Estructuras de datos y algoritmos

6. Implementación de mapas, árboles y grafos

- 6.1. Mapas y conjuntos implementados con tablas de troceado
- 6.2. Árboles
- 6.3. Árboles binarios
- 6.4. Árboles binarios equilibrados y conjuntos ordenados
- 6.5. *B*-árboles
- 6.6. Colas de prioridad
- 6.7. Grafos

6.1. Mapas y conjuntos implementados con tablas de troceado

Las tablas de troceado (o tablas *hash*) permiten realizar operaciones de búsqueda muy eficientes en promedio

- permiten consultar o eliminar elementos conociendo su nombre o identificador
- el tiempo es constante en la mayor parte de los casos: O(1)
- son muy apropiadas por tanto para los mapas, o diccionarios

El principio de funcionamiento es almacenar los datos en una tabla

- se convierte el nombre o identificador a un número entero, que sirve como índice de la tabla
- para que la tabla no sea excesivamente grande, el número entero, llamado *clave*, se compacta a un rango pequeño
  - habitualmente mediante la operación módulo (%)

Implementación mediante tablas hash

La clase del elemento debe disponer de un método `hashCode()` para convertir un objeto de esa clase en un dato numérico llamado *clave*

```
public int hashCode()
```

Los datos se guardan en un array cuyo índice es el tipo `clave`

El principal problema es la resolución de colisiones

- cuando dos datos tienen la misma clave

La función `hashCode` debe distribuir las claves de modo muy homogéneo

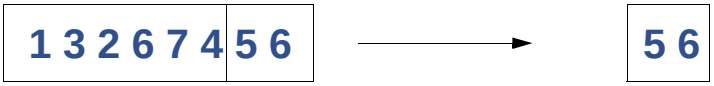
Algunas funciones de troceado para números enteros

Queremos trocear el identificador `id`

Primer método: función módulo

- `id % max`, donde `max` es el tamaño de la tabla
- el inconveniente es que sólo se usa la parte menos significativa del identificador
- si hay muchos identificadores con la misma parte menos significativa, no se obtienen resultados homogéneos

Ejemplo: `id`=13267456, `max`=100



Algunas funciones de troceado para números enteros

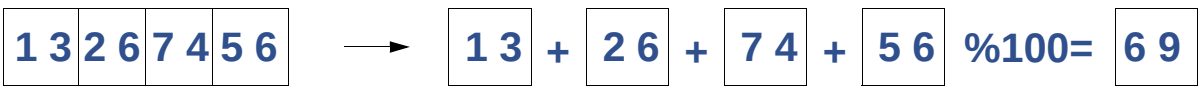
Segundo método: troceado en `n` grupos de cifras

$$Clave = \left(\sum_{i=0}^{n-1} \left(\frac{id}{max^i} \bmod max \right) \right) \bmod max$$

La ventaja es que se utiliza todo el número

- el resultado puede (o no) ser más homogéneo

Ejemplo: `id`=13267456, `max`=100, `n`=4



Algunas funciones de troceado para textos

Para textos occidentales, se usa el hecho de que los caracteres se pueden representar mediante un número pequeño de 7 u 8 bits

Por ejemplo, el texto "hola" es una secuencia de {'h','o','l','a'}

- se podría representar mediante su conversión a cifras de 7 bits

$$clave='h'*128^3+'o'*128^2+'l'*128^1+'a'*128^0$$

- Pero para cadenas largas, este cálculo excedería el valor que se puede guardar en un entero

Algunas funciones de troceado para textos (cont.)

Usaremos la siguiente propiedad de los polinomios

$$A_3X^3 + A_2X^2 + A_1X^1 + A_0X^0$$

se puede evaluar también como

$$(((A_3)X + A_2)X + A_1)X + A_0$$

En la segunda ecuación

- se evita el cálculo de x^i
- se evita un resultado intermedio demasiado grande
- el cálculo se hace de izquierda a derecha, en el orden del string

Algunas funciones de troceado para textos (cont.)

En el cálculo del polinomio anterior, podemos hacer la operación módulo con el tamaño de la tabla a cada paso

- **Así conseguiremos resultados intermedios pequeños**

Si `Cod(i)` es el código numérico del carácter `i`:, la función quedará como

$$Clave_i = (Clave_{i-1} \cdot 128 + Cod(i)) \bmod max$$

partiendo de un valor inicial `Clave=0`

Algunas funciones de troceado para textos (cont.)

Otra alternativa más eficiente es tener en cuenta que las operaciones con enteros admiten en muchos lenguajes (incluido Java) el desbordamiento, haciendo una operación módulo automática con el número de valores que caben en un entero

- **Ello permite hacer una sola operación módulo al final**
- **También podemos usar un número menor para multiplicar la clave, dado que el número de letras distintas suele ser pequeño**

$$Clave_i = Clave_{i-1} \cdot 37 + Cod(i)$$

$$Clave = Clave_n \bmod max$$

- **En este caso, hay que tener en cuenta que el resultado puede ser negativo, para ponerlo en positivo, sumarle `max`**

Ejemplo de función de troceado para strings

```
public static int hashCode(String s, int max) {
    int clave=0;
    for (int i=0; i<s.length(); i++) {
        clave=(clave*37+ s.charAt(i));
    }
    clave = clave % max;
    if (clave<0) {
        clave=clave+max;
    }
    return clave;
}
```


Métodos para la resolución de colisiones

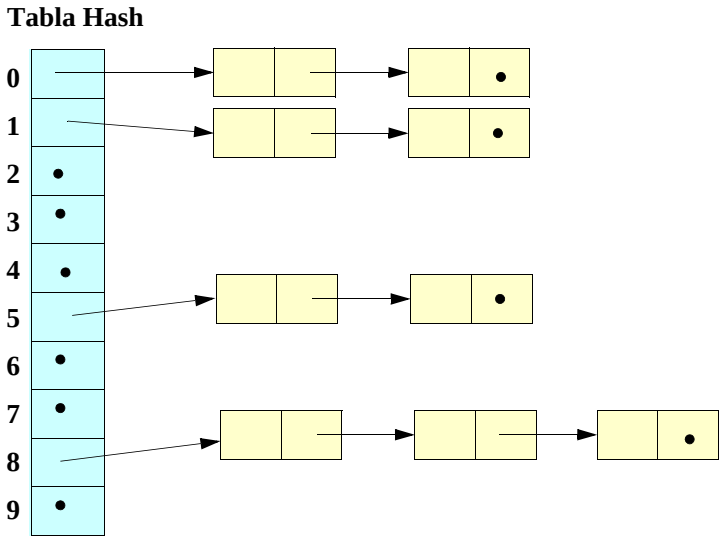
Los tres métodos más simples son:

- encadenamiento o troceado abierto
- por exploración, o troceado cerrado
  - exploración lineal
  - exploración cuadrática
  - otras exploraciones

Su eficiencia es distinta, y depende del grado de ocupación de la tabla

- troceado *abierto*: se puede funcionar bien con una ocupación del 100%, pero no más
- troceado *cerrado*: conviene nunca superar el 75% de ocupación

Resolución de colisiones: troceado abierto



Troceado abierto (cont.)

Cada elemento de la tabla hash es una lista enlazada simple.

- **Cada elemento del mapa se guarda en la lista correspondiente a su clave**

Este método sólo es eficiente cuando la cantidad de colisiones es pequeña, y las listas son cortas

- **Para ello se necesita que la tabla tenga un tamaño mayor o igual que el número de elementos a almacenar**
- **y que, además, la función hash sea muy homogénea**

Operaciones para el troceado abierto

UNIVERSIDAD DE CANTABRIA

***asignaValor*:**

- Calcular la clave de troceado
- Buscar el dato origen en la lista de esa clave
- Si se encuentra, cambiar el destino
- Si no, añadir la pareja (origen-destino) a la lista, por ejemplo al principio (ya que es más eficiente en una lista enlazada simple)

***borra*:**

- Calcular la clave de troceado
- Buscar el dato origen en la lista de esa clave
- Si se encuentra, eliminar el elemento de la lista
- Si no se encuentra, indicar que no existe

Operaciones para el troceado abierto (cont.)

UNIVERSIDAD DE CANTABRIA

***obtenValor*:**

- Calcular la clave de troceado
- Buscar el dato origen en la lista de esa clave
- Si se encuentra, devolver el destino asociado
- Si no se encuentra, indicar que no existe

***hazNulo*:**

- Hacer nulas todas las listas de la tabla

Implementación de un mapa mediante troceado abierto

Por simplicidad, no utilizaremos una extensión de `AbstractMap`

- nos obligaría a implementar `EntrySet`, que es complicado

Haremos una implementación de los métodos básicos

- constructor
- `put`
- `get`
- `remove`
- `containsKey`

Implementación de un mapa mediante troceado abierto

```
/**
 * Clase que representa un mapa con troceado abierto
 */
public class MapaAbierto<K,V>
{
  // atributos privados
  private ListaEnlazadaSimple<Entry<K,V>>[] tabla;

  private static class Entry<K,V> {
    K clave;
    V valor;
    public Entry(K clave, V valor) {
      this.clave=clave;
      this.valor=valor;
    }
```

Implementación de un mapa mediante troceado abierto (cont.)

```
    public boolean equals(Object otra) {
      if (otra instanceof Entry) {
        return this.clave.equals
           (((Entry<K,V>)otra).clave);
      } else {
        return false;
      }
    }
  }
```


Implementación de un mapa mediante troceado abierto (cont.)

```
/**
 * Constructor; se le pasa el tamaño de la tabla*/
public MapaAbierto(int tamano) {
  tabla =
    new ListaEnlazadaSimple[tamano];
  for (int i = 0; i < tamano; i++) {
    tabla[i] =
       new ListaEnlazadaSimple<Entry<K,V>>();
  }
}
// valor para la tabla hash
private int valorHash(K clave) {
  return clave.hashCode() % tabla.length;
}
```

Implementación de un mapa mediante troceado abierto (cont.)

```
public V get(K clave) {
  int hashVal = valorHash(clave);
  Entry<K,V> nuevo= new Entry<K,V>(clave,null);
  Entry<K,V> e=tabla[hashVal].find(nuevo);
  if (e==null) {
      return null;
  } else {
     return e.valor;
  }
}
```


Implementación de un mapa mediante troceado abierto (cont.)

```
public V put(K clave, V valor) {
  int hashVal = valorHash(clave);
  Entry<K,V> nuevo= new Entry<K,V>(clave,valor);
  Entry<K,V> e=tabla[hashVal].find(nuevo);
  if (e==null) {
    tabla[hashVal].addFirst(nuevo);
    return null;
  } else {
    V viejo=e.valor;
    e.valor=valor;
    return viejo;
  }
}
```

Implementación de un mapa mediante troceado abierto (cont.)

```
  public V remove(K clave) {
    int hashVal = valorHash(clave);
    Entry<K,V> nuevo= new Entry<K,V>(clave,null);
    Entry<K,V> e=tabla[hashVal].find(nuevo);
    if (e==null) {
        return null;
    } else {
       V valor=e.valor;
       tabla[hashVal].remove(nuevo);
       return valor;
    }
  }
```


Implementación de un mapa mediante troceado abierto (cont.)

```
  boolean containsKey(K clave) {
    int hashVal = valorHash(clave);
    Entry<K,V> nuevo= new Entry<K,V>(clave,null);
    Entry e=tabla[hashVal].find(nuevo);
    return e!=null;
  }

}
```

Resolución de conflictos por troceado cerrado: exploración lineal

Si se detecta una colisión, se intenta calcular una nueva clave, hasta que se encuentre una vacía

$$Clave_i(x) = (Hash(x) + f(i)) \bmod max$$

La tabla debe ser bastante mayor que el número de relaciones

- **recomendado al menos el 50% de las celdas vacías**

La función $f(i)$ es la estrategia de resolución:

- **lineal: $f(i)=i$**
- **cuadrática: $f(i)=i^2$ (el tamaño de la tabla debe ser primo)**
- **doble troceado: $f(i)=i*hash_2(x)$**
  - **$hash_2(x)$ nunca debe dar cero**
  - **ej.: R-(x mod R), siendo R primo**

Troceado cerrado (cont)

En este caso la tabla "hash" contiene las relaciones

- **parejas {clave,valor}**

Exploración lineal:

- **número de búsquedas en una inserción= $1/(1-\lambda)$, siendo λ la ocupación de la tabla**
- **si la ocupación es menor que la mitad, en promedio se necesitan 2 búsquedas para asignar o calcular**
- **Si la ocupación es mayor, el número de búsquedas puede crecer mucho**
- **Si la función hash no es homogénea la eficiencia puede ser peor**

Troceado cerrado (cont)

***Exploración cuadrática*: es mejor que el lineal si los elementos tienden a agruparse en celdas contiguas**

- **Es preciso que la ocupación sea menor que la mitad, para garantizar que siempre encontramos una celda libre**
- **El tamaño de la tabla debe ser un número primo para que nunca se busque la misma celda dos veces para una clave concreta**
- **el análisis matemático aún no se ha podido hacer, por su complejidad**

***Exploración por doble troceado*:**

- **Permite eliminar la agrupación secundaria que se produce en ocasiones con la exploración cuadrática**

Resolución de conflictos: troceado cerrado (cont)

Si borramos celdas, la resolución de conflictos falla

Debe usarse la técnica del borrado "*perezoso*"

- **marcamos la celda como borrada, pero la mantenemos ocupando memoria**
- **al buscar una relación, si encontramos una celda marcada como borrada continuamos la búsqueda**
- **al insertar una relación nueva, puede meterse en el lugar de una marcada como borrada**

A veces será necesario "*recrear*" la tabla

- **partir de una tabla vacía e insertar en ella todos los elementos útiles**

Implementación de un mapa mediante troceado cerrado

```
import java.util.*;
/**
 * Implementación de un mapa usando troceado cerrado
 */
public class MapaCerrado <K,V>
{
  /**
   * Clase Entry, que define una pareja clave-valor
   * y anota si se ha borrado o no
   */
  private static class Entry<K,V> {
    K clave;
    V valor;
    boolean borrado;
```


Implementación de un mapa mediante troceado cerrado (cont.)

```
    public Entry(K clave, V valor) {
      this.clave = clave;
      this.valor = valor;
      this.borrado=false;
    }

    public boolean equals(Object otra) {
      if (otra instanceof Entry) {
        return this.clave.equals
            (((Entry<K,V>)otra).clave);
      } else {
        return false;
      }
    }
  }
```

Implementación de un mapa mediante troceado cerrado (cont.)

```
// atributos privados de la clase MapaCerrado
private Object[] tabla;
private int num;

/**
 * Constructor al que se le indica el tamaño
 * inicial de la tabla
 */
public MapaCerrado(int tamanoInicial) {
   tabla = new Object[tamanoInicial];
   num=0;
}
```


Implementación de un mapa mediante troceado cerrado (cont.)

```
// función hash
private int valorHash(K clave) {
   return clave.hashCode() % tabla.length;
}
```

Implementación de un mapa mediante troceado cerrado (cont.)

```
// función para buscar una celda
private int buscaIndice(Entry<K,V> e) {
  int pos = valorHash(e.clave);
  // bucle hasta encontrar una celda vacía
  // o una cuya clave coincida con la buscada
  while (tabla[pos] != null &&
         !e.equals(tabla[pos]))
  {
    pos++; // ir a la siguiente celda
    pos = pos % tabla.length;
  }
  return pos;
}
```


Implementación de un mapa mediante troceado cerrado (cont.)

```
public V put(K clave, V valor) {
  Entry<K,V> nuevo= new Entry<K,V>(clave,valor);
  int pos=buscaIndice(nuevo);
  V viejo=null;
  if (tabla[pos]!=null &&
      !((Entry<K,V>)tabla[pos]).borrado)
  {
      viejo=((Entry<K,V>)tabla[pos]).valor;
  }
  if (tabla[pos]==null) {
     num++;
  }
  tabla[pos] = nuevo;
```

Implementación de un mapa mediante troceado cerrado (cont.)

```
    if (num>=tabla.length/2) {
        // recrear la tabla
        Object[] tablaAntigua=tabla;
        tabla= new Object[tablaAntigua.length*2];
        num=0;
        for (int i=0; i<tablaAntigua.length; i++) {
            Entry<K,V> e=(Entry<K,V>)tablaAntigua[i];
            if (e!=null && !e.borrado) {
                put(e.clave, e.valor);
            }
        }
    }
    return viejo;
}
```


Implementación de un mapa mediante troceado cerrado (cont.)

```
public V remove(K clave) {
  Entry<K,V> e=new Entry<K,V>(clave,null);
  int pos=buscaIndice(e);
  if (tabla[pos]==null ||
      ((Entry<K,V>)tabla[pos]).borrado)
  {
      return null;
  } else {
     V valor=((Entry<K,V>)tabla[pos]).valor;
     ((Entry<K,V>)tabla[pos]).borrado=true;
     return valor;
  }
}
```

Implementación de un mapa mediante troceado cerrado (cont.)

```
  public V get(K clave) {
    Entry<K,V> e=new Entry<K,V>(clave,null);
    int pos=buscaIndice(e);
    if (tabla[pos]==null ||
        ((Entry<K,V>)tabla[pos]).borrado)
    {
        return null;
    } else {
       return ((Entry<K,V>)tabla[pos]).valor;
    }
  }
```


Implementación de un mapa mediante troceado cerrado (cont.)

```
  boolean containsKey(K clave) {
    Entry<K,V> e=new Entry<K,V>(clave,null);
    int pos=buscaIndice(e);
    return tabla[pos]!=null &&
           !((Entry<K,V>)tabla[pos]).borrado;
  }
}
```

6.2. Árboles

Los árboles son estructuras de datos que permiten el almacenamiento jerárquico de información

Tienen gran cantidad de aplicaciones; por ejemplo:

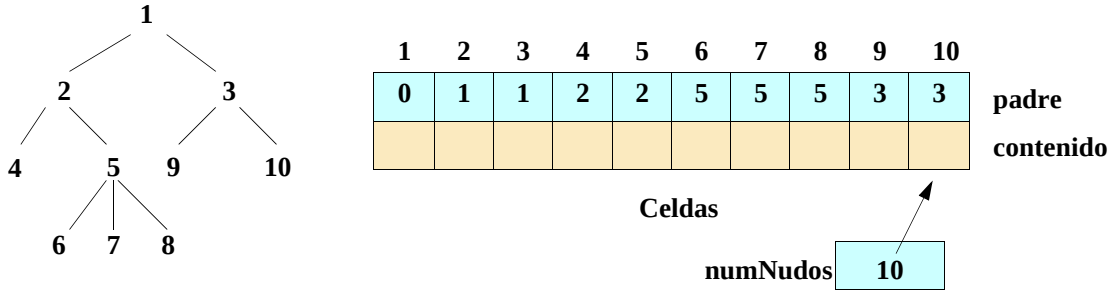
- sistemas de ficheros
- análisis del texto en compiladores
- algoritmos de búsqueda

Estudiaremos diferentes implementaciones

- nudos con referencias al padre
- nudos con listas de hijos
- nudos con referencia al hijo izquierdo y hermano derecho

Implementación mediante arrays con cursor al padre

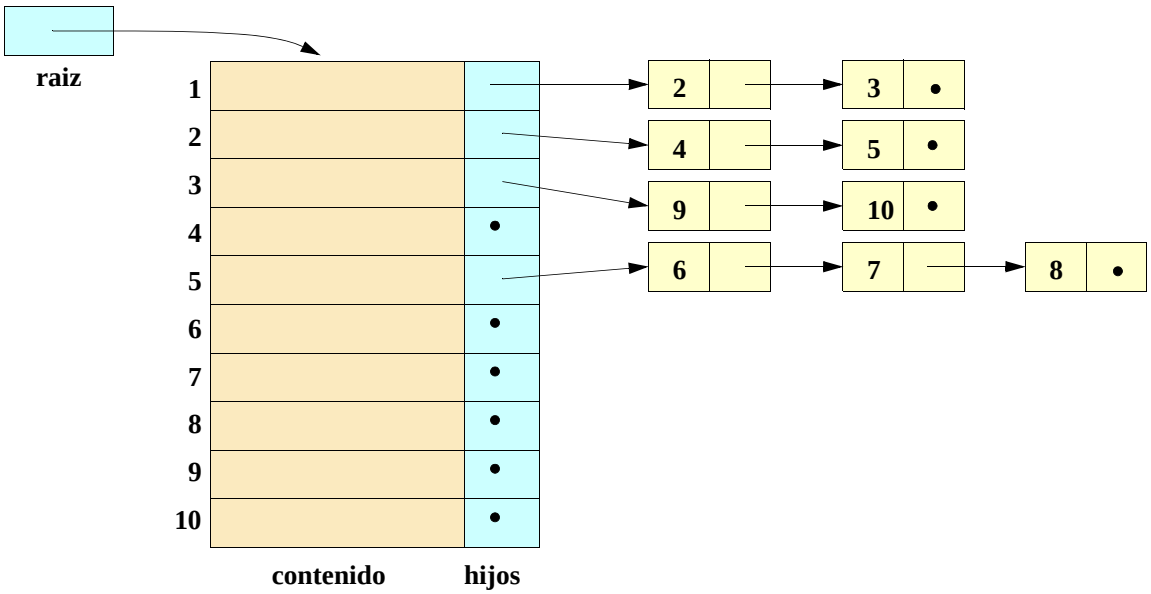
La implementación más sencilla es la de un array en el que cada elemento tiene un cursor al padre



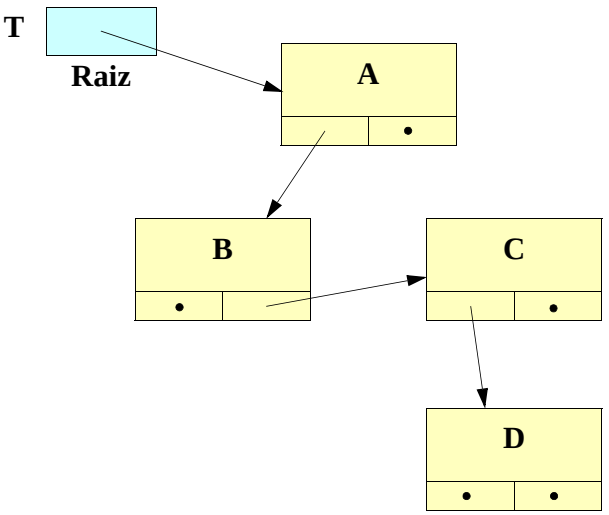
Para que el orden de los nudos esté bien definido

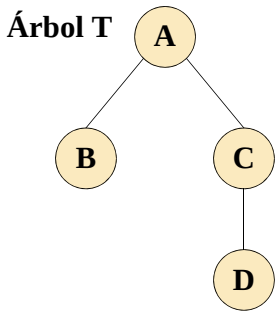
- se numeran los hijos con números mayores al del padre
- se numeran los hermanos en orden creciente de izquierda a derecha

Representación de árboles con listas de hijos

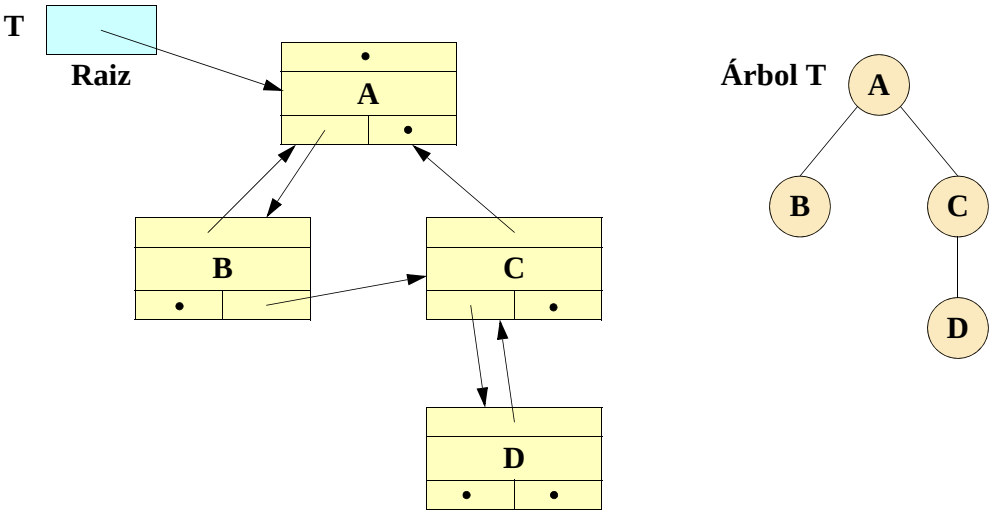


La representación hijo-más-izquierdo, hermano-derecho





La representación padre, hijo-más-izquierdo, hermano-derecho



Implementación de árboles en Java

Realizaremos una implementación de la interfaz `Arbol` del capítulo 3

- usaremos la técnica de anotar en cada nudo el padre, hijo más izquierdo, hermano derecho
- el iterador del árbol contendrá una referencia al nudo actual, y una referencia al árbol original

Implementación de árboles en Java

```
public class ArbolCE <E> implements Arbol<E>
{
  // Clase interna que representa un nudo del arbol
  private static class Nudo<E> {
      Nudo<E> padre, hijoIzquierdo, hermanoDerecho;
      E contenido;
  }

  // El atributo del arbol es su raíz
 private  Nudo<E> raiz;

  /**
   * Constructor que crea el árbol vacío
   */
```


Implementación de árboles en Java (cont.)

```
  public ArbolCE ()
  {
      raiz=null;
  }

  /**
   * Constructor al que se le pasa un elemento
   */
  public ArbolCE (E elementoRaiz)
  {
      raiz=new Nudo<E>();
      raiz.contenido=elementoRaiz;
  }
```

Implementación de árboles en Java (cont.)

```
/**
 * Constructor interno al que se le pasa un nudo,
 * que será la raíz del árbol
 */
ArbolCE (Nudo<E> nudoRaiz)
{
    raiz=nudoRaiz;
}

/**
 * obtener el iterador del arbol
 */
 public IteradorDeArbol<E> iterador() {
    return new IteradorDeArbolCE<E>(this);
 }
```


Implementación de árboles en Java (cont.)

```
/**
 * Dejar el árbol vacío
 */
public void hazNulo() {
    raiz=null;
}

/**
 * Comprobar si el árbol está vacío
 */
public boolean estaVacio() {
    return raiz==null;
}
```

Implementación del iterador del árbol

```
public class IteradorDeArbolCE<E>
    implements IteradorDeArbol<E>
{
  // atributos privados: el árbol y el nudo actual
  private ArbolCE<E> arbol;
  private Nudo<E> nudoActual;

  /**
   * Constructor al que se le pasa el árbol
   * Pone el nudo actual igual a la raiz
   */
  public IteradorDeArbolCE (ArbolCE<E> arbol) {
    this.arbol=arbol;
    this.nudoActual=arbol.raiz;
  }
```


Implementación del iterador del árbol (cont.)

```
  /**
   * Añade un hijo al nudo actual, situado más a la
   * izquierda que los actuales */
  public void insertaPrimerHijo(E elemento)
     throws NoValido
  {
    if (nudoActual==null) {
      throw new NoValido();
    }
    Nudo<E> nuevo=new Nudo<E>();
    nuevo.padre=nudoActual;
    nuevo.hermanoDerecho=nudoActual.hijoIzquierdo;
    nuevo.contenido=elemento;
    nudoActual.hijoIzquierdo=nuevo;
  }
```

Implementación del iterador del árbol (cont.)

```
/**
 * Añade un hijo al padre del nudo actual,
 * situándolo a la derecha del nudo actual
 */
public void insertaSiguienteHermano(E elemento)
   throws EsRaiz, NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  if (nudoActual.padre==null) {
    throw new EsRaiz();
  }
```


Implementación del iterador del árbol (cont.)

```
  Nudo<E> nuevo=new Nudo<E>();
  nuevo.padre=nudoActual.padre;
 nuevo.hermanoDerecho=nudoActual.hermanoDerecho;
  nuevo.contenido=elemento;
  nudoActual.hermanoDerecho=nuevo;
}

/**
 * Si el nudo actual es una hoja, la elimina del
* árbol y hace que el nudo actual sea su padre. Si
 * no es una hoja, lanza NoEsHoja
 */
```

Implementación del iterador del árbol (cont.)

```
public E eliminaHoja() throws NoEsHoja, NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  if (nudoActual.hijoIzquierdo!=null) {
    throw new NoEsHoja();
  }
  ArbolCE<E> rama=(ArbolCE<E>) cortaRama();
  E valor=rama.raiz.contenido;
  return valor;
}
```


Implementación del iterador del árbol (cont.)

```
/**
 * Modifica el contenido del nudo actual
 * reemplazándolo por el elementoNuevo
 */
public E modificaElemento (E elementoNuevo)
  throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  E valor=nudoActual.contenido;
  nudoActual.contenido=elementoNuevo;
  return valor;

}
```

Implementación del iterador del árbol (cont.)

```
/**
 * Elimina la rama del árbol cuya raíz es el nudo
 * actual, y hace que el nudo actual sea su padre.
 * Retorna la rama cortada como un árbol
 */
public Arbol<E> cortaRama() throws NoValido
{
  // comprobar errores
  if (nudoActual==null) {
    throw new NoValido();
  }

  Nudo<E> padre=nudoActual.padre;
  // si es la raiz del arbol, lo dejamos vacio
```


Implementación del iterador del árbol (cont.)

```
  if (padre==null) {
    ArbolCE<E> rama=new ArbolCE<E>(nudoActual);
    arbol.raiz=null;
    return rama;
  } else {
    // comprobar si la hoja es el hijo izquierdo
    if (padre.hijoIzquierdo==nudoActual) {
      padre.hijoIzquierdo=  // es hijo izquierdo
         nudoActual.hermanoDerecho;
    } else {
    // no es hijo izqdo.; buscar hermano izqdo., n
      Nudo<E> n=padre.hijoIzquierdo;
      boolean encontrado=false;
      while (! encontrado) {
        if (n.hermanoDerecho==nudoActual) {
```

Implementación del iterador del árbol (cont.)

```
            encontrado=true;
          } else {
            n=n.hermanoDerecho;
          }
        }
       n.hermanoDerecho=nudoActual.hermanoDerecho;
      }
      // preparar el arbol que vamos a devolver
      ArbolCE<E> rama=new ArbolCE<E>(nudoActual);
      rama.raiz.padre=null;
      rama.raiz.hermanoDerecho=null;
      nudoActual=padre;
      return rama;
    }
  }
```


Implementación del iterador del árbol (cont.)

```
  /**
   * Reemplaza la rama cuya raíz es el nudo actual,
   * sustituyéndola por nuevaRama;
   * Retorna la rama que ha sido reemplazada
   */
  public Arbol<E> reemplazaRama(Arbol<E> nuevaRama)
    throws NoValido
  {
    if (nudoActual==null) {
      throw new NoValido();
    }
    // preparar la vieja rama
    Nudo<E> copiaRaiz=new Nudo<E>();
    copiaRaiz.contenido=nudoActual.contenido;
```

Implementación del iterador del árbol (cont.)

```
    copiaRaiz.padre=null;
    copiaRaiz.hijoIzquierdo=
       nudoActual.hijoIzquierdo;
    copiaRaiz.hermanoDerecho=null;
   ArbolCE<E> viejaRama=new ArbolCE<E>(copiaRaiz);

    // copiar la raiz de la nueva rama
    ArbolCE<E> nr=(ArbolCE<E>) nuevaRama;
    nudoActual.contenido=nr.raiz.contenido;
   nudoActual.hijoIzquierdo=nr.raiz.hijoIzquierdo;
    return viejaRama;
  }
```


Implementación del iterador del árbol (cont.)

```
/**
 * Añade el árbol indicado por nuevaRama haciendo
* que su raíz sea hija del nudo actual, situándola
 * a la derecha de los hijos actuales, si los hay
 */
public void anadeRama(Arbol<E> nuevaRama)
   throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  // Comprobar si el nudo actual tiene hijos
  if (nudoActual.hijoIzquierdo==null) {
    // no tiene hijos;
    // añadir la rama como hijo izquierdo
```

Implementación del iterador del árbol (cont.)

```
      nudoActual.hijoIzquierdo=
        ((ArbolCE<E>)nuevaRama).raiz;
    } else {
      // Buscar el ultimo hijo del nudo actual
      Nudo<E> n=nudoActual.hijoIzquierdo;
      while (! (n.hermanoDerecho==null)) {
        n=n.hermanoDerecho;
      }
    n.hermanoDerecho=((ArbolCE<E>)nuevaRama).raiz;
    }
    ((ArbolCE<E>)nuevaRama).raiz.padre=nudoActual;
  }
```


Implementación del iterador del árbol (cont.)

```
  /**
   * retorna el elemento contenido en el nudo actual
    */
  public E contenido() throws NoValido
  {
    if (nudoActual==null) {
      throw new NoValido();
    }
    return nudoActual.contenido;
  }
```

Implementación del iterador del árbol (cont.)

```
/**
 * Hace que el nudo actual sea la raíz del árbol;
 * valdrá no válido si el árbol está vacío
 */
public void irARaiz()
{
  nudoActual=arbol.raiz;
}
```


Implementación del iterador del árbol (cont.)

```
/**
 * Hace que el nudo actual sea el primer hijo del
 * actual; valdrá no válido si el nudo actual no
 * tiene hijos
 */
public void irAPrimerHijo() throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  nudoActual=nudoActual.hijoIzquierdo;
}
```

Implementación del iterador del árbol (cont.)

```
/**
 * Hace que el nudo actual sea el siguiente hermano
  * del actual; valdrá no válido si el nudo actual
  * no tiene hermanos derechos
  */
public void irASiguienteHermano() throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  nudoActual=nudoActual.hermanoDerecho;
}
```


Implementación del iterador del árbol (cont.)

```
/**
 * Hace que el nudo actual sea el padre del actual;
  * valdrá no válido si el nudo actual era la raíz
  */
public void irAPadre() throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  nudoActual=nudoActual.padre;
}
```

Implementación del iterador del árbol (cont.)

```
/**
 * Retorna un booleano que indica si el nudo
 * actual es una hoja o no
 */
public boolean esHoja()  throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  return nudoActual.hijoIzquierdo==null;
}
```


Implementación del iterador del árbol (cont.)

```
/**
* Retorna un booleano que indica si el nudo actual
 * es la raíz del árbol
 */
public boolean esRaiz() throws NoValido
{
  if (nudoActual==null) {
      throw new NoValido();
  }
  return nudoActual.padre==null;
}
```

Implementación del iterador del árbol (cont.)

```
/**
 *  Retorna un booleano que indica si el nudo
 *  actual es el último hijo de su padre
 */
public boolean esUltimoHijo() throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  return nudoActual.hermanoDerecho==null;
}
```


Implementación del iterador del árbol (cont.)

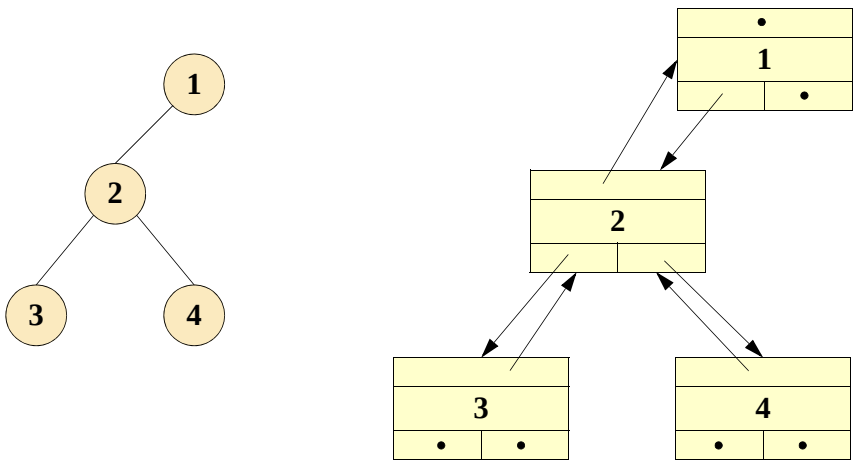
```
/**
* Retorna un booleano que indica si el nudo actual
 * es válido, o no
 */
public boolean esValido() {
    return nudoActual!=null;
}
```

Implementación del iterador del árbol (cont.)

```
    /**
     * Retorna un iterador de árbol que es una copia
     * del actual
     */
    public IteradorDeArbol<E> clone() {
      IteradorDeArbolCE<E> copia =
         new IteradorDeArbolCE<E>(this.arbol);
      copia.nudoActual=this.nudoActual;
      return copia;
    }
  }
}
```


6.3. Árboles binarios

Una posible implementación: cada nudo tiene un puntero al padre, al hijo izquierdo, y al hijo derecho:



Implementación en Java del árbol binario

```
public class ArbolBinCE <E>
  implements ArbolBinario<E>
{
  // Clase interna que representa un nudo del arbol
  private static class Nudo<E> {
    Nudo<E> padre, hijoIzquierdo, hijoDerecho;
    E contenido;
  }

  // El atributo del arbol es su raiz
  Nudo<E> raiz;

  /**
   * Constructor que crea el arbol vacio
   */
```


Implementación en Java del árbol binario (cont.)

```
  public ArbolBinCE ()
  {
    raiz=null;
  }


  /**
   * Constructor al que se le pasa un elemento
   */
  public ArbolBinCE (E elementoRaiz)
  {
    raiz=new Nudo<E>();
    raiz.contenido=elementoRaiz;
  }
```

Implementación en Java del árbol binario (cont.)

```
/**
 * Constructor: se pasan las ramas izqda y drcha
 */
public ArbolBinCE (E elementoRaiz,
    ArbolBinario<E> ramaIzquierda,
    ArbolBinario<E> ramaDerecha)
{
  raiz = new Nudo<E>();
  raiz.contenido = elementoRaiz;
```


Implementación en Java del árbol binario (cont.)

```
  //Enlazar rama izquierda
  Nudo<E> raizIzquierda=
    ((ArbolBinCE<E>) ramaIzquierda).raiz;
  if (raizIzquierda != null) {
     raizIzquierda.padre = raiz;
  }
  raiz.hijoIzquierdo= raizIzquierda;
  //Enlazar rama derecha
  Nudo<E> raizDerecha=
    ((ArbolBinCE<E>) ramaDerecha).raiz;
  if (raizDerecha != null) {
     raizDerecha.padre = raiz;
  }
  raiz.hijoDerecho= raizDerecha;
}
```

Implementación en Java del árbol binario (cont.)

```
/**
 * obtener el iterador del arbol
 */
public IterArbolBin<E> iterador()
{
  return new IterArbolBinCE<E>(this);
}
```


Implementación en Java del árbol binario (cont.)

```
/**
 * Dejar el arbol vacio
 */
public void hazNulo() {
  raiz=null;
}

/**
 * Comprobar si el arbol esta vacio
 */
public boolean estaVacio() {
  return raiz==null;
}
```

Implementación en Java del iterador del árbol binario

```
public class IterArbolBinCE<E>
   implements IterArbolBin<E>
{
  // atributos privados: el arbol y el nudo actual
  private ArbolBinCE<E> arbol;
  private Nudo<E> nudoActual;

  /**
   * Constructor al que se le pasa el arbol
   * Pone el nudo actual igual a la raiz
   */
  public IterArbolBinCE (ArbolBinCE<E> arbol) {
    this.arbol=arbol;
    this.nudoActual=arbol.raiz;
  }
```


Implementación en Java del iterador del árbol binario (cont.)

```
  /**
   * Anade un hijo izquierdo al nudo actual
   * Lanza YaExiste si ya existía un hijo derecho
   */
  public void insertaHijoIzquierdo(E elemento)
     throws NoValido, YaExiste
  {
    if (nudoActual==null) {
      throw new NoValido();
    }
    if (nudoActual.hijoIzquierdo != null) {
      throw new YaExiste();
    }
    Nudo<E> nuevo=new Nudo<E>();
```

Implementación en Java del iterador del árbol binario (cont.)

```
    nuevo.padre=nudoActual;
    nuevo.contenido=elemento;
    nudoActual.hijoIzquierdo=nuevo;
  }

  /**
   * Anade un hijo derecho al nudo actual
   * Lanza YaExiste si ya existía un hijo derecho
   */
  public void insertaHijoDerecho(E elemento)
    throws NoValido, YaExiste
  {
    if (nudoActual==null) {
      throw new NoValido();
    }
```


Implementación en Java del iterador del árbol binario (cont.)

```
    if (nudoActual.hijoDerecho != null) {
      throw new YaExiste();
    }
    Nudo<E> nuevo=new Nudo<E>();
    nuevo.padre=nudoActual;
    nuevo.contenido=elemento;
    nudoActual.hijoDerecho=nuevo;
  }

  /**
   * Si el nudo actual es una hoja, la elimina del
   * arbol y hace que el nudo actual sea su padre.
   * Si no es una hoja, lanza NoEsHoja
   */
```

Implementación en Java del iterador del árbol binario (cont.)

```
public E eliminaHoja() throws NoEsHoja, NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  if ((nudoActual.hijoIzquierdo!=null) ||
      (nudoActual.hijoDerecho!=null))
  {
    throw new NoEsHoja();
  }
  E valor = nudoActual.contenido;

 // si la hoja es raiz del arbol lo dejamos vacio
```


Implementación en Java del iterador del árbol binario (cont.)

```
  if (nudoActual.padre==null) {
    arbol.raiz=null;
    nudoActual=null;
  } else {
    Nudo<E> padre=nudoActual.padre;
    if (padre.hijoIzquierdo==nudoActual) {
      padre.hijoIzquierdo=null;
    } else { // es el hijo derecho
      padre.hijoDerecho=null;
    }
    // cambiamos el nudo actual del iterador
    nudoActual = padre;
  }
  return valor;
}
```

Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * Modifica el contenido del nudo actual
 * reemplazándolo por el elementoNuevo. Retorna
 * el antiguo contenido del nudo actual
 */
public E modificaElemento (E elementoNuevo)
   throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  E valor=nudoActual.contenido;
  nudoActual.contenido=elementoNuevo;
  return valor;
}
```


Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * Reemplaza la rama del arbol cuya raiz es el
 * hijo izquierdo del nudo actual, sustituyéndola
 * por nuevaRama. Retorna la rama reemplazada
 */
public ArbolBinario<E> reemplazaRamaIzquierda
  (ArbolBinario<E> nuevaRama) throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  Nudo<E> hijo = nudoActual.hijoIzquierdo;
  // preparar la vieja rama
  ArbolBinCE<E> viejaRama=new ArbolBinCE<E>();
```

Implementación en Java del iterador del árbol binario (cont.)

```
    if (hijo!= null) {
      hijo.padre=null;
    }
    viejaRama.raiz=hijo;

    // enlazar la nueva rama
    Nudo<E> nuevoHijo=
      ((ArbolBinCE<E>) nuevaRama).raiz;
    if (nuevoHijo!= null) {
      nuevoHijo.padre=nudoActual;
    }
    nudoActual.hijoIzquierdo= nuevoHijo;
    return viejaRama;
  }
```


Implementación en Java del iterador del árbol binario (cont.)

```
  /**
   * reemplaza la rama del arbol cuya raiz es el
   * hijo derecho del nudo actual, sustituyéndola
   * por nuevaRama. Retorna la rama reemplazada
   */
  public ArbolBinario<E> reemplazaRamaDerecha
      (ArbolBinario<E> nuevaRama) throws NoValido
  {
    if (nudoActual==null) {
      throw new NoValido();
    }
    Nudo<E> hijo = nudoActual.hijoDerecho;
    // preparar la vieja rama
    ArbolBinCE<E> viejaRama=new ArbolBinCE<E>();
```

Implementación en Java del iterador del árbol binario (cont.)

```
    if (hijo!= null) {
      hijo.padre=null;
    }
    viejaRama.raiz=hijo;

    // enlazar la nueva rama
    Nudo<E> nuevoHijo=
      ((ArbolBinCE<E>) nuevaRama).raiz;
    if (nuevoHijo!= null) {
      nuevoHijo.padre=nudoActual;
    }
    nudoActual.hijoDerecho= nuevoHijo;
    return viejaRama;
  }
```


Implementación en Java del iterador del árbol binario (cont.)

```
  /**
   * Retorna el elemento contenido en el nudo
   * actual
   */
  public E contenido() throws NoValido
  {
    if (nudoActual==null) {
      throw new NoValido();
    }
    return nudoActual.contenido;
  }
```

Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * Hace que el nudo actual sea la raiz del arbol;
 * valdrá no válido si el arbol esta vacio
 */
public void irARaiz()
{
  nudoActual=arbol.raiz;
}
```


Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * Pasa al hijo izquierdo del actual; valdrá no
 * válido si el nudo actual no tiene hijo
 * izquierdo
 */
public void irAHijoIzquierdo() throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  nudoActual=nudoActual.hijoIzquierdo;
}
```

Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * Pasa al hijo derecho del actual; valdrá no
 * válido si el nudo actual no tiene hijo derecho
 */
public void irAHijoDerecho() throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  nudoActual=nudoActual.hijoDerecho;
}
```


Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * Pasa al padre del actual; valdrá no válido si
 * el nudo actual era la raiz
 */
public void irAPadre() throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  nudoActual=nudoActual.padre;
}
```

Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * Retorna un booleano que indica si el nudo
 * actual es una hoja o no
 */
public boolean esHoja()  throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  return (nudoActual.hijoIzquierdo==null) &&
         (nudoActual.hijoDerecho==null);
}
```


Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * Retorna un booleano que indica si el nudo
 * actual es la raiz del arbol
 */
public boolean esRaiz() throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  return nudoActual.padre==null;
}
```

Implementación en Java del iterador del árbol binario (cont.)

```
/**
 *  Retorna un booleano que indica si el nudo
 *  actual tiene hijo izquierdo o no
 */
public boolean tieneHijoIzquierdo()
  throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  return nudoActual.hijoIzquierdo!=null;
}
```


Implementación en Java del iterador del árbol binario (cont.)

```
/**
 *  Retorna un booleano que indica si el nudo
 *  actual tiene hijo izquierdo o no
 */
public boolean tieneHijoDerecho() throws NoValido
{
  if (nudoActual==null) {
    throw new NoValido();
  }
  return nudoActual.hijoDerecho!=null;
}
```

Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * retorna un booleano que indica si el nudo
 * actual es válido, o no
 */
public boolean esValido() {
  return nudoActual!=null;
}
```


Implementación en Java del iterador del árbol binario (cont.)

```
/**
 * retorna un iterador de arbol que es una copia
 * del actual
 */
public IterArbolBin<E> clone() {
  IterArbolBinCE<E> copia =
     new IterArbolBinCE<E>(this.arbol);
  copia.nudoActual=this.nudoActual;
  return copia;
}
```

Implementación en Java del iterador del árbol binario (cont.)

```
public ArbolBinario<E> clonarArbol() {
  ArbolBinCE<E> clon;
  IterArbolBin<E> iter= this.clone();
  try {
    clon = new ArbolBinCE<E>(iter.contenido());
    IterArbolBin<E> iterClon = clon.iterador();
    if (iter.tieneHijoIzquierdo()) {
      iter.irAHijoIzquierdo();
      ArbolBinario<E> ramaIzquierda =
        iter.clonarArbol();
      iterClon.reemplazaRamaIzquierda
        (ramaIzquierda);
      iter.irAPadre();
    }
    if (iter.tieneHijoDerecho()){
```


Implementación en Java del iterador del árbol binario (cont.)

```
      iter.irAHijoDerecho();
      ArbolBinario<E> ramaDerecha =
        iter.clonarArbol();
     iterClon.reemplazaRamaDerecha(ramaDerecha);
    }
    return clon;
  }  catch (NoValido e) {
    System.out.println("Error inesperado: "+e);
  }   catch (EsRaiz e) {
    System.out.println("Error inesperado: "+e);
  }
  return null;
 }
}
}
```

6.4. Árboles binarios equilibrados y conjuntos ordenados

UNIVERSIDAD DE CANTABRIA

Una de las principales utilidades de los árboles binarios es la búsqueda eficiente y la creación de conjuntos ordenados

- se puede usar para implementar listas en las que la inserción, búsqueda y eliminación son eficientes en promedio: O(log n)
- el arbol binario se ordena de modo que para cada nudo, todos sus descendientes izquierdos sean menores que él, y los derechos mayores
- generalmente evitaremos la duplicidad de elementos
- para mejorar la eficiencia, intentaremos minimizar la profundidad del árbol, haciéndolo equilibrado

En el capítulo 3 se vieron la inserción y búsqueda en árboles binarios ordenados

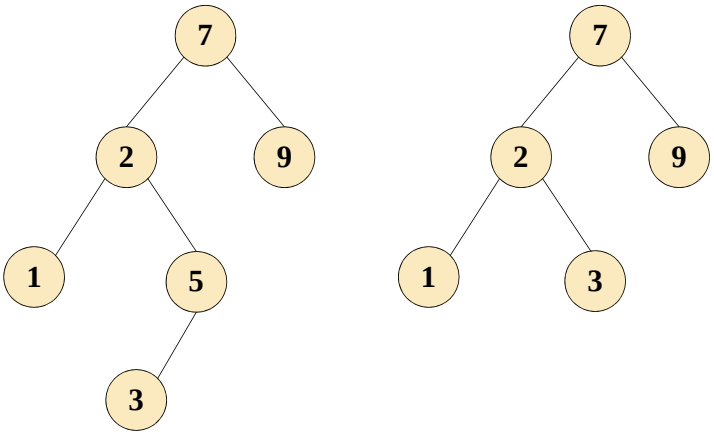
Eliminación en un árbol binario ordenado

UNIVERSIDAD DE CANTABRIA

Se consideren tres casos

a) Si el nudo es una hoja, se puede borrar sin más

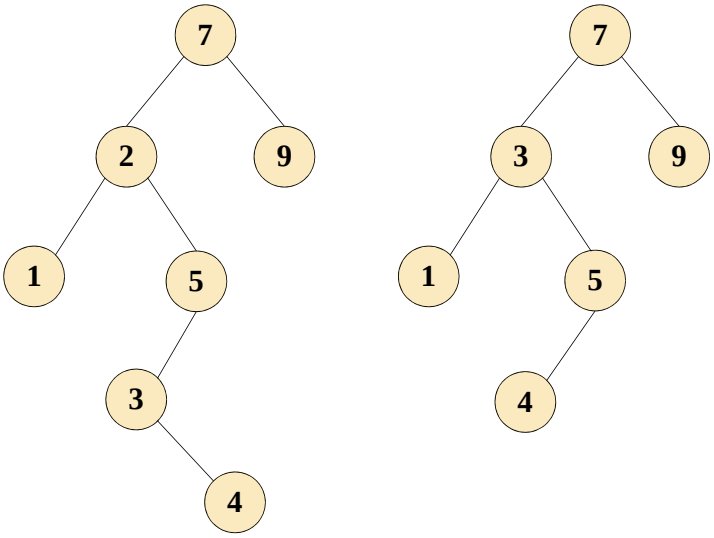
b) Si el nudo tiene un solo hijo se puede eliminar sustituyéndolo en el árbol por ese hijo (con toda su descendencia)
- en la figura se borra el nudo 5



Eliminación en un árbol binario ordenado (cont.)

c) Para eliminar un nudo con dos hijos lo reemplazaremos por el menor elemento de su subárbol derecho, que a su vez eliminaremos

- **este menor elemento eliminado no tiene hijo izquierdo, por lo que se puede borrar con a) o b)**
- **en la figura se elimina el nudo 2**



Eficiencia de las operaciones del árbol

Las operaciones de inserción, búsqueda y eliminación son O(profundidad)

En promedio, para datos ordenados aleatoriamente, la profundidad es O(log n)

Sin embargo, para entradas parcialmente ordenadas, el árbol puede tener mucha profundidad

- **puede estar desequilibrado**
- **en este caso puede interesar utilizar algoritmos que mantengan el árbol equilibrado**

Técnicas para crear árboles binarios equilibrados

Árboles AVL

- deben su nombre a sus autores: Adelson-Velskii y Landis
- aseguran una profundidad que siempre es O(log n)

Árboles rojinegros

- como en el anterior, el peor caso es O(log n)
- presentan la ventaja de que la inserción y eliminación requieren un solo recorrido descendente

Árboles AA

- adaptación de los árboles rojinegros más fácil de implementar
- con un pequeño coste de eficiencia

Técnicas para crear árboles binarios equilibrados (cont.)

B-Árboles

- mejoran la eficiencia usando un árbol ***M-ario***, en lugar de uno binario
- se adaptan bien a estructuras de datos almacenadas en disco, en las que el número de accesos al disco se intenta minimizar
- su implementación es más compleja

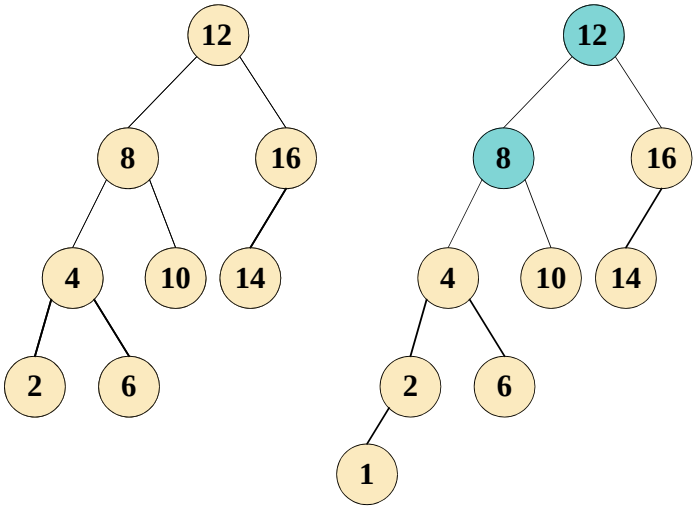
Árboles AVL

Son árboles binarios ordenados con una condición adicional de equilibrio

- las alturas de los hijos derecho e izquierdo de cada nudo sólo pueden diferir, a lo sumo, en 1

Esta propiedad

- garantiza que la profundidad sea siempre O(log n)



Árbol AVL **Árbol no AVL**

Árboles AVL

No garantizan la profundidad mínima

- pero permiten que los algoritmos de inserción y eliminación sean eficientes

Las operaciones de inserción y eliminación se complican con respecto a los árboles binarios ordenados normales

- ya que puede ser necesario restablecer el equilibrio

Inserción en un árbol AVL

Tras una inserción

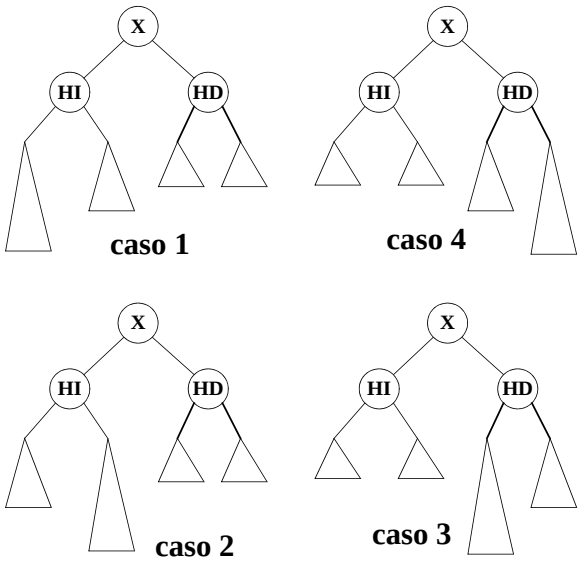
- sólo se ven afectados los nudos en el camino del punto de inserción a la raíz
- restableceremos el equilibrio, si es preciso, en el nudo más profundo que incumple la condición AVL
  - así todo el árbol recupera el equilibrio

Inserción en un árbol AVL

Llamaremos X al nudo que incumple la propiedad AVL

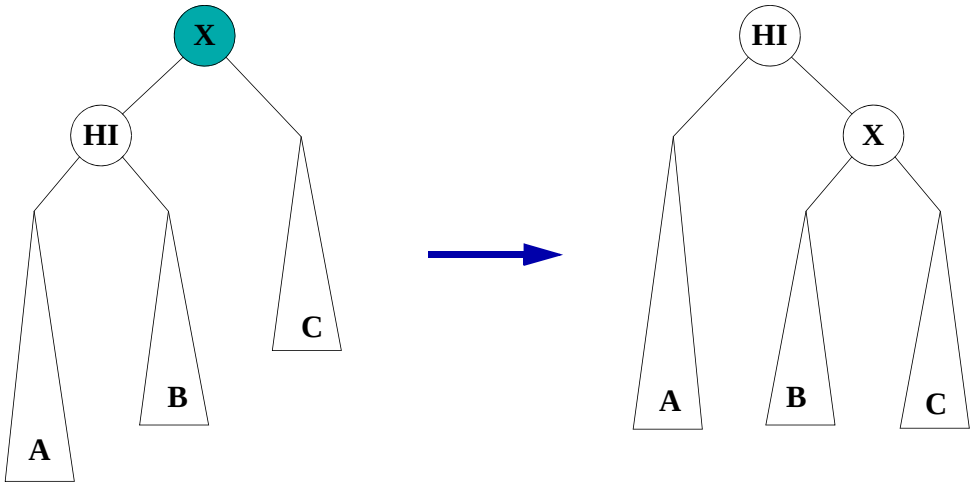
- este nudo tiene a lo sumo dos hijos
- la diferencia entre las profundidades de los dos subárboles es 2

Se dan 4 casos posibles



Operaciones de rotación: rotación simple izquierda de X

Se trata de ajustar la profundidad de dos ramas para hacer el árbol más equilibrado, manteniendo la relación de orden



Se aplica al caso 1

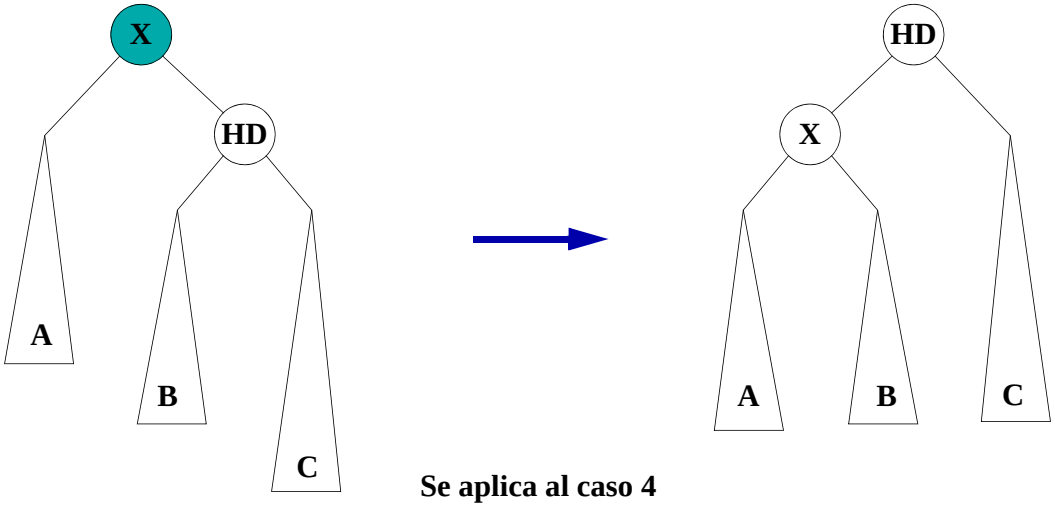
Pseudocódigo de la rotación simple izquierda de X

```
método rotacionSimpleIzquierda (Nudo x) retorna Nudo
   Nudo hi:=x.hijoIzquierdo;
   x.hijoIzquierdo:=hi.hijoDerecho;
   x.hijoIzquierdo.padre=x;
   hi.hijoDerecho:=x;
   x.padre=hi;
   retorna hi;
fmétodo
```

Luego es preciso reinsertar la rama retornada en el lugar donde estaba en el árbol

Operaciones de rotación: rotación simple derecha de X

Es como la anterior, pero elevando la rama derecha



Se aplica al caso 4

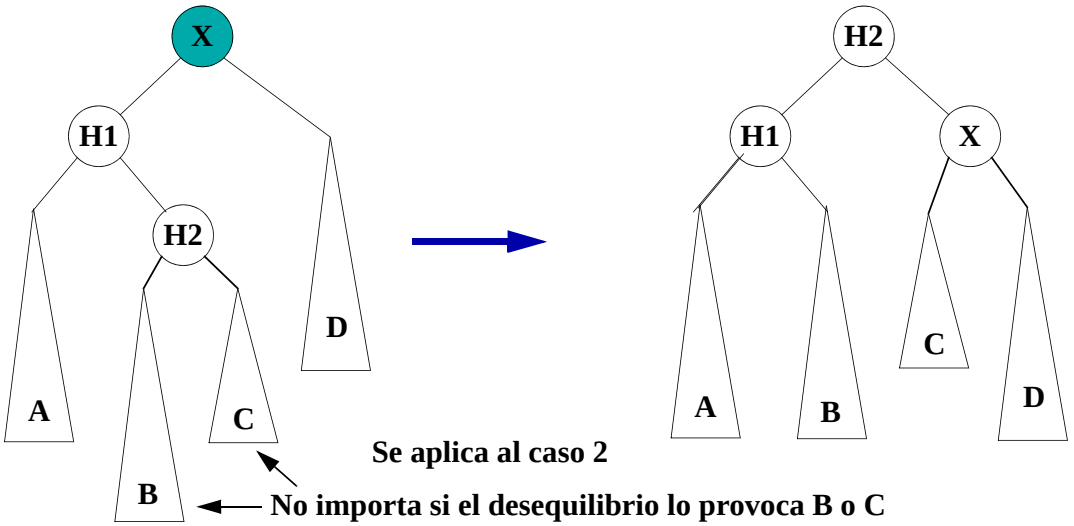
Pseudocódigo de la rotación simple derecha de X

```
método rotacionSimpleDerecha (Nudo x) retorna Nudo
   Nudo hd:=x.hijoDerecho;
   x.hijoDerecho:=hd.hijoIzquierdo;
   x.hijoDerecho.padre=x;
   hd.hijoIzquierdo:=x;
   x.padre=hd;
   retorna hd;
fmétodo
```

Luego es preciso reinsertar la rama retornada en el lugar donde estaba en el árbol

Operaciones de rotación: rotación doble derecha-izquierda de X

Cuando la parte desequilibrada es una rama central, es preciso realizar dos rotaciones: entre H1 y H2 y entre X y H2H



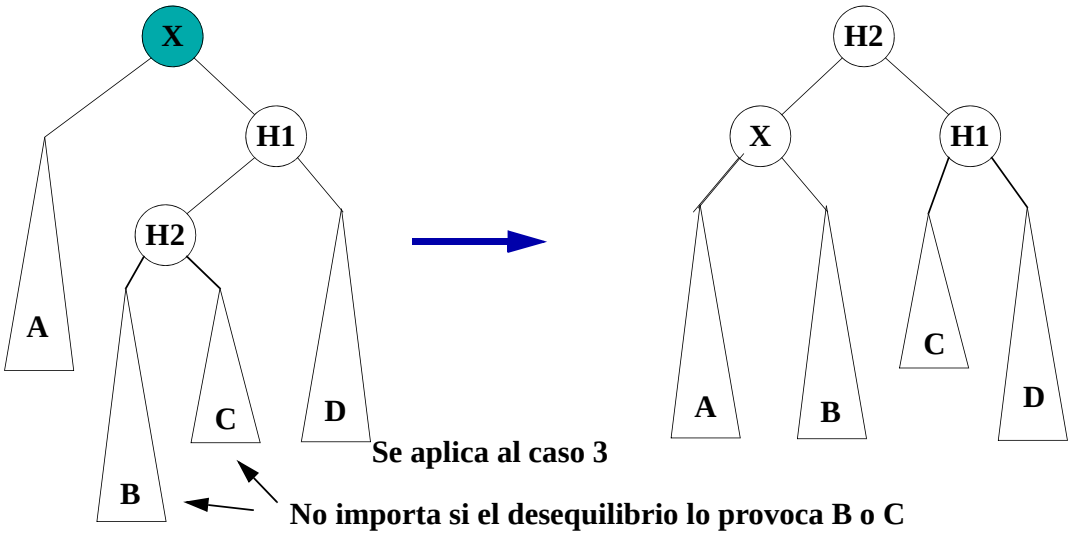
Pseudocódigo de la rotación doble derecha-izquierda

```
método rotacionDobleDI (Nudo x) retorna Nudo
   x.hijoIzquierdo=
      rotacionSimpleDerecha(x.hijoIzquierdo);
   x.hijoIzquierdo.padre=x;
   retorna rotacionSimpleIzquierda(x);
fmétodo
```

Luego es preciso reinsertar la rama retornada en el lugar donde estaba en el árbol

Operaciones de rotación: rotación doble izquierda-derecha de X

Cuando la parte desequilibrada es una rama central, es preciso realizar dos rotaciones: entre H1 y H2 y entre X y H2



Pseudocódigo de la rotación doble izquierda-derecha

```
método rotacionDobleID (Nudo x) retorna Nudo
   x.hijoDerecho=
      rotacionSimpleIzquierda(x.hijoDerecho);
   x.hijoderecho.padre=x;
   retorna rotacionSimpleDerecha(x);
fmétodo
```

Luego es preciso reinsertar la rama retornada en el lugar donde estaba en el árbol

Inserción en el árbol AVL

Se utiliza un algoritmo recursivo para insertar un elemento `e` en el árbol `t`

- si `t` es una hoja, añadimos `e` como hijo izquierdo o derecho, según el orden definido
- si no, insertamos recursivamente `e` en `th`, que es la rama izquierda o la rama derecha de `t`, según el orden definido
- si la altura de `th` no cambia, hemos terminado
- si la altura de `th` cambia, se produce un desequilibrio en `t`, y hay que hacer la correspondiente rotación según el caso en que estemos

Este algoritmo realiza un doble recorrido del arbol, hacia abajo para insertar, y hacia arriba para comprobar el equilibrio

Árboles Rojinegros

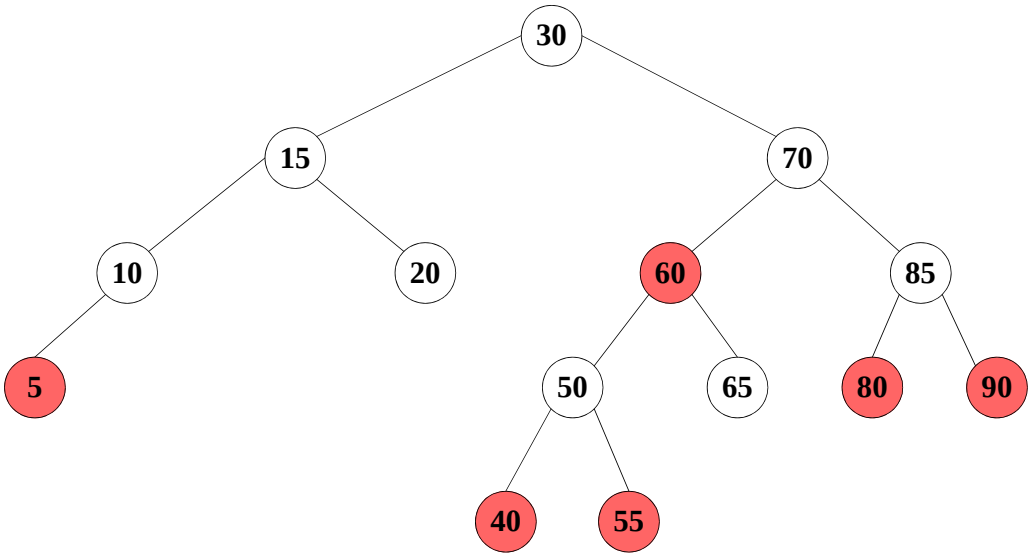
Un árbol rojinegro es un árbol binario ordenado que verifica las siguientes propiedades

- Cada nudo está coloreado de rojo o negro
- La raíz es negra
- Si un nudo es rojo, sus hijos deben ser negros
- Todos los caminos desde un nudo cualquiera a una referencia nula deben tener el mismo número de nudos negros

Evita el doble recorrido del árbol AVL

Estudiaremos ahora las operaciones de inserción y eliminación

Ejemplo de árbol rojinegro



Inserción en un árbol rojinegro

Insertamos un nuevo elemento como una hoja

Para saber dónde insertar hacemos un recorrido descendente como en la inserción no equilibrada

Durante este recorrido, si encontramos un nudo X con dos hijos rojos

- convertimos X en rojo y sus dos hijos en negro
  - el número de nudos negros en un camino no varía
  - pero tendremos dos nudos rojos consecutivos, si el padre de X es rojo
- si el padre de X es rojo, hacemos una rotación simple o doble del padre de X, cambiando los colores de forma apropiada
- Este proceso se repite hasta alcanzar una hoja

Inserción en un árbol rojinegro (cont.)

Pondremos el nuevo nudo rojo

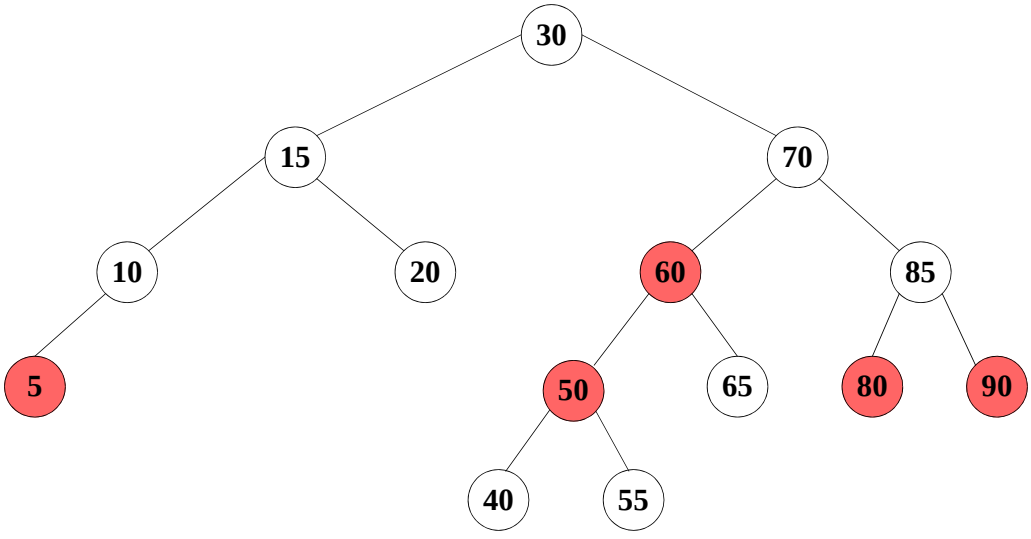
- **si le ponemos negro, incumplimos la última regla**

Por el procedimiento anterior el padre es negro y no hay que hacer nada más

- **p.e., insertar el número 45 en el árbol de la figura anterior**

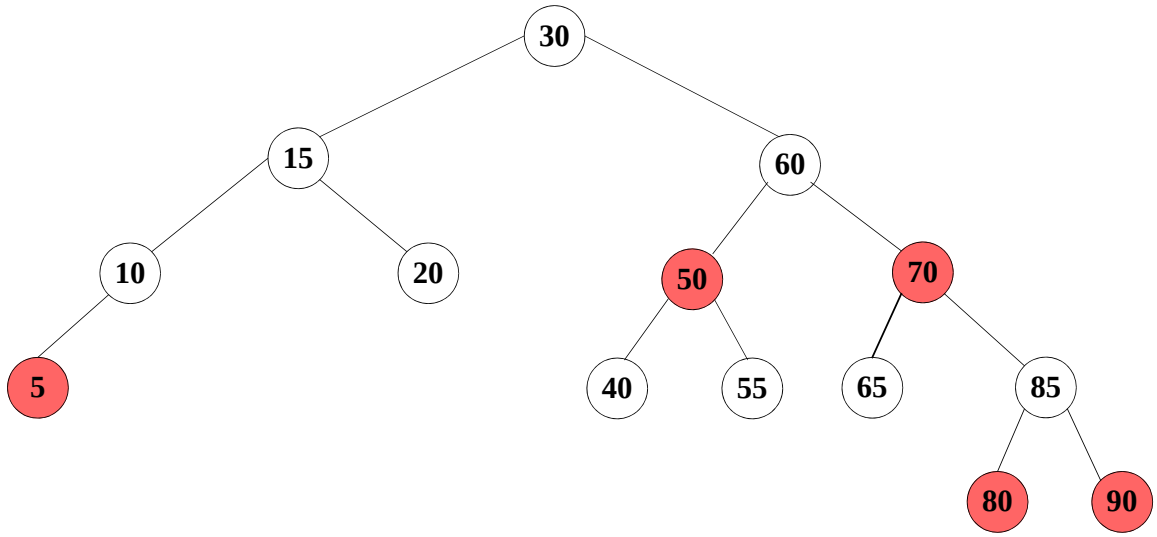
Ejemplo de inserción en un árbol rojinegro (paso 1)

En el recorrido encontramos el nudo 50 con dos hijos rojos, que cambiamos a negros, cambiando luego el 50 a rojo



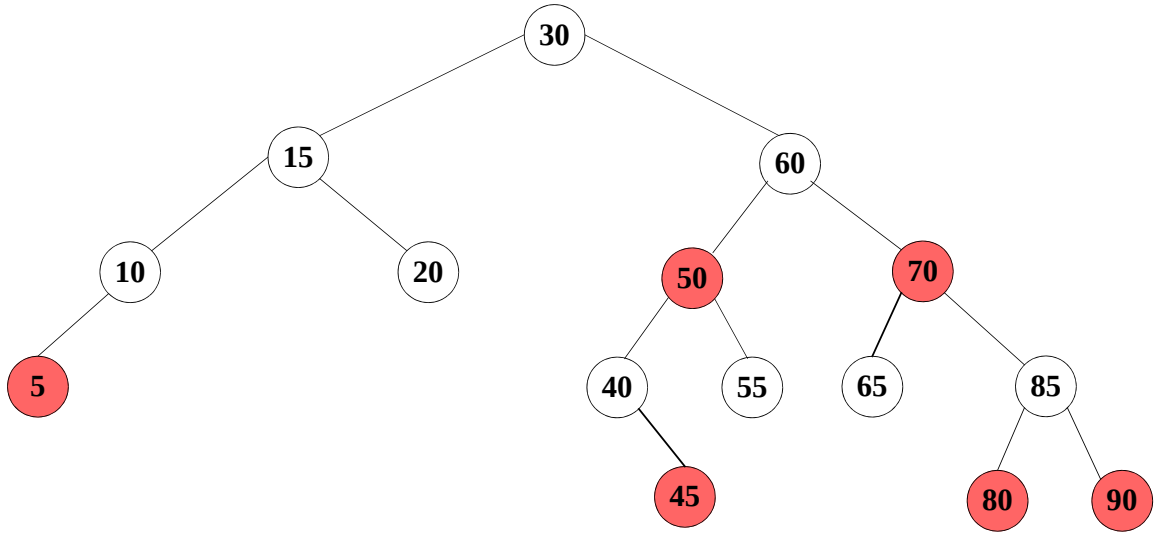
Ejemplo de inserción en un árbol rojinegro (paso 2)

Hacemos una rotación simple de 60 y 70, y reajustamos los colores para recuperar las propiedades



Ejemplo de inserción en un árbol rojinegro (paso 3)

No es preciso hacer más cambios de color ni rotaciones, por lo que finalizamos insertando el nudo 45, de color rojo



Eliminación en árboles rojinegros

Al igual que en el algoritmo de eliminación en árboles no equilibrados

- sólo borramos nudos que son hojas o sólo tienen un hijo
- si tiene dos hijos, se reemplaza su contenido, y se borra otro nudo que es una hoja o sólo tiene un hijo

Si el nudo que vamos a borrar es rojo no hay problema

Si el nudo a borrar es negro, la eliminación hará que se incumpla la 4ª propiedad

- la solución es transformar el árbol para asegurarnos de que siempre borramos un nudo rojo

Eliminación en árboles rojinegros (cont.)

Haremos un recorrido descendente del árbol, comenzando por la raíz; llamaremos

- **X** al nudo actual
- **T** a su hermano
- **P** a su padre

Intentaremos que **X** sea siempre rojo, y mantener las propiedades del árbol a cada paso

- por tanto, al descender, el nuevo **P** será siempre rojo, y el nuevo **X** y el nuevo **T** serán negros

Nudos "centinela"

Para reducir los casos especiales que complicarían el algoritmo, supondremos que existen unos nudos extra o "centinelas", en lugar de los punteros nulos que haya en el árbol

- uno está por encima de la raíz
- si a un nudo le falta un hijo, supondremos un nudo centinela en su lugar
- si un nudo es una hoja, tendrá dos centinelas en lugar de sus hijos

Supondremos los centinelas negros inicialmente

El algoritmo comienza haciendo que **X** sea el centinela por encima de la raíz, y coloreándolo de rojo

Eliminación en árboles rojinegros (cont.)

Existen dos casos principales, además de sus variantes simétricas

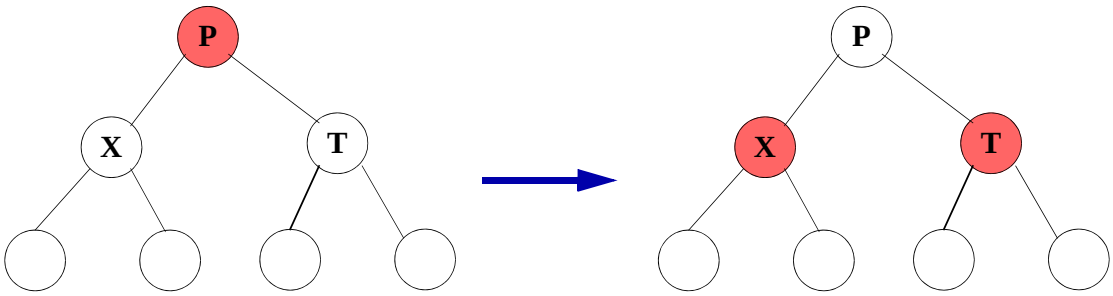
- *caso a*: **X** tiene dos hijos negros
  - *subcaso a1*: **T** tiene dos hijos negros
  - *subcaso a2*: **T** tiene un hijo exterior rojo
  - *subcaso a3*: **T** tiene un hijo interior rojo
  - *subcaso a4*: **T** tiene dos hijos rojos: lo resolvemos como *a2*
- *caso b*: alguno de los hijos de **X** es rojo

Nota

- un hijo es exterior si es un hijo derecho de un hijo derecho o un hijo izquierdo de un hijo izquierdo
- es interior en los otros dos casos

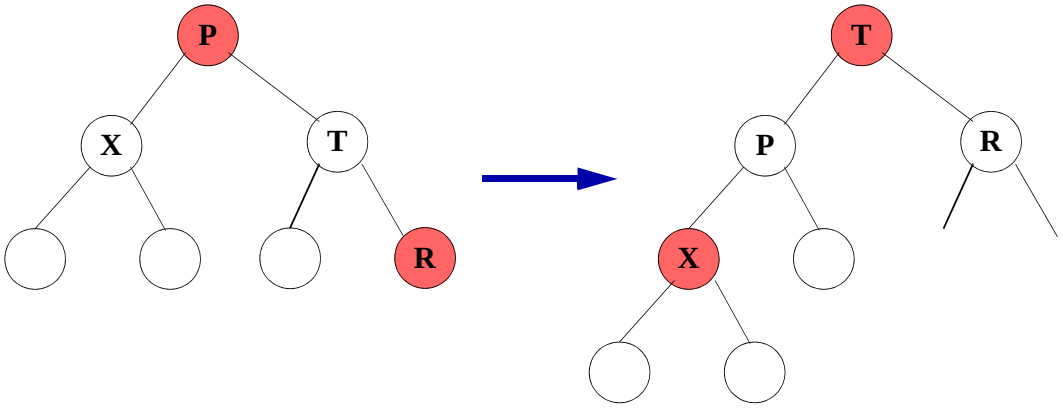
Eliminación en árboles rojinegros: subcaso a1

Hacemos un cambio de color



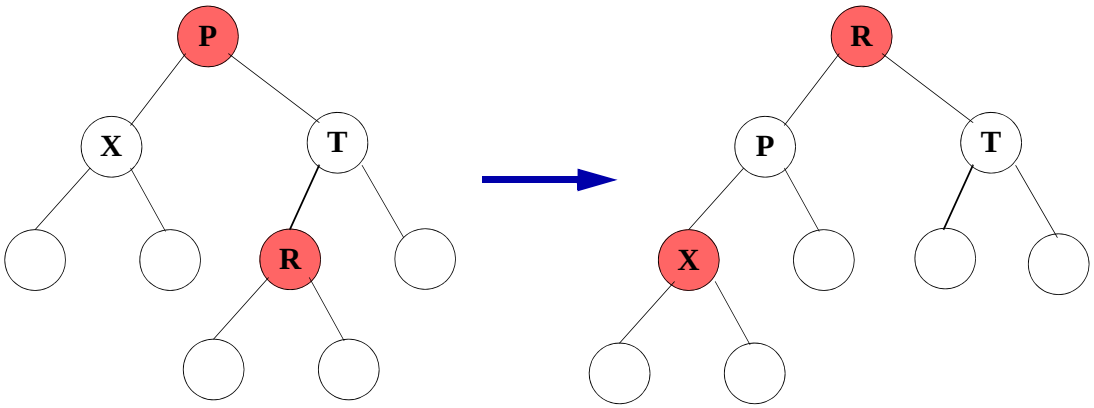
Eliminación en árboles rojinegros: subcasos a2 y a4

Hacemos una rotación simple entre P y T, y los cambios de color que se indican



Eliminación en árboles rojinegros: subcaso a3

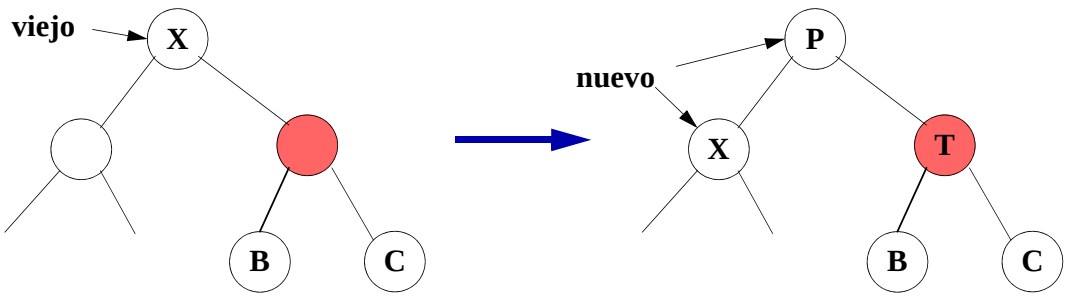
Hacemos una rotación doble entre T y R y luego entre P y R, y los cambios de color que se indican



Eliminación en árboles rojinegros: caso b

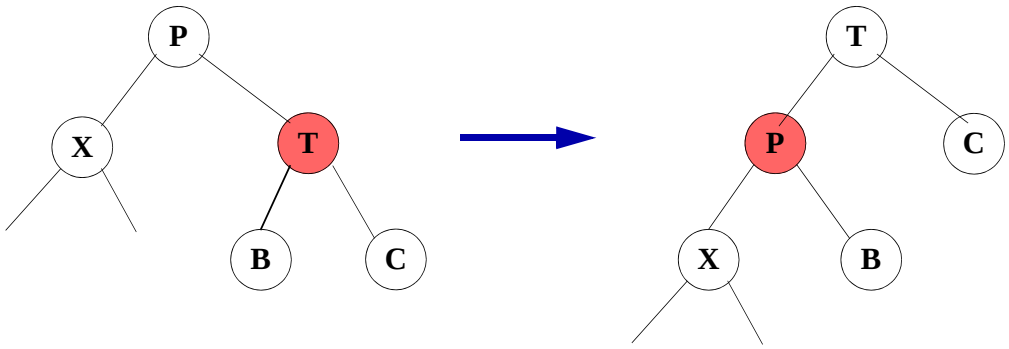
Descendemos al siguiente nivel del árbol obteniendo nuevo X, T, P

- Si el siguiente nudo en el descenso del árbol es rojo, continuamos por él sin necesidad de más cambios
- En caso contrario estamos en esta situación



Eliminación en árboles rojinegros: caso b (cont.)

Hacemos una rotación entre T y P



Y ahora repetimos el algoritmo para tratar de hacer X rojo

Eliminación en árboles rojinegros: caso b (cont.)

El algoritmo siempre finaliza, ya que está garantizado que al llegar al nudo a eliminar habremos alcanzado uno de estos dos casos

- X es una hoja, que se considera que tiene dos hijos negros (caso *a*)
- X tiene un solo hijo
  - si es negro, su "hermano" es un nudo centinela negro, y se aplica el caso *a*
  - si es rojo, eliminamos X y coloreamos ese hijo de negro

6.5. B-árboles

En las estructuras de datos basadas en memoria secundaria la notación $O(n)$ es insuficiente para estudiar el tiempo de ejecución

- **Una operación de E/S en disco tarda varios milisegundos en promedio**
- **equivale a millones de instrucciones de un procesador**

Es preciso encontrar estructuras de datos que minimicen el número de accesos a disco

- **Un árbol binario con 2 millones de registros necesitaría 23 accesos a disco si el caso promedio es log N**
- **Por cada acceso que eliminemos ganaremos mucho tiempo**
  - **incluso aunque compliquemos el código y ejecutemos cientos o miles de instrucciones más**

Árboles M-arios

Cada nudo puede tener hasta M hijos

Ello reduce la profundidad en el caso equilibrado a $\log_M N$

- **Por ejemplo, si M=200, el número de operaciones para N=2 millones de registros es $\log_{200}2000000$=ln 2000000 / ln 200 = 3**

Podemos usar técnicas de ordenación y equilibrado similares a las de los árboles binarios

- **una forma eficiente son los B-árboles**

B-árboles de orden M

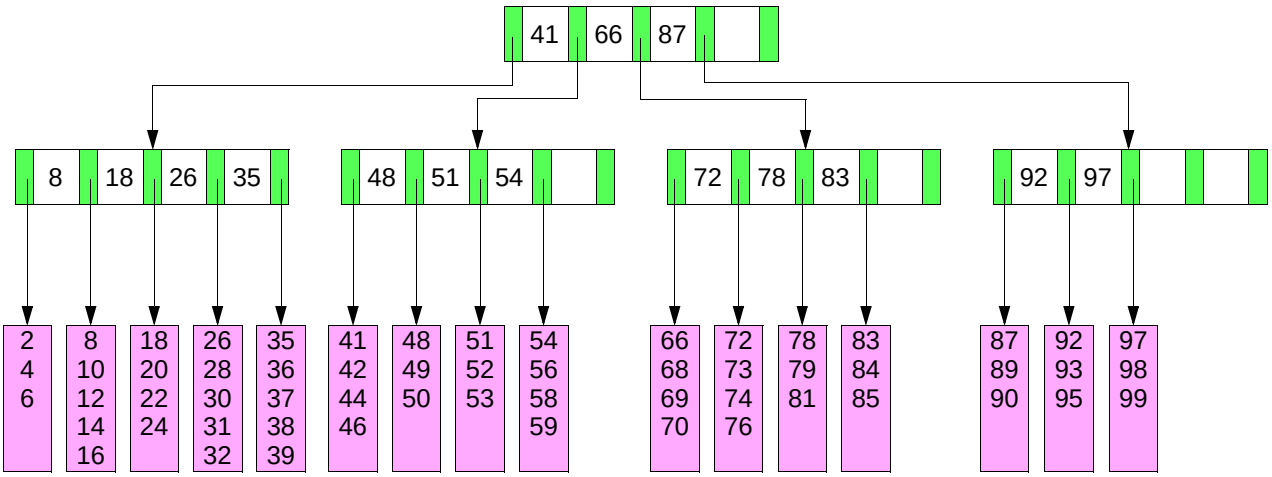
Hay diversas variantes. Por ejemplo los B^+-árboles

Son árboles M-arios que verifican:

1. **Los datos se almacenan en hojas**
2. **Los nudos internos tienen M-1 claves para guiar la búsqueda**
   - **la clave i es la menor clave del subárbol i+1**
3. **La raíz es una hoja o tiene entre 2 y M hijos**
4. **Todos los nudos internos, excepto la raíz, tienen entre $\lceil M/2 \rceil$ y M hijos**
5. **Todas las hojas se encuentran a la misma profundidad y tienen entre $\lceil L/2 \rceil$ y L valores**

Las reglas 3 y 5 se relajan durante las primeras L inserciones

Ejemplo de B-árbol de orden 5, con L=5



B-árboles de orden M Cálculo de L y M

El valor de L se elige en función del tamaño de cada bloque de disco y del tamaño de los datos

- Por ejemplo, $L = \lceil B/R \rceil$
  - B= tamaño del bloque
  - R=tamaño del registro
- Para B=8192 y R=256, L=32

El valor de M se elige en función del tamaño de cada bloque de disco y del tamaño de las claves, C. Cada bloque debe guardar M-1 claves y M cursores o números de bloque

- Por ejemplo, $M = \lceil B/(C*(M-1)+M*4) \rceil$, suponiendo que un número de bloque ocupa 4 bytes
- Para B=8192 y C=32, M=228

Ejemplo

Suponemos N=2000000, B=8192 y R=256, C=32

- L=32
- M=228

El número de hojas es 62500

- La profundidad máxima es 4

Los dos niveles superiores se pueden guardar en memoria principal

- Así, la profundidad máxima de los nudos en memoria secundaria es 2

Operaciones en B-árboles: Búsqueda

UNIVERSIDAD DE CANTABRIA

Hacemos una búsqueda descendiente

- si el nudo es interno
  - comparando la clave con las almacenadas en el nudo sabemos a qué hijo descender
- si el nudo es una hoja
  - buscamos por las claves de los registros, que están ordenadas

Operaciones en B-árboles: Inserción

UNIVERSIDAD DE CANTABRIA

Se comienza por una búsqueda de la hoja

Si el nuevo dato cabe en la hoja, se mete en ella, en el orden adecuado

- una escritura en disco

Si la hoja está llena, se crean dos hojas, metiendo la mitad de los datos en cada una

- dos escrituras para las hojas, y una más para actualizar el padre

Operaciones en B-árboles: Inserción (cont.)

Si el padre estuviese lleno, se divide también, y así sucesivamente
- dos escrituras adicionales por cada división

También es posible dar hijos en adopción a un hermano que tenga sitio para él
- esto ahorra espacio

Si se divide la raíz, se genera una nueva raíz con los dos nudos como hijos
- esto aumenta la profundidad del árbol

Operaciones en B-árboles: Eliminación

Buscamos el dato y lo eliminamos de su hoja

Si la hoja se queda con menos datos que los permitidos
- podemos adoptar un hijo de un hermano, si tiene suficientes
- si no, podemos combinar la hoja con la hermana
- si al hacer esto el padre se queda con pocos hijos, reiteramos el mismo procedimiento
- si la raíz se queda con un solo hijo, la borramos, y ese hijo se convierte en la nueva raíz
  - esto disminuye la profundidad del árbol

6.6. Colas de prioridad

La cola de prioridad se vio en el capítulo 3, y es una estructura de datos para guardar datos comparables, en la que se optimizan

- inserción
- eliminación del elemento mínimo

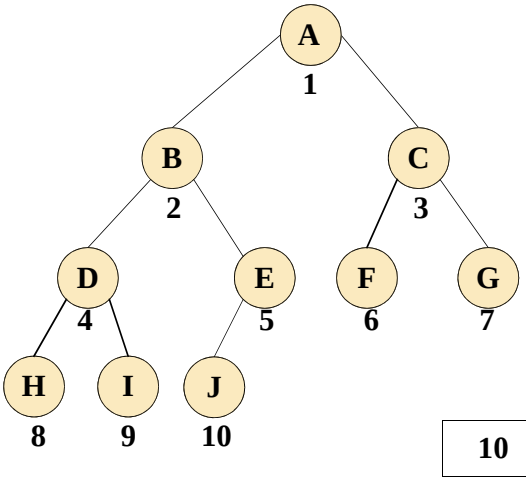
En este apartado veremos una implementación llamada el montículo binario

- tiene tiempos de peor caso O(log n)
- se implementa sobre un array
- es más simple que un árbol equilibrado

El árbol binario completo

Un *árbol binario completo* está completamente lleno, excepto el nivel inferior que se rellena de izquierda a derecha

- su profundidad es O(log n)
- no necesita referencias a los hijos izquierdo y derecho
  - podemos almacenar su recorrido por niveles, en un array
  - y el número de nudos



	A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9	10

El árbol binario completo (cont.)

Es esta estructura, para el elemento `i`:

- El hijo izquierdo está en `2*i`
- El hijo derecho en `2*i+1`
  - si el valor sobrepasa el número de nudos (num), ello indica que ese hijo no existe
- El padre está en `i div 2`

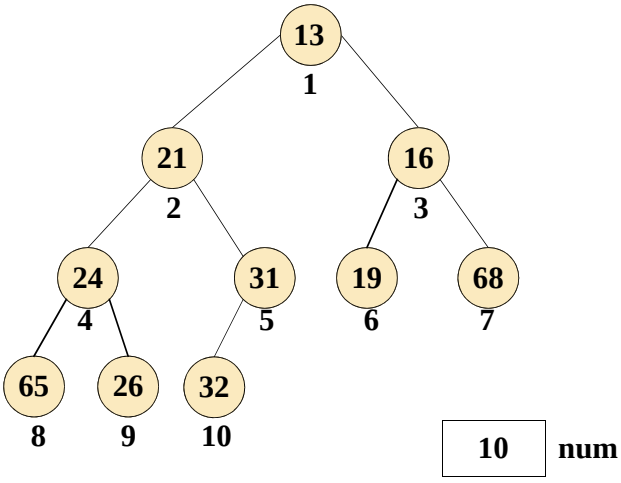
En la posición `0` situaremos un nudo "centinela"

- un padre falso para la raíz
- facilita la implementación

El montículo binario

Un montículo binario es un árbol binario completo ordenado

- el contenido del padre siempre es menor que el de sus hijos
- el menor elemento siempre es la raíz
  - buscar el mínimo es siempre O(1)



	13	21	16	24	31	19	68	65	26	32
0	1	2	3	4	5	6	7	8	9	10

Inserción en montículos binarios

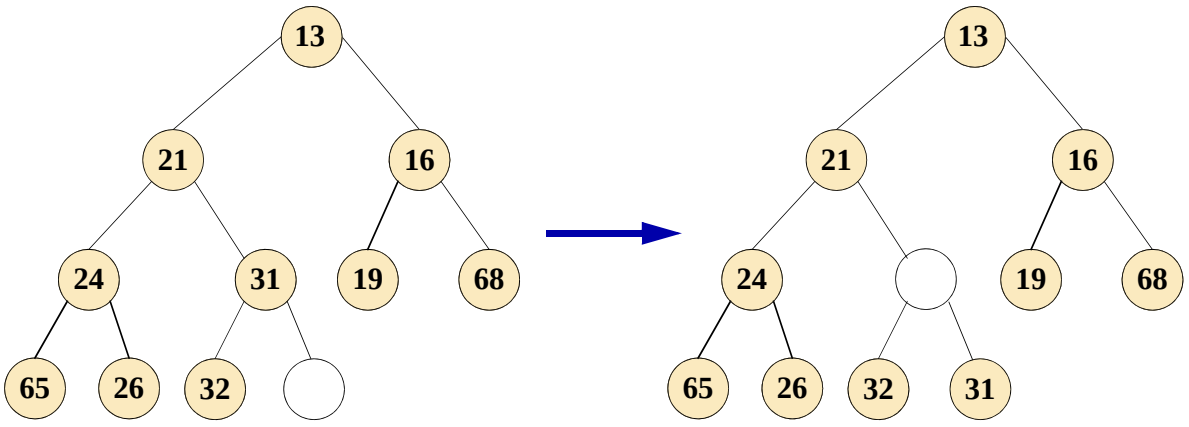
Al añadir un nudo al árbol debemos hacerlo en el siguiente hueco disponible

En caso de que el elemento no quedase bien ordenado, debemos desplazar el hueco

- **lo hacemos intercambiándolo con el padre del hueco**
- **repetimos este paso sucesivas veces hasta alcanzar la relación de orden**
- **a este proceso lo llamamos *reflotamiento***

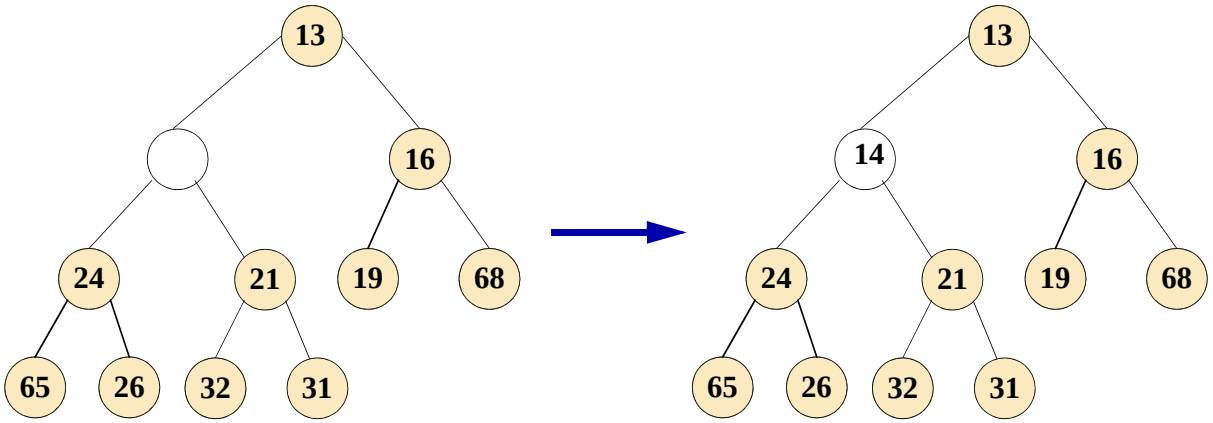
Ejemplo de inserción (paso 1)

Queremos meter el elemento 14; reflotamos el hueco



Ejemplo de inserción (pasos 2 y 3)

Queremos meter el elemento 14; reflotamos el hueco



En promedio se requieren sólo 2.6 comparaciones para insertar

Eliminación en montículos binarios

Queremos eliminar el elemento mínimo, es decir, la raíz

- allí queda un hueco

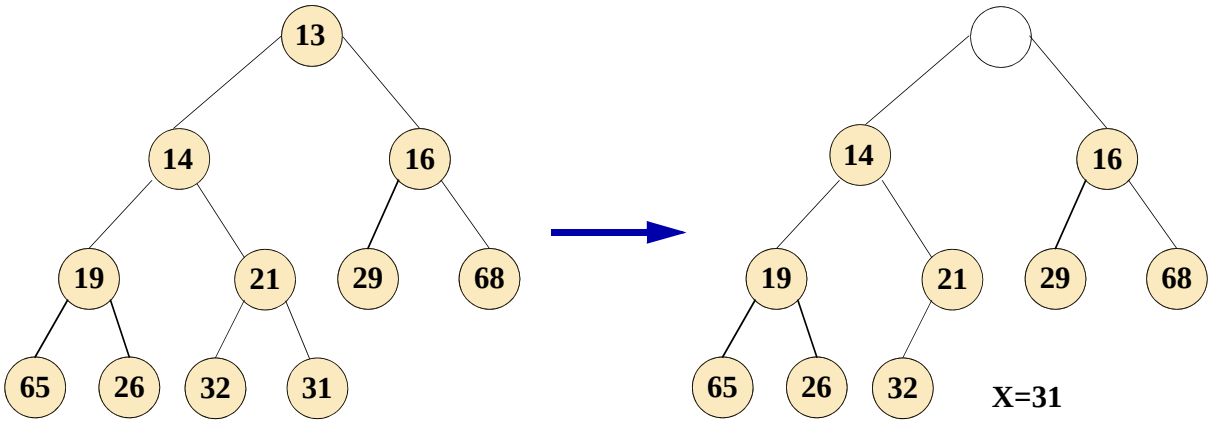
Hay que recolocar el elemento X que ocupa la última posición

Hay que trasladar el hueco hasta una posición adecuada para X

- elegimos el hijo más pequeño del hueco y lo movemos hacia arriba
- repetimos este proceso hasta encontrar una posición apropiada para X
- llamamos a este proceso *hundimiento*

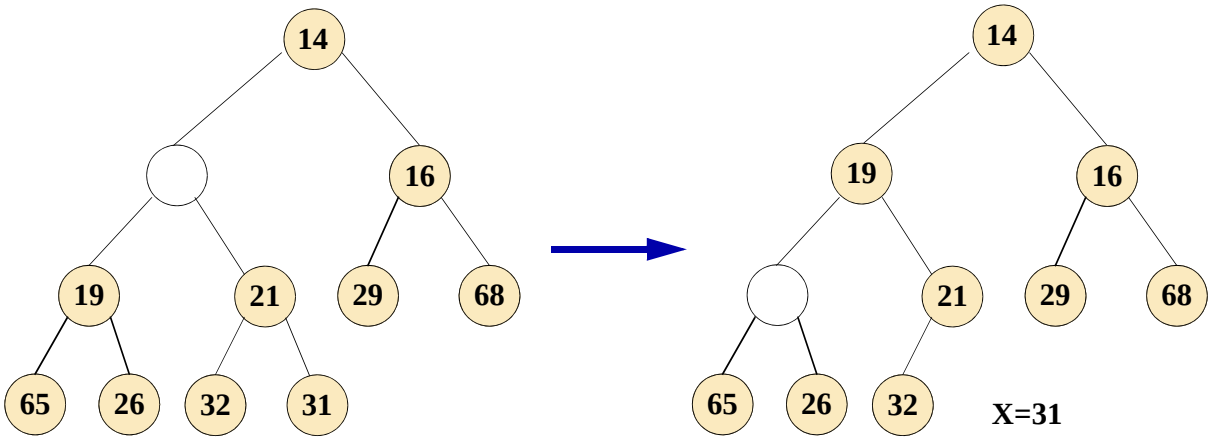
Ejemplo de eliminación (paso 1)

Queremos eliminar la raíz, y recolocar X=31



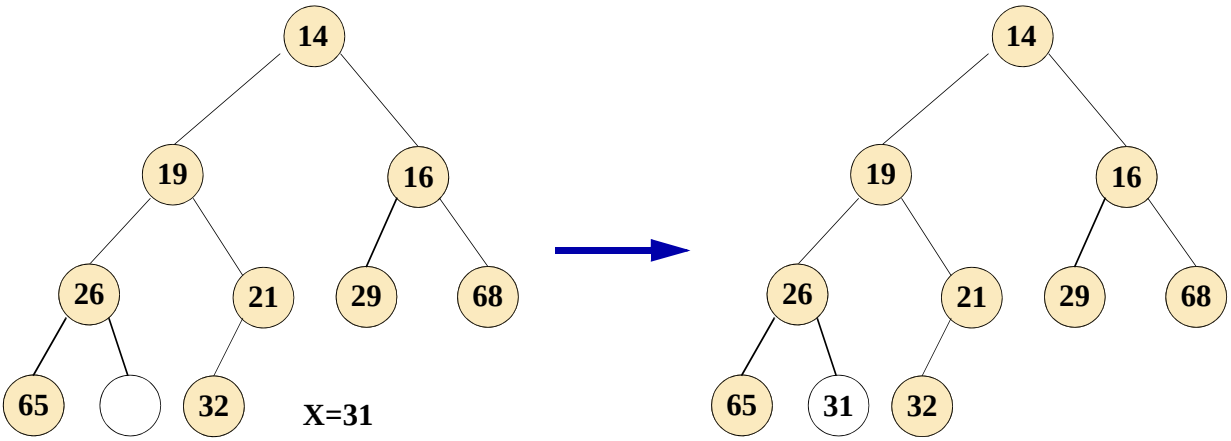
Ejemplo de eliminación (pasos 2 y 3)

Hundimos el hueco



Ejemplo de eliminación (pasos 4 y 5)

Seguimos hundiendo el hueco y colocamos X



Esta operación es O(log n) en el peor caso y en promedio

Implementación Java de los montículos binarios

A continuación se muestra una implementación Java mínima de la cola de prioridad usando los montículos binarios

Implementa las operaciones

- insertar
- eliminar el menor elemento
- mirar el primer elemento, sin borrarlo
- saber si está vacía
- hacer nula

Implementación Java de los montículos binarios (cont.)

```
public class MonticuloBinario
  <E extends Comparable<E>>
{
  // atributos de la clase
  private Object[] tabla;
  private int num;

  /**
   * Constructor al que se le pasa el tamano
   * y un valor menor que todos los demás*/
  public MonticuloBinario(int tamano, E menosInfin) {
    tabla = new Object[tamano];
    tabla[0]=menosInfin;
    num=0;
  }
```


Implementación Java de los montículos binarios (cont.)

```
  /**
   * Inserta un nuevo elemento en el montículo
   */
  public void inserta(E elem) throws NoCabe {
    // comprobar si cabe
    if (num==tabla.length-1) {
      throw new NoCabe();
      // otra alternativa sería recrear la tabla
      // haciéndola más grande
    }
```

Implementación Java de los montículos binarios (cont.)

```
    // insertar el elemento nuevo
    num++;
    int hueco=num;
    // reflotar comparando con el padre
    while(elem.compareTo((E)tabla[hueco/2])<0) {
      tabla[hueco]=tabla[hueco/2];
      hueco=hueco/2;
    }
    tabla[hueco]=elem;
  }
```


Implementación Java de los montículos binarios (cont.)

```
  /**
   * Elimina el elemento menor, y lo retorna
   */
  public E elimina() throws NoHay {
    E borrado=primero();
    E x=(E) tabla[num];
    num--;
    int hueco=1;

    // hundir el hueco
    boolean acabar=false;
    while (! acabar && hueco*2<=num) {
      int hijo=hueco*2;
      // buscar el hijo menor
```

Implementación Java de los montículos binarios (cont.)

```
      if (hijo!=num && ((E)tabla[hijo+1]).
        compareTo((E)tabla[hijo])<0)
      {
        hijo++;
      }
      if (((E)tabla[hijo]).compareTo(x)<0) {
        tabla[hueco]=tabla[hijo];
        hueco=hijo;
      } else {
        acabar=true;
      }
    }
    tabla[hueco]=x;
    return borrado;
  }
```


Implementación Java de los montículos binarios (cont.)

```
  /**
   * Retorna el elemento mínimo sin borrarlo
   */
  public E primero() throws NoHay {
    if (estaVacio()) {
      throw new NoHay();
    } else {
      return (E) tabla[1];
    }
  }
```

Implementación Java de los montículos binarios (cont.)

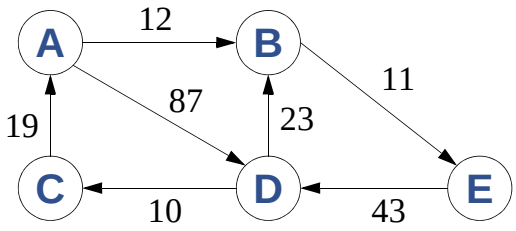
```
   /**
    * Indica si el montículo esta vacio
    */
   public boolean estaVacio() {
     return num==0;
   }

   /**
    * Vacía el montículo
    */
   public void hazNulo() {
     num=0;
   }
}
```


6.7. Grafos

Un *grafo* es una estructura de datos que almacena datos de dos tipos:

- *vértices* o nudos, con un valor almacenado
- *aristas* o arcos: cada una conecta a un vértice con otro, y puede tener un valor almacenado
  - una arista es un par de vértices (*v*,*w*)
  - si el par está ordenado, se dice que el grafo es *dirigido* o que es un *digrafo*



Elementos de la implementación de un grafo

Necesitamos transformar los contenidos de los vértices a identificadores de vértice

- deben ser consecutivos, para poder guardar los vértices en un array

Necesitamos también transformar el identificador a contenido de vértice

Esta estructura se llama *tabla de símbolos*

- para cada nuevo valor del vértice, halla un número entero aún no utilizado
  - el último número usado se guarda en una variable

Elementos de la implementación de un grafo

La transformación de vértice a identificador se guarda en un *mapa* que relaciona vértices con números enteros

- así la transformación es O(1) en promedio

La transformación identificador a vértice se guarda en un vector (`ArrayList`)

- así la transformación es O(1)

Elementos de la implementación de un grafo (cont)

Los vértices del grafo se guardan en una tabla

- el *índice* es el identificador de vértice obtenido de la tabla de símbolos
- la tabla se puede hacer de tamaño variable, con un vector (`ArrayList`)

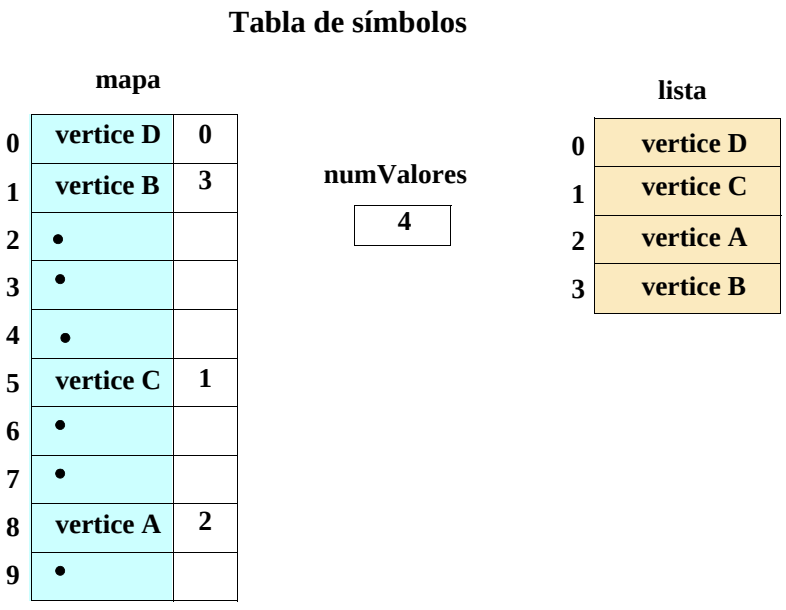
En cada vértice hay que guardar la lista de adyacencia:

- una lista de aristas
- puede ser secuencial (`LinkedList`, lista enlazada simple, o un vector (`ArrayList`)

Por tanto necesitamos un *vector de listas*

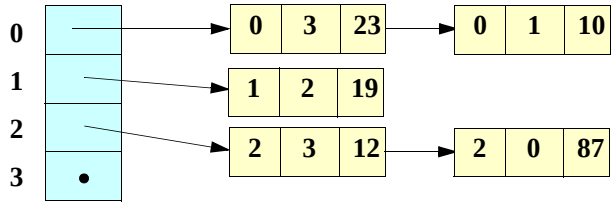
- y además el número de aristas

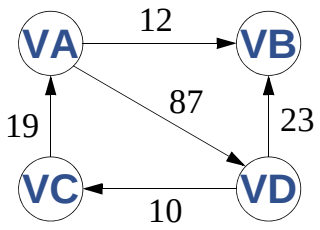
Elementos de la implementación de un grafo (cont)



Elementos de la implementación de un grafo (cont)

Vector de vértices, con listas de aristas





Implementación de grafos: la clase Arista

```
package adts;
public class Arista<A>
{
  private int origen, destino;
  private A contenido;

  public Arista (int origen, int destino, A contenido)
  {
    this.origen=origen;
    this.destino=destino;
    this.contenido=contenido;
  }
```

Implementación de grafos: la clase Arista (cont.)

```
  public int destino() {
    return destino;
  }

  public int origen() {
    return origen;
  }

  public A contenido() {
    return contenido;
  }
}
```


Implementación de grafos: tabla de símbolos

```
package adts;
import java.util.*;
/**
 * Clase que almacena una tabla de identificadores
 * enteros para valores de la clase E
 */
public class TablaIds<E> {
  // mapa que relaciona objetos de la clase E
  // con identificadores enteros
  private Map<E,Integer> mapa=
      new HashMap<E,Integer>();
  // Lista posicional que relaciona
  // identificadores con valores
  private ArrayList<E> lista=new ArrayList<E>();
  private int numValores=0;
```

Implementación de grafos: tabla de símbolos (cont.)

```
/** Retorna el identificador asociado a un valor;
 *  si no esta en la tabla lo anade */
public int createOrGetId(E valor) {
  Integer id=mapa.get(valor);
  if (id==null) {
    // anadir el valor a la tabla
    int ident=numValores;
    numValores++;
    mapa.put(valor,new Integer(ident));
    lista.add(valor);
    return ident;
  } else {
    return id.intValue();
  }
}
```


Implementación de grafos: tabla de símbolos (cont.)

```
public int getId(E valor) throws NoExiste {
  Integer id=mapa.get(valor);
  if (id != null) {
    return id;
  } else {
    throw new NoExiste();
  }
}

/**
 * Retorna el numero de valores almacenados
 */
public int numValores() {
  return numValores;
}
```

Implementación de grafos: tabla de símbolos (cont.)

```
  /**
   * Obtiene el valor a partir del identificador;
   * retorna IdIncorrecto si no existe
   */
  public E valor(int id) throws IdIncorrecto{
    try {
      return lista.get(id);
    } catch (IndexOutOfBoundsException e) {
      throw new IdIncorrecto();
    }
  }
}
```


Implementación del grafo mediante lista de adyacencia

```
package adts;
import java.util.*;
/**
 * Clase que implementa un grafo realizado con
 * listas de adyacencia
 */
public class GrafoAdyacencia<V,A>
   implements Grafo<V,A>
{
  // objeto que almacena la tabla de identificadores
  private TablaIds<V> tabla;
  // lista que almacena las listas de adyacencia
  private ArrayList<List<Arista<A>>>  vertices;
  // numero de aristas
  private int numAristas;
```

Implementación del grafo mediante lista de adyacencia (cont.)

```
/**
 * Constructor, que crea el grafo vacío
 */
public GrafoAdyacencia() {
  tabla=new TablaIds<V>();
  vertices = new ArrayList<List<Arista<A>>>();
  numAristas=0;
}

/**
 * Insertar una nueva arista con peso a partir de
 * las descripciones de sus vertices. Si los
 * vertices son nuevos, los anade al grafo
 */
```


Implementación del grafo mediante lista de adyacencia (cont.)

```
public Arista<A> nuevaArista
  (E vertice1, E vertice2, A contenidoArista)
{
  int idOrigen=tabla.createOrGetId(vertice1);
  int idDestino=tabla.createOrGetId(vertice2);

  // añadir los vértices necesarios
  int idMax=Math.max(idOrigen, idDestino);
  if (idMax>=vertices.size()) {
    for (int i=vertices.size(); i<=idMax; i++) {
      vertices.add(new ArrayList<Arista<A>>());
    }
  }
```

Implementación del grafo mediante lista de adyacencia (cont.)

```
    // añadir la arista a la lista de adyacencia
    Arista<A> a= new Arista<A>
       (idOrigen, idDestino, contenidoArista);
    List<Arista<A>> ady;
    ady=vertices.get(idOrigen);
    ady.add(a);
    numAristas++;
    return a;
  }
```


Implementación del grafo mediante lista de adyacencia (cont.)

```
/**
 * Retorna el identificador del vertice indicado.
 * Lanza NoExiste si el vertice no pertenece al grafo
 */
public int idVertice(V vertice) throws NoExiste {
  return tabla.getId(vertice);
}

/**
 * Retorna el numero de vertices
 */
public int numVertices() {
  return vertices.size();
}
```

Implementación del grafo mediante lista de adyacencia (cont.)

```
  /**
   * Retorna el contenido del vertice cuyo
   * identificador se indica. Lanza IdIncorrecto si
   * ese identificador no pertenece al grafo */
  public V contenido(int idVertice)
    throws IdIncorrecto
  {
    return tabla.valor(idVertice);
  }

  /**
   * Retorna el numero de aristas */
  public int numAristas() {
    return numAristas;
  }
```


Implementación del grafo mediante lista de adyacencia (cont.)

```
  /**
   * Retorna la lista de aristas del vertice de
   * identificador idVertice. Lanza IdIncorrecto si
   * ese identificador no pertenece al grafo
   */
  public List<Arista<A>> listaAristas(int idVertice)
    throws IdIncorrecto
  {
    try {
      return vertices.get(idVertice);
    } catch (IndexOutOfBoundsException e) {
      throw new IdIncorrecto();
    }
  }
}
```