# Estructuras de Datos

MSc. (c) Jhosimar George Arias Figueroa

jariasf03@gmail.com

State University of Campinas
Institute of Computing

# Contenido

# Introducción

- Una estructura de datos consiste en un medio de **almacenar, organizar y recuperar informaciones**.
- Diferentes estructuras de datos poseen complejidades distintas para operaciones como **búsqueda, inserción, eliminación y actualización**.
- Una estructura no resuelve un problema de programación por si solo, pero la elección de una estructura de datos adecuada puede ser la diferencia entre pasar o no la **restricción de tiempo de ejecución**.

# Estructuras de Datos y bibliotecas

- Se asume que el lector de este material tenga familiaridad con estructuras de datos elementares vistas en un curso de pregrado.
- Serán destacadas implementaciones de estructuras de datos usando la **biblioteca STL** (Standard Template Library) de C++.
- Para visualizar el comportamiento de estas estructuras, consulta el siguiente link:

http://visualgo.net/

# Contenido

# Estructuras de Datos lineales

Una estructura de datos es clasificada como lineal si sus elementos forman una secuencia. Tanto en C++ como en Java podemos hacer uso de bibliotecas ya implementadas de las estructuras de datos lineales:

- Vectores estáticos – soporte nativo en C/C++ y Java.
- Vectores dinámicos – C++ STL `vector` (Java `ArrayList`).
- Vectores booleanos – C++ STL `bitset` (Java `BitSet`)
- Listas enlazadas – C++ STL `list` (Java `LinkedList`)
- Pilas – C++ STL `stack` (Java `Stack`)
- Colas – C++ STL `queue` (Java `Queue`)
- Dicolas – C++ STL `deque` (Java `Deque`)

# Vectores dinámicos

- En C++, la librería STL nos brinda una implementación de vector.
- Para usarla es necesario incluir la libreria vector:

```
#include<vector>
```

- Entre las funciones principales en un vector tenemos los siguientes:

```
v.push_back(x); //Inserta un elemento al final del vector O(1)
v.pop_back(x);  //Elimina el ultimo elemento del vector   O(1)
v.clear();      //Elimina todos los elementos del vector  O(n)
v.size();       //Retorna el numero de elementos          O(1)
v.resize(n);    //Cambia el tamano a n elementos          O(n)
```

- Para el acceso del vector podemos hacerlo de manera similar a un arreglo estático (v[ i ]) y también podemos hacerlo mediante iteradores.

# Vectores dinámicos

- En Java el framework Collections nos brinda implementaciones de arreglos dinámicos.
- Para usarla es necesario importar la clase:

```
java.util.ArrayList;
```

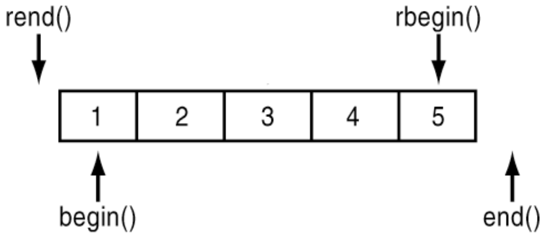
- Entre las funciones principales de la implementación ArrayList tenemos:

```
v.add(x);       //Inserta un elemento al final de la lista  O(1)
v.contains(x);  //Verificar si elemento existe en la lista  O(n)
v.clear();      //Elimina todos los elementos de la lista   O(n)
v.size();       //Retorna el numero de elementos            O(1)
v.remove(i);    //Remueve el elemento en el indice i        O(n)
```

- Para el acceso usamos el método v.get(i) para acceder al i-th elemento.

# Iteradores en C++

- Podemos pensar en un iterador como un puntero a un elemento en el contenedor.
- Todos los contenedores soportan dos funciones principales:
  - Función **begin**, el cual retorna un puntero iterador al inicio del contenedor (1er elemento).
  - Función **end**, el cual retorna un iterador el cual me indica que se ha llegado al final del contenedor.



# Iteradores en C++

La sintaxis para declarar un iterador es la siguiente:

```
std::class_name< template_parameters > :: iterator name;
std::class_name< template_parameters > :: reverse_iterator name;
```

Ejemplo de uso de iteradores:

```
vector< int > :: iterator it;    //Creamos iterador para un vector
vector< int > v;     //Crea un vector vacio

//Agregamos algunos elementos
v.push_back( 1 );
v.push_back( 2 );
v.push_back( 3 );

//Iteramos con for
printf("Elementos de vector\n");
for( int i = 0 ; i < v.size() ; ++i )
    printf("%d " , v[ i ] );
printf("\n");

//Iteramos con iterador
for( it = v.begin() ; it != v.end() ; ++it )
    printf("%d " , *it );
printf("\n");
```

# Iteradores en Java

De manera similar a C++ un iterador en Java me permitirá recorrer una colección:

```
List<Integer> v = new ArrayList<Integer>();         //Crea una lista vacia

//Agregamos elementos a la lista
v.add( 1 );
v.add( 2 );
v.add( 3 );

//Iteramos con for
for( int i = 0 ; i < v.size() ; ++i )
    System.out.println( v.get( i ) );
System.out.println();

    //Creamos un iterador para una lista
Iterator< Integer > it = v.iterator();
//Iteramos con iterador
while( it.hasNext() ){
    System.out.print( it.next() + " " );
}
System.out.println();
```

# Pilas

En C++, la libreria STL nos brinda una implementación de pila.

Para usarla es necesario incluir la libreria:

```
#include<stack>
```

Ejemplo de uso:

```
int main(){
    //stack< tipo > nombre
    stack< int > S;     //Creamos una pila vacia

    //insertamos tres elementos
    S.push( 1 ); S.push( 2 ); S.push( 3 );

    //(LIFO): 3, 2, 1
    while( !S.empty() ) {
        printf("%d " , S.top() );
        S.pop();
    }
    printf("\n");

    return 0;
}
```

# Pilas

En Java, el framework Collections nos brinda una implementación de pila.

Para usarla es necesario importar la clase:

```
java.util.Stack;
```

Ejemplo de uso:

```java
public static void main(String[] args) {
    //Stack< Clase > nombre = new Stack< Clase >();
    //Creamos una pila vacia
    Stack< Integer > S = new Stack< Integer >();

    // insertamos tres elementos
    S.push( 1 ); S.push( 2 ); S.push( 3 );

    //(LIFO): 3, 2, 1
    while( !S.empty() ) {
        System.out.printf("%d " , S.pop() );
    }
    System.out.printf("\n");
}
```

# Colas

En C++, la libreria STL nos brinda una implementación de cola.

Para usarla es necesario incluir la libreria:

```
#include<queue>
```

Ejemplo de uso:

```cpp
int main(){
    //queue< tipo > nombre
    queue< int > Q;    //Creamos una cola vacia

    //insertamos tres elementos
    Q.push( 1 ); Q.push( 2 ); Q.push( 3 );

    //(FIFO): 1, 2, 3
    while( !Q.empty() ) {
        printf("%d " , Q.front() );
        Q.pop();
    }
    printf("\n");

    return 0;
}
```

# Colas

En Java, el framework Collections nos brinda una implementación de pila.

Para usarla es necesario importar la clase:

`java.util.Queue;`

Ejemplo de uso:

```java
public static void main(String[] args) {
    //Queue< Clase > nombre = new LinkedList< Clase >();
    Queue< Integer > Q = new LinkedList<Integer>();

    //insertamos tres elementos
    Q.add( 1 ); Q.add( 2 ); Q.add( 3 );

    //(FIFO): 1, 2, 3
    while( !Q.isEmpty() ) {
        System.out.printf("%d " , Q.remove() );
    }
    System.out.printf("\n");
}
```

# Contenido

# Árboles de búsqueda binaria balanceadas

- Un árbol de búsqueda binario posee la siguiente propiedad: para cada subárbol partiendo de un nodo raíz x, los nodos a la izquierda de x son menores que x, mientras que los nodos a la derecha son mayores que x.


- Un árbol de búsqueda binario es considerado balanceado cuando su altura es asintóticamente limitada por una función logaritmica del número de nodos $h = O(\log n)$
- Cuando el árbol es balanceado, las operaciones de búsqueda, inserción, máximo, mínimo, sucesor, predecesor y eliminación pasan a tener una complejidad $O(\log n)$.

# Árboles de búsqueda binaria balanceadas

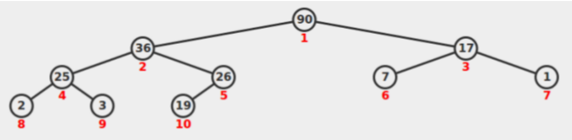
- Las clases `map` y `set` de C++ STL (`TreeMap`, `TreeSet` en Java) son implementaciones de árboles rojo-negras, que corresponden a un tipo de árboles de búsqueda binarias balanceadas.
- La diferencia entre las clases `map` y `set` es que la primera almacena pares clave y valor, mientras la segunda almacena solamente claves.

# Árboles de búsqueda binaria balanceadas

```
set<int> used_values;
map<string, int> mapper;
//Ingreso estatico
mapper["john"] = 78;   used_values.insert(78);
mapper["billy"] = 69;  used_values.insert(69);
mapper["andy"] = 80;   used_values.insert(80);
mapper["steven"] = 77; used_values.insert(77);
mapper["felix"] = 82;  used_values.insert(82);
mapper["grace"] = 75;  used_values.insert(75);
mapper["martin"] = 81; used_values.insert(81);

//                        (grace,75)
//            (billy,69)                (martin,81)
//      (andy,80)   (felix,82)    (john,78)  (steven,77)
for(map<string, int>::iterator it = mapper.begin(); it != mapper.end(); ++it)
    printf("%s %d\n", ((string)it->first).c_str(), it->second);

printf("Score de steven es %d, score de grace es %d\n", mapper["steven"], mapper["grace"]);

//                  (78)
//          (75)            (81)
//      (69)    (77)    (80)    (82)

// busqueda O(log n), encontrado
printf("%d\n", *used_values.find(77));

// busqueda O(log n), no encontrado
if (used_values.find(79) == used_values.end())
    printf("79 no encontrado\n");
```

# Cola de prioridad

- Un **max-heap** es un árbol de búsqueda binaria completa, tal que cada nodo x posee una propiedad de heap, que consiste en la restricción de que todos los hijos del nodo x poseen valores menores que x. Esto implica que la raiz será siempre el mayor elemento del heap.
- Un heap puede ser representado por un vector. En ese caso, los elementos del árbol son visitados de arriba hacia abajo y de izquierda a derecha para ser almacenados secuencialmente en el vector.



# Cola de prioridad

- Un heap es una estructura de datos muy útil para representar una cola de prioridad, donde el elemento de mayor prioridad puede ser removido y un nuevo elemento puede ser insertado en tiempo O(log(n)).
- Son utilizados en problemas importantes de grafos como árboles de expansión mínima (Prim), camino más corto (Dijkstra) y árbol $A^{\star}$.
- Una implementación de cola de prioridades puede ser encontrada en la clase C++ STL `priority_queue` y Java Collections `PriorityQueue`.

# Cola de prioridad

```
priority_queue< pair<int, string> > pq;    // introduciendo 'pair'
pair<int, string> result;

pq.push(make_pair(100, "john"));
pq.push(make_pair(10, "billy"));
pq.push(make_pair(20, "andy"));
pq.push(make_pair(100, "steven"));
pq.push(make_pair(70, "felix"));
pq.push(make_pair(2000, "grace"));
pq.push(make_pair(70, "martin"));

// la clave primaria es dinero (entero), clave secundaria nombre (string)!
//                          (2000,grace)
//             (100,steven)                 (70,martin)
//      (100,john)    (10,billy)      (20,andy)   (70,felix)

//Top 3 personas con mas dinero
result = pq.top();                   // O(1) para acceder al top/max element
pq.pop();           // O(log n) para eliminar el tope y rebalancear la estructura

printf("%s tiene %d $\n", ((string)result.second).c_str(), result.first);
result = pq.top(); pq.pop();
printf("%s tiene %d $\n", ((string)result.second).c_str(), result.first);
result = pq.top(); pq.pop();
printf("%s tiene %d $\n", ((string)result.second).c_str(), result.first);
```

# Contenido

# Ordenamiento y Búsqueda

Dos operaciones muy importantes sobre vectores son la ordenación y búsqueda. Estas operaciones ya estan implementadas en APIs de C++ y Java. Dentro de los algorimos de ordenamiento conocidos tenemos:

- Algoritmos **O($n^2$)** basados en comparación: Bubblesort, Selection Sort, Insertion Sort, etc. Normalmente **deben ser evitados** en competición por ser lentos, pero comprenderlos puede ayudar en la solución de ciertos problemas.
- Algoritmos **O(n log n)** basados en comparación: Mergesort, Heapsort, Quicksort, etc. Este tipo de algoritmos puede ser usados por medio de métodos como `sort`, `stable_sort` de la clase `algorithm` de C++ (`Collections.sort` en Java).
- Algoritmos de propósito específico **O(n)**: Counting sort, Radix sort, Bucket sort, etc. Estos algoritmos asumen características especificas sobre los valores a ser ordenados para reducir la complejidad del algoritmo.

# Ordenamiento y Búsqueda

A continuación son presentados tres métodos para realizar la búsqueda en un vector:

- Búsqueda lineal **O(n)**: recorrer por todos los elementos del vector. Este método **debe ser evitado.**
- Búsqueda binaria **O(log n)**: Esta búsqueda está implementada en C++ a partir de los métodos `lower_bound`, `upper_bound`, `binary_search` de la clase `algorithm` (`Collections.binarySearch` en Java).
- Hashing **O(1)**: Cuando una buena función de hash es seleccionada, las probabilidades de colisión se reducen y el método se vuelve muy rápido. En C++11 se tiene `unordered_map`(`HashMap` en Java).

# Contenido

# Introducción

- Dado un conjunto $\{1, 2, \ldots., n\}$ de n elementos.
- Inicialmente cada elemento es un conjunto diferente.

$$\{1\}, \{2\}, \{3\}, ..., \{n\}$$

- Podemos unir dos conjuntos.

$$\{1, 2\}, \{3\}, \{4, 5, 6\}, ..., \{n\}$$

- Podemos preguntar en que conjunto se encuentra cada elemento.

# Union-Find

Union Find es una estructura de datos que modela una colección de conjuntos disjuntos y esta basado en 2 operaciones:

- **Find(A)**: Determina el conjunto al que pertenece A.
- **Union(A,B)**: Une todo el conjunto al que pertenece A con todo el conjunto al que pertenece B.

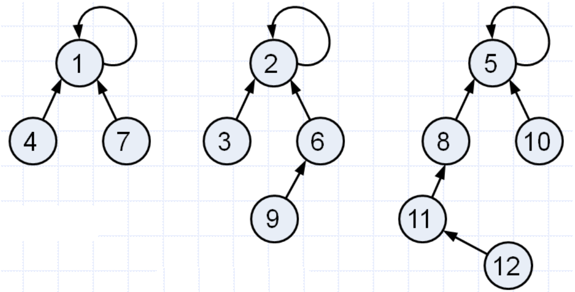
Adicionales a estas operaciones tenemos un método de inicialización al cual llamaremos **MakeSet**.

# Conjuntos como Árboles

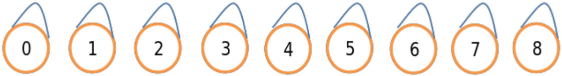
Podemos modelar los conjuntos como árboles donde:

- Cada elemento es un **nodo en el árbol** , el cual contiene un puntero al **nodo padre**.
- La **raíz del árbol es el elemento representativo** de cada conjunto y su puntero al padre es sobre si mismo.

Ejemplo: {1,4,7} {2,3,6,9} {5,8,10,11,12}

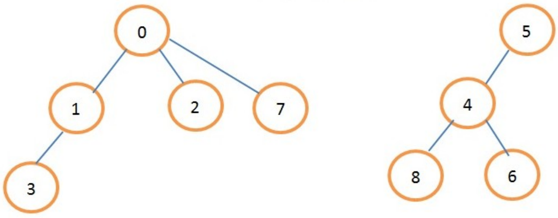


# Make Set



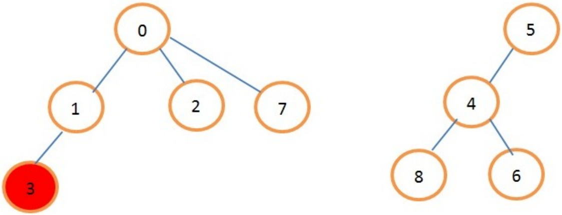

| **Nodos** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| **Padre** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Find



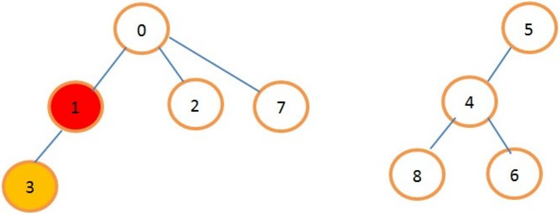

| Nodos | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 0 | 0 | 1 | 5 | 5 | 4 | 0 | 4 |

# Find(3)



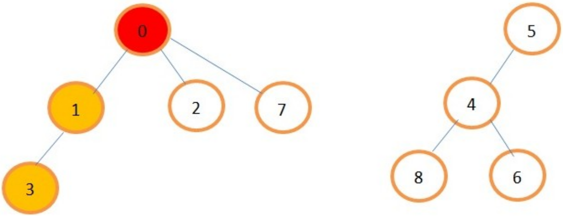

| Nodos | 0 | 1 | 2 | **3** | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| **Padre** | 0 | 0 | 0 | **1** | 5 | 5 | 4 | 0 | 4 |

# Find(3)



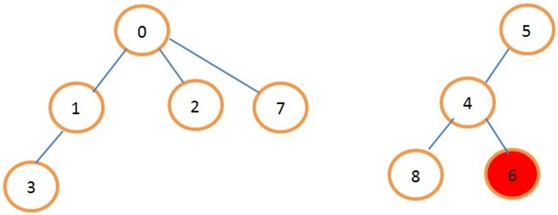

| **Nodos** | 0 | **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| **Padre** | 0 | **0** | 0 | 1 | 5 | 5 | 4 | 0 | 4 |

# Find(3)



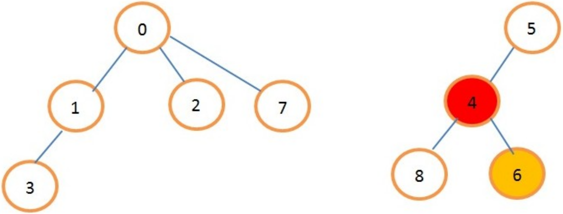

| Nodos | **0** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| **Padre** | **0** | 0 | 0 | 1 | 5 | 5 | 4 | 0 | 4 |

# Find(6)



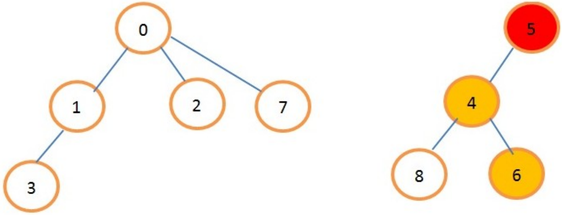

| Nodos | 0 | 1 | 2 | 3 | 4 | 5 | **6** | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| **Padre** | 0 | 0 | 0 | 1 | 5 | 5 | **4** | 0 | 4 |

# Find(6)




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

# Find(6)




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

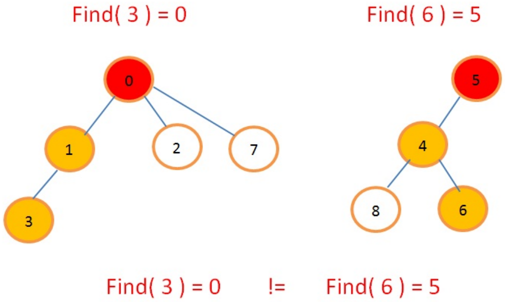
# Implementación Find

Creamos un método llamado Find que devolverá la raíz, el elemento representativo, del árbol.

```
//Metodo para encontrar la raiz del vertice actual X
int Find( int x ){
    if( x == padre[ x ] ){          //Si estoy en la raiz
        return x;                   //Retorno la raiz
    }
    else return Find( padre[ x ] ); //Buscar el padre del vertice actual
}
```

# Aplicación del uso de Find

Podemos crear un método llamado **SameComponent** que me indicará si dos vértices en un grafo están en la misma componente conexa:

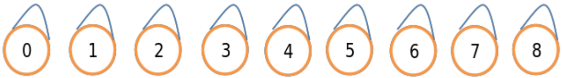


# Union

Este método me permite unir 2 componentes conexas ó conjuntos, ello se realiza de la siguiente manera:

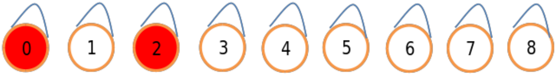
- Obtenemos la raíz del vértice X.
- Obtenemos la raíz del vértice Y.
- Actualizamos el padre de alguna de las raíces, asignandole como padre la otra raíz.

# MakeSet(9)




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

# Union(2,0)



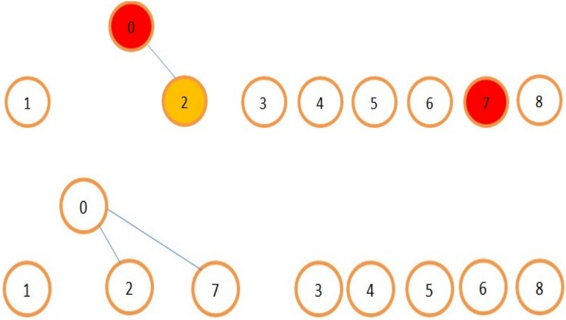

| Nodos | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Union(2,0)


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

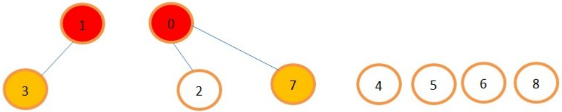
# Union(7,2)

Find(7) = 7
Find(2) = 0

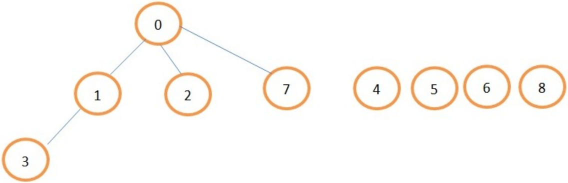



| Nodos | 0 | 1 | 2 | 3 | 4 | 5 | 6 | **7** | 8 |
|---|---|---|---|---|---|---|---|---|---|
| **Padre** | 0 | 1 | 0 | 3 | 4 | 5 | 6 | **0** | 8 |

# Union(3,1) ->Union(3,7)




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

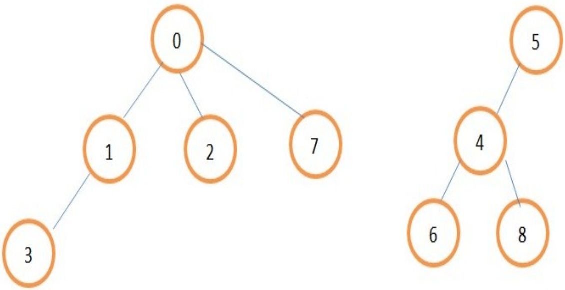



| Nodos | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Padre | 0 | 0 | 0 | 1 | 4 | 5 | 6 | 0 | 8 |

# Union Ejercicio

Cual sería el resultado después de aplicar Union(6,4), Union(8,4) y Union(4,5)?

# Union Ejercicio




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

# Implementación Union

Creamos un método llamado Union que unirá 2 conjuntos en 1.

```
//Metodo para unir 2 componentes conexas
void Union( int x , int y ){
    int xRoot = Find( x );     //Raiz de la componente del vertice X
    int yRoot = Find( y );     //Raiz de la componente del vertice Y
    padre[ xRoot ] = yRoot;    //Mezclar ambos arboles o conjuntos
}
```

# Mejoras

Existen mejoras tanto para el método Find como para el método Union llamados:

- **Find:** Compresión de Caminos.
- **Union:** Unión por Rangos.

Estas mejoras se dan debido a que:

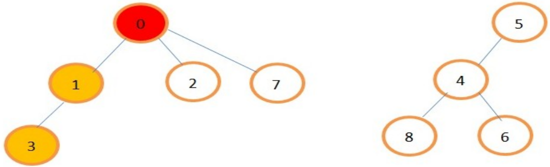
- **Find** recorre el árbol hasta la raíz en cada llamada O(h), peor caso O(n).
- **Union** tiende a crear un árbol desbalanceado.

# Compresión de caminos

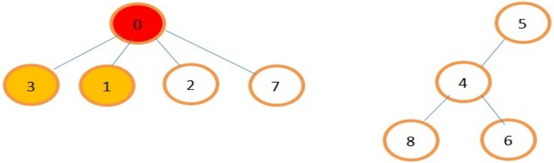
La idea de esta mejora es que cada nodo que visitemos en el camino al nodo raíz puede ser conectado directamente hacia la raíz, es decir, **al terminar de usar el método Find todos los nodos que se visiten tendrán como padre la raíz directamente.**

La compresión de caminos mejora las búsquedas posteriores.

# Compresión de caminos




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




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

# Implementación Compresión de caminos

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

```
//Metodo para encontrar la raiz del vertice actual X
int Find( int x ){
    if( x == padre[ x ] ){          //Si estoy en la raiz
        return x;                   //Retorno la raiz
    }
    //else return Find( padre[ x ] ); //Buscar el padre del vertice actual
    else return padre[ x ] = Find( padre[ x ] ); //Compresion de caminos
}
```

# Union por Rango

Para cada árbol mantenemos un rango: una cota superior a la altura del árbol.

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.

En el peor de los casos (los dos árboles con igual rango) el rango del árbol resultante se incrementa en 1.

# Complejidad con las mejoras

Find aún toma tiempo lineal. Este no es un buen análisis por que si llamamos Find de nuevo, su tiempo de ejecución será constante. Es necesario un **análisis amortizado.**

Si ejecutamos k operaciones de Union y Find, en algún orden desconocido, con parámetros desconocidos. Se puede demostrar que con compresión de caminos, la complejidad será de $O(k * log(n))$.

Union por Rango evitará que algún árbol tenga una profundidad mayor a log(n). Por lo tanto, sin considerar compresión de caminos, la complejidad de ejecutar k operaciones de union será $O(k * log(n))$.

Si combinamos los dos, la complejidad es $O((k + n)\alpha(n))$. Para cualquier aplicación práctica $\alpha(n) \leq 4$.

# Operaciones Adicionales

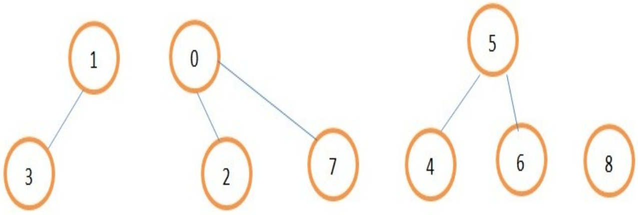
Una vez comprendido Union Find podemos crear métodos adicionales que nos ayuden en diferentes problemas por ejemplo:

- Hallar el número de componentes conexas.
- Hallar el número de vértices por componente conexa.
- Hallar los vértices pertenecientes a cada componente conexa.

A continuación veremos el primero, los demás se dejan como ejercicio.

# Número de componentes conexas

Verificamos si padre[ x ] = x.




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

# Aplicaciones

- Árbol de Expansión Mínima - Algoritmo de Kruskal.
- Encontrar Componentes Conexas en un Grafo.
- Encontrar el Menor Ancestro Común de dos nodos en un Árbol – Algoritmo de Tarjan LCA

# Enlaces Útiles

- C++ STL: http://www.cplusplus.com/reference/stl/
- JAVA Collections: http://docs.oracle.com/javase/tutorial/collections/index.html
- Union Find: https://jariasf.wordpress.com/2012/04/02/disjoint-set-union-find/
- Segment Tree: https://prodeportiva.wordpress.com/2013/02/08/segment-trees/