

# Estructuras de Datos

## Clase 22 – Ordenamiento externo



Dr. Sergio A. Gómez  
<http://cs.uns.edu.ar/~sag>



Departamento de Ciencias e Ingeniería de la Computación  
Universidad Nacional del Sur  
Bahía Blanca, Argentina

# Temario

- Serialización
- Archivos:
  - Secuenciales
  - Directos (de acceso aleatorio)
  - indexados (indizados)
- Estructuras aptas para implementar archivos indexados: Árboles B

# Persistencia con Serialización

- La serialización implica salvar el estado actual de un objeto en un stream y luego poder recuperar el objeto equivalente de tal stream.
- Para poder hacer esto las clases deben implementar la interfaz `java.io.Serializable` (que no tiene métodos)

```
import java.io.Serializable;

public class NombreClase implements Serializable {

}
```

# Ejercicio

- Crear una lista de enteros conteniendo [1,2,3,4,5] y serializarla en un archivo llamado lista.dat.
- Modificaciones requeridas a las implementaciones previas:

```
public class DNode <E> implements Position<E>,  
java.io.Serializable { ... }
```

```
public class DoubleLinkedList <E> implements PositionList<E>,  
java.io.Serializable { ... }
```

```

import java.io.*;
public class SerializarLista {
public static void main( String [] args ) throws FileNotFoundException, IOException {
    PositionList<Integer> lista = new DoubleLinkedList<Integer>();
    lista.addLast( 1 ); lista.addLast( 2 ); lista.addLast( 3 ); lista.addLast( 4 );
    lista.addLast( 5 );

    // Creo un objeto de tipo archivo para almacenar la lista:
    File fileName = new File( "lista.dat" ); //FileNotFoundException

    // Abrir un manejador de archivo para solo escritura:
    ObjectOutputStream output =
        new ObjectOutputStream( new FileOutputStream( fileName ) );

    // Escribir la lista en el stream de objetos output (IOException):
    output.writeObject( lista );
    // Flush fuerza la escritura de cualquier contenido que
    // haya quedado en el buffer del archivo.
    output.flush();
    // Cierro el archivo.
    output.close();
}
}

```

# Ejercicio

- Recuperar la lista de enteros serializada en el archivo lista.dat e imprimir su contenido

```

import java.io.*;
public class RecuperarLista {
public static void main( String [] args )
    throws FileNotFoundException, IOException, ClassNotFoundException,
    InvalidPositionException, BoundaryViolationException {
    DoubleLinkedList<Integer> lista; // Este objeto va a almacenar un objeto leido del archivo
    // Puede producir FileNotFoundException:
    File fileName = new File( "lista.dat" ); // Nombre del archivo
    // Abrir un file handle para solo lectura:
    ObjectInputStream input = new ObjectInputStream( new FileInputStream(fileName) );
    // Leo la única lista que está en el archivo:
    // Puede producir IOException o ClassNotFoundException:
    // Compilar con java RecuperarLista.java -Xlint:unchecked, Produce warning.
    lista = (DoubleLinkedList<Integer>) input.readObject();
    input.close(); // Cierro el archivo:

    // Imprimo la lista:
    Position<Integer> p = lista.first(), ultima = lista.last(); System.out.print( "[" );
    while( p != null ) { System.out.print( p.element() );
        if( p == ultima ) { System.out.print( "]" ); p = null; }
        else { System.out.print( ", " ); p = lista.next(p); }
    }
} /* end main */ } /* end class */

```

# Archivos de registros secuenciales

- La tabla representa una secuencia de registros (records) almacenados en disco con el nombre *agenda.dat*.
- Cada registro (o fila) almacena dos datos (nombre y número de teléfono).
- Los registros se acceden secuencialmente (del primero al último).

Name	Phone
Sergio	15-111-1111
Martin	15-222-2222
Matias	15-333-3333
Carlos	15-444-4444
Marta	15-555-5555
Pablo	15-666-6666
Tito	15-777-7777
Maria	15-888-8888
Juana	15-999-9999

# Clase Registro

```
import java.io.Serializable;

public class Record implements Serializable {

    private String name;
    private String phone;

    public Record(String n, String p) {
        name = n;
        phone = p;
    }

    public Record() { this ("", ""); }

    public void setName(String n) { name = n; }
    public void setPhone(String p) { phone = p; }
    public String getName() { return name; }
    public String getPhone() { return phone; }
}
```

# Creación de la agenda

```
import java.io.*;

public class GenerarArchivoSecuencial {
    public static void main(String [] args) {
        Record r1 = new Record( "Sergio", "15-111-1111" );
        Record r2 = new Record( "Martin", "15-222-2222" );
        Record r3 = new Record( "Matias", "15-333-3333" );
        Record r4 = new Record( "Carlos", "15-444-4444" );
        Record r5 = new Record( "Marta", "15-555-5555" );
        Record r6 = new Record( "Pablo", "15-666-6666" );
        Record r7 = new Record( "Tito", "15-777-7777" );
        Record r8 = new Record( "Maria", "15-888-8888" );
        Record r9 = new Record( "Juana", "15-999-9999" );
    }
}
```

```

try {
    File fileName = new File( "agenda.dat" );

    //Abrir un manejador de archivo para solo escritura:
    ObjectOutputStream output =
        new ObjectOutputStream( new FileOutputStream( fileName ) );

    // Escribir los nueve registros en el archivo output:
    // Puede lanzar excepciones InvalidClassException o NotSerializableException
    output.writeObject( r1 );      output.writeObject( r2 );      output.writeObject( r3 );
    output.writeObject( r4 );      output.writeObject( r5 );      output.writeObject( r6 );
    output.writeObject( r7 );      output.writeObject( r8 );      output.writeObject( r9 );
    output.writeObject( null ); // Grabo marca personal de fin de archivo.

    // Flush fuerza la escritura de cualquier contenido que haya quedado en el buffer del
    // archivo.
    output.flush();
    output.close(); /* Cierro el archivo. */
} catch (InvalidClassException icex) { System.out.println("Clase invalida");
} catch (NotSerializableException nsex) { System.out.println("El objeto no es
serializable");
} catch (IOException e) { System.out.println("Problema al hacer flush o cerrar"); }
}
}

```

# Lectura del archivo

```
import java.io.*;
public class LeerArchivoSecuencial {
public static void main( String [] args ) {
Record r; // Este objeto va a almacenar un objeto leido del archivo
File fileName = new File( "agenda.dat" ); // Nombre del archivo

// Abrir un file handle para solo lectura:
ObjectInputStream input = new ObjectInputStream( new FileInputStream(fileName) );
r = (Record) input.readObject();
while ( r != null ) {
    System.out.println( r.getName() + "; " + r.getPhone() );
    r = (Record) input.readObject();
}
// Cierro el archivo:
input.close();
} catch (EOFException eofex) { System.out.println( "No hay mas registros para leer" );
} catch (ClassNotFoundException cnfex) { System.out.println( "No pude crear el objeto" );
} catch (IOException e ) { System.out.println( "Incapaz de cerrar archivo" ); }
}
}
```

# Archivos de acceso directo

- Los archivos de acceso directo (*RandomAccessFile*) permiten:
  - leer los registros de un archivo en cualquier orden
  - Modificar un registro cualquiera sin necesidad de reescribir todo el archivo
  - Agregar un registro nuevo al final del archivo
- Tendremos una operación más que se llama *seek* que recibe el número de byte (llamado *offset* o desplazamiento) al que hay que mover el puntero de archivo para realizar la próxima operación de lectura o escritura.
- Para utilizar *seek* es necesario conocer el tamaño del registro del archivo subyacente. Ej. Para ir al registro número  $R$ , el *offset* debe ser  $R*S$ , donde  $S$  es el tamaño en bytes del registro y ejecutar: `archivo.seek(R*S)`.

# Archivos Indizados (o indexados)

- Los archivos indexados tienen un orden virtual determinado por el ordenamiento de una clave (en nuestro ejemplo, el nombre de la persona, asumiendo no repetidos).
- Implementación: Por cada tabla, hay al menos dos archivos:
  - Un archivo de datos donde cada registro tiene tamaño fijo
  - Un archivo de índice: Un mapeo de clave en número de registro (contiene además la cantidad de registros del archivo de datos)
  - Cada clave extra de búsqueda requiere otro índice (pero no otro archivo de datos, ej: buscar/ordenar alumnos por legajo, dni, nombre, etc).
- Ver código EjemploArchivoIndizado.zip.

# Implementaciones para el índice

- Arreglo ordenado
  - Búsqueda en orden logarítmico
  - Actualizaciones implican reordenar el arreglo en orden  $n \log(n)$
- Árbol binario de búsqueda:
  - Puede degenerar en orden lineal para buscar y actualizar pero se espera orden logarítmico
- AVL, 2-3:
  - Buscar y actualizar en orden logarítmico
- Árbol B y B+
  - Para grandes volúmenes de datos las opciones anteriores no son viables ya que no entran en RAM y el acceso al disco es del orden de los milisegundos contra el acceso a la RAM que es del orden de los nanosegundos
  - El árbol B generaliza la idea del árbol 2-3 con un gran número  $M$  de claves en cada nodo
  - El árbol B+ almacena entradas (clave,valor) sólo en las hojas, los nodos internos almacenan solo claves que sirven para guiar la búsqueda.

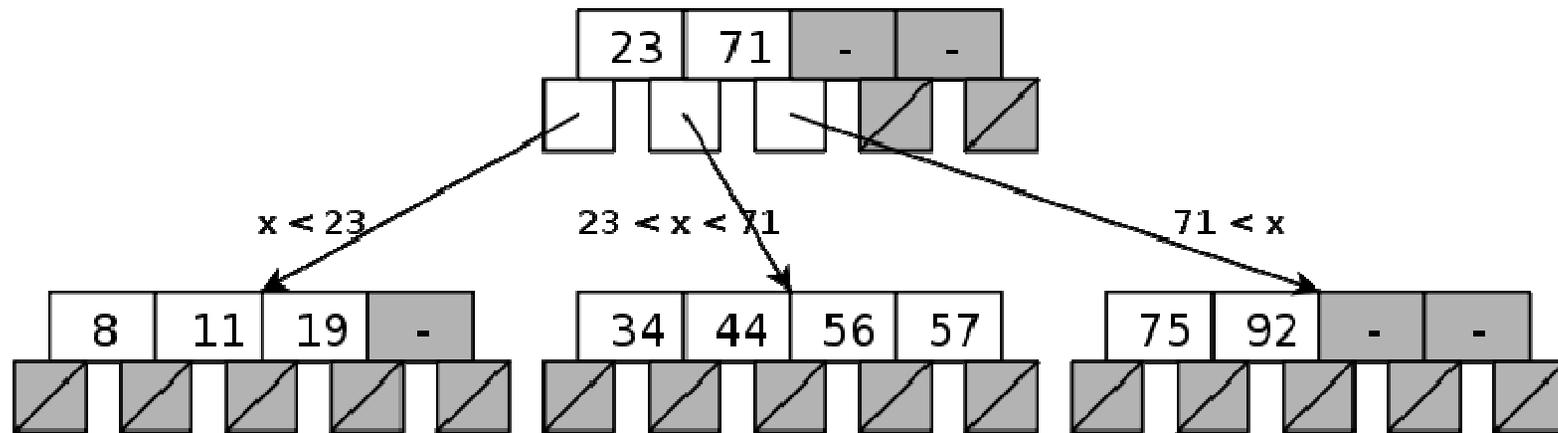
# Árbol B

- Un árbol-B (B-tree) es un árbol balanceado (o auto-balanceante) que mantiene los datos ordenados y permite realizar búsquedas, inserciones y eliminaciones en tiempo logarítmico. Se puede ver también como una generalización del árbol 2-3 porque sus nodos pueden tener más de tres hijos.
- A diferencia de los árboles 2-3, el árbol-B está optimizado para sistemas que leen y escriben grandes bloques de datos.
- Los árboles-B son un buen ejemplo de una estructura de datos para memoria externa y se usan comúnmente en bases de datos y sistemas de archivos.

# Arboles B (o M-arios de búsqueda)

- Los árboles B se optimizan para manejar grandes volúmenes de datos.
- Los árboles B se almacenan en disco y el tamaño del nodo coincide con un múltiplo entero del tamaño del sector del disco.
- Se utilizan para implementar índices en bases de datos.
- El grado de ramificación  $d$  del árbol es un entero, indicando que un nodo tiene entre  $d$  y  $2d$  claves y entre  $d+1$  y  $2d+1$  hijos (excepto la raíz que puede tener menos de  $d$  claves y tiene por lo menos 2 hijos).
- Cuando  $d=1$  tenemos un árbol 2-3.
- Dependiendo de la formalización, al número  $2d+1$  se lo llama  $M$  (i.e. tenemos  $M-1$  claves y  $M$  hijos a lo sumo por nodo; en un árbol 2-3,  $M$  vale 3)

# Árbol B



El tiempo de búsqueda, inserción y eliminación es en tiempo logarítmico puesto que es del orden de la altura del árbol.

Si  $d=1000$ , cada nodo tiene  $d$  claves y  $2d$  hijos por lo menos, entonces la altura del árbol es del orden  $\log_d(n)$  con  $n$  = cantidad de claves del árbol. Ej: si  $d=1.000$  y  $n=1.000.000$ , entonces  $\log_d(1.000.000)=3$  lecturas.

Generalmente hay espacio desperdiciado en los nodos (i.e. fragmentación interna).

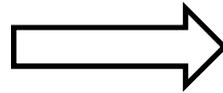
# Árboles B: Inserción

- Comenzamos con un nodo vacío (el cual funciona como un arreglo ordenado).
- Las claves se insertan (en forma lineal y ordenada) en el nodo hasta tener a lo sumo  $2d$  claves.
- Luego, al insertar la siguiente clave, el nodo rebalsa.
- El nodo se parte en dos (cada uno con  $d$  claves) y la clave del medio va al nodo de arriba (incrementando la altura del árbol cuando tal nodo de arriba no existiera).

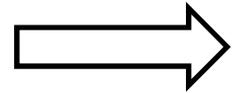
Supongamos que  $d=3$ , entonces tendremos entre  $d=3$  y  $2d=6$  claves por nodo excepto la raíz que puede tener menos claves (es decir,  $M=2d+1=7$ , el B-árbol es de orden 7) y siempre el número de hijos de un nodo es uno más que su cantidad de claves.

Insertemos las claves 1, 40, 5, 3, 45, 8, 9.

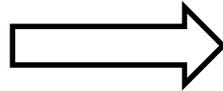
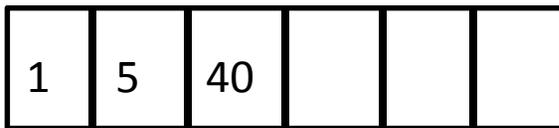
**Inserto 1:**



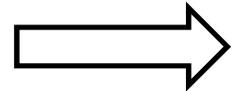
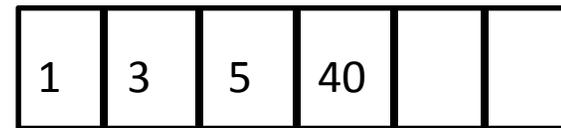
**Inserto 40:**



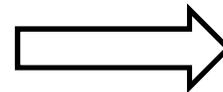
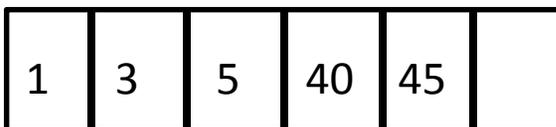
**Inserto 5 (hago shift):**



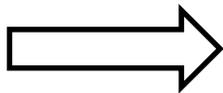
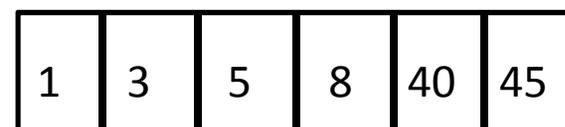
**Inserto 3 (hago shift):**



**Inserto 45:**

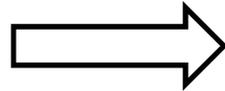
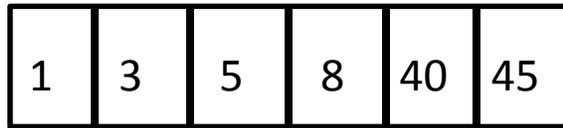


**Inserto 8:**

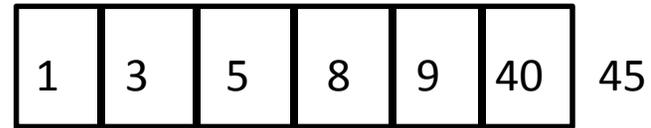


La próxima inserción, es decir la del 9, producirá un rebalse porque el nodo ya está lleno.

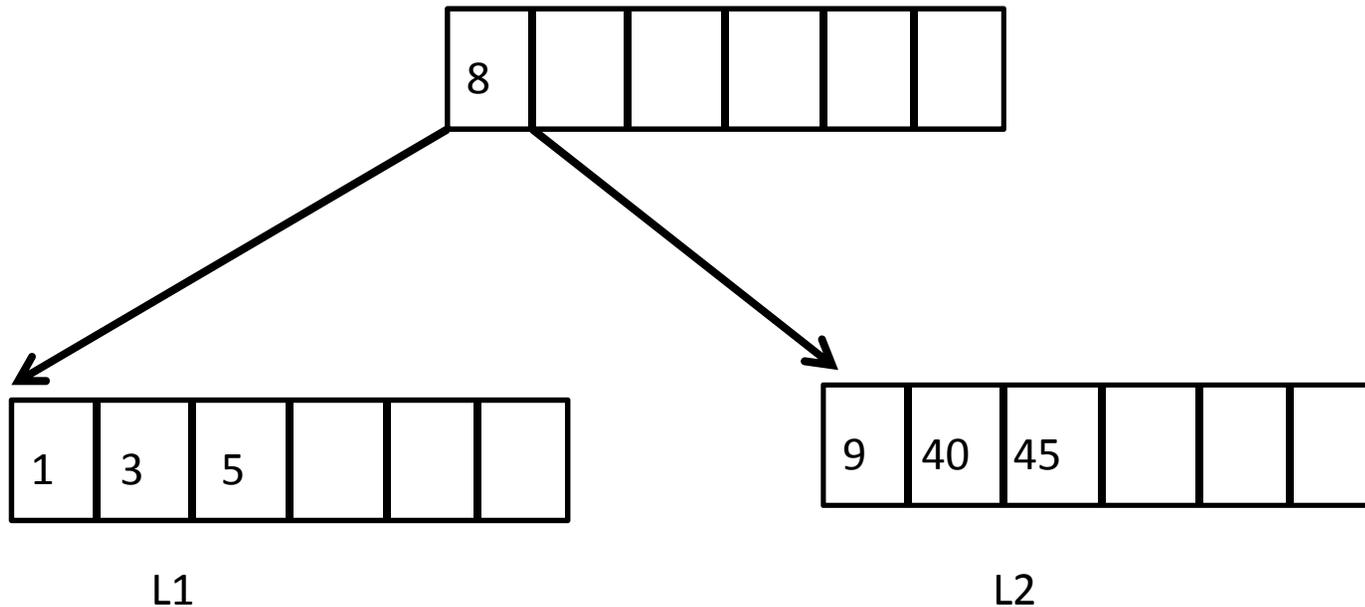
**Estado antes de insertar 9:**



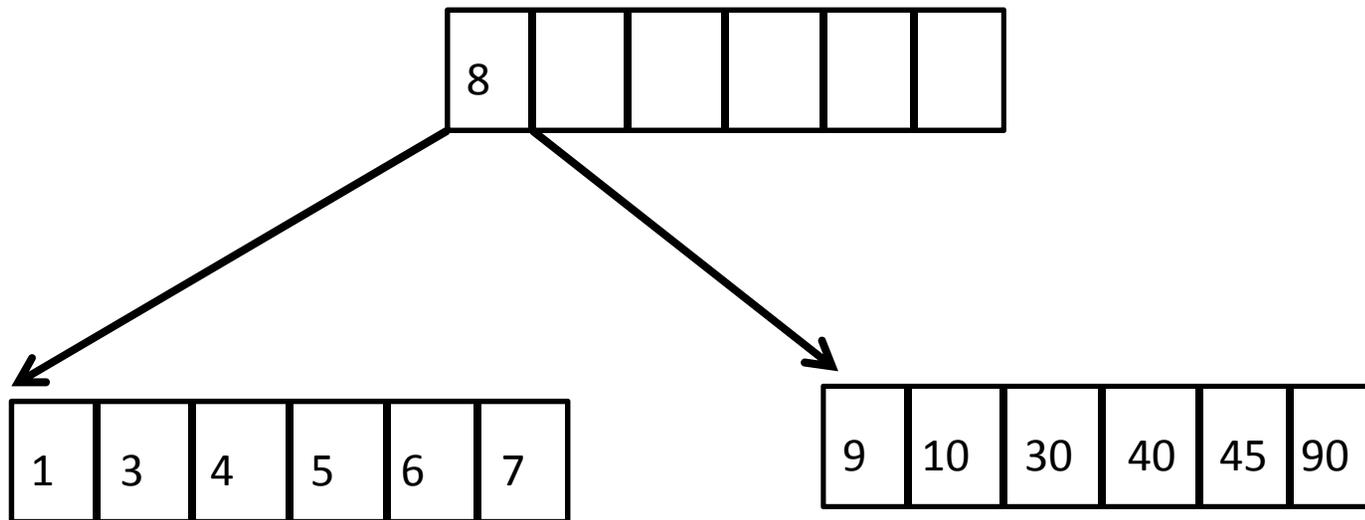
**Al insertar 9, se produce un rebalse!**



Entonces partimos el nodo L en L1 y L2 y creamos un nodo más arriba con L1 y L2 como hijos y como clave la clave del medio de L considerando también al 9, que en este caso será el 8:

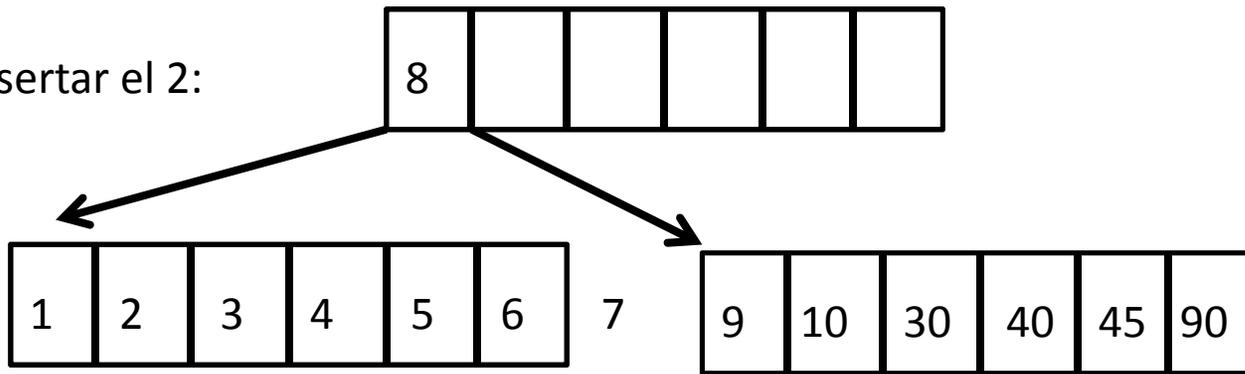


Las inserciones se hacen siempre en las hojas siguiendo el orden de las claves.  
Para insertar una clave, habrá que ver si es menor o mayor a la clave de la raíz.  
Insertar 10, 4, 30, 7, 6, 90 produce el árbol:

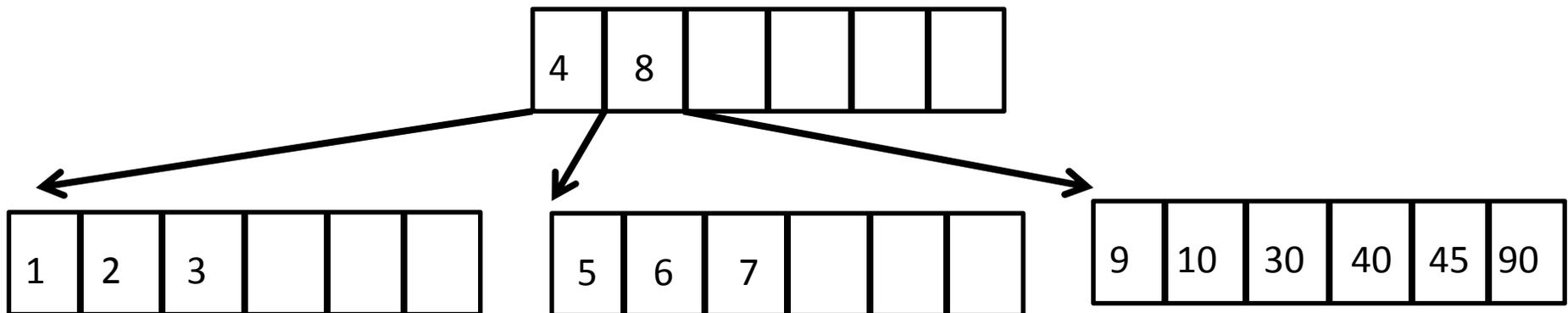


¡Insertar el 2 producirá un rebalse en la hoja de la izquierda!

Estado al insertar el 2:



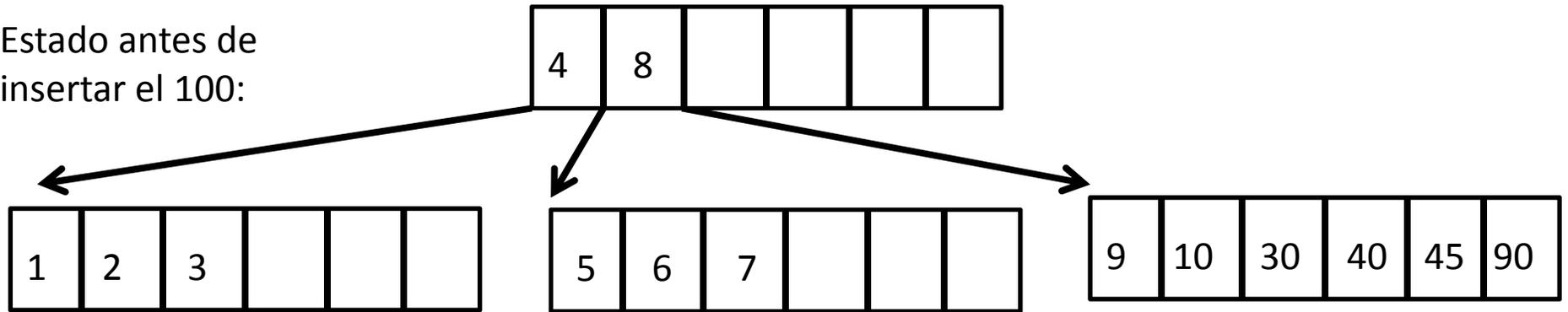
Rebalsó el 7, vamos a partir nodo rebalsado en 2 y mandar la clave del medio (es este caso 4) al padre:



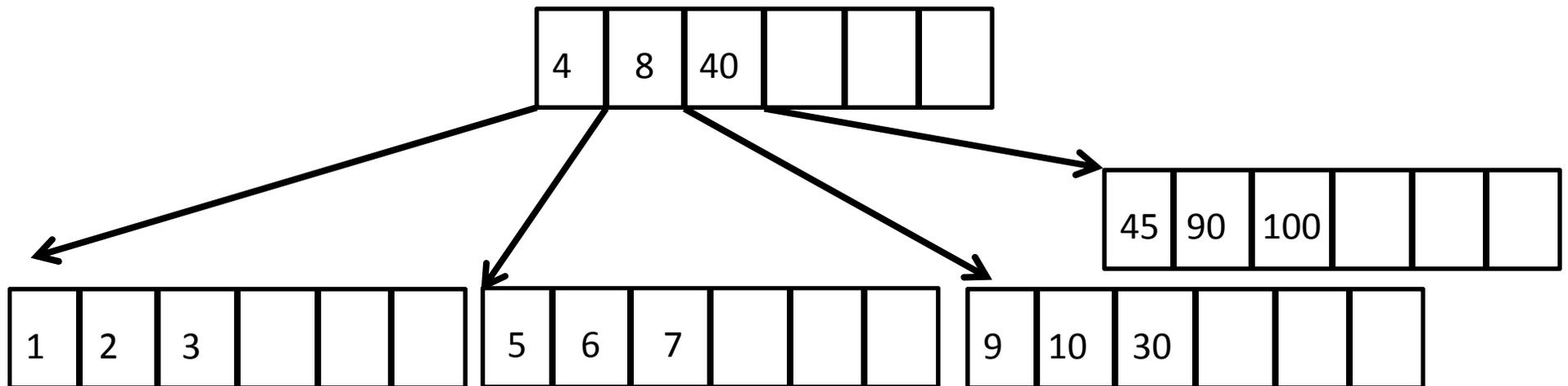
Insertar 100 producirá un rebalse en la hoja de más a la derecha.

El espacio desperdiciado en los arreglos constituye la *fragmentación interna* (*espacio de memoria desperdiciado dentro de las estructuras de datos*).

Estado antes de insertar el 100:



Al insertar el 100, como  $100 > 8$ , va hacia la hoja de más a la derecha, que rebalsa. Entonces la partimos y subimos la clave del medio (en este caso el 40):

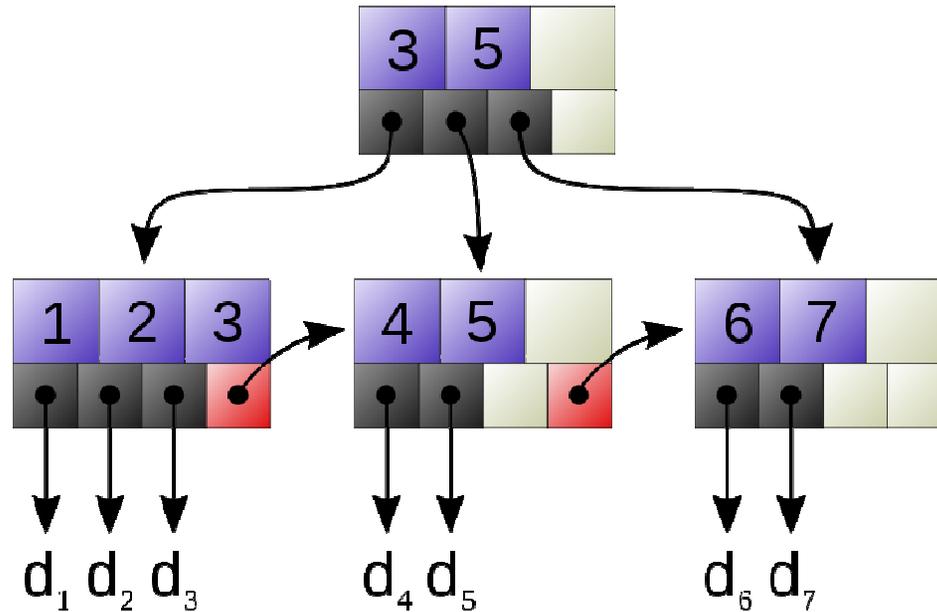


Este proceso continúa hasta que los rebales sucesivos en las hojas van llenando la raíz del árbol. En el siguiente rebalse, se partirá la raíz en 2 y se crea una nueva raíz haciendo crecer el árbol en un nivel.

# Árboles B: Búsqueda de k

- Buscamos a k en el nodo raíz. Si lo encontramos, termino con éxito.
- Si k no está en la raíz, seguimos buscando en el hijo entre las dos claves  $k_i$  y  $k_{i+1}$  del nodo tales que  $k_i \leq k \leq k_{i+1}$ .
- El proceso falla cuando llegamos a una hoja y no encontramos a k.
- Ej: Para buscar el 6, hay que ramificar en el hijo que está entre las claves 4 y 8.

# Variante del árbol B: Árbol B+



El árbol B+ almacena entradas (clave,valor) sólo en las hojas, los nodos internos almacenan solo claves que sirven para guiar la búsqueda.

Ejemplo: 3 en la raíz indica la mayor clave del primer hijo del siguiente nivel.

5 en la raíz indica la mayor clave del segundo hijo del siguiente nivel.

$d_1, d_2, d_3, d_5, d_6, d_7$  corresponden a los valores de las entradas para las claves 1,2,3,4,5,6,7 (usualmente punteros al archivo de datos).