



Escuela Politécnica Superior

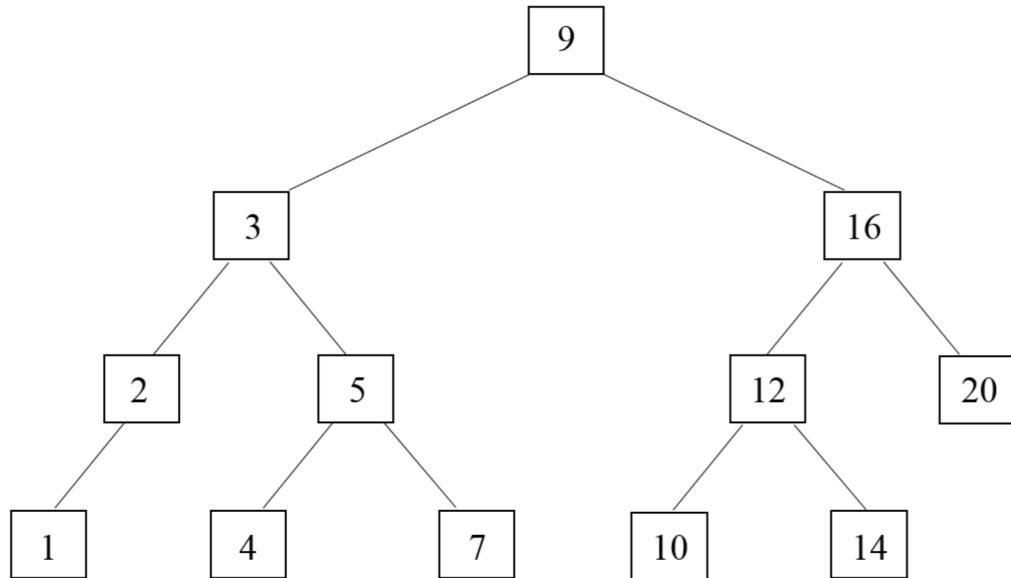
IIN151 – Estructuras de datos y algoritmos

Prácticas de laboratorio 4

Nombre:

Los árboles AVL son árboles binarios de búsqueda autobalanceados. Fueron propuestos por Adelson-Velskii y Landis en 1962.

Un árbol AVL es un árbol binario de búsqueda que cumple que la diferencia entre las alturas de los subárboles de cada uno de sus nodos es, como máximo 1. Los árboles AVL siempre están balanceados, por lo que la complejidad de la búsqueda en un árbol AVL es $O(\log n)$.



Prácticas de laboratorio 4

Un árbol AVL se define:

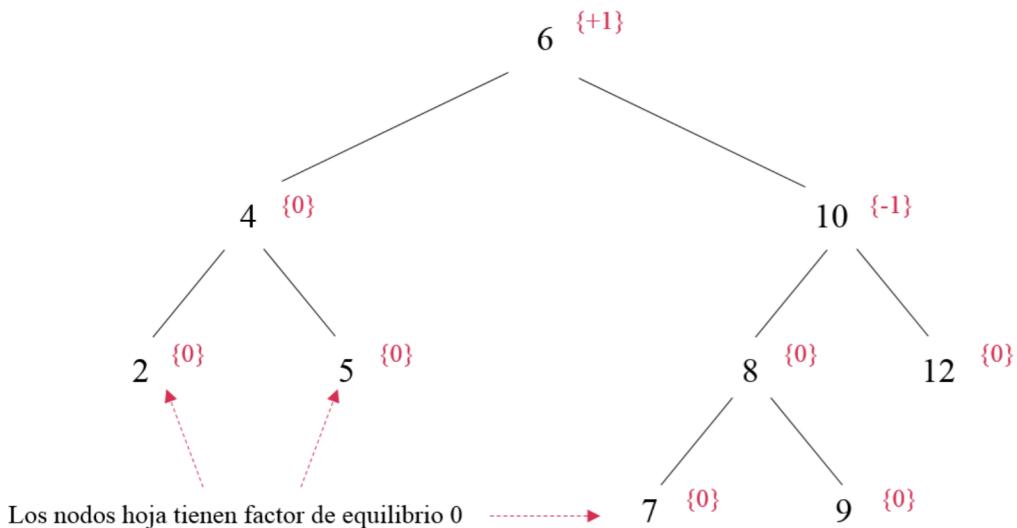
- Un árbol vacío es un árbol AVL
- Si T es un árbol binario no vacío en el que $T_{izquierdo}$ y $T_{derecho}$ son sus ramas izquierda y derecha, T es un árbol AVL si y solo si se cumple:

$T_{izquierdo}$ es un árbol AVL

$T_{derecho}$ es un árbol AVL

$$|\text{altura}(T_{derecho}) - \text{altura}(T_{izquierdo})| \leq 1$$

El factor de equilibrio de un nodo es la altura de su rama derecha menos la altura de su rama izquierda. El factor de equilibrio de los nodos de un árbol AVL balanceado toma los valores +1, 0, -1.

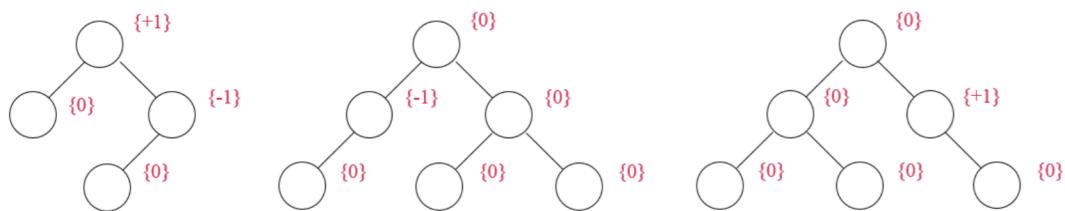
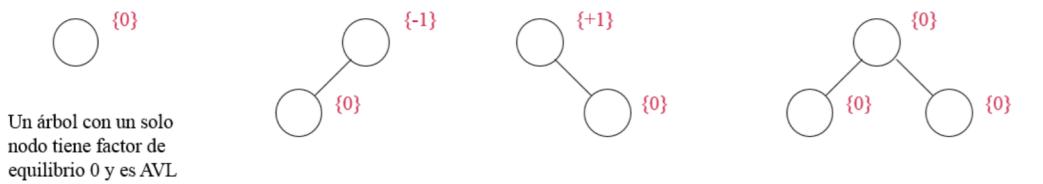


El factor de equilibrio del nodo raíz es +1, la rama derecha tiene altura 3 y la rama izquierda altura 2

Para todos los nodos de un árbol AVL balanceado se cumple:

- Si la altura de la rama derecha de un nodo es mayor que la izquierda, el factor de equilibrio es +1.
- Si el nodo es una hoja o sus ramas izquierda y derecha tienen la misma altura, el factor de equilibrio es 0.
- Si la altura de la rama izquierda de un nodo es menor que la derecha, el factor de equilibrio es -1.

Si el factor de equilibrio de un nodo es mayor que $|1|$ es necesario realizar rotaciones en los nodos para balancear el árbol.



Cuando se inserta o se elimina un nodo de un árbol AVL balanceado se puede perder la propiedad de equilibrio. Para mantener la propiedad de equilibrio de los nodos del árbol AVL se realizan rotaciones de nodos.

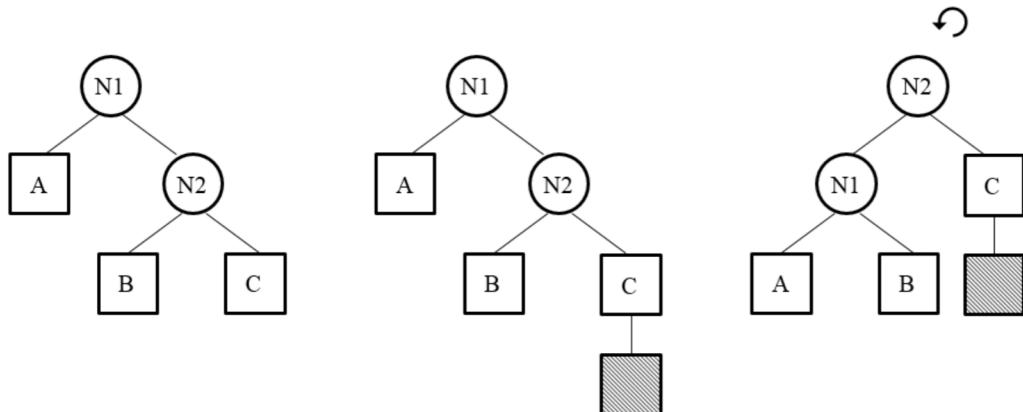
- Rotación simple a la izquierda
- Rotación simple a la derecha
- Rotación doble izquierda, derecha
- Rotación doble derecha, izquierda

Las operaciones de rotación de nodos mantienen el orden de los valores almacenados en los nodos del árbol. Después de realizar una rotación, el árbol resultante es un árbol binario de búsqueda.

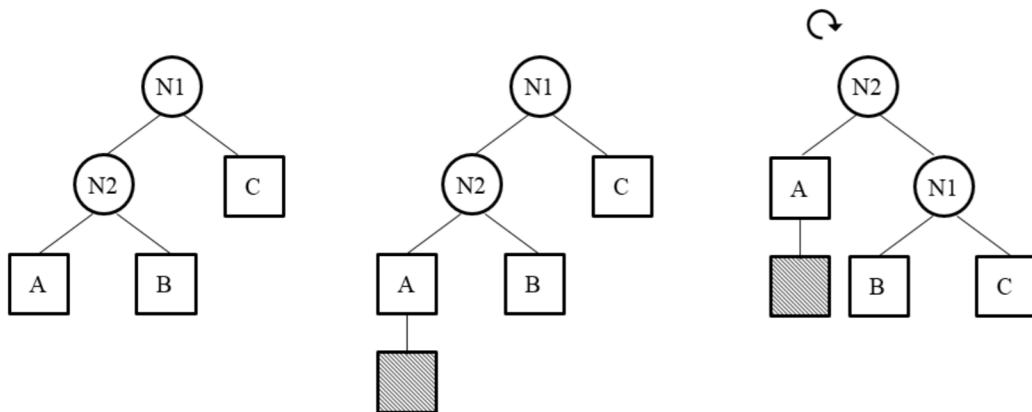
- a) El hijo izquierdo, si no es nulo, almacena un valor menor que el del nodo padre
- b) El hijo derecho, si no es nulo, almacena un valor mayor que el del nodo padre

Prácticas de laboratorio 4

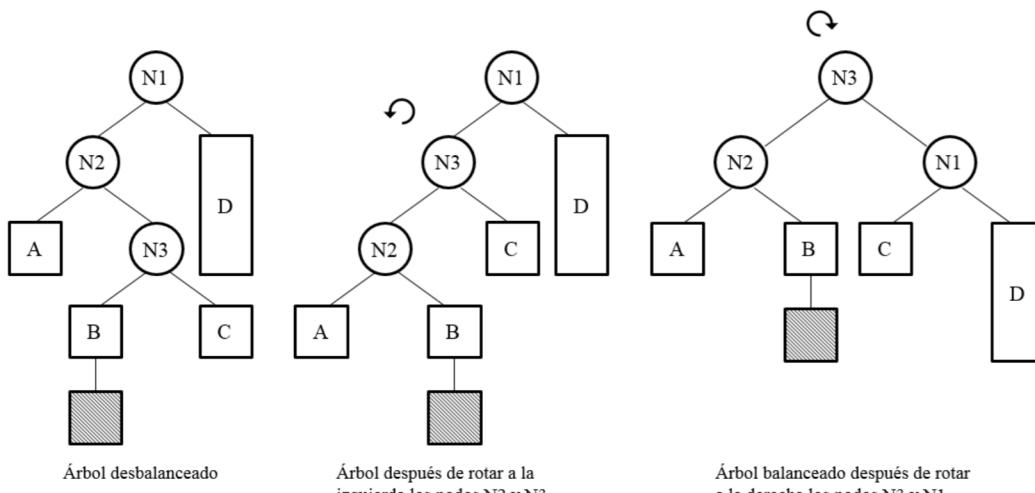
Rotación simple a la izquierda



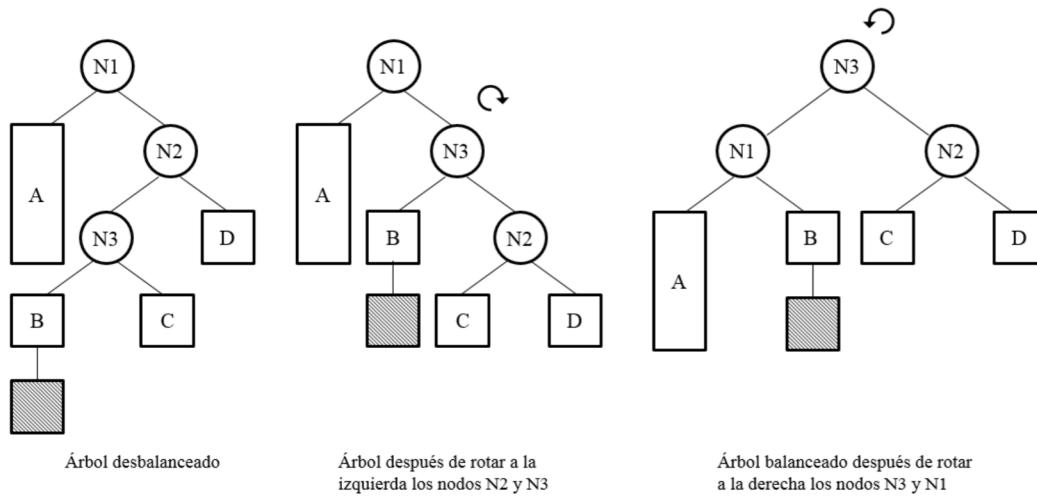
Rotación simple a la derecha



Rotación doble izquierda-derecha

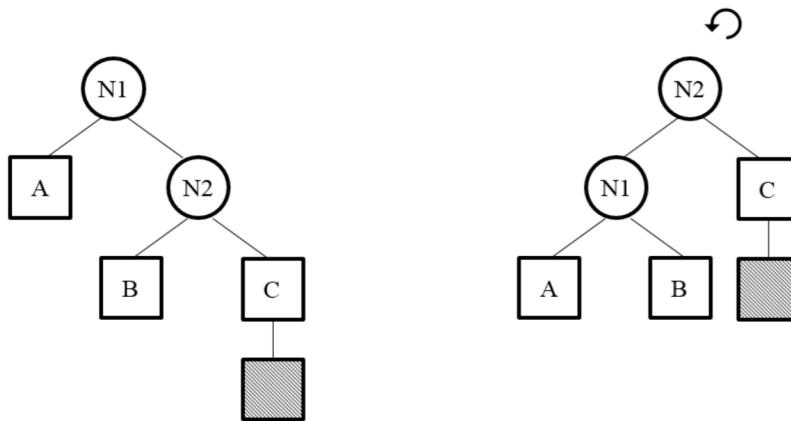


Rotación doble derecha-izquierda



Las operaciones de rotación de nodos mantienen el orden de los valores almacenados en los nodos del árbol. Después de realizar una rotación, el árbol resultante es un árbol binario de búsqueda.

Si el hijo izquierdo no es nulo, almacena un valor menor que el del nodo padre.



En el árbol desbalanceado después de insertar un nodo, se cumple:

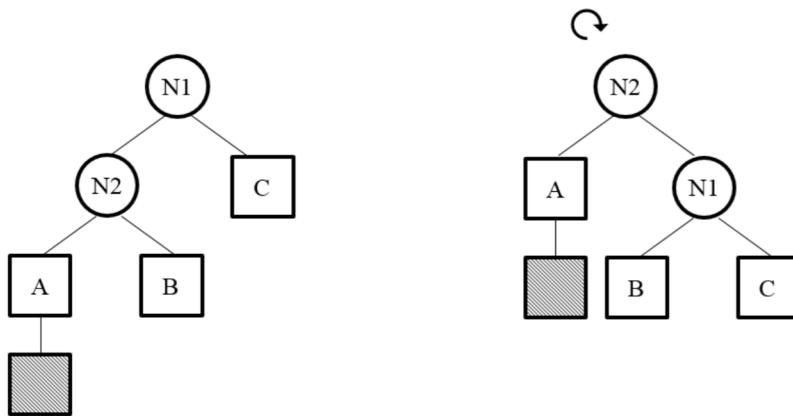
$$A < N1, N1 < N2, N1 < B, N2 < C$$

En el árbol balanceado tras rotar a la izquierda los nodos N1 y N2 se cumple:

$$A < N1, N1 < N2, N1 < B, N2 < C$$

Prácticas de laboratorio 4

Si el hijo derecho no es nulo, almacena un valor mayor que el del nodo padre.



En el árbol desbalanceado después de insertar un nodo, se cumple:

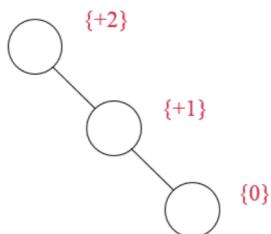
$$A < N2, N2 < N1, B < N1, N1 < C$$

En el árbol balanceado tras rotar a la derecha los nodos N1 y N2 se cumple:

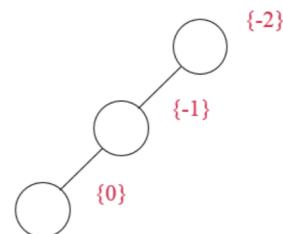
$$A < N2, N2 < N1, B < N1, N1 < C$$

Criterios de rotación de los nodos:

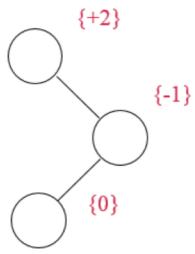
Rotación simple a la izquierda



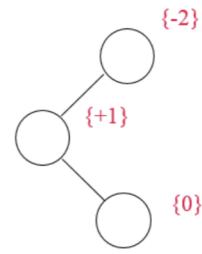
Rotación simple a la derecha



Rotación doble derecha, izquierda



Rotación doble izquierda, derecha



Si un nodo T tiene factor de equilibrio +2, la altura de la rama derecha $T_{derecho}$ es mayor que la altura de la rama izquierda. Para balancear el nodo T se realizan las rotaciones:

- Si $T_{derecho}$ tiene factor de equilibrio +1, se realiza una rotación simple a la izquierda de T .
- Si $T_{derecho}$ tiene factor de equilibrio -1, se realiza una rotación a la derecha de $T_{derecho}$ y una rotación a la izquierda de T .

Si un nodo T tiene factor de equilibrio -2, la altura de la rama izquierda $T_{izquierdo}$ es mayor que la altura de la rama derecha. Para balancear el nodo T se realizan las rotaciones:

- Si $T_{izquierdo}$ tiene factor de equilibrio -1, se realiza una rotación simple a la derecha de T .
- Si $T_{izquierdo}$ tiene factor de equilibrio +1, se realiza una rotación a la izquierda de $T_{izquierdo}$ y una rotación a la derecha de T .

La interfaz del TAD árbol balanceado AVL.

`inicializa(R)` Inicializa R a un árbol vacío

Precondición: Ninguna

Postcondición: Un árbol binario vacío

`vacio(R)` Devuelve verdadero si el árbol R está vacío y falso en cualquier otro caso

Precondición: Ninguna

Postcondición: `true` si el árbol R está vacío y `false` e.o.c.

`inserta(R, v)` Inserta un nodo con el valor v

Precondición: Ninguna

Postcondición: El nodo con el valor v ocupa la posición que le corresponde en el árbol R

Prácticas de laboratorio 4

busca(R, v) Busca el valor v en el árbol R

Precondición: Ninguna

Postcondición: Devuelve el primer nodo de R que almacena el valor v

elimina(R, v) Elimina el nodo de R que almacena el valor v

Precondición: Ninguna

Postcondición: El árbol R ya no tiene un nodo con el valor v

preorder(R) Recorre los nodos del árbol R en “preorder”

Precondición: Ninguna

Postcondición: Ninguna

inorder(R) Recorre los nodos del árbol R en “inorder”

Precondición: Ninguna

Postcondición: Ninguna

postorder(R) Recorre los nodos del árbol R en “postorder”

Precondición: Ninguna

Postcondición: Ninguna

Implemente el TAD Árbol AVL utilizando el siguiente código.

```
struct NodoArbol {
    int dato, altura, equilibrio;
    NodoArbol *izquierdo, *derecho;
};

struct TArbolBalanceadoAVL {
    NodoArbol *raiz;
};

void inicializa(struct TArbolBalanceadoAVL *R) {
    R->raiz = NULL;
}

bool vacio(struct TArbolBalanceadoAVL *R) {
    return (R->raiz == NULL);
}

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

```
int altura(struct NodoArbol *R) {
    int a;

    if (R == NULL)
        a = -1;
    else
        a = max(altura(R->izquierdo), altura(R->derecho)) + 1;

    return a;
}

int equilibrio(struct NodoArbol * R) {
    return (altura(R->derecho) - altura(R->izquierdo));
}

NodoArbol* insertaNodo(int v) {
    NodoArbol *nodo = new NodoArbol;
    nodo->dato = v;
    nodo->altura = 0;
    nodo->equilibrio = 0;
    nodo->izquierdo = NULL;
    nodo->derecho = NULL;

    return nodo;
}
```

Prácticas de laboratorio 4

```
void rotacionIzquierda(struct TArbolBalanceadoAVL *&T,
                        struct NodoArbol *&R) {

    // rotacion simple a la izquierda

    // si el nodo que se ha rotado era la raiz del arbol, se define
    // una nueva raiz

}

void rotacionDerecha(struct TArbolBalanceadoAVL *&T,
                      struct NodoArbol *&R) {

    // rotacion simple a la derecha

    // si el nodo que se ha rotado era la raiz del arbol, se define
    // una nueva raiz

}

void autibalnce(struct TArbolBalanceadoAVL *&T,
                struct NodoArbol *&R) {

    // si el factor de equilibrio es +2

    // la altura de la rama derecha Tderecha es mayor que la altura
    // de la rama izquierda
    //
    // si Tderecho tiene factor de equilibrio +1
    //     rotación a la izquierda de T
    //
    // si Tderecho tiene factor de equilibrio -1
    //     rotación a la derecha de Tderecho y rotación a la
    //     izquierda de T

    // si el factor de equilibrio es -2

    // la altura de la rama izquierda Tizquierda es mayor que la
    // altura de la rama derecha
    //
    // si Tizquierdo tiene factor de equilibrio -1
    //     rotación simple a la derecha de T
    //
    // si Tizquierdo tiene factor de equilibrio +1
    //     rotación a la izquierda de Tizquierdo y una rotación a
    //     la derecha de T

}
```

```
void inserta(struct TArbolBalanceadoAVL *&T,
            struct NodoArbol *&R, int v) {

    if (R == NULL)
        R = insertaNodo(v);
    else {
        if (v < R->dato)
            inserta(T, R->izquierdo, v);
        else
            if (v > R->dato)
                inserta(T, R->derecho, v);

        R->altura = altura(R);
        R->equilibrio = equilibrio(R);

        // si el valor absoluto del factor de equilibrio del nodo es
        // mayor de 1, es necesario balancear el arbol con rotaciones
        // de nodos

        if (abs(R->equilibrio) > 1)
            autobalance(T, R);
    }
}

NodoArbol* busca(struct NodoArbol *R, int v) {
    NodoArbol *nodo = NULL;

    if (R == NULL)
        nodo = R;
    else
        if (R->dato == v)
            nodo = R;
        else
            if (R->dato < v)
                nodo = busca(R->derecho, v);
            else
                nodo = busca(R->izquierdo, v);

    return nodo;
}
```

Prácticas de laboratorio 4

```
int eliminaMinimo(struct NodoArbol *&R) {
    int minimo;

    if (R->izquierdo == NULL) {
        minimo = R->dato;
        R = R->derecho;
    }
    else
        minimo = eliminaMinimo(R->izquierdo);

    return minimo;
}

void elimina(struct TArbolBalanceadoAVL *&T,
             struct NodoArbol *&R, int v) {
    if (R != NULL) {
        if (v < R->dato)
            elimina(T, R->izquierdo, v);
        else
            if (v > R->dato)
                elimina(T, R->derecho, v);
            else
                if (R->izquierdo == NULL && R->derecho == NULL) {
                    NodoArbol *p = R;
                    R = NULL;
                    delete(p);
                }
                else
                    if (R->izquierdo == NULL)
                        R = R->derecho;
                    else
                        if (R->derecho == NULL)
                            R = R->izquierdo;
                        else
                            R->dato = eliminaMinimo(R->derecho);

        if (R != NULL) {
            R->altura = altura(R);
            R->equilibrio = equilibrio(R);

            // si el valor absoluto del factor de equilibrio del nodo
            // es mayor de 1 es necesario balancear el arbol con
            // rotaciones de nodos

            if (abs(R->equilibrio) > 1)
                autobalance(T, R);
        }
    }
}
```

```
void preorder(NodoArbol *R) {
    if (R != NULL) {
        std::cout << R->dato << " {" << R->equilibrio << "}";
        preorder(R->izquierdo);
        preorder(R->derecho);
    }
}

void inorder(NodoArbol *R) {
    if (R != NULL) {
        inorder(R->izquierdo);
        std::cout << R->dato << " {" << R->equilibrio << "}";
        inorder(R->derecho);
    }
}

void postorder(NodoArbol *R) {
    if (R != NULL) {
        postorder(R->izquierdo);
        postorder(R->derecho);
        std::cout << R->dato << " {" << R->equilibrio << "}";
    }
}

void imprimeArbol(struct TArbolBinario *R) {
    std::cout << "\n";
    std::cout << "Preorder   ";
    preorder(R->raiz);
    std::cout << "\n";
    std::cout << "Inorder   ";
    inorder(R->raiz);
    std::cout << "\n";
    std::cout << "Postorder ";
    postorder(R->raiz);
    std::cout << "\n";
}
```

El programa de prueba del TAD Árbol AVL.

```
int main() {
    struct TArbolBalanceadoAVL *R = new TArbolBalanceadoAVL;

    inicializa(R);

    inserta(R, R->raiz, 8);
    inserta(R, R->raiz, 5);
    inserta(R, R->raiz, 4);

    imprimeArbol(R);

    inserta(R, R->raiz, 10);
    inserta(R, R->raiz, 12);

    imprimeArbol(R);

    std::cout << "\nelimina 4 \n";
    elimina(R, R->raiz, 4);

    imprimeArbol(R);

    inserta(R, R->raiz, 3);

    imprimeArbol(R);

    std::cout << "\nelimina 5 \n";
    elimina(R, R->raiz, 5);

    imprimeArbol(R);

    std::cout << "\nelimina 8 \n";
    elimina(R, R->raiz, 8);

    imprimeArbol(R);

    inserta(R, R->raiz, 11);
    inserta(R, R->raiz, 15);

    imprimeArbol(R);

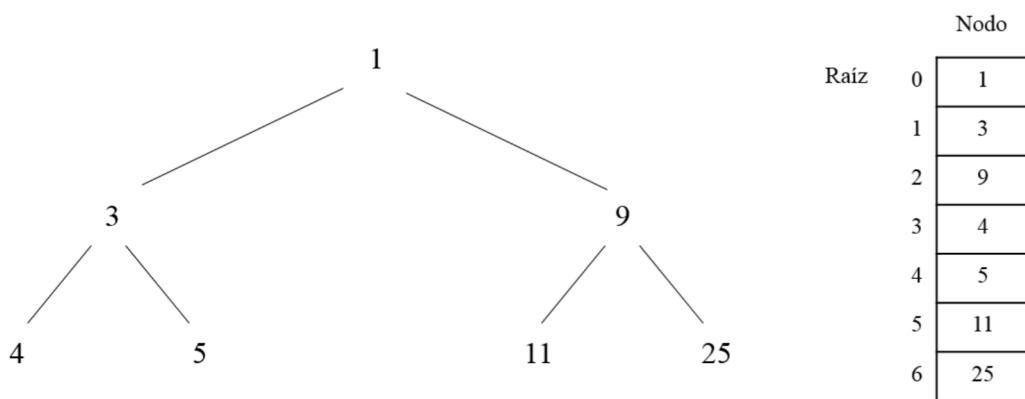
    std::cout << "\nelimina 3 \n";
    elimina(R, R->raiz, 3);

    imprimeArbol(R);
}
```

Un montículo (heap) es un árbol binario completo parcialmente ordenado que permite seleccionar entre un conjunto de valores, el máximo o el mínimo.

En un montículo de mínimos, los nodos padre almacenan un valor menor que el de sus hijos. En un montículo de máximos, los nodos padre almacenan un valor mayor que el de sus hijos.

Un montículo es un árbol binario completo. Dado que los nodos intermedios de un árbol binario completo tienen 2 hijos, un montículo se puede representar utilizando un array.



El nodo almacenado en la posición i de la tabla tiene su hijo izquierdo en la posición $2*i + 1$ y su hijo derecho en la posición $2*i + 2$

Para insertar un elemento en un montículo, el nuevo elemento se inserta en la siguiente posición disponible y se comprueba si se cumple el orden de un montículo de mínimos. Si no se cumple el orden del montículo, se intercambia el valor del nodo con el de su antecesor hasta llegar al nodo raíz.

Cuando se elimina un elemento, siempre se elimina el valor del nodo raíz, que almacena el valor mínimo o máximo del árbol, dependiendo del orden del montículo. Al eliminar un elemento, el nodo raíz toma el valor del último elemento del montículo. Si no se cumple el orden del montículo, se intercambia el valor del nodo raíz con sus descendientes de la rama que almacena el valor menor, hasta que el montículo queda ordenado.

Prácticas de laboratorio 4

La interfaz del TAD Montículo de mínimos.

```
inicializa(M) Inicializa el montículo M a vacío  
vacío(M) Devuelve verdadero si el montículo está vacía y falso en  
cualquier otro caso  
inserta(M, p) Inserta un elemento con prioridad p en el montículo  
eliminaMínimo (M, min) Elimina el elemento mínimo del montículo
```

Implemente el TAD Montículo de mínimos utilizando el siguiente código.

```
#define TAMAÑO_MONTICULO      50  
#define NO_ERROR              0  
#define ERROR_MONTICULO_VACIO 1  
#define ERROR_MONTICULO_LLENO 2  
  
struct TMonticuloMinimos {  
    int ultimo;  
    int prioridad[TAMAÑO_MONTICULO];  
};  
  
void inicializa(TMONTICULO *M) {  
    M->ultimo = -1;  
}  
  
bool vacia(TMONTICULO *M) {  
    return (M->ultimo < 0);  
}  
  
void intercambio(struct TMONTICULO *M, int i, int j) {  
    int v = M->prioridad[i];  
    M->prioridad[i] = M->prioridad[j];  
    M->prioridad[j] = v;  
}  
  
void intercambioAscendente(struct TMONTICULO *M, int hijo) {  
    int padre = (hijo - 1) / 2;  
  
    if (M->prioridad[hijo] < M->prioridad[padre]) {  
        intercambio(M, padre, hijo);  
  
        if (padre != 0)  
            intercambioAscendente(M, padre);  
    }  
}
```

```
void intercambioDescendente(struct TMonticuloMinimos *M,
                             int padre) {

    // intercambia los valores de los nodos, desde la raíz hasta
    // una hoja, para mantener el orden del montículo

}

int inserta(struct TMonticuloMinimos *M, int p) {
    int err = NO_ERROR;

    if (M->ultimo == TAMAÑO_MONTICULO - 1)
        err = ERROR_MONTICULO_LLENO;
    else {
        M->ultimo++;
        M->prioridad[M->ultimo] = p;
        intercambioAscendente(M, M->ultimo);
    }

    return err;
}

bool eliminaMinimo(struct TMonticuloMinimos *M, int &min) {
    int err = NO_ERROR;

    if (vacia(M))
        err = ERROR_MONTICULO_VACIO;
    else {
        min = M->prioridad[0];
        M->prioridad[0] = M->prioridad[M->ultimo--];
        intercambioDescendente(M, 0);
    }

    return err;
}

void imprime(struct TMonticuloMinimos *M) {
    std::cout << "{";

    for (int i = 0; i <= M->ultimo; i++)
        std::cout << M->prioridad[i] << ", ";

    std::cout << "}" \n";
}
```

El programa de prueba del TAD Montículo de mínimos.

```
int main() {
    struct TMonticuloMinimos *M = new TMonticuloMinimos;

    inicializa(M);

    inserta(M, 4);
    inserta(M, 5);
    inserta(M, 6);
    inserta(M, 3);
    inserta(M, 11);
    inserta(M, 7);
    inserta(M, 9);
    inserta(M, 14);
    inserta(M, 10);
    inserta(M, 8);

    imprime(M);

    std::cout << "inserta 2 \n";
    inserta(M, 2);

    imprime(M);

    int min;

    if (eliminaMinimo(M, min) == NO_ERROR)
        std::cout << "elimina minimo=" << min << "\n";
    else
        std::cout << "error elimina minimo\n";

    imprime(M);

    return 0;
}
```