Tema 11- Implementación de Cola de Prioridad y Diccionario Mediante un Árbol Binario de Búsqueda

Germán Moltó
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Tema 11- Implementación de Cola de Prioridad y Diccionario Mediante un ABB

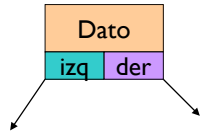
Índice general:

Objetivos

- Estudiar la implementación de un Árbol Binario de Búsqueda en Java.
- Analizar las principales operaciones de un ABB.
- Conocer y calcular el coste de las operaciones de un ABB.
- Estudiar la implementación de los modelos de Cola de Prioridad y de Diccionario utilizando un ABB.
- Analizar el problema de la selección mediante una perspectiva basada en ABBs.

Árbol Binario de Búsqueda: Representación Enlazada

- Representamos un Árbol Binario mediante un enlace a su Nodo Raíz.
  - Cada Nodo del Árbol Binario se corresponderá con un objeto de la clase **NodoABB**.



- Los Nodos de Árbol Binario requieren dos enlaces, tantos como hijos tiene el Nodo representado.
  - Recuerda que los nodos de una Lista Enlazada únicamente contenían un enlace al nodo siguiente.

La clase NodoABB

```
package librerias.estructurasDeDatos.jerarquicos;
class NodoABB<E> {
  E dato;
  NodoABB<E> izq, der;
  int tamanyo;
  NodoABB(E dato, NodoABB<E> izq, NodoABB<E> der){
    this.dato = dato; izq  = izquierdo;  der  = derecho;
    this.tamanyo = 1;
    if (izq!=null) tamanyo+=izq.tamanyo; if (der!=null) tamanyo+= der.tamanyo;
  }
  NodoABB(E dato){
    this(dato, null, null);
  }}
```


Introducción a la Clase ABB (1/2)

- La clase **ABB** puede utilizarse posteriormente para implementar los modelos **Diccionario** y **ColaPrioridad**.
- La clase ABB implementa tres estrategias de inserción diferente, que pueden ser utilizadas, posteriormente, por los modelos de Diccionario y Cola de Prioridad:
  - Insertar sin duplicados, Insertar con duplicados, Actualizar.

ABB<E extends Comparable<E>>
protected NodoABB<E> **insertarSinDuplicados**(E x, NodoABB<E> actual) throws ED
protected NodoABB<E> **insertarConDuplicados**(E x, NodoABB<E> actual)
protected NodoABB<E> **actualizar**(E x, NodoABB<E> actual)
protected NodoABB<E> **eliminar**(E x, NodoABB<E> actual) throws ENE
protected NodoABB<E>***recuperar***(E x, NodoABB<E> actual)

La clase ABB (1/4)

```
package librerias.estructurasDeDatos.jerarquicos;
import librerias.excepciones.*;
public class ABB<E extends Comparable<E>> {
  protected NodoABB<E> raiz;
  protected long numTotalInserciones, numTotalComparaciones;
  private boolean seHaEliminado;
  public ABB(){
    raiz = null; numTotalInserciones = 0;
    numTotalComparaciones = 0; seHaEliminado = false;
  }
 public ABB(E x){
    this();  raiz = new NodoABB<E>(x);
    numTotalInserciones = 1; numTotalComparaciones = 1; }
```


La clase ABB (2/4)

```
public E recuperar(E x) throws ElementoNoEncontrado{ ... }
private NodoABB<E> recuperar(E x, NodoABB<E> n){ ... }

public E recuperarMin(){...}
protected NodoABB<E> recuperarMin(NodoABB<E> actual){ ... }

public void insertarSinDuplicados(E x) throws ElementoDuplicado { ... }
protected NodoABB<E> insertarSinDuplicados(E x, NodoABB<E> actual)
   throws ElementoDuplicado{  ... }

public void insertarConDuplicados(E x) { ... }
protected NodoABB<E> insertarConDuplicados(E x, NodoABB<E>
   actual){ ... }
```

- Fíjate en la utilización de métodos via o lanzadera.

La clase ABB (3/4)

```
public void actualizar(E x) { … }
protected NodoABB<E> actualizar(E x, NodoABB<E> actual){ ... }

public void eliminar(E x) throws ElementoNoEncontrado{ ... }
protected NodoABB<E> eliminar(E x, NodoABB<E> actual) throws
   ElementoNoEncontrado{ ... }

public  E eliminarMin(){…}
protected NodoABB<E> eliminarMin(NodoABB<E> actual){ … }
public int tamanyo(){ … }
public int altura(){ … }
protected int altura(NodoABB<E> actual){ … }
public boolean esVacio() { … }
```


La clase ABB (4/4)

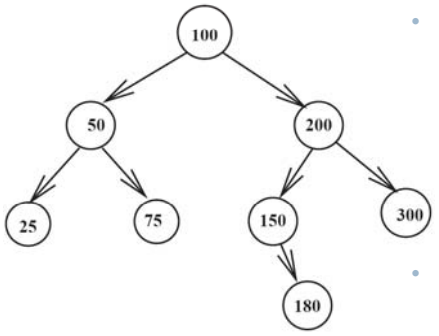
```
public String toStringPostOrden(){  … }
protected String toStringPostOrden(NodoABB<E> actual){ … }
public String toStringPreOrden(){  … }
protected String toStringPreOrden(NodoABB<E> actual){  … }
public String toStringInOrden(){  … }
protected String toStringInOrden(NodoABB<E> actual){…}
public String toStringPorNiveles(){  …}
protected String toStringPorNiveles(NodoABB<E> actual){ …}

public ListaConPI<E> toLPI() { … }
protected void toLPI(NodoABB<E> actual, ListaConPI<E> l){ ... }
public double eMC(){ …}
public double eMCOptimo(){ …}
} /* Fin de la clase ABB */
```


Sobre la Clase ABB

- Es posible insertar en un ABB de tres formas diferentes:
  - Insertar x EXCEPTO si está repetido (**insertarSinDuplicados**)
    - Si x no pertenece al ABB lo inserta, pero si ya estaba entonces se lanza **ElementoDuplicado**.
  - Insertar x INCLUSO si está repetido (**insertarConDuplicados**).
    - Es la estrategia a utilizar al implementar una Cola de Prioridad usando un ABB (ABBColaPrioridad) donde sí se permite la existencia de elementos repetidos.
  - Actualizar x (**actualizar**).
    - Si x no pertenece al ABB lo inserta, pero si ya estaba entonces se actualiza el nodo con el nuevo valor.
    - Puede utilizarse para actualizar de forma eficiente una entrada de un ABBDiccionario.

Diseño Recursivo de *recuperar* en un ABB (I)

- La propiedad de ordenación del árbol permite realizar una búsqueda guiada eficiente sobre el ABB.



- Dado que los objetos almacenados en cada NodoABB<E> del ABB implementan la interfaz **Comparable<E>**, es posible continuar la búsqueda por un subárbol o por otro.
- La búsqueda de un objeto a partir de un NodoABB<E> *actual* define la raíz del ABB donde se va a realizar la búsqueda.

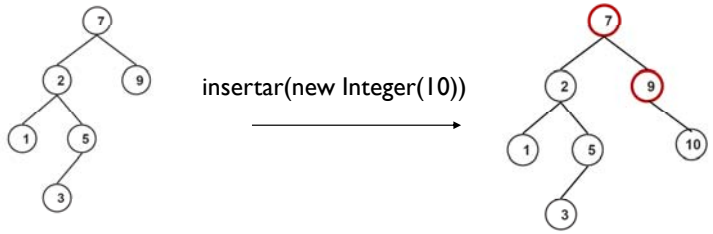
Diseño Recursivo de *recuperar* en un ABB (II)

```
protected NodoABB<E> recuperar(E x, NodoABB<E> n){
    NodoABB<E> res = n;
    if ( n != null ){
        int resC = n.dato.compareTo(x);
        if      ( resC < 0 )   res = recuperar(x, n.der);
        else  if ( resC > 0 )  res = recuperar(x, n.izq);
    }
    return res;
 }
public E recuperar(E x) throws ElementoNoEncontrado{
     NodoABB<E> res = recuperar(x, this.raiz);
     if ( res == null )
        throw new ElementoNoEncontrado("El dato "+x+" no está");
     return res.dato;
  }
```

¿Cuál es la complejidad temporal asintótica del método?

Inserción en un ABB

- La Inserción en un ABB requiere recuperar el lugar de inserción y crear una nueva hoja en el caso de que no esté el elemento.



- La inserción en un ABB requiere un coste temporal, en el peor de los casos, **lineal con la altura del árbol**, similar al de la búsqueda:
  - Para un árbol equilibrado: $\Theta(\log_2(N))$
  - Para un árbol degenerado: $\Theta(N)$

El Método *insertarSinDuplicados*

```
protected NodoABB<E> insertarSinDuplicados(E x, NodoABB<E> actual) throws
   ElementoDuplicado{
    NodoABB<E> res = actual;
    if ( actual == null ){
        numTotalInserciones++;
        res = new NodoABB<E>(x);
    }else{
        int resC =  actual.dato.compareTo(x);
        if ( resC == 0 ) throw new ElementoDuplicado(x +" está duplicado");
        if ( resC < 0 ) res.der = insertarSinDuplicados(x, actual.der);
        else            res.izq = insertarSinDuplicados(x, actual.izq);
        actual.tamanyo++;
        numTotalComparaciones++;
    }
    return res; }
```

¿Por qué se actualiza numTotalComparaciones y el tamaño del nodo después de realizar las llamadas recursivas? ¿Se podría hacer antes?

El Método *insertarConDuplicados*

```
protected NodoABB<E> insertarConDuplicados(E x, NodoABB<E> actual){
    NodoABB<E> res = actual;
    if ( actual == null ){
        numTotalInserciones++;
        res = new NodoABB<E>(x);
    }
    else{
        int resC = actual.dato.compareTo(x);
        numTotalComparaciones++;
        actual.tamanyo++;
       if ( resC < 0 ) res.der = insertarConDuplicados(x, actual.der);
        else            res.izq = insertarConDuplicados(x, actual.izq);
    }
    return res;  }
```

El Método Actualizar

```
protected NodoABB<E> actualizar(E x, NodoABB<E> actual){
  NodoABB<E> res = actual;
  if ( actual == null ){numTotalInserciones++; res = new NodoABB<E>(x); }
  else{
        int resC = (actual.dato).compareTo(x);
        if ( resC == 0 ) res.dato = x;
        else{
           int tamanyoHijosActual = tamanyo(actual)-1;
           if ( resC < 0 ) res.der = actualizar(x, actual.der);
           else            res.izq = actualizar(x, actual.izq);
           if( (tamanyo(res.izq)+tamanyo(res.der)) != tamanyoHijosActual ){
              numTotalComparaciones++;   actual.tamanyo++;
           } }}
   return res;}
```

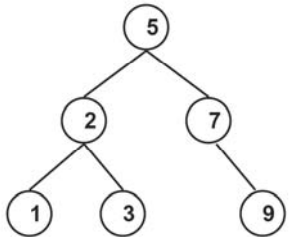

Métodos vía o lanzadera de inserción en ABB

- Los métodos vía o lanzadera simplemente delegan en los correspondientes métodos homónimos para trabajar a partir de la raíz del árbol.

```
public void insertarSinDuplicados(E x) throws ElementoDuplicado {
    this.raiz = insertarSinDuplicados(x, raiz);
 }
 public void insertarConDuplicados(E x) {
    this.raiz = insertarConDuplicados(x, raiz);
 }
 public void actualizar(E x) {
    this.raiz = actualizar(x, raiz);
 }
```


Búsqueda del Mínimo y del Máximo en un ABB

- ¿Cómo se realiza la búsqueda del **máximo** elemento de un ABB?, ¿Qué coste tiene?, ¿Presenta instancias significativas?



- La búsqueda del Dato menor o mayor del ABB requiere el recorrido de una única rama del ABB

- La búsqueda del dato máximo o mínimo en un ABB requiere un coste temporal **lineal con la altura del árbol**, es decir, un coste similar al de la búsqueda: $\theta(\log_2(N))$, para un árbol equilibrado.

Diseño de *recuperarMin*

- El mínimo elemento en un ABB está situado lo más a la izquierda posible. Implementación iterativa:

```
protected NodoABB<E> recuperarMin(NodoABB<E> actual) {
     while (actual.izq != null) actual = actual.izq;
     return actual;
  }

public E recuperarMin() {
     return recuperarMin(this.raiz).dato;
  }
```

¿Qué devuelve el método si es invocado con un actual = null?

Diseño de *eliminarMin*

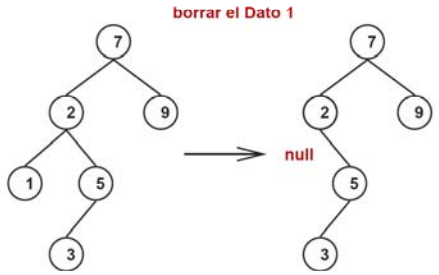
- Borrado del menor elemento del ABB cuya raíz es el NodoABB<E> *actual*. Implementación recursiva:

```
protected NodoABB<E> eliminarMin(NodoABB<E> actual) {
      if ( actual.izq != null ) {
         actual.tamanyo--;  actual.izq = eliminarMin(actual.izq);
       } else {actual = actual.der; seHaEliminado=true;}
      return actual;
}
public  E eliminarMin() {
    E min = recuperarMin();
     this.raiz = eliminarMin(this.raiz);
     return min;
}
```


Borrado en un ABB (I)

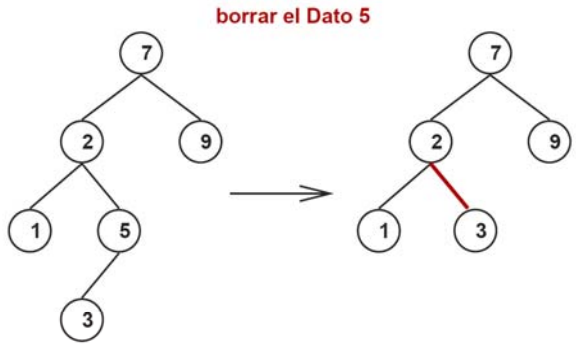
- Borrar un elemento de un ABB implica primero recuperarlo y, tras encontrarlo, eliminarlo. Se pueden dar tres casos diferentes:

1. El dato está en una **Hoja**: Simplemente se elimina el nodo.



Borrado en un ABB (II)

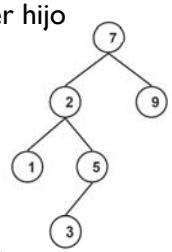
2. El dato está en un nodo que **solo tiene un hijo** (izquierdo o derecho): El padre del nodo a borrar cambia de hijo para tener el hijo único del nodo que se elimina.



Borrado en un ABB (III)

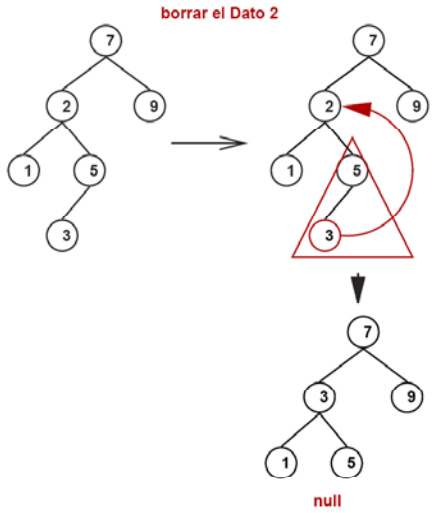
3. El dato está en un nodo que **tiene dos hijos**:
   - Se substituye el dato del nodo a borrar por su **sucesor** y después se elimina este último nodo.

- El sucesor es el menor dato de todos aquellos que son mayores que el del nodo a borrar.
  - Para este caso particular, es el menor dato del subárbol derecho. El nodo que lo contiene no puede tener hijo izquierdo porque sino éste sería aún menor.



- Ejemplo de cálculo de sucesor:
  - sucesor(7) → 9
  - sucesor(2) → 3
  - sucesor(1) → 2

Borrado en un ABB (IV)



- Sustituir el dato del nodo a borrar por el de su sucesor.
- Eliminar el nodo correspondiente al sucesor.

Diseño Recursivo de *eliminar* en un ABB (I)

- Eliminar un elemento supone, primero recuperar el dato y después borrar el NodoABB<E> al que pertenece.
  - El método protegido lanzará la excepción cuando detecte que el elemento a borrar no está en el árbol.

```
public void eliminar(E x) throws ElementoNoEncontrado{
    this.raiz = eliminar(x, this.raiz);
 }
```

protected NodoABB<E> **eliminar**(E x, NodoABB<E> actual) throws ElementoNoEncontrado;

- Elimina el objeto *x* en el ABB del cual *actual* es raíz. Si el elemento no existe, se lanza la excepción ElementoNoEncontrado.

Diseño Recursivo de *eliminar* en un ABB (II)

```
protected NodoABB<E> eliminar(E x, NodoABB<E> actual) throws ElementoNoEncontrado{
    NodoABB<E> res = actual;
    if ( actual == null ) throw new ElementoNoEncontrado("El dato "+x+" no esta");
    int resC = actual.dato.compareTo(x);
    if       ( resC < 0 )  res.der = eliminar(x, actual.der);
    else   if ( resC > 0 )  res.izq = eliminar(x, actual.izq);
           else{
              if ( actual.izq != null  && actual.der != null ){
                  res.dato = recuperarMin(actual.der).dato;
                  res.der =  eliminarMin(actual.der);
              }
              else res = ( actual.izq != null ) ? actual.izq: actual.der;
              this.seHaEliminado = true;
              }
     actual.tamanyo--;
     return res;    }
```


Recorridos de un ABB (I)

```
public String toStringPostOrden(){
    if ( this.raiz != null ) return toStringPostOrden(this.raiz);
    else return "*";
  }
protected String toStringPostOrden(NodoABB<E> actual){
      String res = "";
      if( actual.izq != null ) res += toStringPostOrden(actual.izq);
      if( actual.der != null ) res += toStringPostOrden(actual.der);
      res += actual.dato.toString()+"\n";
      return res;
   }
```

Recorridos de un ABB (II)

```
public String toStringInOrden(){
      if ( this.raiz != null )  return toStringInOrden(this.raiz);  else return "*";
 }
protected String toStringInOrden(NodoABB<E> actual){
       String res = "";
       if( actual.izq != null ) res += toStringInOrden(actual.izq);
       res += actual.dato.toString()+"\n";
       if( actual.der != null ) res += toStringInOrden(actual.der);
       return res;
 }
public String toStringPreOrden(){
       if ( this.raiz != null )   return toStringPreOrden(this.raiz);  else return "*";
   }
protected String toStringPreOrden(NodoABB<E> actual){
       String res = actual.dato.toString()+"\n";
       if( actual.izq != null ) res += toStringPreOrden(actual.izq);
       if( actual.der != null ) res += toStringPreOrden(actual.der);
       return res; }
```


Recorridos de un ABB (III)

```
public String toStringPorNiveles(){
      if ( this.raiz != null )  return toStringPorNiveles(this.raiz);
      else return "*";
  }
protected String toStringPorNiveles(NodoABB<E> actual){
       Cola<NodoABB<E>> q = new ArrayCola<NodoABB<E>>();
       q.encolar(actual);   String res = "";
       while ( !q.esVacia() ){
              NodoABB<E> nodoActual = q.desencolar();
              res += nodoActual.dato.toString()+"\n";
              if ( nodoActual.izq != null ) q.encolar(nodoActual.izq);
              if ( nodoActual.der != null ) q.encolar(nodoActual.der);
       }
       return res;
}
```


Otros Métodos de la Clase ABB

```
public int tamanyo(){ return tamanyo(this.raiz); }
protected int tamanyo(NodoABB<E> actual){
     return (actual != null) ? actual.tamanyo : 0 ;
   }
public int altura(){ return altura(this.raiz); }
protected int altura(NodoABB<E> actual){
     if ( actual == null) return -1;
    else return ( Math.max(altura(actual.izq), altura(actual.der)) + 1 );
 }
public boolean esVacio() { return raiz == null; }
```


Obteniendo una Lista de Elementos

```
public ListaConPI<E> toLPI(){
      ListaConPI<E> l = new LEGListaConPI<E>();
      if (this.raiz != null)  toLPI(this.raiz, l);
      return l;
   }
protected void toLPI(NodoABB<E> actual, ListaConPI<E> l){
       if (actual.izq != null)  toLPI(actual.izq, l);
       l.insertar(actual.dato);
       if (actual.der != null) toLPI(actual.der, l);
   }
```

- Se construye la ListaConPI en el método público y se rellena mediante un recorrido en inorden por el árbol de manera recursiva en el método protegido. Los datos en la ListaConPI quedarán ordenados ascendentemente.

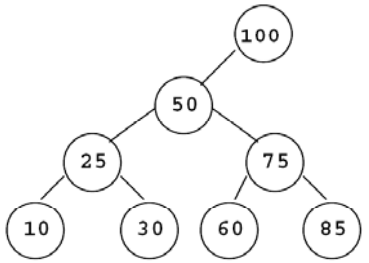
Esfuerzo Medio de Comparación

```
public double eMC(){
  if ( !seHaEliminado ) return Double.NaN;
  return numTotalComparaciones/ (double)numTotalInserciones;
}
```

- Ejercicio propuesto. Piensa en la implementación del eMC óptimo: El eMC que tendría un árbol perfectamente equilibrado con el mismo número de nodos.

Cálculo del Sucesor

- Calculo del sucesor de un nodo:
  - Si tiene subárbol derecho, el sucesor es el mínimo del subárbol derecho.
  - Sino, el sucesor es el ascendiente por la derecha más cercano.



- Cálculo de sucesor:
  - sucesor(50) → 60
  - sucesor(30) → 50
  - sucesor(100) → null
  - sucesor(85) → 100

- El sucesor de un nodo equivale al siguiente nodo que obtendría un recorrido en in-orden.

Resúmen: Operaciones sobre un ABB

- La forma concreta de un ABB con N nodos depende de la secuencia de inserciones y borrados realizados.
  - Su altura está acotada superiormente por $O(N)$ e inferiormente por $O(\log_2(N))$.
- El coste de los métodos insertar, recuperar y eliminar es, en el peor de los casos, **lineal con la altura del ABB**:
  - ABB equilibrado→ $\Theta(\log_2(N))$, ABB degenerado → $\Theta(N)$.
- Su coste se puede estimar de forma experimental contando el número de comparaciones (nodos consultados) durante la operación de inserción.
  - Asumiendo que un ABB se construye en base a inserciones de secuencias aleatorias, su altura media es logarítmica. En promedio la altura de un ABB es solo un 38% peor que en el caso mejor ($1.38 * \log_2(N)$)

Propuesta de Implementación de Diccionario usando un ABB (I)

- El modelo **Diccionario** permite realizar búsquedas por **Clave**: Obtener el valor de una ***Entrada*** del Diccionario.
  - Un Diccionario NO puede tener elementos repetidos, es decir dos objetos entrada que tengan la misma clave.
  - Colección homogénea de Entradas a las que se accede mediante una búsqueda por nombre (la clave de la entrada).

```
package modelos;
public interface Diccionario<C,V> {
        void insertar(C c,V v);
        V recuperar(C c) throws ElementoNoEncontrado;
        void eliminar(C c) throws ElementoNoEncontrado;
        boolean esVacio();
        int talla();
        ListaConPI<C> toLPIClaves();
}
```

Propuesta de Implementación de Diccionario usando un ABB (II)

```
public class EntradaDic<C extends Comparable<C>,V> implements
   Comparable<EntradaDic<C,V>>{
      protected C clave;   protected V valor;
      public EntradaDic(C c,V v) { clave = c; valor = v; }
      public EntradaDic(C c) { clave =c; valor = null; }
      public V getValor(){ return valor; }   public C getClave(){ return clave; }
      public boolean equals(Object o) {
         return this.compareTo( (EntradaDic<C,V>) o) == 0;
      }
     public int compareTo(EntradaDic<C,V> x) {
          return  this.clave.compareTo(x.getClave());
      }
      public String toString(){ return this.clave + " => " + valor;  }  }
```


Propuesta de Implementación de Diccionario usando un ABB (III)

```
public class ABBDiccionario<C extends Comparable<C>,V> implements
   Diccionario<C,V>{
  private ABB<EntradaDic<C,V>> abb;
  public ABBDiccionario(){ abb = new ABB<EntradaDic<C,V>>(); }
  public V recuperar(C c) throws ElementoNoEncontrado{
       return abb.recuperar(new EntradaDic<C,V>(c)).getValor();
   }
   public void eliminar(C c) throws ElementoNoEncontrado{
        abb.eliminar(new EntradaDic<C,V>(c));
  }
  public void insertar(C c,V v){ abb.actualizar(new EntradaDic<C,V>(c,v));}
  public boolean esVacio(){return abb.esVacio();}
```


Propuesta de Implementación de Diccionario usando un ABB (III)

```
public ListaConPI<C> toLPIClaves(){
     ListaConPI<EntradaDic<C,V>> l1 = abb.toLPI();
     ListaConPI<C> l2 = new LEGListaConPI<C>();
     for (l1.inicio(); !l1.esFin(); l1.siguiente())
          l2.insertar(l1.recuperar().getClave());
     return l2;
   }
   public int talla(){
      return abb.tamanyo();
   }
} // Fin de la clase ABBDiccionario
```


Implementación de Cola de Prioridad usando un ABB

- Cola de Prioridad: Colección de datos cuya prioridad determina el orden en el que se accede a dichos datos.
  - La prioridad de los objetos almacenados en la ColaPrioridad se especifica al implementar la interfaz Comparable<T>.

```
package librerias.estructurasDeDatos.modelos;
public interface ColaPrioridad<E extends Comparable<E>> {
        void insertar(E x);
        E recuperarMin();
        E eliminarMin();
        boolean esVacia();
   }
```

  - Precondición: Los métodos *recuperarMin*() y *eliminarMin*() se tienen que aplicar sobre ColaPrioridad no vacías.

La Clase ABBColaPrioridad

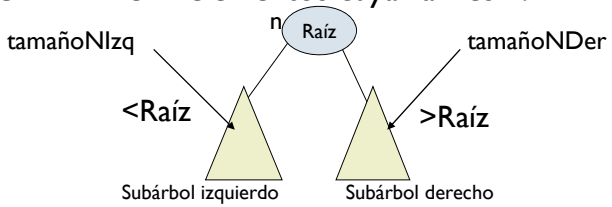
```
package librerias.estructurasDeDatos.jerarquicos;
import librerias.estructurasDeDatos.modelos.*;
import librerias.excepciones.*;
public class ABBColaPrioridad<E extends Comparable<E>>
   extends ABB<E> implements ColaPrioridad<E> {
   public ABBColaPrioridad() {
         super();
   }
//Los métodos recuperarMin y eliminarMin se reciben por herencia
public void insertar(E x){ insertarConDuplicados(x); }
public boolean esVacia(){ return esVacio(); }
} /* Fin de la clase ABBColaPrioridad */
```


El Problema de la Selección

- Dada una colección de N datos comparables, se desea buscar el k-esimo menor Dato de todos ellos.
- Hasta el momento conocemos diversas soluciones (sobre un array):
  1. Ordenación de las k primeras componentes usando una modificación del método de selección directa: $O(k*N)$
  2. Ordenación de todo el vector y acceso al dato en la posición k-1. Si empleamos QuickSort o MergeSort: $O(N*\log_2(N))$
  3. Uso del método de *Partición* sobre el array (selección rápida), estudiado en el tema 5: $O(N)$
- Objetivo: Resolver el problema de la selección empleando un ABB con un coste sublineal: $O(\log_2(N))$

El Problema de la Selección (II)

- Un ABB de N elementos cuya raíz es n:

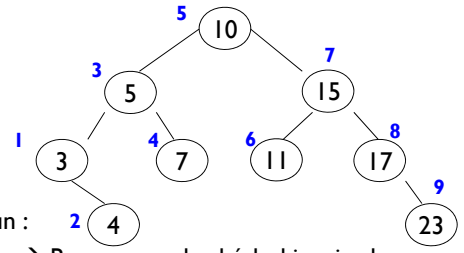


- Se cumple que N = tamañoNIzq + 1 + tamañoNDer
  - Si k == tamañoNIzq + 1, el k-ésimo menor dato es precisamente el de la raíz (n.dato)
  - Si k <= tamañoNIzq, el k-ésimo menor dato se debe buscar en el subárbol izquierdo y, por lo tanto, en n.izq.
  - Si k > tamañoNIzq + 1, el k-ésimo menor dato coincide con el (k–tamañoNIzq–1)-ésimo dato menor del subárbol derecho.

El Problema de la Selección (III)

- Árbol Binario de Búsqueda con N = 9 nodos (etiquetados con el orden que seguiría un recorrido en in-orden).

tamañoNIzq = 4



- Si queremos buscar un :
  - k <= tamañoNIzq → Buscar por el subárbol izquierdo.
  - k > tamañoNIzq + 1 → Buscar por el subárbol derecho el elemento que ocupa la posición k - tamañoNIzq – 1.
    - Ej: k = 8 → Busco el elemento que ocupa la posición 3 en el subárbol derecho.

Diseño del Método recuperarKesimo

```
protected NodoABB<E> recuperarKesimo(int k, NodoABB<E>
    n) throws ElementoNoEncontrado {
    if ( n == null) throw new ElementoNoEncontrado("Al buscar
    k-esimo");
    int tamanyoNIzq = tamanyo(n.izq);
    if ( k <= tamanyoNIzq ) return recuperarKesimo(k, n.izq);
    if ( k == tamanyoNIzq + 1) return n;
    return recuperarKesimo(k – tamanyoNIzq - 1, n.der);
 }
public E recuperarKesimo(int k) throws ElementoNoEncontrado {
    return recuperarKesimo(k,raiz).dato;
}
```

¿Cuál es la complejidad temporal de este algoritmo en el peor de los casos?

Complejidad Temporal de recuperarKesimo

- Suponemos que el ABB está equilibrado:

$$T^{P}_{buscarKesimo}(N) = \begin{cases} T^{P}_{buscarKesimo}(\frac{N}{2}) + \Theta(1) & \text{Si N > 0} \\ k & \text{Si N == 0} \end{cases}$$

- El cálculo del tamaño del subárbol izquierdo únicamente requiere un coste constante.
- Acotando con el Teorema 3 (a = 1, c = 2): $\Theta(\log_2(N))$
- Este coste mejora substancialmente a la estrategia de selección rápida.