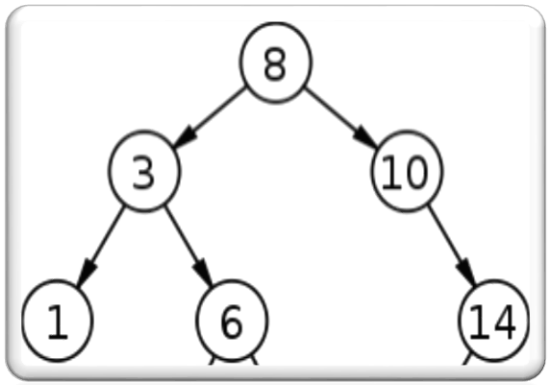
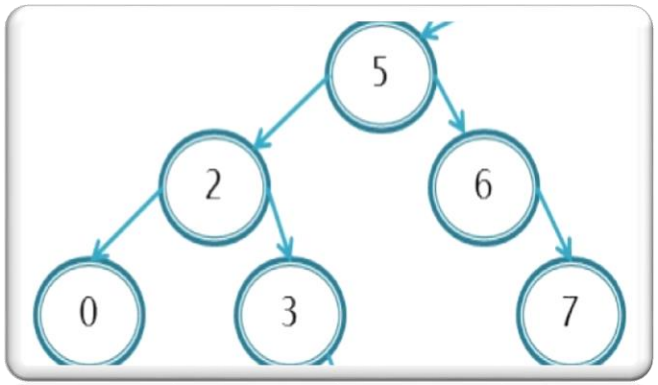
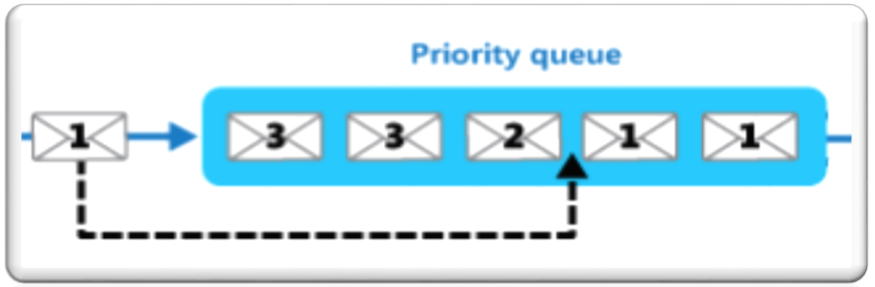
[Estructuras de Datos]







ARBOLES BINARIOS DE BÚSQUEDA.

MAPEOS Y DICCIONARIOS CON ABB.

COLAS CON PRIORIDAD.

Copyright

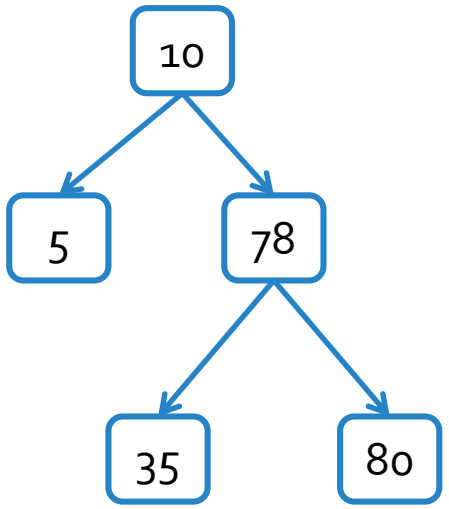
ÁRBOLES BINARIOS DE BÚSQUEDA

Introducción: ¿qué es un árbol binario de búsqueda?

- Un **árbol binario de búsqueda (ABB)** es un **ED** que permite implementar de forma **eficiente** **conjuntos**, **mapeos** y **diccionarios**.
- El **ABB** mantiene un **ordenamiento** en particular de los **elementos** que **almacena**.
- Sea **N** un **nodo** del **ABB**, luego el **ordenamiento** es tal que:
  - Los **elementos** del **subárbol izquierdo** a **N**, son **menores** que el elemento de **N**.
  - Los **elementos** del **subárbol derecho** a **N,** son **mayores** que el elemento de **N**.
- El interés de los **ABB** radica en que la **búsqueda** de un elemento suele **ser muy eficiente**, y que su recorrido **inorden** proporciona los elementos **ordenados** de forma **ascendente**.

Introducción: ¿qué es un árbol binario de búsqueda?

- Sea **N** un **nodo** del **ABB**, luego el **ordenamiento** es tal que:
  - Los **elementos** del **subárbol izquierdo** a **N**, son **menores** que el elemento de **N**.
  - Los **elementos** del **subárbol derecho** a **N,** son **mayores** que el elemento de **N**.



- *Arbol.insertar(10).*
- *Arbol.insertar(5).*
- *Arbol.insertar(78).*
- *Arbol.insertar(80).*
- *Arbol.insertar(35)*
- *Inorden: 5-10-35-78-80*

Notar que en su definición, no es considerado el hecho que dos elementos **puedan ser iguales**. Esto se debe a que esta situación **empeora** la eficiencia en las búsquedas.

ABB :: IMPLEMENTACIÓN MEDIANTE NODOS CON REF. AL PADRE e HIJOS

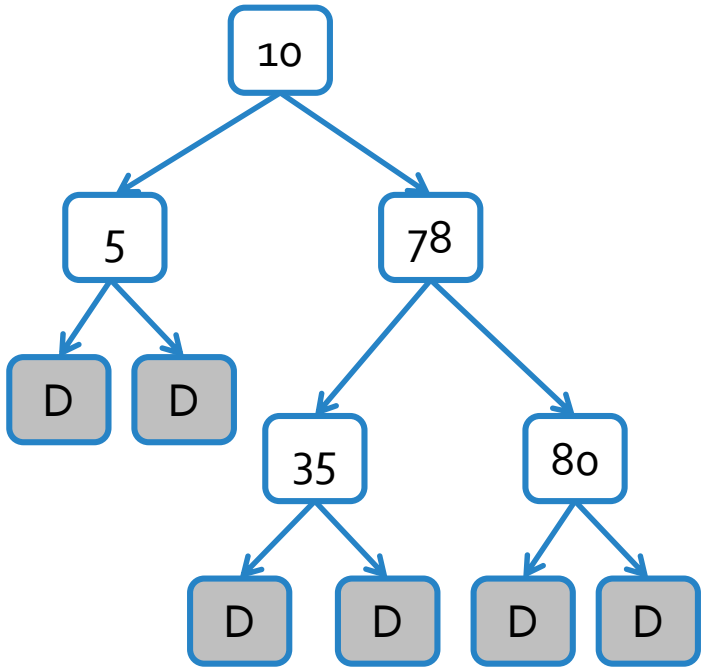
ED NodoABB

- Bajo esta implementación, se define una ED **NodoABB** que mantiene un rótulo, así como una **referencia a tres nodos** que representan los nodos **padre** e **hijo izquierdo** y **derecho**, respectivamente, del nodo modelado.

```
public class NodoABB<E> {

  protected E rotulo;
  protected NodoABB<E> padre, hi, hd;

  public NodoABB(E r, NodoABB<E> p){
    rotulo = r;
    padre = p;
  }

  public void setRotulo(E r){ rotulo = r;}
  public void setParent(NodoABB<E> p){padre = p;}
  public void setLeft(NodoABB<E> i){ hi = i;}
  public void setRight(NodoABB<E> d){hd = d;}
  public E getRotulo(){ return rotulo; }
  public NodoABB<E> getParent(){ return padre; }
  public NodoABB<E> getLeft(){ return hi;}
  public NodoABB<E> getRight(){ return hd; }
}
```

ED ABB

- Bajo esta implementación, se define una ED **ABB** que mantiene un **NodoABB** raíz, y un **comparador** de elementos, con lo que puede acceder a todo el árbol.
- El **ABB** que se implementará contemplando un **árbol propio**, donde cada **hoja** tiene dos nodos ***DUMMY*** como hijos, para facilitar la programación.



ED ABB

- Bajo esta implementación, se define una ED **ABB** que mantiene un **NodoABB** raíz, y un **comparador** de elementos, con lo que puede acceder a todo el árbol.

```
public class ABB<E extends Comparable<E>> {

  protected NodoABB<E> raiz;
  protected Comparator<E> comparador;

  public ABB(Comparator<E> comp) {
    raiz = new NodoABB<E>(null, null);
    if (comp!= null)
      comparador = comp;
    else
      comparador = new DefaultComparator<E>();
  }

  public NodoABB<E> getRoot(){…}
  public boolean insertar(E elemento){…}
  public E eliminar(E elemento){…}
  public boolean pertenece(E elemento){…}
  public NodoABB<E> buscar(E elemento){…}
}
```

*En caso de que **no se parametrice** un **comparador** de elementos E definido por el cliente, se utiliza un **comparador por defecto** que delega la tarea de comparar al tipo E, que es **comparable**.*

Permite recorrer la ED a partir de la raíz.

***Inserta** el elemento, manteniendo el orden establecido por el **ABB**. En caso de que **exista un elemento igual en el ABB**, la inserción falla y retorna false; caso contrario retorna true.*

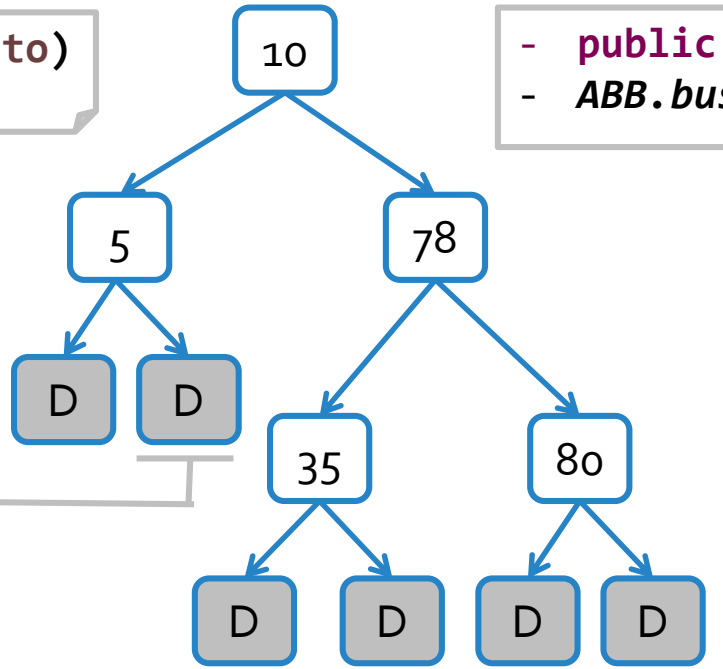
***Elimina** el elemento, manteniendo el orden establecido por el **ABB**. En caso de que **no exista el elemento en el ABB**, la eliminación falla y retorna null; caso contrario retorna el elemento eliminado.*

*Busca el **NodoABB** en el que debe almacenarse el elemento. Como el ABB contempla nodos DUMMY, siempre retorna un **NodoABB**.*

ED ABB

- Bajo esta implementación, se define una ED **ABB** que mantiene un **NodoABB** raíz, y un **comparador** de elementos, con lo que puede acceder a todo el árbol.
- El **ABB** que se implementará contemplando un **árbol propio**, donde cada **hoja** tiene dos nodos ***DUMMY*** como hijos, para facilitar la programación.

```
- public boolean insertar(E elemento)
- ABB.insertar(9)
```

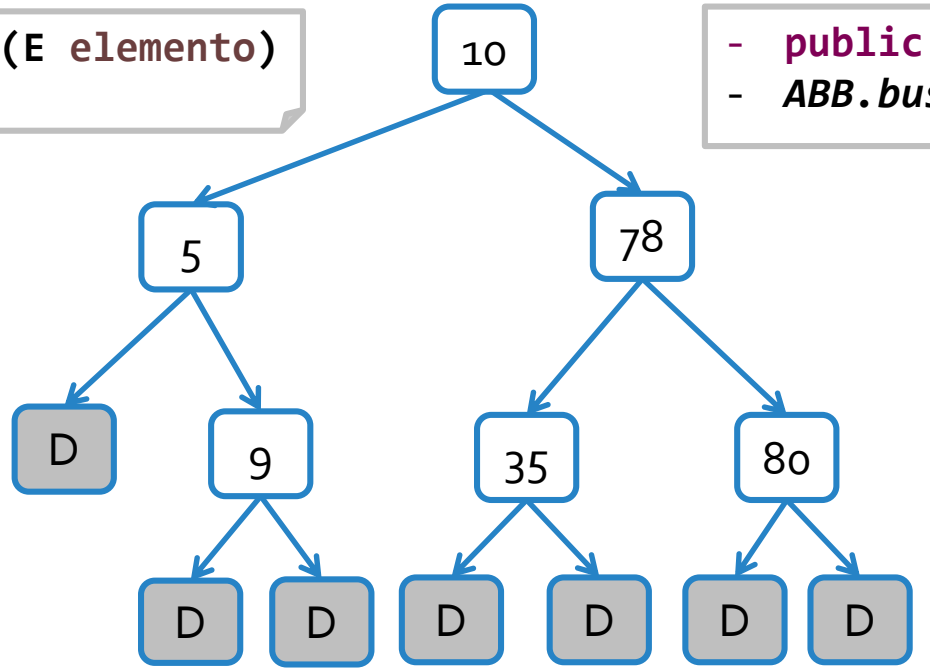



```
- public NodoABB<E> buscar(E elemento)
- ABB.buscar(85)
```

ED ABB

- Bajo esta implementación, se define una ED **ABB** que mantiene un **NodoABB** raíz, y un **comparador** de elementos, con lo que puede acceder a todo el árbol.
- El **ABB** que se implementará contemplando un **árbol propio**, donde cada **hoja** tiene dos nodos ***DUMMY*** como hijos, para facilitar la programación.

```
- public boolean insertar(E elemento)
- ABB.insertar(9)
```




```
- public NodoABB<E> buscar(E elemento)
- ABB.buscar(85)
```

MAPEO Y DICCIONARIO :: IMPLEMENTACIÓN CON ABB

ED MapeoConABB

- Bajo esta implementación, se define una ED **MapeoConABB** que mantiene un **ABB** de Entradas<K,V>. En particular, las **entradas** deben ser **comparables**.

```
public class MapeoConABB<K extends Comparable<K>,V> implements Map<K,V>{

  protected ABB<EntradaComparable<K,V>> abb;
  protected int size;

  public MapeoConABB(){
    abb = new ABB<EntradaComparable<K,V>>(null);
    size = 0;
  }
  ...
}
```

*Al utilizar un **ABB**, se debe **asegurar** que se pueda establecer **una relación de orden** entre las **claves**, por lo que el tipo K **debe ser comparable**. A diferencia de otras implementaciones de mapeo, **no alcanza** con saber cuándo **dos claves son iguales**, sino se debe saber **cuándo son menores o mayores que otras**, también.*

*Al utilizar un **ABB**, se debe **asegurar** que sus elementos son **comparables**. Como el **ABB** almacenará **Entradas<K,V>**, se deberá configurar una **clase de entradas** que sean **comparables**.*

*Como las **Entradas** son **comparables**, el **ABB** puede utilizar un **comparador** por defecto que es suficiente para estimar la relación de orden entre las **entradas** con las que se trabaje.*

ED DiccionarioConABB

- Bajo esta implementación, se define una ED **DiccionarioConABB** que mantiene un **ABB** de Entradas<K,PositionList<V>>. En particular, las **entradas** deben ser **comparables**.

```
public class DiccionarioConABB<K extends Comparable<K>,V> implements Dictionary<K,V>{

  protected ABB<EntradaComparable<K,PositionList<V>>> abb;
  protected int size;

  public DiccionarioConABB(){
    abb = new ABB<EntradaComparable<K,PositionList<V>>>(null);
    size = 0;
  }
  ...
}
```

*Ante **entradas** con **igual clave**, en lugar de **modificar** el valor de la **entrada**, se debe considerar **almacenar** una **lista de valores** correspondientes a dicha clave.*

COLAS CON PRIORIDAD

Introducción: ¿qué es un cola con prioridad?

- Una **cola con prioridad (CPP)** es un colección de elementos, llamados valores, los cuales tienen **asociada una prioridad** que es **provista** en el momento que el elemento es **insertado**.
- Las **prioridades** de cada elemento **insertado** establecen un **orden total**, a partir del cual los **valores** son **ordenados**.
- Operaciones fundamentales de una **CCP**.
  - **insertar(Clave,Valor):** Inserta un valor ***valor*** con prioridad ***clave*** en la **CCP**.
  - **removerMinimo():** Retorna y remueve de la **CCP** una **entrada** con la prioridad más pequeña.
  - Notar que tanto en la **inserción** como en la **eliminación**, se hace uso de la noción de **orden** entre las **prioridades** (p.e.: la prioridad más pequeña).

Introducción: ¿qué es un cola con prioridad?

- Notar que tanto en la **inserción** como en la **eliminación**, se hace uso de la noción de **orden** entre las **prioridades** (p.e.: la prioridad más pequeña).
- Para que una **CCP** funcione adecuadamente, es indispensable que exista un **comparador** de prioridades a fin de establecer la **relación de orden**.
  - Dadas dos claves: ¿Son iguales? ¿Cuál es la menor?¿Cuál es la mayor?
- En las implementaciones de **CCP**, generalmente se establece que el **tipo de dato de las clave (K)** sea **comparable**, a fin de utilizar por defecto el **DefaultComparator** que delega su operación en la operación **compareTo()** del tipo **comparable K**.
- Otra alternativa, es definir un **comparador** de elementos K, que **defina** específicamente **cómo** ordenar las prioridades.

CCP :: Ejemplo de uso

- Asumir que se utilizará una **CCP** para realizar el **ordenamiento** de un conjunto de parciales.
- ¿**Cómo** utilizar la **CCP** para realizar tal tarea?
- ¿**Cuál** será la **prioridad** asociada a un **parcial**?
- Considere que se **ordenan** respecto a las **notas**.
  - ¿**Qué** sucede si **no parametrizo** un **comparador**?
  - ¿**Qué** nota será la de **mayor prioridad**?
  - ¿**Cuál** es el **orden** por **defecto** de **ordenamiento**?
  - ¿**Cómo** se puede **ordenar descendentemente**?

*Se puede definir una **CCP** cuyos **valores** son los **Parciales**, y cuyas **claves** sean datos **comparables** que indican la **prioridad** establecida para **ordenar** los **parciales**.*

*Dependerá del orden con el cual se desean ordenar. Podrían ordenarse por **Nombre**, por **Nota**, por **Fecha**, etc.*

Se ordenarán con el comparador de float por defecto (asumiendo notas como floats).

La menor nota de la CCP.

Redefiniendo el comparador.

Notas ordenadas de menor a mayor.

Fin de la presentación.