# Implementación eficiente del tipo de dato `Conjunto` en `Java`

Algoritmos y Estructuras de Datos

2$^{\text{do}}$ cuatrimestre 2023

# Introducción

¿Cuál es la menor cantidad de preguntas Sí/No que se necesitan para identificar un entero entre 0 y 15?

# Introducción

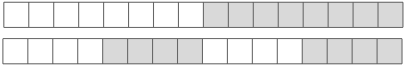
¿Cuál es la menor cantidad de preguntas Sí/No que se necesitan para identificar un entero entre 0 y 15?

1) $x \bmod 16 \geq 8$ 

# Introducción

¿Cuál es la menor cantidad de preguntas Sí/No que se necesitan para identificar un entero entre 0 y 15?

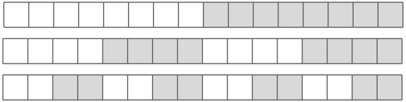
1) $x \bmod 16 \geq 8$
2) $x \bmod 8 \geq 4$



# Introducción

¿Cuál es la menor cantidad de preguntas Sí/No que se necesitan para identificar un entero entre 0 y 15?

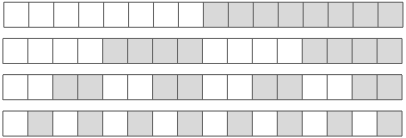
1) $x \bmod 16 \geq 8$
2) $x \bmod 8 \geq 4$
3) $x \bmod 4 \geq 2$



# Introducción

¿Cuál es la menor cantidad de preguntas Sí/No que se necesitan para identificar un entero entre 0 y 15?

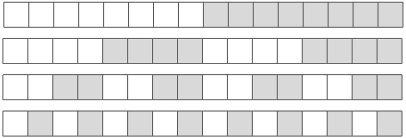
1) $x \bmod 16 \geq 8$
2) $x \bmod 8 \geq 4$
3) $x \bmod 4 \geq 2$
4) $x \bmod 2 \geq 1$



# Introducción

¿Cuál es la menor cantidad de preguntas Sí/No que se necesitan para identificar un entero entre 0 y 15?

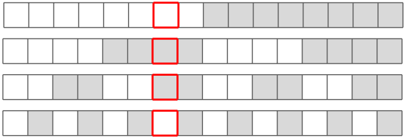
1) $x \bmod 16 \geq 8$
2) $x \bmod 8 \geq 4$
3) $x \bmod 4 \geq 2$
4) $x \bmod 2 \geq 1$



Las preguntas que dividen a la mitad las opciones maximiza la tasa de información.

# Introducción

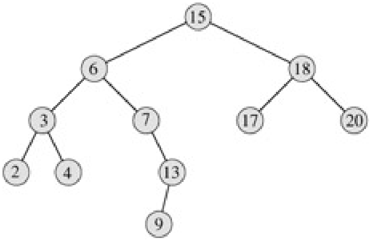
¿Cuál es la menor cantidad de preguntas Sí/No que se necesitan para identificar un entero entre 0 y 15?



1) $x \bmod 16 \geq 8$ `false`
2) $x \bmod 8 \geq 4$ `true`
3) $x \bmod 4 \geq 2$ `true`
4) $x \bmod 2 \geq 1$ `false`

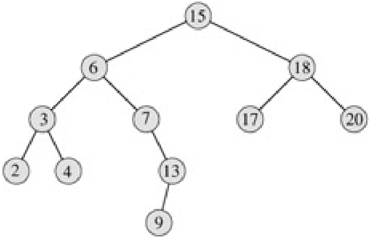
Por eso necesitamos $\lceil \log_2 x \rceil$ bits para representar hasta el número $x$

# Árboles binarios de búsqueda (ABB)



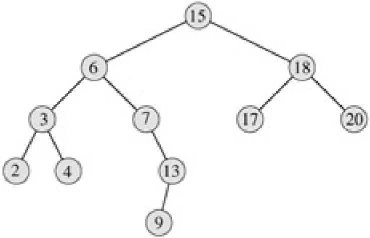

Un árbol binario es un ABB si y sólo si

# Árboles binarios de búsqueda (ABB)



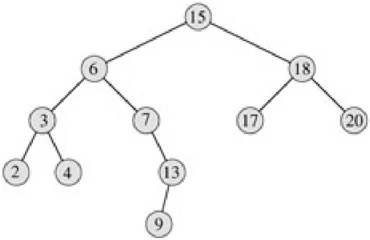

Un árbol binario es un ABB si y sólo si es `null` o

# Árboles binarios de búsqueda (ABB)




Un árbol binario es un ABB si y sólo si es `null` o satisface todas las siguientes condiciones:

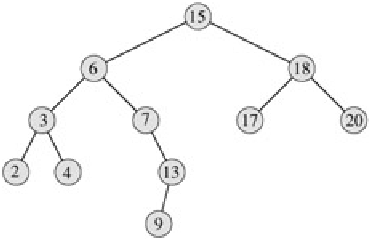
# Árboles binarios de búsqueda (ABB)




Un árbol binario es un ABB si y sólo si es `null` o satisface todas las siguientes condiciones:

- Los valores en todos los nodos del subárbol izquierdo son menores que el valor en la raíz.

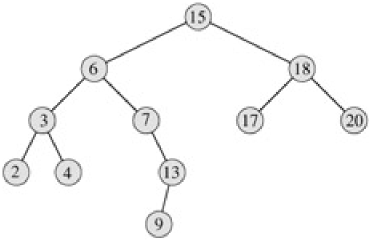
# Árboles binarios de búsqueda (ABB)




Un árbol binario es un ABB si y sólo si es `null` o satisface todas las siguientes condiciones:

- Los valores en todos los nodos del subárbol izquierdo son menores que el valor en la raíz.
- Los valores en todos los nodos del subárbol derecho son mayores que el valor en la raíz.

# Árboles binarios de búsqueda (ABB)




Un árbol binario es un ABB si y sólo si es `null` o satisface todas las siguientes condiciones:

- Los valores en todos los nodos del subárbol izquierdo son menores que el valor en la raíz.
- Los valores en todos los nodos del subárbol derecho son mayores que el valor en la raíz.
- Los subárboles izquierdo y derecho son ABBs.

# Objetivo

Implementar un tipo de datos `Conjunto<T>` en `Java`
usando árboles binarios de búsqueda (ABB)

# Objetivo

Implementar un tipo de datos `Conjunto<T>` en `Java`
usando árboles binarios de búsqueda (ABB)

**¿Memoria dinámica o estática?**

# Implementación (ABB.java)

```
public class ABB<T extends Comparable<T>> implements Conjunto<T> {
```

# Implementación (ABB.java)

```java
public class ABB<T extends Comparable<T>> implements Conjunto<T> {
    private Nodo _raiz;
    // private int _cardinal;
```

# Implementación (`ABB.java`)

El único atributo indispensable es `_raiz`. Pero podríamos usar otros (como `_cardinal`) para tener operaciones $O(1)$.

```
public class ABB<T extends Comparable<T>> implements Conjunto<T> {
    private Nodo _raiz;
    // private int _cardinal;
```

# Implementación (ABB.java)

```java
public class ABB<T extends Comparable<T>> implements Conjunto<T> {
    private Nodo _raiz;
    // private int _cardinal;

    public ABB() {
        _raiz = null;
        // _cardinal = 0;
    }

}
```

## `T extends Comparable<T>`

- Vamos a implementar una clase `Conjunto<T>` paramétrica en un tipo de dato `T` comparable.

# `T extends Comparable<T>`

- Vamos a implementar una clase `Conjunto<T>` paramétrica en un tipo de dato `T` comparable.

Sean dos instancias `elem1` y `elem2`,

- `elem1.compareTo(elem2) > 0` $\Leftrightarrow$ `elem1` es mayor a `elem2`.
- `elem1.compareTo(elem2) == 0` $\Leftrightarrow$ `elem1` es igual a `elem2`.
- `elem1.compareTo(elem2) < 0` $\Leftrightarrow$ `elem1` es menor a `elem2`.

# Representación de los nodos

Definimos la clase `Nodo`. ¿Cuáles son los atributos?

```
private class Nodo {
```

# Representación de los nodos

Declaramos los atributos

```
private class Nodo {
    T valor;
    Nodo izq;
    Nodo der;
    Nodo padre;
```

# Representación de los nodos

Definimos el constructor de Nodo (solo recibe un valor `v` de tipo `T`)

```
private class Nodo {
    T valor;
    Nodo izq;
    Nodo der;
    Nodo padre;

    Nodo(T v) {
```

# Representación de los nodos

¿En qué se diferencia con la estructura de la lista doblemente enlazada?

```
private class Nodo {
    T valor;
    Nodo izq;
    Nodo der;
    Nodo padre;

    Nodo(T v) {
        valor = v;
        izq = null;
        der = null;
        padre = null;
    }
}
```

# Interfaz (Conjunto.java)

La interface Conjunto deberá especificar las siguientes operaciones:

```
interface Conjunto<T> {
```

# Interfaz (Conjunto.java)

La interface Conjunto deberá especificar las siguientes operaciones:

```
interface Conjunto<T> {
    public int cardinal();
```

# Interfaz (Conjunto.java)

La interface Conjunto deberá especificar las siguientes operaciones:

```
interface Conjunto<T> {
    public int cardinal();
    public void insertar(T elem);
```

# Interfaz (Conjunto.java)

La interface Conjunto deberá especificar las siguientes operaciones:

```
interface Conjunto<T> {
    public int cardinal();
    public void insertar(T elem);
    public boolean pertenece(T elem);
```

# Interfaz (Conjunto.java)

La interface Conjunto deberá especificar las siguientes operaciones:

```java
interface Conjunto<T> {
    public int cardinal();
    public void insertar(T elem);
    public boolean pertenece(T elem);
    public void eliminar(T elem);
```

# Interfaz (Conjunto.java)

La interface Conjunto deberá especificar las siguientes operaciones:

```java
interface Conjunto<T> {
    public int cardinal();
    public void insertar(T elem);
    public boolean pertenece(T elem);
    public void eliminar(T elem);
    public String toString();
```

# Interfaz (Conjunto.java)

¿Alguna otra operación que podría resultar útil?

```
interface Conjunto<T> {
    public int cardinal();
    public void insertar(T elem);
    public boolean pertenece(T elem);
    public void eliminar(T elem);
    public String toString();
```

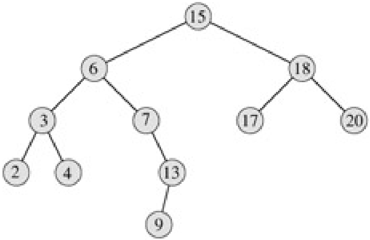
# Interfaz (Conjunto.java)

¿Alguna otra operación que podría resultar útil?

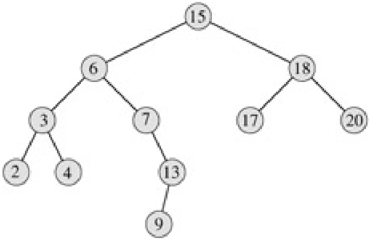
```
interface Conjunto<T> {
    public int cardinal();
    public void insertar(T elem);
    public boolean pertenece(T elem);
    public void eliminar(T elem);
    public String toString();
    public T minimo();
    public T maximo();
}
```

# Algoritmos

`pertenece(T elem)`

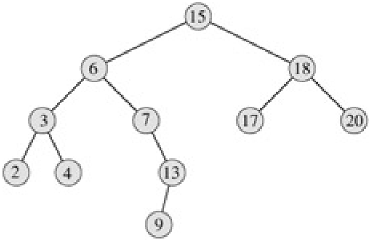


# pertenece(T elem)




busqueda_recursiva(_raiz, elem)

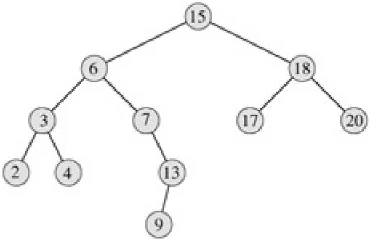
# `pertenece(T elem)`




`busqueda_recursiva(_raiz, elem)`

- `SI` el nodo actual es `null`, devolver `false`.

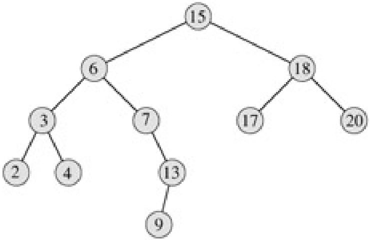
# pertenece(T elem)




`busqueda_recursiva(_raiz, elem)`

- SI el nodo actual es `null`, devolver `false`.
- SI el nodo tiene el elemento, devolver `true`.

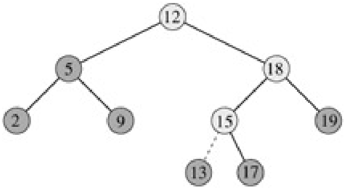
# perteneçe(T elem)




`busqueda_recursiva(_raiz, elem)`

- `SI` el nodo actual es `null`, devolver `false`.
- `SI` el nodo tiene el elemento, devolver `true`.
- `SINO` continuamos la búsqueda recursiva en el nodo que indique `compareTo()`

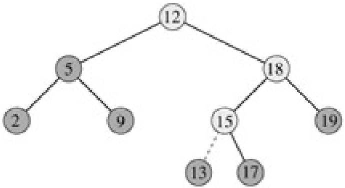
## `insertar(T elem)`




- `ultimo_nodo_buscado = buscar_nodo(_raiz,elem)`

(un algoritmo parecido al anterior, que devuelve el último nodo de la búsqueda)

# insertar(T elem)

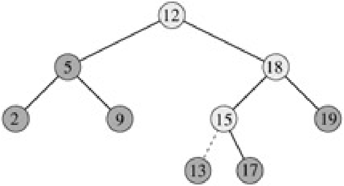



- ultimo_nodo_buscado = buscar_nodo(_raiz,elem)

(un algoritmo parecido al anterior, que devuelve el último nodo de la búsqueda)

- SI lo encontramos, no hacemos nada.

# insertar(T elem)




- `ultimo_nodo_buscado = buscar_nodo(_raiz,elem)`

(un algoritmo parecido al anterior, que devuelve el último nodo de la búsqueda)

- `SI` lo encontramos, no hacemos nada.
- `SINO` lo insertamos como hijo del último nodo de la búsqueda.

`eliminar(T elem)`

## `eliminar(T elem)`

- Buscamos el nodo que tenemos que borrar. Tenemos 4 casos:

# `eliminar(T elem)`

- Buscamos el nodo que tenemos que borrar. Tenemos 4 casos:
  - `SI` no está, no hacemos nada

# `eliminar(T elem)`

- Buscamos el nodo que tenemos que borrar. Tenemos 4 casos:
    - `SI` no está, no hacemos nada
    - `SI` está y no tiene descendencia
      $\rightarrow$ Lo borramos.

## `eliminar(T elem)`

- Buscamos el nodo que tenemos que borrar. Tenemos 4 casos:
  - `SI` no está, no hacemos nada
  - `SI` está y no tiene descendencia
    $\rightarrow$ Lo borramos.
  - `SI` está y tienen un solo hijo.
    $\rightarrow$ El hijo ocupa su lugar.

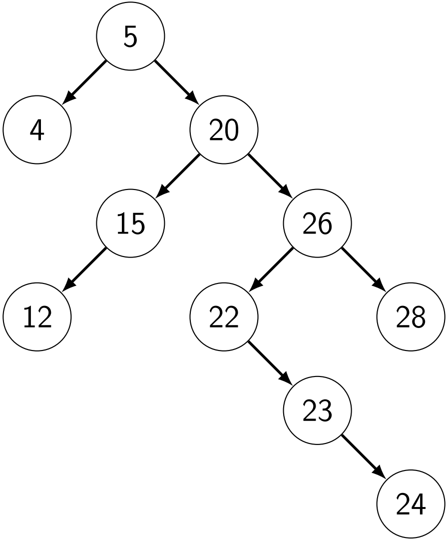
## `eliminar(T elem)`

- Buscamos el nodo que tenemos que borrar. Tenemos 4 casos:
  - `SI` no está, no hacemos nada
  - `SI` está y no tiene descendencia
    $\rightarrow$ Lo borramos.
  - `SI` está y tienen un solo hijo.
    $\rightarrow$ El hijo ocupa su lugar.
  - `SI` está y tiene dos hijos.

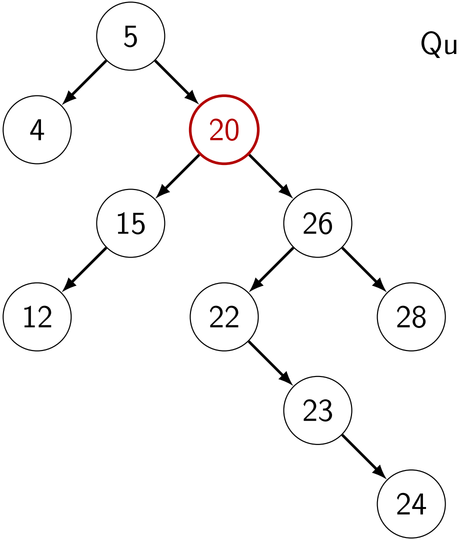
## `eliminar(T elem)`

- Buscamos el nodo que tenemos que borrar. Tenemos 4 casos:
  - `SI` no está, no hacemos nada
  - `SI` está y no tiene descendencia
    $\rightarrow$ Lo borramos.
  - `SI` está y tienen un solo hijo.
    $\rightarrow$ El hijo ocupa su lugar.
  - `SI` está y tiene dos hijos.
    $\rightarrow$ Lo remplazamos por el inmediato sucesor (o predecesor). ¿Dónde está?

`eliminar(T elem)`

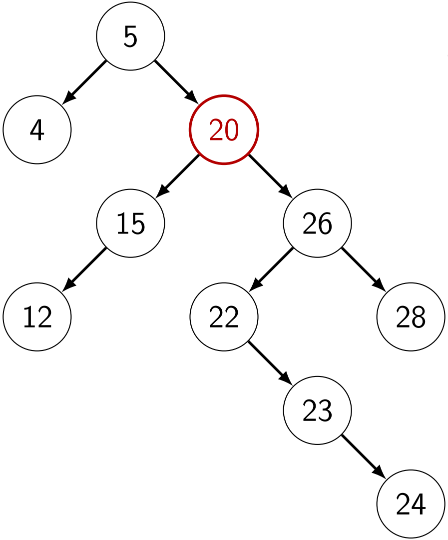


`eliminar(T elem)`



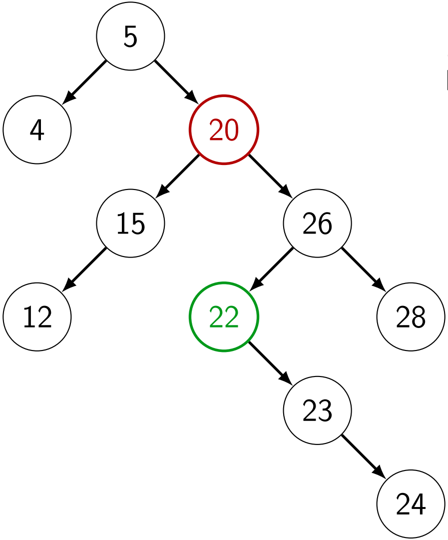

Queremos eliminar el 20.

## `eliminar(T elem)`



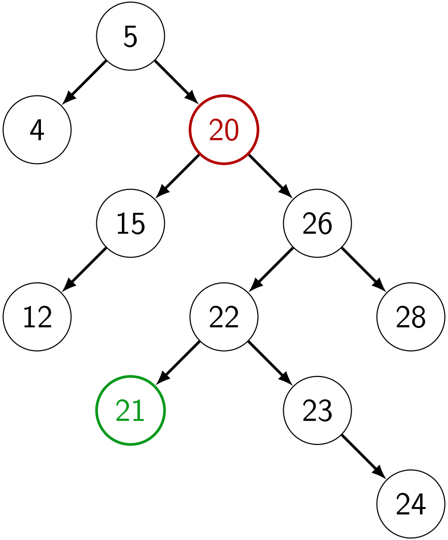

¿Dónde está el sucesor?

`eliminar(T elem)`



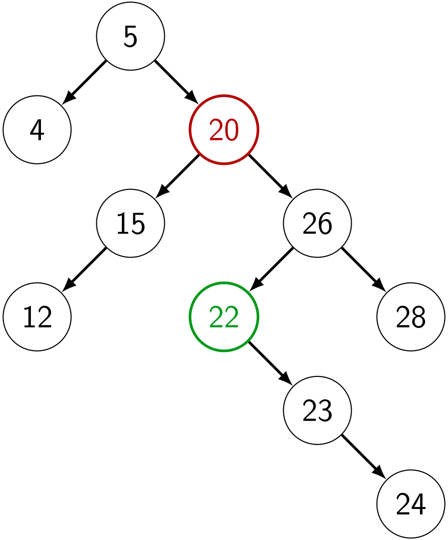

¿Dónde está el sucesor?
El mínimo de la derecha

`eliminar(T elem)`



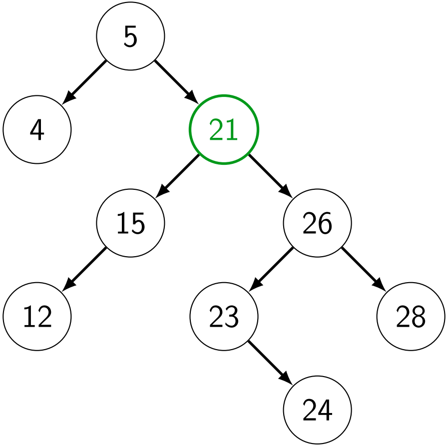

El mínimo siempre es fácil sacarlo

`eliminar(T elem)`




El mínimo siempre es
fácil sacarlo

`eliminar(T elem)`




Removemos el
mínimo derecho
y lo subimos

# Discusión

- ¿Qué complejidades en peor caso tienen las siguientes operaciones?
  donde $N$ es la cantidad de elementos que tiene el conjunto.
  - Pertenece

# Discusión

- ¿Qué complejidades en peor caso tienen las siguientes operaciones?
  donde $N$ es la cantidad de elementos que tiene el conjunto.
  - Pertenece $\rightarrow \mathcal{O}(N)$

# Discusión

- ¿Qué complejidades en peor caso tienen las siguientes operaciones?
  donde $N$ es la cantidad de elementos que tiene el conjunto.
  - Pertenece $\rightarrow \mathcal{O}(N)$
  - Insertar

# Discusión

- ¿Qué complejidades en peor caso tienen las siguientes operaciones?
  donde $N$ es la cantidad de elementos que tiene el conjunto.
  - Pertenece $\to \mathcal{O}(N)$
  - Insertar $\to \mathcal{O}(N)$

# Discusión

- ¿Qué complejidades en peor caso tienen las siguientes operaciones?
  donde $N$ es la cantidad de elementos que tiene el conjunto.
  - Pertenece $\rightarrow \mathcal{O}(N)$
  - Insertar $\rightarrow \mathcal{O}(N)$
  - Borrar

# Discusión

- ¿Qué complejidades en peor caso tienen las siguientes operaciones?
  donde $N$ es la cantidad de elementos que tiene el conjunto.
  - Pertenece $\rightarrow \mathcal{O}(N)$
  - Insertar $\rightarrow \mathcal{O}(N)$
  - Borrar $\rightarrow \mathcal{O}(N)$

# Discusión

- ¿Qué complejidades en peor caso tienen las siguientes operaciones?
  donde $N$ es la cantidad de elementos que tiene el conjunto.
  - Pertenece $\rightarrow \mathcal{O}(N)$
  - Insertar $\rightarrow \mathcal{O}(N)$
  - Borrar $\rightarrow \mathcal{O}(N)$
  - Mínimo/Máximo

# Discusión

- ¿Qué complejidades en peor caso tienen las siguientes operaciones?
  donde $N$ es la cantidad de elementos que tiene el conjunto.
  - Pertenece $\rightarrow \mathcal{O}(N)$
  - Insertar $\rightarrow \mathcal{O}(N)$
  - Borrar $\rightarrow \mathcal{O}(N)$
  - Mínimo/Máximo $\rightarrow \mathcal{O}(N)$ / $\mathcal{O}(1)$

# Discusión

- ¿Qué complejidades en peor caso tienen las siguientes operaciones?
  donde $N$ es la cantidad de elementos que tiene el conjunto.
  - Pertenece $\rightarrow \mathcal{O}(N)$
  - Insertar $\rightarrow \mathcal{O}(N)$
  - Borrar $\rightarrow \mathcal{O}(N)$
  - Mínimo/Máximo $\rightarrow \mathcal{O}(N)$ / $\mathcal{O}(1)$

Las complejidades dependen del orden en el que se hayan ingresado los datos.

# Recorridos de árboles

## Recursivos

`preorder(Bin(`$i$`, `$r$`, `$d$`))` $\equiv$

# Recorridos de árboles

## Recursivos

$\texttt{preorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv \langle r\rangle \ \texttt{\&}\ \texttt{preorder(}i\texttt{)}\ \texttt{\&}\ \texttt{preorder(}d\texttt{)}$

$\texttt{inorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv$

# Recorridos de árboles

## Recursivos

$\texttt{preorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv \langle r \rangle \texttt{ \& preorder(}i\texttt{) \& preorder(}d\texttt{)}$

$\texttt{inorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv \texttt{inorder(}i\texttt{) \& } \langle r \rangle \texttt{ \& inorder(}d\texttt{)}$

$\texttt{postorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv$

# Recorridos de árboles

## Recursivos

$\texttt{preorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv \langle r \rangle \texttt{ \& preorder(}i\texttt{) \& preorder(}d\texttt{)}$

$\texttt{inorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv \texttt{inorder(}i\texttt{) \& } \langle r \rangle \texttt{ \& inorder(}d\texttt{)}$

$\texttt{postorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv \texttt{postorder(}i\texttt{) \& postorder(}d\texttt{)} \;\&\; \langle r \rangle$

# Recorridos de árboles

## Recursivos

$\texttt{preorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv \langle r\rangle \texttt{ \& preorder(}i\texttt{) \& preorder(}d\texttt{)}$

$\texttt{inorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv \texttt{inorder(}i\texttt{) \& }\langle r\rangle\texttt{ \& inorder(}d\texttt{)}$

$\texttt{postorder(Bin(}i\texttt{, }r\texttt{, }d\texttt{))} \equiv \texttt{postorder(}i\texttt{) \& postorder(}d\texttt{)} \ \& \ \langle r\rangle$

## Iterativo

Dar un algoritmo iterativo que recorra todos los nodos de un árbol, en tiempo lineal (i.e. en $\mathcal{O}(n)$),

# Recorridos de árboles

## Recursivos

`preorder(Bin(`$i$`, `$r$`, `$d$`))` $\equiv$ $\langle r \rangle$ `&` `preorder(`$i$`)` `&` `preorder(`$d$`)`

`inorder(Bin(`$i$`, `$r$`, `$d$`))` $\equiv$ `inorder(`$i$`)` `&` $\langle r \rangle$ `&` `inorder(`$d$`)`

`postorder(Bin(`$i$`, `$r$`, `$d$`))` $\equiv$ `postorder(`$i$`)` `&` `postorder(`$d$`)` & $\langle r \rangle$

## Iterativo

Dar un algoritmo iterativo que recorra todos los nodos de un árbol, en tiempo lineal (i.e. en $\mathcal{O}(n)$),

- ¿Por qué, si ya conocemos recorridos recursivos?
  Para (después) poder implementar iteradores sobre árboles.

## Iterador<T>

```
private class ABB_Iterador implements Iterador<T> {
    private Nodo _actual;

    public boolean haySiguiente() {
        /* ... */
    }

    public T siguiente() {
        /* ... */
    }
}

public Iterador<T> iterador() {
    return new ABB_Iterador();
}
```

# Iterador inorder

**Observación**

Si el árbol es un ABB, el recorrido inorder está

# Iterador inorder

**Observación**

Si el árbol es un ABB, el recorrido inorder está ordenado.

# Iterador inorder

### Observación
Si el árbol es un ABB, el recorrido inorder está ordenado.

### Para hacer un recorrido inorder:

- El primer elemento que tenemos que visitar es

# Iterador inorder

**Observación**

Si el árbol es un ABB, el recorrido inorder está ordenado.

**Para hacer un recorrido inorder:**

- El primer elemento que tenemos que visitar es el mínimo. Sabemos cómo encontrarlo: yendo siempre hacia la izquierda.

# Iterador inorder

### Observación

Si el árbol es un ABB, el recorrido inorder está ordenado.

### Para hacer un recorrido inorder:

- El primer elemento que tenemos que visitar es el mínimo. Sabemos cómo encontrarlo: yendo siempre hacia la izquierda.
- Tenemos que hallar el sucesor del mínimo.

# Iterador inorder

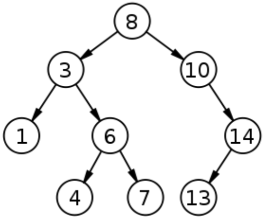
Observación

Si el árbol es un ABB, el recorrido inorder está ordenado.

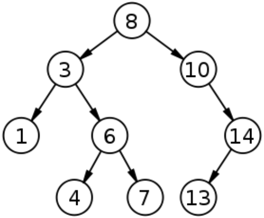
Para hacer un recorrido inorder:

- El primer elemento que tenemos que visitar es el mínimo. Sabemos cómo encontrarlo: yendo siempre hacia la izquierda.
- Tenemos que hallar el sucesor del mínimo.
- Más en general, tenemos que poder hallar el sucesor de un elemento arbitrario del árbol.

`sucesor(T elem)`



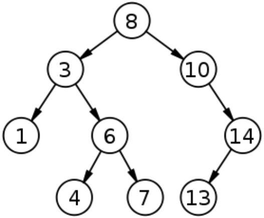
## sucesor(T elem)




MIENTRAS: bajamos de la raíz hasta llegar al nodo X:

SI: el nodo actual es mayor, lo guardamos como posible_sucesor y continuamos (por el sub-árbol izquierdo naturalmente).

## sucesor(T elem)



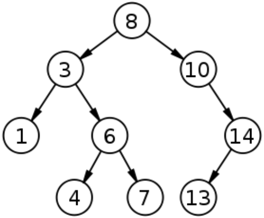

MIENTRAS: bajamos de la raíz hasta llegar al nodo X:

SI: el nodo actual es mayor, lo guardamos como posible_sucesor y continuamos (por el sub-árbol izquierdo naturalmente).

SI: el nodo X no tiene descendencia derecha

Devolvemos el último posible_sucesor (puede ser nulo eventualmente)

## `sucesor(T elem)`




`MIENTRAS:` bajamos de la raíz hasta llegar al nodo `X`:

`SI:` el nodo actual es mayor, lo guardamos como `posible_sucesor` y continuamos (por el sub-árbol izquierdo naturalmente).

`SI:` el nodo `X` no tiene descendencia derecha

Devolvemos el último `posible_sucesor` (puede ser nulo eventualmente)

`SINO:`

Devolvemos el `menor(X.der)`

# Alternativa, sucesor según el capítulo 12 del Cormen

Notas: ($p[x]$ es X.padre) ($\leftarrow$ es asignación) ($=$ es comparación)

```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
    then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
      do x ← y
         y ← p[y]
return y
```

# ¡A programar!

En `ABB.java` está la declaración de la clase, los métodos públicos y la definición de `Nodo` y de `ABB_Iterador`.