

Aug 2025, India

Union-Find



Or

Disjoint-Set

Tutorial

Tutorial By

Vikas Awadhiya

This work is licensed under [CC-BY 4.0](https://creativecommons.org/licenses/by/4.0/)  

Union-Find/Disjoint-Set Tutorial © 2025 by Vikas Awadhiya is licensed under **Creative Commons Attribution 4.0 International (CC-BY 4.0)**.
To view a copy of this license, visit

<https://creativecommons.org/licenses/by/4.0/>



Invented By

Bernard A. Galler

&

Michael J. Fisher

Tutorial By

Vikas Awadhiya

LinkedIn profile: <https://in.linkedin.com/in/awadhiya-vikas>

1 Introduction

Union-Find or Disjoint-Set is an implicit forest (trees) data structure. It contains disjoint sets as trees where the root of a tree uniquely identify the set and used as a representative. union-find/disjoint-set applies merge optimization and branch compression/flatting technique to the trees to optimize time complexity of union/merge and find operations.

As an implicit forest/group of trees data structure, nodes are not used to implement it rather indexing is used to maintain parent-child relationship.

To find the set an element belongs to or to union/merge two sets requires almost constant time complexity. The insert operations requires constant time complexity in best case and linear time complexity in worst case.

Union-Find/Disjoint-Set has $O(n)$ linear space complexity.

2 The Data Structure

In mathematics, set is a collection of elements with no duplicate elements. To represent set in programming `std::set` (Red-Black Tree) or `std::unordered_set` (Hash-Table) are the obvious containers. To insert or find an element requires $O(\log n)$ time complexity with red-black tree and average $O(1)$ constant time complexity and $O(n)$ linear time complexity in worst case with hash-table.

The union-find data structure requires constant (almost constant) time complexity to merge two sets or to find the set an element belongs to, and to insert an element (that is to insert a new set) requires $O(n)$ linear time complexity in worst case and average $O(1)$ constant time complexity.

Union-Find data structure implements tree in an array (sequence container) rather than using nodes. Each element in union-find contains index/key of it's parent that is `arr[i] = j` where i represents an element and value at `arr[i]` represents index/key of parent element. Unlike the red-black tree or hash-table, union-find data structure doesn't contain elements rather indices are used as keys to represent elements. An instance/object of red-black tree or hash-table represents a set but union-find may contains multiple sets. The maximum number of sets it can contain is n sets because at the time of initialization each element represents a set or in other words each element is in a separate set.

The entity uses union-find data structure requires to assign key to elements, it means if there are n elements then each element must be assigned an unique key such as $0 \leq \text{key} < n$. An undirected graph is a good example where the vertices are labeled as numbers from 0 to $n - 1$ and these labels can be used directly as keys to represent vertices.

The union-find data structure doesn't contain elements itself rather contains their numeric references (keys) may not happen always. The insert operation allows to insert a new element/set dynamically and it requires when number of element is not already known as compared to the scenario like a graph where number of vertices are already known. In one of the possible implementation to allow element insertion dynamically union-find contains elements rather than their numeric references/keys. The [insert](#) operation sub-section explains it in details.

At initialization each element is assigned it's own key/index as parent (`arr[i] = i`) that is an invalid parent value indicates a root element of a set. It means at initialization each element represents a set or in other words each element belongs to a separate set. The fig 1 shown below depict union-find with 10 elements where each element belongs to a separate set.



Fig 1

The sets represented by element 0 and by element 1 are merged. Similarly sets of elements 8 and 9 are merged and sets of elements 4 and 5 are merged and then these two resulted sets merged together as shown below in fig 2. Now union-find data structure contains only six sets.

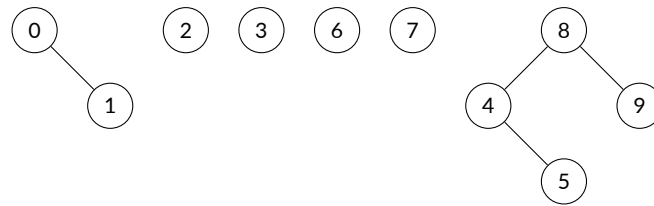


Fig 2

Now it is required to find the set element 5 belongs to. So begins with element 5 and as shown above in fig 2, it is connected with element 4 that means element 4 is the parent element of element 5 ($\text{arr}[5] = 4$), then move to element 4 but element 4 is connected with element 8, so move to element 8. As shown above in fig 2 element 8 is a root element that means it has $\text{arr}[8] = 8$ and further backtracking is not possible. The final answer is, element 5 belongs to the set represented by element 8.

To find the set an element x belongs to requires traverse from element x to the root element of the set and number of nodes requires to traverse is depends on the height of the tree and in worst case it requires $O(n)$ linear time complexity.

To bring time complexity from linear to constant time complexity, union-find data structure applies two optimization techniques. At union/merge of two sets, meta data rank or size is used and with each find operation branch compression/flattening techniques is used. These techniques reduce the heights of the trees and bring the time complexity to almost constant.

2.1 Union Operation

The union/merge of two sets randomly may not always produce an optimal result for example fig 3 as shown below there are two sets, first set is represented by element 1 and other is represented by element 4 and these set can be merged in two possible way.

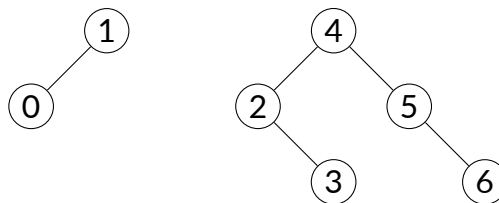


Fig 3

The first possible way is shown below on the left in fig 4 where element 1 represents merged set and assigned as a parent to the element 4. The second possibility is shown below on the right in fig 5 where element 4 represents merged set and it is now parent of element 1.

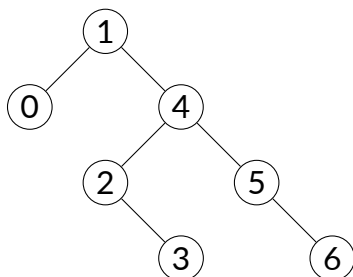


Fig 4

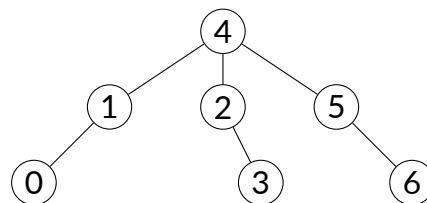


Fig 5

The merged set shown in fig 5 is more efficient because an element belong to this set requires maximum two backtracking steps to find the set it belongs (backtracking from leaf node to root node) but merged set shown in fig 4 requires maximum tree backtracking steps for an element to find which set it belongs to.

There are two techniques, rank based and size based to perform union/merge of sets in efficient manner and any one of them can be used at a time. Union-find allocates an auxiliary array of size equal to the number of elements (initial number of sets) to store rank or size information of each set. Let's see both of these techniques one by one.

2.1.1 Rank

Rank of a set (tree) is based on the height of the tree but the rank never decrease even if the height of tree decreases. At initialization each element (set) is initialized with 0 as a rank value. The rank is shown below in fig 9 as a label on the top right side of each node (element).



Fig 6

If the rank of two set are equal then the root element of any set can be assign as parent of root element of other set and it's rank increased by one as shown below in fig 7 where sets represented by element 0 and element 1 are merged and another merge between the sets of element 4 and element 5.

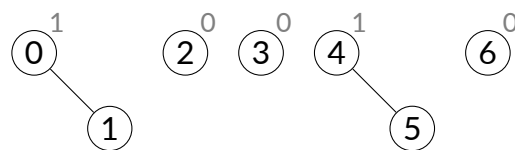


Fig 7

But if the rank of two sets are not equal then the set/tree with lower rank merges in the set with higher rank and height of higher rank set remains same. The fig 8 shown below on the left depict two sets and the set represented by element 1 has rank one but the set represented by element 3 has rank two. At union/merge of these sets element 3 is assigned as a parent element of element 1 (set of element 1 merged in set of element 3) and the rank of set represented by element 3 remain same as shown below on the right in fig 9.

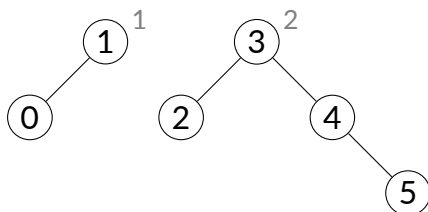


Fig 8

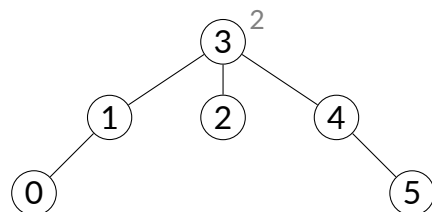


Fig 9

The rank of only root elements are shown in above figures because those ranks are of the interest in union/merge operation.

2.1.2 Size

In size based approach element count of each set is stored in an auxiliary array. At initialization all elements (sets) are assign size 1. The size is shown below in fig 10 as a label on the top right side of each node (element).



Fig 10

If the size of two set are same then any set can be merge in other set and the size doubles but if the size of two sets are different then the set with smaller size merges in the set of bigger size and the size of bigger set increase by size of smaller set.

Two set of different sizes are shown below on the left in fig 11. The set represented by element 1 has size two and the set represented by element 3 has size four. At the time of merge the set of element 1 with smaller size merges in set of bigger size and the size of set represented by element 3 is increased to six by adding the size of smaller set as shown below on the right in fig 12.

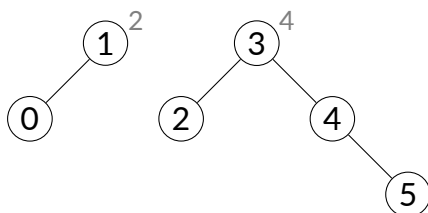


Fig 11

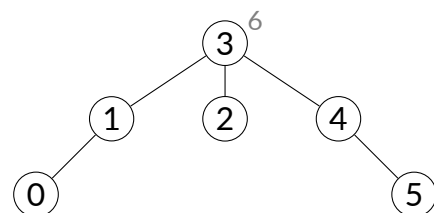


Fig 12

It is very similar to the techniques base on rank the only difference here is that the number of elements in both sets decide the way the union/merge operation is performed.

To union/merge two sets doesn't require root elements of sets rather any element of a set can be passed. The merge operation first find the sets these two given elements belong to by find operation and only perform merge if both of these elements belong to different sets (if representative/root elements of both sets are different).

Let's consider previous example again and rather than passing element 1 and 3 to merge operation let's consider element 2 and element 4 are passed (these are not root elements) but merge operation finds that these elements belong to the same set represented by element 3 as shown above in fig 11. So the merge operation fails because a set can't merge with itself.

2.2 Find Operation

To find the set an element x belongs to, find operation begins from element x and checks it's parent and if the parent is not x itself ($\text{arr}[x] \neq x$) then find continues and move to parent element ($x = \text{arr}[x]$). This continues until reaches to an element which has it's own index as parent value ($\text{arr}[x] == x$) now element(node) is root element and further backtracking is not possible. Find operation then returns root element as a representative of the set.

A ranked based union-find data structure shown below in fig 13 and it has three sets. Now it is required to find the set, element 0 belongs to.

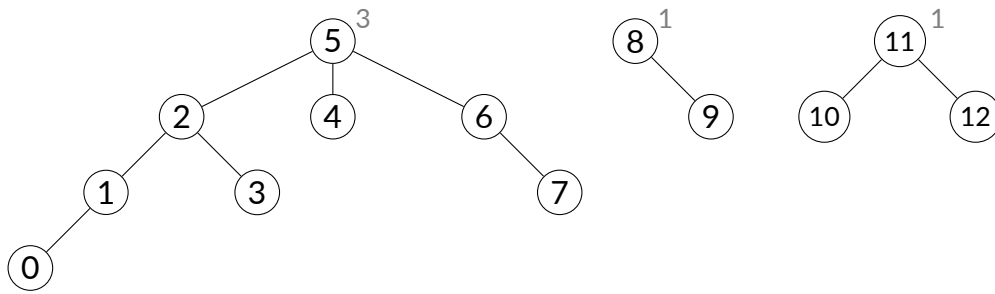


Fig 13

The find begins from element $x = 0$ and check its parent ($\text{arr}[x]$) and element 1 is its parent, so backtrack to parent element that is $x = \text{arr}[0] = 1$. Process continues and checks the parent of element 1 and that is element 2, so the process moves to parent element $x = \text{arr}[1] = 2$ but element 2 has element 3 as parent and process finally moves to element 3. The parent of element 3 is element 3 itself that indicate an invalid value and also means element 3 is a root element and represents the set, element 0 belongs to. Fig 14 show below on the left, highlights the backtracked branch by bold lines. The fig 14 focus only on set represented by element 5 and doesn't show the sets represented by elements 8 and 11 as compared to fig 13.

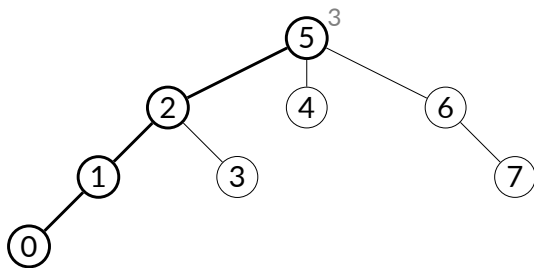


Fig 14

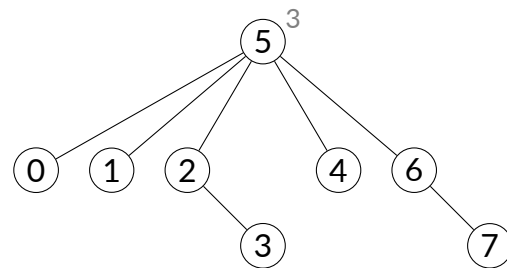


Fig 15

Now element 5 is assigned as parent to element 0 and this directly connect the element 0 to the representative element of the set. This reduces the time complexity to $O(1)$ constant time for future attempt to find the set, element 0 belongs to. This optimization is not limited to element 0 only rather applied to all element in the branched traversed to find the set, as highlight by bold line in fig 14. This is called branch compression/flattening and find operation applies it each time when called if the branch is not already compressed. It reduces time complexity to $O(1)$ constant to find the set of any element which was the part of the branch, compared by find operation. The fig 15 depict the set represented by element 5 after branch compression.

The rank of the set/tree (element 5) still has three as rank value even there is no branch in the tree has height three as shown above in fig 15. As mentioned earlier the rank of set (element) never decrease even when the height of the set/tree decreases after branch compression, to avoid unnecessary overhead but,

The update of rank of a set/tree is not possible at the compression of a branch because it may not be the only branch that keep the rank of the tree/set at this value and there is no way to verify because a child element x knows its parent ($\text{arr}[x]$) but a parent element has no information about its children. That's why it is not possible to find if there is any branch present which keep the rank value same or there is no such branch.

2.3 Insert Operation

In few problems the total number of elements are not known in advance. To handle this scenario elements/sets are required to insert dynamically.

If the element count (initial number of sets) is known that means no insert requires dynamically then array based implementation is enough but if insert functionality is required then there are two ways to implement it.

One possible way is to let the client code to manage new element/set which is using union-find data structure. In this design the entity using union-find data structure checks if the new element is already exist to avoid duplicate element and also assign keys (numeric reference) to elements in increment manner. Assigning the key in increment manner means if there are n elements and from 0 to $n - 1$ values are already assigned as keys then new element must be assign next numeric value ($n = (n - 1) + 1$) as a key. This design has advantage with problems where the total number of elements are not known but there is no possibility of duplicate elements or even if duplicate elements are possible the problem has it's own mechanism to avoid them.

This design makes the union-find simple and reduce implementation complexity and union-find data structure only requires memory reallocation related addition steps to implement insert operation.

Another way is, make the union-find data structure self-sufficient. This approach uses `std::map` (hash table) rather than arrays for both parent and rank/size meta data information. Here elements are used to evaluate hash values or in other words element is itself used as a key (`map[element]`). In this approach union-find contains elements rather than keys (numeric reference) and capable to find if the new element/set is already present or not.

In this design, a hash function has to be provided to use union-find data structure with custom data types. It also introduces the problem of multiple copies of the elements but it can be avoided by implementation technique.

After memory reallocation in array based approach or after finding that element is not already present in hash table based approach the new element is assigned it own value as a parent and rank/size meta data is populated accordingly.

3 Implementation

The C++ implementation of the Union-Find / Disjoint-Set data structure is provided under the MIT License. It also contains the `main.cpp` file to demonstrate its usage.

Implementation doesn't provide `insert` functionality to keep the code simple.

Code available at: <https://github.com/vikasawadhiya/Union-Find-Or-Disjoint-Set>

August 2025, India