```
#ifndef __BINTREE_H__
#define __BINTREE_H__

/**
   TDA bintree.

   Representa un árbol binario con nodos etiquetados con datos del tipo T.

   T debe tener definidas las operaciones:

   − T & operator=(const T & e);
   − bool operator!=(const T & e);
   − bool opertaor==(const T & e);

   Son mutables.
   Residen en memoria dinámica.

   Un ejemplo de su uso:
   @include usobintree.cpp
   @author{Miguel Garcia Silvente}
   @author{Juan F. Huete Guadix}
*/

#include <queue>


template <typename T>
class bintree {
public:
  class node;

  typedef unsigned int size_type;

  /**
     @brief Constructor primitivo por defecto.

     Crea un árbol nulo.
  */
  bintree();

  /**
     @brief Constructor primitivo.

     @param e Etiqueta para la raíz.

     Crea un árbol con un único nodo etiquetado con e.
  */
  bintree(const T & e);

  /**
     @brief Constructor de copia.

     @param a árbol que se copia.

     Crea un árbol duplicado exacto de a.
  */
  bintree (const bintree<T> & a);

  /**
     @brief Reemplaza el receptor por una copia de subárbol.

     @param a Arbol desde el que se copia.
```



```
     @param n nodo raíz del subárbol que se copia.

     El receptor se hace nulo y después se le asigna una copia
     del subárbol de a cuya raíz es n.
  */
  void assign_subtree(const bintree<T> & a, node n);

  /**
     @brief Destructor.

     Destruye el receptor liberando los recursos que ocupaba.
  */
  ~bintree();

  /**
     @brief Operador de asignación.

     @param a: árbol que se asigna.

     Destruye el contenido previo del receptor y le asigna un
     duplicado de a.
  */
  bintree<T> & operator=(const bintree<T> & a);

  /**
     @brief Obtener el nodo raíz.

     @return nodo raíz del receptor.
  */
  node root() const;

  /**
     @brief Podar el subárbol a la izquierda de un nodo.

     @param n: nodo del receptor. !n.null().
     @param dest: subárbol a la izquierda de n. Es MODIFICADO.

     Desconecta el subárbol a la izquierda de n, que pasa a
     ser un árbol nulo. El subárbol anterior se devuelve sobre
     dest.
  */
  void prune_left(node n, bintree<T> & dest);

  /**
     @brief Podar el subárbol a la derecha de un nodo.

     @param n: nodo del receptor. !n.null().
     @param dest: subárbol a la derecha de n. Es MODIFICADO.

     Desconecta el subárbol a la derecha de n, que pasa a
     ser un árbol nulo. El subárbol anterior se devuelve sobre
     dest.
  */
  void prune_right(node n, bintree<T> & dest);

  /**
     @brief Insertar un nodo como hijo a la izquierda de un nodo.

     @param n: nodo del receptor. !n.null().
     @param e: etiqueta del nuevo nodo.

     Desconecta y destruye el subárbol a la izquierda de n, inserta
```

```
   un nuevo nodo con etiqueta e como hijo a la izquierda
*/
void insert_left(const bintree<T>::node & n, const T & e);

/**
   @brief Insertar un árbol como subárbol a la izquierda de un nodo.

   @param n: nodo del receptor. n != nodo_nulo.
   @param rama: subárbol que se inserta. Es MODIFICADO.

   Desconecta y destruye el subárbol a la izquierda de n, le
   asigna el valor de rama como nuevo subárbol a la izquierda
   y rama se hace árbol nulo.
*/
void insert_left(node n, bintree<T> & rama);

/**
   @brief Insertar un nodo como hijo a la derecha de un nodo.

   @param n: nodo del receptor. !n.Nulo().
   @param e: etiqueta del nuevo nodo.

   Desconecta y destruye el subárbol a la derecha de n, inserta
   un nuevo nodo con etiqueta e como hijo a la derecha
*/
void insert_right(node n, const T & e);

/**
   @brief Insertar un árbol como subárbol a la derecha de un nodo.

   @param n: nodo del receptor. !n.Nulo().
   @param rama: subárbol que se inserta. Es MODIFICADO.

   Desconecta y destruye el subárbol a la izquierda de n, le
   asigna el valor de rama como nuevo subárbol a la derecha
   y rama se hace árbol nulo.
*/
void insert_right(node n, bintree<T> & rama);

/**
   @brief Hace nulo un árbol.

   Destruye todos los nodos del árbol receptor y lo hace
   un árbol nulo.
*/
void clear();

/**
   @brief Obtiene el número de nodos.

   @return número de nodos del receptor.
*/
size_type size() const;

/**
   @brief Comprueba si un árbol está vacío (es nulo).

   @return true, si el receptor está vacío (es nulo).
           false, en otro caso.
*/
bool empty() const;
```



```
/**
   @brief Operador de comparación de igualdad.

   @param a: árbol con que se compara el receptor.

   @return  true, si el receptor es igual, en estructura y
               etiquetas a a.
            false, en otro caso.
*/
bool operator==(const bintree<T> & a) const;

/**
   @brief Operador de comparación de desigualdad.

   @param a: árbol con que se compara el receptor.

   @return  true, si el receptor no es igual, en estructura o
                  etiquetas a a.
            false, en otro caso.
*/
bool operator!=(const bintree<T> & a) const;


/**
   @brief Reemplaza el subárbol a partir de pos por una copia de subárbol.

   @param pos nodo a partir del que se colagará la copia
   @param a Arbol desde el que se copia.
   @param n nodo raíz del subárbol que se copia.

   El receptor se modifica colocando a partir de pos una copia
   del subárbol de a cuya raíz es n.
*/
void replace_subtree(node pos, const bintree<T> &a, node n);

/**
   Clase iterator para recorrer el árbol en PreOrden
*/
class preorder_iterator {
public:
  preorder_iterator();
  preorder_iterator(const preorder_iterator & i);
  bool operator!=(const preorder_iterator & i) const;
  bool operator==(const preorder_iterator & i) const;
  preorder_iterator & operator=(const preorder_iterator & i);
  T & operator*();
  preorder_iterator & operator++();
private:
  node elnodo;
  preorder_iterator(node n);
  friend class bintree<T>;
};

preorder_iterator begin_preorder();
preorder_iterator end_preorder();

class const_preorder_iterator
{
public:
  const_preorder_iterator();
  const_preorder_iterator(const const_preorder_iterator & i);
```

```
    const_preorder_iterator(const preorder_iterator & i);
    bool operator!=(const const_preorder_iterator & i) const;
    bool operator==(const const_preorder_iterator & i) const;
    const_preorder_iterator & operator=(const const_preorder_iterator & i);
    const T & operator*() const;
    const_preorder_iterator & operator++();
  private:
    node elnodo;
    const_preorder_iterator(node n);
    friend class bintree<T>;
 };

  const_preorder_iterator begin_preorder() const;
  const_preorder_iterator end_preorder() const;



  /**
     Clase iterator para recorrer el árbol en Inorden
  */

 class inorder_iterator
 {
  public:
    inorder_iterator();
    inorder_iterator(const inorder_iterator & i);
    bool operator!=(const inorder_iterator & i) const;
    bool operator==(const inorder_iterator & i) const;
    inorder_iterator & operator=(const inorder_iterator & i);
    T & operator*();
    inorder_iterator & operator++();
  private:
    node elnodo;
    inorder_iterator(node n);
    friend class bintree<T>;
 };

  inorder_iterator begin_inorder();
  inorder_iterator end_inorder();

class const_inorder_iterator
 {
  public:
    const_inorder_iterator();
    const_inorder_iterator(const const_inorder_iterator & i);
    bool operator!=(const const_inorder_iterator & i) const;
    bool operator==(const const_inorder_iterator & i) const;
    const_inorder_iterator & operator=(const const_inorder_iterator & i);
    const T & operator*() const;
    const_inorder_iterator & operator++();
  private:
    node elnodo;
    const_inorder_iterator(node n);
    friend class bintree<T>;
 };

  const_inorder_iterator begin_inorder() const;
  const_inorder_iterator end_inorder() const;


  /**
     Clase iterator para recorrer el árbol en PostOrden
```



```
  */

 class postorder_iterator
 {
  public:
    postorder_iterator();
    postorder_iterator(const postorder_iterator & i);
    bool operator!=(const postorder_iterator & i) const;
    bool operator==(const postorder_iterator & i) const;
    postorder_iterator & operator=(const postorder_iterator & i);
    T & operator*();
    postorder_iterator & operator++();
  private:
    node elnodo;
    postorder_iterator(node n);
    friend class bintree<T>;
  };

  postorder_iterator begin_postorder();
  postorder_iterator end_postorder();

class const_postorder_iterator
 {
  public:
    const_postorder_iterator();
    bool operator!=(const const_postorder_iterator & i) const;
    bool operator==(const const_postorder_iterator & i) const;
    const T & operator*() const;
    const_postorder_iterator & operator=(const const_postorder_iterator & i);
    const_postorder_iterator & operator++();
  private:
    node elnodo;
    const_postorder_iterator(node n);
    friend class bintree<T>;
  };
  const_postorder_iterator begin_postorder() const;
  const_postorder_iterator end_postorder() const;


  /**
     Clase iterator para recorrer el árbol por niveles
  */

 class level_iterator
 {
  public:
    level_iterator();
    level_iterator(const level_iterator & i);
    bool operator!=(const level_iterator & i) const;
    bool operator==(const level_iterator & i) const;
    level_iterator & operator=(const level_iterator & i);
    T & operator*();
    level_iterator & operator++();
  private:
    std::queue<node> cola_Nodos;
    level_iterator(node n);
    friend class bintree<T>;
  };

  level_iterator begin_level();
  level_iterator end_level();
```

```
class const_level_iterator
 {
  public:
    const_level_iterator();
    bool operator!=(const const_level_iterator & i) const;
    bool operator==(const const_level_iterator & i) const;
    const_level_iterator & operator=(const const_level_iterator & i);
    const T & operator*() const;
    const_level_iterator & operator++();
  private:
    std::queue<node> cola_Nodos;
    const_level_iterator(node n);
    friend class bintree<T>;
  };

  const_level_iterator begin_level() const;
  const_level_iterator end_level() const ;


private:

  // Funciones auxiliares
  /**
     @brief Destruir subárbol.

     @param n: nodo raíz del subárbol que se destruye.
     @doc:
     Destruye el subárbol cuya raíz es n.
  */
  void destroy(bintree<T>::node n);


  /**
     @brief Copia subárbol.

     @param dest: nodo sobre el que se copia. dest.null().
            Es MODIFICADO.
     @param orig: raíz del subárbol que se copia.

     @doc
     Destruye el subárbol con raíz en dest. Sobre éste realiza
     un duplicado del subárbol con raíz en orig.
  */
  void copy(node & dest, const node &orig);


  /**
     @brief Cuenta el número de nodos.

     @param n: raíz del subárbol a contar.

     @return devuelve el número de nodos del subárbol que
             tiene n como raíz.

     Cuenta el número de nodos en el subárbol cuuya raíz es n.
  */
  int count(node n) const;

  /**
     @brief Comparación de igualdad.

     @param n1: raiz del primer subárbol.
```



```
     @param n2: raiz del segundo subárbol.

     @return true, si los dos subárboles son iguales, en
                   estructura y etiquetas.
             false, en otro caso.
  */
  bool equals(node n1, node n2) const;

  // Representación
  node laraiz;
  size_type num_nodos;


  /**
      TDA nodo.
      Modela los nodos del árbol binario.
  */

  class nodewrapper {
  public:
    nodewrapper();
    nodewrapper(const T & e);

    T etiqueta;
    node pad;
    node izda;
    node dcha;
  };

public:

  class node {
  public:
    /**
       @brief Constructor primitivo
    */
    node();

    /**
       @brief Constructor primitivo
       @param e: Etiqueta del nodo
    */
    node(const T & e);

    /**
       @brief Constructor de copia
       @param n: Nodo que se copia
    */
    node(const node & n);

    /**
       @brief Determina si el nodo es nulo
    */
    bool null() const;

    /**
       @brief Devuelve el padre del nodo receptor
       @pre !null()
    */
    node parent() const;

    /**
```

```
      @brief Devuelve el hizo izquierdo del nodo receptor
      @pre !null()
   */
   node left() const;

   /**
      @brief Devuelve el hizo izquierdo del nodo receptor
      @pre !null()
   */
   node right() const;

   /**
      @brief Devuelve la etiqueta del nodo
      @pre Si se usa como consultor, !null()
   */
   T & operator*();

   /**
      @brief Devuelve la etiqueta del nodo
      @pre !null()
     */
   const T & operator*() const;

   /**
      @brief Elimina el nodo actual
      @pre !null()
   */
   void remove();

   /**
      @brief Operador de asignación
      @param n: el nodo a asignar
   */
   node & operator=(const node & n);

   /**
      @brief Operador de comparación de igualdad
      @param n: el nodo con el que se compara
   */
   bool operator==(const node & n) const;

   /**
      @brief Operador de comparación de desigualdad
      @param n: el nodo con el que se compara
   */
   bool operator!=(const node & n) const;

 private:
   // Las siguientes funciones son privadas para uso exclusivo en bintree
   friend class bintree<T>;

   /**
      @brief Coloca el nodo padre de un nodo
      @param n El nodo que ser~ padre del receptor. No nulo.
   */
   inline void parent(node n);

   /**
      @brief Coloca el nodo hijo izquierda de un nodo
      @param n El nodo que ser~ hijo izquierdo del receptor. No nulo
   */
   inline void left(node n);
```



```
   /**
      @brief Coloca el nodo hijo derecho de un nodo
      @param n El nodo que ser~ hijo derecho del receptor No nulo
   */
   inline void right(node n);

   nodewrapper * elnodo;
  };
};

#include "bintree.hxx"
#include "node.hxx"


#endif
```