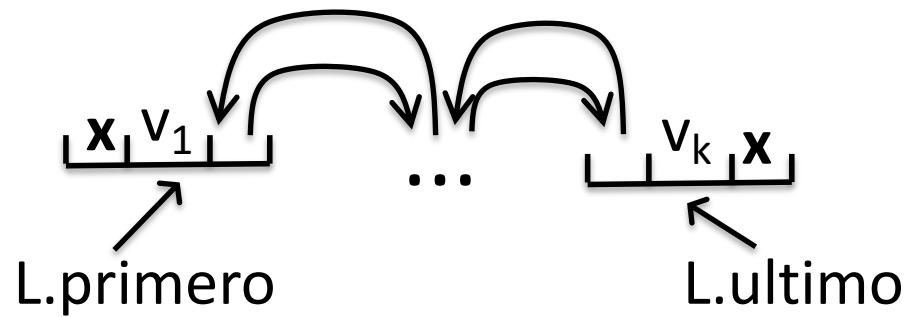
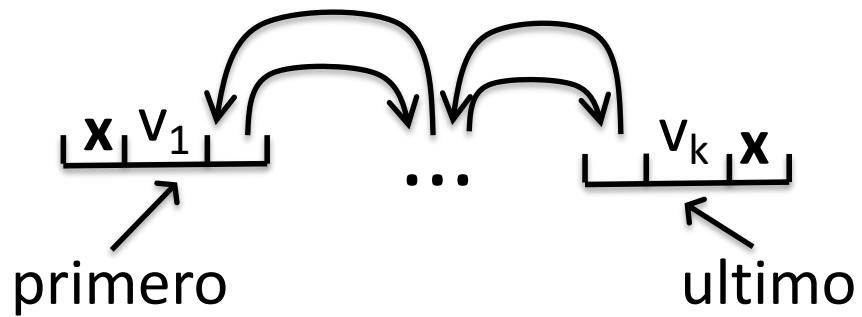
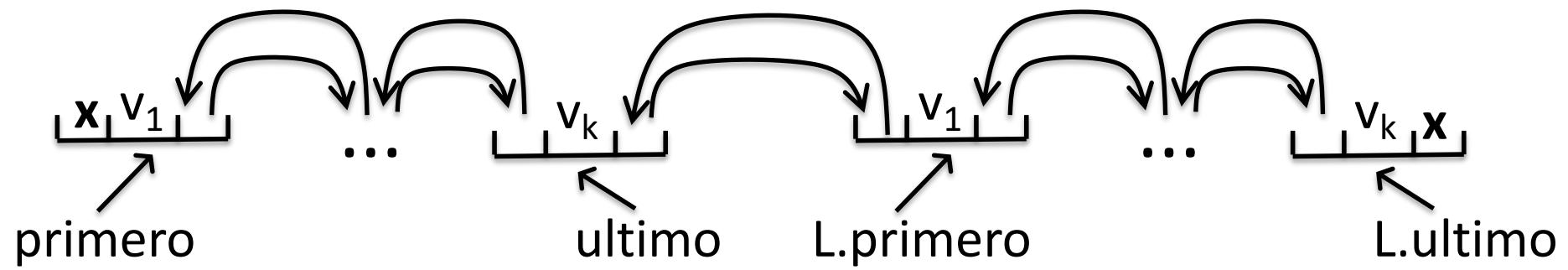
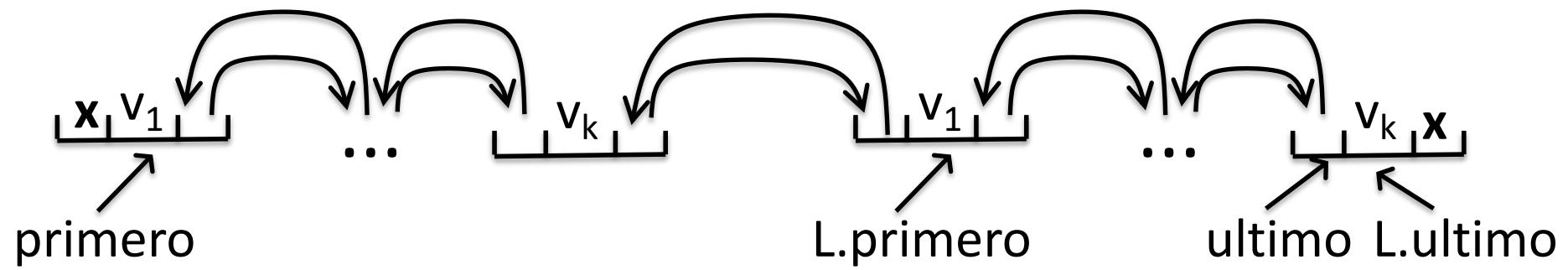
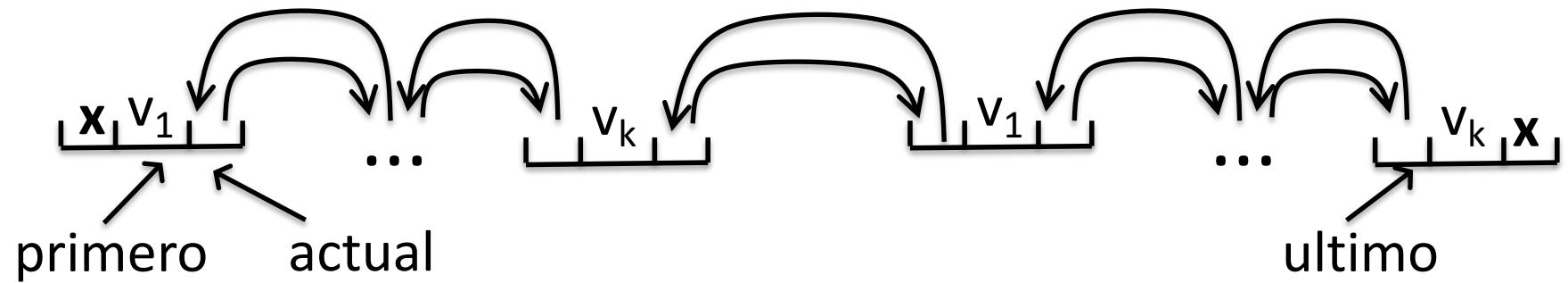


```
// Concatenación
// Pre: true
/* Post: Se han añadido al final los elementos de L, el
punto de interés es el primer elemento a, L queda vacía*/
void concat(Lista & L){
    if (L.longitud > 0) {
        if (longitud == 0) {
            primero = L.primero; ultimo = L.ultimo;
            longitud = L.longitud;
        } else {
            ultimo->sig = L.primero;
            (L.primero)->ant = ultimo; ultimo = L.ultimo;
            longitud = longitud + L.longitud;
        }
        L.primero = L.ultimo = L.act = nullptr;
        L.longitud = 0;
    }
    act = primero;
}
```





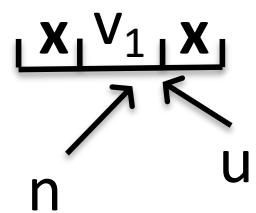
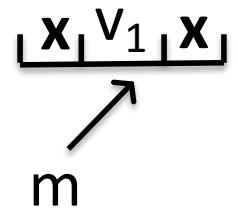




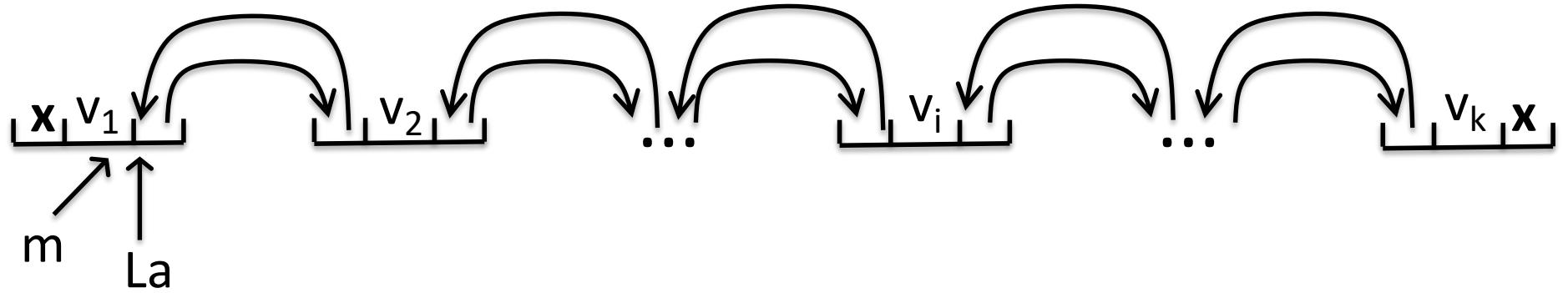
$L.\text{primero} = L.\text{ultimo} = L.\text{act} = \text{nullptr}$

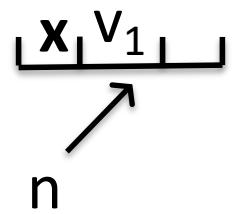
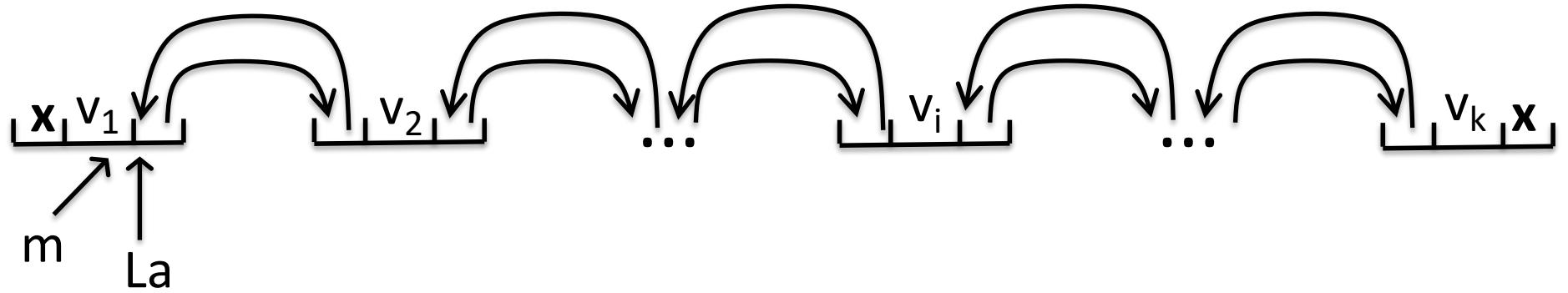
```
// Métodos privados
// Copiar secuencia de nodos
static nodo_Lista* copia_nodo_Lista (
    nodo_Lista* m, nodo_Lista* La,
    nodo_Lista* &u, nodo_Lista* &a);
// Pre: true
/* Post: si m es nullptr el resultado, u y a son nullptr,
   si no, el resultado apunta a una cadena de nodos
   que es una copia de la cadena apuntada por m,
   u apunta al último nodo y a es nullptr si La no apunta
   a ningún nodo de la secuencia, o bien a apunta al nodo
   copia del nodo apuntado por La*/
```

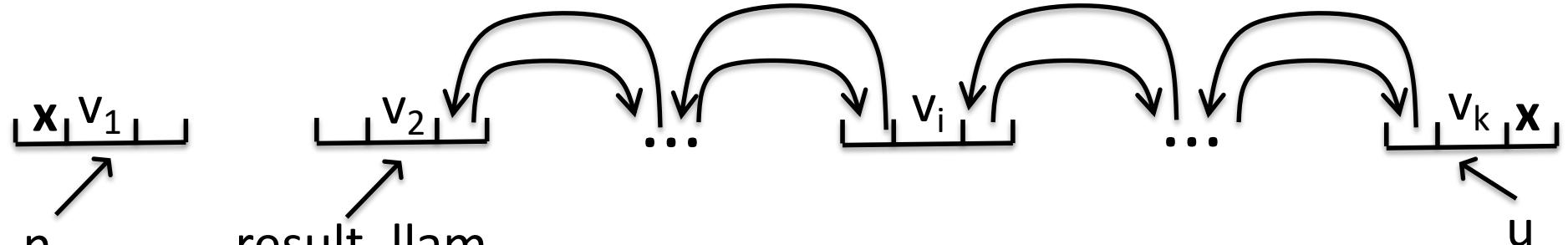
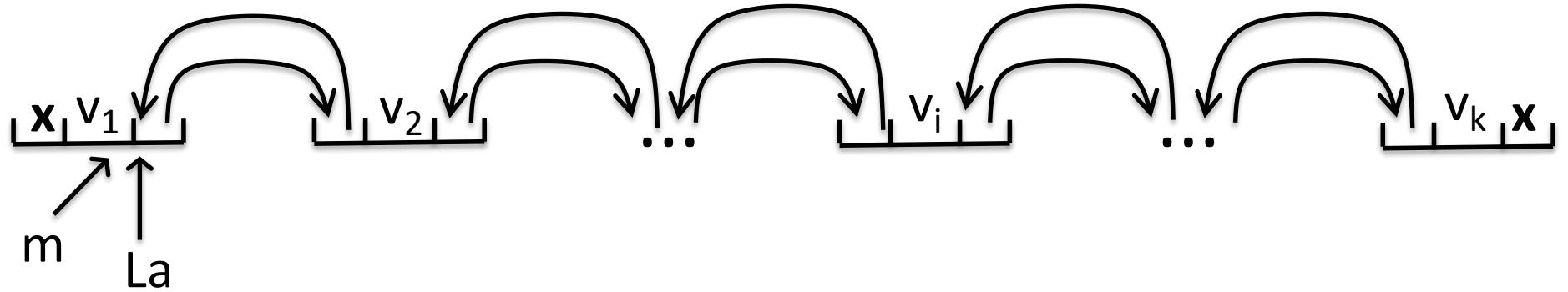
```
static nodo_Lista* copia_nodo_Lista (
    nodo_Lista* m, nodo_Lista* La,
    nodo_Lista* &u, nodo_Lista* &a){
    if (m == nullptr) {u = nullptr; a = nullptr; return nullptr;
}
else {
    nodo_lista* n = new nodo_lista;
    n->info = m->info;
    n->sig = copia_nodo_Lista(m->sig, La, u, a);
    if (n->sig != nullptr) (n->sig)->ant = n;
    else u = n;
    if (m == La) a = n;
    return n;
}
}
```



a = nullptr

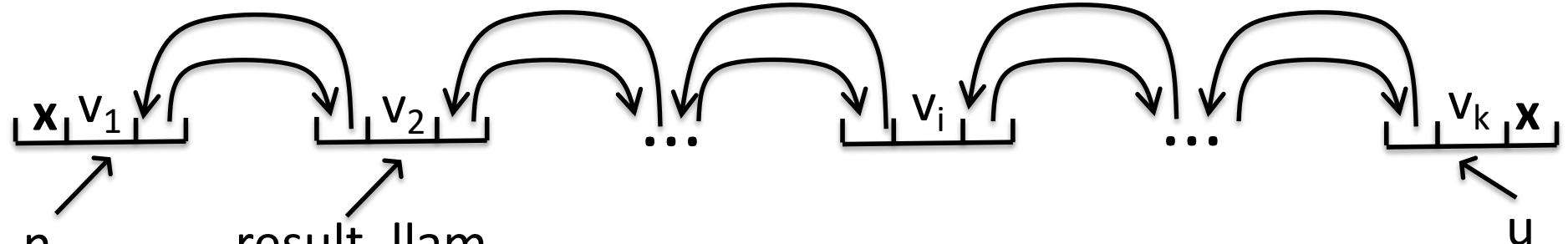
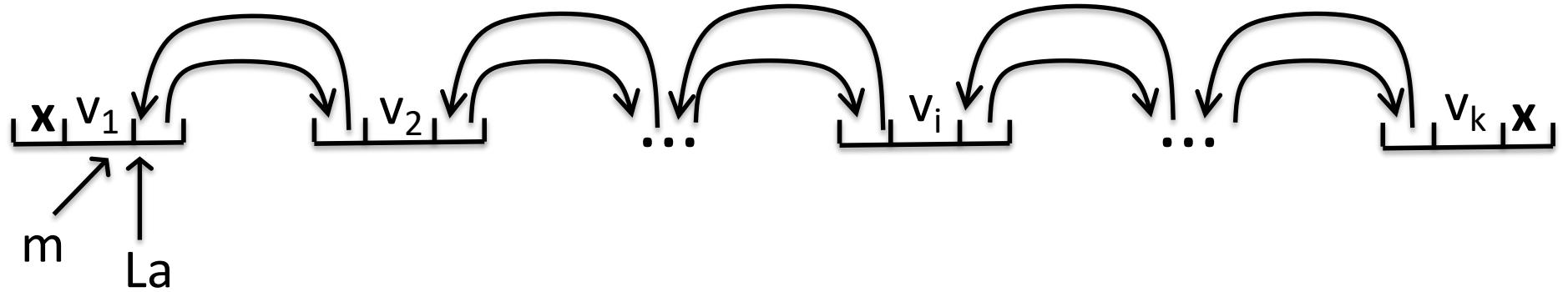






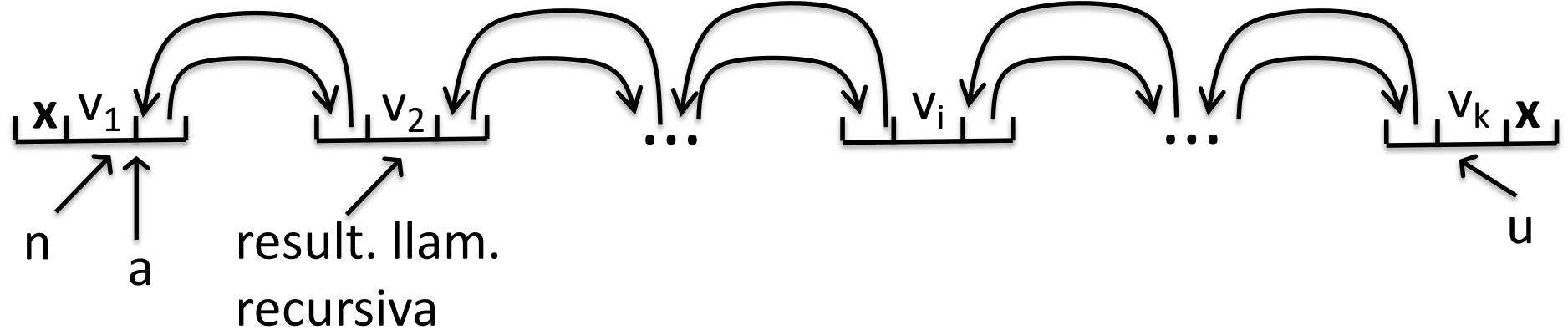
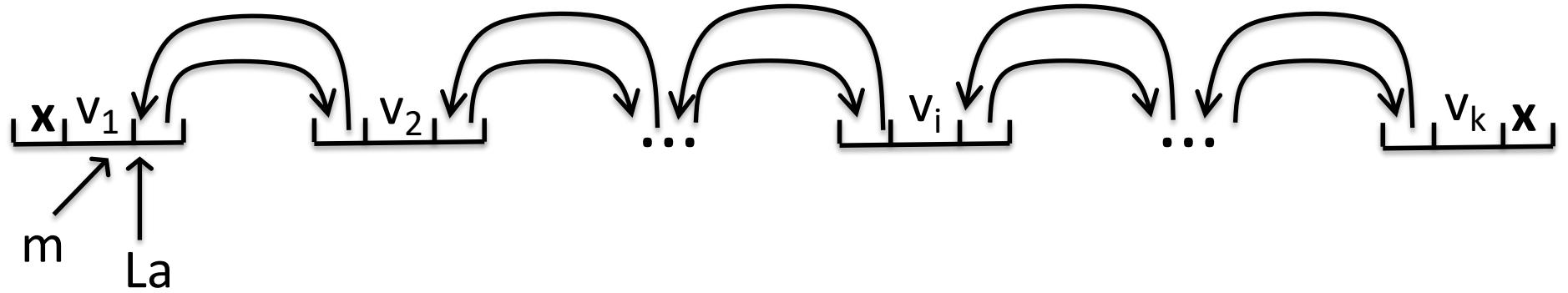
result. llam.
recursiva

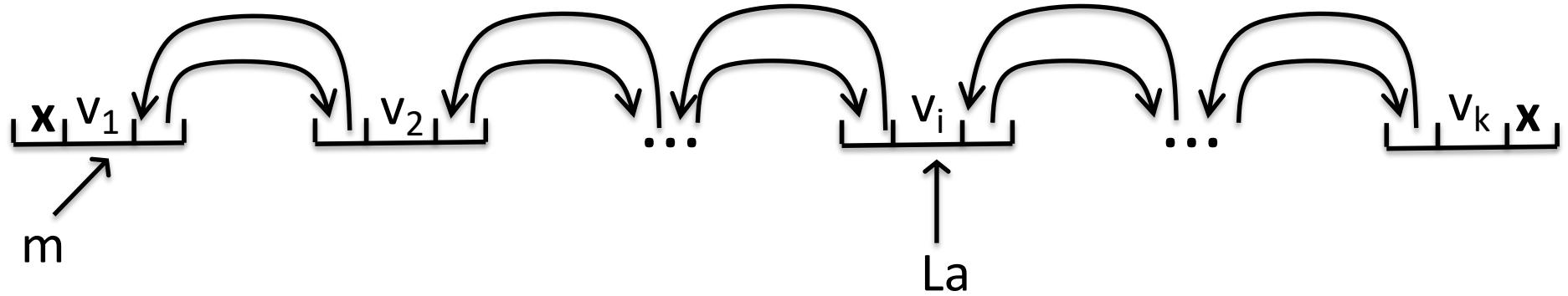
$a = \text{nullptr}$

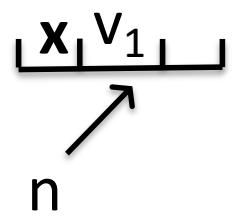
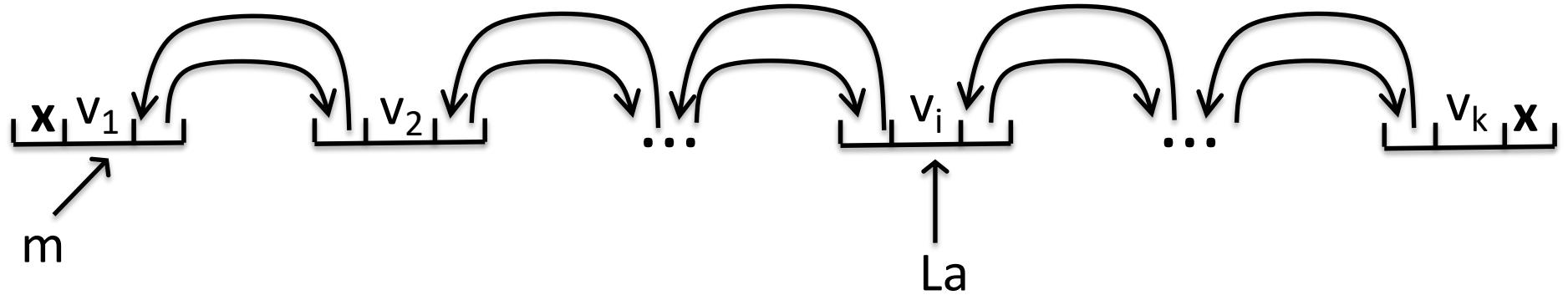


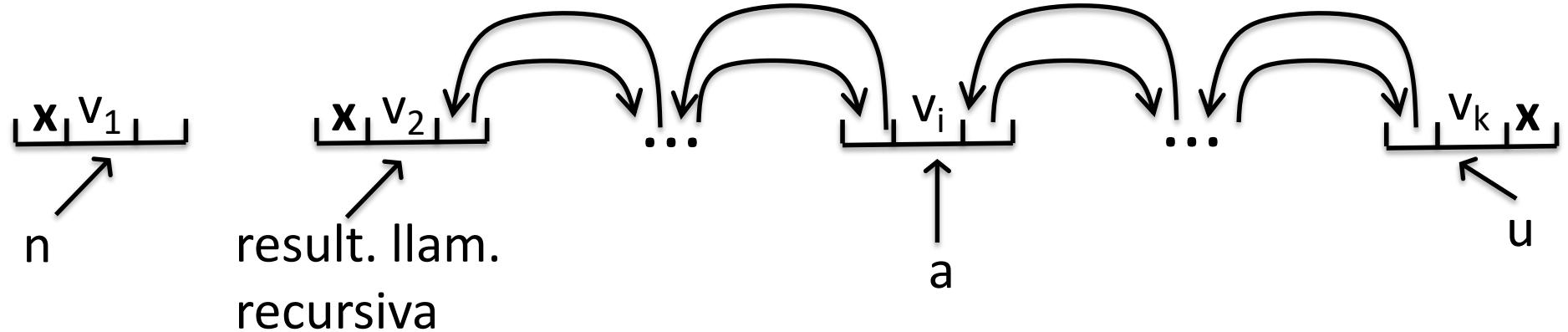
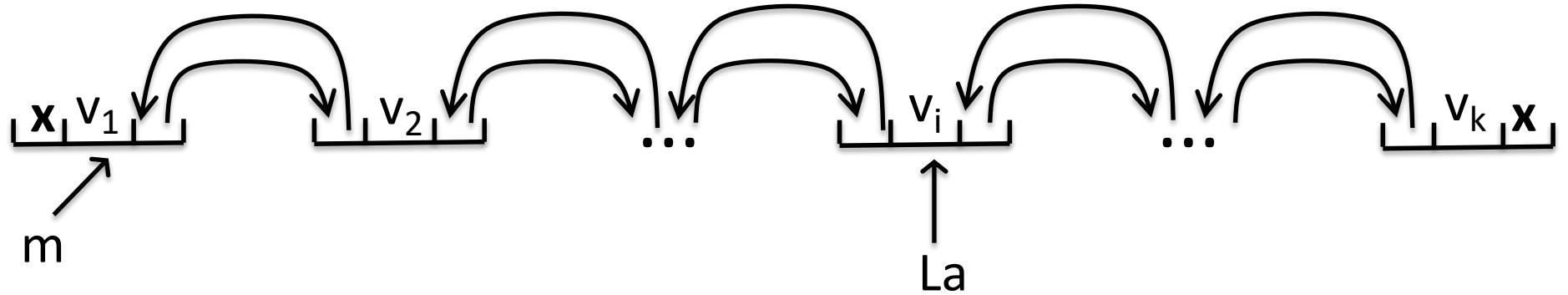
recursiva

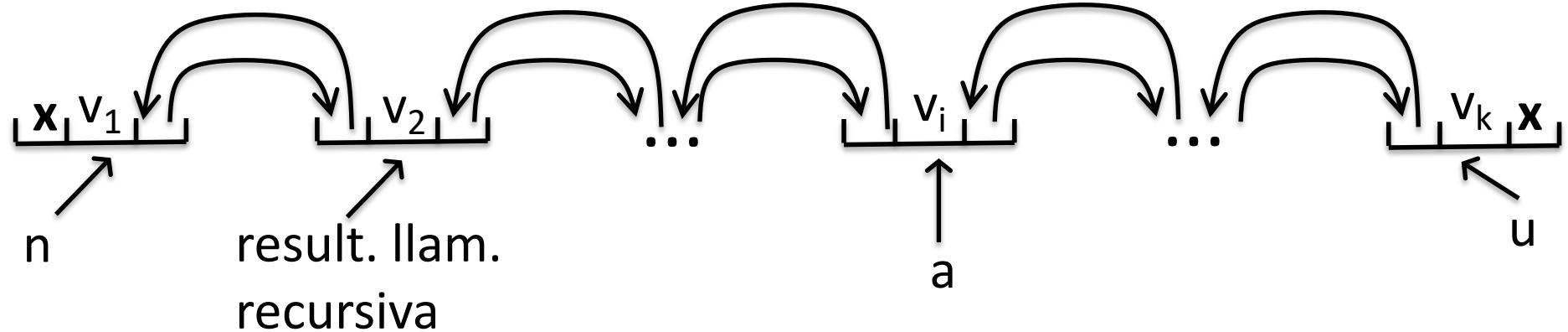
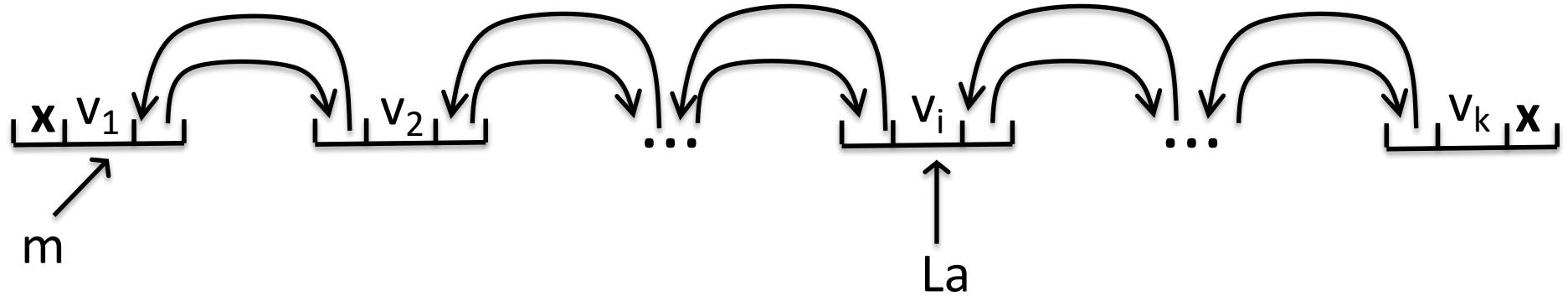
$a = \text{nullptr}$











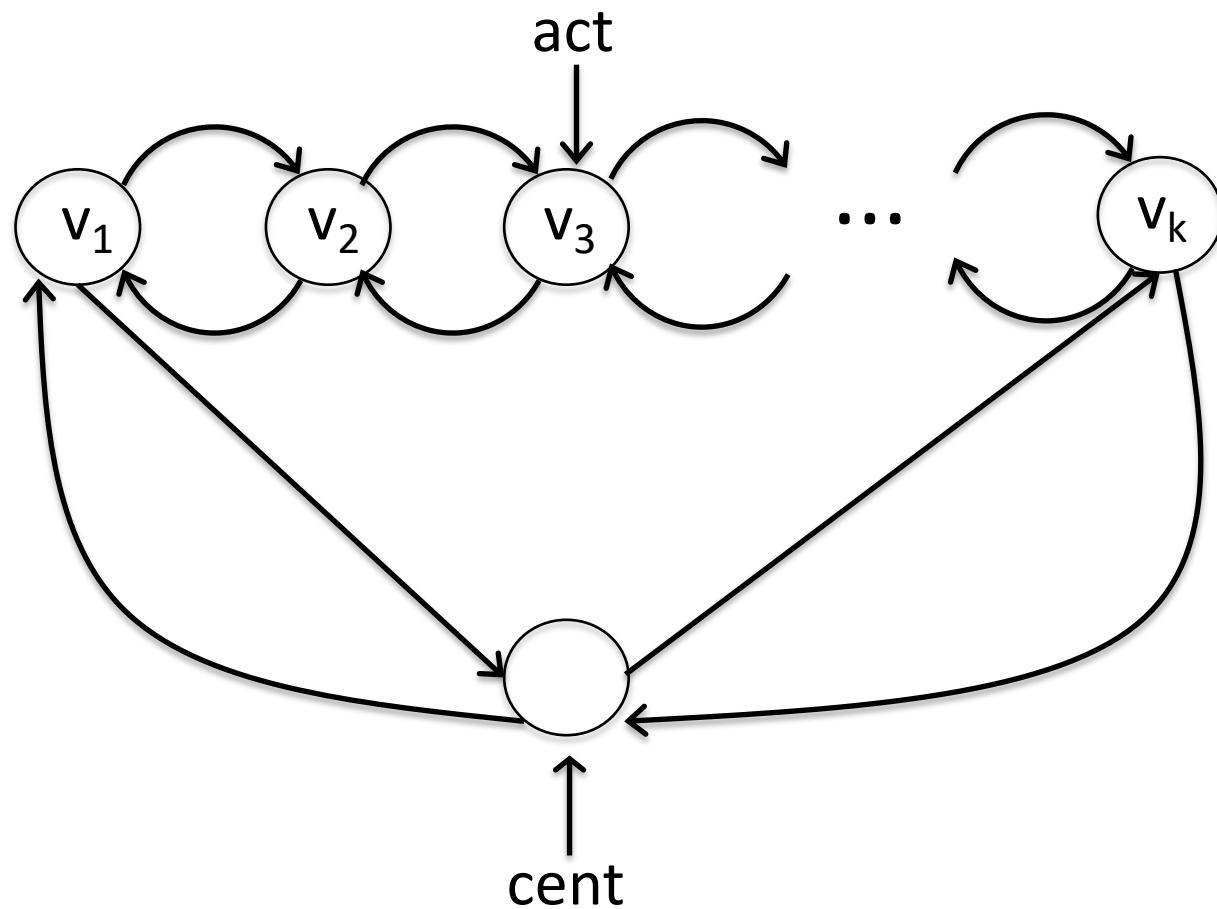
```
// Métodos privados
// Borrar secuencia de nodos
// Pre: true
/* Post: si m es nullptr no hace nada,
   si no libera el espacio ocupado por la cadena de
   nodos apuntada por m */

static void borra_nodo_lista(nodo_lista* m){
    if (m != nullptr) {
        borra_nodo_lista(m->sig);
        delete m;
    }
}
```

// Asignación

```
Lista& operator=(const Lista& L){  
    if (this != &L) {  
        longitud = L.longitud;  
        borra_nodo_lista(primer);  
        primero = copia_nodo_lista(L.primer, L.act,  
                                     ultimo, act);  
    }  
    return *this;  
}
```

Listas con centinela



Listas con centinela

- Objetivo: simplificar algunas operaciones como añadir y eliminar.
- Nodo extra, que no contiene ningún elemento real
- El centinela tiene como siguiente al primer elemento y como anterior al último.
- El centinela es el siguiente del ultimo elemento y el anterior del primero.
- La estructura no tiene valores nullptr: el centinela permite evitarlos
- Si la lista está vacía, el siguiente y anterior del centinela es el centinela y los punteros primero, último y act apuntan al centinela.

Implementación de Lista con centinela

```
template <class T> class Lista {  
    private:  
        struct nodo_lista{  
            T info;  
            nodo_lista* sig;  
            nodo_lista* ant;  
        };  
        int longitud;  
        nodo_lista* cent;  
        nodo_lista* act;  
        ... //operaciones privadas  
    public:  
        ... //operaciones públicas  
}
```

// Constructores y destructores

```
Lista(){
    longitud = 0;
    cent = new nodo_lista;
    act = cent;
    cent->sig = cent;
    cent->ant = cent;
}
Lista(const Lista& L){
    longitud = L.longitud;
    cent = copia_nodo_lista((L.cent)->sig, l.cent,
                           L.act, cent, act);
}
~Lista(){
    borra_nodo_lista(cent->sig, cent);
}
```

```
// Consultoras
```

```
bool es_vacia() const {  
    return longitud == 0;  
}
```

```
int medida() const {  
    return longitud;  
}
```

```
T actual() const {  
    // Pre: La lista no está vacía y el punto de  
    //       interés no es nullptr  
    return act->info;  
}
```

```
/* Consultoras para saber dónde está el punto  
de interés */
```

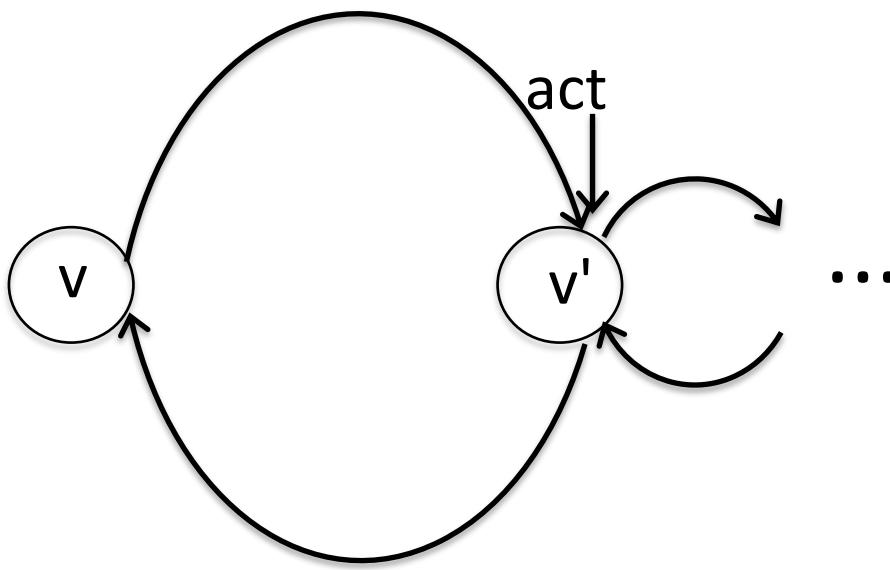
```
bool al_final() const {  
    return act == cent;  
}
```

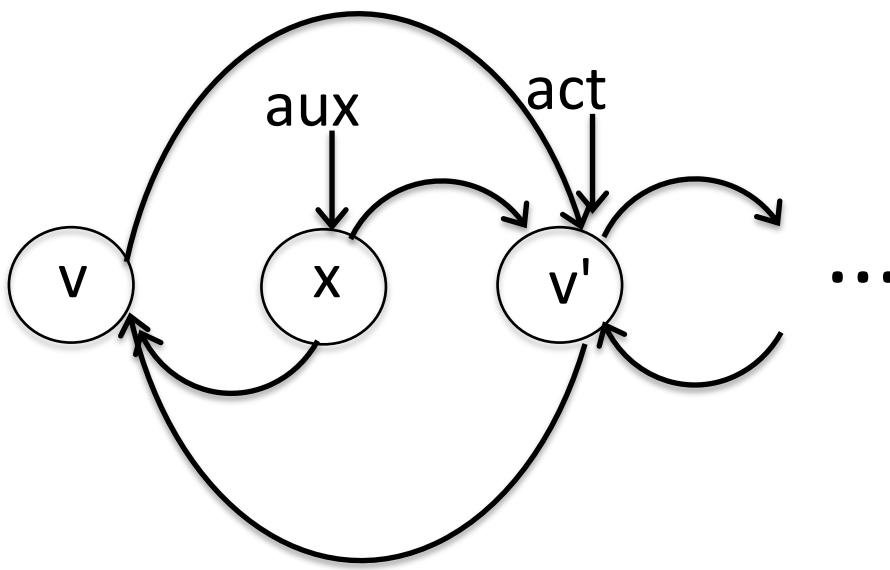
```
int al_principio() const {  
    return act == cent->sig;  
}
```

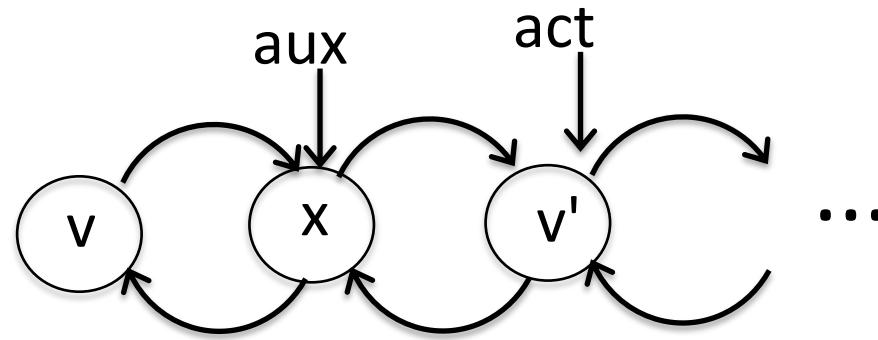
// Modificadoras

```
void l_vacia(){
    borra_nodo_lista(cent->sig, cent);
    longitud = 0;
    cent = new nodo_lista;
    act = cent;
    cent->sig = cent;
    cent->ant = cent;
}
```

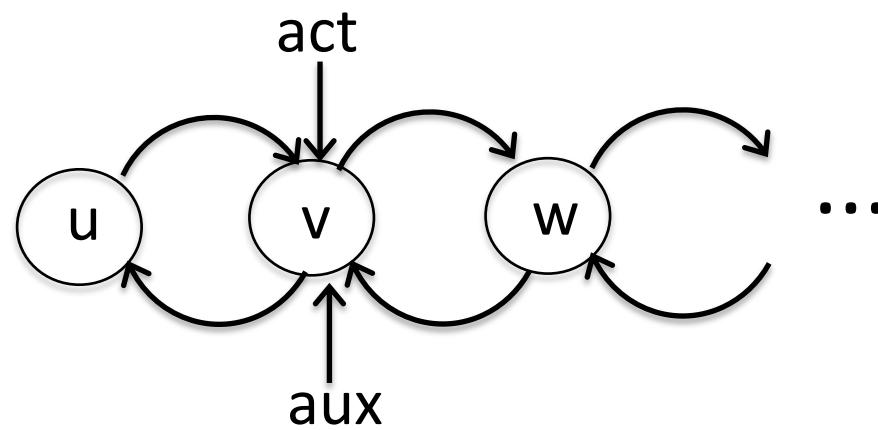
```
// Inserción
// Pre: true
/* Post: Se ha añadido un nodo con el valor x antes
del punto de interés que sigue siendo el mismo que
antes de la operación*/
void añadir(const T& x){
    nodo_lista * aux = new nodo_lista;
    aux->info = x;
    aux->sig = act;
    aux->ant = act->ant;
    (act->ant)->sig = aux;
    act->ant = aux;
    ++longitud;
}
```

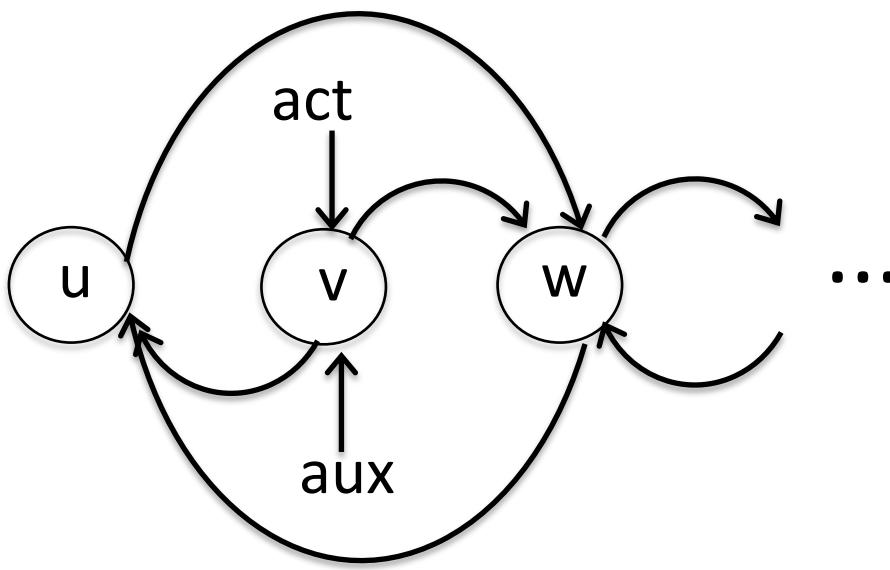


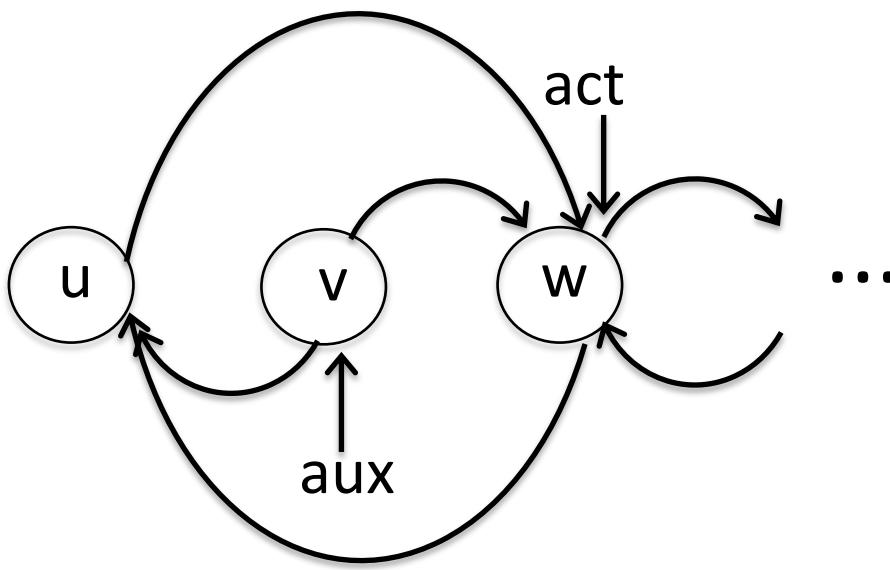


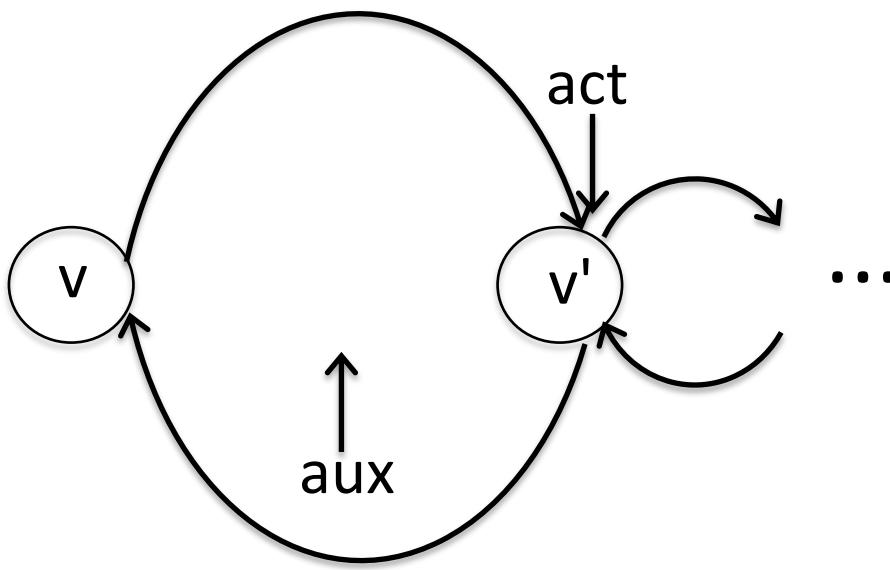


```
// Eliminación  
/* Pre: la lista no está vacía y su punto de interés  
no está al final */  
/* Post: Se ha eliminado el nodo donde estaba el  
punto de interés, el nuevo punto de interés es la  
posición siguiente al nodo eliminado */  
void eliminar(){  
    nodo_lista * aux = act;  
    (act->ant)->sig = act->sig;  
    (act->sig)->ant = act->ant;  
    act = act->sig;  
    delete aux;  
    --longitud;  
}
```









```
// Concatenación
// Pre: true
/* Post: Se han añadido al final los elementos de L, el
punto de interés es el primer elemento, L queda vacía*/
void concat(Lista & L){
    if (L.longitud > 0) {
        if (longitud == 0) swap(cent,L.cent);
        else { (cent->ant)->sig = (L.cent)->sig;
                ((L.cent)->sig)->ant = cent->ant;
                cent->ant = (L.cent)->ant;
                ((L.cent)->ant)->sig = cent;
                (L.cent)->sig = L.cent;
                (L.cent)->ant = L.cent;
        }
        L.act = L.cent;
        longitud = longitud + L.longitud; L.longitud = 0;
    }
    act = cent->sig;
}
```

// modificación y movimiento del punto de interés

```
void modifica_actual(const T & x){  
    act->info = x;  
}
```

```
void inicio(){  
    act = cent->sig;  
}
```

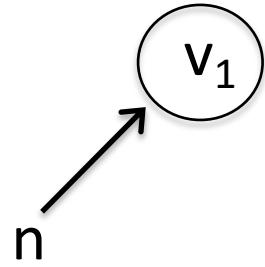
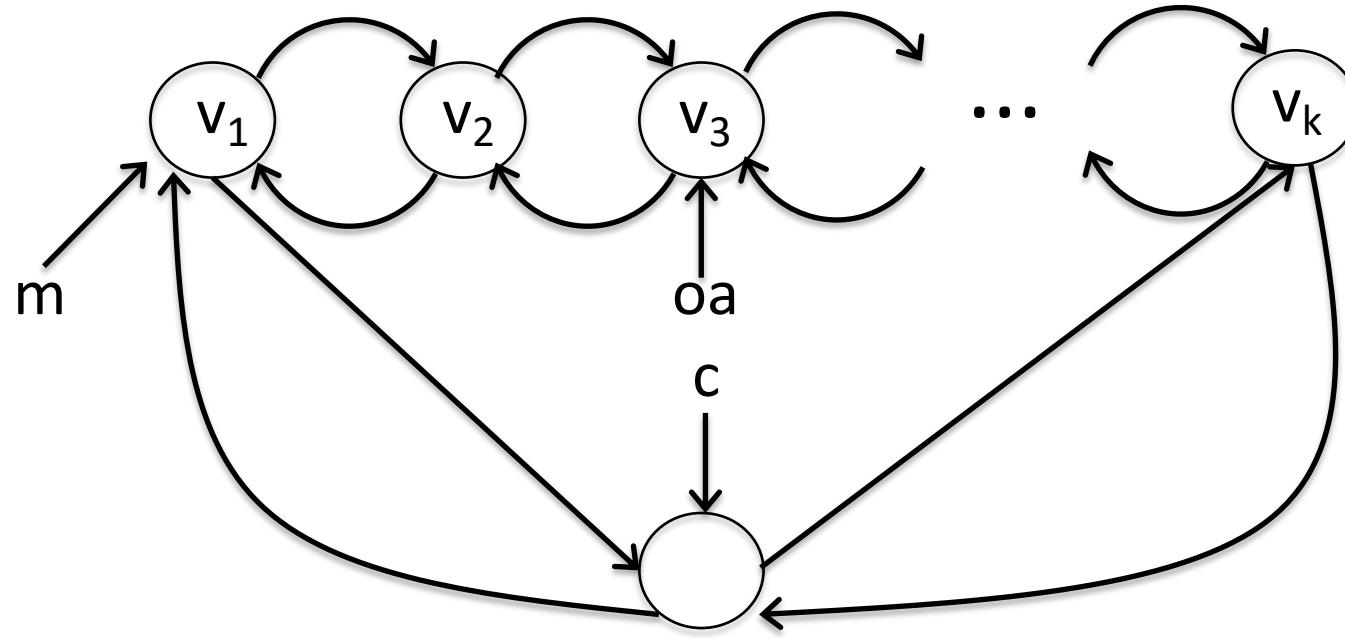
```
void fin(){  
    act = cent;  
}
```

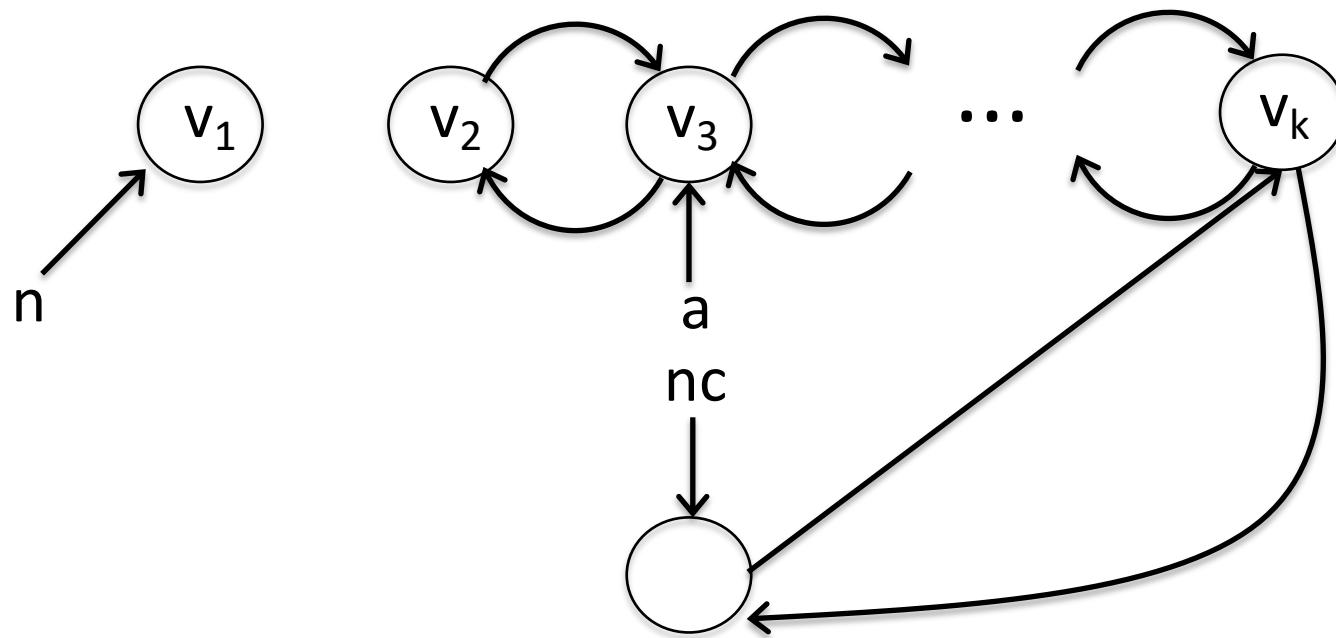
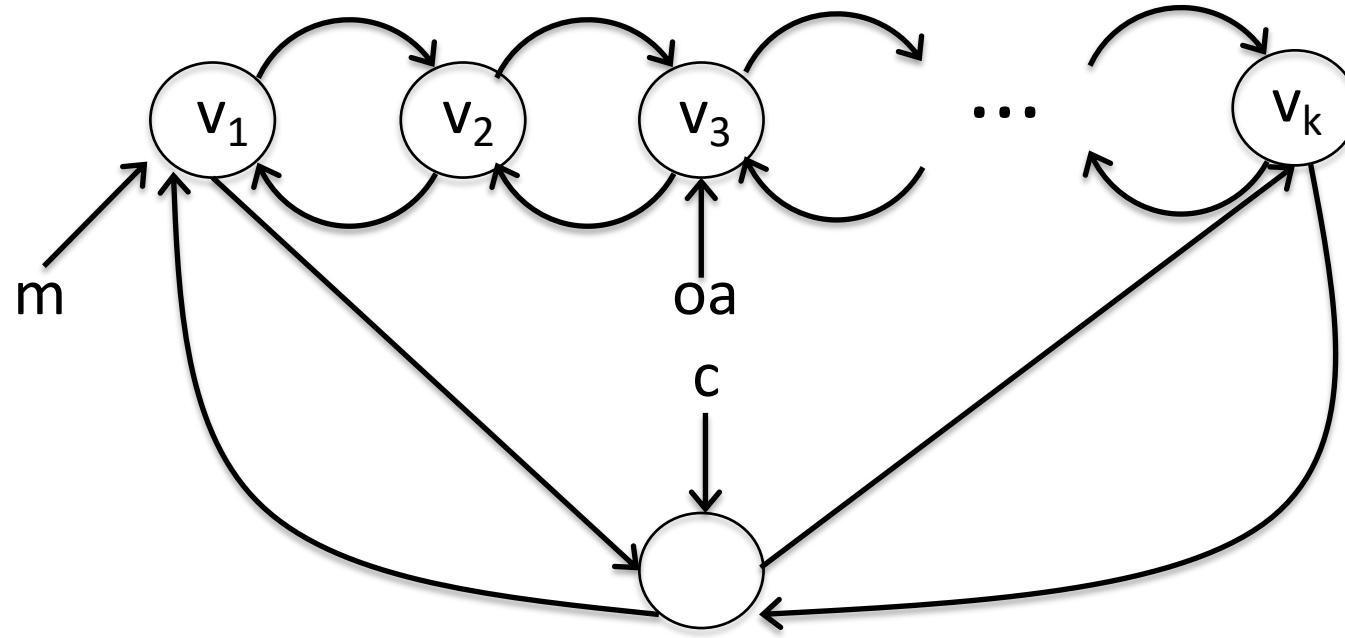
```
void avanza(){  
    act = act->sig;  
}
```

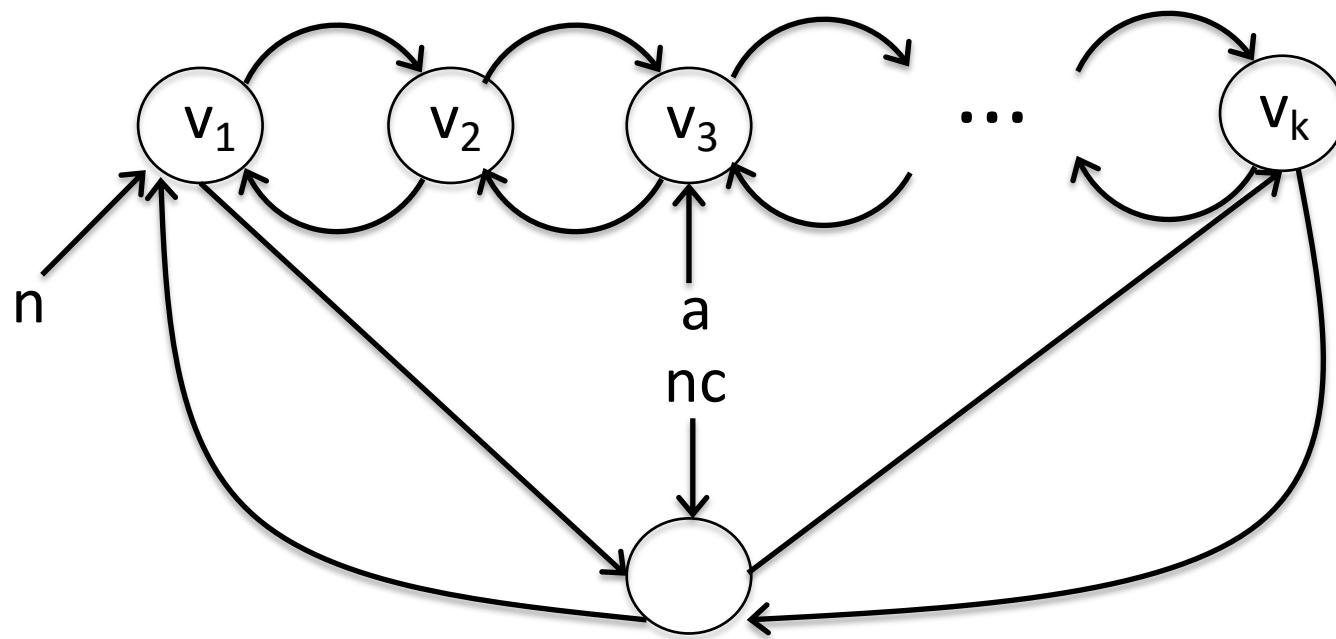
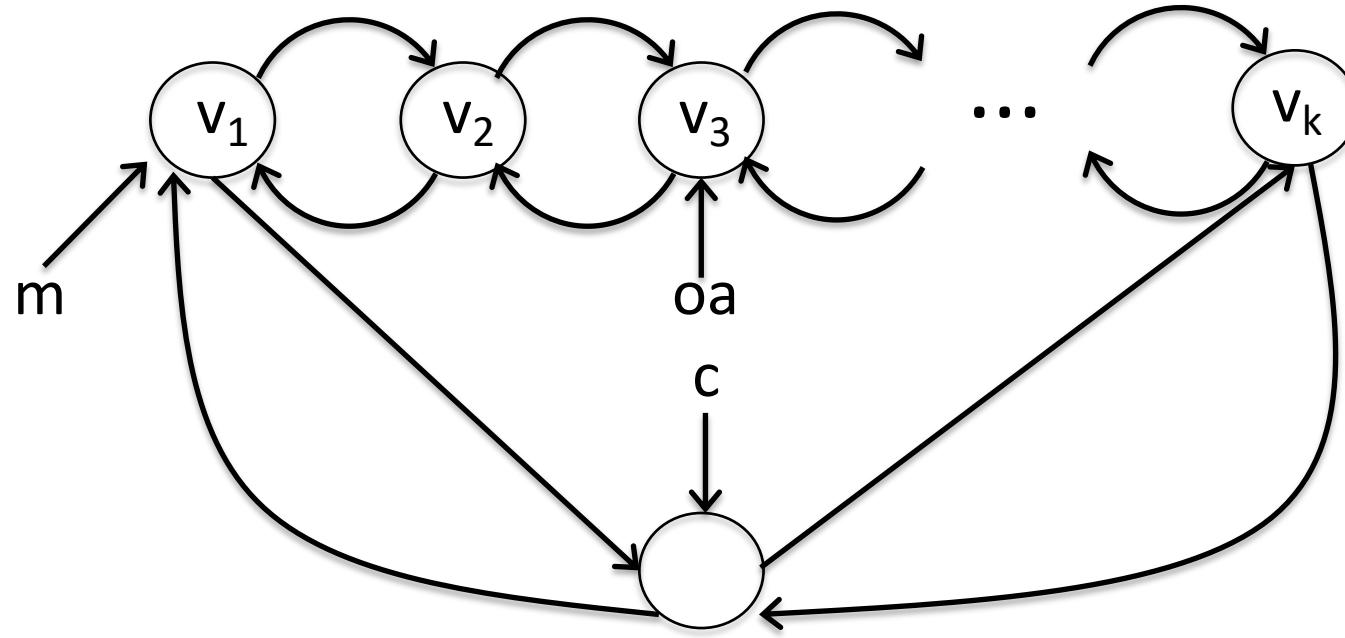
```
void retrocede(){
    act = act->ant;
}
```

```
// Métodos privados
// Copiar secuencia de nodos
static nodo_Lista* copia_nodo_Lista (
    nodo_Lista* m, nodo_Lista* c, nodo_Lista* oa,
    nodo_Lista* &nc, nodo_Lista* &a);
/* Pre: m, oa y c apuntan a nodos de la misma
   secuencia, oa apunta a un nodo entre m y c */
/* Post: retorna como resultado una secuencia de nodos
   que es copia de la secuencia entre m y c, si m y c
   apuntan al mismo nodo, entonces nc y a apuntan al
   mismo nodo, si no, nc apunta a la copia del centinela
   c y a apunta a la copia del nodo apuntado por oa. */
```

```
static nodo_Lista* copia_nodo_Lista (
    nodo_Lista* m, nodo_Lista* c, nodo_Lista* oa,
    nodo_Lista* &nc, nodo_Lista* &a);
nodo_lista* n = new nodo_lista;
if (m == c) {n->ant = n; n->sig = n; nc = n; a = n;}
else {
    n->info = m->info;
    n->sig = copia_nodo_Lista(m->sig, c, oa, nc, a);
    (n->sig)->ant = n;
    nc->sig = n;
    n->ant = nc;
    if (m == oa) a = n;
}
return n;
}
```







```
// Métodos privados
// Borrar secuencia de nodos
/* Pre: c apunta a un centinela, m apunta a la misma
secuencia de nodos */
/* Post: libera el espacio ocupado por la cadena de
nodos entre m* y c*, ambos incluidos/
```



```
static void borra_nodo_lista(nodo_lista* m,
                               nodo_lista* c){
    if (m != c) {
        borra_nodo_lista(m->sig,c);
        delete m;
    }
}
```

// Asignación

```
Lista& operator=(const Lista& L){  
    if (this != &L) {  
        longitud = L.longitud;  
        borra_nodo_lista(primer,cent);  
        nodo_lista* aux = copia_nodo_lista((L.cent)->sig,  
                                            L.cent, L.act, cent, act);  
    }  
    return *this;  
}
```

Árboles binarios

La clase Arbol

- No son exactamente lo mismo que la clase BinTree de C++
- Principales Operaciones:
 - a.plantar(x, a1, a2): a ha de estar vacío, y sea un objeto diferente de a1 y a2. Después de la operación, a1 y a2 quedan vacíos
 - a.hijos(a1, a2), a1 y a2 han de ser vacíos y, al final a, a1 y a2 serán objetos diferentes
 - a.raiz(), retorna el valor en la raíz de a

Implementación de la clase Árbol

```
template <class T> class Arbol {  
    private:  
        struct nodo_arbol{  
            T info;  
            nodo_arbol* sigI;  
            nodo_arbol* sigD;  
        };  
        nodo_arbol* primer_nodo;  
  
    ... //operaciones privadas  
    public:  
    ... //operaciones públicas  
}
```

```
// Constructores y destructores

Arbol(){
    primer_nodo = nullptr;
}

Arbol(const Arbol& A){
    primer_nodo = copia_nodo_arbol(A.primer_nodo);
}

~Arbol(){
    borra_nodo_arbol(primer_nodo);
}
```

```
// Consultoras
```

```
bool es_vacio() const {  
    return primer_nodo == nullptr;  
}
```

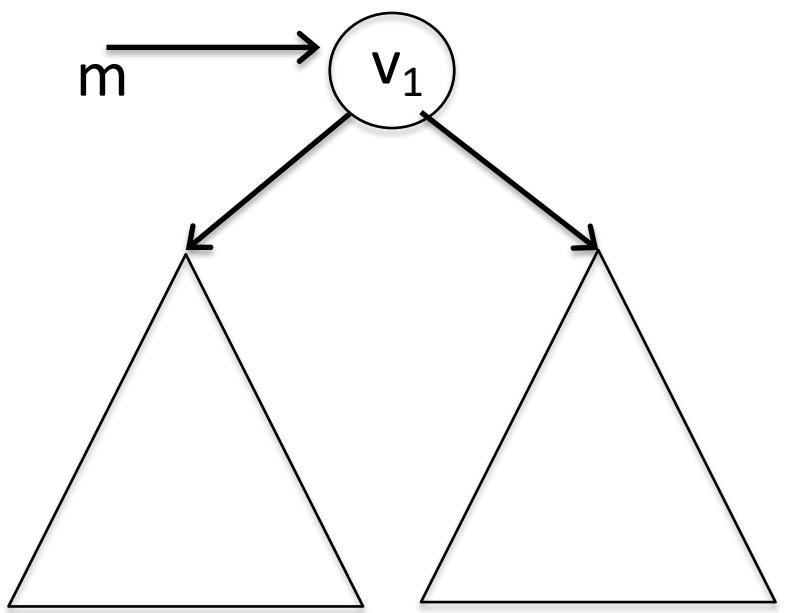
```
T raiz() const {  
// Pre: El arbol no está vacío  
    return primer_nodo->info;  
}
```

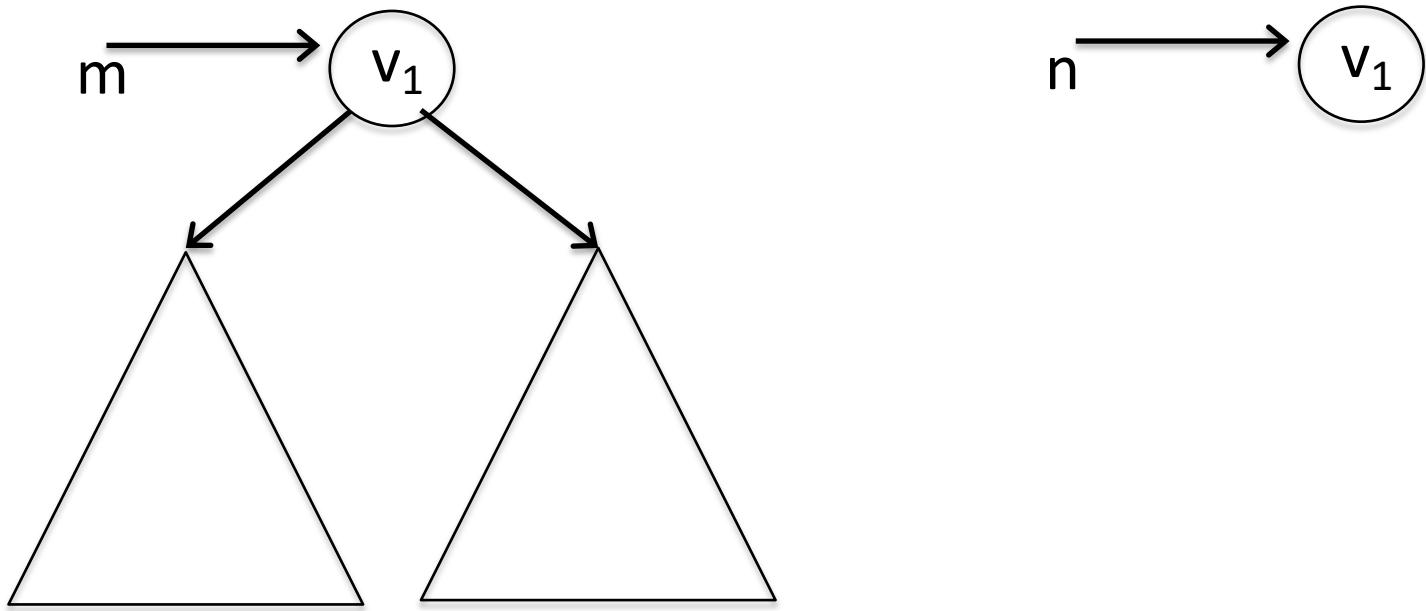
```
// Modificadoras
/* Pre: El p.i. está vacío, a1 = A1, a2 = A2, a1 y a2
   son objetos diferentes del p.i. */
/* Post: El parámetro implícito tiene x en la raiz, A1
   como hijo izquierdo, A2 como hijo derecho, y a1 y a2
   están vacíos */

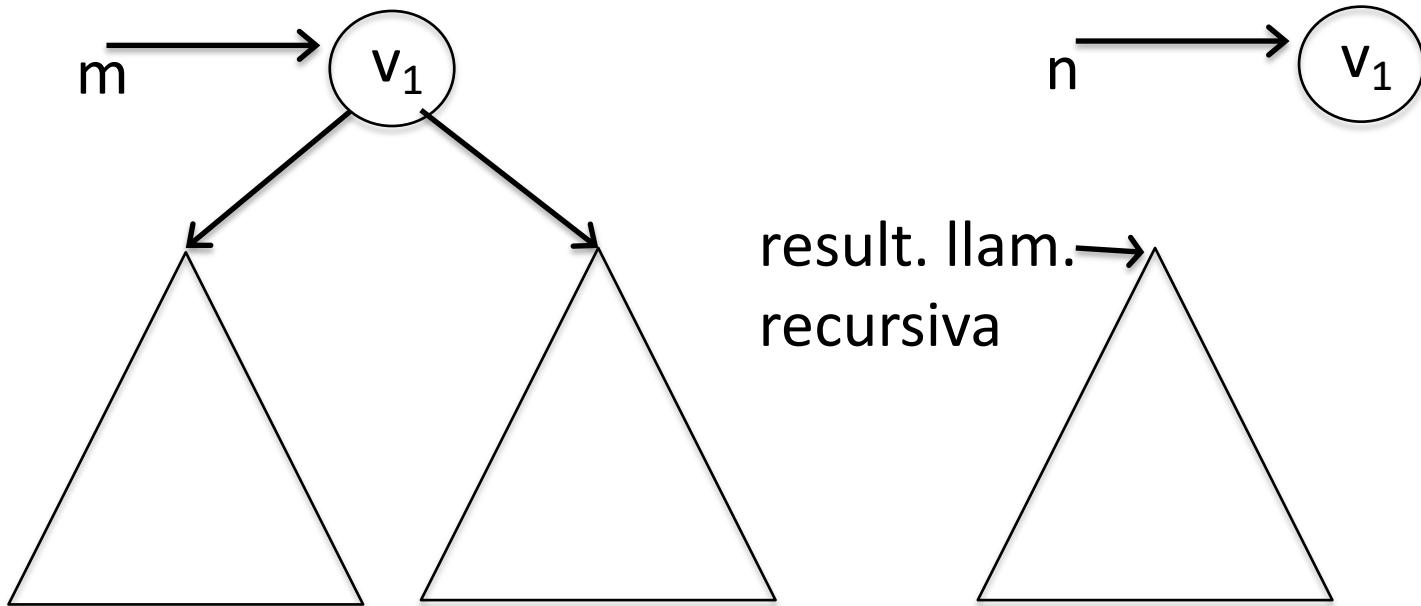
void plantar(const T& x, Arbol &a1, Arbol &a2){
    nodo_arbol * aux = new nodo_arbol;
    aux->info = x;
    aux->sigI = a1.primer_nodo;
    if (a2.primer_nodo != a1.primer_nodo or
        a2.primer_nodo == nullptr)
        aux->sigD = a2.primer_nodo;
    else aux->sigD = copia_nodo_arbol (a2.primer_nodo)
    primer_nodo = aux;
    a1.primer_nodo = nullptr; a2.primer_nodo = nullptr
}
```

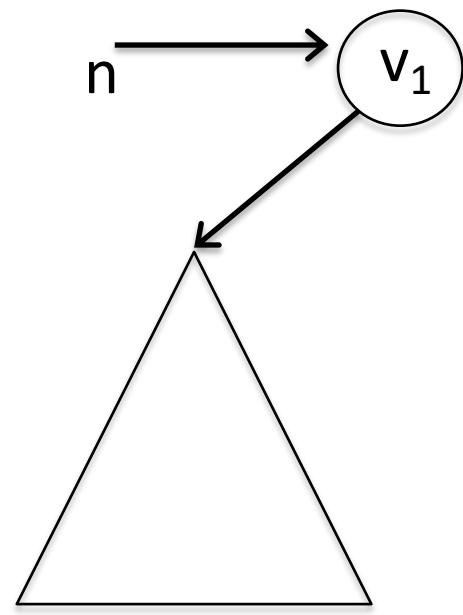
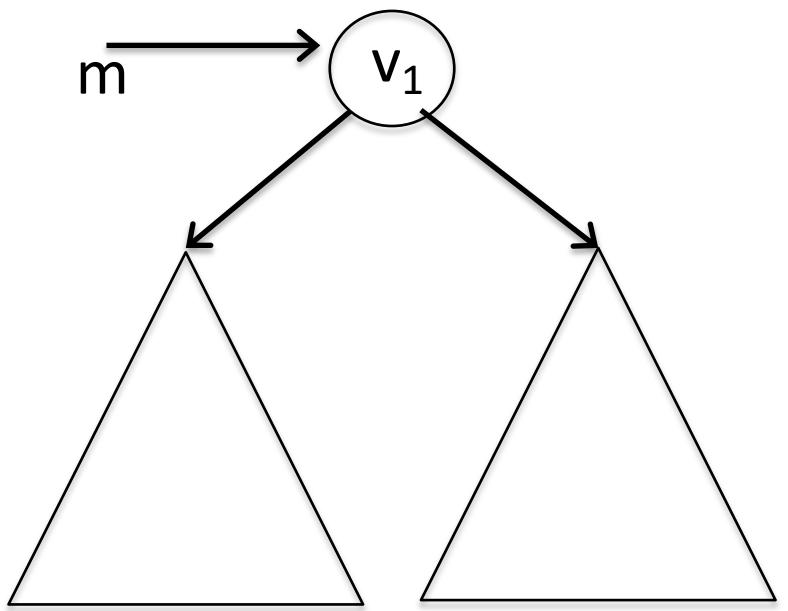
```
/* Pre: Si el parámetro de interés es A, A no está vacío, hi y hd están vacíos y son objetos diferentes */
/* Post: hi es el hijo izqdo de A, hd es el hijo dcho de A, el p.i. está vacío */
void hijos(Arbol &hi, Arbol &hd){
    nodo_lista * aux = primer_nodo;
    hi.primer_nodo = primer_nodo->sigI;
    hd.primer_nodo = primer_nodo->sigD;
    primer_nodo = nullptr;
    delete aux;
}
```

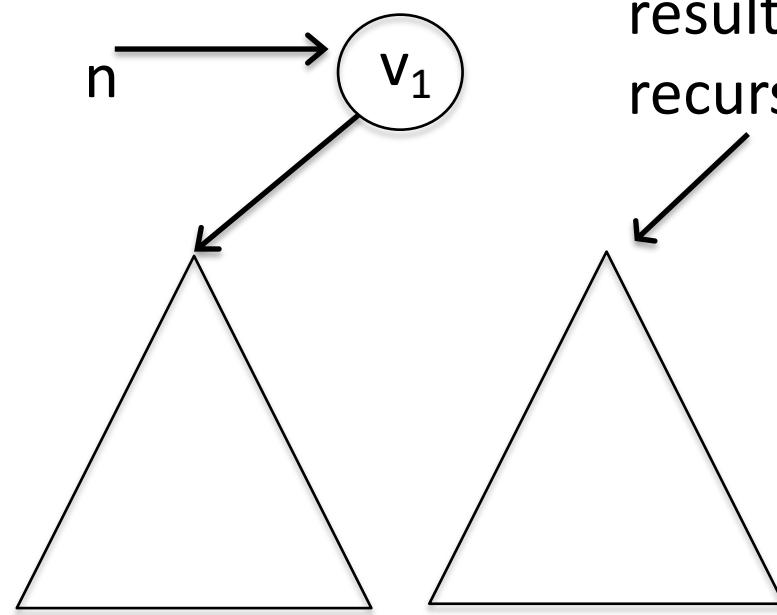
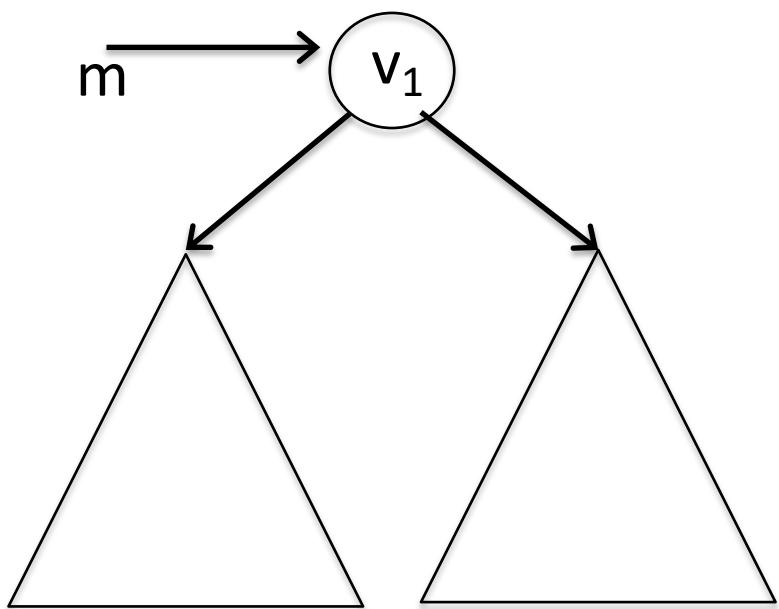
```
// Métodos privados
// Copiar jerarquía de nodos
static nodo_arbol* copia_nodo_arbol (nodo_arbol* m) {
/* Pre: true */
/* Post: Si m es nullptr, retorna nullptr, si no retorna
una copia de la jerarquía de nodos apuntada por m*/
if (m == nullptr) return nullptr;
else {
    nodo_arbol* n = new nodo_arbol;
    n->info = m->info;
    n->sigI = copia_nodo_arbol(m->sigI);
    n->sigD = copia_nodo_arbol(m->sigD);
    return n;
}
}
```

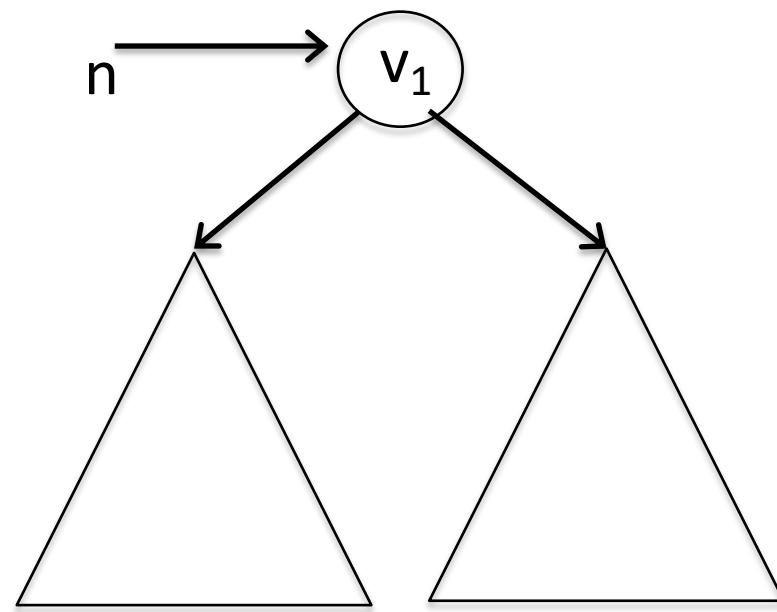
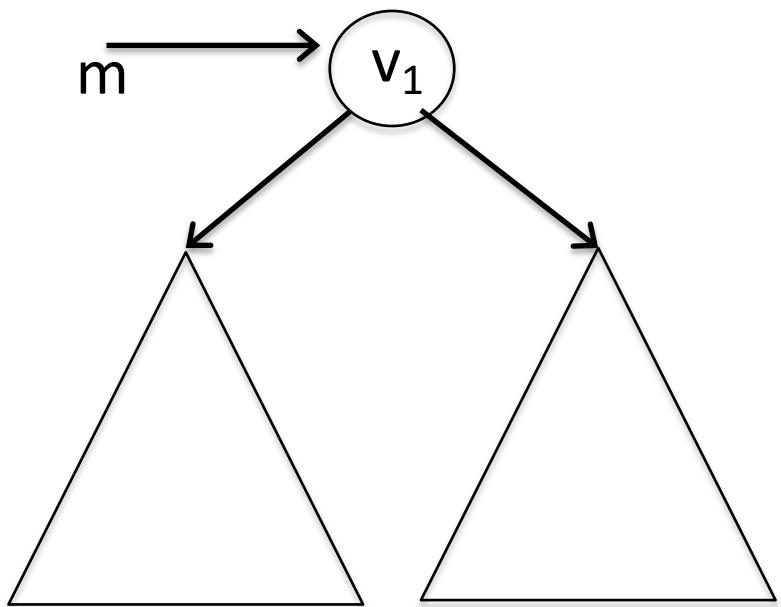












```
// Métodos privados
// Borrar jerarquia de nodos
/* Pre: true */
/* Post: libera el espacio ocupado por todos los
nodos que cuelgan de m*/
static void borra_nodo_arbol(nodo_arbol* m){
    if (m != nullptr) {
        borra_nodo_arbol(m->sigI);
        borra_nodo_arbol(m->sigD);
        delete m;
    }
}
```

// Asignación

```
Arbol& operator=(const Arbol& A){  
    if (this != &A) {  
        borra_nodo_arbol(primer_nodo);  
        primer_nodo= copia_nodo_arbol(A.primer_nodo);  
    }  
    return *this;  
}
```