Chapter 16: Sets and Union-Find

Data Structures and Algorithms with C

Disjoint sets · Union and find · The parent array · Worst case $O(n^2)$ · Weighting rule · Collapsing rule · Inverse Ackermann · Components · Kruskal

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_16_Sets_and_Union_Find.html

Learning Outcomes

Model

Represent pairwise disjoint sets as trees whose links point from child to parent.

Implement

Write `find` and `union` with a single `parent` array.

Find the worst case

Show which sequence of operations degenerates the tree and costs $O(n^2)$.

Improve

Apply the weighting rule and the collapsing rule; interpret the lemmas.

Analyze

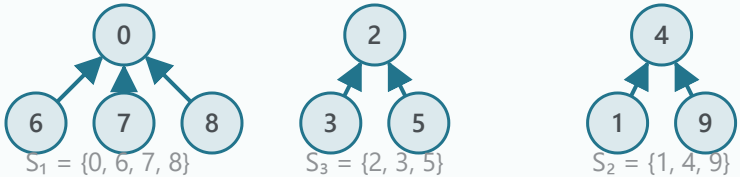
Explain the Tarjan–van Leeuwen result and the inverse Ackermann function.

Use

Apply union-find to counting connected components and to Kruskal's algorithm.

Union and Find on Sets

The universe is {0, 1, …, n−1} and the sets are **pairwise disjoint**: $S_1 \cap S_2 = S_2 \cap S_3 = S_1 \cap S_3 = \emptyset$. For n = 10:



Each set is a tree; links point **from children to parents** (arrows go up).

Instructor *"A very, very basic data structure, yet we use it in many problems. You have to learn it to be able to use it."*

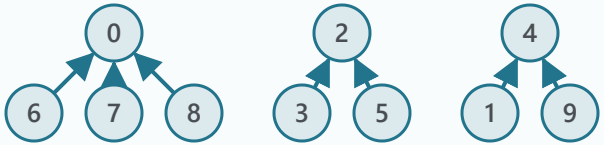
Two Operations, and Two Ways to Unite

1 · Disjoint set union

$S_i \cup S_j = \{x : x \in S_i \vee x \in S_j\}$. After a union the two sets no longer exist separately: $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$.

2 · find(i)

Returns the set that contains i (the set's name = the root of its tree).



The roots are 0, 4 and 2. To unite two sets, make one root's parent link point to the other root.

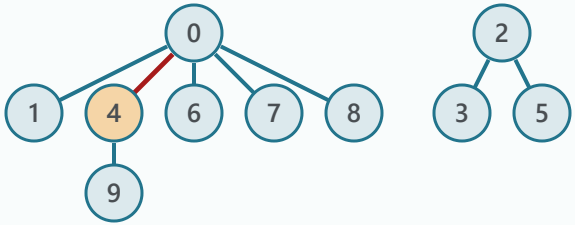
Two Operations, and Two Ways to Unite

1 · Disjoint set union

$S_i \cup S_j = \{x : x \in S_i \vee x \in S_j\}$. After a union the two sets no longer exist separately: $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$.

2 · find(i)

Returns the set that contains i (the set's name = the root of its tree).



$S_1 \cup S_2$: 4's parent becomes 0; the tree of S_2 becomes a subtree of 0.

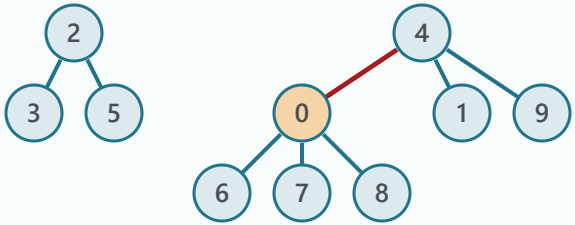
Two Operations, and Two Ways to Unite

1 · Disjoint set union

$S_i \cup S_j = \{x : x \in S_i \vee x \in S_j\}$. After a union the two sets no longer exist separately: $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$.

2 · find(i)

Returns the set that contains i (the set's name = the root of its tree).



$S_2 \cup S_1$: this time 0's parent becomes 4. Both are correct; the choice affects the cost.

A Set Is Named by Its Root

Set name	link (root)
S_1	→ 0
S_2	→ 4
S_3	→ 2

If the names are needed as text, a separate table keeps a link to each root.

For union and find we identify each set with the **root** of its tree: instead of "set S_1" we say "**set 0**".

So `find(8) == find(6)` asks "are 8 and 6 in the same set?"

Instructor *"First we approach the problem like a mathematician, like someone who cannot program; before anything else we must state it correctly in the language of graph theory."*

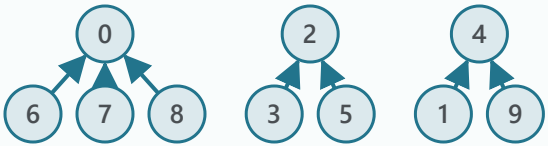
How Should We Implement It?

Method	Union	Find	Problem
Linked list	O(1)	O(n)	slow search
Keep it sorted	—	—	sorting is O(n lg n)
parent array (tree)	O(1)	tree depth	depth must stay small

Instructor *"Is there a way to do it with the array, the most reliable, most robust and fastest data structure we have?"*

The nodes are numbered 0 to n−1, so we use the **number as an index**: each node needs just one field, its parent.

The `parent` Array: One Link per Index



parent[i] = parent of i; −1 at a root

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

- The parent of 1 and 9 is **4**; of 3 and 5, **2**; of 6, 7 and 8, **0**.
- 0, 2 and 4 are roots: `parent[root] = −1` (negative = root).

```
#define N 10
int parent[N] = { -1, 4, -1, 2, -1, 2, 0, 0, 0, 4 };
```

First Version: `find1` and `union1`

```
int find1(int i) {
  for (; parent[i] >= 0; i = parent[i])
    ;                  /* climb to a root */
  return i;
}
void union1(int i, int j) {
  parent[i] = j;       /* put tree i under j */
}
```

find(i)

Follows parent links until it reaches a negative entry (the root) and returns the root.

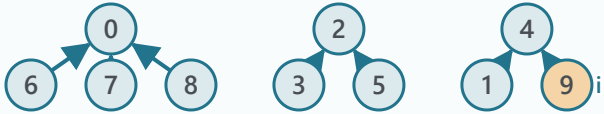
union(i, j)

Given two roots, makes one tree a subtree of the other: O(1).

Precondition

i and j must be **roots**; otherwise call `find1` first.

Code Trace: `find1(9)`, `union1(4, 0)`



find1(9)

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

find1(9): which set is 9 in?

Step 1 / 9

```
▶ int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

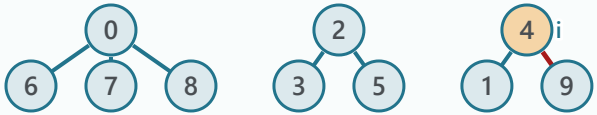
`i = 9`

CALL STACK

`find1(9)`

`main()`

Code Trace: `find1(9)`, `union1(4, 0)`



find1(9)

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

parent[9] = 4 ≥ 0 → i = parent[i] = **4**.

Step 2 / 9

```
1 int find1(int i) {
▶   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

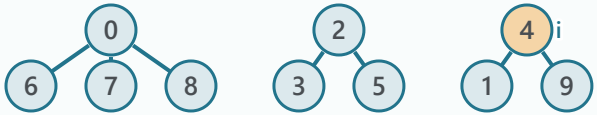
`i = 9` `parent[i] = 4`

CALL STACK

`find1(9)`

`main()`

Code Trace: `find1(9)`, `union1(4, 0)`



find1(9)

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

parent[4] = −1 < 0: a root. **return 4** → 9 is in set 4.

Step 3 / 9

```
1 int find1(int i) {
▶   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
▶   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;      /* put tree i under j */
8 }
```

VARIABLES

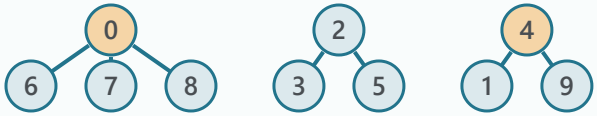
`i = 4` `parent[i] = -1`

CALL STACK

`find1(9)`

`main()`

Code Trace: `find1(9)`, `union1(4, 0)`



union1(4, 0)

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

union1(4, 0): two roots are passed.

Step 4 / 9

```
1 int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
4   return i;
5 }
▶ void union1(int i, int j) {
7   parent[i] = j;      /* put tree i under j */
8 }
```

VARIABLES

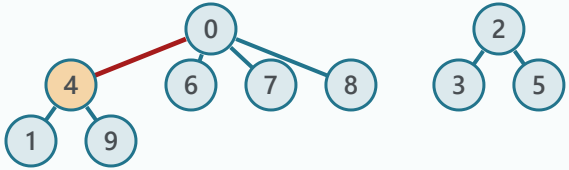
`i = 4` `j = 0`

CALL STACK

`union1(4, 0)`

`main()`

Code Trace: `find1(9)`, `union1(4, 0)`



union1(4, 0)

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

parent[4] = 0: tree 4 now hangs under 0. One assignment: O(1).

Step 5 / 9

```
1 int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
▶   parent[i] = j;      /* put tree i under j */
8 }
```

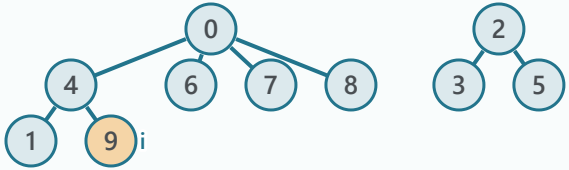
VARIABLES

`i = 4` `j = 0` `parent[4] = 0`

CALL STACK

- `union1(4, 0)`
- `main()`

Code Trace: `find1(9)`, `union1(4, 0)`



find1(9)

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

find1(9) again: the path is one node longer.

Step 6 / 9

```
▶ int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

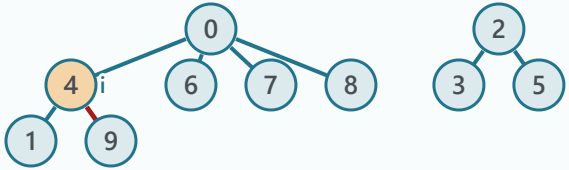
`i = 9`

CALL STACK

`find1(9)`

`main()`

Code Trace: `find1(9)`, `union1(4, 0)`



find1(9)

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

parent[9] = 4 ≥ 0 → i = **4**.

Step 7 / 9

```
1 int find1(int i) {
▶   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

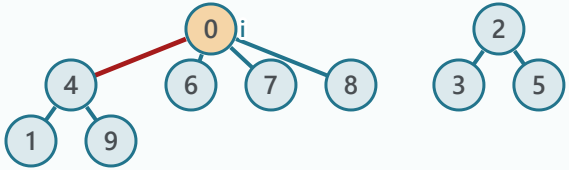
`i = 9` `parent[i] = 4`

CALL STACK

`find1(9)`

`main()`

Code Trace: `find1(9)`, `union1(4, 0)`



find1(9)

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

parent[4] = 0 ≥ 0 → i = **0**.

Step 8 / 9

```
1 int find1(int i) {
▶   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

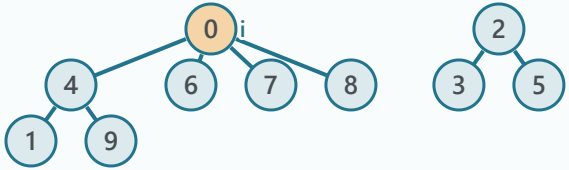
`i = 4` `parent[i] = 0`

CALL STACK

`find1(9)`

`main()`

Code Trace: `find1(9)`, `union1(4, 0)`



find1(9)

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

return 0: 9 is now in set 0. Cost = depth of the node.

Step 9 / 9

```
1 int find1(int i) {
▶   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
▶   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

`i = 0` `parent[i] = -1`

CALL STACK

- `find1(9)`
- `main()`

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



initial

	0	1	2	3	4
parent	-1	-1	-1	-1	-1

n = 5 singleton sets S_i = {i}, all with `parent[i] = -1`.

Step 1 / 11

```
▶ int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;      /* put tree i under j */
8 }
```

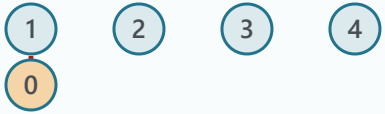
VARIABLES

n = **5**

CALL STACK

main()

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



union1(0, 1)

	0	1	2	3	4
parent	1	-1	-1	-1	-1

union1(0, 1): parent[0] = 1. Each call is O(1).

Step 2 / 11

```
1 int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
▶   parent[i] = j;      /* put tree i under j */
8 }
```

VARIABLES

`i = 0` `j = 1`

CALL STACK

`union1(0, 1)`

`main()`

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



union1(1, 2)

	0	1	2	3	4
parent	1	2	-1	-1	-1

union1(1, 2): parent[1] = 2. Each call is O(1).

Step 3 / 11

```
1 int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
▶   parent[i] = j;      /* put tree i under j */
8 }
```

VARIABLES

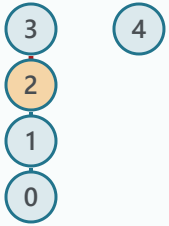
`i = 1` `j = 2`

CALL STACK

`union1(1, 2)`

`main()`

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



union1(2, 3)

	0	1	2	3	4
parent	1	2	3	-1	-1

union1(2, 3): parent[2] = 3. Each call is O(1).

Step 4 / 11

```
1 int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
▶   parent[i] = j;      /* put tree i under j */
8 }
```

VARIABLES

`i = 2` `j = 3`

CALL STACK

`union1(2, 3)`

`main()`

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



union1(3, 4)

	0	1	2	3	4
parent	1	2	3	4	-1

union1(3, 4): parent[3] = 4. Each call is O(1).

Step 5 / 11

```
1 int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
▶   parent[i] = j;      /* put tree i under j */
8 }
```

VARIABLES

`i = 3` `j = 4`

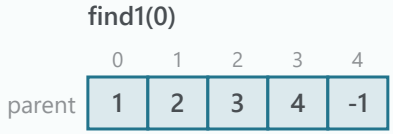
CALL STACK

`union1(3, 4)`

`main()`

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`





The tree has become a **chain** of depth n. Now **find1(0)**.

Step 6 / 11

```
▶ int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;      /* put tree i under j */
8 }
```

VARIABLES

`i = 0` `steps = 0`

CALL STACK

`find1(0)`

`main()`

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



find1(0)

	0	1	2	3	4
parent	1	2	3	4	-1

i = parent[0] = **1** (step 1).

Step 7 / 11

```
1 int find1(int i) {
▶   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

`i = 1` `steps = 1`

CALL STACK

`find1(0)`

`main()`

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



find1(0)

	0	1	2	3	4
parent	1	2	3	4	-1

i = parent[1] = **2** (step 2).

Step 8 / 11

```
1 int find1(int i) {
▶   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

`i = 2` `steps = 2`

CALL STACK

`find1(0)`

`main()`

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



find1(0)

	0	1	2	3	4
parent	1	2	3	4	-1

i = parent[2] = **3** (step 3).

Step 9 / 11

```
1 int find1(int i) {
▶   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

`i = 3` `steps = 3`

CALL STACK

`find1(0)`

`main()`

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



find1(0)

	0	1	2	3	4
parent	1	2	3	4	-1

i = parent[3] = **4** (step 4).

Step 10 / 11

```
1 int find1(int i) {
▶   for (; parent[i] >= 0; i = parent[i])
3     ;                /* climb to a root */
4   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;     /* put tree i under j */
8 }
```

VARIABLES

`i = 4` `steps = 4`

CALL STACK

`find1(0)`

`main()`

ANALYSIS · STEP THROUGH THE CODE

Code Trace: `union1(0,1)` … `union1(3,4)`, `find1(0)`



find1(0)

	0	1	2	3	4
parent	1	2	3	4	-1

return 4 after n−1 = 4 steps. find1(1), (2), … take n−2, n−3, … steps: in total Σ = $O(n^2)$.

Step 11 / 11

```
1 int find1(int i) {
2   for (; parent[i] >= 0; i = parent[i])
3     ;                 /* climb to a root */
▶   return i;
5 }
6 void union1(int i, int j) {
7   parent[i] = j;      /* put tree i under j */
8 }
```

VARIABLES

`i = 4` `steps = 4`

CALL STACK

`find1(0)`

`main()`

Analysis of `find1` and `union1`

```
/* start with n singleton sets */
union1(0, 1);
union1(1, 2);
...
union1(n-2, n-1);
find1(0);
find1(1);
...
find1(n-1);
```

- n−1 unions, each constant → **O(n)** in total; no problem there.
- The tree degenerates into a chain of depth **n**.
- For the element i at level n−i, find1 takes O(n−i): n−1 steps for 0, n−2 for 1, …
- One find1 can cost O(n); n of them cost **$\Sigma\, i = O(n^2)$**.

This sequence ruins the tree. The culprit is not the union itself but *which tree is attached to which*. The fix: the weighting rule.

ANALYSIS

An Aside: Testing the Worst Case

Worst-case input

For worst-case analysis, code should be tested on the data that crashes it or makes it run longest. Finding that data is hard.

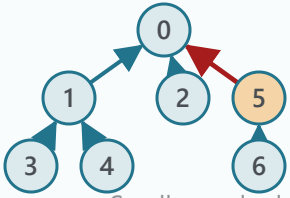
Here

The chain-building union order is exactly such an input for `find1`; random tests easily miss it.

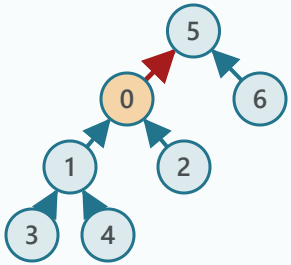
Instructor *"What does 'beta version' mean? 'We may have missed some bugs. Forgive us, and if you find one, tell us so we can fix it.' Software testing is a field of its own; read papers on it too."*

The Weighting Rule

Definition: if tree i has fewer nodes than tree j, make j the parent of i; otherwise make i the parent of j. (The tree with more nodes becomes the parent.)



Smaller under larger: depth 3



Larger under smaller: depth 4

Instructor *"Question: do we need a separate array for the number of nodes in each tree? Answer: every non-root node already stores a nonnegative value in the parent array. So the node count of a root can be stored there too, as a negative number."*

WEIGHTING RULE

`union2`: The Root Stores Minus the Node Count

```
void union2(int i, int j) {
  /* weighting rule; i, j are roots */
  int p = parent[i] + parent[j]; /* -(size) */
  if (parent[i] > parent[j]) {  /* i smaller */
    parent[i] = j;      /* i under j */
    parent[j] = p;      /* j's new size */
  } else {              /* j not larger */
    parent[j] = i;
    parent[i] = p;
  }
}
```

−count at the root

A root holds `-(number of nodes)`: −1 for a singleton, −8 for an 8-element set.

Why `>`?

The values are negative: `parent[i] > parent[j]` means i has **fewer** nodes (−2 > −4).

Cost

Still O(1), with no extra array.

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(0, 1)

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

union2(0, 1): set 0 has 1 node, set 1 has 1 node.

Step 1 / 16

```
▶ void union2(int i, int j) {
2   /* weighting rule; i, j are roots */
3   int p = parent[i] + parent[j]; /* -(size) */
4   if (parent[i] > parent[j]) {  /* i smaller */
5     parent[i] = j;      /* i under j */
6     parent[j] = p;      /* j's new size */
7   } else {              /* j not larger */
8     parent[j] = i;
9     parent[i] = p;
10  }
11 }
```

VARIABLES

`i = 0` `j = 1` `parent[i] = -1` `parent[j] = -1`

CALL STACK

`union2(0, 1)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(0, 1)

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

p = -1 + (-1) = **-2**: the merged set has 2 nodes.

Step 2 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
▶    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 0` `j = 1` `p = -2`

CALL STACK

`union2(0, 1)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(0, 1)

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

-1 > -1 is false (j is not larger) → **else**: j goes under i.

Step 3 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
▶    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 0` `j = 1` `p = -2`

CALL STACK

`union2(0, 1)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(0, 1)

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

parent[1] = 0, **parent[0] = -2**. The deepest node is on level 2.

Step 4 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
▶      parent[j] = i;
▶      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 0` `j = 1` `p = -2`

`parent[j] = 0` `parent[i] = -2`

CALL STACK

`union2(0, 1)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(2, 3)

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

union2(2, 3): set 2 has 1 node, set 3 has 1 node.

Step 5 / 16

```
▶ void union2(int i, int j) {
2   /* weighting rule; i, j are roots */
3   int p = parent[i] + parent[j]; /* -(size) */
4   if (parent[i] > parent[j]) {  /* i smaller */
5     parent[i] = j;      /* i under j */
6     parent[j] = p;      /* j's new size */
7   } else {              /* j not larger */
8     parent[j] = i;
9     parent[i] = p;
10  }
11 }
```

VARIABLES

`i = 2` `j = 3` `parent[i] = -1` `parent[j] = -1`

CALL STACK

`union2(2, 3)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(2, 3)

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

p = -1 + (-1) = **-2**: the merged set has 2 nodes.

Step 6 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
▶    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 2` `j = 3` `p = -2`

CALL STACK

`union2(2, 3)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(2, 3)

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

-1 > -1 is false (j is not larger) → **else**: j goes under i.

Step 7 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
▶    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

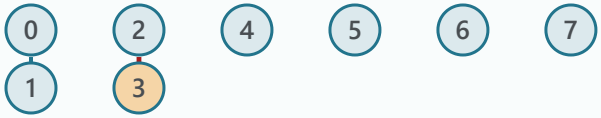
`i = 2` `j = 3` `p = -2`

CALL STACK

`union2(2, 3)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(2, 3)

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

parent[3] = 2, **parent[2] = -2**. The deepest node is on level 2.

Step 8 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
▶      parent[j] = i;
▶      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 2` `j = 3` `p = -2`

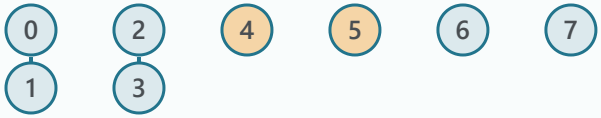
`parent[j] = 2` `parent[i] = -2`

CALL STACK

`union2(2, 3)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(4, 5)

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

union2(4, 5): set 4 has 1 node, set 5 has 1 node.

Step 9 / 16

```
▶ void union2(int i, int j) {
2   /* weighting rule; i, j are roots */
3   int p = parent[i] + parent[j]; /* -(size) */
4   if (parent[i] > parent[j]) {  /* i smaller */
5     parent[i] = j;      /* i under j */
6     parent[j] = p;      /* j's new size */
7   } else {              /* j not larger */
8     parent[j] = i;
9     parent[i] = p;
10  }
11 }
```

VARIABLES

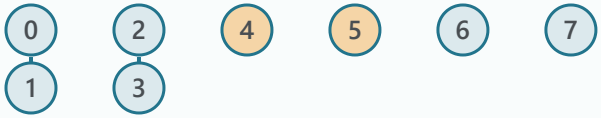
`i = 4` `j = 5` `parent[i] = -1` `parent[j] = -1`

CALL STACK

`union2(4, 5)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(4, 5)

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

p = -1 + (-1) = **-2**: the merged set has 2 nodes.

Step 10 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
▶    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

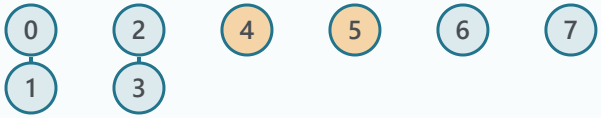
`i = 4` `j = 5` `p = -2`

CALL STACK

`union2(4, 5)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(4, 5)

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

-1 > -1 is false (j is not larger) → **else**: j goes under i.

Step 11 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
▶    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

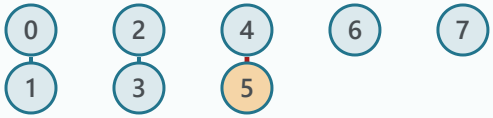
`i = 4` `j = 5` `p = -2`

CALL STACK

`union2(4, 5)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(4, 5)

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

parent[5] = 4, **parent[4] = -2**. The deepest node is on level 2.

Step 12 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
▶      parent[j] = i;
▶      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 4` `j = 5` `p = -2`

`parent[j] = 4` `parent[i] = -2`

CALL STACK

`union2(4, 5)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(6, 7)

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

union2(6, 7): set 6 has 1 node, set 7 has 1 node.

Step 13 / 16

```
▶ void union2(int i, int j) {
2   /* weighting rule; i, j are roots */
3   int p = parent[i] + parent[j]; /* -(size) */
4   if (parent[i] > parent[j]) {  /* i smaller */
5     parent[i] = j;      /* i under j */
6     parent[j] = p;      /* j's new size */
7   } else {              /* j not larger */
8     parent[j] = i;
9     parent[i] = p;
10  }
11 }
```

VARIABLES

`i = 6` `j = 7` `parent[i] = -1` `parent[j] = -1`

CALL STACK

`union2(6, 7)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(6, 7)

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

p = -1 + (-1) = **-2**: the merged set has 2 nodes.

Step 14 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
▶    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 6` `j = 7` `p = -2`

CALL STACK

`union2(6, 7)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(6, 7)

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

-1 > -1 is false (j is not larger) → **else**: j goes under i.

Step 15 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
▶    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

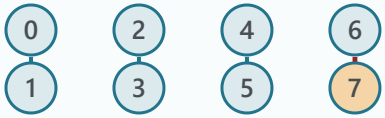
`i = 6` `j = 7` `p = -2`

CALL STACK

`union2(6, 7)`

`main()`

Code Trace: `union2` (0,1) (2,3) (4,5) (6,7)



union2(6, 7)

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

parent[7] = 6, **parent[6] = -2**. The deepest node is on level 2.

Step 16 / 16

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
▶      parent[j] = i;
▶      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 6` `j = 7` `p = -2`

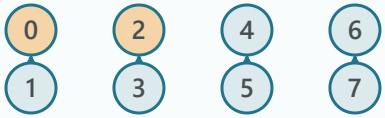
`parent[j] = 6` `parent[i] = -2`

CALL STACK

`union2(6, 7)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(0, 2)

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

union2(0, 2): set 0 has 2 nodes, set 2 has 2 nodes.

Step 1 / 12

```
▶ void union2(int i, int j) {
2   /* weighting rule; i, j are roots */
3   int p = parent[i] + parent[j]; /* -(size) */
4   if (parent[i] > parent[j]) {  /* i smaller */
5     parent[i] = j;      /* i under j */
6     parent[j] = p;      /* j's new size */
7   } else {              /* j not larger */
8     parent[j] = i;
9     parent[i] = p;
10  }
11 }
```

VARIABLES

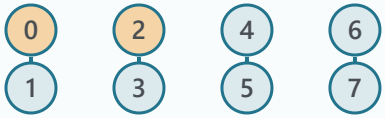
`i = 0` `j = 2` `parent[i] = -2` `parent[j] = -2`

CALL STACK

`union2(0, 2)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(0, 2)

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

p = -2 + (-2) = **-4**: the merged set has 4 nodes.

Step 2 / 12

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
▶    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

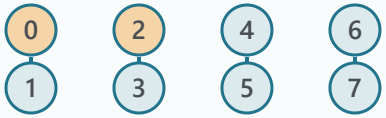
`i = 0` `j = 2` `p = -4`

CALL STACK

`union2(0, 2)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(0, 2)

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

-2 > -2 is false (j is not larger) → **else**: j goes under i.

Step 3 / 12

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
▶    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

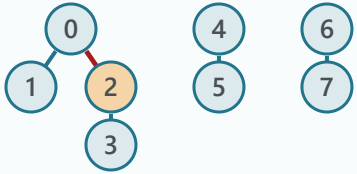
`i = 0` `j = 2` `p = -4`

CALL STACK

`union2(0, 2)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(0, 2)

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

parent[2] = 0, **parent[0] = -4**. The deepest node is on level 3.

Step 4 / 12

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
▶      parent[j] = i;
▶      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 0` `j = 2` `p = -4`

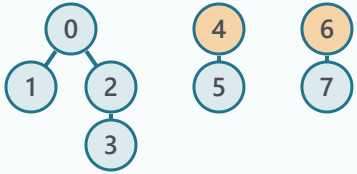
`parent[j] = 0` `parent[i] = -4`

CALL STACK

`union2(0, 2)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(4, 6)

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

union2(4, 6): set 4 has 2 nodes, set 6 has 2 nodes.

Step 5 / 12

```
▶ void union2(int i, int j) {
2   /* weighting rule; i, j are roots */
3   int p = parent[i] + parent[j]; /* -(size) */
4   if (parent[i] > parent[j]) {  /* i smaller */
5     parent[i] = j;      /* i under j */
6     parent[j] = p;      /* j's new size */
7   } else {              /* j not larger */
8     parent[j] = i;
9     parent[i] = p;
10  }
11 }
```

VARIABLES

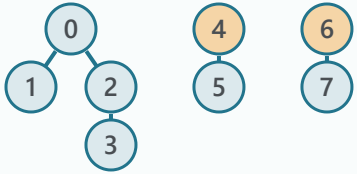
`i = 4` `j = 6` `parent[i] = -2` `parent[j] = -2`

CALL STACK

`union2(4, 6)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(4, 6)

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

p = -2 + (-2) = **-4**: the merged set has 4 nodes.

Step 6 / 12

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
▶    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

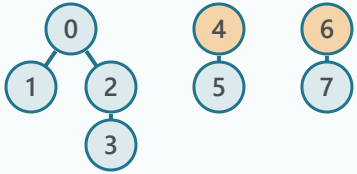
`i = 4` `j = 6` `p = -4`

CALL STACK

`union2(4, 6)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(4, 6)

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

-2 > -2 is false (j is not larger) → **else**: j goes under i.

Step 7 / 12

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
▶    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

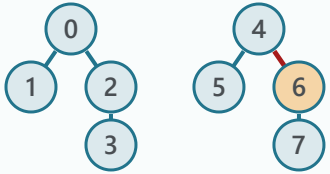
`i = 4` `j = 6` `p = -4`

CALL STACK

`union2(4, 6)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(4, 6)

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

parent[6] = 4, **parent[4] = -4**. The deepest node is on level 3.

Step 8 / 12

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
▶      parent[j] = i;
▶      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 4` `j = 6` `p = -4`

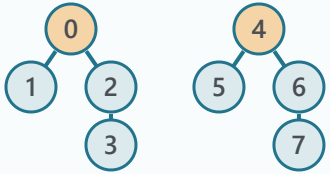
`parent[j] = 4` `parent[i] = -4`

CALL STACK

`union2(4, 6)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(0, 4)

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

union2(0, 4): set 0 has 4 nodes, set 4 has 4 nodes.

Step 9 / 12

```
▶ void union2(int i, int j) {
2   /* weighting rule; i, j are roots */
3   int p = parent[i] + parent[j]; /* -(size) */
4   if (parent[i] > parent[j]) {  /* i smaller */
5     parent[i] = j;      /* i under j */
6     parent[j] = p;      /* j's new size */
7   } else {              /* j not larger */
8     parent[j] = i;
9     parent[i] = p;
10  }
11 }
```

VARIABLES

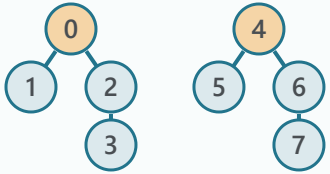
`i = 0` `j = 4` `parent[i] = -4` `parent[j] = -4`

CALL STACK

`union2(0, 4)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(0, 4)

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

p = -4 + (-4) = **-8**: the merged set has 8 nodes.

Step 10 / 12

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
▶    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;      /* i under j */
6      parent[j] = p;      /* j's new size */
7    } else {              /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

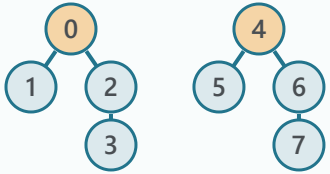
`i = 0` `j = 4` `p = -8`

CALL STACK

`union2(0, 4)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(0, 4)

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

-4 > -4 is false (j is not larger) → **else**: j goes under i.

Step 11 / 12

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
▶    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
8      parent[j] = i;
9      parent[i] = p;
10   }
11 }
```

VARIABLES

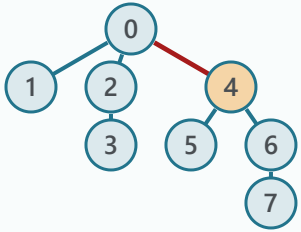
`i = 0` `j = 4` `p = -8`

CALL STACK

`union2(0, 4)`

`main()`

Code Trace: `union2` (0,2) (4,6) (0,4)



union2(0, 4)

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

parent[4] = 0, **parent[0] = -8**. The deepest node is on level 4.

Step 12 / 12

```
1  void union2(int i, int j) {
2    /* weighting rule; i, j are roots */
3    int p = parent[i] + parent[j]; /* -(size) */
4    if (parent[i] > parent[j]) {  /* i smaller */
5      parent[i] = j;       /* i under j */
6      parent[j] = p;       /* j's new size */
7    } else {               /* j not larger */
▶      parent[j] = i;
▶      parent[i] = p;
10   }
11 }
```

VARIABLES

`i = 0` `j = 4` `p = -8`

`parent[j] = 0` `parent[i] = -8`

CALL STACK

`union2(0, 4)`

`main()`

Question: How Badly Can the Tree Degenerate?

Instructor *"The algorithm must be correct, efficient and well engineered. But that is not the end: we ask which data hurts it most. Whatever happens behind the scenes, how long can my code run at most? That is what computer science and computer engineering are about!"*

Instructor *"'Do you all know C, Java and so on?' Yes, they do. 'Fine, you are all hired, same salary!' The computer engineer says: 'Wait a minute! I am a computer engineer. How can it be the same salary? Ask me a question and watch how I solve it!'"*

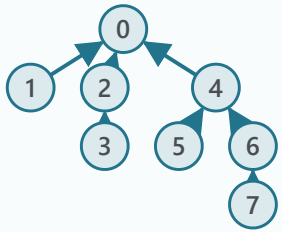
The worst case for the weighting rule is uniting two trees of **equal size** every time (as in the `union2` traces). Each union deepens every node of the smaller tree by 1, but the tree at least doubles.

Lemma: Weighted Union Keeps the Depth Logarithmic

LEMMA (HOROWITZ)

If T is a tree with **m nodes** built by a sequence of `union2` operations, then the depth (number of levels) of T is at most ⌊**lg m**⌋ **+ 1**.

- **Proof idea (induction):** a node gets deeper by 1 only when its tree is attached to a tree that is *no smaller*.
- Then the tree containing the node at least **doubles**.
- A size can double at most lg m times before reaching m → depth ≤ ⌊lg m⌋ + 1.



m = 8 → 4 levels = ⌊lg 8⌋ + 1

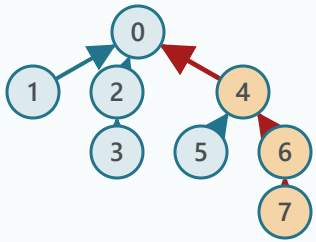
Instructor *"In a mathematics department the question would go on: 'Prove it.' […] We brought the cost down from n to ⌊lg n⌋+1. That is algorithm design! But we do not stop here, on we go…"*

The Collapsing Rule

In the final tree, `find(7)` reaches the root in 3 steps. If a program calls `find(7)` over and over, the cost piles up. Even when the smaller tree goes under the larger one the depth can grow; since we cannot control the depth, we go one step further:

DEFINITION

If j is a node on the path from i to its root (i not the root), then j is **linked directly to the root**.



every node on the path 7 → 6 → 4 → 0 is linked to the root

Instructor *"In graph theory this corresponds to a 'closure'. You will need it in one of the graph theory assignments; I won't say which one. The problem looks different, but this is what lies underneath."*

COLLAPSING RULE

`find2`: Find the Root, Then Link the Path to It

```
int find2(int i) {
  int r, s;
  for (r = i; parent[r] >= 0; r = parent[r])
    ;                    /* find the root */
  while (i != r) {       /* collapse path */
    s = parent[i];       /* save the parent */
    parent[i] = r;       /* link i to root */
    i = s;               /* move up */
  }
  return r;
}
```

Pass 1 (for)

`r` finds the root (the negative entry).

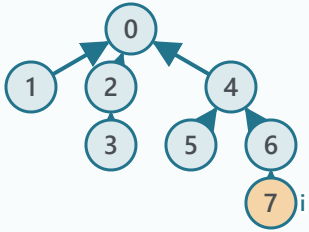
Pass 2 (while)

Every node on the path is linked straight to `r`. It also runs once for a node already linked to the root (useless but harmless).

Payoff

Later finds on the same path reach the root in one step.

Code Trace: `find2(7)`



find2(7)

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

find2(7).

Step 1 / 12

```
▶ int find2(int i) {
2   int r, s;
3   for (r = i; parent[r] >= 0; r = parent[r])
4     ;                    /* find the root */
5   while (i != r) {       /* collapse path */
6     s = parent[i];       /* save the parent */
7     parent[i] = r;       /* link i to root */
8     i = s;               /* move up */
9   }
10  return r;
11 }
```

VARIABLES

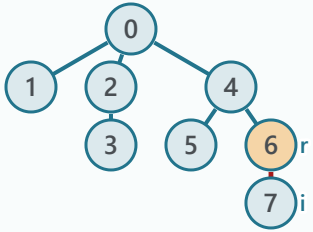
`i = 7`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)`



find2(7)

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

parent[7] = 6 ≥ 0 → r = **6**.

Step 2 / 12

```
1  int find2(int i) {
2    int r, s;
▶    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

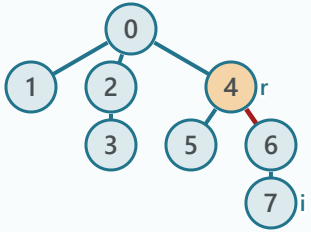
`i = 7` `r = 6`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)`



find2(7)

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

parent[6] = 4 ≥ 0 → r = **4**.

Step 3 / 12

```
1  int find2(int i) {
2    int r, s;
▶    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

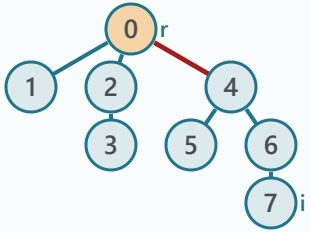
`i = 7` `r = 4`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)`



find2(7)

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

parent[4] = 0 ≥ 0 → r = **0**.

Step 4 / 12

```
1  int find2(int i) {
2    int r, s;
▶    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

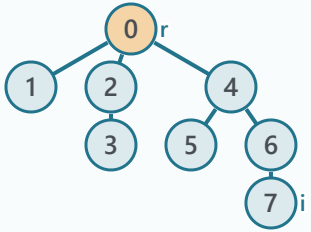
`i = 7` `r = 0`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)`



find2(7)

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

parent[0] = -8 < 0: root **r = 0**.

Step 5 / 12

```
1  int find2(int i) {
2    int r, s;
▶    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

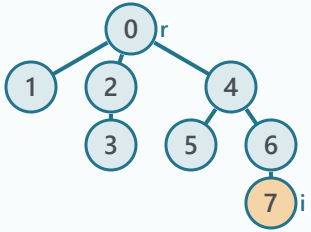
`i = 7` `r = 0`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)`



find2(7)

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

i = 7 ≠ r: save **s = parent[7] = 6**.

Step 6 / 12

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
▶    while (i != r) {       /* collapse path */
▶      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

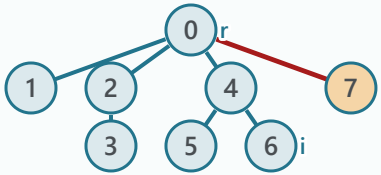
VARIABLES

`i = 7` `r = 0` `s = 6`

CALL STACK

- `find2(7)`
- `main()`

Code Trace: `find2(7)`



find2(7)

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

parent[7] = 0: 7 now points to the root; i = s = 6.

Step 7 / 12

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
▶      parent[i] = r;       /* link i to root */
▶      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

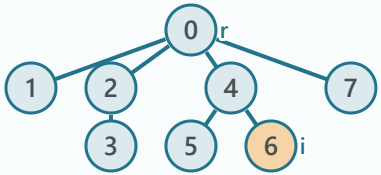
`i = 6` `r = 0` `s = 6`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)`



find2(7)

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

i = 6 ≠ r: save **s = parent[6] = 4**.

Step 8 / 12

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
▶    while (i != r) {       /* collapse path */
▶      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

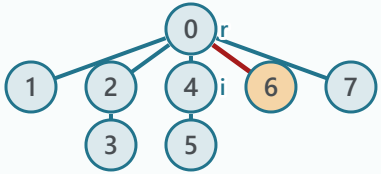
VARIABLES

`i = 6` `r = 0` `s = 4`

CALL STACK

- `find2(7)`
- `main()`

Code Trace: `find2(7)`



find2(7)

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

parent[6] = 0: 6 now points to the root; i = s = 4.

Step 9 / 12

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
▶      parent[i] = r;       /* link i to root */
▶      i = s;               /* move up */
9    }
10   return r;
11 }
```

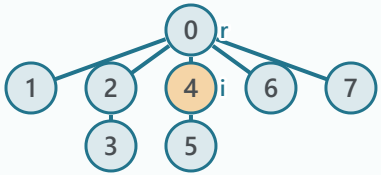
VARIABLES

i = 4 | r = 0 | s = 4

CALL STACK

- find2(7)
- main()

Code Trace: `find2(7)`



find2(7)

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

i = 4 ≠ r: save **s = parent[4] = 0**.

Step 10 / 12

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
▶    while (i != r) {       /* collapse path */
▶      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

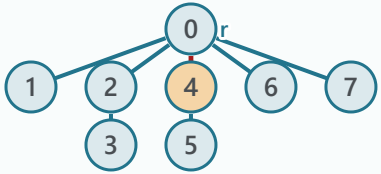
VARIABLES

`i = 4` `r = 0` `s = 0`

CALL STACK

- `find2(7)`
- `main()`

Code Trace: `find2(7)`



find2(7)

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

parent[4] = 0 (already linked to the root: one extra round); i = 0.

Step 11 / 12

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
▶      parent[i] = r;       /* link i to root */
▶      i = s;               /* move up */
9    }
10   return r;
11 }
```

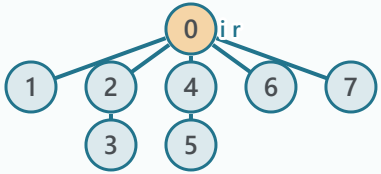
VARIABLES

i = 0 | r = 0 | s = 0

CALL STACK

- find2(7)
- main()

Code Trace: `find2(7)`



find2(7)

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

i == r: the loop ends, **return 0**. parent = [-8, 0, 0, 2, 0, 4, 0, 0].

Step 12 / 12

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
▶    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
▶    return r;
11 }
```

VARIABLES

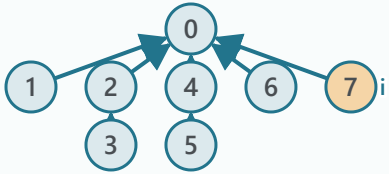
`i = 0` `r = 0`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(7)

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

find2(7).

Step 1 / 15

```
▶ int find2(int i) {
2   int r, s;
3   for (r = i; parent[r] >= 0; r = parent[r])
4     ;                    /* find the root */
5   while (i != r) {       /* collapse path */
6     s = parent[i];       /* save the parent */
7     parent[i] = r;       /* link i to root */
8     i = s;               /* move up */
9   }
10  return r;
11 }
```

VARIABLES

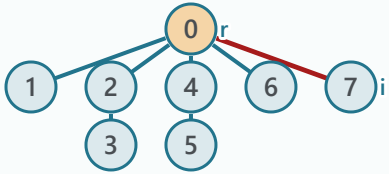
`i = 7`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(7)

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

parent[7] = 0 ≥ 0 → r = **0**.

Step 2 / 15

```
1  int find2(int i) {
2    int r, s;
▶    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

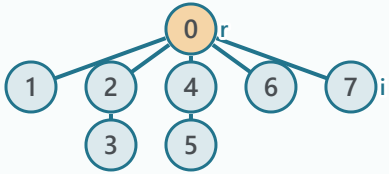
`i = 7` `r = 0`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(7)

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

parent[0] = -8 < 0: root **r = 0**.

Step 3 / 15

```
1  int find2(int i) {
2    int r, s;
▶    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

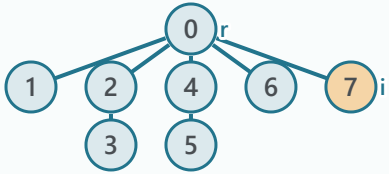
`i = 7` `r = 0`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(7)

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

i = 7 ≠ r: save **s = parent[7] = 0**.

Step 4 / 15

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
▶    while (i != r) {       /* collapse path */
▶      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

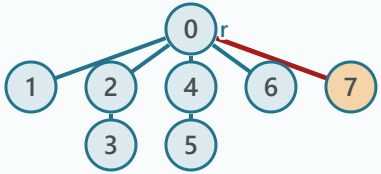
VARIABLES

i = 7 | r = 0 | s = 0

CALL STACK

- find2(7)
- main()

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(7)

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

parent[7] = 0 (already linked to the root: one extra round); i = 0.

Step 5 / 15

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
▶      parent[i] = r;       /* link i to root */
▶      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

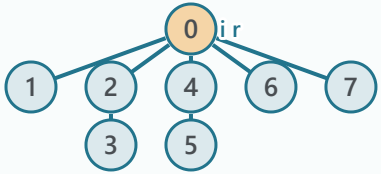
`i = 0` `r = 0` `s = 0`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(7)

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

i == r: the loop ends, **return 0**. parent = [-8, 0, 0, 2, 0, 4, 0, 0].

Step 6 / 15

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
▶    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
▶    return r;
11 }
```

VARIABLES

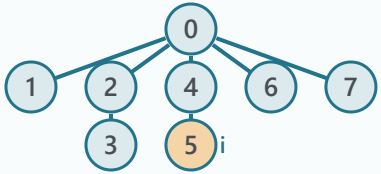
`i = 0` `r = 0`

CALL STACK

`find2(7)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(5)

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

find2(5).

Step 7 / 15

```
▶ int find2(int i) {
2   int r, s;
3   for (r = i; parent[r] >= 0; r = parent[r])
4     ;                    /* find the root */
5   while (i != r) {       /* collapse path */
6     s = parent[i];       /* save the parent */
7     parent[i] = r;       /* link i to root */
8     i = s;               /* move up */
9   }
10  return r;
11 }
```

VARIABLES

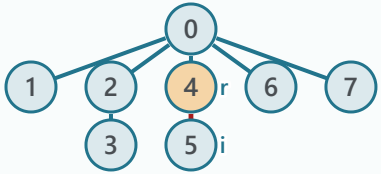
`i = 5`

CALL STACK

`find2(5)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(5)

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

parent[5] = 4 ≥ 0 → r = **4**.

Step 8 / 15

```
1  int find2(int i) {
2    int r, s;
▶    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

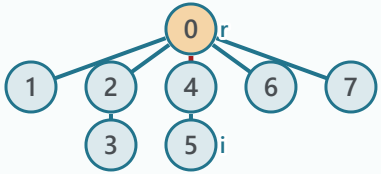
`i = 5` `r = 4`

CALL STACK

`find2(5)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(5)

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

parent[4] = 0 ≥ 0 → r = **0**.

Step 9 / 15

```
1  int find2(int i) {
2    int r, s;
▶    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

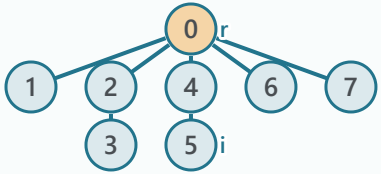
VARIABLES

`i = 5` `r = 0`

CALL STACK

- `find2(5)`
- `main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(5)

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

parent[0] = -8 < 0: root **r = 0**.

Step 10 / 15

```
1  int find2(int i) {
2    int r, s;
▶    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

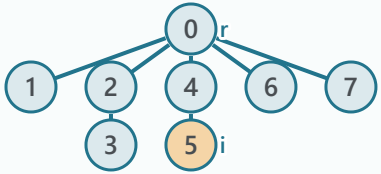
`i = 5` `r = 0`

CALL STACK

`find2(5)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(5)

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

i = 5 ≠ r: save **s = parent[5] = 4**.

Step 11 / 15

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
▶    while (i != r) {       /* collapse path */
▶      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

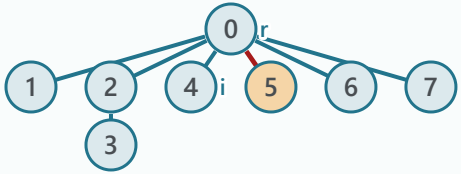
VARIABLES

`i = 5` `r = 0` `s = 4`

CALL STACK

- `find2(5)`
- `main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(5)

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

parent[5] = 0: 5 now points to the root; i = s = 4.

Step 12 / 15

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
▶      parent[i] = r;       /* link i to root */
▶      i = s;               /* move up */
9    }
10   return r;
11 }
```

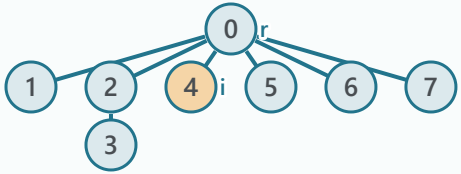
VARIABLES

i = 4 | r = 0 | s = 4

CALL STACK

- find2(5)
- main()

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(5)

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

i = 4 ≠ r: save **s = parent[4] = 0**.

Step 13 / 15

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
▶    while (i != r) {       /* collapse path */
▶      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
10   return r;
11 }
```

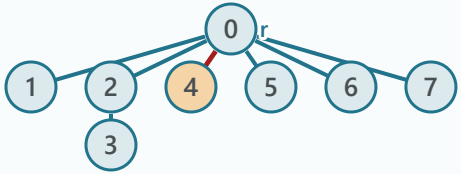
VARIABLES

`i = 4` `r = 0` `s = 0`

CALL STACK

- `find2(5)`
- `main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(5)

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

parent[4] = 0 (already linked to the root: one extra round); i = 0.

Step 14 / 15

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
5    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
▶      parent[i] = r;       /* link i to root */
▶      i = s;               /* move up */
9    }
10   return r;
11 }
```

VARIABLES

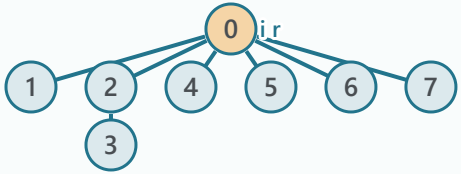
`i = 0` `r = 0` `s = 0`

CALL STACK

`find2(5)`

`main()`

Code Trace: `find2(7)` Again, Then `find2(5)`



find2(5)

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

i == r: the loop ends, **return 0**. parent = [-8, 0, 0, 2, 0, 0, 0, 0].

Step 15 / 15

```
1  int find2(int i) {
2    int r, s;
3    for (r = i; parent[r] >= 0; r = parent[r])
4      ;                    /* find the root */
▶    while (i != r) {       /* collapse path */
6      s = parent[i];       /* save the parent */
7      parent[i] = r;       /* link i to root */
8      i = s;               /* move up */
9    }
▶    return r;
11 }
```

VARIABLES

`i = 0` `r = 0`

CALL STACK

`find2(5)`

`main()`

COLLAPSING RULE

“We Had a for Loop, and Now We Added a while!”

Instructor *“‘We had a for loop and now we added a while loop too. How is that an improvement?’ I don't get it either, let's see. On the second and every later call, find2(7) finds r = 0 in a single step.”*

Call	find1 (steps)	find2 (steps + relinks)
1st find(7)	3	3 + 3
2nd find(7)	3	1 + 1
find(7) 10 times	30	6 + 9·2 = 24
find(5) (afterwards)	2	2 + 2, then 1 + 1

The first call costs about twice as much, but it shortens the path for good. **Amortized** over many operations, the cost drops to almost a constant.

Lemma (Tarjan and Van Leeuwen): Almost Constant

LEMMA

Start with singleton sets and let T(f, u) be the maximum time for **f** finds and **u** unions (weighting rule + collapsing rule) in any interleaved order. For u ≥ n/2:

$$k_1 \cdot (n + f \cdot \alpha(f+n, n)) \leq T(f, u) \leq k_2 \cdot (n + f \cdot \alpha(f+n, n))$$

- α is the **inverse Ackermann** function; it grows extremely slowly.
- Even for as many elements as there are atoms in the universe, **α ≤ 4** in practice.
- Divided by the number of operations, the cost is **almost O(1)**.

```
/* Ackermann: grows very fast */
A(m, n) = n + 1                    m = 0
        = A(m-1, 1)                m > 0, n = 0
        = A(m-1, A(m, n-1))        m > 0, n > 0
```

Instructor *"There are a million elements; one finds them in O(1), the other is still searching a list... As we said, this is the difference between the mathematician and the engineer applying for the same job!"*

The Ackermann Function and Its Inverse

	n = 0	n = 1	n = 2	n = 3	n = 4
m = 0	1	2	3	4	5
m = 1	2	3	4	5	6
m = 2	3	5	7	9	11
m = 3	5	13	29	61	125
m = 4	13	65533	$2^{65536}-3$	…	…

- $A(5, 1) = 2^{2^{\cdot^{\cdot^{2}}}} - 3$ (a tower of 65536 twos): unimaginably large.
- Inverse: $\alpha(m, n) = \min\{ i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \lg n \}$. Since A grows so fast, α grows very slowly.

The Ackermann function has several definitions. With the one above, A(3,4) = 125; with Horowitz's definition ($A(1,j) = 2^j$ …), A(3,1) = 16. The conclusion is the same either way: in practice $\alpha \leq 4$.

Exam Question: Write the Ackermann Function Recursively

EXAM QUESTION

Write the Ackermann function recursively. Then show the call stack of A(1, 2) step by step.

▶ **Solution**

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)

A(1,2)

`ackermann(1, 2)` is called.

Step 1 / 12

```
▶ unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
3   if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
5   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

`m = 1` `n = 2`

CALL STACK

`ackermann(1,2)`

`main()`

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)

A(1,2)

A(1,2): the inner A(1,1) must be computed first.

Step 2 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
3   if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
▶   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

`m = 1` `n = 2`

CALL STACK

`ackermann(1,2)`

`main()`

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)



A(1,1): the inner A(1,0) must be computed first.

Step 3 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
3   if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
▶   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

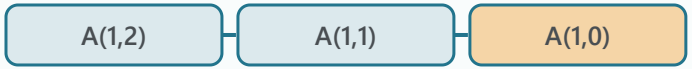
`m = 1` `n = 1`

CALL STACK

- `ackermann(1,1)`
- `ackermann(1,2)`
- `main()`

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)



A(1,0): n = 0 → call A(0, 1).

Step 4 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
3   if (m == 0) return n + 1;
▶   if (n == 0) return ackermann(m - 1, 1);
5   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

`m = 1` `n = 0`

CALL STACK

- **ackermann(1,0)**
- ackermann(1,1)
- ackermann(1,2)
- main()

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)



A(0,1) = 2

A(0,1): m = 0 → **return 2**.

Step 5 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
  if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
5   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

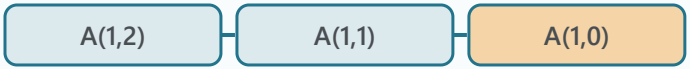
`m = 0` `n = 1`

CALL STACK

- `ackermann(0,1)`
- `ackermann(1,0)`
- `ackermann(1,1)`
- `ackermann(1,2)`
- `main()`

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)



A(1,0) = 2

A(1,0) = A(0,1) = **2**; return.

Step 6 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
3   if (m == 0) return n + 1;
▶   if (n == 0) return ackermann(m - 1, 1);
5   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

`m = 1` `n = 0` `result = 2`

CALL STACK

- **ackermann(1,0)**
- ackermann(1,1)
- ackermann(1,2)
- main()

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)



The inner call gave 2 → outer call A(0,2).

Step 7 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
3   if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
▶   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

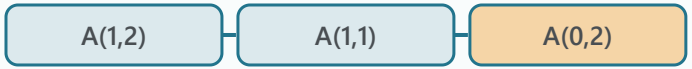
`m = 1` `n = 1` `inner = 2`

CALL STACK

- `ackermann(1,1)`
- `ackermann(1,2)`
- `main()`

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)



A(0,2) = 3

A(0,2): m = 0 → **return 3**.

Step 8 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
  if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
5   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

`m = 0` `n = 2`

CALL STACK

- `ackermann(0,2)`
- `ackermann(1,1)`
- `ackermann(1,2)`
- `main()`

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)



A(1,1) = 3

A(1,1) = **3**.

Step 9 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
3   if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
▶   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

`m = 1` `n = 1` `result = 3`

CALL STACK

- `ackermann(1,1)`
- `ackermann(1,2)`
- `main()`

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)

A(1,2)

The inner call gave 3 → outer call A(0,3).

Step 10 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
3   if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
▶   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

`m = 1` `n = 2` `inner = 3`

CALL STACK

`ackermann(1,2)`

`main()`

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)



A(0,3) = 4

A(0,3): m = 0 → **return 4**.

Step 11 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
  if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
5   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

`m = 0` `n = 3`

CALL STACK

- `ackermann(0,3)`
- `ackermann(1,2)`
- `main()`

Code Trace: `ackermann(1, 2)`

active calls (deeper to the right)

A(1,2)

A(1,2) = 4

A(1,2) = **4**.

Step 12 / 12

```
1 unsigned long ackermann(unsigned long m,
2                         unsigned long n) {
3   if (m == 0) return n + 1;
4   if (n == 0) return ackermann(m - 1, 1);
▶   return ackermann(m - 1, ackermann(m, n - 1));
6 }
```

VARIABLES

m = 1 n = 2 result = 4

CALL STACK

ackermann(1,2)

main()

An Aside: Where Do These Lemmas Come From?

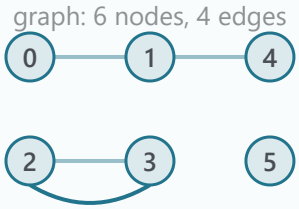
Instructor *"Tarjan is a famous graph theorist with many results to his name. This lemma came out of his joint work with Jan van Leeuwen (1984)."*

Instructor *"An undergraduate degree takes four years and gives you a profession; in a PhD you spend four years on a single problem, one nobody in the world has solved [...] That is why working in a group means learning all the time."*

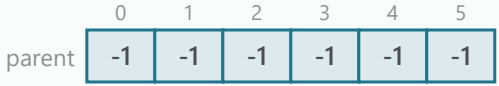
Instructor *"You cannot finish these topics in a week, and you never will if you don't study. Study 2–3 hours every day, and review each week's topic that same week."*

APPLICATION · STEP THROUGH THE CODE

Code Trace: Counting Connected components







Edges u–v: (0,1), (2,3), (1,4), (3,2). How many connected components are there?

Step 1 / 11

```
▶ int components(int n, int m, int u[], int v[]) {
2   for (int i = 0; i < n; i++) parent[i] = -1;
3   int count = n;              /* all separate */
4   for (int e = 0; e < m; e++) {
5     int a = find2(u[e]), b = find2(v[e]);
6     if (a != b) {             /* new component */
7       union2(a, b);
8       count--;
9     }
10  }
11  return count;
12 }
```

VARIABLES

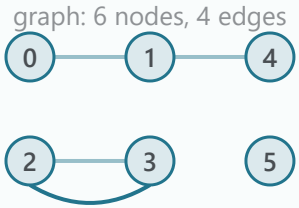
`n = 6` `m = 4`

CALL STACK

`components(6, 4, u, v)`

`main()`

Code Trace: Counting Connected components





count = 6

	0	1	2	3	4	5
parent	-1	-1	-1	-1	-1	-1

Every node is its own set: parent = −1, **count = 6**.

Step 2 / 11

```
1  int components(int n, int m, int u[], int v[]) {
▶    for (int i = 0; i < n; i++) parent[i] = -1;
▶    int count = n;              /* all separate */
4    for (int e = 0; e < m; e++) {
5      int a = find2(u[e]), b = find2(v[e]);
6      if (a != b) {             /* new component */
7        union2(a, b);
8        count--;
9      }
10   }
11   return count;
12 }
```

VARIABLES

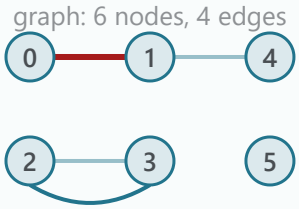
`n = 6` `m = 4` `count = 6`

CALL STACK

`components(6, 4, u, v)`

`main()`

Code Trace: Counting Connected components





count = 6

	0	1	2	3	4	5
parent	-1	-1	-1	-1	-1	-1

Edge (0, 1): find2(0) = 0, find2(1) = 1.

Step 3 / 11

```
1  int components(int n, int m, int u[], int v[]) {
2    for (int i = 0; i < n; i++) parent[i] = -1;
3    int count = n;              /* all separate */
4    for (int e = 0; e < m; e++) {
▶      int a = find2(u[e]), b = find2(v[e]);
6      if (a != b) {             /* new component */
7        union2(a, b);
8        count--;
9      }
10   }
11   return count;
12 }
```

VARIABLES

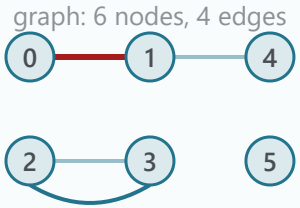
e = 0 | u[e] = 0 | v[e] = 1 | a = 0 | b = 1 | count = 6

CALL STACK

components(6, 4, u, v)

main()

Code Trace: Counting Connected components





count = 5

	0	1	2	3	4	5
parent	-2	0	-1	-1	-1	-1

0 ≠ 1: different components → **union2(0, 1)**, count = 5.

Step 4 / 11

```
1  int components(int n, int m, int u[], int v[]) {
2    for (int i = 0; i < n; i++) parent[i] = -1;
3    int count = n;              /* all separate */
4    for (int e = 0; e < m; e++) {
5      int a = find2(u[e]), b = find2(v[e]);
▶      if (a != b) {             /* new component */
▶        union2(a, b);
▶        count--;
9      }
10   }
11   return count;
12 }
```

VARIABLES

`e = 0` `a = 0` `b = 1`

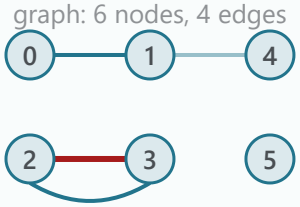
`count = 5`

CALL STACK

`components(6, 4, u, v)`

`main()`

Code Trace: Counting Connected components





count = 5

	0	1	2	3	4	5
parent	-2	0	-1	-1	-1	-1

Edge (2, 3): find2(2) = 2, find2(3) = 3.

Step 5 / 11

```
1  int components(int n, int m, int u[], int v[]) {
2    for (int i = 0; i < n; i++) parent[i] = -1;
3    int count = n;              /* all separate */
4    for (int e = 0; e < m; e++) {
▶      int a = find2(u[e]), b = find2(v[e]);
6      if (a != b) {             /* new component */
7        union2(a, b);
8        count--;
9      }
10   }
11   return count;
12 }
```

VARIABLES

`e = 1` `u[e] = 2` `v[e] = 3`

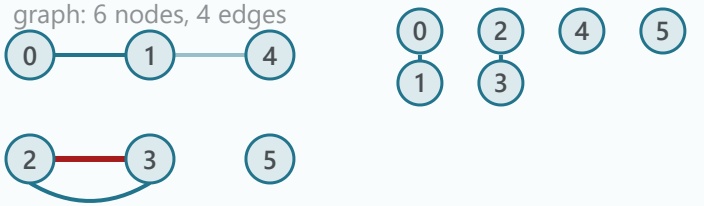
`a = 2` `b = 3` `count = 5`

CALL STACK

`components(6, 4, u, v)`

`main()`

Code Trace: Counting Connected components



count = 4

	0	1	2	3	4	5
parent	-2	0	-2	2	-1	-1

2 ≠ 3: different components → **union2(2, 3)**, count = 4.

Step 6 / 11

```
1  int components(int n, int m, int u[], int v[]) {
2    for (int i = 0; i < n; i++) parent[i] = -1;
3    int count = n;                /* all separate */
4    for (int e = 0; e < m; e++) {
5      int a = find2(u[e]), b = find2(v[e]);
▶      if (a != b) {               /* new component */
▶        union2(a, b);
▶        count--;
9      }
10   }
11   return count;
12 }
```

VARIABLES

`e = 1` `a = 2` `b = 3`

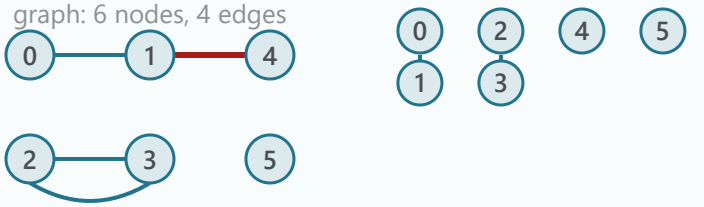
`count = 4`

CALL STACK

`components(6, 4, u, v)`

`main()`

Code Trace: Counting Connected components



count = 4

	0	1	2	3	4	5
parent	-2	0	-2	2	-1	-1

Edge (1, 4): find2(1) = 0, find2(4) = 4.

Step 7 / 11

```
1  int components(int n, int m, int u[], int v[]) {
2    for (int i = 0; i < n; i++) parent[i] = -1;
3    int count = n;              /* all separate */
4    for (int e = 0; e < m; e++) {
▶      int a = find2(u[e]), b = find2(v[e]);
6      if (a != b) {             /* new component */
7        union2(a, b);
8        count--;
9      }
10   }
11   return count;
12 }
```

VARIABLES

`e = 2` `u[e] = 1` `v[e] = 4`

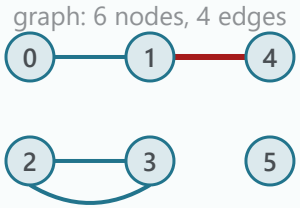
`a = 0` `b = 4` `count = 4`

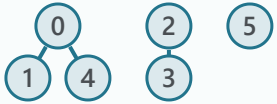
CALL STACK

`components(6, 4, u, v)`

`main()`

Code Trace: Counting Connected components





count = 3

	0	1	2	3	4	5
parent	-3	0	-2	2	0	-1

0 ≠ 4: different components → **union2(0, 4)**, count = 3.

Step 8 / 11

```
1  int components(int n, int m, int u[], int v[]) {
2    for (int i = 0; i < n; i++) parent[i] = -1;
3    int count = n;              /* all separate */
4    for (int e = 0; e < m; e++) {
5      int a = find2(u[e]), b = find2(v[e]);
▶      if (a != b) {             /* new component */
▶        union2(a, b);
▶        count--;
9      }
10   }
11   return count;
12 }
```

VARIABLES

e = 2 | a = 0 | b = 4

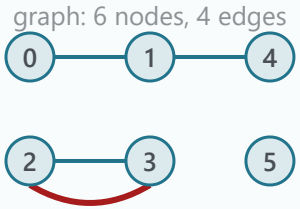
count = 3

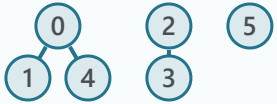
CALL STACK

components(6, 4, u, v)

main()

Code Trace: Counting Connected components





count = 3

	0	1	2	3	4	5
parent	-3	0	-2	2	0	-1

Edge (3, 2): find2(3) = 2, find2(2) = 2.

Step 9 / 11

```
1  int components(int n, int m, int u[], int v[]) {
2    for (int i = 0; i < n; i++) parent[i] = -1;
3    int count = n;                /* all separate */
4    for (int e = 0; e < m; e++) {
▶      int a = find2(u[e]), b = find2(v[e]);
6      if (a != b) {               /* new component */
7        union2(a, b);
8        count--;
9      }
10   }
11   return count;
12 }
```

VARIABLES

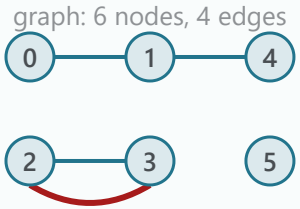
`e = 3` `u[e] = 3` `v[e] = 2` `a = 2` `b = 2` `count = 3`

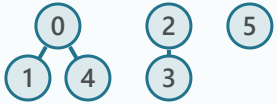
CALL STACK

`components(6, 4, u, v)`

`main()`

Code Trace: Counting Connected components





count = 3

	0	1	2	3	4	5
parent	-3	0	-2	2	0	-1

a == b = 2: 3 and 2 are already in one component; the edge closes a **cycle** and the count stays.

Step 10 / 11

```
1  int components(int n, int m, int u[], int v[]) {
2    for (int i = 0; i < n; i++) parent[i] = -1;
3    int count = n;              /* all separate */
4    for (int e = 0; e < m; e++) {
5      int a = find2(u[e]), b = find2(v[e]);
▶      if (a != b) {             /* new component */
7        union2(a, b);
8        count--;
9      }
10   }
11   return count;
12 }
```

VARIABLES

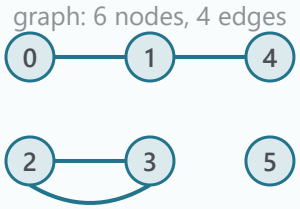
e = 3 | a = 2 | b = 2 | count = 3

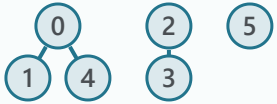
CALL STACK

components(6, 4, u, v)

main()

Code Trace: Counting Connected components





count = 3

	0	1	2	3	4	5
parent	-3	0	-2	2	0	-1

return 3: the components are {0,1,4}, {2,3}, {5}. Total O(n + m·α).

Step 11 / 11

```
1  int components(int n, int m, int u[], int v[]) {
2    for (int i = 0; i < n; i++) parent[i] = -1;
3    int count = n;              /* all separate */
4    for (int e = 0; e < m; e++) {
5      int a = find2(u[e]), b = find2(v[e]);
6      if (a != b) {             /* new component */
7        union2(a, b);
8        count--;
9      }
10   }
▶    return count;
12 }
```

VARIABLES

`count = 3`

CALL STACK

`components(6, 4, u, v)`

`main()`

APPLICATIONS

Where Else Is Union-Find Used?

Equivalence classes
Derive the classes from statements i ≡ j (Horowitz's equivalence problem).

Network connectivity
Answer "are these two computers connected?" online (Sedgewick's opening example).

Image processing
Group adjacent pixels of the same color into regions (connected-component labeling).

Percolation
Is there an open path from top to bottom of a grid? The core of Monte Carlo experiments.

Compilers
Unification in type inference; alias analysis.

Offline LCA
Tarjan's lowest common ancestor algorithm.

Kruskal's algorithm: sort the edges by weight; if find gives different roots for the ends of an edge, take the edge and unite the two sets; if it gives the same root, the edge closes a cycle and is rejected. Cost: sorting $O(m \lg m)$ + union-find $\approx O(m \cdot \alpha)$.

Lab: Counting Components with Union-Find

6 nodes, 4 edges

Input (n=6, edges 0-1, 2-3) `n=6, 0-1, 2-3, 1-4, 3-2` Apply

CODE: components

```
 1 int components(int n, int m,
 2                int u[], int v[]) {
 3   for (int i = 0; i < n; i++) parent[i] = -1;
 4   int count = n; /* all separate */
 5   for (int e = 0; e < m; e++) {
 6     int a = find2(u[e]),
 7         b = find2(v[e]);
 8     if (a != b) { /* new component */
 9       union2(a, b);
10       count--;
11     }
12   }
13   return count;
14 }
```



`count 6`

OUTPUT (none yet)

`components(6, 4, u, v)` : edges (0,1), (2,3), (1,4), (3,2). If the two ends of an edge are in different sets, the sets are merged.

SUMMARY

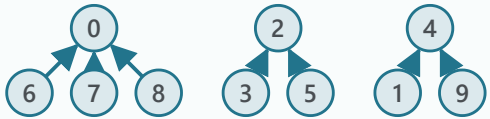
Exam Question: Contents of the Parent Array

EXAM QUESTION

Given union2 and find2, and the sets $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, $S_3 = \{2, 3, 5\}$ (the trees in the figure):

(i) What does the `parent` array contain initially?

(ii) What does `parent` contain after `union2(4, 2); find2(5); union2(0, 4); find2(7); find2(3);` ? *Note: if (i) is wrong, (ii) is not graded.*



▶ **Solution**

SUMMARY

Complexity Summary

Version	Union	Find (worst)	n−1 unions + m finds
union1 + find1	O(1)	O(n)	O(n + m·n)
union2 (weighted)	O(1)	O(lg n)	O(n + m lg n)
union2 + find2	O(1)	O(lg n) per call	O(n + m·α(m+n, n))

Memory: just the `parent` array of n integers. Since $\alpha \leq 4$, the last row is linear in practice.

A union alone is always O(1); the cost is set by the tree depth, that is, by find. The weighting rule bounds the depth; the collapsing rule shortens the paths.

SUMMARY

Common Mistakes

1. Passing an arbitrary node instead of a root to union: `parent[i] = j` cuts off i's subtree.
2. Uniting a root with itself (i == j): the root points to itself, an infinite loop.
3. Misreading the comparison of negative counts in the weighting rule (−2 > −4 means “fewer nodes”).
4. Forgetting to update the node count (p) at the root after a union.
5. Relinking in find2 without saving the parent (s) first: the rest of the path is lost.
6. Not filling the `parent` array with −1 at the start.

SUMMARY

Exercises

1. n = 6, parent all −1. What is parent after `union2(0,1); union2(2,3); union2(0,2); union2(4,5)` ?
2. With the weighting rule, what is the maximum depth of a 16-node tree?
3. Write a union rule that stores the **height** (rank) instead of the weight. What changes?
4. Write find2 recursively (one-line path compression).
5. Why does rejecting an edge in Kruskal's algorithm mean a cycle?
6. What does it cost to count the components of a graph with n nodes and m edges?

▶ **Show answers**

SUMMARY

Chapter Summary

Model

Disjoint sets = trees with child-to-parent links, named by their roots.

Representation

A single `parent` array; a negative value marks a root.

Worst case

Linking without a rule can build a chain: n finds cost $O(n^2)$.

Weighting rule

Smaller tree under larger: depth ≤ ⌊lg m⌋ + 1.

Collapsing rule

The path is linked to the root; later finds take one step.

Result

Together: $\alpha \leq 4$ per operation, almost $O(1)$.

SUMMARY

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Section 5.10 (Set Representation).
- T. H. Cormen et al., *Introduction to Algorithms* (CLRS), chapter “Data Structures for Disjoint Sets”.
- R. Sedgewick, *Algorithms in C*, Chapter 1 (connectivity with union-find).
- R. E. Tarjan, J. van Leeuwen, “Worst-case analysis of set union algorithms”, *J. ACM* 31(2), 1984.

← Chapter 15: Graphs

Course page

Chapter 17: Minimum Spanning Trees

The End

Chapter 16: Sets and Union-Find

Next chapter: Chapter 17: Minimum Spanning Trees · Course page