Árbol binario de búsqueda .

Estructura de datos eficiente que permite buscar, insertar y borrar cualquier elemento o cualquier rango de elementos.

Un árbol binario de búsqueda es un árbol binario, que puede ser vacío y si no es vacío cumple las siguientes propiedades:

- Todos los elementos tienen una clave y no existen dos elementos con igual clave.
- Las claves (si existen) del subárbol izquierdo son menores que la clave en la raíz.
- Las claves (si existen) del subárbol derecho son menores que la clave en la raíz.
- Los subárboles izquierdo y derecho son también árboles binarios de búsqueda.

Árbol de binario búsqueda. Especificación

especificación ARBOL-BUSCA[A es ELEM-ORD] **es**

usa BOOL

instancia privada ARBOL-BIN[B es ELEM] donde B.elem es A.elem

tipos arbol-busca

operaciones

insertar	:	elem arbol-busca	→	arbol-busca
buscar	:	elem arbol-busca	→	arbol-busca
borrar	:	elem arbol-busca	→	arbol-busca
esta	:	elem arbol-busca	→	bool
privada min	:	arbol-busca	→	elem

variables

iz,dr,a	:	arbol-busca
x,y	:	elem

ecuaciones

min(arbol-vacio)	=	$\text{error}_{\text{arbol-busca}}$
min(plantar(iz,x,dr))	=	x ⇐ vacio?(iz)
min(plantar(iz,x,dr))	=	min(iz) ← ¬ vacio?(iz)
insertar(y,arbol-vacio)	=	plantar(arbol-vacio,y,arbol-vacio)
insertar(x,plantar(iz,x,dr))	=	plantar(iz,x,dr)
insertar(y,plantar(iz,x,dr))	=	plantar(insertar(y,iz),x,dr) ⇐ $y < x$
insertar(y, plantar(iz,x,dr))	=	plantar(iz,x,insertar(y,dr)) ⇐ $y > x$
buscar(x,arbol-vacio)	=	arbol-vacio
buscar(x,plantar(iz,x,dr))	=	plantar(iz,x,dr)
buscar(y,plantar(iz,x,dr))	=	buscar(y,iz) ⇐ $y < x$
buscar(y, plantar(iz,x,dr))	=	buscar(y,dr) ⇐ $y > x$

borrar(y,arbol-vacio)	=	arbol-vacio
borrar(x,plantar(iz,x,dr))	=	dr ⇐ vacio?(iz)
borrar(x,plantar(iz,x,dr))	=	iz ⇐ vacio?(dr)
borrar(x,plantar(iz,x,dr))	=	plantar(iz,min(dr),borrar(min(dr),dr)) ⇐ ¬ vacio?(iz) ∧¬ vacio?(dr)
borrar(y,plantar(iz,x,dr))	=	plantar(borrar(y,iz),x,dr) ⇐ $y < x$
borrar(y,plantar(iz,x,dr))	=	plantar(iz,x,borrar(y,dr)) ⇐ $y > x$
esta(x,a)	=	¬ vacio?(buscar(x,a))

fespecificación

Árbol de búsqueda binario. Implementación

```
template <class TipoClave>
class ArbolBinarioBusqueda
{
  public:
    // métodos de árbol binario
    nodo_arbol<TipoClave> * Busqueda(const TipoClave &x);
    nodo_arbol<TipoClave> * Busqueda(nodo_arbol<TipoClave> * b, const TipoClave &x);
    bool insertar( const TipoClave & x );
    bool borrar( TipoClave & x );

  private:
    nodo_arbol<TipoClave> *raiz;

    TipoClave Buscar_Min(nodo_arbol<TipoClave> * );
};
```


Búsqueda de un elemento

```
template <class TipoClave>
    nodo_arbol<TipoClave> * ArbolBinarioBusqueda<TipoClave>
::Busqueda(const TipoClave &x)
{        return Busqueda(raiz,x);   }

template <class TipoClave>
    nodo_arbol<TipoClave> * ArbolBinarioBusqueda<TipoClave>::
       Busqueda(nodo_arbol<TipoClave> * b, const TipoClave &x)
{      if ( b == 0 )
                return 0;
         if ( x == b->elemento )
                return b;
         if ( x < b->elemento )
                return Busqueda(b->izq,x);
         return Busqueda(b->der,x);
}
```

Complejidad: $\mathcal{O}(h)$ donde h es la profundidad del árbol

Búsqueda del elemento menor

```
// Esta función es llamada con al menos un elemento en su hijo derecho
template <class TipoClave>
     TipoClave ArbolBinarioBusqueda<TipoClave>::
       Buscar_Min(nodo_arbol<TipoClave>* aux )
     {
           while (aux -> izq != NULL)
                aux = aux -> izq;
           return aux->elemento;
     }
```

Inserción de un elemento

Complejidad: $\mathcal{O}(h)$ donde h es la profundidad del árbol


```
template <class TipoClave>
 bool ArbolBinarioBusqueda<TipoClave>::insertar( const TipoClave & x )
 {     // busqueda del lugar a insertar x, q es el padre de p
    nodo_arbol<TipoClave> *p = raiz, *q=0 ;
    while( p )
    {     q = p;
          if( x == p-> elemento ) return false;
          if( x < p->elemento ) p = p->izq;
         else p = p->der;
    }

    p = new nodo_arbol<TipoClave>;
    p->izq = p->der = 0;
    p-> elemento = x;
    if (!raiz) raiz = p;
    else if ( x < q->elemento ) q->izq = p;
          else q->der = p;
    return true;
}
```

Borrado de un elemento

- Si el elemento a borrar es una hoja del árbol el campo correspondiente del padre se pone a `null` y se libera la memoria.
- Si el elemento a borrar tiene un único hijo, el hijo toma el lugar del nodo que se elimina y se libera la memoria del nodo que se elimina.
- Si el elemento a borrar tiene dos hijos, el elemento se sustituye por el mayor de los elementos del subárbol izquierdo o por el menor de los elementos del subárbol derecho. A continuación se procede a eliminar este elemento y se libera la memoria.

Complejidad: $\mathcal{O}(h)$ donde h es la profundidad del árbol


```
template <class TipoClave>
bool ArbolBinarioBusqueda<TipoClave>::borrar( TipoClave & x )
{  // busqueda del elemento x, q es el padre de p
    nodo_arbol<TipoClave> *p = raiz, *q=0 ;
    bool encontrado = false;
    while( p && encontrado == false )
    {   if ( x == p->elemento )
            encontrado = true;
        else
        {  q = p;
            if( x < p->elemento )
                p = p->izq;
            else
               p = p->der;
        }
    }
    // p apunta al elemento a borrar
    // q apunta al padre del elemento a borrar
```



```
if( !p )
    return false;         // Si no se ha encontrado acabo
if( p->izq == NULL && p->der == NULL ) // si es una hoja
{      if ( !q && p == raiz)    // si es la raiz
       {
           raiz = NULL;
           delete p;
           return true;
       }

       if( q->izq == p )
           q->izq = NULL;
       else if( q->der == p )
                q->der = NULL;
       delete p;
       return true;         // se elimina y finaliza
}
```



```
if( p->izq != NULL && p->der == NULL ) // si tiene un hijo
{      if ( !q && p == raiz)    // si es la raiz
       {  raiz = p->izq;
           delete p;
           return true;
       }
       if( q->izq == p )
           q->izq = p->izq;
       else if( q->der == p )
                q->der = p->izq;
       delete p;
       return true;         // se elimina y finaliza
}
if( p->izq == NULL && p->der != NULL ) // si tiene un hijo
{      if ( !q && p == raiz)    // si es la raiz
       {  raiz = p->der;
           delete p;
           return true;
       }
```

```
            if( q->izq == p )
                q->izq = p->der;
            else if( q->der == p )
                     q->der = p->der;
            delete p;
            return true;         // se elimina y finaliza
    }
    if( p->izq != NULL && p->der != NULL ) // si tiene dos hijos
    {          TipoClave x = Buscar_Min(p->der);
               borrar(x);
               p->elemento = x;
    }
}
```


Árbol de búsqueda binario balanceado.

La profundidad de un árbol de búsqueda binario de n elementos puede llegar a ser n.

Los Árboles de búsqueda balanceados permiten realizar las operaciones de búsqueda, insercción y borrado con complejidad $\mathcal{O}(log(n))$.

Los tipos más conocidos de árboles de búsqueda balanceados son:

- AVL
- 2-3
- 2-3-4
- rojo/negro
- Árboles B