

Algorithms and More

Blog about programming!

DISJOINT-SET: UNION FIND

Publicado el [abril 2, 2012](#) | [8 comentarios](#)

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

- **Find(A)**: Determina a cual conjunto pertenece el elemento A. Esta operación puede ser usada para determinar si 2 elementos están o no en el mismo conjunto.
- **Union(A, B)**: Une todo el conjunto al que pertenece A con todo el conjunto al que pertenece B, dando como resultado un nuevo conjunto basado en los elementos tanto de A como de B.

Estas operaciones me servirán para la implementación del [algoritmo de Kruskal](#) o problemas que involucran particionamiento como encontrar las componentes conexas en un grafo.

Implementación

Explicaré dos formas de implementar Union-Find, la forma básica y sus mejoras. Al final encontrarán los códigos en C++ y JAVA:

Bosques de Conjuntos Disjuntos

En esta implementación 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:

```
1 | #define MAX 10005
2 | int parent[ MAX ];    //Este arreglo contiene el padre del i-esimo nodo
```

Método MakeSet

Para este tipo de implementación agregaremos un método que me inicializará los conjuntos, el cual llamaremos MakeSet:

```
1 | //Método de inicialización
2 | void MakeSet( int n ){
3 |     for( int i = 0 ; i < n ; ++i ){
4 |         padre[ i ] = i;    //Inicialmente el padre de cada vértice es el
5 |     }
6 | }
```

Tengamos por ejemplo 9 vértices, inicializamos por medio de MakeSet:

MakeSet(9)



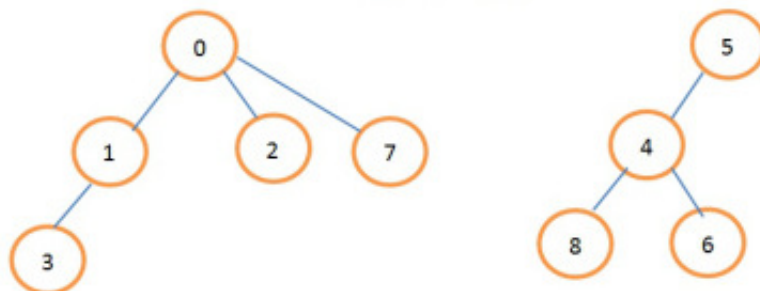
| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Una vez inicializado podemos unir dos componentes conexas o preguntar con el método find si un determinado vértice pertenece a la misma componente conexas de otro vértice.

Método Find – Find(x)

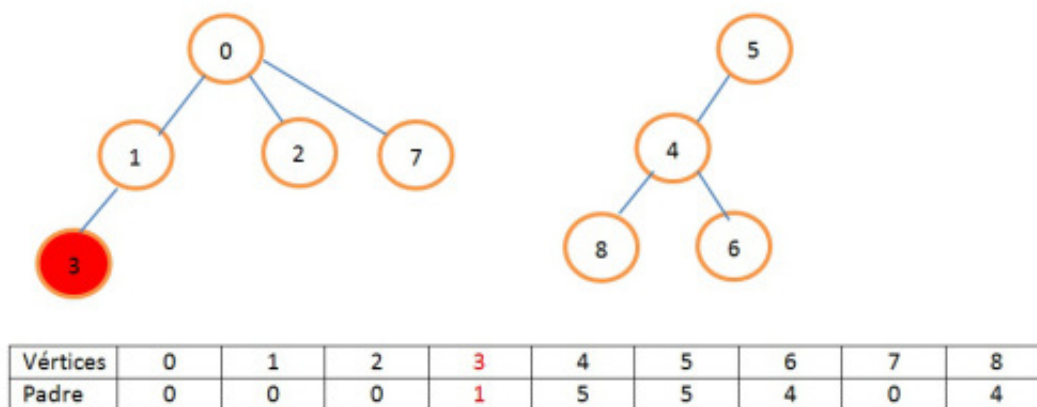
Como se explicó al inicio este método determina a cual componente conexas pertenece un vértice X determinado, ello lo hace retornando el vértice raíz del árbol en el que se encuentra el vértice X.

Por ejemplo tengamos las siguientes componentes conexas vistas como arboles:

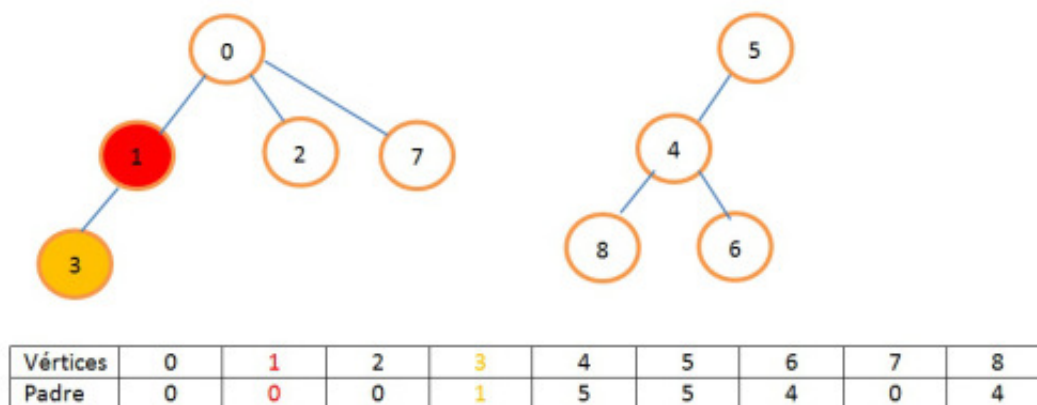


| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 0 | 0 | 1 | 5 | 5 | 4 | 0 | 4 |

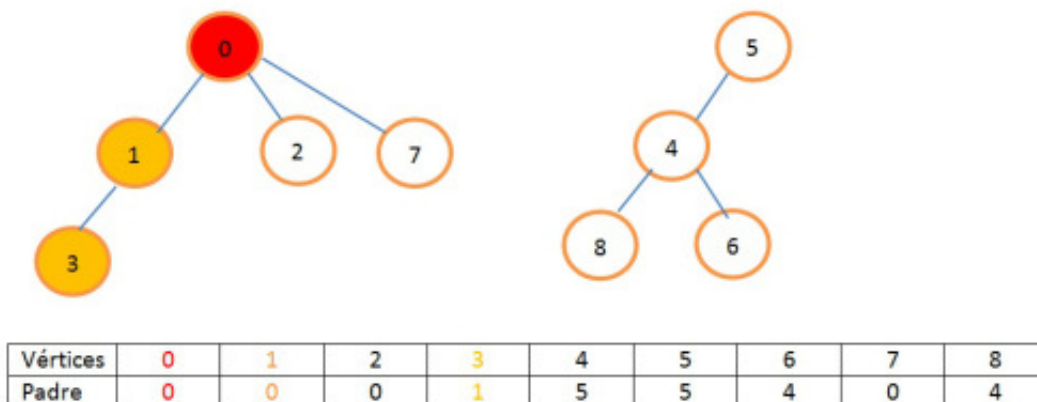
Deseo saber la raíz de la componente conexas ala que pertenece el vértice 3:

Find(3)

Al hacer Find(3) partimos del vértice 3 y subimos hasta la raíz viendo su padre, en este caso el padre[3] = 1 por lo tanto:

Find(3)

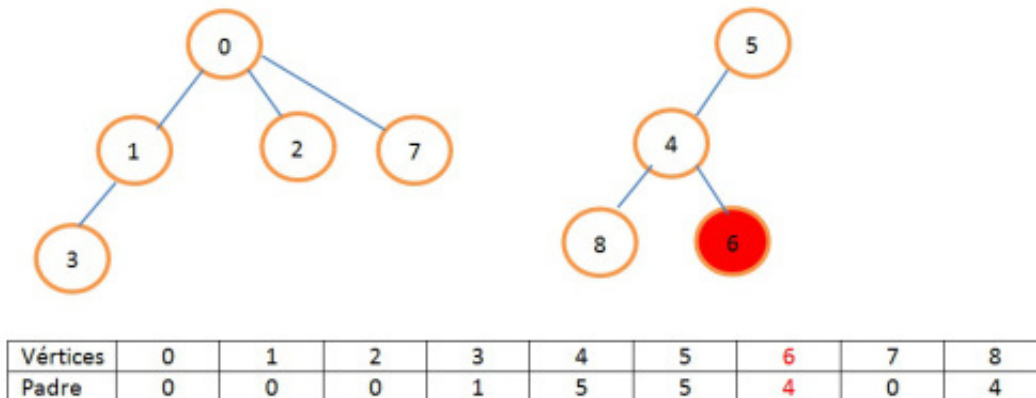
El vértice actual ahora es el vértice 1 y hacemos lo mismo que antes, padre[1] = 0.

Find(3)

Ahora estamos en vértice 0, como el padre[0] = 0 entonces estamos en la raíz y la retornamos.

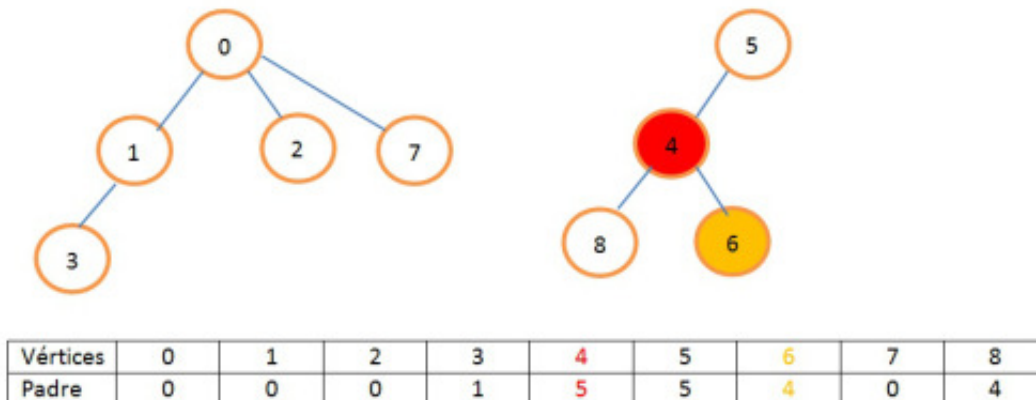
Veamos otro ejemplo, en este caso deseo saber la raíz para el vértice 6:

Find(6)



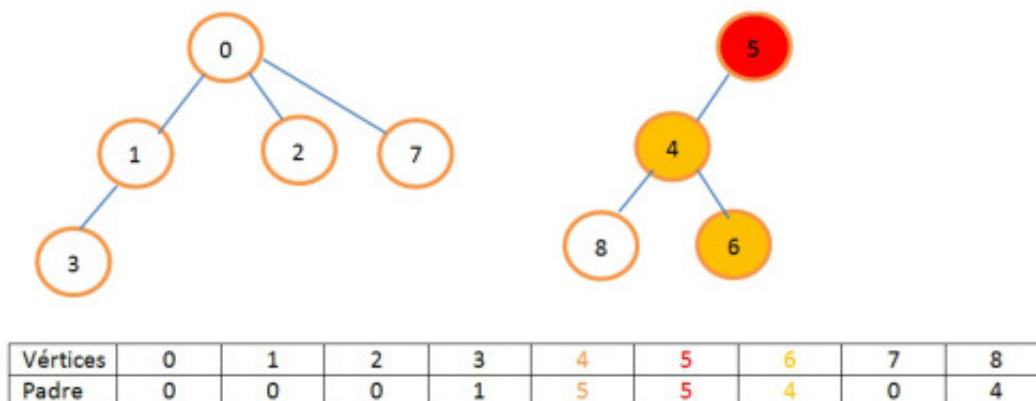
El vértice actual es 6 vemos su padre[6] = 4 y continuamos:

Find(6)



El vértice actual es 4, padre[4] = 5, todavía posee un padre por lo tanto continuamos:

Find(6)



El vértice actual es 5 en este caso padre[5] = 5 hemos llegado a la raíz y la retornamos.

Por la parte de código la implementación se realizara de manera recursiva, teniendo como caso base:

```
1 | if( x == padre[ x ] ){           //Si estoy en la raíz
2 |     return x;                   //Retorno la raíz
3 | }
```

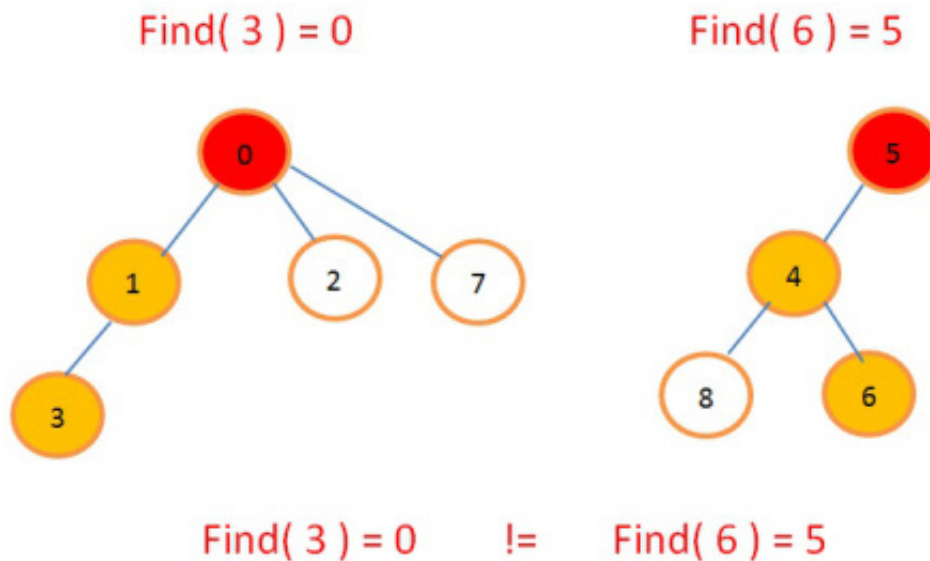
Si no estoy en la raíz continúo subiendo hasta encontrar la raíz y retornarla:

```
1 | else return Find( padre[ x ] ); //De otro modo busco el padre del nodo actu
```

El código completo:

```
1 | //Método para encontrar la raíz del vértice actual x
2 | int Find( int x ){
3 |     if( x == padre[ x ] ){           //Si estoy en la raíz
4 |         return x;                   //Retorno la raíz
5 |     }
6 |     else return Find( padre[ x ] ); //De otro modo busco el padre del vérti
7 | }
```

Del ejemplo anterior podemos agregar un método que me permita saber si dos vértices están o no en la misma componente conexas, veamos:



En este caso no están en la misma componente conexas. Para saber si están en la misma componente conexas pues es simplemente ver si poseen la misma raíz, por lo tanto en código tendríamos:

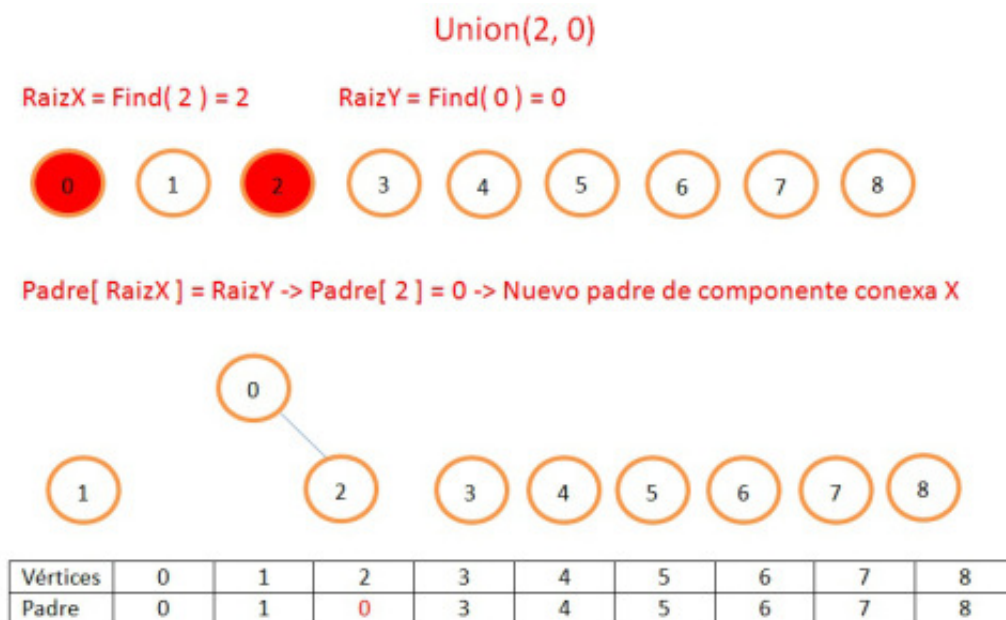
```
1 | //Método que me determina si 2 vértices estan o no en la misma componente c
2 | bool sameComponent( int x , int y ){
3 |     if( Find( x ) == Find( y ) ) return true;   //si poseen la misma raíz
4 |     return false;
5 | }
```

Método Union – Union(x , y)

Como se explicó al inicio este método 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.

Por ejemplo:



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 conexas X. Continuemos:

Union(7, 2)

RaizX = Find(7) = 7

RaizY = Find(2) = 0



Padre[RaizX] = RaizY -> Padre[7] = 0 -> Nuevo padre de componente conexas X



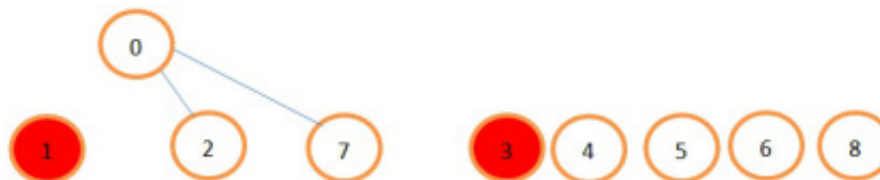
| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 1 | 0 | 3 | 4 | 5 | 6 | 0 | 8 |

Al igual que el caso anterior el nuevo padre del vértice 7 es el vértice 0.

Union (3, 1)

RaizX = Find(3) = 3

RaizY = Find(1) = 1



Padre[RaizX] = RaizY -> Padre[3] = 1 -> Nuevo padre de componente conexas X



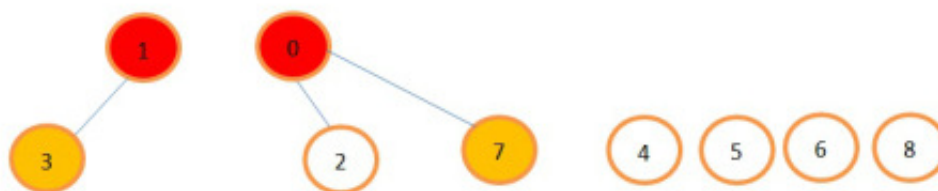
| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 1 | 0 | 1 | 4 | 5 | 6 | 0 | 8 |

En este caso hemos realizado Union(3 , 1), entonces el nuevo padre del vértice 3 es el vértice 1. Hasta el momento tenemos 6 componentes conexas.

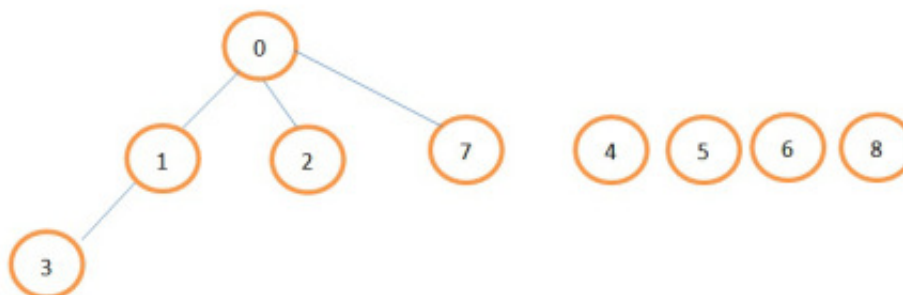
Union(3, 7)

RaizX = Find(3) = 1

RaizY = Find(7) = 0



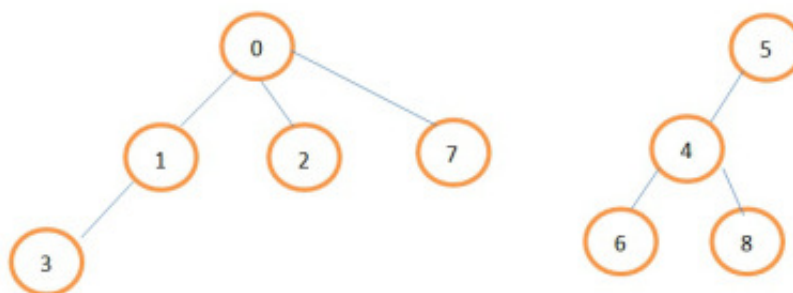
Padre[RaizX] = RaizY -> Padre[1] = 0 -> Nuevo padre de componente conexas X



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

En este caso estamos uniendo dos componentes con más vértices y como se puede observar solo es necesario actualizar el puntero de la raíz de una de las componentes.

Union(6 , 4) -> Union(8 , 4) -> Union(4 , 5)



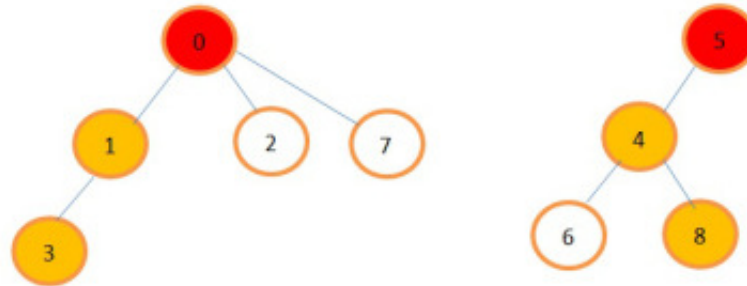
| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 0 | 0 | 1 | 5 | 5 | 4 | 0 | 4 |

En la imagen anterior se hizo Union(6 , 4) -> Union(8 , 4) -> Union(4 , 5) en ese orden.

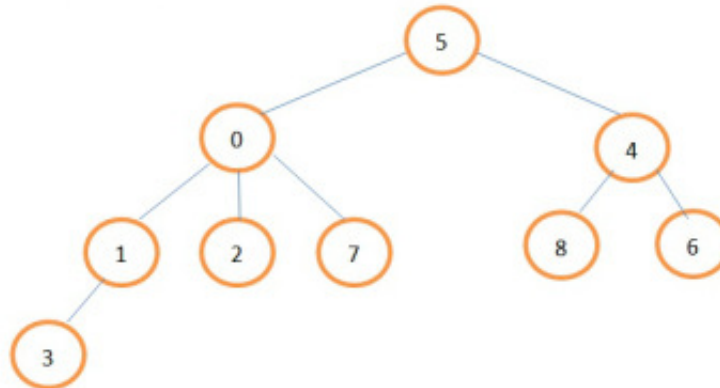
Union(3, 8)

RaizX = Find(3) = 0

RaizY = Find(8) = 5



Padre[RaizX] = RaizY -> Padre[0] = 5 -> Nuevo padre de componente conexas X



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

En esta última imagen hemos unido todos los nodos obteniendo una componente conexas.

En código tendríamos:

```

1 //Método para unir 2 componentes conexas
2 void Union( int x , int y ){
3     int xRoot = Find( x ); //Obtengo la raíz de la componente del vértice x
4     int yRoot = Find( y ); //Obtengo la raíz de la componente del vértice y
5     padre[ xRoot ] = yRoot; //Mezclo ambos arboles o conjuntos, actualiza padre
6 }

```

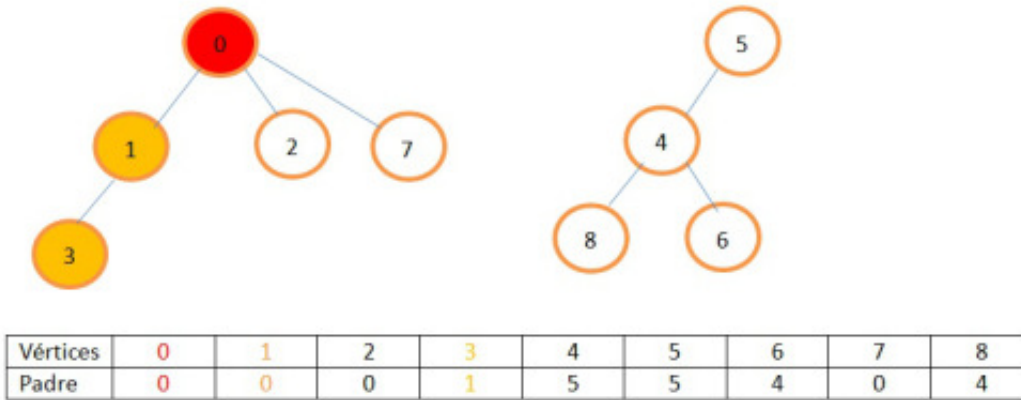
Como pudimos observar cada vez que hagamos unión entre componentes con mayor numero de nodos el árbol tiende a ser un árbol desbalanceado.

Mejoras

Implementación de Find por Compresión de Caminos

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. Veamos la mejora en el primer ejemplo del método Find antes visto:

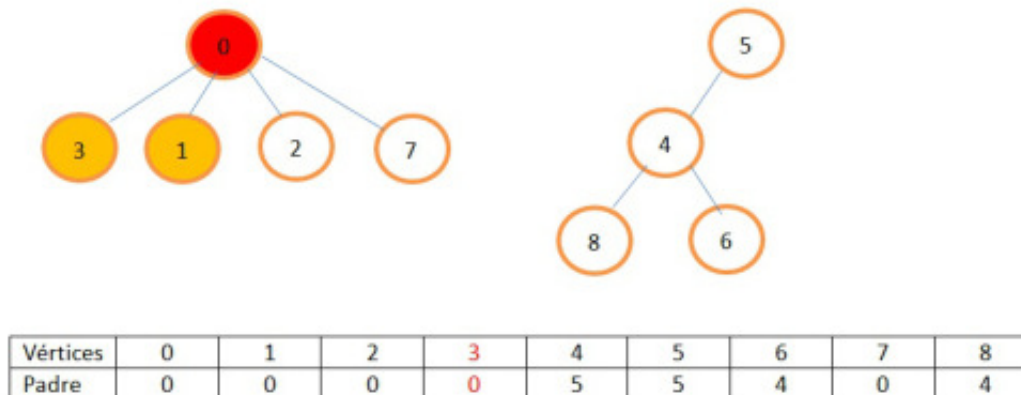
Find(3)



Al aplicar Find(3) estamos en vértice 0, como el padre[0] = 0 entonces estamos en la raíz y la retornamos.

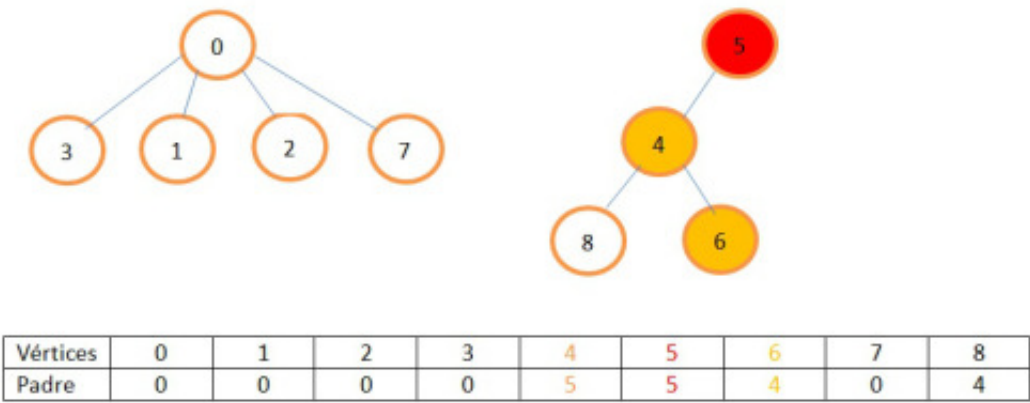
Compresión de Caminos: Al momento de retornar la raíz en la recursión actualizamos el padre de cada vértice visitado como la raíz encontrada.

Find(3)



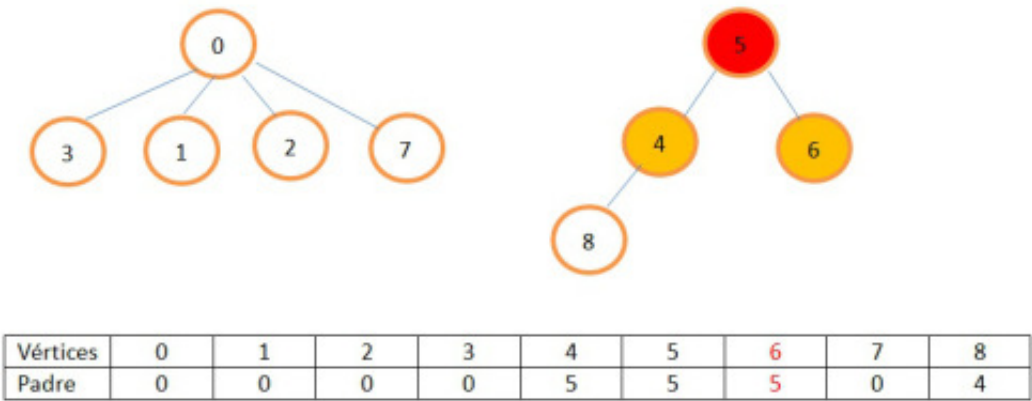
Veamos el otro ejemplo, donde realizamos Find(6):

Find(6)



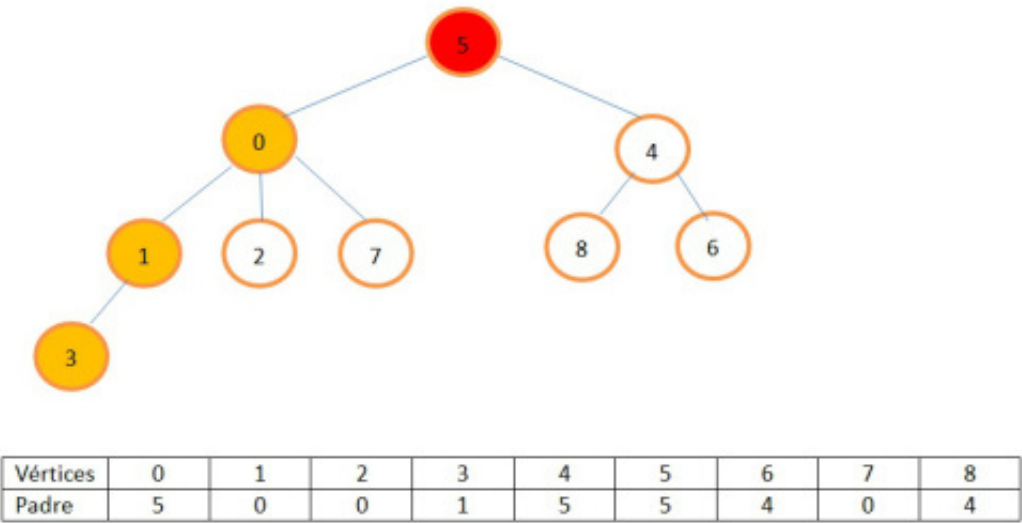
Realizamos la compresión de caminos, obteniendo:

Find(6)

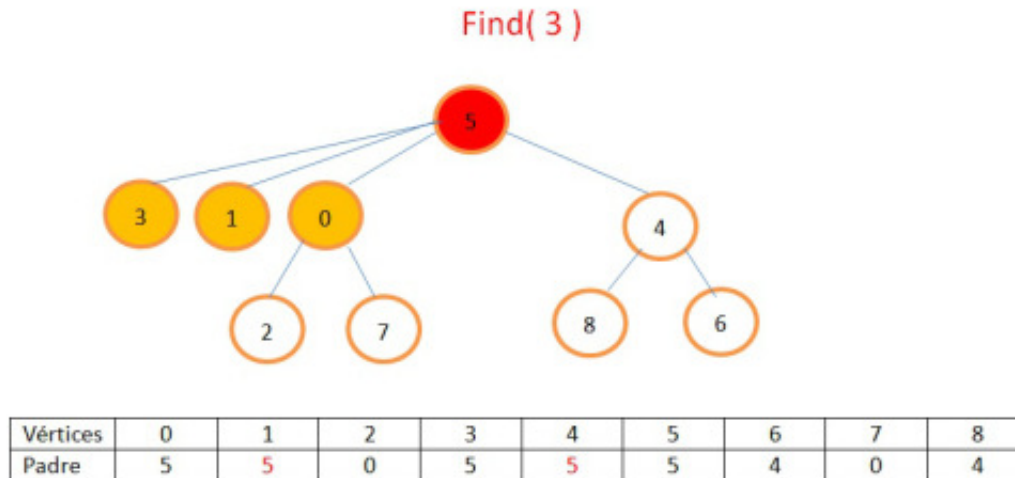


Veamos ahora de la siguiente imagen realizo Find(3):

Find(3)



Aplicamos compresión de caminos obteniendo:



Como hemos podido observar esta mejora me servirá a futuro ya que si deseo saber ahora el Find(3) no es necesario recorrer varias veces el padre del vértice 3 puesto que por la compresión de caminos el vértice 3 esta conectado directamente a la raíz.

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

```
1 //else return Find( padre[ x ] ); //De otro modo busco el padre del vértice
2 else return padre[ x ] = Find( padre[ x ] ); //Compresion de caminos
```

Lo que hacemos es cambiar el puntero actual al padre de un nodo en el recorrido, apuntando a la raíz directamente. Con ello mejoramos la eficiencia por el hecho de que en futuras llamadas de alguno de los nodos vistos ya no tendremos que realizar todo el recorrido hacia la raíz simplemente revisara el padre actual del nodo que es la raíz.

Todo el método:

```
1 //Método para encontrar la raíz del vértice actual X
2 int Find( int x ){
3     if( x == padre[ x ] ){                //Si estoy en la raíz
4         return x;                        //Retorno la raíz
5     }
6     //else return Find( padre[ x ] ); //De otro modo busco el padre del vér
7     else return padre[ x ] = Find( padre[ x ] ); //Compresion de caminos
8 }
```


Implementación de Unión por Rango

La idea de esta mejora básicamente lo que hará será unir el árbol de menor rango a la raíz del árbol con el mayor rango, los rangos los almacenamos en un arreglo y sufrirán modificación al momento de la unión. Los pasos para la implementación son los siguientes:

1. Obtenemos la raíz del vértice x.
2. Obtenemos la raíz del vértice y.
3. Si el rango de X es mayor que el rango de Y entonces.
 - Actualizamos el padre de la raíz de Y asignándole como nuevo padre la raíz de X.
4. De otro modo:
 - Actualizamos el padre de la raíz de X asignándole como nuevo padre la raíz de Y.
 - Si los rangos de ambas raíces son iguales incremento el rango de la nueva raíz en este caso incremento el rango de la raíz del vértice Y.

En resumen cada vez que hagamos unión entre dos componentes, la componente de mayor altura(rango) siempre predominara sobre la de menor altura(rango). Veamos la inicialización:

MakeSet(9)



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

Código:

```

1 //Método de inicialización
2 void MakeSet( int n ){
3     for( int i = 0 ; i < n ; ++i ){
4         padre[ i ] = i;      //Inicialmente el padre de cada vértice es el
5         rango[ i ] = 0;      //Altura o rango de cada vértice es 0
6     }
7 }
```

Veamoslo ahora con ejemplo antes visto en el método Union:

Union(2, 0)

RaizX = Find(2) = 2

RaizY = Find(0) = 0



Rango[RaizX] == Rango[RaizY] --> Rango[2] == Rango[0] == 0 --> Rangos iguales

Padre[RaizX] = RaizY --> Padre[2] = 0 --> Nuevo padre de componente conexa X

Rango[0] = 1 --> Incrementamos rango de nuevo padre.



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

La imagen anterior muestra el caso de rangos iguales, por tanto aumentamos el rango.

Union (7, 2)

RaizX = Find(7) = 7

RaizY = Find(2) = 0



Rango[RaizX] < Rango[RaizY] --> Rango[7] < Rango[0] --> Rangos diferentes

Padre [RaizX] = RaizY --> Padre[7] = 0 --> Nuevo padre de componente conexa X



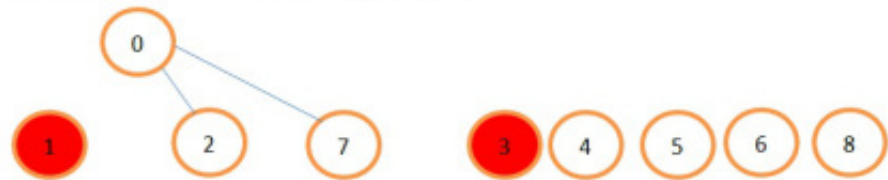
| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 1 | 0 | 3 | 4 | 5 | 6 | 0 | 8 |
| Rango | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

En este caso tenemos rangos diferentes, además no cumple con el paso 3 porque el rango del vértice 7 es 0 y no es mayor que el rango del vértice 0 que es 1, al no tener rangos iguales los rangos quedan como estaban.

Union (3, 1)

RaizX = Find(3) = 3

RaizY = Find(1) = 1



Rango[RaizX] == Rango[RaizY] --> Rango[3] == Rango[1] == 0 --> Rangos iguales

Padre [RaizX] = RaizY --> Padre[3] = 1 --> Nuevo padre de componente conexa X

Rango[1] = 1 --> Incrementamos rango de nuevo padre.



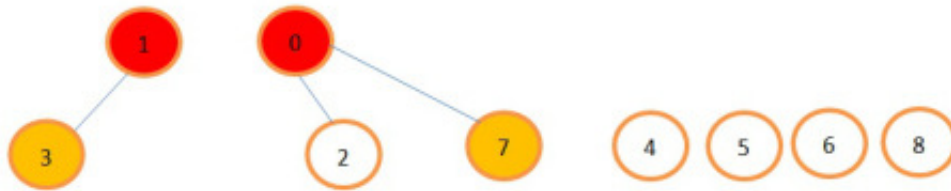
| | | | | | | | | | |
|----------|---|---|---|---|---|---|---|---|---|
| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Padre | 0 | 1 | 0 | 1 | 4 | 5 | 6 | 0 | 8 |
| Rango | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Este caso es similar al primero, por tanto el rango de la raíz de la componente a la que pertenece el vértice Y se incrementa.

Union (3, 7)

RaizX = Find(3) = 1

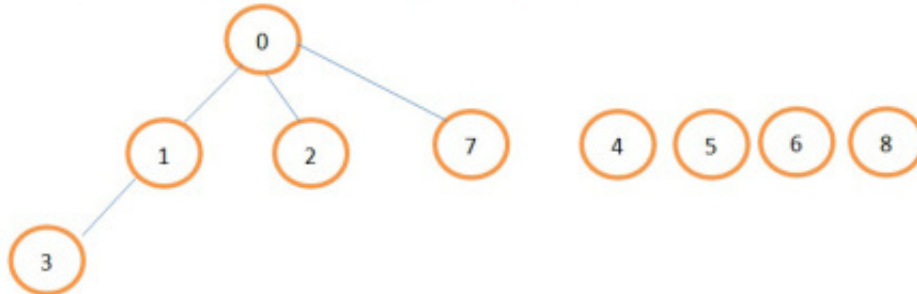
RaizY = Find(7) = 0



Rango[RaizX] == Rango[RaizY] --> Rango[1] == Rango[0] == 1 --> Rangos iguales

Padre [RaizX] = RaizY --> Padre[1] = 0 --> Nuevo padre de componente conexas X

Rango[0] = 2 --> Incrementamos rango de nuevo padre.



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

Caso similar a los anteriores con rangos iguales por parte de las raíces.

Union(6, 4) -> Union(8, 4)

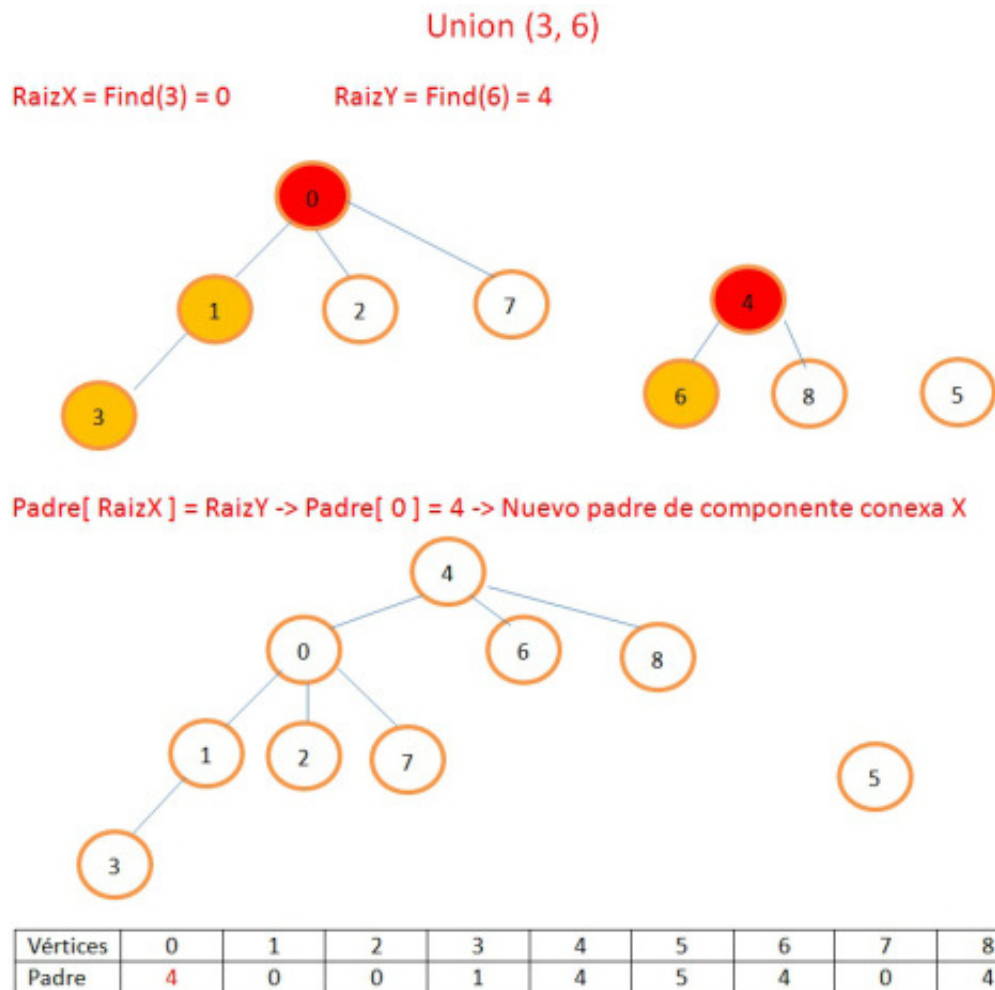


| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 0 | 0 | 1 | 4 | 5 | 4 | 0 | 4 |
| Rango | 2 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

En la imagen anterior se hizo Union(6 , 4) -> Union(8 , 4) -> en ese orden, las actualizaciones las vemos en la tabla de la imagen.

Hasta el momento no hemos notado diferencia alguna entre la Unión normal con la Unión por Rangos, haremos ahora Union(3, 6) por ambos métodos:

1. Unión Normal



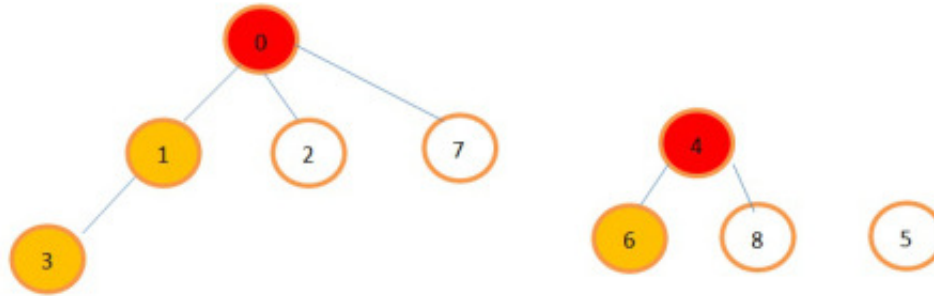
La altura del nuevo árbol formado es de 3. Ahora veamos la diferencia con la unión por rangos.

2. Unión por Rango

Union (3, 6)

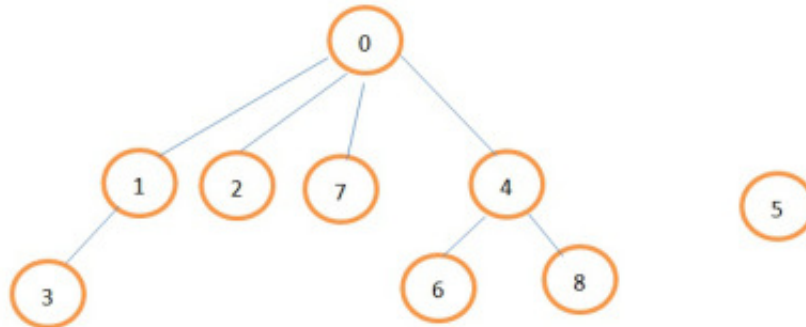
RaizX = Find(3) = 0

RaizY = Find(6) = 4



Rango[RaizX] > Rango[RaizY] --> Rango[0] > Rango[4] --> 2 > 1 --> Rangos diferentes

Padre [RaizY] = RaizX --> Padre[4] = 0 --> Nuevo padre de componente conexa Y



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

La altura del nuevo árbol es 2, este método mantiene el árbol tan balanceado como sea posible, para mayor número de vértices se verá la diferencia en cuanto a la eficiencia. Asimismo podemos decir que el rango del vértice raíz de una componente conexa es la altura de dicha componente.

El código es el siguiente:

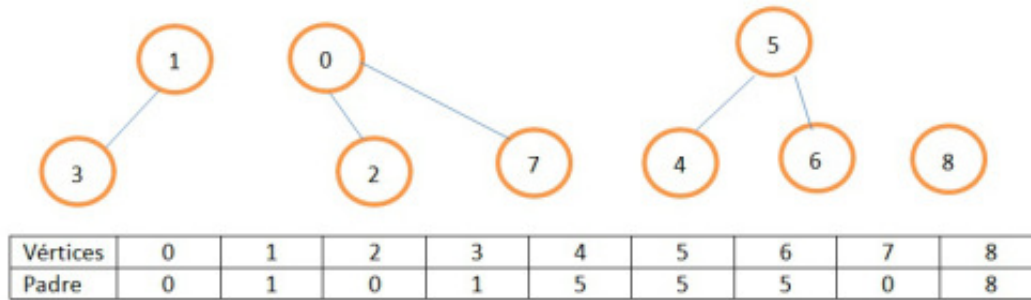
```

1 //Método para unir 2 componentes conexas usando sus alturas (rangos)
2 void UnionbyRank( int x , int y ){
3     int xRoot = Find( x ); //Obtengo la raíz de la componente del vérti
4     int yRoot = Find( y ); //Obtengo la raíz de la componente del vérti
5     if( rango[ xRoot ] > rango[ yRoot ] ){ //en este caso la altura de la
6                                             //mayor que la altura de la con
7         padre[ yRoot ] = xRoot; //el padre de ambas componentes
8     }
9     else{ //en este caso la altura de la componente del
10         padre[ xRoot ] = yRoot; //el padre de ambas componentes
11         if( rango[ xRoot ] == rango[ yRoot ] ){ //si poseen la misma altur
12             rango[ yRoot ]++; //incremento el rango de la nue
13         }
14     }
15 }

```

Número de Componentes Conexas

Para saber el número de componentes conexas mediante unión-find, basta con contar los vértices que sean raíces de su componente conexas, por ejemplo:

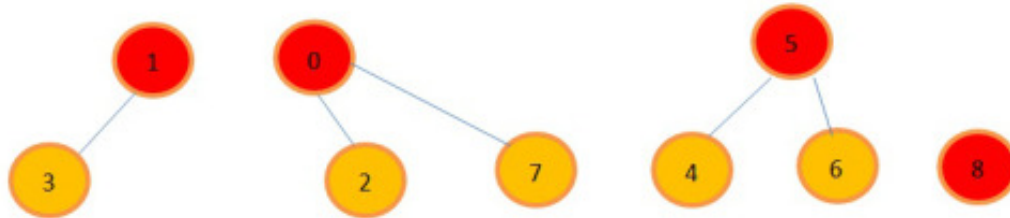


De la imagen podemos hallar la cantidad de componentes conexas de dos formas:

- Verificamos simplemente si $\text{padre}[x] = x$.

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

- Verificamos si $\text{Find}(x) = x$, al llamar a Find realizará la compresión de caminos.



En ambos casos será necesario recorrer el número de vértices y realizar la verificación, en este ejemplo la respuesta será 4 componentes conexas. Adicionalmente podemos almacenar la raíz de cada componente para usos futuros.

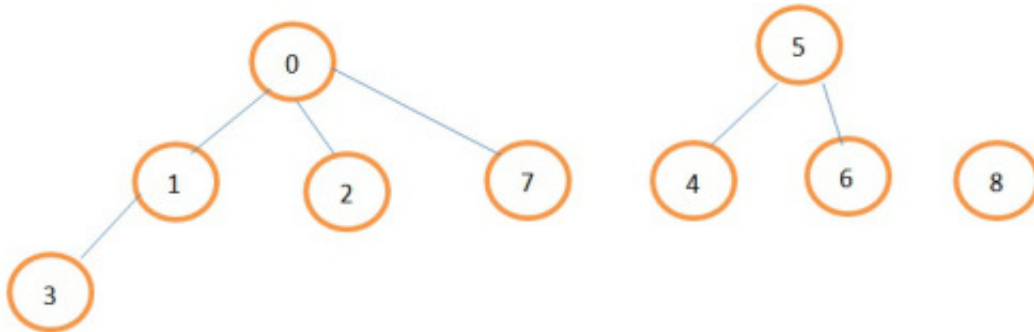
Código:

```

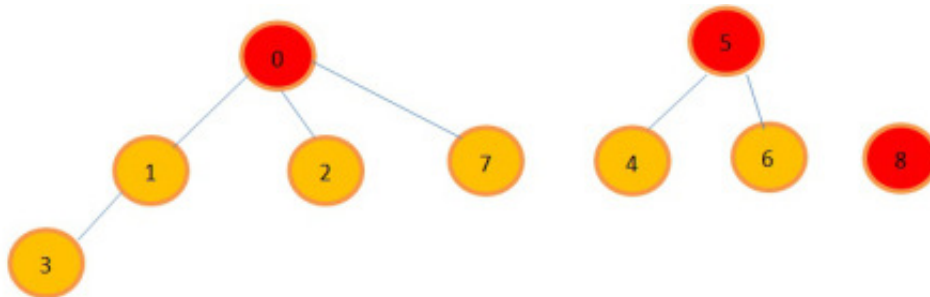
1  int root[ MAX ]; //tendra las raices de las componentes conexas luego de a
2  int numComponentes; //variable para el numero total de componentes conexas
3  //Método para obtener el numero de componentes conexas luego de realizar ]
4  int getNumberConnectedComponents( int n ){
5      numComponentes = 0;
6      for( int i = 0 ; i < n ; ++i ){
7          if( padre[ i ] == i ){ //Si el padre del vértice i es el mismo
8              //if( Find( i ) == i ){ //podemos usamos find para el mismo prop
9                  //para que se realice compresion de cami
10                 root[ numComponentes++ ] = i; //almaceno la raiz de cada nuev
11                 // numComponentes++;
12             }
13         }
14     return numComponentes;
15 }
```

Cantidad de Vértices por Componente Conexa

Para saber la cantidad de Vértices que posee cada componente conexa basta con tener un contador en la raíz de su componente conexa, de tal manera que al recorrer cada vértice incrementamos el contador de la raíz de su componente conexa. Veamos el siguiente ejemplo:



Comencemos haciendo Find(x) desde vértice 0 hasta vértice 8:



| Vértices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|---|
| Find(x) | 0 | 0 | 0 | 0 | 5 | 5 | 5 | 0 | 8 |

Incrementamos en un arreglo el número de vértices, ello lo realizamos solo sobre los vértices que son raíces:

```
1  for( int i = 0 ; i < n ; ++i ){
2      numVertices[ Find( i ) ]++;           //incremento la raíz del
3  }
```

| Raíz | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-------------|---|---|---|---|---|---|---|---|---|
| NumVertices | 5 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 1 |

Código:

```
1  int numVertices[ MAX ]; //almacenara la cantidad de vértices para la i-e
2  //Método para obtener la raíz y el numero de vértices de cada componente c
3  //será necesario primero tener la cantidad de componentes conexas
4  //podemos llamar 1ero al metodo getNumberConnectedComponents o incluir por
5  void getNumberNodes( int n ){
6      memset( numVertices , 0 , sizeof( numVertices ) ); //inicializo mi
7      for( int i = 0 ; i < n ; ++i ){
8          numVertices[ Find( i ) ]++; //incremento la raíz
9      }
```

```
10     for( int i = 0 ; i < numComponentes ; ++i ){
11         printf("Componente %d: Raiz = %d , Nro nodos = %d.\n" , i + 1 , rc
12     }
13 }
```

Problemas de diferentes Jueces

UVA

[459 – Graph Connectivity](#)

[599 – The Forrest for the Trees](#)

[793 – Network Connections](#)

[10158 – War](#)

[10178 – Counting Faces](#)

[10685 – Nature](#)

[11503 – Virtual Friends](#)

[11987 – Almost Union-Find](#)

TIMUS

[1671 – Anansi's Cobweb](#)

TJU

[3499 – Network](#)

POJ

[1703 – Find them, Catch them](#)

[2236 – Wireless Network](#)

Códigos:

Implementación de la estructura en C++: [Union-Find](#)

Implementación de la estructura en JAVA: [Union-Find](#)

Por Jhosimar George Arias Figueroa

SHARE THIS:

Twitter



Facebook



Me gusta

A un bloguero le gusta esto.

Esta entrada fue publicada en [Data Structure](#), [Main](#) y etiquetada [data structure](#), [disjoint set](#), [union find](#). Guarda el [enlace permanente](#).

8 RESPUESTAS A “DISJOINT-SET: UNION FIND”

Señor X | [septiembre 11, 2012 en 6:39 pm](#) | [Responder](#)

Hola, primero felicitarte por tu blog, está genial son super ilustrativos, gracias a tu tutorial y a tus graficos super explicativos aprendí Union-Find a la primera leída, no se si podrías hacer un tutorial de Segment Tree.



jariasf | [septiembre 11, 2012 en 9:16 pm](#) | [Responder](#)

Holas que bueno que fue de ayuda! ese es el objetivo de los tutoriales... Por ahora estoy un poco ocupado por lo que no eh podido actualizar el blog, pero en cualquier momento trataré de publicar el tutorial de segment tree y otros más.



Z | [marzo 1, 2013 en 10:04 pm](#) | [Responder](#)

Muchas gracias por los tutoriales, fáciles de comprender, muy buenos.



david | [junio 6, 2013 en 3:14 pm](#) | [Responder](#)

al final no entiendo como debe ser la estructura del union find `struct union_find { ? }`



pd: muy buen blog, te felicito, y espero que sigas adelante con esto.

hugo | [julio 10, 2013 en 2:55 pm](#) | [Responder](#)

igenial loco!, muy claro. estoy tratando de generar laberintos, empecé a buscar un poco hasta que me decidí por un arbol de expansion minimo mediante kruskal. esto me viene justo... gracias por subirlo



Hailo | [octubre 5, 2013 en 11:49 am](#) | [Responder](#)

Excelente tutorial, sigue subiendo otros por favor



Alberto | [septiembre 20, 2014 en 11:57 pm](#) | [Responder](#)



Excelente!!!

Amigo, sería genial si te haces un tutorial de Segment Tree.

Saludos



Norma | [mayo 14, 2015 en 2:30 pm](#) | [Responder](#)

Hola, mi compañera y yo te queremos agradecer por esta publicación ya que nos ha ayudado bastante, gracias en serio



Blog de WordPress.com. El tema Coraline.