1. Resuelve las siguientes cuestiones:

a) Explica brevemente un problema cuya estructura ideal para resolverlo sea una lista.

Sabemos que las listas son un tipo de estructura de datos abstracta (TAD) que constan de una secuencia de nodos donde almacenar datos y que contienen unas referencias (punteros) a los nodos posteriores o previos.

Todos sus datos han de ser del mismo tipo, pues son estructuras homogeneas. Ademas, son estructuras dinamicas, esto quiere decir, que pueden crecer y decrecer de forma "infinita".

Sabiendo todo esto, un problema cuya estructura ideal a utilizar seria una lista podria ser por ejemplo un "ToDo", una lista de tareas. En ella iriamos insertando las tareas a realizar y a medida que fueramos realizandolas podriamos ir eliminandolas o marcandolas como ejecutadas.

b) Partiendo de una lista vacía, pon un ejemplo de como funcionaría una lista para una inserción normal, una inserción por posición y un borrado por posición (que no tenga éxito). Deses incluir el funcionamiento de forma gráfica y con pseudocódigo.

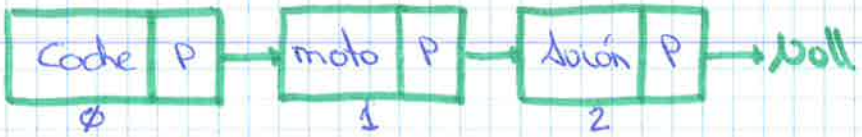
Vamos a crear una lista de medios de transporte (coche, moto, avión)
Lo primero sería inicializar nuestra lista

Lista transportes = new Lista ();

Despues insertaríamos nuestros tres elementos.

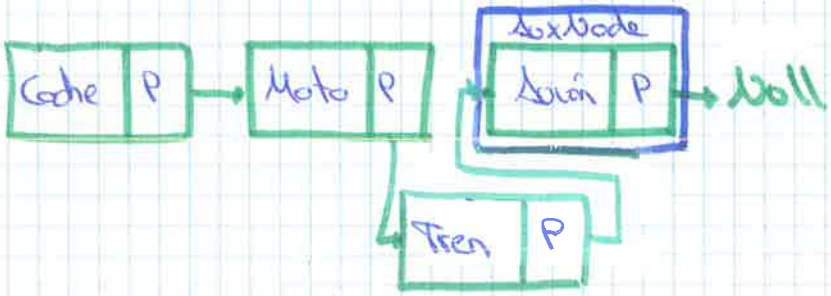
Insertar «coche»
Insertar «moto»
Insertar «avión»

P = Puntero




Ahora insertaremos "Tren" en la posición 2.

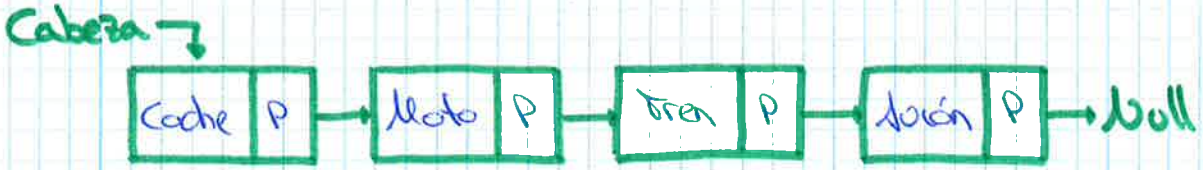
1. Iremos a la posición en la que queremos insertar el nuevo elemento
2. Guardamos el elemento actual en un nodo auxiliar. (AuxNode)
3. El puntero del elemento anterior lo apuntamos al elemento a insertar.
4. El elemento insertado lo apuntamos a AuxNode




```
Proceso insertarNodo [2]{
  auxNode = nodo[2];
  Puntero nodoAnterior = nuevoNodo;
  Puntero nuevoNodo = auxNode;
}
```

Ahora vamos a borra "Barco" de nuestra lista, pero como no esta, se produciria un error y devolvera un false.

1. Nos colocamos en la cabeza de nuestra lista y comparamos el elemento buscado con el elemento, si coincide, procedemos con el borrado, si no coincide pasamos al siguiente elemento y volvemos a comparar.
2. Repetimos el paso 1 hasta encontrar el elemento y borrarlo o hasta que lleguemos al final de la lista.
3. Si encontramos el elemento y lo borramos, devuelve True, si no, devuelve False, por lo tanto, nuestra función sera boolean.




1. Usamos coche y comparamos, como no es pasamos al siguiente.
2. Repetimos la operación anterior hasta encontrarlo.
3. Como llegamos al último y el siguiente apunta a null, salimos de la función devolviendo "False".

```
While (punteroNodo. != null){
    if (nodoActual == nodoBuscado){
        función eliminar nodo;
        return True;
    else{
        punteroNodo = = punteroNextNodo;
    }
}
return False;
```

c) Escribe el código java para dichas operaciones:

```
// Inicializamos nuestra lista:
Lista myList = new Lista(); //esto estaria en el main.

// Insertamos los elementos a nuestra lista
myList.addNode(coche);
myList.addNode(moto);
myList.addNode(avión);
```

} Esto en el main.

```
// Metodo de agregar elemento de la clase Lista:

Public void addNode(String data){
  Nodo myNode = new Nodo(data);
    If (first == null){
      first = myNode;
      last = myNode;
      }else{
        last.setNextNode(myNode);
        last = myNode;
      }
}

// Metodo de insertar elemento por posición:
Public void insertNode(int n; String data){
  Nodo myNode = new Nodo(data);
  if (first == null){
     first = myNode;
  }else{
    int position = 0
   Nodo auxNextNode = first;
  Nodo node = auxNextNode;
    While (position < n && auxNextNode != null){
       node = auxNextNode;
       auxNextNode = auxNextNode.getNextNode();
       position ++;
    {
    node.setNextNode(myNode);
    myNode.setNextNode(auxNextNode);
}}
```

```
//Método de borrar elemento:
Public boolean deleteNode (data){
if (first == null){
  return false;
}else{
  Nodo auxDeleteNode = first;
  Nodo auxNode=first;
  while (auxNode!=null){
    if(auxDeleteNode.getData == data){
      auxNode.setNextNode = auxDeleteNode.getNextNode;
      auxDeleteNode = null;
      return true;
      }else{
       auxNode = auxDeleteNode;
       auxDeleteNode = auxDeleteNode.getNextNode;
       }
  }
  return false;
}
```

Resuelve las siguientes cuestiones:

a) Piensa en un problema que podría resolverse con una tabla Hash y comenta cuál sería la clave y el valor.

Las tablas hash son una estructura de datos que funcionan asociando claves con valores.
Son muy utilizadas para optimizar la búsqueda, 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 tiene asociada una clave única, permitiendo que diferentes entradas de una misma tabla tengan tambien diferentes claves. La clave nos permitira recuperar el valor.
Un problema que podria resolver una tabla hash seria la organización de un grupo de usuarios.
Su clave podria ser su DNI, NIF o Pasaporte y su nombre el valor.

b) Defina una función Hash para una tabla Hash cerrada.
Las tablas hash cerradas suelen estar basadas en arrays, con un número determinado del tamaño que viene dado por el número de elementos el siguiente valor primo.
Para este ejemplo, vamos a utilizar un array de 10 posiciones, que aunque no sea primo, nos permitira realizar las cuentas de manera mas sencilla.
En este caso vamos a utilizar una función hash con resolución de colisiones de exploración lineal.
Indice = clave % 10, en este caso vamos a utilizar el DNI como clave.

| Clave | Valor | INDICE |
|---|---|---|
| 16602395 | Jose | 16602395 % 10 = 5 |
| 16521220 | Juan | 16521220 % 10 = 0 |
| 16277341 | María | 16277341 % 10 = 1 |

En este caso no hay colisiones, pero para evitarlas, utilizaríamos la función hash siguiente:
Indice = (clave + i) % n = (Clave + i) % 10

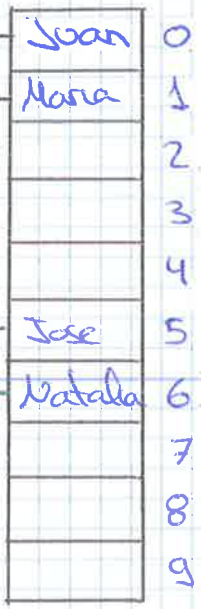
c) Gráficamente muestra como funcionaría una Tabla Hash cerrada. Utiliza un número suficiente de elementos para poder representar diferentes situaciones.

Como en el ejercicio anterior, vamos a utilizar la función hash con resolución de colisiones de exploración lineal.
La clave será el DNI sin letra y el valor el nombre.
Función hash:
Indice = (clave + i) % 10

| CLAVE | VALOR | FUNCION | INDICE |
|---|---|---|---|
| 16602395 | Jose | (16602395 + 0) % 10 = | 5 |
| 16521220 | Juan | (16521220 + 0) % 10 = | 0 |
| 16773741 | Maria | (16773741 + 0) % 10 = | 1 |
| 16292535 | Natalia | (16292535 + 0) % 10 = | 5 |

Tenemos colisión, así que incrementamos "i" para avanzar a la derecha hasta encontrar un hueco libre.

| | | | |
|---|---|---|---|
| 16292535 | Natalia | (16292535 + 1) % 10 = | 6 |




d) Gráficamente, muestra como se realiza el borrado y cómo se buscaría un elemento que se almacenó con colisión después de un elemento que ya se ha borrado (con el que colisiona).

Siguiendo con el ejemplo anterior de tabla hash cerrada y resolución de colisiones con exploración lineal, el borrado de elementos se realiza como lo que se conoce como borrado perezoso, que consiste en marcar la posición como borrada, dando a entender que anteriormente hubo un elemento insertado, así, la búsqueda no se detendrá hasta encontrar una casilla vacía sin borrado perezoso o hasta encontrar el dato a borrar.

Partimos de nuestra tabla hash anterior y vamos a borrar a "Jose".

| CLAVE | VALOR | FUNCION | INDICE |
|---|---|---|---|
| 36602395 | Jose | (36602395+0)%10 | 5 |
| 36521220 | Juan | (36521220+0)%10 | 0 |
| 36773741 | Marca | (36773741+0)%10 | 1 |
| 36292535 | Natalia | (36292535+1)%10 | 6 |

| TABLA | |
|---|---|
| Juan | 0 |
| Marca | 1 |
| | 2 |
| | 3 |
| | 4 |
| Jose | 5 |
| Natalia | 6 |
| | 7 |
| | 8 |
| | 9 |

Ahora borramos a José, así que aplicaremos nuestra función hash de nuevo, compararemos el dato a ver si coincide con el que queremos borrar y si coincide lo borraremos, si no coincide seguiremos recorriendo la tabla hasta encontrar una casilla vacia sin borrado perezoso.

(36602395+0) % 10 = 5

Vamos a la posición 5 y como coincide, borramos el dato y dejamos la marca de borrado perezoso activa (Borrado = True)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Juan | Maria | | | | Borrado = true | Natalia | | | |

Ahora vamos a buscar a Natalia.

(36292535+0) % 10 = 5

Vamos a la posición 5 y esta vacia, pero el borrado perezoso nos indica que anteriormente hubo un dato en dicha posición, con lo que seguiremos con la exploración lineal hasta encontrar el dato o encontrar una posición vacia sin borrado perezoso.

(36292535+1) % 10 = 6

Vamos a la posición 6, comparamos el dato y como coincide, lo hemos encontrado.

2. Resuelve las siguientes cuestiones:

a) Explica que es un ABB.

Los ABB son arboles binarios de busqueda.

Un arbol es una estructura de datos no lineal porque organiza los datos de forma jerarquica.

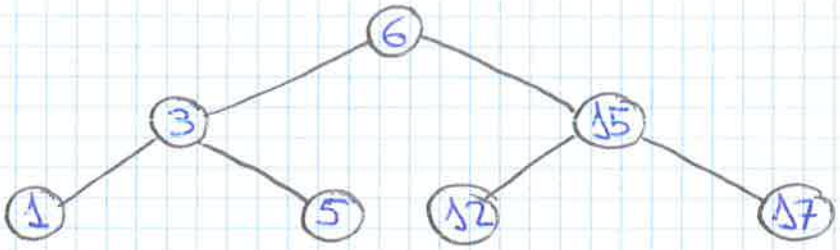
Los arboles binarios de busqueda, tambien llamados arboles ordenados, son arboles que cumplen la propiedad de ordenación.

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

A diferencia que los arboles n-arios, los ABB solo tendran como maximo dos hijos (izquierdo, derecho).

El recorrido por defecto de los ABB es inorder, porque es el que nos muestra los datos ordenados.

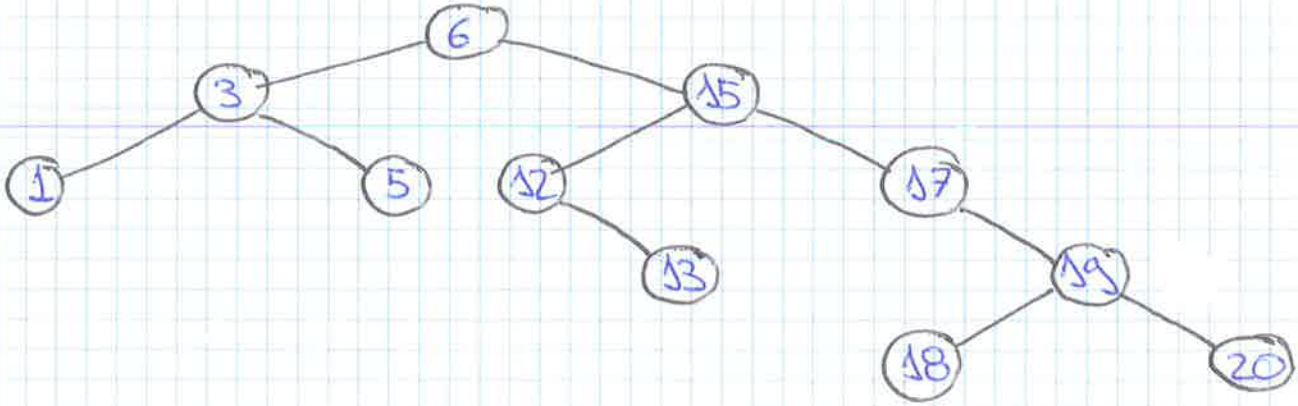
b) Graficamente, representa un ABB con los nodos:
6, 15, 12, 17, 3, 1, 5.




¿Que tipo de arbol es?
Es un arbol binario perfecto, porque todas sus hojas estan a la misma profundidad.

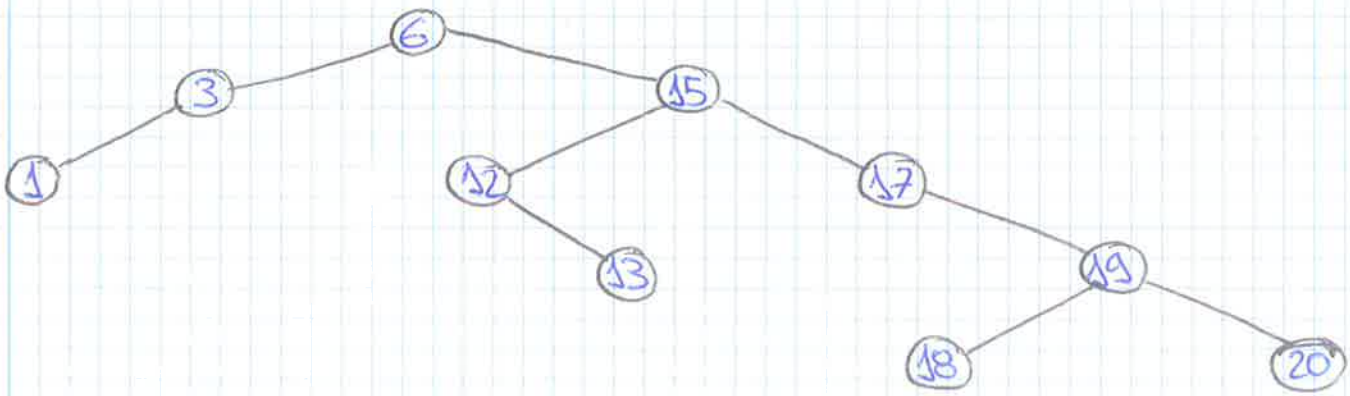
Ahora inserta: 19, 18, 13 y 20.

Partimos del arbol inicial:




Ahora es un arbol ABB, pero no AVL porque no esta equilibrado.

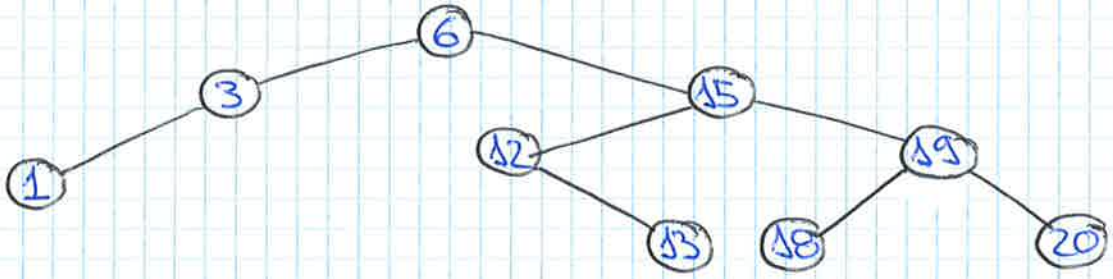
c) Ahora vamos a realizar borrados.
Borra el 5.
Como el nodo 5 es una hoja, bastara con borrarlo.



Borra el 17:

Este caso es diferente al anterior, ya que el nodo a borrar tiene un subarbol hijo.

En este caso borraremos el nodo y daremos su hijo en adopción al padre del nodo a borrar.

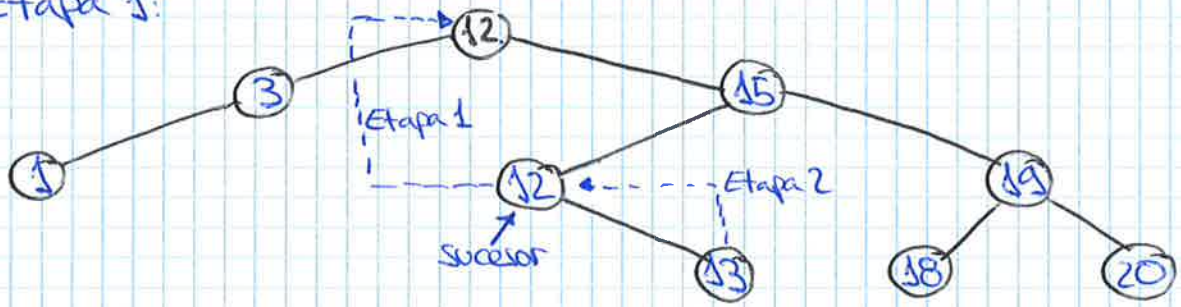



Borra el 6:

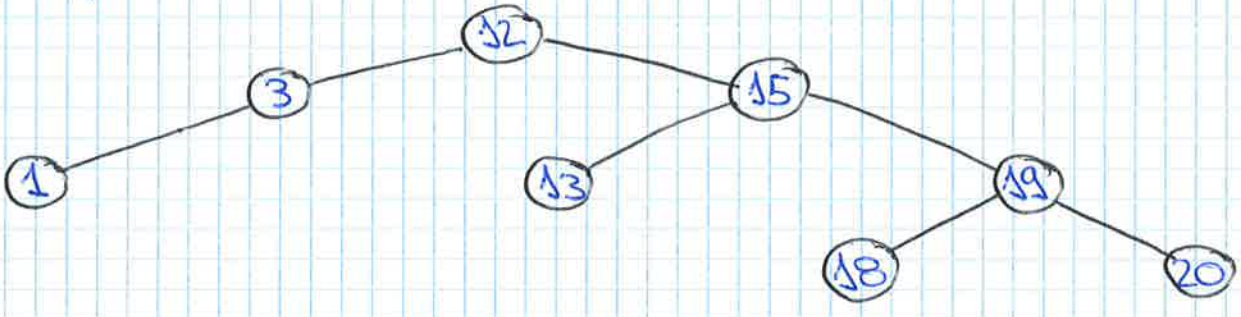
Este caso es el más complejo, pues nuestro nodo a borrar tiene dos subarboles, con lo que habra que realizar el borrado en dos etapas:

- Sustituimos el nodo a borrar por el sucesor.
  El sucesor es el nodo mas pequeño del subarbol derecho, en este caso el "12".
- Una vez reemplazado el nodo, damos el hijo del sucesor en adopción a su abuelo, es decir el 13 en adopción al 15.

Etapa 1:




Etapa 2:



Dibuje una cola dinámica que contenga 3 elementos de tipo entero.

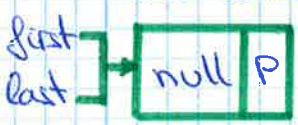
Las colas son estructuras de datos del tipo FIFO (First in First out), es decir, el primer elemento que entra es el primero que sale.

Las colas pueden ser estáticas, almacena los elementos estableciendo previamente un tamaño, o dinámicas, se construye mediante la consecución de nodos (como en las listas).

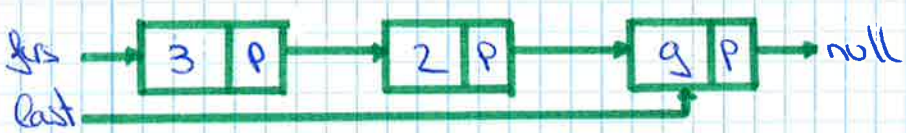
Las operaciones de las colas son diferentes de implementar dependiendo si son estáticas o dinámicas.

Para este ejercicio lo haré con una cola dinámica.

En nuestra clase tendremos dos punteros, uno que apunte al primer nodo y otro que apunte a último nodo. Inicialmente, al estar vacía, los dos apuntaran a la posición ∅ (null).




Ahora introduciremos los elementos (3, 2, 9):

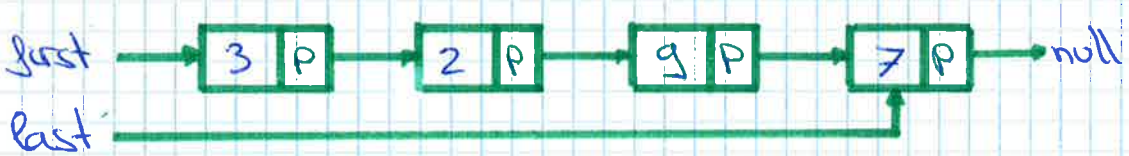



Explique detalladamente y muestre gráficamente como se encolaría un elemento más.

Partiendo de la cola del ejercicio anterior, vamos a proceder a encolar el elemento "7".

El primer paso sería comprobar si nuestra cola esta vacía, si lo estuviera, insertaríamos el elemento y su puntero apuntaría a "null" y los punteros first y last apuntarían al nuevo elemento.

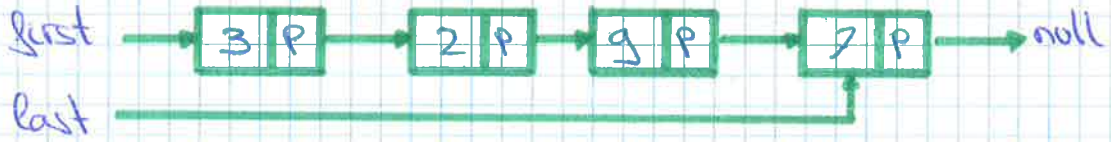
Como nuestra cola no esta vacía y last nos marca el último elemento, lo que hacemos es que el puntero del ultimo elemento apunte al nuevo elemento y last lo actualizamos y apunta tambien al nuevo elemento, que al insertarlo, pasa a ser el último.



Explique de forma detallada y muestre de forma gráfica cómo, a partir de la cola del ejercicio anterior, se desencolaría un elemento.

Como he explicado anteriormente, las colas son estructuras de datos del tipo FIFO (first in first out), el primer elemento en entrar sera el primero en salir.

En este caso hemos encolado los elementos en el siguiente orden: {3, 2, 9, 7}, con lo que nuestra cola tendra el siguiente aspecto.

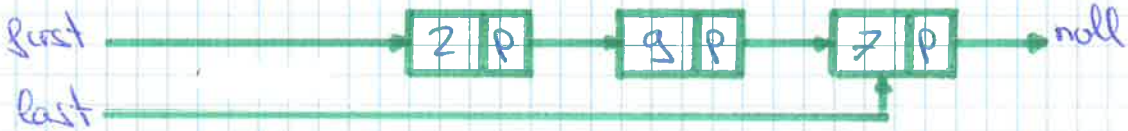



Entonces, nuestro primer elemento a desencolar seria el "3".

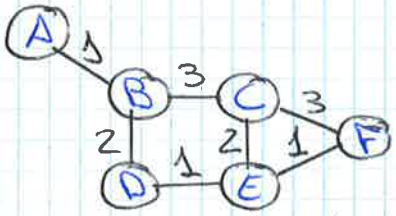
El método es muy sencillo, pues conocemos el primer elemento de la cola, que seria el siguiente a desencolar.

Llamaremos a nuestra función desencolar que lo que haría, seria devolvernos el elemento "3", seguidamente actualizaria first con el valor del puntero de "3", que es "2".

Nuestra cola ahora quedaria así.



Dibuje un grafo conexo ponderado no dirigido




Haga su recorrido en anchura.

Para realizar el recorrido en anchura, partire de un nodo raiz que yo elegiré, en este caso el nodo "1".

Despues visitaremos sus vecinos. Para cada uno de los vecinos exploraremos nuevamente a todos los vecinos no visitados, y asi hasta recorrer el grafo.

Utilizaremos una cola como estructura auxiliar y una lista para almacenar los visitados.

1º- Introducimos el nodo elegido a la cola

Cola | Visitados


2º- Desencolamos el siguiente elemento de la cola, lo introducimos en la lista de visitados y añadimos sus vecinos a la cola.

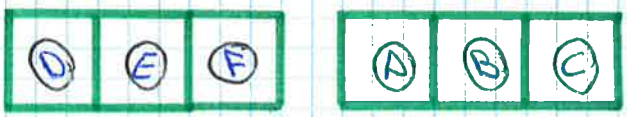
Cola | Visitados


3º- Como la cola no esta vacia, repetimos el paso 2º.

Cola | Visitados

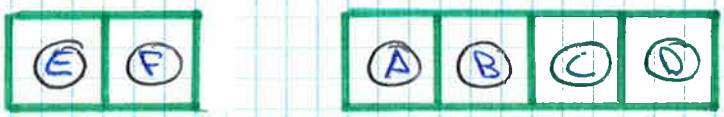

4º- Seguimos con las operaciones hasta que la cola quede vacia.

Cola | Visitados



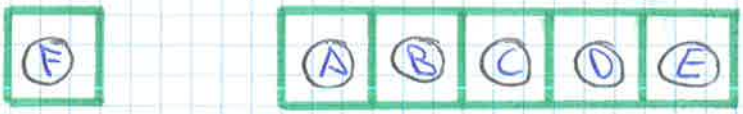

5º- Sacamos "D" y visitamos sus vecinos.

Cola | Visitados



6° Sacamos "E", pero no encolamos porque ya estan visitados o en la cola.

Cola | Visitados




7. Pasamos "F" a visitados y no encolamos nada porque sus vecinos estan visitados

Cola | Visitados


8° Como la cola esta vacia, hemos terminado.

Explique y represente su diseño en una matriz de adyacencia:

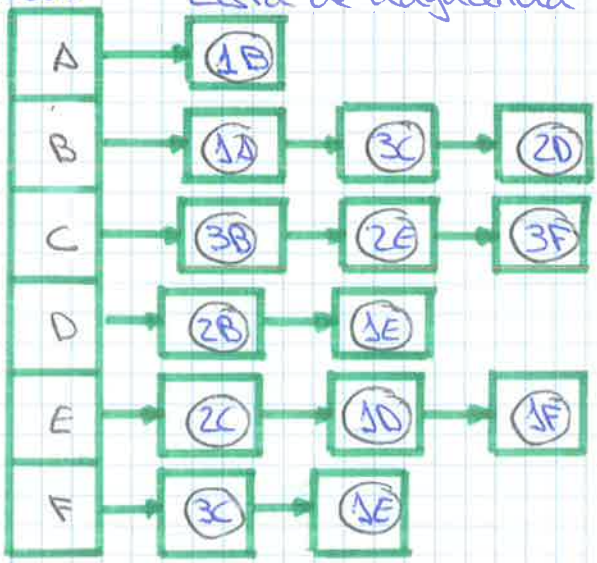
| Origen \ Destino | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | ∅ | 1 | ∅ | ∅ | ∅ | ∅ |
| B | 1 | ∅ | 3 | 2 | ∅ | ∅ |
| C | ∅ | 3 | ∅ | ∅ | 2 | 3 |
| D | ∅ | 2 | ∅ | ∅ | 1 | ∅ |
| E | ∅ | ∅ | 2 | 1 | ∅ | 1 |
| F | ∅ | ∅ | 3 | ∅ | 1 | ∅ |

Como nuestro grafo es ponderado, en nuestra matriz señalamos los pesos de las aristas adyacentes. Las filas marcan el nodo de origen y las columnas el nodo de destino. Si no hay adyacencia se pone un ∅ y si hay se pone un 1 o el peso si es ponderado.

Explique y represente su diseño en una lista de adyacencia:

En este ejercicio se puede hacer de varias maneras, se podria implementar con ArrayList de arrayList, LinkedList de LinkedList, ArrayList de LinkedList, etc.
Nosotros vamos a representarlo con ArrayList de linkedList.

Nodo | Lista de adyacencia



Para una tabla Hash cerrada, defina un nodo que permita trabajar con esta estructura:
Sabiendo que las tablas Hash necesitan de clave y valor, nuestro nodo debera contener como mínimo estos dos datos.
En nuestro caso vamos a utilizar un dato de tipo String para la clave y un dato de tipo String para el valor.
La clave sera el DNI y el objeto el nombre.

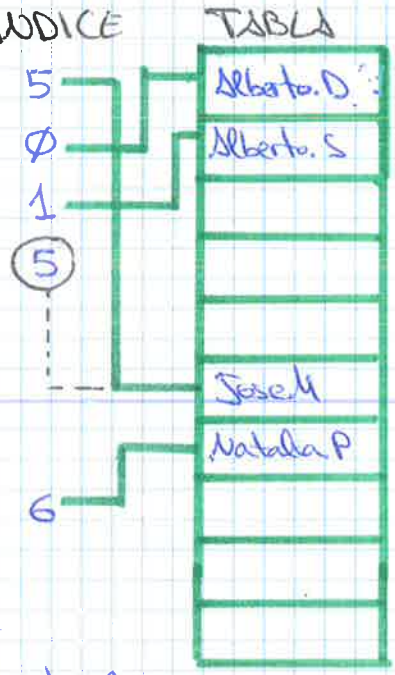
Defina una función Hash de exploración cuadratica:
Las funciones hash de exploración cuadrática son utilizadas para resolver colisiones de forma pasiva, es decir, cuando se ha producido se resolve.
Indice = (hash + $i^2$) % n
Nuestra función Hash se calculara a partir de la clave, en este caso el DNI, del que sacaremos el codigo ASCII de cada caracter y los sumaremos.
Como no me se el código ASCII, voy a inventarme los valores.
Ejemplo:
DNI ⟶ 166023954 → CLAVE
NOMBRE → JOSE MANUEL → VALOR
Hash = ASCII(DNI[0]) + ASCII(DNI[1]) + ... + ASCII(DNI[8])
Hash = 13 + 10 + 2 + 1 + 33 + 8 + 5 + 11 + 2 = 85
Indice para el primer elemento sin colisión:
(85 + $0^2$) % 10 = 5, suponiendo que nuestra tabla tiene un tamaño de 10.

Inserte 4 elementos en dicha tabla hash de forma gráfica.
Para entender bien la inserción de elementos en las tablas Hash y la resolución de colisiones, vamos a provocar una colisión y ver como nuestra función con resolución de conflictos cuadrática lo resuelve.

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

Partimos de un array de 10 posiciones para simplificar las operaciones y los valores del sumatorio ASCII del ejercicio anterior serán inventados por el mismo motivo.

| CLAVE | VALOR | FUNCION |
|---|---|---|
| 16602395Y | Jose Manuel | (85+0)%10 |
| 16521520E | Alberto Diaz | (20+0)%10 |
| 16722547A | Alberto Serna | (71+0)%10 |
| 16213368W | Natalia Padilla | (65+0)%10 |




Vemos como tenemos una colisión a la hora de introducir a Natalia, así que vamos a calcular el indice incrementando "i".

(65+1)%10

Con esto ya hemos evitado la colisión.
En la tabla hemos colocado solo el valor por problemas de espacio, pero debería contener tanto la clave como el valor.

Borre un elemento y explique la utilidad del borrado perezoso en este caso.
Siguiendo con el ejemplo anterior de tabla Hash y resolución de colisiones con exploración cuadratica, el borrado de elementos se realiza como lo que se conoce como borrado perezoso, que consiste en marcar la posición como borrada, dando a entender que anteriormente hubo un elemento insertado, así, a la hora de buscar un elemento, cuando llegue a una posición vacia con borrado perezoso no se detendra.

Partimos de la tabla Hash anterior.

| CLAVE | VALOR | FUNCION | INDICE |
|---|---|---|---|
| 166023954 | Jose Manuel | (85+0)%10 | 5 |
| 16521520E | Alberto Diaz | (20+0)%10 | 0 |
| 16722517A | Alberto Sena | (71+0)%10 | 1 |
| 16213368D | Natalia Padilla | (65+1)%10 | 6 |

TABLA

| | |
|---|---|
| 0 | Alberto Diaz / 16521520E |
| 1 | Alberto Sena / 16722517A |
| 2 | |
| 3 | |
| 4 | |
| 5 | Jose Manuel / 166023954 |
| 6 | Natalia Padilla / 16213368D |
| 7 | |
| 8 | |
| 9 | |

Ahora vamos a borrar a "Jose Manuel", así que aplicaremos nuestra función Hash de nuevo para localizar el indice, compararemos el dato a ver si coincide con el que queremos borrar y si coincide lo borraremos, si no coincide, seguiremos recorriendo la tabla hasta encontrar una casilla vacia sin borrado perezoso.

(85+0)% 10 = 5

Vamos a la posición y como coincide, borramos el dato, dejando una marca de borrado perezoso activa (Borrado=True).

| | |
|---|---|
| 0 | Alberto Diaz |
| 1 | Alberto Sena |
| 2 | |
| 3 | |
| 4 | |
| 5 | Borrado=True |
| 6 | Natalia Padilla |
| 7 | |
| 8 | |
| 9 | |

Supongamos que queremos buscar a Natalia Padilla, aplicamos la función hash para encontrarla.

(65+0)% 10 = 5

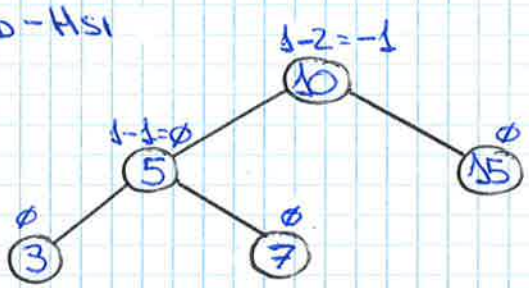
Vamos a la posición 5 y esta vacia, pero el borrado perezoso nos indica que anteriormente hubo un dato en dicha posición, con lo que seguiremos con la exploración lineal hasta encontrar el dato o una posición vacia sin borrado perezoso.

(65+1)% 10 = 6

Vamos a la posición 6, comparamos el dato y como coincide, lo hemos encontrado.

Dibuje un arbol AVL e indique porque lo es:
Un arbol AVL es un arbol que cumple con las propiedades de ordenación y equilibrio.

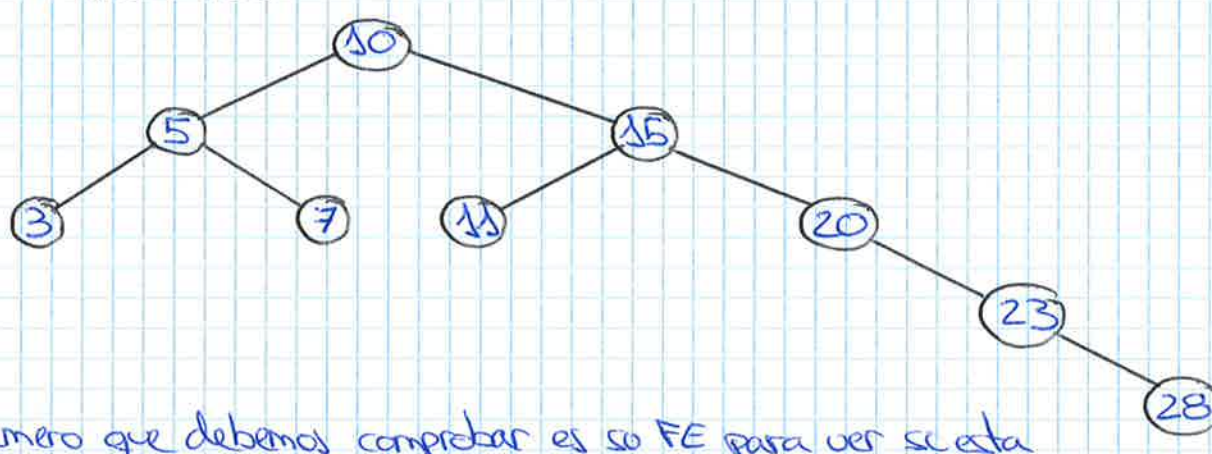
- Ordenación: Los nodos de mayor valor se insertan en el subarbol del hijo derecho y los de menor valor a la izquierda.
- Equilibrio: Cuando su factor de equilibrio es 1, 0 ó -1.
  El factor de equilibrio es la resta de la altura del subarbol derecho menos la altura del subarbol izquierdo.
  $FE = H_{SD} - H_{SI}$




- Las hojas siempre tienen $FE = 0$
- El resto de nodos $FE = S_D - S_I$

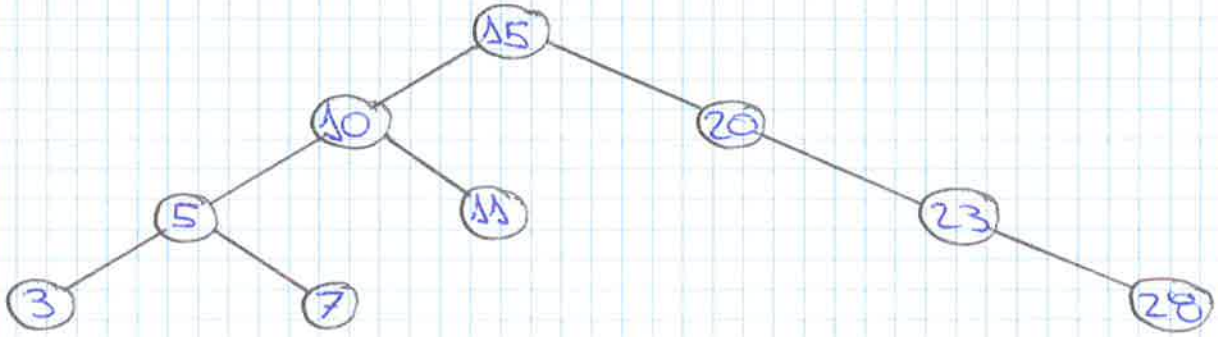
Podemos decir que nuestra arbol está balanceado porque todos sus nodos cumplen el FE.
Este equilibrio nos garantiza un coste computacional $O(\log n)$ en la busqueda.
En resumen, los arboles AVL son arboles ABB que estan siempre balanceados y su FE es menor o igual a 1 punto.

Inserte los elementos necesarios en el arbol anterior para que sea necesaria una rotación a la izquierda e indique porque es necesaria.



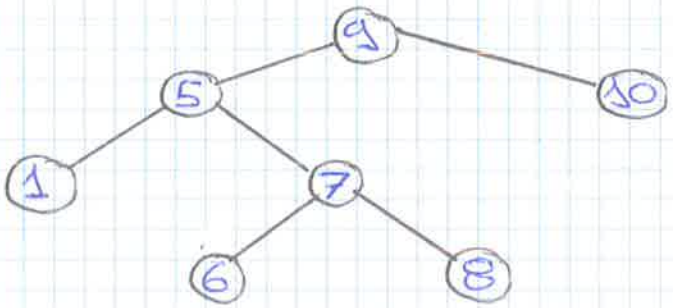

Lo primero que debemos comprobar es su FE para ver si esta o no balanceado y como debemos realizar la rotación.
$FE = H_{SD} - H_{SI} = 4 - 2 = 2$
No esta equilibrado y debemos realizar una rotación simple a la izquierda.

Realice el balanceo del árbol obtenido en el punto anterior:
La raiz sera ahora el hijo derecho.
El hijo derecho de la raiz sera el nieto del actual
El hijo izquierdo sera la raiz anterior.
Del hijo izquierdo de la raiz, su hijo izquierdo sera el mismo y el hijo izquierdo del subarbol derecho anterior sera el hijo derecho del subarbol izquierdo.
Arbol resultante:



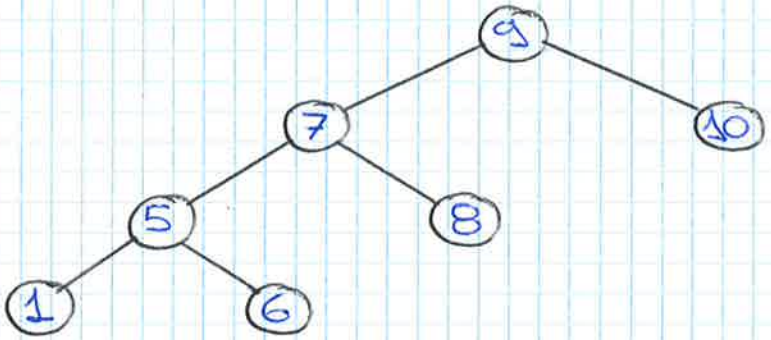

Una vez realizada la rotación comprobamos el FE.
$FE = H_{sd} - H_{si} = 3 - 3 = \emptyset$
Comprobamos que esta balanceado correctamente.

Realice un ejemplo de rotación doble sobre un nuevo arbol y explique el proceso.
Las rotaciones dobles consisten en dos rotaciones simples, en caso de ser doble a la derecha serca una simple a la izquierda del subarbol izquierdo y luego otra simple a la derecha con el arbol resultante.

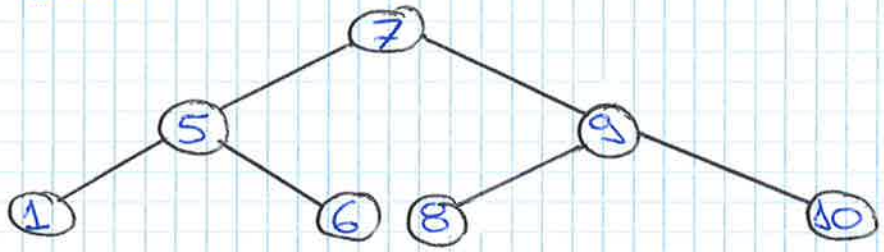



Comprobamos el FE. $FE = 1 - 3 = -2$ esta desequilibrado.
Primero rotamos el subarbol izquierdo a la izquierda.

Arbol resultante:

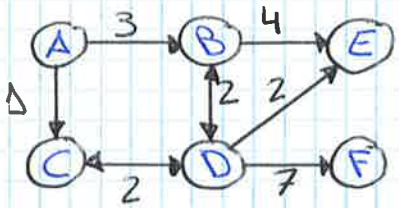



Ahora realizamos la rotación simple a la derecha del arbol resultante.
Arbol final:




Podemos comprobar que nuestro arbol final es un arbol binario perfecto, porque todas sus hojas estan a la misma profundidad, aunque esto no es una consecuencia directa de las rotaciones dobles, en este caso fue casualidad.

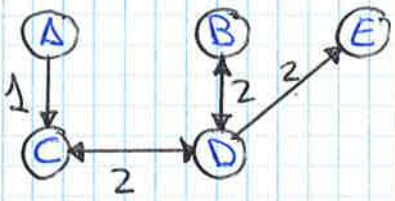
Representa gráficamente un grafo ponderado dirigido con 6 nodos y el número de enlaces que consideres.




Indica un cámino que visite 5 de los nodos. Indica su longitud y explica como lo has hecho.

Comenzare del nodo "A", para recorrerlo ire siguendo las aristas menos pesadas hasta realizar el camino solicitado.

- De "A" voy a "C"
- De "C" voy a "D"
- De "D" voy a "B"
- De "D" voy a "E"




Este recorrido lo he realizado siguiendo un sucedaneo del algoritmo de KruSkal, digo sucedaneo, porque Prim y Kruscal son algoritmos para grafos ponderados no dirigidos.
El camino resultante no es un camino simple porque repite nodos para poder llegar a los 5 vertices.
Su longitud sera

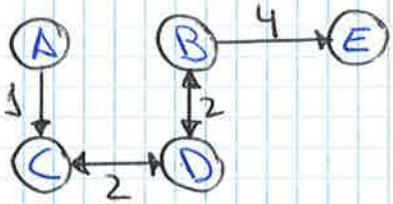
A → C = 1
C → D = 2
D → B = 2
B → D = 2
D → E = 2

} Total una longitud de 9 para visitar 5 nodos

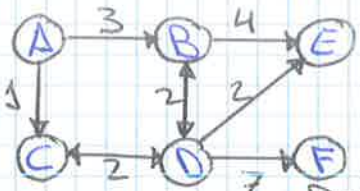
Si hubiera querido realizar un camino simple no hubiera repetido nodos y hubiera seguido el siguiente camino.

A → C = 1
C → D = 2
D → B = 2
B → E = 4

} Total una longitud de 9



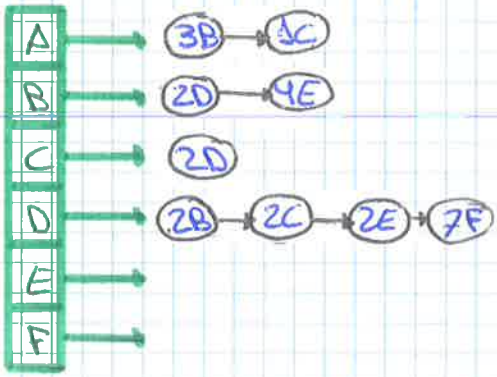
Representa tu grafo con una matriz de adyacencia:




| Origen \ Destino | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | ∅ | 3 | 1 | ∅ | ∅ | ∅ |
| B | ∅ | ∅ | ∅ | 2 | 4 | ∅ |
| C | ∅ | ∅ | ∅ | 2 | ∅ | ∅ |
| D | ∅ | 2 | 2 | ∅ | 2 | 7 |
| E | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ |
| F | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ |

En la matriz de adyacencias representamos los arcos que unen cada par de nodos.

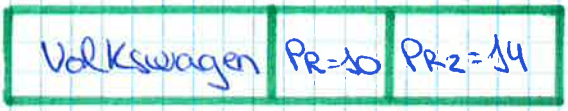
Representa tu grafo con una lista de adyacencia:




La lista de adyacencias indica los nodos que son adyacentes con cada nodo, en este caso, al ser un grafo dirigido, indica los nodos hasta los que llega el arco.

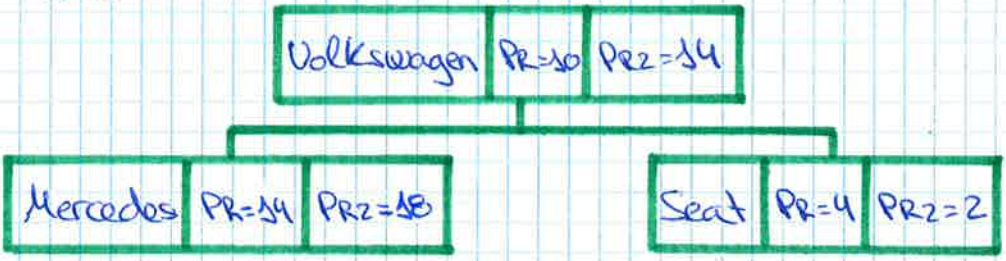
Para representarlo yo utilizaría dos LinkedList, pero en este caso, tal y como lo he dibujado, visualmente tiene el aspecto de un array de 6 posiciones en el que dentro de cada posición del array tenemos un LinkedList.

1. Agregamos Volkswagen:

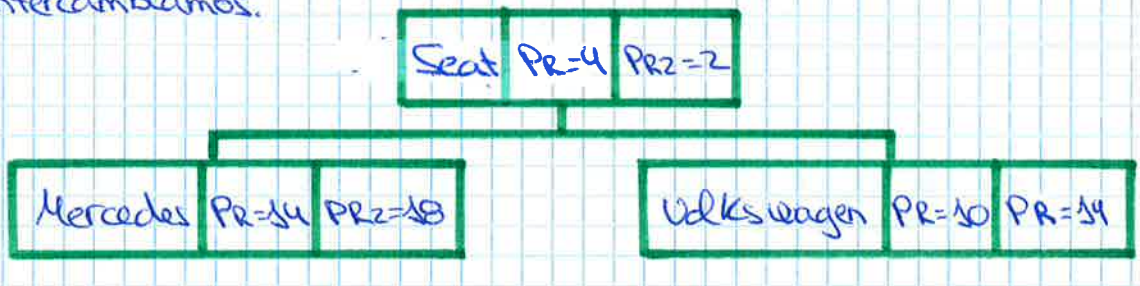



2. Agregamos Mercedes:

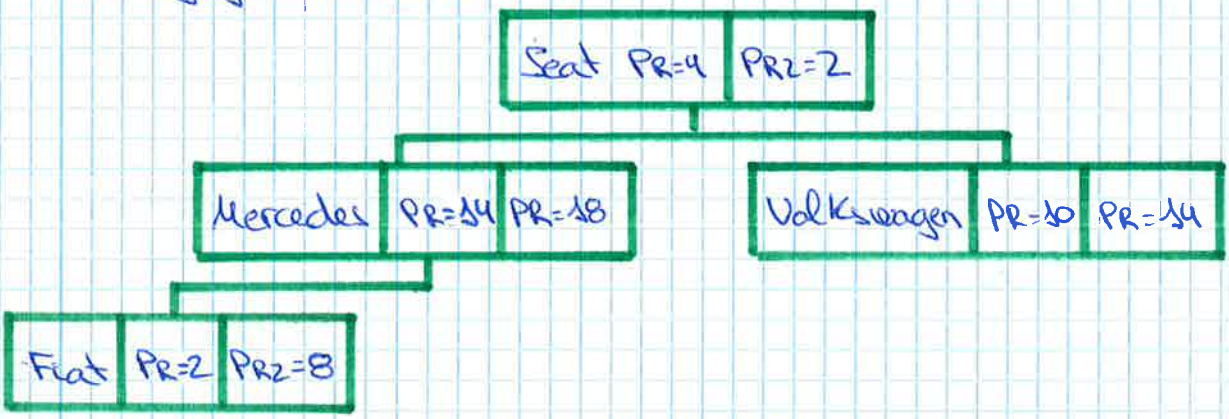

3. Agregamos Seat:



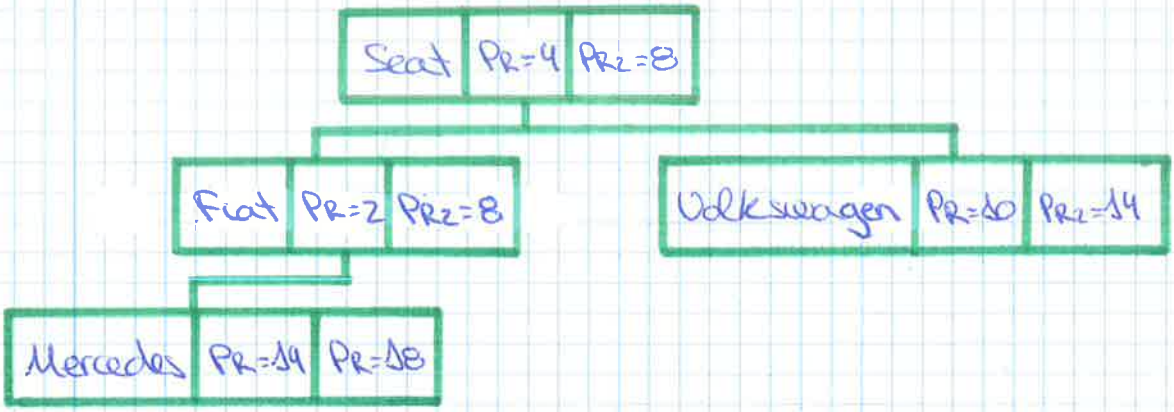

Observamos que ahora no se cumple la condición de orden, ya que seat tiene mayor prioridad que Volkswagen, así que los intercambiamos.



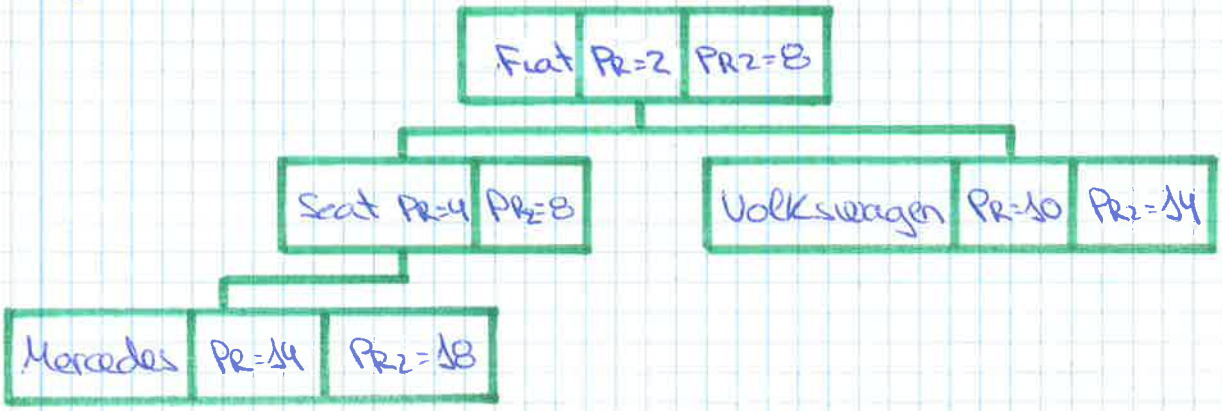

4. Agregamos Fiat:



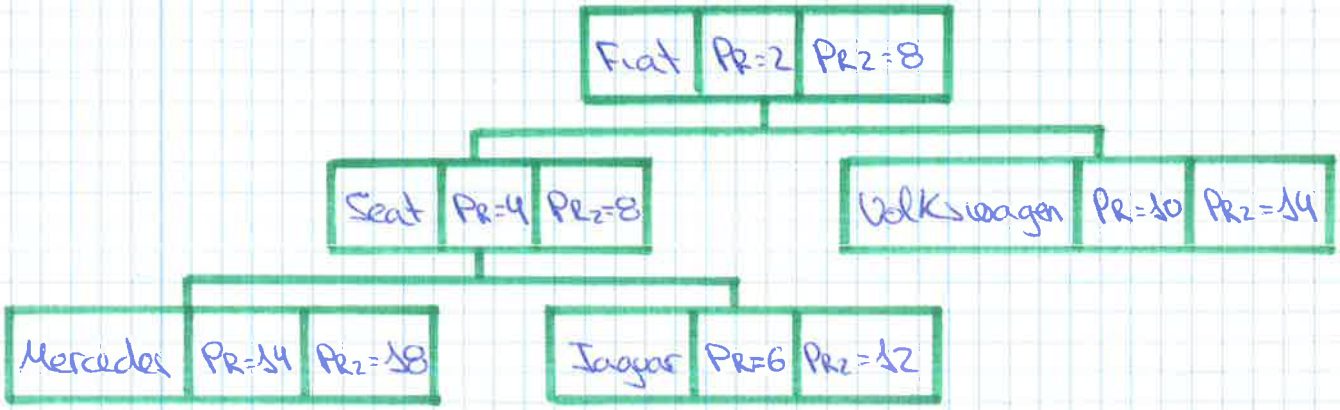

Ahora vuelve a ocurrir que no se cumple la condición de orden entre fiat y mercedes, los intercambiamos.



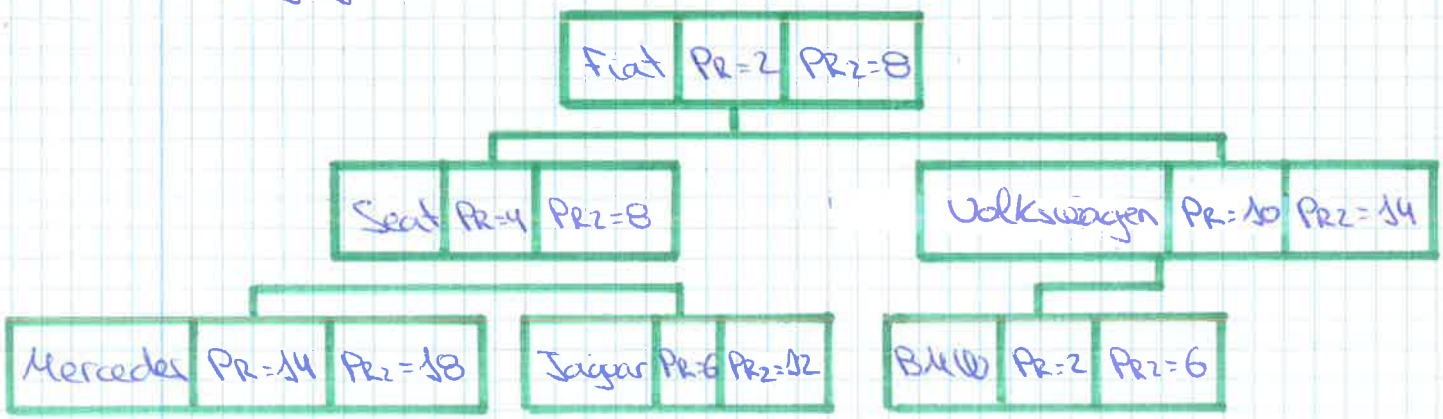

Se sigue incumpliendo la condición de orden, pues Fiat tiene mayor prioridad que Seat, volvemos a intercambiar.



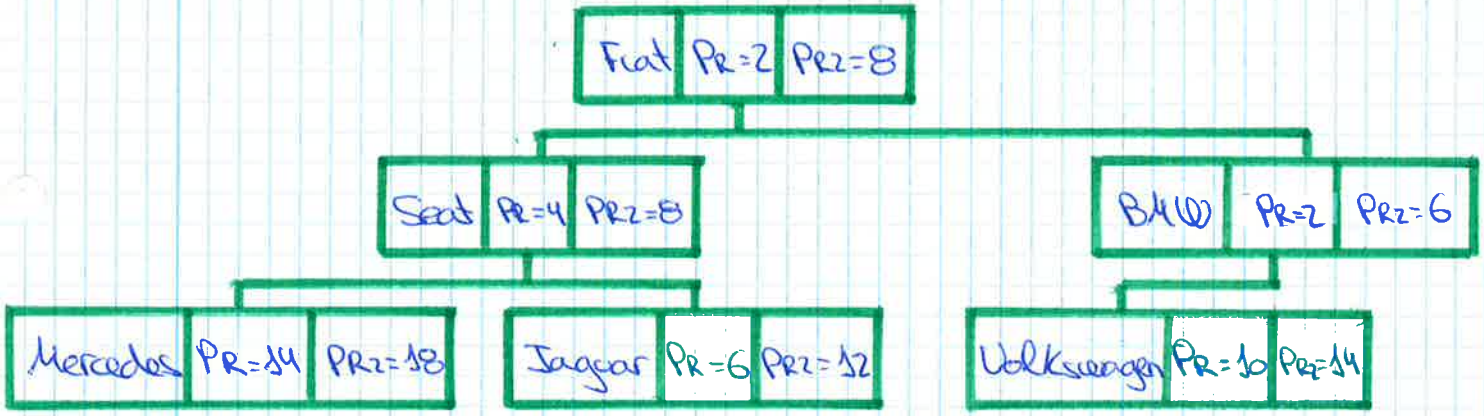

5. Agregamos Jaguar



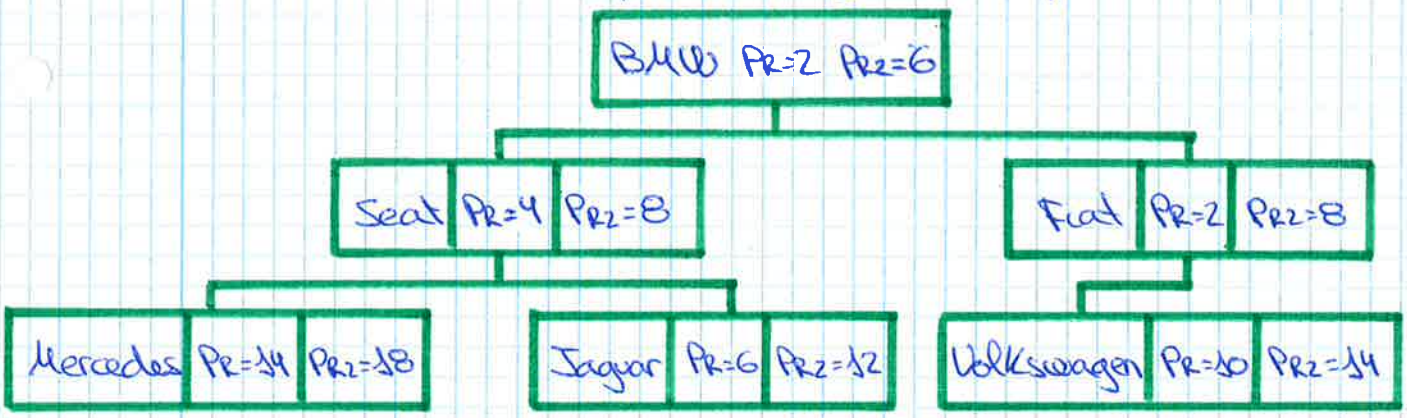

6. Agregamos BMW



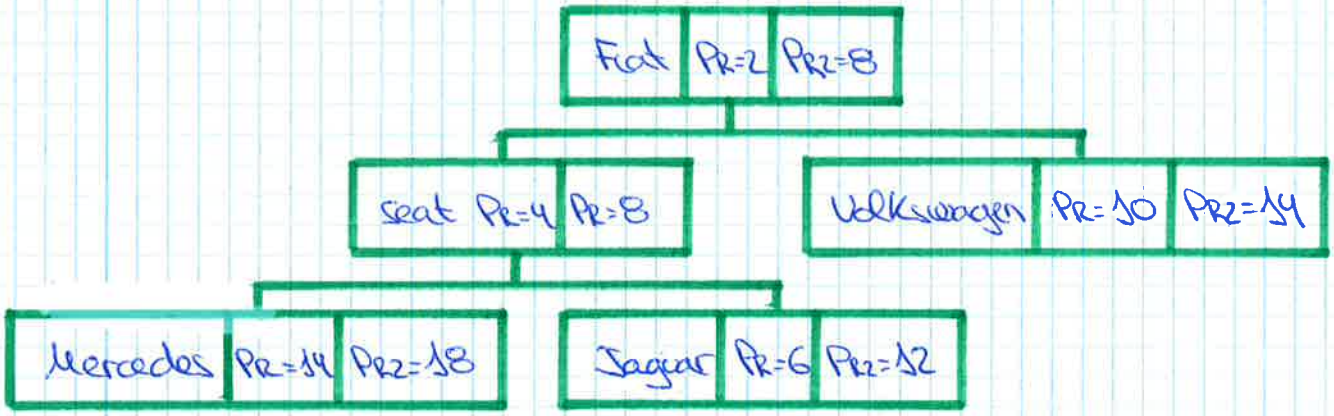

Ahora BMW no cumple la condición de orden y hay que intercambiarlo con Volkswagen



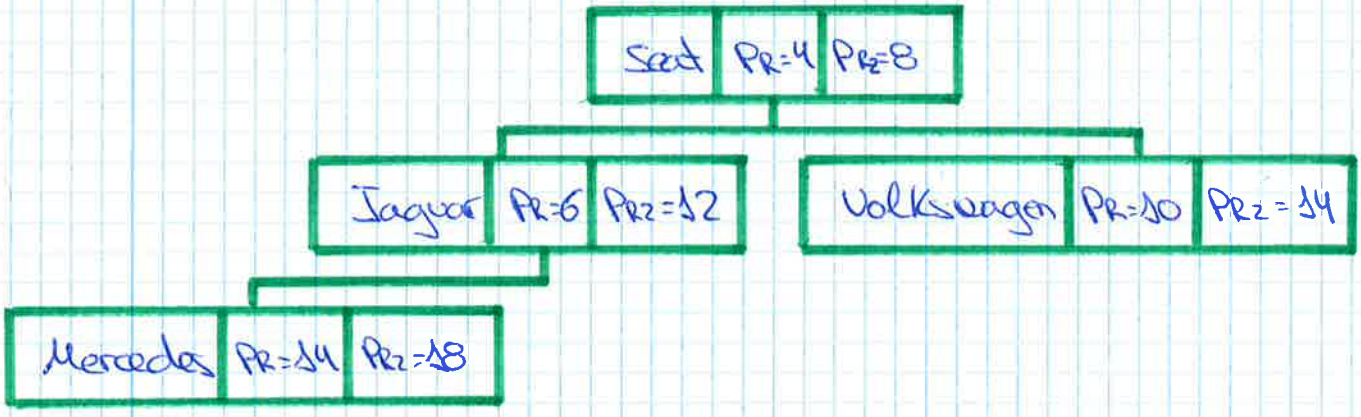

Ahora vemos un conflicto de prioridades, pues la prioridad principal de Fiat y de BMW es la misma, es aquí donde pasamos a comparar la prioridad secundaria; y observamos que no cumplimos con la condición de orden, porque la PR2 de BMW es más prioritaria que la de Fiat, así que los intercambiamos.



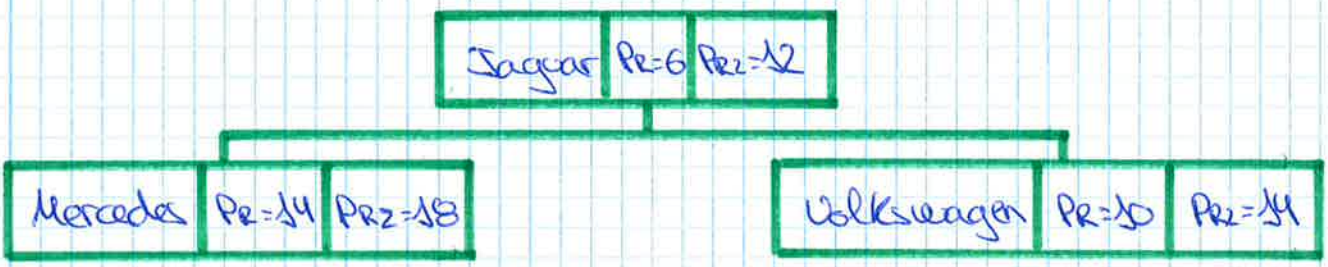
1. Sacamos BMW:



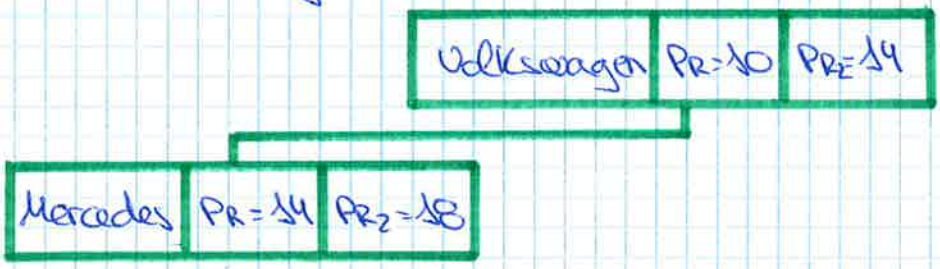

2. Sacamos Fiat:



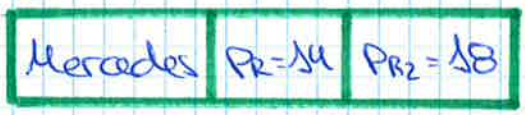

3. Sacamos Seat:




4. Sacamos Jagar:




5. Sacamos Volkswagen:




6. Por último, sacamos Mercedes.