Árboles binarios de búsqueda.

Prof. Isabel Pita

Explicación de las transparencias de clase

Código desarrollado por el profesor A. Verdejo para la asignatura de Estructuras de Datos de la Facultad de Informática de la UCM.

5 de mayo de 2020

1. Árboles binarios de búsqueda.

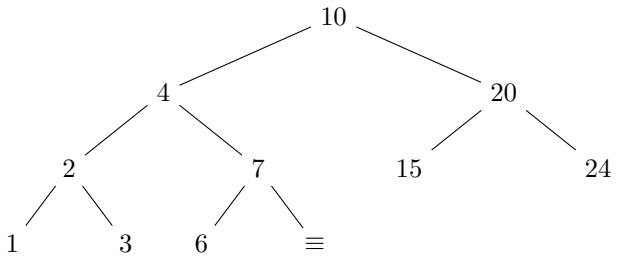
Los árboles binarios de búsqueda son árboles binarios cuyos nodos guardan elementos sobre los cuales hay definido un orden total estricto y que satisfacen la siguiente propiedad adicional: el elemento en cada nodo es mayor que todos sus descendientes izquierdos y menor que todos sus descendientes derechos.

Equivalentemente, o bien el árbol es vacío, o bien el elemento en la raíz es mayor que los elementos del hijo izquierdo y menor que los elementos del hijo derecho, y recursivamente los dos hijos son a su vez árboles binarios de búsqueda.

Los valores de los nodos de un árbol de búsqueda son todos diferentes.

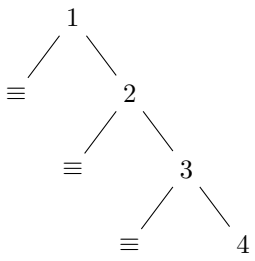
El siguiente árbol es un ejemplo de árbol binario de búsqueda.

1. Es un árbol binario.
2. El valor en la raíz es mayor que todos los elementos del hijo izquierdo (mayor que el máximo del hijo izquierdo).
3. El valor de la raíz es menor que todos los elementos del hijo derecho (menor que el mínimo del hijo derecho).
4. El hijo izquierdo es un árbol binario de búsqueda. Cumple a su vez estas 5 propiedades.
5. El hijo derecho es un árbol binario de búsqueda. Cumple a su vez estas 5 propiedades.

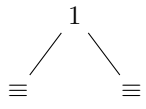


Ejercicio: Indicar cuales de los siguientes son árboles binarios de búsqueda

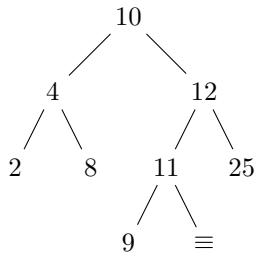
1. a)



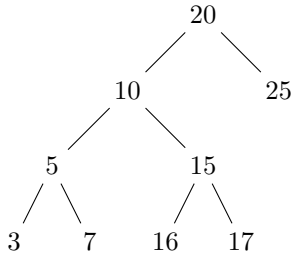
2. b)



3. c)



4. d)



2. Recorridos de un árbol binario de búsqueda

Observad que el recorrido en inorden de un árbol binario de búsqueda obtiene los elementos del árbol en orden. El recorrido en inorden del árbol anterior es: 1 2 3 4 6 7 10 15 20 24.

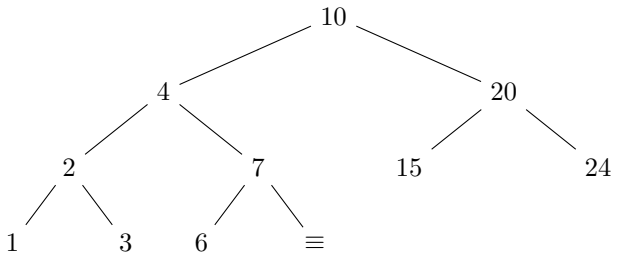
Por lo tanto el iterador del árbol binario que recorre el árbol en inorden, recorre los elementos en orden creciente de su valor.

3. Reconstrucción de un árbol binario de búsqueda

Los árboles binarios de búsqueda son árboles binarios, por lo tanto se pueden reconstruir a partir de los mismos recorridos que los árboles binarios: preorden e inorden y postorden e inorden.

¿Podemos prescindir de alguno de los recorridos y reconstruir un árbol binario de búsqueda a partir únicamente de un recorrido, aprovechando el orden entre los elementos que impone este tipo de árbol?

La respuesta es afirmativa. Observamos en recorrido en preorden del siguiente árbol de búsqueda:



Recorrido en preorden: 10 4 2 1 3 7 6 20 15 24.

El primero elemento del recorrido es la raíz del árbol. ¿Podemos identificar los elementos siguientes que pretenecen al hijo izquierdo y los que pertenecen al hijo derecho?. En los árboles binarios que no son de búsqueda, no se puede y tenemos que utilizar el recorrido en inorden para obtenerlo. Sin embargo en un árbol binario de búsqueda los elementos del hijo izquierdo son menores que la raíz y los elementos del hijo derecho son mayores que la raíz. Por lo tanto en la lista del preorden los elementos del hijo izquierdo son los menores que la raíz y los del hijo derecho los mayores. Los elementos menores están siempre antes que los mayores, porque es un recorrido en preorden.

Solución al ejercicio:

El árbol a) es un árbol binario de búsqueda. Es un árbol binario pues todos los nodos tienen cero o dos hijos, pudiendo ser uno de los hijos vacíos. Los nodos del lado izquierdo tienen valor menor que la

raíz, como no hay nodos en el hijo izquierdo se cumple esta propiedad y los nodos del hijo derecho tienen valor mayor que la raíz. El hijo izquierdo es un árbol binario de búsqueda porque es vacío y los árboles vacíos son binarios de búsqueda y el hijo derecho también es un árbol binario de búsqueda

El árbol b) es un árbol binario de búsqueda. Los hijos son vacíos, luego son árboles binarios de búsqueda y el valor de la raíz cumple las propiedades pedidas, porque no hay valor en los hijos.

El árbol c) no es de búsqueda porque el árbol derecho no cumple que todos sus nodos tengan valores menores que la raíz.

El árbol d) no es un árbol binario de búsqueda porque el árbol derecho del árbol izquierdo de la raíz no es un árbol de búsqueda.

Operaciones sobre árboles de búsqueda.

Prof. Isabel Pita

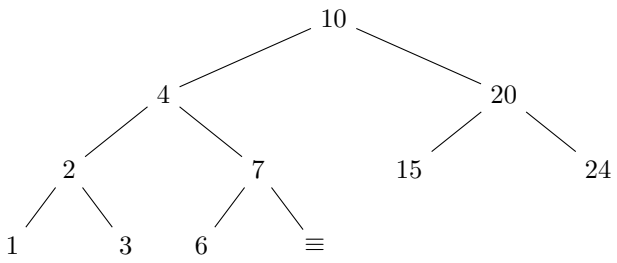
Explicación de las transparencias de clase

Código desarrollado por el profesor A. Verdejo para la asignatura de Estructuras de Datos de la Facultad de Informática de la UCM.

5 de mayo de 2020

1. Operaciones sobre árboles binarios de búsqueda.

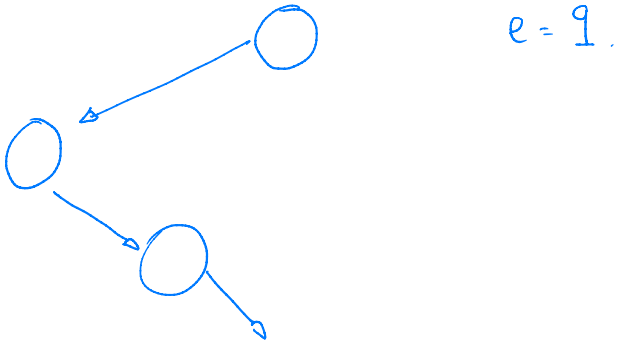
Definimos tres operaciones sobre los árboles binarios de búsqueda: buscar, insertar y eliminar un elemento.



Buscar un elemento en el árbol Para buscar un elemento en un árbol de búsqueda actuamos de forma recursiva.

- Si el árbol es vacío el elemento no se encuentra
- Si el elemento que buscamos es igual al valor de la raíz, ya lo hemos encontrado
- Si el elemento que buscamos es menor que el valor de la raíz, entonces tiene que estar en la parte izquierda del árbol, ya que todos los valores de la parte derecha son mayores que el valor en la raíz del árbol.
- Si el elemento que buscamos es mayor que el valor de la raíz, entonces tiene que estar en la parte derecha del árbol, ya que todos los valores de la parte izquierda son menores que el valor en la raíz del árbol.

Por ejemplo, en el árbol anterior, para buscar el valor 9, lo comparamos con la raíz del árbol. Como es menor que el 10, pasamos a buscar en la parte izquierda del árbol. Comparamos ahora el 9 con la raíz del árbol que es el 4, como el valor que buscamos es mayor pasamos a buscarlo a la parte derecha del árbol. Estamos ahora buscando en el árbol con raíz 7, cómo el 9 es mayor pasamos a buscarlo en la parte derecha del árbol con raíz 7, que es vacía. Por lo tanto el valor no está en el árbol.



La implementación se encuentra en el fichero `set_eda.h`. La operación pública llama a la operación privada con un `Link` a la raíz del árbol y el elemento a buscar. La operación privada sigue el esquema anterior para realizar la búsqueda.

```
bool pertenece(T const& e, Link a) const {
   if (a == nullptr) {
      return false;
   }
   else if (menor(e, a->elem)) {
      return pertenece(e, a->iz);
   }
   else if (menor(a->elem, e)) {
```

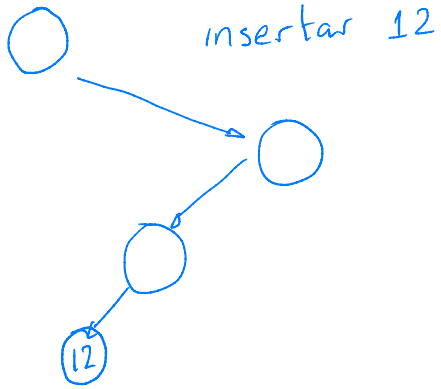
```
        return pertenece(e, a->dr);
    }
    else { // e == a->elem
        return true;
    }
}
```

Insertar un elemento en el árbol Los valores se insertan siempre como hojas del árbol, de esta forma no se pierda la propiedad sobre el orden de los valores del árbol.

La implementación busca en el árbol el lugar que le corresponde al valor a insertar. Esta búsqueda se realiza de la misma manera que la búsqueda de un elemento, se compara el valor a insertar con el valor de la raíz, si el valor a insertar es menor que el valor de la raíz se insertará en el árbol izquierdo, si el valor a insertar es mayor que el valor de la raíz debemos insertarlo en el árbol derecho. Si el valor coincide con el valor de la raíz, el elemento ya se encuentra en el árbol, por lo que no se inserta. Los valores de un árbol binario de búsqueda son todos diferentes. En este caso se devuelve `false` para indicar que el elemento no se ha insertado. Si el árbol es vacío se crea un nodo y se le da el valor. Al cambiar el valor del `Link` para que apunte al nodo, el nodo queda como hijo del nodo que hizo la llamada recursiva. Observad que el parámetro del `Link` es por referencia.

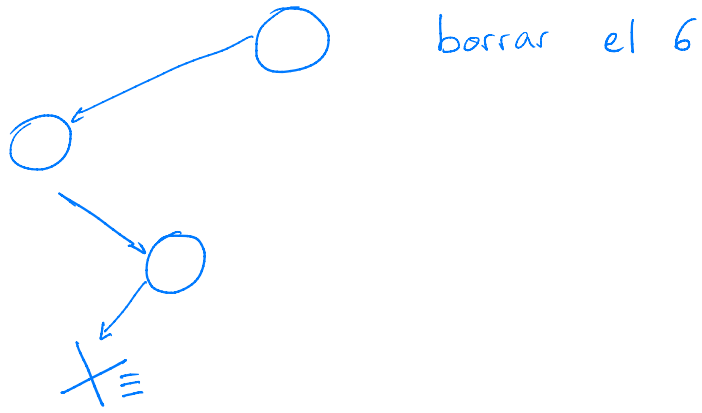
```
bool inserta(T const& e, Link & a) {
    if (a == nullptr) {
        a = new TreeNode(e);
        ++nelems;
        return true;
    }
    else if (menor(e, a->elem)) {
        return inserta(e, a->iz);
    }
    else if (menor(a->elem, e)) {
        return inserta(e, a->dr);
    }
    else // el elemento e ya esta en el arbol
        return false;
}
```

Por ejemplo, si queremos insertar el valor 12, comparamos con el valor de la raíz del árbol, como el valor a insertar es mayor, debe insertarse en la parte derecha del árbol. Comparamos a continuación el valor 12 con el 20, como es menor el valor debe insertarse en la parte izquierda del árbol. A continuación comparamos el valor con el 15, como es menor se inserta en la parte izquierda del árbol. Se llama a la función con el `Link` izquierdo del nodo 15. Este `Link` es el que toma el valor del nodo que se crea con el valor 12.

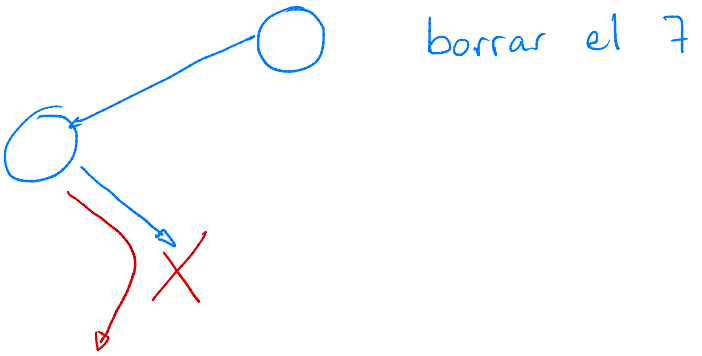


Eliminar un elemento en el árbol Al eliminar un valor del árbol debemos tener cuidado de no perder la propiedad sobre el orden de los valores del árbol. Diferenciamos varios casos

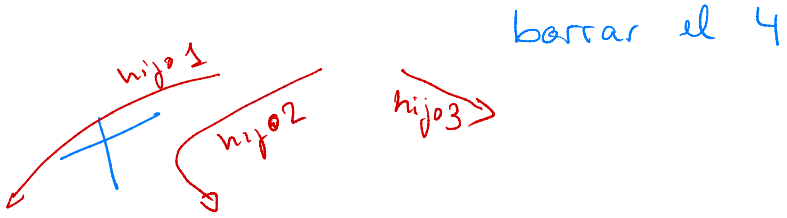
- Si el árbol es vacío el elemento no se encuentra y por lo tanto no se puede eliminar. Se devuelve el valor `false` para indicar que el valor no se ha borrado.
- Si el nodo a borrar el menor que el valor de la raíz del árbol, el elemento a borrar estará en la parte izquierda del árbol.
- Si el nodo a borrar es mayor que el valor de la raíz del árbol, el elemento a borrar estará en la parte derecha del árbol.
- Si el elemento a borrar coincide con el valor de la raíz del árbol, hemos localizado al elemento. Ahora tenemos varios casos:
- Si el elemento es una hoja, podemos eliminarlo y la propiedad sobre el orden de los valores del árbol se mantiene.



- Si el elemento tiene un sólo hijo observamos que si eliminamos el nodo y modificamos el link del padre para que tenga la dirección del único hijo el orden de los valores del árbol no se modifica.

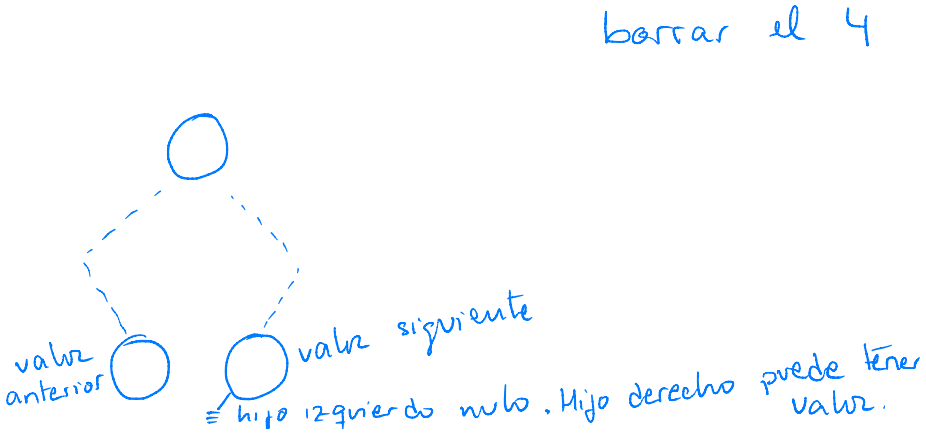


- Si el elemento tiene dos hijos, no podemos eliminarlo, porque obtendríamos un nodo con tres hijos.



Modificar la estructura del árbol para eliminar este nodo no es sencillo, por lo que se utiliza otro enfoque. Cambiaremos el valor del nodo a eliminar por otro valor del árbol que mantenga la propiedad del orden. Este valor puede ser o el siguiente en el orden o el anterior. Por costumbre se utiliza siempre el valor siguiente.

Es importante ver donde se encuentra el valor siguiente a un nodo que tiene dos hijos. Como es un valor mayor que el del nodo, estará en su hijo derecho. Además es el valor siguiente al del nodo, luego tiene que ser el menor valor del hijo derecho. Por lo tanto se encuentra en la rama de más a la izquierda del hijo derecho. Observamos también que la rama más a la izquierda es siempre un nodo con cero o un hijo, y por lo tanto podemos eliminarla sin alterar el orden de los valores del árbol.



Para evitar la copia de los valores de los nodos, en lugar de copiar el valor, modificaremos los valores de los links de forma que el nodo en que se guarda el valor siguiente al que vamos a eliminar quede *en el sitio* del que eliminamos. Una vez que hayamos cambiado los links, devolveremos la memoria del elemento a eliminar al sistema.

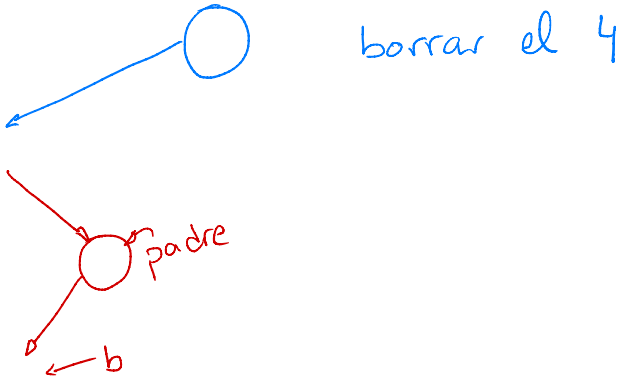
- Observamos en la implementación que primero se diferencian los casos que hemos visto antes. En el caso en que solo tiene un hijo `if (a->iz == nullptr || a->dr == nullptr)`, vemos que se modifica el valor del `Link` a, que es un parámetro por referencia, para darle el valor del hijo no nulo del nodo actual que es el dado por `a`.

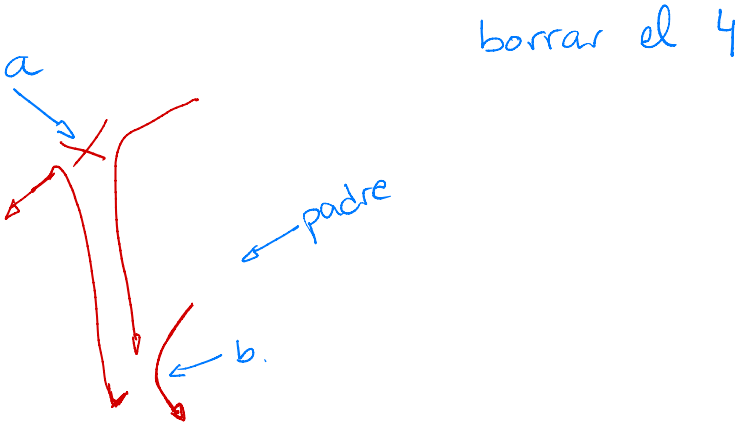
  Cuando los dos hijos no son nulos hacemos uso de una función auxiliar `subirMenorHD`

```
bool borra(T const& e, Link & a)  {
   if (a == nullptr)
      return false;
   else {
      if (menor(e, a->elem)) {
         return borra(e, a->iz);
      }
      else if (menor(a->elem, e)) {
```

```
            return borra(e, a->dr);
        }
        else { // e == a->elem
            if (a->iz == nullptr || a->dr == nullptr) {
                Link aux = a;
                a = (a->iz == nullptr) ? a->dr : a->iz;
                --nelems;
                delete aux;
            }
            else { // tiene dos hijos
                subirMenorHD(a);
                --nelems;
            }
            return true;
        }
    }
}
```

La función `subirMenorHD` recibe el `Link` con el nodo a eliminar. En el `Link b`, guarda el menor del hijo derecho que se obtiene por medio del bucle que accede al hijo izquierdo hasta que no hay más nodos en el hijo izquierdo. Al tiempo que busca el hijo izquierdo para guardarlo en el `Link b`, guarda en el `Link padre` al padre de `b`. Si el hijo derecho no tiene hijo izquierdo el `padre` tendrá el valor `nullptr`. Recordemos que el nodo a eliminar que es el apuntado por el `link a` siempre tiene hijo derecho, ya que partíamos del supuesto de que tiene los dos hijos.



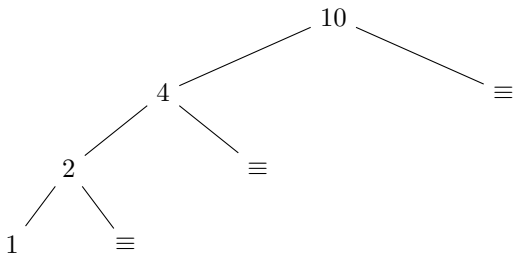



```
static void subirMenorHD(Link & a) {
    // Busca el menor del hijo derecho
    Link b = a->dr, padre = nullptr;
    while (b->iz != nullptr) {
        padre = b;
        b = b->iz;
```

```
        }
        // Modifica los links del arbol
        if (padre != nullptr) { // el nodo derecho tiene hijo izquierdo. Libera el nodo b de s
           padre->iz = b->dr;
           b->dr = a->dr;
        } // Si el nodo derecho no tiene hijo izquierdo el nodo b no necesita ser liberado de
        // Engancha la parte izquierda de a a la parte izquierda de b
        b->iz = a->iz;
        delete a; // Devuelve la memoria del nodo a eliminar
        a = b;  // Da el nuevo valor al link
    }
```

Coste de las operaciones de buscar, insertar y eliminar un elemento Todas las operaciones tienen un coste lineal respecto a la altura del árbol, ya que en cada llamada recursiva se elige uno de los dos lados del árbol, desechando el otro lado. $\mathcal{O}(h)$ siendo h la altura del árbol.

En el caso peor, la altura es lineal respecto al número de nodos. Esto se produce si el árbol es degenerado, es decir casi todos los nodos están en el mismo lado del árbol. Esto produce que una llamada recursiva se haga casi con todos los nodos del árbol, mientras que la otra se realiza con 0 o muy pocos nodos.



La recurrencia que define el coste, siendo n el número de nodos es:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ T(n-1) + c_1 & \text{si } n > 0 \end{cases}$$

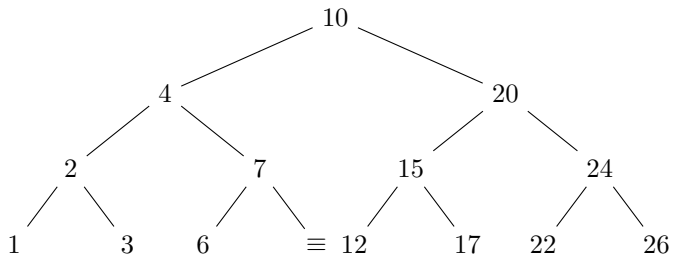
Y si hacemos el desplegado vemos que $T(n) \in \mathcal{O}(n)$

En promedio, la altura es logarítmica respecto al número de nodos. Si el árbol está equilibrado, aproximadamente la mitad de los nodos estarán en el hijo izquierdo y la otra mitad en el hijo derecho. Por lo tanto en cada llamada recursiva desechamos la mitad de los nodos del árbol. Por ello el número de llamadas recursivas es del orden del $\log n$ siendo n el número de nodos. Hay que tener en cuenta que el coste de cada llamada recursiva es consante.

El coste se puede obtener por medio de la recurrencia donde n es el número de nodos, ya que tenemos dos llamadas recurssivas con aproximadamente la mitad de los nodos y el coste de cada llamada recursiva es constante:

$$T(n) = \begin{cases} c_0 & \text{si } n = 1 \\ T(\frac{n}{2}) + c_1 & \text{si } n > 1 \end{cases}$$

Y al realizar el desplegado vemos que el coste es $T(n) \in \mathcal{O}(\log n)$.



Ejercicio: Sobre el siguiente árbol binario de búsqueda realizar las siguientes operaciones

- Insertar el nodo 50
- Insertar el nodo 32
- Borrar el nodo 35
- Borrar el nodo 40
- Borrar el nodo 20
- Borrar el nodo 30