Implementación eficiente del tipo de dato `Conjunto` en Java

Algoritmos y Estructuras de Datos

2do cuatrimestre 2024

Introducción

Vamos a implementar `Conjunto.java`
mediante estructuras de datos eficientes.

Introducción

Vamos a implementar `Conjunto.java`
mediante estructuras de datos eficientes.

```
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();
}
```

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`				
`insertar`				
`borrar`				
`max/min`				

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$			
`insertar`				
`borrar`				
`max/min`				

Introducción

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$			
`insertar`	$\mathcal{O}(N)$			
`borrar`				
`max/min`				

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$			
`insertar`	$\mathcal{O}(N)$			
`borrar`	$\mathcal{O}(N)$			
`max/min`				

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$			
`insertar`	$\mathcal{O}(N)$			
`borrar`	$\mathcal{O}(N)$			
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$			

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`insertar`	$\mathcal{O}(N)$			
`borrar`	$\mathcal{O}(N)$			
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$			

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`borrar`	$\mathcal{O}(N)$			
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$			

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$			

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$		

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$		

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$		
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$		

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$		

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(logN)$
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(logN)$
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(logN)$
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(logN)$
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(1)$

Introducción

Complejidad de las estructura de implementadas hasta ahora:

	Lista enlazada	Lista bi-enlazada	Arreglo redimensionable	Arreglo redimensionable (ordenado)
`pertenece`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(logN)$
`insertar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
`borrar`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
`max/min`	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(N)/\mathcal{O}(1)$	$\mathcal{O}(1)$

Podemos hacer algo mejor

Sabemos que dado un arreglo de tamaño N ordenado podemos verificar pertenencia de un elemento en $\mathcal{O}(logN)$ con **búsqueda binaria**

Introducción

$2 \in [2, 5, 8, 12, 15, 18, 21, 24]$? 8 posibles, 4 candidatos de cada lado

Introducción

$2 \in [2, 5, 8, 12, 15, 18, 21, 24]$? 8 posibles, 4 candidatos de cada lado

$2 \leq 12$. Nos queda $[2, 5, 8, 12]$ 4 posibles, 2 candidatos de cada lado

Introducción

$2 \in [2, 5, 8, 12, 15, 18, 21, 24]$? 8 posibles, 4 candidatos de cada lado

$2 \leq 12$. Nos queda $[2, 5, 8, 12]$ 4 posibles, 2 candidatos de cada lado

$2 \leq 5$. Nos queda $[2, 5]$ 2 posibles, 1 candidato de cada lado

Introducción

$2 \in [2, 5, 8, 12, 15, 18, 21, 24]$? 8 posibles, 4 candidatos de cada lado

$2 \leq 12$. Nos queda $[2, 5, 8, 12]$ 4 posibles, 2 candidatos de cada lado

$2 \leq 5$. Nos queda $[2, 5]$ 2 posibles, 1 candidato de cada lado

$2 \leq 2$. Nos queda $[2]$

Introducción

$2 \in [2, 5, 8, 12, 15, 18, 21, 24]$? 8 posibles, 4 candidatos de cada lado

$2 \leq 12$. Nos queda $[2, 5, 8, 12]$ 4 posibles, 2 candidatos de cada lado

$2 \leq 5$. Nos queda $[2, 5]$ 2 posibles, 1 candidato de cada lado

$2 \leq 2$. Nos queda $[2]$

Necesitamos a lo sumo $\lceil \log_2 N \rceil$ preguntas

Introducción

$2 \in [2, 5, 8, 12, 15, 18, 21, 24]$? 8 posibles, 4 candidatos de cada lado

$2 \leq 12$. Nos queda $[2, 5, 8, 12]$ 4 posibles, 2 candidatos de cada lado

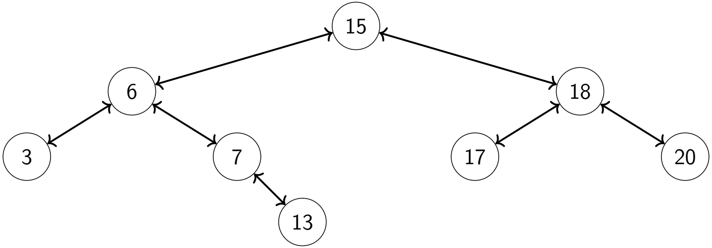
$2 \leq 5$. Nos queda $[2, 5]$ 2 posibles, 1 candidato de cada lado

$2 \leq 2$. Nos queda $[2]$

Necesitamos a lo sumo $\lceil \log_2 N \rceil$ preguntas

¿Podemos usar esta idea para **implementar** un conjunto eficiente?

Árboles binarios de búsqueda (ABB)



Árboles binarios de búsqueda (ABB): invariante de representación

Un objeto es ABB $\iff$

Árboles binarios de búsqueda (ABB): invariante de representación

Un objeto es ABB $\iff$ es `null`

Árboles binarios de búsqueda (ABB): invariante de representación

Un objeto es ABB $\iff$ es `null` o satisface las siguientes condiciones:

Árboles binarios de búsqueda (ABB): invariante de representación

Un objeto es ABB $\iff$ es `null` o satisface las siguientes condiciones:

- Los valores del subárbol izquierdo son menores que el valor de la raíz.

Árboles binarios de búsqueda (ABB): invariante de representación

Un objeto es ABB $\iff$ es `null` o satisface las siguientes condiciones:

- Los valores del subárbol izquierdo son menores que el valor de la raíz.
- Los valores del subárbol derecho son mayores que el valor de la raíz.

Árboles binarios de búsqueda (ABB): invariante de representación

Un objeto es ABB $\iff$ es `null` o satisface las siguientes condiciones:

- Los valores del subárbol izquierdo son menores que el valor de la raíz.
- Los valores del subárbol derecho son mayores que el valor de la raíz.
- Los objetos izquierdos y derechos son ABBs.

Objetivo

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

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;
```

Implementación (ABB.java)

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

Implementación (ABB.java)

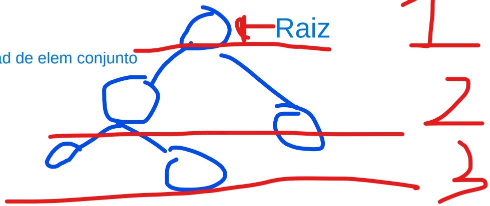
Constructor

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

    public ABB() {
        _raiz = null;
    }
}
```

Implementación (`ABB.java`)

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

```
public class ABB<T extends Comparable<T>> implements Conjunto<T> {
    private Nodo _raiz;
    private int _cardinal;Cantidad de elem conjunto
    private int _altura;

     CONSTRUCTOR
    public ABB() {
        _raiz = null;
        _cardinal = 0;
        _altura = 0;
    }
}
```



Altura = 3

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;
    }
}
```

Algoritmos

`abb.pertenece(elem)`

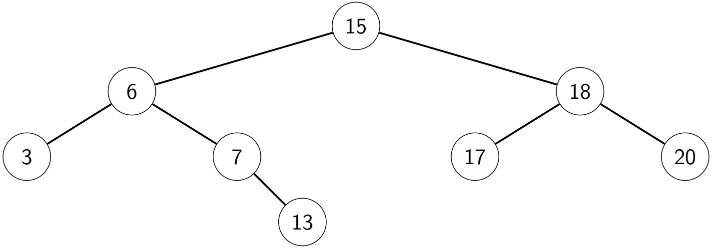


abb.pertenece(elem)

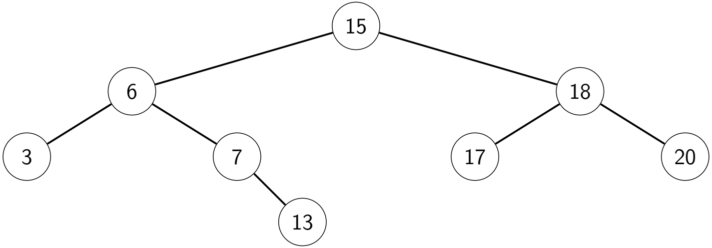
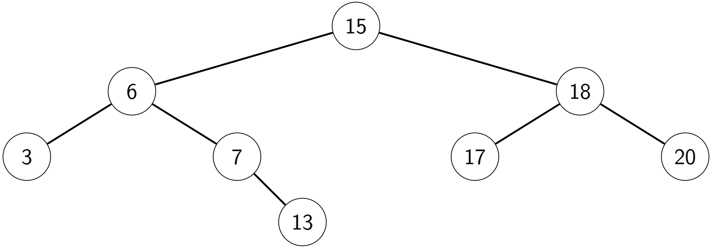


abb.busqueda_recursiva(elem)

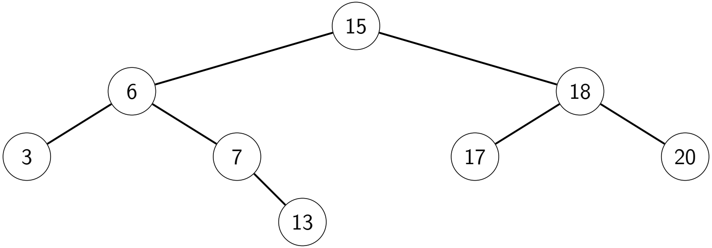
`abb.pertenece(elem)`



`abb.busqueda_recursiva(elem)`

- **Caso Base 1**. Si el ABB es `null`, devolver `false`.
- **Caso Base 2**. Si la raiz contiene el elemento, devolver `true`.

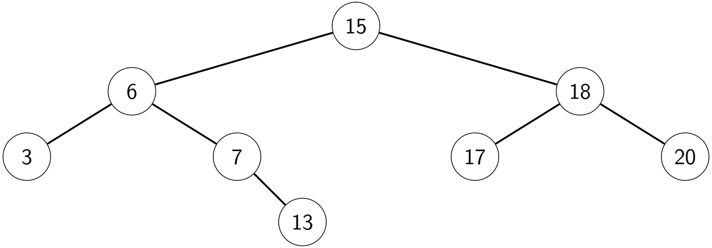
`abb.pertenece(elem)`



`abb.busqueda_recursiva(elem)`

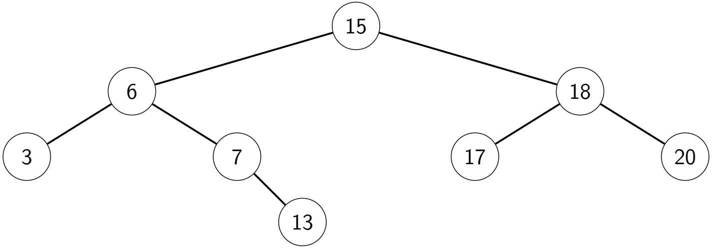
- **Caso Base 1**. Si el ABB es `null`, devolver `false`.
- **Caso Base 2**. Si la raiz contiene el elemento, devolver `true`.
- **Paso Recursivo**. Si no, continuamos la búsqueda recursiva en el sub-árbol que indique `compareTo()`

abb.insertar(elem)



Busco y si esta , no inserto, si no esta inserto

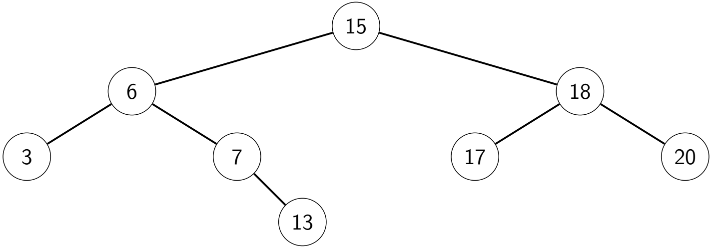
`abb.insertar(elem)`



`ultimo_nodo_buscado = abb.buscar_nodo(elem)`

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

abb.insertar(elem)



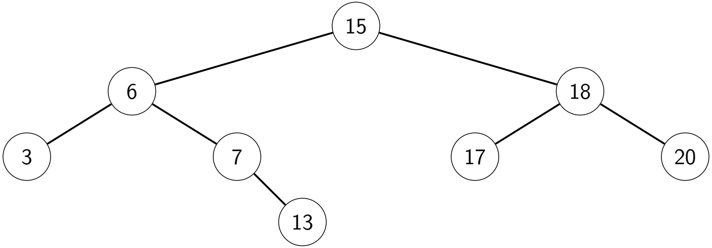
CUANDO BUSCO UN ELEM, LO GUARDO EN UNA VARIABLE

`ultimo_nodo_buscado = abb.buscar_nodo(elem)`

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

- SI lo encontramos, no hacemos nada.

`abb.insertar(elem)`

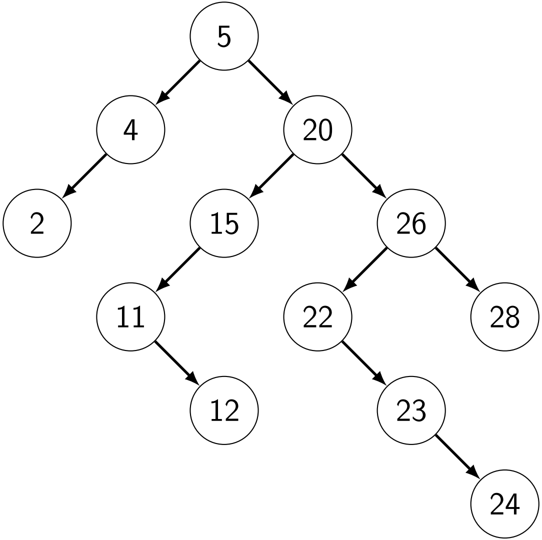


`ultimo_nodo_buscado = abb.buscar_nodo(elem)`

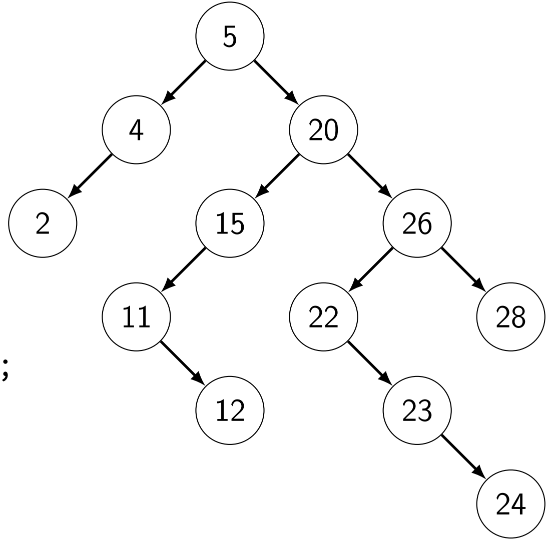
(un algoritmo parecido al anterior, que devuelve el último nodo de la búsqueda)

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

`abb.eliminar(elem)`

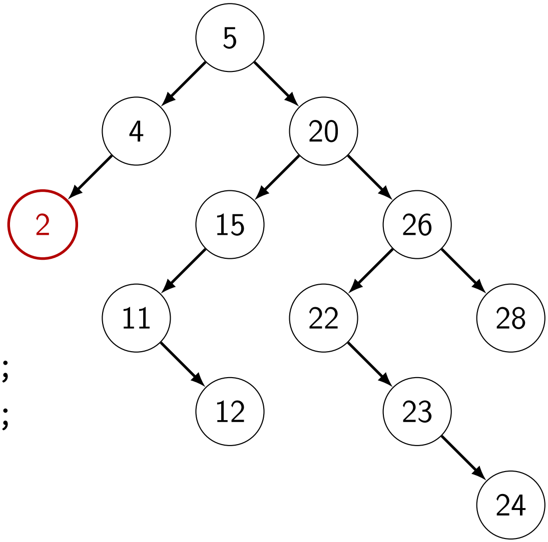


`abb.eliminar(elem)`



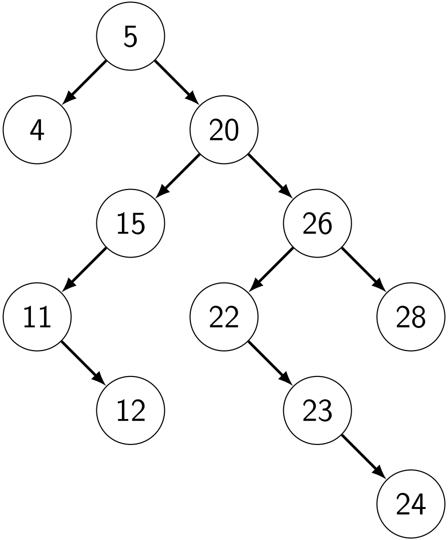

```
abb.eliminar(3);
```

abb.eliminar(elem)




```
abb.eliminar(3);
abb.eliminar(2);
```

`abb.eliminar(elem)`




```
abb.eliminar(3);
abb.eliminar(2);
```

`abb.eliminar(elem)`

```
abb.eliminar(3);
abb.eliminar(2);
abb.eliminar(11);
```

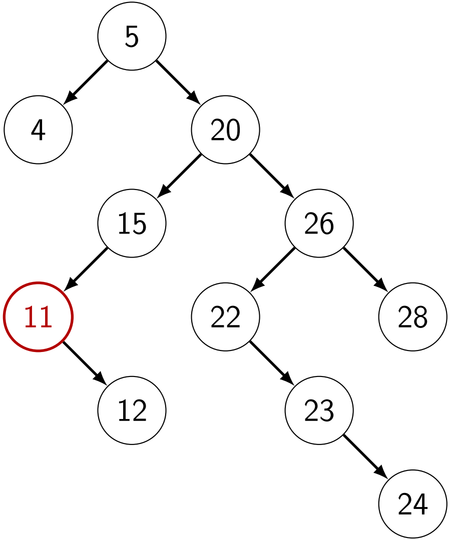
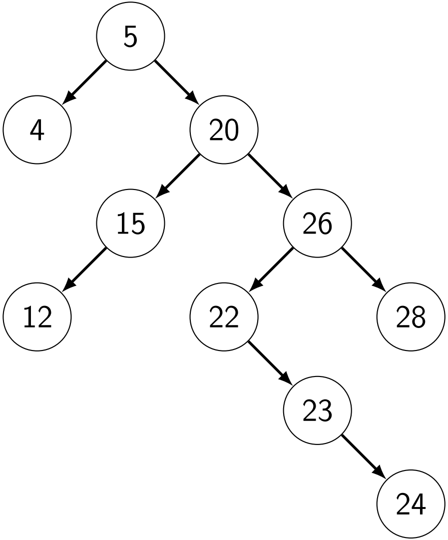
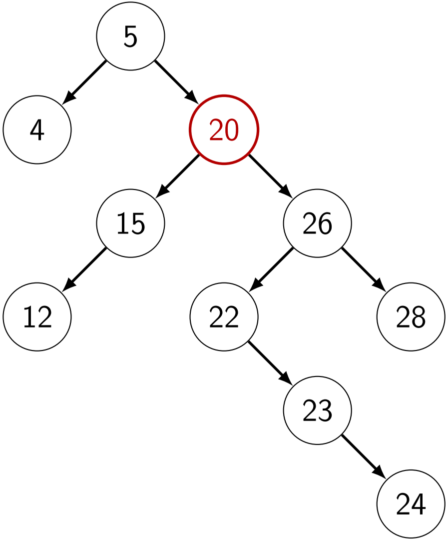


abb.eliminar(elem)



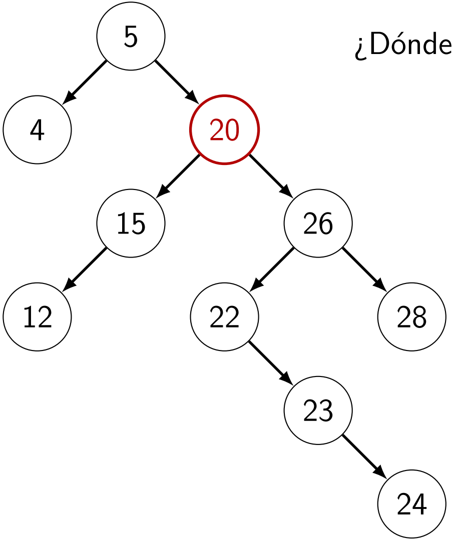

```
abb.eliminar(3);
abb.eliminar(2);
abb.eliminar(11);
```

abb.eliminar(elem)




```
abb.eliminar(3);
abb.eliminar(2);
abb.eliminar(11);
abb.eliminar(20);
```

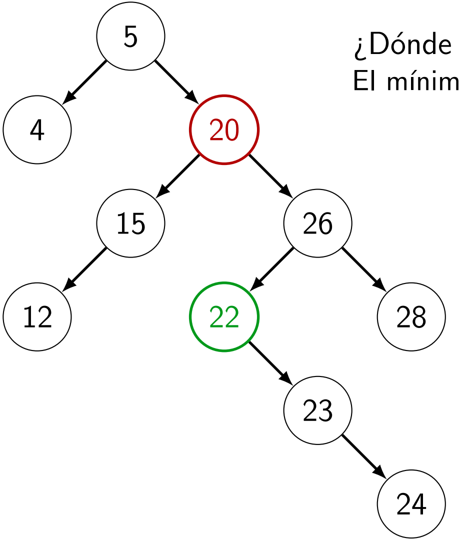
`abb.eliminar(elem)`



¿Dónde está el sucesor?

```
abb.eliminar(3);
abb.eliminar(2);
abb.eliminar(11);
abb.eliminar(20);
```

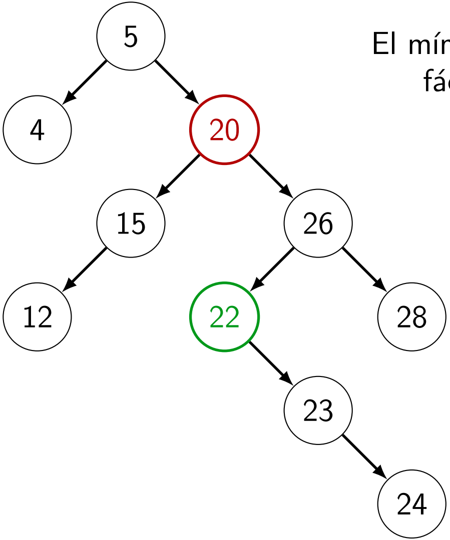
`abb.eliminar(elem)`



¿Dónde está el sucesor?
El mínimo de la derecha

```
abb.eliminar(3);
abb.eliminar(2);
abb.eliminar(11);
abb.eliminar(20);
```

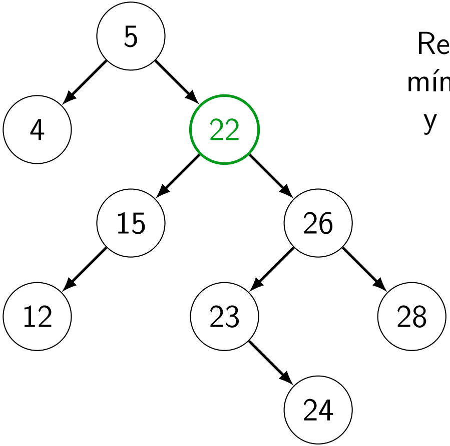
abb.eliminar(elem)



El mínimo siempre es
fácil sacarlo

```
abb.eliminar(3);
abb.eliminar(2);
abb.eliminar(11);
abb.eliminar(20);
```

`abb.eliminar(elem)`



Removemos el mínimo derecho y lo subimos

```
abb.eliminar(3);
abb.eliminar(2);
abb.eliminar(11);
abb.eliminar(20);
```

`abb.eliminar(elem)`

- Tenemos 4 casos:

`abb.eliminar(elem)`

- Tenemos 4 casos:
  - `SI` no está, no hacemos nada

`abb.eliminar(elem)`

- Tenemos 4 casos:
  - `SI` no está, no hacemos nada
  - `SI` está y no tiene descendencia
    $\rightarrow$ Lo borramos.

`abb.eliminar(elem)`

- 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.

`abb.eliminar(elem)`

- 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.

`abb.eliminar(elem)`

- 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).

`abb.sucesor(elem)`

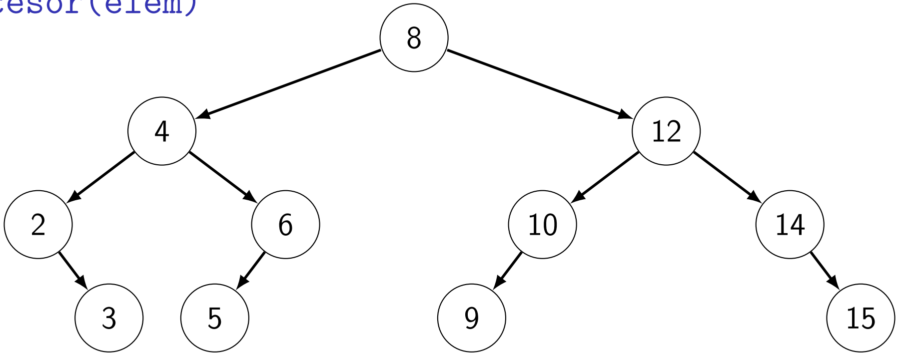
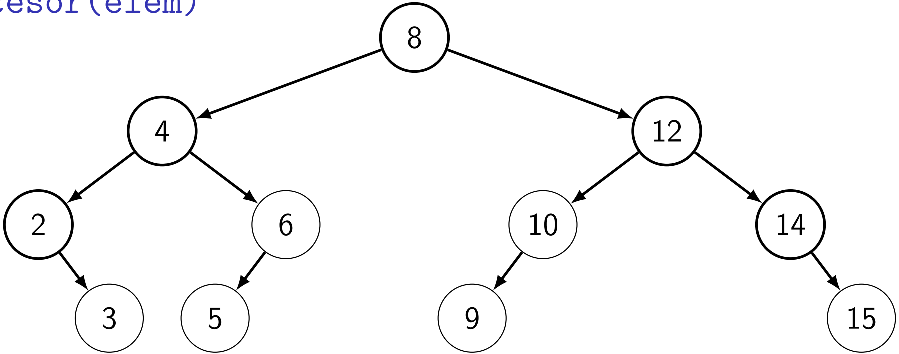
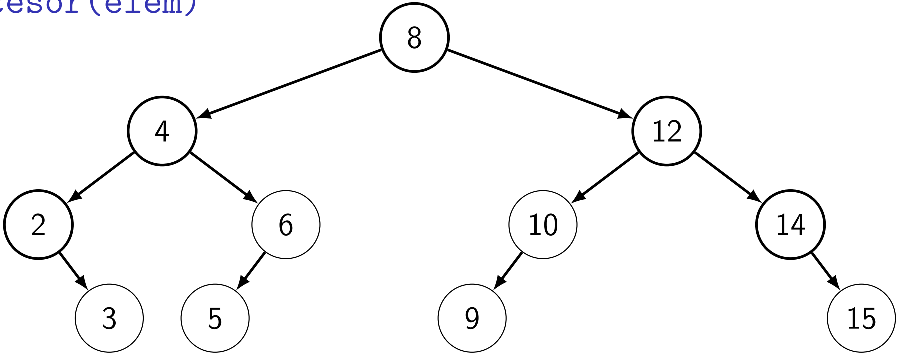


abb.sucesor(elem)



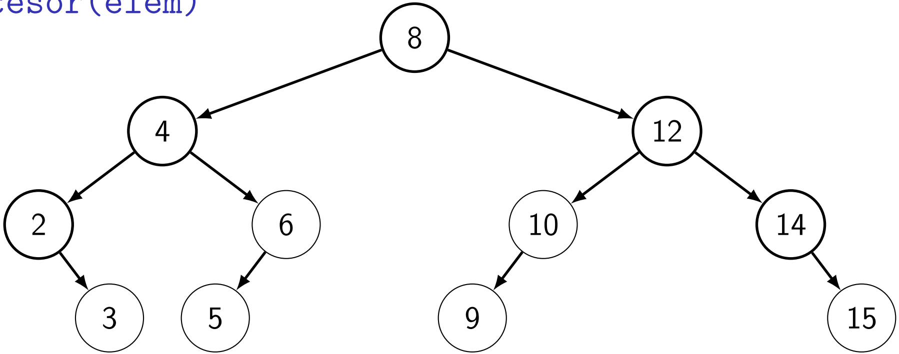
Si tiene_subarbol_derecho(elem):

abb.sucesor(elem)



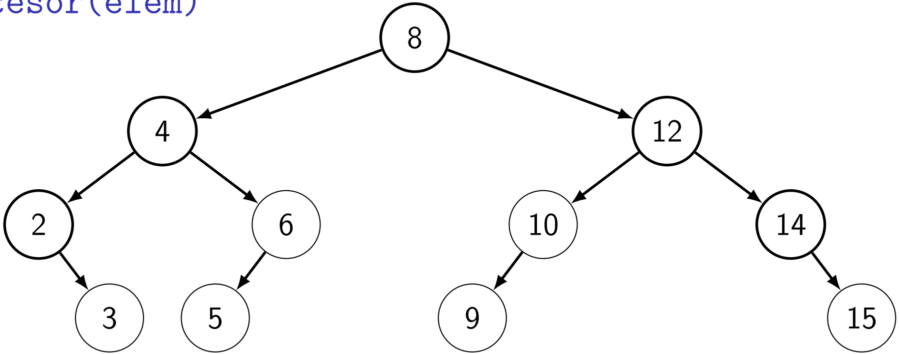

```
Si tiene_subarbol_derecho(elem):
    res = minimo_a_su_derecha(elem)
```

`abb.sucesor(elem)`




```
Si tiene_subarbol_derecho(elem):
    res = minimo_a_su_derecha(elem)
Si no:
```

abb.sucesor(elem)




```
Si tiene_subarbol_derecho(elem):
    res = minimo_a_su_derecha(elem)
Si no:
    res = primer_ancestro_derecho(elem)
```

Que signifca ancestro

abb.sucesor(elem)

```java
private Nodo sucesor(Nodo nodo){
// caso tiene subarbol derecho
Nodo res;
if (nodo._der != null){
    res = nodo._der;
    while (res._izq != null){
        res = res._izq;
    }
} else {

// caso contrario: no tiene subarbol derecho
// el siguiente es el primer padre de un subarbol izquierdo
    res = nodo._padre;
    while (res._der != null && res._der._valor.equals(nodo._valor)) {
        res = res._padre;
    }
}
return res;
}
```

Iterador<T>

```
private class ABB_Iterador implements Iterador<T> {
    private Nodo _actual = this.minimo();

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

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

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

Algoritmos Recursivo

Que significa esto?

```
inorder(ABB) = inorder(ABB.izq) + [ABB.val] + inorder(ABB.der)
```

Iterador

Algoritmos Recursivo

```
inorder(ABB) = inorder(ABB.izq) + [ABB.val] + inorder(ABB.der)
```

Para hacer un `iterador` necesitamos un algoritmo *iterativo*:

Algoritmos Recursivo

```
inorder(ABB) = inorder(ABB.izq) + [ABB.val] + inorder(ABB.der)
```

Para hacer un `iterador` necesitamos un algoritmo *iterativo*:

Constructor: Crear el iterador apuntando al primer elemento (el mínimo).
Siguiente: Devolver el nodo actual y apuntar a su sucesor.

Complejidades

¿Qué complejidades en peor caso tienen las siguientes operaciones?:

- `abb.pertenece(elem)`:
- `abb.insertar(elem)`:
- `abb.min(elem)`:
- `abb.sucesor(elem)`:
- `abb.borrar(elem)`:

Complejidades

¿Qué complejidades en peor caso tienen las siguientes operaciones?:

- `abb.pertenece(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.insertar(elem)`:
- `abb.min(elem)`:
- `abb.sucesor(elem)`:
- `abb.borrar(elem)`:

Complejidades

¿Qué complejidades en peor caso tienen las siguientes operaciones?:

- `abb.pertenece(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.insertar(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.min(elem)`:
- `abb.sucesor(elem)`:
- `abb.borrar(elem)`:

Complejidades

¿Qué complejidades en peor caso tienen las siguientes operaciones?:

- `abb.pertenece(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.insertar(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.min(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.sucesor(elem)`:
- `abb.borrar(elem)`:

Complejidades

¿Qué complejidades en peor caso tienen las siguientes operaciones?:

- `abb.pertenece(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.insertar(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.min(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.sucesor(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.borrar(elem)`:

Complejidades

¿Qué complejidades en peor caso tienen las siguientes operaciones?:

- `abb.pertenece(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.insertar(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.min(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.sucesor(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.borrar(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$

Complejidades

¿Qué complejidades en peor caso tienen las siguientes operaciones?:

- `abb.pertenece(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.insertar(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.min(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.sucesor(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.borrar(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$

Rebalancendo (AVL) tendríamos complejidades $\mathcal{O}(\text{Altura}) = \mathcal{O}(\log N)$

Complejidades

¿Qué complejidades en peor caso tienen las siguientes operaciones?:

- `abb.pertenece(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.insertar(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.min(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.sucesor(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$
- `abb.borrar(elem)`: $\mathcal{O}(\text{Altura}) = \mathcal{O}(N)$

Rebalancendo (AVL) tendríamos complejidades $\mathcal{O}(\text{Altura}) = \mathcal{O}(\log N)$

Agregando un atributo privado, `abb.min(elem)` podría ser $\mathcal{O}(1)$

¡A programar!

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