

# La clase árbol binario

```
class nodoB {  
    public:  
        // Constructor:  
        nodoB(const int dat, nodoB *izq=NULL,  
               nodoB *der=NULL) :  
            dato(dat), izdo(izq), dcho(der) {}  
  
        // Miembros:  
        int dato;    // dato: Cualquier tipo de  
        nodoB *izdo; //          valor a almacenar  
        nodoB *dcho;  
};  
class ArbolB {  
    private:  
        nodoB *raiz;  
  
    public:  
        ...  
};
```

# Operaciones básicas

## Iniciales:

- **crea\_arbol.** Crear un árbol vacío o crear un árbol equilibrado.
- **es\_vacio.** Decir si el árbol es vacío.
- **imprime\_arbol.** Imprimir los nodos del árbol según su estructura.

## Recorridos:

- **pre\_orden.** Recorrer: Raiz, Subarbol izdo, Subarbol dcho.
- **post\_orden.** Recorrer: Subarbol izdo, Subarbol dcho, Raiz.
- **in\_orden.** Recorrer: Suba. izdo, Raiz, Suba. dcho. (*Ord simétrico*)

## Inserciones:

- **inserta\_hoja.**
- **inserta\_raíz.**
- **inserta\_equilibrado.**
- **inserta\_ordenado.**

## Eliminaciones:

- **busca\_datos.**
- **elimina** hoja o nodo interior.
- **elimina** el primer nodo o último en orden.

# Operaciones de Recorrido.

**Preorden**

```
void ArbolB::recorre( nodoB *nodo ) {  
    if (nodo == NULL) return;  
    procesa(nodo->dato);  
    recorre(nodo->izdo);  
    recorre(nodo->dcho);  
}
```

**Inorden**

```
void ArbolB::recorre( nodoB *nodo ) {  
    if (nodo == NULL) return;  
    recorre(nodo->izdo);  
    procesa(nodo->dato);  
    recorre(nodo->dcho);  
}
```

**Postorden**

```
void ArbolB::recorre( nodoB *nodo ) {  
    if (nodo == NULL) return;  
    recorre(nodo->izdo);  
    recorre(nodo->dcho);  
    procesa(nodo->dato);  
}
```

# Recorrido por Niveles.

```
void ArbolB::recorreN( nodoB *Raiz ) {  
    Cola Q;  
    NodoB *nodo;  
    int nivel, Nivel_actual = 0;  
    Q.insertar(Raiz, 0);  
  
    while (!Q.vacia()) {  
        Q.extraer(nodo, nivel);  
        if(nivel > Nivel_actual)  
            Nivel_actual = nivel;           //Incremento de nivel  
        if(nodo != NULL) {  
            Procesar(nodo);  
            Q.insertar(nodo->izdo, nivel+1);  
            Q.insertar(nodo->dcho, nivel+1);  
        }  
        else { //Subarbol vacío}  
    }  
}
```

# Operaciones Iniciales

```
public: // Constructor y destructor
ArbolB() : raiz(NULL) {}
~ArbolB() { Podar(raiz); }

void ArbolB::Podar(nodoB* &nodo) {
    if (nodo == NULL) return ;
    Podar(nodo->izdo); // Podar subarbol izquierdo
    Podar(nodo->dcho); // Podar subarbol derecho
    delete nodo;        // Eliminar nodo
    nodo = NULL;
}

bool EsVacio(nodoB *nodo) {
    return nodo == NULL;
}

bool EsHoja(nodoB *nodo) {
    return !nodo->dcho && !nodo->izdo;
}
```

# Tamaño y altura

```
const int ArbolB::Tam() { return TamRama(raiz); }  
const int ArbolB::TamRama(nodoB* nodo) {  
    if (nodo == NULL) return 0 ;  
    return (1 + TamRama(nodo->izdo) +  
           TamRama(nodo->dcho) );  
}
```

```
const int ArbolB::Alt() { return AltN(raiz); }  
const int ArbolB::AltN(nodoB* nodo) {  
    if (nodo == NULL)  
        return 0 ;  
    int alt_i = AltN(nodo->izdo);  
    int alt_d = AltN(nodo->dcho);  
    if (alt_d > alt_i)  
        return alt_d++;  
    else  
        return alt_i++;  
}
```

# Árbol binario equilibrado

```
const bool ArbolB::Equilibrado() {  
    return EquilibrioRama(raiz); }  
  
const bool ArbolB::EquilibrioRama(nodoB *nodo) {  
    if (nodo == NULL) return true ;  
    int eq = TamRama(nodo->izdo) -  
             TamRama(nodo->dcho);  
    switch (eq) {  
        case -1:  
        case 0:  
        case 1:  
            return EquilibrioRama(nodo->izdo) &&  
                   EquilibrioRama(nodo->dcho);  
        default: return false;  
    }  
}
```

# Inserción en equilibrio

```
void ArbolB::InsertaEquil(const int dato) {
    if (raiz == NULL)
        raiz = new nodoB(dato, NULL, NULL);
    else InsertaEqRama(dato, raiz);
}

void ArbolB::InsertaEquilRama(const int dato, nodoB* nodo) {
    if (Tam(nodo->izdo) < Tam(nodo->dcho) {
        if (nodo->izdo != NULL)
            InsertaEquilRama(dato, nodo->izdo);
        else
            nodo->izdo = new nodoB(dato, NULL, NULL);
    }
    else {
        if (nodo->dcho != NULL)
            InsertaEquilRama(dato, nodo->dcho);
        else
            nodo->dcho = new nodoB(dato, NULL, NULL);
    }
}
```



# La clase árbol de búsqueda binario

```
class nodoBB {
    public:
        // Constructor:
        nodoBB(int dat, int cl, nodoBB *iz=NULL,
               nodoBB *de=NULL) : dato(dat),
                                   clave(cl), izdo(iz), dcho(de) {}
        // Miembros:
        int dato;
        int clave;
        nodoBB *izdo;
        nodoBB *dcho;
};

class ArbolBB {
    private:
        nodoBB *raiz;

    public:
        ...
};
```

# El código de la búsqueda

```
nodoBB* ArbolBB::Buscar( int clave_dada )
{ return BuscarRama(raiz, clave_dada); }

nodoBB* ArbolBB::BuscarRama( nodoBB* nodo,
                             int clave_dada) {

    if (nodo == NULL)
        return NULL ;
    if (clave_dada == nodo->clave)
        return nodo ;
    if (clave_dada < nodo->clave )
        return BuscarRama(nodo->izdo, clave_dada);
    return BuscarRama(nodo->dcho, clave_dada);
}
```

# El código de la inserción

```
void ArbolBB::Insertar( int clave_dada) {  
    InsertarRama( raiz, clave_dada); }  
  
void ArbolBB::InsertarRama( nodoBB* &nodo,  
                           int clave_dada) {  
    if (nodo == NULL)  
        nodo = new nodoBB(clave_dada, clave_dada);  
    else if (clave_dada < nodo->clave)  
        InsertarRama(nodo->izdo, clave_dada);  
    else  
        InsertarRama(nodo->dcho, clave_dada);  
}
```

# El código de la eliminación (I)

```
void ArbolBB::Eliminar( int clave_dada){
    EliminarRama( raiz, clave_dada) ; }

void ArbolBB::EliminarRama( nodoBB* &nodo,
                             int clave_dada) {
    if (nodo == NULL) return NULL ;
    if (clave_dada < nodo->clave)
        EliminarRama(nodo->izdo, clave_dada);
    else if (clave_dada > nodo->clave)
        EliminarRama(nodo->dcho, clave_dada);
    else { //clave_dada == nodo_clave
        .../...
    }
}
```

# El código de la eliminación (II)

```
.../...  
else { //clave_dada == nodo->clave  
    nodoBB* Eliminado = nodo;  
    if (nodo->dcho == NULL)  
        nodo = nodo->izdo;  
    else if (nodo->izdo == NULL)  
        nodo = nodo->dcho;  
    else  
        sustituye(Eliminado, nodo->izdo);  
  
    delete (Eliminado);  
}  
}
```

# Buscando sustituto

```
void ArbolBB::sustituye(nodoBB* &eliminado,
                       nodoBB* &sust) {
    if (sust->dcho != NULL)
        sustituye(eliminado, sust->dcho);
    else {
        eliminado->Info = sust->Info ;
        eliminado->Clave = sust->Clave ;
        eliminado = sust ;
        sust = sust->izdo ;
    }
}
```

# Árbol binario balanceado

```
const bool ArbolB::Balanceado() {  
    return BalanceRama(raiz); }  
  
const bool ArbolB::BalanceRama(nodoB *nodo) {  
    if (nodo == NULL) return true;  
    int balance = Altura(nodo->izdo) -  
                  Altura(nodo->dcho);  
    switch (balance) {  
        case -1:  
        case 0:  
        case 1:  
            return BalanceRama(nodo->izdo) &&  
                   BalanceRama(nodo->dcho);  
        default: return false ;  
    }  
}
```

# La clase árbol AVL

```
class nodoAVL {  
    public:  
        // Constructor:  
        nodoAVL(int dat, int cl, nodoAVL *iz=NULL,  
                nodoAVL *de=NULL) : dato(dat),  
                clave(cl), bal(0), izdo(iz),  
                dcho(de) {}  
        // Miembros:  
        int dato;   int clave;   int bal;  
        nodoAVL *izdo; nodoAVL *dcho;  
};
```

```
class ArbolAVL {  
    private:  
        nodoAVL *raiz;  
  
    public:  
        ...  
};
```



# Rotación II

```
void ArbolAVL::rotacion_II (nodoAVL* &nodo) {  
    nodoAVL nodo1 = nodo->izdo;  
    nodo->izdo = nodo1->dcho;  
    nodo1->dcho = nodo;  
    if (nodo1->bal == 1) {  
        nodo->bal = 0;  
        nodo1->bal = 0;  
    }  
    else { // nodo1->bal == 0  
        nodo->bal = 1;  
        nodo1->bal = -1;  
    }  
    nodo = nodo1;  
}
```

# Rotación DD

```
void ArbolAVL::rotacion_DD (nodoAVL* &nodo) {  
    nodoAVL nodo1 = nodo->dcho;  
    nodo->dcho = nodo1->izdo;  
    nodo1->izdo = nodo ;  
    if (nodo1->bal == -1) {  
        nodo->bal = 0;  
        nodo1->bal = 0;  
    }  
    else { // nodo1->bal == 0  
        nodo->bal = -1;  
        nodo1->bal = 1;  
    }  
    nodo = nodo1;  
}
```

# Rotación ID

```
void ArbolAVL::rotacion_ID (nodoAVL* &nodo) {  
    nodoAVL* nodo1 = nodo->izdo;  
    nodoAVL* nodo2 = nodo1->dcho;  
    nodo->izdo = nodo2->dcho;  
    nodo2->dcho = nodo;  
    nodo1->dcho = nodo2->izdo;  
    nodo2->izdo = nodo1;  
    if (nodo2->bal == -1)  
        nodo1->bal = 1;  
    else nodo1->bal = 0;  
    if (nodo2->bal == 1)  
        nodo->bal = -1  
    else nodo->bal = 0;  
    nodo2->bal = 0;  
    nodo = nodo2;  
}
```

# Rotación DI

```
void ArbolAVL::rotacion_DI (nodoAVL* &nodo) {  
    nodoAVL* nodo1 = nodo->dcho;  
    nodoAVL* nodo2 = nodo1->izdo;  
    nodo->dcho = nodo2->izdo;  
    nodo2->izdo = nodo;  
    nodo1->izdo = nodo2->dcho;  
    nodo2->dcho = nodo1;  
    if (nodo2->bal == 1)  
        nodo1->bal = -1;  
    else nodo1->bal = 0;  
    if (nodo2->bal == -1)  
        nodo->bal = 1  
    else nodo->bal = 0;  
    nodo2->bal = 0;  
    nodo = nodo2;  
}
```

# Inserta y Balancea

```
void ArbolAVL::Insertar( int ClaveDada) {  
    NodoAVL nuevo = nodoAVL( ClaveDada, 0 );  
    bool crece = false;  
    inserta_bal( raiz, nuevo, crece);  
}
```

# Inserta y Balancea

```
void ArbolAVL::inserta_bal( nodoAVL* &nodo,
                           nodoAVL* nuevo, bool& crece) {
    if (nodo == NULL) {
        nodo = nuevo;
        crece = true;
    }
    else if (nuevo->clave < nodo->clave) {
        inserta_bal(nodo->izdo,nuevo,crece);
        if (crece) insert_re_balancea_izda(nodo);
    }
    else {
        inserta_bal(nodo->dcho,nuevo,crece);
        if (crece) insert_re_balancea_dcha(nodo);
    }
}
```

# Re-balancea a la izquierda

```

void ArbolAVL::insert_re_balancea_izda(
                                nodoAVL* &nodo) {

    switch (nodo->bal) {
        case -1: nodo->bal = 0;
                crece = false;
                break;

        case  0: nodo->bal = 1  ;
                break;

        case  1: nodoAVL* nodo1 = nodo->izdo;
                if (nodo1->bal == 1)
                    rotacion_II(nodo);
                else rotacion_ID(nodo);
                crece = false;
    }
}

```

# Re-balancea a la derecha

```
void ArbolAVL::insert_re_balancea_dcha (
                                nodoAVL* &nodo) {

    switch (nodo->bal) {
        case 1: nodo->bal = 0;
                crece = false;
                break;

        case 0: nodo->bal = -1;
                break;

        case -1: nodoAVL* nodo1 = nodo->dcho;
                if (nodo1->bal == -1)
                    rotacion_DD(nodo);
                else rotacion_DI(nodo);
                crece = false;
    }
}
```



# Eliminación con Rebalanceo

```
void ArbolAVL::eliminar( int clave_dada) {  
    bool decrece = false;  
    elimina_rama( raiz, clave_dada, decrece);  
}
```

# Eliminación con Rebalanceo

```

void ArbolAVL::elimina_rama( nodoAVL* &nodo,
                             int ClaveDada, bool& decrece) {
    if (nodo == NULL) return;
    if (clave_dada < nodo->clave) {
        elimina_rama(nodo->izdo, clave_dada, decrece);
        if (decrece)
            eliminar_re_balancea_izda(nodo, decrece);
    }
    else if (clave_dada > nodo->clave) {
        elimina_rama(nodo->dcho, clave_dada, decrece);
        if (decrece)
            eliminar_re_balancea_dcha(nodo, decrece);
    }
    else { // clave_dada == nodo->clave
        ... /...
    }
}

```

# Encontrado a eliminar

```
else { // nodo->clave == clave_dada(encontrado)
    NodoAVL* Eliminado = nodo;
    if (nodo->izdo == NULL) {
        nodo = nodo->dcho;
        decrece = true;
    }
    else if (nodo->dcho == NULL) {
        nodo = nodo->izdo;
        decrece = true;
    }
    else {
        sustituye(Eliminado, nodo->izdo, decrece);
        if (decrece)
            eliminar_re_balancea_izda(nodo, decrece);
    }
    delete Eliminado;
}
```

# Buscando sustituto

```
void ArbolAVL::sustituye(nodoAVL* &eliminado,
                        nodoAVL* &sust, bool &decrece) {
    if (sust->dcho != NULL) {
        sustituye(eliminado, sust->dcho, decrece);
        if (decrece)
            eliminar_re_balancea_dcha(sust, decrece);
    }
    else {
        eliminado->Info = sust->Info;
        eliminado->Clave = sust->Clave;
        eliminado = sust;
        sust = sust->izdo;
        decrece = true;
    }
}
```

# Re-balancea a la izquierda

```

void ArbolAVL::eliminar_re_balancea_izda (
    nodoAVL* &nodo, bool& decrece) {

    switch (nodo->bal) {
        case -1: nodoAVL* nodo1 = nodo->dcho;
            if (nodo1->bal > 0)
                rotacion_DI(nodo);
            else {
                if (nodo1->bal == 0)
                    decrece = false;
                rotacion_DD(nodo);
            }
            break;
        case 0: nodo->bal = -1;
            decrece = false;
            break;
        case 1: nodo->bal = 0;
    }
}

```

# Re-balancea a la derecha

```
void ArbolAVL::eliminar_re_balancea_dcha (
    nodoAVL* &nodo, bool& decrece) {

    switch nodo->bal {
        case 1: nodoAVL* nodo1 = nodo->izdo;
            if (nodo1->bal < 0)
                rotacion_ID(nodo);
            else {
                if (nodo1->bal == 0)
                    decrece = false;
                rotacion_II(nodo);
            }
            break ;
        case 0: nodo->bal = 1;
            decrece = false;
            break;
        case -1: nodo->bal = 0;
    }
}
```