Union By Rank and Path Compression in Union-Find Algorithm - GeeksforGeeks

Source: https://www.geeksforgeeks.org/union-by-rank-and-path-compression-in-union-find-algorithm/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Union By Rank and Path Compression in Union-Find Algorithm Last Updated : 17 Jan, 2026 In the previous post , we introduced the Union-Find algorithm. We employed the union() and find() operations to manage subsets. However, various optimization techniques can be applied, with the primary goal of minimizing the height of the trees representing the disjoint sets. The most common methods are, Path Compression, Union By Rank and Union By Size Path Compression (Used to improve find()): The idea is to flatten the tree when find() is called. When find() is called for an element x, root of the tree is returned. The find() operation traverses up from x to find root. The idea of path compression is to make the found root as parent of x so that we don’t have to traverse all intermediate nodes again. If x is root of a subtree, then path (to root) from all nodes under x also compresses. It speeds up the data structure by compressing the height of the trees. It can be achieved by inserting a small caching mechanism into the Find operation. Take a look at the code for more details: Union by Rank (Modifications to union()) Rank is like height of the trees representing different sets. We use an extra array of integers called rank[] . The size of this array is the same as the parent array Parent[] . If i is a representative of a set, rank[i] is the rank of the element i. Rank is same as height if path compression is not used. With path compression, rank can be more than the actual height. Now recall that in the Union operation, it doesn’t matter which of the two trees is moved under the other. Now what we want to do is minimize the height of the resulting tree. If we are uniting two trees (or sets), let’s call them left and right, then it all depends on the rank of left and the rank of right . If the rank of left is less than the rank of right , then it’s best to move left under right , because that won’t change the rank of right (while moving right under left would increase the height). In the same way, if the rank of right is less than the rank of left, then we should move right under left. If the ranks are equal, it doesn’t matter which tree goes under the other, but the rank of the result will always be one greater than the rank of the trees. C++ #include <iostream> #include <vector> using namespace std ; class DisjointUnionSets { vector < int > rank , parent ; public : // Constructor to initialize sets DisjointUnionSets (int n) { rank . resize (n , 0); parent . resize (n); // Initially, each element is in its own set for (int i = 0 ; i < n ; i ++) { parent [i] = i ; } } // Find the representative of the set that x belongs to int find (int i) { int root = parent [i]; // Path Compression if (parent [root] != root) { return parent [i] = find (root); } return root ; } // Union of sets containing x and y void unionSets (int x , int y) { int xRoot = find (x); int yRoot = find (y); // If they are in the same set, no need to union if (xRoot == yRoot) return ; // Union by rank if (rank [xRoot] < rank [yRoot]) { parent [xRoot] = yRoot ; } else if (rank [yRoot] < rank [xRoot]) { parent [yRoot] = xRoot ; } else { parent [yRoot] = xRoot ; rank [xRoot] ++ ; } } }; int main () { // Let there be 5 persons with ids 0, 1, 2, 3, and 4 int n = 5 ; DisjointUnionSets dus (n); // 0 is a friend of 2 dus . unionSets (0 , 2); // 4 is a friend of 2 dus . unionSets (4 , 2); // 3 is a friend of 1 dus . unionSets (3 , 1); // Check if 4 is a friend of 0 if (dus . find (4) == dus . find (0)) cout << "Yes \n " ; else cout << "No \n " ; // Check if 1 is a friend of 0 if (dus . find (1) == dus . find (0)) cout << "Yes \n " ; else cout << "No \n " ; return 0 ; } Java // A Java program to implement Disjoint Set with // Path Compression and Union by Rank import java.io.* ; import java.util.* ; class DisjointUnionSets { int [] rank , parent ; int n ; // Constructor public DisjointUnionSets (int n) { rank = new int [n] ; parent = new int [n] ; this . n = n ; for (int i = 0 ; i < n ; i ++) { // Initially, all elements are in // their own set. parent [i] = i ; } } // Returns representative of x's set public int find (int i

) { int root = parent [i] ; // Path Compression if (parent [root] != root) { return parent [i] = find (root); } return root ; } // Unites the set that includes x and the set // that includes x void union (int x , int y) { // Find representatives of two sets int xRoot = find (x), yRoot = find (y); // Elements are in the same set, no need // to unite anything. if (xRoot == yRoot) return ; // If x's rank is less than y's rank if (rank [xRoot] < rank [yRoot]) // Then move x under y so that depth // of tree remains less parent [xRoot] = yRoot ; // Else if y's rank is less than x's rank else if (rank [yRoot] < rank [xRoot]) // Then move y under x so that depth of // tree remains less parent [yRoot] = xRoot ; else // if ranks are the same { // Then move y under x (doesn't matter // which one goes where) parent [yRoot] = xRoot ; // And increment the result tree's // rank by 1 rank [xRoot] = rank [xRoot] + 1 ; } } } // Driver code public class Main { public static void main (String [] args) { // Let there be 5 persons with ids as // 0, 1, 2, 3 and 4 int n = 5 ; DisjointUnionSets dus = new DisjointUnionSets (n); // 0 is a friend of 2 dus . union (0 , 2); // 4 is a friend of 2 dus . union (4 , 2); // 3 is a friend of 1 dus . union (3 , 1); // Check if 4 is a friend of 0 if (dus . find (4) == dus . find (0)) System . out . println ("Yes"); else System . out . println ("No"); // Check if 1 is a friend of 0 if (dus . find (1) == dus . find (0)) System . out . println ("Yes"); else System . out . println ("No"); } } Python class DisjointUnionSets : def __init__ (self , n): self . rank = [0] * n self . parent = list (range (n)) def find (self , i): root = self . parent [i] # Path Compression if self . parent [root] != root : self . parent [i] = self . find (root) return self . parent [i] return root def unionSets (self , x , y): xRoot = self . find (x) yRoot = self . find (y) if xRoot == yRoot : return # Union by Rank if self . rank [xRoot] < self . rank [yRoot]: self . parent [xRoot] = yRoot elif self . rank [yRoot] < self . rank [xRoot]: self . parent [yRoot] = xRoot else : self . parent [yRoot] = xRoot self . rank [xRoot] += 1 if __name__ == '__main__' : n = 5 # Let there be 5 persons with ids 0, 1, 2, 3, and 4 dus = DisjointUnionSets (n) dus . unionSets (0 , 2) # 0 is a friend of 2 dus . unionSets (4 , 2) # 4 is a friend of 2 dus . unionSets (3 , 1) # 3 is a friend of 1 # Check if 4 is a friend of 0 if dus . find (4) == dus . find (0): print ('Yes') else : print ('No') # Check if 1 is a friend of 0 if dus . find (1) == dus . find (0): print ('Yes') else : print ('No') C# // A C# program to implement Disjoint Set with // Path Compression and Union by Rank using System ; class DisjointUnionSets { int [] rank , parent ; int n ; // Constructor public DisjointUnionSets (int n) { rank = new int [n]; parent = new int [n]; this . n = n ; for (int i = 0 ; i < n ; i ++) { // Initially, all elements are in // their own set. parent [i] = i ; } } // Returns representative of x's set public int find (int i) { int root = parent [i]; // Path Compression if (parent [root] != root) { return parent [i] = find (root); } return root ; } // Unites the set that includes x and the set // that includes y public void union (int x , int y) { // Find representatives of two sets int xRoot = find (x), yRoot = find (y); // Elements are in the same set, no need // to unite anything. if (xRoot == yRoot) return ; // If x's rank is less than y's rank if (rank [xRoot] < rank [yRoot]) // Then move x under y so that depth // of tree remains less parent [xRoot] = yRoot ; // Else if y's rank is less than x's rank else if (rank [yRoot] < rank [xRoot]) // Then move y under x so that depth of // tree remains less parent [yRoot] = xRoot ; else // if ranks are the same { // Then move y under x (doesn't matter // which one goes where) parent [yRoot] = xRoot ; // And increment the result tree's // rank by 1 rank [xRoot] = rank [xRoot] + 1 ; } } } // Driver code class Program { static void Main (string [] args) { // Let there be 5 persons with ids as // 0, 1, 2, 3 and 4 int n = 5 ; DisjointUnionSets dus = new DisjointUnionSets (n); // 0 is a friend of 2 dus . union (0 , 2); // 4 is a friend of 2 dus . union (4 , 2); // 3 is a friend of 1 dus . union (3 , 1); // Check if 4 is a friend of 0 if (dus . find (4) == dus . find (0)) Console . WriteLine ("Yes"); else Console . WriteLine ("No"); // Check if 1 is a friend of 0 if (dus . find (1) == dus . find (0)) Console . WriteLine ("Yes"); else Console . WriteLine ("No"); } } JavaScript class DisjointUnionSets { constructor (n) { this . rank = new Array (n). fill (0); this . parent = Array . from ({ length : n }, (_ , i) => i); // Initially, each element is in its own set } find (i) { let root = this . parent [i]; // Path Compression if (this . parent [root] !== root) { return this . parent [i] = this . find (root); } return root ; } unionSets (x , y) { const xRoot = this . find (x); const yRoot = this . find (y); // If they are in the same set, no need to union if (xRoot === yRoot) return ; // Union by rank if (this . rank [xRoot] < this . rank [yRoot]) { this . parent [xRoot] = yRoot ; } else if (this . rank [yRoot] < this . rank [xRoot]) { this . parent [yRoot] = xRoot ; } else { this . parent [yRoot] = xRoot ; this . rank [xRoot] ++ ; } } } const n = 5 ; // Let there be 5 persons with ids 0, 1, 2, 3, and 4 const dus = new DisjointUnionSets (n); // 0 is a friend of 2 dus . unionSets (0 , 2); // 4 is a friend of 2 dus . unionSets (4 , 2); // 3 is a friend of 1 dus . unionSets (3 , 1); // Check if 4 is a friend of 0 if (dus . find (4) === dus . find (0)) console . log ('Yes'); else console . log ('No'); // Check if 1 is a friend of 0 if (dus . find (1) === dus . find (0)) console . log ('Yes'); else console . log ('No'); Output Yes No Time complexity : O(n) for creating n single item sets . The two techniques -path compression with the union by rank/size, the time complexity will reach nearly constant time. It turns out, that the final amortized time complexity is O(α(n)), where α(n) is the inverse Ackermann function,

which grows very steadily (it does not even exceed for n<10 600 approximately). Space complexity: O(n) because we need to store n elements in the Disjoint Set Data Structure. Union by Size (Alternate of Union by Rank) We use an array of integers called size[] . The size of this array is the same as the parent array Parent[] . If i is a representative of a set, size[i] is the number of the elements in the tree representing the set. Now we are uniting two trees (or sets), let's call them left and right, then in this case it all depends on the size of left and the size of right tree (or set). If the size of left is less than the size of right , then it's best to move left under right and increase size of right by size of left. In the same way, if the size of right is less than the size of left, then we should move right under left. and increase size of left by size of right. If the sizes are equal, it doesn't matter which tree goes under the other. C++ // C++ program for Union by Size with Path Compression #include <iostream> #include <vector> using namespace std ; class UnionFind { vector < int > Parent ; vector < int > Size ; public : UnionFind (int n) { Parent . resize (n); for (int i = 0 ; i < n ; i ++) { Parent [i] = i ; } // Initialize Size array with 1s Size . resize (n , 1); } // Function to find the representative (or the root // node) for the set that includes i int find (int i) { int root = Parent [i]; if (Parent [root] != root) { return Parent [i] = find (root); } return root ; } // Unites the set that includes i and the set that // includes j by size void unionBySize (int i , int j) { // Find the representatives (or the root nodes) for // the set that includes i int irep = find (i); // And do the same for the set that includes j int jrep = find (j); // Elements are in the same set, no need to unite // anything. if (irep == jrep) return ; // Get the size of i's tree int isize = Size [irep]; // Get the size of j's tree int jsize = Size [jrep]; // If i's size is less than j's size if (isize < jsize) { // Then move i under j Parent [irep] = jrep ; // Increment j's size by i's size Size [jrep] += Size [irep]; } // Else if j's size is less than i's size else { // Then move j under i Parent [jrep] = irep ; // Increment i's size by j's size Size [irep] += Size [jrep]; } } }; int main () { int n = 5 ; UnionFind unionFind (n); unionFind . unionBySize (0 , 1); unionFind . unionBySize (2 , 3); unionFind . unionBySize (0 , 4); for (int i = 0 ; i < n ; i ++) { cout << "Element " << i << ": Representative = " << unionFind . find (i) << endl ; } return 0 ; } Java // Java program for Union by Size with Path // Compression import java.util.Arrays ; class UnionFind { private int [] Parent ; private int [] Size ; public UnionFind (int n) { // Initialize Parent array Parent = new int [n] ; for (int i = 0 ; i < n ; i ++) { Parent [i] = i ; } // Initialize Size array with 1s Size = new int [n] ; Arrays . fill (Size , 1); } // Function to find the representative (or the root // node) for the set that includes i public int find (int i) { int root = Parent [i] ; if (Parent [root] != root) { return Parent [i] = find (root); } return root ; } // Unites the set that includes i and the set that // includes j by size public void unionBySize (int i , int j) { // Find the representatives (or the root nodes) for // the set that includes i int irep = find (i); // And do the same for the set that includes j int jrep = find (j); // Elements are in the same set, no need to unite // anything. if (irep == jrep) return ; // Get the size of i's tree int isize = Size [irep] ; // Get the size of j's tree int jsize = Size [jrep] ; // If i's size is less than j's size if (isize < jsize) { // Then move i under j Parent [irep] = jrep ; // Increment j's size by i's size Size [jrep] += Size [irep] ; } // Else if j's size is less than i's size else { // Then move j under i Parent [jrep] = irep ; // Increment i's size by j's size Size [irep] += Size [jrep] ; } } } public class GFG { public static void main (String [] args) { // Example usage int n = 5 ; UnionFind unionFind = new UnionFind (n); // Perform union operations unionFind . unionBySize (0 , 1); unionFind . unionBySize (2 , 3); unionFind . unionBySize (0 , 4); // Print the representative of each element after // unions for (int i = 0 ; i < n ; i ++) { System . out . println ("Element " + i + ": Representative = " + unionFind . find (i)); } } } Python class UnionFind : def __init__ (self , n): self . Parent = list (range (n)) self . Size = [1] * n # Function to find the representative (or the root # node) for the set that includes i def find (self , i): root = self . Parent [i] if self . Parent [root] != root : self . Parent [i] = self . find (root) return self . Parent [i] return root # Unites the set that includes i and the set that # includes j by size def unionBySize (self , i , j): # Find the representatives (or the root nodes) for # the set that includes i irep = self . find (i) # And do the same for the set that includes j jrep = self . find (j) # Elements are in the same set, no need to unite # anything. if irep == jrep : return # Get the size of i's tree isize = self . Size [irep] # Get the size of j's tree jsize = self . Size [jrep] # If i's size is less than j's size if isize < jsize : # Then move i under j self . Parent [irep] = jrep # Increment j's size by i's size self . Size [jrep] += self . Size [irep] # Else if j's size is less than i's size else : # Then move j under i self . Parent [jrep] = irep # Increment i's size by j's size self . Size [irep] += self . Size [jrep] n = 5 unionFind = UnionFind (n) unionFind . unionBySize (0 , 1) unionFind . unionBySize (2 , 3) unionFind . unionBySize (0 , 4) for i in range (n): print (f 'Element { i } : Representative = { unionFind . find (i) } ') C# // C# program for Union by Size with Path Compression using System ; using System.Linq ; class UnionFind { private int [] Parent ; private int [] Size ; public UnionFind (int n) { // Initialize Parent array Parent = new int [n]; for (int i = 0 ; i < n ; i ++) { Parent [i] = i ; } // Initialize Size array with 1s Size = new int [n]; Array . Fill (Size , 1); } // Function to find the

representative (or // the root node) for the set that includes i public int Find (int i) { int root = Parent [i]; if (Parent [root] != root) { return Parent [i] = Find (root); } return root ; } // Unites the set that includes i and the set // that includes j by size public void UnionBySize (int i , int j) { // Find the representatives (or the root nodes) // for the set that includes i int irep = Find (i); // And do the same for the set that includes j int jrep = Find (j); // Elements are in the same set, no need to unite anything. if (irep == jrep) return ; // Get the size of i's tree int isize = Size [irep]; // Get the size of j's tree int jsize = Size [jrep]; // If i's size is less than j's size if (isize < jsize) { // Then move i under j Parent [irep] = jrep ; // Increment j's size by i's size Size [jrep] += Size [irep]; } // Else if j's size is less than i's size else { // Then move j under i Parent [jrep] = irep ; // Increment i's size by j's size Size [irep] += Size [jrep]; } } } class GFG { public static void Main (string [] args) { int n = 5 ; UnionFind unionFind = new UnionFind (n); unionFind . UnionBySize (0 , 1); unionFind . UnionBySize (2 , 3); unionFind . UnionBySize (0 , 4); // Print the representative of each element after unions for (int i = 0 ; i < n ; i ++) { Console . WriteLine ("Element " + i + ": Representative = " + unionFind . Find (i)); } } } JavaScript class UnionFind { constructor (n) { this . Parent = Array . from ({ length : n }, (_ , i) => i); this . Size = Array (n). fill (1); } // Function to find the representative (or the root // node) for the set that includes i find (i) { let root = this . Parent [i]; if (this . Parent [root] !== root) { return this . Parent [i] = this . find (root); } return root ; } // Unites the set that includes i and the set that // includes j by size unionBySize (i , j) { // Find the representatives (or the root nodes) for // the set that includes i const irep = this . find (i); // And do the same for the set that includes j const jrep = this . find (j); // Elements are in the same set, no need to unite // anything. if (irep === jrep) return ; // Get the size of i's tree const isize = this . Size [irep]; // Get the size of j's tree const jsize = this . Size [jrep]; // If i's size is less than j's size if (isize < jsize) { // Then move i under j this . Parent [irep] = jrep ; // Increment j's size by i's size this . Size [jrep] += this . Size [irep]; } else { // Then move j under i this . Parent [jrep] = irep ; // Increment i's size by j's size this . Size [irep] += this . Size [jrep]; } } } const n = 5 ; const unionFind = new UnionFind (n); unionFind . unionBySize (0 , 1); unionFind . unionBySize (2 , 3); unionFind . unionBySize (0 , 4); for (let i = 0 ; i < n ; i ++) { console . log (`Element ${ i } : Representative = ${ unionFind . find ( i ) } `); } Output Element 0: Representative = 0 Element 1: Representative = 0 Element 2: Representative = 2 Element 3: Representative = 2 Element 4: Representative = 0 Comment Article Tags: Article Tags: Graph DSA union-find