bintree.h

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

    bintree( );
    bintree (const T & e);
    bintree (const bintree<T> & a);
   ~bintree ( );
    bintree <T> & operator = (const bintree<T>
                              & a);

    void assign_subtree (const bintree<T> & a,
                         node n);

    node root ( ) const;

    void prune_left (node n, bintree<T> & dest);
    void prune_right (node n, bintree<T> & dest);

    void insert_left (const bintree<T>:: node & n,
                      const T & e);

    void insert_right (node n, const T & e);

    void insert_left (node n, bintree<T> & rama);
    void insert_right (node n, bintree<T> & rama);
```

```
void clear( );
size_type size( );
bool empty( ) const;
bool operator==(const binTree<T> &a) const;
bool operator!=(const binTree<T> &a) const;
void replace_subtree(node pos, const binTree<T>
                                &a, node n);
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 *lnode;
      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_itaator(const preorder_itaator &i);
  bool operator!=(const const_preorder_itaator &i)
                  const;
  bool operator==(const const_preorder_itaator &i)
                  const;
  const_preorder_itaator & operator=(const
                const_preorder_itaator &i);
  const T & operator*() const;
  const_preorder_itaator & operator++();

private:
  node *node;
  const_preorder_itaator(node n);
  friend class bintree<T>;
};
const_preorder_iterator begin_preorder() const;
const_preorder_itaator end_preorder() const;
```

```
class inorder_iterator
{
...
};
inorder_iterator begin_inorder();
inorder_iterator end_inorder();

class const_inorder_iterator
{
...
};
const_inorder_iterator begin_inorder();
const_inorder_iterator end_inorder();

class postorder_iterator
{
...
};
postorder_iterator begin_postorder();
postorder_iterator endpostorder();

class const_postorder_iterator
{
...
};
const_postorder_iterator begin_postorder() const;
const_postorder_iterator end_postorder() const;
```

```
class level_iterator
{
  ...
};
level_iterator begin_level();
level_iterator end_level();

class const_level_iterator
{
  ...
};
const_level_iterator begin_level() const;
const_level_iterator end_level() const;
```

private:

```
void destroy(binutree<T>::node n)
void copy(node& dest, const node& orig);
int count(node n) const;
bool equals(node n1, node n2) const;
node laraiz;
size_type num_nodos;
class nodewrapper{
 public:
   T etiqueta;
   node pad;
```

```
    node izda;
    node drcha;
    nodewrapper();
    nodewrapper(const T& e);
};
public:
  class node {
    public:
    node();
    node(const T& e);
    node(const node n);
    bool null() const;
    node parent() const;
    node left() const;
    node right() const;
    T& operator*();
   const T& operator*() const;
    void remove();
    node& operator=(const node& n) const;
    bool operator==(const node& n) const;
    bool operator!=(const node& n) const;
```

```
private:   //de uso exclusivo en biutree
  friend dass biutree<T>;
  inliue void parent (node n);
    // pone n como nodo padre del receptor
  inliue void left (node n);
  // pone n como hijo izquierdo del receptor
  inliue void right (node n);
  // pone n como hijo derecha del receptor
  nodeimpl * elnodo;
};
};
# include "biutree.hxx"
# include "node.hxx"
```

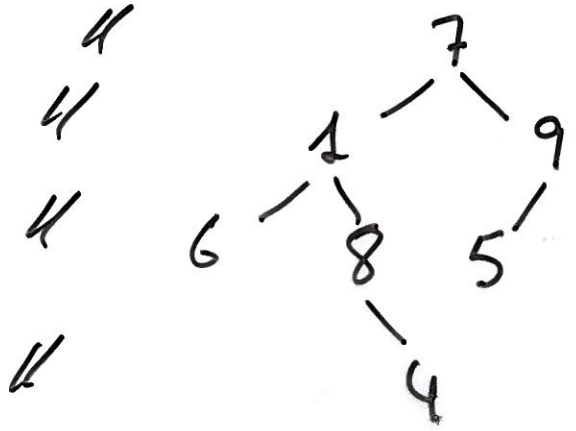
usoarbolbinario.cpp

```
#include <iostream>
#include "bintree.h"
#include <queue>
using namespace std;

template <class T>
bool esHoja (const bintree<T> & A, const
             typename bintree<T>::node &V)

  { return (v.left().null() &&
            v.right().null() );

  }

bool esInterno(const bintree<T> & A, const
             typename bintree<T>::node &v)

  {
   return ( !v.left().null() ||
            !v.right().null() );

  }
```

```
template <class T>
void PreordenBinario (const bintree<T> &A,
        typename bintree<T>::node v)
  {
    if (!v.null())
        {
         cout << *v;
        PreordenBinario (A, v.left());
        Preorden Binario(A, v.right());
        }
  }

template <class T>
void ListarPorNiveles (const bintree <T> &A,
            typename bintree <T>:: node n)
  {
   queue <typename bintree <T>:: node > nodos;
   if (!n.null()) {
       nodos.push(n);
      while (!nodos.empty()) {
         n = nodos.front();
         nodos.pop();
         cout << * n;
       if (!n.left().null()) nodos.push(
                                n.left());
       if (!n.right().null())
                        nodos.push(n.right());
   }
  }
```

```
template <class T>
ostream & operator << (ostream & os,
                       binTree<T> & arb)
{
  cout << "Preorden";
  for (typename binTree<T>::preorder_iterator
         i = arb.begin_preorder();
       i != arb.end_preorder(); ++i)
    cout << *i << " ";
  cout << endl;
}
```

```
int main()
{
  // creamos el árbol
  //
  //
  //
  //
```




```
  typedef binutree<int> bti;
  binutree<int> Arb(7)
  Arb.insert_left(Arb.root(), 1);
  Arb.insert_right(Arb.root(), 9);
  Arb.insert_left(Arb.root().left(), 6);
  Arb.insert_right(Arb.root().left(), 8);
  Arb.insert_right(Arb.root().left().right(), 4);
  Arb.insert_left(Arb.root().right(), 5);

 cout << "Preorden";
for(binutree<int>::preorder_iterator i =
        Arb.begin_preorder();
    i != Arb.end_preorder(); ++i)
        cout << *i << " ";

  cout << endl;
```

```
cout << Inorden";
for (bintree<int>::inorder_iterator i =
     Arb.begin_inorder(); i != Arb.end_inorder()
     ++i)
        cout << *i << " ";
cout << endl;

cout << "Postorden";
for (bintree<int>::postorder_iterator i =
     Arb.begin_postorder(); i != Arb.end_postorder()
          ++i)
     cout << *i << " ";
cout << endl;

cout << "Por Niveles";
for (bintree<int>::level_iterator i =
     Arb.begin_level(); i != Arb.end_level() ++i)
          cout << *i << " ";
cout << endl;
cout << Arb;
}
```

biutree.hxx

```
template <typename T>
void biutree <T>::destroy (typename
                           biutree <T>::node n)
{
   if (!n.null ())
     {
      destroy (n.left());
      destroy (n.right());
       n.remove();
     }
}
template <typename T>
int biutree <T>:: count (biutree <T>::node n) cons
    {
     if (n.null())
       return 0;
     else
      return 1 + count (n.left()) +
                 + cout (n.right());
    }
```

```
template <typename T>
 inline
void bintree<T>::prune_left(
                typename bintree<T>::node n,
                bintree<T> & orig)

{
   assert(!n.null());
   destroy(orig.lavaiz);
   num_nodos = num_nodos - count(n.left());
   orig.lavaiz = n.left();
   if (!orig.lavaiz.null())
     orig.lavaiz.parent(typename bintree<T>
                        ::node());
     n.left(typename bintree<T>::node());
}
```

```
template <typename T>
inline
void biutree<T>::insert_left(const
    typename biutree<T>::node & n,
                          const T & e)

  {
    biutree<T> aux(e);
    insert_left(n, aux);
  }
template <typename T>
inline
void biutree<T>::insert_left(typename
       biutree<T>::node n, biutree<T>& rama)

 {
   assert(!n.null());
  num_nodos = num_nodos - count(n.left())
                        + rama.num_nodos;

  typename biutree<T>:: node aux(n.left());
  destroy(aux);
  n.left(rama.laraiz);
  if (!n.left.null()) n.left().parent(n);
  rama.laraiz = typename biutree<T>::node();

 }
```

```
//iterator preorder

template <typename T>
inline
binTree<T>::preorder_iterator::preorder_iterator()
{
    elnode = typename binTree<T>::node();
}

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

template <typename T>
inline
binTree<T>::preorder_iterator::preorder_iterator(
    binTree<T>::node n): elnode(n) {}

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

```
template <typename T>
inline
bool binTree<T>::preorder_iterator::operator==(
    const binTree<T>::preorder_iterator& i) const
{
  return elnode == i.elnode;
}
template <typename T>
inline
typename binTree<T>::preorder_iterator&
binTree<T>::preorder_iterator::operator=(
          const binTree<T>::preorder_iterator& i)
{
  elnode = i.elnode;
  return *this;
}
template <typename T>
inline
T& binTree<T>::preorder_iterator::operator*()
{
  return *elnode;
}
```

```
template <typename T>
 inline
 typename binTree <T>::preorder_iterator
          binTree <T>:: begin_preorder()
    {
      return preorder_iterator (last);
    }
template <typename T>
 inline
 typename binTree <T>::preorder_iterator
   binTree <T>:: end_preorder()
    {
    return preorder_iterator (typename
                binTree <T>:: node ());
    }
```

```
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;
}
```

node.hxx

```
template <typename T>
inline
  bool biutree <T>::node::null() const
    {
      return elnode==0;
    };
template <typename T>
 inline
  bool biutree <T>::parent(typename biutree<T>::node n)
    {
      elnode->pad=n
    }
  template <typename T>
  inline
   typename biutree <T>::node biutree <T>::node
                          ::parent() const
    {
      return (elnode->pad);
    }
```