Chapter 13: Sets and Disjoint Sets

Data Structures

Set ADT | Union, intersection, difference | Hash- and tree-based sets | Multisets | Union-Find | Path compression, rank | Connected components

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/data_structures/slides/Chapter_13_Sets_and_Disjoint_Sets.html

INTRODUCTION

Learning Outcomes

The set model

Explain the set ADT and Java's `Set` interface.

Multisets

Use a multiset for problems where items repeat.

Union-Find

Implement path compression and union by rank.

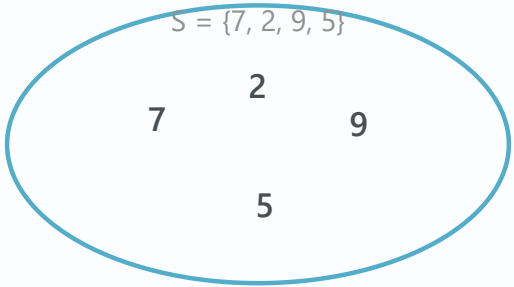
Applications

Apply Union-Find to connectivity problems such as counting connected components.

SET ADT

Set

- A set is a collection of **unique** elements: no element appears twice.
- It is the data-structure counterpart of a set in mathematics.
- Order does not matter: {1, 2, 3} = {3, 1, 2}.
- The key question, **"is x in the set?"** (the membership test), must be answered fast.



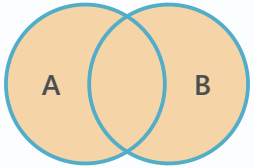
add(2) → no change (2 already there)

Removing duplicates
The distinct words in a list.
`new HashSet<>(list)`

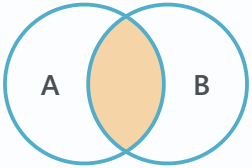
Membership
Banned users, visited nodes.

Set algebra
Mutual friends = intersection; all tags = union.

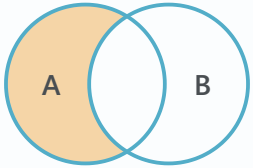
Set Operations



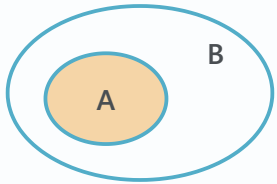
Union A ∪ B
in A *or* in B



Intersection A ∩ B
in *both* A and B



Difference A − B
in A but not in B



Subset A ⊆ B
every element of A is in B

A = {1, 2, 3, 4, 5}, B = {3, 4, 5, 6, 7}	**Result**
Union A ∪ B	{1, 2, 3, 4, 5, 6, 7}
Intersection A ∩ B	{3, 4, 5}
Difference A − B / B − A	{1, 2} / {6, 7}
Subset?	A ⊄ B (1 ∉ B) · {3, 4} ⊆ A ✓

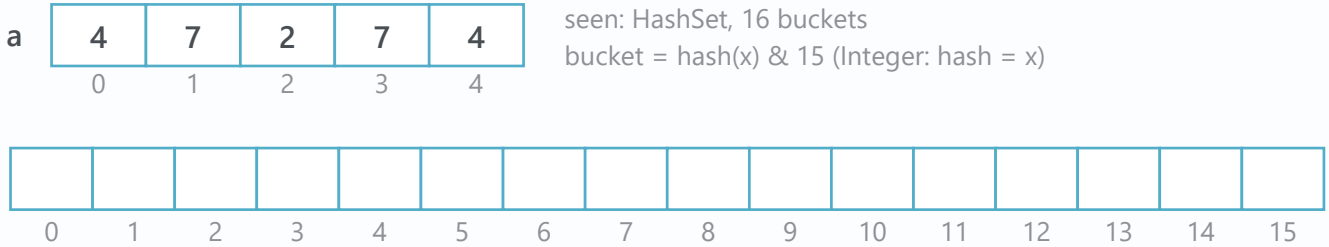
The java.util.Set Interface

Method	Description
`add(e)`	adds; `false` if present
`remove(o)`, `contains(o)`	removes; tests membership
`addAll(T)`	**union**: S ← S ∪ T
`retainAll(T)`	**intersection**: S ← S ∩ T
`removeAll(T)`	**difference**: S ← S − T

```
Set<Integer> a = new HashSet<>(List.of(1,2,3,4,5));
Set<Integer> b = new HashSet<>(List.of(3,4,5,6,7));
Set<Integer> uni = new HashSet<>(a);
uni.addAll(b);      // [1..7]
Set<Integer> both = new HashSet<>(a);
both.retainAll(b); // [3, 4, 5]
Set<Integer> diff = new HashSet<>(a);
diff.removeAll(b); // [1, 2]
a.containsAll(b);  // false
```

`addAll`, `retainAll`, and `removeAll` **modify the set in place**; copy a first if you need to keep it.

Code Trace: First Repeat via the Result of `add`



We create an empty `HashSet`: 16 empty buckets. Each element goes to bucket `hash & 15`.

Step 1 / 10

```
1 int firstRepeat(int[] a) {
▶   Set<Integer> seen = new HashSet<>();
3   for (int x : a) {
4     if (!seen.add(x))   // not added?
5       return x;         // already there: a repeat
6   }
7   return -1;            // no repeats
8 }
```

VARIABLES

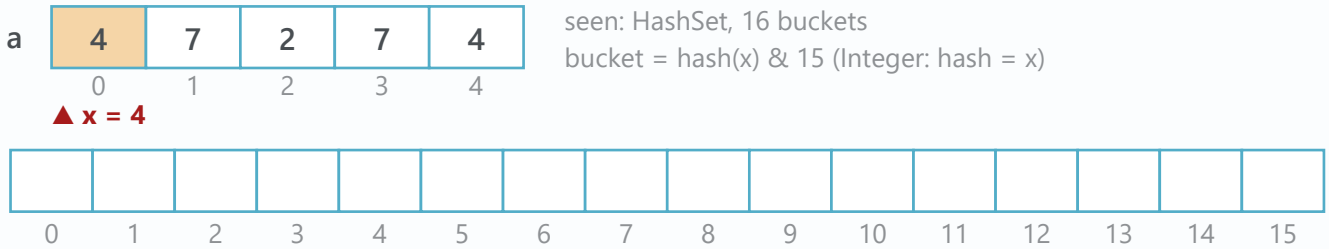
`a = [4, 7, 2, 7, 4]` `seen = {}`

CALL STACK

`firstRepeat(a)`

`main()`

Code Trace: First Repeat via the Result of `add`



The loop takes the next element: **x = 4**.

Step 2 / 10

```
1  int firstRepeat(int[] a) {
2    Set<Integer> seen = new HashSet<>();
▶    for (int x : a) {
4      if (!seen.add(x))   // not added?
5        return x;         // already there: a repeat
6    }
7    return -1;            // no repeats
8  }
```

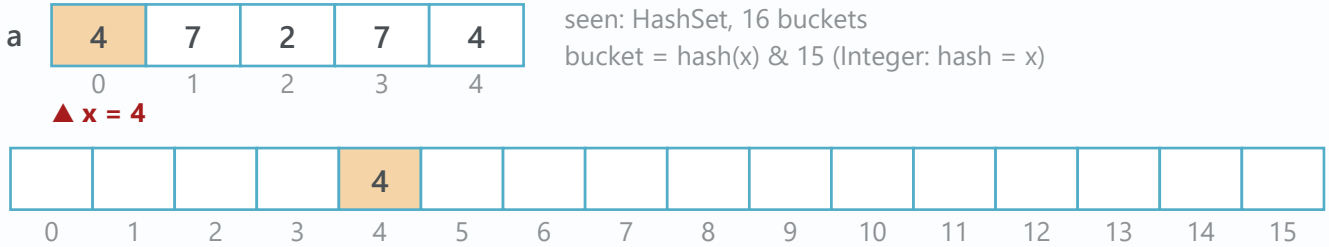
VARIABLES

`a = [4, 7, 2, 7, 4]` `seen = {}` `x = 4`

CALL STACK

- `firstRepeat(a)`
- `main()`

Code Trace: First Repeat via the Result of `add`



`add(4)` : bucket 4 is empty, so the element goes in and **true** comes back → `!true` is false and the loop continues. One bucket checked: O(1).

Step 3 / 10

```
1  int firstRepeat(int[] a) {
2    Set<Integer> seen = new HashSet<>();
3    for (int x : a) {
▶      if (!seen.add(x))   // not added?
5        return x;           // already there: a repeat
6    }
7    return -1;              // no repeats
8  }
```

VARIABLES

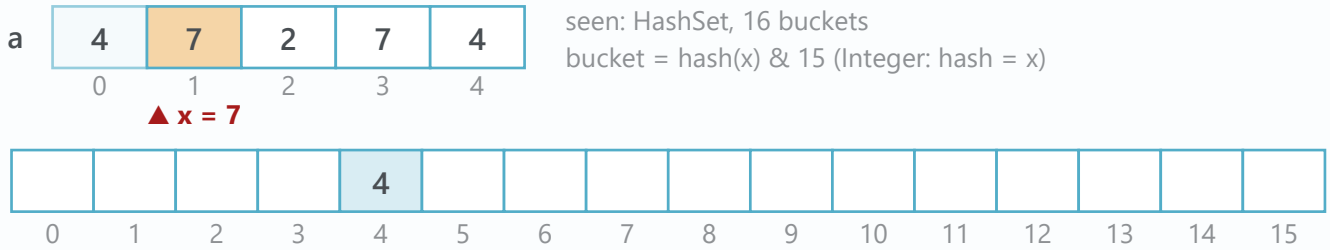
`a = [4, 7, 2, 7, 4]` `seen = {4}` `x = 4` `add = true`

CALL STACK

`firstRepeat(a)`

`main()`

Code Trace: First Repeat via the Result of `add`



The loop takes the next element: **x = 7**.

Step 4 / 10

```
1  int firstRepeat(int[] a) {
2    Set<Integer> seen = new HashSet<>();
▶    for (int x : a) {
4      if (!seen.add(x))   // not added?
5        return x;         // already there: a repeat
6    }
7    return -1;            // no repeats
8  }
```

VARIABLES

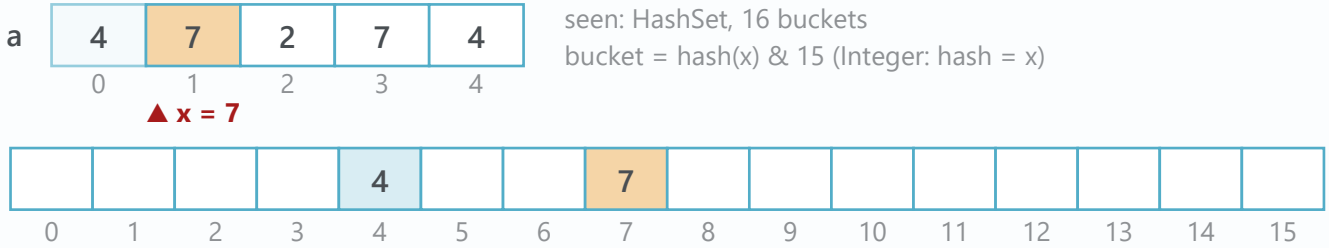
`a = [4, 7, 2, 7, 4]` `seen = {4}` `x = 7`

CALL STACK

`firstRepeat(a)`

`main()`

Code Trace: First Repeat via the Result of `add`



`add(7)` : bucket 7 is empty, so the element goes in and **true** comes back → `!true` is false and the loop continues. One bucket checked: O(1).

Step 5 / 10

```
1 int firstRepeat(int[] a) {
2   Set<Integer> seen = new HashSet<>();
3   for (int x : a) {
▶     if (!seen.add(x))   // not added?
5       return x;         // already there: a repeat
6   }
7   return -1;            // no repeats
8 }
```

VARIABLES

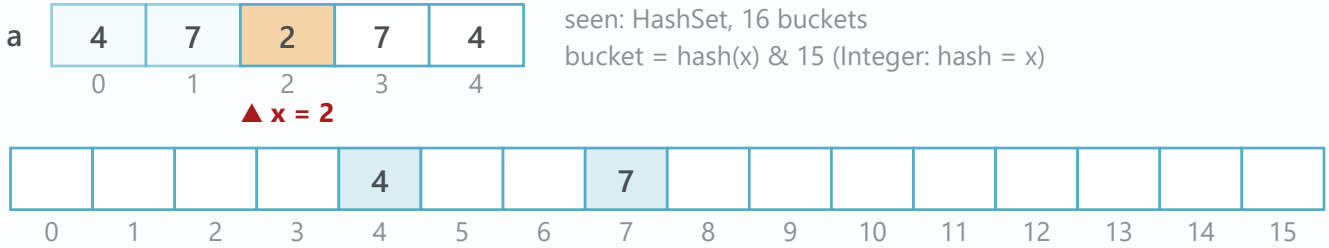
`a = [4, 7, 2, 7, 4]` `seen = {4, 7}` `x = 7` `add = true`

CALL STACK

`firstRepeat(a)`

`main()`

Code Trace: First Repeat via the Result of `add`



The loop takes the next element: **x = 2**.

Step 6 / 10

```
1  int firstRepeat(int[] a) {
2    Set<Integer> seen = new HashSet<>();
▶    for (int x : a) {
4      if (!seen.add(x))   // not added?
5        return x;         // already there: a repeat
6    }
7    return -1;            // no repeats
8  }
```

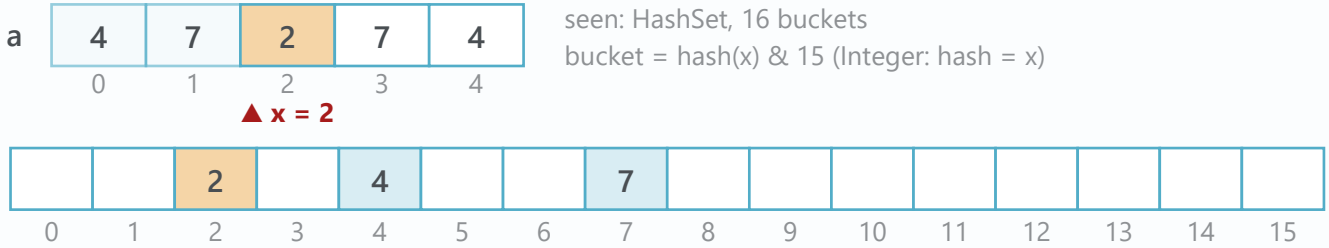
VARIABLES

`a = [4, 7, 2, 7, 4]` `seen = {4, 7}` `x = 2`

CALL STACK

- `firstRepeat(a)`
- `main()`

Code Trace: First Repeat via the Result of `add`



`add(2)` : bucket 2 is empty, so the element goes in and **true** comes back → `!true` is false and the loop continues. One bucket checked: O(1).

Step 7 / 10

```
1 int firstRepeat(int[] a) {
2   Set<Integer> seen = new HashSet<>();
3   for (int x : a) {
▶     if (!seen.add(x))   // not added?
5       return x;         // already there: a repeat
6   }
7   return -1;            // no repeats
8 }
```

VARIABLES

`a = [4, 7, 2, 7, 4]`

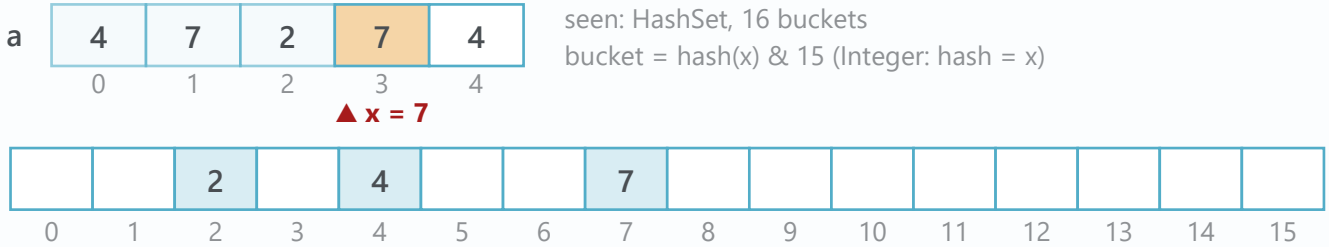
`seen = {2, 4, 7}` `x = 2` `add = true`

CALL STACK

`firstRepeat(a)`

`main()`

Code Trace: First Repeat via the Result of `add`



The loop takes the next element: **x = 7**.

Step 8 / 10

```
1  int firstRepeat(int[] a) {
2    Set<Integer> seen = new HashSet<>();
▶    for (int x : a) {
4      if (!seen.add(x))   // not added?
5        return x;         // already there: a repeat
6    }
7    return -1;            // no repeats
8  }
```

VARIABLES

`a = [4, 7, 2, 7, 4]`

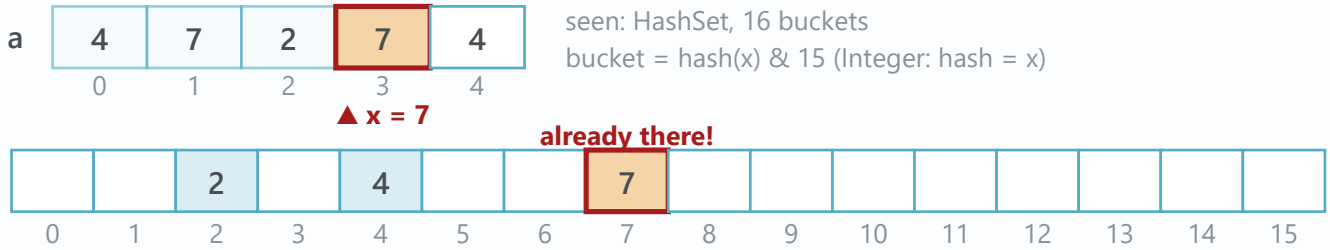
`seen = {2, 4, 7}` `x = 7`

CALL STACK

`firstRepeat(a)`

`main()`

Code Trace: First Repeat via the Result of `add`



`add(7)` : **7 is already** in bucket 7; the set is unchanged and **false** comes back → the condition holds.

Step 9 / 10

```
1 int firstRepeat(int[] a) {
2   Set<Integer> seen = new HashSet<>();
3   for (int x : a) {
▶     if (!seen.add(x))   // not added?
5       return x;           // already there: a repeat
6   }
7   return -1;              // no repeats
8 }
```

VARIABLES

`a = [4, 7, 2, 7, 4]`

`seen = {2, 4, 7}` `x = 7`

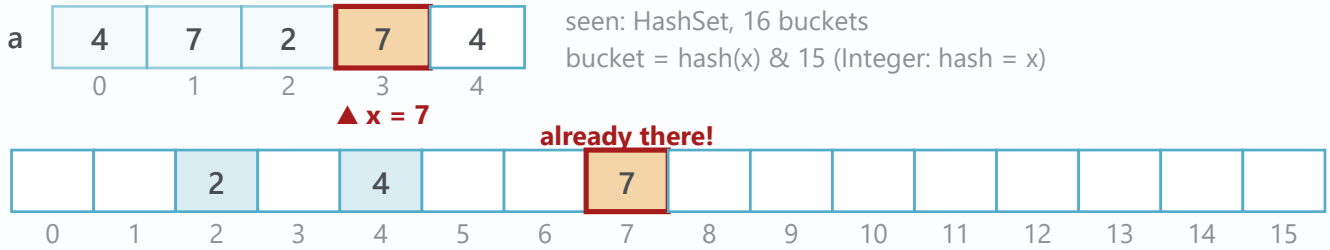
`add = false`

CALL STACK

`firstRepeat(a)`

`main()`

Code Trace: First Repeat via the Result of `add`



The first repeated element, **7**, is returned. Total work O(n): one `add` per element, no nested loop.

Step 10 / 10

```
1 int firstRepeat(int[] a) {
2   Set<Integer> seen = new HashSet<>();
3   for (int x : a) {
4     if (!seen.add(x))    // not added?
▶       return x;          // already there: a repeat
6   }
7   return -1;             // no repeats
8 }
```

VARIABLES

`a = [4, 7, 2, 7, 4]`

`seen = {2, 4, 7}` `x = 7`

`returned = 7`

CALL STACK

`firstRepeat(a)`

`main()`

SET IMPLEMENTATIONS

java.util.Set Implementations

Class	Internal structure	Iteration order	add / contains / remove
HashSet	hash table	depends on hash codes, unpredictable	O(1) average
LinkedHashSet	hash table + doubly linked list	insertion order	O(1) average
TreeSet	red-black tree	sorted (ascending, alphabetical)	O(log n)

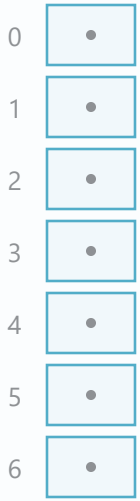
```
for (String s : List.of("lemon", "peach", "fig", "lemon")) {
    hash.add(s);  linked.add(s);  tree.add(s);
}
// HashSet       → [lemon, fig, peach]   (bucket order: depends on hash)
// LinkedHashSet → [lemon, peach, fig]   (insertion order)
// TreeSet       → [fig, lemon, peach]   (alphabetical)
```

The repeated "lemon" appears **once** in all three. If order doesn't matter, use HashSet; for insertion order, LinkedHashSet; for sorted access, TreeSet.

SET IMPLEMENTATIONS

Hash-Based Set

- The set is stored as a **hash table** that holds keys only, with no values.
- Each element's position is computed quickly from its hash code.
- Insertion, removal, and lookup take **O(1)** on average; with a bad hash, O(n) in the worst case.
- Java's `HashSet` uses a `HashMap` internally.

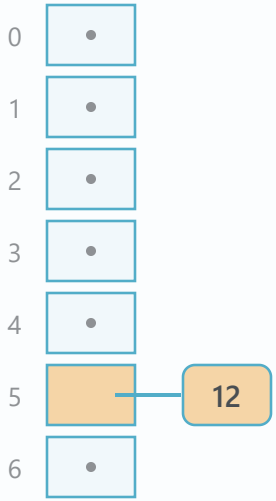


Start: an empty table with 7 buckets. Position = `hashCode(x) mod 7`.

Step 1 / 7

Hash-Based Set

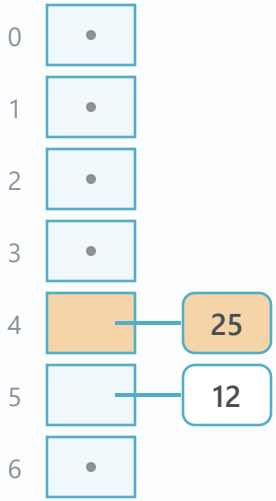
- The set is stored as a **hash table** that holds keys only, with no values.
- Each element's position is computed quickly from its hash code.
- Insertion, removal, and lookup take **O(1)** on average; with a bad hash, O(n) in the worst case.
- Java's `HashSet` uses a `HashMap` internally.



add(12): 12 mod 7 = **5** → bucket 5 is empty, so 12 goes in.

Hash-Based Set

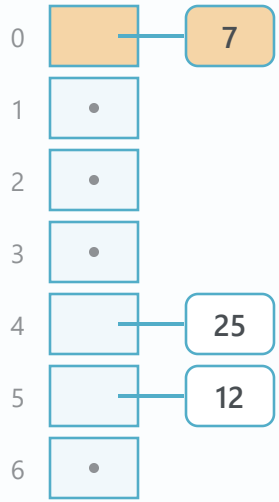
- The set is stored as a **hash table** that holds keys only, with no values.
- Each element's position is computed quickly from its hash code.
- Insertion, removal, and lookup take **O(1)** on average; with a bad hash, O(n) in the worst case.
- Java's `HashSet` uses a `HashMap` internally.



add(25): 25 mod 7 = **4** → into bucket 4.

Hash-Based Set

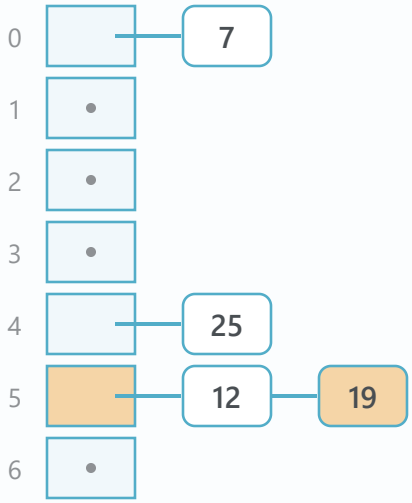
- The set is stored as a **hash table** that holds keys only, with no values.
- Each element's position is computed quickly from its hash code.
- Insertion, removal, and lookup take **O(1)** on average; with a bad hash, O(n) in the worst case.
- Java's `HashSet` uses a `HashMap` internally.



add(7): 7 mod 7 = **0** → into bucket 0.

Hash-Based Set

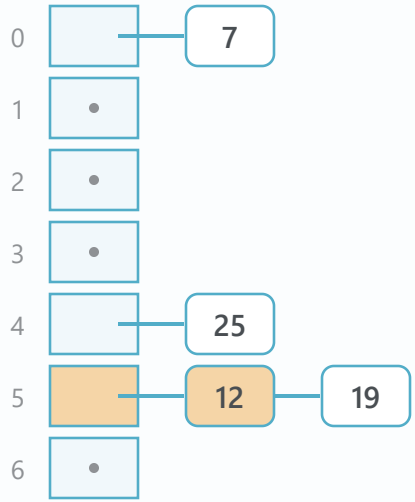
- The set is stored as a **hash table** that holds keys only, with no values.
- Each element's position is computed quickly from its hash code.
- Insertion, removal, and lookup take **O(1)** on average; with a bad hash, O(n) in the worst case.
- Java's `HashSet` uses a `HashMap` internally.



add(19): 19 mod 7 = **5** → collision! Bucket 5 holds 12; 19 goes at the end of the chain.

Hash-Based Set

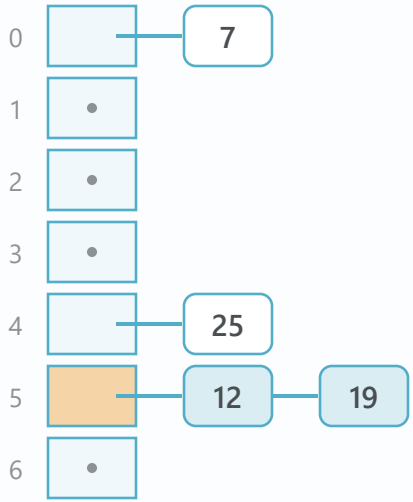
- The set is stored as a **hash table** that holds keys only, with no values.
- Each element's position is computed quickly from its hash code.
- Insertion, removal, and lookup take **O(1)** on average; with a bad hash, O(n) in the worst case.
- Java's `HashSet` uses a `HashMap` internally.



add(12) again: bucket 5 is scanned and 12 is found → **not added**, `false` is returned. This is what makes a set a set.

Hash-Based Set

- The set is stored as a **hash table** that holds keys only, with no values.
- Each element's position is computed quickly from its hash code.
- Insertion, removal, and lookup take **O(1)** on average; with a bad hash, O(n) in the worst case.
- Java's `HashSet` uses a `HashMap` internally.



contains(26): 26 mod 7 = 5 → only bucket 5 (12, 19) is scanned; 26 is not found → `false`. The other buckets are never examined.

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {          // x ≥ 0
▶    int i = x % m;               // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `add(12)`
- `main()`



`add(12)`: bucket no. = 12 % 7 = **5**. The bucket is empty, so the scan ends at once.

Step 1 / 18

VARIABLES

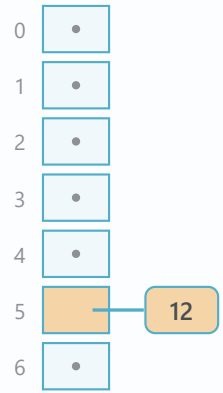
`x = 12` `i = 5` `n = 0`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
▶    bucket[i].add(x);
▶    n++;
▶    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `add(12)`
- `main()`



12 is not in the bucket → appended to the chain, **n = 1**, `return true`.

VARIABLES

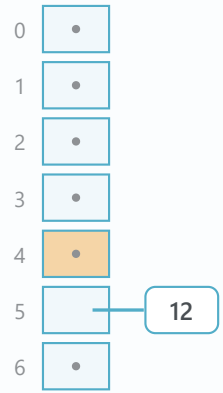
`x = 12` `i = 5` `n = 1` `returned = true`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {          // x ≥ 0
▶    int i = x % m;               // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `add(25)`
- `main()`



`add(25)`: bucket no. = 25 % 7 = **4**. The bucket is empty, so the scan ends at once.

Step 3 / 18

VARIABLES

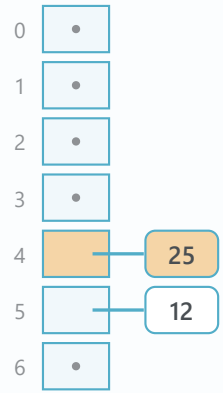
`x = 25` `i = 4` `n = 1`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {            // x ≥ 0
3    int i = x % m;                 // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
▶    bucket[i].add(x);
▶    n++;
▶    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `add(25)`
- `main()`



25 is not in the bucket → appended to the chain, **n = 2**, `return true`.

VARIABLES

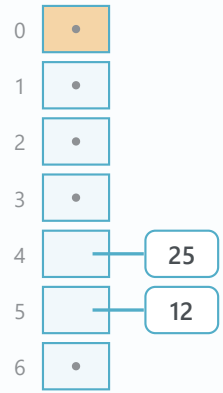
`x = 25` `i = 4` `n = 2` `returned = true`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {            // x ≥ 0
▶    int i = x % m;                 // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `add(7)`
- `main()`



`add(7)` : bucket no. = 7 % 7 = **0**. The bucket is empty, so the scan ends at once.

Step 5 / 18

VARIABLES

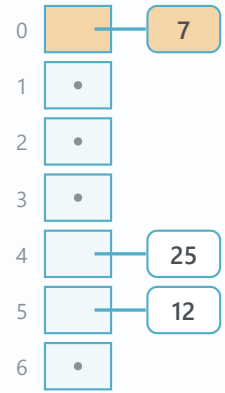
`x = 7` `i = 0` `n = 2`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
▶    bucket[i].add(x);
▶    n++;
▶    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- **add(7)**
- main()



7 is not in the bucket → appended to the chain, **n = 3**, `return true`.

Step 6 / 18

VARIABLES

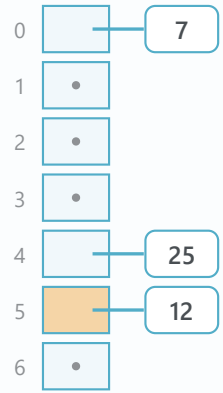
`x = 7` `i = 0` `n = 3` `returned = true`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {          // x ≥ 0
▶    int i = x % m;               // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `add(19)`
- `main()`



`add(19)`: bucket no. = 19 % 7 = **5**. The chain in the bucket must be scanned: 12.

Step 7 / 18

VARIABLES

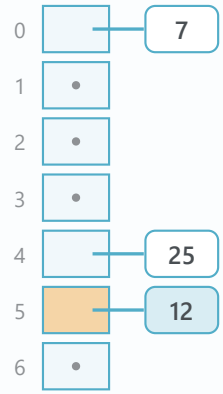
`x = 19` `i = 5` `n = 3`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // bucket no.
4    for (int y : bucket[i])
▶      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `add(19)`
- `main()`



y = 12 ≠ 19 → move along the chain.

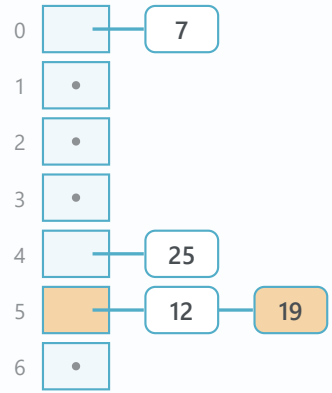
Step 8 / 18

VARIABLES

`x = 19` `i = 5` `n = 3` `y = 12`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already there
▶    bucket[i].add(x);
▶    n++;
▶    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```



19 is not in the bucket → appended to the chain, **n = 4**, `return true`. (Collision: bucket 5 now holds 2 elements.)

Step 9 / 18

CALL STACK

- `add(19)`
- `main()`

VARIABLES

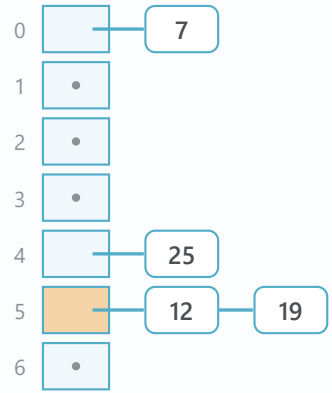
`x = 19` `i = 5` `n = 4` `returned = true`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {            // x ≥ 0
▶    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;   // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `add(12)`
- `main()`



`add(12)`: bucket no. = 12 % 7 = **5**. The chain in the bucket must be scanned: 12, 19.

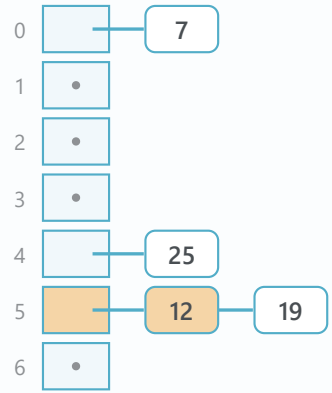
Step 10 / 18

VARIABLES

`x = 12` `i = 5` `n = 4`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // bucket no.
4    for (int y : bucket[i])
▶      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```



y = 12 == 12 → the element is **already in the set**: `return false`, nothing is added. This is what makes a set a set.

Step 11 / 18

CALL STACK

- `add(12)`
- `main()`

VARIABLES

`x = 12` `i = 5` `n = 4` `y = 12`

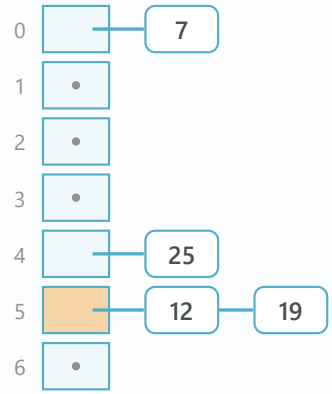
`returned = false`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
▶    int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `contains(26)`
- `main()`



`contains(26)` : 26 % 7 = **5** → only bucket 5 is scanned; the other buckets are never examined.

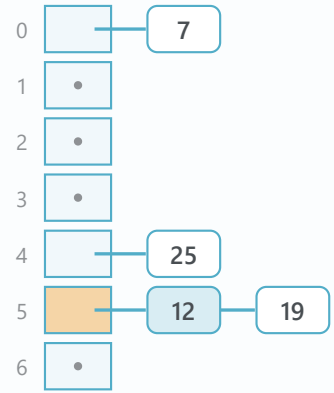
Step 12 / 18

VARIABLES

`x = 26` `i = 5` `n = 4`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
▶      if (y == x) return true;
14   return false;
15 }
```



y = 12 ≠ 26 → keep going.

Step 13 / 18

CALL STACK

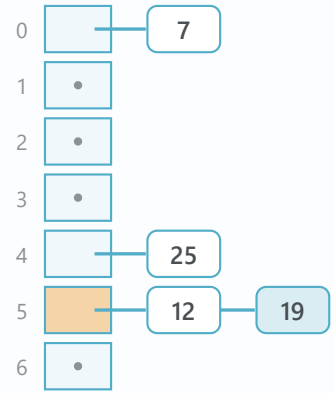
- `contains(26)`
- `main()`

VARIABLES

`x = 26` `i = 5` `n = 4` `y = 12`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
▶      if (y == x) return true;
14   return false;
15 }
```



y = 19 ≠ 26 → keep going.

Step 14 / 18

CALL STACK

- `contains(26)`
- `main()`

VARIABLES

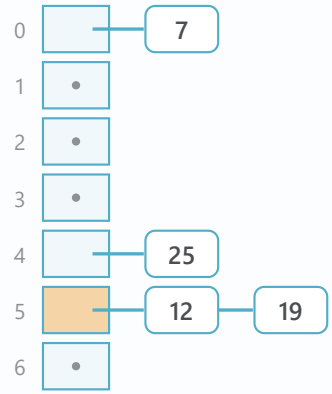
`x = 26` `i = 5` `n = 4` `y = 19`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
▶    return false;
15 }
```

CALL STACK

- `contains(26)`
- `main()`



End of chain, no 26 → `return false`. The average chain length n/m is constant → **O(1)**.

Step 15 / 18

VARIABLES

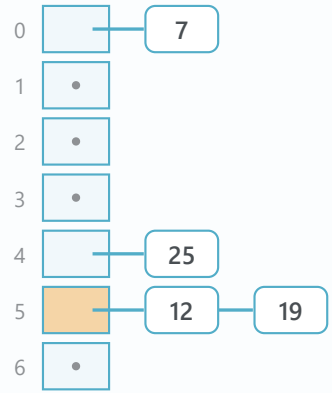
`x = 26` `i = 5` `n = 4` `returned = false`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
▶    int i = x % m;
12   for (int y : bucket[i])
13     if (y == x) return true;
14   return false;
15 }
```

CALL STACK

- `contains(19)`
- `main()`



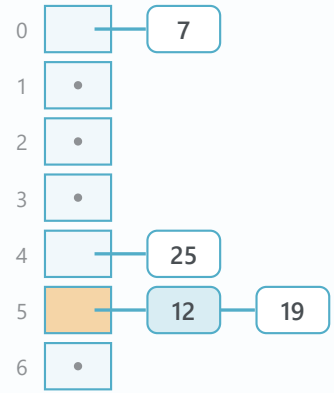
`contains(19)` : 19 % 7 = **5** → only bucket 5 is scanned; the other buckets are never examined.

VARIABLES

`x = 19` `i = 5` `n = 4`

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
▶      if (y == x) return true;
14   return false;
15 }
```



y = 12 ≠ 19 → keep going.

Step 17 / 18

CALL STACK

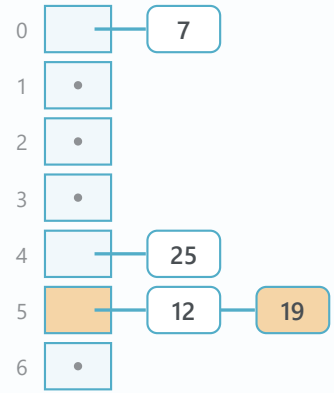
- **contains(19)**
- main()

VARIABLES

x = 19 | i = 5 | n = 4 | y = 12

Code Trace: `add` / `contains` (m = 7)

```
1  // fields: List<Integer>[] bucket; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // bucket no.
4    for (int y : bucket[i])
5      if (y == x) return false;  // already
   there
6    bucket[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : bucket[i])
▶      if (y == x) return true;
14   return false;
15 }
```



y = 19 == 19 → **found**, `return true`.

Step 18 / 18

CALL STACK

- `contains(19)`
- `main()`

VARIABLES

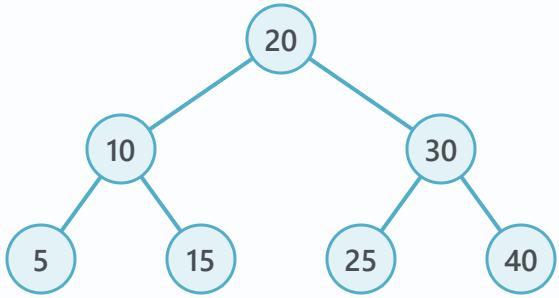
`x = 19` `i = 5` `n = 4` `y = 19`

`returned = true`

SET IMPLEMENTATIONS

Tree-Based (Sorted) Set

- The set is stored as a **balanced binary search tree** (Java's `TreeSet` : a red-black tree).
- Elements are kept **sorted**; an inorder traversal produces them in order.
- Lookup, insertion, and removal take **O(log n)**.
- Elements must be comparable (`Comparable` or `Comparator`).
- `TreeSet` treats two elements as the same when `compare(a, b) == 0` , so the ordering must be consistent with `equals` .



Extra operations a sorted structure provides (S = {5, 10, 15, 20, 25, 30, 40}):

first / last

`5 / 40`

floor(17)

largest ≤ 17

`15`

ceiling(17)

smallest ≥ 17

`20`

subSet(10, 30)

range query

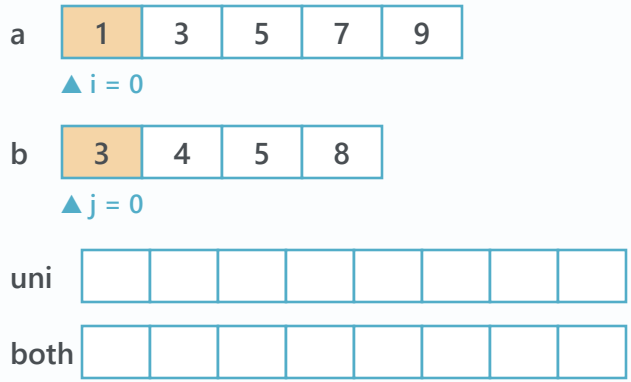
`[10, 15, 20, 25]`

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
▶    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) uni.add(a[i++]);
6      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
8        uni.add(a[i]);  both.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) uni.add(a[i++]);
13   while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 0` `j = 0` `a[i] = 1` `b[j] = 3`



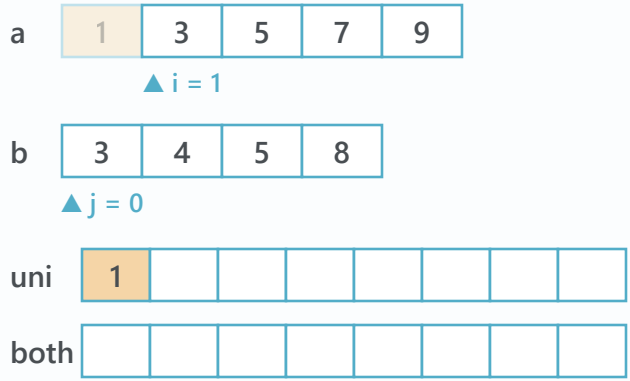
The sets are in **sorted** arrays. Like *merge* in merge sort, two pointers advance in a single pass (Goodrich: *generic merge*).

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
▶      if (a[i] < b[j]) uni.add(a[i++]);
6      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
8        uni.add(a[i]);  both.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) uni.add(a[i++]);
13   while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 0` `j = 0` `a[i] = 1` `b[j] = 3`



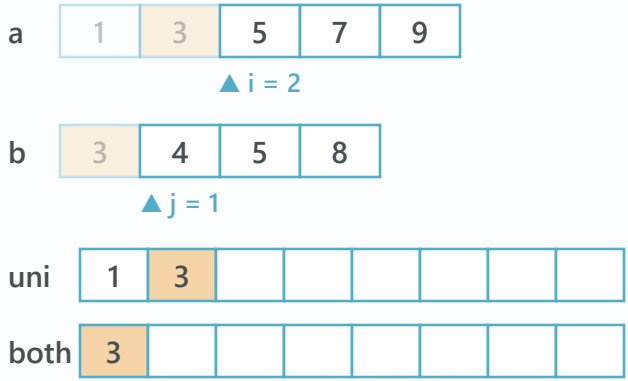
a[i] = 1 < b[j] = 3 → 1 is only in a: added to the **union**, i++.

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) uni.add(a[i++]);
6      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
▶        uni.add(a[i]);  both.add(a[i]);
▶        i++;  j++;
10     }
11   }
12   while (i < a.length) uni.add(a[i++]);
13   while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 1` `j = 0` `a[i] = 3` `b[j] = 3`



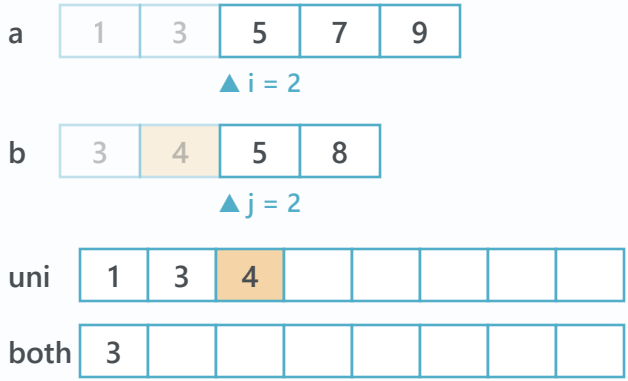
a[i] = b[j] = 3 → **in both**: added once to the union and to the intersection; both pointers advance.

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) uni.add(a[i++]);
▶      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
8        uni.add(a[i]);  both.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) uni.add(a[i++]);
13   while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 2` `j = 1` `a[i] = 5` `b[j] = 4`



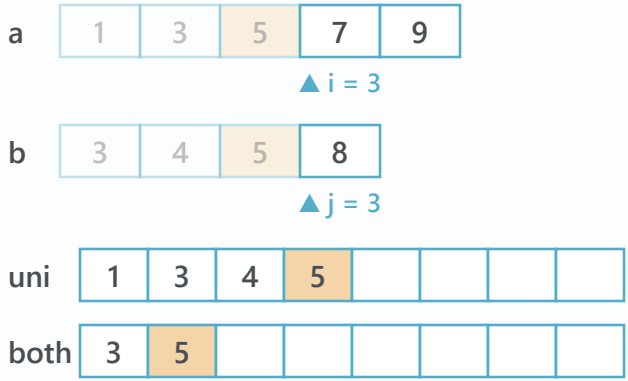
a[i] = 5 > b[j] = 4 → 4 is only in b: added to the **union**, j++.

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) uni.add(a[i++]);
6      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
▶        uni.add(a[i]);  both.add(a[i]);
▶        i++;  j++;
10     }
11   }
12   while (i < a.length) uni.add(a[i++]);
13   while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 2` `j = 2` `a[i] = 5` `b[j] = 5`



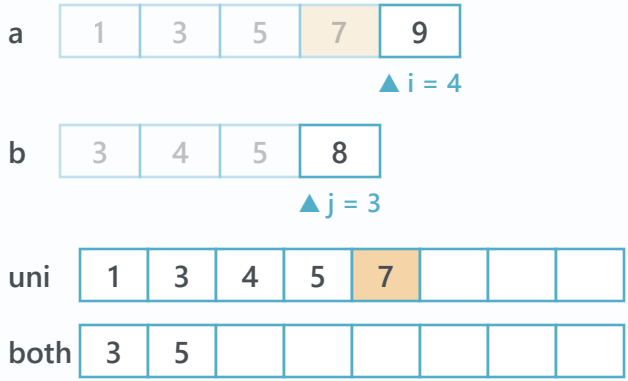
a[i] = b[j] = 5 → **in both**: added once to the union and to the intersection; both pointers advance.

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
▶      if (a[i] < b[j]) uni.add(a[i++]);
6      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
8        uni.add(a[i]);  both.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) uni.add(a[i++]);
13   while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 3` `j = 3` `a[i] = 7` `b[j] = 8`



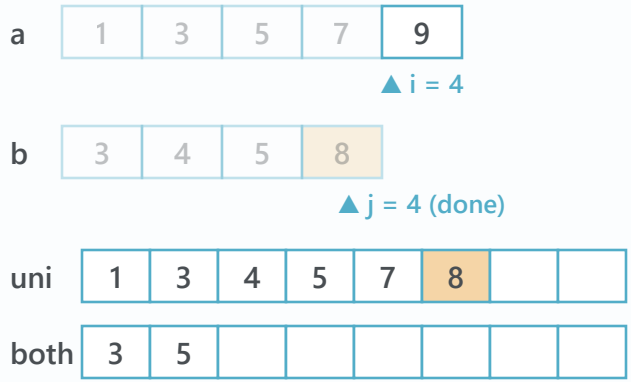
a[i] = 7 < b[j] = 8 → 7 is only in a: added to the **union**, i++.

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) uni.add(a[i++]);
▶      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
8        uni.add(a[i]);  both.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) uni.add(a[i++]);
13   while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 4` `j = 3` `a[i] = 9` `b[j] = 8`



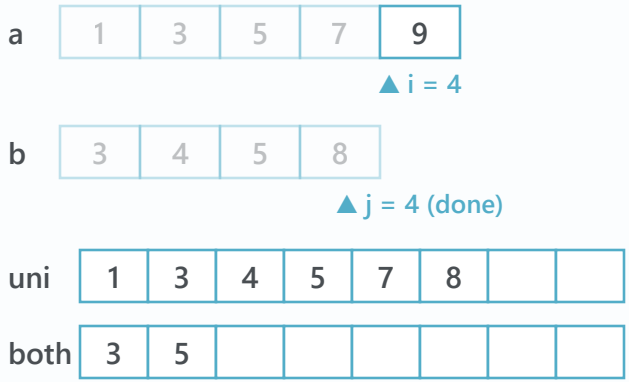
a[i] = 9 > b[j] = 8 → 8 is only in b: added to the **union**, j++.

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
3    int i = 0, j = 0;
▶    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) uni.add(a[i++]);
6      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
8        uni.add(a[i]);  both.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) uni.add(a[i++]);
13   while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 4` `j = 4` `a[i] = 9`



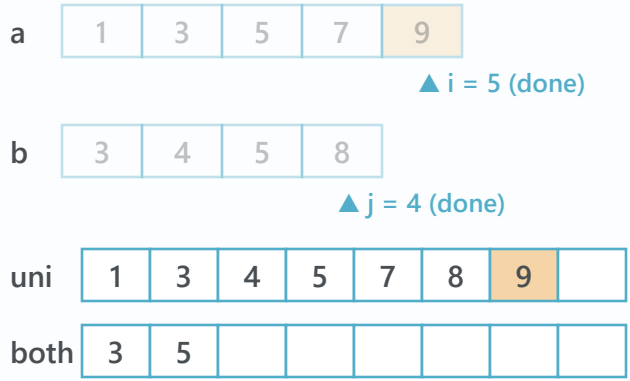
j = 4 = b.length → b is exhausted and the main loop exits. The intersection is complete: no more common elements are possible.

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) uni.add(a[i++]);
6      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
8        uni.add(a[i]);  both.add(a[i]);
9        i++;  j++;
10     }
11   }
▶  while (i < a.length) uni.add(a[i++]);
13   while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 4` `j = 4` `a[i] = 9`



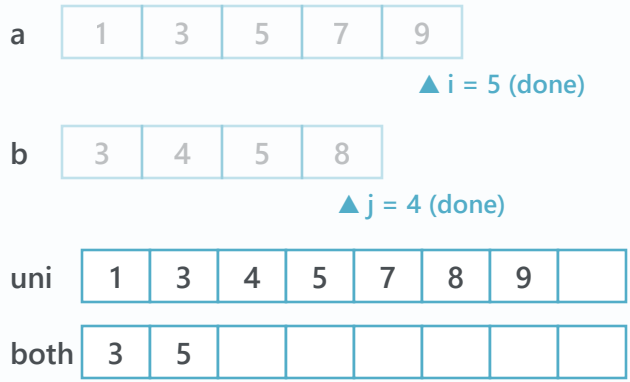
The 9 left in a goes straight into the union, i++.

Union and Intersection of Sorted Arrays (Two Pointers)

```
1  void unionIntersection(int[] a, int[] b,
2      List<Integer> uni, List<Integer> both) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) uni.add(a[i++]);
6      else if (a[i] > b[j]) uni.add(b[j++]);
7      else {                          // in both
8        uni.add(a[i]);  both.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) uni.add(a[i++]);
▶    while (j < b.length) uni.add(b[j++]);
14 }
```

VARIABLES

`i = 5` `j = 4`



Nothing is left in b. Union [1, 3, 4, 5, 7, 8, 9], intersection [3, 5]. Each step advances one pointer → at most n + m steps: **O(n + m)**.

SET IMPLEMENTATIONS

Comparing Set Implementations

Operation (|A| = n, |B| = m)	Unsorted list	Sorted array	Hash table	Balanced BST
`contains`	O(n)	O(log n)	**O(1)***	O(log n)
`add` / `remove`	O(n)†	O(n)	**O(1)***	O(log n)
A ∪ B, A ∩ B, A − B	O(n·m)	O(n + m)	O(n + m)*	O(m log(n+m))
Sorted iteration	O(n log n)	**O(n)**	O(n log n)	**O(n)**
Min / floor / range	O(n)	O(log n)	O(n)	O(log n)

* average case (worst case O(n)). † a lookup is needed first to rule out duplicates. Union with a hash table: `contains` + `add` for each element of B.

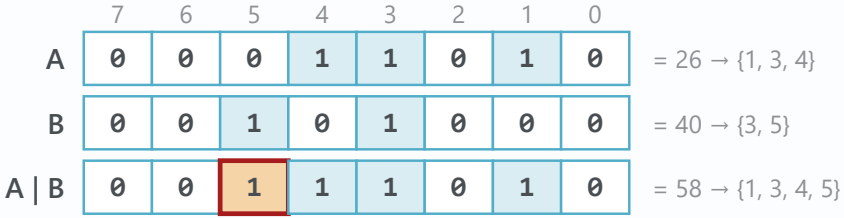
Default choice: hash

The fastest membership test when order doesn't matter.

Need order? Use a tree

Every operation O(log n), and ordered queries come free.

Sets as Bit Vectors (Bitsets)



Idea

With universe 0..U−1, bit i is 1 ⇔ i is in the set.

One element: O(1)

Add, remove, membership = a single bit operation.

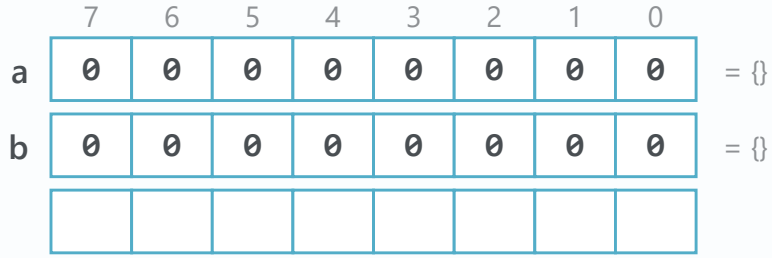
Set algebra: O(U / 64)

∪ ∩ − word by word with `|` `&` `&~`.

Operation	int mask (U ≤ 32)	java.util.BitSet
add x / contains x?	`a |= 1 << x` · `(a >> x & 1) == 1`	`set(x)` · `get(x)`
union / intersection	`a | b` · `a & b`	`or(b)` · `and(b)`
difference / size	`a & ~b` · `Integer.bitCount(a)`	`andNot(b)` · `cardinality()`

Memory is U bits: a universe of 10^6 elements ≈ 122 KB. If the universe is large but the set sparse (e.g. 3 elements, U = 10^9), use a hash-based set.

Code Trace: Set Algebra with Bit Operations



Two empty sets: all bits 0.

Step 1 / 10

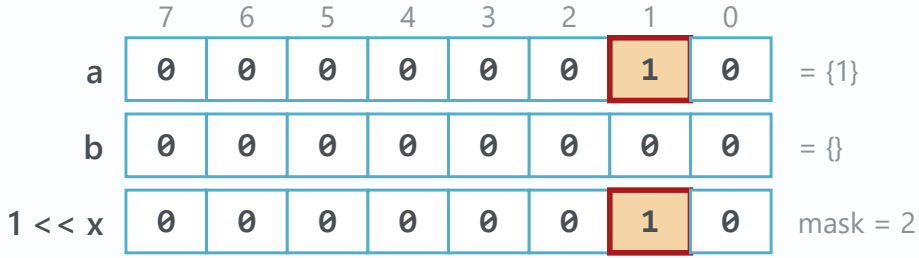
```
▶ int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int uni = a | b;        // union
7 int both = a & b;       // intersection
8 int diff = a & ~b;      // difference
9 boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

a = 0 (00000000)

b = 0 (00000000)

Code Trace: Set Algebra with Bit Operations



`1 << 1` = mask 2; `a |= mask` → bit 1 becomes 1: 1 is added to the set.

Step 2 / 10

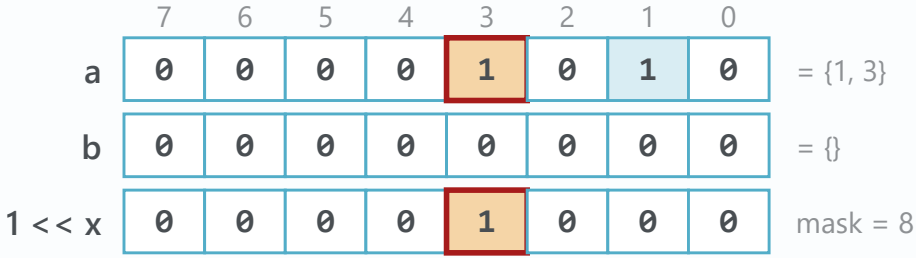
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
▶   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int uni = a | b;        // union
7 int both = a & b;       // intersection
8 int diff = a & ~b;      // difference
9 boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

`a = 2 (00000010)`

`b = 0 (00000000)` `x = 1`

Code Trace: Set Algebra with Bit Operations



`1 << 3` = mask 8; `a |= mask` → bit 3 becomes 1: 3 is added to the set.

Step 3 / 10

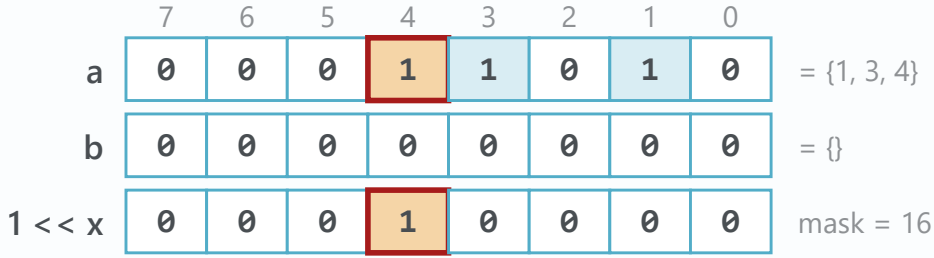
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
▶   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int uni = a | b;        // union
7 int both = a & b;       // intersection
8 int diff = a & ~b;      // difference
9 boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

`a = 10 (00001010)`

`b = 0 (00000000)` `x = 3`

Code Trace: Set Algebra with Bit Operations



`1 << 4` = mask 16; `a |= mask` → bit 4 becomes 1: 4 is added to the set.

Step 4 / 10

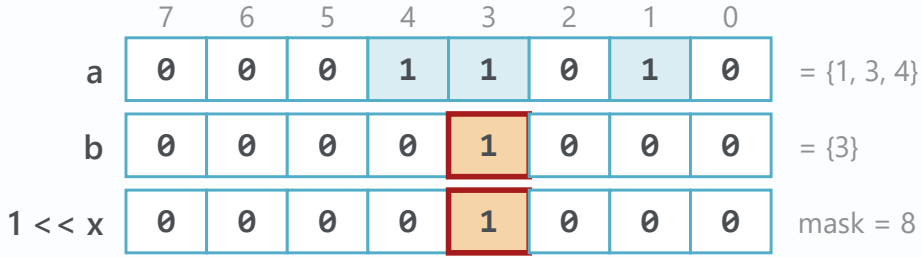
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
▶   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int uni = a | b;        // union
7 int both = a & b;       // intersection
8 int diff = a & ~b;      // difference
9 boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

a = 26 (00011010)

b = 0 (00000000) x = 4

Code Trace: Set Algebra with Bit Operations



Add 3 to set b: bit 3 turns on.

Step 5 / 10

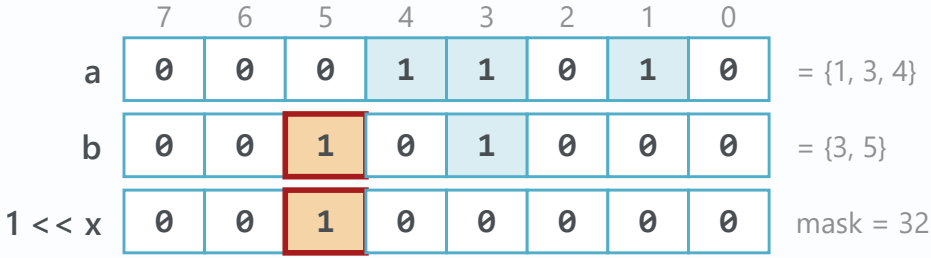
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
▶   b |= 1 << x;
6 int uni = a | b;        // union
7 int both = a & b;       // intersection
8 int diff = a & ~b;      // difference
9 boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

a = **26 (00011010)**

b = **8 (00001000)** x = **3**

Code Trace: Set Algebra with Bit Operations



Add 5 to set b: bit 5 turns on.

Step 6 / 10

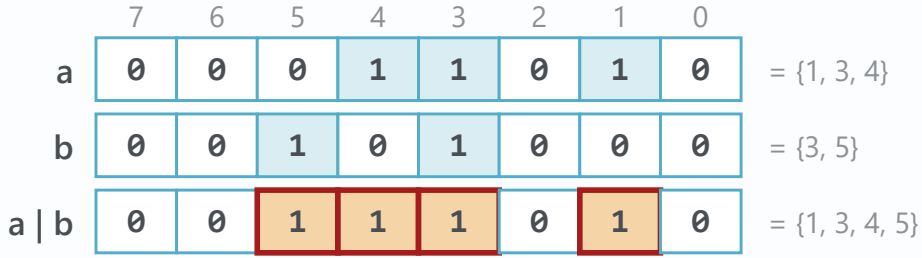
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
▶   b |= 1 << x;
6 int uni = a | b;        // union
7 int both = a & b;       // intersection
8 int diff = a & ~b;      // difference
9 boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

a = **26 (00011010)**

b = **40 (00101000)** x = **5**

Code Trace: Set Algebra with Bit Operations



OR: 1 if either bit is 1 → union {1, 3, 4, 5}. A single machine instruction.

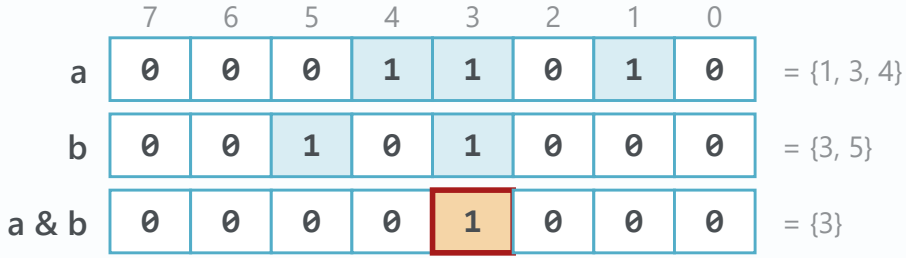
Step 7 / 10

```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
▶ int uni = a | b;        // union
7 int both = a & b;       // intersection
8 int diff = a & ~b;      // difference
9 boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

- a = 26 (00011010)
- b = 40 (00101000)
- uni = 58 (00111010)

Code Trace: Set Algebra with Bit Operations



AND: 1 if both bits are 1 → intersection {3}.

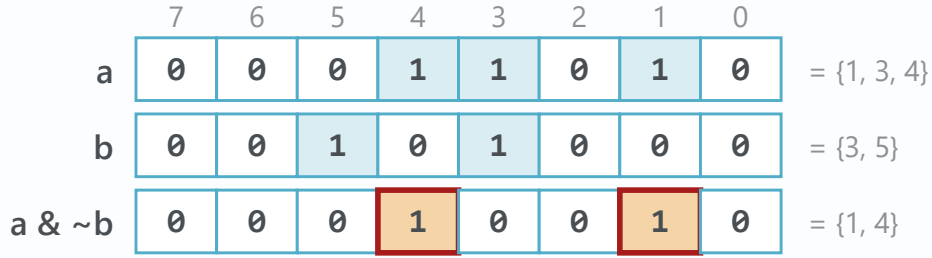
Step 8 / 10

```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int uni = a | b;        // union
▶ int both = a & b;       // intersection
8 int diff = a & ~b;      // difference
9 boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

- a = **26 (00011010)**
- b = **40 (00101000)**
- uni = **58 (00111010)**
- both = **8 (00001000)**

Code Trace: Set Algebra with Bit Operations



`~b` flips the bits of b; `a & ~b` : in a but not in b → difference {1, 4}.

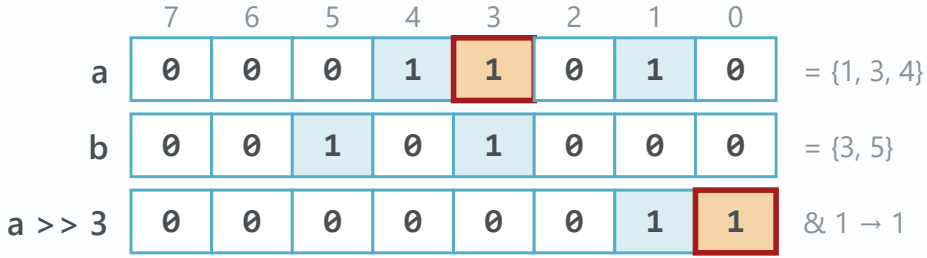
Step 9 / 10

```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int uni = a | b;        // union
7 int both = a & b;       // intersection
▶ int diff = a & ~b;      // difference
9 boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

- a = **26 (00011010)**
- b = **40 (00101000)**
- uni = **58 (00111010)**
- both = **8 (00001000)**
- diff = **18 (00010010)**

Code Trace: Set Algebra with Bit Operations



Membership test: `a >> 3` shifts bit 3 to the far right and `& 1` isolates it → 1, so **3** ∈ **a**. O(1).

Step 10 / 10

```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // set bit x
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int uni = a | b;        // union
7 int both = a & b;       // intersection
8 int diff = a & ~b;      // difference
▶ boolean has3 = (a >> 3 & 1) == 1;
```

VARIABLES

- a = **26 (00011010)**
- b = **40 (00101000)**
- uni = **58 (00111010)**
- both = **8 (00001000)**
- diff = **18 (00010010)**
- has3 = **true**

Multiset (Bag)

- A **multiset** (or *bag*) is a set in which the same element can appear **more than once**.
- Order still doesn't matter; what matters is **how many times** each element occurs (its multiplicity).
- Natural implementation: an **element → count** map.
- Examples: word frequencies in a text, a shopping cart, a vote count.

Element	Count
a	2
b	1
c	3

M = {a, a, b, c, c, c} → map {a:2, b:1, c:3}

Multiset ADT (Goodrich)	Meaning	With hashing
`add(e)` / `add(e, n)`	add e once / n times (count += n)	O(1)*
`count(e)`	how many times is e there? (0 if absent)	O(1)*
`remove(e)` / `remove(e, n)`	decrement the count; delete the entry at 0	O(1)*
`size()`	number of elements, repeats included (6 here)	O(1)

A Multiset in Java

```
class Multiset<E> {
    private final Map<E, Integer> counts = new HashMap<>();
    private int n = 0;                                  // number of elements, repeats included
    void add(E e) { counts.merge(e, 1, Integer::sum); n++; }  // 1 if absent, else +1
    int count(E e) { return counts.getOrDefault(e, 0); }
    boolean remove(E e) {
        Integer c = counts.get(e);
        if (c == null) return false;                    // e is absent
        if (c == 1) counts.remove(e);                      // counter would hit 0: delete entry
        else counts.put(e, c - 1);
        n--;
        return true;
    }
    int size() { return n; }
}
```

Usage

```
add("a"); add("a");
count("a") → 2
```

Unlike a set

In a Set the second `add("a")` has no effect; here the count goes up.

Sorted multiset

Use `TreeMap<E, Integer>` for sorted iteration.

Code Trace: `Multiset` Counters

```
1  void add(E e) {
▶    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

- **add("a")**
- main()

counts (element → count)

element	count
"a"	1

n

1

(repeats included)

M = {a}

`merge` : "a" was absent → added with count **1**; n = 1.

Step 1 / 13

VARIABLES

e = **"a"** n = **1**

Code Trace: `Multiset` Counters

```
1  void add(E e) {
▶    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

- **add("b")**
- main()

counts (element → count)

element	count
"a"	1
"b"	1

n

2

(repeats included)

M = {a, b}

`merge` : "b" was absent → added with count **1**; n = 2.

Step 2 / 13

VARIABLES

e = **"b"** n = **2**

Code Trace: `Multiset` Counters

```
1  void add(E e) {
▶    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

- **add("a")**
- main()

counts (element → count)

element	count
"a"	2
"b"	1

n

3

(repeats included)

M = {a, a, b}

`merge` : "a" was present → `Integer::sum` gives 1 + 1 = **2**; n = 3. A Set would not have changed.

Step 3 / 13

VARIABLES

`e = "a"` `n = 3`

Code Trace: `Multiset` Counters

```
1  void add(E e) {
▶    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

- **add("c")**
- main()

counts (element → count)

element	count
"a"	2
"b"	1
"c"	1

n

4

(repeats included)

M = {a, a, b, c}

`merge` : "c" was absent → added with count **1**; n = 4.

Step 4 / 13

VARIABLES

e = **"c"** n = **4**

Code Trace: `Multiset` Counters

```
1  void add(E e) {
2    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
▶    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

`count("a")`

`main()`

counts (element → count)

element	count
"a"	2
"b"	1
"c"	1

n

4

(repeats included)

M = {a, a, b, c}

count("a"): `getOrDefault` → **2**. A single hash lookup: O(1).

Step 5 / 13

VARIABLES

`e = "a"` `returned = 2`

Code Trace: `Multiset` Counters

```
1  void add(E e) {
2    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
▶    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

- **remove("a")**
- main()

counts (element → count)

element	count
"a"	2
"b"	1
"c"	1

n = 4 (repeats included)

M = {a, a, b, c}

remove("a"): `c = counts.get(e)` = **2**.

Step 6 / 13

VARIABLES

`e = "a"` `c = 2` `n = 4`

Code Trace: `Multiset` Counters

```
1  void add(E e) {
2    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
▶    else counts.put(e, c - 1);
▶    n--;
▶    return true;
14 }
```

CALL STACK

- **remove("a")**
- main()

counts (element → count)

element	count
"a"	1
"b"	1
"c"	1

M = {a, b, c}

n

3

(repeats included)

c = 2 > 1 → the count drops to **1** (the entry stays); n = 3.

Step 7 / 13

VARIABLES

`e = "a"` `c = 2` `n = 3` `returned = true`

Code Trace: `Multiset` Counters

```
1  void add(E e) {
2    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
▶    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

- **remove("b")**
- main()

counts (element → count)

element	count
"a"	1
"b"	1
"c"	1

M = {a, b, c}

n

3

(repeats included)

remove("b"): `c = counts.get(e)` = **1**.

Step 8 / 13

VARIABLES

`e = "b"` `c = 1` `n = 3`

Code Trace: `Multiset` Counters

```
1  void add(E e) {
2    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
9    if (c == null) return false;
▶    if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
▶    n--;
▶    return true;
14 }
```

CALL STACK

- **`remove("b")`**
- `main()`

counts (element → count)

element	count
"a"	1
"c"	1
"b"	0 (deleted)

M = {a, c}

n

2

(repeats included)

c == 1 → the count would reach 0, so the entry is **deleted entirely** (zero counts are not stored); n = 2.

Step 9 / 13

VARIABLES

`e = "b"` `c = 1` `n = 2` `returned = true`

Code Trace: `Multiset` Counters

```
1  void add(E e) {
2    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
▶    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

- **`remove("x")`**
- `main()`

counts (element → count)

element	count
"a"	1
"c"	1

"x" not in the map

n

2

(repeats included)

M = {a, c}

remove("x"): `c = counts.get(e)` = **null**.

Step 10 / 13

VARIABLES

`e = "x"` `c = null` `n = 2`

Code Trace: `Multiset` Counters

```
1  void add(E e) {
2    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
▶    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

- `remove("x")`
- `main()`

counts (element → count)

element	count
"a"	1
"c"	1

"x" not in the map

n

2

(repeats included)

M = {a, c}

c == null → the element is absent: `return false`, n is unchanged.

Step 11 / 13

VARIABLES

`e = "x"` `c = null` `n = 2`

`returned = false`

Code Trace: `Multiset` Counters

```
1  void add(E e) {
2    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
▶    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
14 }
```

CALL STACK

- `count("b")`
- `main()`

counts (element → count)

element	count
"a"	1
"c"	1

"b" not in the map

n = 2 (repeats included)

M = {a, c}

count("b"): `getOrDefault` → **0**. A single hash lookup: O(1).

Step 12 / 13

VARIABLES

`e = "b"` `returned = 0`

Code Trace: `Multiset` Counters

```
1  void add(E e) {
2    counts.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return counts.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = counts.get(e);
9    if (c == null) return false;
10   if (c == 1) counts.remove(e);
11   else counts.put(e, c - 1);
12   n--;
13   return true;
▶ }
```

CALL STACK

`main()`

counts (element → count)

element	count
"a"	1
"c"	1

n

2

(repeats included)

M = {a, c}

Result: M = {a, c}, n = 2. Every operation is one hash access → **O(1)** on average.

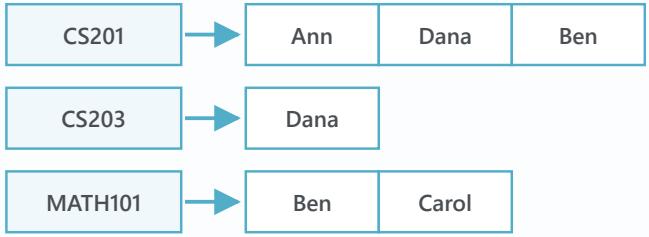
Step 13 / 13

VARIABLES

n = 2

Multimap

- In a **multimap**, one key can map to **several values** (course → students).
- Goodrich ADT: `get(k)` returns all values; `put(k, v)`, `remove(k, v)`, `removeAll(k)`; `size()` counts all (k, v) pairs.
- Java has no built-in class: build one from `Map<K, List<V>>`.




```
Map<String, List<String>> m = new HashMap<>();
void put(String k, String v) {   // create an empty list if absent
  m.computeIfAbsent(k, x -> new ArrayList<>()).add(v);
}
List<String> get(String k) {
  return m.getOrDefault(k, List.of());
}
```

Unlike a multiset, it stores the values themselves, not just *how many*. With hashing, `put` / `get` are O(1)* and `remove(k, v)` is O(1 + list length).

Disjoint Sets and Union-Find

- **Disjoint sets**: a collection of sets that share no elements. Every element belongs to exactly **one** set.
- **Union-Find** (Disjoint Set Union, DSU) is the data structure that maintains them. It has two main operations:

find(x)

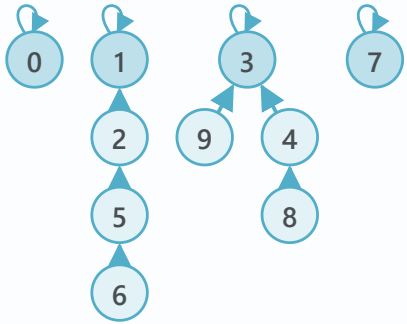
Returns the *representative* (root) of the set containing x.

`find(x) == find(y) ⇔ same set`

union(x, y)

Merges the sets of x and y into one.

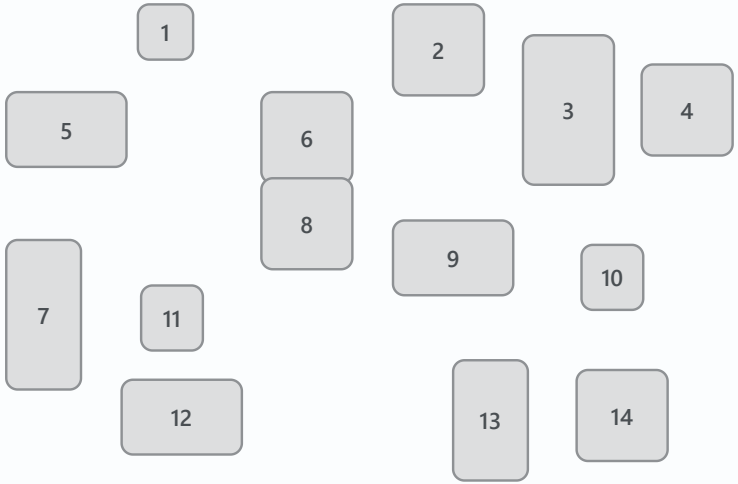
`{1,2} ∪ {3} → {1,2,3}`



Each set is a tree: every node points to its parent; the root points to itself and represents the set.

Magnet Example

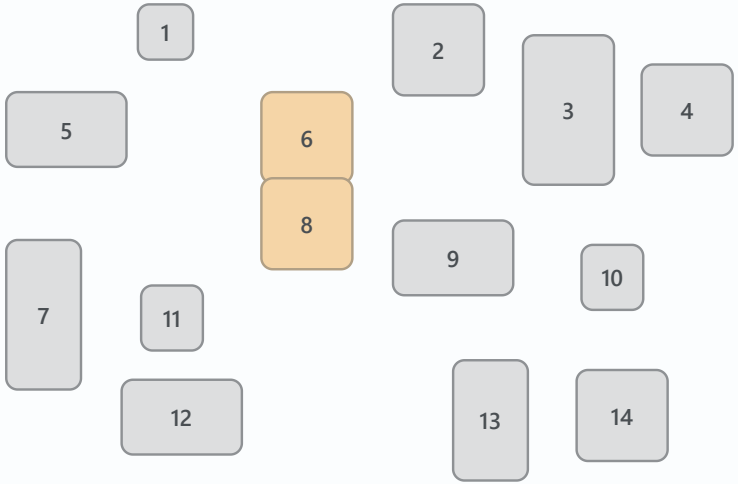
- Picture strong magnets: once two touch, they **never come apart**.
- As they merge, we want to keep track of which magnet is in which group.
- First, each magnet gets a **number**.
- A merge is a `union`; "same group?" compares two `find`s.



14 magnets, each in its own group: 14 disjoint sets. find(x) = x.

Magnet Example

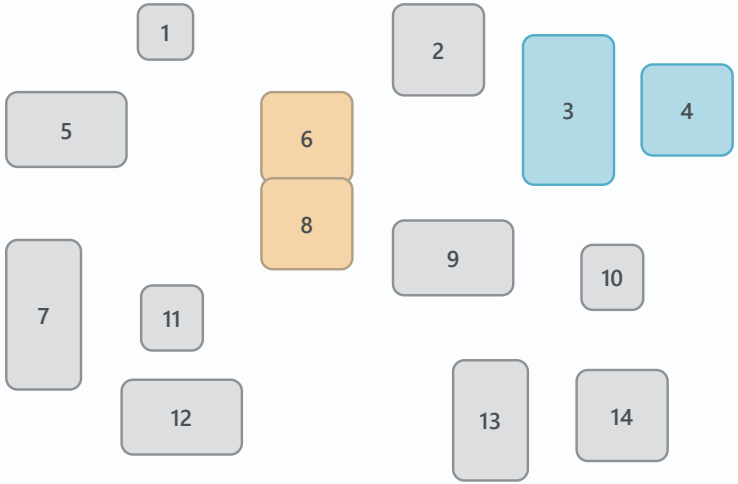
- Picture strong magnets: once two touch, they **never come apart**.
- As they merge, we want to keep track of which magnet is in which group.
- First, each magnet gets a **number**.
- A merge is a `union`; "same group?" compares two `find`s.



union(6, 8): the closest pair, 6 and 8, merge; now find(6) = find(8).

Magnet Example

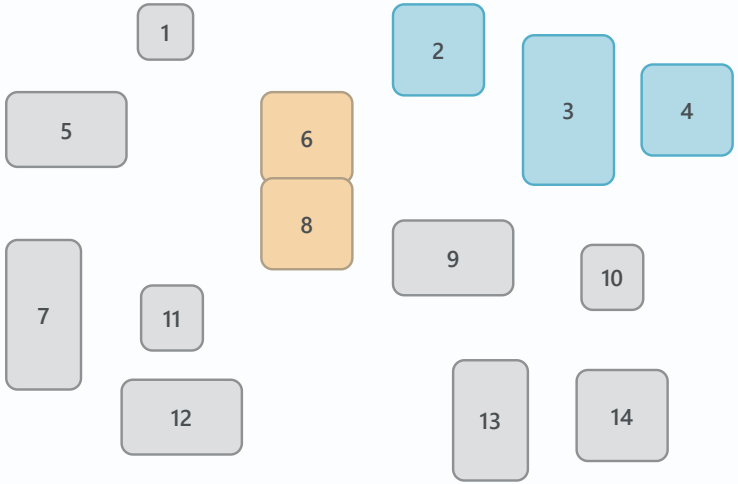
- Picture strong magnets: once two touch, they **never come apart**.
- As they merge, we want to keep track of which magnet is in which group.
- First, each magnet gets a **number**.
- A merge is a `union`; "same group?" compares two `find`s.



union(3, 4): 3 and 4 attract, forming the new group {3, 4}.

Magnet Example

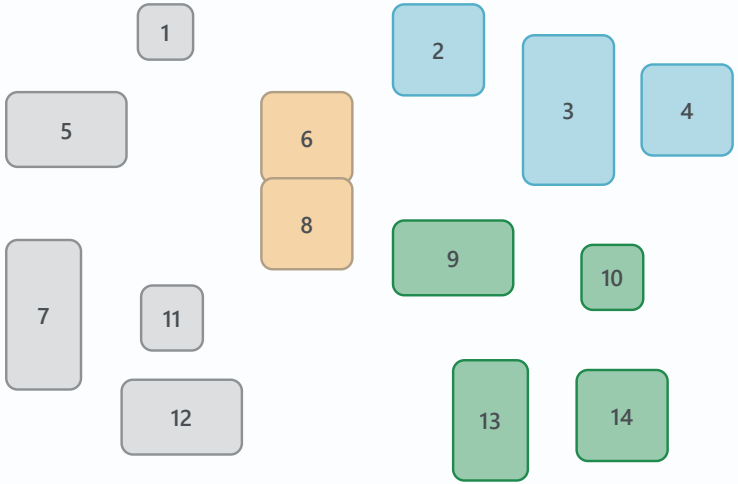
- Picture strong magnets: once two touch, they **never come apart**.
- As they merge, we want to keep track of which magnet is in which group.
- First, each magnet gets a **number**.
- A merge is a `union`; "same group?" compares two `find`s.



union(2, 3): 2 joins the group {3, 4} → {2, 3, 4}.

Magnet Example

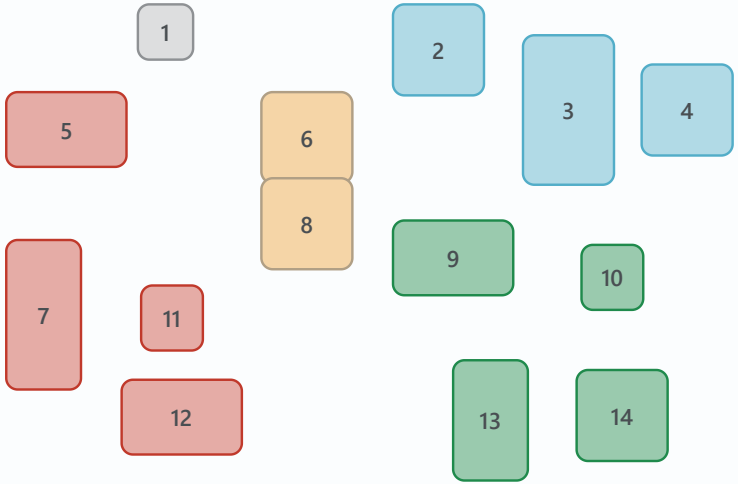
- Picture strong magnets: once two touch, they **never come apart**.
- As they merge, we want to keep track of which magnet is in which group.
- First, each magnet gets a **number**.
- A merge is a `union`; "same group?" compares two `find`s.



union(9, 10), union(13, 14), union(9, 13) → {9, 10, 13, 14}.

Magnet Example

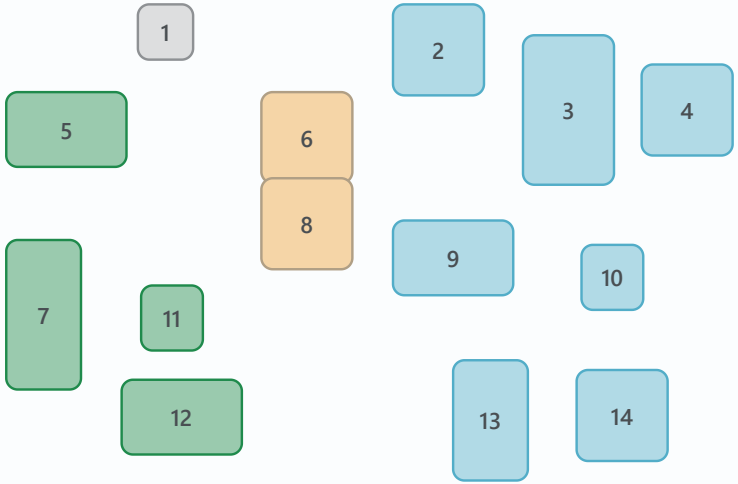
- Picture strong magnets: once two touch, they **never come apart**.
- As they merge, we want to keep track of which magnet is in which group.
- First, each magnet gets a **number**.
- A merge is a `union`; "same group?" compares two `find`s.



union(5, 7), union(7, 11), union(11, 12) → {5, 7, 11, 12}. Five groups remain (1 is alone).

Magnet Example

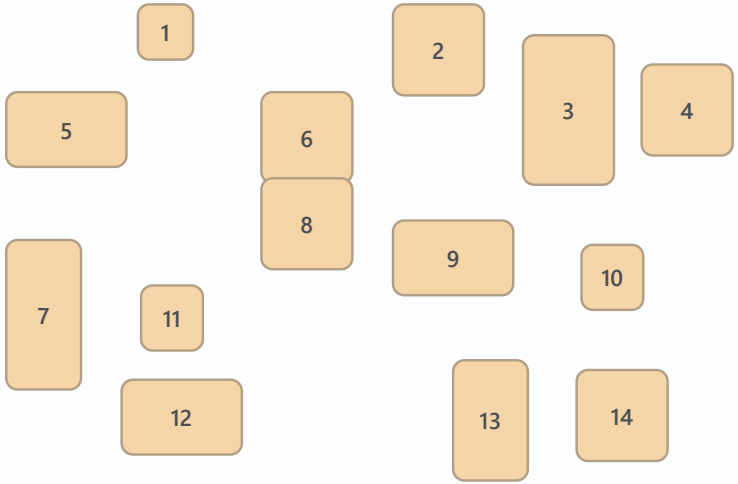
- Picture strong magnets: once two touch, they **never come apart**.
- As they merge, we want to keep track of which magnet is in which group.
- First, each magnet gets a **number**.
- A merge is a `union`; "same group?" compares two `find`s.



Are 4 and 10 in the same group? find(4) ≠ find(10) → no. **union(4, 10)** merges the two groups.

Magnet Example

- Picture strong magnets: once two touch, they **never come apart**.
- As they merge, we want to keep track of which magnet is in which group.
- First, each magnet gets a **number**.
- A merge is a `union`; "same group?" compares two `find`s.



The merging goes on until all magnets end up in **one group**: find(x) gives the same root for everyone.

The Find and Union Operations

- **find(x)** returns the name (root) of the set containing x by following parent links until it reaches the root.
- This is not the *search* of a search tree: no value is looked for; we simply climb to the root.
- **union(x, y)** replaces two sets with their union by attaching one root below the other.

The sets are disjoint to begin with, so no duplicate check is needed: only **one parent link** changes.

```
find(x):
  if x.parent != x:
    x.parent = find(x.parent)  // path compression
  return x.parent

union(x, y):
  root1 = find(x)
  root2 = find(y)
  if root1 != root2:
    // make the root of one set
    // a child of the other's root
    root1.parent = root2
```

Mapping Objects to Indices

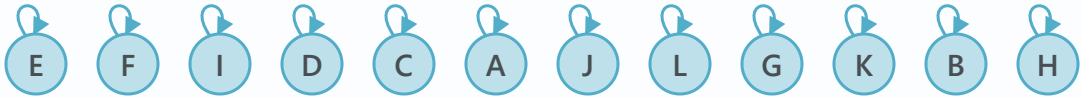
- Union-Find is most efficient on an **array**, with elements numbered 0..n−1.
- If the elements are letters, cities, people…, each object is first mapped to an index **arbitrarily (but permanently)**, and the mapping is stored in a hash table.
- Then `parent[i] = i` starts every element as the root of its own set (`makeSet`).

`E → 0`	`F → 1`	`I → 2`	`D → 3`	`C → 4`	`A → 5`
`J → 6`	`L → 7`	`G → 8`	`K → 9`	`B → 10`	`H → 11`

```
Map<Character, Integer> index = new HashMap<>();   // 'E'→0, 'F'→1, …
int[] parent = new int[12];
for (int i = 0; i < 12; i++) parent[i] = i;              // 12 singleton sets
```

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	0	1	2	3	4	5	6	7	8	9	10	11



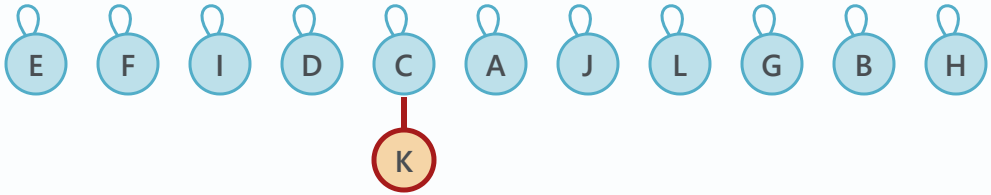
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

Start: 12 singleton sets, `parent[i] = i`. Rule: **the root of the smaller tree goes under the root of the larger**; on a tie, the second root goes under the first. No path compression.

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	0	1	2	3	4	5	6	7	8	4	10	11



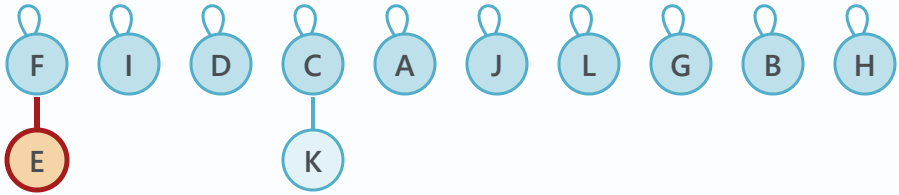
Operations

union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)

union(C, K): find(C) = C, find(K) = K; sizes equal (1) → **K** goes under C (parent[9] = 4).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	1	2	3	4	5	6	7	8	4	10	11



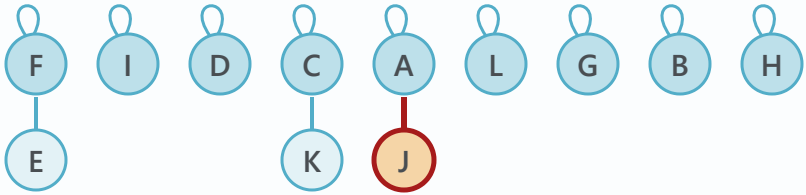
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(F, E): find(F) = F, find(E) = E; sizes equal (1) → **E** goes under F (parent[0] = 1).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	1	2	3	4	5	5	7	8	4	10	11



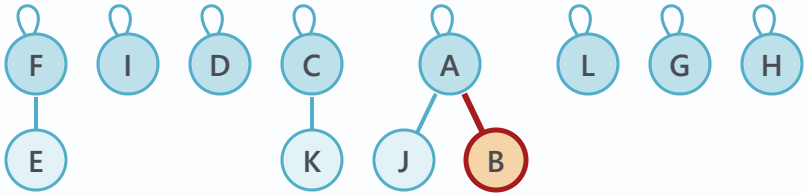
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(A, J): find(A) = A, find(J) = J; sizes equal (1) → **J** goes under A (parent[6] = 5).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	1	2	3	4	5	5	7	8	4	5	11



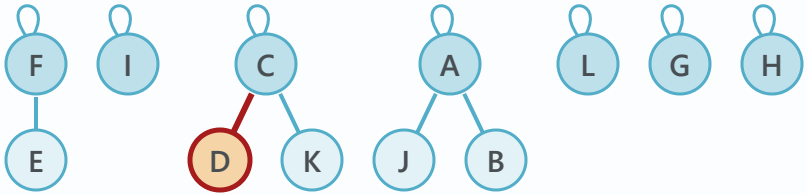
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(A, B): find(A) = A, find(B) = B; size 1 < 2 → **B** goes under A (parent[10] = 5).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	1	2	4	4	5	5	7	8	4	5	11



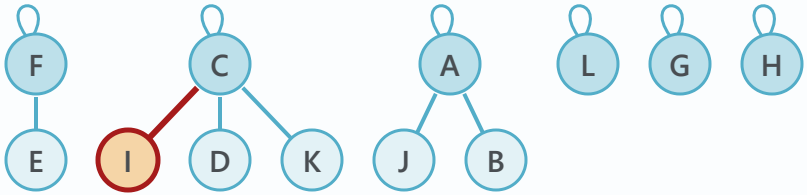
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(C, D): find(C) = C, find(D) = D; size 1 < 2 → **D** goes under C (parent[3] = 4).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	1	4	4	4	5	5	7	8	4	5	11



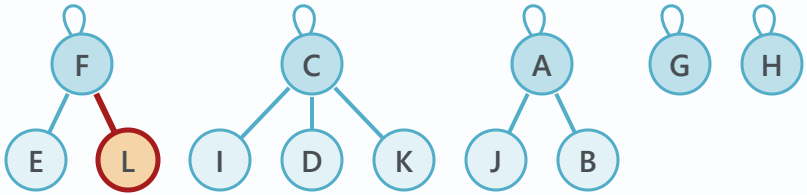
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(D, I): find(D) = C, find(I) = I; size 1 < 3 → **I** goes under C (parent[2] = 4).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	1	4	4	4	5	5	1	8	4	5	11



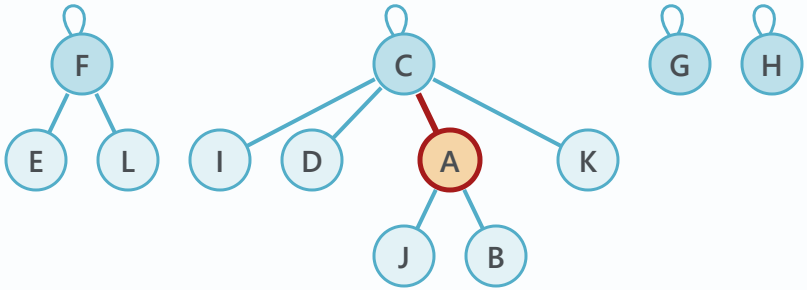
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(L, F): find(L) = L, find(F) = F; size 1 < 2 → **L** goes under F (parent[7] = 1).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	1	4	4	4	4	5	1	8	4	5	11



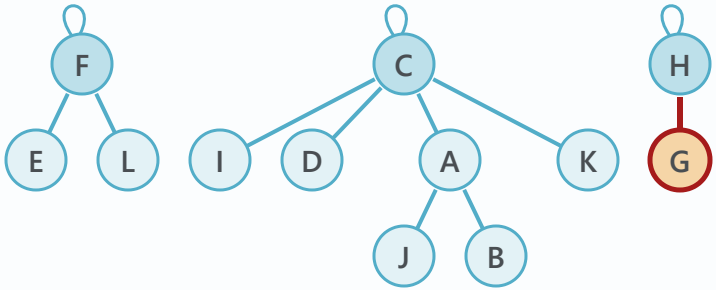
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(C, A): find(C) = C, find(A) = A; size 3 < 4 → **A** goes under C (parent[5] = 4).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	1	4	4	4	4	5	1	11	4	5	11



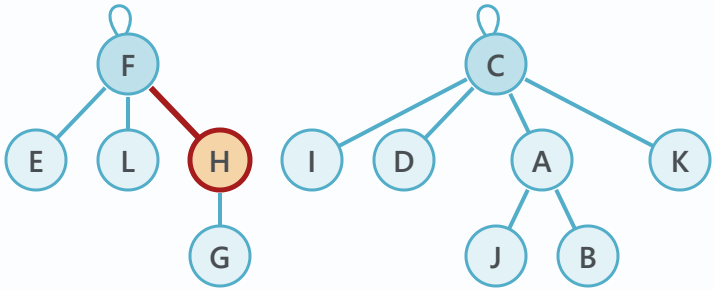
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(A, B): find(A) = find(B) = C → already in the same set; nothing changes. **union(H, G)**: find(H) = H, find(G) = G; sizes equal (1) → **G** goes under H (parent[8] = 11).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	1	4	4	4	4	5	1	11	4	5	1



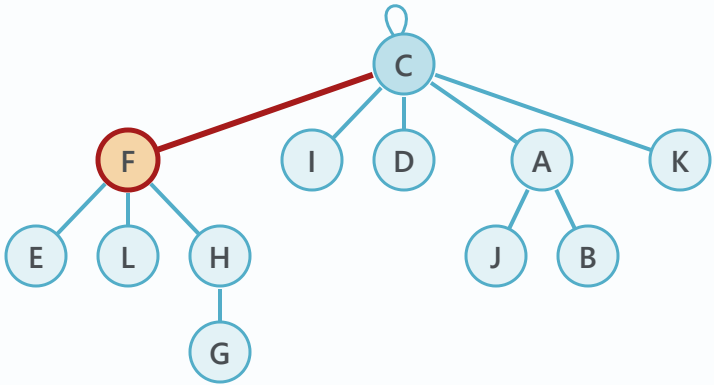
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(H, F): find(H) = H, find(F) = F; size 2 < 3 → **H** goes under F (parent[11] = 1).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	1	4	4	4	4	4	5	1	11	4	5	1



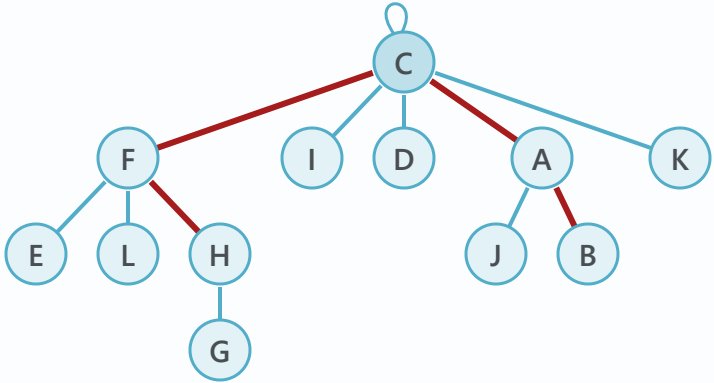
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

union(H, B): find(H) = F, find(B) = C; size 5 < 7 → **F** goes under C (parent[1] = 4).

The Union Operation Step by Step

element	E	F	I	D	C	A	J	L	G	K	B	H
index	0	1	2	3	4	5	6	7	8	9	10	11
parent	**1**	**4**	**4**	**4**	**4**	**4**	**5**	**1**	**11**	**4**	**5**	**1**



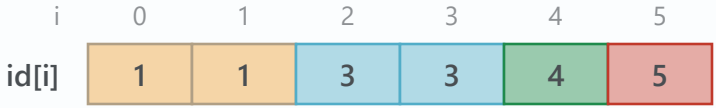
Operations

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

Result: one set with root **C**. Are H and B in the same set? find(H): H → F → C, find(B): B → A → C; same root → **yes** (4 steps in all).

Another Idea: Quick-Find

- A flat array instead of a tree: `id[i]` = the **set name** of i; elements of the same set share an id. **find(p)** = `id[p]` → O(1).
- **union(p, q)**: set *every* entry equal to `id[p]` to `id[q]` → the whole array is scanned, **O(n)**.

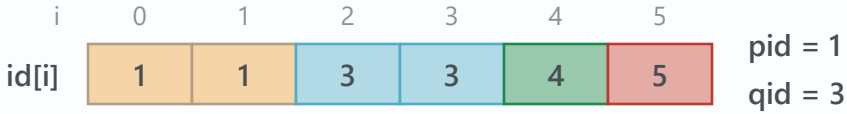


sets: {0, 1} {2, 3} {4} {5}

after union(0,1), union(2,3): 4 sets (same color = same set)

Method (Sedgewick)	find	union	total for n − 1 unions
Quick-Find (array, set name)	**O(1)**	O(n)	$O(n^2)$
Quick-Union (parent tree, naive)	O(h), worst case O(n)	O(h)	worst case $O(n^2)$
Rank/size + path compression	O(α(n)) amortized		≈ O(n)

Code Trace: Quick-Find `union`



sets: {0, 1} {2, 3} {4} {5}

union(1, 3): p's set name is **pid = id[1] = 1**, q's is **qid = id[3] = 3**.

Step 1 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
▶   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
6     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 1` `q = 3` `pid = 1` `qid = 3`

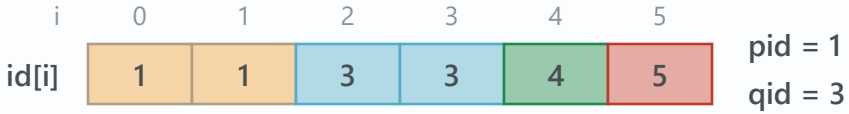
`id = [1, 1, 3, 3, 4, 5]`

CALL STACK

`union(1, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1} {2, 3} {4} {5}

pid ≠ qid: different sets, so they must be merged. Now the **whole array** is scanned.

Step 2 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
▶  if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
6     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 1` `q = 3` `pid = 1` `qid = 3`

`id = [1, 1, 3, 3, 4, 5]`

CALL STACK

`union(1, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 2, 3} {1} {4} {5}

id[0] = 1 = pid → set **id[0] = 3**.

Step 3 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
▶       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 0` `id = [3, 1, 3, 3, 4, 5]`

CALL STACK

`union(1, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

id[1] = 1 = pid → set **id[1] = 3**.

Step 4 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
▶       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 1` `id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(1, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

id[2] = 3 ≠ pid: left alone (but still examined).

Step 5 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 2` `id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(1, 3)`

`main()`

Code Trace: Quick-Find `union`



id[3] = 3 ≠ pid: left alone (but still examined).

Step 6 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 3` `id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(1, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

id[4] = 4 ≠ pid: left alone (but still examined).

Step 7 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 4` `id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(1, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

id[5] = 5 ≠ pid: left alone (but still examined).

Step 8 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 1` `q = 3` `pid = 1` `qid = 3`

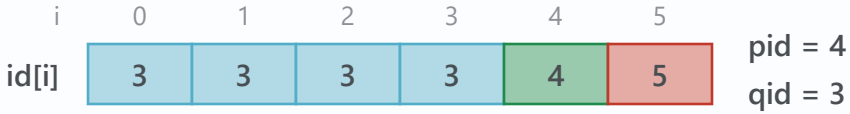
`i = 5` `id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(1, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

union(4, 3): p's set name is **pid = id[4] = 4**, q's is **qid = id[3] = 3**.

Step 9 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
▶   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
6     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 4` `q = 3` `pid = 4` `qid = 3`

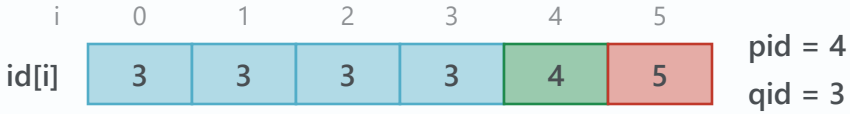
`id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(4, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

pid ≠ qid: different sets, so they must be merged. Now the **whole array** is scanned.

Step 10 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
▶   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
6     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 4` `q = 3` `pid = 4` `qid = 3`

`id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(4, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

id[0] = 3 ≠ pid: left alone (but still examined).

Step 11 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 4` `q = 3` `pid = 4` `qid = 3`

`i = 0` `id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(4, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

id[1] = 3 ≠ pid: left alone (but still examined).

Step 12 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 4` `q = 3` `pid = 4` `qid = 3`

`i = 1` `id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(4, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

id[2] = 3 ≠ pid: left alone (but still examined).

Step 13 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 4` `q = 3` `pid = 4` `qid = 3`

`i = 2` `id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(4, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3} {4} {5}

id[3] = 3 ≠ pid: left alone (but still examined).

Step 14 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 4` `q = 3` `pid = 4` `qid = 3`

`i = 3` `id = [3, 3, 3, 3, 4, 5]`

CALL STACK

`union(4, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3, 4} {5}

id[4] = 4 = pid → set **id[4] = 3**.

Step 15 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
▶       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 4` `q = 3` `pid = 4` `qid = 3`

`i = 4` `id = [3, 3, 3, 3, 3, 5]`

CALL STACK

`union(4, 3)`

`main()`

Code Trace: Quick-Find `union`



sets: {0, 1, 2, 3, 4} {5}

id[5] = 5 ≠ pid: left alone. Each union examined **all 6 cells**: O(n) for n elements; n − 1 unions → $O(n^2)$.

Step 16 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) scan
8 }
```

VARIABLES

`p = 4` `q = 3` `pid = 4` `qid = 3`

`i = 5` `id = [3, 3, 3, 3, 3, 5]`

CALL STACK

`union(4, 3)`

`main()`

Why Do We Need Improvements?

- find takes as many steps as the node's distance from the root. A bad sequence of unions turns the tree into a **chain** → **O(n)** in the worst case.

Naive: old root under new



height 0 · find(1) = 0 steps

By size: smaller under larger



height 0 · find(1) = 0 steps

Start: 5 singleton sets; each node is its own root. We will do union(1,2), union(2,3), union(3,4), union(4,5) in order.

Step 1 / 5

Two simple ideas keep the trees flat: **(1) union by size or rank** and **(2) path compression**. Together they give nearly constant time per operation: **O(α(n))**.

Why Do We Need Improvements?

- find takes as many steps as the node's distance from the root. A bad sequence of unions turns the tree into a **chain** → **O(n)** in the worst case.

Naive: old root under new



height 1 · find(1) = 1 step

By size: smaller under larger



height 1 · find(1) = 1 step

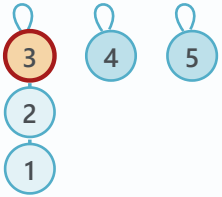
union(1, 2): two singleton sets; both methods put 1 under 2.

Two simple ideas keep the trees flat: **(1) union by size or rank** and **(2) path compression**. Together they give nearly constant time per operation: **O(α(n))**.

Why Do We Need Improvements?

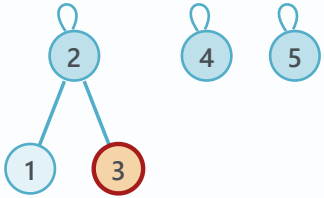
- find takes as many steps as the node's distance from the root. A bad sequence of unions turns the tree into a **chain** → **O(n)** in the worst case.

Naive: old root under new



height 2 · find(1) = 2 steps

By size: smaller under larger



height 1 · find(1) = 1 step

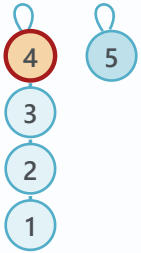
union(2, 3): the naive method puts root 2 (a 2-element tree) under the singleton 3 → the chain grows. By size: the singleton 3 goes under 2, the root of the larger tree.

Two simple ideas keep the trees flat: **(1) union by size or rank** and **(2) path compression**. Together they give nearly constant time per operation: **O(α(n))**.

Why Do We Need Improvements?

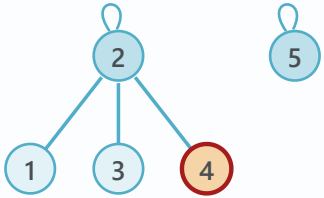
- find takes as many steps as the node's distance from the root. A bad sequence of unions turns the tree into a **chain** → **O(n)** in the worst case.

Naive: old root under new



height 3 · find(1) = 3 steps

By size: smaller under larger



height 1 · find(1) = 1 step

union(3, 4): the naive chain grows one more level; by size, 4 again joins the root.

Two simple ideas keep the trees flat: **(1) union by size or rank** and **(2) path compression**. Together they give nearly constant time per operation: **O(α(n))**.

Why Do We Need Improvements?

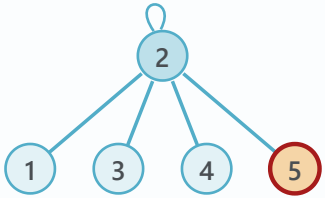
- find takes as many steps as the node's distance from the root. A bad sequence of unions turns the tree into a **chain** → **O(n)** in the worst case.

Naive: old root under new



height 4 · find(1) = 4 steps

By size: smaller under larger



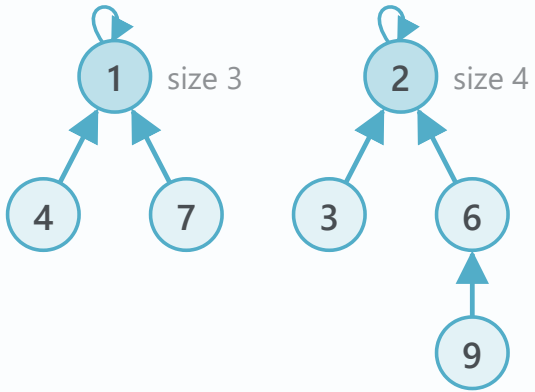
height 1 · find(1) = 1 step

union(4, 5): the naive tree is now a **chain** (height n − 1 → find is O(n)); the by-size tree is a **star** (height 1).

Two simple ideas keep the trees flat: **(1) union by size or rank** and **(2) path compression**. Together they give nearly constant time per operation: **O(α(n))**.

Union by Size

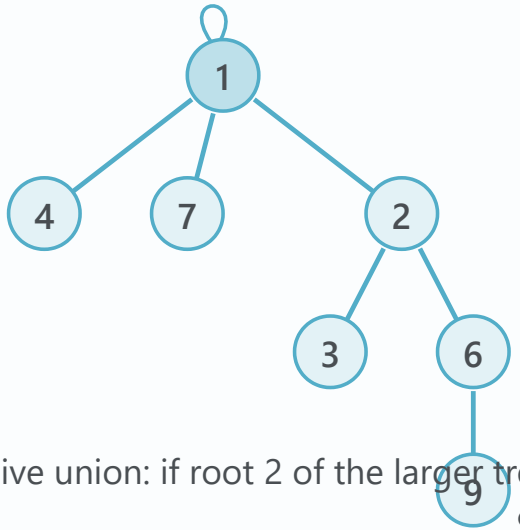
- Store each tree's **size** (number of elements) at its root.
- When merging, attach the **root of the smaller tree to the root of the larger**; the new size is the sum of the two.



Two sets: {1, 4, 7} (size 3) and {2, 3, 6, 9} (size 4). Heights 1 and 2.

Union by Size

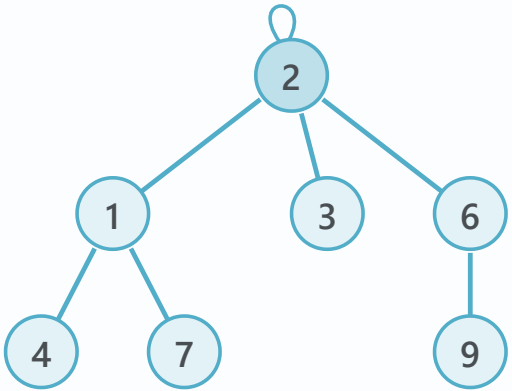
- Store each tree's **size** (number of elements) at its root.
- When merging, attach the **root of the smaller tree to the root of the larger**; the new size is the sum of the two.



Naive union: if root 2 of the larger tree goes under the smaller one, the height rises to **3** and 9's find gets longer.

Union by Size

- Store each tree's **size** (number of elements) at its root.
- When merging, attach the **root of the smaller tree to the root of the larger**; the new size is the sum of the two.



By size: root 1 of the smaller tree goes under root 2 of the larger. The height stays **2**, and size[2] = 3 + 4 = **7**.

Union by Rank

- The **rank** is an upper bound on the height; a new set has rank 0.
- The root of lower rank goes under the root of higher rank.
- If the ranks are equal, the new root's rank goes up by one.
- This rule keeps the height O(log n) even without path compression.

Path compression lowers the height but does not update the rank, so the rank is only an *upper bound*.

```
makeSet(x):
  parent[x] = x
  rank[x] = 0

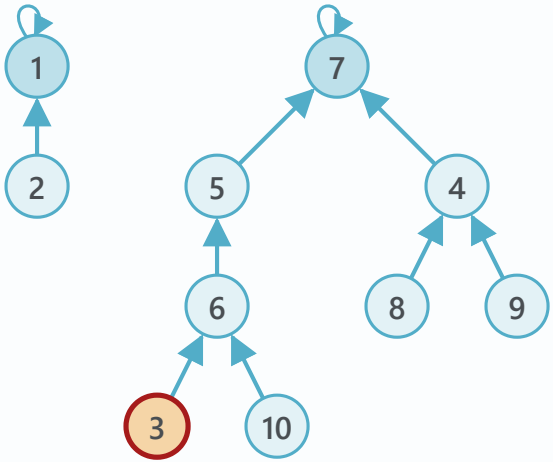
find(x):   // with path compression
  if x != parent[x]:
    parent[x] = find(parent[x])
  return parent[x]

link(x, y):   // x, y are roots
  if rank[x] > rank[y]:
    parent[y] = x
  else:
    parent[x] = y
    if rank[x] == rank[y]:
      rank[y] = rank[y] + 1

union(x, y):
  link(find(x), find(y))
```

Path Compression

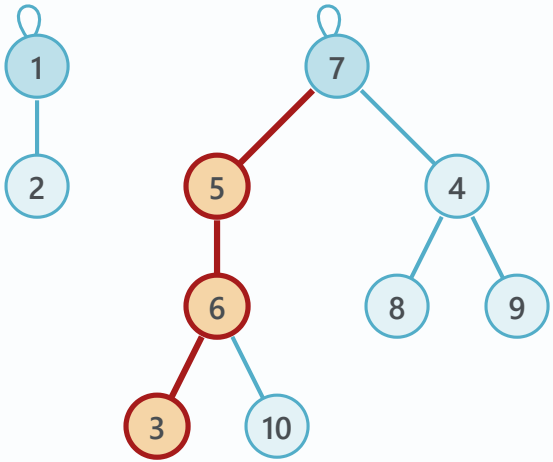
Path compression: during find, attach every node you pass **directly to the root**. Later finds finish in one step.



Start: 3's root is 7, but the path is 3 → 6 → 5 → 7 (3 steps).

Path Compression

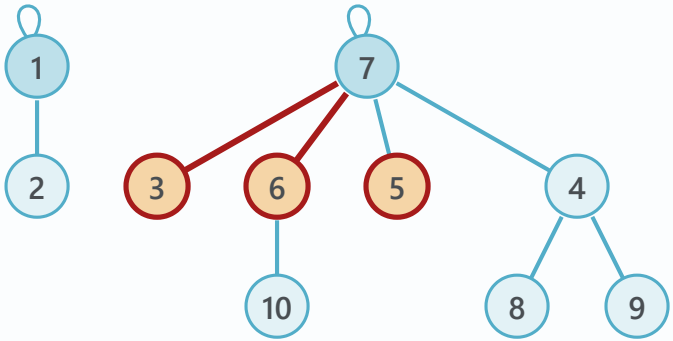
Path compression: during find, attach every node you pass **directly to the root**. Later finds finish in one step.



find(3): the recursion climbs 3 → 6 → 5 → 7 and finds the root, 7.

Path Compression

Path compression: during find, attach every node you pass **directly to the root**. Later finds finish in one step.



On the way back, each node's parent becomes 7: `parent[3] = parent[6] = 7` (5 was already on 7). Now find(3) and find(6) take one step, and 10's path is shorter too.

Union-Find Complexity

Implementation	find	union	Note
Naive (arbitrary linking)	O(n)	O(n)	the tree can become a chain
By size or rank	O(log n)	O(log n)	height ≤ $\log_2 n$
Path compression only	O(log n) amortized		for m operations
Rank + path compression	**O(α(n)) amortized**		constant in practice

α(n): the inverse Ackermann function

It grows extremely slowly: α(n) ≤ 4 even for as many elements as there are atoms in the universe.

Memory

Two integer arrays, `parent[n]` and `rank[n]` (or `size[n]`) → O(n).

Tarjan (1975): a sequence of m operations finishes in O(m · α(n)) time. Used in Kruskal's algorithm, connected components, image processing (region labeling), and network connectivity queries.

Java: Node-Based (Object) Union-Find

```
class SetNode {
  int data;
  int rank;   // bound on height
  SetNode parent;

  SetNode(int data) {
    this.data = data;
    this.parent = this;
    this.rank = 0;
  }
}

SetNode find(SetNode node) {
  if (node != node.parent)
    node.parent = find(node.parent); // compre
  return node.parent;
}
```

```
void union(SetNode x, SetNode y) {
  SetNode rootX = find(x);
  SetNode rootY = find(y);
  if (rootX == rootY) return;
  // union by rank
  if (rootX.rank < rootY.rank) {
    rootX.parent = rootY;
  } else if (rootX.rank > rootY.rank) {
    rootY.parent = rootX;
  } else { // equal: pick one
    rootX.parent = rootY;
    rootY.rank++;
  }
}
```

Java: Array-Based Union-Find

```java
class UnionFind {
  private final int[] parent, rank;
  UnionFind(int n) {
    parent = new int[n];  rank = new int[n];
    for (int i = 0; i < n; i++) parent[i] = i;
  }
  int find(int x) {
    if (parent[x] != x) parent[x] = find(parent[x]);
    return parent[x];
  }
  boolean union(int x, int y) {
    int rx = find(x), ry = find(y);
    if (rx == ry) return false;
    if (rank[rx] < rank[ry]) parent[rx] = ry;
    else if (rank[rx] > rank[ry]) parent[ry] = rx;
    else { parent[rx] = ry; rank[ry]++; }
    return true;
  }
}
```

Usage

```
UnionFind uf = new UnionFind(5);
uf.union(0, 1); uf.union(3, 4);
uf.find(1) == uf.find(0) → true
uf.find(1) == uf.find(3) → false
```

Why does `union` return boolean?

It returns true if a merge happened: in Kruskal's algorithm, that answers "does this edge form a cycle?"

Deep recursion

For very large n, find can be written with a loop instead.

Code Trace: `find` + `union`

```
▶ int find(int x) {
2   if (parent[x] != x) parent[x] = find(parent[x]);
3   return parent[x];
4 }
5 boolean union(int x, int y) {
6   int rx = find(x), ry = find(y);
7   if (rx == ry) return false;
8   if (rank[rx] < rank[ry]) parent[rx] = ry;
9   else if (rank[rx] > rank[ry]) parent[ry] = rx;
10  else { parent[rx] = ry; rank[ry]++; }
11  return true;
12 }
```

CALL STACK

`main()`



i	0	1	2	3	4	5
parent	0	1	2	3	4	5
rank	0	0	0	0	0	0

6 elements, `parent[i] = i`, `rank[i] = 0`: 6 singleton sets (each root points to itself).

Step 1 / 14

VARIABLES

—

Code Trace: `find` + `union`

```
 1 int find(int x) {
 2   if (parent[x] != x) parent[x] = find(parent[x]);
 3   return parent[x];
 4 }
 5 boolean union(int x, int y) {
 6   int rx = find(x), ry = find(y);
 7   if (rx == ry) return false;
 8   if (rank[rx] < rank[ry]) parent[rx] = ry;
 9   else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(0)`
- `union(0, 1)`
- `main()`



i	0	1	2	3	4	5
parent	0	1	2	3	4	5
rank	0	0	0	0	0	0

find(0): parent[0] == 0 → **0 is a root** and is returned.

Step 2 / 14

VARIABLES

`x = 0`

Code Trace: `find` + `union`

```
 1 int find(int x) {
 2   if (parent[x] != x) parent[x] = find(parent[x]);
 3   return parent[x];
 4 }
 5 boolean union(int x, int y) {
 6   int rx = find(x), ry = find(y);
 7   if (rx == ry) return false;
 8   if (rank[rx] < rank[ry]) parent[rx] = ry;
 9   else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(1)`
- `union(0, 1)`
- `main()`



i	0	1	2	3	4	5
parent	0	1	2	3	4	5
rank	0	0	0	0	0	0

find(1): parent[1] == 1 → **1 is a root** and is returned.

Step 3 / 14

VARIABLES

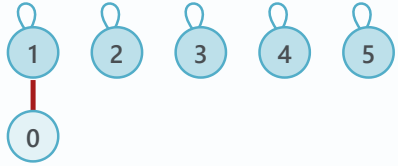
`x = 1`

Code Trace: `find` + `union`

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
▶    else { parent[rx] = ry; rank[ry]++; }
▶    return true;
12 }
```

CALL STACK

- `union(0, 1)`
- `main()`



i	0	1	2	3	4	5
parent	1	1	2	3	4	5
rank	0	1	0	0	0	0

Equal ranks (0) → parent[0] = 1, and the new root's rank goes up: **rank[1] = 1**.

Step 4 / 14

VARIABLES

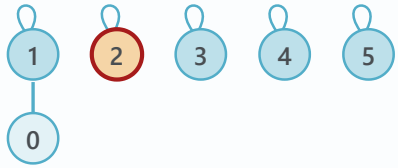
`x = 0` `y = 1` `rx = 0` `ry = 1` `returned = true`

Code Trace: `find` + `union`

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- **find(2)**
- union(2, 3)
- main()



i	0	1	2	3	4	5
parent	1	1	2	3	4	5
rank	0	1	0	0	0	0

find(2): parent[2] == 2 → **2 is a root** and is returned.

Step 5 / 14

VARIABLES

x = 2

Code Trace: `find` + `union`

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- **find(3)**
- union(2, 3)
- main()



i	0	1	2	3	4	5
parent	1	1	2	3	4	5
rank	0	1	0	0	0	0

find(3): parent[3] == 3 → **3 is a root** and is returned.

Step 6 / 14

VARIABLES

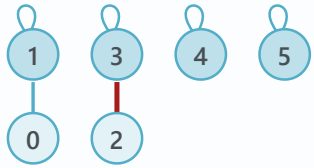
x = 3

Code Trace: `find` + `union`

```
 1 int find(int x) {
 2   if (parent[x] != x) parent[x] = find(parent[x]);
 3   return parent[x];
 4 }
 5 boolean union(int x, int y) {
 6   int rx = find(x), ry = find(y);
 7   if (rx == ry) return false;
 8   if (rank[rx] < rank[ry]) parent[rx] = ry;
 9   else if (rank[rx] > rank[ry]) parent[ry] = rx;
▶    else { parent[rx] = ry; rank[ry]++; }
▶    return true;
12 }
```

CALL STACK

- **union(2, 3)**
- main()



i	0	1	2	3	4	5
parent	1	1	3	3	4	5
rank	0	1	0	1	0	0

Equal ranks (0) → parent[2] = 3, and the new root's rank goes up: **rank[3] = 1**.

Step 7 / 14

VARIABLES

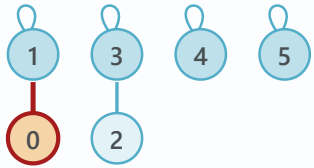
`x = 2` `y = 3` `rx = 2` `ry = 3` `returned = true`

Code Trace: `find` + `union`

```
 1 int find(int x) {
 2   if (parent[x] != x) parent[x] = find(parent[x]);
 3   return parent[x];
 4 }
 5 boolean union(int x, int y) {
 6   int rx = find(x), ry = find(y);
 7   if (rx == ry) return false;
 8   if (rank[rx] < rank[ry]) parent[rx] = ry;
 9   else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(0)`
- `union(0, 2)`
- `main()`



i	0	1	2	3	4	5
parent	1	1	3	3	4	5
rank	0	1	0	1	0	0

find(0): parent[0] = 1 ≠ 0 → not a root; climb with `find(1)`.

Step 8 / 14

VARIABLES

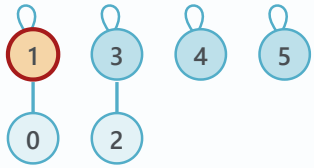
`x = 0`

Code Trace: `find` + `union`

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(1)`
- `find(0)`
- `union(0, 2)`
- `main()`



i	0	1	2	3	4	5
parent	1	1	3	3	4	5
rank	0	1	0	1	0	0

find(1): parent[1] == 1 → **1 is a root** and is returned.

Step 9 / 14

VARIABLES

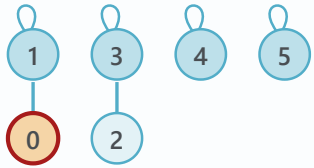
`x = 1`

Code Trace: `find` + `union`

```
 1 int find(int x) {
 2   if (parent[x] != x) parent[x] = find(parent[x]);
 3   return parent[x];
 4 }
 5 boolean union(int x, int y) {
 6   int rx = find(x), ry = find(y);
 7   if (rx == ry) return false;
 8   if (rank[rx] < rank[ry]) parent[rx] = ry;
 9   else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- **find(0)**
- union(0, 2)
- main()



i	0	1	2	3	4	5
parent	1	1	3	3	4	5
rank	0	1	0	1	0	0

On return, parent[0] = 1 (already a child of the root, unchanged); find(0) returns 1.

Step 10 / 14

VARIABLES

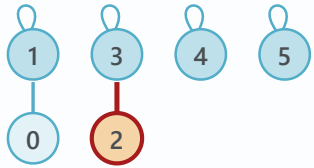
x = 0

Code Trace: `find` + `union`

```
1  int find(int x) {
▶    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(2)`
- `union(0, 2)`
- `main()`



i	0	1	2	3	4	5
parent	1	1	3	3	4	5
rank	0	1	0	1	0	0

find(2): parent[2] = 3 ≠ 2 → not a root; climb with `find(3)`.

Step 11 / 14

VARIABLES

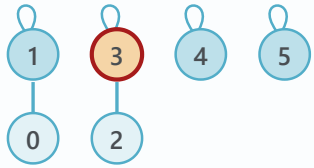
`x = 2`

Code Trace: `find` + `union`

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(3)`
- `find(2)`
- `union(0, 2)`
- `main()`



i	0	1	2	3	4	5
parent	1	1	3	3	4	5
rank	0	1	0	1	0	0

find(3): parent[3] == 3 → **3 is a root** and is returned.

Step 12 / 14

VARIABLES

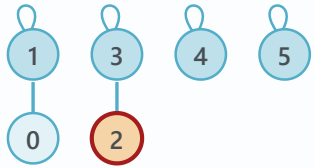
`x = 3`

Code Trace: `find` + `union`

```
 1 int find(int x) {
 2   if (parent[x] != x) parent[x] = find(parent[x]);
 3   return parent[x];
 4 }
 5 boolean union(int x, int y) {
 6   int rx = find(x), ry = find(y);
 7   if (rx == ry) return false;
 8   if (rank[rx] < rank[ry]) parent[rx] = ry;
 9   else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(2)`
- `union(0, 2)`
- `main()`



i	0	1	2	3	4	5
parent	1	1	3	3	4	5
rank	0	1	0	1	0	0

On return, parent[2] = 3 (already a child of the root, unchanged); find(2) returns 3.

Step 13 / 14

VARIABLES

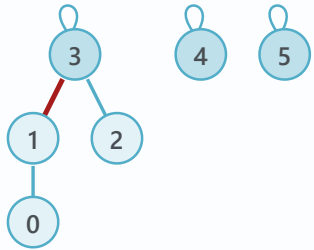
x = 2

Code Trace: `find` + `union`

```
 1 int find(int x) {
 2   if (parent[x] != x) parent[x] = find(parent[x]);
 3   return parent[x];
 4 }
 5 boolean union(int x, int y) {
 6   int rx = find(x), ry = find(y);
 7   if (rx == ry) return false;
 8   if (rank[rx] < rank[ry]) parent[rx] = ry;
 9   else if (rank[rx] > rank[ry]) parent[ry] = rx;
▶    else { parent[rx] = ry; rank[ry]++; }
▶    return true;
12 }
```

CALL STACK

- **union(0, 2)**
- main()



i	0	1	2	3	4	5
parent	1	3	3	3	4	5
rank	0	1	0	2	0	0

Equal ranks (1) → parent[1] = 3, and the new root's rank goes up: **rank[3] = 2**.

Step 14 / 14

VARIABLES

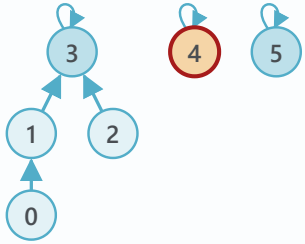
`x = 0` `y = 2` `rx = 1` `ry = 3` `returned = true`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- **find(4)**
- union(4, 0)
- main()



i	0	1	2	3	4	5
parent	1	3	3	3	4	5
rank	0	1	0	2	0	0

find(4): parent[4] == 4 → **4 is a root** and is returned.

Step 1 / 15

VARIABLES

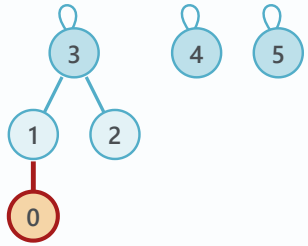
x = **4**

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
▶    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(0)`
- `union(4, 0)`
- `main()`



i	0	1	2	3	4	5
parent	1	3	3	3	4	5
rank	0	1	0	2	0	0

find(0): parent[0] = 1 ≠ 0 → not a root; climb with `find(1)`.

Step 2 / 15

VARIABLES

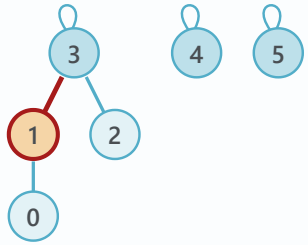
`x = 0`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
▶    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(1)`
- `find(0)`
- `union(4, 0)`
- `main()`



i	0	1	2	3	4	5
parent	1	3	3	3	4	5
rank	0	1	0	2	0	0

find(1): parent[1] = 3 ≠ 1 → not a root; climb with `find(3)`.

Step 3 / 15

VARIABLES

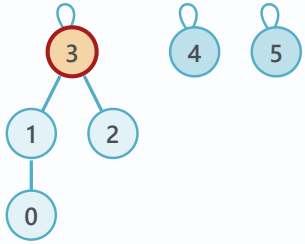
`x = 1`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(3)`
- `find(1)`
- `find(0)`
- `union(4, 0)`
- `main()`



i	0	1	2	3	4	5
parent	1	3	3	3	4	5
rank	0	1	0	2	0	0

find(3): parent[3] == 3 → **3 is a root** and is returned.

Step 4 / 15

VARIABLES

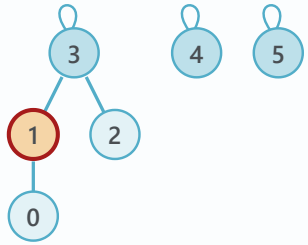
`x = 3`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
▶    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(1)`
- `find(0)`
- `union(4, 0)`
- `main()`



i	0	1	2	3	4	5
parent	1	3	3	3	4	5
rank	0	1	0	2	0	0

On return, parent[1] = 3 (already a child of the root, unchanged); find(1) returns 3.

Step 5 / 15

VARIABLES

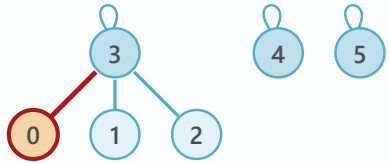
x = 1

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
▶    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(0)`
- `union(4, 0)`
- `main()`



i	0	1	2	3	4	5
parent	3	3	3	3	4	5
rank	0	1	0	2	0	0

Path compression: on return, parent[0] changes from 1 to **3**; 0 now links directly to the root. find(0) = 3.

Step 6 / 15

VARIABLES

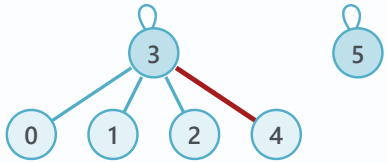
`x = 0`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
▶    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
▶    return true;
12 }
```

CALL STACK

- **union(4, 0)**
- main()



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

rank[4] = 0 < rank[3] = 2 → 4, the root of the shorter tree, goes under 3: parent[4] = 3. The height does not grow.

Step 7 / 15

VARIABLES

`x = 4` `y = 0` `rx = 4` `ry = 3`

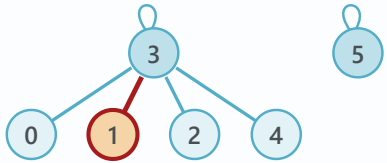
`returned = true`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
▶    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(1)`
- `union(1, 4)`
- `main()`



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

find(1): parent[1] = 3 ≠ 1 → not a root; climb with `find(3)`.

Step 8 / 15

VARIABLES

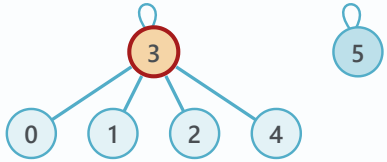
`x = 1`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- **find(3)**
- find(1)
- union(1, 4)
- main()



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

find(3): parent[3] == 3 → **3 is a root** and is returned.

Step 9 / 15

VARIABLES

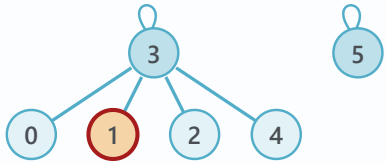
x = 3

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(1)`
- `union(1, 4)`
- `main()`



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

On return, parent[1] = 3 (already a child of the root, unchanged); find(1) returns 3.

Step 10 / 15

VARIABLES

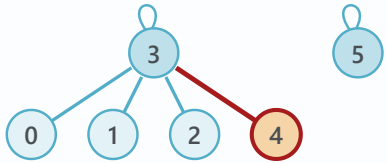
`x = 1`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(4)`
- `union(1, 4)`
- `main()`



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

find(4): parent[4] = 3 ≠ 4 → not a root; climb with `find(3)`.

Step 11 / 15

VARIABLES

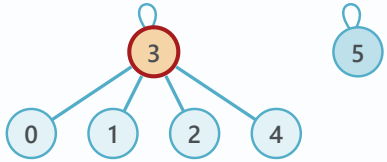
`x = 4`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- `find(3)`
- `find(4)`
- `union(1, 4)`
- `main()`



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

find(3): parent[3] == 3 → **3 is a root** and is returned.

Step 12 / 15

VARIABLES

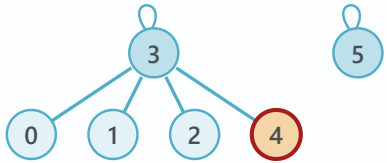
`x = 3`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

- **find(4)**
- union(1, 4)
- main()



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

On return, parent[4] = 3 (already a child of the root, unchanged); find(4) returns 3.

Step 13 / 15

VARIABLES

x = 4

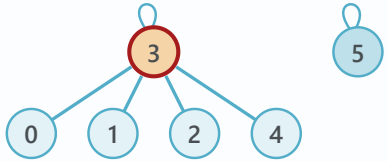
Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
3    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
▶    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

`union(1, 4)`

`main()`



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

rx = ry = 3 → 1 and 4 are **already in the same set**: `return false` (in Kruskal: a cycle).

Step 14 / 15

VARIABLES

`x = 1` `y = 4` `rx = 3` `ry = 3`

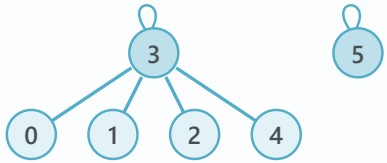
`returned = false`

Code Trace: `find` + `union` (cont.)

```
1  int find(int x) {
2    if (parent[x] != x) parent[x] = find(parent[x]);
▶    return parent[x];
4  }
5  boolean union(int x, int y) {
6    int rx = find(x), ry = find(y);
7    if (rx == ry) return false;
8    if (rank[rx] < rank[ry]) parent[rx] = ry;
9    else if (rank[rx] > rank[ry]) parent[ry] = rx;
10   else { parent[rx] = ry; rank[ry]++; }
11   return true;
12 }
```

CALL STACK

`main()`



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

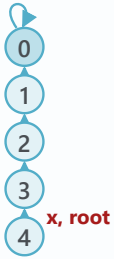
One set, root 3. Path compression moved 0 up to the root; rank[3] = 2 is only an upper bound. Together: ≈ **O(α(n))** per operation.

Step 15 / 15

VARIABLES

—

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	2	3

The chain 4 → 3 → 2 → 1 → 0 (the result of bad unions). find(4) first looks for the root: **root = 4**.

Step 1 / 16

```
 1 int find(int x) {
 2   int root = x;
 3   while (parent[root] != root) root = parent[root];
 4   while (parent[x] != root) {    // path compression
 5     int next = parent[x];
 6     parent[x] = root;
 7     x = next;
 8   }
 9   return root;
10 }
```

VARIABLES

`x = 4` `root = 4`

`parent = [0, 0, 1, 2, 3]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	2	3

Loop 1: parent[4] = 3 ≠ 4 → move up, **root = 3**.

Step 2 / 16

```
1  int find(int x) {
2    int root = x;
▶    while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 4` `root = 3`

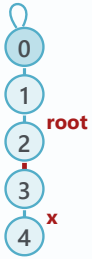
`parent = [0, 0, 1, 2, 3]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	2	3

Loop 1: parent[3] = 2 ≠ 3 → move up, **root = 2**.

Step 3 / 16

```
1  int find(int x) {
2    int root = x;
▶   while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 4` `root = 2`

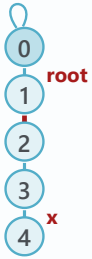
`parent = [0, 0, 1, 2, 3]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	2	3

Loop 1: parent[2] = 1 ≠ 2 → move up, **root = 1**.

Step 4 / 16

```
1  int find(int x) {
2    int root = x;
▶   while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 4` `root = 1`

`parent = [0, 0, 1, 2, 3]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	2	3

parent[1] = 0 → **root = 0**; since parent[0] = 0, the loop ends: root found (4 steps).

Step 5 / 16

```
1  int find(int x) {
2    int root = x;
▶    while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 4` `root = 0`

`parent = [0, 0, 1, 2, 3]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	2	3

Loop 2: parent[4] = 3 is not the root → this node will be compressed.

Step 6 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
▶    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 4` `root = 0`

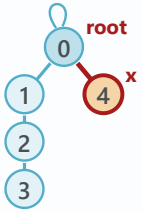
`parent = [0, 0, 1, 2, 3]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	2	0

First the old parent is saved (**next = 3**), then **parent[4] = 0**: 4 now links directly to the root.

Step 7 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
▶      int next = parent[x];
▶      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 4` `root = 0` `next = 3`

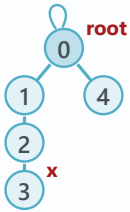
`parent = [0, 0, 1, 2, 0]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	2	0

One step up the path: **x = 3**.

Step 8 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
▶      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 3` `root = 0` `next = 3`

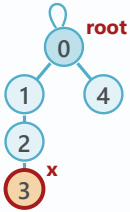
`parent = [0, 0, 1, 2, 0]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	2	0

Loop 2: parent[3] = 2 is not the root → this node will be compressed.

Step 9 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
▶    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 3` `root = 0`

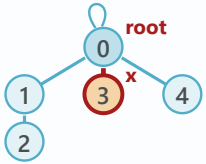
`parent = [0, 0, 1, 2, 0]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	0	0

First the old parent is saved (**next = 2**), then **parent[3] = 0**: 3 now links directly to the root.

Step 10 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
▶      int next = parent[x];
▶      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 3` `root = 0` `next = 2`

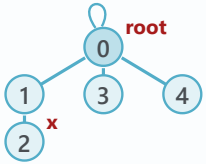
`parent = [0, 0, 1, 0, 0]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	0	0

One step up the path: **x = 2**.

Step 11 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
▶      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 2` `root = 0` `next = 2`

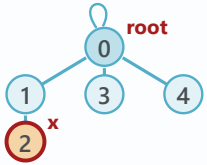
`parent = [0, 0, 1, 0, 0]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	1	0	0

Loop 2: parent[2] = 1 is not the root → this node will be compressed.

Step 12 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
▶    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 2` `root = 0`

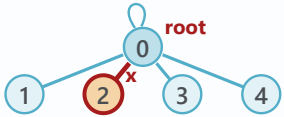
`parent = [0, 0, 1, 0, 0]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	0	0	0

First the old parent is saved (**next = 1**), then **parent[2] = 0**: 2 now links directly to the root.

Step 13 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
▶      int next = parent[x];
▶      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 2` `root = 0` `next = 1`

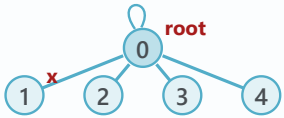
`parent = [0, 0, 0, 0, 0]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	0	0	0

One step up the path: **x = 1**.

Step 14 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
▶      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 1` `root = 0` `next = 1`

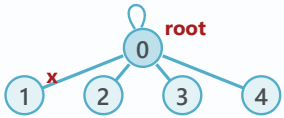
`parent = [0, 0, 0, 0, 0]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	0	0	0

parent[1] = 0 = root: 1 already links to the root, so the loop ends.

Step 15 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
▶    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
7      x = next;
8    }
9    return root;
10 }
```

VARIABLES

`x = 1` `root = 0`

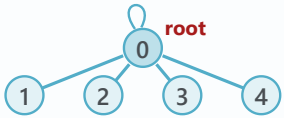
`parent = [0, 0, 0, 0, 0]`

CALL STACK

`find(4)`

`main()`

Code Trace: Iterative `find` with Path Compression



i	0	1	2	3	4
parent	0	0	0	0	0

The root **0** is returned. Every node on the path now links to the root, so later finds take 1 step. No recursion → no risk of stack overflow either.

Step 16 / 16

```
1  int find(int x) {
2    int root = x;
3    while (parent[root] != root) root = parent[root];
4    while (parent[x] != root) {    // path compression
5      int next = parent[x];
6      parent[x] = root;
7      x = next;
8    }
▶    return root;
10 }
```

VARIABLES

`x = 1` `root = 0`

`parent = [0, 0, 0, 0, 0]`

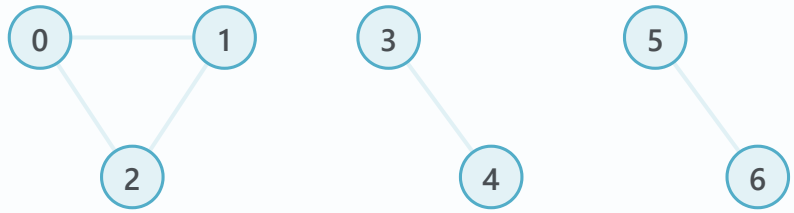
`returned = 0`

CALL STACK

`find(4)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	0	1	2	3	4	5	6
rank	0	0	0	0	0	0	0

count = 7

7 nodes, 5 edges. `UnionFind(7)` : each node is the root of its own set (parent[i] = i).

Step 1 / 12

```
1  int countComponents(int n, int[][] edges) {
▶    UnionFind uf = new UnionFind(n);
3    int count = n;          // n singleton sets
4    for (int[] e : edges)
5      if (uf.union(e[0], e[1]))
6        count--;            // two components merged
7    return count;
8  }
```

VARIABLES

`n = 7`

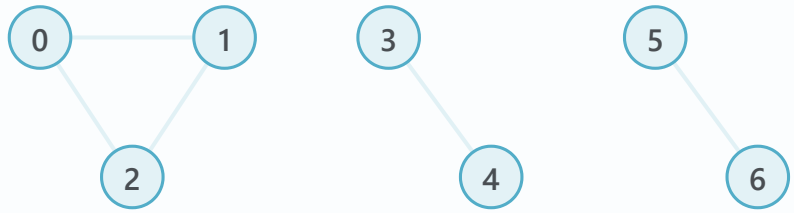
`edges = [[0,1], [1,2], [3,4], [0,2], [5,6]]`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	0	1	2	3	4	5	6
rank	0	0	0	0	0	0	0

count = 7

No edges yet: **7 components**.

Step 2 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
▶   int count = n;          // n singleton sets
4   for (int[] e : edges)
5     if (uf.union(e[0], e[1]))
6       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

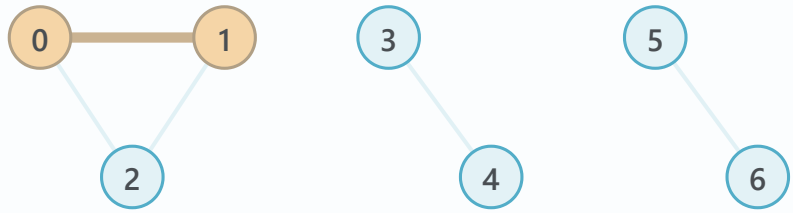
`n = 7` `count = 7`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	1	1	2	3	4	5	6
rank	0	1	0	0	0	0	0

count = 7

Edge (0, 1): different roots → `union` merges the sets and returns **true**.

Step 3 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
▶   for (int[] e : edges)
▶     if (uf.union(e[0], e[1]))
6       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

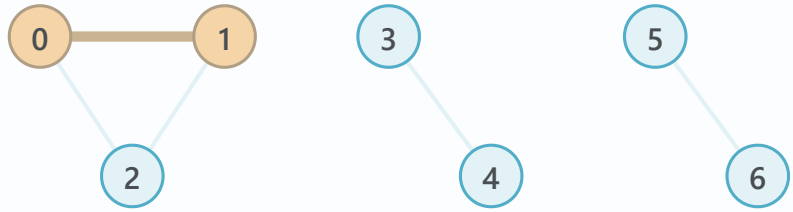
`n = 7` `count = 7` `e = [0, 1]` `union = true`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	1	1	2	3	4	5	6
rank	0	1	0	0	0	0	0

count = 6

Two components became one: **count = 6**.

Step 4 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
4   for (int[] e : edges)
5     if (uf.union(e[0], e[1]))
▶       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

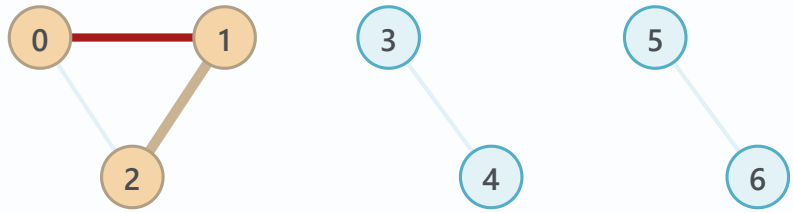
`n = 7` `count = 6` `e = [0, 1]` `union = true`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	1	1	1	3	4	5	6
rank	0	1	0	0	0	0	0

count = 6

Edge (1, 2): different roots → `union` merges the sets and returns **true**.

Step 5 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
▶   for (int[] e : edges)
▶     if (uf.union(e[0], e[1]))
6       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

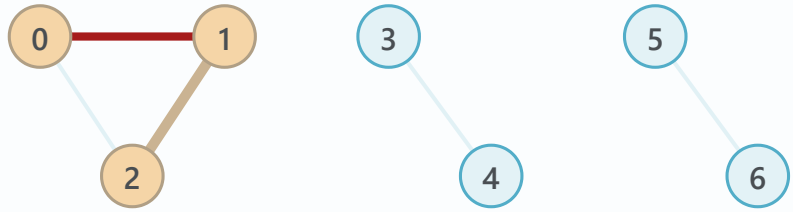
`n = 7` `count = 6` `e = [1, 2]` `union = true`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	1	1	1	3	4	5	6
rank	0	1	0	0	0	0	0

count = 5

Two components became one: **count = 5**.

Step 6 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
4   for (int[] e : edges)
5     if (uf.union(e[0], e[1]))
▶       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

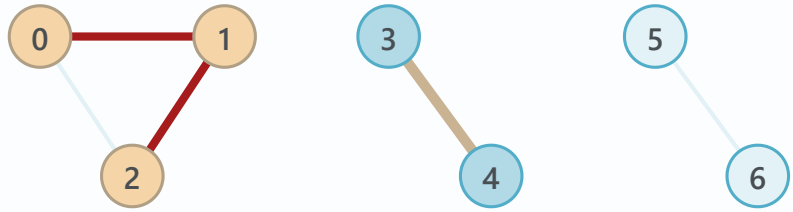
`n = 7` `count = 5` `e = [1, 2]` `union = true`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	1	1	1	4	4	5	6
rank	0	1	0	0	1	0	0

count = 5

Edge (3, 4): different roots → `union` merges the sets and returns **true**.

Step 7 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
▶   for (int[] e : edges)
▶     if (uf.union(e[0], e[1]))
6       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

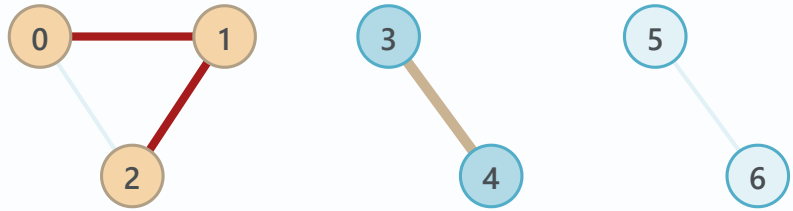
`n = 7` `count = 5` `e = [3, 4]` `union = true`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	1	1	1	4	4	5	6
rank	0	1	0	0	1	0	0

count = 4

Two components became one: **count = 4**.

Step 8 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
4   for (int[] e : edges)
5     if (uf.union(e[0], e[1]))
▶       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

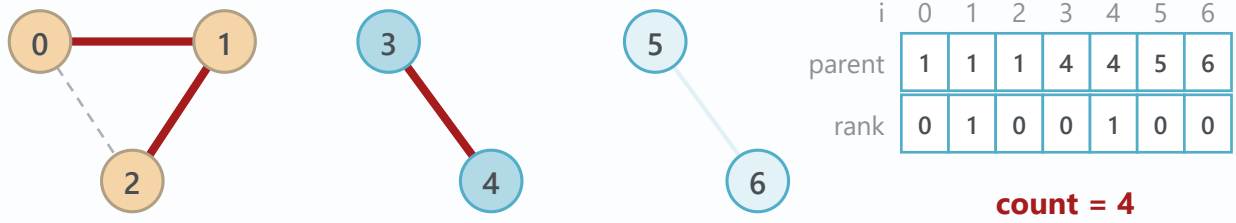
`n = 7` `count = 4` `e = [3, 4]` `union = true`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



Edge (0, 2): find(0) = find(2) → already in the **same component**; `union` returns false and count is unchanged (this edge closes a cycle).

Step 9 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
▶   for (int[] e : edges)
▶     if (uf.union(e[0], e[1]))
6       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

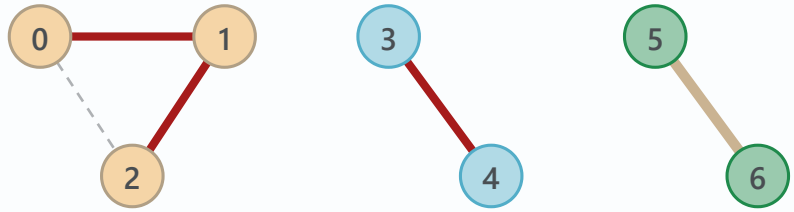
`n = 7` `count = 4` `e = [0, 2]` `union = false`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	1	1	1	4	4	6	6
rank	0	1	0	0	1	0	1

count = 4

Edge (5, 6): different roots → `union` merges the sets and returns **true**.

Step 10 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
▶   for (int[] e : edges)
▶     if (uf.union(e[0], e[1]))
6       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

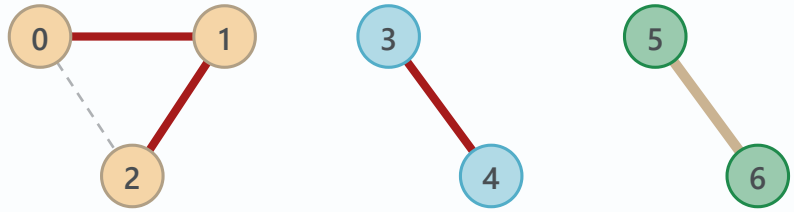
`n = 7` `count = 4` `e = [5, 6]` `union = true`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



i	0	1	2	3	4	5	6
parent	1	1	1	4	4	6	6
rank	0	1	0	0	1	0	1

count = 3

Two components became one: **count = 3**.

Step 11 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
4   for (int[] e : edges)
5     if (uf.union(e[0], e[1]))
▶       count--;            // two components merged
7   return count;
8 }
```

VARIABLES

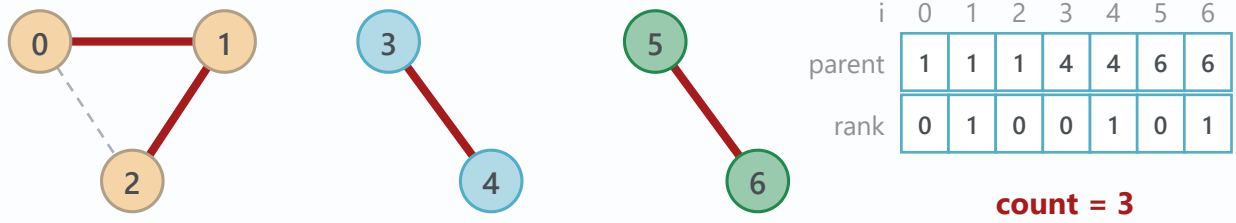
`n = 7` `count = 3` `e = [5, 6]` `union = true`

CALL STACK

`countComponents(7, edges)`

`main()`

Code Trace: Counting Connected Components with Union-Find



Result: **3 components**: {0, 1, 2}, {3, 4}, {5, 6}. Total O(n + m·α(n)); even if edges arrive later, the count is always current (dynamic connectivity).

Step 12 / 12

```
1 int countComponents(int n, int[][] edges) {
2   UnionFind uf = new UnionFind(n);
3   int count = n;          // n singleton sets
4   for (int[] e : edges)
5     if (uf.union(e[0], e[1]))
6       count--;            // two components merged
▶   return count;
8 }
```

VARIABLES

n = 7 | count = 3 | returned = 3

CALL STACK

countComponents(7, edges)

main()

SUMMARY

Common Mistakes

1. Changing an element inside a set

An object whose hashCode changes after insertion stays in the wrong bucket, so `contains` returns false.

```
p.x = 5; set.contains(p) → false
```

2. equals without hashCode

Two "equal" objects will probably land in different buckets, so the HashSet holds them **twice**. Override both together.

3. Removing while iterating

`set.remove(x)` inside for-each → `ConcurrentModificationException`.

```
set.removeIf(x -> x % 2 == 0);
```

4. Linking elements instead of roots

`parent[x] = y` is wrong: it splits x's tree. Correct:

```
parent[find(x)] = find(y);
```

5. Forgetting to bump equal ranks

Without `rank[ry]++`, rank no longer tracks height and the O(log n) guarantee is lost.

6. A compare inconsistent with TreeSet's needs

Of distinct objects with `compare(a, b) == 0`, only one is kept (e.g. when sorting by surname alone).

Lab: Set Operations

Sorted union and intersection: {1, 3, 5, 7, 9} and {3, 4, 5, 8}

Input (sorted a | sorted b) `1, 3, 5, 7, 9 | 3, 4, 5, 8`

CODE · **step 1 / 22**

```
1 void unionIntersection(int[] a, int[] b,
2     List<Integer> union, List<Integer> inte…
3   int i = 0, j = 0;
4   while (i < a.length && j < b.length) {
5     if (a[i] < b[j]) union.add(a[i++]);
6     else if (a[i] > b[j]) union.add(b[j++]);
7     else {                        // in both
8       union.add(a[i]);  inter.add(a[i]);
9       i++;  j++;
10    }
11  }
12  while (i < a.length) union.add(a[i++]);
13  while (j < b.length) union.add(b[j++]);
14 }
```

a 1 3 5 7 9

b 3 4 5 8

union empty

inter empty

RESULT –

Called with the sorted arrays a = {1, 3, 5, 7, 9} and b = {3, 4, 5, 8}.

SUMMARY

Summary

Structure	Core operations	Cost	When?
HashSet	add, remove, contains	O(1)*	order not needed
TreeSet	+ first, floor, range	O(log n)	sorted access needed
Multisets	add, count, remove	O(1)*	repeat counts matter
Union-Find	find, union	O(α(n))	disjoint sets, connectivity queries

* average case. Union-Find cost is amortized, with rank + path compression.

Choice: `HashSet` for membership, `TreeSet` for sorted access, Union-Find for dynamic connectivity.

SUMMARY

Interactive Simulators

Set ADT

Adding and removing elements, membership tests, union, intersection, and difference.

Disjoint Set Union (Union-Find)

find and union, union by rank, and path compression, shown on trees.

SUMMARY

Sample Problems

1. For A = {2, 4, 6, 8} and B = {1, 2, 3, 4}, what are A ∪ B, A ∩ B, and B − A?
2. Which Set implementation would you choose to print the words of a dictionary in alphabetical order, without duplicates?
3. Store the letters of "kayak" in a multiset. What do count('a') and size() return?
4. On 0..5 we do union(0,1), union(2,3), union(1,3), union(4,5) in order (by rank; on a tie, parent[x] = y). How many sets remain? What is find(0)?
5. On 0..4 we do union(0,1), union(1,2), union(3,4), union(2,0) in order. Which union returns false? How many connected components remain?

▶ **Show answers**

SUMMARY

Exercises

1. 12, 28, 44, 7 are added to an empty HashSet (16 buckets, bucket = x & 15). Which buckets? What does `add(28)` return?
2. U = {0..7}: A = 1011 0010_2, B = 0111 0100_2. What sets are A | B, A & B, and A & ~B?
3. With Quick-Find, n − 1 unions on n elements: how many cells are examined in total?
4. With `parent = [1, 2, 3, 3, 3]`, what is `parent` after an iterative `find(0)`?
5. In union by rank, how large can a rank get? Why?
6. Edges arrive one at a time; after each one: "is the graph connected?" Which structure, at what cost?

▶ **Answers / hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022.
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011.
- Oracle, Java SE 17 API documentation for `Set`, `HashSet`, and `TreeSet`.

The End

Chapter 13: Sets and Disjoint Sets

Next deck: Chapter 14: Graphs · Study questions · Simulators: Set ADT Simulator, Disjoint Sets Simulator · Glossary