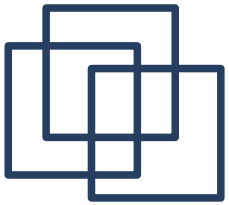
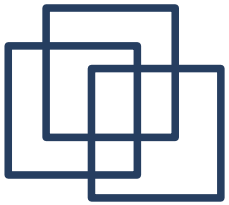
Contenedores Avanzados: Árboles

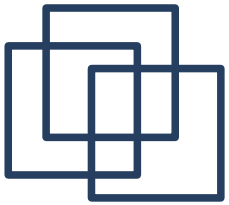


bintree

- **bintree** es un contenedor que almacena objetos del tipo T.
- Define una jerarquía entre los elementos del tipo.
  - los elementos están por encima (antecesores) y por debajo (descendientes).
- Representa un árbol binario (un nodo tiene a lo sumo dos hijos (left y right)
- Al insertar/borrar elementos no se invalidan las posiciones del bintree

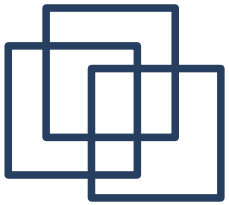


- Especificación de bintree<T> que representa el concepto abstracto árbol binario
  - bintree<int> un_arbol
- Especificación del bintree<T>::node que representa el concepto abstracto nodo de un árbol binario
  - bintree<int>::node unnodo



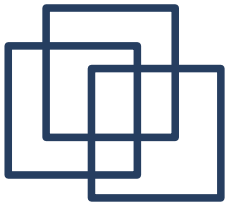
Ejemplos ... (IV)

```
int altura(const bintree<T> & A, bintree<T>::node
   v)
{    if ( v.null() ) return -1;
     else {
        int h, alt_izq, alt_dch;
        alt_izq = altura(A,v.left() );
        alt_dch = altura(A,v.right() );
        h = (alt_izq>alt_dch)? alt_izq:alt_dch;
        return 1+h;
        }
    }
```



Ejemplos V

```
template <typename T>
pair<typename bintree<T>::node,bool>
  find( const typename bintree<T>::node &  n,
  const T & x ) //
{
 pair<typename bintree<T>::node,bool> salida;
 salida.second = false;
 if (!n.null()) {
   if(*n==x){ salida.first = n;salida.second =
   true;}  else {
            salida = find(n.left(),x);
            if (salida.second != true)
                  salida = find(n.right(),x);
           }
 return salida;
}
```



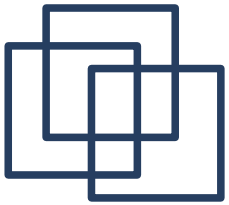
bintree<T> Representacion

Representación Matricial:

Se guardan los elementos en un vector.

Representación por Celdas enlazadas:

Los elementos se guardan en celdas independientes



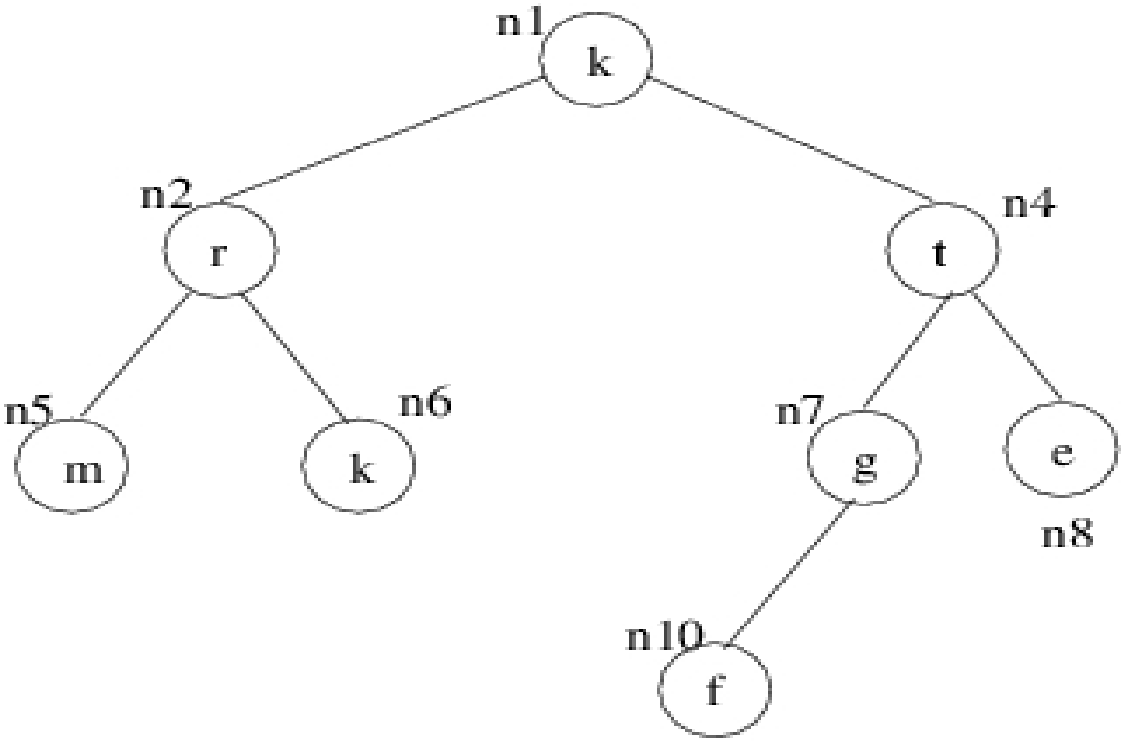
bintree<T> Representación Matricial

Se guardan los elementos en un vector.

Un nodo se representa por una posición del vector.

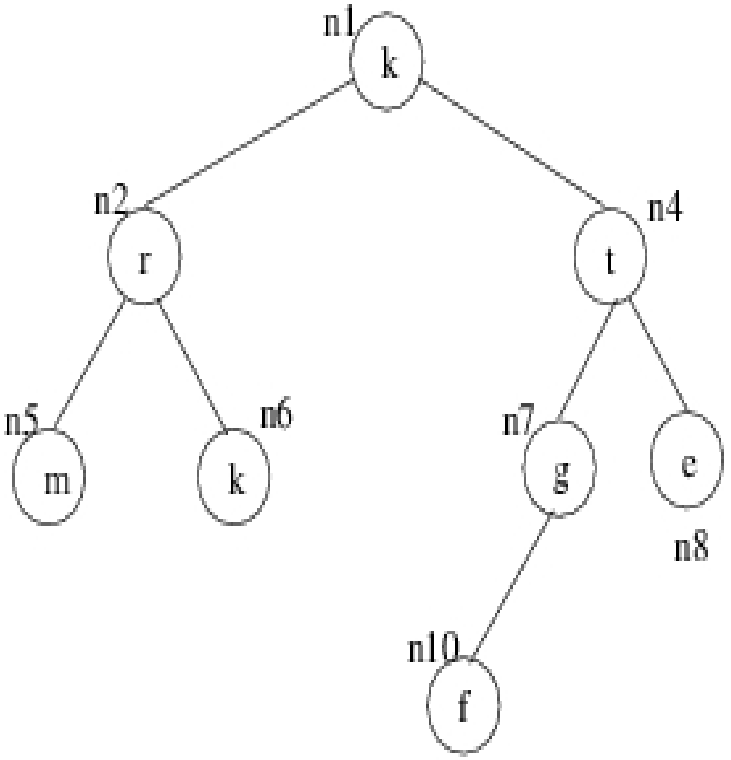
- El nodo raíz es el nodo=0
- node::parent() es (nodo - 1)/2
- node::left() es 2 * nodo + 1
- node::right() es 2 * nodo + 2

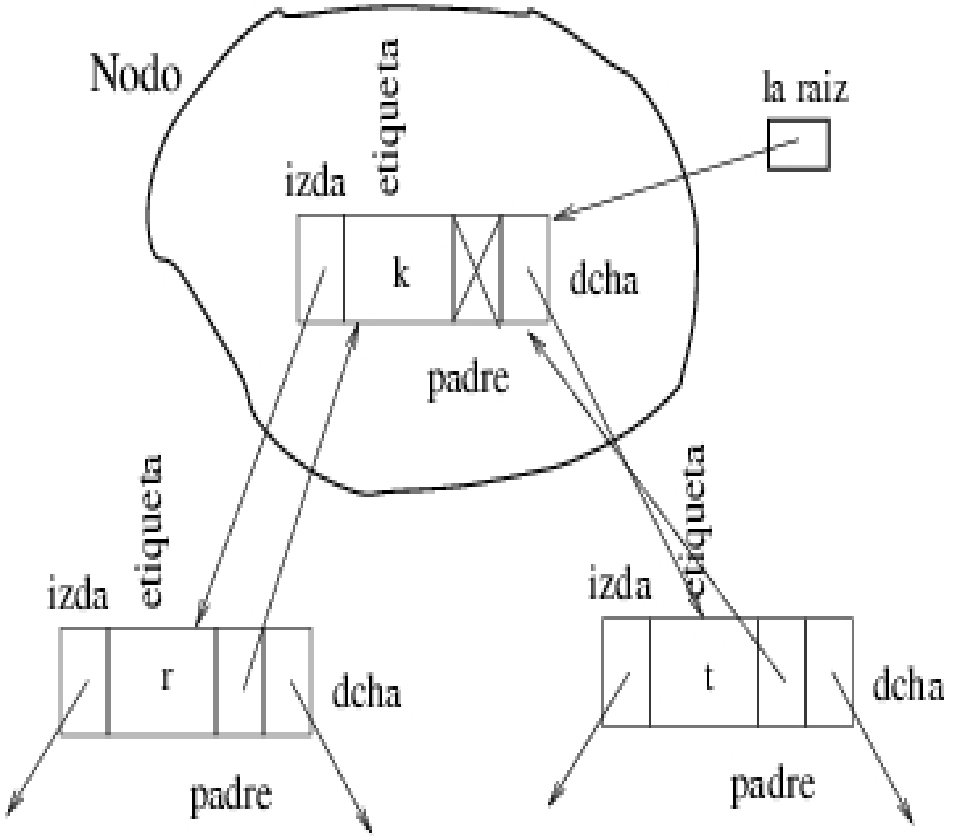
Representación matricial

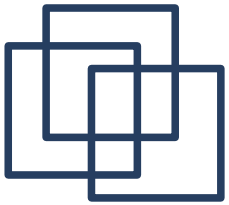


	0	1	2	3	4	5	6	7
rep	k	r	t	m	k	g	e	f

Representación con celdas enlazadas







Representación con celdas enlazadas

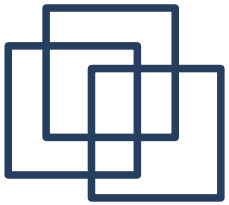
- Clase node

Función de Abstracción:

Dado el objeto del tipo rep r, r = {elnodo} y representa un nodo etiquetado con “etiqueta” y con hijos “izda” y “dcha” y padre “padre”.

Invariante de la representación:

Si “padre”, “izda” y “dcha” no son nulos, entonces son todos distintos.



Representación celdas enlazadas

Función de Abstracción:

Dado el objeto del tipo rep r, r = {laraiz}, el objeto abstracto al que representa es:

a) Arbol nulo, si r.laraiz.null().

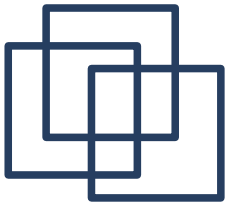
b) Arbol con un único nodo: Nodo de etiqueta *r.laraiz, si r.laraiz.left().null() y r.laraiz.right().null()

c)

*r.laraiz

/ \

Arbol(r.laraiz.left()) Arbol(r.laraiz.right())

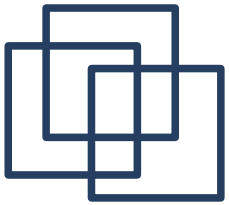


Representación por celdas enlazadas

Invariante de Representación:

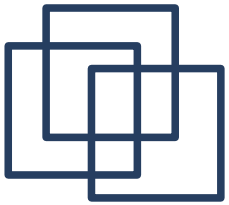
Si !r.laraiz.null() entonces

r.laraiz.parent().null().



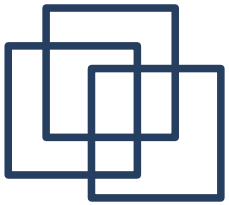
Esquema general

```
class bintree{
    public:
        bintree();
        bintree( const T & e);
        .......
        class node { .... };
        class preorder_iterator {  .... };
        class postorder_iterator { .... };
        class inorder_iterator { .... };
```



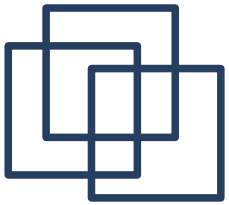
Esquema general

```
class bintree{
    public:
        .......
    private:
        struct un_nodo { ... };
        typedef un_nodo * ptrnode;
        ptrnode laraiz;
};
```



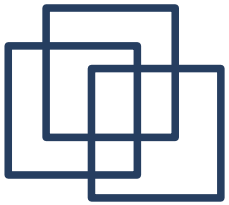
Estructura del nodo

```
struct un_nodo {
  un_nodo()
     {parent= left= right= 0;};
  un_nodo(const T & e)
     {etiqueta = e;
      parent= left = right = 0;}
  T etiqueta;
  un_nodo *parent;
  un_nodo *left; //hijo izquierda
  un_nodo *right; //hijo derecha
};
```



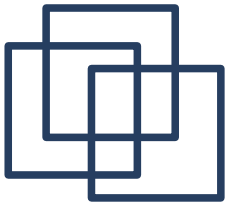
Clase bintree<T>::node

```
class node {
 public:
   node();
   node & operator=(const node & n);
   T& operator*();
   node parent() const;
   node left() const;
   node right() const;
   bool null() const;
   bool operator==(const node & i) const;
   ....
```



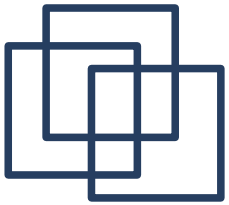
Clase tree<T>::node

```
class node {
 public:
  ......
 private:
   friend class bintree;

   node( ptrnode  n,  bintree  *arb);
   ptrnode elnodo;
   tree *elarbol;
};
```



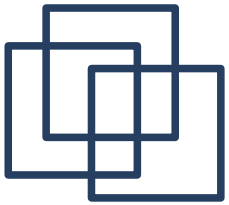
bintree<T>::inorder_iterator

```
class inorder_iterator {
 public:
   inorder_iterator();
   inorder_iterator & operator=
                (const inorder_iterator & it);
   T& operator*();
   inorder_iterator & operator++( );
   bool operator==(const node & i) const;
   ....
```



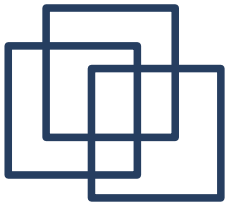
bintree<T>::inorder_iterator

```
class inorder_iterator{
 public:
  ......
 private:
   friend class bintree;
   node( ptrnode  n,  bintree  *arb);
   ptrnode elnodo;
   tree *elarbol;
};
```



Implementaciones

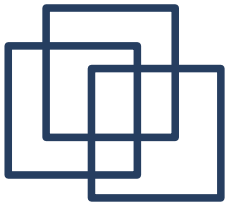
- bintree<T>
- bintree<T>::node
- bintree<T>::preorder_iterator
- bintree<T>::inorder_iterator
- bintree<T>::postorder_iterator



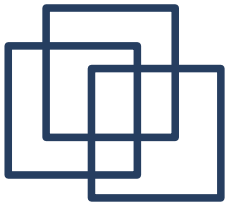
bintree<T>

Constructores

```
template <typename T>
inline bintree<T>::bintree()
{ laraiz =  0; }

template <typename T>
bintree<T>::bintree(const T& e)
{ laraiz = new un_nodo(e);  }
```

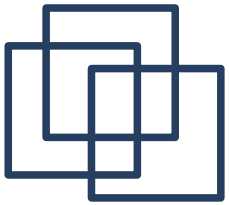


```
template <typename T>
typename bintree<T>::size_type
bintree<T>::size() const
{  return count(laraiz); }

template <typename T>
typename bintree<T>::size_type
bintree<T>::count(bintree<T>::ptrnode n) const
{  if (n==0)  return 0;
  else return 1 + count(n->left) + count(n->right);
}
```



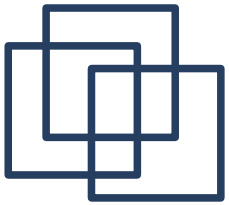
beginInorder

```
template <typename T>
typename bintree<T>::inorder_iterator
bintree<T>::beginInorder()
{
  ptrnode n = laraiz;
  if (n!=0)
    while (n->left!=0)
      n = n->left;
  return typename
            bintree<T>::inorder_iterator(n, this);
}
```



destructor

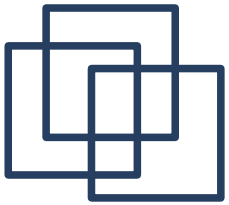
```
template <...> bintree<T>::~bintree()
{   destroy(laraiz);}

template < ...> void
  bintree<T>::destroy(typename
  bintree<T>::ptrnode n)
{    if (n!=0) {
        destroy(n->left);
        destroy(n->right);
        delete n;
        n=0;   }
}
```



insert_left

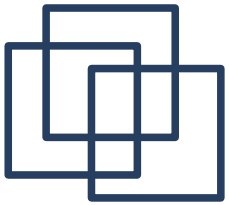
Inserta una rama como hija izquierda del nodo n, si n ya tiene un hijo este se destruye.
Post: rama pasa a ser un arbol vacío.

```
template <typename T>
typename bintree<T>::node
bintree<T>::insert_left(
                typename bintree<T>::node  n,
                bintree<T>& rama)
{ ..... }
```



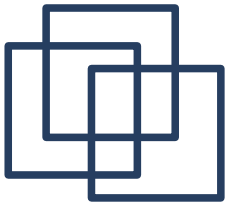
insert_left(n, rama)

```
{  assert(n.elnodo!=0);

  destroy(n.elnodo->left);
  n.elnodo->left=rama.laraiz;
  if (n.elnodo->left!=0) {
    n.elnodo->left->parent = n.elnodo;
  }
  rama.laraiz = 0; //Hacemos rama el arb. vacio
  return typename
          bintree<T>::node(n.elnodo->left,this);
}
```



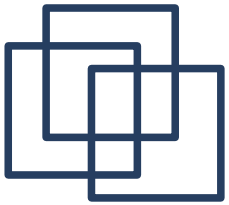
Operador asignación

```
Template < ... >  bintree<T> &
bintree<T>::operator=(const bintree<T>& a)
{
  if (&a != this) {
    destroy(laraiz);
    copy(laraiz, a.laraiz);
    if (laraiz!=0)
      laraiz->parent=0;
  }
  return *this;
}
```



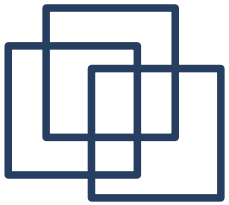
Copia de subárboles

```
Template < ... > void
   bintree<T>::copy(bintree<T>::ptrnode &dest,
                    const bintree<T>::ptrnode &orig)
{
  if (orig==0)   dest = 0;
  else {
      dest = new un_nodo(orig->etiqueta);
      copy (dest->left, orig->left);
      copy (dest->right, orig->right);
      if   (dest->left!=0)   dest->left->parent=dest;
      if (dest->right!=0) dest->right->parent=dest;
     }
}
```

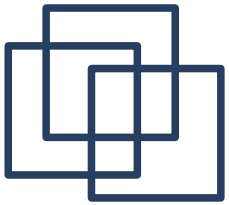


bintree<T>::node

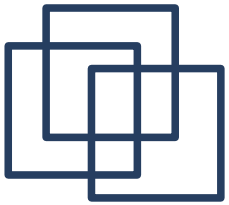
```
template <typename T>
bintree<T>::node::node( )
{ elnodo = 0 ; elarbol= 0;
}

template <typename T>
bintree<T>::node::node(
        const bintree<T>::node & it)
{ elnodo = it.elnodo;  elarbol= it.elarbol;}
```



```
template <typename T>
T & bintree<T>::node::operator*()
{  return elnodo->etiqueta; }

template <typename T>
typename  bintree<T>::node
bintree<T>::node::parent() const
{
  return typename
         bintree<T>::node(elnodo-
   >parent,elarbol);
}
```

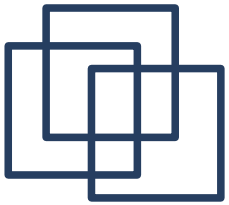


```
template <typename T>
typename  bintree<T>::node
bintree<T>::node::right() const
{  return typename
          bintree<T>::node(elnodo->right,elarbol);
}

template <typename T>
bool  bintree<T>::node::null() const
{
 return (elnodo==0);
}
```



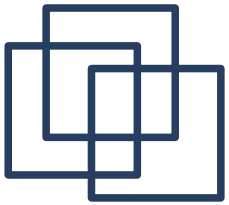
bintree<T>::preorder_iterator

```
template <typename T>
typename bintree<T>::preorder_iterator &
bintree<T>::preorder_iterator::operator++( )
{   if (elnodo==0)
       return *this;
  if (elnodo->left!=0)  elnodo = elnodo->left;
  else if (elnodo->right!=0)  elnodo = elnodo->right;
  else {
     hay que ascender por el árbol
  }

  return *this;
}
```



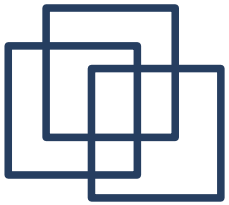
bintree<T>::preorder_iterator

```
preorder_iterator &   operator++( )
{    casos simples (transparencia anterior)

  else {  // Ascendemos
    while (  ( elnodo->parent!=0) &&
                (    elnodo->parent->right == elnodo ||
                    elnodo->parent->right==0 )   )
         elnodo = elnodo->parent;
    if (elnodo->parent==0)   elnodo = 0;
    else   elnodo = elnodo->parent->right;
  }
  return *this;
}
```



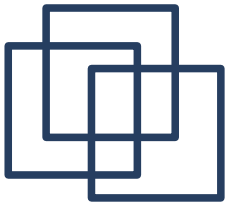
bintree<T>::inorder_iterator

```
template <typename T>
typename bintree<T>::inorder_iterator &
bintree<T>::inorder_iterator::operator++( )
{   if (elnodo==0)  return *this;
  if (elnodo->right!=0) {
      elnodo = elnodo->right;
      while (elnodo->left!=0)
            elnodo = elnodo->left;
  }  else {
        hay que ascender por el árbol
     }
  return *this;
}
```



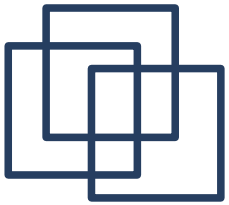
bintree<T>::inorder_iterator

```
inorder_iterator &   operator++( )
{    casos simples (transparencia anterior)

  else {  // Ascendemos
     while ( elnodo->parent!=0 &&
                  elnodo->parent->right == elnodo)
        elnodo = elnodo->parent;
         // Si (padre de elnodo es nulo), fin.  En caso
         // contrario, el siguiente ptrnode es el padre
     elnodo = elnodo->parent;
    }
  return *this;
}
```



bintree<T>::postorder_iterator

```
template <typename T>
typename bintree<T>::postorder_iterator &
bintree<T>::postorder_iterator::operator++( )
{    if (elnodo==0)  return *this;
     if (elnodo->parent==0) elnodo = 0;
    else   if (elnodo->parent->left != elnodo)
             // elnodo es hijo a la derecha de su padre
          elnodo = elnodo->parent;
   else   if (elnodo->parent->right==0)
 // elnodo es hijo a la izq de su padre, pero sin hdcha
      elnodo = elnodo->parent;
  else {  hay que ascender por el árbol }
 return *this;
}
```



bintree<T>::postorder_iterator

```
postorder_iterator &   operator++( )
{    casos simples (transparencia anterior)
  else {
       elnodo = elnodo->parent->right;
       do {
        while (elnodo->left!=0)
              elnodo = elnodo->left;
         if (elnodo->right!=0)  elnodo = elnodo->right;
       } while ( elnodo->left!=0 || elnodo->right!=0 );
     }
 return *this;
}
```