```
//−*−Mode: C++;−*−


/*
************************************************************
* Implementaci¨®n
************************************************************
*/

/*
  Funci¨®n de Abstracci¨®n:
  ------------------------
  Dado el objeto del tipo rep r, r = {laraiz}, el objeto
  abstracto al que representa es:
  a) Arbol nulo, si r.laraiz.null().
  b) Arbol con un ¨²nico nodo de etiqueta *(r.laraiz)
     si r.laraiz.left().null() y r.laraiz.dcha().null()

  c)                    *(r.laraiz)
                         /       \
                        /         \
      Arbol(r.laraiz.left())    Arbol(r.laraiz.right())


  Invariante de Representaci¨®n:
  -----------------------------
  Si !r.laraiz.null(),
  −   r.laraiz.parent().null().

  Para cualquier nodo n del ¨¢rbol:
  Si !n.left().null()
      n.left().parent() == n;
  Si !n.right().null()
      n.right().parent() == n;

*/


#include <cassert>


/*____________________________________________________________________ */
/*____________________________________________________________________ */
//                FUNCIONES PRIVADAS
/*____________________________________________________________________ */
/*____________________________________________________________________ */

template <typename T>
void bintree<T>::destroy(typename bintree<T>::node n)
{
  if (!n.null()) {
    destroy(n.left());
    destroy(n.right());
    n.remove();
  }
}

/*____________________________________________________________________ */

template <typename T>
void bintree<T>::copy(bintree<T>::node & dest, const bintree<T>::node & orig)
{
```



```
  if (orig.null())
    dest = typename bintree<T>::node();
  else {
      dest = node(*orig);
      typename bintree<T>::node aux(dest.left());
      copy (aux, orig.left());
      dest.left(aux);
      if (!dest.left().null())
        dest.left().parent(dest);
      aux = dest.right();
      copy (aux, orig.right());
      dest.right(aux);
      if (!dest.right().null())
        dest.right().parent(dest);
    }
}

/*____________________________________________________________________ */

template <typename T>
int bintree<T>::count(bintree<T>::node n) const
{
  if (n.null())
    return 0;
  else
    return 1 + count(n.left()) + count(n.right());
}

/*____________________________________________________________________ */

template <typename T>
bool bintree<T>::equals(bintree<T>::node n1, bintree<T>::node n2) const
{
  if (n1.null() && n2.null())
    return true;
  if (n1.null() || n2.null())
    return false;
  if (*n1 != *n2)
    return false;
  if (!equals(n1.left(),n2.left()))
    return false;
  if (!equals(n1.right(),n2.right()))
    return false;
  return true;
}


/*____________________________________________________________________ */
/*____________________________________________________________________ */
//                FUNCIONES PUBLICAS
/*____________________________________________________________________ */
/*____________________________________________________________________ */


template <typename T>
inline
bintree<T>::bintree()
  : num_nodos(0)
{
}

/*____________________________________________________________________ */
```

```
template <typename T>
inline
bintree<T>::bintree(const T & e)
  : laraiz(e), num_nodos(1)
{
}

/*_____________________________________________________________ */

template <typename T>
inline
bintree<T>::bintree(const bintree<T> & a)
{
  copy(laraiz, a.laraiz);
  if (!laraiz.null())
    laraiz.parent(typename bintree<T>::node());
  num_nodos = a.num_nodos;
}

/*_____________________________________________________________ */

template <typename T>
void bintree<T>::assign_subtree(const bintree<T> & a,
                                typename bintree<T>::node n)
{
  if (&a != this) {
    destroy(laraiz);
    num_nodos = count(n);
    copy(laraiz, n);
    if (!laraiz.null())
      laraiz.parent(typename bintree<T>::node());
  } else {  // Reemplazar el receptor por un subárbol suyo.
    if (laraiz != n) {
      typename bintree<T>::node nod(laraiz);
      num_nodos = count(n);
      laraiz = n;
      if (!n.null()) {
        typename bintree<T>::node aux(n.parent());
        if (n.parent().left() == n)
          aux.left(typename bintree<T>::node());
        else
          aux.right(typename bintree<T>::node());
      }
      destroy(nod);
      laraiz.parent(typename bintree<T>::node());
    }
  }
}

/*_____________________________________________________________ */

template <typename T>
inline
bintree<T>::~bintree()
{
  destroy(laraiz);
}

/*_____________________________________________________________ */

template <typename T>
```



```
inline
bintree<T> &
bintree<T>::operator=(const bintree<T> & a)
{
  if (&a != this) {
    destroy(laraiz);
    num_nodos = a.num_nodos;
    copy(laraiz, a.laraiz);
    if (!laraiz.null())
      laraiz.parent(typename bintree<T>::node());
  }
  return *this;
}

/*_____________________________________________________________ */

template <typename T>
inline
typename bintree<T>::node bintree<T>::root() const
{
  return laraiz;
}

/*_____________________________________________________________ */

template <typename T>
inline
void bintree<T>::prune_left(typename bintree<T>::node n,
                            bintree<T> & orig)
{
  assert(!n.null());

  destroy(orig.laraiz);
  num_nodos = num_nodos − count(n.left());
  orig.laraiz = n.left();
  if (!orig.laraiz.null())
    orig.laraiz.parent(typename bintree<T>::node());
  n.left(typename bintree<T>::node());
}


/*_____________________________________________________________ */

template <typename T>
inline
void bintree<T>::prune_right(typename bintree<T>::node n,
                             bintree<T> & orig)
{
  assert(!n.null());

  destroy(orig.laraiz);
  num_nodos = num_nodos − count(n.right());
  orig.laraiz = n.right();
  num_nodos = num_nodos − count(n.right());
  if (!orig.laraiz.null())
    orig.laraiz.parent(typename bintree<T>::node());
  n.right(typename bintree<T>::node());
}

/*_____________________________________________________________ */
template <typename T>
inline
```

```
void bintree<T>::insert_left(const typename bintree<T>::node & n, const T & e)
{
  bintree<T> aux(e);
  insert_left(n, aux);
}

/*_____________________________________________________________ */
template <typename T>
inline
void bintree<T>::insert_left(typename bintree<T>::node n,
                             bintree<T> & rama)
{
  assert(!n.null());

  num_nodos = num_nodos − count(n.left()) + rama.num_nodos;
  typename bintree<T>::node aux(n.left());
  destroy(aux);
  n.left(rama.laraiz);
  if (!n.left().null())
    n.left().parent(n);
  rama.laraiz = typename bintree<T>::node();
}

/*_____________________________________________________________ */
template <typename T>
inline
void bintree<T>::insert_right(typename bintree<T>::node n, const T & e)
{
  bintree<T> aux(e);
  insert_right(n, aux);
}
/*_____________________________________________________________ */

template <typename T>
inline
void bintree<T>::insert_right(typename bintree<T>::node n,
                              bintree<T> & rama)
{
  assert(!n.null());

  num_nodos = num_nodos − count(n.right()) + rama.num_nodos;
  typename bintree<T>::node aux(n.right());
  destroy(aux);
  n.right(rama.laraiz);
  if (!n.right().null())
    n.right().parent(n);
  rama.laraiz = typename bintree<T>::node();
}

/*_____________________________________________________________ */

template <typename T>
void bintree<T>::clear()
{
  destroy(laraiz);
  laraiz = bintree<T>::node();
}

/*_____________________________________________________________ */

template <typename T>
inline
```



```
typename bintree<T>::size_type bintree<T>::size() const
{
  return num_nodos;
}

/*_____________________________________________________________ */

template <typename T>
inline
bool bintree<T>::empty() const
{
  return laraiz == bintree<T>::node();
}

/*_____________________________________________________________ */

template <typename T>
inline
bool bintree<T>::operator==(const bintree<T> & a) const
{
  return equals(laraiz, a.laraiz);
}

/*_____________________________________________________________ */

template <typename T>
inline bool bintree<T>::operator!=(const bintree<T> & a) const
{
  return !(*this == a);
}

/*
  ************************************************************
  Iteradores
  ************************************************************
  */


/*
  Iterador para recorrido en Preorder
*/

template <typename T>
inline
bintree<T>::preorder_iterator::preorder_iterator()
{
  elnodo = typename bintree<T>::node();
}


template <typename T>
inline
bintree<T>::preorder_iterator::preorder_iterator(
                  const bintree<T>::preorder_iterator & i)
  : elnodo(i.elnodo)
{
}


template <typename T>
inline
bintree<T>::preorder_iterator::preorder_iterator(bintree<T>::node n)
```

```
  : elnodo(n)
{
}

template <typename T>
inline
bool bintree<T>::preorder_iterator::operator!=(
        const bintree<T>::preorder_iterator & i) const
{
  return elnodo != i.elnodo;
}


template <typename T>
inline
bool bintree<T>::preorder_iterator::operator==(
             const bintree<T>::preorder_iterator & i) const
{
  return elnodo == i.elnodo;
}


template <typename T>
inline
typename bintree<T>::preorder_iterator &
bintree<T>::preorder_iterator::operator=(
                      const bintree<T>::preorder_iterator & i)
{
  elnodo = i.elnodo;
  return *this;
}


template <typename T>
inline
T & bintree<T>::preorder_iterator::operator*()
{
  return *elnodo;
}


template <typename T>
typename bintree<T>::preorder_iterator &
bintree<T>::preorder_iterator::operator++()
{
  if (elnodo.null())
    return *this;

  if (!elnodo.left().null())
    elnodo = elnodo.left();
  else if (!elnodo.right().null())
      elnodo = elnodo.right();
  else {
    while ((!elnodo.parent().null()) &&
               (elnodo.parent().right() == elnodo ||
                elnodo.parent().right().null()))
      elnodo = elnodo.parent();
    if (elnodo.parent().null())
      elnodo = typename bintree<T>::node();
    else
      elnodo = elnodo.parent().right();
  }
```



```
  return *this;
}


template <typename T>
inline
typename bintree<T>::preorder_iterator
bintree<T>::begin_preorder()
{
  return preorder_iterator(laraiz);
}

template <typename T>
inline
typename bintree<T>::preorder_iterator
bintree<T>::end_preorder()
{
  return preorder_iterator(typename bintree<T>::node());
}

/*_____________________________________________________________ */

/*
  Iterador para recorrido en Inorder
*/


template <typename T>
inline
bintree<T>::inorder_iterator::inorder_iterator()
{
}


template <typename T>
inline bintree<T>::inorder_iterator::inorder_iterator(
  bintree<T>::node n)
  : elnodo(n)
{
}

template <typename T>
inline bool bintree<T>::inorder_iterator::operator!=(
const typename bintree<T>::inorder_iterator & i) const
{
  return elnodo != i.elnodo;
}


template <typename T>
inline bool bintree<T>::inorder_iterator::operator==(
const typename bintree<T>::inorder_iterator & i) const
{
  return elnodo == i.elnodo;
}


template <typename T>
inline
typename bintree<T>::inorder_iterator &
bintree<T>::inorder_iterator::operator=(
```

```
    const bintree<T>::inorder_iterator & i)
{
  elnodo = i.elnodo;
  return *this;
}


template <typename T>
inline T & bintree<T>::inorder_iterator::operator*()
{
  return *elnodo;
}


template <typename T>
typename bintree<T>::inorder_iterator &
bintree<T>::inorder_iterator::operator++()
{
  if (elnodo.null())
    return *this;

  if (!elnodo.right().null()) {
      elnodo = elnodo.right();
      while (!elnodo.left().null())
        elnodo = elnodo.left();
  }
  else {
      while (!elnodo.parent().null() &&
                 elnodo.parent().right() == elnodo)
        elnodo = elnodo.parent();
      // Si (padre de elnodo es nodo_nulo), hemos terminado
      // En caso contrario, el siguiente node es el padre
      elnodo = elnodo.parent();
    }
  return *this;
}

template <typename T>
typename bintree<T>::inorder_iterator
bintree<T>::begin_inorder()
{
  node n(laraiz);

  if (!n.null())
    while (!n.left().null())
      n = n.left();
  return inorder_iterator(n);
}


template <typename T>
inline
typename bintree<T>::inorder_iterator
bintree<T>::end_inorder()
{
  return inorder_iterator(typename bintree<T>::node());
}

/*_____________________________________________________________ */

/*
  Iterador para recorrido en Postorder
```



```
*/


template <typename T>
inline
bintree<T>::postorder_iterator::postorder_iterator()
{
}

template <typename T>
inline
bintree<T>::postorder_iterator::postorder_iterator(
  bintree<T>::node n)
  : elnodo(n)
{
}

template <typename T>
inline
bool bintree<T>::postorder_iterator::operator!=(
const bintree<T>::postorder_iterator & i) const
{
  return elnodo != i.elnodo;
}

template <typename T>
inline
bool bintree<T>::postorder_iterator::operator==(
const bintree<T>::postorder_iterator & i) const
{
  return elnodo == i.elnodo;
}


template <typename T>
inline
typename bintree<T>::postorder_iterator &
bintree<T>::postorder_iterator::operator=(
    const bintree<T>::postorder_iterator & i)
{
  elnodo = i.elnodo;
  return *this;
}


template <typename T>
inline
T & bintree<T>::postorder_iterator::operator*()
{
  return *elnodo;
}


template <typename T>
typename bintree<T>::postorder_iterator &
bintree<T>::postorder_iterator::operator++()
{
  if (elnodo.null())
    return *this;

  if (elnodo.parent().null())
    elnodo = typename bintree<T>::node();
```

```
  else
    if (elnodo.parent().left() == elnodo) {
      if (!elnodo.parent().right().null()) {
        elnodo = elnodo.parent().right();
        do {
                while (!elnodo.left().null())
            elnodo = elnodo.left();
          if (!elnodo.right().null())
            elnodo = elnodo.right();
        } while ( !elnodo.left().null() ||
                  !elnodo.right().null());
      }
      else
        elnodo = elnodo.parent();
    }
    else // elnodo es hijo a la derecha de su padre
      elnodo = elnodo.parent();

  return *this;
}

template <typename T>
inline
typename bintree<T>::postorder_iterator bintree<T>::begin_postorder()
{
  node n(laraiz);

  do {
    while (!n.left().null())
      n = n.left();
    if (!n.right().null())
      n = n.right();
  } while (!n.left().null() || !n.right().null());

  return postorder_iterator(n);
}

template <typename T>
inline
typename bintree<T>::postorder_iterator
bintree<T>::end_postorder()
{
  return postorder_iterator(typename bintree<T>::node());
}

/*_____________________________________________________________ */

/*
  Iterador para recorrido por Niveles
*/

template <typename T>
inline
bintree<T>::level_iterator::level_iterator()
{
}

template <typename T>
inline bintree<T>::level_iterator::level_iterator(
  bintree<T>::node n)
{
  cola_Nodos.push(n);
```



```
}
template <typename T>
inline bool bintree<T>::level_iterator::operator!=(
const bintree<T>::level_iterator & i) const
{
  if (cola_Nodos.empty() && i.cola_Nodos.empty())
    return false;
  if (cola_Nodos.empty() || i.cola_Nodos.empty())
    return true;
  if (cola_Nodos.front() != i.cola_Nodos.front())
    return false;
  if (cola_Nodos.size() != i.cola_Nodos.size())
    return false;
  return (cola_Nodos == i.cola_Nodos);
}

template <typename T>
inline
bool bintree<T>::level_iterator::operator==(
const bintree<T>::level_iterator & i) const
{
  return  !(*this != i);
}

template <typename T>
inline
typename bintree<T>::level_iterator &
bintree<T>::level_iterator::operator=(
    const bintree<T>::level_iterator & i)
{
  cola_Nodos = i.cola_Nodos;
  return *this;
}


template <typename T>
inline
T & bintree<T>::level_iterator::operator*()
{
  return *cola_Nodos.front();
}


template <typename T>
typename bintree<T>::level_iterator &
bintree<T>::level_iterator::operator++()
{
  if (!cola_Nodos.empty()) {
    typename bintree<T>::node n = cola_Nodos.front();
    cola_Nodos.pop();
    if (!n.left().null())
      cola_Nodos.push(n.left());
    if (!n.right().null())
      cola_Nodos.push(n.right());
  }

  return *this;
}


template <typename T>
```

```
inline typename bintree<T>::level_iterator
bintree<T>::begin_level()
{
  if (!root().null())
    return level_iterator(laraiz);
  else
    return level_iterator();
}

template <typename T>
inline typename bintree<T>::level_iterator
bintree<T>::end_level()
{
  return level_iterator();
}

/*
  ************************************************************
  Iteradores constantes
  ************************************************************
  */


/*
  Iterador cosntante para recorrido en Preorder
*/

template <typename T>
inline
bintree<T>::const_preorder_iterator::const_preorder_iterator()
{
}


template <typename T>
inline
bintree<T>::const_preorder_iterator::const_preorder_iterator(
    bintree<T>::node n)
  : elnodo(n)
{
}

template <typename T>
bintree<T>::const_preorder_iterator::const_preorder_iterator(const typename bint
ree<T>::preorder_iterator & i) {
  elnodo = i.elnodo;
}

template <typename T>
inline
bool bintree<T>::const_preorder_iterator::operator!=(
     const bintree<T>::const_preorder_iterator & i) const
{
  return elnodo != i.elnodo;
}


template <typename T>
inline
bool bintree<T>::const_preorder_iterator::operator==(
   const bintree<T>::const_preorder_iterator & i) const
{
```



```
  return elnodo == i.elnodo;
}


template <typename T>
inline
typename bintree<T>::const_preorder_iterator &
bintree<T>::const_preorder_iterator::operator=(
    const bintree<T>::const_preorder_iterator & i)
{
  elnodo = i.elnodo;
  return *this;
}


template <typename T>
inline
const T & bintree<T>::const_preorder_iterator::operator*() const
{
  return *elnodo;
}


template <typename T>
typename bintree<T>::const_preorder_iterator &
bintree<T>::const_preorder_iterator::operator++()
{
  if (elnodo.null())
    return *this;

  if (!elnodo.left().null())
    elnodo = elnodo.left();
  else if (!elnodo.right().null())
      elnodo = elnodo.right();
  else {
    while ((!elnodo.parent().null()) &&
           (elnodo.parent().right() == elnodo
            || elnodo.parent().right().null()))
        elnodo = elnodo.parent();
    if (elnodo.parent().null())
      elnodo = typename bintree<T>::node();
    else
      elnodo = elnodo.parent().right();
  }

  return *this;
}


template <typename T>
inline
typename bintree<T>::const_preorder_iterator
bintree<T>::begin_preorder() const
{
  return const_preorder_iterator(laraiz);
}

template <typename T>
inline
typename bintree<T>::const_preorder_iterator
bintree<T>::end_preorder() const
{
```

```
  return const_preorder_iterator(typename bintree<T>::node());
}

/*_____________________________________________________________ */

/*
  Iterador constante para recorrido en Inorder
*/


template <typename T>
inline
bintree<T>::const_inorder_iterator::const_inorder_iterator()
{
  elnodo = typename bintree<T>::node();
}

template <typename T>
inline
bintree<T>::const_inorder_iterator::const_inorder_iterator(const const_inorder_i
terator &i)
{
  elnodo = i.elnodo;
}

template <typename T>
inline
bintree<T>::const_inorder_iterator::const_inorder_iterator(
  bintree<T>::node n)
  : elnodo(n)
{
}

template <typename T>
inline
bool bintree<T>::const_inorder_iterator::operator!=(
const bintree<T>::const_inorder_iterator & i) const
{
  return elnodo != i.elnodo;
}


template <typename T>
inline
bool bintree<T>::const_inorder_iterator::operator==(
const bintree<T>::const_inorder_iterator & i) const
{
  return elnodo == i.elnodo;
}

template <typename T>
inline
typename bintree<T>::const_inorder_iterator &
bintree<T>::const_inorder_iterator::operator=(
    const bintree<T>::const_inorder_iterator & i)
{
  elnodo = i.elnodo;
  return *this;
}


template <typename T>
```



```
inline
const T & bintree<T>::const_inorder_iterator::operator*() const
{
  return *elnodo;
}


template <typename T>
typename bintree<T>::const_inorder_iterator &
bintree<T>::const_inorder_iterator::operator++()
{
  if (elnodo.null())
    return *this;

  if (!elnodo.right().null()) {
    elnodo = elnodo.right();
    while (!elnodo.left().null())
      elnodo = elnodo.left();
  }
  else {
      while (!elnodo.parent().null() &&
                    elnodo.parent().right() == elnodo)
        elnodo = elnodo.parent();
      // Si (el padre de elnodo es nodo_nulo), hemos terminado
      // En caso contrario, el siguiente Nodo es el padre
      elnodo = elnodo.parent();
    }
  return *this;
}

template <typename T>
inline
typename bintree<T>::const_inorder_iterator
bintree<T>::begin_inorder() const
{
  node n(laraiz);

  if (!n.null())
    while (!n.left().null())
      n = n.left();
  return const_inorder_iterator(n);
}


template <typename T>
inline
typename bintree<T>::const_inorder_iterator
bintree<T>::end_inorder() const
{
  return const_inorder_iterator(typename bintree<T>::node());
}

/*_____________________________________________________________ */

/*
  Iterador constante para recorrido en Postorder
*/


template <typename T>
inline
bintree<T>::const_postorder_iterator::const_postorder_iterator()
```

```
{
  elnodo = typename bintree<T>::node();
}


template <typename T>
inline
bintree<T>::const_postorder_iterator::const_postorder_iterator(
  typename bintree<T>::node n)
  : elnodo(n)
{
}

template <typename T>
inline
bool bintree<T>::const_postorder_iterator::operator!=(
const bintree<T>::const_postorder_iterator & i) const
{
  return elnodo != i.elnodo;
}


template <typename T>
inline
bool bintree<T>::const_postorder_iterator::operator==(
const bintree<T>::const_postorder_iterator & i) const
{
  return elnodo == i.elnodo;
}


template <typename T>
inline
typename bintree<T>::const_postorder_iterator &
bintree<T>::const_postorder_iterator::operator=(
    const bintree<T>::const_postorder_iterator & i)
{
  elnodo = i.elnodo;
  return *this;
}


template <typename T>
inline
const T & bintree<T>::const_postorder_iterator::operator*() const
{
  return *elnodo;
}


template <typename T>
typename bintree<T>::const_postorder_iterator &
bintree<T>::const_postorder_iterator::operator++()
{
  if (elnodo.null())
    return *this;

  if (elnodo.parent().null())
    elnodo = typename bintree<T>::node();
  else
    if (elnodo.parent().left() == elnodo) {
      if (!elnodo.parent().right().null()) {
```



```
        elnodo = elnodo.parent().right();
        do {
                while (!elnodo.left().null())
                  elnodo = elnodo.left();
                if (!elnodo.right().null())
                  elnodo = elnodo.right();
              } while ( !elnodo.left().null() ||
                              !elnodo.right().null());
      }
      else
        elnodo = elnodo.parent();
   }
  else // elnodo es hijo a la derecha de su padre
      elnodo = elnodo.parent();

  return *this;
}


template <typename T>
inline
typename bintree<T>::const_postorder_iterator
bintree<T>::begin_postorder() const
{
  node n = root();

  do {
    while (!n.left().null())
      n = n.left();
    if (!n.right().null())
      n = n.right();
  } while (!n.left().null() || !n.right().null());

  return const_postorder_iterator(n);
}

template <typename T>
inline
typename bintree<T>::const_postorder_iterator
bintree<T>::end_postorder() const
{
  return const_postorder_iterator(typename bintree<T>::node());
}

/*_____________________________________________________________ */

/*
  Iterador cosntante para recorrido por Niveles
*/


template <typename T>
inline
bintree<T>::const_level_iterator::const_level_iterator()
{
}


template <typename T>
inline
bintree<T>::const_level_iterator::const_level_iterator(
  bintree<T>::node n)
```

```
{
  cola_Nodos.push(n);
}


template <typename T>
inline
bool bintree<T>::const_level_iterator::operator!=(
const bintree<T>::const_level_iterator & i) const
{
  if (cola_Nodos.empty() && i.cola_Nodos.empty())
    return false;
  if (cola_Nodos.empty() || i.cola_Nodos.empty())
    return true;
  return cola_Nodos.front() != i.cola_Nodos.front();
}


template <typename T>
inline
bool bintree<T>::const_level_iterator::operator==(
const bintree<T>::const_level_iterator & i) const
{
  return !(*this != i);
}


template <typename T>
inline
typename bintree<T>::const_level_iterator &
bintree<T>::const_level_iterator::operator=(
    const bintree<T>::const_level_iterator & i)
{
  cola_Nodos = i.cola_Nodos;
  return *this;
}


template <typename T>
inline const T & bintree<T>::const_level_iterator::operator*() const
{
  return *cola_Nodos.front();
}


template <typename T>
typename bintree<T>::const_level_iterator &
bintree<T>::const_level_iterator::operator++()
{
  if (!cola_Nodos.empty()) {
    typename bintree<T>::node n = cola_Nodos.front();
    cola_Nodos.pop();
    if (!n.left().null())
      cola_Nodos.push(n.left());
    if (!n.right().null())
      cola_Nodos.push(n.right());
  }

  return *this;
}
```



```
template <typename T>
inline
typename bintree<T>::const_level_iterator
bintree<T>::begin_level() const
{
  if (!root().null())
    return const_level_iterator(laraiz);
  else
    return const_level_iterator();
}

template <typename T>
inline
typename bintree<T>::const_level_iterator
bintree<T>::end_level() const
{
  return const_level_iterator();
}

template <typename T>
void bintree<T>::replace_subtree(typename bintree<T>::node pos, const bintree<T>
& a, typename bintree<T>::node n)
{
   if (&a != this) {
     if (pos == laraiz) {  // pos es la raiz
       destroy(laraiz);
       copy(laraiz, n);
       if (!laraiz.null())
         laraiz.parent(typename bintree<T>::node());
     }
     else {  // Pos no esta en la raiz
       typename bintree<T>::node padre = pos.parent(), aux;
       if (padre.left()==pos) {
         destroy(padre.left());
         copy(aux, n);
         padre.left(aux);
       }
       else {
         destroy(padre.right());
         copy(aux, n);
         padre.right(aux);
       }
     }
   }
}
```