

Fundamentos de Organización de Datos

Bienvenidos

La Cátedra

Profesores:

Mg. Rodolfo Bertone

Mg. Thomas Pablo

Trabajos Prácticos:

JTP: Lic. Sobrado Ariel

JTP: Lic. Nucilli Emanuel

JTP: APU Cipollone Juan

Clases

- **Clases**

- Teóricas
- Explicaciones de Prácticas (donde se presentan ejemplos)
- Prácticas
- Se utilizará plataforma moodle para mensajería y material: **asignaturas.info.unlp.edu.ar**

Enlace provisorio de material práctico:

<https://drive.google.com/drive/folders/1icJHyH90OSXCu95qWicEb91sGanitZdh?usp=sharing>



Temas a ver:

- Persistencia de Datos:
 - Archivos
- Acceso a datos, performance en el acceso:
 - Acceso secuencial indizado (árboles)
 - Acceso Directo (Hashing)



Evaluación

- Se evalúan todos los temas vistos
- Cada tema se aprueba de forma independiente.

Fechas de Examen:

- **1º Fecha Martes 10/6**
- **2º Fecha Martes 1/7**
- **3º Fecha Martes 15/7**



Cambios de turno

Selección de turno en <https://fod.info.unlp.edu.ar/>
Para los cambios de turnos(solo con certificado laboral o cambio con otro compañero de otro turno) tienen como fecha límite el **Martes 25 de Marzo sin excepción.**

Bibliografía

- Introducción a las Bases de Datos. Conceptos Básicos (Bertone, Thomas)
- Estructuras de Archivos (Folk-Zoellick)
- Files & Databases: An Introduction (Smith-Barnes)
- Fundamentos de Bases de Datos (Korth Silvershatz)

Fundamentos de Organización de Datos

Archivos

Tipos de Archivos

Registros de longitud fija o tipados (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.

Bloques de bytes (File)

Operaciones básicas

Definición de Archivos tipados

Dos formas:

- **var** archivo_logico: **file of** tipo_de_dato;

- **type**

archivo = **file of** tipo_de_datos;

var archivo_logico: archivo

Ejemplo

11

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');
```

Operaciones

Apertura y creación de archivos

rewrite (nombre_logico) ; → Crea un archivo

reset (nombre_logico) ; → Abre un archivo existente

Ejemplo:

rewrite (enteros) ;

reset (personas) ;

Operaciones

Cierre de archivos

```
close (nombre_logico) ;
```

Transfiere la información del buffer al disco.

Ejemplo:

```
close (enteros) ;
```

```
close (personas) ;
```

Operaciones

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;

Operaciones adicionales

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.


```
program creacion_archivo;  
type  
    persona = record  
        dni: string[8]  
        apellidoyNombre: string[30];  
        direccion: string[40];  
        sexo      : char;  
        salario   : real;  
    end;  
    archivo_personas = file of  
persona;  
  
var  
    personas: archivo_personas;  
    nombre_fisico: string[12];  
    per: persona;
```

begin

```
write('Ingrese el nombre del archivo: ');  
readln(nombre_fisico);
```

{enlace entre el nombre lógico y el nombre físico}

```
assign(personas, nombre_fisico);
```

{apertura del archivo para creación}

```
rewrite(personas);
```

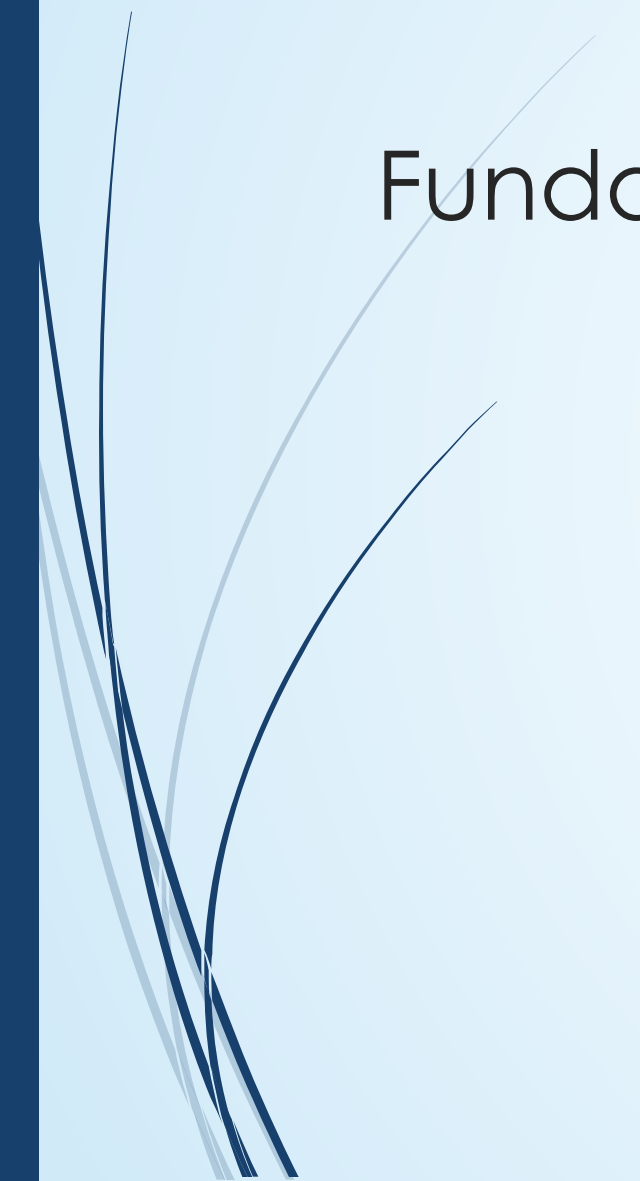
```
        {lectura del DNI una persona}
write('Ingrese el dni de la persona: ');
readln(per.dni);
while (per.dni <> '') do begin
    {lectura del resto de los datos de la persona}
    write('Ingrese el apellido y nombre de la persona:
');
        readln(per.apellidoyNombre);
        write('Ingrese la dirección de la persona: ');
readln(per.direccion);
        write('Ingrese el sexo de la persona: ');
readln(per.sexo);
        write('Ingrese el salario de la persona: ');
        readln(per.salario);
        {escritura del registro de la persona en el archivo}
        write(personas, per);
        {lectura del DNI de una nueva persona}
        write('Ingrese otro dni o blanco para terminar: ');
        readln(per.dni);
end;
{cierre del archivo}
close(personas);
end.
```



¿Dudas?

QR acceso provisorio al material de práctica





Fundamentos de Organización de Datos

Archivos de texto y binarios

Ejemplo-archivo de texto- binario

Type

```
tRegistroVotos=Record
```

```
  codProv: integer;
```

```
  codLoc: integer;
```

```
  nroMesa: integer;
```

```
  cantVotos: integer;
```

```
  desc:String;
```

```
end;
```

```
tArchVotos=File of tRegistroVotos;
```

Var

```
opc: Byte;
```

```
nomArch, nomArch2: String;
```

```
arch:
```

```
tArchVotos;
```

```
carga: Text { archivo de texto con datos de los votos,  
se lee de el y se genera archivo binario.}
```

```
votos: tRegistroVotos;
```

Ejemplo-archivo de texto- binario

3

```
Begin
WriteLn('VOTOS');
WriteLn;
WriteLn('0. Terminar el Programa');
WriteLn('1. Crear un archivo binario desde un arch
texto');
WriteLn('2. Abrir un archivo binario y exportar a texto');
Repeat
Write('Ingresa el nro. de opcion: '); ReadLn(opc);
If (opc=1) or (opc=2) then begin
WriteLn;
Write('Nombre del archivo de votos: ');
ReadLn(nomArch);
Assign(arch, nomArch);
end;
```


Ejemplo-archivo de texto- binario

4

{Opción 1 crea el archivo binario desde un texto}

Case opc of

1: begin

**Que sucede cuando tengo más de 1 campo
String en un archivo de texto?**

Write('Nombre del archivo de carga: ');

ReadLn(nomArch2);

Assign(carga, nomArch2);

Reset(carga); *{abre archivo de texto con datos}*

Rewrite(arch); *{crea nuevo archivo binario}*

while (not eof(carga)) **do begin**

ReadLn(carga, votos.codProv, votos.codLoc,
votos.nroMesa, votos.cantVotos, votos.desc); *{lectura del
archivo de texto}*

Write(arch, votos); *{escribe binario}*

end;

Write('Archivo cargado.');

ReadLn;

Close(arch); **Close**(carga); *{cierra los dos
archivos}* **end;**

Ejemplo-archivo de texto- binario

5

{Opcion 2 exporta el contenido del binario a un texto}

2: begin

Write('Nombre del archivo de texto: ');

ReadLn(nomArch2);

Assign(carga, nomArch2);

Reset(arch); *{abre archivo binario}*

Rewrite(carga); *{crea archivo de texto, se utiliza el mismo de opcion 1 a modo ejemplo}*

while (not eof(arch)) do begin

Read(arch, votos); *{lee votos del arch binario}*

WriteLn(carga,votos.codProv,' ',votos.codLoc,' ',
votos.nroMesa,' ',votos.cantVotos,' ',votos.desc); *{escribe en el archivo texto los campos separados por el carácter blanco}*

end;

Close(arch); Close(carga)

end;

Until (opc = 0);

End



¿Dudas?

Fundamentos de Organización de Datos

Archivos

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.

Ejemplo: *archivo conteniendo las ventas sobre esos productos.*

Algorítmica clásica sobre archivos

Importante: Analizar las precondiciones de cada caso particular.

Los algoritmos a desarrollar deben tener en cuenta estas precondiciones, caso contrario determina la falla de su ejecución.

Actualización de un archivo maestro con un archivo detalle - 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: Definición de tipos

type

producto = record

cod: **string**[4];

descripcion: **string**[30];

pu: **real**; *{precio unitario}*

stock: **integer**;

end;

venta_prod = record

cod: **string**[4];

cant_vendida: **integer**;

end;

maestro = file of producto;

detalle = file of venta_prod;

Ejemplo: variables y operaciones

var

mae: maestro;

det: detalle;

regm: producto;

regd: venta_prod;

begin *{ Inicio del programa }*

assign (mae, 'maestro.dat');

assign (det, 'detalle.dat');

reset (mae);

reset (det);

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.
```

Actualización de un archivo maestro con un archivo detalle

- 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.

Ejemplo: Definición de tipos

type

producto = record

cod: **string**[4];

descripcion: **string**[30];

pu: **real**;

stock: **integer**;

end;

venta_prod = record

cod: **string**[4];

cant_vendida: **integer**;

end;

detalle = file of venta_prod;

maestro = file of producto;

Ejemplo: variables y operaciones

var

mae: maestro;

det: detalle;

regm: producto;

regd: venta_prod;

cod_actual: **string**[4];

tot_vendido: **integer**;

begin *{Inicio del programa}*

assign (mae, 'maestro');

assign (det, 'detalle');

reset (mae);

reset (det);

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 totaliza la cantidad vendida del detalle}

```
cod_actual := regd.cod;
```

```
tot_vendido := 0;
```

```
while (regd.cod = cod_actual) do begin
```

```
    tot_vendido:=tot_vendido+regd.cant_vendida;
```

```
    read(det, regd);
```

```
end;
```

{Se actualiza el stock del producto con la cantidad vendida del mismo}

```
regm.stock := regm.stock - tot_vendido;
```



```
{se reubica el puntero en el maestro}  
seek (mae, filepos (mae) -1) ;  
{se actualiza el maestro}  
write (mae, regm) ;  
end ;  
close (det) ;  
close (mae) ;  
end .
```

¿Diferencia entre este ejemplo y el anterior?

Se agrega una iteración que permite agrupar todos los registros del detalle que modificarán a un elemento del maestro.

¿Inconvenientes de esta solución?

La segunda operación **read** sobre el archivo detalle se hace sin controlar el fin de datos del mismo. Podría solucionarse agregando un **if** que permita controlar dicha operación, pero cuando finaliza la iteración interna, al retornar a la iteración principal se lee otro registro del archivo detalle, perdiendo así un registro.

Actualización de un archivo maestro con un archivo detalle

```
const valoralto = 'ZZZZ';  
type str4 = string[4];  
  producto = record  
    cod: str4;  
    descripcion: string[30];  
    pu: real;  
    stock: integer;  
  end;  
  venta_prod = record  
    cod: str4;  
    cant_vendida: integer;  
  end;  
detalle = file of venta_prod;  
maestro = file of producto;
```

Ejemplo

var

mae: maestro;
det: detalle;
total: **integer**;

regm: producto;
regd: venta_prod;
aux: str4;

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.stock := regm.stock - 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.
```



¿Dudas?

Fundamentos de Organización de Datos

Archivos

Corte de Control

Algorítmica clásica sobre archivos

Corte de Control

Proceso mediante el cual la información de un archivo es presentada en forma organizada de acuerdo a la estructura que posee el archivo.

Algorítmica clásica sobre archivos

Ejemplo

Se almacena en un archivo la información de ventas de una cadena de electrodomésticos. Dichas ventas han sido efectuadas por los vendedores de cada sucursal de cada ciudad de cada provincia del país.

Es necesario informar al gerente de ventas de la empresa, el total vendido en cada sucursal, ciudad y provincia, así como el total final.

Ejemplo – Formato

Provincia:

Ciudad:

Sucursal:

Vendedor 1 Total \$\$

...

Vendedor N Total \$\$

Total Sucursal: Total \$\$

Sucursal:

Vendedor 1 Total \$\$

...

Vendedor N Total \$\$

Total Sucursal: Total \$\$

.....

Total Ciudad: \$\$

Ciudad:

...

Total Ciudad: \$\$

Total Provincia: \$\$

Provincia:

...

Total Ciudad: \$\$

Total Provincia: \$\$

Total Empresa: \$\$

Ejemplo: Precondiciones

- El archivo se encuentra ordenado por provincia, ciudad y sucursal
- En diferentes provincias pueden existir ciudades con el mismo nombre, y en diferentes ciudades pueden existir sucursales con igual denominación.

Ejemplo

```
program ejemplo;  
  const valor_alto = 'ZZZ';  
  type  
    nombre = string[30];  
    reg_venta = record  
      vendedor: integer;  
      monto: real;  
      sucursal: nombre;  
      ciudad: nombre;  
      provincia: nombre;  
  end;  
  
  ventas = file of reg_venta;
```

var

reg: reg_venta;

archivo: ventas;

total, totProv, totCiudad, totSuc: **real**;

prov, ciudad, sucursal: nombre;

procedure leer(**var** archivo: ventas;
 var dato: reg_venta);**begin** **if** (**not** (**EOF**(archivo))) **then** **read** (archivo, dato) **else**

dato.provincia := valor_alto;

end;


```
{programa principal}
```

```
begin
```

```
  assign (archivo, 'archivo_ventas');
```

```
  reset (archivo);
```

```
  leer (archivo, reg);
```

```
  total := 0;
```

```
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.
```

Fundamentos de Organización de Datos

Archivos
Merge

Algorítmica clásica sobre archivos

Merge

Proceso mediante el cual se genera un nuevo archivo a partir de otros archivos existentes.

Ejemplo – Merge

```
program ejemplo;  
  const valor_alto = 999999;  
  type  
    producto = record  
      codigo: LongInt;  
      descripcion: string[30];  
      pu: real;  
      cant: integer;  
    end;  
  arc_productos = file of producto;
```

var

```
det1, det2, det3, mae: arc_productos;  
min, regd1, regd2, regd3: producto;
```

```
procedure leer (var archivo: arc_productos;  
                var dato: producto);
```

begin

```
    if (not(EOF(archivo))) then
```

```
        read (archivo, dato)
```

```
    else
```

```
        dato.codigo := valor_alto;
```

end;


```
procedure minimo(var det1, det2, det3: arc_productos;  
                 var r1, r2, r3, min: producto);  
begin  
    if (r1.codigo<=r2.codigo) and (r1.codigo<=r3.codigo) then begin  
        min := r1;  
        leer(det1, r1);  
    end  
    else  
        if (r2.cod <= r3.cod) then begin  
            min := r2;  
            leer(det2, r2);  
        end  
        else begin  
            min := r3;  
            leer(det3, r3)  
        end;  
end;
```

{programa principal}

begin

assign (mae, 'maestro');

assign (det1, 'detalle1');

assign (det2, 'detalle2');

assign (det3, 'detalle3');

rewrite (mae);

reset (det1);

reset (det2);

reset (det3);

leer (det1, regd1);

leer (det2, regd2);

leer (det3, regd3);

minimo (det1, det2, det3,
 regd1, regd2, regd3, min);

*{se procesan todos los registros de los
archivos detalle}*

```
while (min.codigo <> valoralto) do begin  
    write (mae, min);  
    minimo (det1, det2, det3,  
            regd1, regd2, regd3, min);  
  
end;  
close (det1);  
close (det2);  
close (det3);  
close (mae);  
end.
```

Otra variante – Productos repetidos en los archivos detalles

```
while (min.codigo <> valoralto) do begin
  aux:= min;
  total := 0;
  while (min.codigo = aux.codigo) do begin
    total := total + min.cant;
    minimo (det1, det2, det3,
           regd1, regd2, regd3, min);
  end;
  aux.cant := total;
  write (mae, aux);
end;
```

Fundamentos de Organización de Datos

Archivos
Bajas

Algorítmica clásica sobre archivos

¿Qué es una baja?

Se denomina proceso de baja a aquel proceso que permite quitar información de un archivo.

El proceso de baja puede llevarse a cabo de dos modos diferentes:

- **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 Física

Se realiza baja física sobre un archivo cuando un elemento es efectivamente quitado del archivo, decrementando en uno la cantidad de elementos.

VENTAJA: En todo momento, se administra un archivo de datos que ocupa el lugar mínimo necesario.

DESVENTAJA: Performance de los algoritmos que implementan esta solución.

Técnicas de Baja Física

- Generar un nuevo archivo con los elementos válidos → sin copiar los que se desea eliminar
- Utilizar el mismo archivo de datos, generando los reacomodamientos que sean necesarios. (Solo para archivos sin ordenar)

Ejemplo: algoritmo

```
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);
{renombrar el archivo original para dejarlo
como respaldo}
rename(archivo, 'arch_empleados_old');
{renombrar el archivo temporal con el nombre
del original}
rename(archivo_nuevo, 'arch_empleados');
end.
```


Ejemplo: Baja lógica

```
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.
```

Técnicas

- **Recuperación de espacio:** Se utiliza el proceso de baja física periódicamente para realizar un proceso de **compactación del archivo**.


Quita los registros marcados como eliminados, utilizando cualquiera de los algoritmos vistos para baja física.
- **Reasignación de espacio:** Recupera el espacio utilizando los lugares indicados como eliminados para el ingreso de nuevos elementos al archivo (altas).

Ejemplo Reasignación de espacio

Marca de eliminado

Archivos de enteros

1156 115	304 304	228 228	998 998	116 116	504 504	824 824	597 597	715 715
------------------------	-----------------------	-----------------------	-----------------------	-----------------------	-----------------------	-----------------------	-----------------------	-----------------------

NRR

0

1

2

3

4

5

6

7

8

Eliminación de claves:

- 116
- 304
- 824

¿Desventajas de esta técnica?

Ejemplo Reasignación de espacio

Lista invertida

Archivos de enteros

-79	156	304	5228	988	0169	5040	8242	597
197	25	597	15					
0	1	2	3	4	5	6	7	8

Registro
Cabecera

Eliminación de claves:

- 116
- 304
- 824

Fundamentos de Organización de Datos

Árboles B

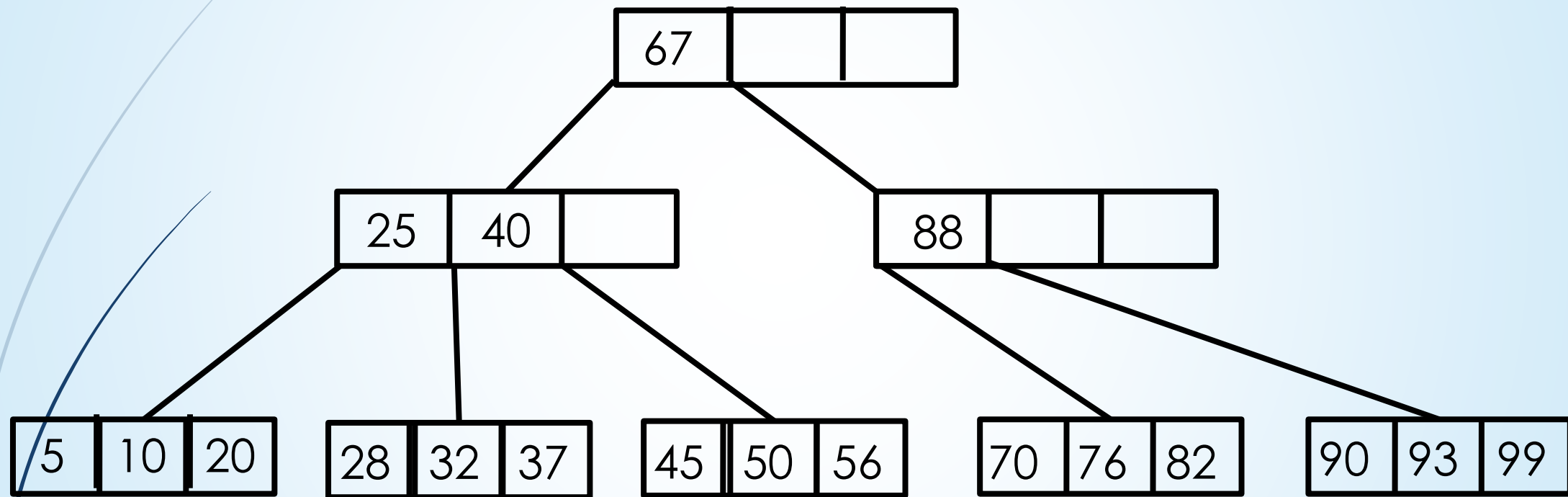
Árboles B y B+

Los árboles B son **árboles multcamino** con una **construcción especial en forma ascendente** que permite mantenerlos balanceados a bajo costo.

Propiedades de un Árbol B de orden M

- Cada nodo del árbol puede contener como **máximo M descendientes directos (hijos)**.
- La raíz no posee descendientes directos o tiene al menos dos.
- Un nodo interno con **X hijos** contiene **$X-1$ elementos**.
- Todos los nodos (salvo la raíz) tienen como **mínimo $\lceil M/2 \rceil - 1$ elementos** y como **máximo $M-1$ elementos**.
- Todos los **nodos terminales** se encuentran al **mismo nivel**.
- Cada nodo tiene sus **elementos ordenados por clave**.
Además, todos los elementos en el subárbol izquierdo de un elemento son menores o iguales que dicho elemento, mientras que todos los elementos en el subárbol derecho son mayores que ese elemento

Ejemplo de Árbol B de orden 4



¿Para qué usamos los árboles B?

Alternativas:

- Organizar el archivo de datos como un árbol B
- Organizar el archivo índice un árbol B

Archivo de datos como árbol B

const $M = \dots$; {orden del árbol}

type

TDato = **record**

codigo: longint;

nombre: string[50];

end;

TNodo = **record**

cant_datos: **integer**;

datos: **array**[1..M-1] **of** TDato;

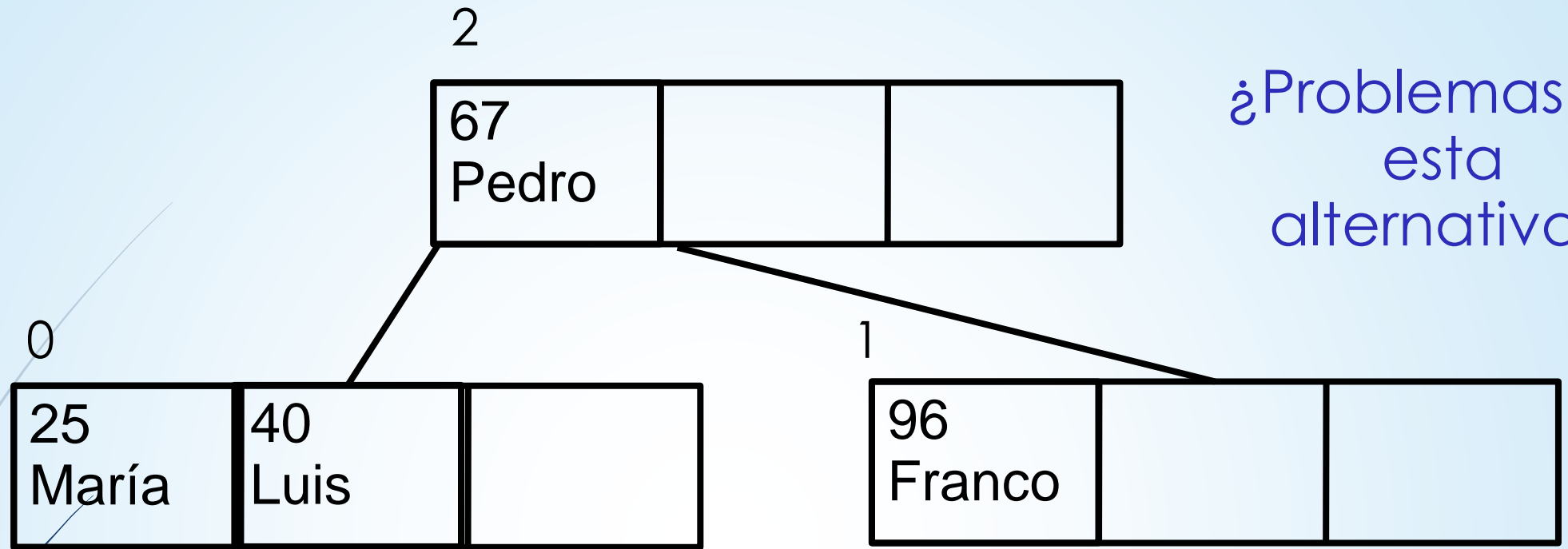
hijos: **array**[1..M] **of** integer;

end;

arbolB = **file of** TNodo;

var

archivoDatos: arbolB;



¿Problemas de
esta
alternativa?

Archivo:

datos:

25 María	40 Luis		cd: 2	96 Franco			cd: 1	67 Pedro			cd: 1
1	2	3		1	2	3		1	2	3	
-1	-1	-1		-1	-1			0	1		
1	2	3	4	1	2	3	4	1	2	3	4

NRR 0

NRR 1

NRR 2

Archivo índice como árbol B

const M = ... ; {orden del árbol}

type

TDato = **record**

codigo: longint;

nombre: string[50];

end;

TNodo = **record**

cant_claves: **integer**;

claves: **array**[1..M-1] **of longint**;

enlaces: **array** [1..M-1] **of integer**;

hijos: **array**[1..M] **of integer**;

end;

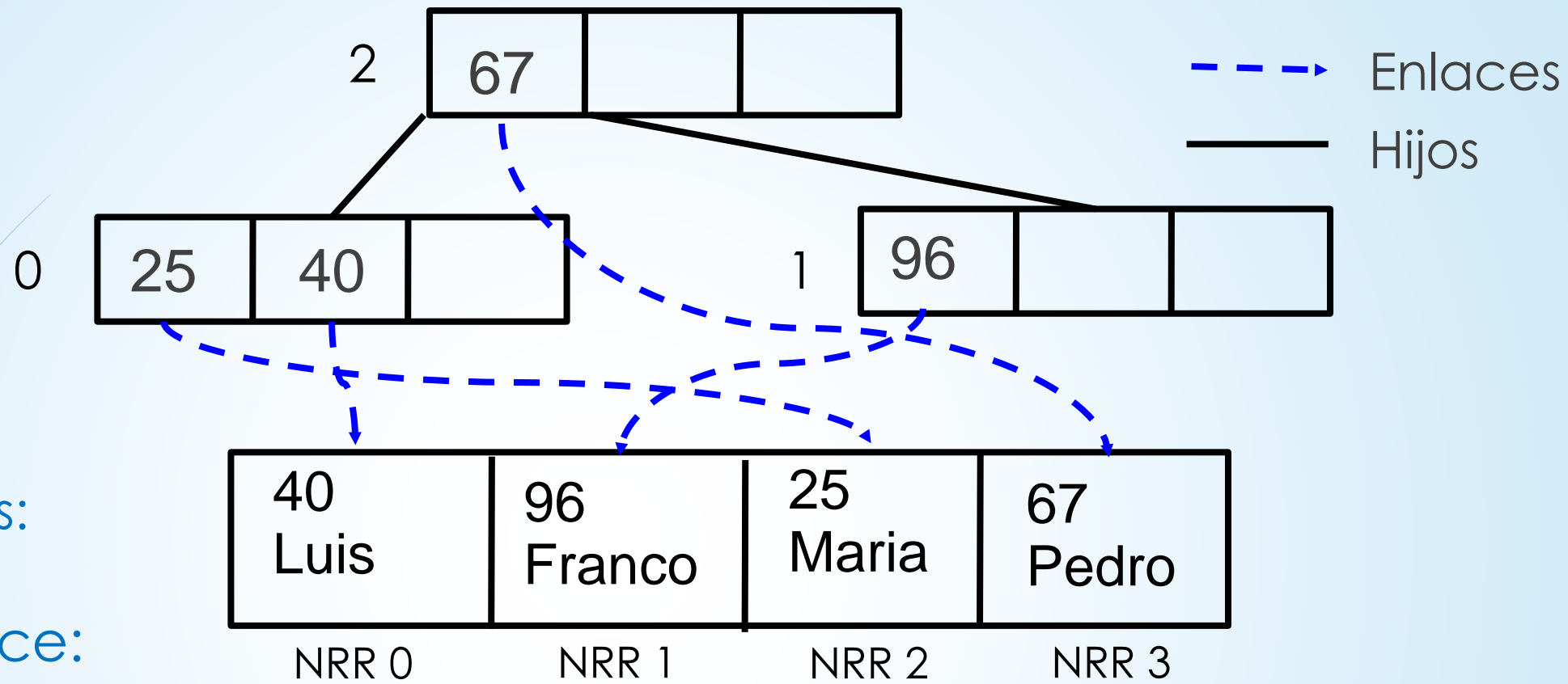
TArchivoDatos = **file of** TDato;

arbolB = **file of** TNodo;

var

archivoDatos: TArchivoDatos;

archivoIndice: arbolB;



Archivo Datos:

Archivo Indice:

claves:

25	40		cc: 2
----	----	--	-------

1 2 3

hijos:

-1	-1	-1	
----	----	----	--

1 2 3 4

enlaces:

2	0	
---	---	--

1 2 3

96			cc: 1
----	--	--	-------

1 2 3

-1	-1		
----	----	--	--

1 2 3 4

1			
---	--	--	--

1 2 3

67			cc: 1
----	--	--	-------

1 2 3

0	1		
---	---	--	--

1 2 3 4

3			
---	--	--	--

1 2 3

Ejemplo – Árbol B de orden 4

Árbol Inicial

Nodo 0

25	40	96
---------------	---------------	----

+40, +96, +25, +67

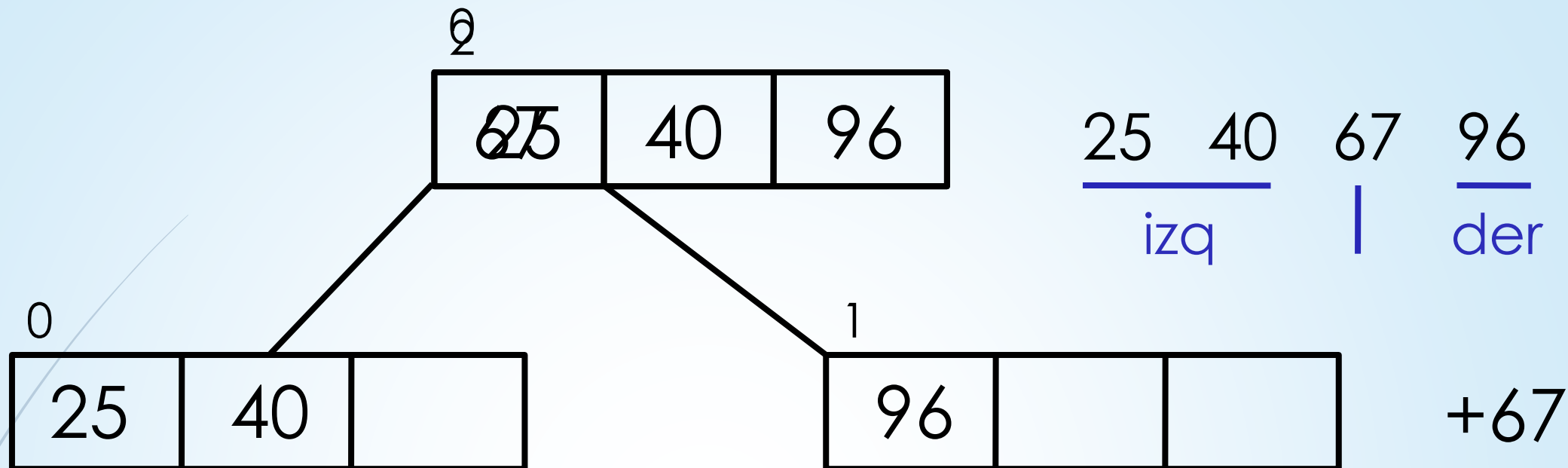
Archivo:

cant_claves: 0	25	40	96	
claves:	1	2	3	
hijos:	-1	-1	-1	-1
	1	2	3	4

NRR 0

Overflow

- 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.



División de la raíz. Se incrementa la altura del árbol.

Archivo:

claves:	25	40		cc: 2	96			cc: 1	67			cc: 1
	1	2	3		1	2	3		1	2	3	
hijos:	-1	-1	-1		-1	-1			0	1		
	1	2	3	4	1	2	3	4	1	2	3	4

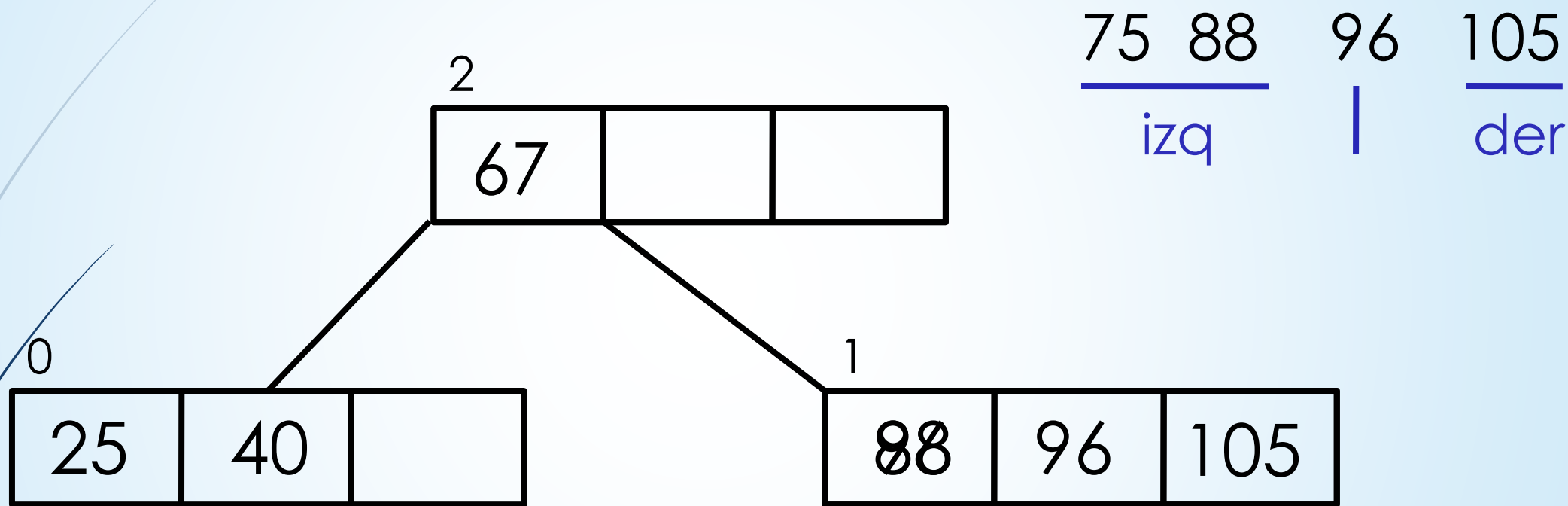
NRR 0

NRR 1

NRR 2

¡Notar la numeración de los nodos!

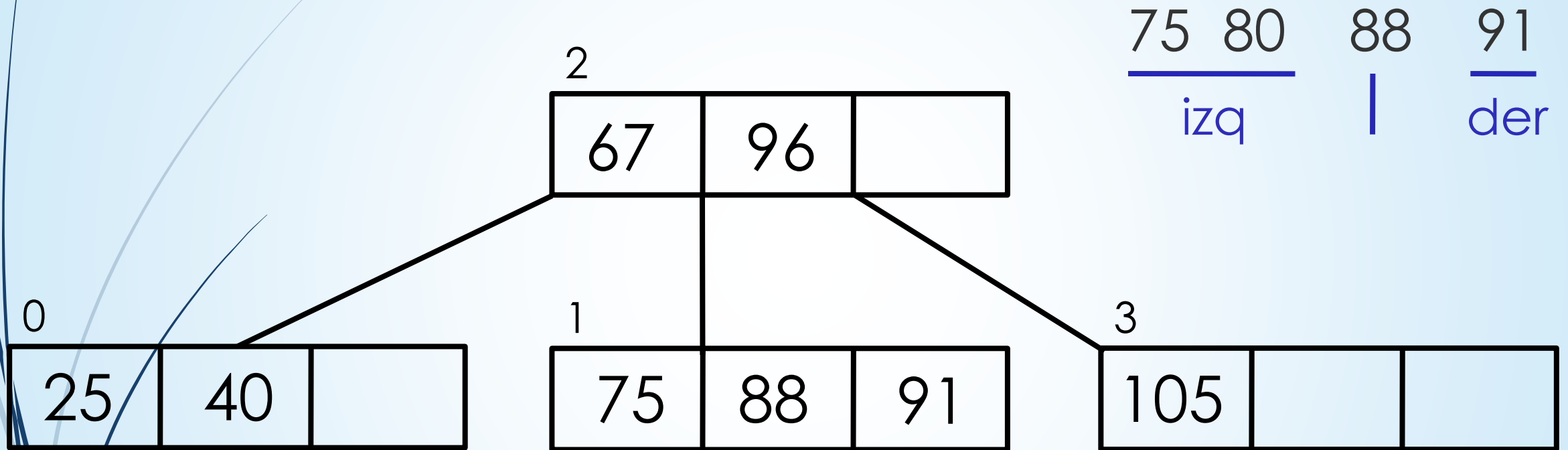
Ejemplo – Árbol B de orden 4



+88 , +105 , +75

Overflow en el nodo 1. División del mismo y promoción de la clave 96.

Ejemplo – Árbol B de orden 4

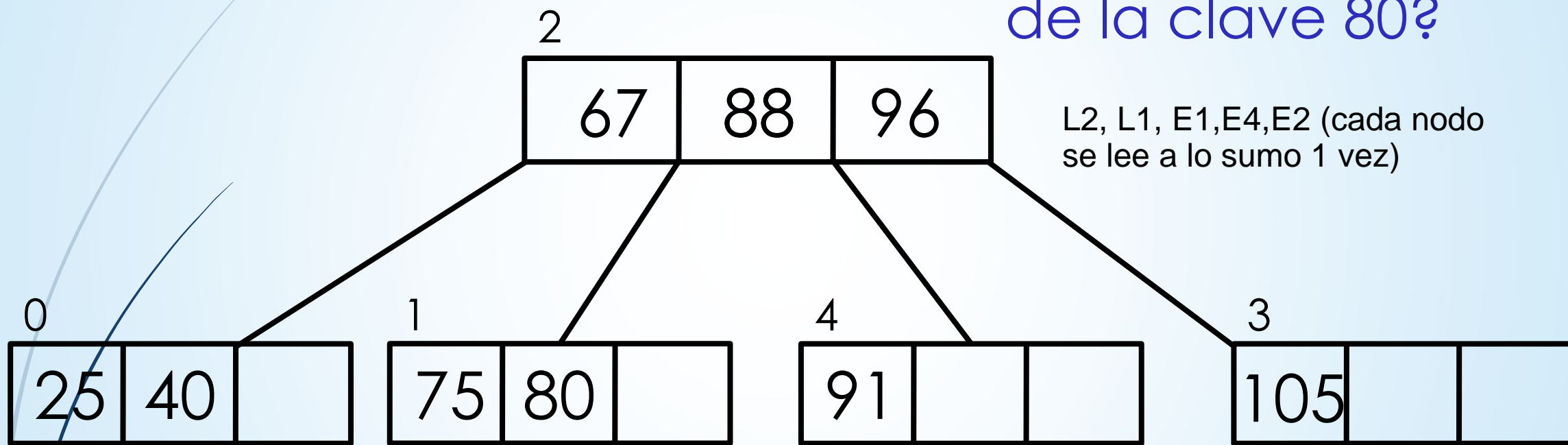


+75, +91, +80

Overflow en el nodo 1. División del mismo y promoción de la clave 88.

Ejemplo – Árbol B de orden 4

¿L/E necesarias para el alta de la clave 80?



+80 Completamos el árbol con las altas de:
+86, +120, +230, +95, +55

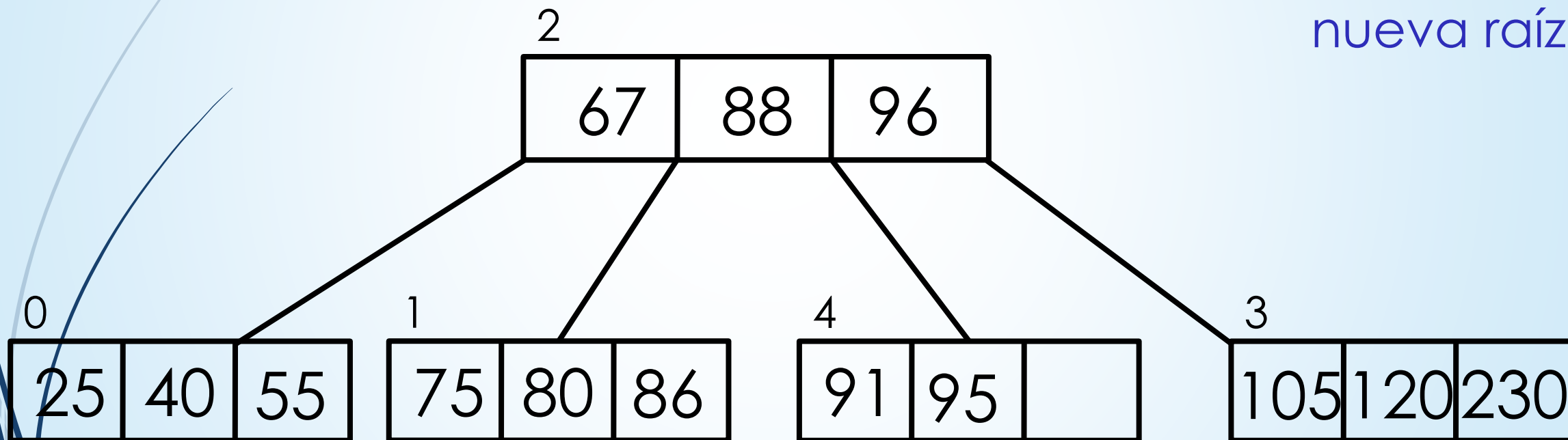
Overflow en el nodo 1. División del mismo y promoción de la clave 80.

Propagación del overflow a la raíz. División de la misma y aumento en la altura del árbol.

70	75	80	86
<hr/>			<hr/>
izq			der

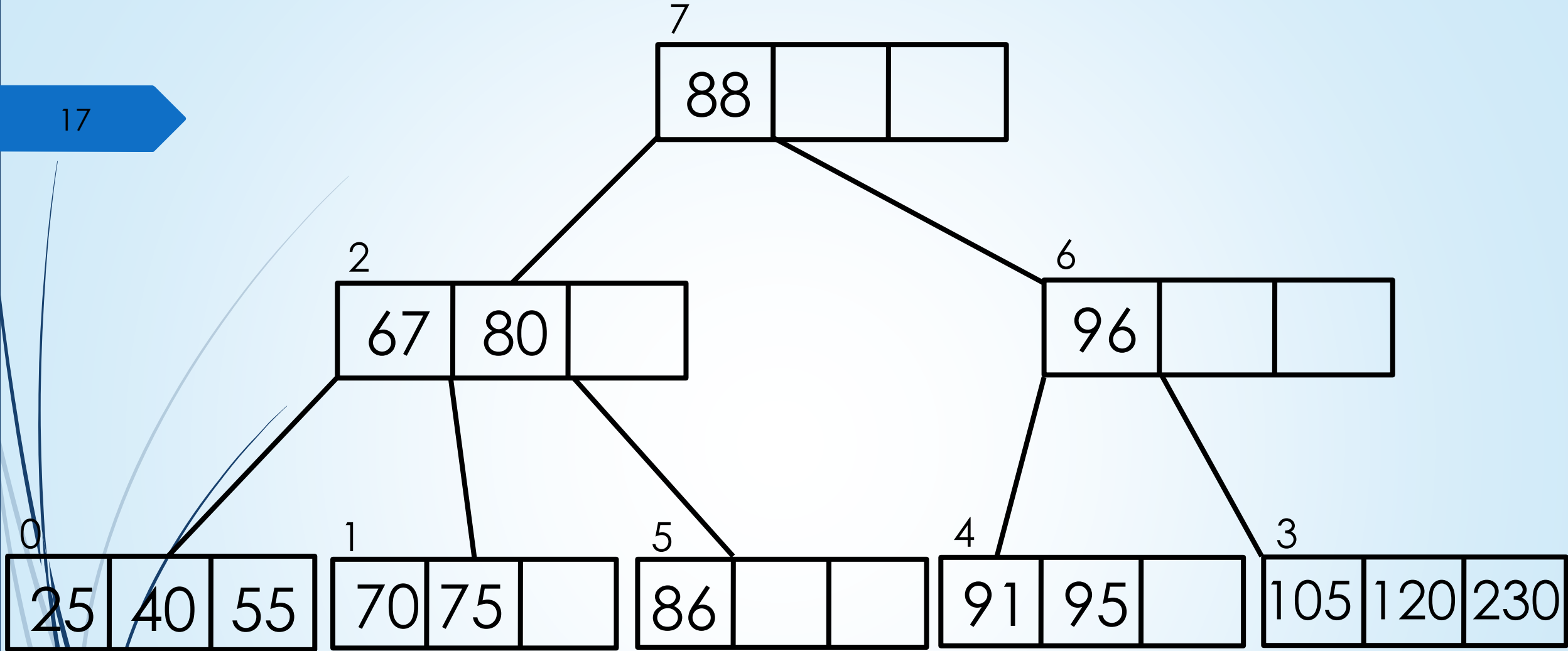
67	80	88	96
<hr/>			<hr/>
izq			der

nueva raíz



+86 , +120 , +230 , +95 , +55 . Alta de +70

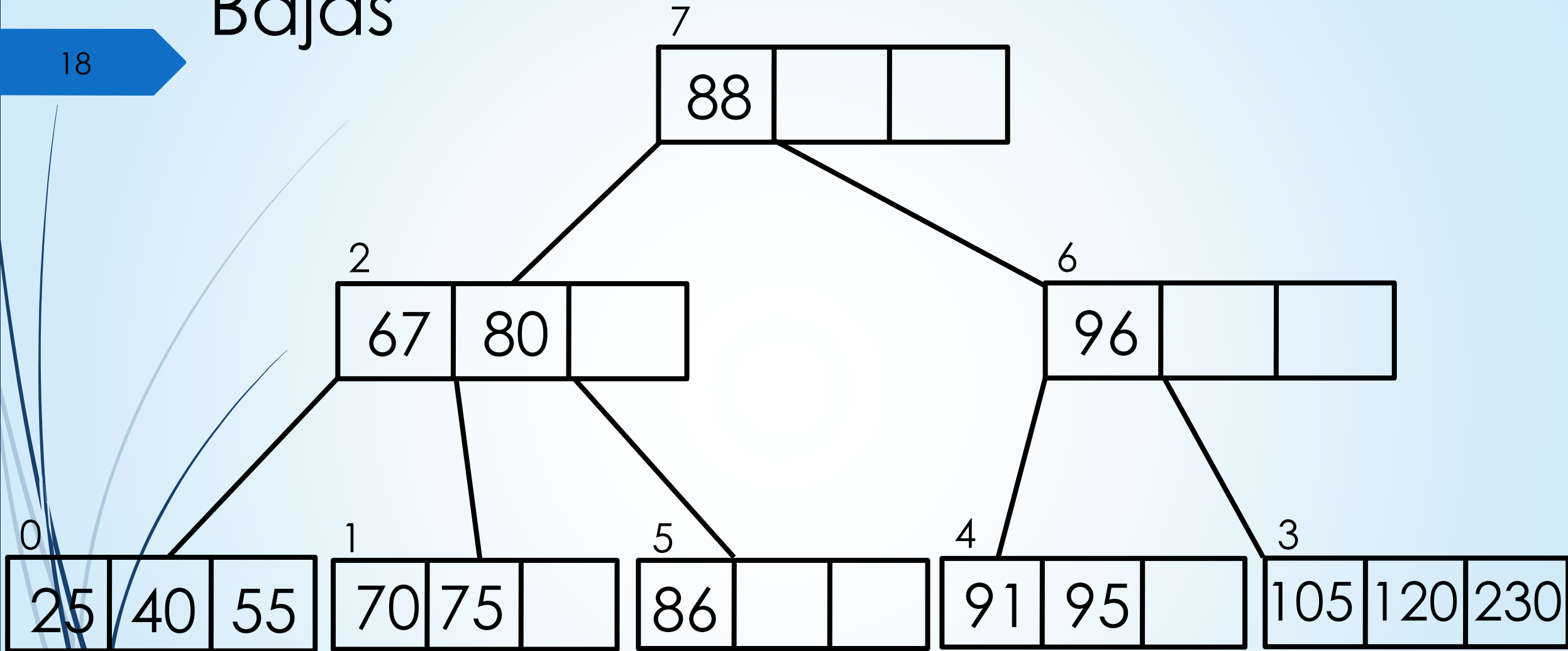
17



+70

Bajas

18



-75

Bajas

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

Bajas - 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.

Políticas para la resolución de underflow:

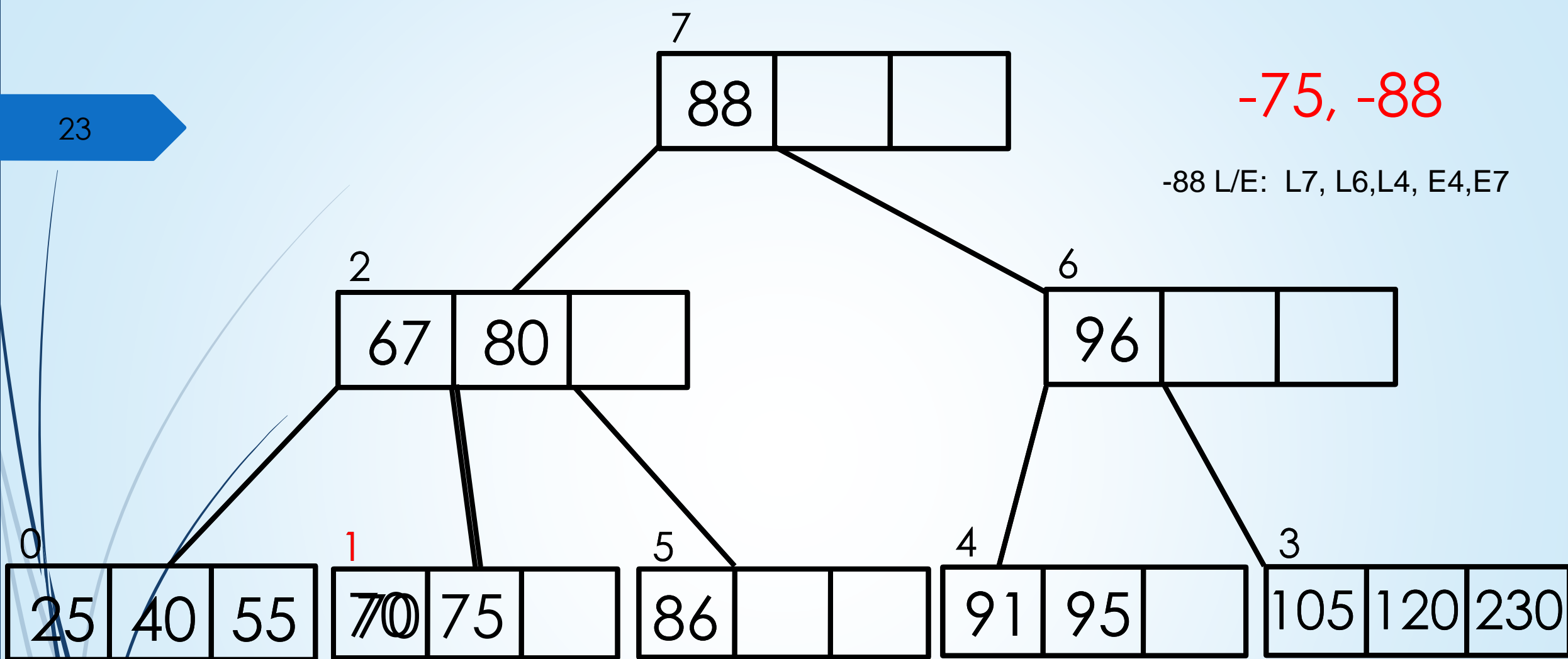
1. **Política izquierda:** se intenta redistribuir con el hermano adyacente izquierdo, si no es posible, se fusiona con hermano adyacente izquierdo.
2. **Política derecha:** se intenta redistribuir con el hermano adyacente derecho, si no es posible, se fusiona con hermano adyacente derecho.
3. **Política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

Políticas para la resolución de underflow:

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

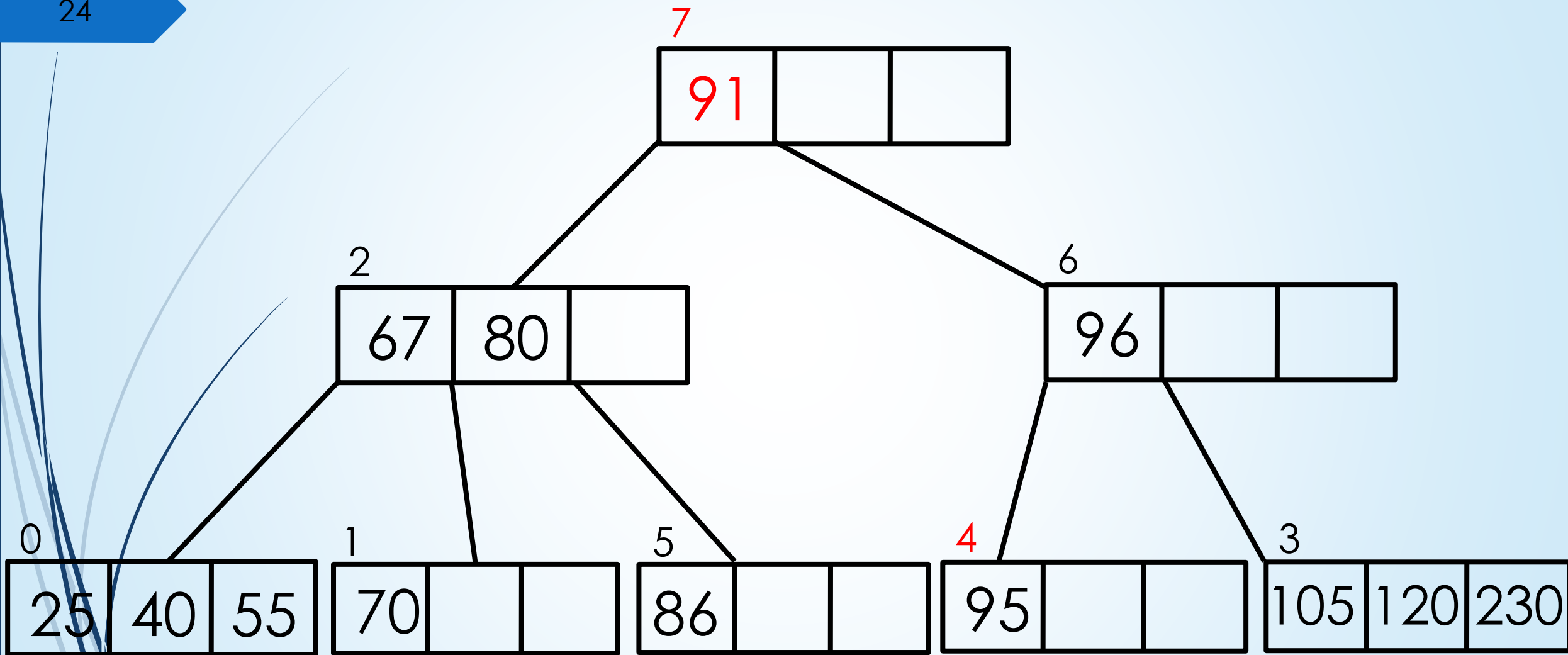


Eliminación de la clave 75 en el nodo 1.

Baja del 88, se reemplaza la clave por la menor clave del subárbol derecho.

No se genera underflow en la hoja

Ejemplo política derecha o izquierda



-88 , -70

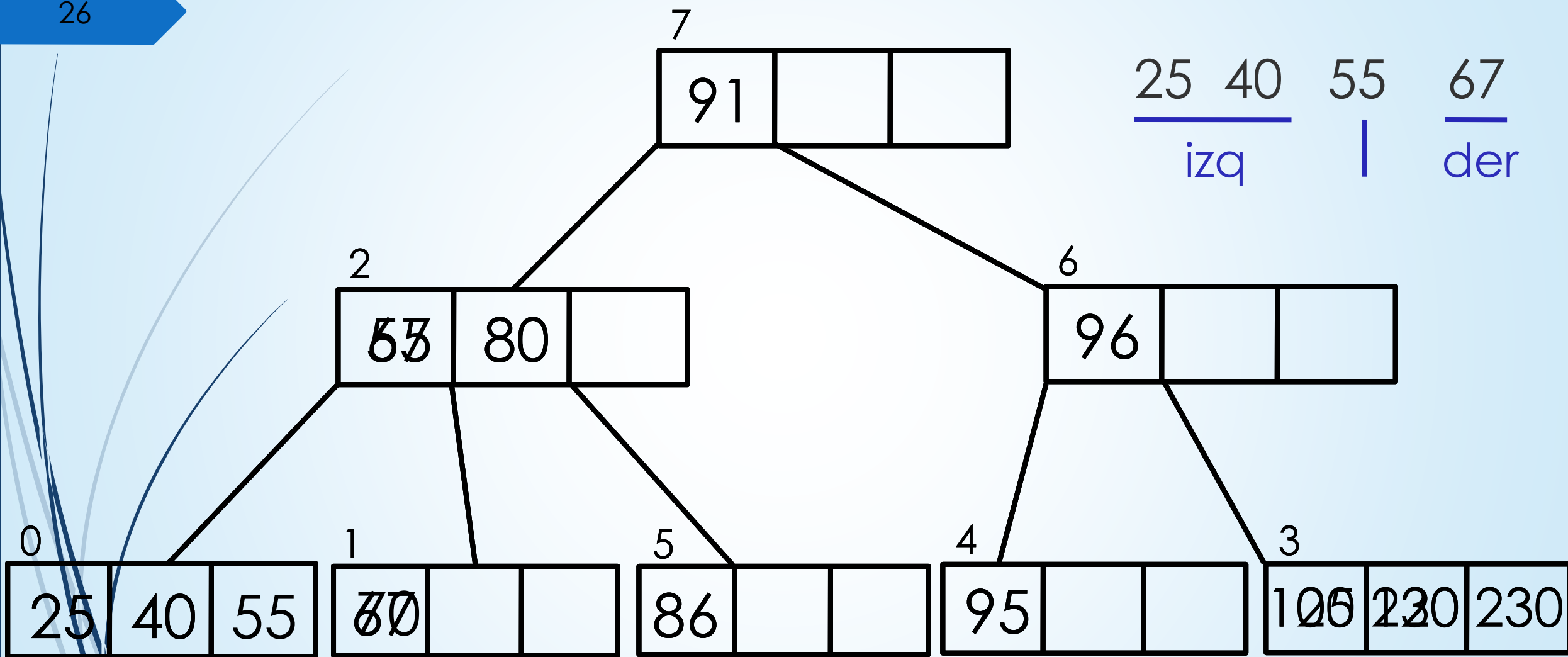
Baja de la clave 70 - política derecha o izquierda

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

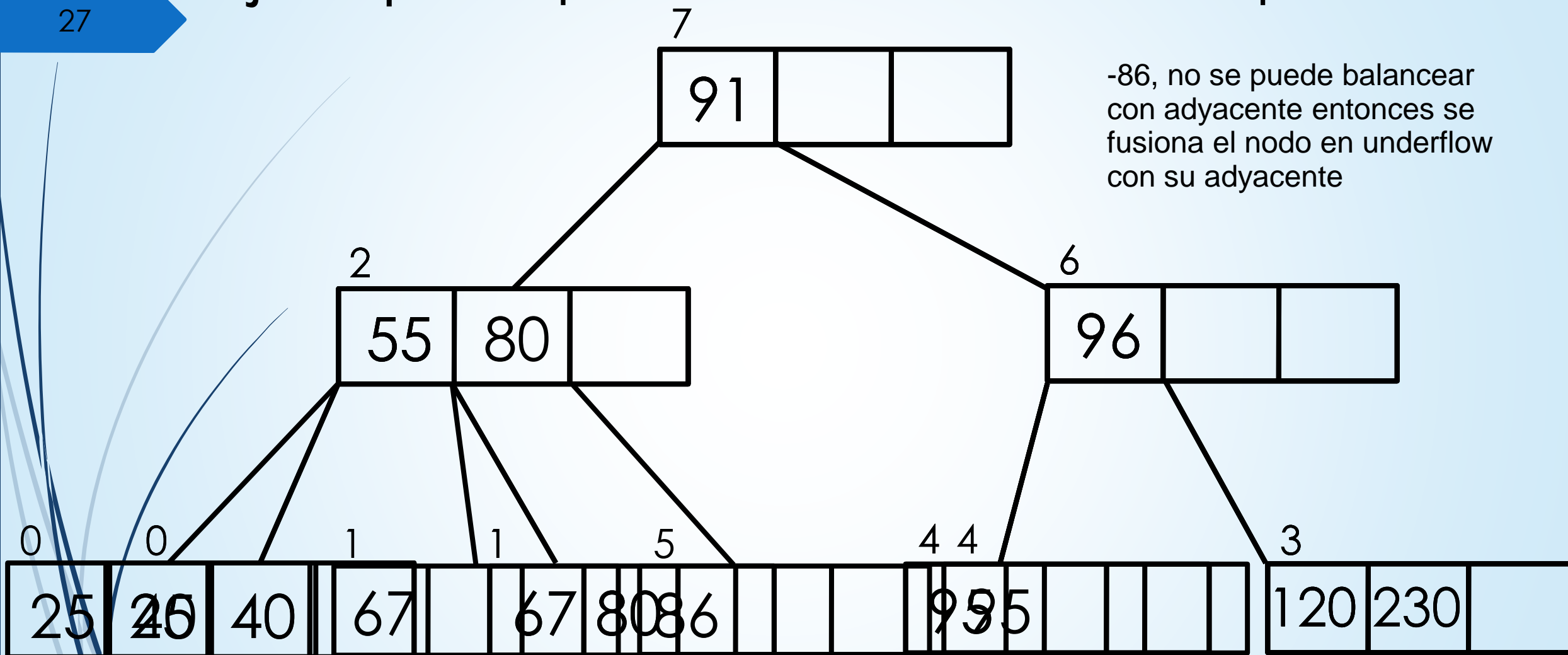
Se intenta redistribuir con el hermano derecho. No es posible ya que el nodo contiene la cantidad mínima de claves.

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

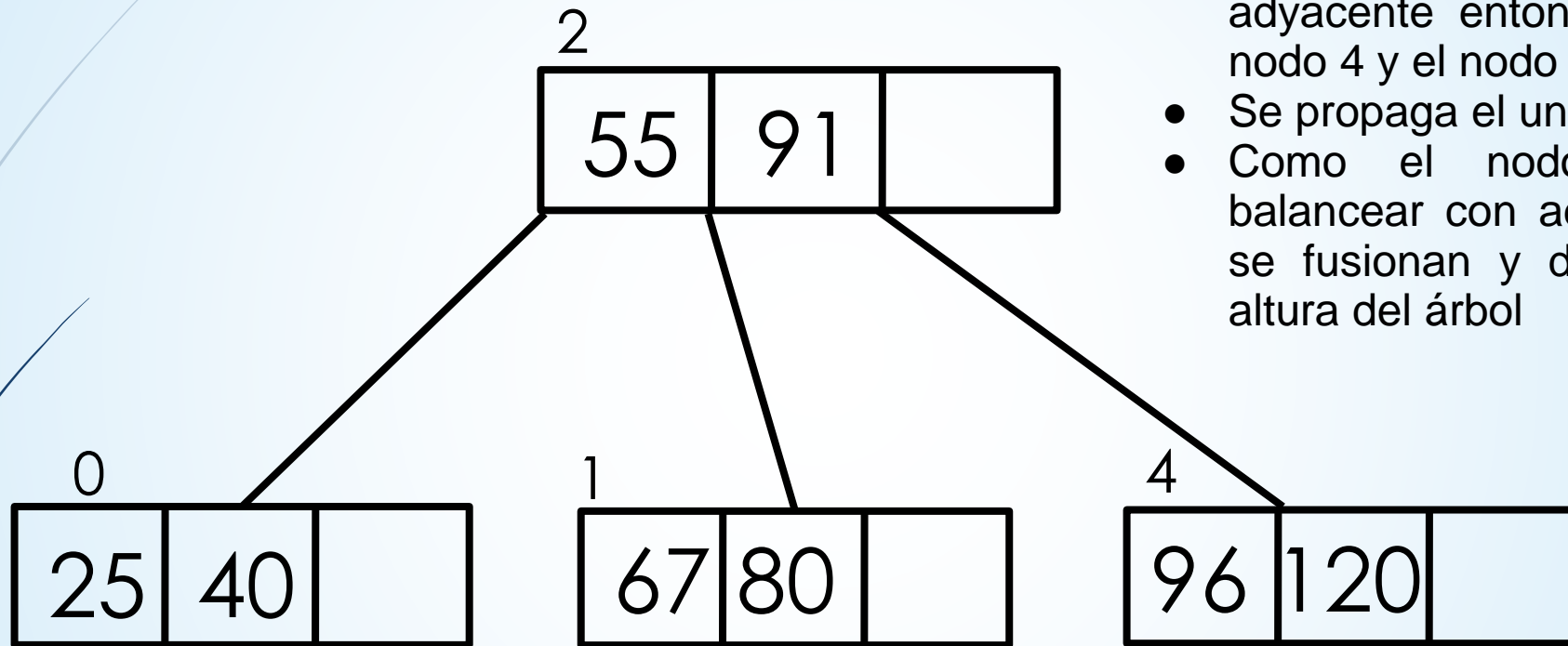


Ejemplo - política derecha o izquierda



-86, -230, -95

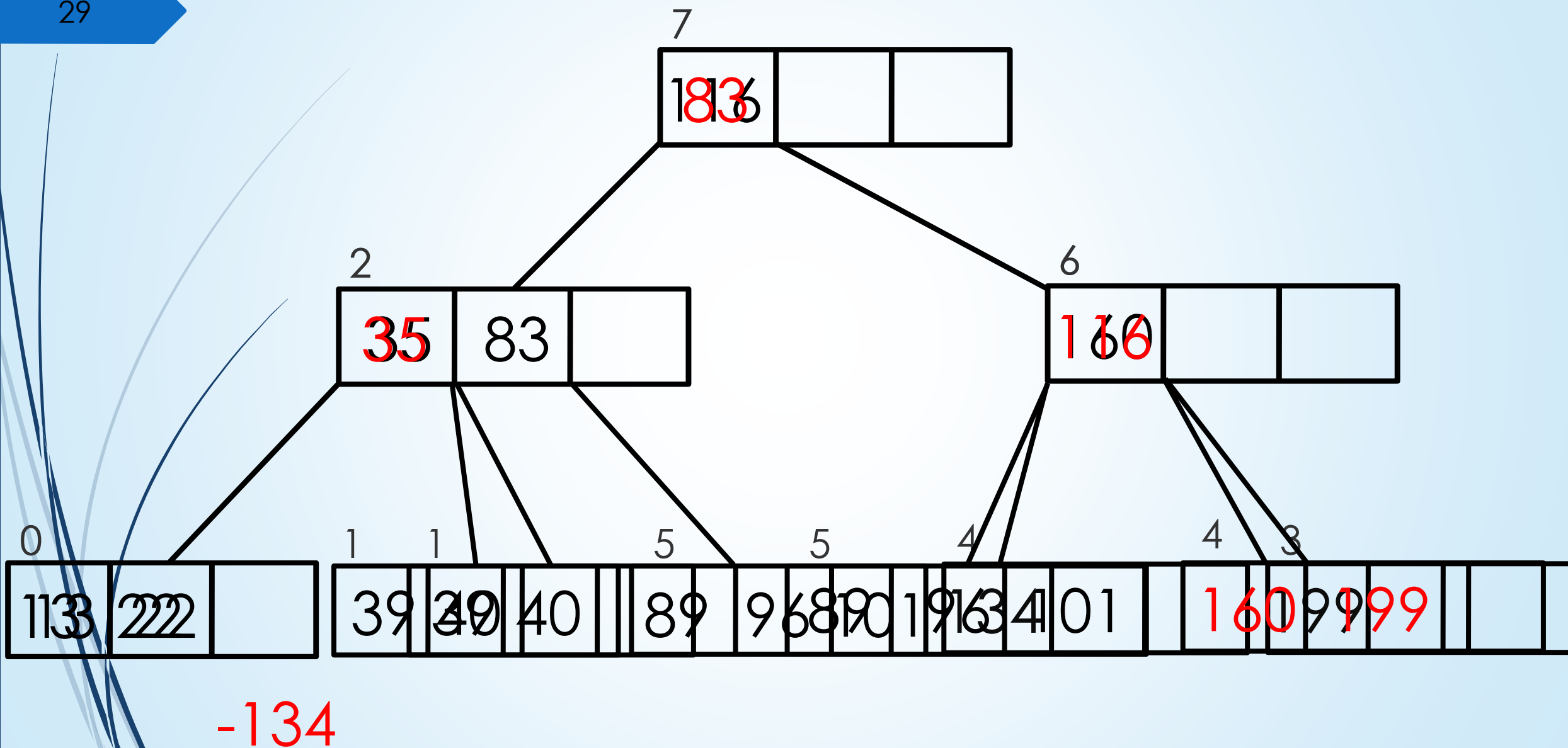
Ejemplo - política derecha o izquierda



- -95, no se puede balancear con adyacente entonces se fusiona el nodo 4 y el nodo 3
- Se propaga el underflow al nodo 6
- Como el nodo 6 no puede balancear con adyacente (nodo 2) se fusionan y disminuye en 1 la altura del árbol

-95

Ej: Redistribución en nodo interno



Fundamentos de Organización de Datos

Árboles B+

Árboles B+

- ✓ Constituyen una mejora sobre los árboles B, pues conservan la propiedad de acceso indizado rápido y permiten además un recorrido secuencial rápido.
- ✓ **Conjunto índice:** Proporciona acceso indizado a los registros. Todas las claves se encuentran en las hojas, duplicándose en la raíz y nodos interiores aquellas que resulten necesarias para definir los caminos de búsqueda.
- ✓ **Conjunto secuencia:** Contiene todos los registros del archivo. Las hojas se vinculan para facilitar el recorrido secuencial rápido. Cuando se lee en orden lógico, lista todos los registros por el orden de la clave.


Búsqueda B+

La operación de búsqueda en árboles B+ es similar a la operación de búsqueda en árboles B. El proceso es simple, ya que todas las claves se encuentran en las hojas, deberá continuarse con la búsqueda hasta el último nivel del árbol.

Inserción B+

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.

Bajas en B+

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 $\lceil M/2 \rceil - 1$, 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.

Bajas en B+

Underflow

- Si al eliminar una clave, la cantidad de llaves es menor a $\lceil M/2 \rceil - 1$, 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.

Políticas para la resolución de underflow:

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.

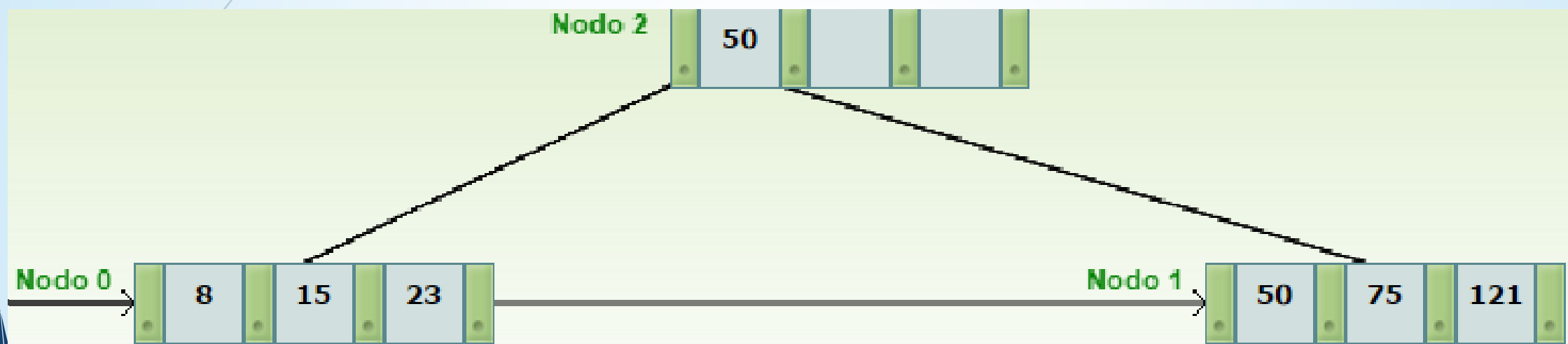
Árboles B+

Ejemplo con árbol de orden 4

Claves:

+50, +75, +23

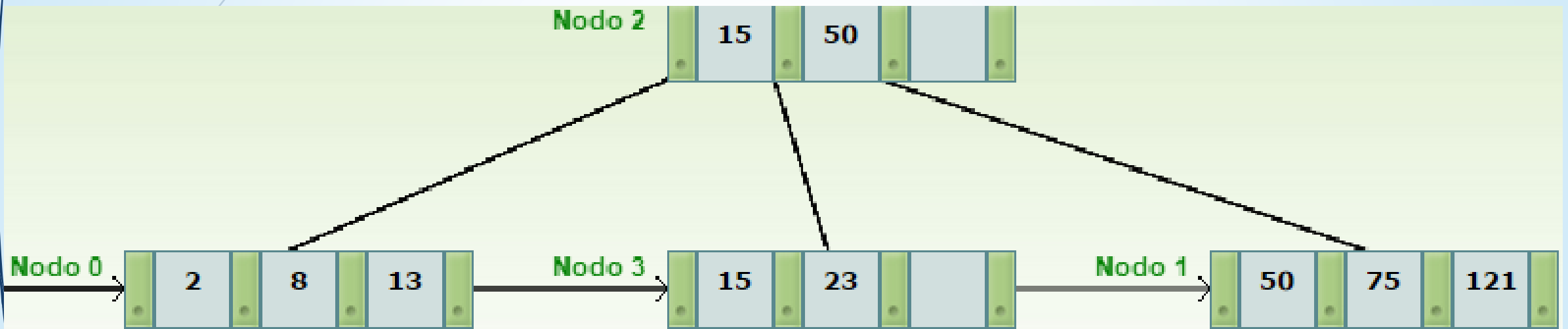
$\frac{8 \ 23}{\text{izq}}$ $\frac{50 \ 75}{\text{der}}$



+8 , +121, +15 , +2

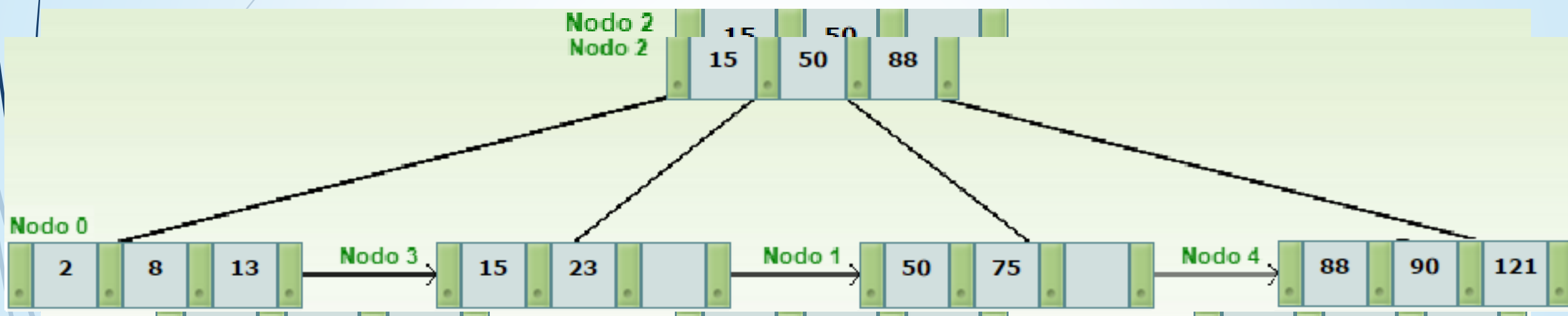
+2, +13, +88

$\frac{2 \ 8}{\text{izq}}$ $\frac{15 \ 23}{\text{der}}$



+88, +90, +100

50 75 88 121
 izq der

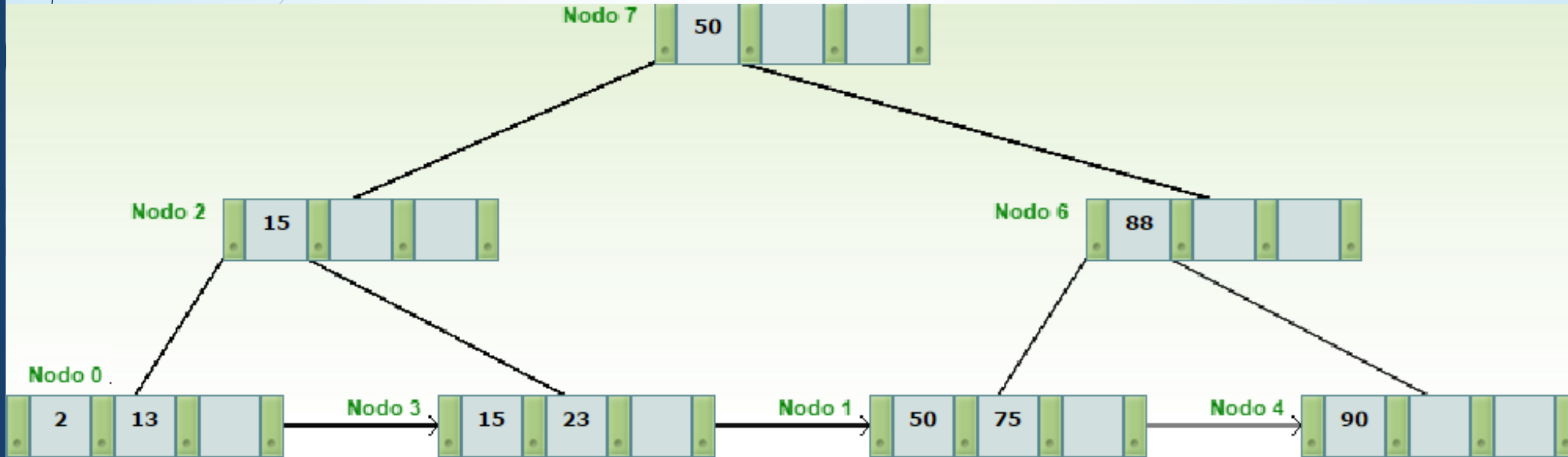


88 90 100 121
 izq der

15 50 88 100
 izq | der
 nueva raíz

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.

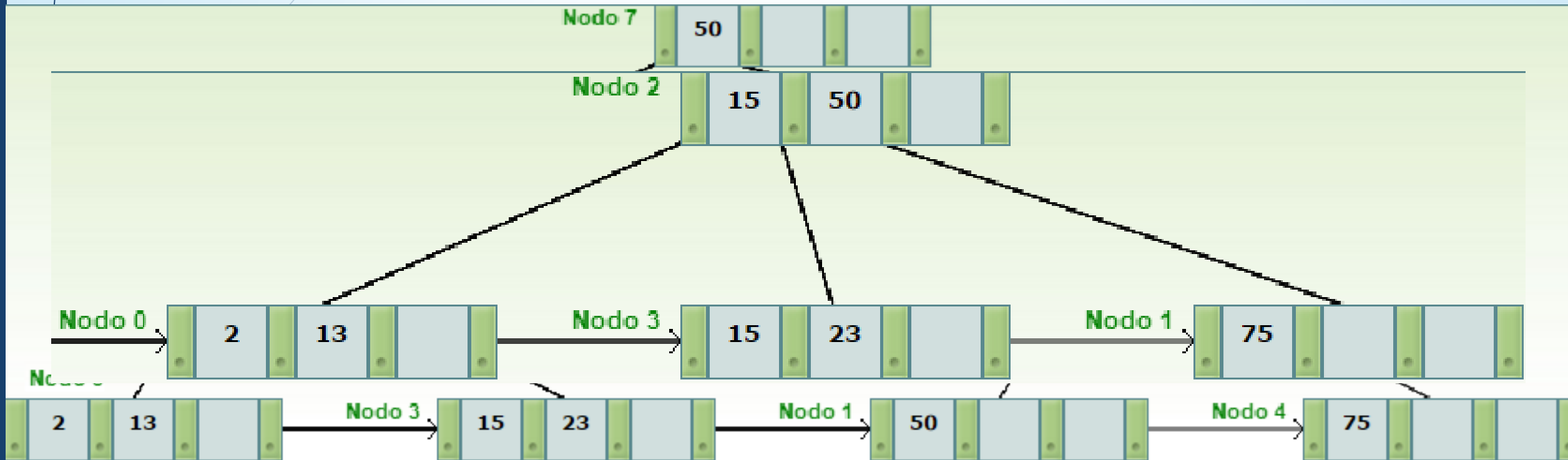
15	50	88
<hr/>		<hr/>
izq		der
nueva raíz		



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

-90, -50

Underflow en nodo 1. No es posible redistribuir y se fusionan los nodos 1 y 4. Se libera el nodo 4. Se propaga el underflow. No es posible redistribuir, se fusiona nodo 2 y 6 liberando el nodo 6 y decrementando la altura del árbol liberando el nodo 7.



Fundamentos de Organización de Datos

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.

Tipos de Dispersión

Direccionamiento estático

El espacio disponible para dispersar los registros del archivo está fijado previamente.

Direccionamiento dinámico

El espacio disponible para dispersar los registros del archivo aumenta o disminuye en función de las necesidades.

Parámetros a considerar

Parámetros que influyen sobre el desempeño del ambiente de dispersión:

- ✓ Capacidad de almacenamiento de cada dirección
- ✓ Densidad de empaquetamiento
- ✓ Función de hash
- ✓ Método de tratamiento de desbordes

Dispersión de Archivos

Función de dispersión

Caja negra que a partir de una clave genera la dirección física donde debe almacenarse el registro.

Colisión

Situación en la que un registro es asignado, por función de dispersión, a una dirección que ya posee uno o más registros.

Dispersión de Archivos

Desborde

Situación en la cual una clave carece de lugar en la dirección asignada por la función de dispersión.

Densidad de empaquetamiento

Relación entre el espacio disponible para el archivo de datos y la cantidad de registros que integran el mismo.

$$DE = \text{número de registros} / \text{espacio Total}$$

Dispersión de Archivos

Aunque la función de dispersión sea eficiente y la densidad de empaquetamiento sea baja, es probable que ocurran **desbordes**.

Métodos aplicables para resolver colisiones con desborde en *dispersión estática*:

- ✓ *Saturación progresiva*
- ✓ *Saturación progresiva encadenada*
- ✓ *Saturación progresiva con área de desborde por separado*
- ✓ *Dispersión doble*

No vemos ejercicios prácticos de hashing estático



Hashing Extensible

Técnica de resoluciones: Hashing Extensible

Ejemplo:

- Función de dispersión: Retorna 10 bits.
- Capacidad para 2 registros por dirección.
- Se van a dispersar 8 claves en total.

Hashing Extensible

Claves a dispersar

1- Colapinto (1011001100)

2- Verstappen (1110101000)

3- Russell (1010001001)

4- Stroll (1010101010)

5- Alonso (1010001000)

6- Hamilton (1001001011)

7- Sainz (1010001111)

8- Leclerc (1010100111)

Hashing Extensible

Estado inicial del archivo:

Tabla de dispersión Bits de dispersión: 0	
Sufijo	#Bloque
(0)	0

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	0		

Que el número de bits de dispersión esté en 0, indica que no es necesario ningún bit de la secuencia obtenida por la función de dispersión.

Hashing Extensible

Se agregan Colapinto (1011001100) y Verstappen (1110101000)

Tabla de dispersión Bits de dispersión: 0	
Sufijo	#Bloque
(0)	0

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	0	Colapinto (1011001100)	Verstappen (1110101000)

Ambos se agregan sin inconvenientes en el bloque 0.

Hashing Extensible

Se agrega Russell (1010001001) → produce desborde

Tabla de dispersión Bits de dispersión: 1	
Sufijo	#Bloque
(0)	1
(1)	0

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	1	Russell (1010001001)	
1	1	Colapinto (1011001100)	Verstappen (1110101000)

Se produce desborde en el bloque 0. Aumentamos en uno los bits de dispersión locales del bloque 0 y creamos un nuevo bloque (bloque 1) con la misma cantidad de bits locales. Luego comparamos los bits locales del bloque con los bits de dispersión globales de la tabla: como ahora el bloque tiene más bits locales que los bits globales de la tabla, aumentamos en uno los bits globales de la tabla y duplicamos la cantidad de direcciones. Finalmente, las claves se redistribuyen entre el bloque original y el nuevo según el bit menos significativo correspondiente. La dirección donde ocurrió el desborde ahora apunta al nuevo bloque.

Hashing Extensible

Se agrega Stroll (1010101010) → produce desborde

Tabla de dispersión Bits de dispersión: 2	
Sufijos	#Bloque
(00)	2
(01)	0
(10)	1
(11)	0

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	1	Russell (1010001001)	
1	2	Stroll (1010101010)	
2	2	Colapinto (1011001100)	Verstappen (1110101000)

Se produce desborde en el bloque 1. Aumentamos en uno los bits de dispersión locales del bloque 1 y creamos el bloque 2 con la misma cantidad de bits locales. Como los bits locales superan a los bits globales de la tabla, se aumenta en uno los bits de dispersión de la tabla y se duplican la cantidad de direcciones. Finalmente, las claves se redistribuyen entre el bloque original y el nuevo según los dos bits menos significativos. La dirección donde ocurrió el desborde ahora apunta al nuevo bloque.

Hashing Extensible

Se agrega Alonso (1010001000) → produce desborde

Se agrega Hamilton (1001001011) → entra normal

Se agrega Sainz (1010001111) → produce desborde

Se agrega Leclerc (1010100111) → produce desborde

Tabla de dispersión Bits de dispersión: 3	
Sufijo	#Bloque
(000)	3
(001)	0
(010)	1
(011)	4
(100)	2
(101)	0
(110)	1
(111)	5

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	2	Russell (1010001001)	
1	2	Stroll (1010101010)	
2	3	Colapinto (1011001100)	
3	3	Alonso (1010001000)	Verstappen (1110101000)
4	3	Hamilton (1001001011)	
5	3	Leclerc (1010100111)	Sainz (1010001111)

Hashing Extensible

Doy de baja a Verstappen (1110101000): se borra normal

Tabla de dispersión Bits de dispersión: 3	
Sufijo	#Bloque
(000)	3
(001)	0
(010)	1
(011)	4
(100)	2
(101)	0
(110)	1
(111)	5

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	2	Russell (1010001001)	
1	2	Stroll (1010101010)	
2	3	Colapinto (1011001100)	
3	3	Alonso (1010001000)	Verstappen (1110101000)
4	3	Hamilton (1001001011)	
5	3	Leclerc (1010100111)	Sainz (1010001111)

Al dar de baja a Alonso (1010001000), **el bloque 3 queda vacío**. Este bloque tenía nivel de dispersión local igual al global (3). Se identifica su bloque hermano: aquel que comparte el mismo sufijo salvo en el bit más significativo utilizado para el direccionamiento, en este caso el bloque 2 (sufijo 100). Como el bloque hermano también tiene un nivel de dispersión local de 3 y contiene claves válidas, **se libera el bloque 3** y su rango es absorbido por el bloque hermano. Luego, se reduce en uno el nivel de dispersión local del bloque resultante y se actualizan las entradas de la tabla que apuntaban al bloque eliminado para que redirijan al bloque que permanece.

Tabla de dispersión Bits de dispersión: 3	
Sufijo	#Bloque
(000)	2
(001)	0
(010)	1
(011)	4
(100)	2
(101)	0
(110)	1
(111)	5

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	2	Russell (1010001001)	
1	2	Stroll (1010101010)	
2	2	Colapinto (1011001100)	
3	3	Alonso (1010001000)	
4	3	Hamilton (1001001011)	
5	3	Leclerc (1010100111)	Sainz (1010001111)

Al eliminar Stroll (1010101010), el bloque 1 queda vacío. El mismo tenía 2 bits de dispersión local y estaba referenciado por las direcciones de la tabla que terminan en 10. Para saber si se puede liberar el bloque 1, se evalúa si sus direcciones hermanas apuntan al mismo bloque. Son hermanas aquellas que terminan en 00, ya que si se redujera la dispersión local compartirían el mismo sufijo. **Las entradas terminadas en 00 apuntan al mismo bloque (2)**, entonces se puede liberar el bloque 1: se sustituyen las referencias al bloque 1 por referencias al bloque 2, se reduce en uno la dispersión local, y el bloque 1 puede eliminarse. Se mantiene la integridad del direccionamiento y se optimiza el uso de bloques.

Tabla de dispersión Bits de dispersión: 3	
Sufijo	#Bloque
(000)	2
(001)	0
(010)	2
(011)	4
(100)	2
(101)	0
(110)	2
(111)	5

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	2	Russell (1010001001)	
1	2	Stroll (1010101010)	
2	1	Colapinto (1011001100)	
3	3	Alonso (1010001000)	
4	3	Hamilton (1001001011)	
5	3	Leclerc (1010100111)	Sainz (1010001111)

Al eliminar Russell (1010001001), el bloque 0 queda vacío. El mismo tenía 2 bits de dispersión local y estaba referenciado por las direcciones de la tabla que terminan en 01. Para saber si se puede liberar el bloque 0, se evalúa si sus direcciones hermanas apuntan todas al mismo bloque. Se consideran hermanas aquellas que terminan en 11, ya que si se redujera la dispersión local compartirían el mismo sufijo. **Las entradas terminadas en 11 no apuntan al mismo bloque** (011 apunta al 4 y 111 apunta al 5), por ende no se puede liberar el bloque 0. **Se escribe el bloque 0 vacío.**

Tabla de dispersión Bits de dispersión: 3	
Sufijo	#Bloque
(000)	2
(001)	0
(010)	2
(011)	4
(100)	2
(101)	0
(110)	2
(111)	5

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	2	Russell (1010001001)	
1	2	Stroll (1010101010)	
2	1	Colapinto (1011001100)	
3	3	Alonso (1010001000)	
4	3	Hamilton (1001001011)	
5	3	Leclerc (1010100111)	Sainz (1010001111)

Hashing Extensible

Estado final:

Tabla de dispersión Bits de dispersión: 3	
Sufijo	#Bloque
(000)	2
(001)	0
(010)	2
(011)	4
(100)	2
(101)	0
(110)	2
(111)	5

Archivo de datos			
#Bloque	Bits	Clave R1	Clave R2
0	2		
1	2	Stroll (1010101010)	
2	1	Colapinto (1011001100)	
3	3	Alonso (1010001000)	
4	3	Hamilton (1001001011)	
5	3	Leclerc (1010100111)	Sainz (1010001111)