this is true.

The Ordered Set ADT

- An extension of Set that includes support for a total ordering of the elements.
- The standard methods are expanded:
  - **pollFirst**: Return and remove the smallest element.
  - **pollLast**: Return and remove the largest element.
  - **Ceiling**: Return the smallest element that's greater than or equal to a given element.
  - **Floor**: Return the largest element that's less than or equal to a given element.
  - **Lower**: Returns the greatest element less than a given element.
  - **Higher**: Returns the smallest element greater than a given element.

How About Java and Ordered Sets?

- The **NavigableSet** interface adds the Ordered Set methods to the Set interface.
- This interface is implemented by both **ConcurrentSkipListSet**, which implements an Ordered Set as a Skip List, and **TreeSet**, which implements it as a Red-Black Tree.
- These are both still descendants of Set, which means both of them, HashSet, and the other Set variants **can be cast as Sets and used by methods or classes that take in Sets**.

It’s Time To Do Math To Sets

- Now that we’ve introduced the Set basics, we can start applying the rules found in Set Theory to them.
- The first algorithms we’ll consider are ones that manage **combining two sets** in different ways:
  - Union
  - Intersection
  - Subtraction

Unions

- The Union of two sets is pretty straightforward – it **combines two sets into one**.
- Don’t forget, since the elements of a set are unique, there **shouldn’t be any duplicates**.

(3, 2, 6) U (5, 6, 9) -> (3, 2, 6, 5, 9)

Intersections

- The Intersection of two sets will give you the subset of elements that **appeared in both sets**.

(3, 2, 6) ∏ (5, 6, 9) -> (6)

(it’s actually supposed to be the upside-down capital U but I couldn’t find that one)

Subtractions

- Subtracting one set from another gives you the difference, that is, the **elements of one set that did not appear in the other**.

(3, 2, 6) - (5, 6, 9) -> (3, 2)

Sets and the Generic Merge

- All three of these methods can be described as types of a **generic Merge algorithm**, which looks essentially like the **Merge step from our Merge-Sort algorithm**.
- You take both Sets to be merged, compare the next elements of both, and decide what to do with them based on the outcome of the comparison and the type of Merge you're doing.
- This general description gives us the Set subtype of **Mergable Sets**.

The Mergable Set ADT

- A Set which supports being combined with other Sets.
- Adds the following methods:
  - **Union**: Replaces the set with the union of itself and a given set.
  - **Intersect**: Replaces the set with the intersection of itself and a given set.
  - **Subtract**: Replaces itself with the difference of itself and a given set.

Mergable Sets, Java, and Implementation

- There isn't a particular Mergable Set class or interface in Java because each of the practical results we describe from the theory **can be achieved through using regular functions**, like using .addAll to unite two sets.
- Mergable Sets are still a useful data structure for bridging theory to practice, however, and gives us a chance to introduce the **Template Method** design pattern.

Implementing the General Merge

- Since we know that the three ways to combine Sets are variants of a general Merge, we can write a **general Merge function** that can then be specialized as needed.
- Writing functions in this way that minimizes the amount of duplication between very similar methods follows the Template Method approach, where a mostly-complete template is used **as the basis for each variant of Merge**.

```
abstract class Merge<E> {
    private E a, b;
    private Iterator<E> iterA, iterB;
```

```
private boolean advanceA()
{
    if (iterA.hasNext())
    {
        a = iterA.next(); return true;
    }
    return false;
}
private boolean advanceB()
{
    if (iterB.hasNext())
    {
        b = iterB.next(); return true;
    }
    return false;
}
```

```java
public void merge(Set<E> A, Set<E> B,
                  Comparator<E> comp, Set<E> C) {
    iterA = A.iterator();
    iterB = B.iterator();
    boolean aExists = advanceA();
    boolean bExists = advanceB();
    while (aExists && bExists) {
        int x = comp.compare(a, b);
        if (x < 0) {
            aIsLess(a, C);
            aExists = advanceA();
        } else if (x == 0) {
            bothAreEqual(a,b,C); aExists = advanceA(); bExists = advanceB();
        } else {
            bIsLess(b, C); bExists = advanceB();
        }
    }
    while (aExists) {aIsLess(a, C); aExists = advanceA();}
    while (bExists) {bIsLess(b, C); bExists = advanceB();}
}
```

```
protected void aIsLess(E a, Set<E> C) {

}
protected void bothAreEqual(E a, E b, Set<E> C) {

}
protected void bIsLess(E b, Set<E> C)
{

}
```

- Now we only need to fill in aIsLess, bothAreEqual, and bIsLess as each method requires. For example, for Union:

```
protected  void aIsLess(E a, Set<E> C) {
    C.add(a);
}
protected  void bothAreEqual(E a, E b, Set<E> C) {
    C.add(a);
}
protected  void bIsLess(E b, Set<E> C)
{
    C.add(b);
}
```

What About Splitting Sets Instead of Merging Them?

- Since the Merge operations show how to combine Sets in different ways, we should also have a way to **divide them**.
- A **Partition** describes a collection of **disjoint Sets**, meaning separate sets with no elements in common.
- It's also another distinct type of data structure with its own methods.

The Partition ADT

- A collection of disjoint Sets.
- Reintroduces the notion of positions to Sets,
- Includes the following standard methods:
  - **makeSet**: Create a single-element set out of a given element and returns the position storing it.
  - **Union**: Returns the union of two given sets, while removing them.
  - **Find**: Returns the set containing the element in a given position.

Partitions as Union-Find Structures

- Partitions aren't just another layer of structure that goes on top of Sets – instead, a Partition of Sets is treated by theory as another **distinct data structure** called a **Union-Find Structure**.
- Union-Find Structures are built around efficiently determining which Set among a collection of Sets contains a particular element (the **Find**), and operations for efficiently building and combining Sets (the **Union**).

Java, Partitions, and Union-Finds

- Java **doesn't support Partitioning directly**, at least not in the way that it supports Sets.
- You can **recreate the practical effects** of a Partition with an array or list, plus using existing Set methods.
  - Make a new Set and add a single element, or use the **.addAll()** function to add one set to another, or use **.contains()** on each Set in a collection of Sets to find a particular element.
- This **doesn't capture the efficiency or theoretical simplicity of a Union-Find Structure**, which means we'll have to implement our own..

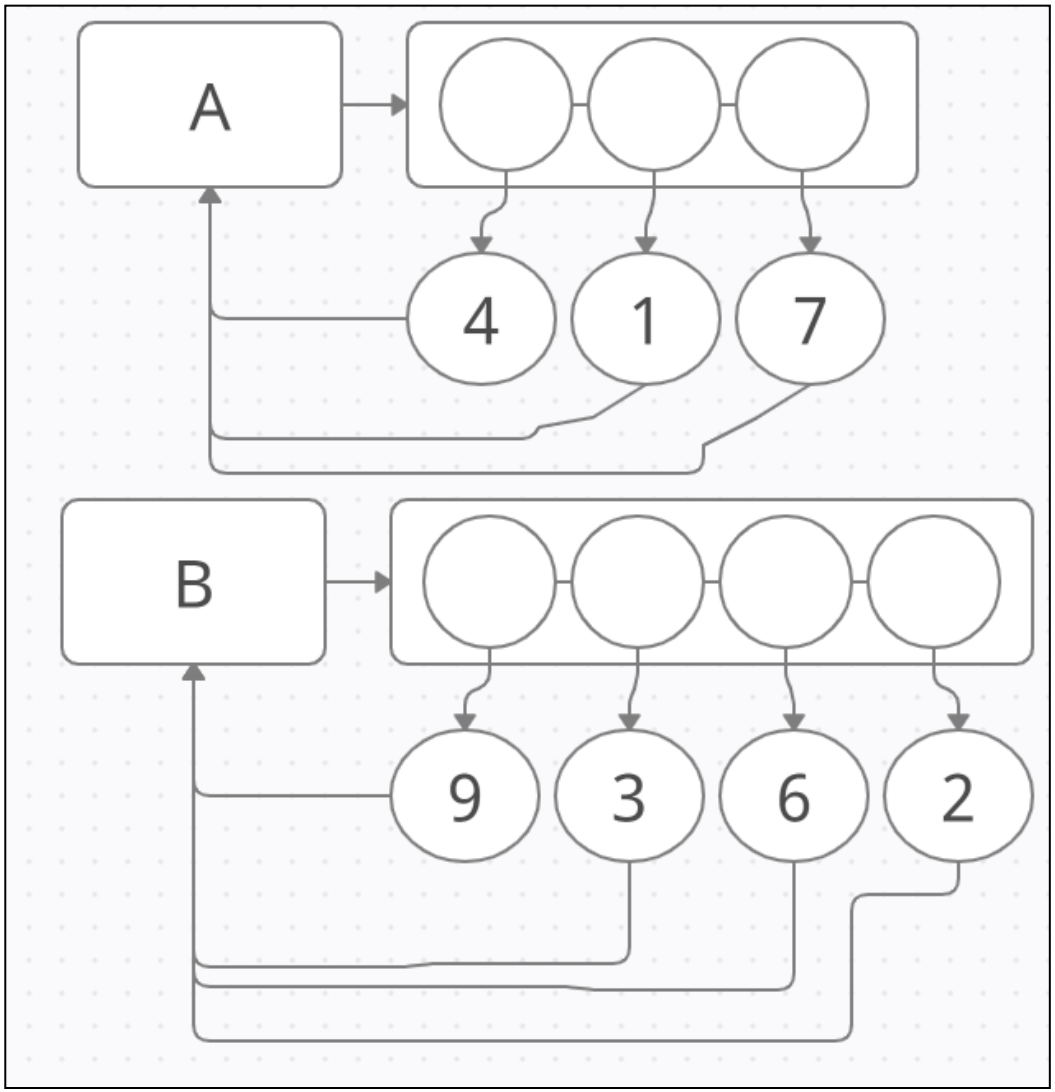
Implementing Union-Find Structures

- Setting aside Sets and Partitions, the goal of a Union-Find Structure is to **organize n elements across some number of groups** (which we'll still call sets, just not Sets).
- The simples way is to have a **collection of sequences**, like a list of lists.
- The top-level list **stores a reference to each list**, representing a set.

Using Positions and Elements

- Each list-set will store **Positions**, a type of node that contains **a reference to the element** stored in that position.
- It also stores **a reference to the list-set that contains it**, making our elements location-aware. This makes finding what set a given element belongs to or creating a new set a constant-time operation, as the relevant information is stored locally.

List-Based Union-Find Structure



Uniting Two List-Sets in a Union-Find

- When **performing a Union on two sets**, remove all of the positions from the smaller list and add them to the end of the larger one.
- Don’t forget to **update each position’s set reference** to point to their new list-set.
- Since at worst this will take O(n/2), we can say this **Union** operation is **O(n)**.

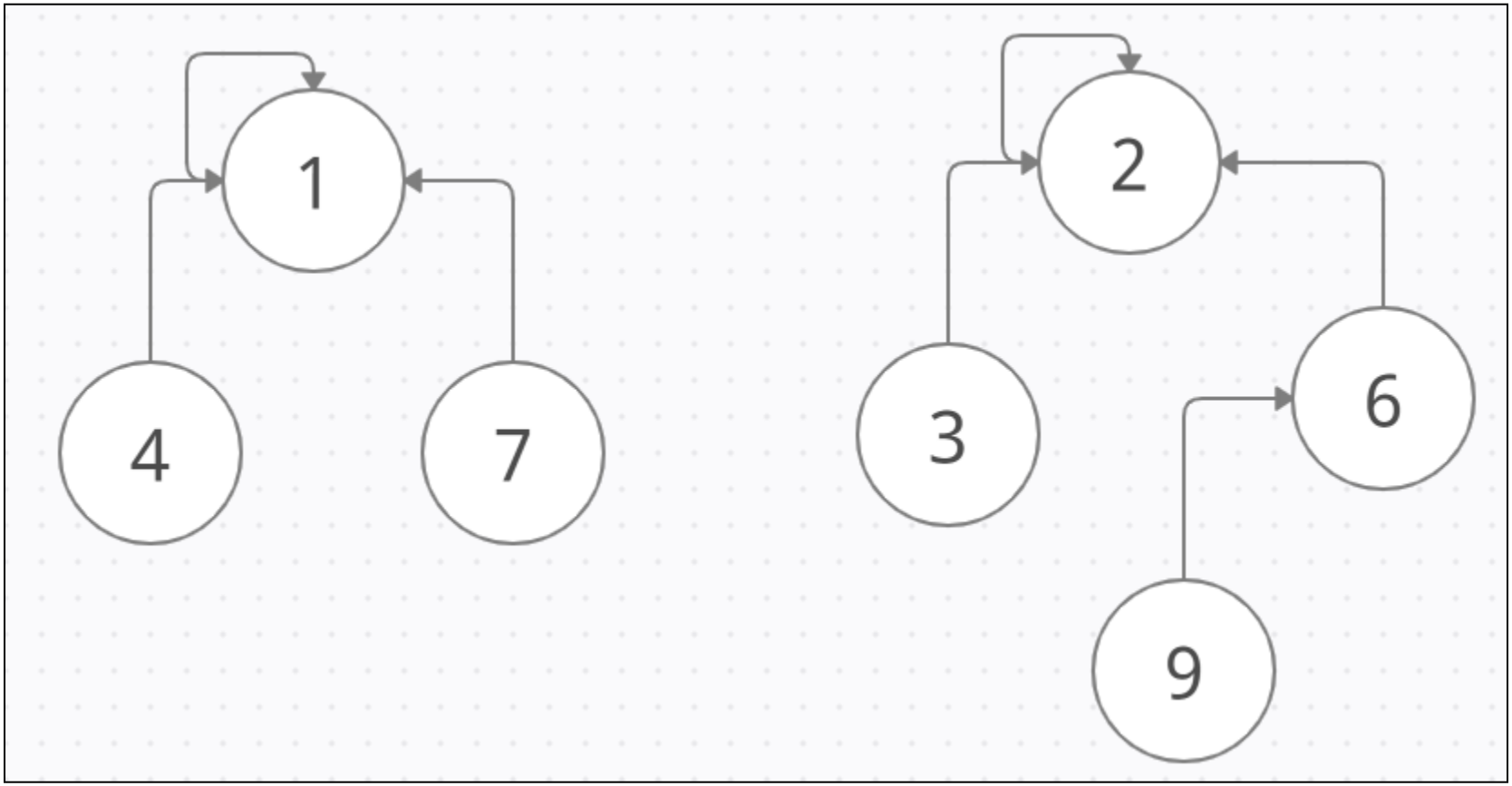
Performance of the List-Based Union-Find Structure

- Performance for a Union-Find structure is essentially measured by how long it takes to build all of the sets (one for each element), then **use Union and Find to assemble them into the desired collection of disjoint sets**, possibly up to one big Set containing everything.
- This will take us **O(n log n)**, since Making and Finding for sets is constant while Unions are O(n), and it'll take O(log n) to go from n Sets of size 1 to 1 Set of size n (remember what we learned about Tree height!)

Tree-Based Union-Find Structure

- An **alternative approach** to implementing a Union-Find Structure would be using **Trees**, where each Tree is a separate set.
- Aside from changing the Position nodes to be Tree nodes storing references to multiple children, we also no longer have a Set reference pointing to a parent list – instead, **the parent reference of each Tree node is also a Set reference**, and the **root of each separate Tree-Set points back to itself**.

Tree-Based Union-Find Structure

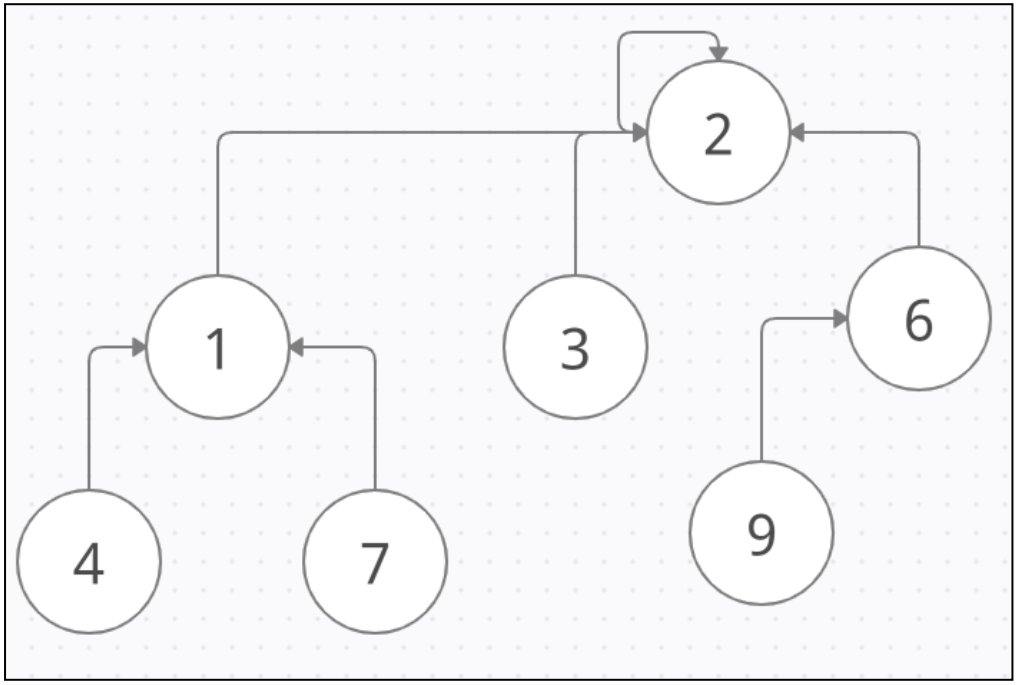


Performance Differences With a Tree-Based Union-Find

- Now uniting two sets can be achieved by **making one Tree a subtree of the other**, by changing one Tree's root's parent reference from itself to the root of the other Tree, a constant-time operation.
- Find, on the other hand, requires **walking up from a given element's Position to the root**, to find out what Set they're in, which in the worst case can take **O(n)**.

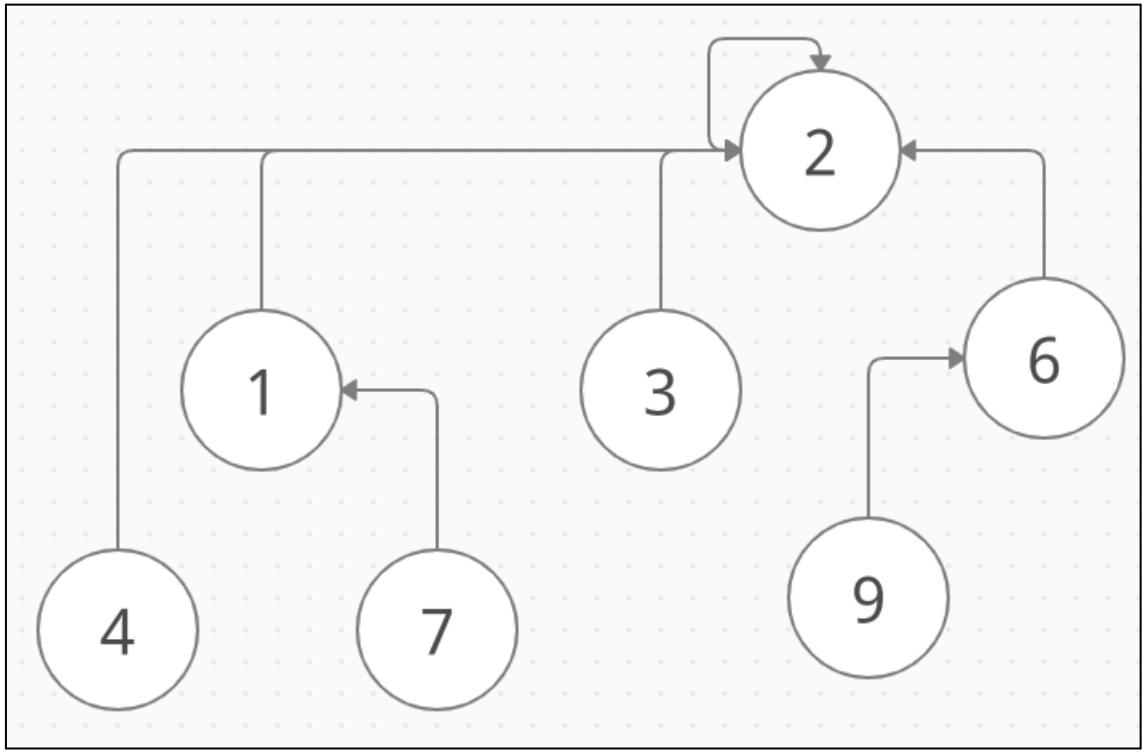
Two Performance-Improving Heuristics

- **Union-by-Size**: Add a size variable to the root of each subtree storing how many positions make up that Tree, and choose the smaller subtree to be anchored to the larger one during Union operations (don't forget to update the size variable for the now-even-larger subtree!).



Two Performance-Improving Heuristics

- **Path Compression**: During a Find operation, on the way to the root, reset the parent reference for each node visited to point to the root of the subtree.



Performance of the Augmented Tree-Based Union-Find Structure

- Performing a series of **n Union and Find operations now takes O(n log* n)** time, where log * n is the **log-star function**, the inverse of the tower-of-twos function.
- Log*n is “the number of times that one can iteratively take the logarithm (base 2) of a number before getting a number smaller than 2”.
- Essentially what this means is “Yes it’s O(n log n) but it’s a faster O(n log n) than if you did it the other way.

A Reminder of Why We're Doing This

- Set-based data structures, including Partitions and Union-Find Structures, are all about **establishing an element's membership**.
- There will be times while programming where what you need is to collect data according to their group and confirm that everything is stored in the right Set, without being particular about how they're stored within that Set – **Union-Find Structures may be the right basis for that kind of work**.
- As for doing other operations with a Set, like finding the largest or smallest element, we'll be tackling those next in **Selections**.

Recap – Set the Record Straight

- **Sets** are a very general type of data structure that collects distinct elements.
- Sets are very high up in the theory and Java programming hierarchies, leading to **Ordered Sets** and **Mergable Sets**.
- Mergable Sets can be implemented with the **generic Merge method**, built over a **Template Me**thod design pattern that can be specialized into three different methods.
- **Partitions** are disjoint collections of Sets, while Union-Find Structures are a distinct data structure based on the Partition.
- We can implement our own **Union-Find Structure** using either **Lists** or **Trees**.