RESUMEN FOD

ARCHIVOS

TIPOS

- Registros de longitud fija (File of <tipo_dato>)
- Texto(Text): Caracteres estructurados en líneas. Lectura/escritura con conversión automática de tipos. El acceso es exclusivamente secuencial. Útiles para importar y exportar datos.

DEFINICION

type archivo = file of tipo_de_datos;

var archivo_logico: archivo

Ejemplo

```
type
    persona = record
          dni: string[8];
       apellido: string[25];
          nombre: string[25];
          direccion: string[25];
        sexo: char;
    end;
      archivo_enteros  = file of integer;
      archivo_string   = file of string;
      archivo_personas = file of persona;
var
      enteros: archivo_enteros;
   texto: archivo_string;
   personas: archivo_personas;
```

OPERACIONES

```
assign(nombre_logico, nombre_fisico);
```

Realiza una correspondencia entre el archivo lógico y archivo físico.

Ejemplo:

```
assign(enteros, 'c:\archivos\enteros.dat');
assign(texto,' c:\archivos\texto.dat');
assign(personas, 'c:\archivos\personas.dat');
```

Apertura y creación de archivos

```
rewrite(nombre_logico);
```
→ Crea un archivo

```
reset(nombre_logico);
```
→ Abre un archivo existente

Ejemplo:

```
rewrite(enteros);
reset(personas);
```

Cierre de archivos

```
close(nombre_logico);
```

Transfiere la información del buffer al disco.

Ejemplo:

```
close(enteros);
close(personas);
```

Lectura y escritura de archivos

```
read(nombre_logico, var_dato);
write(nombre_logico, var_dato);
```

El tipo de dato de la variable `var_dato` es igual al tipo de datos de los elementos del archivo.

Ejemplo:

```
read(enteros, e);
```
→ e : integer;

```
write(personas, p);
```
→ p : persona;

EOF(nombre_logico);
Controla el fin de archivo.

fileSize(nombre_logico);
Devuelve el tamaño de un archivo.

filePos(nombre_logico);
Devuelve la posición actual del puntero en el archivo.
En longitud fija, los registros se numeran de 0..N-1.

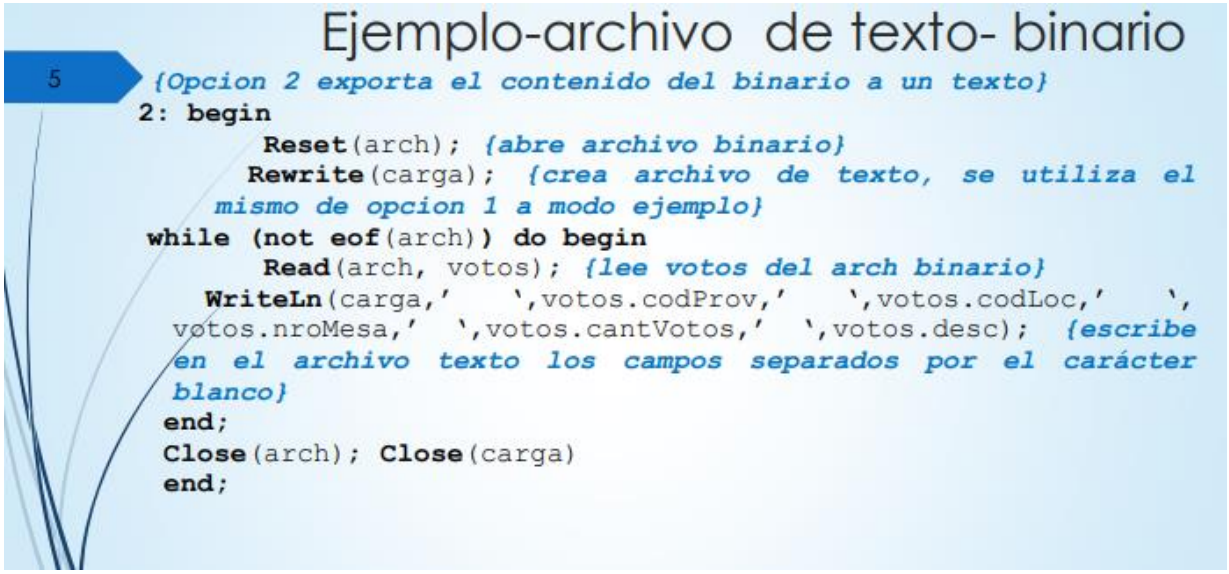
seek(nombre_logico, pos);
Establece la posición del puntero en el archivo.

Siempre que se lee/escribe se mueve el puntero.

ARCHIVO DE TEXTO

DE TEXTO A BINARIO

DE BINARIO A TEXTO



LOS STRING AL FINAL DE CADA LINEA.

CREAR ARCHIVO

```
procedure crear (var arc_log:archivo; nombre:string);
var
    e:emp;
begin
    rewrite (arc_log);
    leer (e);
    while (e.ap <> 'fin') do begin
        write (arc_log,e);
        leer (e);
    end;
    close (arc_log);
end;
```

AÑADIR

```
procedure aniadir (var arc_log:archivo);
var
    c:celu;
begin
    reset (arc_log);
    seek (arc_log, fileSize (arc_log));
    leer (c);
    while (c.cod <> -1) do begin
        write (arc_log,c);
        leer (c);
    end;
    close (arc_log);
end;
```

MAESTRO – DETALLE

Algorítmica clásica sobre archivos

Archivo maestro: Resume información sobre el dominio de un problema específico.

Ejemplo: *El archivo de productos de una empresa.*

Archivo detalle: Contiene movimientos realizados sobre la información almacenada en el maestro.

1. PRECONDICIONES
   - Existe un archivo maestro.
   - Existe un único archivo detalle que modifica al maestro.
   - Cada registro del detalle modifica a un solo registro del maestro que seguro existe.
   - No todos los registros del maestro son necesariamente modificados.
   - Cada elemento del maestro que se modifica, es alterado por un solo un elemento del archivo detalle.
   - Ambos archivos están ordenados por igual criterio.

Ejemplo: algoritmo

```
{Loop archivo detalle}
while not(EOF(det)) do begin
    read(mae, regm); // Lectura archivo maestro
    read(det, regd); // Lectura archivo detalle

    {Se busca en el maestro el producto del
    detalle}
    while (regm.cod <> regd.cod) do
      read(mae, regm);
```

```
    {Se modifica el stock del producto con la
    cantidad vendida de ese producto}
    regm.stock := regm.stock-regd.cant_vendida;
    {Se reubica el puntero en el maestro}
     seek(mae, filepos(mae)-1);
    {Se actualiza el maestro}
     write(mae, regm);
  end; // Fin while archivo detalle
  close(det);
  close(mae);
end.
```

2. PRECONDICIONES
    - Existe un archivo maestro.
    - Existe un único archivo detalle que modifica al maestro.
    - Cada registro del detalle modifica a un registro del maestro que seguro existe.
    - No todos los registros del maestro son necesariamente modificados.
    - Cada elemento del archivo maestro puede no ser modificado, o ser modificado por uno o más elementos del detalle.
    - Ambos archivos están ordenados por igual criterio.

```
procedure leer( var archivo: detalle; var dato: venta_prod);
begin
    if (not(EOF(archivo))) then
       read (archivo, dato)
    else
         dato.cod := valoralto;
end;
```

```
{programa principal}
begin
    assign(mae, 'maestro');
    assign(det, 'detalle');
    reset(mae);
    reset(det);
    read(mae, regm);
    leer(det, regd);
```

```
{Se procesan todos los registros del archivo detalle}
while (regd.cod <> valoralto) do begin
  aux := regd.cod;
  total := 0;

  {Se totaliza la cantidad vendida de productos iguales en el archivo de detalle}
  while (aux = regd.cod) do begin
      total := total + regd.cant_vendida;
      leer(det, regd);
   end;
```

```
{se busca el producto del detalle en el maestro}
    while (regm.cod <> aux) do
      read (mae, regm);
    {se modifica el stock del producto con la cantidad total vendida de ese producto}
    regm.cant := regm.cant - total;
    {se reubica el puntero en el maestro}
    seek(mae, filepos(mae)-1);
    {se actualiza el maestro}
    write(mae, regm);
      {se avanza en el maestro}
    if (not(EOF(mae))) then
        read(mae, regm);
  end;
  close(det);
  close(mae);
end.
```

3. PRECONDICIONES
   - Existe un archivo maestro.
   - Existe varios archivos detalle que modifica al maestro.
   - Cada registro del detalle modifica a un registro del maestro que seguro existe.
   - No todos los registros del maestro son necesariamente modificados.
   - Cada elemento del archivo maestro puede no ser modificado, o ser modificado por uno o más elementos del detalle.
   - Ambos archivos están ordenados por igual criterio.

```
procedure minimo (var registro:reg_detalle; var min:suc ; var deta: ar_detalle);
var
i,indiceMin: integer;
begin
    indiceMin:= 0;
    min.cod:= valorAlto;
    for i:= 1 to n do
        if (registro[i].cod <> valorAlto) then
            if (registro[i].cod < min.cod ) then begin
                min:= registro[i];
                indiceMin:= i;
            end;
    if (indiceMin <> 0) then begin
        leer(deta[indiceMin], registro[indiceMin]);
    end;
end;
```

```
procedure actualizar (var arc_maestro:maestro; var deta:ar_detalle; var registro:reg_detalle);
var
min: suc;
i:integer;
aString:string;
m:producto;
cantVendida:integer;
codActual:integer;
begin
    reset (arc_maestro);
    for i:=1 to n do begin
        Str (i,aString);
        Assign (deta[i],'detalle'+aString);
        reset (deta[i]);
        leer (deta[i],registro[i]); //leo el primer registro de los 30 archivos y lo escribo en el array de registros
        writeln (registro[i].cod);
    end;
    minimo (registro,min,deta); //obtengo minimo
    while (min.cod <> valorAlto) do begin  //mientras ese minimo sea distinto a valor alto
        codActual:= min.cod;
        cantVendida:= 0;
        while (min.cod = codActual) do begin //mientras el minimo sea el codigo actual
            cantVendida:= cantVendida + min.cantidadVendida;  //voy sumando las cantidades vendidas
            minimo (registro,min,deta); //busco otro minimo, si es el mismo el codigo va a seguir siendo el mismo al actual y se van a ir sumando
        end;
        read (arc_maestro,m);
        while(m.cod <> codActual)do begin  //busco el codigo de maestro para que coincida con el minimo
            read (arc_maestro,m);
        end;
```

```
        seek (arc_maestro,filePos (arc_maestro)-1); //ubico puntero
        m.stockD := m.stockD - cantVendida; //actualizo el stock
        write (arc_maestro,m); //actualizo maestro
    end;
    for i:=1 to n do
        close (deta[i]);
    close (arc_maestro);
end;
```

CORTE DE CONTROL

```
while (reg.provincia <> valor_alto)do begin
   writeln('Provincia:', reg.provincia);
   prov := reg.provincia;
   totProv := 0;
   while (prov = reg.provincia) do begin
      writeln('Ciudad:', reg.ciudad);
      ciudad := reg.ciudad;
      totCiudad := 0;
      while (prov = reg.provincia) and
            (ciudad = reg.ciudad) do begin
         writeln('Sucursal:',reg.sucursal);
         sucursal := reg.sucursal;
         totSuc := 0;
```

```
  while (prov = reg.provincia) and
        (ciudad = reg.ciudad) and
        (sucursal = reg.sucursal) do begin

     write("Vendedor:", reg.vendedor);
     writeln(reg.monto);
     totSuc := totSuc + reg.monto;
     leer(archivo, reg);
  end;
```

```
            writeln("Total Sucursal",totSuc);
            totCiudad := totCiudad + totSuc;
        end;{while (prov = reg.provincia) and
            (ciudad = reg.ciudad)}
        writeln("Total Ciudad", totCiudad);
        totProv := totProv + totCiudad;
    end;{while(prov = reg.provincia)}
    writeln("Total Provincia", totProv);
    total := total + totProv,
end;{while(reg.provincia <> valor_alto)}
writeln("Total Empresa", total);
close(archivo);
end.
```

BAJAS

- **Baja física**

Consiste en borrar efectivamente la información del archivo, recuperando el espacio físico.

- **Baja lógica**

Consiste en borrar la información del archivo, pero sin recuperar el espacio físico respectivo.

BAJA FISICA

```
begin {se sabe que existe Carlos Garcia}
  assign (archivo, 'arch_empleados');
  assign (archivo_nuevo, 'arch_nuevo');
  reset (archivo);
  rewrite (archivo_nuevo);
  leer (archivo, reg);
  {se copian los registros previos a Carlos Garcia}
  while (reg.nombre <> "Carlos Garcia") do begin
    write (archivo_nuevo, reg);
    leer (archivo, reg);
  end;
```

```
    {se descarta a Carlos Garcia}
    leer(archivo, reg);

    {se copian los registros restantes}
    while (reg.nombre <> valoralto) do begin
      write(archivo_nuevo, reg);
      leer(archivo, reg);
    end;

    close(archivo_nuevo);
    close(archivo);
end.
```

MISMO ARCHIVO (NO TIENE QUE TENER ORDEN)

```
procedure eliminar (var arc_log: archivo; nro:integer);
var
    e:emp;
    ok:boolean;
    f:emp;
begin
    ok:= false;
    seek (arc_log,FileSize(arc_log)-1);
    read (arc_log,f);
    seek (arc_log,0);
    while not eof(arc_log) and not (ok) do begin
        read (arc_log,e);
        if (e.nro = nro) then begin
            ok:= true;
            seek (arc_log,filepos (arc_log)-1);
            write (arc_log,f);
            seek (arc_log,FileSize(arc_log)-1);
            truncate(arc_log);
        end;
    end;
    if (ok = false) then
        writeln ('NO SE ENCONTRO NUMERO DE EMPLEADO')
    else
        writeln ('SE ELIMINO CON EXITO');
    writeln ('');
    close (arc_log);
end;
```

TRUNCO EL ULTIMO

BAJA LOGICA

```
Begin {se sabe que existe Carlos Garcia}
  assign(archivo, 'arch_empleados');
  reset(archivo);
  leer(archivo, reg);
  {Se avanza hasta Carlos Garcia}
  while (reg.nombre <> "Carlos Garcia") do
    leer(archivo, reg);
  {Se genera una marca de borrado}
  reg.nombre := "***";
  {Se borra lógicamente a Carlos Garcia}
  seek(archivo, filepos(archivo)-1 );
  write(archivo, reg);
  close(archivo);
end.
```

```
procedure eliminar (var arc_log:archivo);
var
a:asistente;
begin
    reset(arc_log);
    leerArc (arc_log,a);
    while (a.nro <> valorAlto) do begin
        if (a.nro < 1000) then begin
            a.AyN:= '@Eliminado';
            seek (arc_log,filePos(arc_log)-1);
            write(arc_log,a);
        end;
        leerArc (arc_log,a);
    end;
end;
```

REASIGNACION ESPACIO

Alta en los lugares borrados

LISTA INVERTIDA

```
procedure eliminar (var arc_log:archivo);
var
n,indice:novela;
codigo:integer;
ok:boolean;
begin
    ok:= false;
    write ('INGRESE CODIGO DE LA NOVELA QUE DESEA ELIMINAR: '); readln (codigo);
    writeln ('');
    leerArc(arc_log,indice); //leo el indice que esta en el cabecera
    leerArc (arc_log,n);
    while (n.cod <> valorAlto) and not(ok) do begin
        if (n.cod = codigo) then begin
            ok:= true;
            n.cod:= indice.cod; //copio el indice que estaba en el reg 0 en el que elimino para tener la lista invertida
            seek (arc_log,filePos(arc_log)-1);
            indice.cod:= filePos(arc_log) * -1; //paso el indice a negativo
            write(arc_log,n);
            seek(arc_log,0);
            write(arc_log,indice); //el indice que esta en el registro cabecera lo remplazo con el del reg que acabo de eliminar
        end
        else
            leerArc (arc_log,n);
    end;
    if (ok) then
        writeln ('NOVELA ELIMINADA')
    else
        writeln ('NO SE ENCONTRO NOVELA');
end;
```

COMPACTACION ARCHIVO

Quita los registros marcados como eliminados, utilizando cualquiera de los algoritmos vistos para baja física.

```
procedure eliminar (var arc_log:archivo; codigo:integer);
var
n:prendas;
begin
    reset (arc_log);
    leerArc(arc_log,n);
    while (n.cod <> codigo) do //se supone que todas las prendas existen en el maestro.
        leerArc (arc_log,n);
    seek (arc_log,filePos(arc_log)-1);
    n.stock:= n.stock * -1;
    write (arc_log,n);
    close (arc_log);
end;
```

```
procedure actualizar (var arc_log,aEliminar:archivo);
var
n:prendas;
begin
    reset (aEliminar);
    leerArc (aEliminar,n);
    while (n.cod <> valorAlto) do begin
        eliminar (arc_log,n.cod);
        leerArc(aEliminar,n);
    end;
    close (aEliminar);
end;
```

```
procedure compactar (var arc_log,aux:archivo);
var
n:prendas;
begin
    reset(arc_log);
    rewrite(aux);
    leerArc(arc_log,n);
    while(n.cod <> valorAlto) do begin
        if (n.stock >= 0) then begin
            write (aux,n);
        end;
        leerArc(arc_log,n);
    end;
    close (arc_log);
    close(aux);
end;
```

```
    actualizar (arc_log,aEliminar);
    compactar (arc_log,arch_nuevo);
    erase (arc_log);
    rename(arch_nuevo,'maestro.dot');
    writeln('NUEVO MAESTRO');
    writeln ('');
    mostrar(arch_nuevo);
end.
```

ARBOLES

POLITICAS

1. **Política izquierda:** se intenta redistribuir con el hermano adyacente izquierdo, si no es posible, se fusiona con hermano adyacente izquierdo.
2. **Política derecha:** se intenta redistribuir con el hermano adyacente derecho, si no es posible, se fusiona con hermano adyacente derecho.
3. **Política izquierda o derecha:** se intenta redistribuir con el hermano adyacente izquierdo, si no es posible, se intenta con el hermano adyacente derecho, si tampoco es posible, se fusiona con hermano adyacente izquierdo.
4. **Política derecha o izquierda:** se intenta redistribuir con el hermano adyacente derecho, si no es posible, se intenta con el hermano adyacente izquierdo, si tampoco es posible, se fusiona con hermano adyacente derecho.

Casos especiales: en cualquier política si se tratase de un nodo hoja de un extremo del árbol debe intentarse redistribuir con el hermano adyacente que el mismo posea.

Aclaración:
- En caso de underflow lo primero que se intenta **SIEMPRE** es redistribuir y el

hermano adyacente se encuentra en condiciones de hacer la redistribución y no se produce underflow en el.

- Si se hace una fusión siempre me quedo con el nodo izquierdo.
- Siempre le dejo mas elementos al izquierdo si se trata de orden impar
- Las lecturas se hacen 1 sola vez.

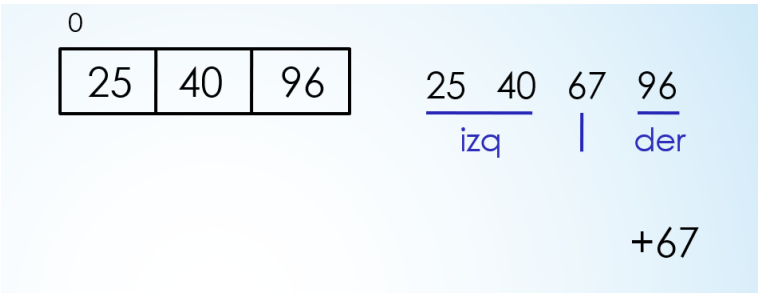
ARBOL B

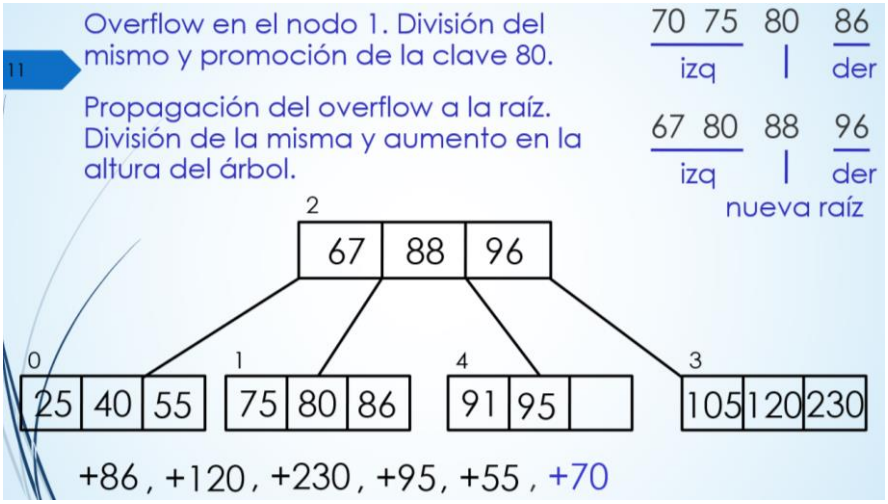
- Cada nodo del árbol puede contener como máximo M descendientes y M-1 elementos.
- La raíz no posee descendientes directos o tiene al menos dos.
- Un nodo con X descendientes directos contiene X-1 elementos.
- Todos los nodos (salvo la raíz) tienen como mínimo [M/2] – 1 elementos y como máximo M-1 elementos.
- Todos los nodos terminales se encuentran al mismo nivel.

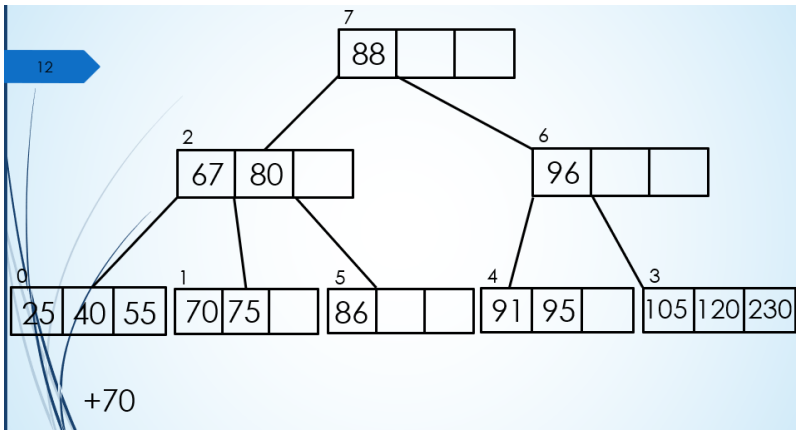
TRATAMIENTO OVERFLOW

ALTA

- Se crea un nuevo nodo.
- La primera mitad de las claves se mantiene en el nodo con overflow.
- La segunda mitad de las claves se traslada al nuevo nodo.
- La menor de las claves de la segunda mitad se promociona al nodo padre.
- Propagación del overflow a la raíz. División de la misma y aumento en la altura del árbol.

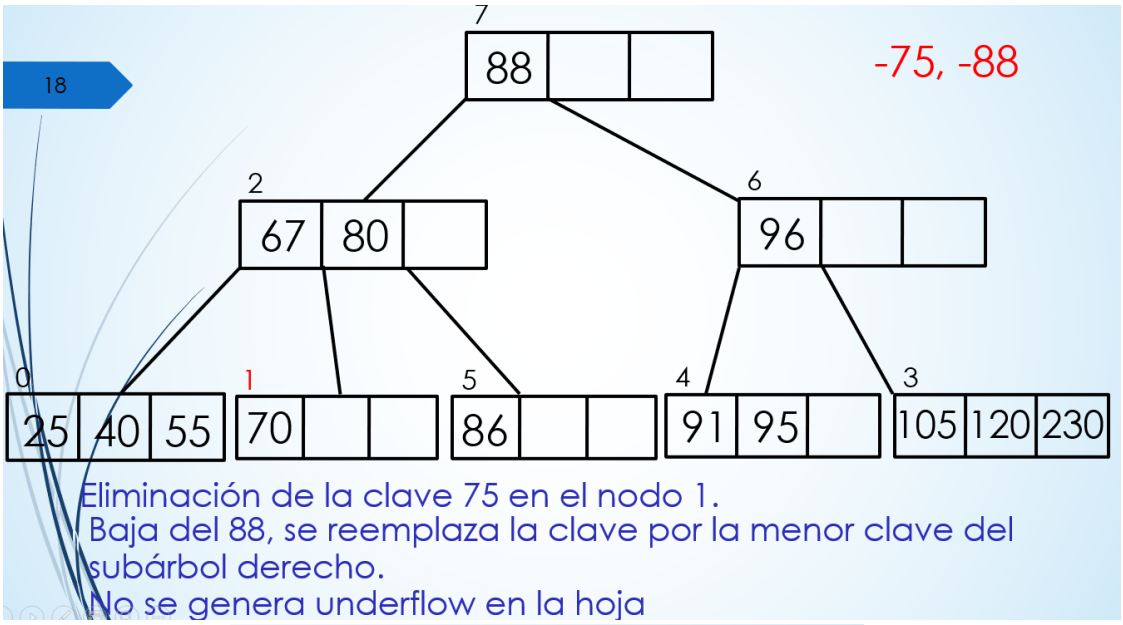




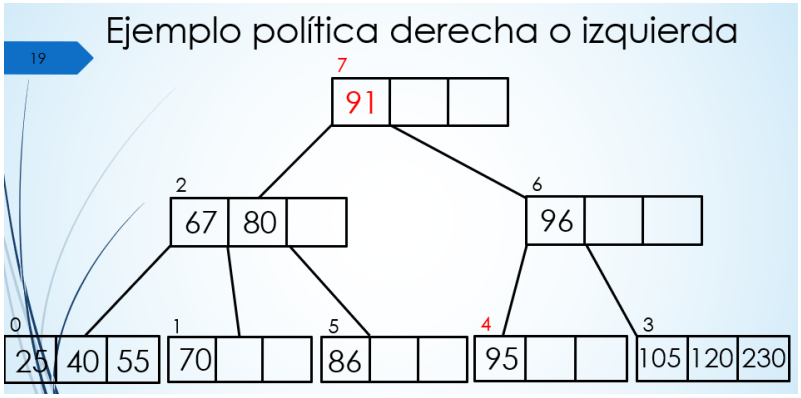


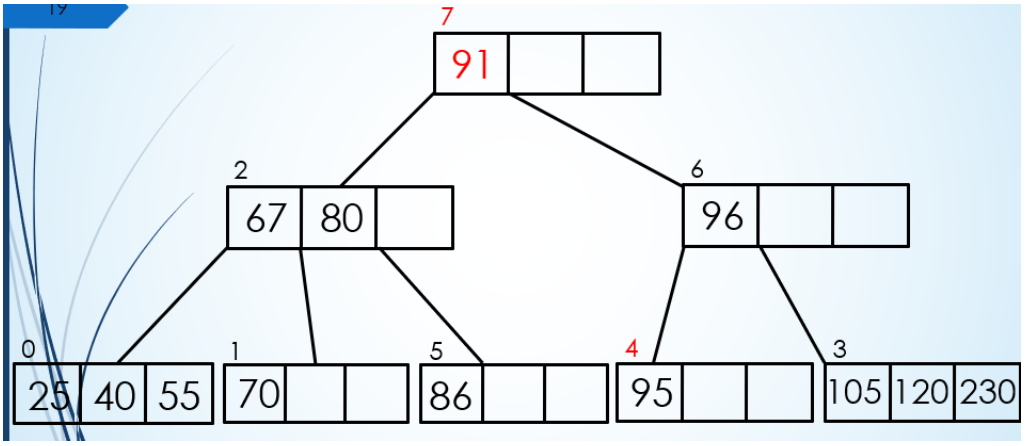
BAJA

1. Si la clave a eliminar no está en una hoja, se debe reemplazar con la menor clave del subárbol derecho.
2. Si el nodo hoja contiene por lo menos el mínimo número de claves, luego de la eliminación, no se requiere ninguna acción adicional.
3. En caso contrario, se debe tratar el underflow
4. Primero *se intenta* **redistribuir** con un hermano adyacente. La redistribución es un proceso mediante el cual se trata de dejar cada nodo lo más equitativamente cargado posible.
5. Si la redistribución no es posible, entonces se debe fusionar con el hermano adyacente.



-88 L/E: L7, L6,L4, E4,E7


Ejemplo política derecha o izquierda
19
7
91
2
67
80
6
96
0
25
40
55
1
70
5
86
4
95
3
105
120
230


7
91
2
67
80
6
96
0
25
40
55
1
70
5
86
4
95
3
105
120
230

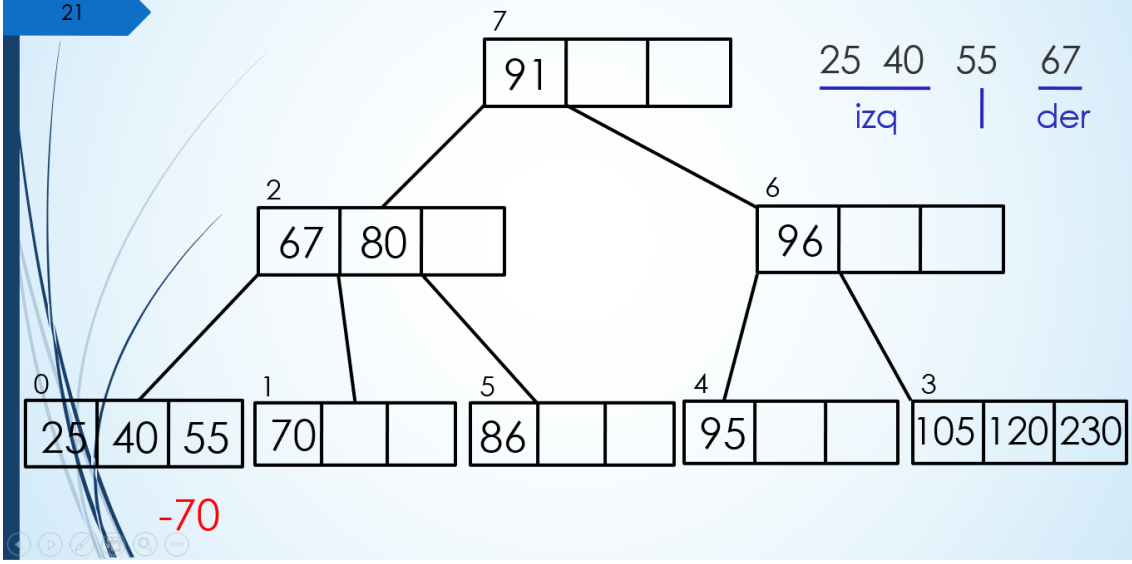
La eliminación de la clave 70 en el nodo 1 produce underflow.

Se intenta redistribuir con el hermano derecho. No es posible ya que el nodo contiene la cantidad mínima de claves.

Se intenta redistribuir con el hermano izquierdo. La operación es posible y se rebalancea la carga entre los nodos 1 y 0.

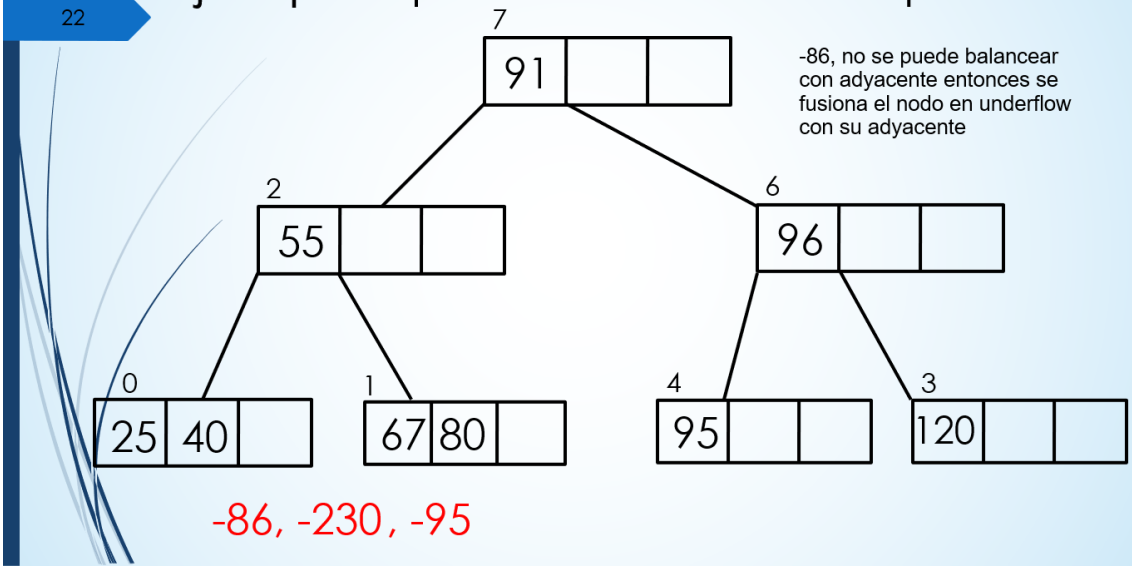
Ejemplo - política derecha o izquierda

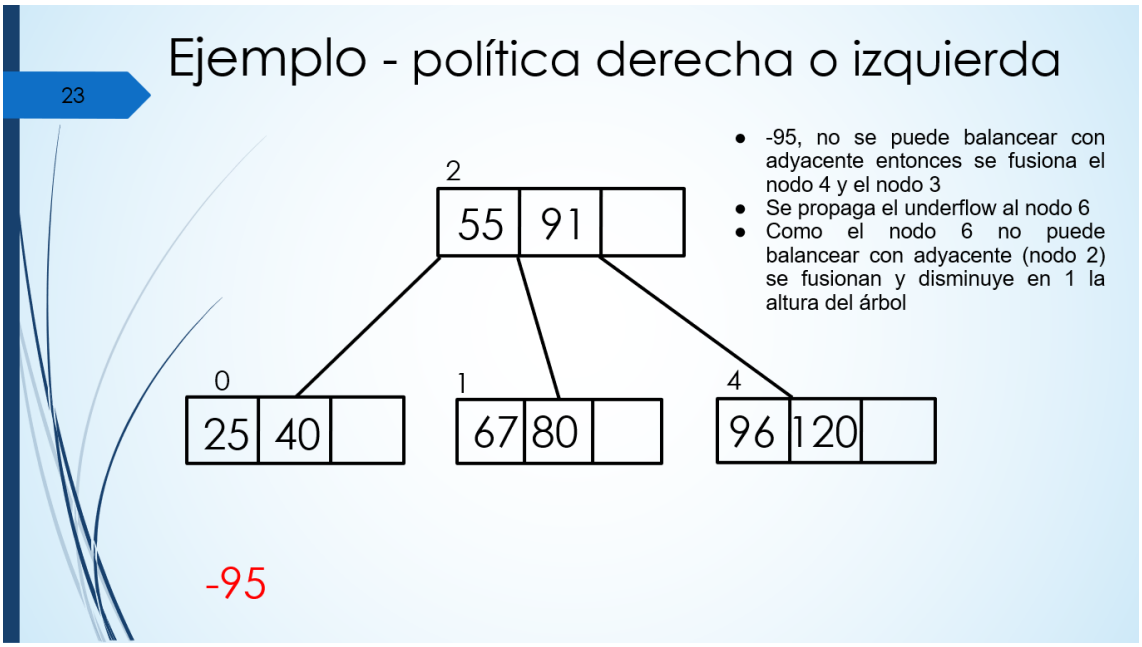
21



Ejemplo - política derecha o izquierda

22





ARBOL B+

ALTA

Dificultad: Inserción en nodo lleno (overflow).

El nodo afectado se divide en 2, distribuyéndose las claves **lo más equitativamente posible**. Una **copia** de la clave del medio o de la menor de las claves mayores (casos de overflow con cantidad pares de elementos) se promociona al nodo padre. El nodo con overflow se divide a la mitad.

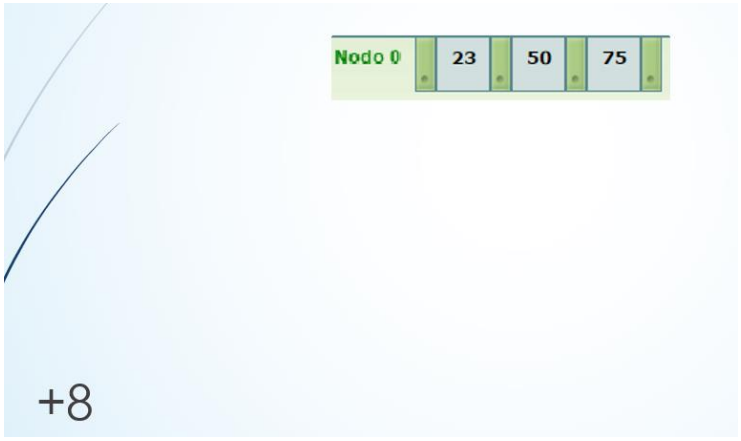
La copia de la clave sólo se realiza en un overflow ocurrido a nivel de hoja.

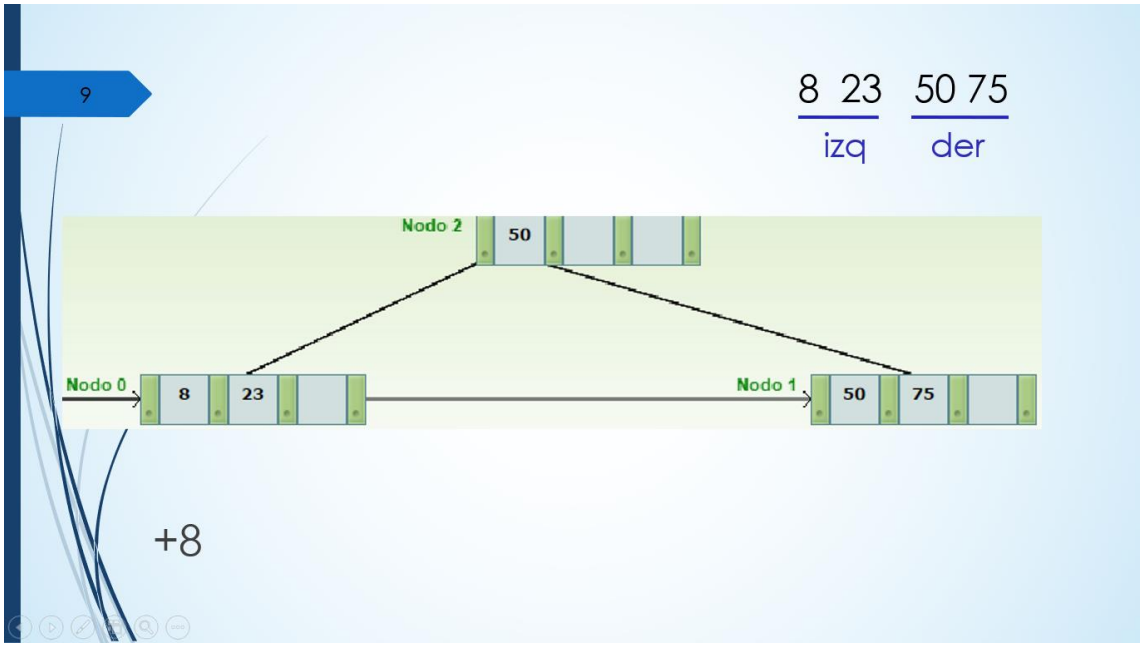
Caso contrario -> igual tratamiento que en árboles B.

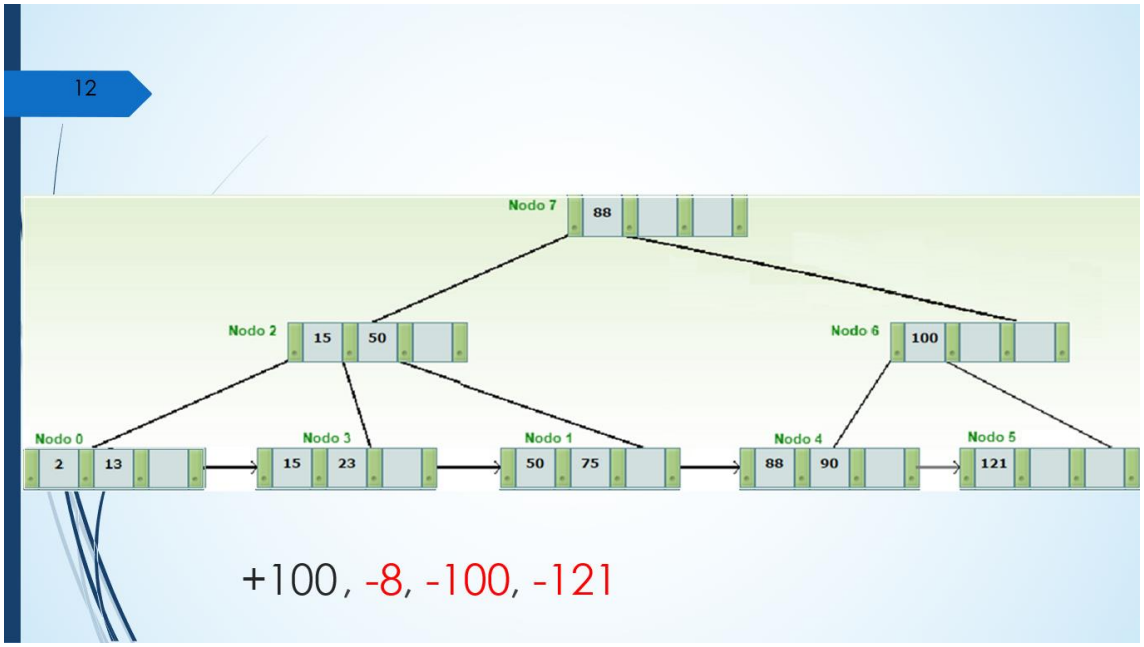
BAJA

La operación de eliminación en árboles B+ es más simple que en árboles B. Esto ocurre porque las claves a eliminar siempre se encuentran en las páginas hojas. En general deben distinguirse los siguientes casos, dado un árbol B+ de orden M:

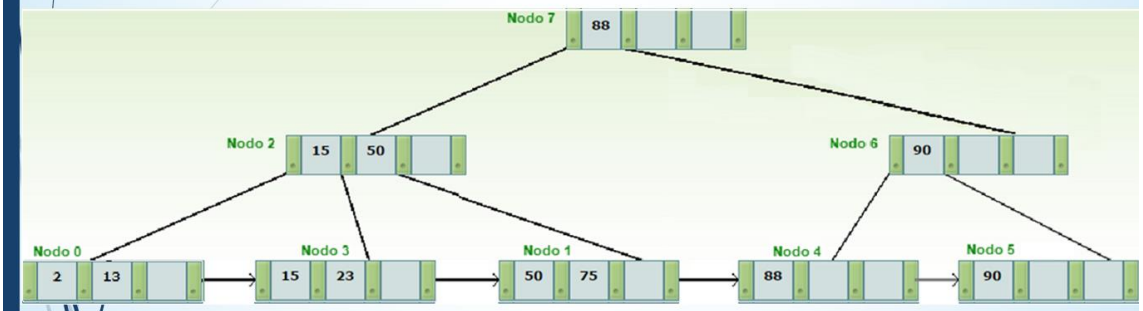
- Si al eliminar una clave, la cantidad de claves que queda es mayor o igual que [M/2]-1, entonces termina la operación. Las claves de los nodos raíz o internos no se modifican por más que sean una copia de la clave eliminada en las hojas.
- Si al eliminar una clave, la cantidad de llaves es menor a [M/2]-1, entonces debe realizarse una **redistribución** de claves, tanto en el índice como en las páginas hojas.
- Si la redistribución no es posible, entonces debe realizarse una **fusión** entre los nodos.


Nodo 0
23
50
75
+8


9
8 23 50 75
izq der
Nodo 2
50
Nodo 0
8
23
Nodo 1
50
75
+8


12
Nodo 7
88
Nodo 2
15
50
Nodo 6
100
Nodo 0
2
13
Nodo 3
15
23
Nodo 1
50
75
Nodo 4
88
90
Nodo 5
121
+100, -8, -100, -121

12



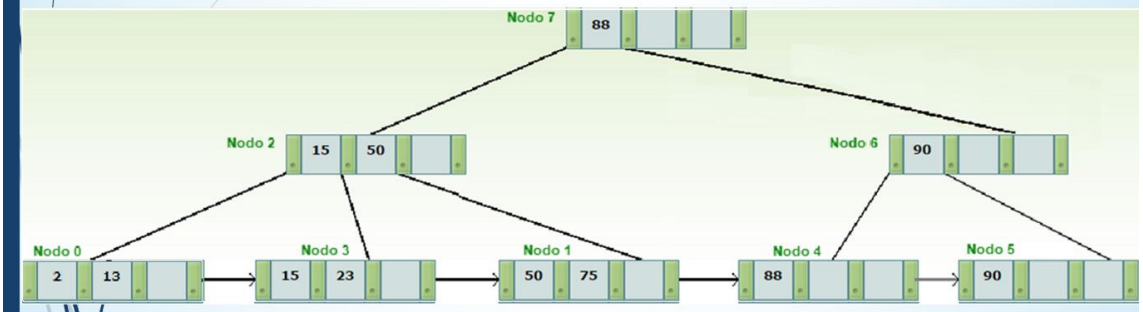
+100, -8, -100, -121

12

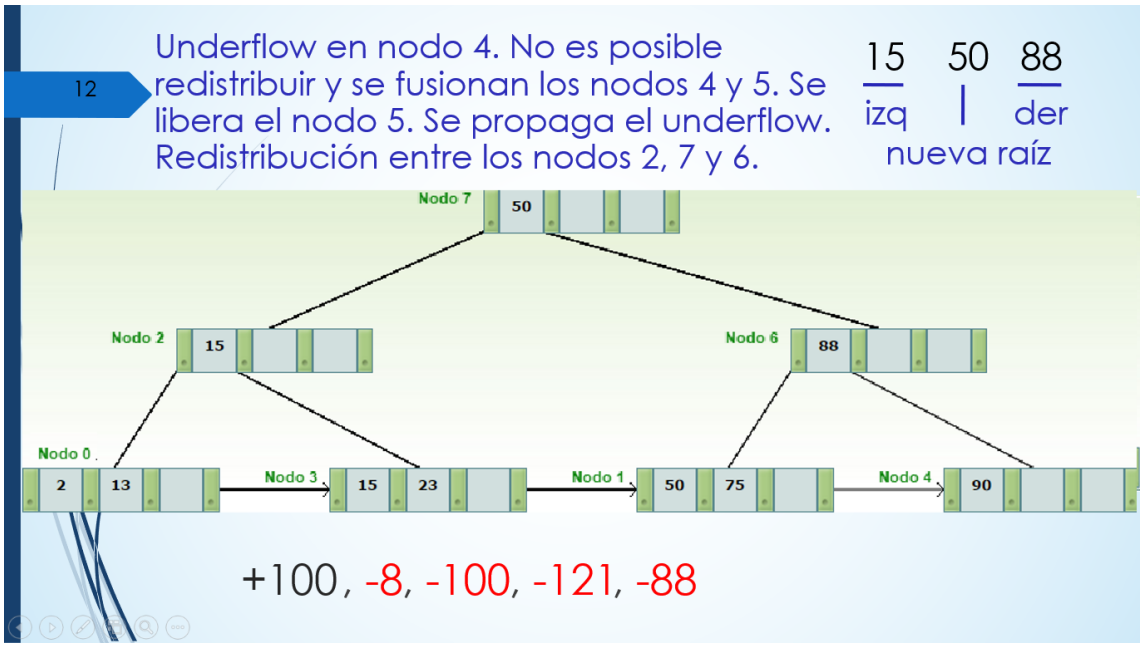
Underflow en nodo 4. No es posible redistribuir y se fusionan los nodos 4 y 5. Se libera el nodo 5. Se propaga el underflow. Redistribución entre los nodos 2, 7 y 6.

15 50 88

izq der



+100, -8, -100, -121, -88



HASHING

Dispersión de Archivos

- Técnica para generar una dirección base única para una clave dada.
- Convierte la clave en un número aleatorio, que luego sirve para determinar dónde se almacena la clave.
- Utiliza una función de dispersión para mapear cada clave con una dirección física de almacenamiento.
- Utilizada cuando se requiere acceso rápido por clave.

TERMINOS

- Sinonimo: Las claves que pertenecen a un mismo registro son sinonimos.
- Colision: situación en la que un registro es asignado, por función de dispersión, a una dirección que ya posee uno o más registros.
- Registro en saturación: Situación en la que un registro es direccionado a un nodo que no dispone de capacidad para almacenarlo.

DENSIDAD DE EMPAQUETAMIENTO

DE= número de registros / cantidad de espacio total (capacidad de nodo x nro de nodos)

HASHING ESTATICO

TIPOS TRATAMIENTO OVERFLOW

- Saturación progresiva
  - Cuando se completa el nodo, se busca el próximo hasta encontrar uno libre.
  - Cuando hay una baja marco con ### la cubeta si se elimina una clave de una cubeta llena y la siguiente cubeta tiene datos.

- Saturación progresiva encadenada
    - Similar a saturación progresiva, pero los reg. de saturación se encadenan y “no ocupan” necesariamente posiciones contiguas.
    - Si la clave en la dirección base a insertar es intrusa, se le asigna un nuevo lugar y a partir de la dirección base de la intrusa se modifica el enlace a su nueva dirección. Luego se inserta la nueva clave en su dirección base.
    - Cuando inserto siguiendo un enlace, es como en lista invertida. Si inserto una nueva clave, en el enlace pongo la dirección que referenciaba la dirección base, y en la dirección base pongo la dirección en donde se almaceno la nueva clave.
    - Si doy de baja una clave que esta en el medio de una cadena de sinónimos, el enlace que tenía la dirección de esa clave lo copio en la que referenciaba la dirección de la clave eliminada.
    - Si doy de baja una clave que está en el inicio de una cadena de sinónimos, reemplazo el contenido de la dirección por el contenido de la dirección que estaba apuntando el nodo de la clave eliminada.
    - Si esta al final simplemente modifico el enlace del que lo apuntaba.

NOTA: Cada nodo lo leo una sola vez.

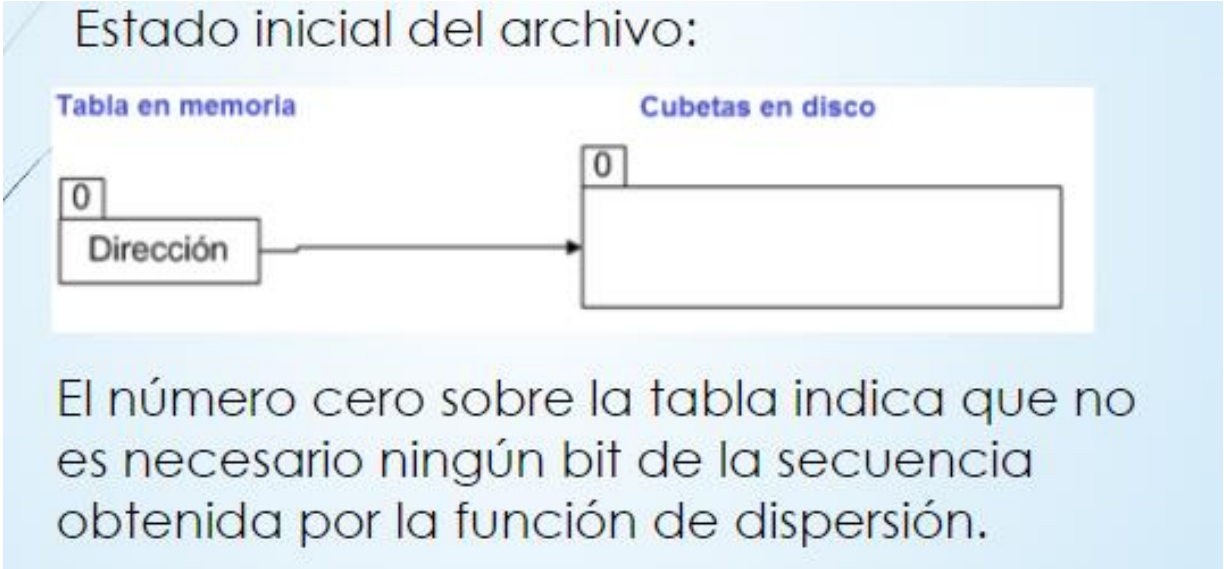
- Saturación progresiva encadenada con área de desborde separada
    - Las claves que general Overflow van a nodos especiales.
    - El enlace se trata como la Saturación progresiva encadenada.
    - Si la baja es en el área principal solo escribo la cubeta sin la clave a eliminar, no se modifican los enlaces
- Dispersión doble
    - Se resuelven overflows aplicando una segunda función de dispersion a la clave para producir un desplazamiento. La segunda funcion se suma a la primera (dirección original) tantas veces como sea necesario hasta encontrar una dirección con espacio.
    - Siempre que se elimina se deja una marca.

HASHING DINAMICO

Van aumentando la cantidad de nodos a medida que los necesite.

Función de dispersión: Retorna 32 bits.

Se empieza tomando 0 bits de la función de Hashing, por lo que se agregan sin problemas.



Cuando se produce Overflow:

1) Se incrementa en uno el valor asociado a la cubeta saturada.

2) Se genera una nueva cubeta con el mismo valor asociado a la cubeta saturada.

3) Se compara el valor de la cubeta con el valor asociado a la tabla.

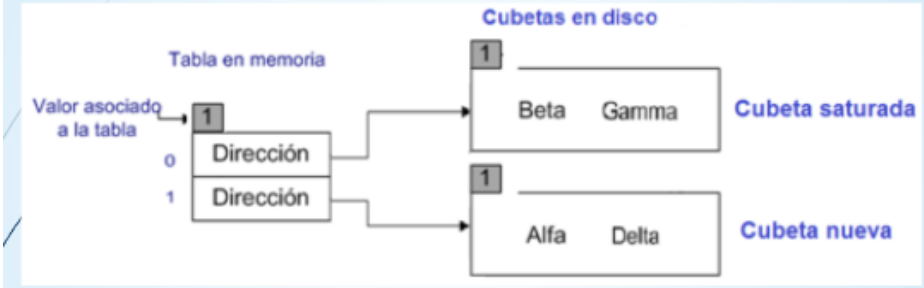
Si el primero es mayor que el segundo: la tabla no dispone de entradas suficientes para direccionar a la nueva cubeta, hace falta generar más direcciones por lo que la cantidad de celdas de la tabla se duplica y el valor asociado a la tabla se incrementa en uno.

Si el primero no es mayor la tabla posee direcciones suficientes para direccionar a la nueva cubeta y la cantidad de celdas NO debe ser duplicada

Clave	f(clave)
Alfa	00.....1001
Beta	00.....0100
Gamma	00.....0010
Delta	**00.....1111**

Inserción de Epsilon

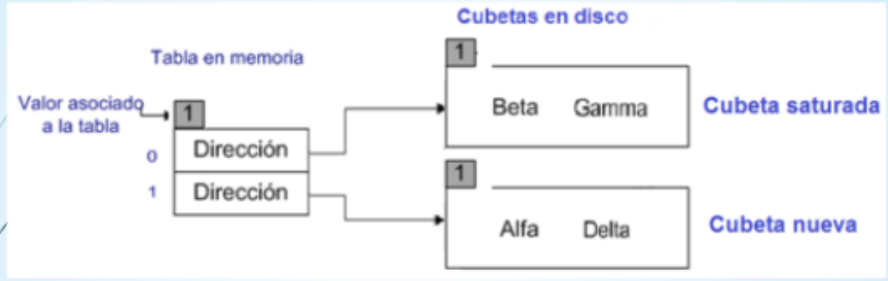
Clave	f(clave)
Epsilon	00.....0000



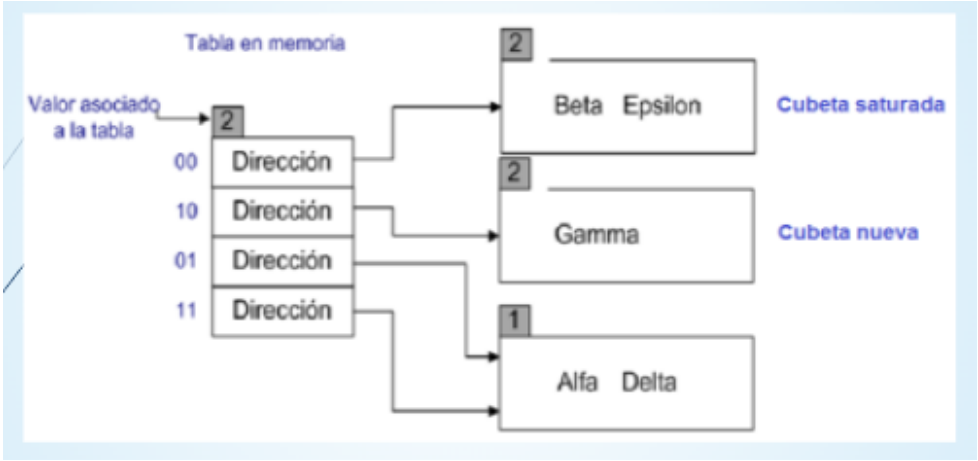
Epsilon debe ser almacenado en la cubeta asociada a la celda 0 de la tabla. La misma se encuentra completa lo que genera un nuevo **desborde**.

Inserción de Epsilon

Clave	f(clave)
Epsilon	00.....0000



Al no disponer de celdas suficientes en la tabla en memoria principal, se **duplica** el espacio disponible, que a partir de este momento necesita 2 bits de la función de hash para poder direccionar un registro.



Se redispersan solamente las claves de las cubetas involucrada.