Implementación ABB

```
...

función está?(a:abb; e:elemento) devuelve booleano
principio
    si a=nil entonces
        devuelve falso
    sino
        selección
            e<a↑.dato: devuelve está?(a↑.izq,e);
            e=a↑.dato: devuelve verdad;
            e>a↑.dato: devuelve está?(a↑.der,e)
        fselección
    fsi
fin
...
```

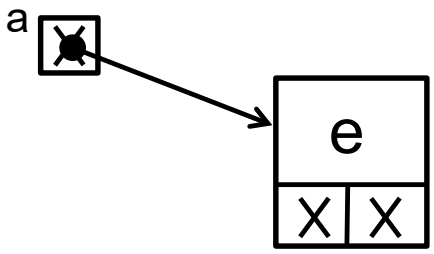
{Se recorre un camino desde la raíz: hasta encontrar el elemento buscado o llegar a árbol vacío…}

¿Coste?

En el caso peor: coste lineal en la altura del árbol

Implementación ABB

```
...
 procedimiento insertar(e/s a:abb; ent e:elemento)
 principio
   si a=nil entonces
      nuevoDato(a);
      a↑.dato:=e;
      a↑.izq:=nil;
      a↑.der:=nil
   sino
      si e≤a↑.dato entonces
         insertar(a↑.izq,e)
      sino
         insertar(a↑.der,e)
      fsi
    fsi
 fin
...
```



{Se recorre un camino desde la raíz, y el nuevo elemento se coloca como una nueva hoja…}

¿Cómo evitar elementos repetidos?

¿Coste?

En el caso peor: coste lineal en la altura del árbol

Implementación ABB

```
...
función min( a: abb) devuelve elemento
{Precondición: a no es un árbol vacío}
principio
    si a↑.izq=nil entonces
        devuelve (a↑.dato)
    sino
        devuelve min(a↑.izq)
    fsi
fin

función max(a: abb) devuelve elemento
{Precondición: a no es un árbol vacío}
principio
    si a↑.der=nil entonces
        devuelve (a↑.dato)
    sino
        devuelve max(a↑.der)
    fsi
fin
...
```

¿Coste?

En el caso peor: coste lineal en la altura del árbol

Implementación ABB

```
...
procedimiento borrar(e/s a:abb; ent e:elemento)
  variable aux:abb
 principio
   si a≠nil entonces
     selección
       e<a↑.dato: borrar(a↑.izq,e);
       a↑.dato<e: borrar(a↑.der,e);
       e=a↑.dato:
          si a↑.izq=nil entonces
               aux:=a; a:=a↑.der; disponer(aux)
          sino_si a↑.der=nil entonces
               aux:=a; a:=a↑.izq; disponer(aux)
          sino
               borrarMáximo(a↑.izq,a↑.dato)
          fsi
     fselección
   fsi
fin
```

{Se recorre un camino desde la raíz hasta encontrar el elemento buscado…

(si no está, llegaremos a un nodo, y al buscarlo en su hijo vacío acabamos)}

{… si lo encontramos hay que borrarlo, pero el árbol deberá seguir siendo un ABB}

{… localiza el máximo elemento en a↑.izq, lo devuelve (quedando almacenado en a↑.dato), y además modifica el árbol (a↑.izq) borrando dicho elemento}

¿Coste?

En el caso peor: coste lineal en la altura del árbol

Implementación ABB

```
...

procedimiento borrarMax(e/s a: abb; sal e:elemento)
{(Precondición: a no es un árbol vacío.) Devuelve en e un elemento
    máximo del árbol a, y además elimina ese elemento del árbol a.}
variable aux:abb
principio
    si a↑.der=nil entonces {la raíz de a es máximo en a}
        e:= a↑.dato;
        aux:= a;
        a:=a↑.izq;
        disponer(aux)
    sino {el máximo en a está en su hijo derecho}
        borrarMax(a↑.der,e)
    fsi
fin
...
```

¿Coste?

En el caso peor: coste lineal en la altura del árbol

Implementación ABB

```
procedimiento borrar_alternativo (e/s a:abb; ent e:elemento)
  variable aux:abb; auxMax:abb
 principio
   si a≠nil entonces
     selección
       e<a↑.dato: borrar_alternativo(a↑.izq,e);
       a↑.dato<e: borrar_alternativo(a↑.der,e);
       e=a↑.dato:
          si a↑.izq=nil entonces
              aux:=a; a:=a↑.der; disponer(aux)
         sino_si a↑.der=nil entonces
              aux:=a; a:=a↑.izq; disponer(aux)
         sino { localiza el nodo con máximo elemento
           en a↑.izq, lo desengancha del árbol a↑.izq
           y después colocamos ese nodo en lugar del que tiene el elemento a borrar: }
              desenganchar_Maximo(a↑.izq,auxMax);
              {le ponemos los hijos que tiene el nodo que vamos a eliminar: }
              auxMax↑.izq:=a↑.izq;  auxMax↑.der:=a↑.der;
              {intercambiamos en a el nodo que vamos a eliminar por auxMax : }
              aux:=a;  a:= auxMax;
              {liberamos la memoria del nodo que tiene el elemento a borrar}
              disponer(aux)
           fsi
     fselección
   fsi
 fin
```

```
procedimiento desenganchar_Maximo(e/s a: abb; sal maxNodo:abb)
{(Precondición: a no es un árbol vacío.) Devuelve en maxNodo el
puntero al nodo con el máximo elemento del árbol a, y además
desencadena ese nodo del árbol a (dejando a como un abb correcto.}
principio
      si a↑.der=nil entonces {la raíz de a tiene el máximo en a}
          maxNodo:= a;
          {desengancha su nodo del árbol a: }
          a:=a↑.izq;    maxNodo↑.izq := nil;
      sino {el máximo en a está en su hijo derecho}
          desenganchar_Maximo (a↑.der, maxNodo)
      fsi
fin
```

¿Coste? *Idéntico al de la primera implementación, aunque evita hacer copia (asignación) del elemento*

Con esta implementación, cualquier puntero que tuviésemos apuntando al nodo que hemos cambiado de sitio en el árbol, seguirá siendo válido y apuntando al mismo nodo (que tiene el mismo elemento que antes)