Chapter 13
Sets and Union-Find

13.1 Sets and Set Operations

※ 13.1.1 *Set Terminology*

A **set** is a well-defined collection of *distinct* elements. In practice, sets are often used to check if a specific value exists within a group of elements. However, before we begin discussing use cases of sets, let us first introduce some set terminology:

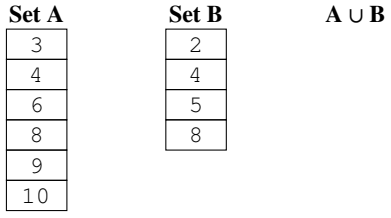
- The **union** ($\cup$) of two sets consists of all elements that are in one set or the other. For example, if set A consists of the elements {1, 2, 3} and set B consists of the elements {1, 3, 5}, the union of sets A and B (denoted as A $\cup$ B) is the set consisting of the elements {1, 2, 3, 5}.
- The **intersection** ($\cap$) of two sets consists of all elements that are members of both sets. Using the same example as above, the intersection of sets A and B (denoted as A $\cap$ B) consists of the elements {1, 3}, since only 1 and 3 are members of both sets A and B.
- The **set difference** ($\setminus$ or –) of two sets consists of all elements that are members of one set but not the other. In the provided example, the set difference A $\setminus$ B (or A – B) consists of the element {2}, since 2 is the only element in set A that is not in set B. The set difference B $\setminus$ A (or B – A) consists of the element {5}, since 5 is the only element in set B that is not in set A.
- The **symmetric difference** ($\oplus$) of two sets consists of all elements that are in either set but not both. In the provided example, the symmetric difference A $\oplus$ B consists of the elements {2, 5}, since 2 and 5 are the only elements that exist in only one of A or B.

※ 13.1.2 *Set Implementation*

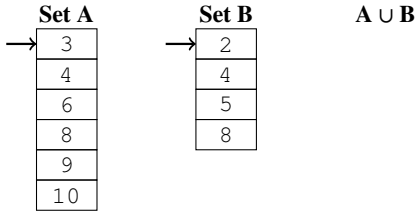
How would you implement a set? There are several ways to implement a set that efficiently supports the operations specified above. For instance, the STL implements its sets using hash tables (`std::unordered_set<>`) and self-balancing trees (`std::set<>`), both of which will be covered in later chapters. In this chapter, we will consider a simple set implementation that uses a sorted vector as its underlying container. Sorted vectors work well because binary search (see chapter 15) can be used to check an element's membership in $\Theta(\log(n))$ time.

To find the union of two sets whose elements are stored in a sorted vector, you would first initialize iterators to the beginning of both vectors. Then, compare the values at each of these iterators and push the smaller element to a destination vector representing the set union. The iterator pointing to the element that was pushed in should then be incremented. If the elements at both iterators are identical, only one copy of the element is pushed to the destination vector and both iterators are incremented (this is done to avoid duplication, which is not allowed in a set). This process is repeated until both iterators point to the end of their respective containers. If one iterator reaches the end before the other, iterate through the remaining elements of the other vector and push them into the destination vector.

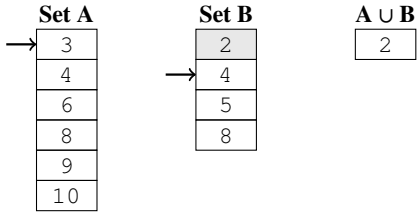
To illustrate this, consider the following example. Here, set A contains the elements {3, 4, 6, 8, 9, 10} and set B contains the elements {2, 4, 5, 8}. Both sets are implemented using a sorted vector. We want to build the set A ∪ B, which stores the elements in the union of A and B.



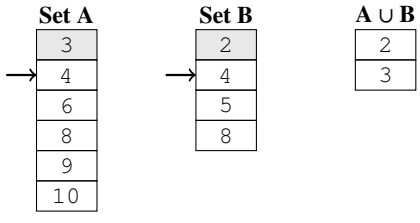
First, we initialize two iterators at the beginning of the two underlying vectors. The two elements at these iterators are then compared, with the smaller value being sent to the output set A ∪ B. This process continues until one of the iterators reaches the end of its respective sorted vector.



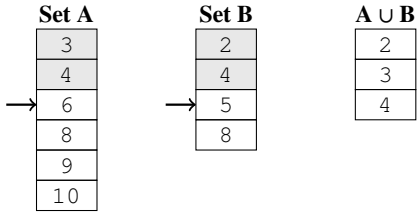
2 is smaller than 3, so we push 2 into the output vector and increment the iterator for set B.



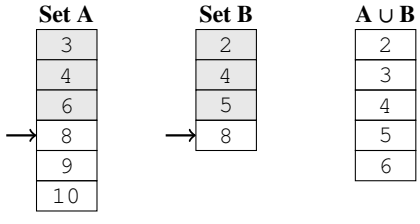
3 is smaller than 4, so we push 3 into the output vector and increment the iterator for set A.



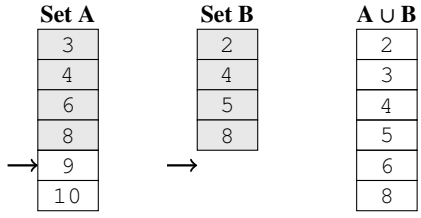
Both iterators now point to the same value, so we push 4 into the output vector and increment both iterators.



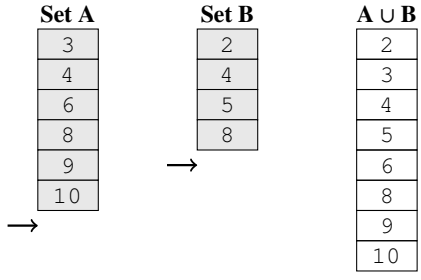
Continuing this process, we push 5 into the union, followed by 6. After these values are added, both iterators would point to 8.



Since both iterators point to the same value, we push 8 into the output vector and increment both iterators.



After 8 is added to the union, notice that the iterator for set B is pointing off the end of the underlying vector. When this happens, all the remaining elements in the other set (in this case, set A) are pushed into the union. This is okay since both sets are traversed in sorted order. If one set reaches the end before another, the other set must contain elements that are larger than the contents of the set that reached the end. Thus, the output vector would remain sorted and all values would remain unique.



A potential implementation of the set union process is shown below. The following code takes in iterators to the underlying containers of two sets and an iterator pointing to the output vector. The comparator `comp(a, b)` returns `true` if `a` is less than `b`.

```
1  template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
2  OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
3                           InputIterator2 first2, InputIterator2 last2,
4                           OutputIterator result, Compare comp) {
5    // loop until one set reaches the end
6    while (first1 != last1 && first2 != last2) {
7      // if element at first1 is less than element at first2, add element
8      // at first1 to result and increment first1
9      if (comp(*first1, *first2)) {
10       *result++ = *first1++;
11     } // if
12     // else if element at first2 is less than element at first1, add
13     // element at first2 to result and increment first2
14     else if (comp(*first2, *first1)) {
15       *result++ = *first2++;
16     } // else if
17     // else if both elements are equal, add one copy of the element to
18     // result and increment both iterators
19     else {
20       *result++ = *first1++;
21       ++first2;
22     } // else
23   } // while
24   // if first2 reaches last2 first, copy remaining elements in container 1
25   // to the result set
26   while (first1 != last1) {
27     *result++ = *first1++;
28   } // while
29   // if first1 reaches last1 first, copy remaining elements in container 2
30   // to the result set
31   while (first2 != last2) {
32     *result++ = *first2++;
33   } // while
34   // returns iterator that points one past the end of the sorted union
35   // this is the behavior expected for standard algorithm functions
36   return result;
37 } // set_union()
```

What is the time complexity of the `set_union()` function? With each iteration, either `first1` or `first2` is incremented. As a result, if m and n are the sizes of the two sets, the algorithm iterates through a total of $m+n$ elements. Thus, `set_union()` runs in $\Theta(m+n)$ time.

How would we change the `set_union()` algorithm to a `set_intersection()` algorithm? When dealing with a set intersection, we only add the element to the output vector if it belongs in both sets. Thus, only the `else` block on line 19 of the previous code should write the element to the output vector. We can also get rid of lines 24 to 33; once the first set reaches the end, we know that the elements remaining in the other set cannot be in the set intersection. This leads us to the following modified code for `set_intersection()`.

```
1  template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
2  OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
3                                  InputIterator2 first2, InputIterator2 last2,
4                                  OutputIterator result, Compare comp) {
5    // loop until one set reaches the end
6    while (first1 != last1 && first2 != last2) {
7      // if element at first1 is less than element at first2, do not add
8      // to result and only increment first1
9      if (comp(*first1, *first2)) {
10       ++first1;
11     } // if
12     // else if element at first2 is less than element at first1, do not
13     // add to result and only increment first2
14     else if (comp(first2, first1)) {
15       ++first2;
16     } // else if
17     // else if both elements are equal, add one copy of the element to
18     // result and increment both iterators
19     else {
20       *result++ = *first1++;
21       ++first2;
22     } // else
23   } // while
24   // return iterator that points one past the end of the sorted union
25   return result;
26 } // set_intersection()
```

The same template can be used to implement `set_difference()` and `set_symmetric_difference()`. The only difference is in the values that are added to the result vector. In the case of `set_difference()`, only the `if` block on line 9 of the `set_union()` code should write its value to the result vector, and the `while` loop on lines 26-28 should be kept.

```
1  template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
2  OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
3                                InputIterator2 first2, InputIterator2 last2,
4                                OutputIterator result, Compare comp) {
5    // loop until one set reaches the end
6    while (first1 != last1 && first2 != last2) {
7      // if element at first1 is less than element at first2, add element
8      // at first1 to result and increment first1
9      if (comp(*first1, *first2)) {
10       *result++ = *first1++;
11     } // if
12     // else if element at first2 is less than element at first1, increment first2
13     else if (comp(*first2, *first1)) {
14       ++first2;
15     } // else if
16     // else if both elements are equal, increment both iterators
17     else {
18       ++first1;
19       ++first2;
20     } // else
21   } // while
22   // if first2 reaches last2 first, copy remaining elts in ctn 1 to result set
23   while (first1 != last1) {
24     *result++ = *first1++;
25   } // while
26   // returns iterator that points one past the end of the sorted union
27   return result;
28 } // set_difference()
```

The implementation of `set_symmetric_difference()` (elements that exist in either set but not both) is also similar to the implementation of `set_union()`, except that the `else` block on line 19 should *not* write a value to the result vector. The code is shown below:

```
1   template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
2   OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
3                                           InputIterator2 first2, InputIterator2 last2,
4                                           OutputIterator result, Compare comp) {
5     while (first1 != last1 && first2 != last2) {
6       if (comp(*first1, *first2)) {
7         *result++ = *first1++;
8       } // if
9       else if (comp(*first2, *first1)) {
10        *result++ = *first2++;
11      } // else if
12      else {
13        ++first1;
14        ++first2;
15      } // else
16    } // while
17    while (first1 != last1) {
18      *result++ = *first1++;
19    } // while
20    while (first2 != last2) {
21      *result++ = *first2++;
22    } // while
23    return result;
24  } // set_symmetric_difference()
```

※ 13.1.3 *Set Operations in the STL*

These four set operations for sorted containers: `set_union()`, `set_intersection()`, `set_difference()`, and `set_symmetric_difference()`, are all implemented in the STL's `<algorithm>` library and behave similarly to the examples above. All four functions take in iterator ranges to two *sorted* sequence containers, an output iterator that points to the destination container, and an optional comparator. These four functions all return an iterator one past the last element of the constructed range.

```
template <typename InputIterator1, typename InputIterator2, typename OutputIterator, typename Compare>
OutputIterator std::set_union(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result, Compare comp);
```

Constructs a sorted range beginning at `result` with the contents of the set union of the elements in the two sorted ranges `[first1, last1)` and `[first2, last2)` and returns an iterator one past the last element in the constructed range. The comparator is optional, and `operator<` is used by default if it is not specified.

```
template <typename InputIterator1, typename InputIterator2, typename OutputIterator, typename Compare>
OutputIterator std::set_intersection(InputIterator1 first1, InputIterator1 last1,
                                     InputIterator2 first2, InputIterator2 last2,
                                     OutputIterator result, Compare comp);
```

Constructs a sorted range beginning at `result` with the contents of the set intersection of the elements in the two sorted ranges `[first1, last1)` and `[first2, last2)` and returns an iterator one past the last element in the constructed range. The comparator is optional, and `operator<` is used if it is not specified.

```
template <typename InputIterator1, typename InputIterator2, typename OutputIterator, typename Compare>
OutputIterator std::set_difference(InputIterator1 first1, InputIterator1 last1,
                                   InputIterator2 first2, InputIterator2 last2,
                                   OutputIterator result, Compare comp);
```

Constructs a sorted range beginning at `result` with the contents of the set difference of the elements in the sorted range `[first1, last1)` with respect to the elements in the sorted range `[first2, last2)` and returns an iterator one past the last element in the constructed range. The comparator is optional, and `operator<` is used by default if it is not specified.

```
template <typename InputIterator1, typename InputIterator2, typename OutputIterator, typename Compare>
OutputIterator std::set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                                             InputIterator2 first2, InputIterator2 last2,
                                             OutputIterator result, Compare comp);
```

Constructs a sorted range beginning at `result` with the contents of the set symmetric difference of the elements in the two sorted ranges `[first1, last1)` and `[first2, last2)` and returns an iterator one past the last element in the constructed range. The comparator is optional, and `operator<` is used if it is not specified.

For example, the following code uses `std::set_union()` in the `<algorithm>` library.

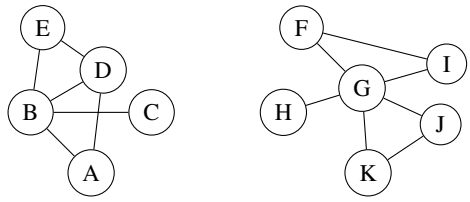
```
1   std::vector<int32_t> v1 = {1, 2, 3};
2   std::vector<int32_t> v2 = {1, 3, 5};
3   std::vector<int32_t> v_union;
4   std::set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(v_union));
5   for (int32_t val : v_union) {
6     std::cout << val << " ";  // prints 1 2 3 5
7   } // for val
```

13.2 Union-Find and Path Compression

※ 13.2.1 *Disjoint Sets and the Union-Find Data Structure*

In the previous section, we introduced the concept of a set. One benefit of using a set is that we are able to quickly determine if any element is part of that set. However, certain problems go beyond simply querying if an element exists in a set.

One such problem involves something known as a **disjoint set**. We consider two sets to be disjoint if they do not share any elements. In disjoint set problems, we are often required to organize a collection of objects into different groups and efficiently determine whether two objects belong to the same group. As an example, suppose we wanted to implement a program that can determine whether two computers in a network are connected to each other. Consider the following diagram, where each node represents a computer, and each edge represents a network connection that exists between two computers.

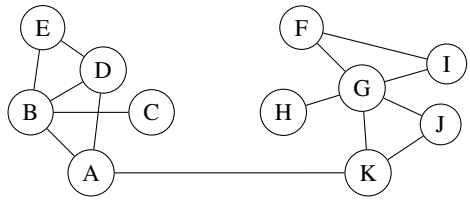


Given this setup, we have two disjoint sets, one that includes computers A-E, and one that includes computers F-K.

Suppose we were asked if data can be sent through the network from computer C to computer E. How would we answer this question? In this case, the answer would be yes, since computers C and E belong to the same disjoint set; to send data from C to E, we could first send the data from C to B, and then from B to E.

What if we were instead asked if data could be sent through the network from computer C to computer I? The answer to this question would be no, since computers and C and I belong to two different disjoint sets — there is no way to use the existing connections to travel from C to I.

Now, let's suppose we added a connection between computers A and K, as shown below.

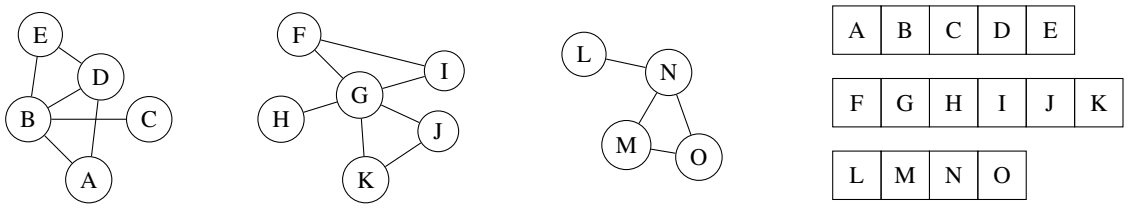


What happens to our answers to the previous two questions? In the case of computers C and E, the answer would still be yes, since C and E still belong to the same disjoint set. However, the answer for C and I would change to yes, since the new connection ended up joining C and I into the same disjoint set! Even though neither C nor E had their connections directly modified, an external connection caused their disjoint set membership to be the same.

If we want to efficiently solve problems involving disjoint sets, we would need to store our data in a container that allows us to keep track of multiple disjoint sets and membership within these sets. This is done using a container known as a **union-find data structure** (alternatively known as a *disjoint-set data structure*). The union-find data structure is a container that relies on two primary operations, union and find:

- `union_set(x, y)` is used to merge the set containing element `x` with the set containing element `y`. Using the above example, `union_set(A, K)` would add a connection between computers A and K.[1]
- `find_set(x)` is used to identify the disjoint set that element `x` belongs to.

How should we implement a union-find data structure to support efficient union and find operations? The simplest approach would be to use the same sorted vector approach to keep track of our disjoint sets. Each disjoint set would be stored in its own sorted vector, as shown:



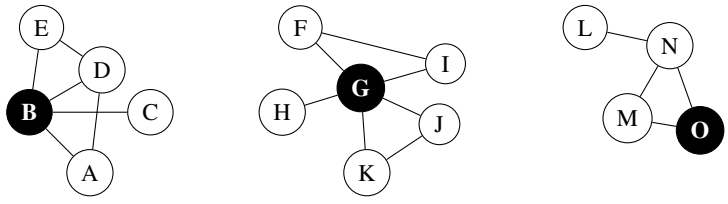
How would union and find be implemented using this approach? If `union_set()` is called on two elements, we would first check if the two elements are in the same sorted vector. If they are not, we would have to shift all the elements in one vector to the other to merge the two sets. For example, if we wanted to union computers A and K, we would have to merge the first two sorted vectors (A-E and F-K). This takes linear time. If `find_set()` is called on an element, we would have to check every vector for the existence of that element. In the worst case, this also takes linear time. Because both union and find are linear-time operations using this approach, this implementation is inefficient, especially as the number of disjoint sets in the container increases. It would be better to store all the elements in the *same* container, rather than a separate container for each disjoint set.

[1] The reason we are using `union_set()` and not `union()` is that the keyword `union` is reserved in C++ and cannot be used to name a custom function or variable. Similarly, `set_union()` is not used because that is the name of a function in the STL algorithm library.

However, if we do store all our data in a single container, we will need to store additional information to keep track of which disjoint set each element belongs to. We can do this using a *representative system*, where each element in the union-find data structure remembers a unique representative that identifies which set it belongs in.

To use an analogy, suppose you are in a room full of people from all over the world, and you want to organize the people into disjoint sets based on citizenship. If everyone is mingled together, how would you be able to identify which country everyone is from? One way to solve this problem is to ask each person who the leader of their country is. If they answer with the name of the United States president, they must be from the United States. If they answer with the name of the Canadian Prime Minister, they must be from Canada. You can then use this information to organize the people into their correct groups.

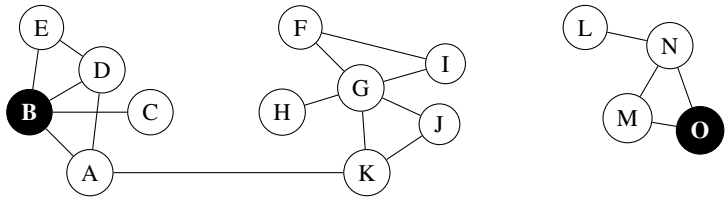
The representative system works similarly for data in a union-find data structure. Each element in a disjoint set is assigned a representative that acts as the "leader" of its set. Every member of the set then remembers who their representative is. An example is shown below, where computers B, G, and O are arbitrarily chosen as the representative of each of the disjoint sets.



Element	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Representative	B	B	B	B	B	G	G	G	G	G	G	O	O	O	O

Here, computer B serves as the representative of all computers in the first disjoint set, computer G serves as the representative of all computers in the second disjoint set, and computer O serves as the representative of all computers in the third disjoint set. Compared to the previous implementation that kept a sorted vector for each disjoint set, the `find_set()` operation is much more efficient with this approach. To determine which set an element belongs to, `find_set()` would just need to check that element's representative. To determine if two elements are in the same disjoint set, simply compare their representatives. This procedure of forcing each element to remember its ultimate representative is known as the *quick-find* implementation of union-find, since `find_set()` can be completed in $\Theta(1)$ time.

However, the union operation is still inefficient here. If we wanted to union computers A and K together, we would have to change the representatives of all the elements in one of the sets — this would require a linear traversal of the container.



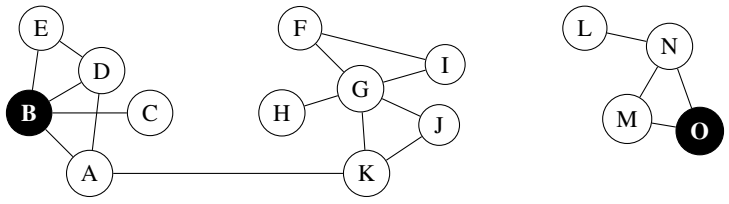
Element	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Representative	B	B	B	B	B	**B**	**B**	**B**	**B**	**B**	**B**	O	O	O	O

We can make the union process faster by removing the need for each element to remember its ultimate representative. It is okay if each element remembers an element that may *not* be the ultimate representative itself, as long as the element it remembers either (1) knows who the ultimate representative is itself or (2) remembers another element that knows who the ultimate representative is. The ultimate representative serves as the leader of the entire disjoint set, and its representative is itself. This implementation is known as the *quick-union* implementation of union-find.

Returning to the citizenship analogy, this process is similar to asking everyone to name a person who is a citizen of the same country rather than a direct leader (or ultimate representative). The idea here is that we will eventually be able to figure out which country each person belongs to just by following the chain of representatives (e.g., if Alice says Bob is her representative, Bob says Cathy is his representative, and Cathy says that the president of the United States is her representative, we would know that Alice, Bob, and Cathy are all Americans).

When using this implementation, a call to `union_set()` would only need to modify the ultimate representative of one of the disjoint sets that was merged. In the example, the ultimate representative of the first disjoint set was computer B, and the ultimate representative of the second disjoint set was computer G. If a connection were added between computers A and K, we would first check if A and K are already part the of the same disjoint set by comparing their ultimate representatives using two calls to `find_set()`. If they are not part of the same set, we would modify the ultimate representative of either A or K so that it refers to the ultimate representative of the other disjoint set.

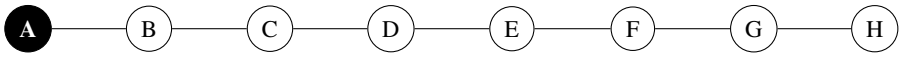
In the following example, since K's ultimate representative is G and A's ultimate representative is B, we could change G's representative to B. Note that we could have changed B's representative to G instead; the decision to change G's representative instead of B's was made arbitrarily.



Element	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Representative	B	B	B	B	B	G	**B**	G	G	G	G	O	O	O	O

This reduces the amount of work needed for union, since we no longer need to traverse the entire container and reset the representative of every element whose representative was G. However, the find operation will likely be slower than before, since we may need to traverse an entire chain of elements to find the disjoint set that an element belongs to. For instance, suppose we called find on computer F. We know that F's representative is G. However, G's representative was changed to B. Because of this, we know that computer F is a member of the disjoint set whose ultimate representative is B (we know B is an ultimate representative because its representative is itself).

However, this is a simple case. Since we removed the requirement for each element to directly remember its ultimate representative, we could end up with something like this:



Element	A	B	C	D	E	F	G	H
Representative	A	A	B	C	D	E	F	G

Suppose we called `find_set()` on computer H using this setup. H's representative is G, but G's representative is F, and F's representative is E, and E's representative is D, … we would have to traverse through every element in the container to figure out that H's ultimate representative is computer A. This is a worst-case $\Theta(n)$ traversal for a single find operation! In addition, since union itself calls find twice (to determine the ultimate representatives of the two elements to join), the union operation here also runs in $\Theta(n)$ time (even if it is slightly faster on average in most situations). How can we adjust our implementation so that both union and find can be done efficiently? The solution is to use a strategy known as *path compression*.

※ 13.2.2 *Path Compression*

When we forced each element to directly remember its ultimate representative, find could be done quickly, but union was inefficient. However, if we removed this requirement and allowed elements to store any representative that belongs to its own disjoint set, union became faster but find became less efficient.

A key insight to notice here is that, even though it took a $\Theta(n)$ traversal to learn that computer H's ultimate representative was computer A, this work only needs to be done once. If we call `find_set()` on H again, we should not need to traverse the entire chain of elements again; we already know that its ultimate representative is A! As a result, after calling `find_set()` on H the first time, we can speed up subsequent calls to `find_set()` by changing H's representative to A, so that further `find_set()` calls on H would no longer require a $\Theta(n)$ traversal.

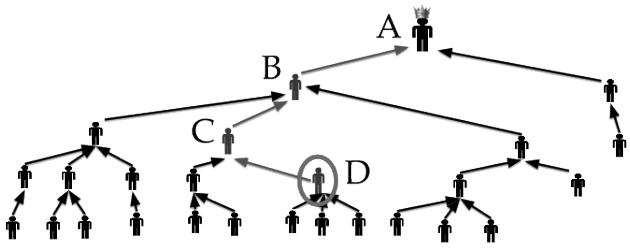
Element	A	B	C	D	E	F	G	H
Representative	A	A	B	C	D	E	F	**A**

However, that is not all we can do! By traveling down the entire chain of elements, we learned more than the fact that H's ultimate representative is A. Because H's old representative was G, H and G must be part of the same disjoint set. Thus, if H's ultimate representative is A, G's ultimate representation must also be A. And since G's representative is F, G and F must be part of the same disjoint set, so F's ultimate representative must also be A. Not only can we change the representative of H to A, but we can change the representative of *every element on the entire chain* to A as well!

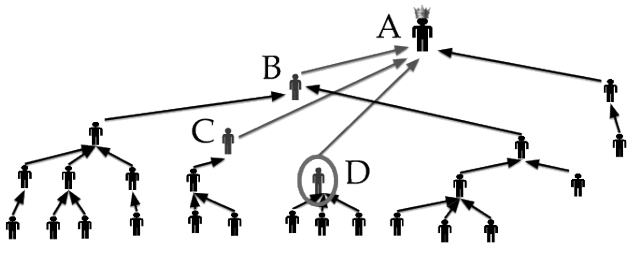
Element	A	B	C	D	E	F	G	H
Representative	A	A	**A**	**A**	**A**	**A**	**A**	**A**

Now, every subsequent call to `find_set()` can be done quicker since every node now stores its ultimate representative. This process of moving elements closer to its ultimate representative — so that future calls to find will require less work — is known as **path compression**. The benefit of path compression is further illustrated in the figures below.

Suppose we have a large disjoint set of people in our union-find container. Person A is the ultimate representative of everyone in the set. Person B has person A as their representative, person C has person B as their representative, and person D has person C as their representative.

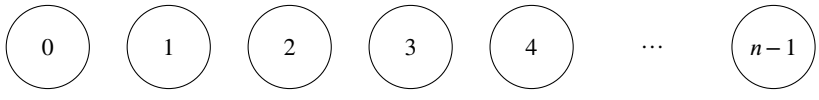


If we call `find_set()` on person D to identify the disjoint set they belong to, we would move up the tree from D and discover that person A is the ultimate representative. However, after this call to find, we would also discover that persons B, C, and D all share the same ultimate representative! If we ever call `find_set()` on B, C, or D, we would not want to repeat the work of walking up the tree to find the ultimate representative. With path compression, we can bring B, C, and D closer to their ultimate representative so that future calls to find run faster. This is done by updating each of their representatives to person A, as shown below:



13.3 Implementing a Union-Find Container

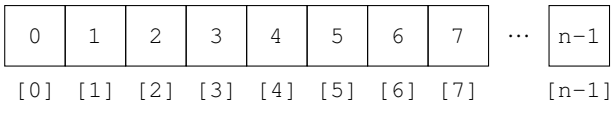
In this section, we will implement a union-find data structure using the concepts covered in the previous section. Suppose we are given a collection of n elements that we want to organize into disjoint sets:



As mentioned before, the union-find container will need to keep track of each element's representative, so we will initialize an underlying vector of indices as a private member variable of the `UnionFind` class. Each index of the vector stores the representative of the element at that index.

```
1  class UnionFind {
2    // vector that keeps track of representatives
3    // the value at index i is the index of the ith element's representative
4    std::vector<int32_t> reps;
5    ...
6  };
```

How should the `reps` vector be initialized? We do not want to default initialize all the representatives to 0, since that would imply that element 0 is the representative of every element in the collection. Instead, we want each element to be in its own disjoint set (of size 1) when the container is first initialized. This means that every element starts with *itself* as its representative (that is, the value at index `i` of the `reps` vector should be initialized to `i`).



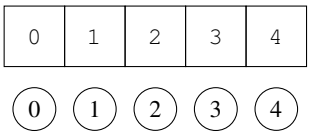
The following constructor takes in a size and initializes the representatives in the `reps` vector:

```
1   class UnionFind {
2     std::vector<int32_t> reps;
3   public:
4     // the reps vector should have each index initialized to the
5     // value of the index itself
6     UnionFind(int32_t size) {
7       reps.reserve(size);
8       for (int32_t i = 0; i < size; ++i) {
9         // at the beginning, every node represents itself
10        reps.push_back(i);
11      } // for i
12    } // UnionFind()
13    ...
14  };
```

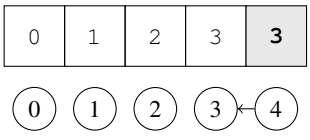
The `for` loop can also be condensed into one line using the `std::iota()` function:

```
std::iota(reps.begin(), reps.end(), 0);
```

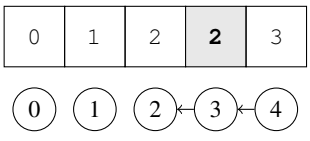
Now, we have to implement the `union_set()` and `find_set()` methods for our union-find container. To implement these, let's first look at how these operations should operate visually. Suppose we had five elements that all start off disjoint in their own set.



Suppose we called `union_set()` on elements 3 and 4. When this happens, 3 and 4 are merged into the same set. To reflect this change in our union-find container, we would need to change 4's representative to 3 to indicate that 3 and 4 are together in the same set (note that this choice to update 4 is arbitrary; we could have instead updated 3's representative to 4 if we wanted to).

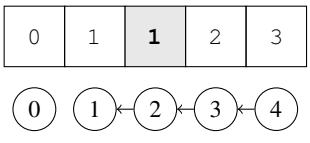


Now, suppose we union elements 2 and 3. To update our union-find container, we will update 3's representative to 2 to indicate that 2 and 3 belong to the same set.



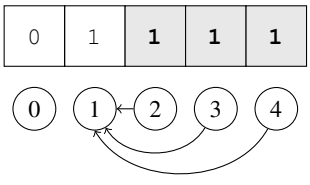
Notice that elements 2, 3, and 4 form a set in this container, with 2 as the ultimate representative of every element in this set. We know this because 2 is the only element in this set whose ultimate representative is itself.

Continuing on, if we unioned together elements 1 and 2, 2's representative would be updated to 1:

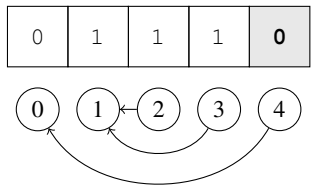


Now, let's call `find_set()` on element 4. When we call find, we want to return the ultimate representative of 4, which serves as an identifier for the disjoint set it belongs in. To do this, we will search up the chain of representatives from 4 until we encounter an element whose representative is itself. Here, 4's representative is 3, 3's representative is 2, 2's representative is 1, and 1's representative is itself. Thus, 1 is the ultimate representative is 4.

However, that is not all we have to do. If we were to call find on 4 again, we would not want to look down its chain of representatives all over again. To address this, we can implement path compression and change the representative of every element along the path from 4 to 1 to the ultimate representative of 1.

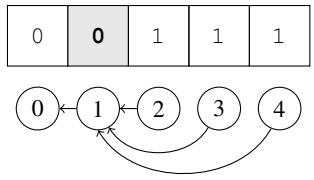


Now, let's union together 0 and 4. This is a bit tricky, since we cannot just change 4's representative to 0. Doing so would produce the following incorrect result:



If we just changed 4's representative, 4 would not longer be in the same set as 1, 2, and 3 after the union call. This is invalid, since 4 was moved out of its original set, which should not happen. Instead, the intended behavior of unioning 0 and 4 is for all five elements to be in the same set.

To fix this issue, we should update the representative of 4's *ultimate representative*, rather than 4 itself. In this case, 1 is the ultimate representative of 4, so we modify 1's representative to 0. By doing so, 4 is able to join 0 without breaking its previous connections to 1, 2, and 3.



We now have enough information to implement the `find_set()` and `union_set()` functions of the union-find container. With `find_set()`, we would first find the ultimate representative of a given element by looping through its representatives until we find an element that represents itself. Then, we would utilize path compression to set every element along the chain of representatives we visited to this ultimate representative.

```
1   // find_set(x) returns the ultimate representative of x
2   int32_t find_set(int32_t x) {
3     int32_t current = x;
4     // pass 1: find the ultimate representative
5     while (reps[x] != x) {
6       x = reps[x];
7     } // while
8     // x now stores the ultimate representative
9     // pass 2: path compression, set all reps along path to x
10    while (reps[current] != x) {
11      int32_t next = reps[current];
12      reps[current] = x;
13      current = next;
14    } // while
15    // return the ultimate representative
16    return x;
17  } // find_set()
```

We can also use recursion to shorten the find process into the following lines of code:

```
1   int32_t find_set(int32_t x) {
2     if (x == reps[x]) {
3       return x;
4     } // if
5     reps[x] = find_set(reps[x]);
6     return reps[x];
7   } // find_set()
```

What about `union_set()`? To implement the union operation, we would want to find the ultimate representative of each element we want to union and update one representative to point to the other representative:

```
1   // union_set() unions the elements x and y into the same set
2   void union_set(int32_t x, int32_t y) {
3     int32_t x_rep = find_set(x);
4     int32_t y_rep = find_set(y);
5     reps[y_rep] = x_rep;
6   } // union_set()
```

In the above implementation, we arbitrarily decided that `y`'s representative will always be the one that is updated. This works, but it is not always ideal. In the next section, we will discuss methods to strategically select which representative gets updated in order to further improve the efficiency of union-find.

Example 13.1 The EECS 281 staff is testing out a brand new social media site to promote social interaction among EECS students! Suppose there are a total of n students that are invited to the site, and each student is assigned a unique integer ID from 0 to $n-1$. You are given the number of students in the network, `num_students`, and a vector of activity logs, where each log stores an integer timestamp and the IDs of two students on the site:

```
struct Log {
  int32_t timestamp;
  int32_t id_A;
  int32_t id_B;
};
```

Each `Log` object represents the time when students `id_A` and `id_B` became friends. The timestamp is represented in YYYYMMDD format (e.g., February 26, 2020 is represented as the integer 20200226). This format allows you to compare timestamps by directly comparing the integers themselves. Friendship is also symmetric: if A is friends with B, then B is friends with A.

Let's say that person A is *acquainted* with person B if A and B are in the same friend group; that is, if A is friends with B, or A is a friend of someone acquainted with B. Implement the `earliest_acquaintance()` function, which returns the earliest timestamp for which every person in the network is acquainted with every other person. Return `-1` if there is no such earliest time.

```
int32_t earliest_acquaintance(std::vector<Log>& friendships, int32_t num_students);
```

Example 1: Given `num_students = 5` and `friendships = [{20200229, 2, 3}, {20200227, 1, 4}, {20200303, 0, 3}, {20200228, 0, 4}, {20200301, 1, 2}, {20200226, 0, 2}]`, you would return `20200229`, since that is the first timestamp for which all five students are acquained (on the 29th, person 0 is direct friends with 2 and 4, acquainted to 1 via 4, and acquainted to 3 via 2).

Example 2: Given `num_students = 5` and `friendships = [{20200304, 1, 4}, {20200307, 0, 2}, {20200306, 3, 4}]`, you would return `-1`. This is because person 0 and person 2 are in a separate friend group and are never acquainted with 1, 3, or 4.

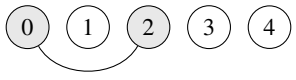
In this example, we want to find the earliest timestamp during which every student on the network is acquainted. The key insight to notice is that two students are acquainted if they are part of the same disjoint set! This makes the problem a good candidate for the union-find data structure.

Let us look at the examples visually. Consider the first example: at the very beginning, no one is friends with anyone else, and thus none of students start off as acquaintances.

Current timestamp: < 20200226

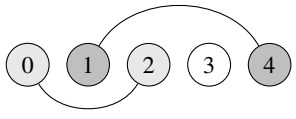
On the 26th, the earliest timestamp in the vector, 0 becomes friends with 2. As a result, these two students become part of the same friend group (i.e., the same disjoint set), and thus are acquainted.

Current timestamp: 20200226



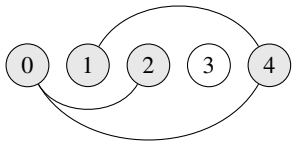
On the 27th, 1 becomes friends with 4, forming another distinct friend group:

Current timestamp: 20200227



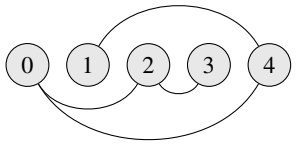
On the 28th, 0 becomes friends with 4. Notice that this combines the two exisiting friend groups; at this point, all but student 3 are acquainted.

Current timestamp: 20200228



On the 29th, 2 becomes friends with 3, which ends up joining 3 with the friend group containing all the other students. There is only one disjoint set remaining after this friendship, so everyone is acquainted, and we can return the timestamp of 20220229.

Current timestamp: 20200229

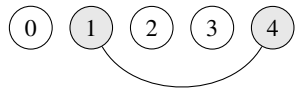


Let's consider the second example. Again, no one is friends with each other at the beginning, so each student starts off in their own disjoint set.

Current timestamp: < 20200304

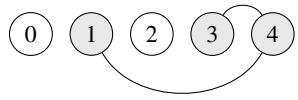
On the 4th, 1 becomes friends with 4, which puts them in the same disjoint set.

Current timestamp: 20200304



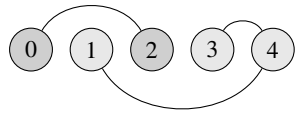
On the 6th, 3 becomes friends with 4, which adds 3 to the disjoint set containing 1 and 4.

Current timestamp: 20200306



On the 7th, 0 becomes friends with 2, forming their own disjoint set.

Current timestamp: 20200307



However, no additional friendships are made. There are multiple disjoint sets left over, so not every person in the network is acquainted with every other person (e.g., 0 and 2 are not acquainted with 1, 3, or 4). Because of this, we will return `-1`.

From these examples, we can come up with the steps needed to solve this problem. Our algorithm is as follows:

1. Sort the vector of logs by timestamp — this allows us to traverse the logs from the earliest timestamp to the latest timestamp.
2. Initialize the students using a union-find container, with each student starting off as its own representative (i.e., each student initially belongs in its own disjoint set).
3. Loop over the vector of logs. Every time you encounter a new friendship, union them together.
4. If only one disjoint set remains, return the timestamp at which this occurs. Otherwise, if all the logs are traversed and there are still multiple disjoint sets, return `-1`.

The final step is not as straightforward as it seems: how do we determine if there is only one disjoint set remaining? This can be done by identifying the number of ultimate representatives that exist in our union-find container. Recall that the ultimate representative is the element that serves as the "leader" of its disjoint set, and they have the unique property of being their own ultimate representative. However, if we want to count the number of elements whose ultimate representative is equal to itself, we would have to iterate over the entire union-find container — this would take $\Theta(n)$ time given n students. Is there a way to do better?

Instead of iterating over all the students and counting the number of ultimate representatives after every new friendship, a more efficient approach would be to count the number of times two students in *different* disjoint sets are unioned together. *This works because the total number of disjoint sets decreases by one every time two non-acquainted students are unioned together.* Since we start off with n disjoint sets (as no one is friends with each other), we are guaranteed to have one disjoint set remaining after making exactly $n-1$ connections between non-acquainted students. Notice that we can easily determine whether two students are non-acquainted or not by calling `find_set()` on each of them and verifying that their ultimate representatives are different. By utilizing this strategy, we would only need to increment or decrement a counter after every friendship, which takes $\Theta(1)$ time.

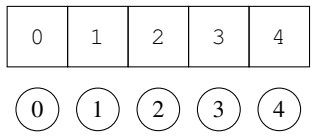
There are two ways to keep track of the counter: you can either start the counter at 0 and increment the counter every time you union two non-acquainted students, stopping once the counter reaches $n-1$; or, you can start the counter at n and decrement the counter every time you union two non-acquainted students, stopping the counter once you reach 1. The first approach counts the number of non-acquainted friendships that are made, while the second approach counts the total number of disjoint sets that exist among the students. Both approaches would get you to the same answer. A full implementation of the problem solution is shown below:

```
1   struct LogComp {
2     bool operator() (const Log& lhs, const Log& rhs) const {
3       return lhs.timestamp < rhs.timestamp;
4     } // operator()()
5   };
6
7   // find the representative (using path compression)
8   int32_t find(std::vector<int32_t>& reps, int32_t id) {
9     if (id == reps[id]) {
10      return id;
11    } // if
12    reps[id] = find(reps, reps[id]);
13    return reps[id];
14  } // find()
15
16  // ... the rest of the solution is continued on the next page ...
```

```
17  int32_t earliest_acquaintance(std::vector<Log>& friendships, int32_t num_students) {
18    std::sort(friendships.begin(), friendships.end(), LogComp()); // can also use a lambda
19    std::vector<int32_t> reps(num_students);
20    int32_t count = 0; // can also start at n and decrement
21    for (int32_t i = 0; i < num_students; ++i) {
22      // set each student to its own representative (can also use std::iota() to save lines)
23      reps[i] = i;
24    } // for i
25    for (const Log& f : friendships) {
26      int32_t rep_A = find(reps, f.id_A);
27      int32_t rep_B = find(reps, f.id_B);
28      // if rep_A and rep_B are different, then A and B are part of different disjoint sets when they
29      // are unioned together, so increment the counter (or decrement if you started the counter at n)
30      if (rep_A != rep_B) {
31        reps[rep_B] = rep_A;
32        ++count;
33      } // if
34      // done when n - 1 connections are made (check count == 1 if decrementing from n)
35      if (count == num_students - 1) {
36        return f.timestamp;
37      } // if
38    } // for f
39    // return -1 if all logs are visited and fewer than n - 1 non-acquainted unions are made
40    return -1;
41  } // earliest_acquaintance()
```

13.4 Union-by-Size and Union-by-Rank (⁎)

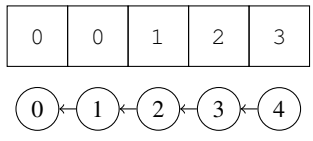
So far, our union implementation blindly sets the ultimate representative of the second disjoint set to that of the first. However, this process could still give a worst-case scenario where union and find both take $\Theta(n)$ time.



Suppose we called the following sequence of union operations:

- `union_set(3, 4)`
- `union_set(2, 3)`
- `union_set(1, 2)`
- `union_set(0, 1)`

If we did this, 4's representative would become 3, 3's representative would become 2, 2's representative would become 1, and 1's representative would become 0. This is shown accordingly:

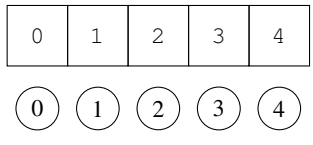


We have now run into a situation where the worst-case complexity of union and find both become $\Theta(n)$. If you wanted to call find on element 4, you would have to do a linear traversal of the entire vector to discover that 4's ultimate representative is 0.

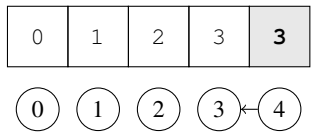
At this point, you may be wondering: shouldn't path compression handle this case? After we do the $\Theta(n)$ traversal, we would change the representatives of all the elements to 0 so that future calls to union and find become cheaper — as such, wouldn't we need to do the traversal only once? Although this is true, the issue is that we may have to do a $\Theta(n)$ traversal in the first place! Even though path compression improves the *amortized* cost of union and find, it does not improve the *worst-case* cost of either operation. That is, path compression does not prevent you from ending up with a worst-case "stick" formation of elements, such as in the example above. To address this, there are two optimization strategies we can use: *union-by-size* and *union-by-rank*.

⁎ 13.4.1 *Union-by-Size (⁎)*

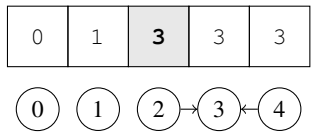
In a **union-by-size** approach, we additionally keep track of the sizes of each disjoint set in our union-find data structure. When two disjoint sets are unioned together, we always update the representative of the smaller set (ties are broken arbitrarily). Consider the same sequence of union operations from above, but this time using union-by-size to determine which representative is updated.



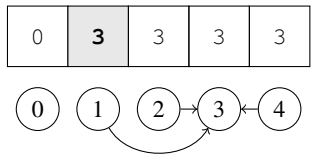
When we union 3 and 4, are are able to arbitrarily select which representative we want to update (since both sets are the same size). In this case, let's suppose we updated 4's representative to be 3.



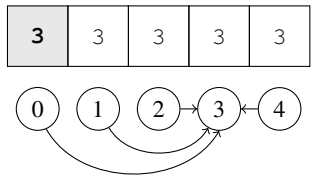
Now, we want to union 2 and 3. The disjoint set that 2 belongs to has a size of 1, and the disjoint set that 3 belongs to has a size of 2. Since 2's disjoint set is smaller, we assign 2's representative to 3's ultimate representative, or 3.



Now, we want to union 1 and 2. The disjoint set that 1 belongs to has a size of 1, and the disjoint set that 2 belongs to has a size of 3. Since 1's disjoint set is smaller, we assign 1's representative to 2's ultimate representative, or 3.



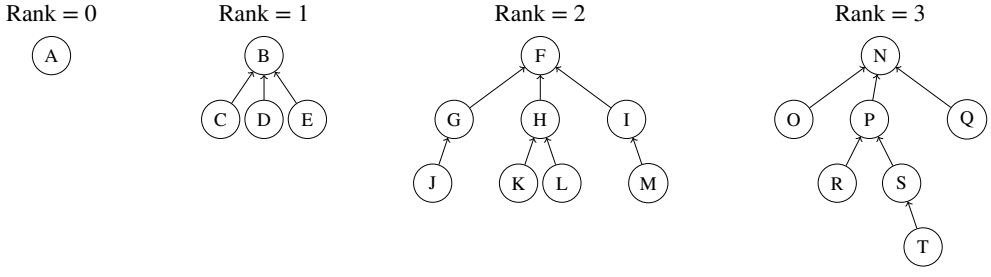
Now, we want to union 0 and 1. The disjoint set that 0 belongs to has a size of 1, and the disjoint set that 2 belongs to has a size of 4. Since 0's disjoint set is smaller, we assign 0's representative to 1's ultimate representative, or 3.



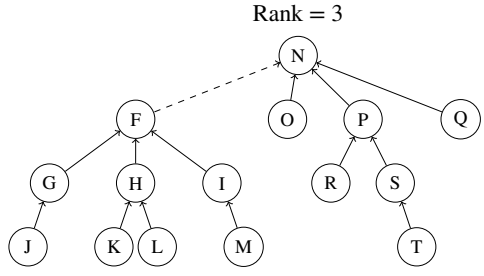
Notice here that we no longer have a stick-like chain of representatives. In fact, if union-by-size is used, the worst-case time complexity of a single union or find call drops from $\Theta(n)$ to $\Theta(\log(n))$, where n is the number of elements in the container. This is because union-by-size ensures that the longest possible chain of representatives has a length of at most $\Theta(\log(n))$.

※ 13.4.2 *Union-by-Rank* (✻)

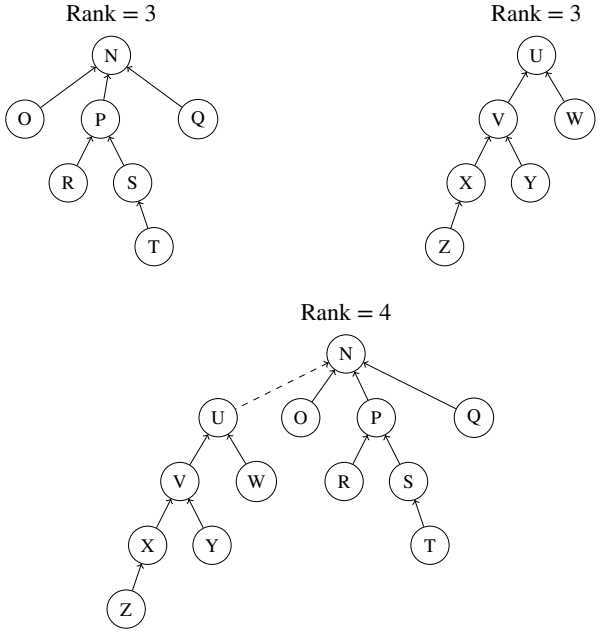
Another optimization technique is **union-by-rank**, which keeps track of the approximate height (or *rank*) of each disjoint set in the data structure instead of its size. When two disjoint sets are unioned together under the union-by-rank optimization approach, we always update the representative of the set with the smaller rank. Some examples of rank are shown below.



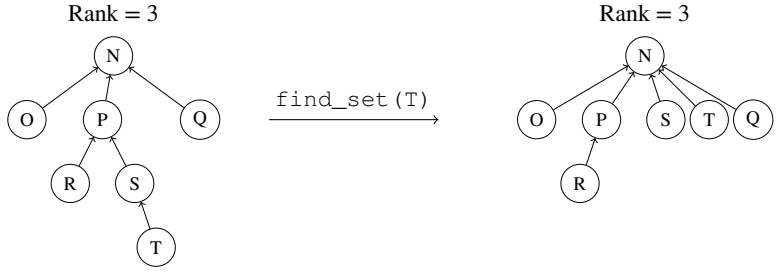
For instance, if we wanted to union elements F and N, we would update F's representative to N because F's disjoint set has a smaller rank. The rank of N, however, would still be 3 after the sets are joined together.



Rank is incremented if two disjoint sets with the same rank are unioned together. When unioning two disjoint sets with the same rank, it does not matter which representative is updated to the other.



Note that we consider rank as an *approximation* for the longest representative chain that exists between an element and its ultimate representative. This is because rank is *not* recomputed if path compression changes the actual maximum length. For example, if we call `find_set(T)` on the following disjoint set using union-by-rank, both S and T's representatives would change to N, but the rank of the entire set would still be 3.



※ 13.4.3 *Union-Find Complexity Analysis* (✻)

So far, we have discussed several union-find optimization strategies, including path compression, union-by-size, and union-by-rank. Path compression can be used to move elements closer to their ultimate representatives, which speeds up future calls to union and find. Union-by-size or union-by-rank can be used to build the connections in a way that prevents the length of any representative chain from growing beyond $\Theta(\log(n))$. However, how do these optimizations impact the overall time complexities of union and find?

The answer is not trivial, so we will not be going into the full details here. If we look at a union-find container that only implements path compression but not union-by-rank or union-by-size, then the time complexity of $n-1$ union operations and m find operations is bounded by $\Theta(n+m\log(n))$ for $m < n$, and $\Theta(m\log(n))$ for $m \geq n$. On a similar vein, if we have a union-find container that only implements union-by-rank or union-by-size, but not path compression, then the time complexity of $n-1$ union operations and m find operations is also bounded by a worst-case time complexity of $\Theta(m\log(n))$. This is because the costs of find and union are both bounded by the largest rank or size among the disjoint sets, which is itself bounded by $\Theta(\log(n))$.

However, if we combine path compression with either union-by-size or union-by-rank, the time complexity of any m union-find operations drops down to $\Theta(m\alpha(n))$, which also means that the amortized costs of find and union both become $\Theta(\alpha(n))$. Here, $\alpha(n)$ represents the **inverse Ackermann function**, which is a function that grows so slowly that its value can practically be treated as a constant for reasonable values of n. In other words, if path compression is combined with union-by-size or union-by-rank, `find_set()` and `union_set()` can both be done in (essentially) amortized constant time![2]

Disjoint sets and union-find have practical use cases in several important computer science problems, such as image processing or counting connected components. We will revisit the concept of union-find in chapter 20 when we discuss Kruskal's algorithm, which can be used to find the minimum cost required to connect all nodes in a graph.

[2]The analyses in this section were obtained from the research paper "Worst-Case Analysis of Set Union Algorithms" by Tarjan and van Leeuwen (1984), and chapter 21 of "Introduction to Algorithms (Third ed.) by Cormen, Leiserson, Rivest, and Stein (2009).