* Introducción a la programación en Java.

Java es un lenguaje de programación orientado a objetos (POO).

- Clase: Es un molde para definir una entidad.

Una clase consta de:

- Atributos: Define las características de la entidad a definir.
- Métodos: Las operaciones que realiza la clase.
- Constructores: Su objetivo es inicializar al objeto que se generará cuando se instancia una clase.

Una vez definida la clase, podremos crear objetos.
Un **objeto** hace referencia a una entidad que existe dentro de la memoria, con identidad propia y que es diferente a otras instancias (de la misma o diferentes clases)

Para definir una clase utilizamos la palabra reservada «class».
Los nombres de las clases siempre empiezan con mayúscula.
En primer lugar se define el nivel de privacidad de una clase.

- Public: Permite que cualquier objeto pueda acceder a cualquier elemento.
- Protected: Solamente podemos acceder si estamos dentro de la clase y sus hijos. (Herencia)
- Private: Solamente podemos acceder si estamos dentro de la clase.

Clase: public class Persona { }

Para introducir los atributos de la clase usaremos la estructura:

<tipo de acceso> <tipo de modificador> <tipo de dato> <nombre>;

- Tipo de acceso: define quien puede acceder a los atributos.
- Tipo de modificador: define restricciones sobre el atributo: estático (static), constante (final)... (Parámetro opcional).
- Tipo de dato: Indica el dato que va a contener (int, float...).
- Nombre: Cualquiera que no sea una palabra reservada.

Para los nombres de métodos, atributos y variables, lowerCamelCase
Para los nombres de las clases, UpperCamelCase.

Atributo: private string nombre;

Los atributos se definen dentro de las llaves de la clase.

```
public class Persona{
    private string nombre;
    private int edad
}
```

Los métodos son las instrucciones que estan definidas dentro de una clase. Estan programados mediante algoritmos para realizar una tarea concreta y se les puede llamar a traves de su nombre. Su sintaxis es:

```
<tipo de acceso><tipo de retorno><nombre> (<argumentos opcionales>){
    [implementación del método]
    <retorno (si aplica)>
}
```

- Tipo de acceso: public, protected o private.
- Tipo de retorno: tipo de dato que va a devolver (int, char...)
- Argumentos: parametros que puede recibir. Se debe definir el tipo y el nombre de la variable.
- Retorno: valor que devuelve el método (siempre y cuando el tipo de retorno no sea void).

```
public class Persona{
    private string nombre;
    private int edad;

    public float andar (int distancia){
     //aqui irea la implementación del algoritmo
      return float;
    }
    private void digerir() {
     //implementación
    }
}
```

Los constructores tienen como objetivo inicializar al objeto que se generará cuando se instancia una clase.
En Java existe lo que se conoce como constructor por defecto, es el que no recibe ningún parametro y existe siempre que no se haya creado ninguno.
Si creamos un constructor, el constructor por defecto desaparece, y para volver a optar por el, debe crearse de forma explícita.
Su sintaxis es:

<tipo de acceso> <nombre de clase>(<argumentos>)

El constructor no devuelve nada.
Se pueden crear tantos constructores como queramos, siempre que no coincidan en los tipos recibidos.

```
public Persona (String n){
    nombre=n;
}
```

Los getters son metodos que nos devuelven un valor cuando lo solicitemos a una clase con atributos privados.

```
public String getNombre (){
    return nombre;
}
```

Los setters sirven para controlar el valor de nuestros atributos.

```
public void setNombre (string n){
    nombre = n;
}
```

Una clase quedaria definida:

```
public class Persona{
    //atributos
    //constructores
    //getters
    //setters
    //resto de métodos
}
```

- Objeto: es una instancia a una clase. Un objeto permite dar vida a lo que la clase representa.
Cuando se instancia una clase y creamos un objeto, tenemos acceso a todos los metodos y atributos publicos de la clase y podremos trabajar con diferentes objetos.
Para instanciar un objeto se debe usar la palabra reservada new. Su sintaxis es:
<nombre de clase><nombre de objeto> = new <nombre clase>(argum)
Por ejemplo, vamos a crear un objeto llamado p1, instanciando a la clase persona desde la clase principal.

```
public class Principal{
    Persona p1 = new Persona (Juan);
} //Ahora p1 en el atributo nombre se llama Juan.
```

- Variable: Espacio de memoria que se reserva para almacenar un valor. En Java pueden ser de dos categorias:
  - Datos primitivos: Δ los tipos a los que se accede al valor directamente (int, float, string...)
  - Referenciadas: No almacenan un valor concreto, sino un espacio en memoria, y en esa dirección de memoria es donde accedemos a los datos que componen la variable (un objeto de una clase).

  Siempre se debe tipar el tipo de variable:

  ```
  int numero = Ø;
  ```

El new crea una instancia a una clase asignando la cantidad de memoria necesaria deacuerdo al tipo de objeto.
El operador new se utiliza en conjunto con un constructor.

- Expresiones condicionales y bucles:

· Condicionales:

== igual a

!= distinto de

\> mayor que

< menor que

\>= mayor o igual que

<= menor o igual que

&& es equivalente a AND, si y solo si todas las condiciones se cumplen.

|| es equivalente a OR, cuando se cumpla al menos una condición.

If si la condición es evaluada como verdadera se ejecuta, en caso contrario ejecutara else.

```
If (n != 0) {
  System.out.println(n);
} else {
  System.out.println("su valor es" + n);
}
```

· Bucles: Sentencia que se ejecuta de forma repetida hasta que una una condición se cumple.

for: número concreto de iteraciones.

Sintaxis: for <tipo><variable contadora>=<valor inicial>; <condicion>; <incremento>

```
for (int i = 0; i < 10; i++){
    System.out.println("Hola mundo")
} // Imprimira "Hola mundo" 10 veces.
```

while: número de iteraciones controladas por condición.

Sintaxis: while (<condición>)

```
i = 0
while (i < 10) {
    System.out.println ("Hola mundo");
    i++;
} // Imprimira "Hola mundo" 10 veces.
```

do-while realiza al menos una iteración y se controla por condición.

Sintaxis
```
do{
  //codigo
}while (<condición>);
```

```
i=11
do{
  Sistem.out.println(i);
}while (i<10);
```

//imprimira 11, esto es porque ejecuta una vez el código y despues entra ya en el while.

*Tipos Abstractos de Datos (TAD o TDA):

Un tipo de dato define un conjunto de valores y las operaciones sobre estos valores.

- Tipos de datos primitivos: Son aquellos a los que se accede al valor directamente.

TIPO	REPRESENTACIÓN
Caracteres	char
Númericos (Enteros)	byte short int long
Númericos (reales)	float double
Booleanos	boolean

- Tipos de datos compuestos: Son aquellos que derivan de más de un tipo primitivo. Por ejemplo, string es la combinación de varios char, una clase da lugar al concepto objeto, y a su vez, al tipo de dato abstracto.

El objetivo de un TAD es poder representar un nuevo tipo de dato de forma abstracta.

Estructura de datos: Es una forma particular de almacenar y organizar datos de tal forma que estos puedan ser usados de forma eficiente. En resumen, un elemento que nos permite agrupar colecciones de datos.

Arrays: Zonas de almacenamiento continuo que contienen una serie de elementos del mismo tipo.

Hay dos clases de arrays (arreglos):

-Vectores: Son arrays de una sola dimensión, por ejemplo, una lista de cosas:

Avión	Barco	Coche	Tren	Moto

La forma de crear un vector en Java es:

<tipo><nombre de variable>[ ] = new <tipo>[<tamaño>]

Por ejemplo, para crear un array que contenga 5 numeros enteros.

int numeros [ ] = new int[5];

Tambien se le pueden asignar valores a un array directamente.

int numeros[ ] = {1, 7, 3, 8, 2};

Si queremos acceder a la posición de un array:

int num = numeros [2] //asignamos a "num" el valor "3".

Si queremos remplazar el valor de la posición 4 del array:

numeros [4] = 6 // el array resultante sera {1, 7, 3, 8, 6}

-Arrays n-dimensionales: Un array puede tener tantas dimensiones como queramos.

-Arrays bidimensionales o matrices: Son array de dos dimensiones.

Si queremos crear una matriz de 4x3 vacia:

char matriz [ ] [ ] = new char [4][3];

Asignar valores directamente en la matriz:

char matriz [ ] [ ] = {{'A','B','C'}, {'D','E','F'}, {'G','H','I'}, {'J','K','L'}};

Se divide en llaves cada fila y dentro de cada fila se introducen los valores.

Para acceder a un dato de la matriz o modificarlo, lo haremos de la misma manera que con los vectores.

char letra = matriz [2] [1];

Fila ↑ ↑ Columna.

*Estructuras de datos lineales:

- TAD Lista: Son secuencias de nodos que permiten almacenar datos y que contienen un determinado número de referencias con punteros a los nodos posteriores y/o previos.

Las listas son estructuras de datos homogeneas, es decir, todos sus datos son del mismo tipo.

Son contenedores de datos dinamicos, es decir, que pueden crecer y decrecer de forma "infinita".

Las operaciónes mas habituales de las listas son:

- Inicializar(): Genera una nueva lista vacía.
  Si se inicializa una lista que contenga datos, los datos son suprimidos para dar lugar a la nueva lista.
- Insertar(elemento): Inserta el elemento en la lista.
- Borrar(elemento): Borra el elemento que se le pase como parametro.
- Borrar(posición): Borra el elemento que se encuentre en la posición recibida.
- Obtener(posición): Devuelve el elemento que se encuentre en la posición recibida.
- Localizar(elemento): Devuelve la posición en la que se encuentra el elemento.
- Tamaño(): Devuelve el tamaño de la lista.

- Listas enlazadas: Se trata de una serie de nodos los cuales contienen una referencia que apunta al siguiente nodo. (puntero)

Como vemos en la imagen, cada nodo contendra un dato y un puntero que apunta al siguiente nodo.

A parte de crear nuestra lista de nodos, necesitamos otros elementos para poder operar totalmente con ella.

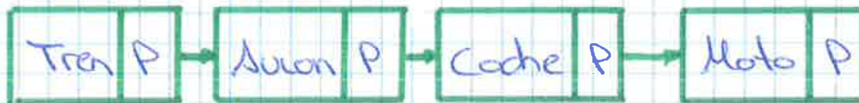
- Puntero al primer elemento
- Puntero al ultimo elemento.

Los punteros son necesarios para saber donde empieza la lista y donde se debe insertar un elemento.

- Añadir un elemento:

Tenemos la siguiente lista y queremos añadir un elemento (moto):

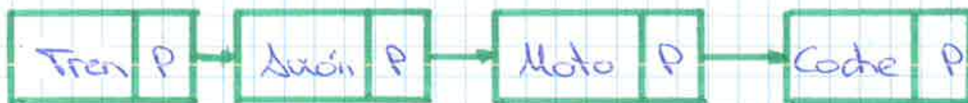
Como tenemos la información del último nodo, solo tendremos que añadirlo a la cola y actualizar "ultimo elemento".



- Insertar un elemento:

Disponemos del puntero "primer nodo" = Tren
Ahora queremos añadir la moto en la posición 2 (coche).
Nuestro algoritmo debe de ser capaz de colocar la moto en la posición 2 y mover el resto de elementos a la derecha, con lo que el coche pasara a la posición 3.
Para ello debemos posicionarnos en la posición anterior a la que queremos insertar.

- Se guarda en un nodo auxiliar el nodo que esta en la posición a modificar.
- En el nodo 1, se le dira que apunte al nuevo (moto)
- En el nodo nuevo le diremos que apunte al nodo auxiliar.



- Borrar un elemento:

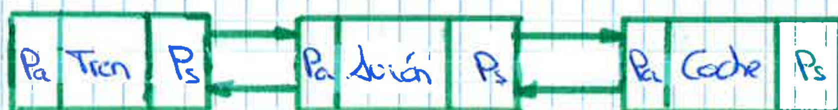
La operación de borrado se puede implementar de varias formas:

- Borrado del ultimo nodo: Si tenemos el puntero que apunta a ultimo nodo, bastara con decir que lo borre y actualizar el puntero "ultimo nodo".
  Si no, deberemos recorrer la lista buscando un nodo A, que apunte a un nodo B, y que este apunte a null. Se borra el nodo B y actualizamos el puntero del nodo A a null.

• Borrado del primer nodo:
Debemos colocarnos al principio de la lista y comprobar si hay algo detrás. Si no hay nada, la lista queda vacía al borrar el nodo. No es problema, pero se comprueba para actualizar los nodos cabeza y cola.
Supongamos nuestra lista con varios nodos.
Creamos un nodo auxiliar que guarde la cabeza actual de la lista (Tren).
Actualizamos el puntero a la cabeza diciendo que es el siguiente (Avion). [primer_nodo = Avion]
Borramos el nodo que contenía la antigua cabeza.

• Borrado de cualquier otro nodo:
Imaginemos que queremos borrar el nodo 1 (Avión).
1. Nos situamos en la cabeza de la lista.
2. Buscamos el siguiente al que queremos borrar y lo guardamos en un auxiliar (auxSiguiente).
3. Buscamos el nodo que queremos borrar y lo guardamos en un auxiliar (auxNodoBorrado).
4. Buscamos el nodo anterior al que queremos borrar y establecemos un enlace entre este y el nodo AuxSiguiente.
5. Borramos el nodo AuxNodoBorrado dandole valor nulo.

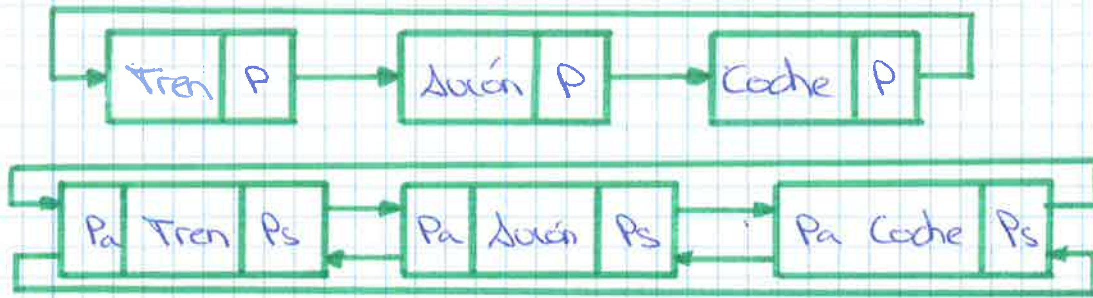
- Listas doblemente enlazadas: Son practicamente como las listas enlazadas, solo que estas añaden tambien un puntero al nodo anterior.



Pa: Puntero nodo anterior
Dato: Es el dato
Ps: Puntero nodo siguiente.

Mientras que en las listas doblemente enlazadas es mas costoso añadir o borrar elementos, las operaciones son mas simples y eficientes, porque no es necesario tener un seguimiento del nodo previo durante el recorrido, o no es necesario recorrer la lista para encontrar el nodo previo a uno dado.
La principal ventaja de este tipo de listas es que permite ser recorrida en ambos sentidos, lo que disminuye su coste computacional.

- Listas circulares: Son una implementación particular de las listas simples y doblemente enlazadas, donde el primer y ultimo nodo se unen generando una estructura cerrada.



*Listas en Java:

Java dispone de varios tipos de implementaciones de listas.

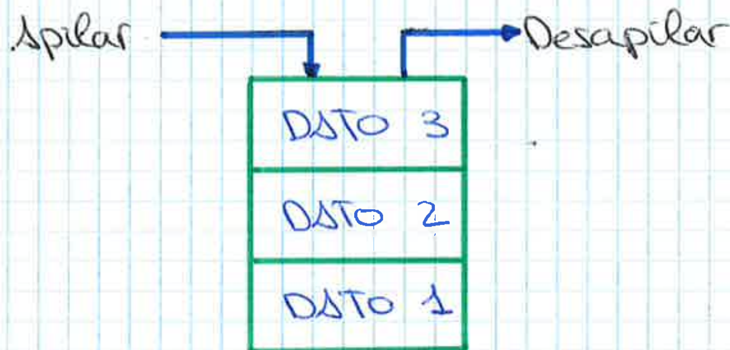
- ArrayList: Esta implementado como un array dinámico, es decir, un array que puede modificar su tamaño.
  Para lograr esto, cuando el array se llena, crea uno nuevo mayor y copia los datos de uno a otro.

- LinkedList: Esta implementado como una lista doblemente enlazada.

Cada una de estas implementaciones tiene sus pros y sus contras, por ejemplo, para operaciones de acceso a un dato, ArrayList es inmediato al comportarse como un array $O(1)$, en cambio LinkedList tiene un coste computacional de $O(n)$.
Por contra, ArrayList tiene que redimensionarse cada vez que necesita mas espacio, y esto computacionalmente lleva tiempo.

*Pilas y colas:

-Pila: Es una estructura de datos del tipo LIFO (Last In First Out), esto quiere decir que, los ultimos elementos en entrar en la estructura de datos seran siempre los primeros en salir.

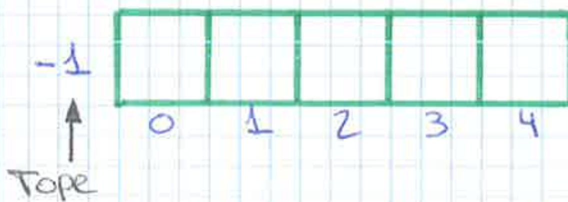


Las pilas pueden ser implementadas como pilas estáticas o pilas dinámicas, en funcion de que establezcamos o no un limite de elementos.
En las pilas estaticas utilizaremos arrays para almacenar los datos mientras que en las pilas dinámicas seguiremos la idea de las listas, donde tendremos nodos con punteros.

Las operaciones básicas de la pila son:

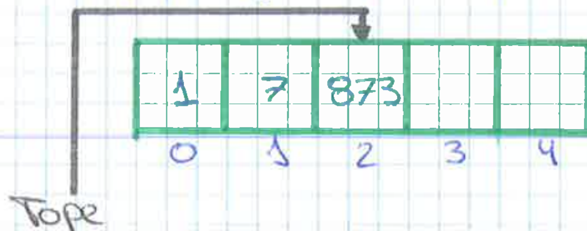
- Inicializar(): genera una pila nueva vacia.
  Si se inicializa una pila que contiene datos, estos son suprimidos.
- Apilar(elemento): Se encarga de añadir el elemento a la pila. Devolvera True o False en función de si se pudo añadir el elemento (pilas estáticas).
- Desapilar(): Obtiene el elemento de la pila y lo borra.
- Vacia(): Devuelve un booleano en función de si la pila esta vacia o no. Esta función deberia ejecutarse antes de realizar un desapilar, ya que si la pila esta vacia, no tiene sentido sacar un elemento que no existe.
- Cima: Obtiene el dato que esta en la cima pero sin borrarlo. Sirve para obtener una referencia al ultimo elemento de la pila.
- Limpiar: Borra completamente la pila, desapilando sus elementos y dejando la pila vacia.

- Pila estática: Almacena los elementos estableciendo previamente un tamaño maximo.
Su implementación vendrá dada por un array.

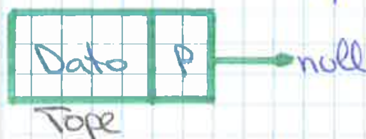


Tendremos un puntero que establece el tope.
El tope es un puntero que define cual es la siguiente posición libre o donde esta la cima (depende del valor con que se inicialice).

A la hora de añadir elementos, solo debemos desplazar el tope y ocupar la posicion que marca.

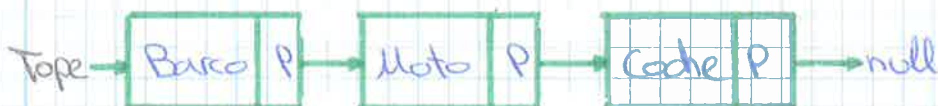


- Pila dinámica: Se construye mediante la consecución de nodos, al igual que las listas, con la diferencia de que no se dispone de un array en el que se introducen elementos.
Al crear la pila, como no tenemos elementos, tendremos un nodo vacio que apunta a null.



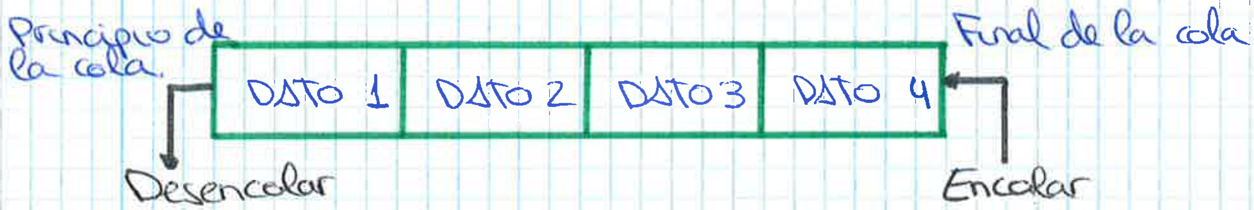
Cuando insertemos un dato, tope sera un nodo con valor y apuntara a null.

A medida que vamos añadiendo datos, se van insertando delante



La operación desapilar devuelve el elemento al que apunta tope y actualiza la referencia.

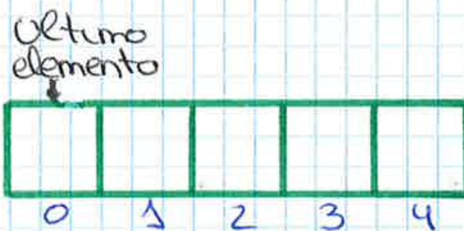
- Cola: Es una estructura de datos del tipo FIFO (First in First out), el primer elemento que entra es el primero en salir.



Los operadores basicos de la cola son:

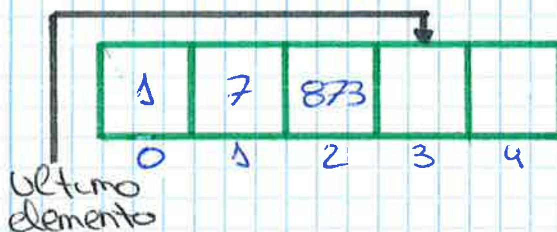
- Inicializar(): Corresponde al proceso que inicia la cola.
- Encolar(elemento): Añade un elemento a la cola. Devuelve un True o False indicando si se añadio correctamente.
- Desencolar(): Obtiene el primer elemento de la cola y lo borra
- Primero(): Devuelve el primer elemento de la cola sin borrarlo.
- Vacia(): Comprueba si la cola esta vacia.

- Cola estática: Almacena los elementos estableciendo previamente un tamaño máximo.
Su implementación viene dada por un array.

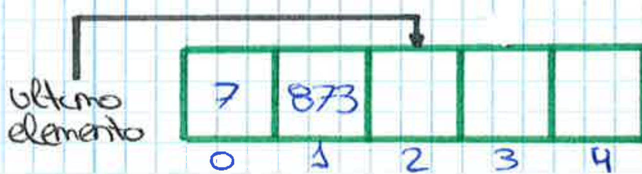


Tenemos un puntero que apunta donde estara el ultimo elemento.

A la hora de añadir elementos, solamente tenemos que ocupar la posición que marca el ultimo elemento y desplazarlo, siempre comprobando que la cola no este llena.



Para desencolar, primero se comprueba que la cola no este vacia. Si no esta vacia, se obtiene el primer valor y luego se desplazan todos los elementos de la posición 1 hacia la izquierda.



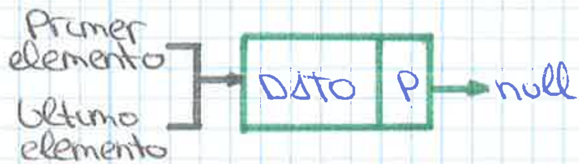
Saldria el "1" y el resto de los elementos se desplazaria a la izquierda.

- Cola dinámica: La cola dinámica se construye mediante la consecución de nodos.

Las operaciones son iguales que en las listas y el funcionamiento es muy similar.

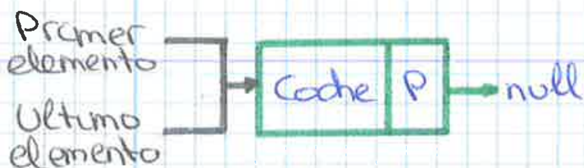
Cuando se crea una cola dinamica debemos partir de dos referencias básicas para controlar que hay en cada momento en cada posición de la cola.

- Primer elemento: Es el elemento que saldra al desencolar.
- Ultimo elemento: Es donde se insertara un nuevo nodo cuando se encole.



De primeras, ambas referencias apuntan al mismo sitio (null).

Si insertamos un elemento, ambas apuntaran a dicho elemento.



A medida vallamos insertando datos, iremos moviendo el puntero de ultimo elemento.

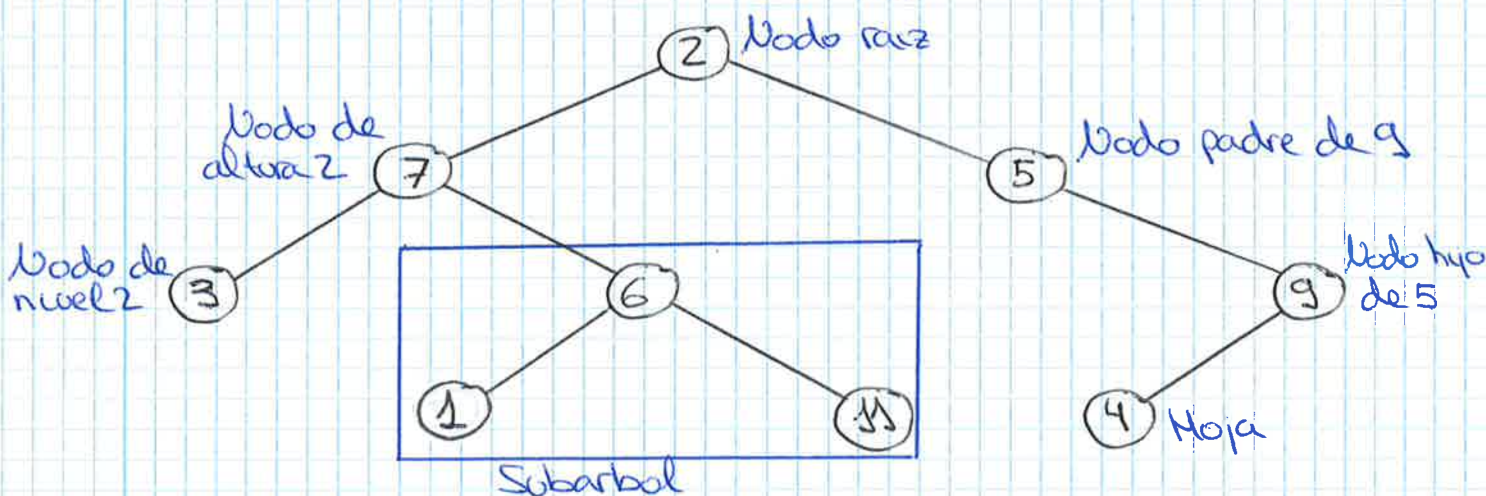
Cuando desencolemos, devolvera el valor al que apunta primer elemento y actualizara la referencia para que primer elemento apunte al siguiente elemento.

* Estructuras de datos jerarquicas:

- Arboles: Son una estructura de datos no lineal, porque organizan los datos en nodos de forma jerarquica.

Elementos del arbol:

- Nodo: Almacenan un dato y sus relaciones con los nodos hijo.
- Nodo raiz: Es el nodo del que derivan todos los demas.
- Nodo padre: Todo nodo tiene un solo padre, el cual es el nodo antecesor.
- Nodo hijo: El nodo hijo es el descendiente directo de un nodo. Un nodo puede tener cero, uno o mas hijos.
- Nodo hoja: Son nodos que no tienen descendencia.
- Subarbol: Cualquier nodo y sus hijos.
- Arbol nulo: Es un arbol vacio.
- Grado de un nodo: Número de hijos que salen del nodo.
- Nivel de un nodo: Número de ancestros que tiene desde la raiz.
- Camino: La secuencia de nodos que recorremos de un nodo a otro.
- Longitud del camino: Número de nodos que conforma el camino sin contar el nodo inicial.
- Altura de un nodo: La longitud del camino más largo desde un nodo hasta una hoja.
- Altura del arbol: La altura del nodo raiz.
- Grado del arbol: Máximo del grado de sus nodos



Camino ② a ③: 2, 7, 3

Longitud del camino ② a ③: 2

Grado nodo 6: 2, sus hijos son 1 y 11.

Altura del arbol: 3

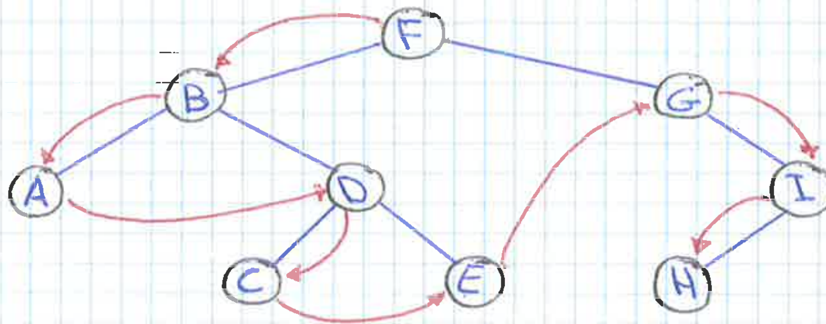
Grado del arbol: 2

Recorridos de los arboles: consiste en recorrer todos sus nodos.

- Formas de recorrer un arbol:

• Preorder: R I D

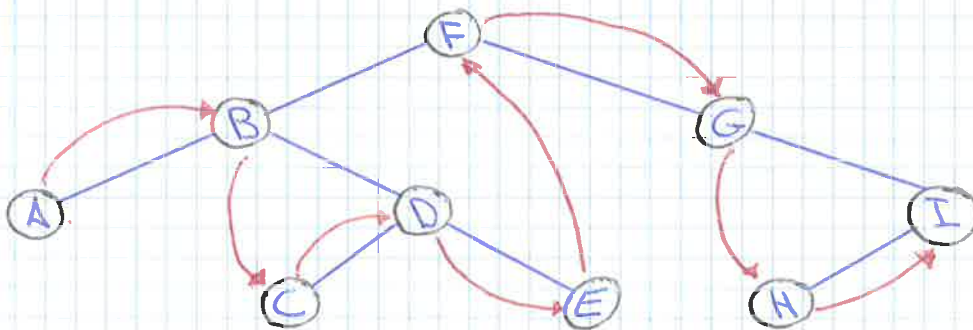
1. Visitamos la raíz
2. Recorremos en preorder el subarbol izquierdo.
3. Recorremos en preorder el subarbol derecho.



Recorrido: F, B, A, D, C, E, G, I, H

• Inorder: I R D

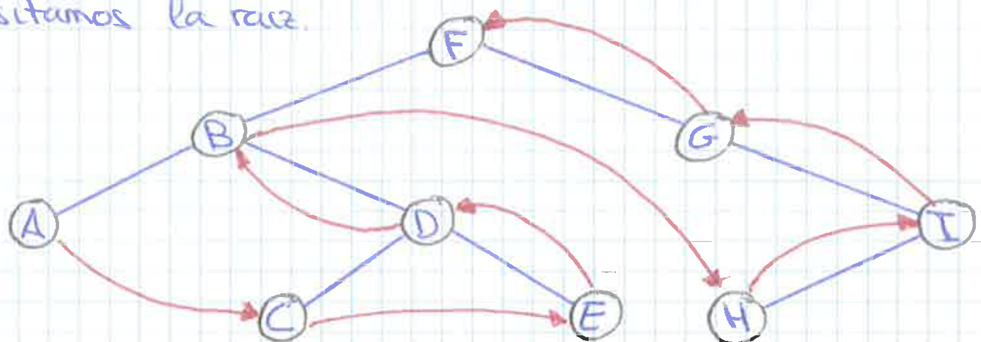
1. Recorremos inorder el subarbol izquierdo
2. Visitamos la raiz
3. Recorremos inorder el subarbol derecho.



Recorrido: A, B, C, D, E, F, G, H, I

• Postorder: I D R Se recorre de izquierda a derecha, de abajo hacia arriba y acabando en la raiz.

1. Recorremos en postorder el subarbol izquierdo
2. Recorremos en postorder el subarbol derecho
3. Visitamos la raiz.



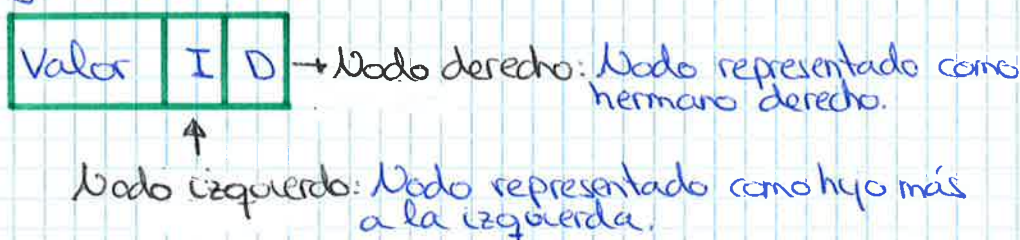
Recorrido: A, C, E, D, B, H, I, G, F

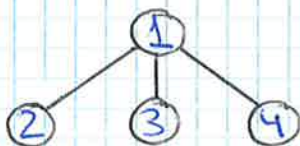
Arboles n-arios: Son arboles en el que cada nodo tiene n-hijos.

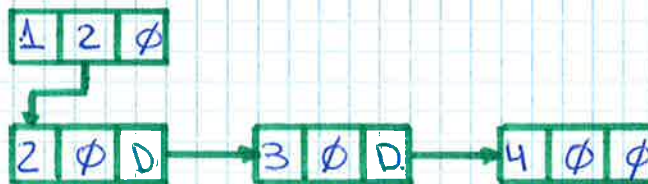
Operaciones básicas:

- Crear (Valor V): Crea un arbol con "V" en la raiz.
- Obtener (Nodo n): Obtiene el valor del nodo "n".
- Asignar (Nodo n, Valor v): Asigna el valor "v" al nodo "n"
- Insertar izquierdo (Nodo n, Valor v): Crea un nodo con valor "v" como hijo a la izquierda del nodo "n".
- Insertar derecho (Nodo n, valor v): Crea un nodo con valor "v" como hermano derecho del nodo "n".
- Eliminar izquierdo (Nodo n): Elimina el nodo hijo más a la izquierda del nodo "n".
- Eliminar derecha (Nodo n): Elimina el nodo hermano derecho del nodo "n".
- Inicio (): Se situa en el nodo raiz del arbol.
- Final (): Se situa en el nodo final del arbol.
- Izquierdo (Nodo n): Obtiene el hijo más a la izquierda del nodo "n".
- Derecho (Nodo n): Obtiene el hermano a la derecha del nodo "n".
- Padre (Nodo n): Obtiene el padre del nodo "n".

La representación de un nodo de un arbol de estas caracteristicas tendra la siguiente estructura:



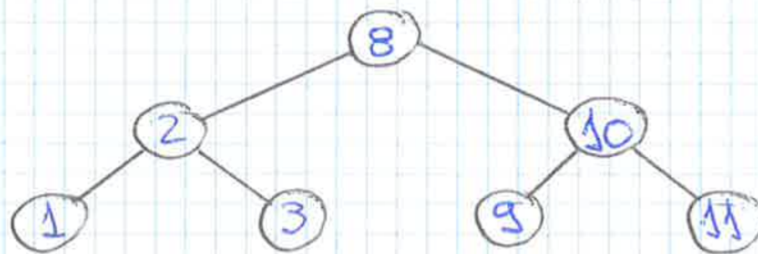




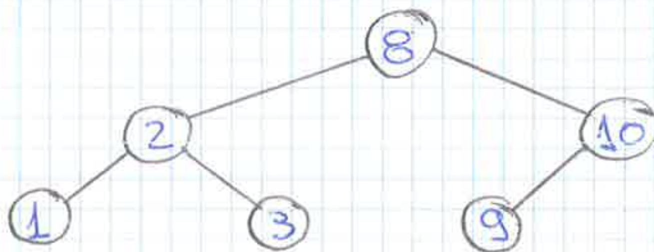
Arboles binarios: Son arboles n-arios donde cada nodo tiene como maximo dos hijos. (Grado 2)

- Arbol binario perfecto: Cuando todas sus hojas estan a la misma profundidad.
- Arbol binario completo: Es un arbol perfectamente equilibrado hasta el penultimo nivel, y en el ultimo nivel todos sus nodos estan agrupados a la izquierda.
- Arbol equilibrado: La altura de los subarboles no difiere en mas de uno.

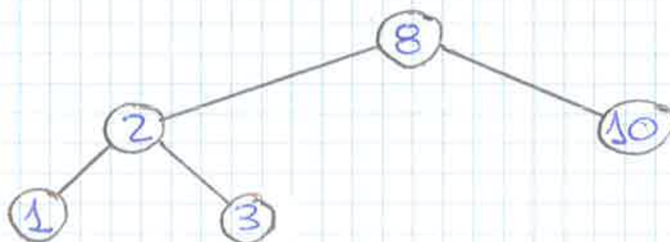
• Arbol perfectamente equilibrado: Binario perfecto



• Arbol completo:



• Arbol equilibrado:



- Arboles binarios de busqueda (ABB):

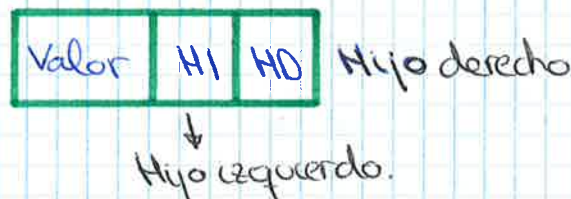
Un arbol binario de busqueda, tambien llamado arbol ordenado, es un arbol que cumple con la propiedad de ordenación.

- Propiedad de ordenación:
  - Si el elemento es menor a la raiz, sera un nodo del subarbol izquierdo.
  - Si el elemento es mayor a la raiz, sera un nodo del subarbol derecho.

Esto quiere decir, que va de menor a mayor de izquierda a derecha.

El recorrido de un arbol binario ordenado es el recorrido inorder y es el que nos devuelve los elementos ordenados.

A diferencia de los arboles n-arios, los arboles ABB tendran la siguiente estructura:



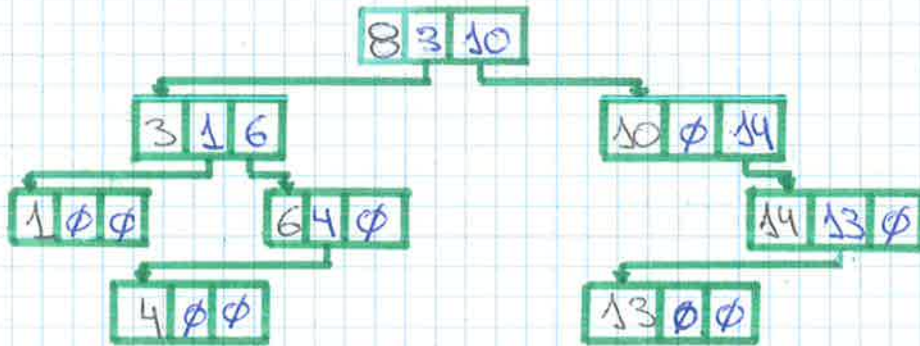
Operaciones:

- Busqueda:
  - Se va al nodo raiz, si coincide con el elemento se finaliza la busqueda.
  - Si el elemento es menor, se busca en el subarbol izquierdo.
  - Si el elemento es mayor, se busca en el subarbol derecho.
  - Si alcanzamos un nodo hoja y el elemento no coincide, significa que el arbol no tiene ese elemento.

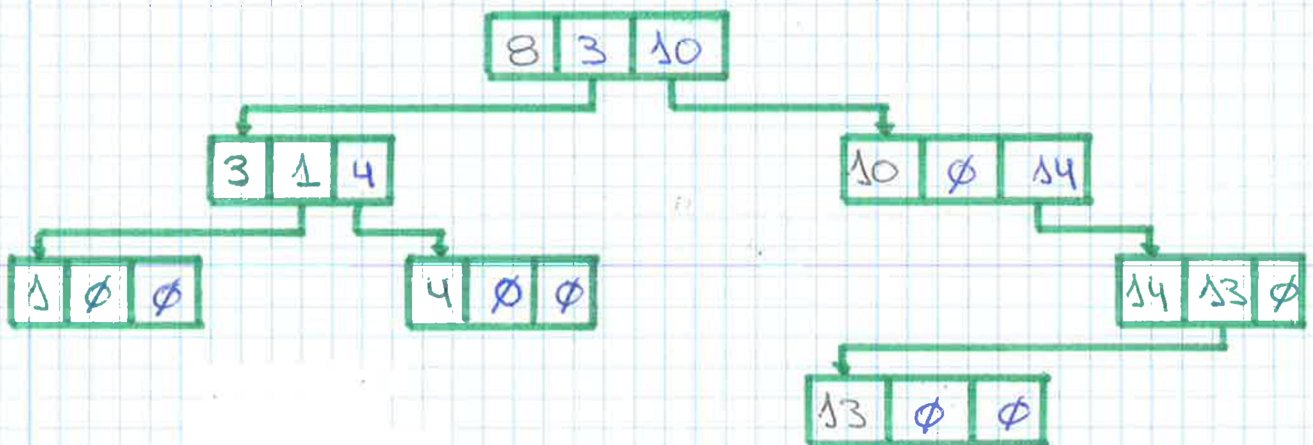
- Insertar:
  - Si el elemento es menor, se inserta en el subarbol izquierdo.
  - Si el elemento es mayor, se inserta en el subarbol derecho.

- Borrado: El prodeso de borrado es mas complejo y existen diferentes casos en los que se procedera de forma diferente.
  - Borrar un nodo hoja: Se elimina el elemento y el padre ahora apuntara a "null".

• Borrar un nodo con un subárbol hijo: Se borra el nodo y se asigna su hijo a su padre.



Supongamos que queremos borrar el nodo (6), entonces, su nodo padre, deberá apuntar a su nodo hijo: 3 → 4.

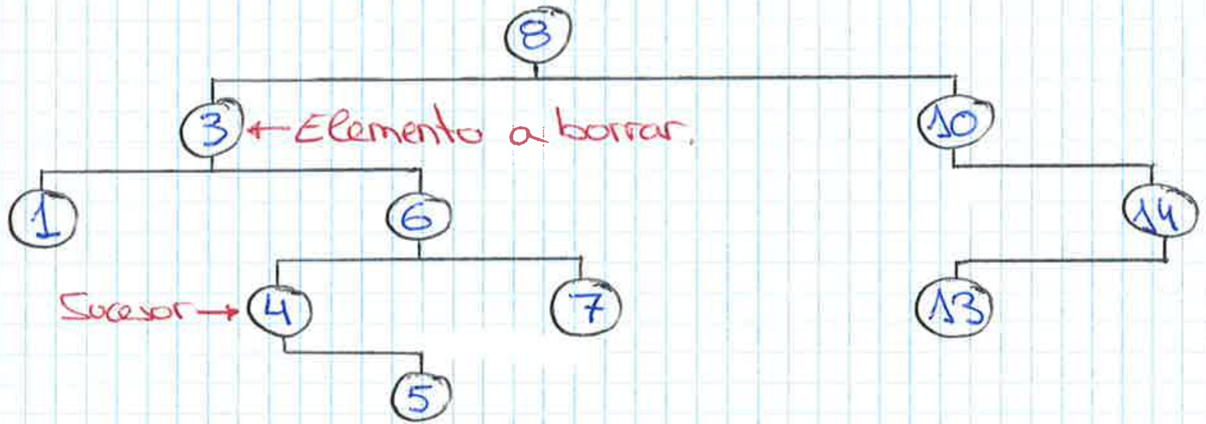


• Borrar un nodo con dos subárboles: Se realiza en dos etapas, pero hay que conocer ciertos conceptos:

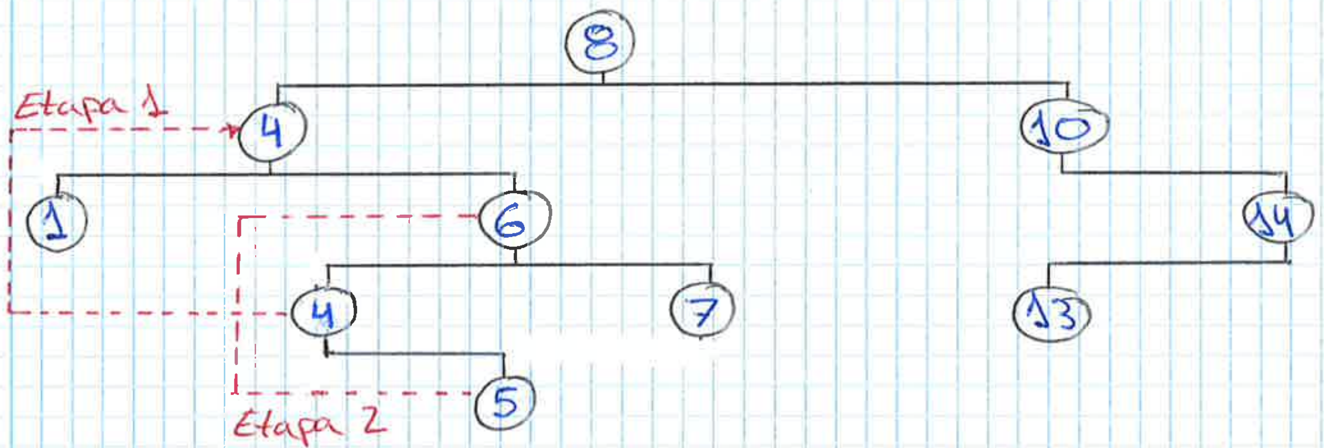
- Sucesor: Dentro de la rama de elementos mayores, el nodo de valor más pequeño, es decir, el nodo más pequeño al que se accede a través de la rama derecha del elemento a borrar.

Etapa 1: Se reemplaza el nodo a borrar con el sucesor.
Etapa 2: Una vez reemplazado el nodo, se da el hijo de dicho nodo en adopción a su abuelo.

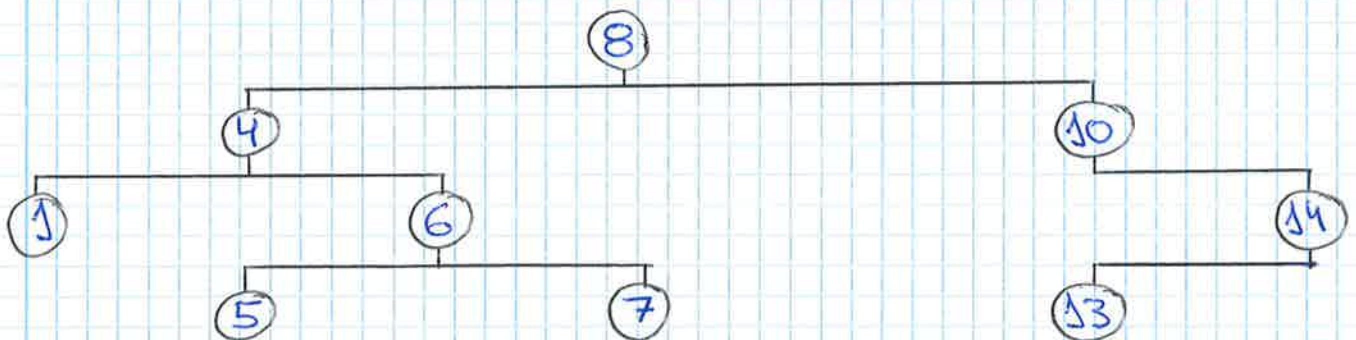
Supongamos el siguiente arbol ABB que queremos eliminar el elemento 3.



Etapa 1: Se reemplaza el nodo a borrar con el sucesor, es decir, del subarbol derecho del elemento a borrar, el más pequeño.
Nuestro arbol resultante seria el siguiente.



Etapa 2: El hijo del sucesor lo damos en adopción a su abuelo.
Nuestro arbol quedaria así:



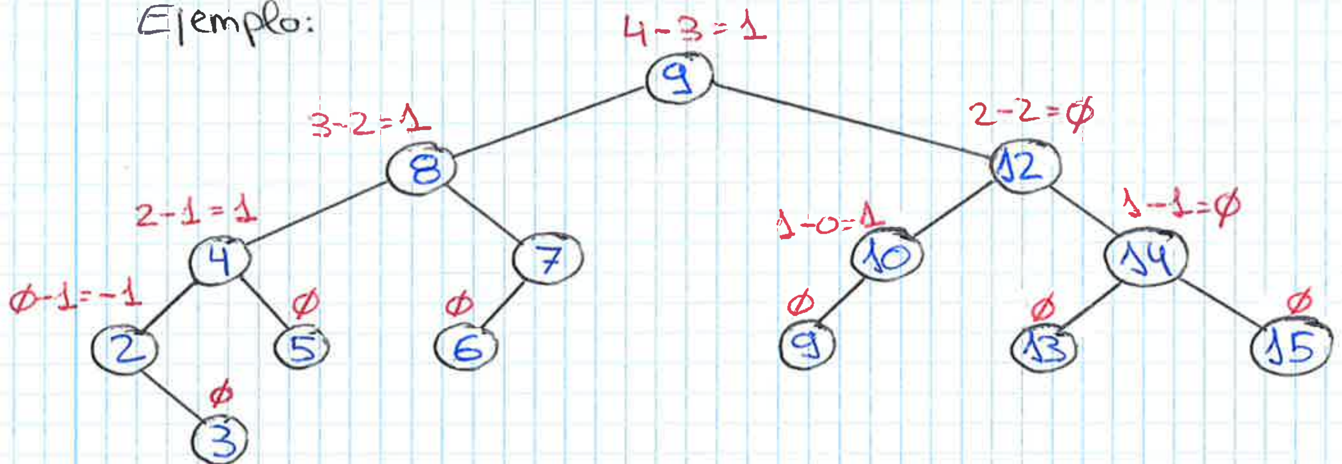
*Arboles AVL:

Un arbol AVL es un arbol binario ordenado y balanceado. La eficiencia con la que podemos localizar un nodo viene dada por el recorrido que se realiza desde la raiz hasta dicho nodo. Esta caracteristica se denomina balance o equilibrio. (Factor de equilibrio = FE).

FE = ALTURA SUBARBOL Iz - ALTURA SUBARBOL Dr

La diferencia debe ser 1, ∅ o -1 para que este balanceado

Ejemplo:



- Todas las hojas tienen FE=∅.
- El resto de nodos FE=SI-SD

Podemos decir que nuestro arbol esta balanceado porque todos los subarboles cumplen el FE.

Este equilibrio garantiza que el coste computacional de busqueda es siempre $O(\log n)$.

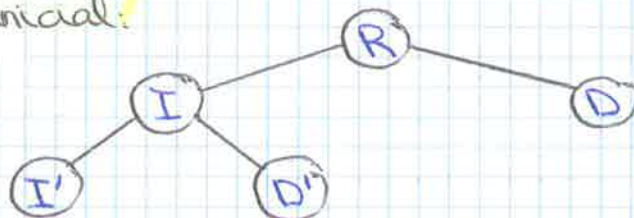
En resumen: Los arboles AVL son arboles ABB que estan siempre balanceados y su FE es menor o igual a 1 punto.

Operaciones: Generalmente se usan los mismos algoritmos que el los arboles binarios de busqueda, la diferencia son una serie de operaciones adicionales que permiten reestructurar el arbol para que se mantenga ordenado y balanceado. Estas operaciones son llamadas rotaciones.

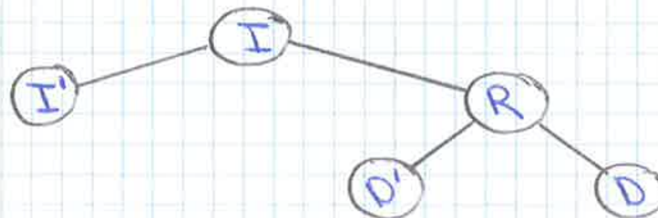
- **Rotaciones:** Existen dos casos; rotaciones simples o dobles, que a su vez, pueden ser hacia la izquierda o hacia la derecha.

  - **Rotación simple a la derecha:** consiste en la formación de un nuevo arbol
    La raiz sera el hijo izquierdo del actual.
    El hijo izquierdo sera el nieto izquierdo del actual.
    El hijo derecho es un nuevo arbol cuya
    raiz es la raiz inicial, el
    hijo derecho el hijo derecho inicial y el
    hijo izquierdo el nieto derecho inicial.

Arbol inicial:

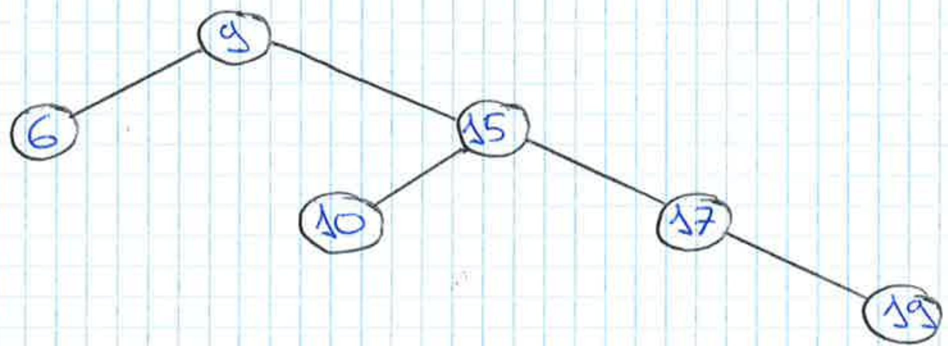


Nuevo Arbol:

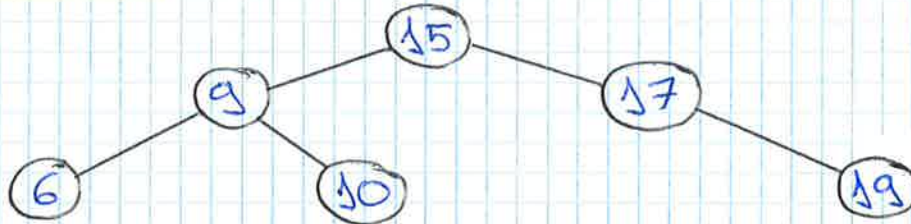


  - **Rotación simple a la izquierda:** es el inverso que la rotación simple a la derecha.
    La raiz sera el hijo derecho actual.
    El hijo derecho sera el nieto derecho del actual.
    El hijo izquierdo es un nuevo arbol cuya
    raiz es la raiz inicial, el
    hijo izquierdo el hijo izquierdo inicial y el
    hijo derecho el nieto izquierdo inicial.

Arbol Inicial:



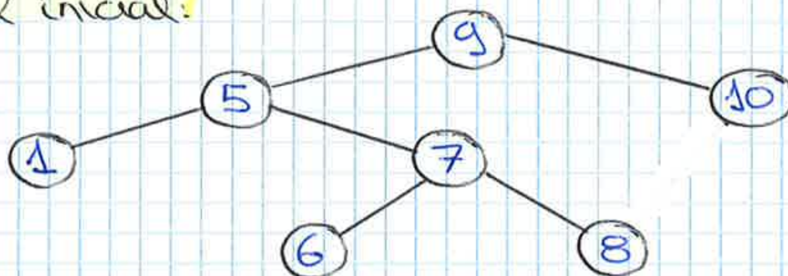
Arbol nuevo:



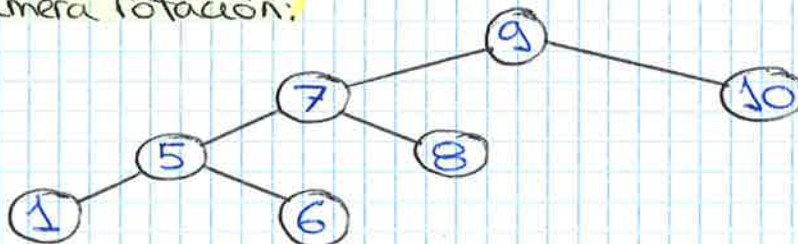
Truco: Primero movemos la nueva raiz y segundo comenzamos a ordenar el arbol como si de un ABB se tratara.

Rotación doble a la derecha: Consiste en dos rotaciones simples, primero una a la izquierda del subarbol y luego otra a la derecha con el arbol resultante.

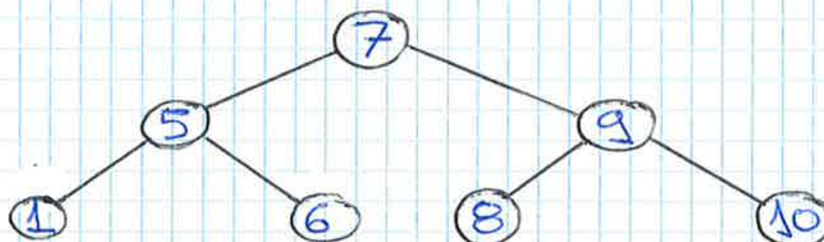
Arbol inicial:



Primera rotación:

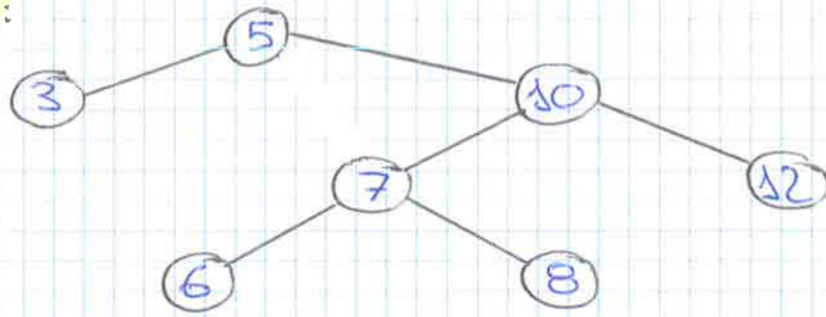


Segunda rotación:

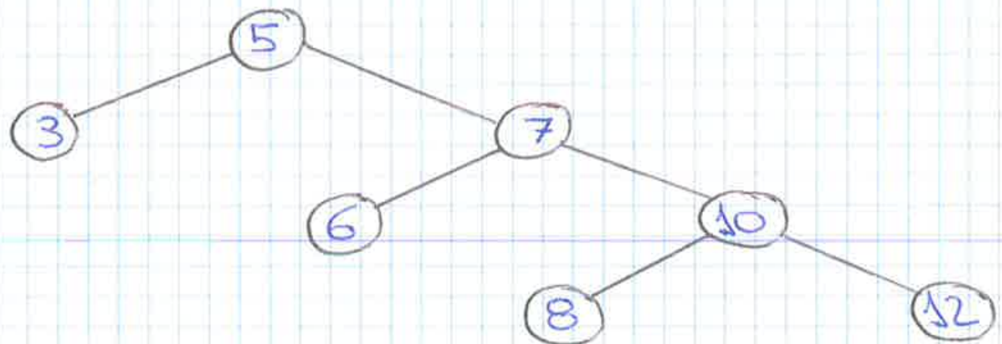


Rotación doble a la izquierda: Consiste en dos rotaciones simples, primero una a la derecha del subarbol y luego otra a la izquierda con el arbol resultante.

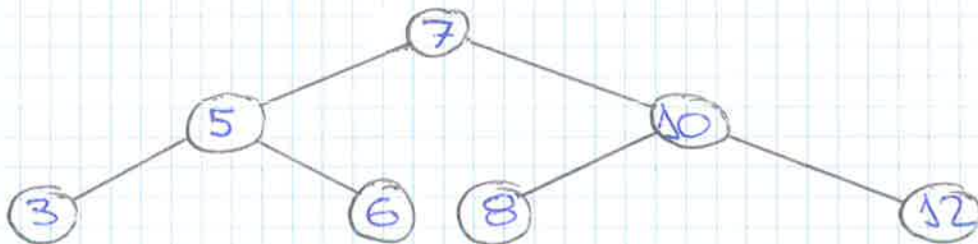
Arbol inicial:



Primera rotación:



Segunda rotación:



Estructura y diseño de un arbol AVL:

La implantación de un arbol AVL se basa en la misma que la de los arboles ABB, basada en la creación de nodos del arbol que contenían punteros a sus nodos hijos. Además, se le añade una propiedad mas a los nodos. Existen dos posibles formas:

- Añadir la altura del nodo: Cada vez que se añade un nodo a un arbol, tendremos una propiedad que nos indicara la altura de dicho nodo.

(V) → Valor
[A] → Altura

Ejemplo: Añadimos en orden 9, 10, 5, 1, ∅

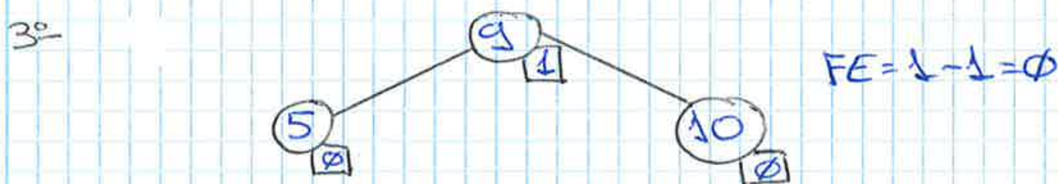
1º

FE = ∅ - ∅ = ∅

2º

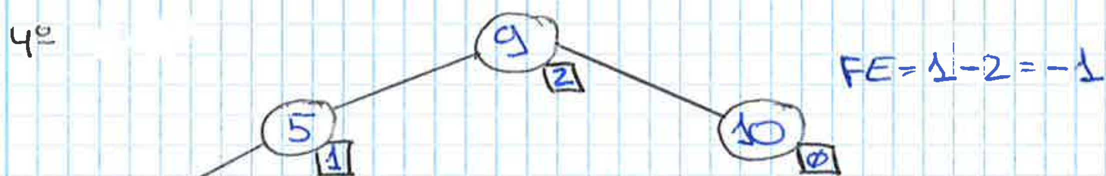
FE = 1 - ∅ = 1

3º



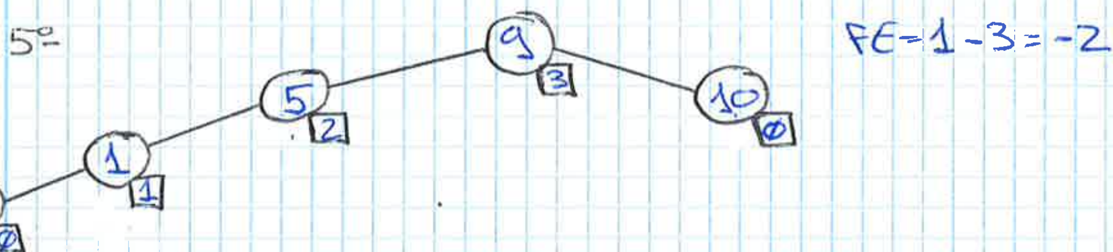
FE = 1 - 1 = ∅

4º



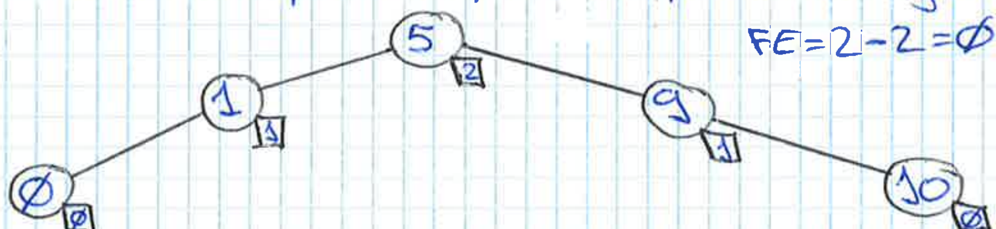
FE = 1 - 2 = -1

5º



FE = 1 - 3 = -2

6º Como esta desequilibrado, lo reequilibramos girando a la derecha:



FE = 2 - 2 = ∅

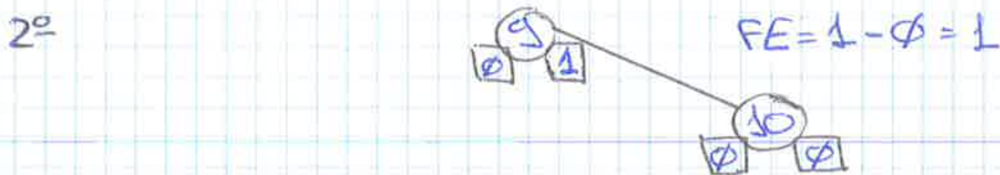
Mediante una referencia a la altura de cada nodo, es posible ir calculando el factor de equilibrio, aunque es necesario echar mano de los subarboles para realizar este calculo.

Añadir la altura de los subarboles: Se guardan en cada nodo la altura del subarbol izquierdo y derecho, asi calcularemos el FE mas rapido y directo.

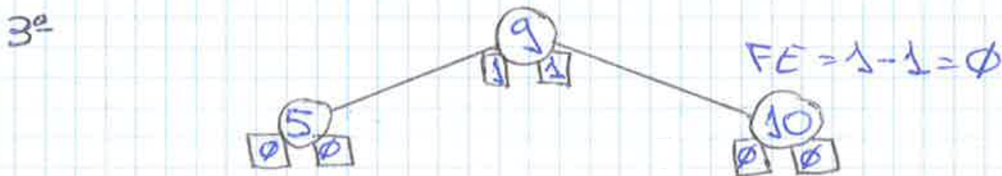
Ejemplo: añadimos en orden 9, 10, 5, 1, ∅.

1º

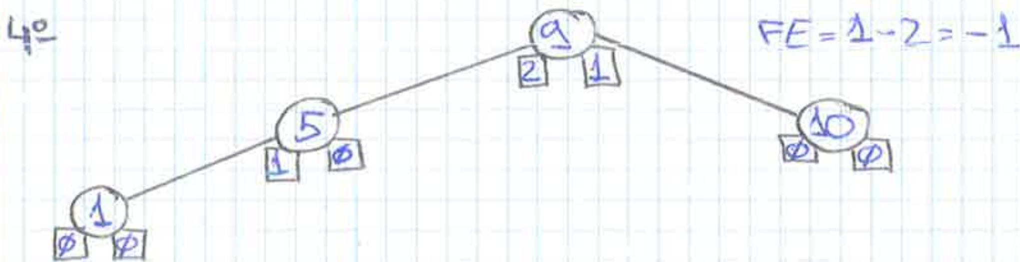
2º



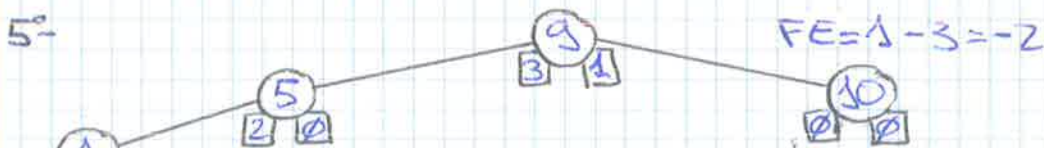
3º



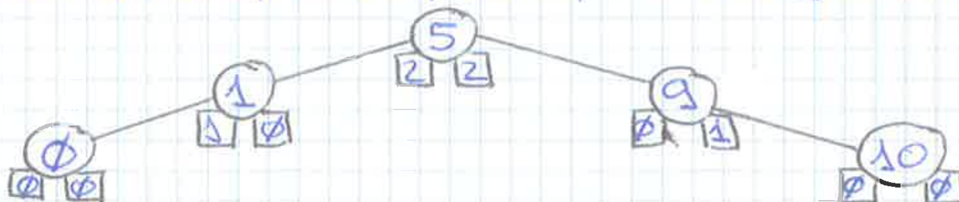
4º



5º

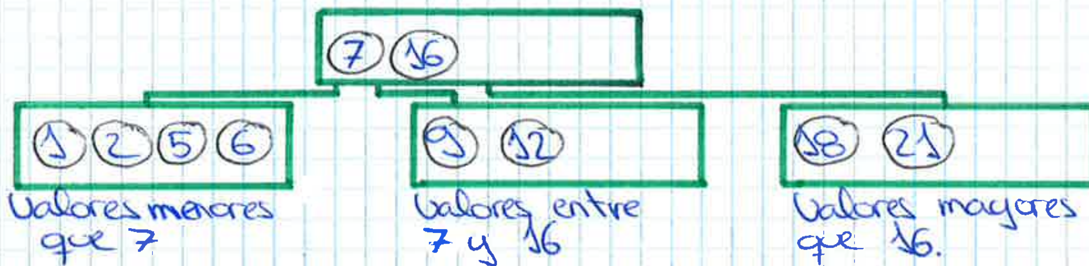


6º Como esta desequilibrado, lo equilibramos girando a la derecha:

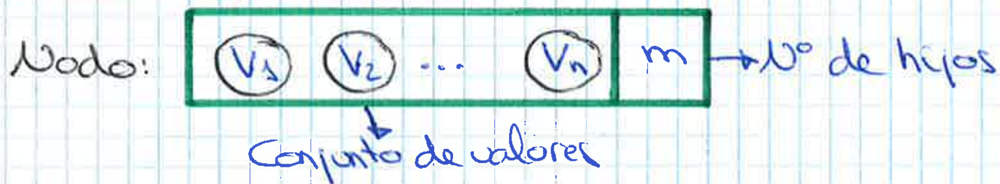


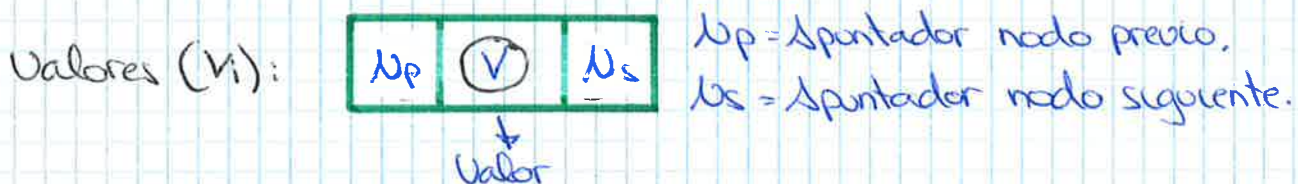
***Arboles multicamino:** Son arboles cuyo grado es superior a 2 y con cota superior a un valor "g".
Un arbol binario es un arbol multicamino con un valor de g=2.
Existen diferentes variantes de arboles multicamino:

- **Arboles B:** son arboles AVL multicamino, donde los nodos internos han de tener un número variable de nodos hijo dentro de un rango predefinido.
  Se utilizan para BBDD y sistemas de ficheros, ya que estan optimizados para leer y escribir grandes bloques de datos.
  Cuando se modifica la estructura del arbol, el número de nodos no varia, simplemente son unidos o separados.
  Tienen la ventaja de no necesitar un rebalanceo tan frecuente, pero producen un mayor gasto de espacio.
  Cada nodo tiene un numero de pares clave-valor.



Diseño e implementación:

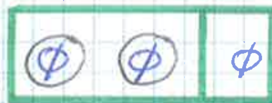




Adicionalmente podremos tener el número de claves y la altura.

Operaciones (2 claves, 3 hijos):

- Creación: Se crea el nodo con dos nodos vacíos y ∅ hijos.



- Inserción:

Primero buscamos en que hoja tengo que insertar.

- Si cabe, se introduce el nodo.
- Si no cabe, hay que dividir:

  El valor mediano se lleva al padre.

  Los valores menores al hijo izquierdo.

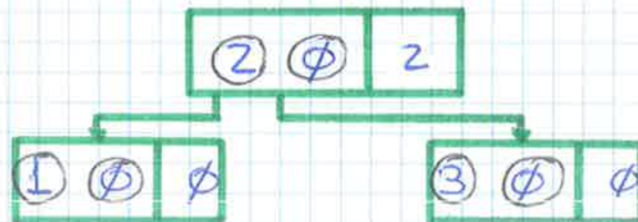
  Los valores mayores al hijo derecho.

Ejemplo: Insertamos del 1 hasta el 7 en orden:

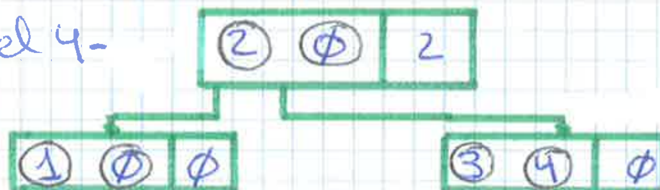
Inserto el 1 -

Inserto el 2 -

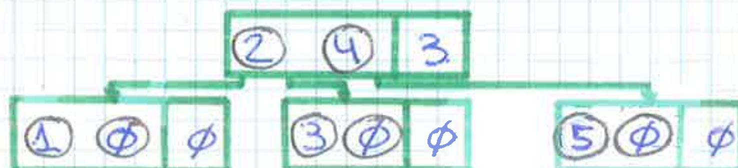
Inserto el 3, como no cabe, dividimos:



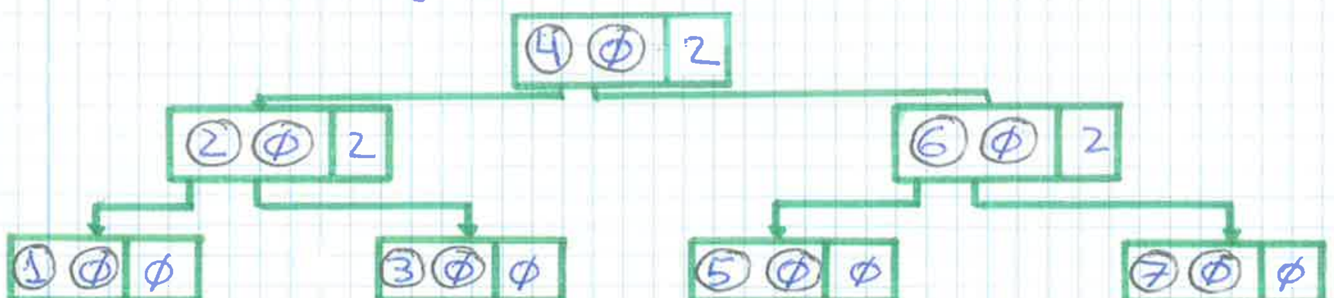
Inserto el 4 -



Inserto el 5, como no cabe, dividimos:



Inserto el 6 y seguido el 7, pero como no cabe, dividimos:



Busqueda:

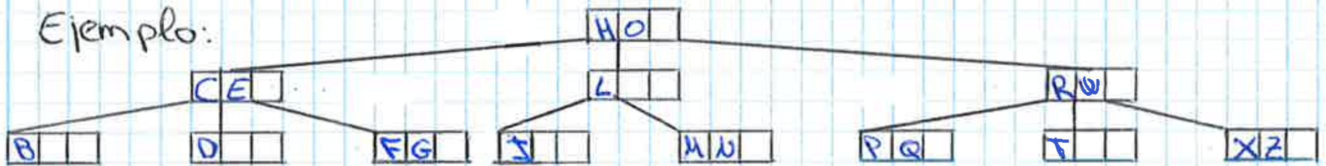
- Si la raiz contiene el valor, fin.
- Si es hoja y el valor no esta, el valor no se encuentra.
- Si no esta en la raiz:
  - Si el valor es menor que la primera clave, izquierda.
  - Si esta entre Ki y Ki+1 se sigue ese camino.
  - Si es mayor que Ki+1 a la derecha.

Borrado:

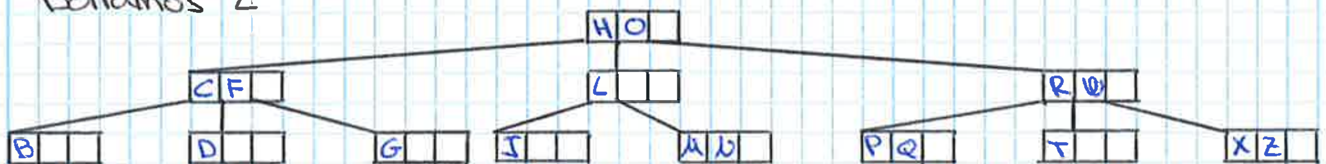
- Encontrar el valor que queramos borrar:
  - Si es un nodo hoja y tiene suficientes claves, fin.
  - Si el vecino tiene mas de $\frac{m}{2}$ claves, repartimos valores
  - Si hay menos valores del minimo, se combina con el padre.

  - Si no es hoja, sustituir el valor por el mayor de la izquierda o el menor de la derecha de los hijos.

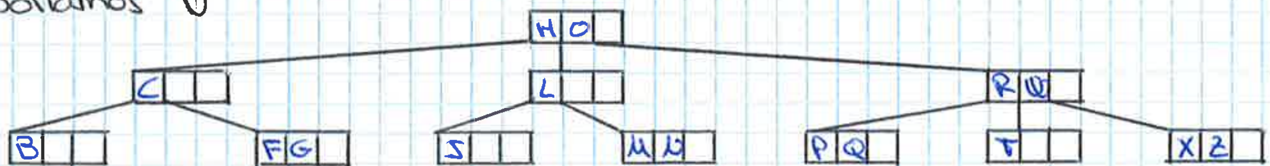
Ejemplo:



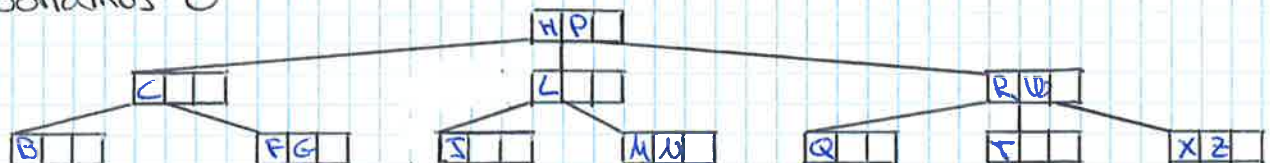
Borramos "E"



Borramos "D"



Borramos "O"



*Montículos (heaps) y colas de prioridad:

Los heaps es una estructura tipo arbol cuya información pertenece a un conjunto ordenado.

Su implementación esta basada en array.

Para que un arbol sea considerado montículo debe cumplir dos condiciones:

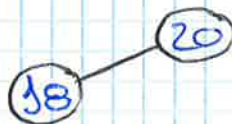
- Ser un arbol binario completo: Es aquel que todos los niveles estan llenos, con la excepción del último nivel, que esta lleno de izquierda a derecha.
- Condición de orden:
  - De máximos: Cada padre tiene un valor mayor que sus hijos
  - De mínimos: Cada padre tiene un valor menor que sus hijos.

Operaciones:

- Inserción: Para realizar la inserción, solamente deberemos insertar los nodos cumpliendo las dos condiciones de los heaps.
  Ejemplo, insertamos 20, 18, 9, 8, 10, 12.

1º Insertamos el 20.

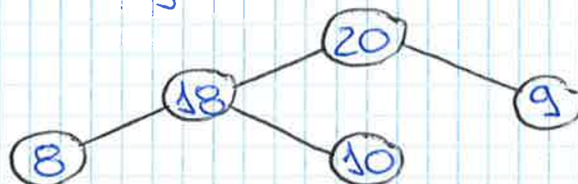
2º Insertamos el 18 siguiendo las condiciones:



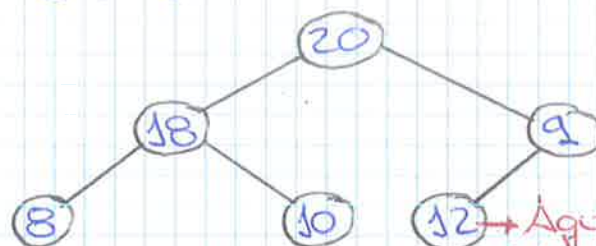
3º Insertamos el 9:

En un arbol de busqueda se incertaria a la izquierda del 18, pero aquí, al tratarse de un heap, ha de cumplir las características, en este caso ser un arbol completo.

4º Insertamos el 8 y el 10:

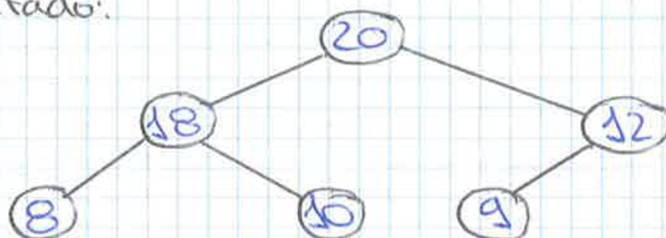


5º- Insertamos el 12:



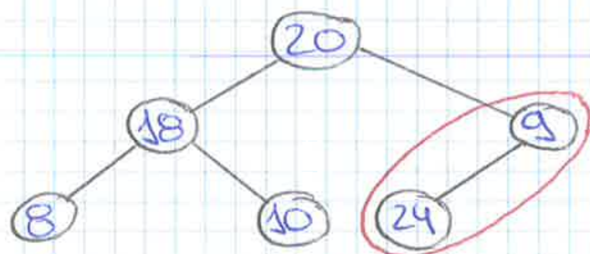
→ Aquí se incumple la segunda condición, esto implica que deben intercambiarse el hijo por el padre.

Resultado:



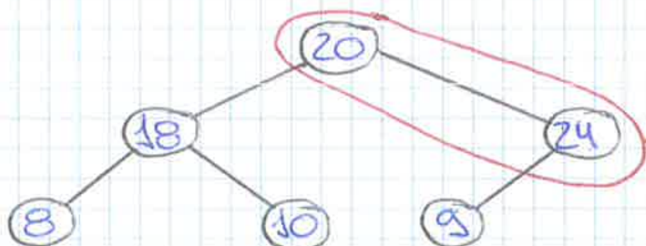
Si hubieramos insertado por ejemplo el 24, se incumpliría dos veces la segunda condición, así que seguiríamos intercambiando el hijo por el padre hasta que este correcto.

1º.



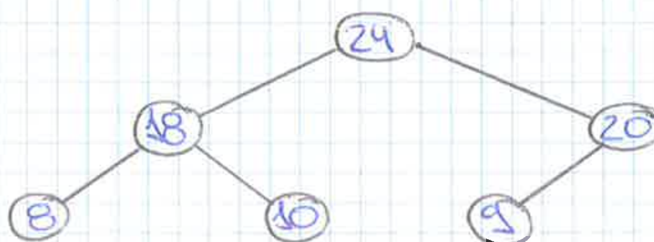
→ Incumple 2ª condición, Intercambiamos.

2º



→ Incumple 2ª condición, volvemos a intercambiar.

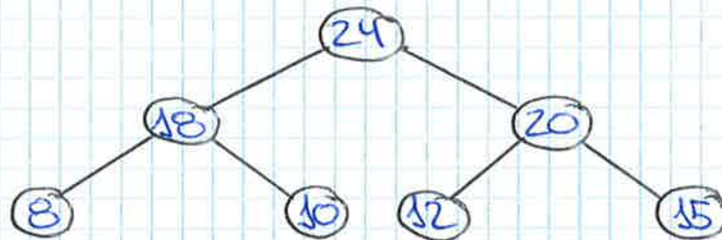
3º



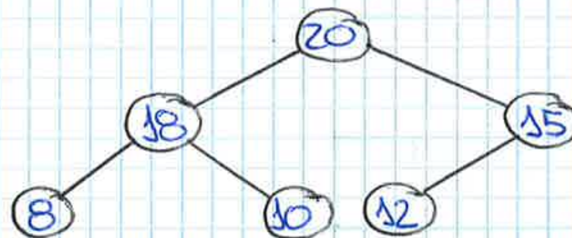
En resumen, en caso de que al insertar un nodo se viole la regla del montículo de máximos o mínimos, aplicaremos el intercambio padre-hijo tantas veces como sea necesario hasta que el montículo cumpla las reglas.

-Borrado: Como los heaps son utilizados para implementar colas de prioridad, el borrado de elementos se realiza siempre borrando la raiz y se reestructura el arbol, cumpliendo siempre con las condiciones de los heaps.

Queremos borrar el siguiente elemento, en este caso el 24:



Sacamos el 24 y reestructuramos:



Colas de prioridad: Tiene la misma estructura que una cola, con la diferencia de que cada elemento tiene una prioridad asociada, siendo el elemento de mayor prioridad el que sale el primero de la cola.
Un ejemplo de una cola de prioridad sería el triaje de un hospital.
A la hora de crear una cola de prioridad ha de establecerse un criterio de prioridad, por el que una prioridad es mayor que otra.
Generalmente viene indicada por un valor numérico:

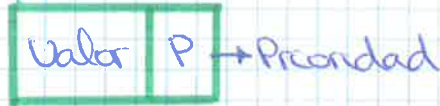
Mayor valor → Mayor prioridad }
Menor valor → Mayor prioridad } Determina si el heap es de máximos o mínimos.

Operaciones:

- Inicializar (): Proceso en el que se inicia la cola.
- Encolar (elemento): Añade un elemento a la cola.
  Devuelve True o False indicando si se ha añadido correctamente.
- Desencolar (): Devuelve el elemento con mayor prioridad.
- Primero (): Devuelve el elemento que ocupa la primera posición de la cola.
- Vacia(): Comprueba si la cola esta vacia.

Ejemplo:

Nodo



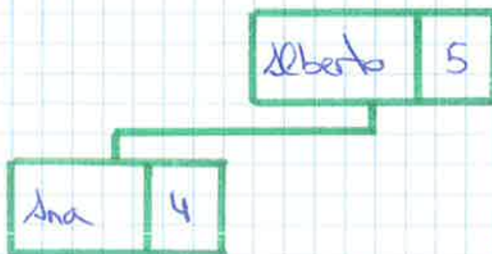
Tenemos pacientes con distintas lesiones, los priorizamos según su gravedad y urgencia:

Paciente	Prioridad
Alberto	Azul – 5
Ana	Verde – 4
María	Amarillo-3
Andrés	Rojo – 1
Aitana	Amarillo-3

Aquí ya tenemos ejecutado y decidido nuestro criterio de prioridad, en el que el valor más pequeño indicará la mayor prioridad, por lo tanto, sigue una condición de mínimos.

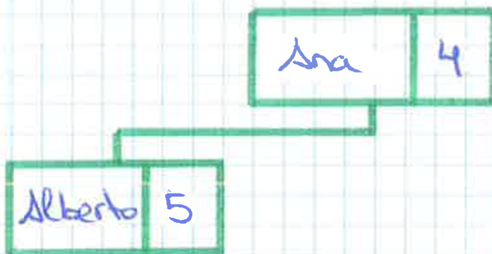
Insertamos Alberto:

Insertamos Ana



Incumple condición de orden

Reestructuramos:



Insertamos María:

Se vuelve a incumplir la condición de orden.

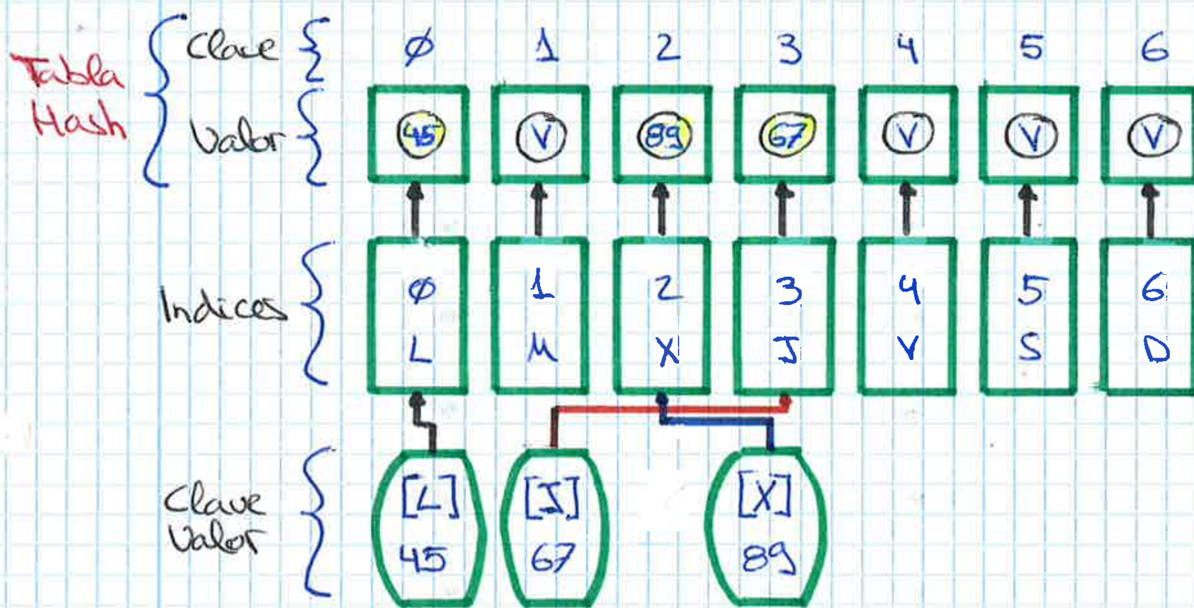
Reestructuramos:

Seguiremos insertando hasta completar el árbol.

*Tablas Hash:

Las tablas hash son una estructura de datos que funcionan asociando claves con valores.
Son muy utilizadas para optimizar la busqueda, ya que utilizan una clave generada para acceder a los elementos almacenados.
Resumiendo, es una estructura como una tabla con un conjunto de entradas. Cada entrada (valor) tiene asociada una clave única, permitiendo que diferentes entradas de una misma tabla tengan tambien diferentes claves. La clave nos permitira recuperar el valor.

Ejemplo: Imaginemos que queremos organizar los periodicos que compramos a diario, de tal forma que se puedan ubicar de forma rápida.
Cogemos una caja con 7 huecos, asi que los ordenaremos por dias.



Los indices devuelven la posición a la que se debe acceder en la propia tabla.
Clave-Valor contiene el dato a insertar y la clave mediante la cual calcularemos nuestro indice para insertar en la tabla el valor.

Funcionamiento:

Las tablas Hash se implementan sobre vectores unidimensionales (arrays). Esto es para que la función hash solo tenga que calcular un valor que se ajuste al tamaño del array.
Tambien se podría implementar en una matriz, pero deberíamos calcular el parametro fila y columna para insertar nuestro elemento. Es decir, dos claves diferentes.

Función hash:

- Determinista: Mismo valor tiene siempre la misma clave
- No invertible: No se puede reconstruir el dato a partir del hash generado.
- Aleatoria.
- Eficiente.

A medida introducimos datos en nuestra tabla hash, hay más posibilidades de que haya colisiones (dos salidas iguales).

Ejemplo de función hash: Siguiendo el ejemplo anterior, nuestra función hash creara un indice unico a partir de la clave introducida.

Criterios para diseñar una buena función hash:

- Buscar un buen rendimiento tratando de evitar las colisiones siempre que sea posible.
- La función debe proporcionar una distribución uniforme de los valores hash.

Una función con una distribución no uniforme da lugar a aglomeramientos (valores tienden a caer unos muy cerca de otros).

Se suelen utilizar vectores cuyo tamaño sea primo.

Colisiones: Se producen cuando una función hash produce un mismo valor para dos entradas diferentes.

Las colisiones son practicamente imposibles de evitar, debido a esto, es necesario implementar estrategias de resolución de colisiones:

- Protección activa: Tratar de evitar la colisión.
- Protección pasiva: Tratar la colisión cuando se produzca.
- Redispersión: Aumentar o disminuir el tamaño del vector sobre el que se soporta la tabla en función del número de elementos.

Redispersión:

Para saber cuando vamos a realizar una redispersión, nos basaremos en unos umbrales del factor de carga (FC)

El factor de carga es una propiedad de las tablas hash que nos permite saber la "estabilidad computacional".

$$FC = \frac{S}{n} \quad \begin{matrix} \rightarrow \text{Nº de elementos.} \\ \rightarrow \text{Tamaño del array.} \end{matrix}$$

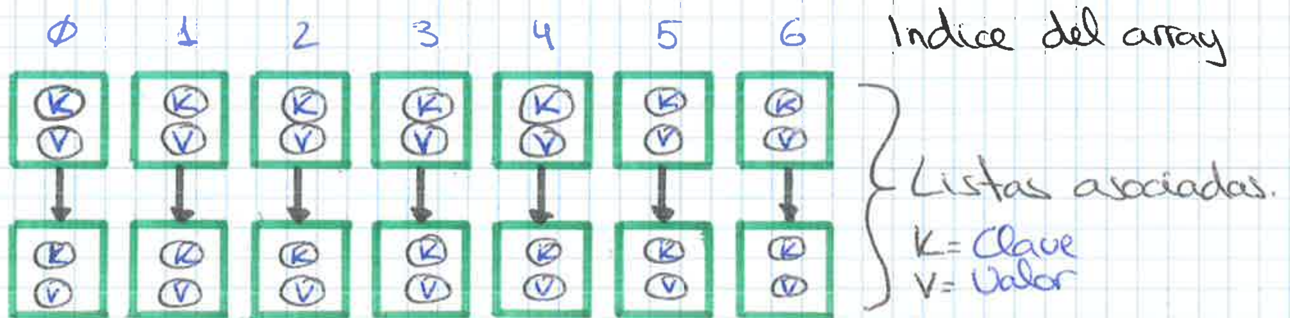
La redispersión consiste en mejorar el FC hasta niveles aceptables.

- Se creara una nueva tabla con el nuevo tamaño (siguiente primo mas cercano a $2 \cdot n$).
- Mover los elementos a la nueva tabla:
  - Se obtiene cada elemento de la tabla antigua.
  - Se calcula la nueva posición y se inserta.

La redispersión inversa se realiza cuando el número de elementos decrece mucho.

Se realiza el proceso inverso al anterior.

Tablas hash abiertas: Son tablas hash que utilizan un método pasivo de resolución de colisiones. Se basa en disponer en cada celda del array de la tabla hash de una estructura de datos dinámica, que es donde se almacenarán los elementos. Cuando hay colisión, se añade una posición a la estructura dinámica.



En estos casos de tablas abiertas, hay que guardar en cada posición la clave y el valor, ya que si tenemos que buscar un elemento, solo podremos distinguirlo con la clave original, pues puede haber varios elementos en la misma posición.

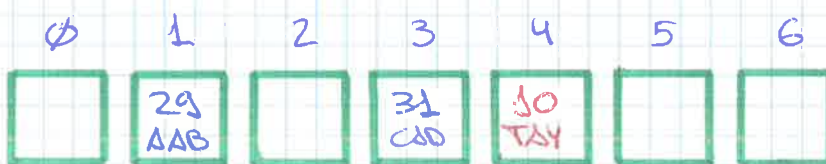
Tablas hash cerradas: Son tablas hash que utilizan un método pasivo de resolución de colisiones. Se basa en la búsqueda de posiciones alternativas cercanas para almacenar el elemento.

Existen tres estrategias a la hora de buscar donde se almacenará:

- Exploración lineal:
  - Se modifica la función hash:

    Indice = hash % n → Indice = (hash + i) % n
    ↳ valores de Ø a n.

  Si hay colisión, ir a la derecha hasta encontrar posición libre.



↳ como la posición estaba ocupada, la hemos pasado a la siguiente.

A la hora de buscar un valor, realizaremos el recorrido de la misma manera y en caso de llegar a una posición vacía, significará que el valor no se encuentra en la tabla.

Borrado: Si realizamos el borrado de forma convencional, tenemos el problema que, al ir a buscar un elemento que no se encuentra en la posición ideal e ir recorriendo el array, nos encontremos con una posición vacía y dejemos de recorrerlo (ERROR)

¿Como solucionamos esto?

Realizando un borrado perezoso, que consiste en marcar la casilla como eliminada, dando a entender que previamente hubo elementos, con lo que al buscar debe continuarse.

La exploración lineal da problemas de agrupamiento, haciendo que las inserciones resulten más costosas.

Exploración cuadratica: Su objetivo es mejorar el agrupamiento de la exploración lineal.

$$Indice = hash \% n \rightarrow Indice = (hash + i^2) \% n$$

Esta exploración trata de dar "pasos más grandes" a la hora de buscar una nueva casilla donde insertar.

El resto de procesos se realiza de forma identica a la exploración lineal.

Exploración doble: Su objetivo es resolver el agrupamiento secundario de la exploración cuadratica.

$$Indice = hash \% n \rightarrow (hash + i \cdot h(x)) \% n$$

↳ Segunda función hash de dispersión.

Su calculo de la función de dispersión es más elevado.

En conclusión, la exploración cuadratica es la más simple y rápida en la practica.

Redispersión: Ampliar tamaño del array
- Tablas abiertas: $FC > 1$
- Tablas cerradas: $FC > 0'5$

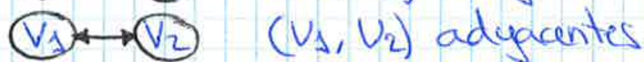
Redispersion inversa: Disminuir tamaño del array
- Tablas abiertas: $FC < 0'33$
- Tablas cerradas: $FC < 0'16$

$$FC = \frac{S}{n}$$ (S → Nº elem, n → T. array)

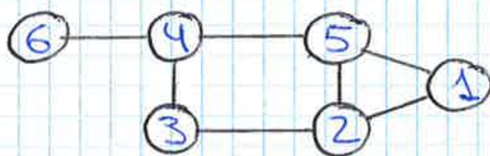
*Grafos:

Los grafos son un tipo abstracto de datos que se basa en una serie de nodos (V=vertices) y aristas (E=edge).
Los vertices son un conjunto finito y pueden tener asociado algun valor.
Los edge son un conjunto de pares de V.
Un grafo es un par ordenado G=(V,E), donde:
- V: es el conjunto de vertices o de nodos.
- E: es el conjunto de aristas o arcos que relacionan los nodos.

Grado del grafo: es el número de vertices que contiene.
Grado del vertice: es el número de aristas que inciden en él.
Adyacencia: Dos vertices son adyacentes en el grafo si están en la misma arista del grafo.
La arista no es dirigida o es bidireccional.



Camino: Cualquier combinacion de vertices y aristas para ir de un nodo a otro.

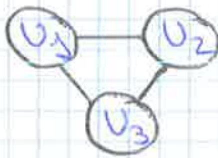


Caminos de 1 a 6:
[1,5,4,6] [1,2,3,4,6]
[1,2,5,4,6] [1,5,2,3,4,6]

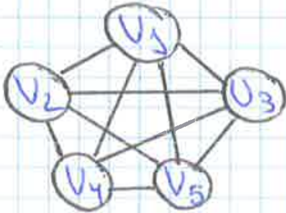
Longitud del camino: Número de aristas de un camino o número de vertices menos 1.
Camino simple: Cuando en un camino no se repiten vertices.
Circuito o ciclo: Cuando el vertice de origen y destino son el mismo.

- Tipos de grafo:

· Grafo simple: Es un grafo donde no hay bucles ni aristas paralelas.

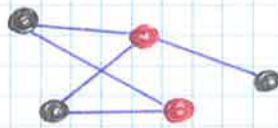


· Grafo completo: Es un grafo donde sus aristas unen todos los posibles vertices.

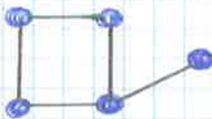


Todos los vertices estan unidos entre si.
Número de aristas $= \frac{n(n-1)}{2}$

· Grafo bipartito: es un grafo no dirigido que puede ser particionado en dos conjuntos de vertices, de forma que los vertices de cada subconjunto no estan conectados entre si y cada arista conecta un vertice V_1 con un vertice V_2.

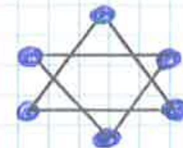


· Grafo conexo: es donde cada par de vertices estan conectados
· Grafo disconexo: cuando el grafo no es conexo



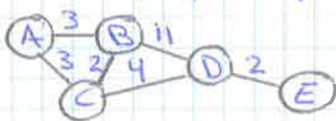
Conexo

Disconexo



Disconexo

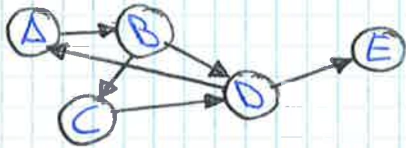
Grafo ponderado: grafo donde se le asigna un valor a las aristas, que representan pesos, costes, distancias ...



Longitud del camino: suma del peso de las aristas.

Camino de A hasta D:
[A, C, D] → Longitud = 3 + 4 = 7

· Grafos dirigidos: grafos donde las aristas tienen dirección.



Se indican $\langle V_1, V_2 \rangle$

V_1 = Origen
V_2 = Destino
V_1 es adyacente hacia V_2
V_2 es adyacente desde V_1

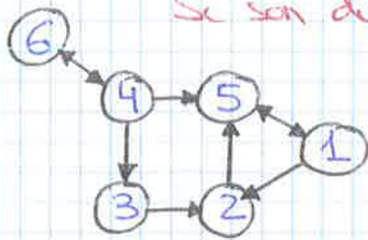
Adyacencia de entrada: Nodos que llegan al nodo. B: A (grado 1)
Adyacencia de salida: Nodos que salen del nodo. B: C, D (grado

Operaciones básicas:
- añadir (x): añadir un nodo.
- borrar (x): borrar nodo y referencias.
- adyacente (x, y): booleano si "x" e "y" son adyacente = True.
- vecinos (x): nodos adyacentes a "x".
- añadir (x, y): añadir arista que une "x" e "y".
- borrar (x, y): borrar arista que une "x" e "y".
- obtenerValorNodo (x): obtener el objeto del nodo "x".
- establecerValorNodo (x, a): asignar "a" como valor de "x".
- obtenerValorArco (x, y): obtener el valor del arco que une "x" e "y".
- establecerValorArco (x, y, a): asignar "a" como valor del arco que une "x" e "y".

- Formas de representar un grafo:

  · Matriz de adyacencia: Es una matriz bidimensional que representa los arcos entre cada par de nodos con un 1.

  Si son dirigidos se marca de origen a destino.



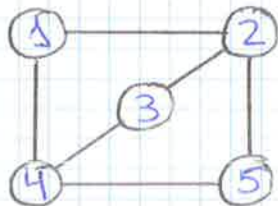
Origen (filas) / Destino (columnas)

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	0	0	0	0	1	0
3	0	1	0	0	0	0
4	0	0	1	0	1	1
5	1	0	0	0	0	0
6	0	0	0	1	0	0

Cuando borramos un nodo, hay que redimensionar la matriz y renombrar los nodos en una lista de índices.

  · Listas de adyacencia: Indica los nodos que son adyacentes a cada nodo.

  Si el grafo es dirigido indica los vertices hasta los que llega el arco.



Nodo	Lista de adyacencia
1	→ 2, 4
2	→ 1, 3, 5
3	→ 2, 4
4	→ 1, 3, 5
5	→ 2, 4

* Algoritmos y conceptos avanzados sobre grafos:

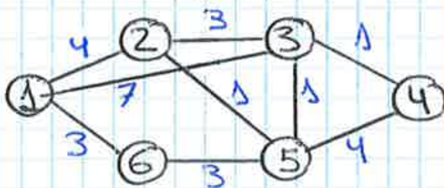
- Problemas sobre grafos:
  - Camino más corto:
    1. La suma de los pesos de los arcos sea mínima. (grafos ponderados).
    2. El número de nodos visitados sea mínimo. (grafos no ponderados).
  - Camino más largo:
    1. La suma del peso de los arcos sea máxima. (grafos ponderados).
    2. El número de nodos visitados sea máximo. (grafos no ponderados).

- Algoritmos de recorridos:
  - Prim y Kruskal: son algoritmos utilizados en grafos conexos ponderados no dirigidos.

  Estan asociados al "arbol recubridor mínimo":
  - Es un grafo en forma de arbol que contiene todos los nodos del grafo inicial.
  - La suma de las aristas del arbol es mínima.

Prim: Dos reglas:
- No formar ciclos.
- Asegurar aristas de menor peso.

Ejemplo:



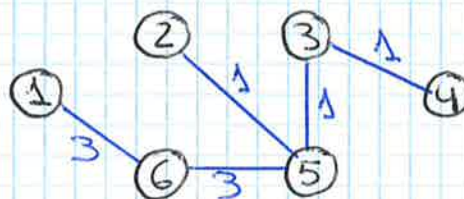
1º Elegimos un nodo de inicio, en este caso elegimos el 1.

2º Elegimos la arista de menor peso para conectar con otro nodo, en este caso (1-6).

3º Ahora buscamos la arista de menos peso para conectar los nodos ya agregados con el siguiente nodo (6-5).

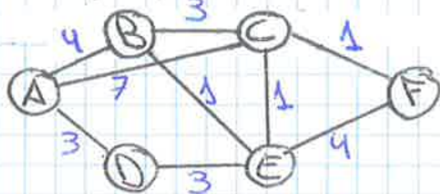
4º Seguimos con este mecanismo hasta agregar todos los nodos.

- Grafo resultante:



Kruskal: Este algoritmo es lo que se conoce como un algoritmo voraz.
En este algoritmo se ordenan las aristas por orden creciente de pesos y en cada etapa se decide que hacer.
Si el arco no forma un ciclo con los ya seleccionados se incluye, si no se descarta.

Ejemplo.



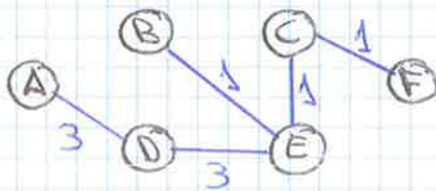
1º Ordenamos todas las aristas de menor a mayor:

BE	EC	CF	BC	DE	AD	AB	EF	AC
1	1	1	3	3	3	4	4	7

2º Selecionamos la primera arista, luego vamos seleccionando el resto de aristas siempre que no formen ciclos.

BE	EC	CF	BC	DE	AD	AB	EF	AC
1	1	1	3	3	3	4	4	7
SI	SI	SI	NO	SI	SI	NO	NO	NO

-Grafo resultante:

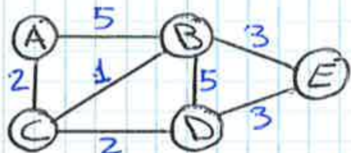


- Algoritmo de Dijkstra:

Usado para grafos conexos ponderados con pesos positivos.
Es utilizado para encontrar el camino más corto desde un nodo origen hasta el resto de nodos.
Tambien se usa para encontrar el camino más corto a un nodo en concreto.

Funcionamiento:

Partimos de un grafo conexo ponderado de "n" nodos



Nodo origen "x" (en este caso A)
Nodo actual "a" (al comienzo a = x)
Usamos un array para almacenar las distancias de "x" a "a".

Paso 1: Inicializamos el array con valores infinitos, excepto el origen que sera Ø.

Ø[A]	1[B]	2[C]	3[D]	4[E]
Ø	∞	∞	∞	∞

Paso 2: Indicamos el nodo actual.

Ø[A]	1[B]	2[C]	3[D]	4[E]
Ø	∞	∞	∞	∞

Paso 3: Recorremos los nodos adyacentes no marcados y apuntamos la distancia:

$D_b = 5$

$D_c = 2$

Paso 4: Calculamos la distancia tentativa. $Dt(v_i) = D_a + d(a, v_i)$

$Dt_b = Ø + 5 = 5$

$Dt_c = Ø + 2 = 2$

Paso 5: Comparamos Si $Dt(v_i) < D_{vi} \rightarrow D_{vi} = dt(v_i)$

Ø[A]	1[B]	2[C]	3[D]	4[E]
Ø	5	2	∞	∞

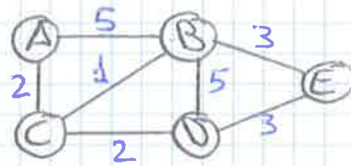
Paso 6: Marcamos el nodo actual como visitado.

1[A]	2[B]	3[C]	4[D]	5[E]
Ø	5	2	∞	∞

Volvemos al paso 3 mientras existan nodos no marcados visitados.

Paso 3 y 4:

∅	5	2	∞	∞
A	B	C	D	E



El nodo actual será el más pequeño no marcado.

$D_B = 1$ $Dt_C = 2 + 1 = 3$

$D_D = 2$ $Dt_D = 2 + 2 = 4$

Paso 5 y 6:

A	B	C	D	E
∅	3	2	4	∞

$Dt_B < D_B = 3 < 5$ True

$Dt_D < D_D = 4 < \infty$ True

Paso 3 y 4:

A	B	C	D	E
∅	3	2	4	∞

$D_E = 3$ $Dt_E = 3 + 3 = 6$

$D_D = 5$ $Dt_D = 5 + 3 = 8$

Paso 5 y 6:

A	B	C	D	E
∅	3	2	4	6

$Dt_D < D_D = 8 < 4$ False

$Dt_E < D_E = 6 < \infty$ True

Paso 3 y 4:

A	B	C	D	E
∅	3	2	4	6

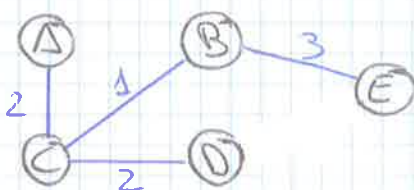
$D_E = 3$ $Dt_E = 4 + 3 = 7$

Paso 5 y 6:

A	B	C	D	E
0	3	2	4	6

$Dt_D < D_D = 7 < 6$

Grafo resultante:



Distancias de A al resto de nodos

A	B	C	D	E
∅	3	2	4	6

Ciclos Eulerianos:

Camino que recorre todas las aristas de un grafo, pasando una vez por cada una y llega al vertice de salida.
Los vertices pueden repetirse.

Un grafo conexo es euleriano si todos sus vertices son de grado par.
Un grafo conexo tiene recorrido euleriano no cerrado si tiene exactamente, dos vertices de grado impar.

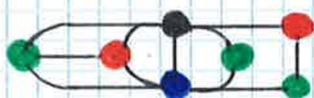
Camino Hamiltonianos:

Camino que recorre todos los vertices pasando solamente una vez por cada uno.
Si el ultimo vertice y el primero son el mismo, es un ciclo Hamiltoniano.

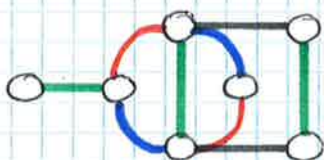
Coloración de grafos:

Consiste en etiquetar el grafo con colores en vez de números o letras.
La representación de los colores es utilizando números.
Hay tres modos:

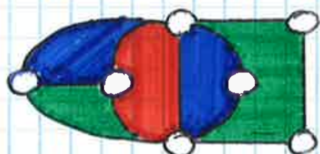
- Vertice coloración:
  - Se colorean los vertices, teniendo en cuenta que dos vertices adyacentes no pueden tener el mismo color.



- Aristas coloración:
  - Se colorean las aristas, teniendo en cuenta que dos aristas adyacentes no pueden tener el mismo color.



- Cara coloración
  - Se colorean las caras, teniendo en cuenta que dos caras con frontera común no pueden tener el mismo color.



- Algoritmos de busqueda y recorridos:

Consiste en visitar los nodos de un grafo.

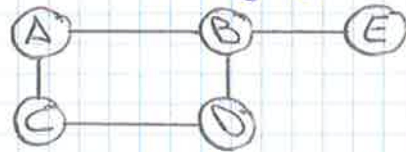
- La busqueda se detiene cuando encuentra el nodo requerido.
- El recorrido visita todos los nodos del grafo.

  En los recorridos de grafos se puede visitar un nodo más de una vez, pero hay que intentar que sea lo menos posible.

Recorrido en anchura:

Partimos de un nodo raiz (cualquier nodo que elijamos) y se procede a explorar todos los vecinos de dicho nodo. Para cada uno de los vecinos, se exploran nuevamente todos sus vecinos, y asi hasta recorrer el grafo.

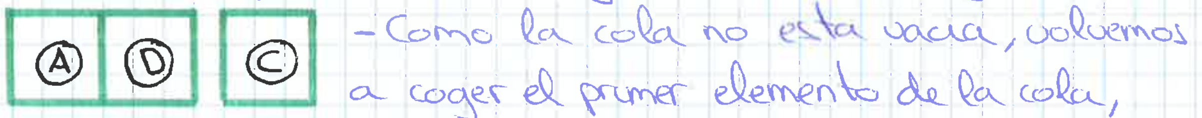
Algoritmo:



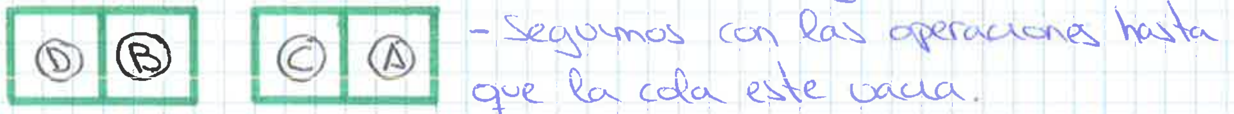
- Usaremos una cola como extructura auxiliar.
- Añadimos el nodo raiz (C) a nuestra cola.

Cola Nodos visitados

- Sacamos el primer elemento de la cola y comprobamos sus vecinos, metiendo "A" y "D" a la cola y "C" a visitados



- Como la cola no esta vacia, volvemos a coger el primer elemento de la cola, comprobamos sus vecinos no visitados y movemos "A" a visitados



- Seguimos con las operaciones hasta que la cola este vacia.

- Sacamos "D" y visitamos sus vecinos no visitados (ninguno).

- Sacamos "B" y visitamos sus vecinos no visitados (E).

- Sacamos "E" y visitamos sus vecinos no visitados (ninguno)
- Metemos "E" en la lista y como la cola esta vacia terminamos

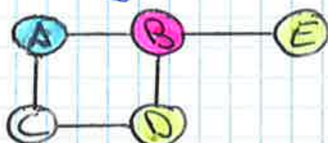
- Recorrido en profundidad:

Consiste en recorrer todos los nodos de forma recursiva a traves de un camino dado.

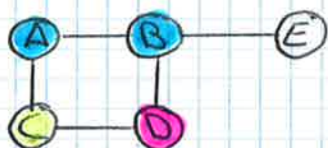
Cuando los nodos del camino se acaban, volvemos atras. (Backtracking)

Algoritmo:

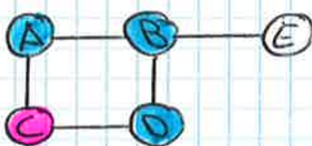
- Elegimos un nodo raiz, en este caso "A".
- Tenemos dos opciones de recorrido: "B" o "C"
- Elegimos una (en este caso alfabeticamente): "B"



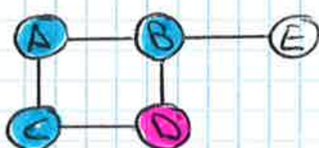
- Ahora otras dos posibilidades de camino: "E y D"
- Elegimos D.



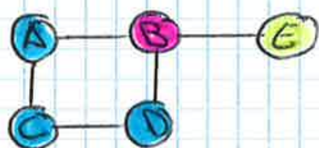
- Dos opciones de camino: "C" y "B" (ya visitado)
- Nos queda "C".



- No tenemos opciones de camino no visitado, volvemos hacia atras "D".

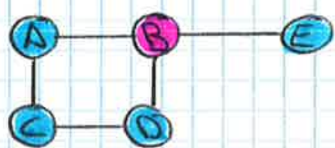


- En "D" tampoco tenemos opciones de no visitado, volvemos atras "B".

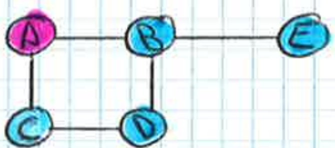


- Tenemos la opción de "E", la escogemos.

- No tenemos opciones de no visitadas, volvemos atras, "B".



- No tenemos opciones de no visitadas, volvemos atra, "A".



- Hemos vuelto al inicio "A".
- Finalizamos.