Disjoint Set

Set: collection of unnique items/elements : Check, Add, Remove -> O(1)

Disjoint Set: collection of sets. intersection of each set is empty.

- Find -> O(1)
- Union

Union Find

In [ ]:

```
```

In [1]:

```
class DisjointSet:

    def __init__(self, values):
        self.__data = {} # hash map
        for v in values:
            self.__data[v] = v

    def print(self):
        print(self.__data)


s1 = DisjointSet([0,1,2,3,4,5])
s1.print()
```

```
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}
```

In [ ]:

```
```

In [3]:

```
# add find operation
class DisjointSet:

    def __init__(self, values):
        self.__data = {} # hash map
        for v in values:
            self.__data[v] = v

    def find(self, element):
        if element not in self.__data:
            raise Exception("not found")

        if element == self.__data[element]:
            return element

        return self.find(self.__data[element])

    def print(self):
        print(self.__data)


s1 = DisjointSet([0,1,2,3,4,5])
s1.print()
print(s1.find(0), s1.find(1), s1.find(5))
```

```
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}
0 1 5
```

In [ ]:

```

```

In [6]:

```
# add union operation
class DisjointSet:

    def __init__(self, values):
        self.__data = {} # hash map
        for v in values:
            self.__data[v] = v

    def find(self, element):
        if element not in self.__data:
            raise Exception("not found")

        if element == self.__data[element]:
            return element

        return self.find(self.__data[element])

    def union(self, first, second):
        p1 = self.find(first)
        p2 = self.find(second)
        self.__data[p1] = p2

    def print(self):
        print(self.__data)


s1 = DisjointSet([0,1,2,3,4,5])
s1.print()

print()
s1.union(0,1)
s1.print()
print(s1.find(0) == s1.find(1), s1.find(1) == s1.find(4))

print()
s1.union(3,4)
s1.print()

print()
s1.union(0,3)
s1.print()
print(s1.find(0) == s1.find(1), s1.find(1) == s1.find(4))
```

```
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}

{0: 1, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}
True False

{0: 1, 1: 1, 2: 2, 3: 4, 4: 4, 5: 5}

{0: 1, 1: 4, 2: 2, 3: 4, 4: 4, 5: 5}
True True
```

In [ ]:

```
```

In [11]:

```
# add path compression
class DisjointSet:

    def __init__(self, values):
        self.__data = {} # hash map
        for v in values:
            self.__data[v] = v

    def find(self, element):
        if element not in self.__data:
            raise Exception("not found")

        if element == self.__data[element]:
            return element

        parent = self.find(self.__data[element])
        self.__data[element] = parent
        return parent

    def union(self, first, second):
        p1 = self.find(first)
        p2 = self.find(second)
        self.__data[p1] = p2

    def print(self):
        print(self.__data)


s1 = DisjointSet([0,1,2,3,4,5])
s1.print()

print()
s1.union(0,1)
s1.union(0,2)
s1.print()

print()
s1.union(3,4)
s1.union(4,5)
s1.print()

# (0,1,2)  (3,4,5)
print()
s1.union(0,4)
s1.print()

print()
print(s1.find(0) == s1.find(3))
s1.print()
```

```
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}

{0: 1, 1: 2, 2: 2, 3: 3, 4: 4, 5: 5}

{0: 1, 1: 2, 2: 2, 3: 4, 4: 5, 5: 5}

{0: 2, 1: 2, 2: 5, 3: 4, 4: 5, 5: 5}

True
{0: 5, 1: 2, 2: 5, 3: 5, 4: 5, 5: 5}
```

In [ ]:

Java

```java
package org.example.graph.disjoint;

import java.util.HashMap;
import java.util.Map;

public class DisjointSet {
    private Map<Integer, Integer> data;

    public DisjointSet(int[] values) {
        data = new HashMap<>();
        for (int v : values) {
            data.put(v, v);
        }
    }

    public int find(int element) {
        if (!data.containsKey(element)) {
            throw new IllegalArgumentException("Element not found");
        }
        if (element == data.get(element)) {
            return element;
        }
        int parent = find(data.get(element));
        data.put(element, parent);
        return parent;
    }

    public void union(int first, int second) {
        int p1 = find(first);
        int p2 = find(second);
        data.put(p1, p2);
    }

    public void print() {
        System.out.println(data);
    }

    public static void main(String[] args) {
        int[] values = {0, 1, 2, 3, 4, 5};
        DisjointSet s1 = new DisjointSet(values);
        s1.print();
        System.out.println();

        s1.union(0, 1);
        s1.union(0, 2);
        s1.print();
        System.out.println();

        s1.union(3, 4);
```

```
            s1.union(4, 5);
            s1.print();
            System.out.println();

            s1.union(0, 4);
            s1.print();
            System.out.println();

            System.out.println(s1.find(0) == s1.find(3));
            s1.print();
        }
    }
```

C++

```
#include <iostream>
#include <unordered_map>
#include <exception>
#include <vector>

class DisjointSet {
private:
    std::unordered_map<int, int> data;

public:
    DisjointSet(std::vector<int> values) {
        for (int v : values) {
            data[v] = v;
        }
    }

    int find(int element) {
        if (data.find(element) == data.end()) {
            throw std::runtime_error("Not found");
        }
        if (element == data[element]) {
            return element;
        }
        int parent = find(data[element]);
        data[element] = parent;
        return parent;
    }

    void unionSets(int first, int second) {
        int p1 = find(first);
        int p2 = find(second);
        data[p1] = p2;
    }

    void print() {
        for (const auto& pair : data) {
            std::cout << pair.first << ": " << pair.second << std::endl;
        }
        std::cout << std::endl;
    }
};
int main() {
    std::vector<int> values = {0, 1, 2, 3, 4, 5};
    DisjointSet ds(values);
    ds.print();
    ds.unionSets(0, 1);
    ds.unionSets(1, 2);
    ds.print();
```

```
        ds.unionSets(3,4);
        ds.unionSets(4,5);
        ds.print();

        ds.unionSets(0,4);
        ds.print();

        std::cout << (ds.find(0) == ds.find(3)) << std::endl;
        ds.print();

        return 0;
    }
```

In [ ]:

In [ ]:

Union By weigh (https://pybook.leangaurav.dev/chapters/datastructures/Disjoint%20Set.html#disjoint-set-using-union-by-size-weight)
Union By Height/Rank (https://pybook.leangaurav.dev/chapters/datastructures/Disjoint%20Set.html#disjoint-set-using-union-by-rank-height)

In [ ]:

Cycle Detection in Directed Graph

- BFS(Kahn's)
- DFS

Cycle Detection in Undirected Graph
Check if a graph is a tree

- Disjoint Set

In [ ]:

https://leetcode.com/problems/redundant-connection/description/

Java

```java
class Solution {
    public int[] findRedundantConnection(int[][] edges) {
         int parent[] = new int [1001];
        int rank[] = new int [1001];

        Arrays.fill(rank,0);
        Arrays.fill(parent,-1);

        for(int [] e: edges){
            int u = e[0];
            int v = e[1];
            int parentU = findSet(parent,u);
            int parentV = findSet(parent,v);
            if(parentU == parentV)
                return e;
            else
                makeUnion(parent,rank , u,v);
        }
        return new int []{-1,-1};
    }

    public int findSet(int parent[] , int x){
        if(parent[x] == -1 ) return x;
        int result = findSet(parent, parent[x]);

        return parent[x] = result;
    }
    public void makeUnion(int parent[] , int rank[] , int x , int y){
        int parX = findSet(parent,x);
        int parY = findSet(parent,y);

        if(parX == parY) return;

        if(rank[parX] < rank[parY]){
            parent[parX] = parY;
        } else if(rank[parY] < rank[parX]){
            parent[parY] = parX;
        } else {
            rank[parX]++;
            parent[parY] = parX;
        }
    }
}
```

Python

```
class Solution:
    def findRedundantConnection(self, edges: List[List[int]]) -> List[in
t]:

        class DisjointSet:
            def __init__(self, values):
                self.__data = {} # hash map
                for v in values:
                    self.__data[v] = v

            def find(self, element):
                if element not in self.__data:
                    raise Exception("not found")

                if element == self.__data[element]:
                    return element

                parent = self.find(self.__data[element])
                self.__data[element] = parent
                return parent

            def union(self, first, second):
                p1 = self.find(first)
                p2 = self.find(second)
                self.__data[p1] = p2

            def print(self):
                print(self.__data)

        vertices = []
        for i in range(1, len(edges) + 1):
            vertices.append(i)

        s = DisjointSet(vertices)
        for e in edges:
            v1 = e[0]
            v2 = e[1]

            if s.find(v1) == s.find(v2):
                return e

            s.union(v1, v2)

        return []
```

C++

```
# include <unordered_map>

class DisjointSet {

    std::unordered_map<int, int> _set;

    public:
        DisjointSet(int maxKey) {
            for (int i = 1; i <= maxKey; i++) {
                _set[i] = i;
            }
        }

        int find(int key) {
            if (_set[key] == key) {
                return key;
            }
            _set[key] = find(_set[key]);
            return _set[key];
        }

        void _union(int k1, int k2) {
            int p1 = find(k1);
            int p2 = find(k2);

            _set[p2] = p1;
        }

        bool isSameSet(int k1, int k2) {
            return find(k1) == find(k2);
        }
};

class Solution {
public:
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {
        vector<int> res;
        if (edges.size() == 0) {
            return res;
        }

        int maxVertex = edges[0][0];
        for (auto edge: edges) {
            if (edge[0] > maxVertex) {
                maxVertex = edge[0];
            }

            if (edge[1] > maxVertex) {
                maxVertex = edge[1];
```

```
                }
            }

            DisjointSet s = DisjointSet(maxVertex);

            for (auto edge: edges) {
                int v1 = edge[0];
                int v2 = edge[1];

                if (s.find(v1) == s.find(v2)) {
                    res = edge;
                    break;
                }
                s._union(v1, v2);
            }
            return res;
        }
    };
```

In [ ]:

In [ ]:

Spanning tree (E:Edges, V: Vertices)

- Set of vertices connected by edges, such that no cycle exists
- Min no. of edges -> (V-1)
- only one path to reach from one node to another

MST: Minimum Spanning Tree of a Graph

- A spanning tree of a graph that has minimum weight

Kruskals Algorithm (Uses Sorting)

```
- Sort the edges by weight
- create a disjoint set.
- count=0
- totalWeight=0
- while count != (V-1)
    pick an edge from list of edges
    check if the edge forms a cycle:
        skip

    add the edge to disjoint set
    increase count (+=1)
    update totalWeight
```

Prims Algorithm (Uses heap)

DIY

In [ ]:

```
https://leetcode.com/problems/min-cost-to-connect-all-points/description/
```

Python

```
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:

        class Edge:
            def __init__(self, x1,y1, x2, y2):
                self.x1 = x1
                self.y1 = y1
                self.x2 = x2
                self.y2 = y2
                self.dist = abs(x2-x1) + abs(y2-y1)

            def __lt__(self, other):
                return self.dist < other.dist

            def __repr__(self):
                return f"({self.x1}, {self.y1}, {self.x2}, {self.y2}, {se
lf.dist})"

        class DisjointSet:

            def __init__(self, values):
                self.__data = {} # hash map
                for v in values:
                    self.__data[v] = v

            def find(self, element):
                if element not in self.__data:
                    raise Exception("not found")

                if element == self.__data[element]:
                    return element

                return self.find(self.__data[element])

            def union(self, first, second):
                p1 = self.find(first)
                p2 = self.find(second)
                self.__data[p1] = p2

            def print(self):
                print(self.__data)

        edges = []
        for i in range(len(points)):  # for (int i=0; i < n; i++)
            for j in range(i+1, len(points)): # for(int j=i+1, j<n;j++)
                p1 = points[i]
                p2 = points[j]
                edges.append( Edge(p1[0], p1[1], p2[0], p2[1]))
```

```
    edges.sort()

    points_tuple = [tuple(p) for p in points]

    s = DisjointSet(points_tuple)
    count = len(points) - 1
    totalWt = 0
    i = 0
    while count > 0 and i < len(edges):
        edge = edges[i]
        i += 1

        p1 = (edge.x1, edge.y1)
        p2 = (edge.x2, edge.y2)
        if s.find(p1) == s.find(p2):
            continue

        count -= 1
        totalWt += edge.dist
        s.union(p1, p2)

    return totalWt
```

C++

```
# include <unordered_map>

class DisjointSet {

    std::unordered_map<int, int> _set;

    public:
        DisjointSet(int maxKey) {
            for (int i = 1; i <= maxKey; i++) {
                _set[i] = i;
            }
        }

        int find(int key) {
            if (_set[key] == key) {
                return key;
            }

            // int root = key;
            // while (root != _set[root] )
            //     root = _set[root];

            // while (key != root)
            _set[key] = find(_set[key]);
            return _set[key];
        }

        void _union(int k1, int k2) {
            int p1 = find(k1);
            int p2 = find(k2);

            _set[p2] = p1;
        }

        bool isSameSet(int k1, int k2) {
            return find(k1) == find(k2);
        }
};


struct Edge {
    int src;
    int dest;
    int weight;

    Edge(int src, int dest, int weight) {
        this->src = src;
        this->dest = dest;
        this->weight = weight;
```

```
        }

        int operator < (Edge other) {
            return weight < other.weight;
        }

    };


    class Solution {

        int distance(int x1,int x2,int y1,int y2) {
            return  abs(x2-x1) + abs(y2-y1);
        }

    public:
        int minCostConnectPoints(vector<vector<int>>& points) {
            // generate all edge pair weights
            // each point in input is denoted by its index.
            vector<Edge> edges;
            for (int i = 0; i < points.size(); i++) {
                for (int j = i + 1; j < points.size();j++) {
                    int d = distance(points[i][0], points[j][0], points[i]
    [1], points[j][1]);
                    edges.push_back(Edge(i,j, d));
                }
            }

            DisjointSet ds = DisjointSet(points.size());

            std::sort(edges.begin(), edges.end());

            int distSum = 0;
            int count = 0;
            for (auto &edge: edges) {
                // V-1 edges have been added, break
                if (count == points.size() - 1) {
                    break;
                }
                if (ds.find(edge.src) == ds.find(edge.dest)) {
                    continue;
                }

                count += 1;
                distSum += edge.weight;

                ds._union(edge.src, edge.dest);
            }

            return distSum;
```

```
            }
        }
```