ARBOLES BALANCEADOS

La búsqueda más eficiente se efectúa en un árbol binario balanceado. Desafortunadamente, la función Inserta no asegura que el árbol permanezca balanceado, el grado de balance depende del orden del orden en que son insertados los nodos en el árbol.

La altura de un árbol binario es el nivel máximo de sus hojas (profundidad). La altura del árbol nulo se define como –1. Un árbol binario balanceado es un árbol binario en el cual las alturas de los dos subárboles de todo nodo difiere a lo sumo en 1. El balance de un nodo en un árbol binario se define como la altura de su subárbol izquierdo menos la altura de su subárbol derecho. Cada nodo en un árbol binario balanceado tiene balance igual a 1, -1 o 0, dependiendo de si la altura de su subárbol izquierdo es mayor que, menor que o igual a la altura de su subárbol derecho.

Supóngase que tenemos un árbol binario balanceado, y usamos la función para insertar un nodo en dicho árbol. Entonces el árbol resultante puede o no permanecer balanceado.

Es fácil ver que el árbol se vuelve desbalanceado si y solo si el nodo recién insertado es un descendiente izquierdo de un nodo que tenia de manera previa balance de 1, o si es un hijo derecho descendiente de un nodo que tenia de manera previa balance –1.

Para que el árbol se mantenga balanceado es necesario realizar una transformación en el mismo de manera que:

1. El recorrido en orden del árbol transformado sea el mismo que para el árbol original (es decir, que el árbol transformado siga siendo un árbol de búsqueda binaria).
2. El árbol transformado esté balanceado.

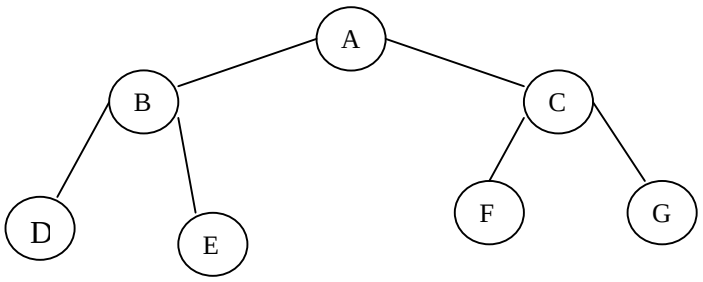
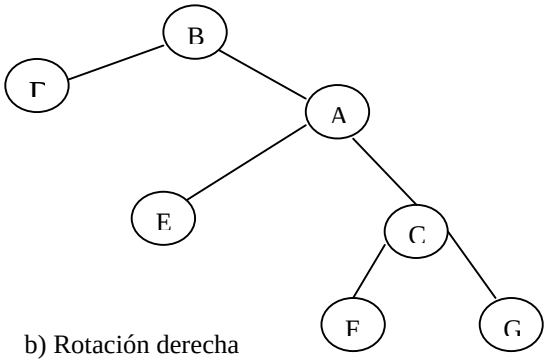
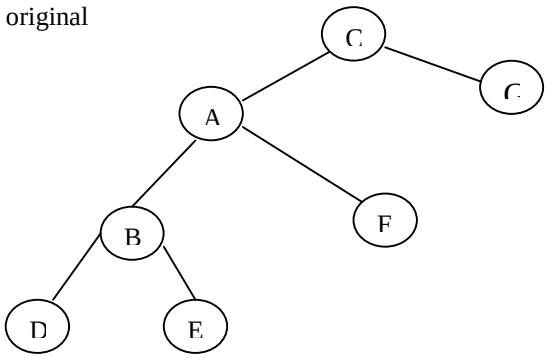


Figura 1

a) Árbol original



b) Rotación derecha



c) Rotación izquierda

El árbol de la figura 1b es una rotación derecha del árbol con raíz en A de manera similar, el árbol de la figura 1c se dice que es una rotación izquierda del árbol con raíz en A.

Un algoritmo para implantar una rotación izquierda de un subárbol con raíz en *p* es el siguiente:

```
q= right(p);
hold=left(q);
left(q)=p;
right(p)=hold;
```

Llamemos a esta operación *leftrotation(p). Rightrotation(P),* puede definirse de manera similar. Por supuesto, en cualquier rotación debe cambiarse el valor de la raíz del subárbol que está siendo rotado para que apunte a la nueva raíz. (En el caso de la rotación izquierda anterior, esta nueva raíz es q). Obsérvese que el orden de los nodos en un recorrido en orden se preserva en ambas rotaciones: izquierda y derecha. Por consiguiente, se deduce que cualquier número de rotaciones (izquierda o derecha) pueden ejecutarse en un árbol desbalanceado para obtener uno balanceado, sin perturbar el orden de los nodos en un recorrido en orden.

Supóngase que se realiza una rotación derecha en el subárbol con raíz en A de la figura 2 a. El árbol resultante se muestra en la figura 3 a. Obsérvese que el árbol de la figura 3 a produce el mismo recorrido en orden que el de la figura 2 a y también está balanceado. También, como la altura del subárbol de la figura 2 a era n+2 antes de la inserción y la altura del subárbol de la figura 3 a es n+2 con el nodo insertado, el balance de cada ancestro del nodo A no se altera. Así, reemplazando el subárbol de la figura 2ª por su rotación derecha de la figura 3ª, garantizamos que se mantenga como árbol de búsqueda binaria balanceado.

En la figura 2b donde el nodo recién creado se inserta en el subárbol derecho del B. Sea C el hijo derecho de B. (Hay tres casos: C puede ser el nodo recién insertado en cuyo caso n=-1, o el nodo recién insertado puede estar en el subárbol derecho o izquierdo de C. La figura 2b ilustra el caso en que está en el subárbol izquierdo; el análisis de los otros casos es análogo). Supóngase que una rotación izquierda del subárbol con raíz en B precede a una rotación derecha del subárbol con raíz en A. La figura 3b ilustra el árbol resultante.

El siguiente algoritmo busca e inserta en un árbol binario balanceado que no esté vacío. Cada nodo del árbol contiene 5 campos: k y r, que guardan la llave y el registro de manera respectiva left y right son apuntadores a los subárboles izquierdo y derecho de manera respectiva y bal, cuyos valores es 1, -1 o 0, dependiendo del balance del nodo. En la primera parte del algoritmo, si la llave deseada aún no se encuentra en el árbol, se inserta un nuevo nodo en el árbol de búsqueda binario, sin importar el balance. La primera fase también toma en cuenta al ancestro más joven, ya que puede desbalancearse tras la inserción. El algoritmo hace uso de la función maketree descrita con anterioridad y de las rutinas rigthrotation y leftrotation, que aceptan un apuntador a la raíz de un subárbol y ejecutan la rotación deseada.

La altura máxima de un árbol de búsqueda binaria balanceado es 1.44log2n, de manera que una búsqueda en un árbol así nunca requiere más de 44% de comparaciones que las necesarias en un árbol balanceado de manera completa. En la práctica los árboles de búsqueda binaria balanceados se comportan aún mejor, produciendo tiempos de búsqueda del orden de log2n +0.25 para valores grandes de n. En promedio se requiere una rotación en el 46.5 % de las inserciones.

El algoritmo para eliminar un nodo de un árbol de búsqueda binaria balanceado conservando su balance l es aún más complejo. Mientras que la inserción requiere a lo sumo una rotación doble, la eliminación puede requerir una rotación (doble o simple) en cada nivel del árbol o O(log n) rotaciones. Sin embargo, en la práctica, se ha visto que solo se requiere un promedio de 0.214 rotaciones (simples o dobles) por eliminación.

Los árboles de búsqueda binaria balanceados que hemos visto se llaman **árboles de altura balanceada** porque su altura se usa como criterio para el balanceo. Un método, se define **peso** del árbol como el número de nodos externos en el mismo (que es igual al número de apuntadores nulos). Si el cociente del peso del subárbol izquierdo de todo nodo entre el peso del subárbol con raíz en el nodo está entre alguna fracción a y 1-a, el árbol es un **árbol de pesos balanceados de razón a** o se dice que está en la clase wb[a]. Cuando una inserción o eliminación ordinaria en un árbol de clase wb[a] elimina al árbol de dicha clase, se usan rotaciones para restaurar la propiedad de pesos balanceados.

Otro tipo de árbol llamado por Tarjan, un árbol binario balanceado requiere que para todo nodo nd, la longitud del camino más largo de nd a un nodo externo se a lo sumo dos veces la longitud del camino más corto de nd a un nodo externo (recuérdese que los nodos externos son nodos agregados al árbol en cada apuntador nulo). De nuevo se usan rotaciones para mantener el balance después de una inserción o eliminación. Los árboles balanceados de Tarjan tienen la propiedad de que, tras una eliminación o inserción, puede restaurarse el balance aplicando a lo sumo una rotación doble y una simple, en contraste con las posibles O(log n) rotaciones tras la eliminación en un árbol de altura balanceada.

Los árboles balanceados también pueden usarse para una implantación eficiente de colas de prioridad. La inserción de un nuevo elemento requiere a lo sumo O(log n), pasos para encontrar la posición adecuada y O(1) pasos para accesar el elemento (siguiendo los apuntadores izquierdos hasta la hoja de la extrema izquierda) y O(log n) o O(1) pasos para eliminar esta hoja. Así al igual que una cola de prioridad implantada un heap, una cola de prioridad implantada usando un árbol balanceado puede ejecutar cualquier secuencia de n inserciones y eliminaciones mínimas en O(nlog n) pasos.

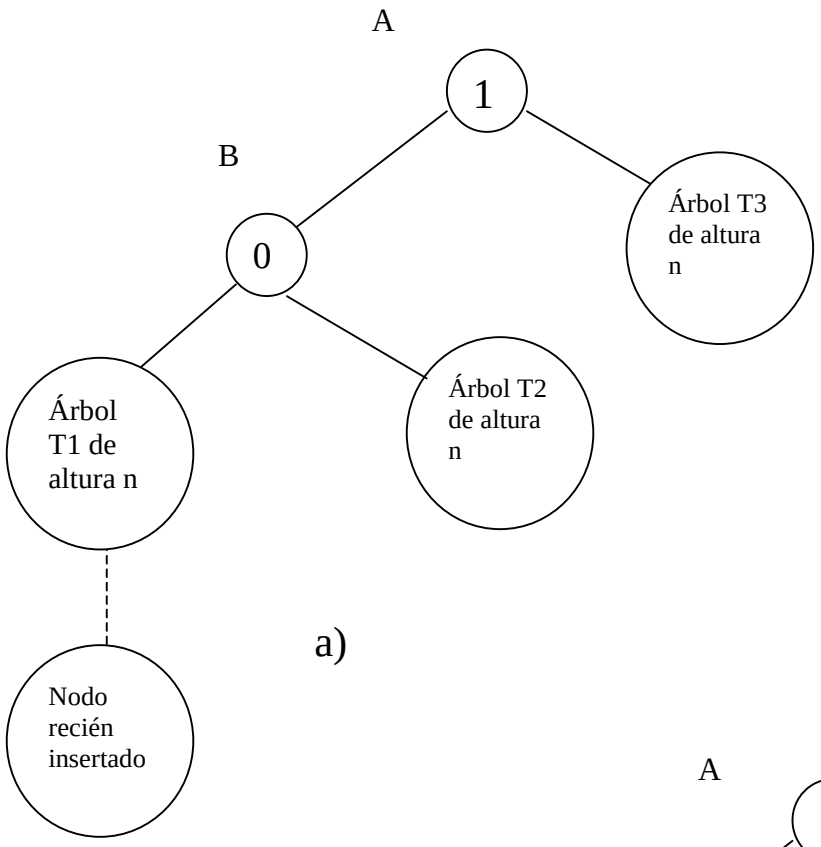
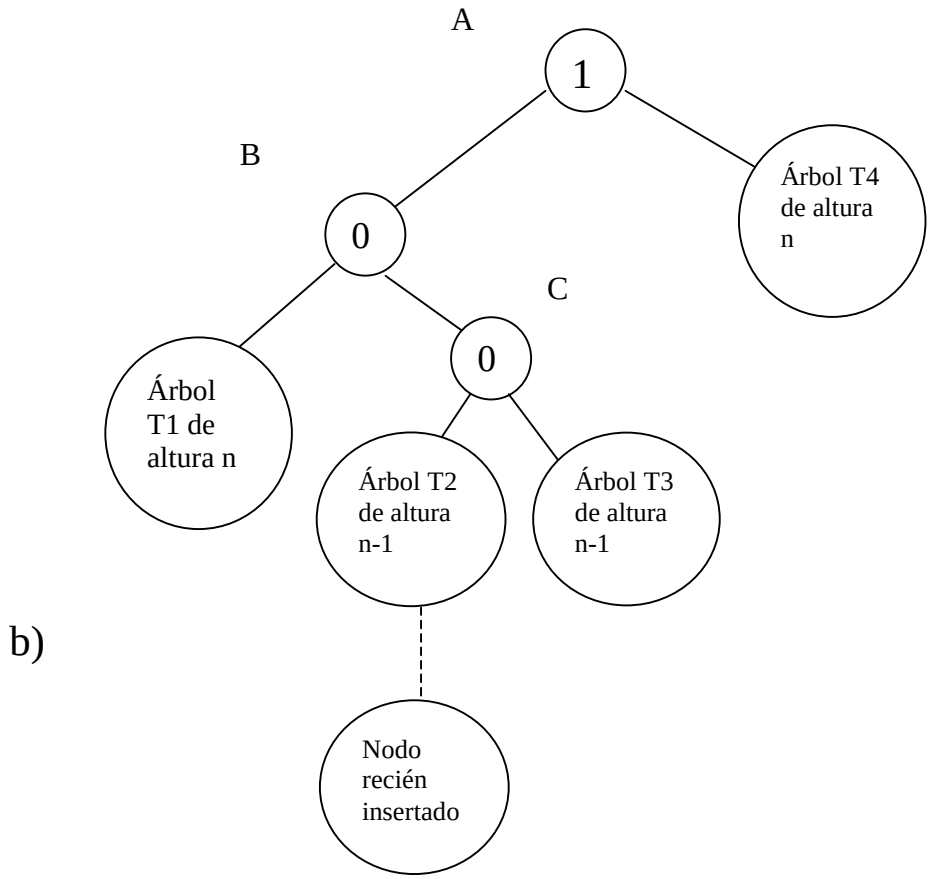

A
1
B
0
Árbol T3
de altura
n
Árbol T2
de altura
n
Árbol
T1 de
altura n
Nodo
recién
insertado

Figura 2

a)


A
1
B
0
Árbol T4
de altura
n
C
0
Árbol
T1 de
altura n
Árbol T2
de altura
n-1
Árbol T3
de altura
n-1
Nodo
recién
insertado

b)

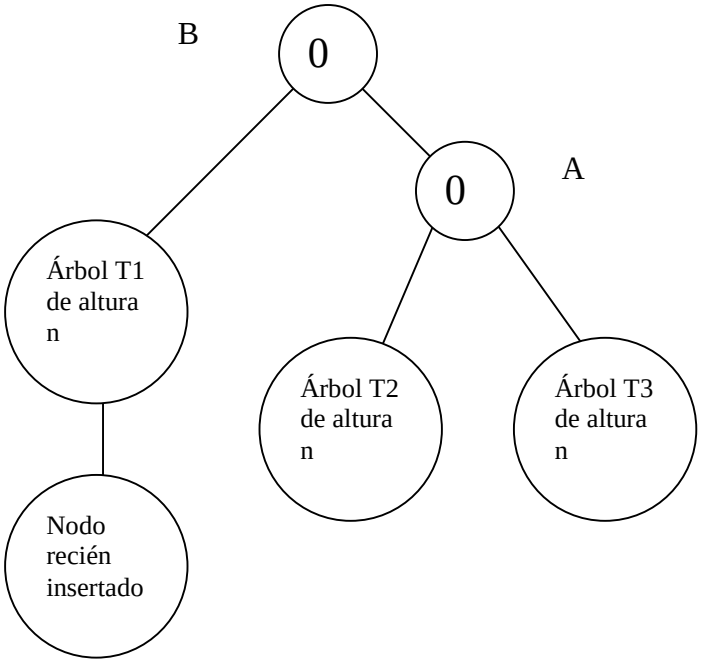
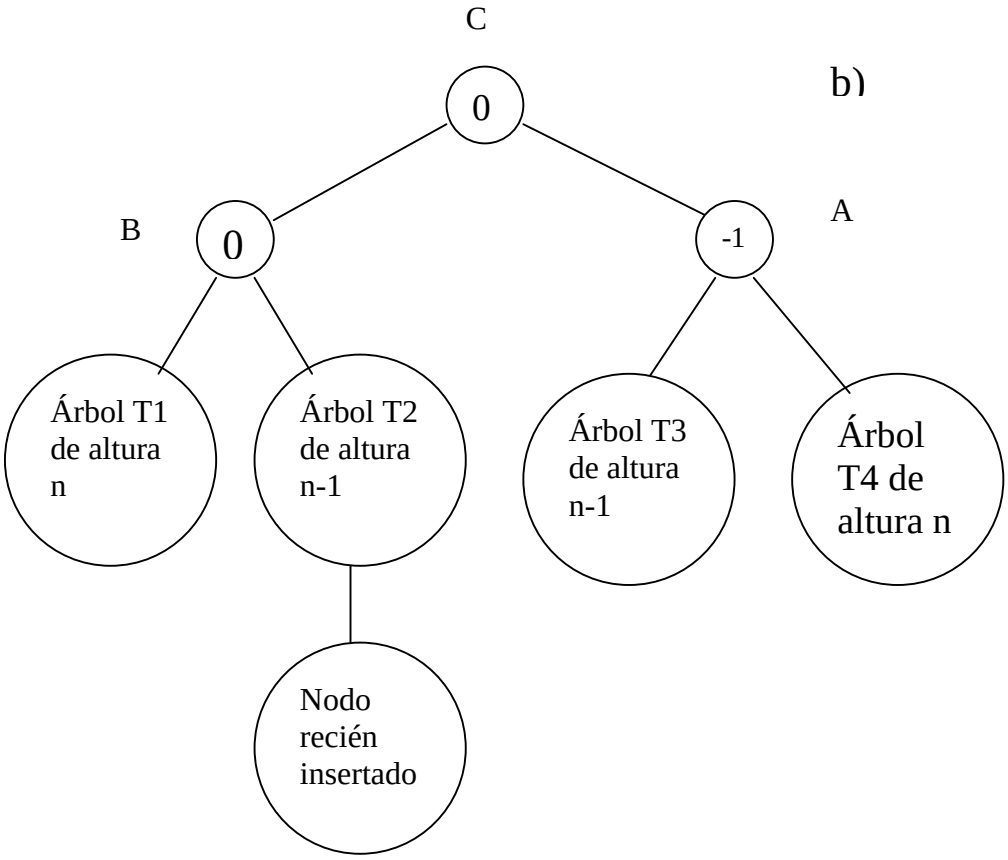

B
0
A
0
a)
Árbol T1
de altura
n
Árbol T2
de altura
n
Árbol T3
de altura
n
Nodo
recién
insertado

Figura 3


C
0
b)
B
0
A
-1
Árbol T1
de altura
n
Árbol T2
de altura
n-1
Árbol T3
de altura
n-1
Árbol
T4 de
altura n
Nodo
recién
insertado

CODIGO DE EJEMPLO

```
/*      busqueda e insercion en el arbol binario
                PARTE 1
*/
fp=NULL;
p=tree;
fya=NULL;
ya=p;
/*   ya apunta al ancestro mas joven que puede
     llegar a desbalancearse.fya señala al padre
     de ya, y fp al padre de p
*/
while(p!=NULL){
          if(key==k(p))
          return (p);
  q=(key<k(p) ? left(p):right(p);
  if(q!=NULL)
          if(bal(q)!=0){
          fya=p;
          ya=q;
    }     //      fin del if
  fp=p;
  p=q;
}         //              fin del while

// inserta nuevo registro
q=maketree(rec,key);
bal(q)=0;
(key<k(fp)) ?  left(fp)=q:right(fp)=q;

/*      el balance de todos los nodos entre node(ya) y node(q)
                debera alterarse,modificando su valor de 0
*/
p= (key < k(ya)) ?  left(ya) : right(ya);
s=p;
while(p!=q){
     if(key<k(p)){
          bal(p)= 1;
       p=left(p);
     }
     else{
          bal(p)=-1;
       p=right(p);
     }    //              fin del if
}         //              fin del while
```

```
/*
    PARTE 2

                determinar si el arbol se encuentra desbalanceado o no.
                si lo esta, q es el nodo recien insertado, ya es su ancestro
                desbalanceado mas joven, fya es el padre de ya y s es el hijo
                de ya en la dirección del desbalance.
*/
imbal=(key<k(ya)) ? 1:-1;
if(bal(ya)==0){
  /*    se le ha agregado otro nivel al arbol.
                El arbol permanece balanceado.
  */
        (bal(ya))=imbal;
        return(q);
}       //              fin del if
if(bal(ya)!=imbal){
/*   el nodo agregado se ha colocado en la direccion opuesta
     del desbalance. El arbol permanece balanceado.
*/
bal(ya)=0;
return (q);
}       //              fin del if

/*
   PARTE 3

        el nodo adicional a desbalanceado al arbol. restablecer
   el balance efectuando la rotacion requerida, ajustando
   despues los valores de balance de los nodos involucrados.
*/

if(bal(s)==imbal){
//      ya y s se han desbalanceado en la misma direccion.
        p=s;
        if(imbal==1)
        rightrotation(ya);
  else
        leftrotation(ya);
  bal(s)=0;
  bal(s)=0;
}
else{
        /*  ya y s se encuentran desbalanceados en direcciones
      opuestas.
  */
```

```
  if(imbal==1){
        p=right(s);
    leftrotation(s);
    left(ya)=p;
    rightrotation(ya);
  }
  else{
        p=left(s);
    right(ya)=p;
    rightrotation(s);
    leftrotation(ya);
  }     //              fin del if

  //    ajustar el campo bal para los nodos involucrados
  if(bal(p)==0){
  //    p fue un nodo insertado
        bal(ya)=0;
    bal(s)=0;
  }
  else
        if(bal(p)==imbal){
        bal(ya)=-imbal){
      bal(s)=0;
    }
    else
        bal(ya)=0;
      bal(s)=imbal;
    }   //              fin de if
  bal(p)=0;
}  //            fin de if
//      ajustar el epuntador del subarbol rotado
if(fya==NULL)
        tree=p;
else
        (ya==right(fya)) ? right(fya)=p: left(fya)=p;
  return (q);
}
```