Guía para la implementación de un árbol AVL

Programación y Estructuras de Datos

Curso 2019-2020

Universidad de Alicante

Elaborado por Víctor M. Sánchez Cartagena

Esta guía pretende facilitar al alumnado la realización del cuadernillo 3 de la asignatura Programación y Estructuras de Datos. Los consejos que aquí se recogen no son de obligado cumplimiento. El alumnado es libre de implementar la práctica de la manera que considere. Los únicos requisitos son los que se recogen en el enunciado del cuadernillo 3.

En el cuadernillo 3 se propone la implementación de un árbol equilibrado con respecto a la altura (AVL). Los métodos tienen exactamente la misma interfaz (parte pública) que en el cuadernillo 2.[1] La única diferencia de comportamiento radica en que, tras una inserción o un borrado, se podrán producir rotaciones.

Por tanto, es recomendable que, antes de comenzar con el cuadernillo 3, el alumno se haya asegurado de que su cuadernillo 2 pase todas las pruebas publicadas. Después, el primer paso consistiría en hacer una copia del cuadernillo 2, renombrar los atributos y las clases según el enunciado del cuadernillo 3, y comprobar que todo sigue funcionando correctamente.

La implementación de un árbol AVL partiendo del código del cuadernillo 2 implica dos tareas adicionales: mantenimiento de los factores de equilibrio y rotaciones.

Mantenimiento de factores de equilibrio

Diseño

Para poder asegurarnos de que el árbol siempre está equilibrado con respecto a la altura, es necesario conocer el factor de equilibrio de cada nodo, que se almacenará en el atributo `fe` de la clase `TNodoAVL`.

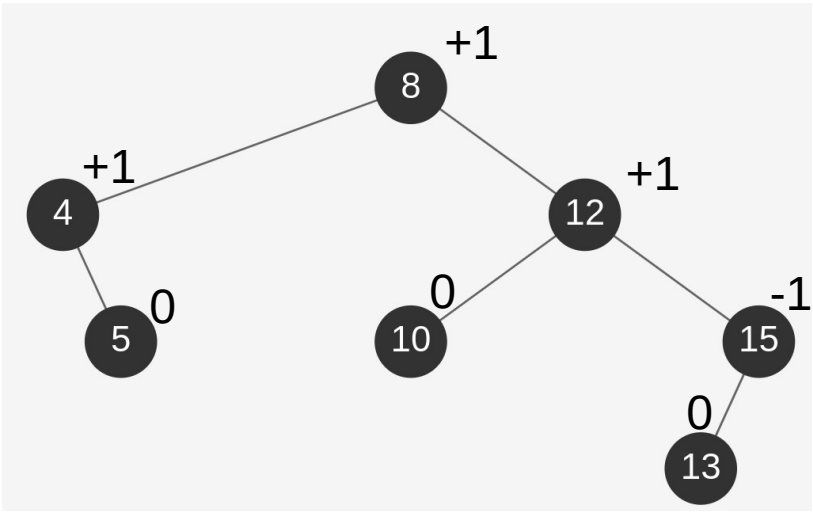
No será necesario calcular el factor de equilibrio para cada nodo mediante el cómputo directo de la altura de sus dos hijos. Será suficiente con establecer el factor de equilibrio a 0 para los nuevos nodos hoja, y actualizar dicho factor en el camino desde el punto donde se ha realizado una inserción/borrado hasta la raíz. Este camino se recorrerá de manera recursiva, pero no será necesario implementar ninguna llamada recursiva adicional a las implementadas en el cuadernillo 2. Simplemente, habrá que incluir código que actualice el factor de equilibrio tras cada llamada recursiva existente a los métodos `Insertar`/`Borrar`.[2] Para actualizar dicho factor de equilibrio, es necesario conocer si la altura del hijo ha crecido cuando estamos insertando o si ha decrecido cuando estamos borrando.

---

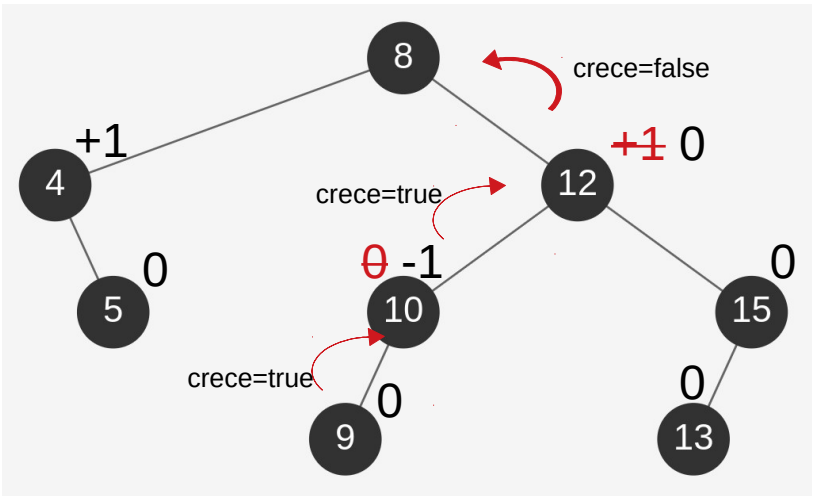
1 En el cuadernillo 3 se ha añadido el método `bool operator!=( TAVLCom &)`.

2 Veremos más adelante en este documento que, en realidad, los métodos a los que llamaremos recursivamente serán `InsertarAux` y `BorrarAux`.

Por ejemplo, en el siguiente árbol AVL, con los factores de equilibrio ya calculados, vamos a insertar el número “9”:



El siguiente esquema muestra cómo los factores de equilibrio se actualizarían desde el nodo hoja donde se ha insertado el elemento hasta la raíz. La variable `crece` nos indica si cada subárbol ha crecido en altura tras la inserción. Por ejemplo, el subárbol donde se ha insertado el 9 ha crecido en altura, pues antes estaba vacío; el subárbol cuya raíz es 10 también ha crecido. Sin embargo, el subárbol cuya raíz es 12 no ha crecido en altura: sigue teniendo altura 3 tras la inserción.



Actualizar el factor de equilibrio de un nodo y saber si el subárbol cuya raíz es ese nodo ha crecido en altura tras una inserción es sencillo si contamos con la información de cuál de los hijos ha crecido en altura. Se pueden dar los siguientes casos (FE es el factor de equilibrio, “crece” indica si el subárbol ha crecido en altura tras la inserción):

- ninguno de los hijos ha crecido: FE no cambia, no crece
- el hijo izquierdo ha crecido y FE=1, o el hijo derecho ha crecido y FE=-1: FE=0, no crece
- el hijo izquierdo ha crecido y FE=0: FE=-1, crece
- el hijo derecho ha crecido y FE=0: FE=1, crece
- el hijo izquierdo ha crecido y FE=-1: FE=**-2** (debe realizarse una rotación)
- el hijo derecho ha crecido y FE=1: FE=**2** (debe realizarse una rotación)

En los dos últimos casos, el factor de equilibrio no llegará a actualizarse, se realizará una rotación y el factor de equilibrio resultante y crecimiento del árbol dependerán de la rotación.

La actualización de factores de equilibrio tras un borrado se realizará de manera parecida.

Implementación

A continuación se describen las modificaciones que habría que realizar en el código del cuadernillo 2 para actualizar los factores de equilibrio tras una inserción. La implementación de la actualización de factores de equilibrio tras un borrado se deja a criterio del alumnado.

El factor de equilibrio deberá inicializarse a 0 en el constructor por defecto de `TNodoAVL:` un nodo hoja siempre tendrá un factor de equilibrio 0 porque la diferencia entre las alturas de sus hijos es 0 (0 – 0 = 0).

Se recomienda que la transmisión de la información sobre si un subárbol ha crecido se realice añadiendo un parámetro adicional pasado por referencia llamado `crece` al método `Insertar`. El método escribirá en `crece` el valor `true` si la altura del árbol ha crecido tras la inserción, y `false` en caso contrario. El valor de retorno del método mantendrá el mismo significado que en el cuadernillo 2: indicará si se ha podido realizar la inserción.

Para cumplir con las condiciones establecidas en el enunciado, tendremos que crear un método nuevo `InsertarAux` con el parámetro adicional en lugar de añadir el parámetro directamente a `Insertar`. El método `Insertar` ahora sólo contendrá una llamada a `InsertarAux`, como se muestra a continuación:

```
bool TAVLCom::Insertar(TComplejo &c){
  bool crece=false;
  return InsertarAux(c,crece);
}
```

Un buen punto de partida para `InsertarAux` es el método `Insertar` del cuadernillo 2, pues ya contiene la gestión de todos los casos posibles: que el elemento a insertar se encuentre ya en el árbol, que el árbol esté vacío y por tanto se cree un nuevo nodo, o que haya que hacer una llamada recursiva para insertar el elemento en el subárbol derecho o en el subárbol izquierdo.

En el caso de inserción en un árbol vacío, simplemente habrá que asignar a `crece` el valor `true` y poner el factor de equilibrio a 0 (debería hacerlo el constructor por defecto de `TNodoAVL`).

En el caso de tener que hacer una llamada recursiva, habrá que incluir el código para actualizar el factor de equilibrio. El siguiente fragmento de código muestra un ejemplo de las modificaciones que deben realizarse tanto en la llamada recursiva como después de esta. Las variables booleaneas `creceHijoIz` y `creceHijoDe` nos indican cuál de los dos hijos ha crecido en altura (si es que lo ha hecho alguno). El fragmento de código que empieza con `if (creceHijoIz || creceHijoDe)` actualiza el factor de equilibrio según los casos discutidos anteriormente, aunque el ejemplo sólo contiene la implementación de los dos primeros.

```
bool TAVLCom::InsertarAux(TComplejo &c, bool &crece){

   bool creceHijoIz=false, creceHijoDe=false, insercionOK=false;

// ...
//si el árbol no está vacío
   //si c debe insertarse en el hijo izquierdo
   insercionOK=nodo->iz.InsertarAux( c , creceHijoIz);

   //si c debe insertarse en el hijo derecho
   insercionOK=nodo->de.InsertarAux( c , creceHijoDe);

  //actualización de factores de equilibrio
  // solo es necesario actualizar si uno de los hijos ha crecido
   if (creceHijoIz || creceHijoDe){
     if ( (creceHijoIz && nodo->fe ==1) || (creceHijoDe && nodo-
>fe ==-1) ) {
        //Le indicamos al padre que este subárbol no ha crecido
        crece=false;
        nodo->fe=0;
     }
     //..

  }else{
     //Le indicamos al padre que este subárbol no ha crecido
     crece=false;
  }

  // ...

}
```

Rotaciones

Con las indicaciones dadas hasta ahora en este documento, es posible tener siempre actualizados los factores de equilibrio de todos los nodos del árbol, pero no realizar las rotaciones necesarias cuando un nodo tiene un factor de equilibrio no permitido en un árbol equilibrado con respecto a la altura. En esta sección, primero se presenta un algoritmo en pseudocódigo para realizar rotaciones y después se detalla cómo implementar ese algoritmo en el cuadernillo 3 e integrarlo con el código de gestión de factores de equilibrio descrito anteriormente.

Diseño

A continuación se presenta el pseudocódigo[3] de un algoritmo que realiza la rotación correspondiente en un nodo que, tras una inserción, tiene un factor de equilibrio de -2. Es decir, realiza una rotación II o una rotación ID. Tanto las rotaciones DD y DI tras una inserción como las rotaciones tras un borrado guardan un gran parecido con este algoritmo.

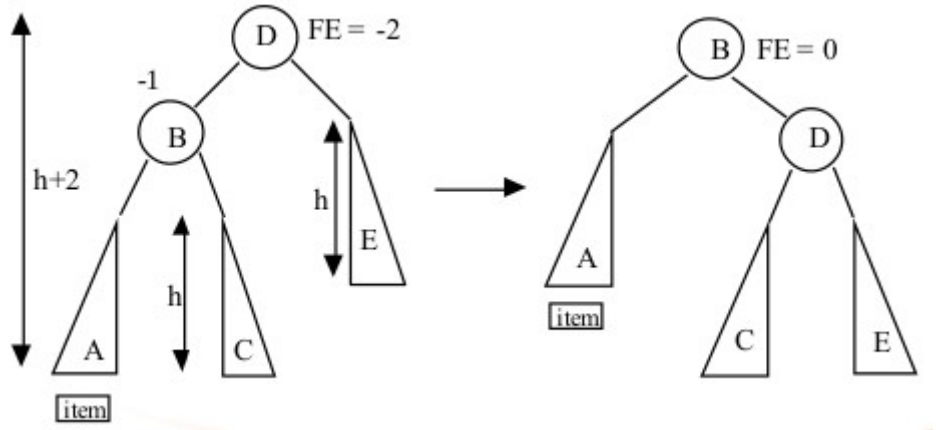
En este pseudocódigo, utilizaremos *iteradores* para representar *referencias* a ciertos nodos de un árbol. `I` es un iterador que apunta al nodo que tiene factor de equilibrio -2 tras realizar una inserción y `J` y `K` son iteradores auxiliares. `FE` es una función que devuelve el factor de equilibrio del nodo apuntado por el iterador, mientras que `HijoIzq` e `HijoDe` devuelven un iterador al hijo izquierdo o hijo derecho, respectivamente, del nodo apuntado por el iterador que se le pasa como parámetro.

3 Este pseucodcódigo, junto con el del algoritmo para el mantenimiento de los factores de equilibrio, puede encontrarse en las diapositivas del tema 3.2: Árboles AVL.

La función `Mover`[4] mueve el nodo apuntado por el segundo iterador que se le pasa como parámetro al lugar del árbol al que apunta el primer iterador. Inicialmente, los iteradores auxiliares `J` y `K` apuntan a árboles vacíos.

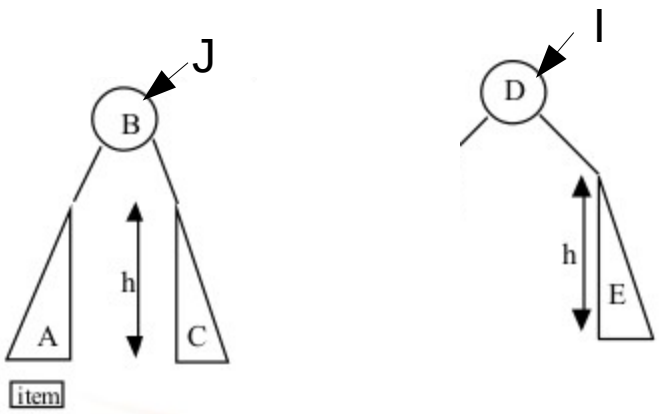
```
VAR I, J, K: Iterador; E2: int;

si ( FE (HijoIzq (I) = −1 entonces
  //ROTACIÓN II
  Mover (J, HijoIzq (I));
  Mover (HijoIzq (I), HijoDer(J));
  Mover (HijoDer (J), I);
  FE (J) = 0; FE (HijoDer (J))=0;
  Mover (I,J);
sino
  //ROTACIÓN ID
  Mover (J, HijoIzq (I));
  Mover (K, HijoDer (J));
  E2 = FE (K);
  Mover (HijoIzq (I), HijoDer(K));
  Mover (HijoDer (J), HijoIzq(K));
  Mover (HijoIzq (K), J);
  Mover (HijoDer (K), I);
  FE (K) = 0;

  caso de E2
   −1: FE (HijoIzq (K)) = 0; FE (HijoDer (K)) = 1;
   +1: FE (HijoIzq (K)) = −1; FE (HijoDer (K)) = 0;
   0: FE (HijoIzq (K)) = 0; FE (HijoDer (K)) = 0;
  fcaso
  Mover (I, K);
fsi
```

El primer bloque de código dentro de la instrucción `if` realiza una rotación II. A continuación se detallan las implicaciones de cada paso comparando con el esquema de rotación visto en clase de teoría:

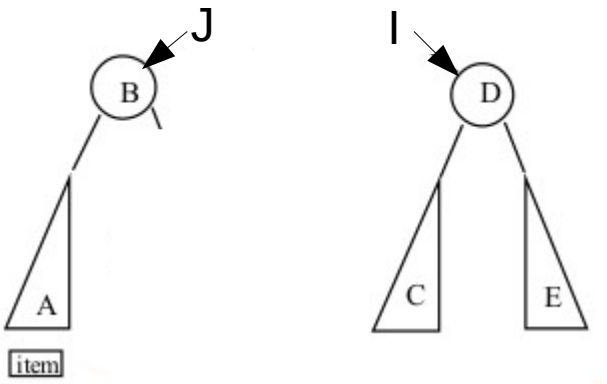


4 Pueden encontrarse más ejemplos del funcionamiento de la función `Mover` en las diapositivas del tema 3.1: Árboles.

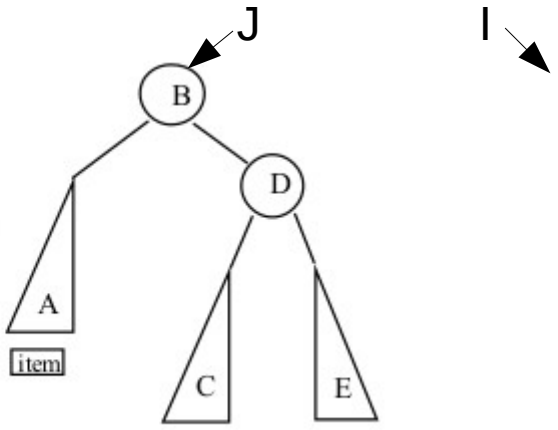
El nodo apuntado por el iterador `I` es `D`, la raíz del subárbol que va a rotarse. `Mover (J, HijoIzq (I))` hace que el hijo izquierdo de `D` (es decir el subárbol cuya raíz es `B`) se elimine del árbol y pase a estar apuntado por el iterador auxiliar `J`, como se muestra en el siguiente dibujo:



Después, `Mover (HijoIzq (I), HijoDer(J))` hace que el hijo derecho del nodo apuntado por `J` pase a ser el hijo izquierdo del nodo apuntado por `I`. El nodo apuntado por `J` es `B` y su hijo derecho es `C`. Como resultado, `C` queda colocado como hijo izquierdo de `D`:



Finalmente, `Mover (HijoDer (J), I)` hace que el subárbol cuya raíz es `D` pase a ser el hijo derecho de `B`. Con este paso hemos conseguido construir el subárbol resultado de la rotación, que está apuntado por el iterador auxiliar `J`:



Como consecuencia de los movimientos que hemos hecho, el lugar en el árbol principal donde debería estar el subárbol rotado está vacío. Finalmente, sólo nos queda mover este árbol rotado a su lugar en el árbol original con `Mover (I,J)`.

Implementación

De nuevo, vamos a detallar cómo se implementarían las rotaciones tras una inserción. En el bloque de código donde actualizamos los factores de equilibrio si la altura de un hijo ha crecido, se puede dar el caso de que un factor de equilibrio pase a ser +2 o -2. En ese caso, habrá que realizar una rotación. Por ejemplo, si el hijo izquierdo ha crecido y FE=-1:

```
if (creceHijoIz && nodo->fe ==-1){
    EquilibrarIzquierda()
    crece=false;
}
```

El método `EquilibrarIzquierda` realiza una rotación II o ID, dependiendo del factor de equilibrio del hijo izquierdo del árbol que lo invoca e implementa el pseudocódigo que se acaba de describir. Las rotaciones se llevarán a cabo cambiando el puntero `nodo` de `TAVLCom`. No será necesario liberar ni reservar memoria. Por supuesto, también será necesario implementar un método `EquilibrarDerecha`.

En el caso de los borrados, además de implementar métodos para realizar la rotación, habrá que tener en cuenta que la altura del árbol decrecerá o no dependiendo del tipo de rotación y, por tanto, habrá que actualizar la variable pasada por referencia que indica si la altura del árbol ha decrecido.

Los iteradores utilizados en el pseudocódigo se pueden implementar como punteros a las clases `TNodoAVL` o `TAVLCom`. El siguiente fragmento de código C++ muestra cómo se implementarían las primeras instrucciones del algoritmo de rotación empleando punteros a `TNodoAVL` para implementar los iteradores. `I` es el puntero `nodo` de `TAVLCom`, mientras que `J` y `K` serán punteros auxiliares que deberemos declarar.

```
void TAVLCom::EquilibrarIzquierda(){
     TNodoAVL* j,k;
     if (nodo->iz.nodo->fe == -1){
          //Mover (J, HijoIzq (I));
          j=nodo->iz.nodo;
          nodo->iz.nodo=NULL;

          //Mover (HijoIzq (I), HijoDer(J));
          nodo->iz.nodo=j->de.nodo;
          j->de.nodo=NULL;

          // ...
     }


}
```