Advanced Object-Oriented Programming

CPT204 – Lecture 9

Erick Purwanto

Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT204 Advanced Object-Oriented Programming

Lecture 9

Defensive Programming, Immutability, Iterator, Disjoint Sets

Welcome !

- Welcome to Lecture 9 !

- In this lecture we are going to
  - learn about how to avoid debugging, in particular, using defensive programming by assertion
  - continue to learn about immutability
  - learn about iterator
  - learn about a sophisticated data structure called Disjoint Sets (DS)

Part 1: Avoiding Debugging

- The first part of the lecture is about how to avoid debugging entirely, or keep it easy when you have to do it

- We're going to discuss about two general ways to avoid painful debugging:

  - **Make bugs impossible by design**
    We'll review some techniques that we've already talked about:
    static checking, dynamic checking, and immutability
    - *we will talk more about immutability in second part*

  - **Localize bugs to a small part of the program**,
    so that finding the bug becomes easier,
    with the following ideas:
    - *assertions*
    - *incremental development*
    - *scope minimization*

First Defense: Make Bugs Impossible

- The best defense against bugs is to make them impossible by design
- **Static checking** is one way that we've already talked about
  - For example, Java prevents you from inadvertently passing an int value to a function expecting a String parameter, and it won't even allow your code to compile
  - Static checking eliminates many bugs by catching them at compile time
- We also saw some examples of **dynamic checking**
  - For example, Java makes array overflow bugs impossible by catching them dynamically
  - If you try to use an index outside the bounds of an array or a List, then Java automatically produces an error at runtime
  - Older languages like C and C++ *silently allow* the bad access, which leads to bugs and security vulnerabilities such as buffer overflow

Make bugs impossible by Immutability (1)

- **Immutability** (immunity from change) is another design principle that prevents bugs
  - **An immutable type** is a type whose *values* can *never change* once they have been created
- Recall that String is an immutable type
  - There are no methods that you can call on a String that will change the sequence of characters that it represents — Strings can be passed around and shared without fear that they will be modified by other code
- Recall also Java gives us immutable references: variables declared with the keyword **final**, which can be assigned once but never reassigned
  - It's good practice to use final for declaring the parameters of a method and as many local variables as possible
  - Like the type of the variable, these declarations are important documentation, useful to the reader of the code and statically checked by the compiler

Make bugs impossible by Immutability (2)

- Consider this example:
  - `final char[] letters = new char[] {'a', 'e', 'i', 'o', 'u'};`
  - The vowels variable is declared `final`, but is it really unchanging?

- Which of the following statements will be illegal (caught statically by the compiler) and which will be allowed?
  - `letters = new char[] {'x', 'y', 'z'};`
  - `letters[0] = 'z';`

- You will have to answer in Lecture Quiz 9
- Be *careful* about what final means!
  - It *only* makes the reference immutable,
    not necessarily the object that the reference points to!

Second Defense: Localize Bugs (1)

- If we can't prevent bugs, we can try to **localize** them to a small part of the program so that we don't have to look too hard to find the cause of a bug
- When localized to a single method or small module, bugs may be found simply by studying the program text
- We already talked about *fail fast*: the earlier a problem is observed (the closer to its cause), the easier it is to fix
- We'll talk about a particular technique for failing fast, called **assertions**
  - Let's begin with a simple example:

```
/**
 * @param x   requires x >= 0
 * @return approximation to square root of x
 */
public double sqrt(double x) { ... }
```

Second Defense: Localize Bugs (2)

- Now suppose somebody calls sqrt with *a negative argument*

- What's the best behavior for sqrt?
  - Since the caller has failed to satisfy the requirement that x should be nonnegative, sqrt is *no longer* bound by the terms of its contract, so it is technically free to do whatever it wants: return an arbitrary value, enter an infinite loop, or even melt down the CPU

- Since the bad call indicates a bug in the caller, however, the most useful behavior would point out the bug as early as possible

Second Defense: Localize Bugs (3)

- We do this by inserting a runtime check of the precondition, for example:

```java
/**
 * @param x  requires x >= 0
 * @return approximation to square root of x
 */
public double sqrt(double x) {
    if (! (x >= 0)) throw new IllegalArgumentException("required
                                        x >= 0, but was: " + x);

    ...
}
```

- When the precondition is not satisfied, this code terminates the program by
  throwing an unchecked `IllegalArgumentException`
    - The effects of the caller's bug are prevented from propagating

Assertions (1)

- Checking preconditions is an example of **defensive programming**
- Real programs are rarely bug-free
  - Defensive programming offers a way to mitigate the effects of bugs even if you don't know where they are

- It is *common practice* to define a procedure for these kinds of defensive checks, usually called `assert`:

```
assert (x >= 0);
```

- This approach abstracts away from what exactly happens when the assertion fails.
  - The failed assert might exit, might record an event in a log file, or might even email a report to a maintainer

Assertions (2)

- Assertions have the added benefit of documenting an assumption about the state of the program at that point
  - To somebody reading your code, `assert` `(x >= 0)` says "at this point, it should always be true that x >= 0."
  - Unlike a comment, however, an assertion is executable code that enforces the assumption at runtime

- In Java, runtime assertions are a built-in feature of the language
  - The simplest form of the assert statement takes a boolean expression, and *throws an* `AssertionError` if the boolean expression evaluates to false:

    `assert x >= 0;`

Assertions (3)

- An assert statement may also include ***a description expression***, which is usually a string, but may also be a primitive type or a reference to an object
  - The description is printed in an error message when the assertion fails, so that it can be used to provide additional details to the programmer about the cause of the failure
  - The description follows the asserted expression, separated by a colon
    - For example:
      ```
      assert (x >= 0) : "x is " + x;
      ```
  - If `x == -1`, then this assertion fails with the error message: `x is -1` along with a stack trace that tells you where the assert statement was found in your code and the sequence of calls that brought the program to that point
    - This information is often enough to get started in finding the bug

Assertions (4)

- A serious problem with Java assertions is that assertions are **off by default**
  - If you just run your program as usual, none of your assertions will be checked!
- Java's designers did this because checking assertions can sometimes be costly to performance
  - For example, a procedure that searches an array using binary search has a requirement that the array be sorted
  - Asserting this requirement requires scanning through the entire array, however, turning an operation that should run in logarithmic time into one that takes linear time
- You should be willing to pay this cost *during testing*, since it makes debugging much easier, but not after the program is released to users
- For most applications, however, assertions are not expensive compared to the rest of the code and the benefit they provide in bug-checking is worth that small cost in performance

Assertions (5)

- So you have to enable assertions explicitly by passing **-ea** (which stands for enable assertions) to the Java virtual machine

  - In IntelliJ, enable assertions by going to Run → Edit Configurations → Configuration, putting -ea in the VM options, and click OK (may need to click Modify options → Add VM options first)

Assertions and JUnit (1)

- It's always a good idea to have assertions turned on when you're running JUnit tests
- You can ensure that assertions are enabled using the following test case:

```java
@Test(expected=AssertionError.class)
public void testAssertionsEnabled() {
    assert false;
}
```

- If assertions are turned on as desired, then assert false throws an AssertionError
  - The annotation (expected=AssertionError.class) on the test expects and requires this error to be thrown, so the test passes
  - If assertions are turned off, however, then the body of the test will do nothing, failing to throw the expected exception, and JUnit will mark the test as failing

Assertions and JUnit (2)

- Note that the Java assert statement is a **different** mechanism than the JUnit methods `assertTrue()`, `assertEquals()`, etc
- They all assert a predicate about your code, but are designed for use in different contexts

- The `assert` statement should be used *in implementation code* for defensive checks inside the implementation
- JUnit `assert...()` methods should be used *in JUnit tests* to check the result of a test

- The `assert` statements don't run without -ea, but the JUnit `assert...()` methods always run

What to Assert (1)

Here are some things you should assert:

- **Method argument requirements**, like we saw for `sqrt`
- **Method return value requirements**
  - This kind of assertion is sometimes called a self check
  - For example, the sqrt method might square its result to check whether it is reasonably close to x:

```java
public double sqrt(double x) {
    assert x >= 0;
    double r;
    ... // compute result r
    assert Math.abs(r*r - x) < .0001;
    return r;
}
```

What to Assert (2)

- **Covering all cases**
  - If a conditional statement or switch does not cover all the possible cases, it is good practice to use an assertion to *block the illegal cases*:

```
switch (vowel) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u': return "A";
  default: Assert.fail();
}
```

  - The assertion in the default clause has the effect of asserting that vowel must be one of the five vowel letters

What to Assert (3)

- When should you write runtime assertions?
    - As you write the code, **not** after the fact
    - When you're writing the code, you have the **invariants** in mind

- If you postpone writing assertions, you're less likely to do it, and you're liable to omit some important invariants

What *not* to Assert (1)

- Runtime assertions are not free
  - They can clutter the code, so they must be used judiciously
- Avoid trivial assertions, just as you would avoid uninformative comments
  - For example:

```
// don't do this:
x = y + 1;
assert x == y + 1;
```

- This assertion doesn't find bugs in your code
  - It finds bugs in the compiler or Java virtual machine, which are components that you **should trust** until you have good reason to doubt them
  - If an assertion is obvious from its local context, *leave it out*

What *not* to Assert (2)

- Never use assertions to test conditions that are **external** to your program, such as the existence of files, the availability of the network, or the correctness of input typed by a human user
  - Assertions test the internal state of your program to ensure that it is within the bounds of its specification

- When an assertion fails, it indicates that the program has run off the rails, in some sense, into a state in which it was not designed to function properly
  - Assertion failures therefore indicate **bugs**

- External failures are not bugs and there is no change you can make to your program in advance that will prevent them from happening
  - External failures should be handled using **exceptions** instead

What *not* to Assert (3)

- Many assertion mechanisms are designed so that assertions are executed only during testing and debugging and turned off when the program is released to users
  - Java's assert statement behaves this way

- Since assertions may be disabled, the correctness of your program should **never** depend on whether or not the assertion expressions are executed
  - In particular, asserted expressions should **not** have side-effects
  - For example, if you want to assert that an element removed from a list was actually found in the list, don't write it like this:

```
// don't do this:
assert list.remove(x);
```

- If assertions are disabled, the entire expression is skipped, and x is never removed from the list
  - Write it like this instead:

```
boolean found = list.remove(x);
assert found;
```

Incremental Development

- A great way to localize bugs to a tiny part of the program is **incremental development**
  - Build only a bit of your program at a time and test that bit thoroughly before you move on
  - That way, when you discover a bug, it's more likely to be in the part that you just wrote, rather than anywhere in a huge pile of code

- Recall that our lectures on testing have talked about two techniques that help with incremental development:
  - **Unit testing**: when you test a module in isolation, you can be confident that any bug you find is in that unit — or maybe in the test cases themselves
  - **Regression testing**: when you're adding a new feature to a big system, run the regression test suite as often as possible. If a test fails, the bug is probably in the code you just changed

Modularity

- **Modularity** means dividing up a system into components, or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system
  - The opposite of a modular system is a **monolithic** system — big and with all of its pieces tangled up and dependent on each other

- A program consisting of a single, very long `main()` function is monolithic — harder to understand and harder to isolate bugs in
  - By contrast, a program broken up into small functions and classes is more modular

Encapsulation (1)

- **Encapsulation** means building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior and bugs in other parts of the system can't damage its integrity

- One kind of encapsulation is **access control**: using public and private to control the visibility and accessibility of your variables and methods
  - A *public* variable or method can be accessed by any code (assuming the class containing that variable or method is also public)
  - A *private* variable or method can only be accessed by code in the same class. Keeping things private as much as possible, especially for variables, provides encapsulation, since it limits the code that could inadvertently cause bugs

Encapsulation (2)

- Another kind of encapsulation comes from **variable scope**

  - The scope of a variable is the portion of the program text over which that variable is defined, in the sense that expressions and statements can refer to the variable

  - A method parameter's scope is the body of the method
  - A local variable's scope (in Java) extends from its declaration to the closing curly brace of the block around the declaration

  - Keeping variable scopes ***as small as possible***makes it much easier to reason about where a bug might be in the program

Encapsulation (3)

- For example, suppose you have a loop like this:

```
for (i = 0; i < 100; i++) {

    ...

    doSomeThings();

    ...

}
```

and you've discovered that this loop keeps running forever, `i` never reaches 100

- Somewhere, somebody is changing `i`
  - But where?

Encapsulation (4)

- If `i` is declared as a global variable like this:

```java
public static int i;
...
for (i = 0; i < 100; i++) {
    ...
    doSomeThings();
    ...
}
```

then its scope is the entire program!

- It might be changed anywhere in your program: by `doSomeThings()`, by some other method that `doSomeThings()` calls, or by a concurrent thread running some completely different code

Encapsulation (5)

- But if `i` is instead declared as a local variable with a narrow scope, like this:

```
for (int i = 0; i < 100; i++) {
    ...
    doSomeThings();
    ...
}
```

then the only place where `i` can be changed is within the for statement — in fact, only in the `...` parts that we've omitted

- You don't even have to consider `doSomeThings()` because `doSomeThings()` doesn't have access to this local variable

Minimizing Variable Scope (1)

Minimizing the scope of variables is a powerful practice for bug localization

Here are a few rules that are good for Java:

- Always declare a loop variable in the for-loop initializer

    - Don't declare it before the loop:

    ```java
    int i; // don't do this
    for (i = 0; i < 100; i++) {
    ```
    This makes the scope of the variable the entire rest of the outer curly-brace block containing this code

    - Do this instead:

    ```java
    for (int i = 0; i < 100; i++) {
    ```
    which makes the scope of i limited just to the for loop

Minimizing Variable Scope (2)

- Declare a variable only when you first need it and in the innermost curly-brace block that you can
  - Variable scopes in Java are **curly-brace blocks**, so put your variable declaration in the **innermost** one that contains all the expressions that need to use the variable
  - Don't declare all your variables at the start of the function — it makes their scopes unnecessarily large
  - But note that in languages without static type declarations, like Python and Javascript, the scope of a variable is normally the entire function anyway, so you **cannot** restrict the scope of a variable with curly braces

Minimizing Variable Scope (3)

- Avoid global variables
  - Using global variables is very bad idea, especially as programs get large
  - Global variables are often unwisely used as a shortcut to provide a parameter to several parts of your program, but that's a **terrible** idea
  - It's better to just pass the parameter into the code that needs it, rather than putting it in global space where it can inadvertently reassigned

Part 2: Mutability, Immutability, Iterator

- In the second part of the lecture, we are going to discuss more about the dangers of mutable data types, the advantages of immutability, and how to create an iterator

- Recall from when we discussed snapshot diagrams that some objects are immutable: once created, they always represent the same value
- Other objects are mutable: they have methods that change the value of the object

- `String` is an example of an immutable type
  - a `String` object always represents the same string
- `StringBuilder` is an example of a mutable type
  - it has methods to delete parts of the string, insert or replace characters, etc

Mutability (1)

- Since `String` is immutable, once created, a `String` object always has the same value
  - To add something to the end of a `String`, you have to create a new `String` object:

```
String s = "a";
s = s.concat("b");
// s+="b" and s=s+"b" also mean the same thing
```

- By contrast, `StringBuilder` objects are mutable
  - This class has methods that change the value of the object, rather than just returning new values:

```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```
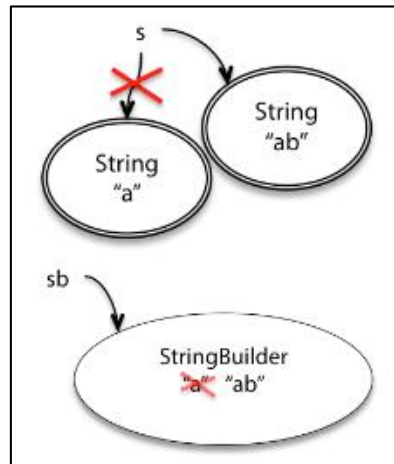
- `StringBuilder` has other methods as well, for deleting parts of the string, inserting in the middle, or changing individual characters

Mutability (2)

- *So what?*
  - In both cases, you end up with `s` and `sb` referring to the string of characters `"ab"`

- The difference between mutability and immutability doesn't matter much when there's only one reference to the object
  - But there are big differences in how they behave when *there are other references to the object*

Mutability (3)

- For example, when another variable `t` points to the same `String` object as `s`, and another variable `tb` points to the same `StringBuilder` as `sb`, then the differences between the immutable and mutable objects become more evident:

```
String t = s;
t = t + "c";


StringBuilder tb = sb;
tb.append("c");
```
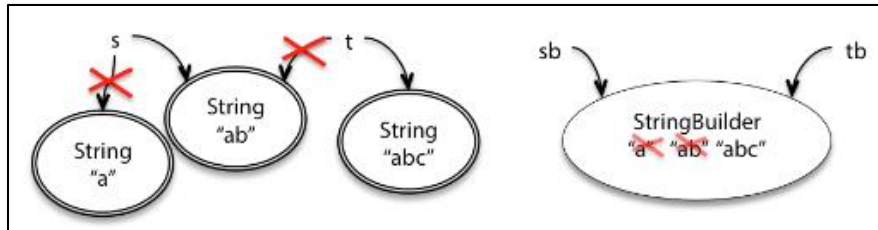

- This shows that changing `t` had **no effect** on `s`,
  but changing `tb` **affected** `sb` **too** — possibly to the surprise of the programmer
- That's the essence of the problem we're going to look at in this lecture

StringBuilder (1)

- Since we have the immutable `String` class already, why do we even need the mutable `StringBuilder` in programming?
  - A common use for it is to concatenate a large number of strings together
  - Consider this code:

```
String s = "";
for (int i = 0; i < n; i++) {
    s = s + i;
}
```

- Using immutable strings, this makes a lot of temporary copies — the first number of the string ("0") is actually copied n times in the course of building up the final string, the second number is copied n-1 times, and so on
  - It actually costs $O(n^2)$ time just to do all that copying, even though we only concatenated n elements

StringBuilder (2)

- StringBuilder is designed to minimize this copying
    - It uses a simple but clever internal data structure to avoid doing any copying at all until the very end when you ask for the final String with a toString() call:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; i++) {
    sb.append(String.valueOf(i));
}
String s = sb.toString();
```

- Getting good performance is one reason why we use mutable objects
- Another reason is convenient sharing: two parts of your program can communicate more conveniently by sharing a common mutable data structure

Risks of Mutation

- Mutable types seem much more powerful than immutable types
  - If you were shopping in the Datatype Supermarket and you had to choose between a boring immutable String and a super-powerful-do-anything mutable `StringBuilder`, *why would you choose the immutable one?*
  - `StringBuilder` should be able to do everything that `String` can do, plus `set()` and `append()` and everything else

- The answer is that immutable types are safer from bugs, easier to understand, and more ready for change
  - Mutability makes it harder to understand what your program is doing and much harder to enforce contracts

- Let us look at the following two examples that illustrate why

Risky Example 1: Passing Mutable Values (1)

- Let's start with a simple method that sums the integers in a list:

```java
/*
 * @return the sum of the numbers in the list
 */
public static int sum(List<Integer> list) {
    int sum = 0;
    for (int x : list)
        sum += x;
    return sum;
}
```

Risky Example 1: Passing Mutable Values (2)

- Suppose we also need a method that sums the absolute values
- Following good DRY practice (Don't Repeat Yourself), the implementer writes a method that uses `sum()`:

```java
/** @return the sum of the absolute values of the numbers in the list */
public static int sumOfAbsoluteValues(List<Integer> list) {
    // let's reuse sum() because DRY, so first we take absolute values
    for (int i = 0; i < list.size(); ++i)
        list.set(i, Math.abs(list.get(i)));
    return sum(list);
}
```

- Notice that this method does its job by *mutating the list directly*
  - It seemed sensible to the implementer because it's more efficient to reuse the existing list

In-Class Quiz 1

- But the resulting behavior will be very surprising to anybody who uses it!
  - Here is an example:

```java
// meanwhile, somewhere else in the code
public static void main(String[] args) {
    // ...
    List<Integer> myData = Arrays.asList(-5, -2, -10);
    System.out.println(sumOfAbsoluteValues(myData));
    System.out.println(sum(myData));
}
```

- What will this code print?

17 and 17

Risky Example 1: Passing Mutable Values (4)

- Let's think about the key lessons of the risky example 1 here:

  - *Is it safe from bugs?*

    - In this example, it's easy to blame the implementer of sumAbsolute() for going beyond what its spec allowed

    - But really, passing mutable objects around is ***a latent bug***

    - It's just waiting for some programmer to inadvertently mutate that list, often with very good intentions like reuse or performance, but resulting in a bug that may be very hard to track down.

  - *Is it easy to understand?*

    - When reading `main()`, what would you assume about `sum()` and `sumOfAbsoluteValues()`?

    - It is not visible to the reader that `myData` gets changed by one of them

Risky Example 2: Returning Mutable Values (1)

- We just saw an example where passing a mutable object to a function caused problems : what about returning a mutable object?

- Let's consider `Date`, one of the built-in Java classes
  - Date happens to be a **mutable** type
  - Suppose we write a method that determines the first day of spring:

```java
/** @return the first day of spring this year */
public static Date startOfSpring() {
    return askGroundhog();
}
```

Simply think that this method somehow returns a `Date` object

- Here we're using the Groundhog algorithm (watch the movie Groundhog Day!) for determining when spring starts

Risky Example 2: Returning Mutable Values (2)

- Clients start using this method, for example to plan their big parties:

```java
// somewhere else in the code…
public static void partyPlanning() {
    Date partyDate = startOfSpring();
    // ...
}
```

- All the code works and people are happy
- Now, independently, two things happen

Risky Example 2: Returning Mutable Values (3)

- First, the implementer of `startOfSpring()` realizes that the groundhog is starting to get annoyed from being constantly asked when spring will start
- So the code is rewritten to ask the groundhog at most once and then cache the groundhog's answer for future calls:

```java
/** @return the first day of spring this year */
public static Date startOfSpring() {
    if (groundhogAnswer == null) groundhogAnswer = askGroundhog();
    return groundhogAnswer;
}
private static Date groundhogAnswer = null;
```

(Aside: note the use of a private static variable for the cached answer. Would you consider this a global variable or not?)

Risky Example 2: Returning Mutable Values (4)

- Second, one of the clients of `startOfSpring()` decides that the actual first day of spring is too cold for the party, so the party will be exactly a month later instead:

```
// somewhere else in the code…
public static void partyPlanning() {
    // let's have a party one month after spring starts!
    Date partyDate = startOfSpring();
    partyDate.setMonth(partyDate.getMonth() + 1);
    // uh-oh... what just happened?
}
```

(Aside: this code also has a latent bug in the way it adds a month.
Why? What does it implicitly assume about when spring starts?)

Risky Example 2: Returning Mutable Values (5)

- What happens when these two decisions interact?
  - Even worse, think about who will first discover this bug — will it be `startOfSpring()`? Will it be `partyPlanning()`?
- The key lessons from this second example:
  - Is it safe from bugs? No — again we had a latent bug that reared its ugly head
  - Is it ready for change? Obviously the mutation of the date object is a change, but that's not the kind of change we're talking about when we say "ready for change"
    - Instead, the question is whether the code of the program can be easily changed without rewriting a lot of it or introducing bugs
    - Here we had two apparently independent changes by different programmers that interacted to produce a bad bug

Avoiding Bug by Immutability (1)

- In both of these examples — the `List<Integer>` example and the `Date` example — the problems would have been completely avoided if the list and the date had been immutable types — the bugs would have been impossible by design

- In fact, you should ***never use*** `Date`!
  - Use one of the classes from package java.time: `LocalDateTime`, `Instant`, etc.
  - **All** guarantee in their specifications that **they are immutable**

- This example also illustrates why using mutable objects can actually be bad for performance
  - The simplest solution to this bug, which avoids changing any of the specifications or method signatures, is for `startOfSpring()` to always return a **deep copy** of the groundhog's answer:

    ```
    return new Date(groundhogAnswer.getTime());
    ```

Avoiding Bug by Immutability (2)

- This pattern is **defensive copying**
  - we'll see much more of it when we talk about abstract data types (ADT)

- The defensive copy means `partyPlanning()` can freely edit the returned date without affecting `startOfSpring()`'s cached date

- But defensive copying forces `startOfSpring()` to do extra work and use extra space for every client — even if 99% of the clients never mutate the date it returns
  - We may end up with lots of copies of the first day of spring throughout memory
  - If we used an immutable type instead, then different parts of the program could safely share the same values in memory, so less copying and less memory space is required

- Immutability can be more efficient than mutability, because immutable types *never* need to be defensively copied

Aliasing is what makes mutable types risky (1)

- Actually, using mutable objects is just fine if you are using them entirely locally within a method and with only one reference to the object
  - What led to the problem in the two examples we just looked at was *having multiple references*, also called **aliases**, *for the same mutable object*

- In the List example, the same list is pointed to by both list (in `sum` and `sumOfAbsoluteValues`) and myData (in `main`)
  - One programmer (`sumOfAbsoluteValues`'s) thinks it's ok to modify the list; another programmer (`main`'s) wants the list to stay the same
  - Because of the aliases, `main`'s programmer loses

Aliasing is what makes mutable types risky (2)

- In the `Date` example, there are two variable names that point to the `Date` object: `groundhogAnswer` and `partyDate`
  - These aliases are in completely different parts of the code under the control of different programmers who may have no idea what the other is doing

- Draw snapshot diagrams on paper first, but your real goal should be to develop the snapshot diagram in your head, so you can visualize what's happening in the code

Specifications for Mutating Methods (1)

- At this point it should be clear that when a method performs mutation, it is crucial to include that mutation in the method's spec
  - Now we've seen that even when a particular method doesn't mutate an object, that object's mutability can still be a source of bugs

- Here's an example of a ***mutating*** method:

```
static void sort(List<String> list)
  requires:  nothing
  effects:   puts list in sorted order, i.e. list[i] <= list[j]
             for all 0 <= i < j < list.size()
```

Specifications for Mutating Methods (2)

- And an example of a method that ***does not mutate*** its argument:

```
static List<String> toLowerCase(List<String> list)
  requires: nothing
  effects:   returns a new list t where t[i] == list[i].toLowerCase()
```

- If the effects do **not** explicitly say that an input can be mutated, then let us assume
  mutation of the input is **implicitly disallowed**
  - Virtually all programmers would assume the same thing
  - Surprise mutations lead to terrible bugs

Iterator (1)

- The next mutable object we're going to look at is **an iterator** — an object that steps through a collection of elements and returns the elements one by one
- Iterators are used *under the covers* in Java when you're using *an enhanced for* (... : ...) *loop* to step through a List or array
- This code:

```java
List<String> list = ...;
for (String str : list) {
    System.out.println(str);
}
```

is rewritten by the compiler into something like this:

```java
List<String> list = ...;
Iterator iter = list.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    System.out.println(str);
}
```

Iterator (2)

- An iterator has two methods:
  - **`next()`** returns the next element in the collection
  - **`hasNext()`** tests whether the iterator has reached the end of the collection

- In addition, the `next()` method is a **mutator** method that ***not only*** returns an element ***but also*** **advances** the iterator so that the subsequent call to `next()` will return a different element

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure

the iterator starts somewhere,
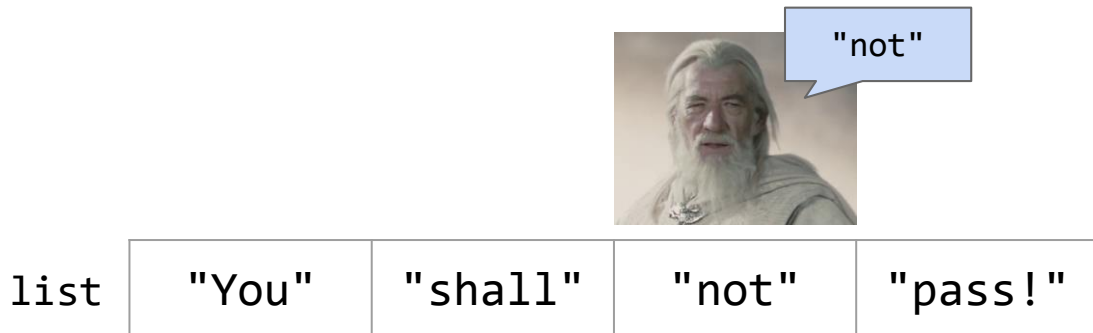usually the beginning of the data structure

list

"You"	"shall"	"not"	"pass!"

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure

true

hasNext() asks, is there an item in where you are currently?

iter.hasNext()

list | "You" | "shall" | "not" | "pass!"

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure

"You"

1) first, `next()` asks the iterator to return the item where it is currently

```
iter.hasNext()
iter.next()
```
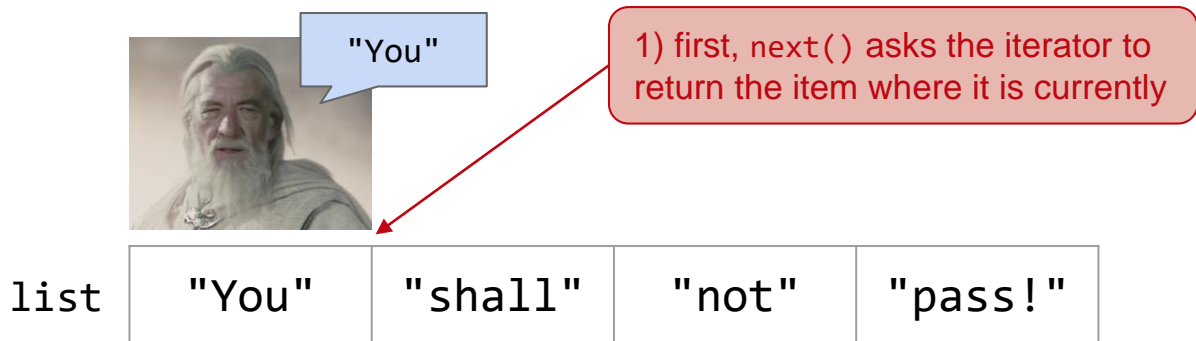
list | "You" | "shall" | "not" | "pass!"

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure

2) then, the iterator moves (mutates) itself to the next place

```
iter.hasNext()
iter.next()
```

list | "You" | "shall" | "not" | "pass!" |

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure


```
iter.hasNext()
iter.next()
iter.hasNext()
```

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure


```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
```

"You"	"shall"	"not"	"pass!"

list

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure


```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
```

list

| "You" | "shall" | "not" | "pass!" |

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure


```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
```

list

| "You" | "shall" | "not" | "pass!" |

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure


```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
```

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure

"You"	"shall"	"not"	"pass!"

list

```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
```

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure

true

| "You" | "shall" | "not" | "pass!" |

list

```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
```

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure

"pass!"

| list | "You" | "shall" | "not" | "pass!" |

```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
```

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure


```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
```

list | "You" | "shall" | "not" | "pass!"

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure

false

```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
```

list | "You" | "shall" | "not" | "pass!"

How an Iterator Work

- Think of an iterator as a powerful wizard that can look inside your data structure


```
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
iter.next()
iter.hasNext()
```

list "Y   !"

MyIterator (1)

- To better understand how an iterator works, here's a simple implementation of **an iterator for `ArrayList<String>`**

- Class `MyIterator` declaration and instance variables:

```java
/**
 * A MyIterator is a mutable object that iterates over
 * the elements of an ArrayList<String>, from first to last.
 * This is just an example to show how an iterator works.
 * In practice, you should use the ArrayList's own iterator
 * object, returned by its iterator() method.
 */
public class MyIterator {

    private final ArrayList<String> list;
    private int index;
```

`list[index]` is the next element that will be returned by `next()`

`index == list.size()` means no more elements to return

MyIterator (2)

- The constructor:

```java
/**
 * Make an iterator.
 * @param list list to iterate over
 */
public MyIterator(ArrayList<String> list) {
    this.list = list;
    this.index = 0;
}
```

start iterating from the beginning

- The hasNext() method:

```java
/**
 * Test whether the iterator has more elements to return.
 * @return true if next() will return another element,
 *         false if all elements have been returned
 */
public boolean hasNext() {
    return index < list.size();
}
```

MyIterator (3)

- The `next()` method:

```java
/**
 * Get the next element of the list.
 * Requires: hasNext() returns true.
 * Modifies: this iterator to advance it to the element
 *           following the returned element.
 * @return next element of the list
 */
public String next() {
    final String element = list.get(index);
    ++index;
    return element;
}
```

MyIterator (4)

- Here's a snapshot diagram showing a typical state for a `MyIterator` object in action:

- Recall that we draw the arrow from list with a double line to indicate it's final

- That means the arrow can't change once it's drawn

- But the `ArrayList` object it points to is mutable — elements can be changed within it — and declaring list as final has no effect on that
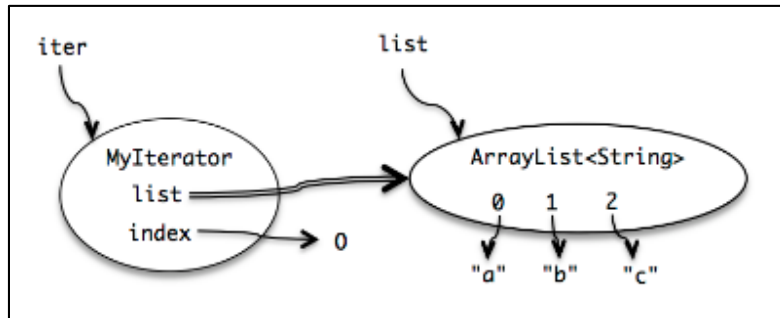
Why Iterator?

- Why do iterators exist?
- There are many kinds of collection data structures (linked lists, maps, hash tables) with different kinds of internal representations
- The iterator concept allows a single uniform way to access them all, so that client code is simpler and the collection implementation can change without changing the client code that iterates over it

- Most modern languages (including Python, C#, and Ruby) use the notion of an iterator
- It's an effective **design pattern** (a well-tested solution to a common design problem)
  - We'll see many other design patterns as we move through the lecture

Mutation Undermines an Iterator (1)

- Let's try using our iterator for a simple job
- Suppose we have a list of subjects represented as strings, like ["a100", "b100", "c200"] We want a method dropCourseA that will delete the "a" subjects/courses from the list, leaving the other subjects behind
- Following good practices, we first write the spec:

```java
/**
 * Drop all subjects that are from a.
 * Modifies subjects list by removing subjects that start with "a".
 * @param subjects list of course subject numbers
 */
public static void dropCourseA(ArrayList<String> subjects)
```

Note that dropCourseA has a clause in its contract that warns the client that its list argument will be mutated

Mutation Undermines an Iterator (2)

- Next, following test-first programming, we devise a testing strategy that partitions the input space and choose test cases to cover that partition:

```
// Testing strategy:
//    subjects.size: 0, 1, n
//    contents: no axxx, one axxx, all axxx
//    position: axxx at start, axxx in middle, axxx at end

// Test cases:
//    [] => []
//    ["b100"] => ["b100"]
//    ["c200", "d200", "e100"] => ["c200", "d200", "e100"]
//    ["b100", "a100", "c100"] => ["b100", "c100"]
//    ["a100", "a200", "a300"] => []
```

Mutation Undermines an Iterator (3)

- Finally, we implement it:

```java
public static void dropCourseA(ArrayList<String> subjects) {
    MyIterator iter = new MyIterator(subjects);
    while (iter.hasNext()) {
        String subject = iter.next();
        if (subject.startsWith("a")) {
            subjects.remove(subject);
        }
    }
}
```

In-Class Quiz 2

- What is the output of `dropCourseA` on input `["a100", "a200", "a300"]` ?

```java
public static void dropCourseA(ArrayList<String> subjects) {
    MyIterator iter = new MyIterator(subjects);
    while (iter.hasNext()) {
        String subject = iter.next();
        if (subject.startsWith("a")) {
            subjects.remove(subject);
        }
    }
}
```

[a200]

Mutation Undermines an Iterator (4)

- The last test case fails:

```
// dropCourseA(["a100", "a200", "a300"])
//    expected [], actual ["a200"]
```

- We got the wrong answer: `dropCourseA` left a course behind in the list!

- Why?
  - Trace through what happens!
  - It will help to use a snapshot diagram showing the `MyIterator` object and the `ArrayList` object and update the diagram while you work through the code

Mutation Undermines an Iterator (5)

- So we've seen that MyIterator can be broken by mutating the underlying list of strings while you're iterating over it
- But this isn't just a bug in our `MyIterator`!
  - The **built-in iterator** in `ArrayList` **suffers from the same problem** and so does the for loop that's syntactic sugar for it
  - The problem just has a different symptom — if you used this code instead:

```java
for (String subject : subjects) {
    if (subject.startsWith("a")) {
        subjects.remove(subject);
    }
}
```

  then you'll get a `ConcurrentModificationException`
  - The built-in iterator detects that you're changing the list under its feet and cries foul (How do you think it does that?)

Mutation Undermines an Iterator (6)

- How can you fix this problem? One way is to use the **remove() method of Iterator** so that the iterator *adjusts its index* appropriately:

```
Iterator iter = subjects.iterator();
while (iter.hasNext()) {
    String subject = iter.next();
    if (subject.startsWith("a")) {
        iter.remove();
    }
}
```

instead of `subjects.remove(subject)`

- This is actually more efficient as well, it turns out, because `iter.remove()` already knows where the element it should remove is, while `subjects.remove()` had to search for it again
- But this doesn't fix the whole problem
  - What if there are *other Iterators* currently active over the same list?
  - They won't all be informed!

Mutable objects can make simple contracts very complex (1)

- This is a fundamental issue with mutable data structures
  - Multiple references to the same mutable object (also called **aliases for the object**) may mean that multiple places in your program — possibly widely separated — are relying on that object to remain consistent

- To put it in terms of specifications, contracts can't be enforced in just one place anymore, e.g. between the client of a class and the implementer of a class

- Contracts involving mutable objects now depend on the good behavior of everyone who has a reference to the mutable object

Mutable objects can make simple contracts very complex (2)

- As a symptom of this non-local contract phenomenon, consider the Java collections classes, which are normally documented with very clear contracts on the client and implementer of a class
  - Try to find where it documents the crucial requirement on the client that we've just discovered — that you **can't** modify a collection while you're iterating over it
  - Who takes responsibility for it? Iterator? List? Collection? Can you find it?

- The need to reason about global properties like this make it much harder to understand, and be confident in the correctness of, programs with mutable data structures
  - We still have to do it — for performance and convenience — but we pay a big cost in bug safety for doing so

Mutable objects reduce changeability (1)

- Mutable objects make the contract between a client and an implementer more complicated and reduce the freedom of the client and implementer to change
- In other words, using objects that are allowed to change makes the code harder to change
- Here's an example to illustrate the point, where the crux of our example will be the specification for this method, which looks up a username in a university database and returns the user's 9-digit identifier:

```
/**
 * @param username username of person to look up.
 * @return the 9-digit university identifier for username.
 * @throws NoSuchUserException if nobody with username is in university database
 */
public static char[] getUnivId(String username) throws NoSuchUserException {
    // ... look up username in university's database and return the 9-digit ID
}
```

Mutable objects reduce changeability (2)

- Suppose we have a client using this method to print out a user's identifier:

```
char[] id = getUnivId("abcde");
System.out.println(id);
```

- Now, **both** the client and the implementer **separately decide to make a change**
- The client is worried about the user's privacy and decides to obscure the first 5 digits of the id:

```
char[] id = getUnivId("abcde");
for (int i = 0; i < 5; ++i) {
    id[i] = '*';
}
System.out.println(id);
```

Mutable objects reduce changeability (3)

- The implementer is worried about the speed and load on the database, so the implementer introduces a cache that remembers usernames that have been looked up:

```java
private static Map<String, char[]> cache = new HashMap<String, char[]>();

public static char[] getUnivId(String username) throws NoSuchUserException {
    // see if it's in the cache already
    if (cache.containsKey(username)) {
        return cache.get(username);
    }
    // ... look up username in university database ...

    // store it in the cache for future lookups
    cache.put(username, id);
    return id;
}
```

Mutable objects reduce changeability (4)

- These two changes have created a subtle bug

- When the client looks up "abcde" and gets back a char array, now both the client and the implementer's cache are pointing to the same char array

- The array is **aliased**

  - That means that the client's obscuring code is actually overwriting the identifier in the cache

  - so future calls to getUnivId("abcde") will not return the full 9-digit number, like "928432033", but instead the obscured version "*****2033"

Sharing a mutable object complicates a contract (1)

- If this contract failure went to software engineering court, it would be contentious. Who's to blame here? Was the client obliged not to modify the object it got back? Was the implementer obliged not to hold on to the object that it returned?

- Here's one (bad) way we could have clarified the spec:

```
public static char[] getUnivId(String username) throws NoSuchUserException
```
requires: nothing
effects: returns an array containing the 9-digit university identifier of username, or throws NoSuchUserException if nobody with username is in university database. *Caller may never modify the returned array*.

Sharing a mutable object complicates a contract (2)

- That is a bad way to do it
- The problem with this approach is that it means the contract has to be in force for the entire rest of the program — it's a lifetime contract!
- The other contracts we wrote were much narrower in scope; you could think about the precondition just before the call was made, and the postcondition just after, and you didn't have to reason about what would happen for the rest of time
- Here's a spec with a similar problem:

```
public static char[] getUnivId(String username) throws NoSuchUserException
  requires: nothing
  effects: returns a new array containing the 9-digit university identifier of username,
          or throws NoSuchUserException if nobody with username is in university
          database
```

Sharing a mutable object complicates a contract (3)

- That doesn't entirely fix the problem either

- That spec at least says that the array has to be fresh
- But does it keep the implementer from holding an alias to that new array?
  Does it keep the implementer from changing that array or reusing it in the future
  for something else?

Sharing a mutable object complicates a contract (4)

- Here's a much better spec:

```
public static String getUnivId(String username) throws NoSuchUserException
 requires: nothing
 effects: returns the 9-digit university identifier of username, or throws
         NoSuchUserException if nobody with username is in university database
```

- The immutable String return value provides a guarantee that the client and the implementer will never step on each other the way they could with char arrays
- It doesn't depend on a programmer reading the spec comment carefully
  - `String` is immutable
  - Not only that, but this approach (unlike the previous one) gives the implementer the freedom to introduce a cache — a performance improvement

Useful Immutable Types (1)

Since immutable types avoid so many pitfalls, let us enumerate some commonly-used immutable types in the Java API:

- **The primitive types** and **primitive wrappers** are all immutable
  - If you need to compute with large numbers, `BigInteger` and `BigDecimal` are immutable

- Don't use mutable Dates but instead use the appropriate immutable type from **java.time** based on the granularity of timekeeping you need

Useful Immutable Types (2)

The usual implementations of Java's collections types — `List`, `Set`, `Map` — are all mutable: `ArrayList`, `HashMap`, etc.

- The `Collections` utility class has methods for obtaining unmodifiable views of these mutable collections:
  - **`Collections.unmodifiableList`**
  - **`Collections.unmodifiableSet`**
  - **`Collections.unmodifiableMap`**

  - You can think of the unmodifiable view as ***a wrapper*** around the underlying list/set/map
  - A client who has a reference to the wrapper and tries to perform mutations — add, remove, put, etc. — will trigger an `UnsupportedOperationException`

Useful Immutable Types (3)

- Before we pass a mutable collection to another part of our program, we can wrap it in an unmodifiable wrapper
- We should be careful at that point to *forget our reference to the mutable collection*, lest we accidentally mutate it (one way to do that is to let it go out of scope)
- Just as a mutable object behind a final reference can be mutated, the mutable collection inside an unmodifiable wrapper ***can still be modified*** by someone with a reference to it, defeating the wrapper

- `Collections` also provides methods for obtaining immutable empty collections: **`Collections.emptyList`**, etc.
  - Nothing's worse than discovering your definitely very empty list is suddenly definitely not empty!

Part 3: Disjoint Sets and Dynamic Connectivity

- For the third part of the lecture, let us take a break from the *boring* example of List, and explore this new data structure

- We will build a data structure called the **Disjoint Sets**, also known as **Union Find**
  - to solve the **Dynamic Connectivity** problem

- Furthermore, we will see:
  - How a data structure design can evolve from *basic* to *sophisticated*
  - How our choice of *underlying abstraction* can affect the running time
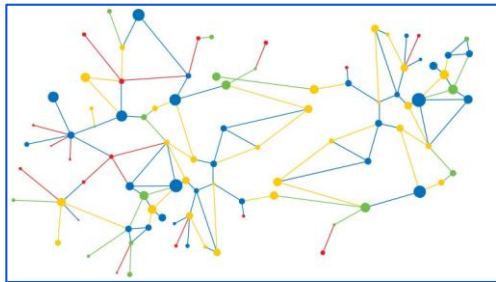
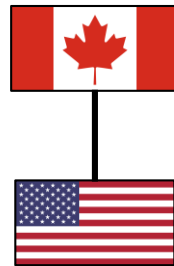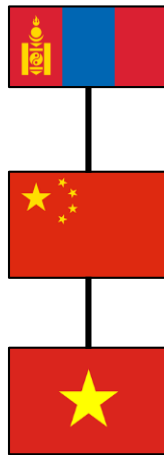# Disjoint Sets

- 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

        Note that connections can be transitive, i.e. they *don't need* to be direct
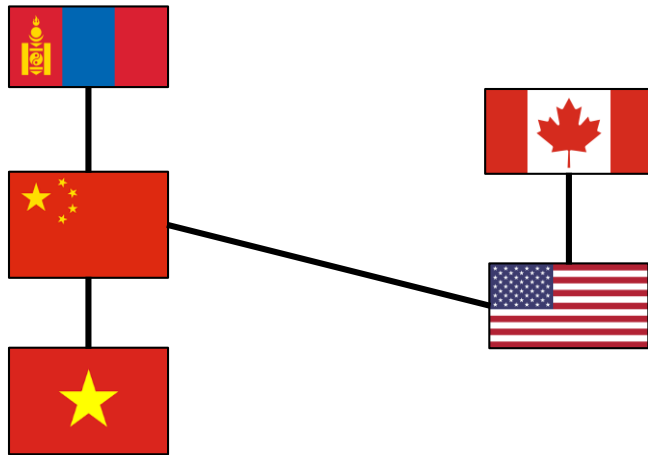
Disjoint Sets

- 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

        Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- connect(China, Vietnam)

Disjoint Sets

- 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

    Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- connect(China, Vietnam)
- connect(China, Mongolia)

Disjoint Sets

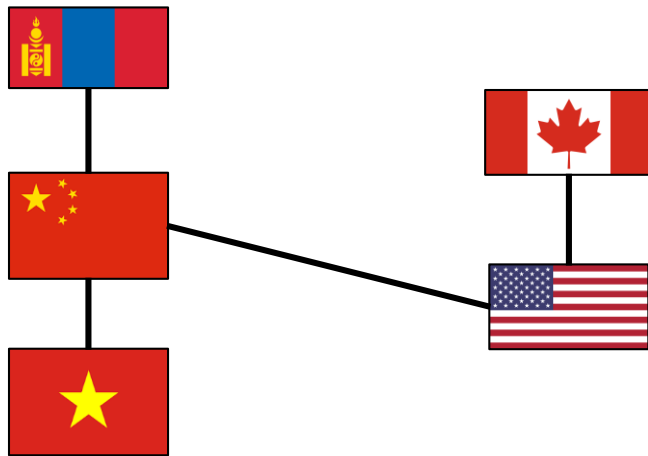- Disjoint Sets data structure has two operations:
    - **connect**(x, y)   :   Connects x and y
    - **isConnected**(x, y)   :   Returns `true` if x and y are connected
      Note that connections can be transitive, i.e. they *don't need* to be direct

Example:
- connect(China, Vietnam)
- connect(China, Mongolia)
- isConnected(Vietnam, Mongolia) → ?

Disjoint Sets

- 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
    Note that connections can be transitive, i.e. they *don't need* to be direct

Example:
- connect(China, Vietnam)
- connect(China, Mongolia)
- isConnected(Vietnam, Mongolia) → `true`
- connect(USA, Canada)

Disjoint Sets

- 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
    Note that connections can be transitive, i.e. they *don't need* to be direct

Example:
- connect(China, Vietnam)
- connect(China, Mongolia)
- isConnected(Vietnam, Mongolia) → `true`
- connect(USA, Canada)
- isConnected(Canada, Mongolia) → ?
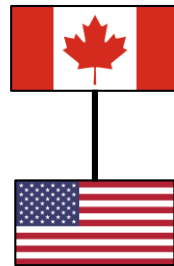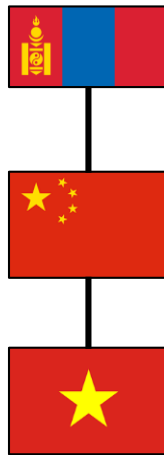
Disjoint Sets

- 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

    Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- connect(China, Vietnam)
- connect(China, Mongolia)
- isConnected(Vietnam, Mongolia) → `true`
- connect(USA, Canada)
- isConnected(Canada, Mongolia) → `false`
- connect(China, USA)

Disjoint Sets

- 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
    Note that connections can be transitive, i.e. they *don't need* to be direct

Example:
- connect(China, Vietnam)
- connect(China, Mongolia)
- isConnected(Vietnam, Mongolia) → `true`
- connect(USA, Canada)
- isConnected(Canada, Mongolia) → `false`
- connect(China, USA)
- isConnected(Canada, Mongolia) → ?

Disjoint Sets

- 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
    Note that connections can be transitive, i.e. they *don't need* to be direct

Example:
- connect(China, Vietnam)
- connect(China, Mongolia)
- isConnected(Vietnam, Mongolia) → `true`
- connect(USA, Canada)
- isConnected(Canada, Mongolia) → `false`
- connect(China, USA)
- isConnected(Canada, Mongolia) → `true`

the answer can change!

Disjoint Sets

- 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
      Note that connections can be transitive, i.e. they *don't need* to be direct

- This data structure is useful for, for example:
    - Percolation theory in Computational Chemistry
    - Kruskal Algorithm in computing MST
    - and many more

Disjoint Sets on Integers

- Let's start simple, we let:
    - the items are integers instead of arbitrary data, and we
    - declare the number of items in advance, where everything is *disconnected*

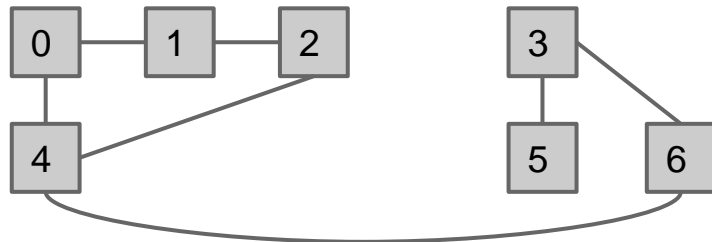# Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is *disconnected*

`ds = DisjointSets(7)`

| 0 | 1 | 2 | 3 |

| 4 | | 5 | 6 |

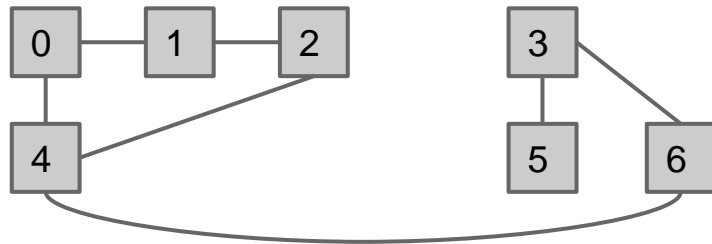# Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
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)
```


these are called
connected components
there are 3 connected
components currently

Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
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

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
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)
```
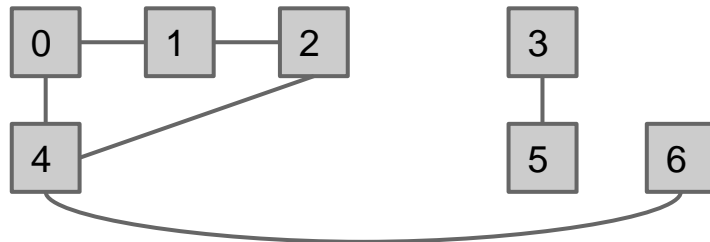
Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
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) → ?
```

Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
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
```

Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

- We want to implement two methods: (and a constructor)

```
/** Connects two items p and q. */
void connect(int p, int q) {
    ...
}

/** Decides if two items p and q are connected. */
boolean isConnected(int p, int q) {
    ...
}
```

also called **union**

also called **find**

  - where the **number of items** N and the **number of operations** M can be *very large*
  - and the two methods are called in *arbitrary* order

Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

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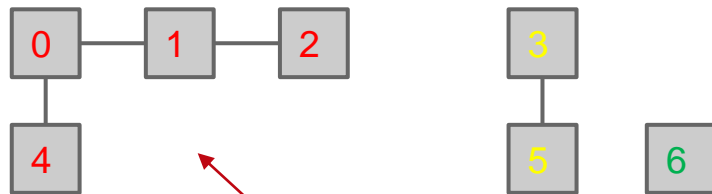

the question is: how do we store these connected components?

Naive Approach

- Let us start to think how we store the connectivity information
- A naive approach:
  - Connect: Record every single connecting line in some data structure
  - isConnected: Iterate over the lines to see if one item can be reached from the other
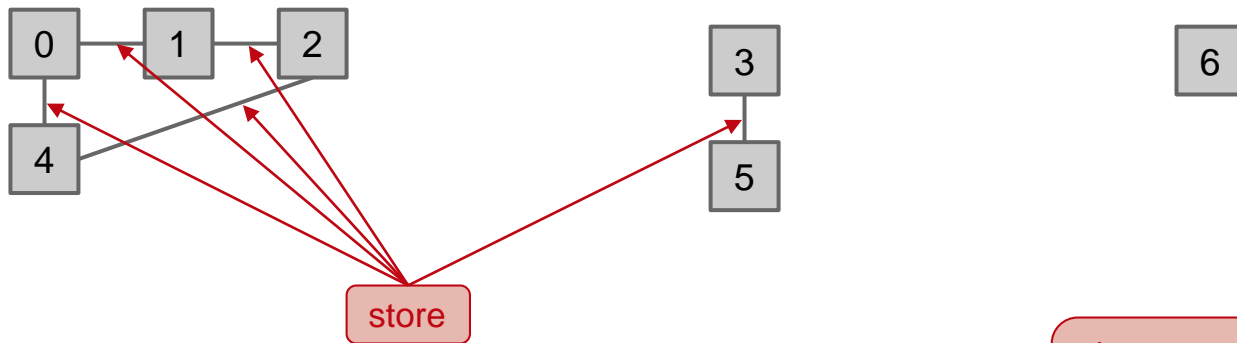
since connectivity is transitive, this idea is redundant
what should we store instead?

Connected Components / Sets

- A better idea would be to record **the sets** that each item belongs to
  - each set represents a connected component

```
ds = DisjointSets(7)
```
                                    {0}, {1}, {2}, {3}, {4}, {5}, {6}

Connected Components / Sets

- A better idea would be to record **the sets** that each item belongs to
    - each set represents a connected component

```
ds = DisjointSets(7)     {0}, {1}, {2}, {3}, {4}, {5}, {6}
ds.connect(0, 1)
```

merge / union

Connected Components / Sets

- A better idea would be to record **the sets** that each item belongs to
    - each set represents a connected component

```
ds = DisjointSets(7)      {0}, {1}, {2}, {3}, {4}, {5}, {6}
ds.connect(0, 1)          {0, 1}, {2}, {3}, {4}, {5}, {6}
ds.connect(1, 2)
```

merge / union

Connected Components / Sets

- A better idea would be to record **the sets** that each item belongs to
  - each set represents a connected component

```
ds = DisjointSets(7)        {0}, {1}, {2}, {3}, {4}, {5}, {6}
ds.connect(0, 1)            {0, 1}, {2}, {3}, {4}, {5}, {6}
ds.connect(1, 2)            {0, 1, 2}, {3}, {4}, {5}, {6}
ds.connect(0, 4)            {0, 1, 2, 4}, {3}, {5}, {6}
ds.connect(3, 5)            {0, 1, 2, 4}, {3, 5}, {6}
ds.isConnected(2, 4)
ds.isConnected(3, 0)
ds.connect(4, 2)
```
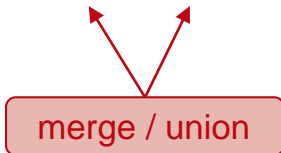
no change

Connected Components / Sets

- A better idea would be to record **the sets** that each item belongs to
  - each set represents a connected component

```
ds = DisjointSets(7)        {0}, {1}, {2}, {3}, {4}, {5}, {6}
ds.connect(0, 1)            {0, 1}, {2}, {3}, {4}, {5}, {6}
ds.connect(1, 2)            {0, 1, 2}, {3}, {4}, {5}, {6}
ds.connect(0, 4)            {0, 1, 2, 4}, {3}, {5}, {6}
ds.connect(3, 5)            {0, 1, 2, 4}, {3, 5}, {6}
ds.isConnected(2, 4)
ds.isConnected(3, 0)
ds.connect(4, 2)            {0, 1, 2, 4}, {3, 5}, {6}
ds.connect(4, 6)            {0, 1, 2, 4, 6}, {3, 5}
ds.connect(3, 6)
```

merge / union

Connected Components / Sets

- A better idea would be to record the sets that each item belongs to
    - each set represents a connected component
    - if two items are **in the same set**, they are connected

```
ds = DisjointSets(7)      {0}, {1}, {2}, {3}, {4}, {5}, {6}
ds.connect(0, 1)          {0, 1}, {2}, {3}, {4}, {5}, {6}
ds.connect(1, 2)          {0, 1, 2}, {3}, {4}, {5}, {6}
ds.connect(0, 4)          {0, 1, 2, 4}, {3}, {5}, {6}
ds.connect(3, 5)          {0, 1, 2, 4}, {3, 5}, {6}
ds.isConnected(2, 4)
ds.isConnected(3, 0)
ds.connect(4, 2)          {0, 1, 2, 4}, {3, 5}, {6}
ds.connect(4, 6)          {0, 1, 2, 4, 6}, {3, 5}
ds.connect(3, 6)          {0, 1, 2, 3, 4, 5, 6}
ds.isConnected(3, 0)
```

asking: are 3 and 6 in the same set ?

Selecting Underlying Data Structure

- So, what data structure should we should to track the set membership?
  - in other words, how do we keep track which item in which set?

Before `connect(2, 3)` operation:
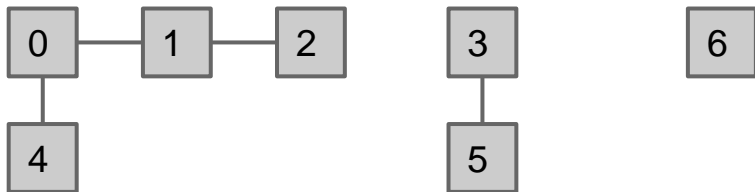
After `connect(2, 3)` operation:

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

5 in here before
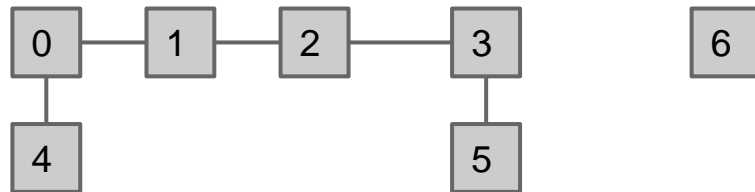
`{0, 1, 2, 4, 3, 5},`   `{6}`

5 in here after

Idea 1: List of Sets

- Using `List<Set<Integer>>` from Java Library

  - A very intuitive idea

  - However, imagine in the beginning, we have this List of Sets of integers:
    `[{0}, {1}, {2}, {3}, {4}, {5}, {6}]`

  - We have to iterate through all the sets to find an item

    - 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 is linear or O(N)

    - similarly for connect

Performance Summary

- Let's keep track our progress so far

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

List of Sets
Disjoint Sets

Initialize
N items

need to
find items

Idea 2: Array of Integers (1)

- Using an array of integers, where the ith entry gives **the set number/id** of item i


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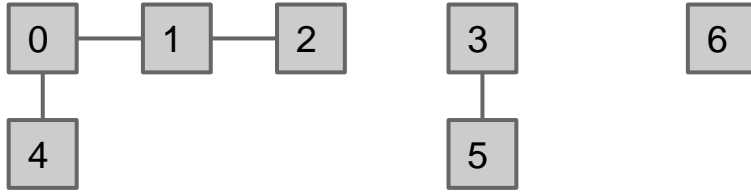
int[] id

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

item 0, 1, 2, and 4 belong to set number 4

the set number 4 does not matter, could be any int

Idea 2: Array of Integers (2)

- Using an array of integers, where the ith entry gives the set number/id of item i
  - **isConnected**(p, q) simply checks whether id[p] and id[q] are *the same*


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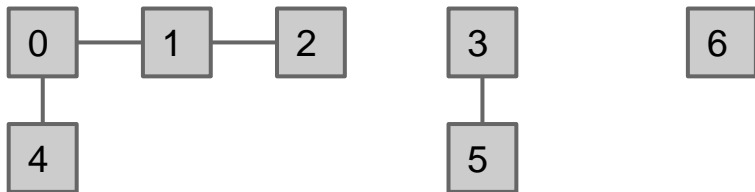
```
int[] id   4  4  4  5  4  5  6
           0  1  2  3  4  5  6
```

items 2 and 4 are connected

items 5 and 6 are not connected

Idea 2: Array of Integers (3)

- Using an array of integers, where the ith entry gives the set number/id of item i
  - **connect**(p, q) simply changes **all** entries that equal id[p] to id[q]

Before connect(2, 3) operation:
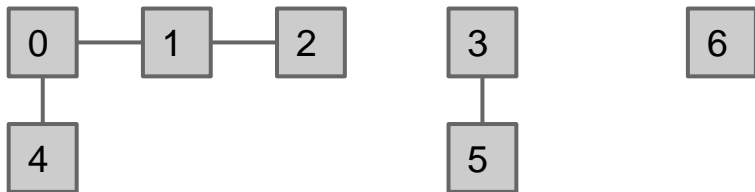
After connect(2, 3) operation:

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

int[] id

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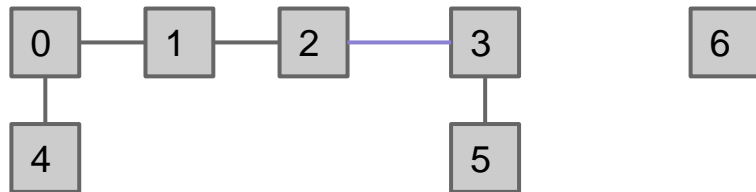
{0, 1, 2, 4, 3, 5},   {6}

int[] id

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

the set number of 0, 1, 2, 4 changes from 4 to 5, which is the set number of 3

Idea 2: Array of Integers (4)

- This implementation of Disjoint Sets data structure is called `QuickFindDS`

```java
public class QuickFindDS {

    private int[] id;

    /** Construct a new disjoint sets data structure of N items. */
    public QuickFindDS(int N){
        id = new int[N];
        for (int i = 0; i < N; i++){
            id[i] = i;
        }
    }

    /** Connects two items p and 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;
            }
        }
    }

    /** Decides if two items p and q are connected. */
    public boolean isConnected(int p, int q){
        return (id[p] == id[q]);
    }
}
```

because the **isConnected**/**find** operation is *very fast*, just two array accesses
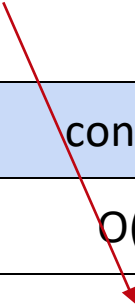
initially, each item belong to each set

however, as seen in the for loop here, **connect**/**union** is *slow*, taking O(N) time

Performance Summary

- Our second attempt made some progress with isConnected,
  - but connecting two items taking linear time each time is unacceptable!

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

Next Idea

- Because we may need to change *many values*, `connect` of `QuickFindDS` is slow
  - we still want to represent the connected components as an array of integers
  - however, values will be chosen so that connect is fast

- How could we change our connected components representation so that combining two sets requires changing only **one value**?

can you solve this? any ideas?

Idea 3: Array of Parents

- How could we change our set representation so that combining two sets requires changing only **one value**?
  - Assign each item a parent (instead of an id), and set the roots to have value -1
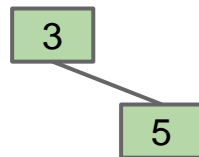 - Resulting in a forest (a set of trees)

parent

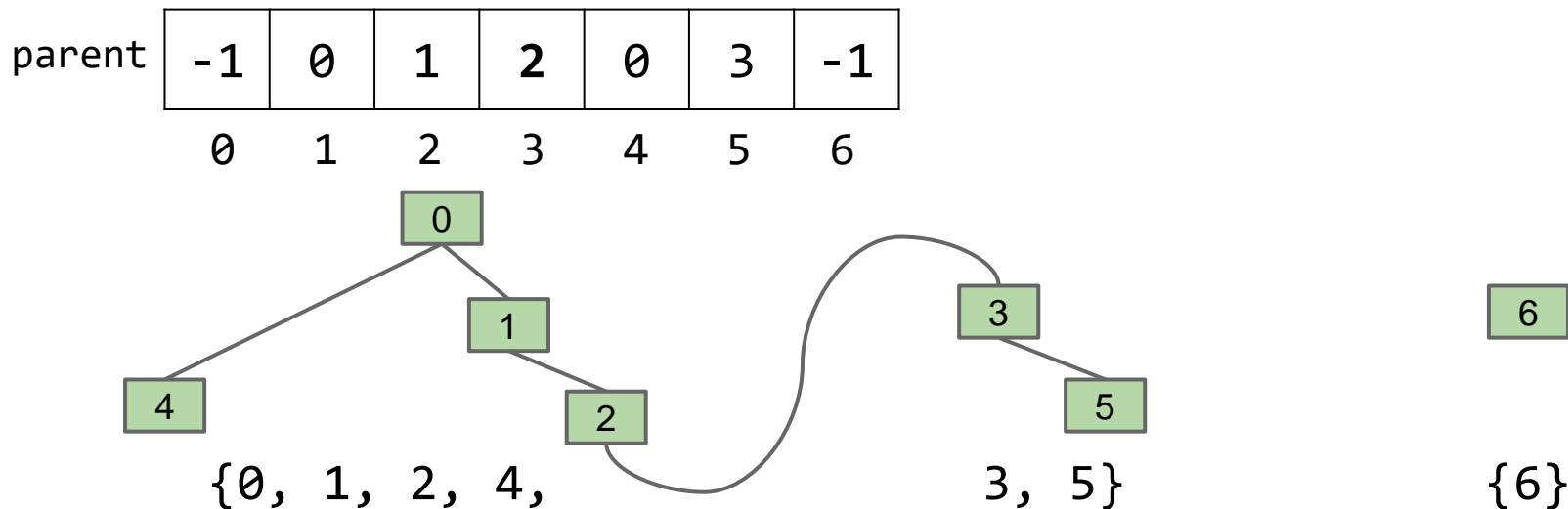| -1	0	1	-1	0	3	-1
0	1	2	3	4	5	6

the parent of 5 is 3
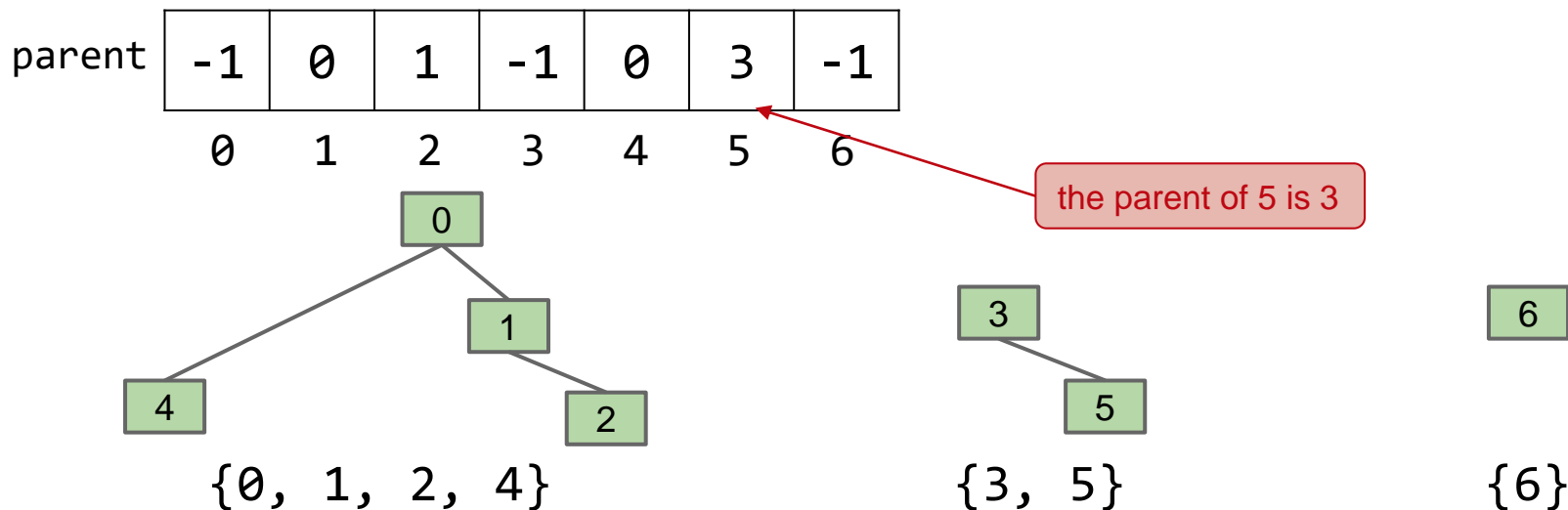
{0, 1, 2, 4}   {3, 5}   {6}

Improving Connect (1)

- How should we change the parent array if we call **connect(5, 2)**?
  - Can we just set parent[5] to 2?

parent

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

```
        0
       / \
      4   1
           \
            2
   {0, 1, 2, 4}
```

```
    3
     \
      5
  {3, 5}
```

```
    6
  {6}
```
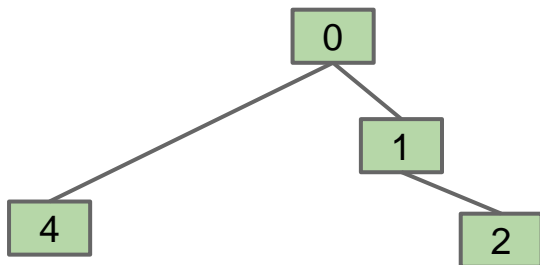
Improving Connect (2)

- How should we change the parent array if we call **connect(5, 2)**?
  - Can we just set parent[5] to 2? No, 3, the root of 5, will be disconnected
  - One possible idea is to set parent[3] to 2, but this results in taller tree, longer path to root
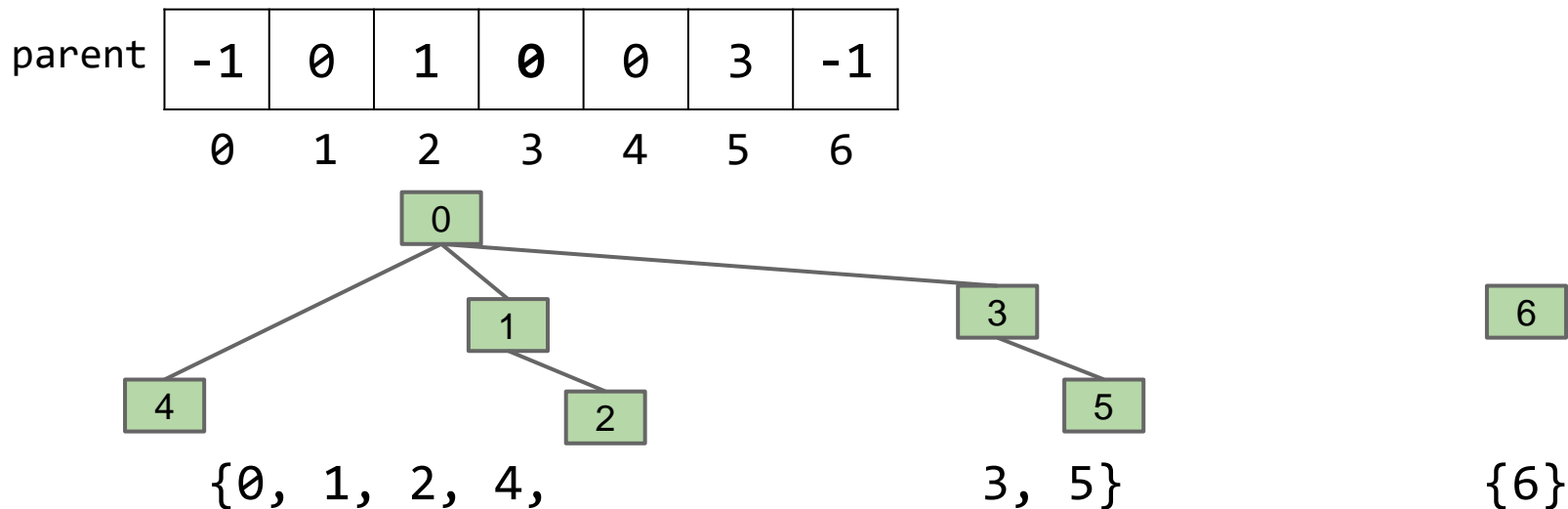
parent

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

```
        0
       / \
      4   1
           \
            2

            3
           / \
          5

            6
```

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

Improving Connect (3)

- How should we change the parent array if we call **connect(5, 2)**?
  - Another idea is to do the following:

    find the root 5, find the root of 2,

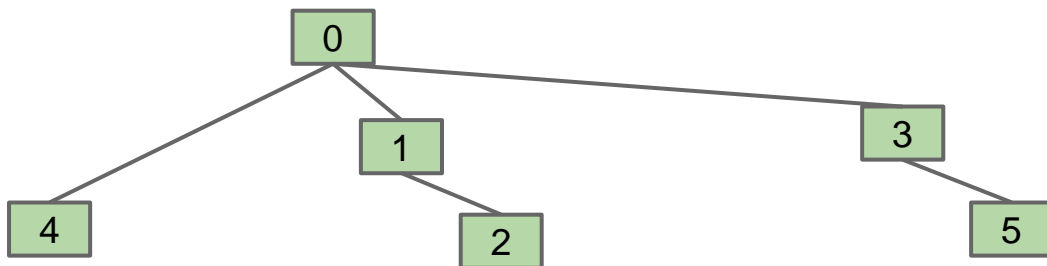   and then set the value of the root of 5 to be the root of 2

parent

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

```
        0
      / | \
     /  |  \
    4   1   3
        |   |
        2   5
```

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

Improving Connect (3)

- How should we change the parent array if we call **connect(5, 2)**?
  - Another idea is to do the following:

    find the root 5, find the root of 2,

    and then set the value of the root of 5 to be the root of 2

parent

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

seems like a good idea…
can things go wrong?

```
        0
      / |   \
     4  1    3
        |    |
        2    5
```

0

1

3

6

4

2

5

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

The Worst-Case

- Consider the following series of operations:
  - ...
  - connect(4, 3)
  - connect(3, 2)
  - connect(2, 1)
  - connect(1, 0)

- For N items, what is the worst case runtime of:
  - connect(p, q)?
  - isConnected(p, q)?
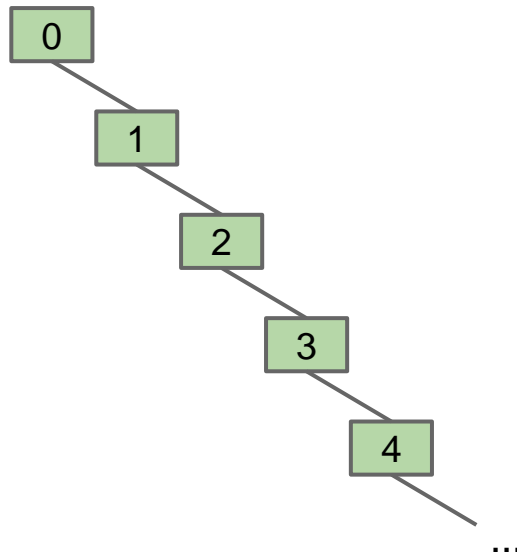
The Worst-Case

- Consider the following series of operations:
  - ...
  - connect(4, 3)
  - connect(3, 2)
  - connect(2, 1)
  - connect(1, 0)

- For N items, what is the worst case runtime of:
  - connect(p, q)?   O(N)
  - isConnected(p, q)?   O(N)
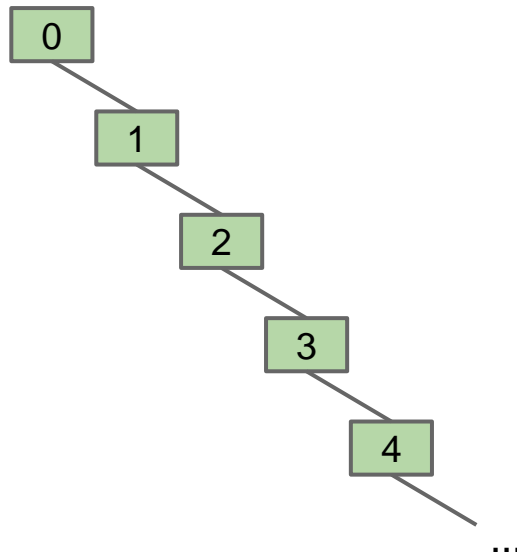
Idea 3: Array of Parents

- This implementation of Disjoint Sets data structure is called **QuickUnionDS**

```java
public class QuickUnionDS {

    private int[] parent;

    /** Construct a new disjoint sets data structure of N elements. */
    public QuickUnionDS(int N) {
        parent = new int[N];
        for (int i = 0; i < N; i++) {
            parent[i] = -1;
        }
    }

    // Finding the root.
    private int find(int p) {
        while (parent[p] >= 0) {
            p = parent[p];
        }
        return p;
    }

    /** Connects two elements p and q. */
    public void connect(int p, int q) {
        int i = find(p);
        int j = find(q);
        parent[i] = j;
    }

    /** Decides if two elements p and q are connected. */
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }
}
```

finding the root costs O(N) in worst-case

causing connect and isConnected to cost O(N) worst-case running time as well

Performance Summary

- Our third attempt may perform well in average,
  but in the worst case, we actually get a worse performance!

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

Next Idea

- The defect of `QuickUnionDS` is that the trees can get tall
  - resulting in potentially even worse performance than `QuickFindDS` if tree is imbalanced

- What should we do to keep our trees balanced?

can you solve this? any ideas?

In-Class Quiz 3

- Suppose you are trying to connect(5, 2) — which one is better :
    a. set 5's root to be a child of 2's root
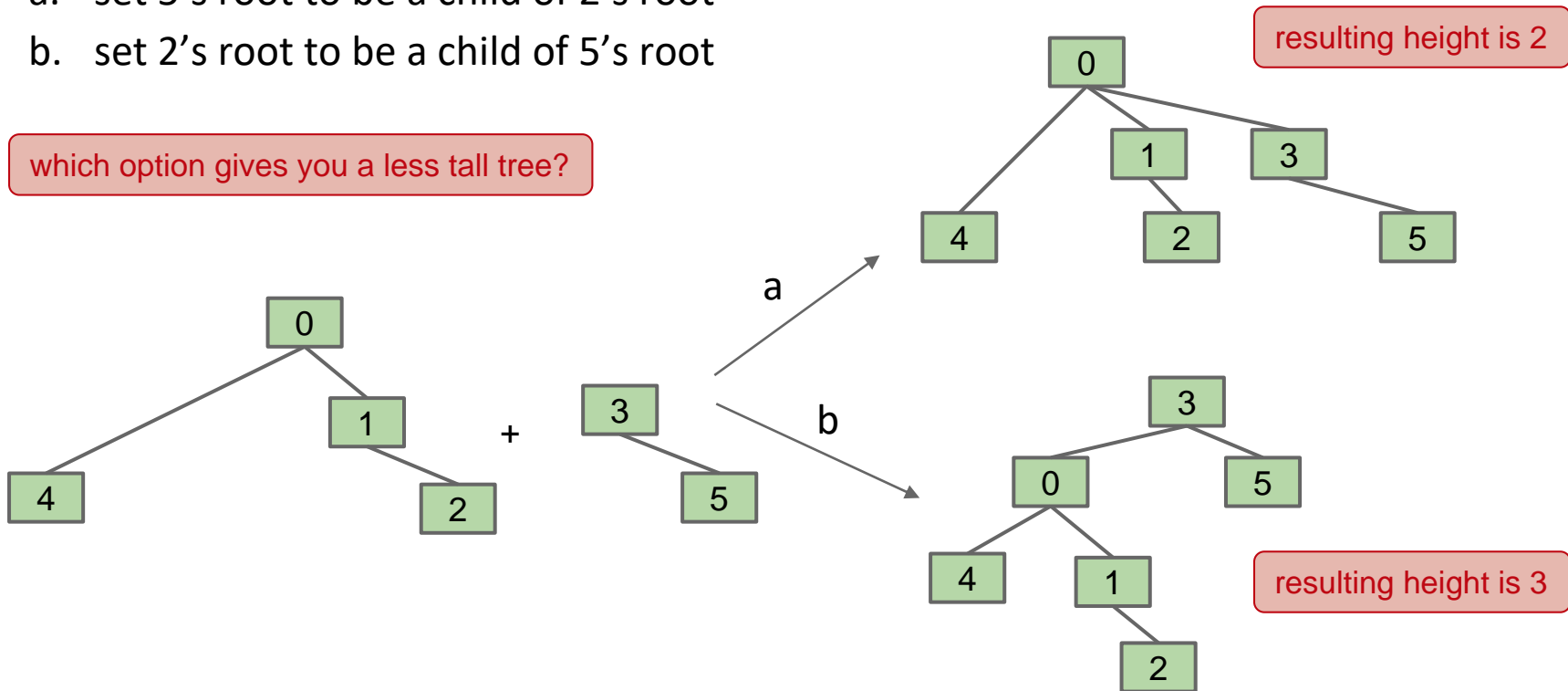   b. set 2's root to be a child of 5's root
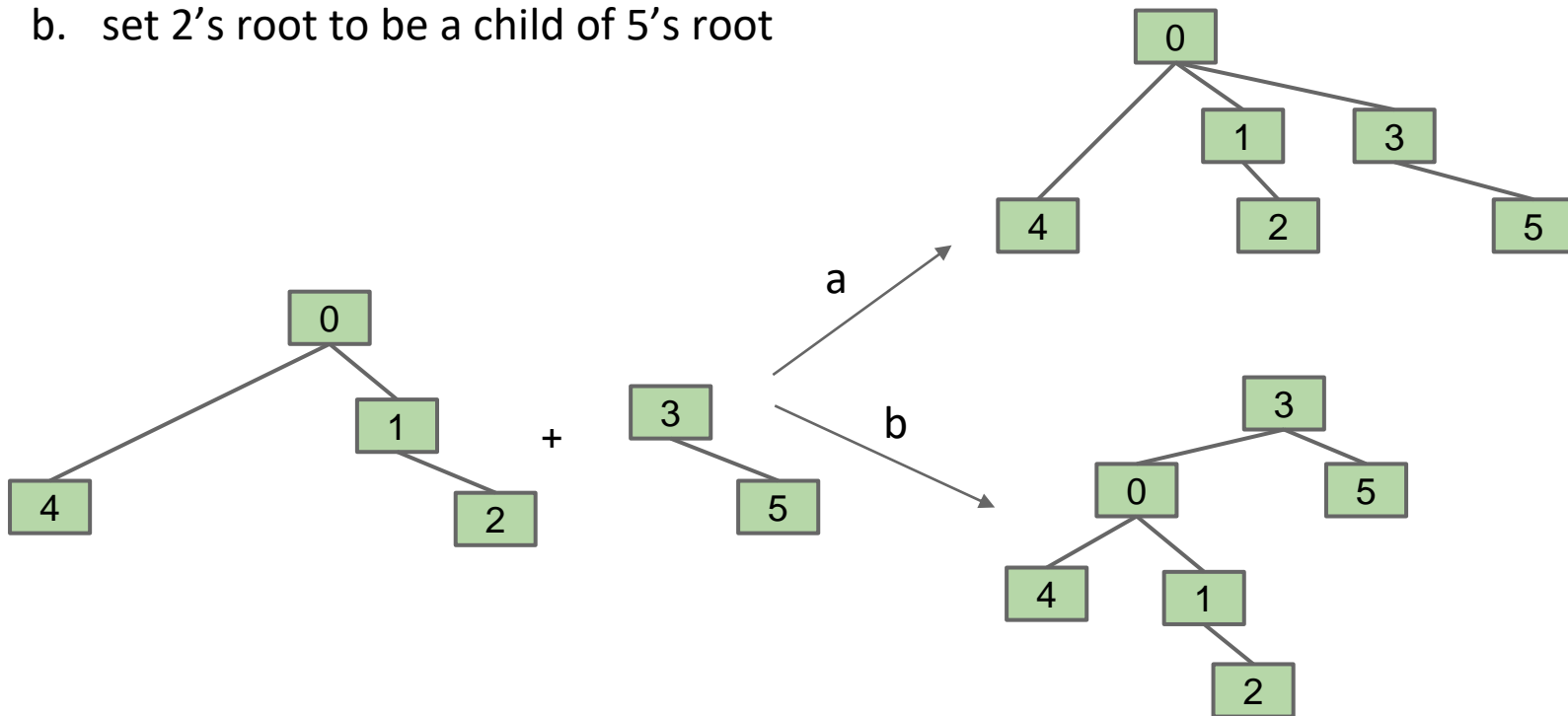
An Observation

- Suppose you are trying to connect(5, 2) — which one is better :
  a. set 5's root to be a child of 2's root
  b. set 2's root to be a child of 5's root

resulting height is 2

which option gives you a less tall tree?

resulting height is 3

Idea 4: Link Smaller to Larger

- Rather than always links the root of *first* tree to the root of *second* tree during connect as in QuickUnionDS:
    - keep track of the **tree size** (its number of elements)
    - link the the root of *smaller* tree to the root of *larger* tree

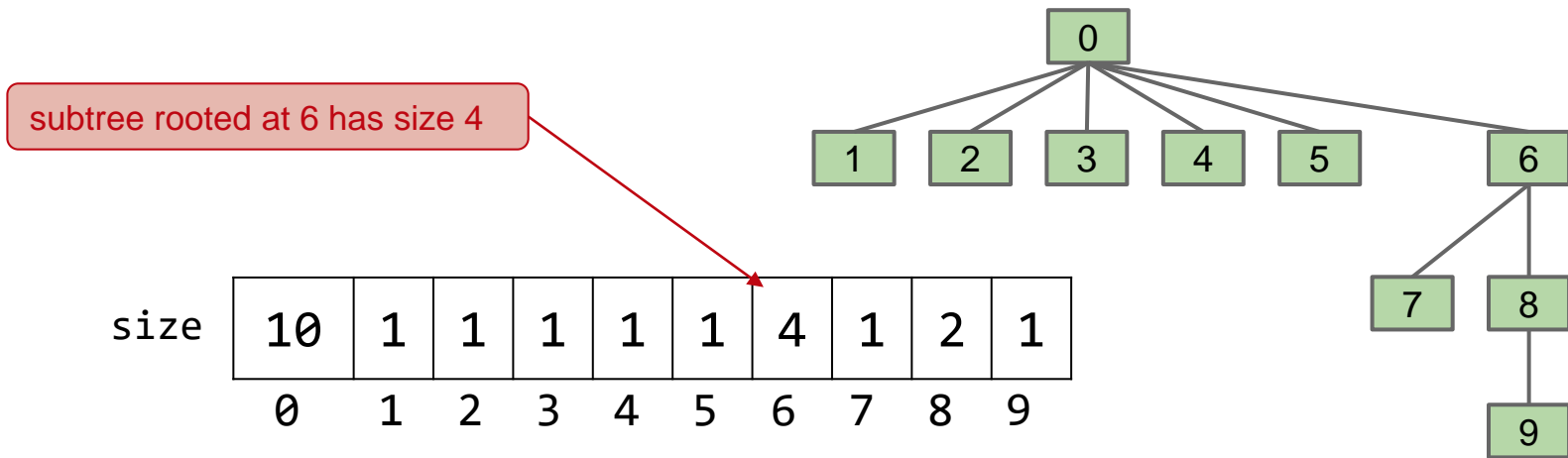# Example Linking Smaller to Larger (1)

- Rather than always links the root of *first* tree to the root of *second* tree during connect as in QuickUnionDS:
    - keep track of the **tree size** (its number of elements)
    - link the the root of ***smaller*** tree to the root of ***larger*** tree

- Example: what will connect(3, 8) change ?
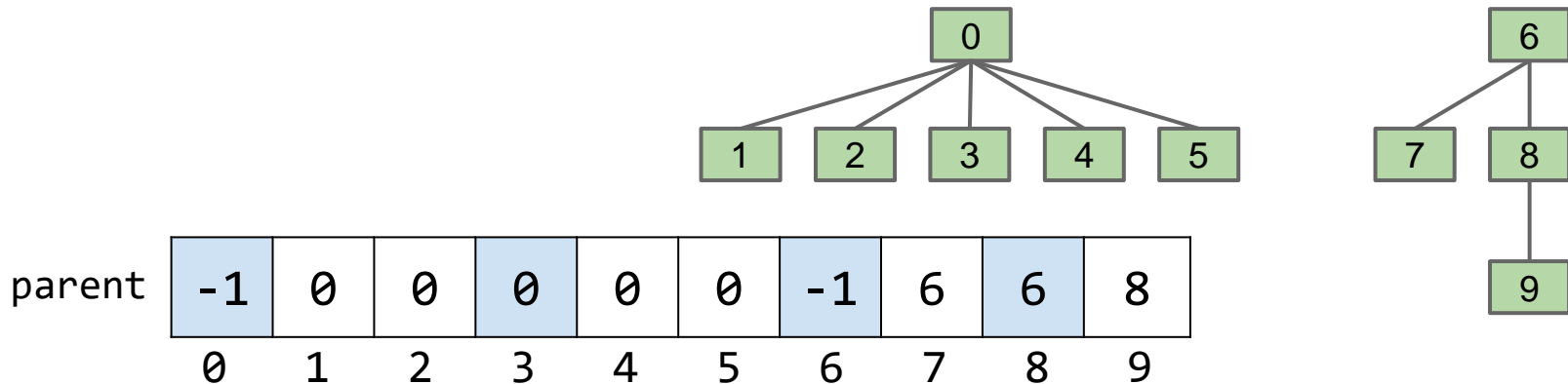
Example Linking Smaller to Larger (2)

- Rather than always links the root of *first* tree to the root of *second* tree during connect as in QuickUnionDS:
  - keep track of the **tree size** (its number of elements)
  - link the the root of *smaller* tree to the root of *larger* tree

- Example: what will connect(3, 8) change ?
  - the tree of 8 has smaller size,
    so we change the root of 6

parent	-1	0	0	0	0	0	0	6	6	8
	0	1	2	3	4	5	6	7	8	9
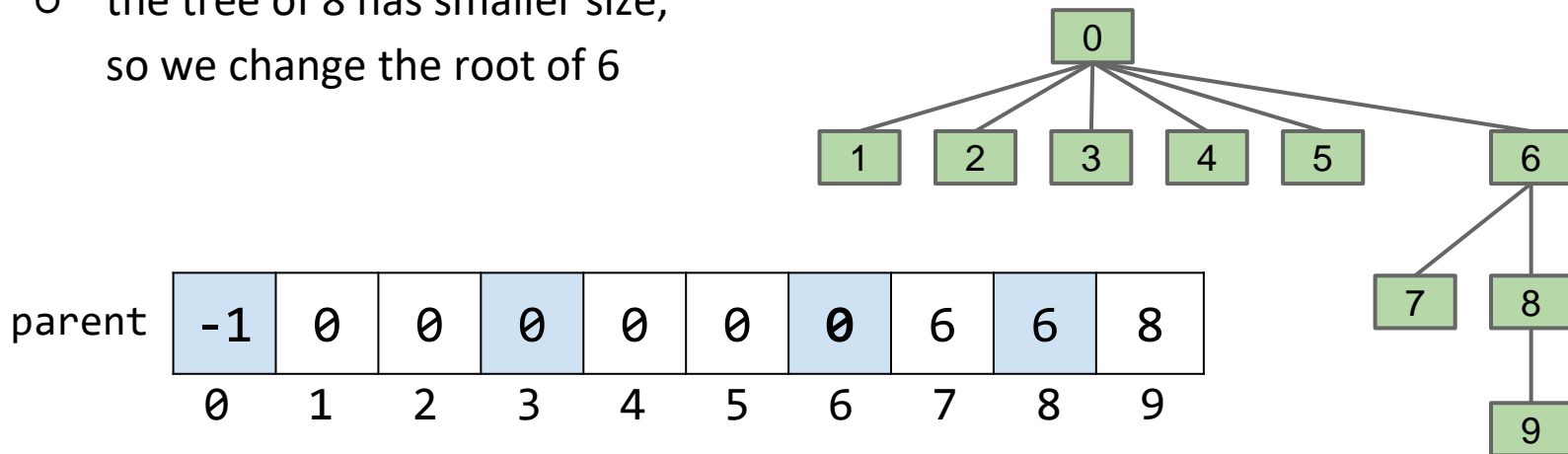
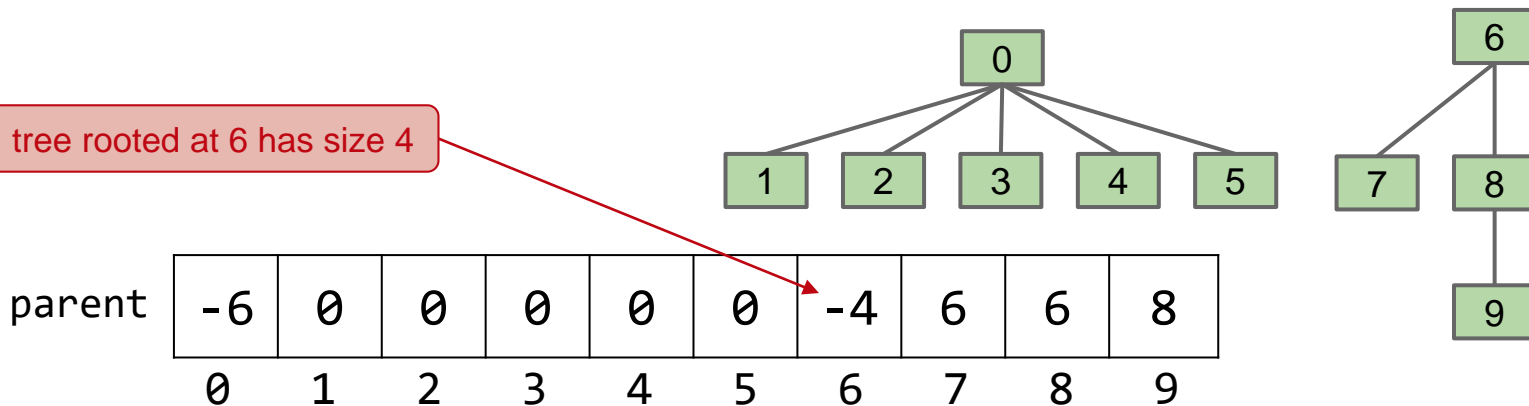# Idea 4: Weighted Quick Union (1)

- How do we implement this? What is the underlying data structure(s)?
  - we still use array of parents as in QuickUnionDS
  - to keep track the sizes:
    1. we could have another array size[] to store **the size of the subtree rooted at index i**

subtree rooted at 6 has size 4

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

Idea 4: Weighted Quick Union (2)

- How do we implement this? What is the underlying data structure(s)?
  - we still use array of parents as in QuickUnionDS
  - to keep track the sizes:
    1. we could have another array size[] to store the size of the subtree rooted at index i
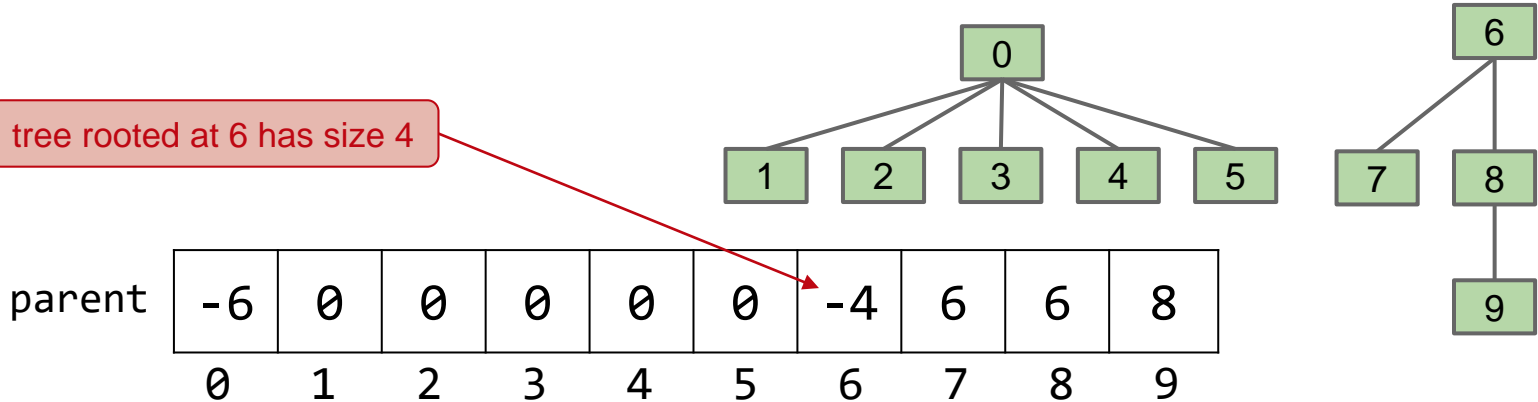   2. instead of -1, we store **negative size** of the trees in the root

tree rooted at 6 has size 4

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

Idea 4: Weighted Quick Union (2)

- How do we implement this? What is the underlying data structure(s)?
  - we still use array of parents as in QuickUnionDS
  - to keep track the sizes:
    1. we could have another array size[] to store the size of the subtree rooted at index i
    2. instead of -1, we store **negative size** of the trees in the root

tree rooted at 6 has size 4

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

let's use Idea 4.2. in our **Lab 9** !

Relationship between Weight and Height (1)

- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)
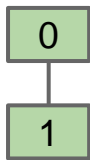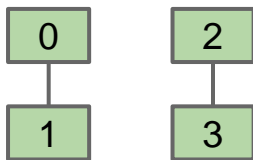
N	H
1	0

when N = 1, the tallest possible tree has height H = 0

0

Relationship between Weight and Height (2)

- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)
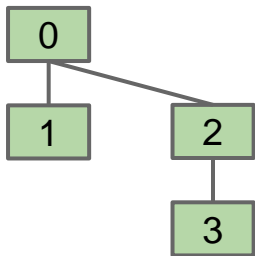
N	H
1	0
2	1

```
0
|
1
```

when N = 2, the tallest possible tree has height H = 1

Relationship between Weight and Height (3)

- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)

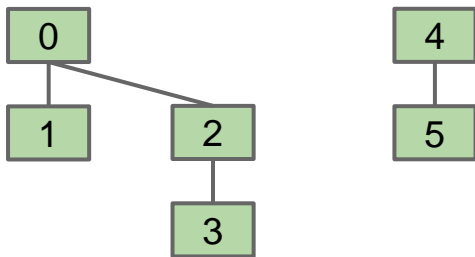| N	H
1	0
2	1

```
0       2
|       |
1       3
```

we will connect/union those two trees next

Relationship between Weight and Height (4)

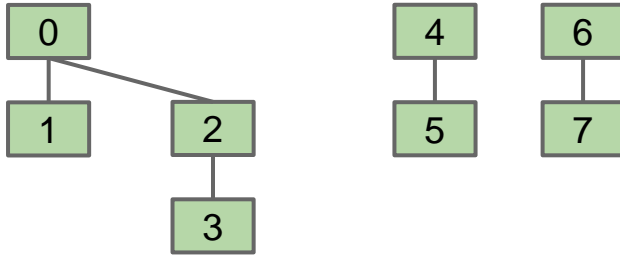- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)

N	H
1	0
2	1
4	2

when N = 4, the tallest possible tree has height H = 2

- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)

N	H
1	0
2	1
4	2

Relationship between Weight and Height (6)

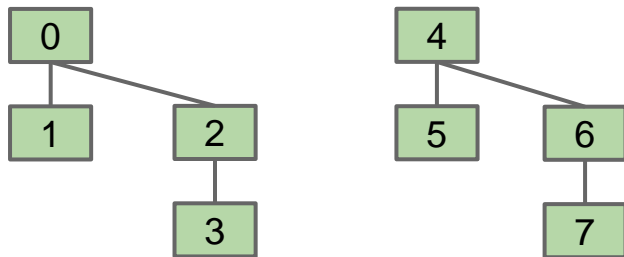- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)

N	H
1	0
2	1
4	2

Relationship between Weight and Height (7)

- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)
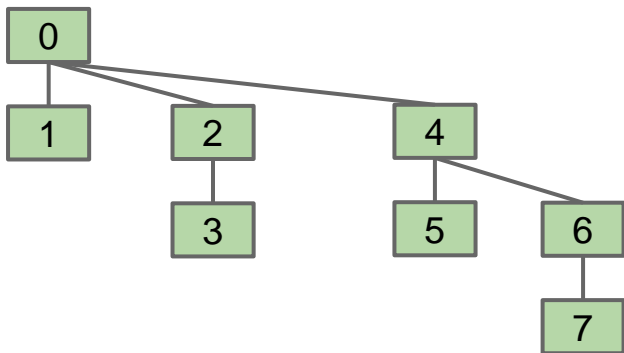
N	H
1	0
2	1
4	2

connect those two to get a height-3 tree

Relationship between Weight and Height (8)

- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)

N	H
1	0
2	1
4	2
8	3

when N = 8, the tallest possible tree has height H = 3

Relationship between Weight and Height (9)

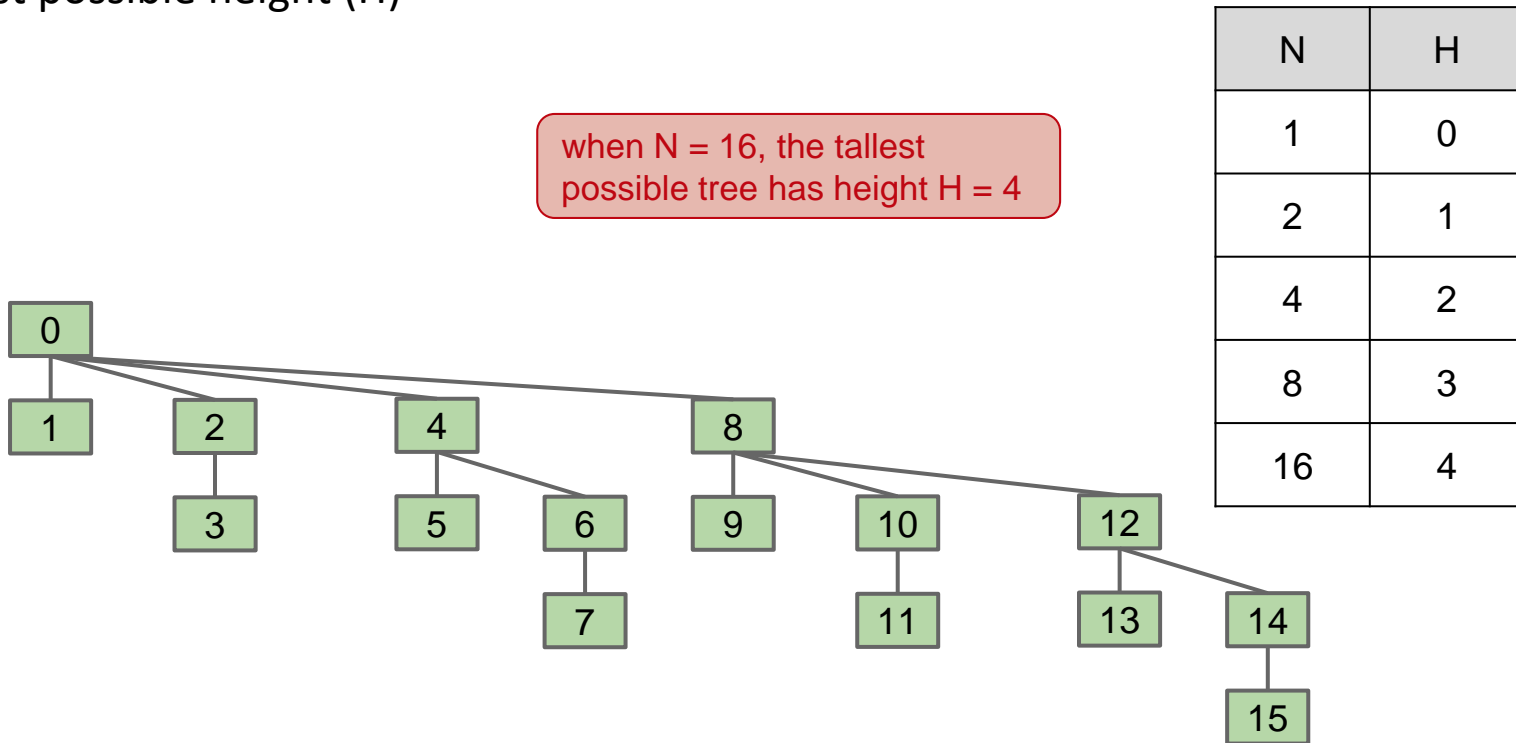- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)

when N = 16, the tallest possible tree has height H = 4

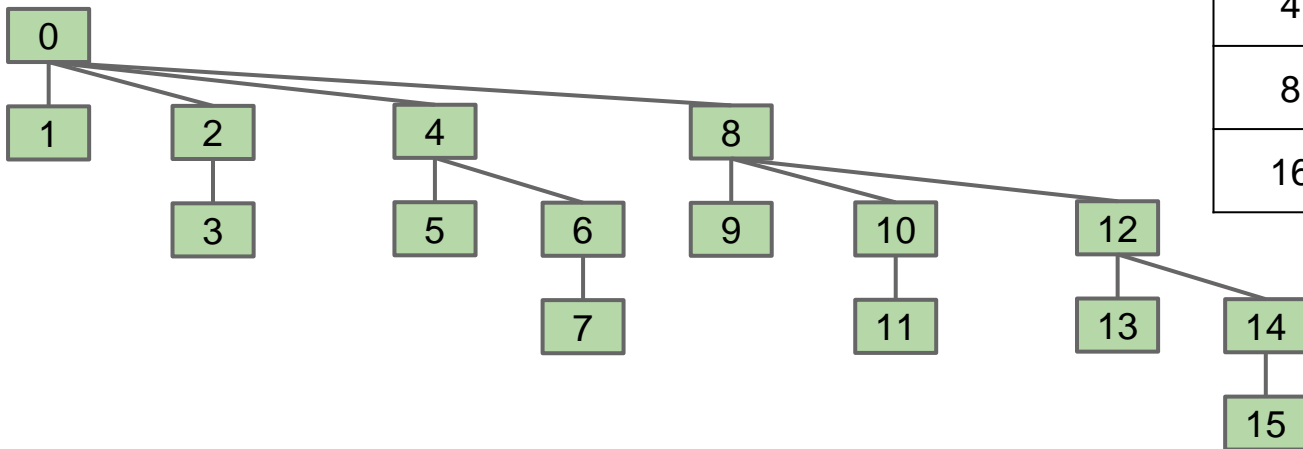| N	H
1	0
2	1
4	2
8	3
16	4

Relationship between Weight and Height (10)

- We now illustrate the relationship between the weight (N, number of items) and the worst possible height (H)
    - the worst-case tree height H = O(log N)
    - since connect and isConnected both depends on find, and find depends on the height, they are also O(log N)

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

Performance Summary

- Our fourth attempt has finally achieved a good performance!
  - logarithmic time is fast enough for practical problems

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

Thank you for your attention !

- In this lecture, you have learned:
  - about assertions, incremental development, and scope minimization
  - about encapsulation
  - to identify aliasing and understand the dangers of mutability
  - to use immutability to improve correctness, clarity, and changeability
  - to create an iterator that iterate through your data structures
  - to create an efficient disjoint sets data structure

- Please continue to Lecture Quiz 9 and Lab 9:
  - to do Lab Exercise 9.1 - 9.4, and
  - to do Exercise 9.1 - 9.3