Implementación eficiente del tipo de dato `Conjunto` en Java

Algoritmos y Estructuras de Datos

1er cuatrimestre 2024

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

Introducción

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

```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();
    public T minimo();
    public T maximo();
}
```

Introducción

Complejidad de las estructura de implementadas hasta ahora:

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

Introducción

Complejidad de las estructura de implementadas hasta ahora:

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

Introducción

Complejidad de las estructura de implementadas hasta ahora:

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

Complejidad de las estructura de implementadas hasta ahora:

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

Complejidad de las estructura de implementadas hasta ahora:

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

Complejidad de las estructura de implementadas hasta ahora:

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

Complejidad de las estructura de implementadas hasta ahora:

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

Complejidad de las estructura de implementadas hasta ahora:

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

Introducción

Complejidad de las estructura de implementadas hasta ahora:

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

Introducción

Complejidad de las estructura de implementadas hasta ahora:

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

¿Podemos hacer algo mejor?

¿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?

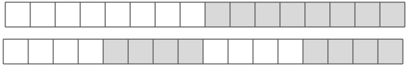
1) $x \bmod 16 \geq 8$ 

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 mod $16 \geq 8$ 

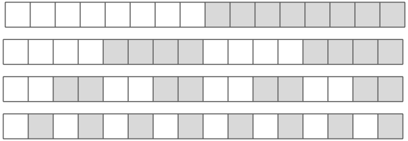
2) x mod $8 \geq 4$ 

3) x mod $4 \geq 2$ 

Introducción

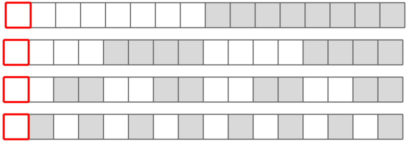
¿Cuál es la menor cantidad de preguntas Sí/No que se necesitan para identificar un entero entre 0 y 15?

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



Introducción

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



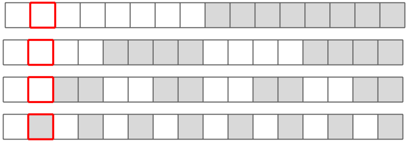
1) $x \bmod 16 \geq 8$ `false`
2) $x \bmod 8 \geq 4$ `false`
3) $x \bmod 4 \geq 2$ `false`
4) $x \bmod 2 \geq 1$ `false`

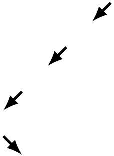
0

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

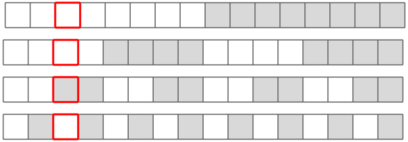




1

Introducción

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



1) $x \bmod 16 \geq 8$ `false`

2) $x \bmod 8 \geq 4$ `false`

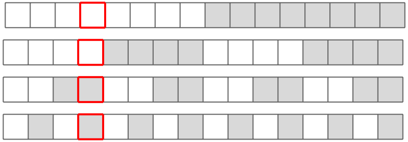
3) $x \bmod 4 \geq 2$ `true`

4) $x \bmod 2 \geq 1$ `false`

2

Introducción

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



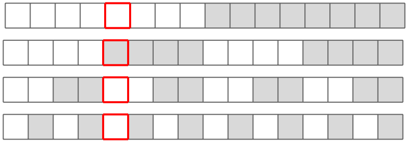
1) $x \bmod 16 \geq 8$ `false`
2) $x \bmod 8 \geq 4$ `false`
3) $x \bmod 4 \geq 2$ `true`
4) $x \bmod 2 \geq 1$ `true`

3

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) $x \bmod 4 \geq 2$ false
4) $x \bmod 2 \geq 1$ false

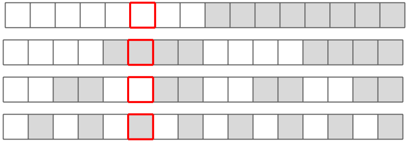


4

Introducción

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

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

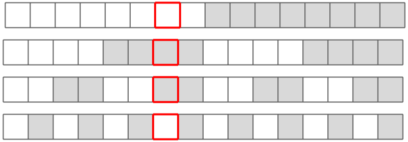


5

Introducción

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

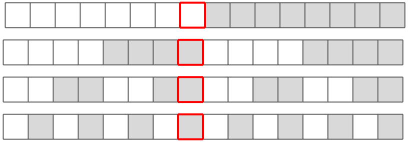
1) x mod $16 \geq 8$ `false`
2) x mod $8 \geq 4$ `true`
3) x mod $4 \geq 2$ `true`
4) x mod $2 \geq 1$ `false`

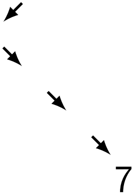


6

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

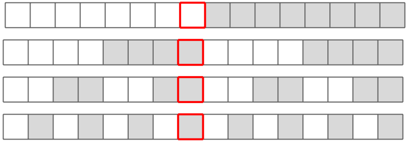
1) x mod $16 \geq 8$ `false`
2) x mod $8 \geq 4$ `true`
3) x mod $4 \geq 2$ `true`
4) x mod $2 \geq 1$ `true`





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) $x \bmod 2 \geq 1$ `true`

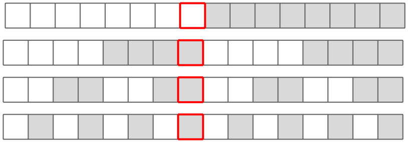
7

Necesitamos $\lceil \log_2 x \rceil$ preguntas (bits) para **representar** un número x

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) $x \bmod 2 \geq 1$ `true`

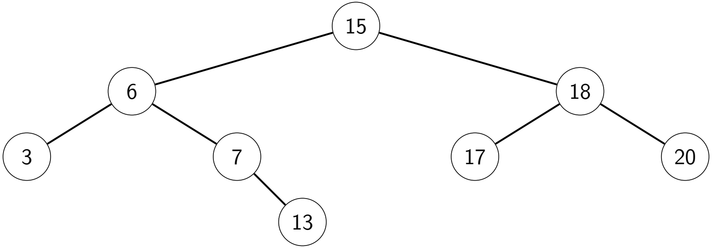


7

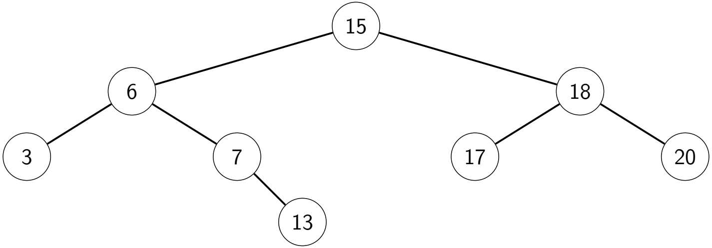
Necesitamos $\lceil \log_2 x \rceil$ preguntas (bits) para **representar** un número x

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

Árboles binarios de búsqueda (ABB):

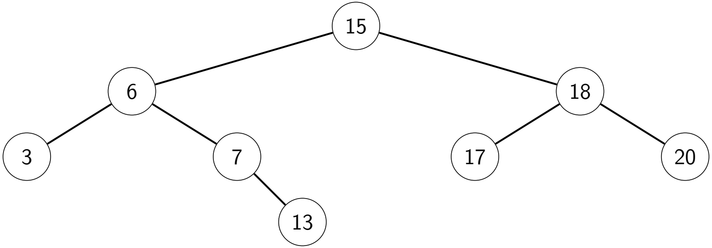


Árboles binarios de búsqueda (ABB): invariante de representación



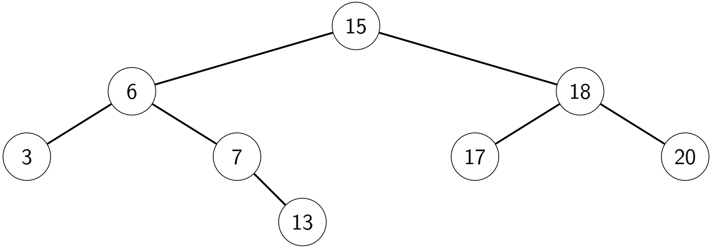
Un objeto es ABB $\iff$

Árboles binarios de búsqueda (ABB): invariante de representación



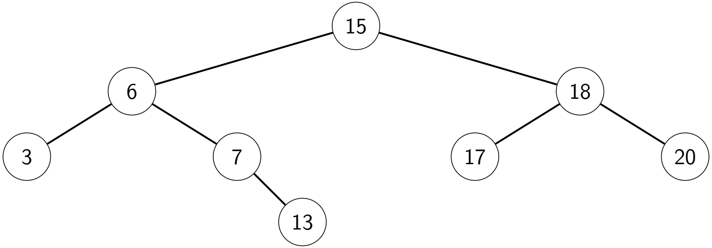
Un objeto es ABB $\iff$ es `null`

Árboles binarios de búsqueda (ABB): invariante de representación



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

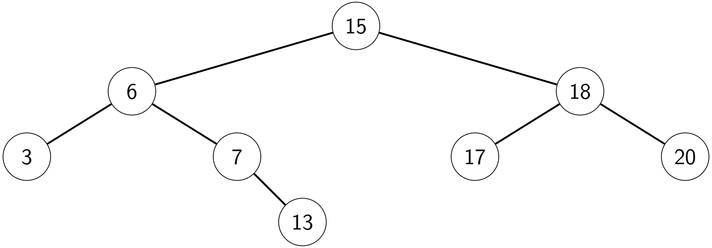
Árboles binarios de búsqueda (ABB): invariante de representación



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.

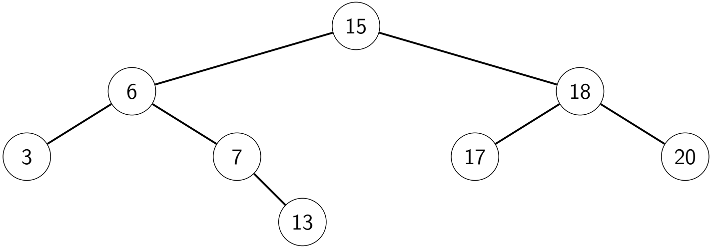
Árboles binarios de búsqueda (ABB): invariante de representación



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)

```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;
    // private int _altura;
```

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;
    // private int _altura;
```

Implementación (ABB.java)

Constructor

```java
public class ABB<T extends Comparable<T>> implements Conjunto<T> {
    private Nodo _raiz;
    // private int _cardinal;
    // private int _altura;
    public ABB() {
        _raiz = null;
        // _cardinal = 0;
        // _altura = 0;
    }
}
```

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 arriba;
```

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 arriba;
    Nodo(T v) {
```

Representación de los nodos

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

```java
private class Nodo {
    T valor;
    Nodo izq;
    Nodo der;
    Nodo arriba;
    Nodo(T v) {
        valor = v;
        izq = null;
        der = null;
        arriba = null;
    }
}
```

Algoritmos

`abb.pertenece(elem)`

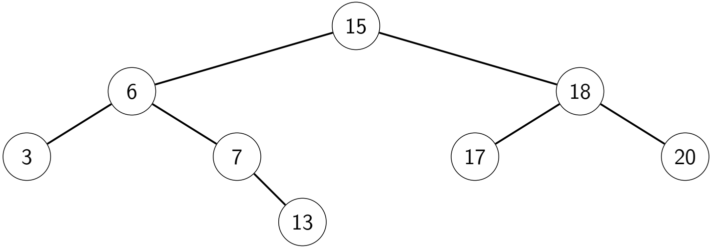


abb.pertenece(elem)

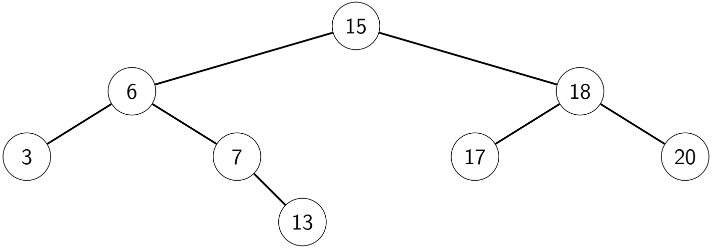
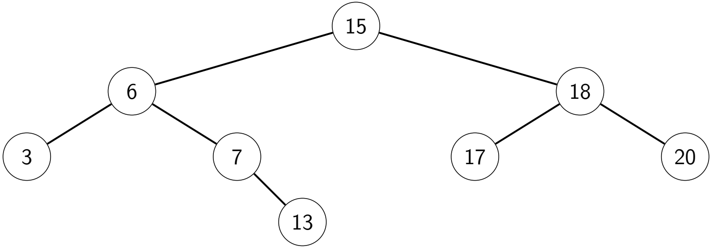


abb.busqueda_recursiva(elem)

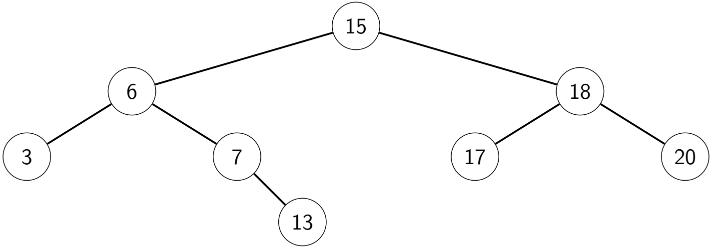
`abb.pertenece(elem)`



`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)`

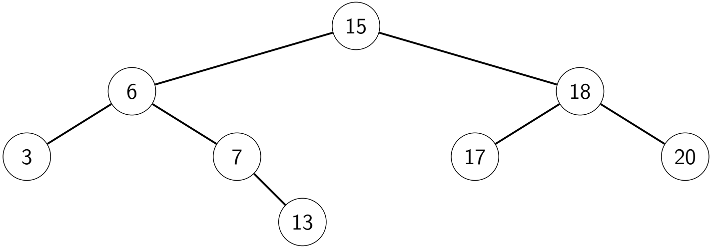
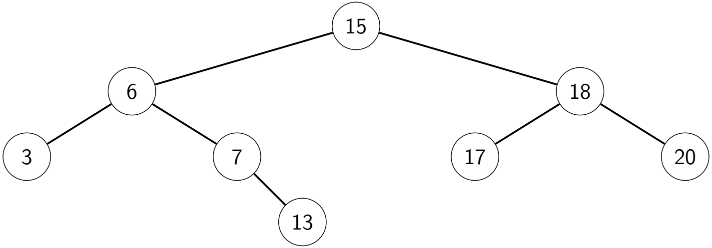


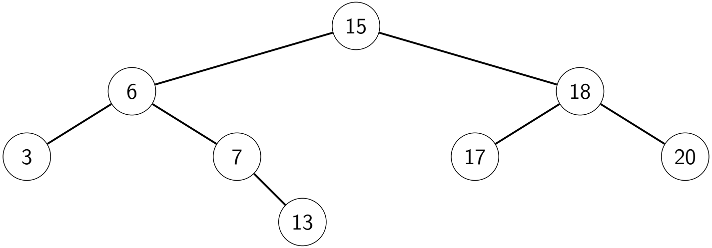
abb.insertar(elem)



`ultimo_nodo_buscado = abb.buscar_nodo(elem)`

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

abb.insertar(elem)

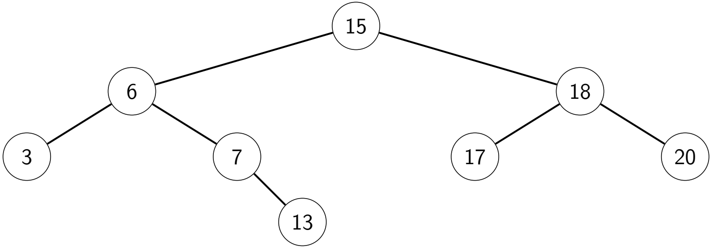


`ultimo_nodo_buscado = abb.buscar_nodo(elem)`

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

- SI lo encontramos, no hacemos nada.

abb.insertar(elem)

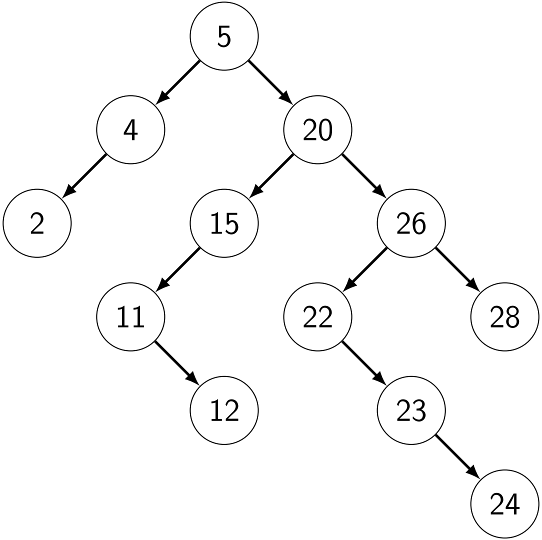


`ultimo_nodo_buscado = abb.buscar_nodo(elem)`

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

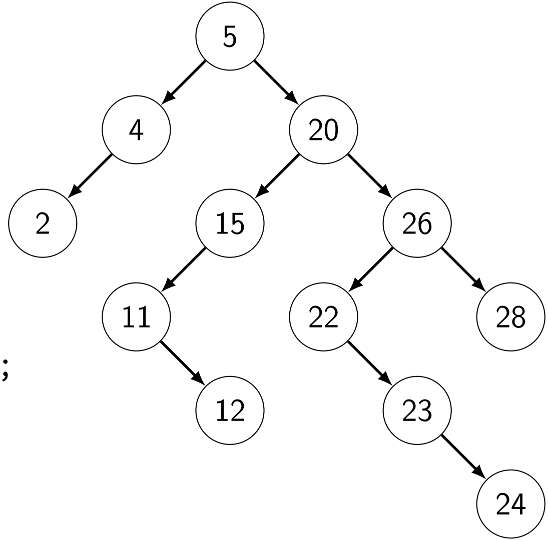
- `SI` lo encontramos, no hacemos nada.
- `SINO` lo insertamos como hijo del último nodo de la búsqueda.

`abb.eliminar(elem)`

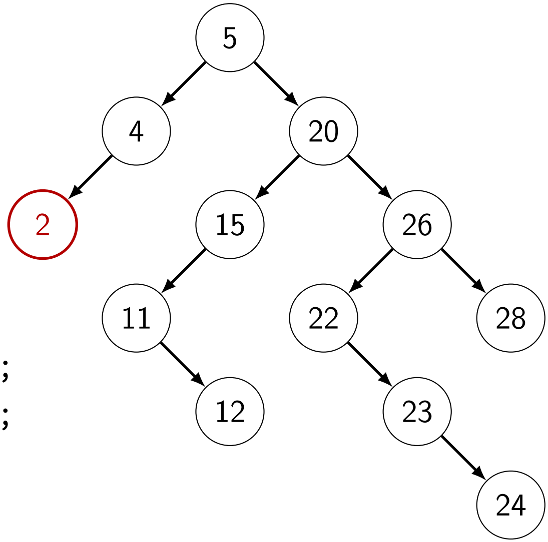


`abb.eliminar(elem)`

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

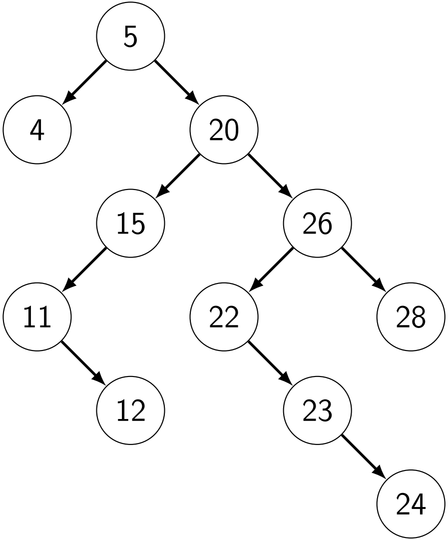


`abb.eliminar(elem)`



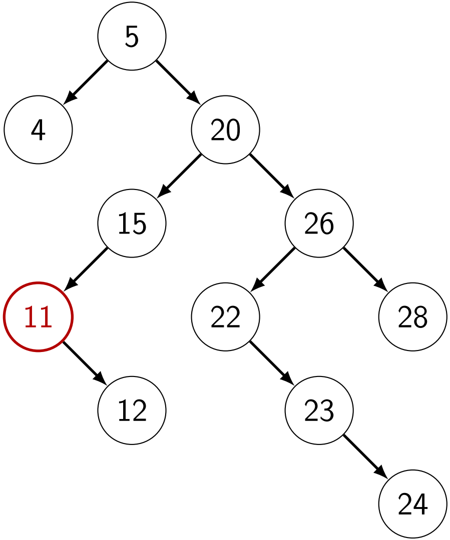

```
abb.eliminar(3);
abb.eliminar(2);
```

`abb.eliminar(elem)`



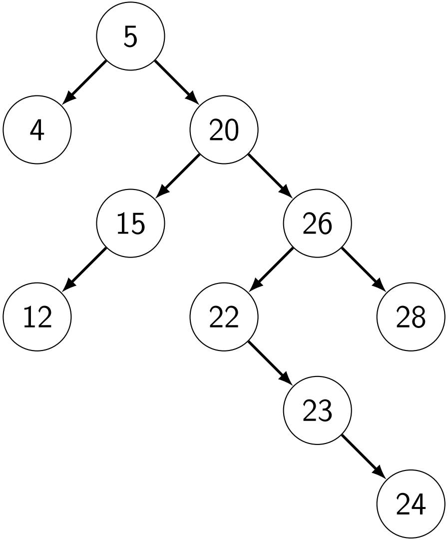

```
abb.eliminar(3);
abb.eliminar(2);
```

abb.eliminar(elem)



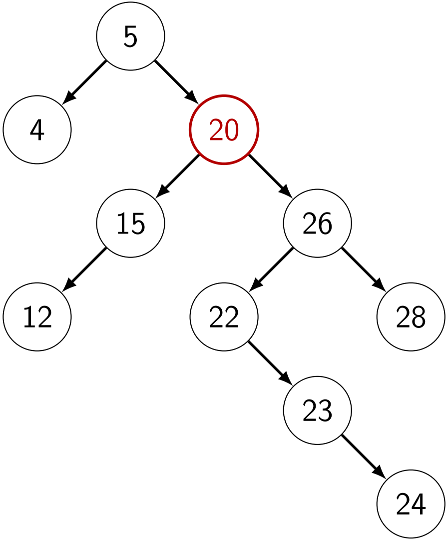

```
abb.eliminar(3);
abb.eliminar(2);
abb.eliminar(11);
```

abb.eliminar(elem)



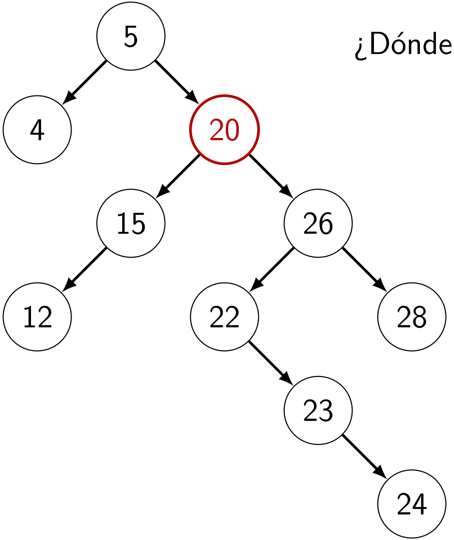

```
abb.eliminar(3);
abb.eliminar(2);
abb.eliminar(11);
```

`abb.eliminar(elem)`




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

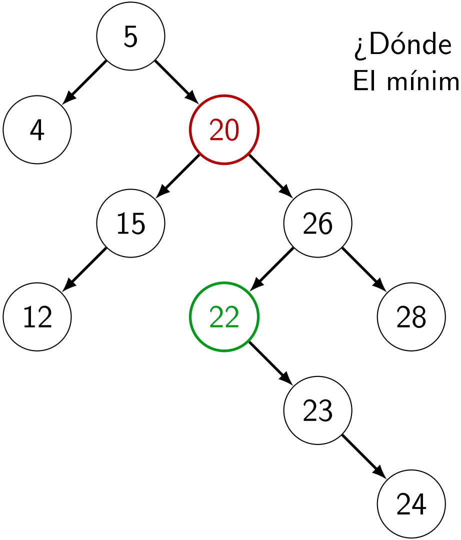
`abb.eliminar(elem)`



¿Dónde está el sucesor?

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

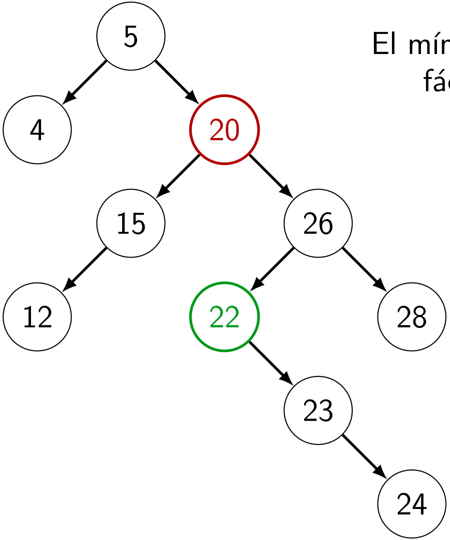
```
abb.eliminar(elem)
```



¿Dónde está el sucesor?
El mínimo de la derecha

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

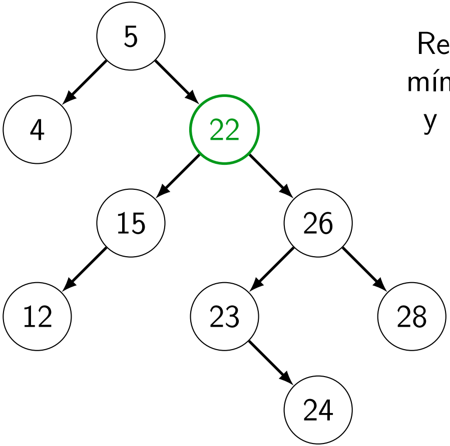
`abb.eliminar(elem)`



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.

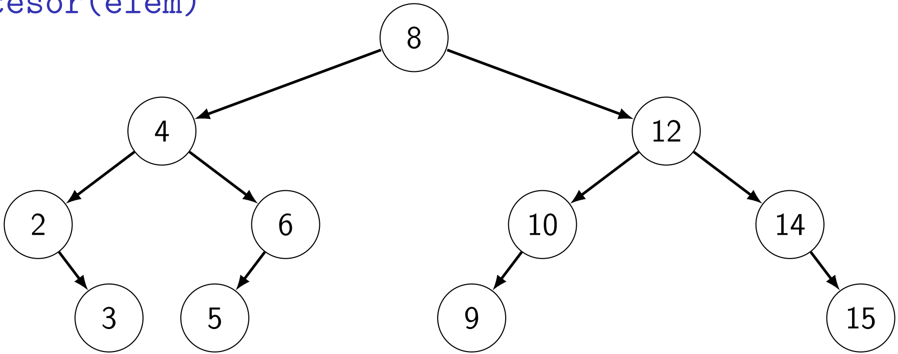
`abb.eliminar(elem)`

- 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.

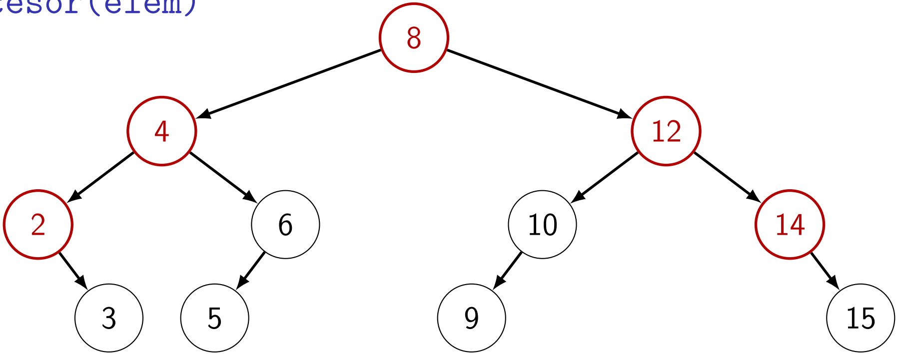
`abb.eliminar(elem)`

- 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)`

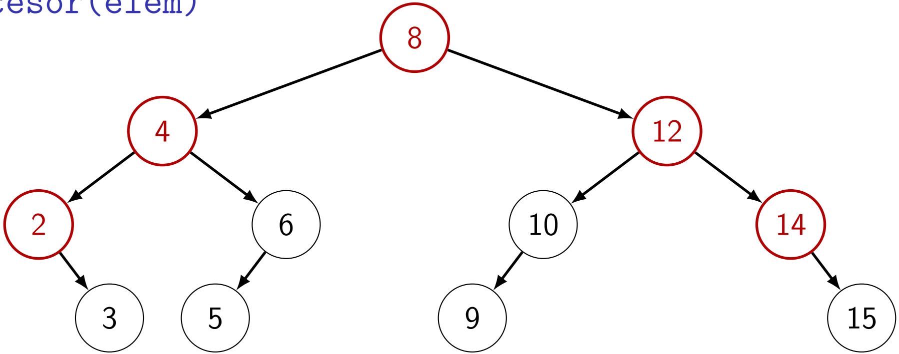


`abb.sucesor(elem)`



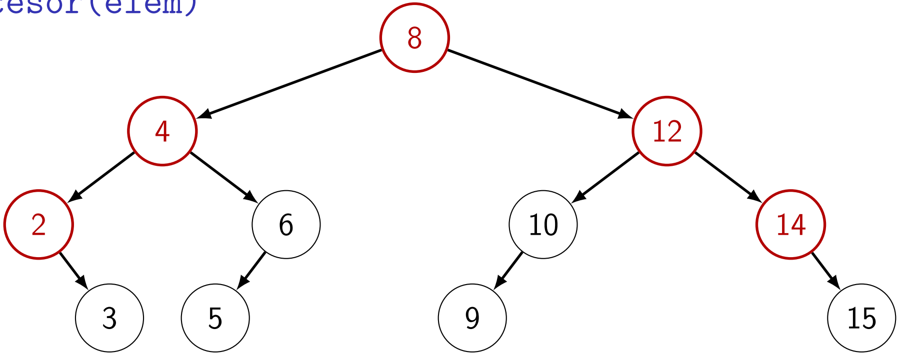
`Si tiene_subarbol_derecho(elem):`

`abb.sucesor(elem)`



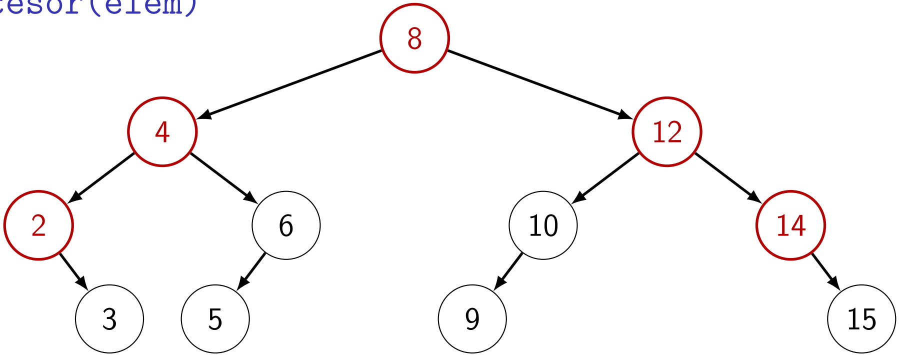

```
Si tiene_subarbol_derecho(elem):
    res = minimo_a_su_derecha(elem)
```

`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)
```

`abb.sucesor(elem)`: según capítulo 12 del Cormen

(`p(x)` es `raiz(x)`) ($\leftarrow$ 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
```

Iterador<T>

```java
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();
}
```

Iterador

Algoritmos Recursivo

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

Algoritmos Recursivo

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

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

Iterador

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

Complejidades

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

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

Complejidades

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

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

Complejidades

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

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

Complejidades

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

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

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

¡A programar!

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