Estructura de datos y algoritmos

estructuras de datos

→ forma de organizar un conjunto de datos elementales con el de facilitar su manipulación.

Lista enlazada

→ Nodo: contenedor de datos, espacio, información, puntero.

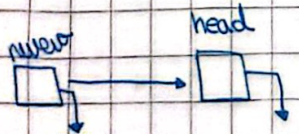
→ estructura de datos que posee una secuencia de nodos, donde se guardan campos arbitrarios que pueden "apuntar" al nodo sgte o al anterior.

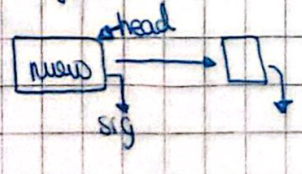
→ nodo simple.

* posee uno o mas atributos, atributo adicional - puntero sgte.
* metodos: accesadores - para rescatar datos (gets), mutadores: para modificar datos.

Codigo: (C++)

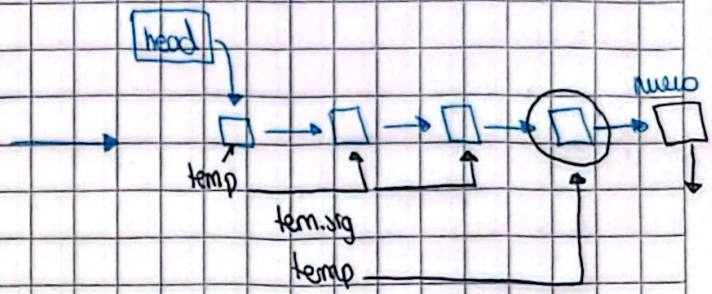
```
Class ListaSimple
{
    Nodo * head;
ListaSimple(){
    head = null;
  }
bool getVacio() ——→ (lista vacia)
{
 If(head == null)
  {
    return true;
}
  }
void pushInicio(Nodo * nuevo) ——→ (agrega a la lista de inicio).
{
 If(getVacio() == true)
 {
    head = nuevo; ——→
 }
 nuevo.sig = head;
   head = nuevo;
}
}
```





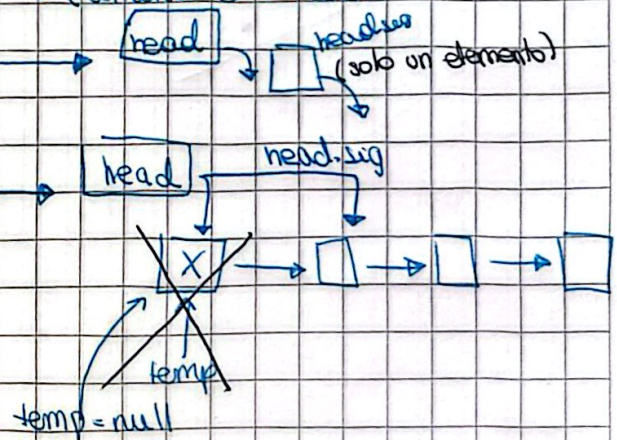
```
void pushfinal (Nodo *nuevo)
{
  if(getVacio() == true) {
      head = nuevo;
  }
  else {
      Nodo * temp = head;
while ( temp.sig != NULL) {
    temp = temp.sig;
}
  temp.sig = nuevo;
}
  }
```

→ agregar al final.



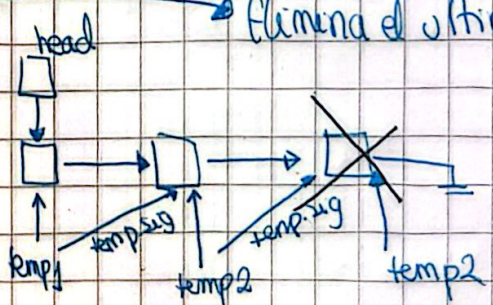

```
void deleteInicio() {
  if( getVacio() == false) {
  if(head.sig == NULL) {
      head = NULL;
  }
  else {
    Nodo * temp = head;
    head = head.sig;
    temp = null;
}
  }
  }
```

→ Elimina el inicio




```
void deleteUltimo() {
  if( getVacio() == false) {
  if(head.sig == null) {
      head = null;
  }
else {
  Nodo * temp1 = head;
  Nodo * temp2 = head.sig;
while(temp2.sig != null) {
    temp1 = temp1.sig;
    temp2 = temp2.sig;
}
  temp2 = null;
}}
```

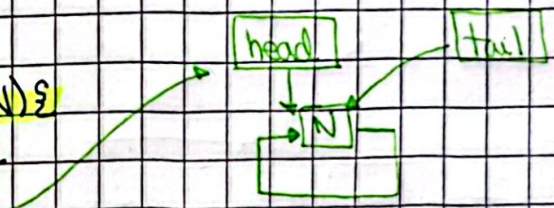
→ Elimina el ultimo

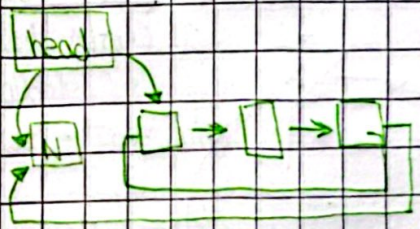


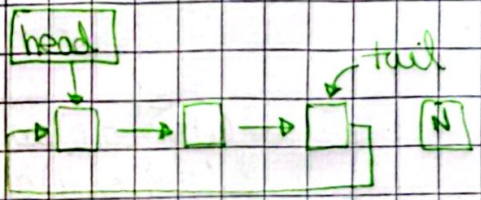
PROARTE

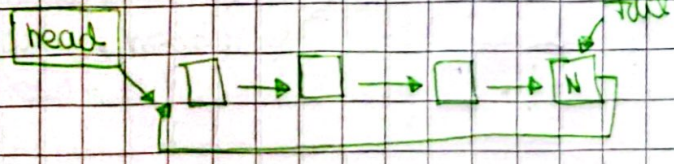
Lista enlazada circular

```
Class ListaCircular () {
   Nodo * head;
   Nodo * tail;
ListaCircular () {
  head = null;
  tail = null;
}
bool getVacio() {
  if ( head == null) {
   return true;
} }
void pushInicio (Nodo *N) {
  if ( getVacio() == true) {
     head = tail = N;
     temp.sig = head;
}
else
{
  N.sig = head;
  head = N;
 tail.sig = head;
}}
void pushFinal (Nodo *N) {
  if ( getVacio() == true) {
     head = tail = N;
     tail.sig = head;
}
else {
     tail.sig = N;
     tail = N;
  tail.sig = head;
}
}
```









PROARTE

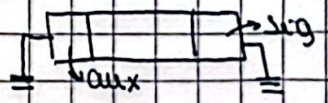
```
void deleteFinal(){
  if(getVacio() == false){
  if(head == tail){
     head = null;
     tail = null;
  }else{
    Nodo temp = tail;
    Nodo * temp2 = head;
while(temp2.sig != tail){
    temp2 = temp.sig;
}
    tail = temp2;
    tail.sig = head;
    temp = null;
}}}
```

LISTA DOBLEMENTE ENLAZADA SIMPLE

```
Class NodoD {
  int dato;
  Nodo * sig;
  Nodo * ant;
NodoD(int d){
  sig = null;
  ant = null;
  dato = d;
}
 int getDato;
 {
  return dato;
}};
```

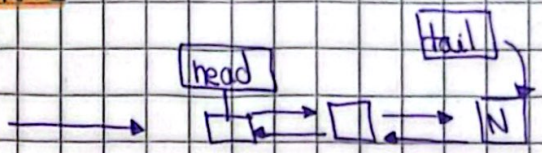
```
Class ListaDoble{
  Nodo * head;
  Nodo * tail;
ListaDoble(){
  head = tail = null;
}
 bool getVacio(){
   if(head = null){
     return true;
 }}
void pushIncio(Nodo * N){
   if(getVacio() == true){
      head = tail = N;
   }else{
     head.ant = N;
     N.sig = head;
     head = N;
   }
   }.
```

Nodo dDoble

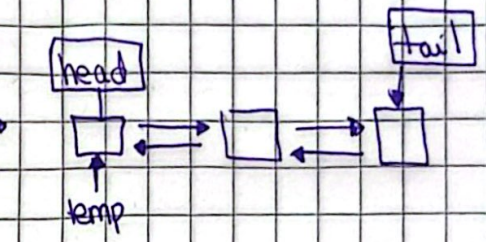


PROARTE

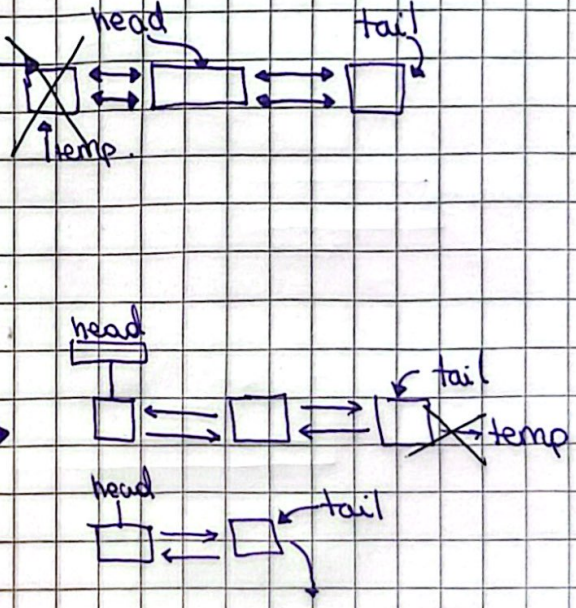
```
void pushFinal (NodoD * N) {
 if (getVacio() == true) {
     head = tail = N;
  } else {
    tail.sig = N;
    N.ant = tail;
    tail = N;
}}.
```




```
void deleteInicio() {
 if (getVacio == false) {
 if ( head = tail) {
   head = tail = null;
 } else
 {
   Nodo D * temp = head;
    head = head.sig;
     temp = null;
 }}}.
```




```
void deleteFinal () {
 if ( getVacio() == false) {
     if (head = tail) {
      head = tail = null
  }
   else {
    Nodo * temp = tail;
    tail = tail.ant;
    temp = null;
}
  }
    }
```

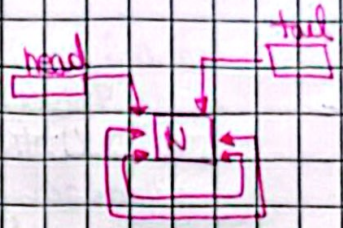


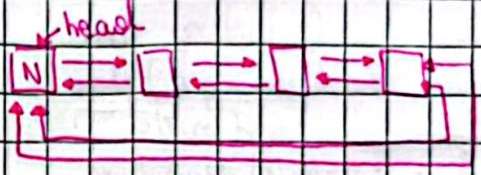
LISTA DOBLEMENTE CIRCULAR

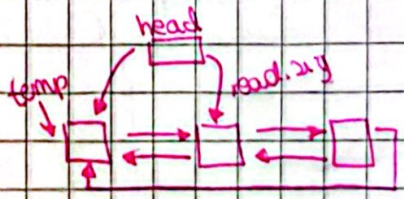
```
Class ListaCircularDoble {
    NodoD * head;
    NodoD * tail;
ListaCircularDoble () {
 head = tail = null;
}
 bool getVacio () {
   if ( head == null) {
   return true;
}}
```

PROARTE

```
void pushInicio(NodoD * N){
  if(getVacio() == true){
     head = tail = N;
     head.ant = tail;
     tail.sig = head;
  } else {
     N.sig = head;
     head.ant = N;
     head = N;
     head.ant = tail;
     tail.sig = head;
  }
}
void pushFinal (NodoD * N){
     (*)
  } else {
     N.ant = tail;
     tail.sig = N;
     tail = N;
     head.ant.tail;
     tail.sig = head;
  }}
void DeleteUnico(){
  if( getVacio() == false){
     if( head == tail){
        head = tail = null;}
  } else
  {
     NodoD * temp = head;
     head = head.sig;
     tail.sig = head;
     head.ant = tail;
     temp = null;
  }}
void DeleteFinal(){
     (*)
} else {
  NodoD * temp = tail;
   tail = tail.ant;
tail.sig = head;
head.ant = tail;
```







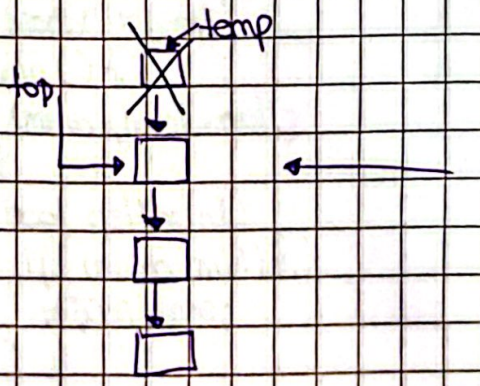

```
temp = null;
}
}
};
```

PROARTE

PILA (tipo LIFO, last in, first out)

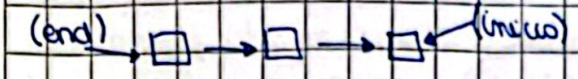
```
Class NodoP {
  int dato;
  NodoP * pabajo;
NodoP (int d) {
  dato = d;
  pabajo = null;
}
 int getDato() {
   return dato;
 }
}
```

```
Class Pila {
 NodoP * top;
Pila () {
  top = null;
}
bool getVacio () {
 if ( top == null ) {
  return true;
void push (NodoP * N) {
 if ( getVacio () = true) {
    top = N;
}
else {
  N.pabajo = top;
    top = N;
}
}
void pop () {
 if ( getVacio = false) {
 if ( top.pabajo == null ) {
    top = null;
}
else {
  NodoP * temp = top;
   top = top.pabajo;
   temp = null;
  }
   }
   }
```



PROARTE

COLA (FIFO, First In, First Out)




```
Class NodoC {
   int dato;
   NodoC * pAtras;
   NodoC (int d) {
      dato = d;
      pAtras = null;
  int getDato () {
   return dato;
}
  };

Class Cola {
  NodoC * inicio;
  NodoC * final;
Cola () {
  inicio = final = null;
}
  bool getVacio()
   if(inicio = null) {
     return true;
  }
  }
void encolar (NodoC * N) {
    if (getVacio () == true) {
       inicio = final = N;
} else
  {
     final.patras = N;
       final = N;
  }
  }
void desencolar () {
  if (getVacio() == false) {
  if (inicio == fin) {
     inicio = fin = null;
} else
  {
```

```
NodoC * temp = inicio;
inicio = inicio.patras;
   temp = null;
} }
  }
    };
```

PROARTE

Metodos de ordenamiento:

SELECTION SORT: $(O(n^2))$

→ compara todos los elementos de un arreglo buscando el mínimo elemento entre una posicion y el final de la lista e intercambian dicho mínimo con el elemento de dicha posicion i-esima

8	5	7	1	9	3	(n-1) first
1	5	7	8	9	3	(n-2) second.
1	3	7	8	9	5	(n-3)
1	3	5	8	9	7	2
1	3	5	7	9	8	1
1	3	5	7	8	9	0.

INSERTION SORT

→ cuando hay k elementos ordenados de menor a mayor, se forma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniendose cuando se encuentra un elemento menor (todos los mayores se desplazan una posicion a la derecha)

menor ← → despues del primero

7	8	5	2	4	6	3
7	8	5	2	4	6	3
5	7	8	2	4	6	3
2	5	7	8	4	6	3
2	4	5	7	8	6	3
2	4	5	6	7	8	3
2	3	4	5	6	7	8

ej: 12, 11, 13, 5, 6 → arreglo.

i = 1 to i = 4

i = 1 → 11, 12, 13, 5, 6.

i = 2 → 11, 12, 13, 5, 6.

i = 3 → 5, 11, 12, 13, 6

i = 4 → 5, 6, 11, 12, 13 //

PROARTE

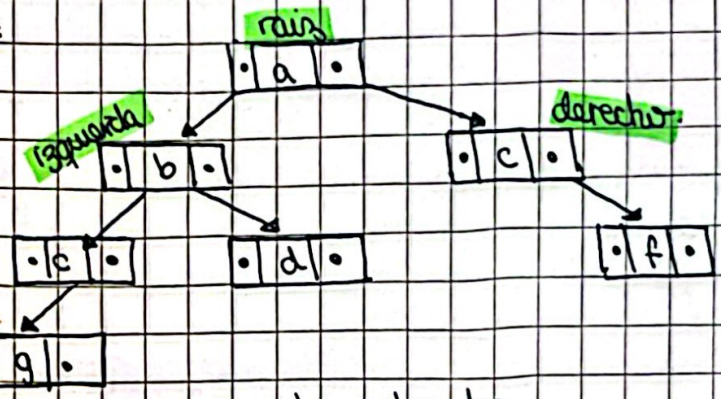
ARBOL BINARIO

→ tipo de arbol que se caracteriza porque cada nodo puede tener un maximo de dos hijos (izquierdo y derecho).

→ **caracteristicas:**

- compuesto por cero o mas nodo.
- cada nodo tiene valor, referencia al hijo izquierdo
- " " " " derecho.

Ej:

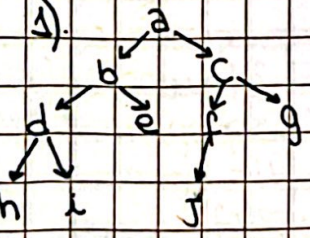


→ se usan para estar ordenados.

→ **tipos:**

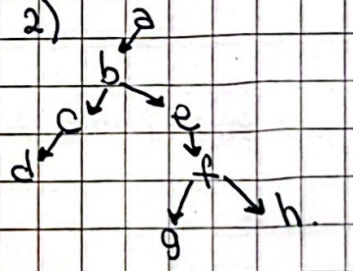
1. arbol balanceado = nodos equilibrados
2. arbol desbalanceado = nodos no equilibrados.

1).



* cada nivel excepto el ultimo debe estar balanceado.

2)



→ Los nodos son objetos con 2 datos base y N datos adicionales

dato base:

- **puntero izquierdo**: elemento tipo nodo comienza ~~contiene~~ en nulo y sirve para por hijos cuyo peso sea menor o igual que el del padre.
- **puntero derecho**: elemento tipo nodo, comienza en nulo y nos sirve para poner hijos cuyo peso sea mayor que el padre.
- **peso (dato)**: numero que permite insertar ordenadamente en el arbol.

PROARTE

→ codigo asociado.

```
Class Arbol {
Public:
   Hoja *raiz;
Arbol () {
  raiz = NULL;
}
  bool getVacio () {
  if(raiz == NULL) {
     return true;
  }
    }
      };
```

```
Class Hoja {
Public:
   Hoja * izq;
   Hoja * der;
   int peso;
  Hoja (int p) {
   peso = p;
   izq = NULL;
   der = NULL;
  }
    int getPeso () {
      return peso;
    }
      };
```

```
void insertarhoja (Hoja *H) {
  if(getVacio() == true) {
     raiz = H;
  }
  else {
    bool seguir = true;
    Hoja * temp = raiz;
while (seguir) {
if( temp.getpeso >= H.getPeso and temp.izq == NULL)
     temp.izq = H;
     seguir = false;
  }
else if (temp.getPeso() >= H.getPeso()) {
     temp = temp.izq;
  }
else if ( temp.getPeso() < H.getPeso() and temp.der = NULL) {
    temp.der = H;
   seguir = false;
  }
else if ( temp.getPeso() < H.getPeso()) {
   temp = temp.der;
  }
   }
   }
```

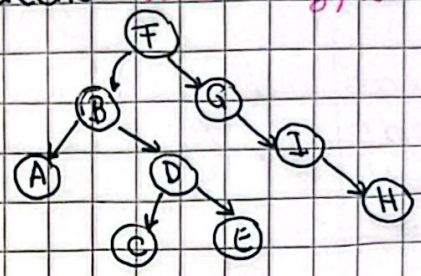
PROARTE

Algoritmo de recorrido:

→ procedimiento que nos permite recorrer una vez cada hoja de un arbol.

- **PreOrden**: tipo de orden que se realiza cierta accion sobre la raiz, luego trata el subarbol izquierdo cuando concluya el subarbol derecho. (nodo raiz, nodo izquierdo, nodo derecha)

Ej.:



Preorden = F-B-A-D-C-E-G-I-H

- **PostOrden**: nodo izquierdo, derecho, nodo raiz trabaja arbol subizquierdo, arbol subderecho y al final la hoja raiz.

PostOrden: A-C-E-D-B-G-I-H-F

(nodo izquierda, nodo derecha, nudo raiz)

- **InOrden**: orden izquierda, raiz, derecha. Se trabaja el subarbol izquierdo, despues nodo raiz y por ultimo el subarbol derecho.

InOrden: A-B-C-D-E-F-G-H-I

(nodo izquierda, nodo raiz, nodo derecha).

- **Arbol AVL**:

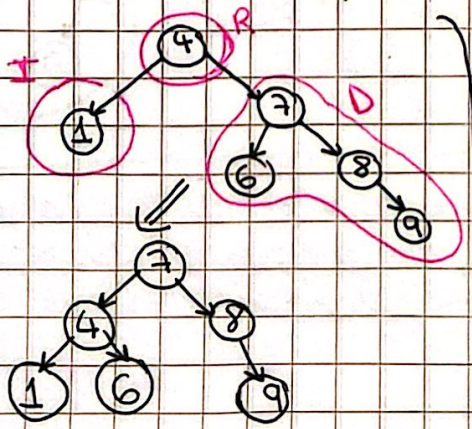
→ arbol binario "autobalanceado".

→ factor de balance es el alto del subarbol derecho, menos el alto del subarbol izquierdo.

FB = h(subarbol der) - h(subarbol izq).

FB = 1, 0 o 1

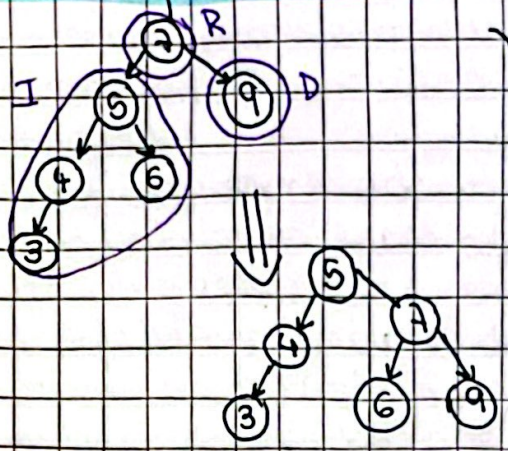
rotacion simple a izquierda:



- Lado izquierdo I, derecho D, raiz R.
- Arbol nuevo, raiz nueva corresponde a la raiz del hijo derecho.
- como hijo derecho, se coloca al hijo derecho de D.
- hijo izquierdo, arbol nuevo, la raiz del arbol original, el hijo izquierdo de D sera el derecho y el hijo izquierdo del arbol original sera el hijo izquierdo del nuevo subarbol

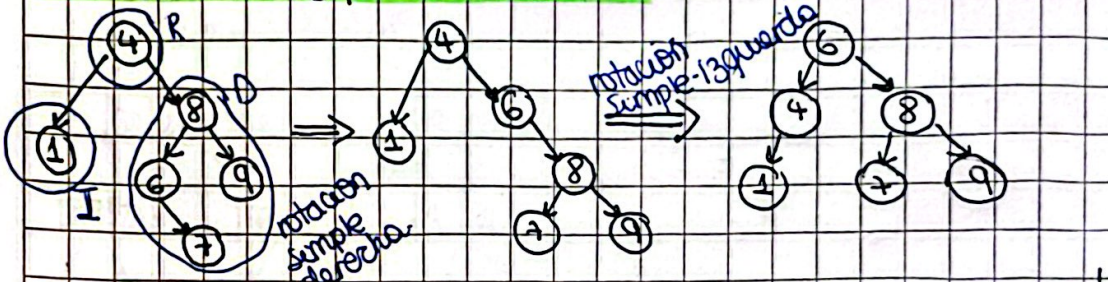
PROARTE

rotacion simple a la derecha:



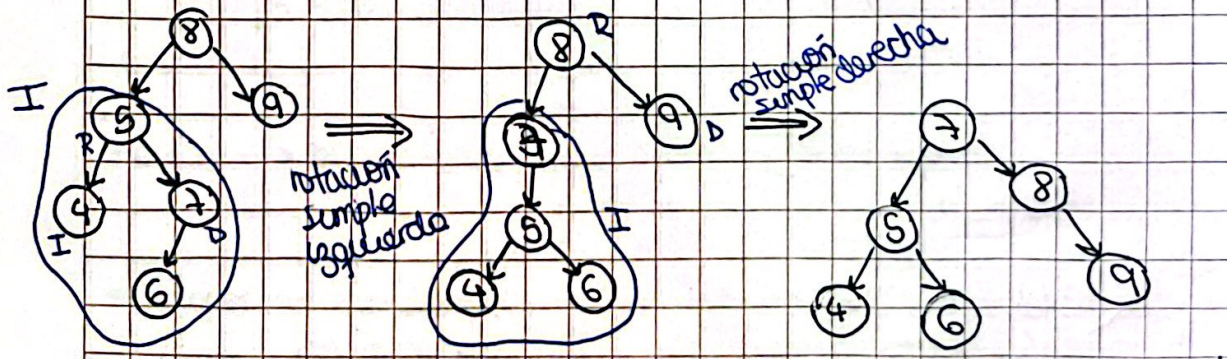
- raiz nueva, es la raiz del hijo izquierdo.
- hijo izquierdo, es el hijo izquierdo de I.
- como hijo derecho, arbol nuevo donde la raiz es la del arbol original, el hijo derecho de I es el izquierdo y el hijo derecho de arbol original sera el hijo derecho del nuevo subarbol

rotacion doble de izquierda a derecha:



- Rotamos simple a la derecha sobre el subarbol derecho y luego hacemos una rotacion simple a la izquierda sobre el arbol completo.

rotacion doble de derecha a izquierda:



- se hace una rotación simple a la izquierda sobre el subarbol izquierdo y luego se hace una rotacion simple a la derecha sobre el arbol completo.

PROARTE

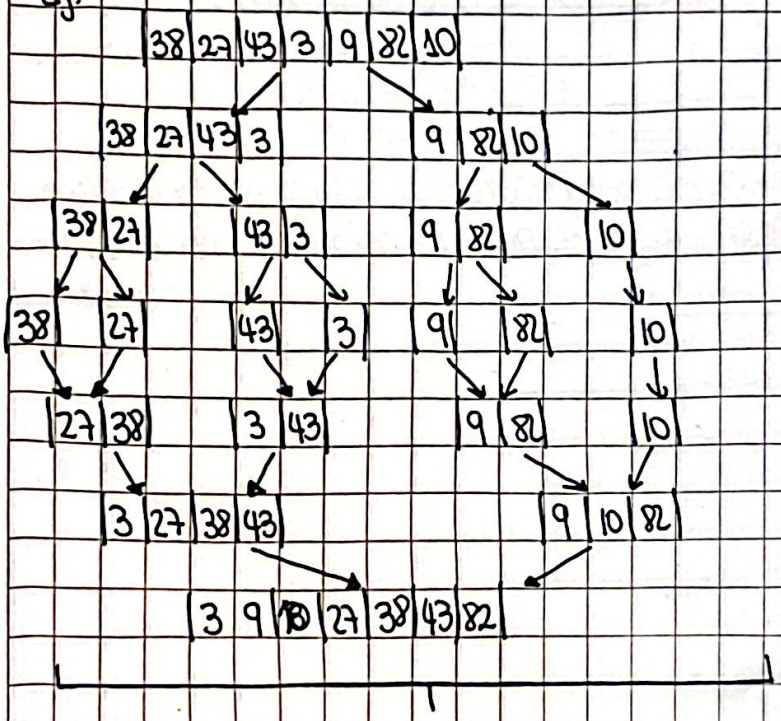
DIVIDE AND CONQUER

→ Metodo que consiste en solucionar un problema dividiendo el mismo, de manera recursiva en dos o mas subproblemas similares.

• **MergeSort**: algoritmo de ordenamiento del tipo out-place basado en la tecnica divide and conquer. Funciona si la longitud de la lista es 0 o 1 entonces esta ordenada, sino divide la lista en 2 sublistas de aprox la mitad del tamaño, ordena cada sublista aplicando el orden por mezcla y mezcla las dos sublistas en una sola ordenada.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & \text{if } n>1 \end{cases}$$

Ej:



• Divide → el arreglo se separa en 2 partes / tiempo constante $D(n) = \Theta(1)$.

• Conquer → resolver recursivamente en dos subproblemas de tamaño n/2 cada uno / $2T(n/2)$.

• merge → combinar las dos mitades ordenadas, en un tercer arreglo ordenado. Funcion de trabajo con n combinaciones en promedio / $C(n) = \Theta(n)$

DECRESE AND CONQUER

→ disminuye el tamaño del problema en cada paso.

→ enfoques:

- top down: comienza con el problema original, el que se va haciendo mas pequeño en cada paso.
- bottom up: se comienza por las soluciones muy pequeñas a parte del problema original.

→ formas de decrecer:

- decrecer por una constante: el algoritmo se hace decrecer el tamaño del problema en un valor constante.

$$a^n = a^{n-1} \cdot n$$

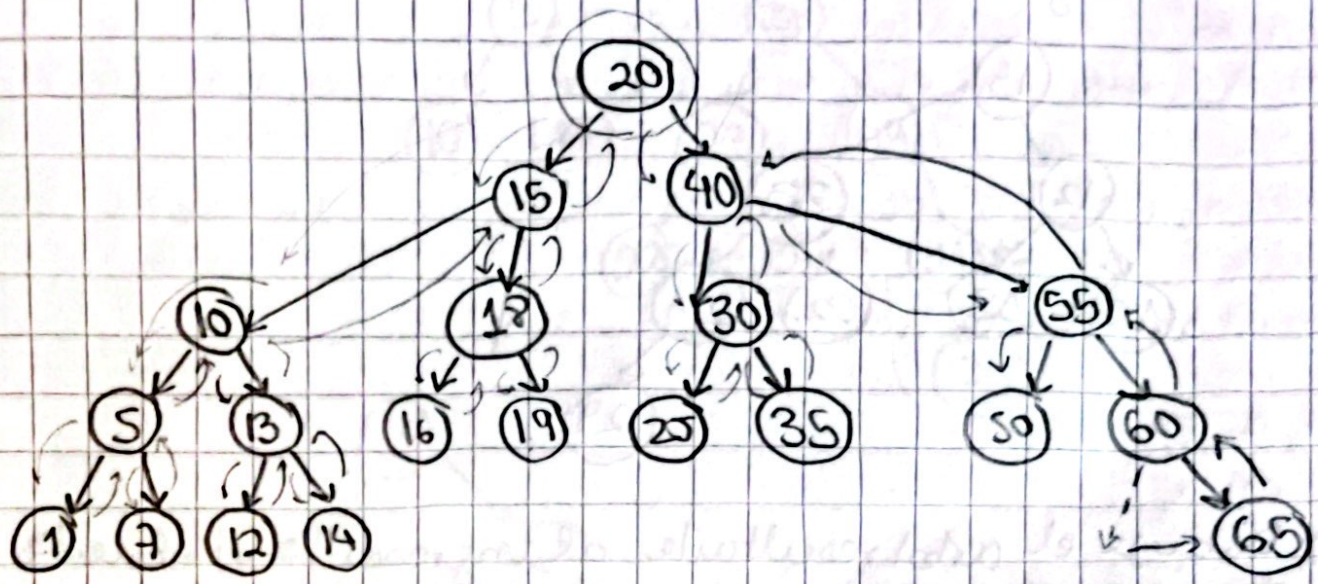
- decrecer por factor constante: el algoritmo se hace decrecer el tamaño del problema en un factor constante.

$$a^n = (a^{n/2})^2$$

- decrecer por factor variable: hace decrecer el tamaño del problema en un factor variable, es decir, no el mismo factor en cada paso.

Ejercicio Prueba 2

1- de acuerdo al siguiente arbol:



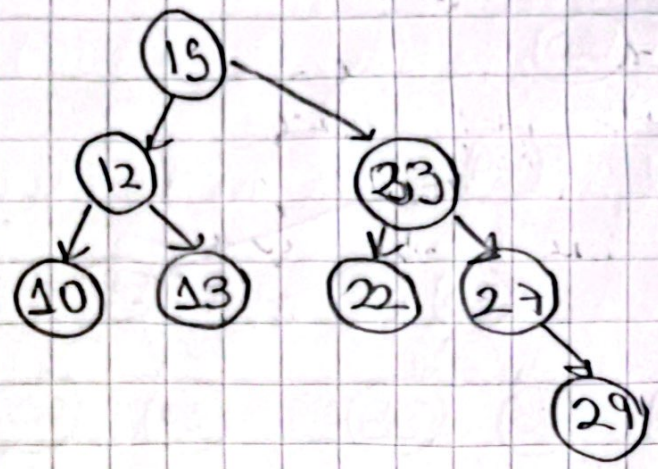
Indique el resultado que se obtiene al recorrer e imprimir el valor de cada nodo utilizando los algoritmos In orden, Preorden, Postorden.

✓ PreOrden: R-I-D
20-15-10-5-1-7-13-12-14-18-16-19-40-30-25-35-55-50-60-65

PostOrden: I-D-R.
1-7-5-12-14-13-10-16-19-18-15-25-35-30-50-65-60-55-40-20.

✓ In Orden: I-R-D
1-5-7-10-12-13-14-15-16-18-19-20-25-30-35-40-50-55-60-65

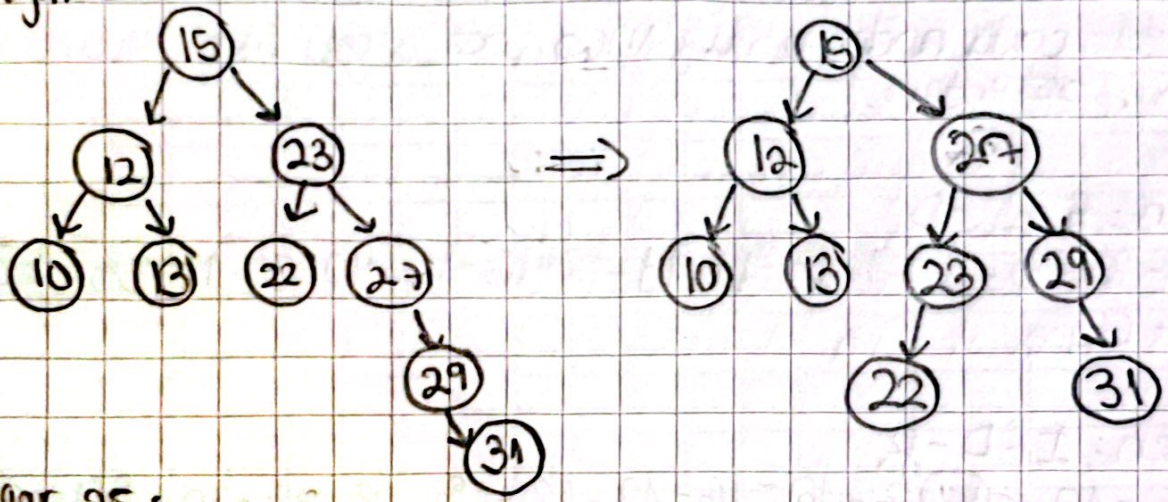
ejercicio 2
Dado el siguiente arbol AVL



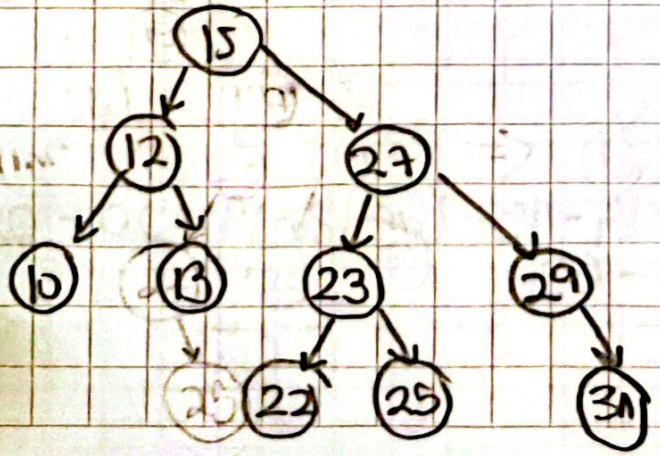
a) Dibuje el arbol resultante al ingresar los siguientes elementos
31, 25, 39 y 37

10 - 12 - 13 - 15 - 22 - 23

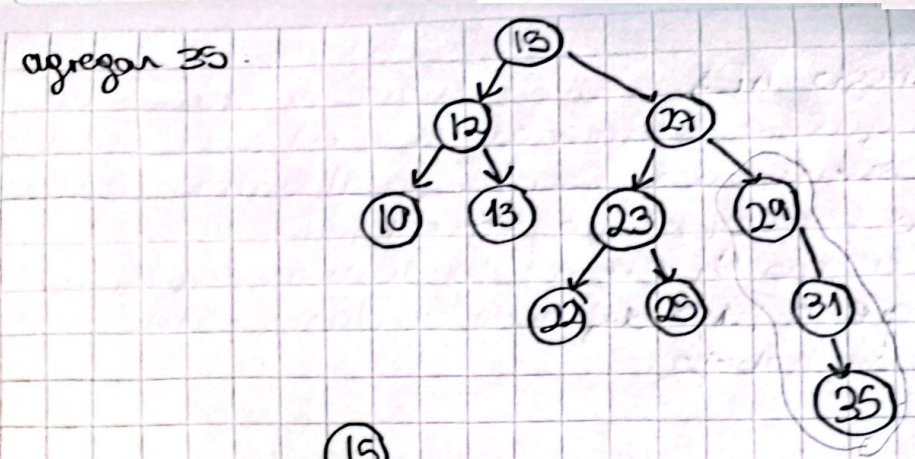
agregar 31:

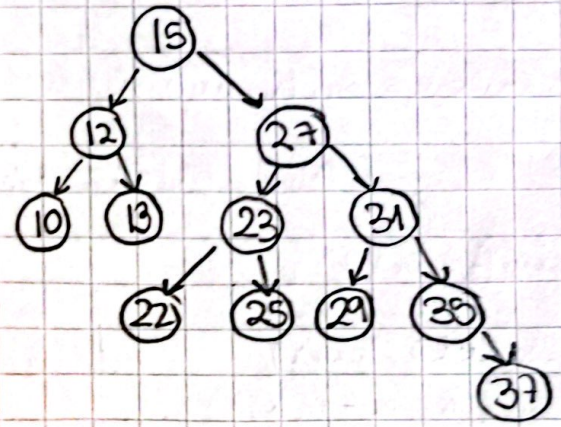


agregar 25:



agregar 35.





agregar 27.