

## Resumen Algo II - Buchwald

### 1. División y Conquista:

- Dividir un problema en mas de una parte (significativamente, por ejemplo /2)
- Hacer algo para solucionar cada parte del problema
- Juntar todo

Usualmente (no se si siempre) es de forma recursiva.

Teorema maestro

Si tenemos un algoritmo cuya ecuación de recurrencia es:

$$T(n) = A * T(n / B) + \mathcal{O}(n^C)$$

A: cantidad de llamados recursivos

B: proporción del tamaño original con el que llamamos recursivamente

$\mathcal{O}(n^C)$ : el costo de *partir y juntar* (todo lo que no son llamados recursivos)

Si tenemos un algoritmo cuya ecuación de recurrencia es:

$$T(n) = A * T(n / B) + \mathcal{O}(n^C)$$

$$< \rightarrow T(n) = \mathcal{O}(n^C)$$

Si:  $\log_B(A) = C \rightarrow \mathcal{O}(n^C \log_B(n)) = \mathcal{O}(n^C \log(n))$

$$> \rightarrow \mathcal{O}(n^{\log_B(A)})$$

### 2. Ordenamientos comparativos

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(N)$
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(N)$
Heap Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(1)$

Sorting Algorithms	In - Place	Stable
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Quick Sort	Yes	No
Merge Sort	No (because it requires an extra array to merge the sorted subarrays)	Yes
Heap Sort	Yes	No

```

func BubbleSort(array[] int)[]int {
    for i:=0; i< len(array)-1; i++ {
        for j:=0; j < len(array)-i-1; j++ {
            if (array[j] > array[j+1]) {
                array[j], array[j+1] = array[j+1],
                array[j]
            }
        }
    }
    return array
}

func mergeSort(arr []int) []int {
    if len(arr) <= 1 {
        return arr
    }
    middle := len(arr) / 2
    left := mergeSort(arr[:middle])
    right := mergeSort(arr[middle:])
    return merge(left, right)
}

func merge(left, right []int) []int {
    result := make([]int, len(left)+len(right))
    i, j := 0, 0
    for k := 0; k < len(result); k++ {
        if i >= len(left) {
            result[k] = right[j]
            j++
        } else if j >= len(right) {
            result[k] = left[i]
            i++
        } else if left[i] < right[j] {
            result[k] = left[i]
            i++
        } else {
            result[k] = right[j]
            j++
        }
    }
    return result
}

```

```

func Selection_Sort(array[] int, size int) []int {
    var min_index int
    var temp int
    for i := 0; i < size - 1; i++ {
        min_index = i
        // Find index of minimum element
        for j := i + 1; j < size; j++ {
            if array[j] < array[min_index] {
                min_index = j
            }
        }
        temp = array[i]
        array[i] = array[min_index]
        array[min_index] = temp
    }
    return array
}

func insertionSort(arr []int) []int {
    for i := 1; i < len(arr); i++ {
        key := arr[i]
        j := i - 1
        for j >= 0 && arr[j] < key {
            arr[j+1] = arr[j]
            j = j - 1
        }
        arr[j+1] = key
    }
    return arr
}

```

```

func quickSort(arr []int, low, high int) {
    if low < high {
        var pivot = partition(arr, low, high)
        quickSort(arr, low, pivot)
        quickSort(arr, pivot + 1, high)
    }
}

func partition(arr []int, low, high int) int {
    var pivot = arr[low]
    var i = low
    var j = high

    for i < j {
        for arr[i] <= pivot && i < high {
            i++
        }
        for arr[j] > pivot && j > low {
            j--
        }
        if i < j {
            var temp = arr[i]
            arr[i] = arr[j]
            arr[j] = temp
        }
    }
    arr[low] = arr[j]
    arr[j] = pivot
    return j
}

```

```

func HeapSort[T comparable](elementos []T, funcion_cmp func(T, T) int) {
    heapify(elementos, funcion_cmp)
    for j := len(elementos) - 1; j >= 0; j-- {
        elementos[0], elementos[j] = elementos[j], elementos[0]
        downheap(elementos, funcion_cmp, j, 0)
    }
}

func heapify[T comparable](arreglo []T, funcion_cmp func(T, T) int) {
    for i := len(arreglo) - 1; i >= 0; i-- {
        downheap(arreglo, funcion_cmp, len(arreglo), i)
    }
}

func downheap[T comparable](datos []T, funcion_cmp func(T, T) int, cant, index int) {
    pos_hijo_izq := 2*index + 1
    pos_hijo_der := 2*index + 2

    mayor := mayor(datos, funcion_cmp, cant, index, pos_hijo_izq, pos_hijo_der)
    if mayor != index {
        datos[index], datos[mayor] = datos[mayor], datos[index]
        downheap(datos, funcion_cmp, cant, mayor)
    }
}

```

### 3. Ordenamientos no comparativos

Sorting Algorithms	Special Input Condition	Time Complexity			Space Complexity
		Best Case	Average Case	Worst Case	
<b>Counting Sort</b>	Each input element is an integer in the range 0- K	$\Omega(N + K)$	$\Theta(N + K)$	$O(N + K)$	$O(K)$
<b>Radix Sort</b>	Given n digit number in which each digit can take on up to K possible values	$\Omega(NK)$	$\Theta(NK)$	$O(NK)$	$O(N + K)$
<b>Bucket Sort</b>	Input is generated by the random process that distributes elements uniformly and independently over the interval [0, 1]	$\Omega(N + K)$	$\Theta(N + K)$	$O(N^2)$	$O(N)$

```

func countsort(array []int, exp int) {
    m := len(array)
    output := make([]int, m)
    count := make([]int, 10)

    for a := 0; a < 10; a++ {
        count[a] = 0
    }

    for a := 0; a < m; a++ {
        index := (array[a] / exp) % 10
        count[index]++
    }

    for a := 1; a < 10; a++ {
        count[a] += count[a-1]
    }

    for a := m - 1; a >= 0; a-- {
        index := (array[a] / exp) % 10
        output[count[index]-1] = array[a]
        count[index]--
    }

    for a := 0; a < m; a++ {
        array[a] = output[a]
    }
}

```

```

func radsorting(array []int) {
    maximum := findmaximum(array)

    for exp := 1; maximum/exp > 0; exp *= 10 {
        countsort(array, exp)
    }
}

func bucketSort(arr []int) []int {
    maxVal := 0
    for _, val := range arr {
        if val > maxVal {
            maxVal = val
        }
    }

    buckets := make([][]int, maxVal+1)
    for i := range buckets {
        buckets[i] = make([]int, 0)
    }

    for _, val := range arr {
        buckets[val] = append(buckets[val], val)
    }

    result := make([]int, 0)
    for _, bucket := range buckets {
        result = append(result, bucket...)
    }

    return result
}

```

## 4. TDAs

### 1. Pila

```
type pilaDinamica[T any] struct {
    datos []T
    cantidad int
}
```

Primitivas:

- func (pila \*pilaDinamica[T]) EstaVacia() bool O(1)
- func (pila \*pilaDinamica[T]) VerTope() T O(1)
- func (pila \*pilaDinamica[T]) Apilar(valor T) O(1)
- func (pila \*pilaDinamica[T]) Desapilar() T O(1)

Redimensionar:

```
func (pila *pilaDinamica[T]) _redimensionar(nuevaCapacidad int) {
    nuevoDatos := make([]T, nuevaCapacidad)
    _ = copy(nuevoDatos, pila.datos)
    pila.datos = nuevoDatos
}
```

### 2. Cola

```
type nodo[T any] struct {
    dato T
    proximo *nodo[T]
}

type colaEnlazada[T any] struct {
    // Primer nodo de la cola, primero en salir al desencolar.
    primerNodo *nodo[T]

    // Ultimo nodo de la cola, el ultimo que fue encolado.
    ultimoNodo *nodo[T]
}
```

Primitivas:

- func (cola \*colaEnlazada[T]) EstaVacia() bool O(1)
- func (cola \*colaEnlazada[T]) VerPrimero() T O(1)
- func (cola \*colaEnlazada[T]) Encolar(dato T) O(1)
- func (cola \*colaEnlazada[T]) Desencolar() T O(1)

### 3. Lista Enlazada

<pre>type nodoLista[T any] struct {     dato T     proximo *nodoLista[T] }  type listaEnlazada[T any] struct {     primerNodo *nodoLista[T]     ultimoNodo *nodoLista[T]     largo int }  type iteradorExterno[T any] struct {     lista *listaEnlazada[T]     nodoAnterior *nodoLista[T]     nodoActual *nodoLista[T] }</pre>	<p>Primitivas:</p> <ul style="list-style-type: none"> <li>- func (lista *listaEnlazada[T]) EstaVacia() bool O(1)</li> <li>- func (lista *listaEnlazada[T]) InsertarPrimero(elem T) O(1)</li> <li>- func (lista *listaEnlazada[T]) InsertarUltimo(elem T) O(1)</li> <li>- func (lista *listaEnlazada[T]) BorrarPrimero() T O(1)</li> <li>- func (lista listaEnlazada[T]) VerPrimero() T O(1)</li> <li>- func (lista listaEnlazada[T]) VerUltimo() T O(1)</li> <li>- func (lista listaEnlazada[T]) Largo() int O(1)</li> <li>- func (lista *listaEnlazada[T]) Iterar(visitar func(T) bool) O(n)</li> <li>- func (lista *listaEnlazada[T]) Iterador() IteradorLista[T] O(1)</li> </ul> <p>Primitivas Iterador Externo:</p> <ul style="list-style-type: none"> <li>- func (iter *iteradorExterno[T]) VerActual() T O(1)</li> <li>- func (iter *iteradorExterno[T]) HaySiguiente() bool O(1)</li> <li>- func (iter *iteradorExterno[T]) Siguiente() O(1)</li> <li>- func (iter *iteradorExterno[T]) Insertar(elem T) O(1)</li> <li>- func (iter *iteradorExterno[T]) Borrar() T O(1)</li> </ul>
--	--

### 4. Hash

<pre>type parClaveValor[K comparable, V any] struct {     clave K     dato V }  type hashAbierto[K comparable, V any] struct {     tabla []TDALista.Lista[parClaveValor[K, V]]     cantidad int }  type iteradorExterno[K comparable, V any] struct {     hash *hashAbierto[K, V]     posActualTabla int     iteradorLista TDALista.IteradorLista[parClaveValor[K, V]] }</pre>	<p>Funciones Aux:</p> <pre>func buscarParClaveValor[K comparable, V any]{     iterLista TDALista.IteradorLista[parClaveValor[K, V]],     clave K) (bool, parClaveValor[K, V]) {      // La función recibe una clave del diccionario y un iterador     // de la lista de la posición de la tabla de hash     correspondiente     // a esa clave. Si la clave está en el diccionario, devuelve     true     // y su par clave valor. Caso contrario, devuelve false y un     par     // clave-valor nulo.      for iterLista.HaySiguiente() {         if iterLista.VerActual().clave == clave {             return true, iterLista.VerActual()         }         iterLista.Siguiente()     }     return false, *new(parClaveValor[K, V]) }</pre>
<p>Primitivas:</p> <ul style="list-style-type: none"> <li>- func (hash *hashAbierto[K, V]) Guardar(clave K, dato V) O(1) - O(n) → Peor Caso</li> <li>- func (hash *hashAbierto[K, V]) Perteneciente(clave K) bool O(1) - O(n) → Peor Caso</li> <li>- func (hash *hashAbierto[K, V]) Obtener(clave K) V O(1) - O(n) → Peor Caso</li> <li>- func (hash *hashAbierto[K, V]) Borrar(clave K) V O(1) - O(n) → Peor Caso</li> <li>- func (hash *hashAbierto[K, V]) Cantidad() int O(1)</li> <li>- func (hash *hashAbierto[K, V]) Iterar(visitar func(clave K, dato V) bool) O(n)</li> <li>- func (hash *hashAbierto[K, V]) Iterador() IteradorDiccionario[K, V] O(n)</li> </ul>	
<p>Primitivas Iterador Externo:</p> <ul style="list-style-type: none"> <li>- func (iter *iteradorExterno[K, V]) HaySiguiente() bool O(1)</li> <li>- func (iter *iteradorExterno[K, V]) VerActual() (K, V) O(1)</li> <li>- func (iter *iteradorExterno[K, V]) Siguiente() O(1)</li> </ul>	

## 5. Árboles

<pre>type nodoAbb[K comparable, V any] struct {     izquierdo **nodoAbb[K, V]     derecho  **nodoAbb[K, V]     clave     K     dato      V }</pre>	<pre>type abb[K comparable, V any] struct {     raiz      **nodoAbb[K, V]     cantidad  int     funcion_comp func(clave1,     clave2 K) int }</pre>	<pre>type iteradorABB[K comparable, V any] struct {     abb      *abb[K, V]     desde   *K     hasta   *K     pilainOrder     TDAPila.Pila[**nodoAbb[K, V]] }</pre>
--	---	---

Primitivas:

- func (abb \*abb[K, V]) Guardar O(log n)
- func (abb \*abb[K, V]) Pertenece(clave K) bool O(log n)
- func (abb \*abb[K, V]) Obtener(clave K) V O(log n)
- func (abb \*abb[K, V]) Borrar(clave K) V O(log n)
- func (abb \*abb[K, V]) Cantidad() int O(1)
- func (abb \*abb[K, V]) Iterar(visitar func(clave K, dato V) bool) O(n)
- func (abb \*abb[K, V]) IterarRango(desde, hasta \*K, visitar func(clave K, dato V) bool) O(n)
  - func (nodo \*nodoAbb[K, V]) iterarNodo(abb \*abb[K, V], desde, hasta \*K, visitar func(clave K, dato V) bool) bool O(n)
- func (abb \*abb[K, V]) Iterador() IterDiccionario[K, V] O(log n)
- func (abb \*abb[K, V]) IteradorRango(desde \*K, hasta \*K) IterDiccionario[K, V] O(log n + m), donde n es la cantidad de elementos en el árbol y m es la cantidad de elementos en el rango especificado.

Primitivas Iterador Externo:

- func (iter \*iteradorABB[K, V]) HaySiguiente() bool O(1)
- func (iter \*iteradorABB[K, V]) VerActual() (K, V) O(1)
- func (iter \*iteradorABB[K, V]) Siguiente() O(1)

```
func (abb *abb[K, V]) buscarNodoABB(nodoPadre, nodo **nodoAbb[K, V], clave K) (bool, **nodoAbb[K, V], **nodoAbb[K, V]) { O(log n)
    if *nodo == nil {
        return false, nodoPadre, nodo
    }
    if abb.funcion_comp((*nodo).clave, clave) == 0 {
        return true, nodoPadre, nodo
    }
    if abb.funcion_comp((*nodo).clave, clave) > 0 {
        return abb.buscarNodoABB(nodo, (*nodo).izquierdo, clave)
    }
    return abb.buscarNodoABB(nodo, (*nodo).derecho, clave)
}
```

```
func (iter *iteradorABB[K, V]) apilarHijosIzquierdos(nodo **nodoAbb[K, V]) {
    if *nodo == nil {
        return
    }
    iter.pilainOrder.Apilar(nodo)
    iter.apilarHijosIzquierdos((*nodo).izquierdo)
}
```

```
func (iter *iteradorABB[K, V]) iterarHastaInicioRango() {
    if iter.desde == nil || iter.abb.cantidad == 0 {
        return
    }
    claveActual := (*iter.pilainOrder.VerTope()).clave
    for iter.HaySiguiente() && iter.abb.funcion_comp(claveActual, *iter.desde) < 0 {
        iter.Siguiente()
        if iter.HaySiguiente() {
            claveActual = (*iter.pilainOrder.VerTope()).clave
        }
    }
}
```

## Recorridos

Recorrer un árbol significa pasar por cada uno de los nodos. Nombramos:

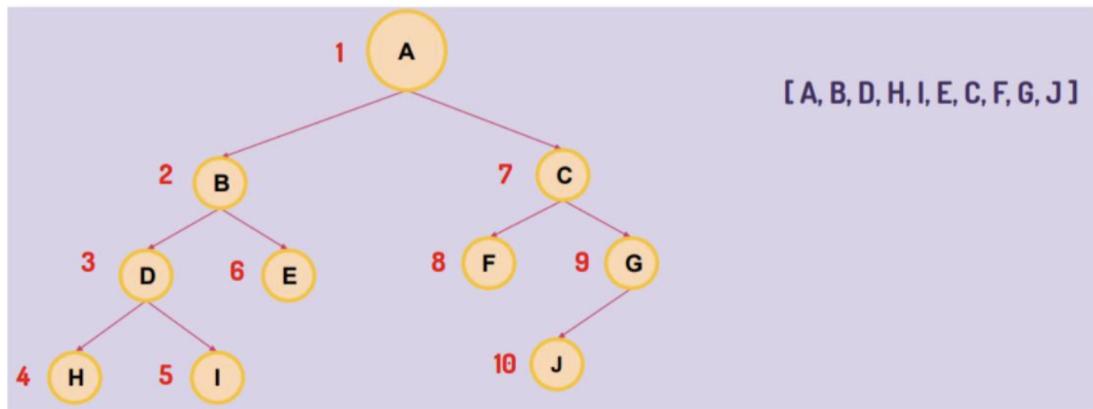
- N: nodo actual,
- D: subárbol derecho
- I: subárbol izquierdo



### Preorder

Primero se visita el nodo actual, luego el subárbol izquierdo y luego el derecho

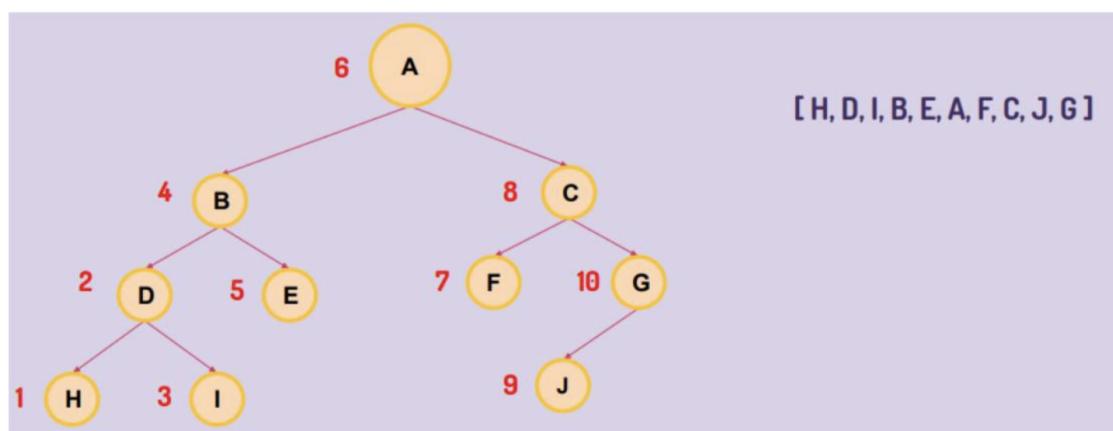
Se suele utilizar: Cuando quiero **clonar** árboles.



### Inorder

Primero se visita el subárbol izquierdo, luego el nodo actual y por último el subárbol derecho

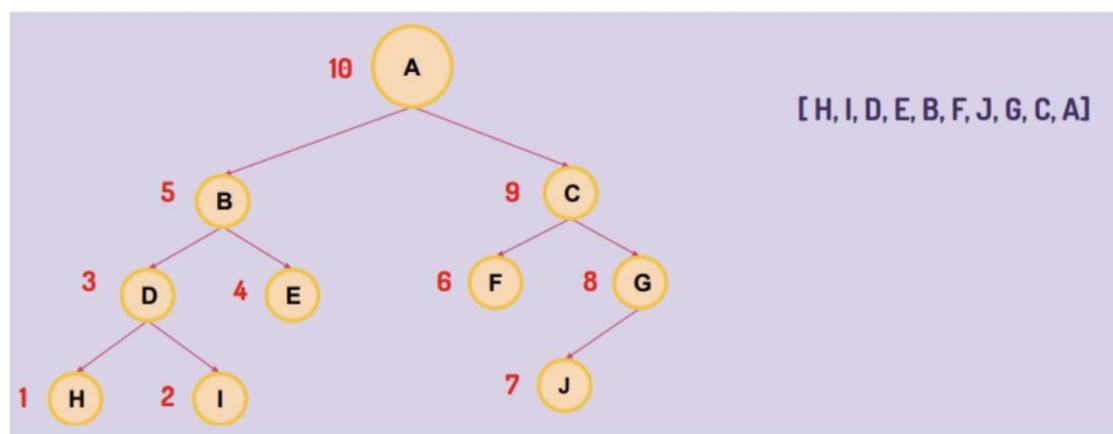
Se suele utilizar: Cuando quiero un recorrido **ordenado** del árbol.



## Postorder

Primero se visita el subárbol izquierdo, luego el derecho y por último el nodo actual.

Se suele utilizar: Cuando quiero **destruir** el árbol.

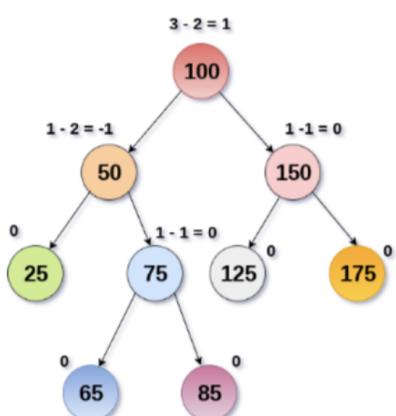


### a) AVL

Árbol binario de búsqueda auto balanceado. Cada hijo difiere en altura de un valor de -1, 0 o 1

- 0 → el nodo está equilibrado y sus sub-arboles tienen exactamente la misma altura
- 1 → el nodo está equilibrado y su sub-árbol derecho es un nivel más alto
- -1 → el nodo está equilibrado y su sub-árbol izquierdo es un nivel más alto

Si el  $|F\_e| \geq 2$  es necesario equilibrar.

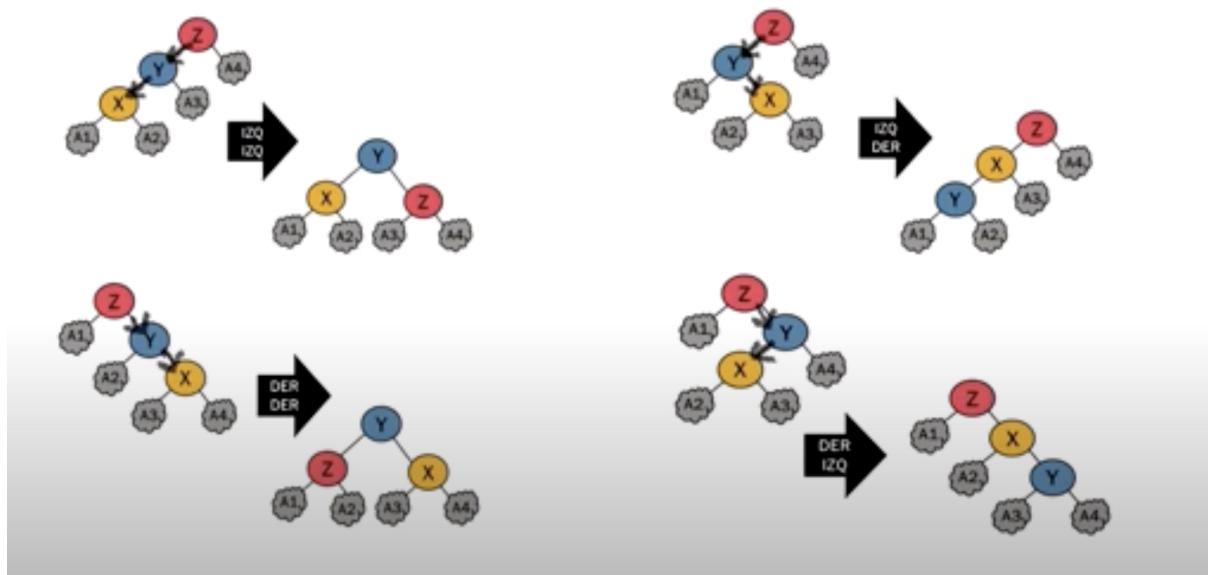


## Implementación

Un nodo de un AVL tiene los siguientes datos (como mínimo):

- Clave.
- Factor de balanceo.
- Puntero a un AVL a la derecha.
- Puntero a un AVL a la izquierda.

## Rotaciones



- **Post** rotación **IZQ - DER** hay que **hacer** rotación **IZQ - IZQ**
- **Post** rotación **DER - IZQ** hay que **hacer** rotación **DER - DER**

## 6. Heap

Un heap binario es un árbol binario que tiene ciertas características. Presenta cierta relación entre elementos, ya que hay un **criterio de orden parcial**.

- No hay un criterio de orden total
- No hay relación entre los hermanos, sobrinos y tíos

Este tipo de árbol se lo denomina casi completo → si recorro por niveles solo puedo tener elementos faltantes en el último nivel. El "agujero" siempre está del lado derecho. Siempre se contemplan los niveles de forma ordenada, no puedo empezar un nuevo nivel si hay espacios en el actual.

## Propiedades para ser Heap binario

1. Árbol ordenado por rama (la raíz es del árbol es el menor o mayor elemento)+
2. Todos los niveles del heap, exceptuando el último, están completos (árboles casi completos)
3. Solo se puede leer, buscar y borrar la raíz del heap.

Si estoy en la posición  $i$ :

- ¿Cuáles son las posiciones de sus hijos (si existen)?

$$\text{hijo\_izq} = 2*i + 1 \quad \text{hijo\_der} = 2*i + 2$$

- ¿Cuál es la posición de su padre?

$$\text{padre} = (i - 1) // 2$$

<pre>type heap[T comparable] struct {     datos    []T     cantidad int     funcion_cmp func(elem1,                       elem2 T) int }</pre>	<p>Primitivas:</p> <ul style="list-style-type: none"><li>- func (hp *heap[T]) EstaVacia() bool</li><li>- func (hp *heap[T]) Encolar(elem T)</li><li>- func (hp *heap[T]) VerMax() T</li><li>- func (hp *heap[T]) Desencolar() T</li><li>- func (hp *heap[T]) Cantidad() int</li></ul>
<pre>func upheap[T comparable](datos []T, funcion_cmp func(T, T) int, cant, index int) {     if index == 0 {         return     }      pos_padre := (index - 1) / 2     if funcion_cmp(datos[index], datos[pos_padre]) &gt; 0 {         datos[index], datos[pos_padre] = datos[pos_padre], datos[index]         upheap(datos, funcion_cmp, cant, pos_padre)     } }</pre>	
<pre>func downheap[T comparable](datos []T, funcion_cmp func(T, T) int, cant, index int) {     pos_hijo_izq := 2*index + 1     pos_hijo_der := 2*index + 2      mayor := mayor(datos, funcion_cmp, cant, index, pos_hijo_izq, pos_hijo_der)     if mayor != index {         datos[index], datos[mayor] = datos[mayor], datos[index]         downheap(datos, funcion_cmp, cant, mayor)     } }</pre>	

```

func HeapSort[T comparable](elementos []T, funcion_cmp func(T, T) int) {
    heapify(elementos, funcion_cmp)
    for j := len(elementos) - 1; j >= 0; j-- {
        elementos[0], elementos[j] = elementos[j], elementos[0]
        downheap(elementos, funcion_cmp, j, 0)
    }
}

```

```

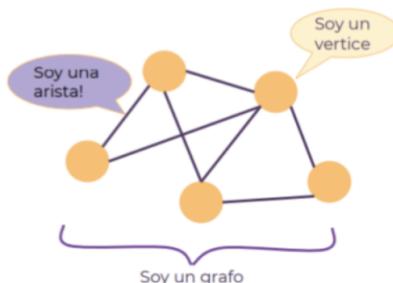
func heapify[T comparable](arreglo []T, funcion_cmp func(T, T) int) {
    for i := len(arreglo) - 1; i >= 0; i-- {
        downheap(arreglo, funcion_cmp, len(arreglo), i)
    }
}

```

## 7. Grafos

Un grafo es un par ordenado  $G = (V, E)$  donde:

- $V$  es un conjunto de vértices (o nodos)
- $E$  es un conjunto de aristas (o arcos)

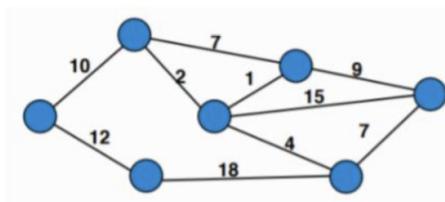


### ii. Definiciones grafo:

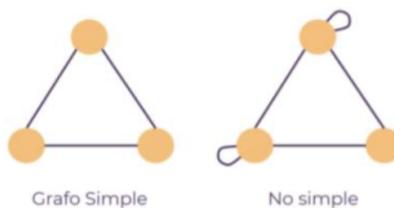
- **Orden de un grafo:** es la cantidad de vértices.
- **Tamaño de un grafo:** es la cantidad de aristas.
- **Grado de un vértice:** es la cantidad de aristas incidentes (ingresan al grafo)
- Grado de entrada: cantidad de aristas que entran al vértice
- Grado de salida: cantidad de aristas que salen de cada vértice

Grafo pesado: si sus **aristas tienen pesos asignados**.

Vendría a ser el "costo" a la hora de tener que transitar de un vértice hacia otro.



Grafo simple: es aquel que no posee aristas múltiples ni lazos. En un par de vértices hay solo una arista.



## B. REPRESENTACION

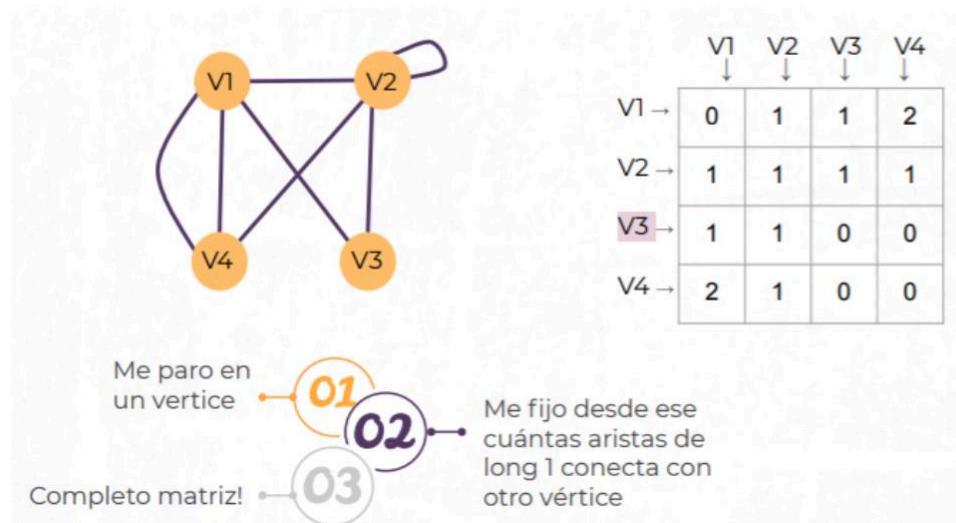
Las siguientes dos son las representaciones más utilizadas de un gráfo.

1. Matriz de adyacencia
2. Lista de adyacencia
3. Matriz de adyacencia

La elección de la representación del gráfo es específica de la situación. Depende totalmente del tipo de operaciones que se realizarán y la facilidad de uso.

### i. Matriz de adyacencia

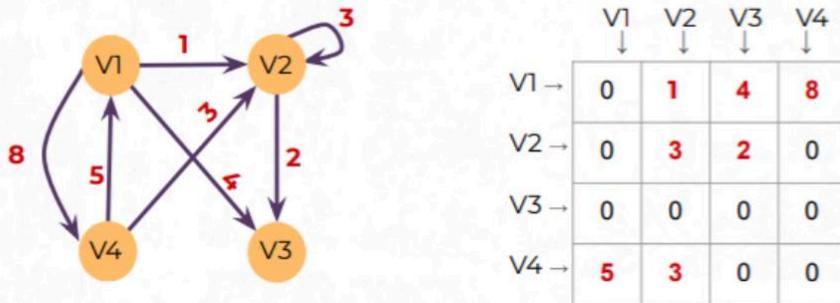
- La matriz de adyacencia para el grafo no dirigido es siempre simétrica.



- En un grafo dirigido la matriz de adyacencia solo registra el vértice de inicio y al vértice final,  $A[i][j] = 1$ , si y solo si la arista parte del vértice i hacia el vértice j:



- Si el grafo tiene pesos, completo con el peso



## Ventajas

Las operaciones básicas como agregar una arista, eliminar una arista o verificar si hay una arista desde un vértice  $i$  a un vértice  $j$  son operaciones de tiempo constante  $O(1)$ .

Si el grafo es **denso**, la matriz de adyacencia debe ser la primera opción. La representación es más fácil de implementar y seguir.

## Desventajas

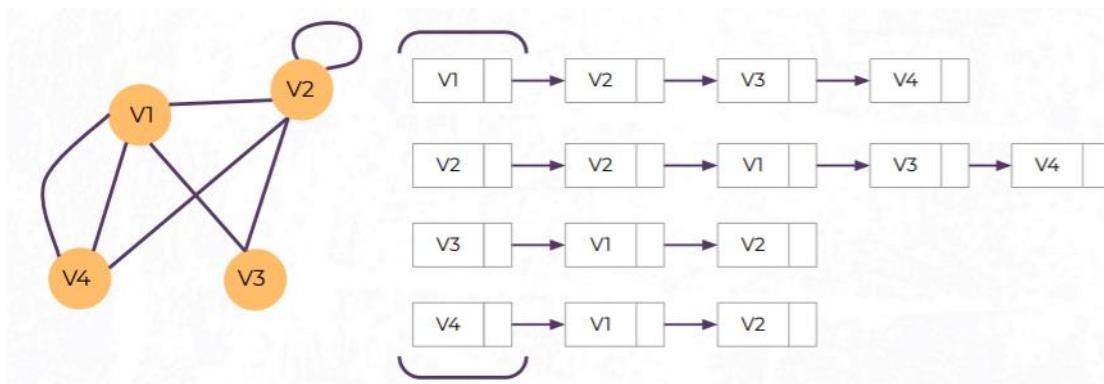
Se utiliza mucho espacio ya que hay que reservar espacio para cada enlace posible entre todos los vértices.

### ii. Lista de adyacencia

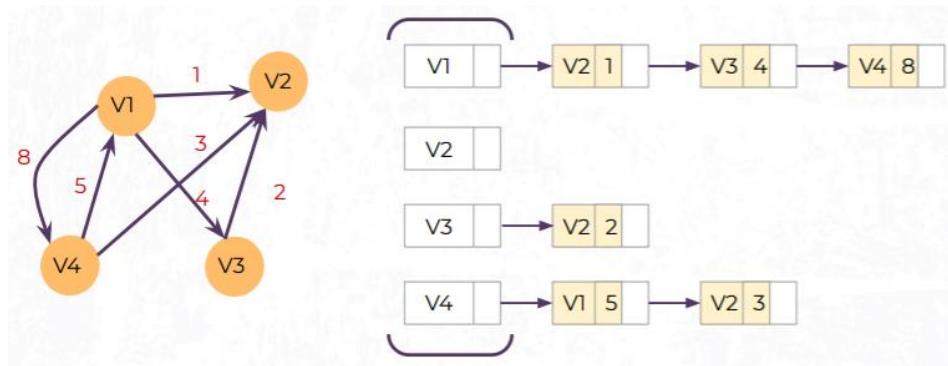
Cada arista posee una lista simplemente enlazada de a qué vértices está unida.

Una lista de adyacencia representa un grafo como un arreglo de listas vinculadas (ese arreglo, en vez de ser un arreglo, también puede ser otra lista y cada nodo de la misma, salen otras listas).

El tamaño del arreglo es igual al números de vértices. El índice del arreglo representa un vértice y cada elemento en su lista enlazada representa los otros vértices que forman una arista con el vértice.



Si el grafo es dirigido, represento las direcciones. Si tienen pesos las aristas, agrego ese dato:



## Ventajas

Es eficiente en términos de almacenamiento porque solo se necesitan almacenar los valores para las aristas.

Usa espacio  $O(|V| + |E|)$

## Desventajas

Las consultas si hay una arista de vértice  $u$  a vértice  $v$  no son eficientes y pueden ser  $O(n)$

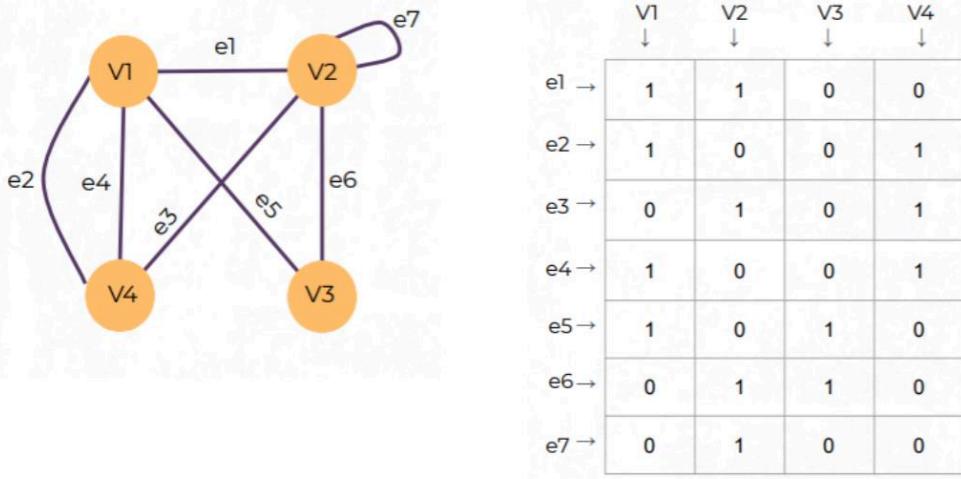
### iii. Matriz de incidencia

Necesito identificar las aristas

Esta matriz de  $Z^{m \times n}$  donde  $m$  es la cantidad de aristas y  $n$  la cantidad de vértices.

Para cada arista identificamos que nodos inciden con él y lo marcamos en el grafo no dirigido con un 1. Si el grafo a representar es dirigido: se le asigna -1 al vértice de salida y 1 al vértice de entrada.

Si el grafo fuera dirigido y con peso, en vez de utilizar el número 1 se utiliza el número del peso de cada arista.



## C. RECORRIDOS

Proceso de búsqueda o forma de atravesar la estructura de datos de un grafo que involucra la visita de cada vértice en el grafo. El orden en el cual los vértices son visitados es la forma en que se clasifica el tipo de recorrido.

- **Visitar:** marcar como visitado el nodo.
- **Explorar:** política en la que vamos a definir cómo se comportará el algoritmo (explorar los vecinos, hijos, etc.)

Existen dos grandes recorridos: en anchura o en profundidad.

### Recorrido en profundidad DFS

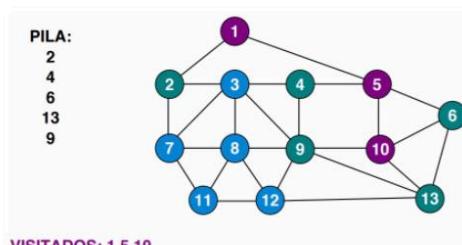
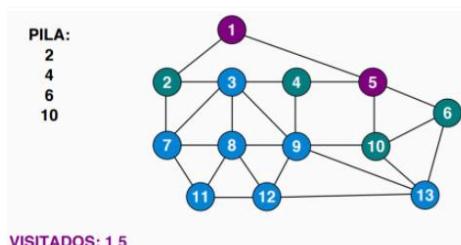
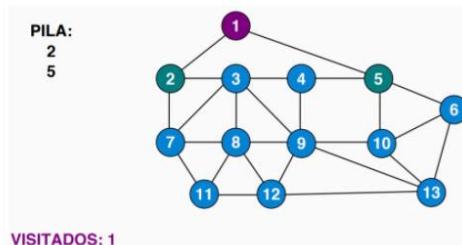
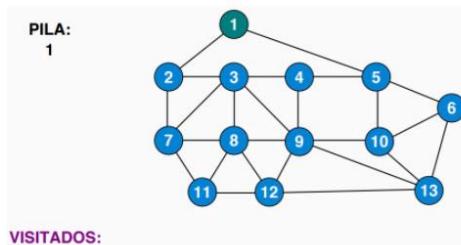
En inglés significa "Depth First Search"

Se visitan los nodos hijos primero, avanzando hasta que no se pueda continuar. Luego se "vuelve" hasta un hijo donde se tenían más caminos y se vuelve a realizar la misma lógica.

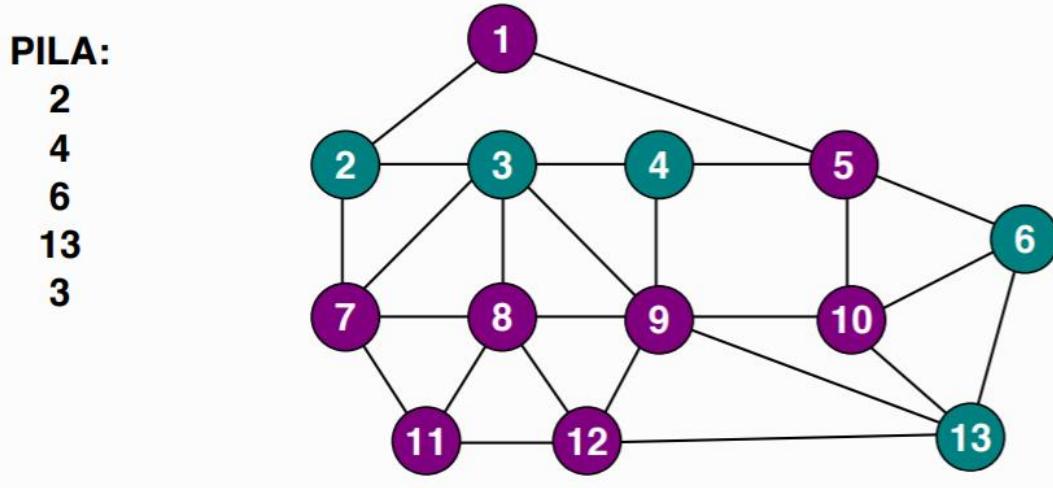
El recorrido consiste en ir recorriendo el grafo empezando desde un vértice cualquiera y, a cada paso, se visita un vértice adyacente al último visitado (se recorre lo más posible). Se continua el bucle hasta que no haya más vértices por visitar. Luego retrocede, se verifica si hay otros sin visitar y los recorre.

### Mejor forma de pensarla es con una PILA

1. Apilar un vértice
2. Quitar un vértice de la pila, visitarlo
3. Apilar los vértices adyacentes al actual
4. Repetir desde 3 hasta quedarse sin vértices



Cuando ya no me quedan mas vértices por visitar, desapilo todos los elementos de la pila y los visito.



RECORRIDO DFS EN PYTHON:

```
def DFS(grafo, vertice_inicial):
    visitados = [vertice_inicial]
    pila = [vertice_inicial]

    while len(pila) > 0:
        vertice = pila.pop() #desapilo 1
```

```
print(vertice) #lo imprimo por pantalla
```

```
for ady in grafo.adyacencias(vertice):
```

```
    if ady not in visitados:
```

```
        pila.append(ady)
```

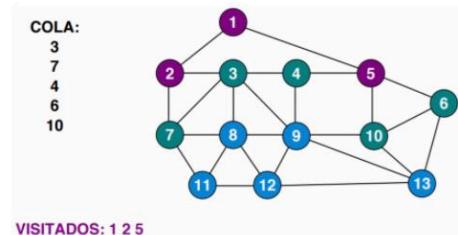
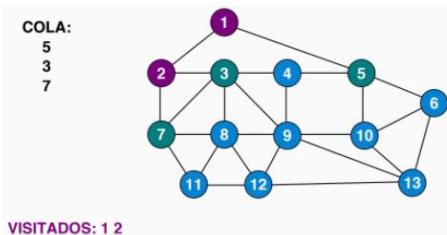
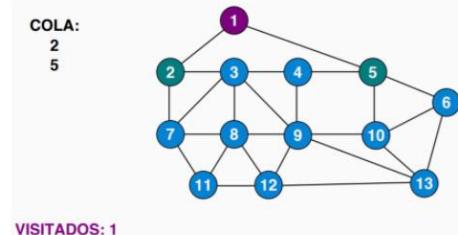
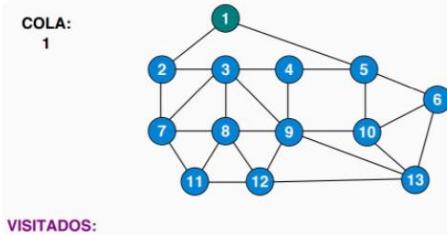
```
        visitados.append(ady)
```

## Recorrido en ancho BFS

El recorrido a lo ancho consiste en ir recorriendo el grafo empezando desde un vértice cualquiera y luego se van visitando los vértices adyacentes más cercanos.

**Mejor forma de pensarlo es con una COLA:**

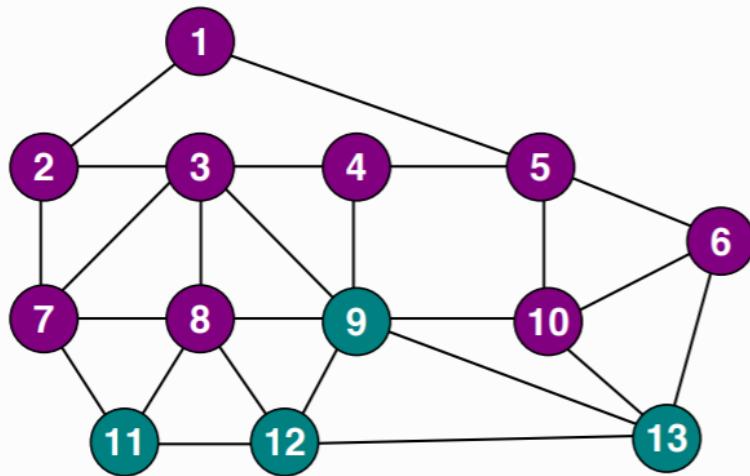
1. Encolar un vértice
2. Quitar un vértice de la cola y visitarlo.
3. Encolar los vértices adyacentes al actual
4. Repetir desde 2 hasta quedarse sin vértices



Cuando ya me quedaron todos los vértices encolados, los desencolo y visito.

**COLA:**

9  
11  
13  
12



**VISITADOS: 1 2 5 3 7 4 6 10 8**

RECORRIDO BFS EN PYTHON:

```
def BFS(grafo, vertice_inicial):  
  
    visitados = [vertice_inicial]  
  
    cola = [vertice_inicial]  
  
    while len(cola) > 0:  
  
        vertice = cola.pop(0) #saco desde el principio de la cola  
  
        print(vertice) #lo imprimo por pantalla  
  
        for ady in grafo.adyacencias(vertice):  
  
            if ady not in visitados:  
  
                cola.append(ady)  
  
                visitados.append(ady)
```

## D. ORDEN TOPOLOGICO

La idea del orden topológico es la de procesar los vértices de una grafo **acíclico** de forma tal que si el grafo contiene la arista dirigida  $uv$  entonces el nodo  $u$  aparece antes del nodo  $v$

## 14. ALGOS CON GRAFOS

[https://servicios.algoritmos7541mendez.com.ar/clases/11\\_-\\_diapos\\_grafos.pdf](https://servicios.algoritmos7541mendez.com.ar/clases/11_-_diapos_grafos.pdf)

### ALGORITMOS PARA ENCONTRAR EL CAMINO MAS CORTO

#### A. DIJKSTRA

El algoritmo de Dijkstra es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices de un grafo que tiene peso en cada arista.

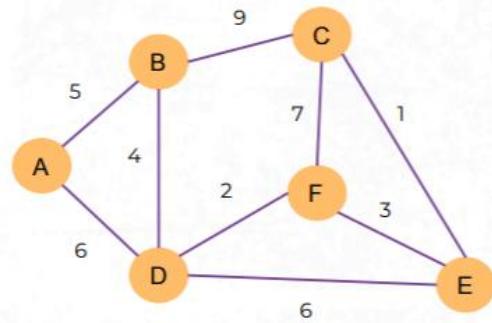
La idea consiste en ir exportando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices.

Cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene.

##### i. Pasos

1. Elegir un vértice para comenzar.
2. Se crean dos listas de nodos, una lista de nodos Visitados y otra de nodos no visitados, que contiene a todos los nodos del grafo.
3. Se crea una tabla con 3 columnas: vértice, distancia mínima V y el nodo anterior por el cual se llegó.
4. Se toma el vértice V como vértice inicial y se calcula su distancia a sí mismo, que es 0 y la del resto de los vértices la inicializamos en  $\infty$ .
5. Se visita el vértice NO VISITADO con menor distancia conocida desde el primer vértice V, que es el vértice con el que comenzamos ya que la distancia a ese es 0 y las demás infinito.
6. Se calcula la distancia entre los vértices sumando los pesos de cada uno con la distancia de V.
7. Si la distancia calculada de los vértices conocidos es menor a la que está en la tabla se actualiza y también los vértices desde donde se llegó.
8. Se pasa el vértice V a la lista de Vértices visitados y se continua con el vértice no visitado con menor distancia desde ese.

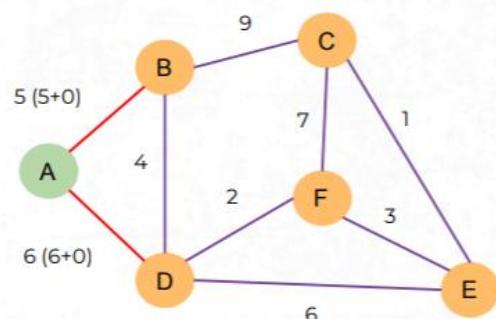
Vértice	Distancia	V. Anterior
A	0	-
B	$\infty$	-
C	$\infty$	-
D	$\infty$	-
E	$\infty$	-
F	$\infty$	-



Visitados: []

No visitados: [A,B,C,D,E,F]

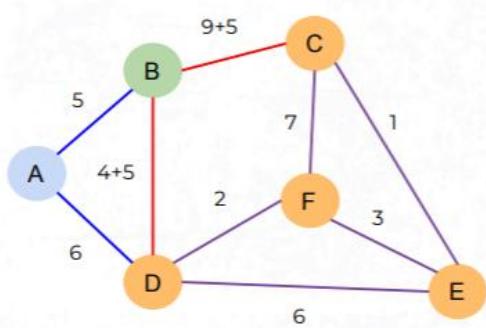
Vértice	Distancia	V. Anterior
A	0	-
B	5	A
C	$\infty$	-
D	6	A
E	$\infty$	-
F	$\infty$	-



Visitados: [A]

No visitados: [B,C,D,E,F]

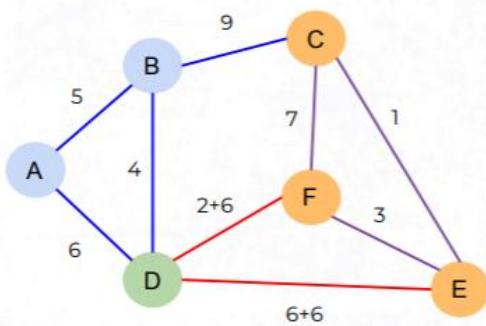
Vértice	Distancia	V. Anterior
<b>A</b>	0	-
<b>B</b>	5	A
C	14	B
D	6	A
E	$\infty$	-
F	$\infty$	-



Visitados: [A,B]

No visitados: [C,D,E,F]

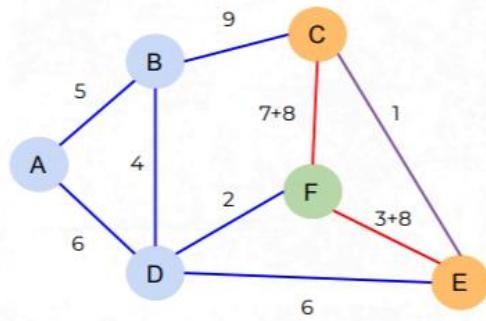
Vértice	Distancia	V. Anterior
<b>A</b>	0	-
<b>B</b>	5	A
C	14	B
<b>D</b>	6	A
E	12	D
F	8	D



Visitados: [A,B,D]

No visitados: [C,E,F]

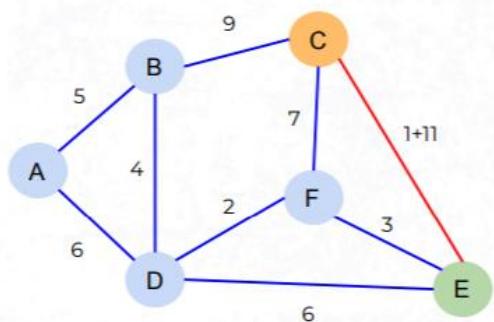
Vértice	Distancia	V. Anterior
<b>A</b>	0	-
<b>B</b>	5	A
C	14	B
<b>D</b>	6	A
E	<b>11</b>	<b>F</b>
<b>F</b>	8	D



Visitados: [A,B,D,F]

No visitados: [C,E]

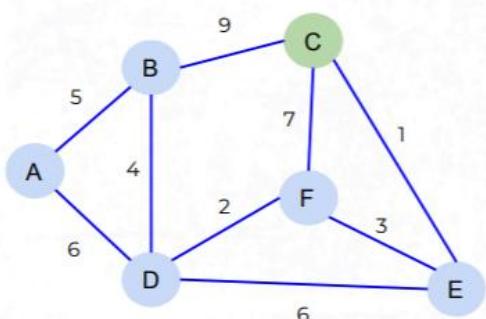
Vértice	Distancia	V. Anterior
<b>A</b>	0	-
<b>B</b>	5	A
C	<b>12</b>	<b>E</b>
<b>D</b>	6	A
<b>E</b>	11	F
<b>F</b>	8	D



Visitados: [A,B,D,F,E]

No visitados: [C]

Vértice	Distancia	V. Anterior
<b>A</b>	0	-
<b>B</b>	5	A
<b>C</b>	12	E
<b>D</b>	6	A
<b>E</b>	11	F
<b>F</b>	8	D

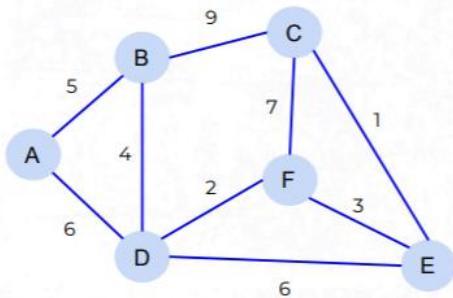


Visitados: [A,B,D,F,E,C]

No visitados: []



Vértice	Camino Minimo	Costo
A → B	A → B	5
A → C	A → D → F → E → C	12
A → D	A → D	6
A → E	A → D → F → E	11
A → F	A → D → F	8



```

def dij(grafo, vertice_inicial):
    tabla = {}
    visitados = []

    for v in grafo.vertices():
        tabla[v] = {}
        tabla[v]["distancia"] = math.inf
        tabla[v]["anterior"] = None

    tabla[vertice_inicial]["distancia"] = 0

    def minimo_no_visitado():
        no_visitados = [v for v in grafo.vertices() if v not
in visitados]
        #armo un listado de todos los no visitados

        menor = None
        distancia = math.inf
        for nv in no_visitados:
            if tabla[nv]["distancia"] < distancia:
                distancia = tabla[nv]["distancia"]
                menor = nv
        return menor

    while vertice := minimo_no_visitado():

        visitados.append(vertice)
        distancia_actual = tabla[vertice]["distancia"]

        #me fijo con que distancia llegue a ese vertice y despues de los
        #vecinos

        for ady in grafo.adyacencias(vertice):
            nueva_distancia = distancia_actual +
grafo.distancia(vertice,ady)
            if nueva_distancia < tabla[ady]["distancia"]:
                tabla[ady]["distancia"] =
nueva_distancia
                tabla[ady]["anterior"] = vertice

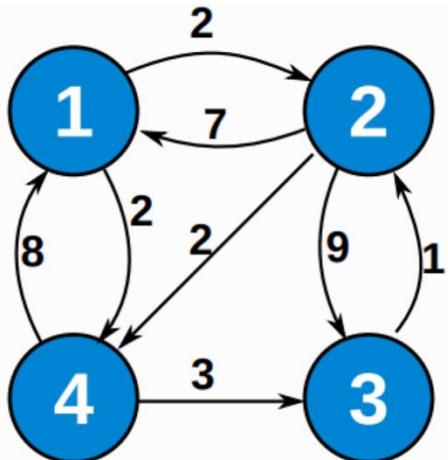
    return tabla
  
```

## B. FLOYD-WARSHALL

Calcula el camino mínimo entre todos los pares de vértices. La implementación son 3 for anidados

Ejemplo: yo sé que de 1 a 3 es infinito (no tengo un camino directo), entonces es más barato:

- ir de 1 a 3 directo
- ir de 1 a 4 a 3
- ir de 1 a 2 a 3



Armo una matriz de distancias. Tomo un vértice  $v_0$  y otro par de vértices  $v_1$  y  $v_2$ . Me quedo con la menor distancia entre:

- $v_1 \mapsto v_2$
- $v_1 \mapsto v_0 \mapsto v_2$

Por ejemplo, tomo  $v_0 = 1, v_1 = 2, v_2 = 3$

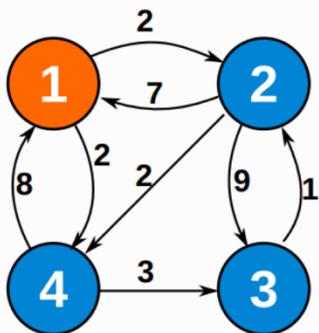
y me quedaría:

- $2 \mapsto 3 = 9$
- $2 \mapsto 1 \mapsto 3 = \text{infinito}$

Me quedo que ir directo de 2 hacia 3 es más corto

	1	2	3	4
1	0	2	$\infty$	2
2	7	0	9	2
3	$\infty$	1	0	$\infty$
4	8	$\infty$	3	0

Si los tomo de a pares a partir del 1, el 2,3,4 se van a modificar. Siempre me tengo que fijar en la matriz



	1	2	3	4
1	0	2	$\infty$	2
2	7	0	9	2
3	$\infty$	1	0	$\infty$
4	8	$\infty$	3	0

COMO 1 ES NUESTRO VERTICE INTERMEDIO, NO ME INTERESAN LAS ARISTAS QUE SALEN O LLEGAN DE EL.

	1	2	3	4
1	0	2	$\infty$	2
2	7	0	9	2
3	$\infty$	1	0	$\infty$
4	8	$\infty$	3	0

	1	2	3	4
1	0	2	$\infty$	2
2	7	0	9	2
3	$\infty$	1	0	$\infty$
4	8	$\infty$	3	0

?  $2 \Rightarrow 1 \Rightarrow 3 < 2 \Rightarrow 3?$

?  $2 \Rightarrow 1 \Rightarrow 4 < 2 \Rightarrow 4?$

	1	2	3	4
1	0	2	$\infty$	2
2	7	0	9	2
3	$\infty$	1	0	$\infty$
4	8	$\infty$	3	0

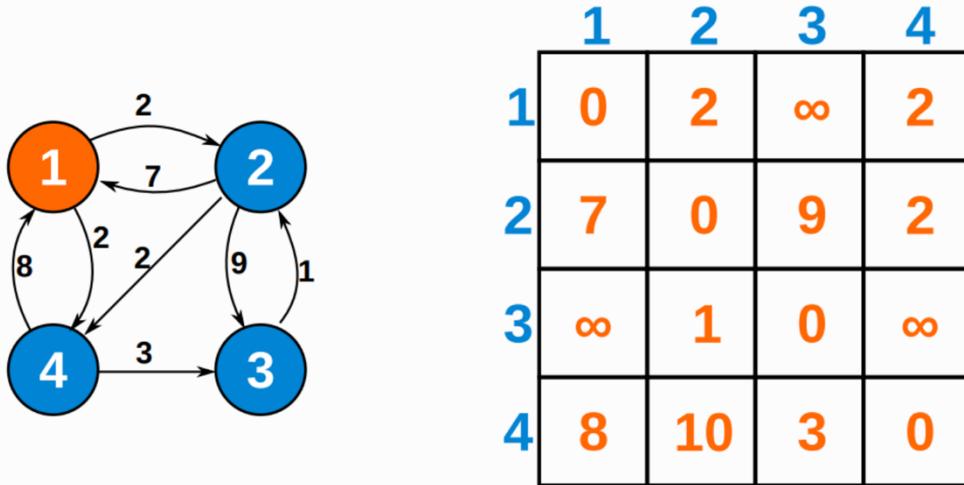
	1	2	3	4
1	0	2	$\infty$	2
2	7	0	9	2
3	$\infty$	1	0	$\infty$
4	8	$\infty$	3	0

$\dot{c}3 \Rightarrow 1 \Rightarrow 2 < 3 \Rightarrow 2?$

$\dot{c}3 \Rightarrow 1 \Rightarrow 4 < 3 \Rightarrow 4?$

	1	2	3	4
1	0	2	$\infty$	2
2	7	0	9	2
3	$\infty$	1	0	$\infty$
4	8	$\infty$	3	0

$\dot{c}4 \Rightarrow 1 \Rightarrow 2 < 4 \Rightarrow 2?$



## ARBOL DE EXPANSION MINIMO

### C. PRIM

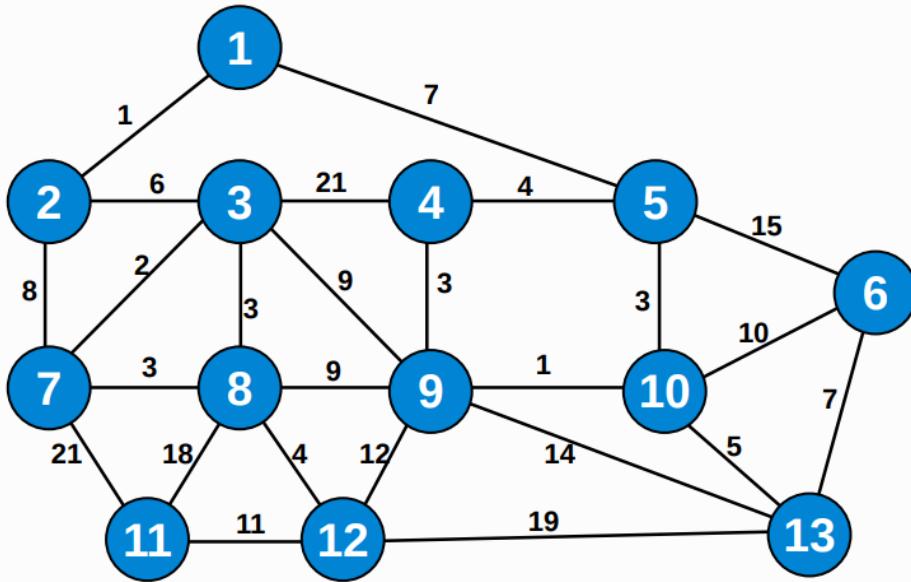
Un árbol (spanning tree) → grafo sin ciclos y conexo

Este algoritmo busca el spanning tree mínimo: al recorrer los pesos de las aristas, tenga que gastar lo menor posible para llegar de un vértice a otro. Quiero tener todos conectados entre sí con el menor peso posible.

- Forman un árbol que incluya todos los vértices
- Tiene la suma mínima de pesos entre todos los árboles que se pueden formar a partir del gráfo.

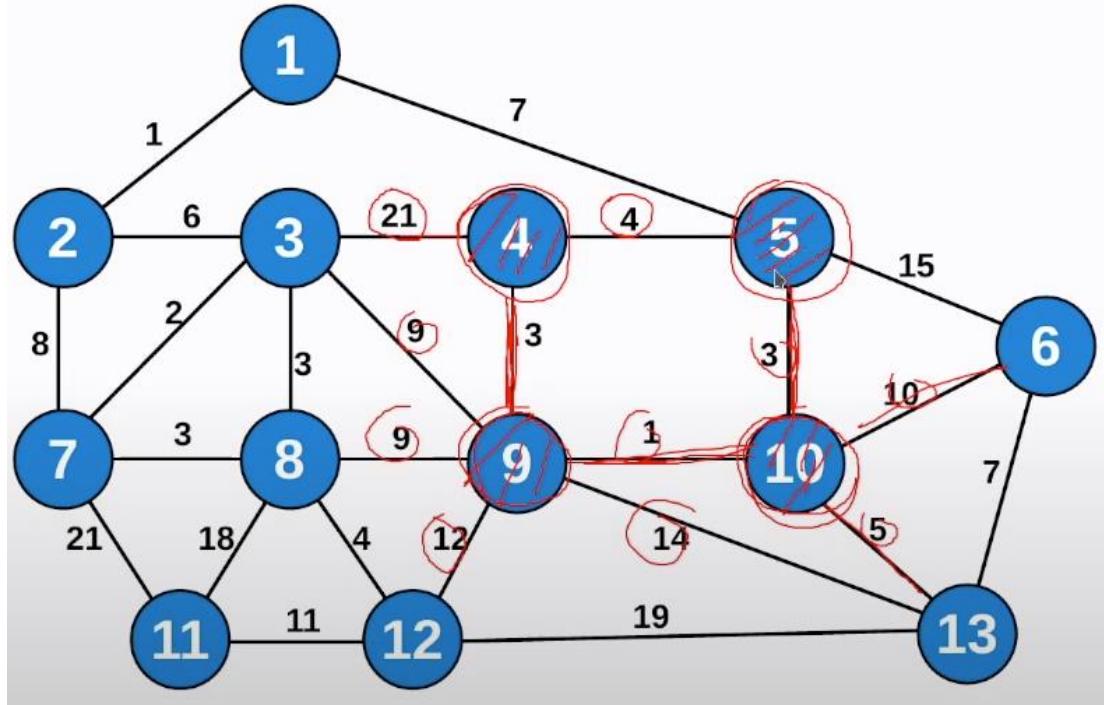
Los pasos para implementar el algoritmo de Prim son los siguientes:

1. Inicializar el árbol de expansión mínimo con un vértice elegido al azar.
2. Encontrar todas las aristas que conectan el vértice a nuevos vértices, elegir la mínima y agregarla al árbol. A medida que se agregan vértices al árbol se amplia la cantidad de aristas que podemos elegir, la única condición es que esa arista no lleve a un vértice ya agregado al árbol.
3. Seguir repitiendo el paso 2 hasta que obtengamos un árbol de expansión mínimo.



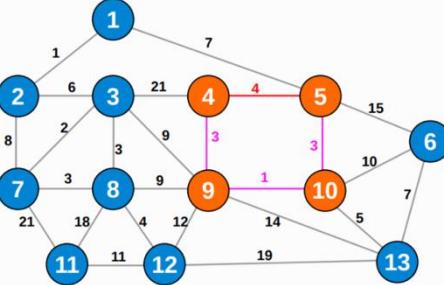
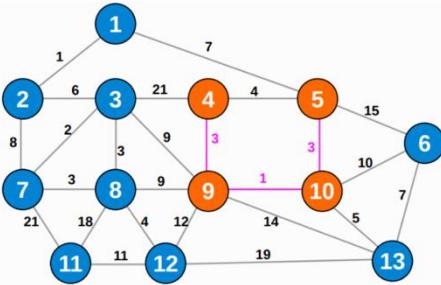
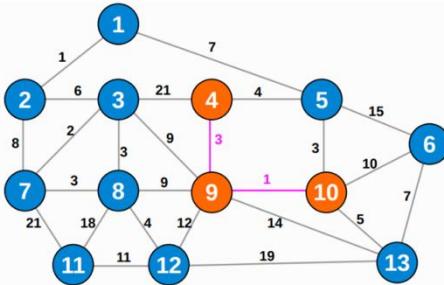
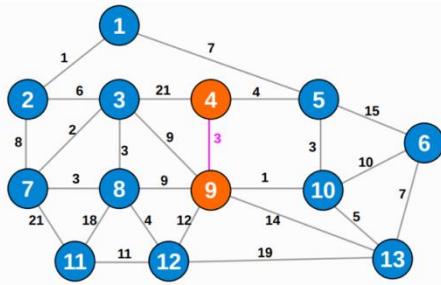
COMENZANDO POR UN VERTICE ARBITRARIO, MARCARLO COMO VISITADO E IR AGREGANDO LA ARISTA DE MENOR PESO QUE CONECTA UN VERTICE YA VISITADO CON UNO NO VISITADO.

Voy agregando al spanning tree un vértice a la vez. Cuando me quedo con la arista mínima, osea me quedo con el vértice también y repito. Empecé por el 4, me conecte al 9 después al 10 y por ultimo al 5 (esto sigue pero a modo de ejemplo). Después la mínima sería 4 (de 5 a 4) pero ya esta conectado entonces eso no lo puedo hacer.

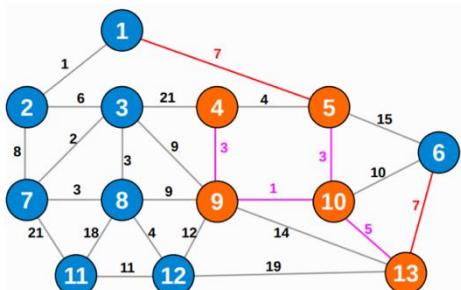
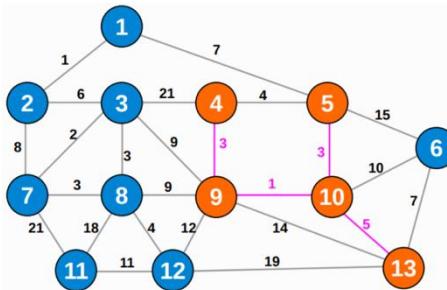


Ahora veamoslo más lindo. Lo que pasa es que empiezo con un vértice y hago crecer mi árbol agregando aristas (todas las posibles son las que tengo un vértice en mi

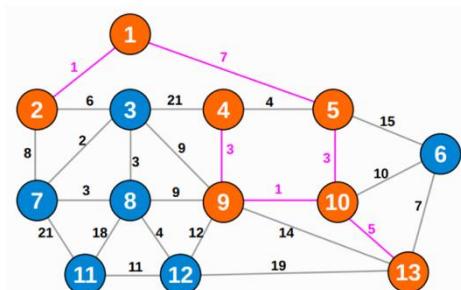
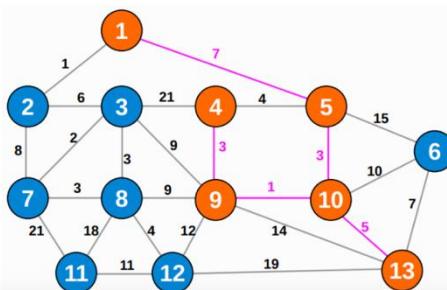
árbol)



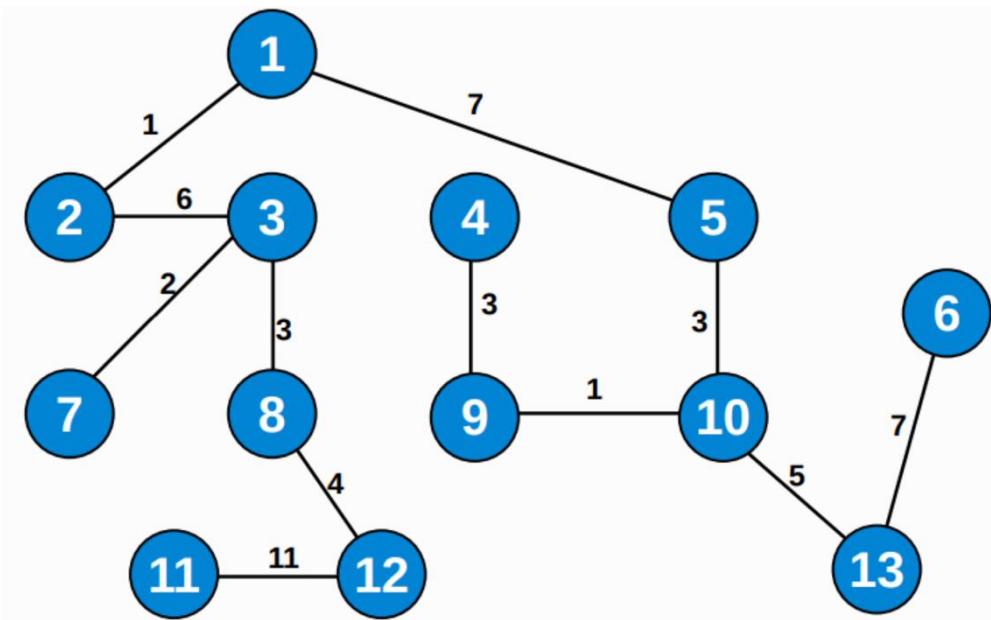
Esta arista que me genera un ciclo no la agrego, sino que agrego la que sigue menor



Tenía dos con peso 7 pero eso lo defino en mi algoritmo, para donde ir (es indiferente)



En síntesis, mi árbol queda así

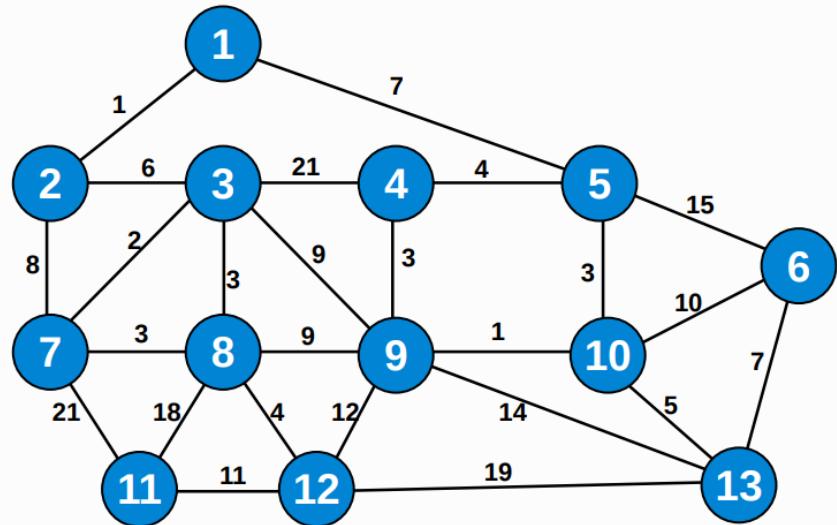


#### D. KRUNSKAL

En lugar de comenzar desde un vértice, el algoritmo de Kruskal ordena todos las aristas de bajo peso a alto y sigue agregando las aristas más bajas, ignorando aquellas aristas que crean un ciclo.

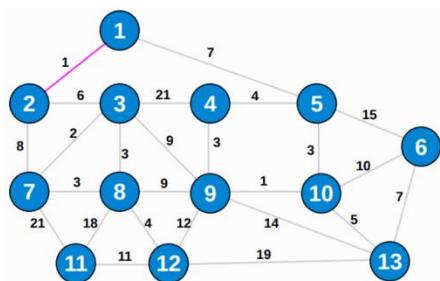
Se parte de las aristas con menor peso y se siguen agregando aristas hasta llegar al objetivo. Los pasos para implementar el algoritmo de Kruskal son los siguientes:

1. Ordenar todos las aristas de bajo peso a alto.
2. Tomar la arista con el peso más bajo y agregarlo al árbol de expansión. Si agregar la arista crea un ciclo, rechazar esta arista.
3. Seguir agregando aristas hasta llegar a todos los vértices.

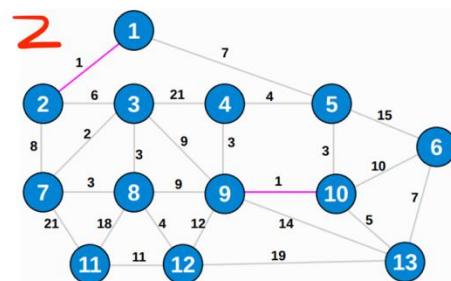


CREO UN ARBOL SEPARADO PARA CADA VERTICE.  
VOY TOMANDO CADA ARISTA DE FORMA ASCENDENTE Y ME QUEDO CON LAS QUE UNEN 2 ARBOLES.

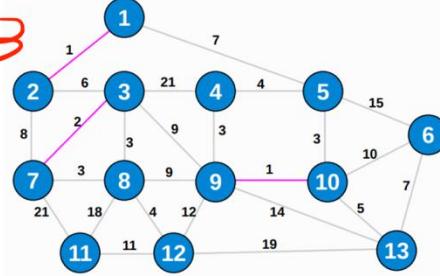
1



2



3



4

