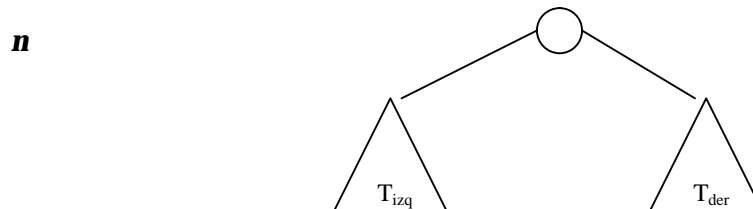


ÁRBOL BINARIO

- Un árbol binario puede definirse como un árbol que en cada nodo puede tener como mucho grado 2, es decir, a lo más 2 hijos. Los hijos suelen denominarse hijo a la izquierda e hijo a la derecha, estableciéndose de esta forma un orden en el posicionamiento de los mismos.

- Si n es un nodo y T_{izq} , T_{der} son árboles binarios, entonces podemos construir un nuevo árbol binario que tenga como raíz el nodo n y como subárboles T_{izq} y T_{der} (subárbol izquierdo y subárbol derecho de n , respectivamente). Un árbol vacío es un árbol binario.



- Árbol binario homogéneo es aquel cuyos nodos tienen grado 0 ó 2 (no hay ninguno de grado 1).
- Árbol binario completo es aquel que tiene todos los niveles llenos excepto, quizás, el último en cuyo caso los huecos deben quedar a la derecha.
- Número máximo de nodos por nivel en un árbol binario en un nivel i es 2^{i-1} .

TDA ÁRBOL BINARIO

- Una instancia del tipo de dato abstracto Árbol Binario sobre un dominio Tbase se puede construir como:
 - Un objeto vacío (árbol vacío) si no contiene ningún elemento. Lo denotamos {}.
 - Un árbol que contiene un elemento destacado, el nodo raíz, con un valor e en el dominio Tbase (denominado etiqueta), y dos subárboles (T_i izquierdo y T_d derecho) del TDA Árbol Binario sobre Tbase. Se establece una relación padre-hijo entre cada nodo y los nodos raíz de los subárboles (si los hubiera) que cuelgan de él. Lo denotamos {e, $\{T_i\}$, $\{T_d\}$ }
- El espacio requerido para el almacenamiento es $O(n)$. Donde n es el número de nodos del árbol.

IMPLEMENTACIÓN DEL ÁRBOL BINARIO

- Un árbol binario se compone de un nodo **raíz** que es un puntero al primer nodo del árbol, vale 0 si el árbol es vacío, nodo es una estructura donde almacenamos una etiqueta del árbol que se implementa como un conjunto de nodos enlazados según la relación padre-hijo, los componentes de la estructura nodo son los siguientes:

- ☞ **Etiqueta** -> es el elemento almacenado en un nodo del árbol, es de tipo Tbase, int, char, etc.
- ☞ **Izqda** -> en este campo de almacena un puntero al nodo raíz del subárbol izquierdo o el valor 0 si no tiene.
- ☞ **Drcha** -> en este campo de almacena un puntero al nodo raíz del subárbol derecho o el valor 0 si no tiene.

☞ **Padre** -> en este campo se almacena un puntero al nodo padre o el valor 0 si es la raíz del árbol.

Un nodo nulo es un nodo que apunta a null, no tiene nada y se denota en esta implementación con el número cero.

CLASE ABSTRACTA ÁRBOL BINARIO

```
Class ArbolBinario
{
    Public:
        typedef struct nodo *Nodo;
        static const Nodo nodo_nulo = 0;

        ArbolBinario();
        ArbolBinario(const Tbase& e);
        ArbolBinario (const ArbolBinario<Tbase>& v);
        ArbolBinario<Tbase>& operator=(const ArbolBinario<Tbase> &v);
        void AsignaRaiz(const Tbase& e);
        Nodo raiz() const;
        Nodo izquierda(const Nodo n) const;
        Nodo derecha(const Nodo n) const;
        Nodo padre(const Nodo n) const;
        Tbase& etiqueta(const Nodo n);
        const Tbase& etiqueta(const Nodo n) const;
        void asignar_subarbol(const ArbolBinario<Tbase>& orig, const Nodo nod);
        void podar_izquierda(Nodo n, ArbolBinario<Tbase>& dest);
        void podar_derecha(Nodo n, ArbolBinario<Tbase>& dest);
        void insertar_izquierda(Nodo n, ArbolBinario<Tbase>& rama);
```

```

void insertar_derecha(Nodo n, ArbolBinario<Tbase>& rama);
void clear();
int size() const;
bool empty() const;
bool operator == (const ArbolBinario<Tbase>& v) const;
bool operator != (const ArbolBinario<Tbase>& v) const;
template<class T>
friend istream& operator >> (istream& in, ArbolBinario<T>& v);
template<class T>
friend ostream& operator << (ostream& out, const ArbolBinario<T>& v);
~ArbolBinario();

```

Private:

```

struct nodo
{
    Tbase etiqueta;
    struct nodo *izqda;
    struct nodo *drcha;
    struct nodo *padre;
};

struct nodo *laraiz;

void destruir(nodo * n);
void copiar(nodo *& dest, nodo * orig);
void contar(nodo * n);
bool soniguales(nodo * n1, nodo * n2);
void escribe_arbol(ostream& out, nodo * nod) const;
void lee_arbol(istream& in, nodo *& nod);
}

```

IMPLEMENTACIÓN

```
template <class Tbase>
inline ArbolBinario<Tbase>::ArbolBinario()
{
    laraiz=0;
}
```

```
template <class Tbase>
ArbolBinario<Tbase>::ArbolBinario(const Tbase& e)
{
    laraiz = new nodo;
    laraiz->padre= laraiz->izqda= laraiz->drcha= 0;
    laraiz->etiqueta= e;
}
```

```
template <class Tbase>
ArbolBinario<Tbase>::ArbolBinario (const ArbolBinario<Tbase>& v)
{
    copiar (laraiz,v.laraiz);
    if (laraiz!=0)
        laraiz->padre= 0;
}
```

```
template <class Tbase>
void ArbolBinario<Tbase>::destruir(Nodo n)
{
    if (n!=0) {
        destruir(n->izqda);
        destruir(n->drcha);
        delete n;
    }
}
```

```
}
```

```
template <class Tbase>
void ArbolBinario<Tbase>::copiar(Nodo& dest, Nodo orig)
{
    if (orig==0)
        dest= 0;
    else {
        dest= new nodo;
        dest->etiqueta= orig->etiqueta;
        copiar (dest->izqda,orig->izqda);
        copiar (dest->drcha,orig->drcha);
        if (dest->izqda!=0)
            dest->izqda->padre= dest;
        if (dest->drcha!=0)
            dest->drcha->padre= dest;
    }
}
```

```
template <class Tbase>
void ArbolBinario<Tbase>::contar(Nodo n)
{
    if (n==0)
        return 0;
    else return 1+contar(n->izqda)+contar(n->drcha);
}
```

```
template <class Tbase>
bool ArbolBinario<Tbase>::soniguales(Nodo n1, Nodo n2)
{
    if (n1==0 && n2==0)
        return true;
```

```

if (n1==0 || n2==0)
    return false;
if (n1->etiqueta!=n2->etiqueta)
    return false;
if (!soniguales(n1->izqda,n2->izqda))
    return false;
if (!soniguales(n1->drcha,n2->drcha))
    return false;
return true;
}

```

```

template <class Tbase>
void ArbolBinario<Tbase>::lee_arbol(istream& in, Nodo& nod)
{
    char c;
    in >> c;
    if (c=='n')
    {
        nod= new nodo;
        in >> nod->etiqueta;
        lee_arbol(in,nod->izqda);
        lee_arbol(in,nod->drcha);
        if (nod->izqda!=0)
            nod->izqda->padre=nod;
        if (nod->drcha!=0)
            nod->drcha->padre=nod;
    }
    else nod= 0;
}

```

```

template <class Tbase>
void ArbolBinario<Tbase>::escribe_arbol(ostream& out, Nodo nod) const
{

```

```

if (nod==0)
    out << "x ";
else {
    out << "n "<< nod->etiqueta << " ";
    escribe_arbol(out,nod->izqda);
    escribe_arbol(out,nod->drcha);
}
}

```

```

template <class Tbase>
ArbolBinario<Tbase>& ArbolBinario<Tbase>::operator=(const
ArbolBinario<Tbase>& v)
{
    if (this!=&v)
    {
        destruir(laraiz);
        copiar (laraiz,v.laraiz);
        if (laraiz!=0)
            laraiz->padre= 0;
    }
    return *this;
}

```

```

template <class Tbase>
void ArbolBinario<Tbase>::AsignaRaiz(const Tbase& e)
{
    destruir(laraiz);
    laraiz = new nodo;
    laraiz->padre= laraiz->izqda= laraiz->drcha= 0;
    laraiz->etiqueta= e;
}

```



```
template <class Tbase>
inline ArbolBinario<Tbase>::Nodo ArbolBinario<Tbase>::raiz() const
{
    return laraiz;
}
```

```
template <class Tbase>
inline ArbolBinario<Tbase>::Nodo ArbolBinario<Tbase>::izquierda(const Nodo p) const
{
    assert(p!=0);
    return (p->izqda);
}
```

```
template <class Tbase>
ArbolBinario<Tbase>::Nodo ArbolBinario<Tbase>::derecha(const Nodo p) const
{
    assert(p!=0);
    return (p->drcha);
}
```

```
template <class Tbase>
ArbolBinario<Tbase>::Nodo ArbolBinario<Tbase>::padre(const Nodo p)
const
{
    assert(p!=0);
    return (p->padre);
}
```

```
template <class Tbase>
Tbase& ArbolBinario<Tbase>::etiqueta(const Nodo p)
{
}
```

```
    assert(p!=0);  
    return (p->etiqueta);  
}
```

```
template <class Tbase>  
const Tbase& ArbolBinario<Tbase>::etiqueta(const Nodo p) const  
{  
    assert(p!=0);  
    return (p->etiqueta);  
}
```

```
template <class Tbase>  
void ArbolBinario<Tbase>::asignar_subarbol(const  
ArbolBinario<Tbase>& orig,  
    const Nodo nod)  
{  
    destruir(laraiz);  
    copiar(laraiz,nod);  
    if (laraiz!=0)  
        laraiz->padre=0;  
}
```

```
template <class Tbase>  
void ArbolBinario<Tbase>::podar_izquierda(Nodo n,  
    ArbolBinario<Tbase>& dest)  
{  
    assert(n!=0);  
    destruir(dest.laraiz);  
    dest.laraiz=n->izqda;  
    if (dest.laraiz!=0)  
    {  
        dest.laraiz->padre=0;  
    }
```

```
        n->izqda=0;
    }
}
```

```
template <class Tbase>
void ArbolBinario<Tbase>::podar_derecha(Nodo n,
                                         ArbolBinario<Tbase>& dest)
{
    assert(n!=0);
    destruir(dest.laraiz);
    dest.laraiz=n->drcha;
    if (dest.laraiz!=0)
    {
        dest.laraiz->padre=0;
        n->drcha=0;
    }
}
```

```
template <class Tbase>
void ArbolBinario<Tbase>::insertar_izquierda(Nodo n,
                                              ArbolBinario<Tbase>& rama)
{
    assert(n!=0);
    destruir(n->izqda);
    n->izqda=rama.laraiz;
    if (n->izqda!=0)
    {
        n->izqda->padre= n;
        rama.laraiz=0;
    }
}
```

```
template <class Tbase>
void ArbolBinario<Tbase>::insertar_derecha (Nodo n,
                                           ArbolBinario<Tbase>& rama)
{
    assert(n!=0);
    destruir(n->drcha);
    n->drcha=rama.laraiz;
    if (n->drcha!=0)
    {
        n->drcha->padre= n;
        rama.laraiz=0;
    }
}
```

```
template <class Tbase>
void ArbolBinario<Tbase>::clear()
{
    destruir(laraiz);
    laraiz= 0;
}
```

```
template <class Tbase>
inline int ArbolBinario<Tbase>::size() const
{
    return contar(laraiz);
}
```

```
template <class Tbase>
inline bool ArbolBinario<Tbase>::empty() const
{
    return laraiz==0;
}
```

```
template <class Tbase>
inline      bool      ArbolBinario<Tbase>::operator      ==      (const
ArbolBinario<Tbase>& v) const
{
    return soniguales(laraiz,v.laraiz);
}
```

```
template <class Tbase>
inline      bool      ArbolBinario<Tbase>::operator      !=      (const
ArbolBinario<Tbase>& v) const
{
    return !(*this==v);
}
```

```
template <class Tbase>
inline istream& operator>> (istream& in, ArbolBinario<Tbase>& v)
{
    v.lee_arbol(in,v.laraiz);
    return in;
}
```

```
template <class Tbase>
inline      ostream&      operator<<      (ostream&      out,      const
ArbolBinario<Tbase>& v)
{
    v.escribe_arbol(out,v.laraiz);
    return out;
}
```

```
template <class Tbase>
```

```
inline ArbolBinario<Tbase>::~~ArbolBinario()  
{  
    destruir(laraiz);  
}
```

RECORRIDOS EN EL ÁRBOL BINARIO

RECORRIDO EN PREORDEN DE UN ÁRBOL BINARIO

- Esta función escribe en la salida estándar el preorden del subárbol que cuelga del nodo del árbol a.

```
void preorden (const ArbolBinario<int>& a, const ArbolBinario<int>::Nodo n)  
{  
    if (n != ArbolBinario<int>::nodo_nulo)  
    {  
        cout << a.etiqueta(n) << ' ' ;  
        preorden(a,a.izquierda(n));  
        preorden(a,a.derecha(n));  
    }  
}
```

RECORRIDO EN INORDEN DE UN ÁRBOL BINARIO

- Esta función escribe en la salida estándar el inorden del subárbol que cuelga del nodo del árbol a.

```
void inorden(const ArbolBinario<int>& a, const ArbolBinario<int>::Nodo n)  
{  
    if (n != ArbolBinario<int>::nodo_nulo)
```

```

{
    inorden(a,a.izquierda(n));
    cout << a.etiqueta(n) << ' ';
    inorden(a,a.derecha(n));
}
}

```

RECORRIDO EN POSTORDEN DE UN ÁRBOL BINARIO

- Esta función escribe en la salida estándar el postorden del subárbol que cuelga del nodo del árbol a.

```

void      postorden      (const      ArbolBinario<int>&      a,      const
ArbolBinario<int>::Nodo n)
{
    if (n!=ArbolBinario<int>::nodo_nulo)
    {
        postorden(a,a.izquierda(n));
        postorden(a,a.derecha(n));
        cout << a.etiqueta(n) << ' ';
    }
}

```