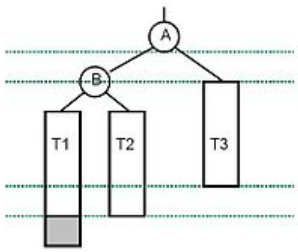
Factor de equilibrio

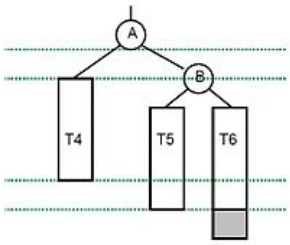
Rotación

```
T1, T2 and T3 are subtrees of the tree rooted with y (on the left side) or x (on the right
side)
     |                               |
     y                               x
    / \     Right Rotation          / \
   x   T3   - - - - - - - >        T1  y
  / \       < - - - - - - -           / \
 T1 T2      Left Rotation             T2 T3

Keys in both of the above trees follow the following order
     keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.
```

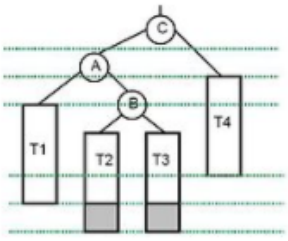
Ejercicio 1: Indica la secuencia de rotaciones para que el árbol se autobalancee



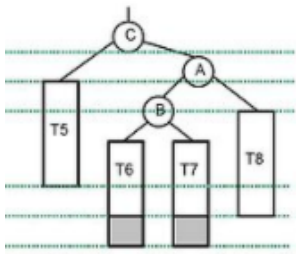
Ejercicio 2: Indica la secuencia de rotaciones para que el árbol se autobalancee



Ejercicio 3: Indica la secuencia de rotaciones para que el árbol se autobalancee



Ejercicio 4: Indica la secuencia de rotaciones para que el árbol se autobalancee



Factor de equilibrio:

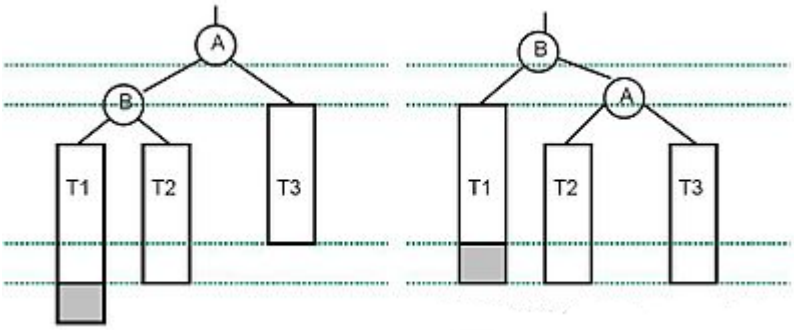
- Factor de Equilibrio (fe) = altura hijo derecho - altura hijo izquierdo
- altura del nodo vacío es -1
- altura de un único nodo es 0

Ejercicio 5: Determina la altura y el factor de equilibrio de cada nodo en los ejercicios 1, 2, 3, 4.

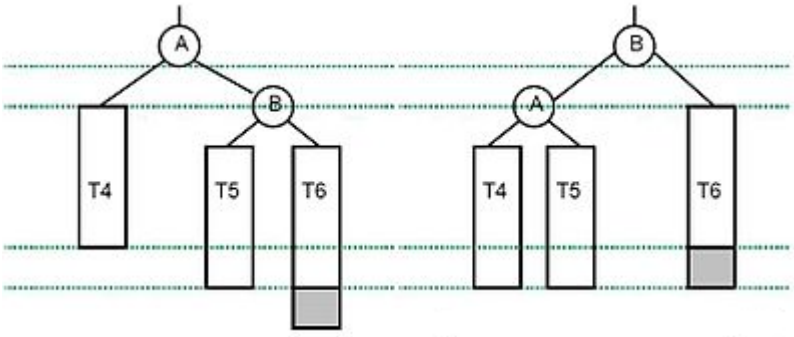
Ejercicio 6: Calcula los factores de equilibrio que determina las rotaciones en los ejercicios 1, 2, 3, 4.

Inserción

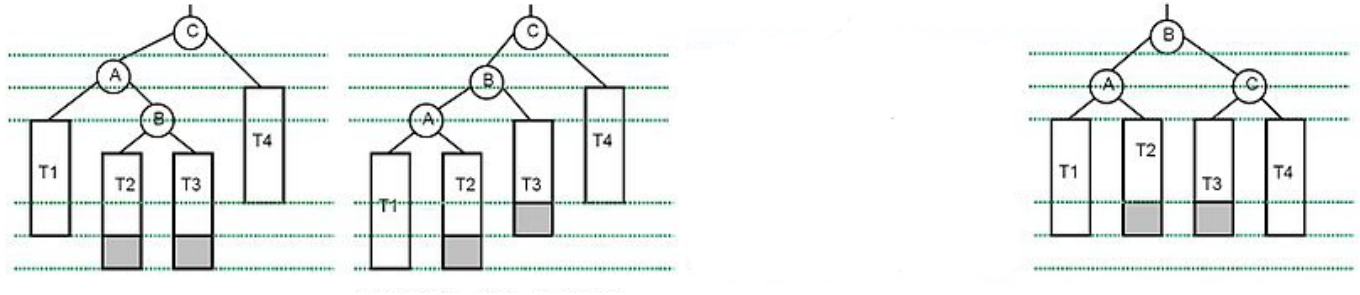
nodo->fe == -2 && nodo->hi->fe == -1 → Rotación simple a Derecha (o Derecha-Derecha)

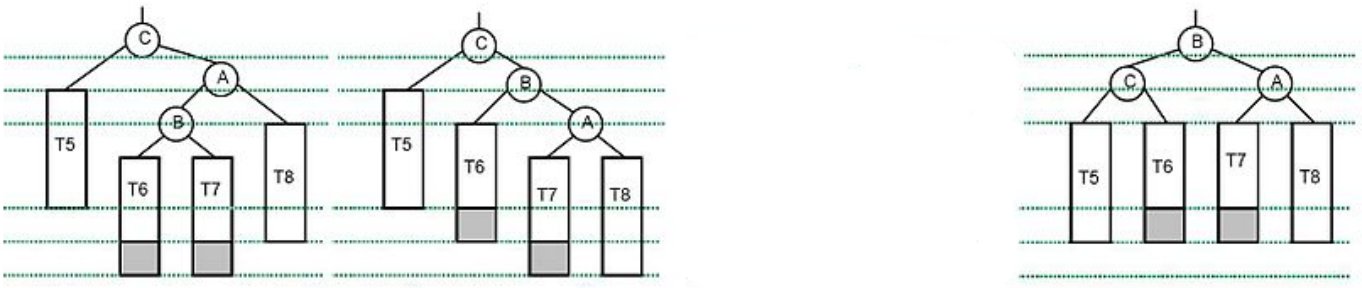


Solución 2: nodo->fe == 2 && nodo->hd->fe == 1 → Rotación simple a Izquierda (o Izquierda-Izquierda)



nodo->fe == -2 && nodo->hi->fe == 1 → Rotación Izquierda-Derecha



`nodo->fe == 2 && nodo->hd->fe == -1 → Rotación Derecha-Izquierda`

Borrado
`nodo->fe == 2 && nodo->hd->fe == 0 → Rotación Simple Izquierda (o Izquierda-Izquierda)`
`nodo->fe == -2 && nodo->hi->fe == 0 → Rotación Simple Derecha (o Derecha-Derecha)`

Implementación (El código puesto es incoherente con el gráfico)

1. A partir de la clase ABB cambiar el nombre a AVL.
2. Añadir la altura y el factor de equilibrio a la estructura Nodo.
3. Modificar la función de imprimir para que imprima la altura y el factor de equilibrio del nodo.
4. Añadir la función actualizar_altura_fe a la de insertar para que se actualice la altura y el factor de equilibrio del nodo cada vez que se inserte un nodo.
5. Añadir la función de balancear a la de insertar para balancear el nodo después de la inserción.

```
/******************************************************************************
Welcome to GDB Online.
GDB online is an online compiler and debugger tool for C, C++, Python, Java, PHP, Ruby, Perl,
C#, VB, Swift, Pascal, Fortran, Haskell, Objective-C, Assembly, HTML, CSS, JS, SQLite, Prolog.
Code, Compile, Run and Debug online from anywhere in world.
******************************************************************************/
#include <stdio.h>
#include <iostream>
using namespace std;

struct Nodo {
    int val;
    Nodo* hi;
    Nodo* hd;
};

class AVL {
    Nodo* raiz;
public:
    AVL() {
        raiz = NULL;
    }

    Nodo* insertarR(Nodo* nodo, int val) {
        if (!nodo) {
            Nodo* aux = new Nodo;
            aux->val = val;
            aux->hi = NULL;
            aux->hd = NULL;
            return aux;
        } else {
            if (val < nodo->val)
                nodo->hi = insertarR(nodo->hi, val);
            else if (val > nodo->val)
                nodo->hd = insertarR(nodo->hd, val);
            return equilibrar(nodo);
        }
    }

    void insertar(int val) {
        raiz = insertarR(raiz, val);
    }


/*
Minimo(ConstruirArbolBin(i, r, d)) =
    SI EsArbolBinVacio(i)
        r
    | Minimo(i)
*/

    int Minimo(Nodo* nodo) {
        if (!nodo->hi) // nodo->hi == NULL
            return nodo->val;
        else
            return Minimo(nodo->hi);
    }

/*
Eliminar(e, CrearArbolBinVacio) = CrearArbolBinVacio
Eliminar(e, ConstruirArbolBin(i, r, d)) =
    SI e=r
        SI EsArbolBinVacio(d)
```

```
            i
        | ConstruirArbolBin(i, Minimo(d), Eliminar(Minimo(d), d))
    | SI Mayor(e, r)
            ConstruirArbolBin(i, r, Eliminar(e, d))
        | ConstruirArbolBin(Eliminar(e, i), r, d)
*/

    Nodo* EliminarR(int e, Nodo* nodo) {
        if (!nodo) // raiz == NULL el arbol es EsArbolBinVacio
            return NULL;
        else {
            if (e == nodo->val) {
                if (nodo->hd == NULL) {
                    Nodo* aux = nodo->hi;
                    delete nodo;
                    return aux;
                } else {
                    int minimo = Minimo(nodo->hd);
                    nodo->val = minimo;
                    nodo->hd = EliminarR(minimo, nodo->hd);
                    nodo->hd = equilibrar(nodo->hd);
                    return nodo;
                }

            } else {
                if (e > nodo->val) {
                    nodo->hd = EliminarR(e, nodo->hd);
                    nodo->hd = equilibrar(nodo->hd);
                    return nodo;
                } else {
                    nodo->hi = EliminarR(e, nodo->hi);
                    nodo->hi = equilibrar(nodo->hi);
                    return nodo;
                }
            }
        }

        equilibrar(nodo);
    }

    void Eliminar(int e) {
        EliminarR(e, this->raiz);
    }


    void imprimirR(Nodo* nodo, int tab) {
        if (nodo) {
            for(int i = 0; i < tab; i++)
                cout << "\t";
            cout << nodo->val << "(" << FE(nodo)<< "," << altura(nodo) << ")" << endl;
            imprimirR(nodo->hi, tab + 1);
            imprimirR(nodo->hd, tab + 1);
        }
    }

    void imprimir() {
        if (this->raiz)
            imprimirR(raiz, 0);
        else
            cout << "El arbol está vacío" << endl;
    }

    void borrarR(Nodo* nodo) {
```

```
    if (nodo) {
        if (nodo->hi) borrarR(nodo->hi);
        if (nodo->hd) borrarR(nodo->hd);
        delete nodo;
    }
}

Nodo* rotacionDD(Nodo* nodo) {
    Nodo* y = nodo;
    Nodo* x = nodo->hi;
    //Nodo* T1 = nodo->hi->hi;
    Nodo* T2 = nodo->hi->hd;
    //Nodo* T3 = nodo->hd;

    //x->hi = T1;
    x->hd = y;
    y->hi = T2;
    //y->hd = T3;

    return x;
}

Nodo* rotacionII(Nodo* nodo) {
    Nodo* x = nodo;
    Nodo* y = nodo->hd;
    //Nodo* T1 = nodo->hi;
    Nodo* T2 = nodo->hd->hi;
    //Nodo* T3 = nodo->hd->hd;

    //y->hd = T3;
    y->hi = x;
    //x->hi = T1;
    x->hd = T2;

    return y;
}

Nodo* rotacionID(Nodo* nodo) {
    nodo->hi = rotacionII(nodo->hi);
    nodo = rotacionDD(nodo);

    return nodo;
}

Nodo* rotacionDI(Nodo* nodo) {
    //Nodo* C = nodo;
    //Nodo* A = nodo->hd;
    nodo->hd = rotacionDD(nodo->hd);
    nodo = rotacionII(nodo);

    return nodo;

}

int max(int a, int b) {
    return a > b ? a : b;
}

int altura(Nodo* nodo) {
    if (!nodo) // nodo == NULL
        return -1; // Por conveniencia para que cuando haya un único nodo, su altura sea 0.
    else {
        return max(altura(nodo->hd), altura(nodo->hi)) + 1;
```

```
        }
    }

    int FE(Nodo* nodo) {
        if (nodo) {
            return altura(nodo->hd) - altura(nodo->hi);
        } else {
            return 0;
        }
    }

    Nodo* equilibrar(Nodo* nodo) {
        if (nodo) {
            if (FE(nodo) == -2 && FE(nodo->hi) == -1) return rotacionDD(nodo);
            if (FE(nodo) == 2 && FE(nodo->hd) == 1) return rotacionII(nodo);
            if (FE(nodo) == -2 && FE(nodo->hi) == 1) return rotacionID(nodo);
            if (FE(nodo) == 2 && FE(nodo->hd) == -1) return rotacionDI(nodo);
            if (FE(nodo) == 2 && FE(nodo->hd) == 0) return rotacionII(nodo);
            if (FE(nodo) == -2 && FE(nodo->hi) == 0) return rotacionDD(nodo);
        }
        return nodo;
    }

    ~AVL() {
        cout << "~ABB" << endl;
        borrarR(raiz);
    }
};

int main()
{
    AVL avl;

    avl.imprimir();
    avl.insertar(5);
    avl.insertar(6);
    avl.insertar(4);
    avl.insertar(7);
    avl.insertar(8);
    avl.insertar(3);
    avl.insertar(2);
    avl.imprimir();

    return 0;
}
```