# Heap

# ¿Qué es un heap?

El heap es una estructura de datos basada en árboles binarios. El autor Mark Allen Weiss (2013) señala dos propiedades esenciales. La primera, la **denomina propiedad estructural** y se refiere a que el heap es un árbol binario completo con excepción del nivel inferior, es decir, no es necesario que posea todas las hojas por esta razón, se le suele decir árbol casi completo. En el caso de agregar nuevos nodos la inserción se realiza de arriba hacia abajo y de izquierda a derecha en la posición correspondiente. Un árbol completo garantiza una altura o profundidad que se calculará como:

ℎ = 𝑙𝑜𝑔 (𝑛)

2

Donde h es la altura y n es la cantidad de nodos que tiene el heap. Más adelante se retomará esta ecuación para analizar los costos temporales de los métodos internos del heap.

*En la Figura 1 se propone un ejemplo de un árbol binario que no cumple con la propiedad estructural.*

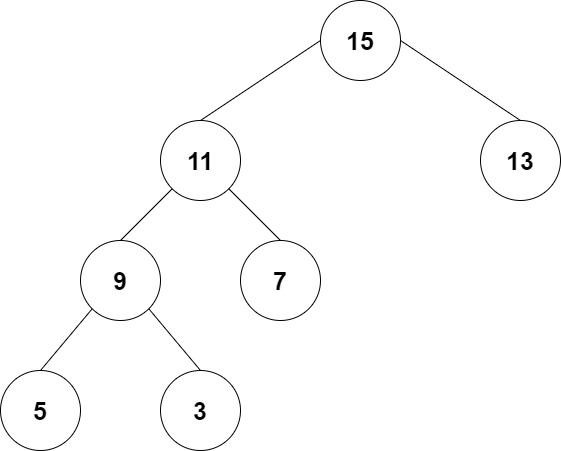


Figura 1

La segunda propiedad, el autor, la denomina **propiedad de ordenamiento**. En la que se describe la relación que existe entre un nodo padre y su o sus nodos hijos. Para un heap min, el valor del nodo padre siempre es menor o igual que el valor de sus hijos y para un heap max, el valor del nodo padre siempre es mayor o igual al valor de sus hijos. Esta propiedad se debe conservar en todos los subárboles para que el heap sea consistente. De esta forma, el valor mínimo o máximo se ubicará siempre en la raíz según el heap que se decida utilizar.

*En la Figura 2 se observa la diferencia entre en heap y un árbol binario que no cumple con la propiedad de ordenamiento del heap.*

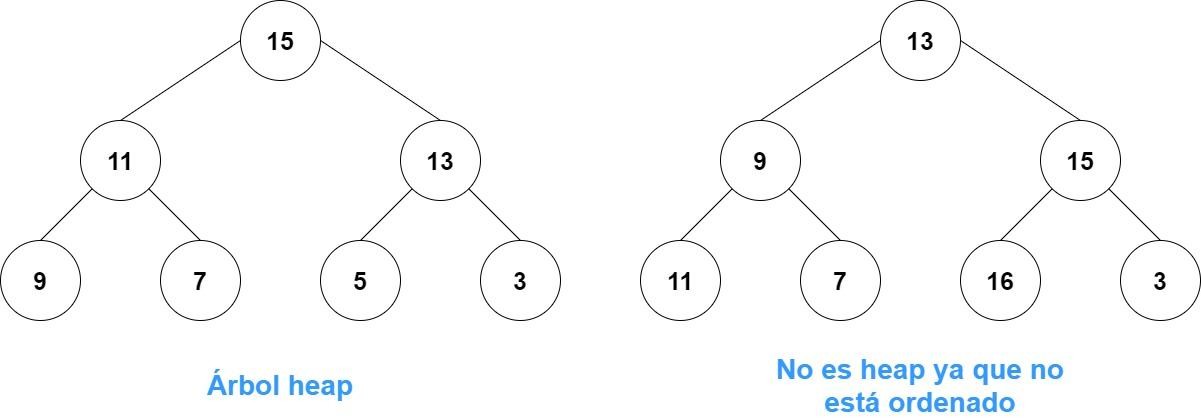


Figura 2

*En la Figura 3 se muestra un ejemplo de Max heap y de Min Heap, en el cual se observa el valor máximo o mínimo de la raíz según corresponda.*

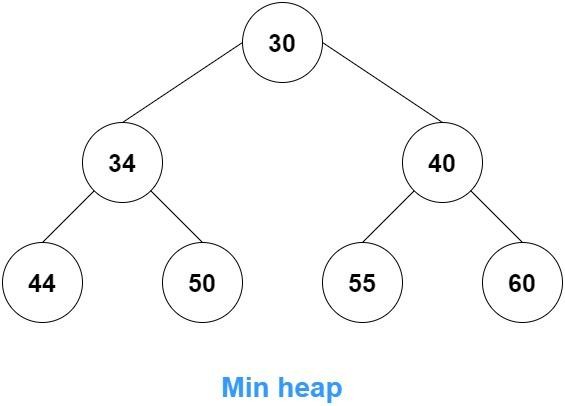
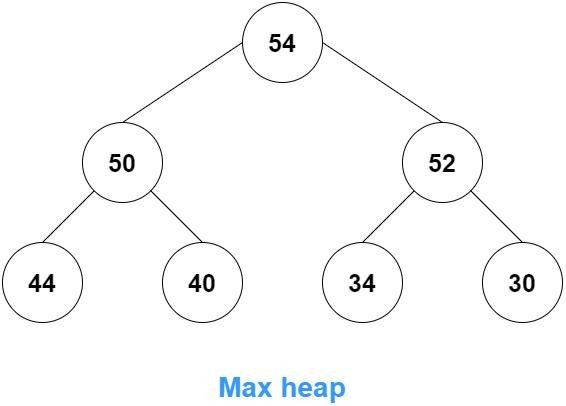


Figura 3

## **Conservación del heap**

Debido a las propiedades mencionadas anteriormente sabemos que el heap nos provee el valor mínimo o máximo simplemente consultando la raíz. Esta característica resulta de relevancia al momento de implementar una cola con prioridad, ya que en este tipo abstracto de datos se requiere acceder al valor de mayor prioridad. Sin embargo, cabe preguntarse qué sucede cuando se agregan nuevos elementos al árbol, o cuando se eliminan elementos porque al modificar el heap se puede incurrir en una violación de alguna o ambas propiedades. En consecuencia, al realizar alguna de estas operaciones, se debe realizar una verificación y adecuación del heap para conservar sus propiedades. En la sección de métodos, se describirá con mayor detalle cómo se realiza la adecuación luego de insertar o eliminar un elemento.

## **¿Cómo se implementa?**

Si bien un heap es un tipo de árbol binario y, a priori, uno supone que se implementa con nodos conectados; se suele implementar mediante un arreglo según Jay Wengrow (2017). Partiendo de la propiedad estructural (árbol completo) se puede establecer una asociación entre la posición de un elemento en el árbol con el índice de un arreglo. En consecuencia, un heap puede ser un tipo abstracto de datos que utiliza un arreglo para su implementación. Para ilustrar este punto, veamos la Figura 4. Se observa en el arreglo los valores de cada elemento, en una posición que sigue un patrón. La raíz se ubica en el índice 1 del arreglo, luego es posible acceder a los valores de los hijos partiendo del padre, o partiendo de un hijo conocer el valor del padre. El patrón está determinado de la siguiente manera: el hijo izquierdo de un padre (en la posición i) es el valor que se encuentra en la posición 2i en el arreglo. Del mismo modo, el hijo derecho del padre está en la posición 2i+1 del arreglo. Para encontrar el padre de cualquier nodo en el árbol, podemos simplemente usar la división entera.

Dado un nodo en la posición n del arreglo, el padre estará en la posición n/2.

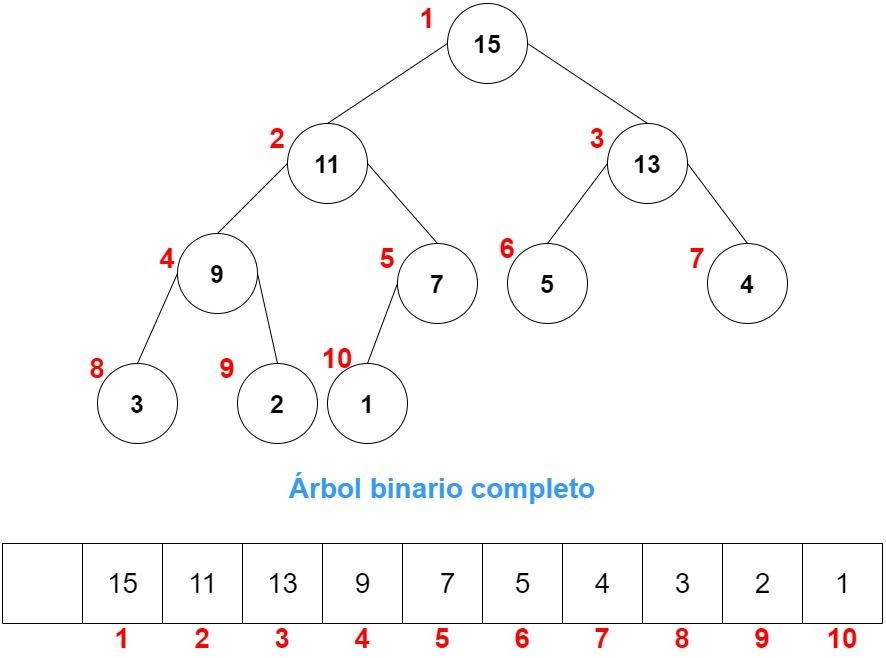


Figura 4

## **Implementación de heaps**

##### **Inserción de elementos en un heap:**

Para insertar un elemento a un heap min, contamos con un método denominado Push. El mismo consiste en agregar el elemento en un nuevo nodo al final del árbol.

Como este nuevo nodo puede romper la consistencia del heap, es decir, el orden, cada vez que se inserta un nuevo elemento se deberá recorrer el heap desde la posición del elemento agregado hacia la raíz, En caso de que el elemento agregado sea menor que el padre, se los debe intercambiar y se debe realizar la comparación en los diferentes niveles del árbol hasta llegar a la raíz. En nuestra implementación este método lo denominamos heapifyUp().

En la Figura 5 se ilustra el procedimiento paso a paso de inserción de un nuevo elemento “7” y su ordenamiento posterior hasta recuperar la consistencia. Este método posee un costo temporal estimado en Log2 (n), que recuerda a la ecuación mencionada para el cálculo de la altura del heap, ya que va a ser necesario (como máximo) realizar Log2 (n) intercambios entre hijo-padre para mantener las propiedades del heap.

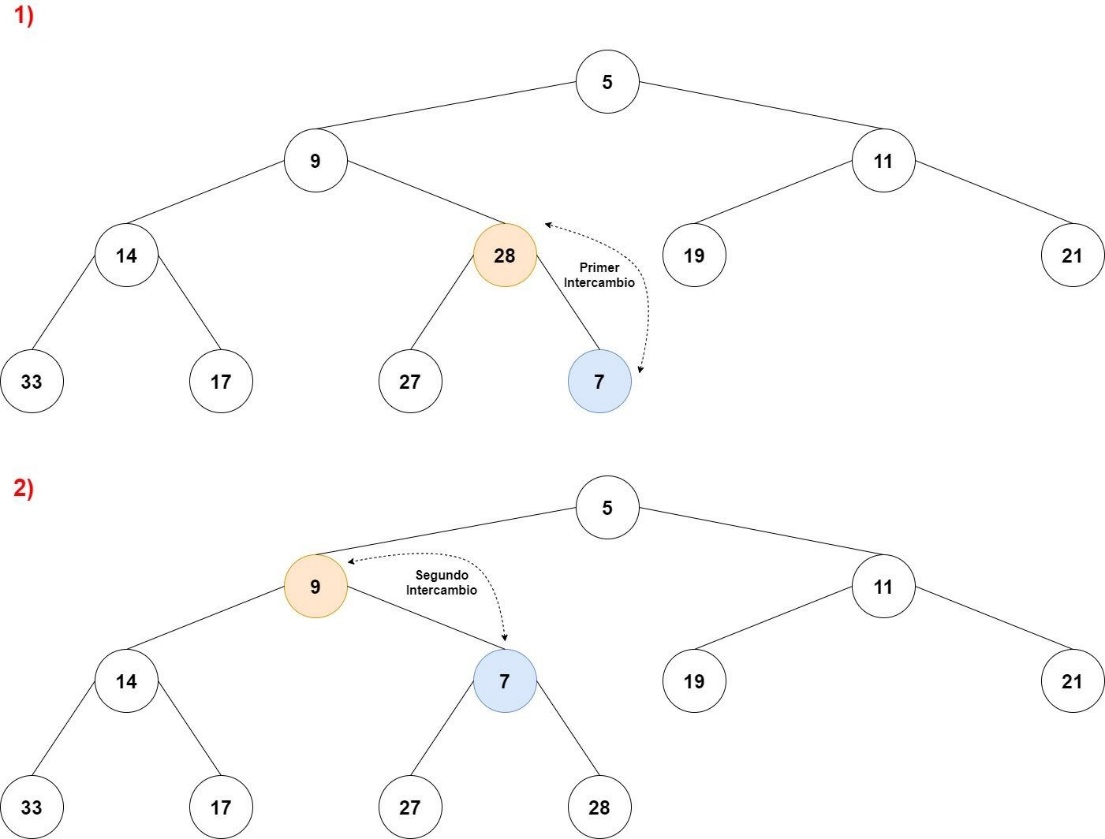
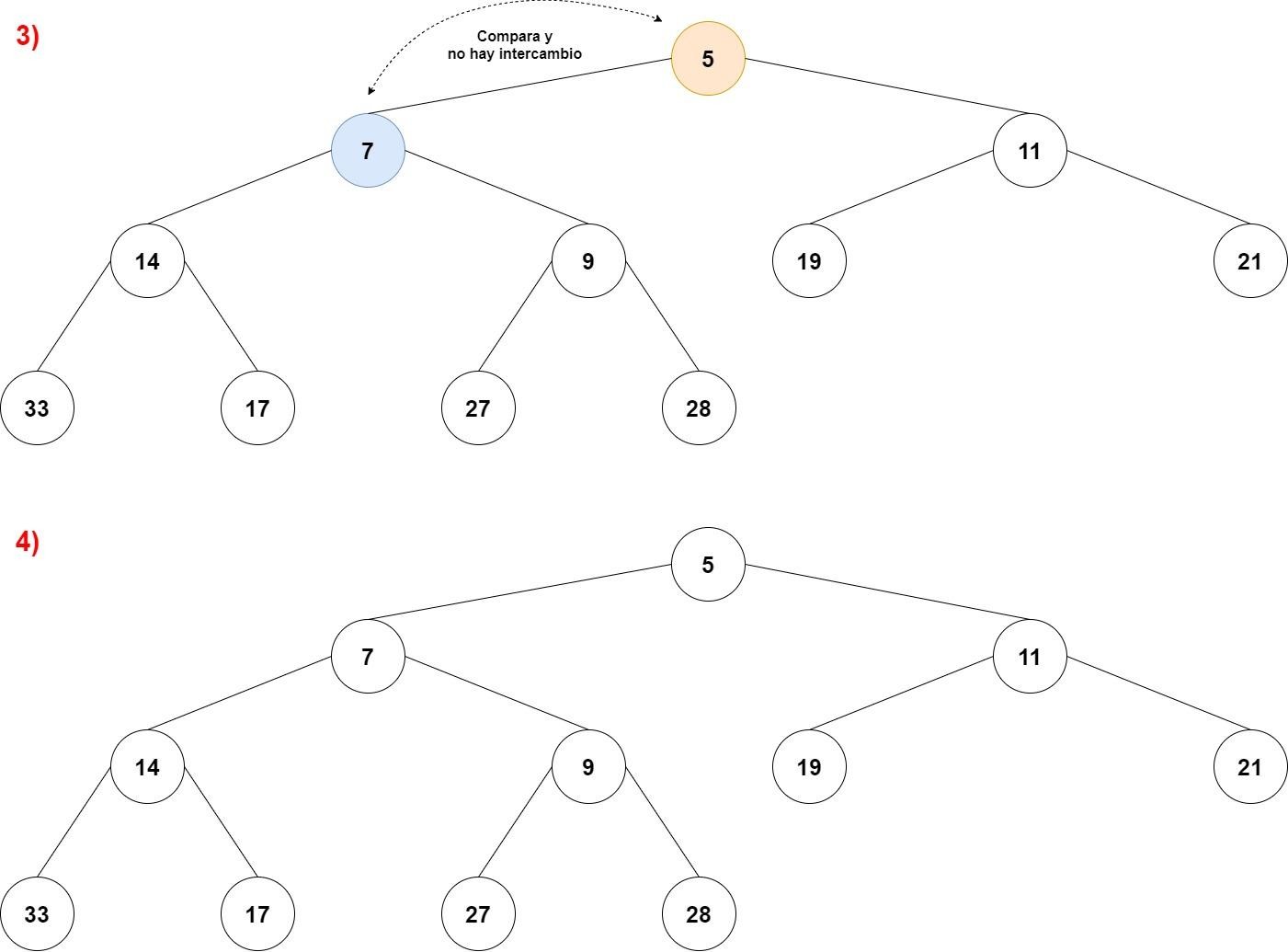


Figura 5



##### **Eliminación de elementos de un heap:**

En un heap, eliminamos el elemento que se encuentra en la raíz. Puede ser el máximo en un heap max o un mínimo en un heap min. En esta operación se elimina un elemento ubicado en la raíz y se lo reemplaza con el último elemento del árbol. Esta operación puede generar una inconsistencia en el heap, ya que puede romper el orden, como ocurre cuando se agrega un nuevo elemento. En este caso, como el problema se encuentra en la raíz, se debe comparar este elemento con sus hijos e intercambiarlo por el menor. Este recorrido se realiza desde arriba hacia abajo hasta que el heap recupera su consistencia. En nuestra implementación dentro del método Pop(), se encuentra el método heapifyDown() que es responsable de restablecer el orden del heap luego de la eliminación del valor que está raíz. Este método posee un costo temporal estimado en Log2 (n), como el método de inserción, ya que va a ser necesario

(como máximo) realizar Log2 (n) intercambios entre padre-hijo para mantener las propiedades del heap.

En la Figura 6 se observa como es el procedimiento que elimina el valor “5” y las sucesivas correcciones.

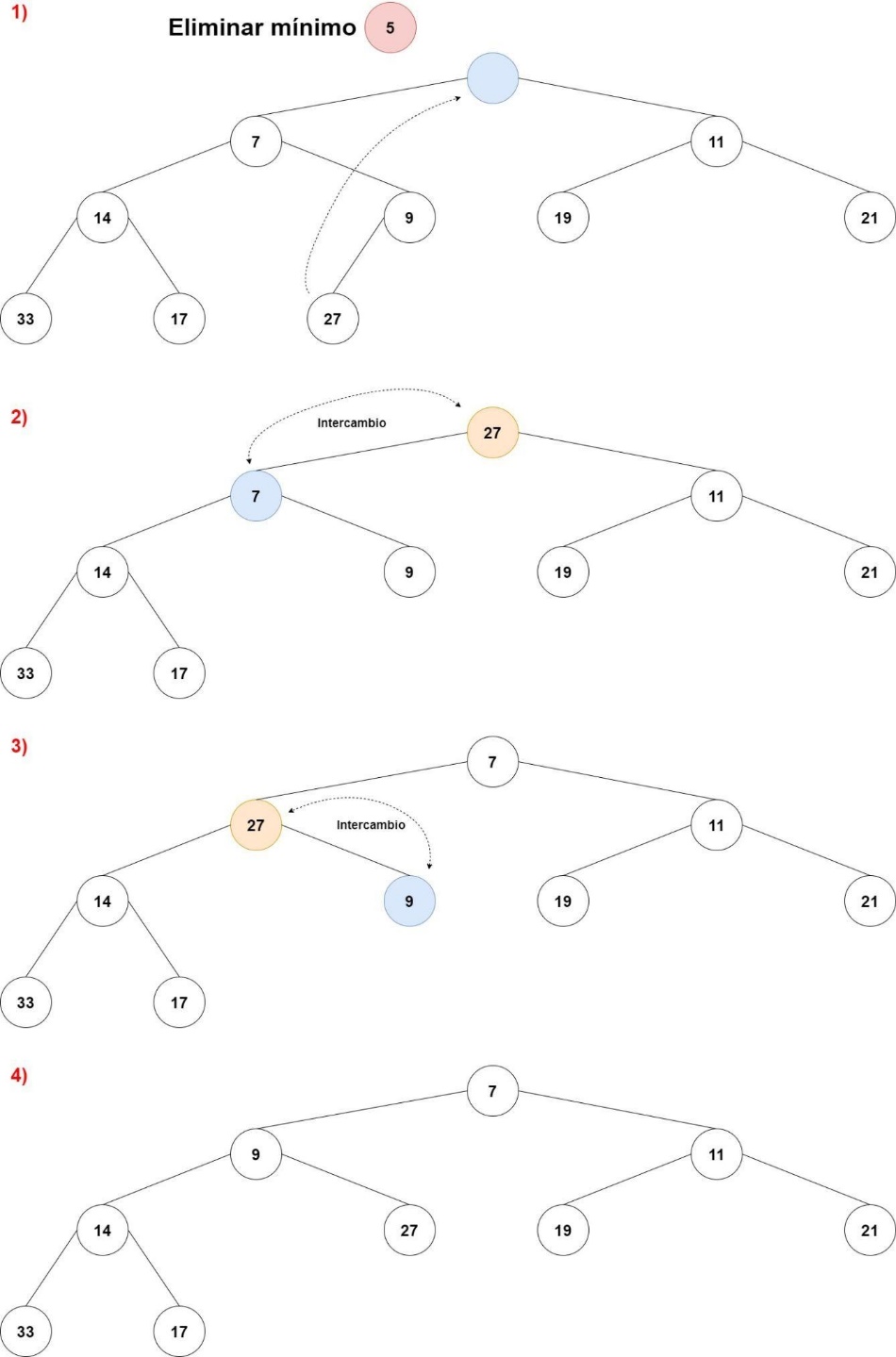


Figura 6

## **¿Para qué se utilizan los heaps?**

Habiendo explicado el funcionamiento y las características del heap se puede comprender porque son útiles para la implementación de las colas con prioridad. La necesidad de acceso inmediato al valor prioritario aprovecha la naturaleza del heap ya que siempre este valor estará en la raíz del heap. Además, una vez que se haya cumplido con tal tarea y se decida acceder a la siguiente (es decir, se elimine dicha prioridad del heap), la siguiente tarea subirá hasta colocarse en la raíz. Como se indicó en la parte de costos temporales del heap, tanto la inserción como la eliminación de un elemento tienen un costo estimado *logarítmico* en función de la cantidad de elementos del heap. En la comparación propuesta por Wengrow entre los costos de implementar una cola con prioridad con un heap o un arreglo ordenado se observa los siguiente:

|  |  |  |
| --- | --- | --- |
| Operación | **Arreglo ordenado** | **Heap** |
| Insertar | O(n) | O(Log(n)) |
| Eliminar | O(1) | O(Log(n)) |

Si bien puede parecer que comparando en ambas operaciones se compensan el en arreglo ordenado, es más conveniente utilizar una estructura de datos que siempre mantenga su costo como es el heap, en comparación con el arreglo ordenado que para una operación es rápido y lento para la otra. No obstante, queda a discreción del programador si encuentra razones válidas para utilizar el arreglo ordenado.

# Cola con Prioridad para Kruskal:

Para el algoritmo de Kruskal utilizaremos una cola con prioridad para ordenar los pesos de las aristas con sus nodos respectivos de un grafo. Para implementar la cola con prioridad utilizamos la técnica del heap mínimo así obtenemos un ordenamiento de las aristas de menor a mayor. Utilizamos dos arreglos, uno será el arreglo que tendrá la prioridad que es el peso de la arista y el otro arreglo contendrá el par de nodos que une la arista, en este arreglo guardaremos en cada posición un objeto llamado Par. La posición de la arista se moverá en simultaneo con la del otro arreglo, para guardar la información de la prioridad y su par de nodos. La complejidad de los métodos se detalla a continuación, pero los métodos tienen entre complejidad constante y logarítmica, que es lo interesante de utilizarla.

# Interfaz de Cola con Prioridad:

**package api;**

**import implementation.ColaPrioridad.Par;**

**public interface ColaPrioridadTDA {**

**/\*\***

**\* @Tarea\_Nombre: InicializarCola.**

**\* @Tarea\_Descripción: Se crea una estructura vacía lista para ser utilizada.**

**\* @Parámetros: Recibe la cantidad de elementos de la cola con prioridad.**

**\* @Devuelve: No retorna ya que es de tipo void.**

**\* @Precondiciones: La cantidad de elementos que se recibe es igual o mayor a la cantidad de aristas del grafo.**

**\* @Postcondiciones: No tiene postcondiciones.**

**\* @Excepción: No tiene excepciones.**

**\*/**

**public void inicializarCola(int capacidad);//constante O(1)**

**/\*\***

**\* @Tarea\_Nombre: AcolarPrioridad.**

**\* @Tarea\_Descripción: Se acola según el peso de la arista del par de nodos.**

**\* @Parámetros: Recibe nodo origen, nodo destino y peso de la arista que une esos nodos.**

**\* @Devuelve: No retorna ya que es de tipo void.**

**\* @Precondiciones: No tiene precondición.**

**\* @Postcondiciones: La cola con prioridad mantiene su orden al haber insertado el nuevo elemento.**

**\* @Excepción: No tiene excepciones.**

**\*/**

**public void acolarPrioridad(int nodoOrigen, int nodoDestino, int pesoArista);//logarítmico O(log(n))**

**/\*\***

**\* @Tarea\_Nombre: Desacolar.**

**\* @Tarea\_Descripción: Se elimina el primer elemento de la cola.**

**\* @Parámetros: No recibe parámetros.**

**\* @Devuelve: No retorna ya que es de tipo void.**

**\* @Precondiciones: La cola no debe estar vacía.**

**\* @Postcondiciones: La cola con prioridad mantiene su orden al haber eliminado el elemento.**

**\* @Excepción: No tiene excepciones.**

**\*/**

**public void desacolar();//logarítmico O(log(n))**

**/\*\***

**\* @Tarea\_Nombre: ColaVacia.**

**\* @Tarea\_Descripción: Verifica si la cola esta vacía.**

**\* @Parámetros: No recibe parámetros.**

**\* @Devuelve: Retorna un booleano, false si tiene elementos y true si no tiene elementos.**

**\* @Precondiciones: No tiene precondición.**

**\* @Postcondiciones: No tiene postcondiciones.**

**\* @Excepción: No tiene excepciones.**

**\*/**

**public boolean colaVacia();//constante O(1)**

**/\*\***

**\* @Tarea\_Nombre: Primero.**

**\* @Tarea\_Descripción: Retorna el primero elemento de la cola.**

**\* @Parámetros: No recibe parámetros.**

**\* @Devuelve: Retorna una clase Par que tiene la información de un par de nodos.**

**\* @Precondiciones: La cola no debe estar vacía.**

**\* @Postcondiciones: No tiene postcondiciones.**

**\* @Excepción: No tiene excepciones.**

**\*/**

**public Par primero ();//constante O(1)**

**/\*\***

**\* @Tarea\_Nombre: Prioridad.**

**\* @Tarea\_Descripción: Devuelve la prioridad del primero elemento de la cola.**

**\* @Parámetros: No recibe parámetros.**

**\* @Devuelve: Retorna un tipo de dato int.**

**\* @Precondiciones: La cola no debe estar vacía.**

**\* @Postcondiciones: No tiene postcondiciones.**

**\* @Excepción: No tiene excepciones.**

**\*/**

**public int prioridad();//constante O(1)**

**}**

# Implementación de Cola con Prioridad utilizando la estructura de heap:

**package** implementation;

**import** api.ColaPrioridadTDA;

**public** **class** ColaPrioridad **implements** ColaPrioridadTDA {

**public** **class** Par {

**public** **int** V\_origen;

**public** **int** V\_destino;

}

**int** [] prioridad;

Par[] valor;

**int** size;

Par vertices;

**public** **void** inicializarCola(**int** capacidad) {

prioridad= **new** **int** [capacidad+1]; //constante O(1)+

valor= **new** Par [capacidad+1]; //constante O(1)+

size=0; //constante O(1)=

} //constante O(1)

**private** **void** swap(**int** i, **int** j){//el valor y prioridad se modifican juntos

**int** tmp = prioridad[i]; //constante O(1)+

Par tmp2 = valor[i]; //constante O(1)+

prioridad[i] = prioridad[j]; //constante O(1)+

valor[i] = valor[j]; //constante O(1)+

prioridad[j] = tmp; //constante O(1)+

valor[j] = tmp2; //constante O(1)=

} //constante O(1)

**private** **void** heapifyUp(**int** k) {

**while** (prioridad[k] < prioridad[k/2] && (k/2)!=0 ) {//log n\*(

swap(k , k/2); //constante O(1)+

k = k/2; //constante O(1))=

}

} //logarítmico O(log(n))

**public** **void** acolarPrioridad(**int** nodoOrigen, **int** nodoDestino, **int** pesoArista) {

prioridad[++size]=pesoArista; //constante O(1)+

Par vertices = **new** Par ();

vertices.V\_destino=nodoDestino; //constante O(1)+

vertices.V\_origen=nodoOrigen; //constante O(1)+

valor[size]=vertices; //constante O(1)+

heapifyUp(size); //logarítmico O(log(n))=

//logarítmico O(log(n))

}

**private** **void** heapifyDown(**int** k) {

**int** posMenor = k; //constante O(1) +

**int** posIzquierdo = 2\*k; //constante O(1) +

**int** posDerecho = 2\*k + 1;//constante O(1) +

**if** (posIzquierdo <= size && prioridad[posIzquierdo] < prioridad[posMenor] ){ // constante O(1) +

posMenor = posIzquierdo; //constante O(1) +

}

**if** (posDerecho <= size && prioridad[posDerecho] < prioridad[posMenor] ) { //constante O(1) +

posMenor = posDerecho; //constante O(1) +

}

**if** (posMenor != k ) { //constante O(1) +

swap(k, posMenor); //constante O(1) +

heapifyDown(posMenor); //logarítmico O(log(n))=

}

} //logarítmico O(log(n))

**public** **void** desacolar() {

**if**(colaVacia()) { //constante O(1) +

**throw** **new** RuntimeException("La cola esta vacia")//constante O(1)

}

prioridad[1] = prioridad[size]; //+constante O(1) +

valor[1]=valor[size]; //constante O(1) +

size--; //constante O(1) +

heapifyDown(1); //logarítmico O(log(n))=

} //logarítmico O(log(n))

**public** **boolean** colaVacia() {

**return** size==0 ; //constante O(1)

}

**public** Par primero() {

**if**(colaVacia()) {

**throw** **new** RuntimeException("La cola esta vacia");//constanteO(1)=

}**else** {

**return** valor[1];

} //constante O(1)

}

**public** **int** prioridad() {

**if**(colaVacia()) { //constante O(1) +

**throw** **new** RuntimeException("La cola esta vacia");//constante O(1)=

}**else** {

**return** prioridad[1];}

} //constante O(1)

}

# DisjointSet

# ¿Qué es un Dijointset?

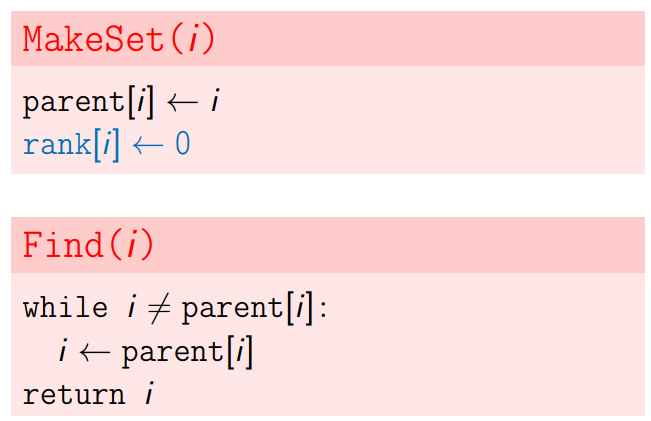
Un conjunto disjunto es una [estructura de datos](https://es.wikipedia.org/wiki/Estructura_de_datos) que mantiene un conjunto de elementos [particionados](https://es.wikipedia.org/wiki/Partici%C3%B3n_(matem%C3%A1tica)) en un número de conjuntos disjuntos(no se solapan los conjuntos). Ósea dos conjuntos son disjuntos si su intersección es vacía. Un algoritmo Unión-Buscar es un algoritmo que realiza tres importantes operaciones en esta estructura de datos:

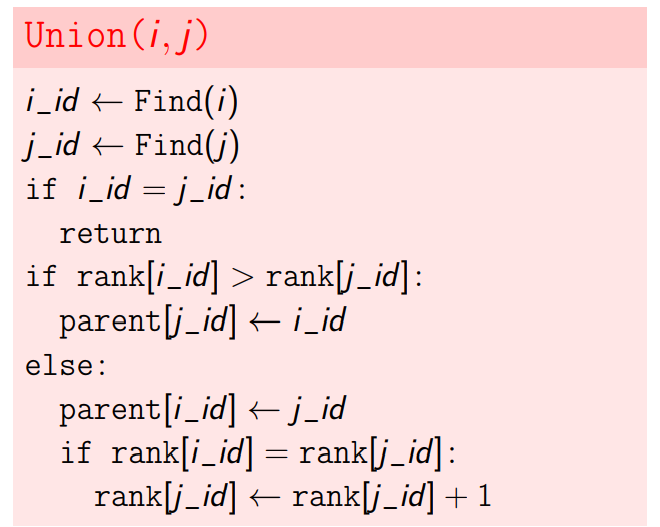
Buscar: Determina a cuál subconjunto pertenece un elemento. Esta operación puede usarse para verificar si dos elementos están en el mismo conjunto.

Unión: Une dos subconjuntos en uno solo.

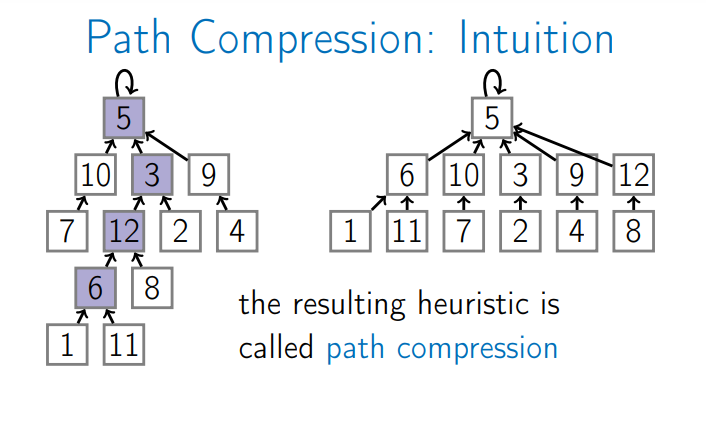
Crear un conjunto: Generalmente trivial, esta crea un conjunto con un elemento dado.

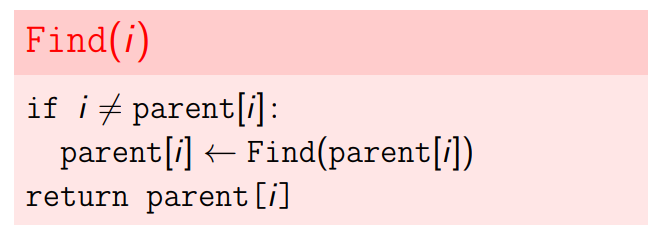
La **unión por rank**, consiste en siempre añadir el árbol más pequeño a la raíz del árbol más grande. Como la profundidad del árbol afecta el tiempo de ejecución del algoritmo, el árbol con menor profundidad es añadido a la raíz del árbol con mayor profundidad, el cual aumenta su profundidad solo si sus profundidades son iguales. En el contexto de este algoritmo, el término rank se usa en vez de profundidad porque este deja de ser igual a la profundidad si se usa la *compresión de camino* (descrita más abajo). Los árboles con un solo elemento tienen rank igual a cero, y cualesquiera dos árboles del mismo rank r son combinados, el rank resultante es r+1. Solo aplicando esta técnica se obtiene complejidad O (log n) para buscar y unión, debido a la altura del árbol.





La segunda mejora, llamada **compresión de camino**, es una forma de aplanar la estructura del árbol cuando se aplique Buscar. La idea es que cada nodo visitado en el camino hacia la raíz puede ser añadido directamente a la raíz; todos ellos comparten el mismo representativo. Para lograr este efecto, mientras Buscar recursivamente se mueve hacia la raíz, este cambia la referencia del padre del nodo a la raíz que encuentre. El árbol resultante es más aplanado, acelerando operaciones futuras no solo en estos elementos sino también en aquellos que referencian a estos.





Estas dos técnicas se complementan una a otra**; aplicadas juntas**, el tiempo [amortizado](https://es.wikipedia.org/wiki/An%C3%A1lisis_de_amortizaci%C3%B3n) es solo **{\displaystyle O(\alpha (n))}O(α(n)).** Donde **α(n)** es la [inversa de la función](https://es.wikipedia.org/w/index.php?title=Inversa_de_la_funci%C3%B3n&action=edit&redlink=1) *n* = f(*x*) = A(*x,x*) y A es la  [*función de Ackermann*](https://es.wikipedia.org/wiki/Funci%C3%B3n_de_Ackermann) . Esta función crece muy rápidamente, por lo tanto, su inversa crece muy lentamente y su resultado es inferior a 4 para entradas prácticamente de cualquier tamaño, de manera que se asimila a una función constante.

Crear el conjunto tiene una complejidad constante O (1) y consideramos como precondición que cuando se agregue un elemento el mismo no ha sido añadido antes.

**¿Para qué se utiliza DisjointSet?**

|  |  |
| --- | --- |
| **Operación** | **Complejidad** |
| Inicializar Disjointset | O(1) |
| Agregar | O(1) |
| Fusionar | O( α(n) ) |
| Id | O( α(n) ) |

Las estructuras de datos para conjuntos disjuntos modelan el [particionamiento de conjuntos](https://es.wikipedia.org/wiki/Partici%C3%B3n_(matem%C3%A1tica)), por ejemplo las [componentes conexas](https://es.wikipedia.org/wiki/Componente_conexa) de un [grafo no dirigido](https://es.wikipedia.org/wiki/Grafo_no_dirigido). Este modelo también puede utilizarse para verificar si dos vértices pertenecen a una misma componente conexa, o para comprobar si al añadir una arista entre dos vértices se forma un ciclo. El algoritmo Unión-Buscar es usado es usado en la implementación del [algoritmo de Kruskal](https://es.wikipedia.org/wiki/Algoritmo_de_Kruskal) para hallar el [árbol de recubrimiento mínimo](https://es.wikipedia.org/wiki/%C3%81rbol_recubridor_m%C3%ADnimo) de un grafo ya que nos indica si el agregado de una arista genera un ciclo o no.

# Interfaz de Disjointset:

**package** api;

**public** **interface** DisjointSetTDA {

/\*\*

\* **@Tarea\_Nombre**: InicializarDisjointSet.

\* **@Tarea\_Descripción**: Se crea una estructura vacía lista para ser utilizada.

\* **@Parámetros**: Recibe la cantidad de elementos del DisjointSet.

\* **@Devuelve**: No retorna ya que es de tipo void.

\* **@Precondiciones**: No tiene precondición.

\* **@Postcondiciones**: No tiene postcondiciones.

\* **@Excepción**: No tiene excepciones.

\*/

**public** **void** inicializarDisjointSet(**int** x); //COSTO: constante O(1)

/\*\*

\* **@Tarea\_Nombre**: Agregar.

\* **@Tarea\_Descripción**: Agrega un nuevo elemento a la estructura y se crea un nuevo conjunto unitario que contiene el nuevo elemento.

\* **@Parámetros**: Recibe un tipo de dato int que es el elemento.

\* **@Devuelve**: No retorna ya que es de tipo void.

\* **@Precondiciones**: Disjointset inicializado y el nuevo elemento **no** es parte del conjunto.

\* **@Postcondiciones**: El identificador del nuevo elemento es el elemento mismo.

\* **@Excepción**: No tiene excepción.

\*/

**public** **void** agregar(**int** x) ; //COSTO: constante O(1)

/\*\*

\* **@Tarea\_Nombre**: Id

\* **@Tarea\_Descripción**: Recibe como parámetro un elemento ingresado a la estructura y devuelva el identificador correspondiente.

\* **@Parámetros**: Recibe un tipo de dato int.

\* **@Devuelve**: Retorna un tipo de dato int.

\* **@Precondiciones**: Disjointset inicializado y el elemento x debe existir en el conjunto.

\* **@Postcondiciones**: No tiene postcondiciones.

\* **@Excepción**: No tiene excepción.

\*/

**public** **int** id(**int** x) ; //COSTO: función inversa de Ackermann , tiempo amortizado casi constante O(α(n))

/\*\*

\* **@Tarea\_Nombre**: Fusionar

\* **@Tarea\_Descripción**: Recibe como parámetro dos elementos de la estructura y fusiona los respectivos conjuntos a los que pertenecen.

\* **@Parámetros**: Recibe un tipo de dato int.

\* **@Devuelve**: No retorna ya que es de tipo void.

\* **@Precondiciones**: Disjointset inicializado y los elementos x e y deben existir en el conjunto.

\* **@Postcondiciones**: Ambos elementos quedan con el mismo identificador.

\* **@Excepción**: No tiene excepción.

\*/

**public** **void** fusionar(**int** x,**int** y) ; //COSTO: función inversa de Ackermann , tiempo amortizado casi constante O(α(n))

}

# Implementación de Disjointset utilizando unión por rank y compresión de camino:

**package** implementation;

**import** api.DisjointSetTDA;

**public** **class** DisjointSet **implements** DisjointSetTDA{

**private** **int**[] rank;

**private** **int**[] parent;

**public** **void** inicializarDisjointSet(**int** x) {

rank = **new** **int**[x]; //constante O(1)+

parent = **new** **int**[x]; //constante O(1)=

//constante O(1)

}

**public** **void** agregar(**int** n) {

parent[n]=n; //constante O(1)+

rank[n]=0; //constante O(1)=

//constante O(1)

}

**public** **int** id(**int** x) {

**if** (parent[x] != x) {

parent[x] = id(parent[x]);

}

**return** parent[x];

//O( α(n) )

}

**public** **void** fusionar(**int** x, **int** y)

{

**int** xRoot = id(x), yRoot = id(y); // O( α(n) )+

**if** (xRoot == yRoot) //constante O(1)+

**return**;

**if** (rank[xRoot] > rank[yRoot]) {//constante O(1)+

parent[yRoot] = xRoot; //constante O(1)+

}

**else** {

parent[xRoot] = yRoot; //constante O(1)+

**if**(rank[xRoot] == rank[yRoot]) //constante O(1)+

rank[yRoot] = rank[yRoot] + 1; //constante O(1)

}

// = O( α(n) )

}

# Kruskal

El algoritmo de Kruskal es un [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) de la [teoría de grafos](https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos) para encontrar un [árbol de recubrimiento mínimo](https://es.wikipedia.org/wiki/%C3%81rbol_recubridor_m%C3%ADnimo) en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada [componente conexa](https://es.wikipedia.org/w/index.php?title=Componente_conexa_(teor%C3%ADa_de_los_grafos)&action=edit&redlink=1)).

Primero ordenaremos las aristas del grafo por su peso de menor a mayor que lo hacemos con la Cola de Prioridad. Mediante la técnica greedy Kruskal intentará unir cada arista siempre y cuando no se forme un ciclo, ello se hará con el Disjoinset. Como hemos ordenado las aristas por peso comenzaremos con la arista de menor peso, si los vértices que contienen dicha arista no están en la misma componente conexa no formaran un ciclo por lo cual tomamos esa arista como valida. Tendremos n-1 aristas seleccionadas, donde n es la cantidad de nodos del grafo.

# Función para agregar las aristas de un grafo a la cola con prioridad:

Recibimos un grafo G en formato de matriz, recorremos toda la matriz preguntando en cada posición que valor numérico posee, de ser un cero no hay arista existente, de lo contrario se agrega la arista a la cola con prioridad Q y los respectivos nodos que une esa arista. Al final se retorna la cola con prioridad Q. El coste de esta función es O (n^2 . log n) debido a que tenemos que recorrer toda la matriz lo cual ya nos da un coste de n^2 , pero también en cada iteración debemos agregar en la cola de prioridad, la prioridad y los nodos lo cual tiene un coste de log n .

**public** **static** ColaPrioridadTDA AristasG\_ColaPrioridad(**int**[][] G , ColaPrioridadTDA Q ) {

**for**(**int** i = 1; i < G.length; i++) //n.(

{

**for**(**int** y = 1; y < G.length; y++) //n.(

{

**if**(G[i][y]!=0) {

**int** peso = G[i][y]; // O(1)+

Q.acolarPrioridad(i, y, peso);//+ O(log(n))

}

}

// ) = O( n^2 . log n )

}

**return** Q;

}

# Función del algoritmo Kruskal – “KruskalCode”:

Recibimos un grafo *G* en forma de matriz, un valor entero *x* que corresponde al número de mayor valor mas 1 que se ingresara en el disjointset , por ejemplo si queremos ingresar en el disjointset los elementos 10 , 22 y 54 , el valor de x será : 54+1 = 55 debido a que como esta implementado el disjointset es condición necesaria. Luego se ingresará un *x2* el cual debe corresponder a la cantidad de aristas que tiene el grafo G. Luego con estos parámetros recibidos la función agregara cada nodo del grafo en el dijointset para crear los árboles de un único elemento. Usaremos la función *AristaG\_ColaPrioridad* en donde se agregan todas las aristas del grafo y los nodos que unen esa arista en la cola con prioridad *Q*. En Q las aristas serán ordenadas de menor a mayor, donde siempre obtendremos la menor al llamar al primer elemento de la cola. Una vez que la Q contiene todas las aristas , iteramos n-1 veces consultando en Q por el primer elemento y luego descolándolo, con este elemento primero podes extraer el par de nodos y consultar si pertenecen o no al mismo conjunto a través del método *id* del disjointset , si los *id* de esos nodos son distintos los fusionamos y esa arista es seleccionada para ser parte del árbol mínimo de recubrimiento , por lo cual **se mostrara por pantalla al usuario la arista , que peso tiene y que nodos unió** , caso contrario no se tomara en cuenta la arista ya que no queremos ciclos , para no perder iteraciones en ese caso se incrementa el contador . Con respecto a la complejidad lo que refiere el disjointset se itera m veces y se utiliza *fusión* y *id*, con lo cual ahí contamos con O(m.α(n)) , debido al desacolar de Q de coste logarítmico termina quedando O(m . log n), pero también tenemos la función *AristaG\_ColaPrioridad* la cual eleva la complejidad a O(n^2 . log n). Por lo cual la complejidad total de esta función queda como O (n^2 . log n).

**public** **static** **void** KruskalCode(**int**[][] G , **int** x , **int** x2) {

DisjointSetTDA T = **new** DisjointSet(); // O(1)+

T.inicializarDisjointSet(x);// O(1)+

ColaPrioridadTDA Q = **new** ColaPrioridad();// O(1)+

Q.inicializarCola(x2);// O(1)+

**for**(**int** i = 1; i < G.length ; i++) // n.(

{

T.agregar(i);//arboles de un solo elemento , O(1)

}// )= O(n)

//pesos de las aristas y nodos agregados a la cola con prioridad

Q = *AristasG\_ColaPrioridad*( G , Q ); // O( n^2 . log n )+

**int** n =G.length-1;// O(1)+

**for**(**int** i = 1; i < n ; i++) //n(

{

Par par = Q.primero();// O(1)+

**int** v = par.V\_origen;// O(1)+

**int** u = par.V\_destino;// O(1)+

**int** uset= T.id(u);// O( α(n) )+

**int** vset= T.id(v);// O( α(n) )+

**if**(uset != vset) {// O(1)+

T.fusionar(uset, vset);// O( α(n) )+

System.***out***.println("Nodos unidos :"+ v + "-" +u + " | Peso de Arista : " + Q.prioridad()); // O(1)+

}**else** {

n++; // no perdemos la iteración de los casos que rechazamos una arista , O(1)+

}

Q.desacolar(); // O(log n) )

}

//= O( n . log n )

**return** ;

// costo : O( n^2 . log n )

}

# Ejemplo utilizando “KruskalCode”:

**public** **static** **void** main(String[] args) {

**int** [][] Grafo1 = **new** **int** [8][8] ;

//agregamos las aristas

Grafo1 [1][2]= 4;

Grafo1 [1][3]= 7;

Grafo1 [1][6]= 5;

Grafo1 [1][7]= 1;

Grafo1 [2][3]= 3;

Grafo1 [3][7]= 3;

Grafo1 [3][4]= 5;

Grafo1 [4][5]= 7;

Grafo1 [4][7]= 9;

Grafo1 [5][7]= 4;

Grafo1 [5][6]= 8;

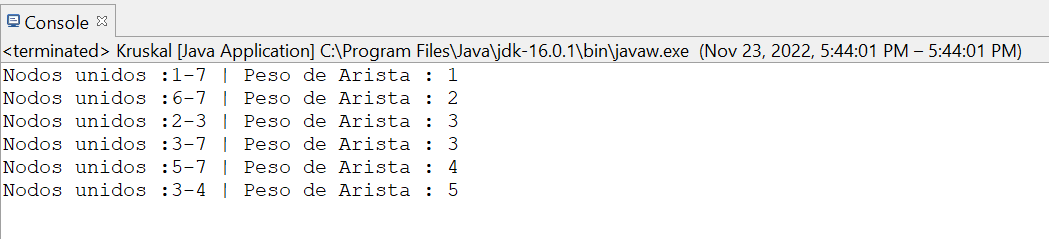
Grafo1 [6][7]= 2;

**int** elementosDisjunt = 9;// número del elemento más grande que se ingresara más 1 en el DisjontSet

**int** cantidadAristas = 12; // cantidad de aristas en el grafo

*KruskalCode*(Grafo1 , elementosDisjunt , cantidadAristas);//imprime por pantala que aristas son las seleccionadas y que nodos une

}



# Bibliografía

# <https://es.wikipedia.org/wiki/Estructura_de_datos_para_conjuntos_disjuntos>

# <https://www.techiedelight.com/es/disjoint-set-data-structure-union-find-algorithm/#:~:text=Disjoint%E2%80%93set%20forests%20son%20estructuras,del%20%C3%A1rbol%20de%20ese%20conjunto>.

* + https://es.wikipedia.org/wiki/Algoritmo\_de\_Kruskal
  + Alexander S. Kulikov Steklov Institute of Mathematics at St. Petersburg Russian Academy of Sciences :

https://paperzz.com/doc/7141210/disjoint-sets--naive-implementation