# UNION-FIND ( DISJOINT-SET)
## explicación e implementacion
## Data Estructure

Union Find es una estructura de datos que modela una colección de conjuntos disjuntos (disjoint-set) y esta basado en 2 operaciones:

1. **Find( X ):** determina a cual conjunto pertenece el elemento **X**. Esta operación puede ser usada para determinar si 2 elementos están o no en el mismo conjunto.

2. **Union( X , Y ):** Une todo el conjunto al que pertenece **X** con todo el conjunto al que pertenece **Y**, dando como resultado un nuevo conjunto basado en los elementos tanto de **X** como de **Y**.

Pueden sonar extraños los nombres de las operaciones. Se llaman así porque se puede pensar en cada componente conexa como un conjunto. Añadir una arista entre dos vértices es equivalente a unir dos conjuntos. Saber si dos vértices se encuentran en la misma componente conexa se puede interpretar como saber que dos vértices se encuentran en el mismo conjunto, esta misma se puede expresar con teoría de conjuntos.

representamos los conjuntos como un árbol, donde cada nodo mantiene la información de su nodo padre, la raíz del árbol será el elemento representativo de todo el conjunto. Por lo tanto basta con declarar un arreglo que contenga los elementos del padre de un determinado elemento

```
#include<iostream>
using namespace std;

Const int MAXN = int(1e6);

int  padre[MAXN];
```

Para esta representacion es necesario inicializar cada padre como nodo independiente de cualquier conjunto para este ejemplo podremos crear una funcion llamada **INICIALIZAR( ARR[], N )** en donde mandamos como parametros nuestro arreglo de padres y su limite dependiendo del problema:

```
void inicializar( int arr[], int n )
{
        for ( int c = 0; c<=n; c++ )
                padre[c] = c;
        return;
}
```

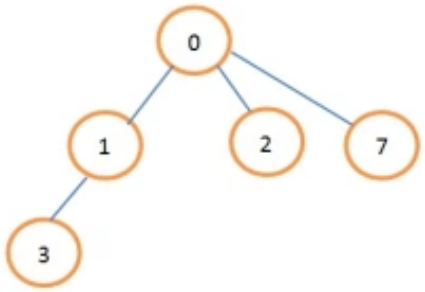
Joseph Hynimoto Aguilar Lopez

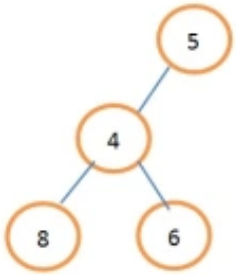
**Fucnion Find – Find( x )**

Esta funcion determina a cual componente conexa pertenece un vértice X determinado, ello lo hace retornando el vértice raíz del árbol en el que se encuentra el vértice X.
Recordemos que en un árbol entre todo par de vértices hay un único camino que los une(claro que descartando los caminos que repiten vértices). Consideremos un árbol T = (V, A), vamos a elegir arbitrariamente un vértice r, al cual llamaremos raíz. Está claro que para cualquier vértice v ∈ V existe un único camino que lo une con r, vamos a llamar P[r] al segundo vértice de dicho camino(el primer vértice del camino claramente es v), si v = r entonces el camino consta solamente de un vértice y por conveniencia diremos que P[r] = r

```
int find(x)
{
        if( padre[x] != x )
                return find(x);
        else
          return x;
}
```

Por ejemplo tengamos las siguientes componentes conexas vistas como arboles:




| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 0 | 0 | 1 | 5 | 5 | 4 | 0 | 4 |

Al aplicar la funcion Find(3) este nos **retornaria** un entero **1** se muesta en la ilustracion.

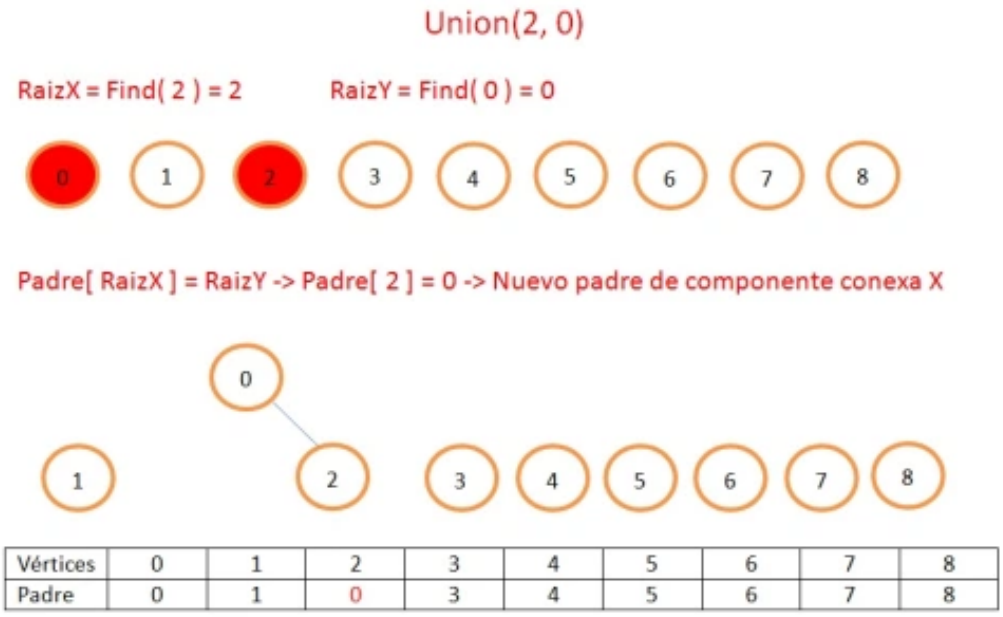
Joseph Hynimoto Aguilar Lopez

**Funcion Union – Union( x , y )**

Esta funcion me permite unir 2 componentes conexas, ello se realiza por lo siguiente:

1. Obtenemos la raíz del vértice x.
2. Obtenemos la raíz del vértice y.
3. Actualizamos el padre de alguna de las raíces, asignándole como nuevo padre la otra raíz.

Podemos tomar el siguiente ejemplo:




| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 1 | 0 | 3 | 4 | 5 | 6 | 7 | 8 |

Como se pudo observar primero realizamos los pasos 1 y 2 para hallar las raíces de ambos vértices y finalmente el paso 3 para actualizar el padre de una de las componentes conexas, en este caso se actualizará el padre de la componente conexa X.
si en lugar de guardar todo el grafo simplemente guardamos un árbol de expansión del grafo original entonces podemos a cada componente conexa asignarle una raíz, si hiciéramos esto ya no necesitaríamos usar listas de adyacencia ni búsquedas, simplemente guardaríamos para cada vértice v el valor de P[v], y de esta manera desde cualquier vértice podríamos llegar a la raíz de su componente conexa

**Si dos vértices estan conectados, entonces estan en la misma componente conexa. Si estan en la misma componente conexa entonces estan conectados con la misma raíz. Análogamente si estan conectados con la misma raíz entonces están en la misma componente conexa y por tanto estan conectados.**

Si queremos unir dos componentes conexas X y Y, basta con hacer que la raíz de X apunte a la raíz de Y(o viceversa), de esa manera todos los nodos X estarán conectados con la misma raíz que todos los nodos de Y.

```
void union ( int x, int y )
{
        padre[find(x)] = find(y);
        return;

}
```

Joseph Hynimoto Aguilar Lopez

**COMO PODEMOS SABER SI DOS VERTICES ESTAN CONECTADOS ?**

Despues de ver la funcion union podremos habernos percatado de que hay un inconveniente, al momento de querer saber si un vertice X esta conectado con un vertice Y, nos podremos dar cuenta de la logica que implica esto, y es que si un vertice X pertenece a un conjunto y el vertice X esta conectado a un Vertice Y perteneciente a un conjunto distinto al de X entonces el conjunto de X esta conectado al conjunto Y, en otras palabras X pertenece a Y.
Y como dos vértices estan conectados sí y solo sí estan conectados con la misma raíz entonces nuestro procedimiento de pertenencia queda de la siguiente manera:

```
bool pertenencia( int x, int y )
{
        if( find(x) == find(y ) )
                return true;
        return false;

}
```

Joseph Hynimoto Aguilar Lopez

# Mejorando el Rendimiento de Unión-Find

La idea de esta mejora es que cada nodo que visitemos en el camino al nodo raíz puede ser conectado directamente hacia la raíz, es decir al terminar de usar el método Find todos los nodos que visite tendrán como padre la raíz directamente.

- **Compresión de Caminos:** Al momento de retornar la raíz en la recursión actualizamos el padre de cada vértice visitado como la raíz encontrada.

Para la implementación basta con una pequeña modificación en el método Find y es la de modificar lo siguiente:

```
int find(x)
{
        if( padre[x] != x )
                return  padre[x] = find(x);
        else
          return x;
}
```

Como cada arista que no apunta directamente a la raíz se recorre a lo más una vez y todas las aristas son creadas mediante una operación de unión, entonces el algoritmo toma un tiempo de O(u + f) donde u es el número de llamadas a la operación unión, y f es el número de llamadas a la operación find.

Joseph Hynimoto Aguilar Lopez

**BIBLIOGRAFIA UTILIZADA**


- Arias, J. A. F. jariasf. (2012b, 2 abril). DISJOINT-SET: UNION FIND [Publicación en un blog]. Recuperado 16 noviembre, 2019, de https://jariasf.wordpress.com/2012/04/02/disjoint-set-union-find/

- Vargas Azcona, L. E. V. A. Luis E.. (2010). *Problemas y Algoritmos* (Ed. rev.). Recuperado de https://omegaup.com/img/libropre3.pdf