Register for SDE Sheet Challenge

takeUforward

Signin/Signup

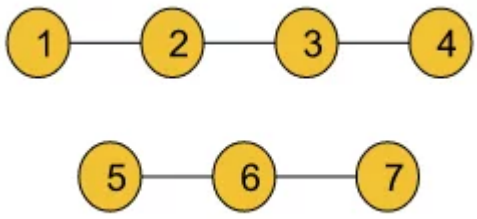
Striver's SDE Sheet | Striver's A2Z DSA Course/Sheet | Striver's DSA Playlists | CS Subjects | Interview Prep Sheets | Striver's CP Sheet

December 15, 2022 ▪ Data Structure / Graph

Disjoint Set | Union by Rank | Union by Size | Path Compression: G-46

In this article, we will discuss the ***Disjoint Set*** data structure which is a very important topic in the entire graph series. Let's first understand ***why we need a Disjoint Set data structure using the below question:***

Question: Given two components of an undirected graph



The question is whether node 1 and node 5 are in the same component or not.

Approach:

Search

Search

Latest Video on takeUforward

Strive

Latest Video on Striver

Now, in order to solve this question we can use either the DFS or BFS traversal technique like if we traverse the components of the graph we can find that node 1 and node 5 are not in the same component. This is actually the ***brute force*** approach whose time complexity is **O(N+E)***(N = no. of nodes, E = no. of edges).* But ***using a Disjoint Set data structure we can solve this same problem in constant time***.

The disjoint Set data structure is generally used for ***dynamic graphs***.

Dynamic graph:

A dynamic graph generally refers to a graph that keeps on changing its configuration. Let's deep dive into it using an example:

- Let's consider the edge information for the given graph as: {{1,2}, {2,3}, {4,5}, {6,7}, {5,6}, {3,7}}. Now if we start adding the edges one by one, in each step the structure of the graph will change. So, after each step, if we perform the same operation on the graph while updating the edges, the result might be different. In this case, the graph will be considered a dynamic graph.
- For example, after adding the first 4 edges if we look at the graph, we will find that node 4 and node 1 belong to different components but after adding all 6 edges if

Open

Recent Posts

Implement Upper Bound

Implement Lower Bound

2023 – Striver's SDE Sheet Challenge

Top Array Interview Questions – Structured Path with Video Solutions

Longest Subarray with sum K | [Postives and Negatives]

we search for the same we will figure out that node 4 and node 1 belong to the same component.

- So, ***after any step, if we try to figure out whether two arbitrary nodes u and v belong to the same component or not, Disjoint Set will be able to answer this query in constant time.***

Functionalities of Disjoint Set data structure:

The disjoint set data structure generally provides two types of functionalities:

- Finding the parent for a particular node (***findPar()***)
- Union (in broad terms this method basically adds an edge between two nodes)

- Union by rank
- Union by size

First, we will be discussing Union by rank and then Union by size.

Union by rank:

Before discussing Union by rank we need to discuss some terminologies:

Rank:

The rank of a node generally refers to the distance (the number of nodes including the leaf node) between the furthest leaf node and the current node. Basically rank includes all the nodes beneath the current node.

Ultimate parent:

The parent of a node generally refers to the node right above that particular node. But

the ultimate parent refers to the topmost node or the root node.

Now let's discuss the implementation of the union by rank function. In order to implement Union by rank, we basically need two arrays of size N(no. of nodes). One is the ***rank*** and the other one is the ***parent***. The rank array basically stores the rank of each node and the parent array stores the ultimate parent for each node.

Algorithm:

Initial configuration:

rank array: This array is initialized with zero.

parent array: The array is initialized with the value of nodes i.e. parent[i] = i.

The algorithm steps are as follows:

1. Firstly, the Union function requires two nodes(***let's say u and v***) as arguments. Then we will find the ultimate parent (using the findPar() function that is discussed later) of u and v. Let's consider the ultimate parent of u is ***pu*** and the ultimate parent of v is ***pv***.
2. After that, we will find the rank of ***pu*** and ***pv***.
3. Finally, we will connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. But if the ranks are equal, we can connect any parent to the other parent and we will increase the rank by one for the parent node to whom we have connected the other one.

Let's understand it further using the below example.

Given the edges of a graph are: {{1,2}, {2,3}, {4,5}, {6,7}, {5,6}}

After applying the union by rank function to every edge the graph and the arrays will look like the following:

Observation 1:

If we carefully observe, we are only concerned about the ultimate parent but not the immediate parent.

Let's see ***why we need to find the ultimate parents***.

- After union by rank operations, if we are asked (refer to the above picture) if node 5 and node 7 belong to the same component or not, the answer must be yes. If we carefully look at their immediate parents, they are not the same but if we consider their ultimate parents they are the same i.e. node 4. So, we can determine the answer by considering the

ultimate parent. That is why we need to find the ultimate parent.

So, here comes the **findPar() function** which will help us to find the ultimate parent for a particular node.

findPar() function:

This function actually takes a single node as an argument and finds the ultimate parent for each node.

Observation 2:

Now, if we try to find the ultimate parent(typically using recursion) of each query separately, it will end up taking O(logN) time complexity for each case. But we want the operation to be done in a constant time. This is where the ***path compression technique*** comes in.

Using the ***path compression technique*** we can reduce the time complexity nearly to constant time. It is discussed later on why the time complexity actually reduces.

What is path compression?

Basically, connecting each node in a particular path to its ultimate parent refers to path compression. Let's understand it using the following illustration:

How the time complexity reduces:

- Before path compression, if we had tried to find the ultimate parent for node 4, we had to traverse all the way back to node 1 which is basically the height of size logN. But after path compression, we can easily access the ultimate parent with a single step. Thus the traversal reduces and as a result the time complexity also reduces.

Though using the path compression technique it seems like the rank of the node is also changing, we cannot be sure about it. So, we will not make any changes to the rank array while applying path compression. The following example depicts an example:

Note: *We cannot change the ranks while applying path compression.*

Overall, findPar() method helps to reduce the time complexity of the ***union by the rank*** method as it can find the ultimate parent within constant time.

Algorithm:

This process is done using the backtracking method.

The algorithm steps are as follows:

1. **Base case:** If the node and the parent of the node become the same, it will return the node.
2. We will call the findPar() function for a node until it hits the base case and while backtracking we will update the parent of the current node with the returned value.

Note: *The actual time complexity of union by rank and findPar() is O(4) which is very small and close to 1. So, we can consider 4 as a constant. Now, this 4 term has a long mathematical derivation which is not required for an interview.*

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code of Union by rank and findPar():

Code:

C++ Code

```
#include <bits/stdc++.h>
using namespace std;
class DisjointSet {
    vector<int> rank, parent;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        for (int i = 0; i <= n; i++)
            parent[i] = i;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUP
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v
            parent[ulp_u] = ulp_v;
```

```
            }
            else if (rank[ulp_v] < rank[
                parent[ulp_v] = ulp_u;
            }
            else {
                parent[ulp_v] = ulp_u;
                rank[ulp_u]++;
            }
        }
    };
    int main() {
        DisjointSet ds(7);
        ds.unionByRank(1, 2);
        ds.unionByRank(2, 3);
        ds.unionByRank(4, 5);
        ds.unionByRank(6, 7);
        ds.unionByRank(5, 6);
        // if 3 and 7 same or not
        if (ds.findUPar(3) == ds.findUPa
            cout << "Same\n";
        }
        else cout << "Not same\n";

        ds.unionByRank(3, 7);

        if (ds.findUPar(3) == ds.findUPa
            cout << "Same\n";
        }
        else cout << "Not same\n";
        return 0;
    }
```

Output:

Not same

Same

Time Complexity: The actual time complexity is O(4) which is very small and close to 1. So, we can consider 4 as a constant.

Java Code

```
import java.io.*;
import java.util.*;
class DisjointSet {
    List<Integer> rank = new ArrayLi
    List<Integer> parent = new Array
    public DisjointSet(int n) {
        for (int i = 0; i <= n; i++)
            rank.add(0);
            parent.add(i);
        }
    }

    public int findUPar(int node) {
        if (node == parent.get(node)
            return node;
        }
        int ulp = findUPar(parent.ge
        parent.set(node, ulp);
        return parent.get(node);
    }

    public void unionByRank(int u, i
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank.get(ulp_u) < rank.g
            parent.set(ulp_u, ulp_v)
        } else if (rank.get(ulp_v) <
            parent.set(ulp_v, ulp_u)
        } else {
            parent.set(ulp_v, ulp_u)
            int rankU = rank.get(ulp
            rank.set(ulp_u, rankU +
        }
    }

}

class Main {
    public static void main (String[
        DisjointSet ds = new Disjoin
        ds.unionByRank(1, 2);
        ds.unionByRank(2, 3);
```

```
        ds.unionByRank(4, 5);
        ds.unionByRank(6, 7);
        ds.unionByRank(5, 6);

        // if 3 and 7 same or not
        if (ds.findUPar(3) == ds.fin
            System.out.println("Same
        } else
            System.out.println("Not

        ds.unionByRank(3, 7);
        if (ds.findUPar(3) == ds.fin
            System.out.println("Same
        } else
            System.out.println("Not
    }
}
```

Output:

Not same

Same

Time Complexity: The actual time complexity is O(4) which is very small and close to 1. So, we can consider 4 as a constant.

Follow-up question:

In the union by rank method, why do we need to connect the smaller rank to the larger rank?

- Let's understand it using the following example:

In this case, the traversal time to find the ultimate parent for nodes 3, 4, 5, 6, 7, and 8 increases and so the path compression time also increases. But if we do the following

- the traversal time to find the ultimate parent for only nodes 1 and 2 increases. So the path compression time becomes relatively lesser than in the previous case. So, we can conclude that we should

always connect a smaller rank to a larger one with the goal of

- ***shrinking the height of the graph.***
- ***reducing the time complexity as much as we can.***

Observation 3:

Until now, we have learned union by rank, the findPar() function, and the path compression technique. Now, if we again carefully observe, after applying path compression the rank of the graphs becomes distorted. So, rather than storing the rank, we can just store the size of the components for comparing which component is greater or smaller.

So, here comes the concept of ***Union by size***.

Union by size:

This is as same as the Union by rank method except this method uses the size to compare the components while connecting. That is why we need a ***'size'*** array of size N(no. of nodes) instead of a ***rank*** array. The size array will be storing the size for each particular node i.e. size[i] will be the size of the component starting from node i.

Typically, the size of a node refers to the number of nodes that are connected to it.

Algorithm:

Initial configuration:

size array: This array is initialized with one.

parent array: The array is initialized with the value of nodes i.e. parent[i] = i.

The algorithm steps are as follows:

1. Firstly, the Union function requires two nodes(***let's say u and v***) as arguments. Then we will find the ultimate parent (using the findPar() function discussed earlier) of u and v. Let's consider the ultimate parent of u is ***pu*** and the ultimate parent of v is ***pv***.
2. After that, we will find the size of ***pu*** and ***pv*** i.e. size[pu] and size[pv].
3. Finally, we will connect the ultimate parent with a smaller size to the other ultimate parent with a larger size. But if the size of the two is equal, we can connect any parent to the other parent. While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

Let's understand it further using the below example.

Given the edges of a graph are {{1,2}, {2,3}, {4,5}, {6,7}, {5,6}, {3,7}}

After applying the union by size function to every edge the graph and the arrays will look like the following:

Note: *It seems much more intuitive than union by rank as the rank gets distorted after path compression.*

Note: *The findPar() function remains the exact same as we have discussed earlier.*

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Disjoint Set data structure implementation:

Code:

C++ Code

```
#include <bits/stdc++.h>
using namespace std;
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++)
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUP
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[
            parent[ulp_v] = ulp_u;
        }
```

```
            else {
                parent[ulp_v] = ulp_u;
                rank[ulp_u]++;
            }
        }

        void unionBySize(int u, int v) {
            int ulp_u = findUPar(u);
            int ulp_v = findUPar(v);
            if (ulp_u == ulp_v) return;
            if (size[ulp_u] < size[ulp_v
                parent[ulp_u] = ulp_v;
                size[ulp_v] += size[ulp_
            }
            else {
                parent[ulp_v] = ulp_u;
                size[ulp_u] += size[ulp_
            }
        }
    };
    int main() {
        DisjointSet ds(7);
        ds.unionBySize(1, 2);
        ds.unionBySize(2, 3);
        ds.unionBySize(4, 5);
        ds.unionBySize(6, 7);
        ds.unionBySize(5, 6);
        // if 3 and 7 same or not
        if (ds.findUPar(3) == ds.findUPa
            cout << "Same\n";
        }
        else cout << "Not same\n";

        ds.unionBySize(3, 7);

        if (ds.findUPar(3) == ds.findUPa
            cout << "Same\n";
        }
        else cout << "Not same\n";
        return 0;
    }
```

Output:

Not Same

Same

Time Complexity: The time complexity is O(4) which is very small and close to 1. So, we can consider 4 as a constant.

Java Code

```
import java.io.*;
import java.util.*;
class DisjointSet {
    List<Integer> rank = new ArrayLi
    List<Integer> parent = new Array
    List<Integer> size = new ArrayLi
    public DisjointSet(int n) {
        for (int i = 0; i <= n; i++)
            rank.add(0);
            parent.add(i);
            size.add(1);
        }
    }

    public int findUPar(int node) {
        if (node == parent.get(node)
            return node;
        }
        int ulp = findUPar(parent.ge
        parent.set(node, ulp);
        return parent.get(node);
    }

    public void unionByRank(int u, i
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank.get(ulp_u) < rank.g
            parent.set(ulp_u, ulp_v)
        } else if (rank.get(ulp_v) <
            parent.set(ulp_v, ulp_u)
        } else {
            parent.set(ulp_v, ulp_u)
            int rankU = rank.get(ulp
            rank.set(ulp_u, rankU +
        }
    }
```

```
    public void unionBySize(int u, i
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size.get(ulp_u) < size.g
            parent.set(ulp_u, ulp_v)
            size.set(ulp_v, size.get
        } else {
            parent.set(ulp_v, ulp_u)
            size.set(ulp_u, size.get
        }
    }
}

class Main {
    public static void main (String[
        DisjointSet ds = new Disjoin
        ds.unionByRank(1, 2);
        ds.unionByRank(2, 3);
        ds.unionByRank(4, 5);
        ds.unionByRank(6, 7);
        ds.unionByRank(5, 6);

        // if 3 and 7 same or not
        if (ds.findUPar(3) == ds.fin
            System.out.println("Same
        } else
            System.out.println("Not

        ds.unionByRank(3, 7);
        if (ds.findUPar(3) == ds.fin
            System.out.println("Same
        } else
            System.out.println("Not
    }
}
```

Output:

Not Same

Same

Time Complexity: The time complexity is O(4) which is very small and close to 1. So,

we can consider 4 as a constant.

> Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

« Previous Post
Striver Graph Series : Top Graph Interview Questions

Next Post »
Prim's Algorithm – Minimum Spanning Tree – C++ and Java: G-45

Load Comments

takeUforward

The best place to learn data structures, algorithms, most asked coding interview questions, real interview experiences free of cost.

Follow Us

DSA Playlist	DSA Sheets	Contribute
Array Series	Striver's SDE Sheet	Write an Article
Tree Series	Striver's A2Z DSA Sheet	
Graph Series	SDE Core Sheet	
DP Series	Striver's CP Sheet	

Copyright © 2023 takeuforward | All rights reserved