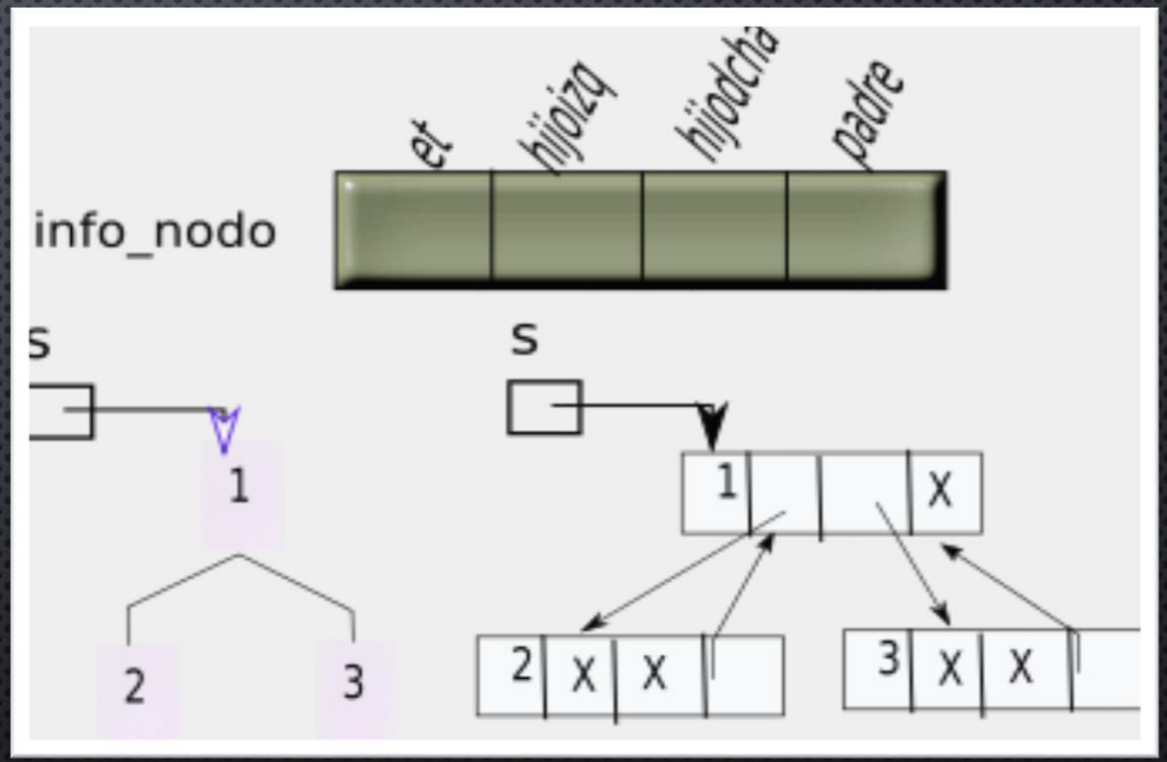
ARBOLES: ARBOLES BINARIOS (AB)

Arbol Binario.- Son árboles en el que cada nodo puede tener 0, 1 o 2 hijos. El árbol vacío es un árbol binario.

Representación (Aproximacíón 1).-

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```



ARBOLES: ARBOLES BINARIOS (AB)

Arbol Binario.- Son árboles en el que cada nodo puede tener 0, 1 o 2 hijos. El árbol vacío es un árbol binario.

Representación (Aproximacíón 1).-

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

Operaciones:

- Modificadores: Copiar, Borrar, Podar, Insertar
- Consultores: getPadre, getHijoIzq,getHijoDere,etiqueta, numero de nodos
- Escritura/Lectura
- Recorrido en Profundidad y Anchura.

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoiizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
info_nodo<T> * getPadre(info_nodo<T> *n){
    return n->padre;
}

template <class T>
info_nodo<T> * getHijoIzq (info_nodo<T> *n){
    return n->hijoizq;
}

template <class T>
info_nodo<T> * getHijoDer (info_nodo<T> *n){
    return n->hijoder;
}
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void Copiar(const info_nodo<T> *s, info_nodo<T> *&d){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void BorrarInfo(info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void InsertarHijoDerecha(info_nodo<T> *n,
                         info_nodo<T>*sub){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void InsertarHijoDerecha(info_nodo<T> *n,
                         const T &e){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void InsertarHijoIzquierda(info_nodo<T> *n,
                           info_nodo<T>*sub){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void InsertarHijoIzquierda(info_nodo<T> *n,
                           const T &e){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void PodarHijoIzquierda(info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
info_nodo<T>
*PodarHijoIzquierdaGetSubtree(info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void PodarHijoDerecha (info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
info_nodo<T>
*PodarHijoDerechaGetSubtree(info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void RecorridoPreorden(info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void RecorridoInorden(info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void RecorridoPostorden(info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void RecorridoNiveles (info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
bool iguales (info_nodo<T> *n1,info_nodo<T> *n2){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
bool numero_nodos(info_nodo<T> *n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void Lee(istream & is, info_nodo<T> * &n){
```

Arbol Binario.-Implementación

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    info_nodo(){
      padre=0;
      hijozq=hijoder=0;
    }
    info_nodo(const T &e){
     et=e;
     padre=0;
     hijozq=hijoder=0;
    }
};
```

```
template <class T>
void Escribir(ostream & os, info_nodo<T> *n){
```

ARBOLES: ARBOLES BINARIOS (AB)

Arbol Binario.-Resumen de la representación con struct info_nodo.

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    …
};
```

```
info_nodo<T> & getPadre(info_nodo<T> &n);
info_nodo<T> & getHijoIzq(info_nodo<T> &n);
info_nodo<T> & getHijoDer(info_nodo<T> &n);
void Copiar(const info_nodo<T>* s, info_nodo<T> *&d);
void BorrarInfo(info_nodo<T>*n);
void InsertarHijoDerecha(info_nodo<T> *n,
                         info_nodo<T>*sub);
void InsertarHijoDerecha(info_nodo<T> *n,
                               const T*e);
void InsertarHijoIzquierda (info_nodo<T> *n,
                         info_nodo<T>*sub);
void InsertarHijoIzquierda(info_nodo<T> *n,
                               const T*e);
void PodarHijoIzquierda(info_nodo<T> *n);

info_nodo<T>
*PodarHijoIzquierdaGetSubtree(info_nodo<T> *n);

void PodarHijoDerecha (info_nodo<T> *n);

info_nodo<T> *PodarHijoDerechaGetSubtree(info_nodo<T>
*n);
```

Arbol Binario.-Resumen de la representación con struct info_nodo.

```
template <class T>
struct info_nodo{
    T et;
    info_nodo * padre;
    info_nodo * hijoizq;
    info_nodo * hijoder;
    …
};
```

```
void RecorridoPreorden (info_nodo<T> *n);
void RecorridoIntorden (info_nodo<T> *n);
void RecorridoPostorden (info_nodo<T> *n);
void RecorridoNiveles (info_nodo<T> *n);
bool iguales (info_nodo<T> *n1,info_nodo<T>*n2);
bool numero_nodos(info_nodo<T> *n);
void Escribir(ostream & os, info_nodo<T> *n);
void Lee(istream & is, info_nodo<T> *n);
```

Arbol Binario.-Representación mediante la clase ArbolBinario (bintree)

```
template <class T>
class ArbolBinario{
 private:
   struct info_nodo{
     …
   };
   //Operaciones de las
   //anteriores  transparencias

   info_nodo *raíz;

 public:
```

```
Arbolbinario():raiz(nullptr){}

Arbolbinario(const T & e){
     raíz = new info_nodo<T>(e );
}
Arbolbinario(const Arbolbinario<T> & ab): {
     Copiar(ab.raiz,raiz);
}

Arbolbinario<T> & operator=(const Arbolbinario<T>
&ab){
     if (this!=&ab){
          BorrarInfo();
          Copiar(ab.raiz,raiz);
     }
     return *this;
}
```

Arbol Binario.-Representación mediante la clase ArbolBinario (bintree)

```
template <class T>
class ArbolBinario{
 private:
   struct info_nodo{
     …
   };
   //Operaciones de las
   //anteriores  transparencias

   info_nodo *raíz;

 public:
    class nodo{
        private:
          info_nodo *p;
```

```
public:
    nodo(info_nodo <T>*i){ p = i; }
    nodo():p(nullptr){}
    nodo (const nodo <T> &s):p(s.p){}
    const T & operator*(){ return p->et;}
    const T & operator*()const{ return p->et;}
    nodo padre() const{ return p->padre;}
    nodo hizq() const{ return p->hijoizq;}
    nodo hder() const{ return p->hijodrch;}
    bool null()const{ return p==nullptr;}
    bool operator!=(const nodo & s){
        return p!=s.p;}
    bool operator==(const nodo &s){
        return p==s.p;
    }
    friend class Arbolbinario;
    friend class preorder_iterator;
    friend class inorder_iterator;
    friend class postorder_iterator;
};//end nodo
```

Arbol Binario.-Representación mediante la clase ArbolBinario (bintree)

```
template <class T>
class ArbolBinario{
 private:
  struct info_nodo{
     …
  };
  info_nodo *raíz;
  public:
```

```
typedef typename arbolbinario<T>::nodo  tipoNodo;

tipoNodo getraiz(){
        if (raiz!=0)
          return tipoNodo(raiz);
        else
        return tipoNodo();
};
tipoNodo Insertar_hi(tipoNodo n, const <T>&e);
        InsertarHijoIzquierda(n.p,e);
        return tipoNodo(n.p->hizq);
}

tipoNodo Insertar_hi(tipoNodo n, arbolbinario <T>& tree){
        InsertarHijoIzquierda(n.p,tree.raiz);
        tree.raiz=0;
        return tipoNodo(n.p->hizq);
}
```

Arbol Binario.-Representación mediante la clase ArbolBinario (bintree)

```
template <class T>
class ArbolBinario{
 private:
  struct info_nodo{
     …
  };
  info_nodo *raíz;
  public:
```

```
typedef typename Arbolbinario<T>::nodo  tipoNodo;

tipoNodo Insertar_hd(tipoNodo n, const <T>&e){
   InsertarHijoDerecha(n.p,e);
   return tipoNodo(n.p->hder);
}
tipoNodo Insertar_hd(tipoNodo pos, Arbolbinario <T>& tree)
{
        InsertarHijoDerecha(n.p,tree.raiz);
        tree.raiz=0;
        return tipoNodo(n.p->hder);
}

void Poda_hd(tipoNodo n){
   PodarHijoDerecha (n.p);
}
```

Arbol Binario.-Representación mediante la clase ArbolBinario (bintree)

```
template <class T>
class ArbolBinario{
 private:
  struct info_nodo{
     …
  };
  info_nodo *raíz;
  public:
```

```
typedef typename Arbolbinario<T>::nodo  tipoNodo;

void Poda_hi(tipoNodo n){
   PodarHijoIzquierda(n.p);
}
ArbolBinario<T> PodarHi_GetSubtree(tipoNodo pos){
  typename ArbolBinario<T>::info_nodo *
       aux=PodarHijoIzq_GetSubtree(pos.p);
  if (aux!=0) aux->padre=0;
  ArbolBinario<T> anuevo.raiz=aux;
  return anuevo;
}
ArbolBinario<T> PodarHd_GetSubtree(tipoNodo pos){
…
}
```

Arbol Binario.-Representación mediante la clase ArbolBinario (bintree)

```
template <class T>
class ArbolBinario{
 private:
  struct info_nodo{
     …
  };
  info_nodo *raíz;
  public:
```

```
typedef typename Arbolbinario<T>::nodo  tipoNodo;
void clear(){
        BorrarInfo(raiz);
        raiz=0;
}
bool empty()const {
        return raiz==0;
}
unsigned int size()const {
        return numero_nodos(raiz);
}
bool operator==(const ArbolBinario<T> &a)const {
        return iguales(raiz,a.raiz);
}
bool operator!=(const ArbolBinario<T> &a)const{
        return !(*this==a);
}
```

Arbol Binario.-Representación mediante la clase ArbolBinario (bintree)

```
template <class T>
class ArbolBinario{
 private:
  struct info_nodo{
     …
  };
  info_nodo *raíz;
  public:
```

```
typedef typename Arbolbinario<T>::nodo  tipoNodo;
void RecorridoPreOrden(ostream &os)const {
        RecorridoPreorden(os,raiz);
}
void RecorridoInOrden(ostream &os)const {
        RecorridoInorden(os,raiz);
}
voidRecorridoPostOrden(ostream &os)const {
        RecorridoPostorden(os,raiz);
}
void RecorridoNiveles(ostream &os)const {
        RecorridoNiveles(os,raiz);
}
friend ostream & operator<<(ostream &os,const
                            ArbolBinario<T> &ab){
        ab.Escribe(os,ab.raiz); return os;
}
friend istream & operator>>(istream &is,ArbolBinario<T> &ab){
        ab.Leer (is,ab.raiz); return is;
}
```

Arbol Binario.-Representación mediante la clase ArbolBinario (bintree)

```
template <class T>
class ArbolBinario{
 private:
   struct info_nodo{
     …
   };

   info_nodo *raíz;

 public:
    class nodo{
        private:
          info_nodo *p;
```

```
public:
    …
    friend class arbolbinario;
    friend class preorder_iterator;
    friend class inorder_iterator;
    friend class postorder_iterator;
};//end nodo

class preorder_iterator{
  private:
    info_nodo *p;
  public:
     preorden_iterator(const nodo &n):p(n.p){}
     //resto operaciones igual que nodo
     //excepto operator ++
     …
}
```

```
template <class T>
class ArbolBinario{
 private:
   struct info_nodo{
     …
   };

   info_nodo *raíz;

 public:
```

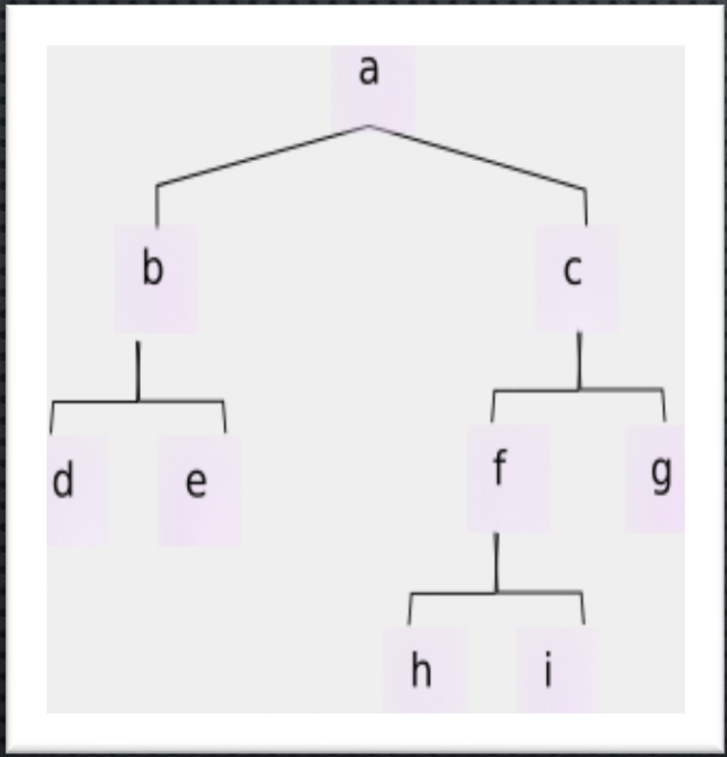
```
preorden_iterator begin_preorden();
preorden_iterator end_preorden();

inorden_iterator begin_inorden();
inorden_iterator end_inorden();


postorden_iterator begin_postorden();
postorden_iterator end_postorden();
```

ARBOLES: ARBOLES BINARIOS (AB)

Ejercicio 1.- Dar las sentencias para crear el AB que se muestra en la figura.



ARBOLES: ARBOLES BINARIOS (AB)

Ejercicio 2.-Crear una función que dado un AB de enteros nos devuelva true si todos los elementos del árbol son pares o false en caso contrario

ARBOLES: ARBOLES BINARIOS (AB)

Ejercicio 3.-Crear una función que dado un AB de enteros obtenga un nuevo árbol que es el reflejado del árbol de entrada.

ARBOLES: ARBOLES BINARIOS (AB)

Ejercicio 4.-Crear una función que dado un AB de enteros obtenga la suma de los nodos por niveles.

ARBOLES: ARBOLES BINARIOS (AB)

Ejercicio 5.-Crear una función que dado un AB de enteros nos devuelva true si el recorrido en preorden es igual al recorrido en inorden. (Por ejemplo si todas las etiquetas tienen el mismo valor será true).

ARBOLES: ARBOLES BINARIOS (AB)

Ejercicio 6.-Crear una función que dado dos AB (plantilla) nos diga si son reflejados, true, o en caso contrario false.