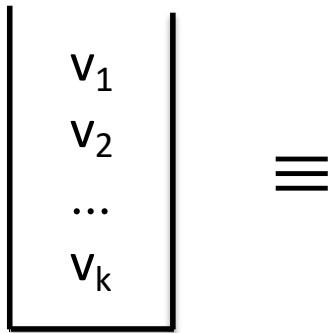


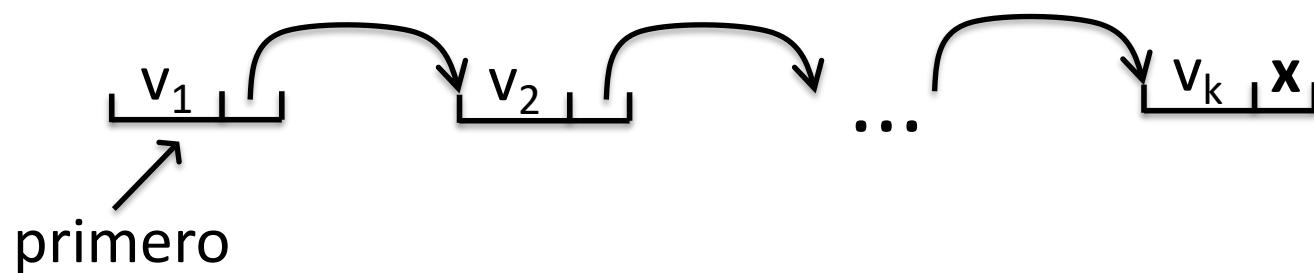
Pilas

Implementación de pilas

```
template <class T> class stack {  
    private:  
        // tipo privado nuevo  
        struct nodo_pila{  
            T info;  
            nodo_pila* sig;  
        };  
        int altura;  
        nodo_pila* primero;  
        ... //operaciones privadas  
    public:  
        ... //operaciones públicas  
}
```



=



primero

// Constructores

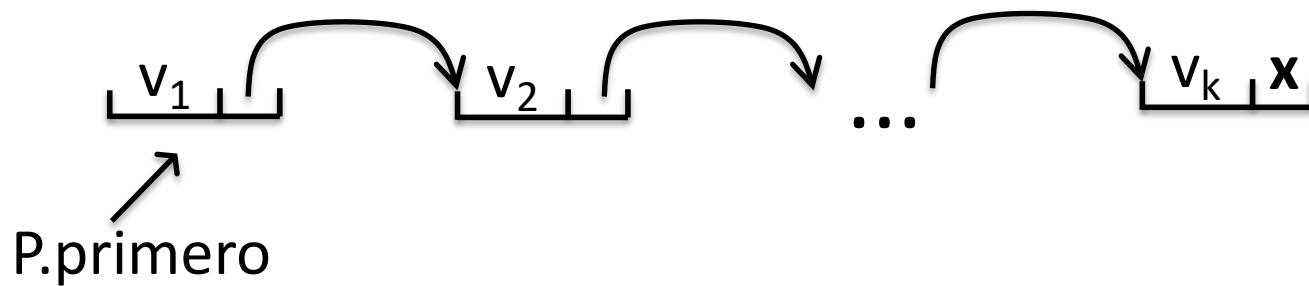
```
stack(){
    altura = 0;
    primero = nullptr;
}
```

```
stack(const stack& P){
```

```
}
```

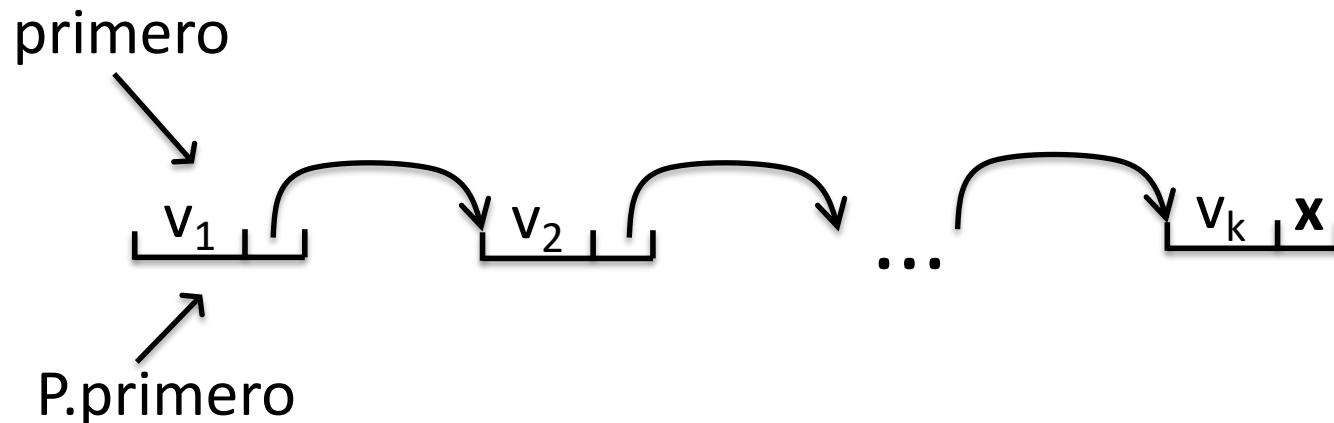
// Constructores

```
stack(const stack& P){  
    altura = p.altura;  
    primero = p.primero  
}
```



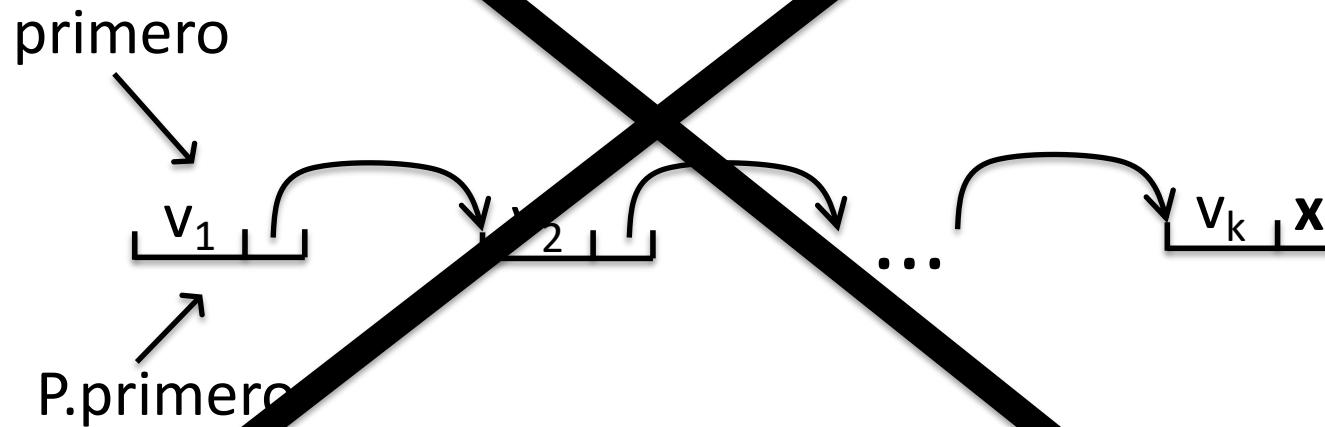
// Constructoras

```
Si stack(const stack& P){  
    altura = p.altura;  
    primero = p.primero  
}
```



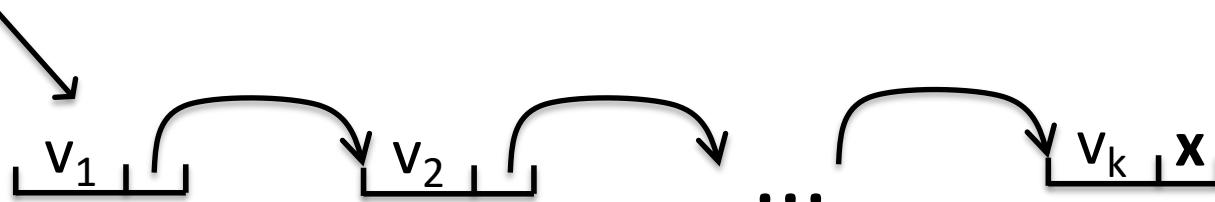
// Constructores

Si ~~stack(const stack& P){~~
~~altura = p.altura;~~
~~primero = p.primero~~
}

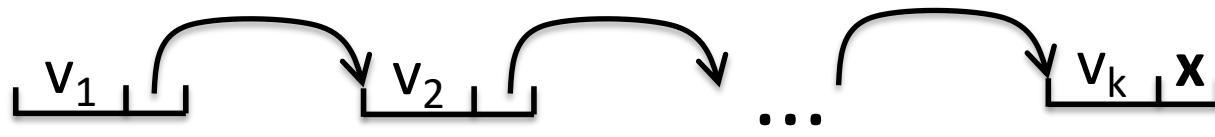


// Constructoras

primero



P.primero



// Constructores

```
stack(){  
    altura = 0;  
    primero = nullptr;  
}  
  
stack(const stack& P){  
    altura = P.altura;  
    primero = copia_nodo_pila(P.primer);  
        //retorna una copia de todo  
        //lo que cuelga del parámetro  
}
```

```
// Destrucción
```

```
~stack(){
    borra_nodo_pila(primer);
    //elimina todo lo que cuelga
    //del parámetro
}
```

// Consultoras

```
T top() const {
    // Pre: la pila no está vacía
    return primero->info;
}

bool empty() const {
    return altura == 0;
}

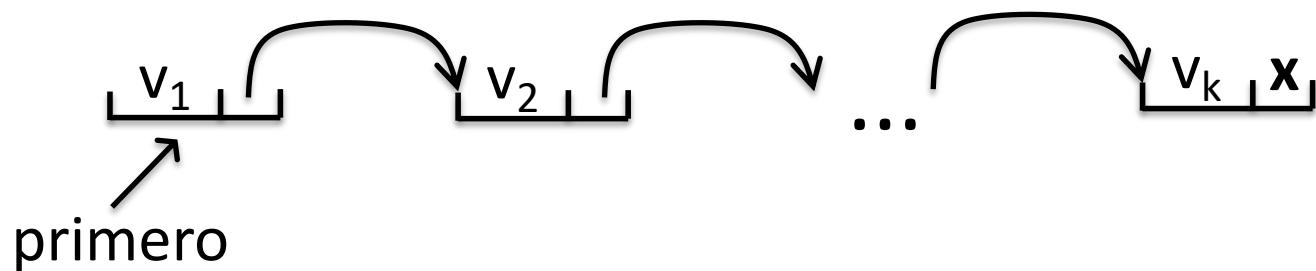
int size() const {
    return altura;
}
```

// Modificadoras

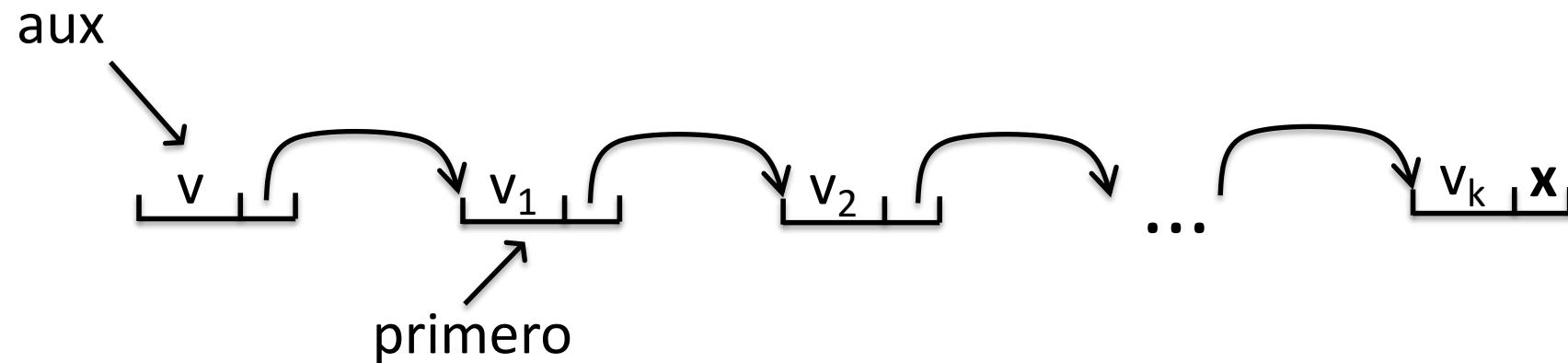
```
void clear(){
    borra_nodo_pila(primer);
    altura = 0;
    primero = nullptr;
}

void push(const T& v){
    nodo_pila * aux = new nodo_pila;
    aux->info = v;
    aux->sig = primero;
    primero = aux;
    ++altura;
}
```

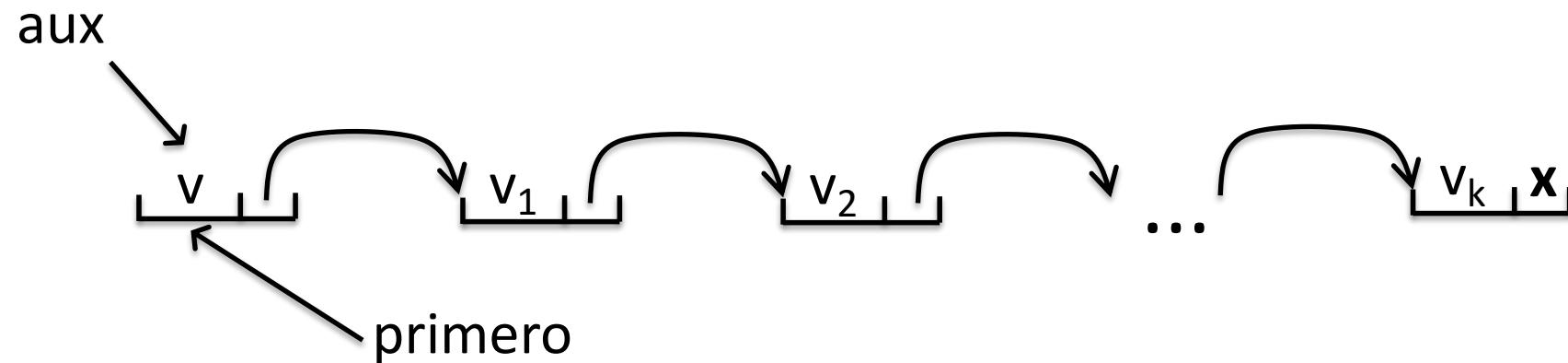
push(v):



push(v):



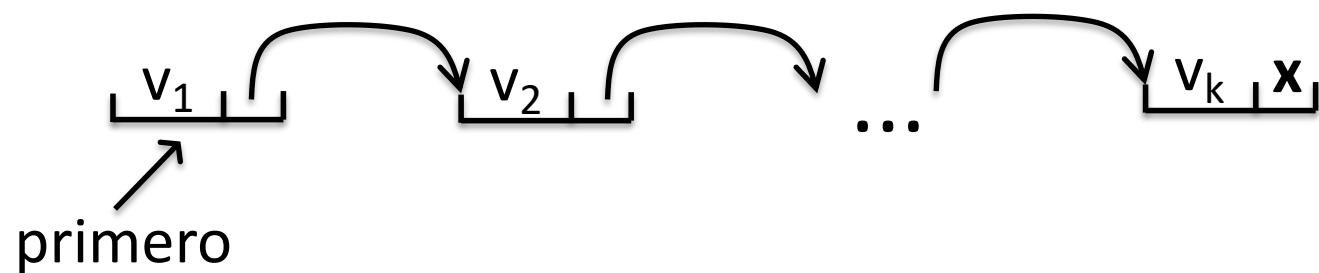
push(v):



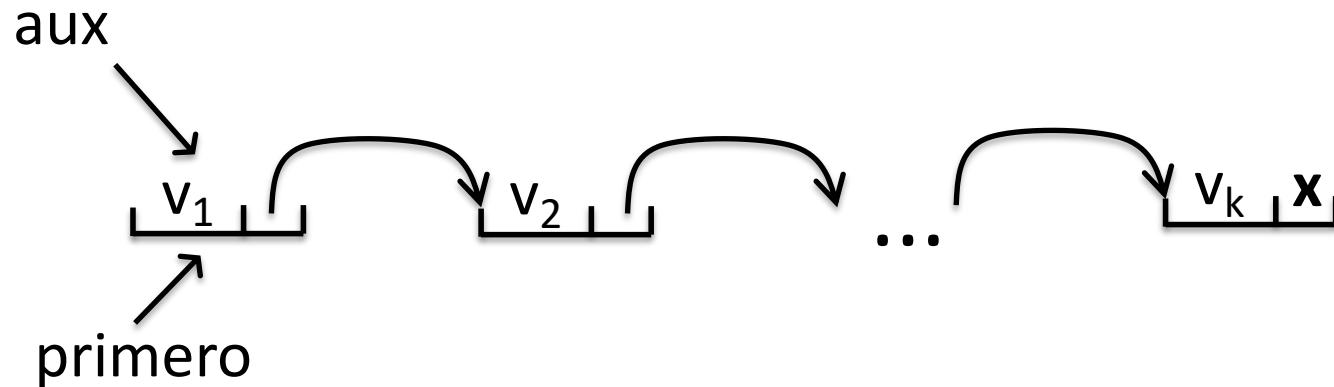
// Modificadoras

```
void pop(){
    // Pre: la pila no está vacía
    nodo_pila * aux = primero;
    primero = primero->sig;
    delete aux;
    --altura;
}
```

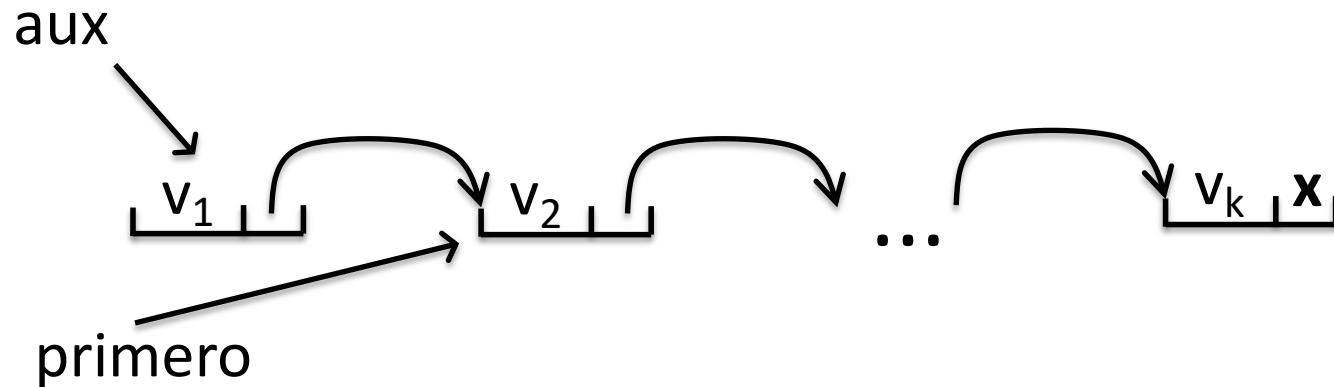
pop():



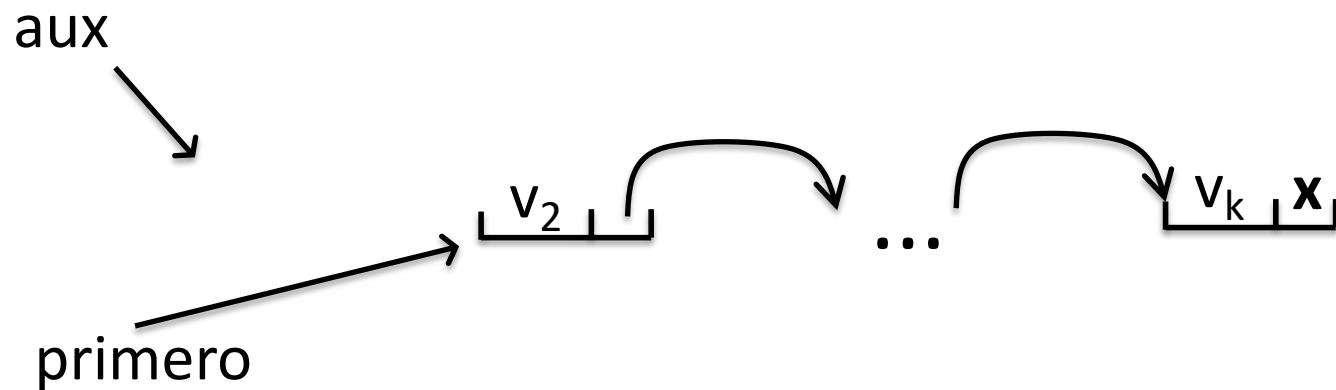
pop():



pop():



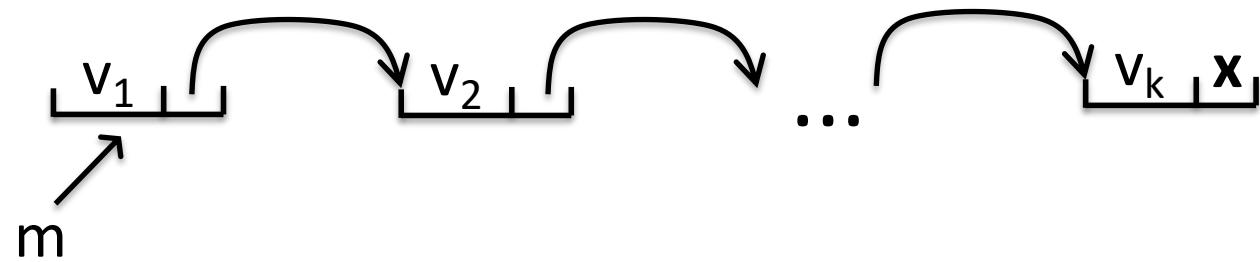
pop():



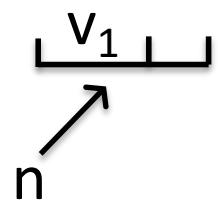
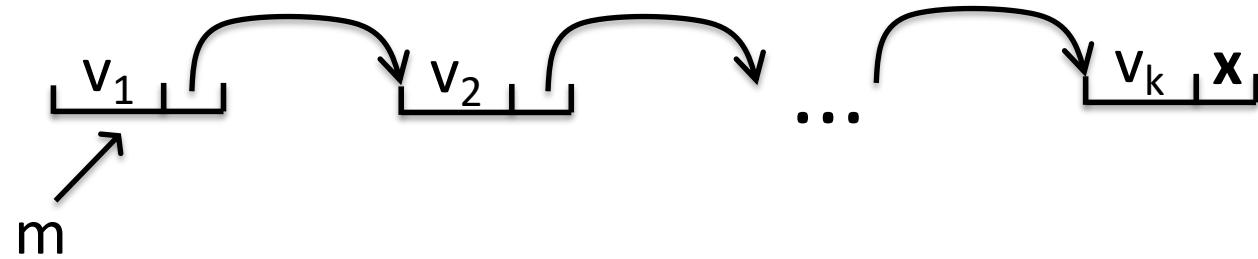
```
// Métodos privados
// Pre: true
/* Post: si m es nullptr el resultado es nullptr,
   si no el resultado apunta a una cadena de nodos
   que es una copia de la cadena apuntada por m */

static nodo_pila* copia_nodo_pila(nodo_pila* m){
    if (m == nullptr) return nullptr;
    else{
        nodo_pila* n = new nodo_pila;
        n->info = m->info;
        n->sig = copia_nodo_pila(m->sig);
        return n;
    }
}
```

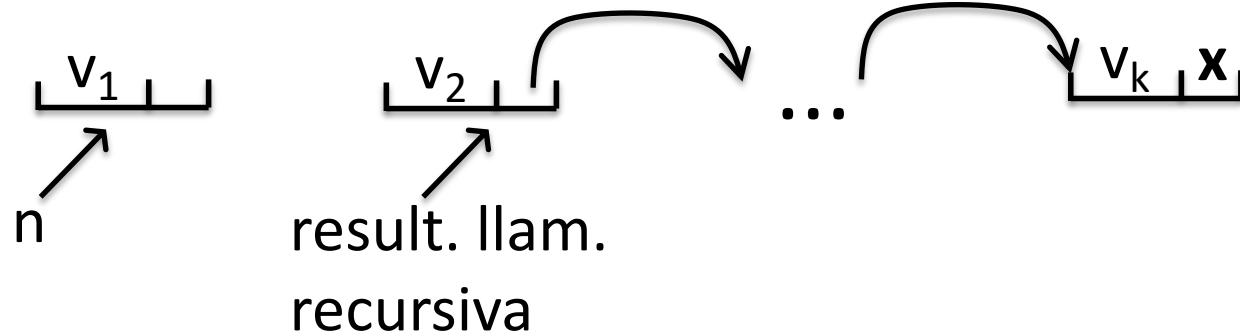
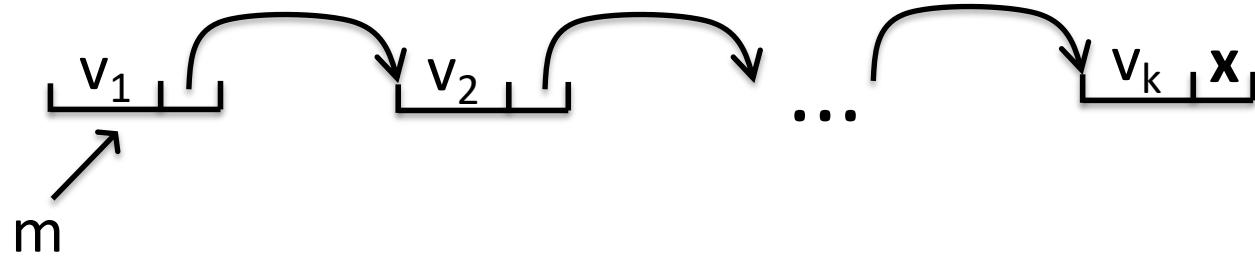
copia_nodo_pila(m):



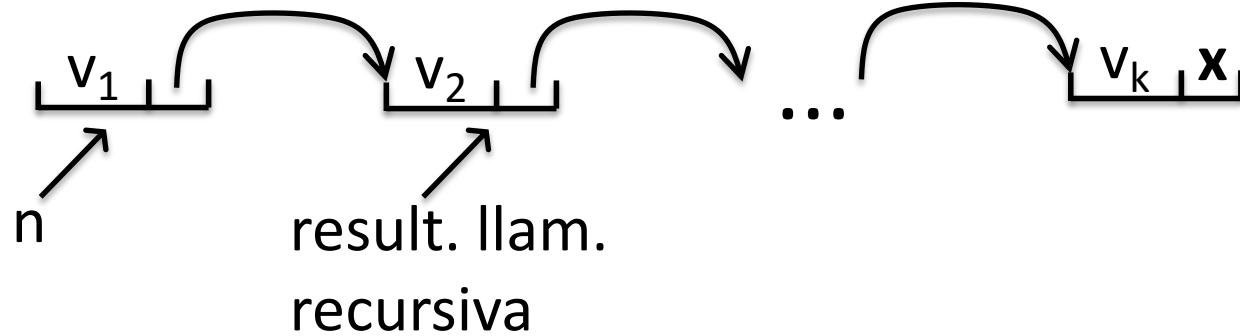
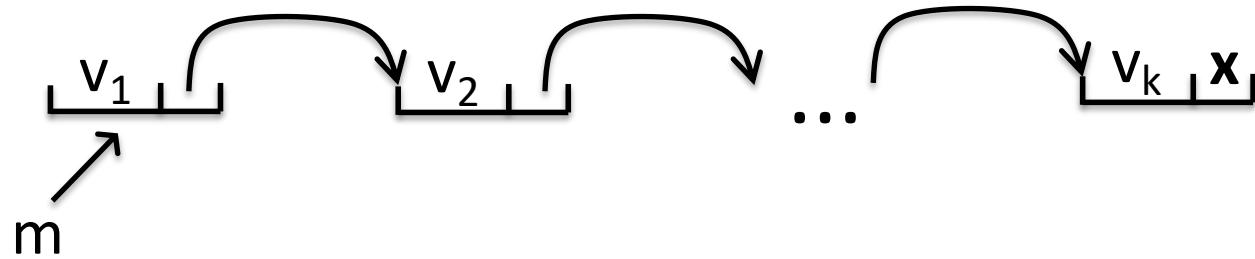
copia_nodo_pila(m):



copia_nodo_pila(m):



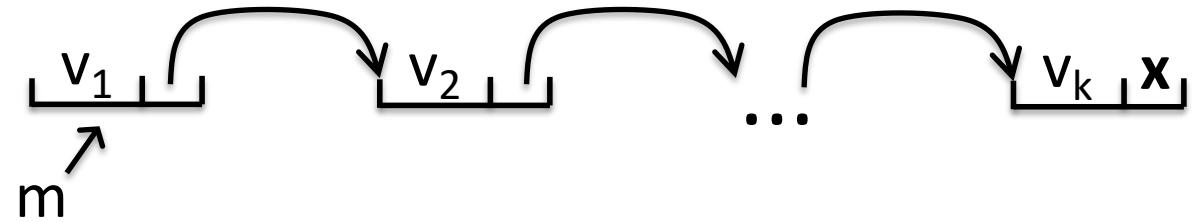
copia_nodo_pila(m):



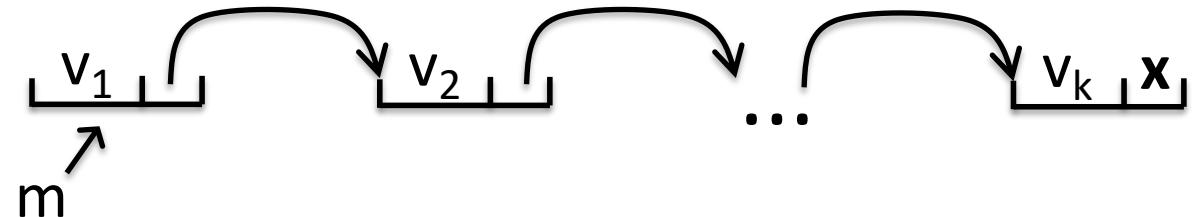
```
// Métodos privados
// Pre: true
/* Post: si m es nullptr no hace nada,
   si no libera el espacio ocupado por la cadena de
   nodos apuntada por m */

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

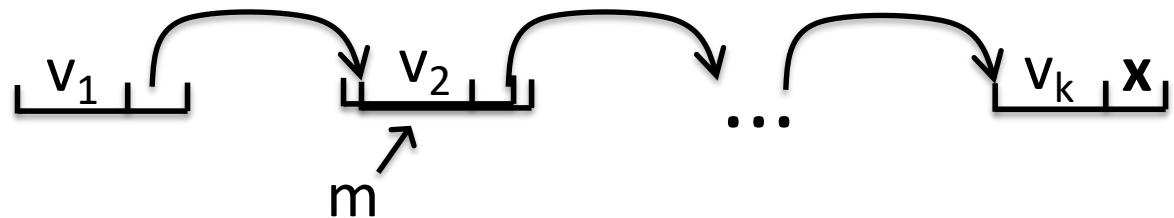
borra_n_p(m)

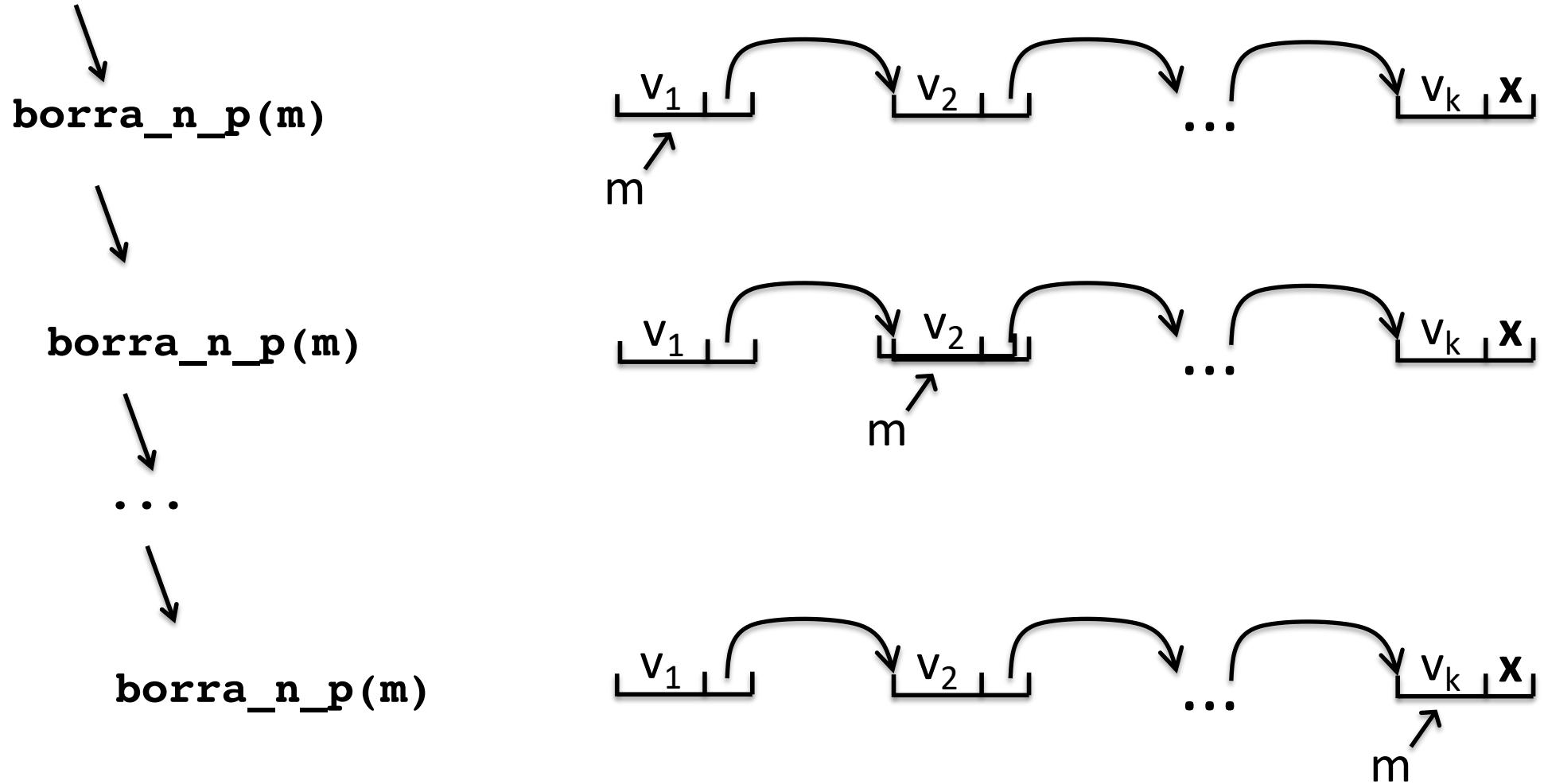


borra_n_p(m)



borra_n_p(m)





\downarrow

borra_n_p(m)

\downarrow

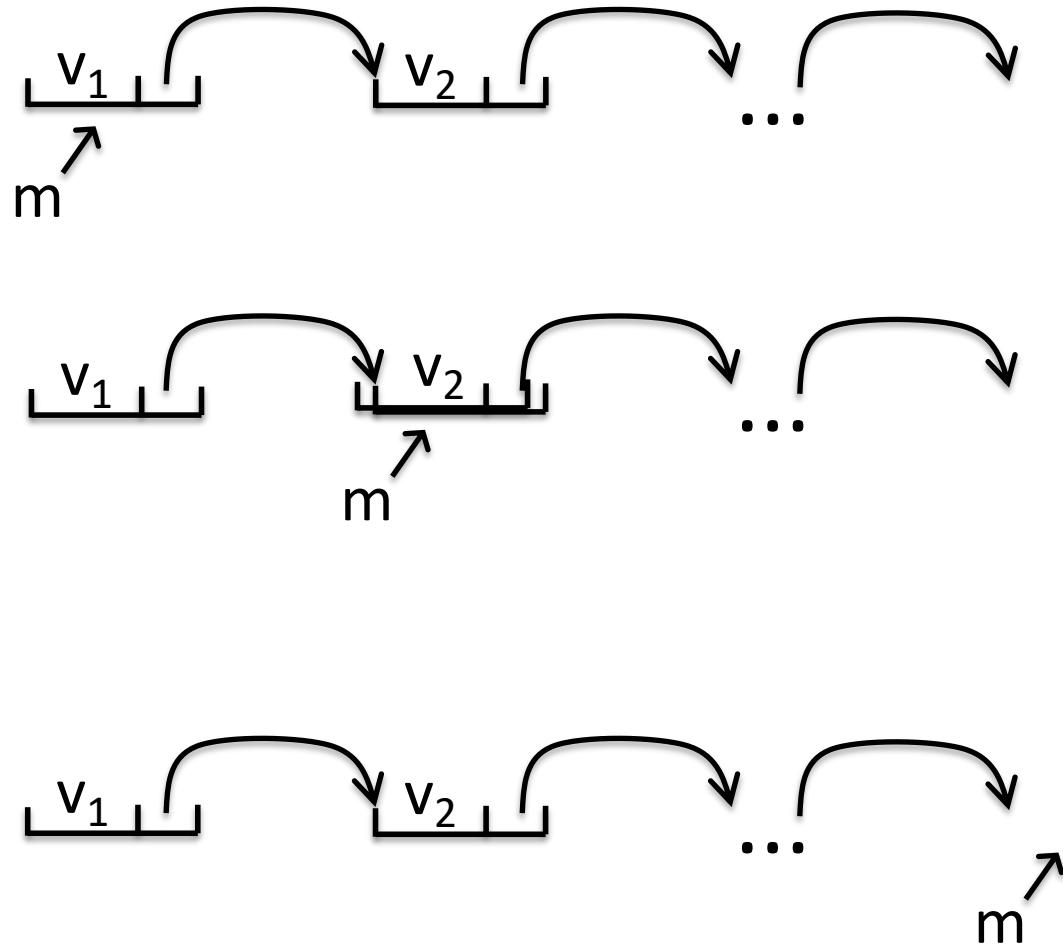
borra_n_p(m)

\dots

\downarrow

borra_n_p(m)

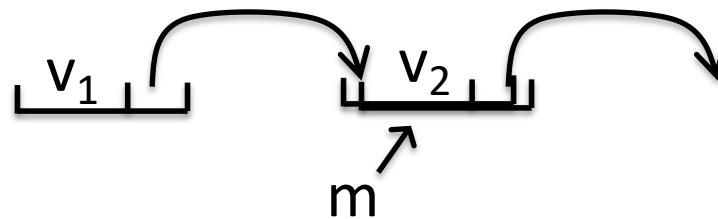
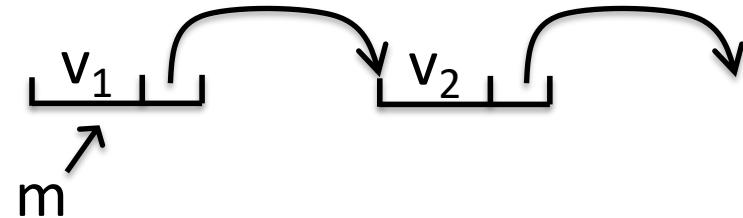
\swarrow



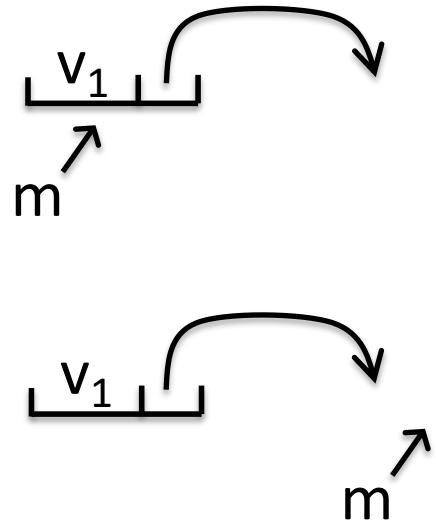
borra_n_p(m)



borra_n_p(m)



\downarrow
 $\text{borra_n_p}(m)$
 \downarrow
 $\text{borra_n_p}(m)$

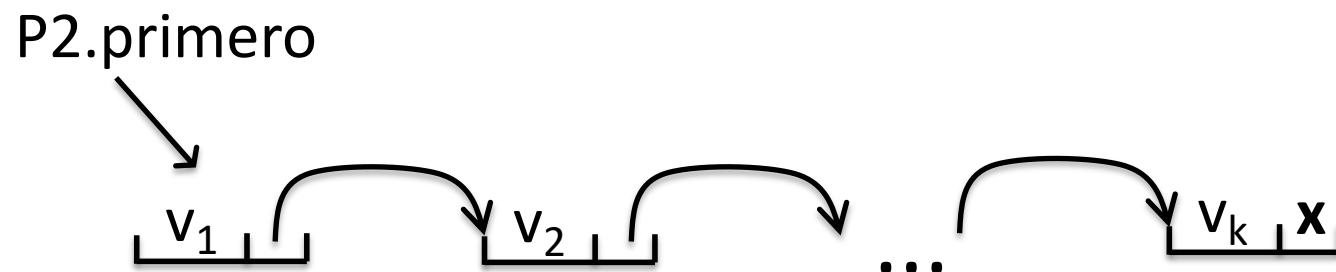


borra_n_p(m)

m

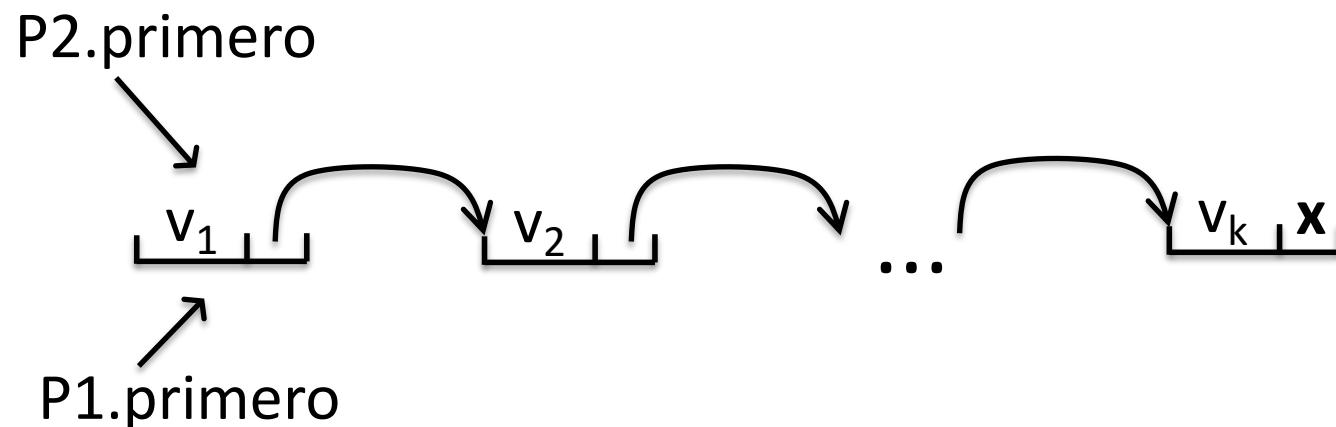
// La asignación "normal"

P1 = P2



// La asignación "normal"

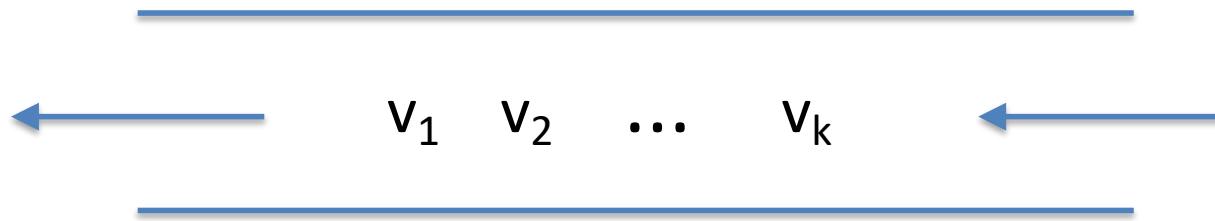
P1 = P2



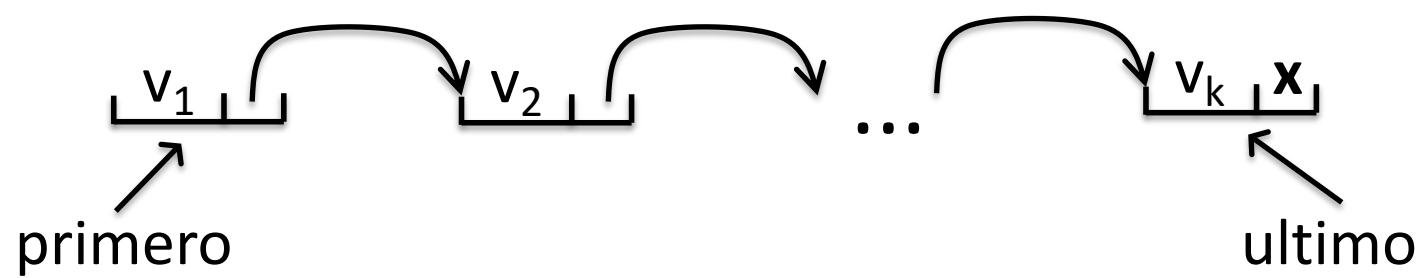
// La asignación

```
stack& operator=(const stack& S){  
    if (this != &S) {  
        altura = S.altura;  
        borra_nodo_pila(primer);  
        primer = copia_nodo_pila(S.primer);  
    }  
    return *this;  
}
```

Colas



≡



Implementación de colas

```
template <class T> class queue {  
    private:  
        // tipo privado nuevo  
        struct nodo_colas{  
            T info;  
            nodo_colas* sig;  
        };  
        int longitud;  
        nodo_colas* primero;  
        nodo_colas* ultimo;  
        ... //operaciones privadas  
    public:  
        ... //operaciones públicas  
}
```

// Constructores y destructores

```
queue(){  
    longitud = 0;  
    primero = nullptr;  
    ultimo = nullptr;  
}  
  
queue(const queue& C){  
    longitud = C.longitud;  
    primero = copia_nodo_colas(C.primer, ultimo);  
}  
  
~queue(){  
    borra_nodo_colas(primer);  
}
```

// Consultoras

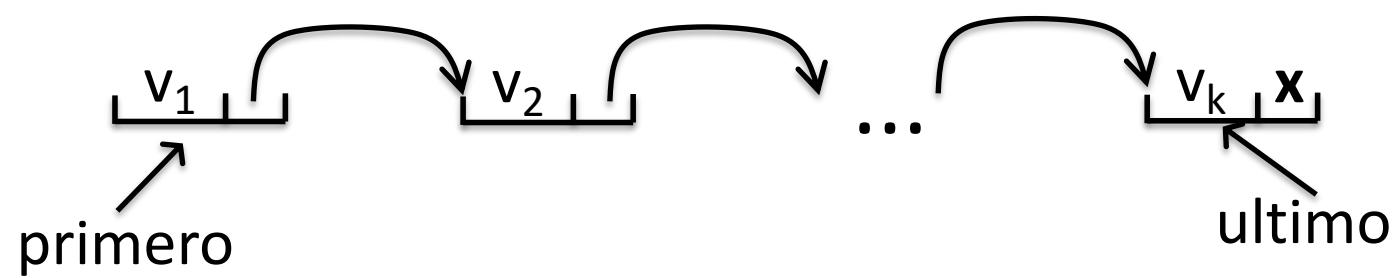
```
T front() const {
    // Pre: la cola no está vacía
    return primero->info;
}

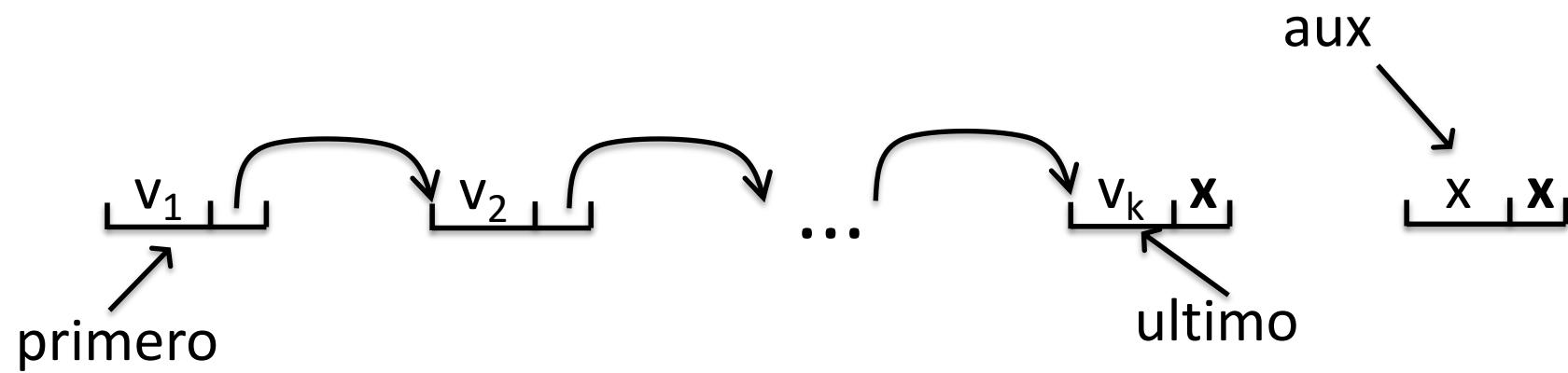
bool empty() const {
    return longitud == 0;
}

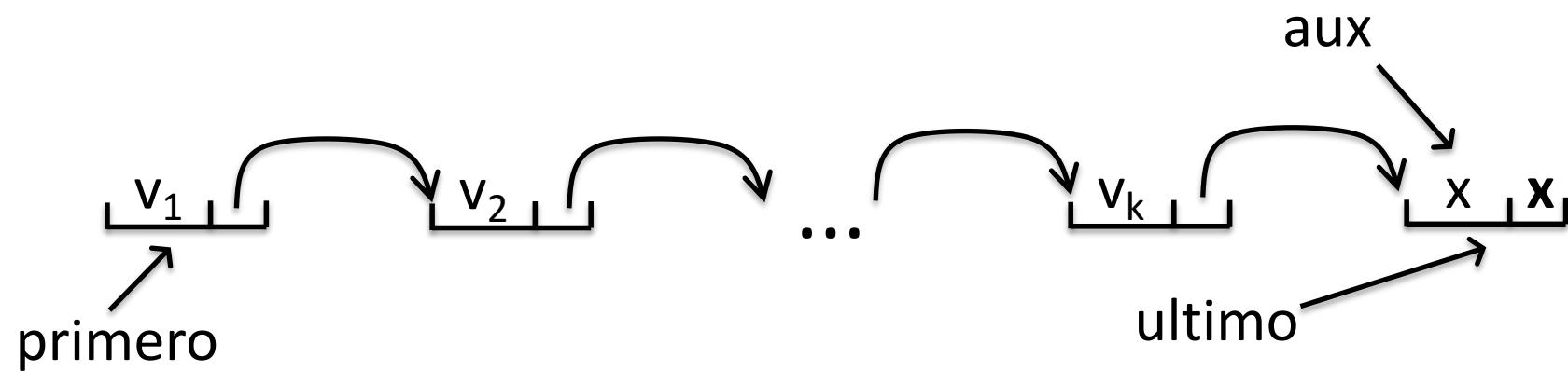
int size() const {
    return longitud;
}
```

// Modificadoras

```
void clear(){
    borra_nodo_col(a);
    longitud = 0;
    primero = nullptr;
    ultimo = nullptr;
}
void push(const T& x){
    nodo_col * aux = new nodo_col;
    aux->info = x;
    aux->sig = nullptr;
    if (primero == nullptr) primero = aux;
    else ultimo->sig = aux;
    ultimo = aux; ++longitud;
}
```

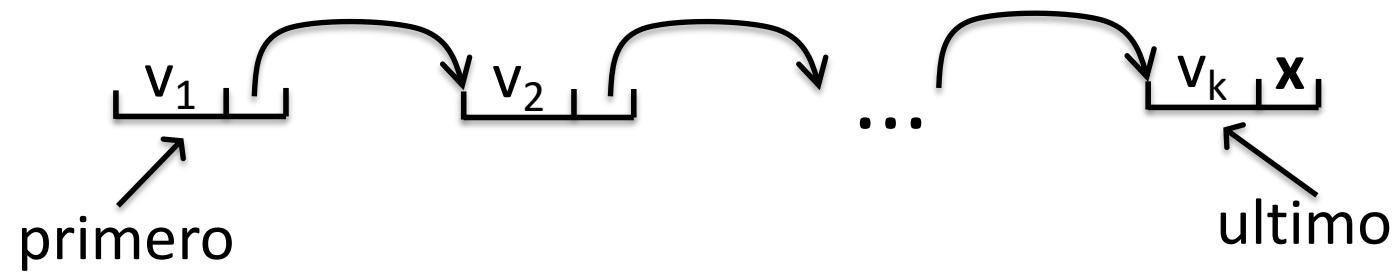


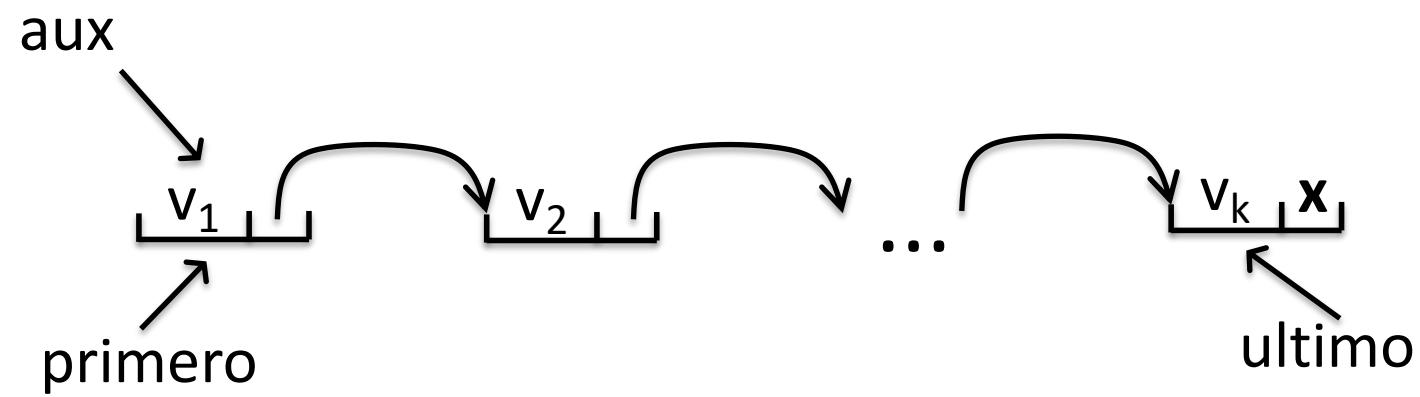


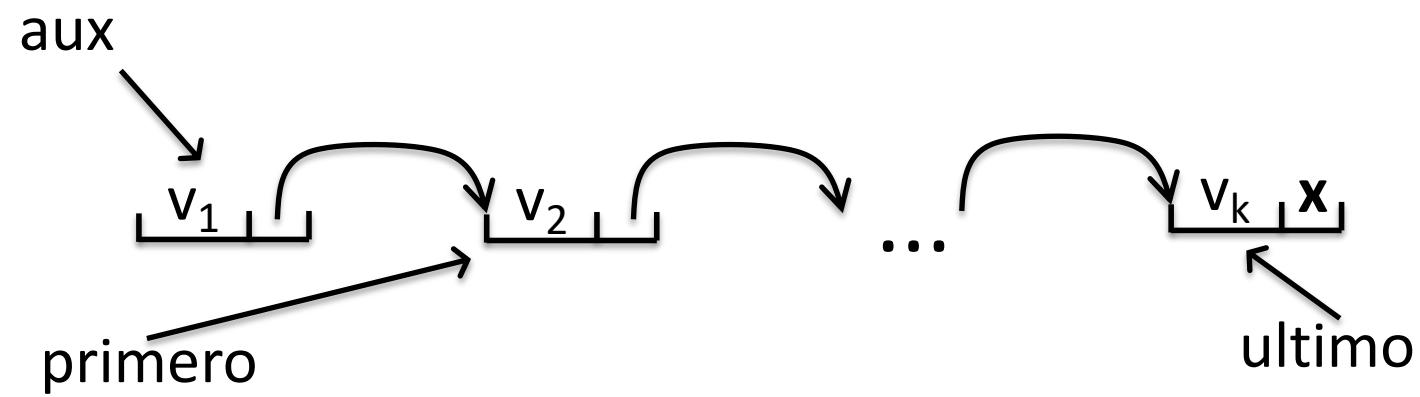


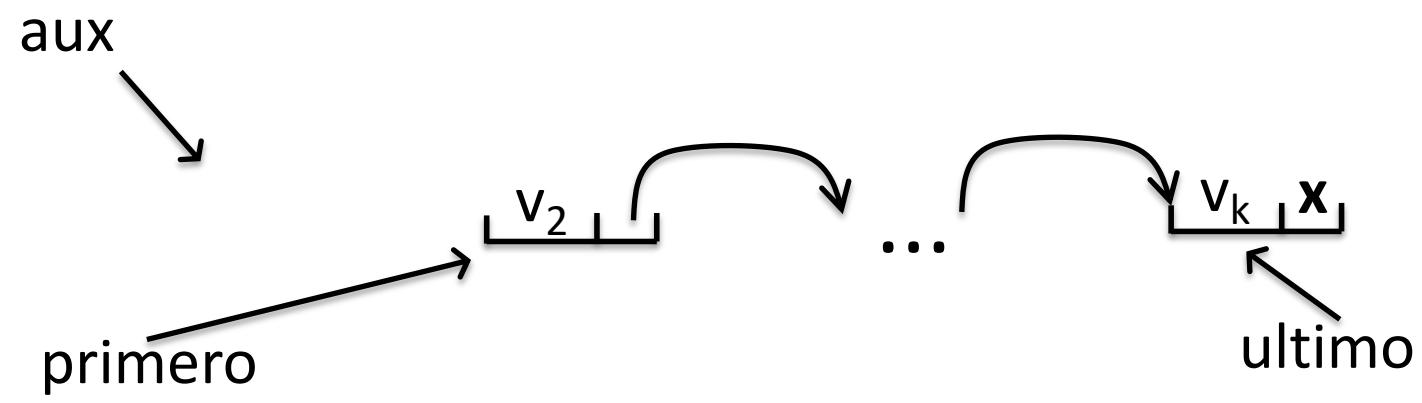
// Modificadoras

```
void pop(){
    // Pre: la cola no está vacía
    nodo_colas * aux = primero;
    if (primero->sig == nullptr) {
        primero = nullptr;
        ultimo = nullptr;
    }
    else
        primero = primero->sig;
    delete aux;
    --longitud;
}
```

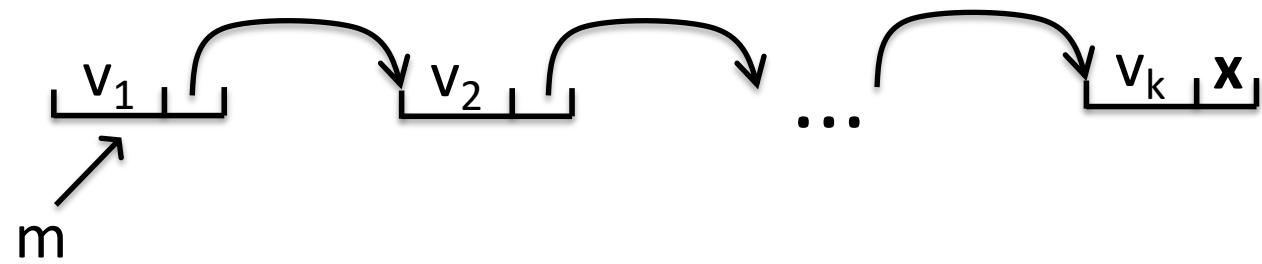


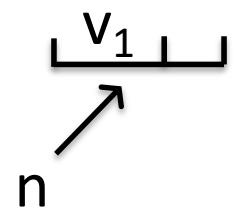
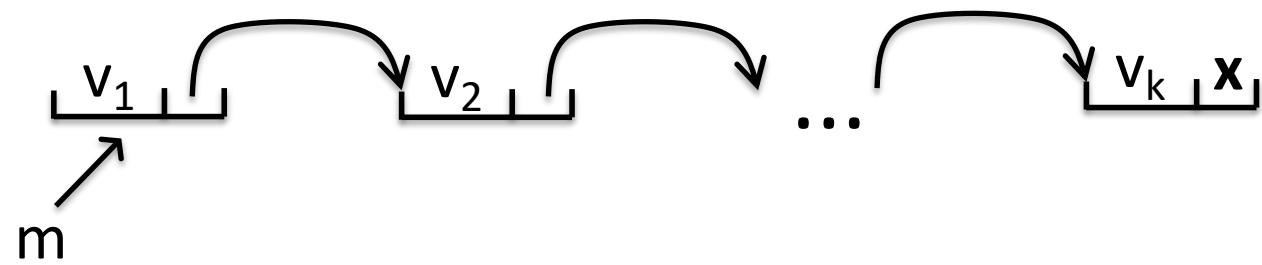


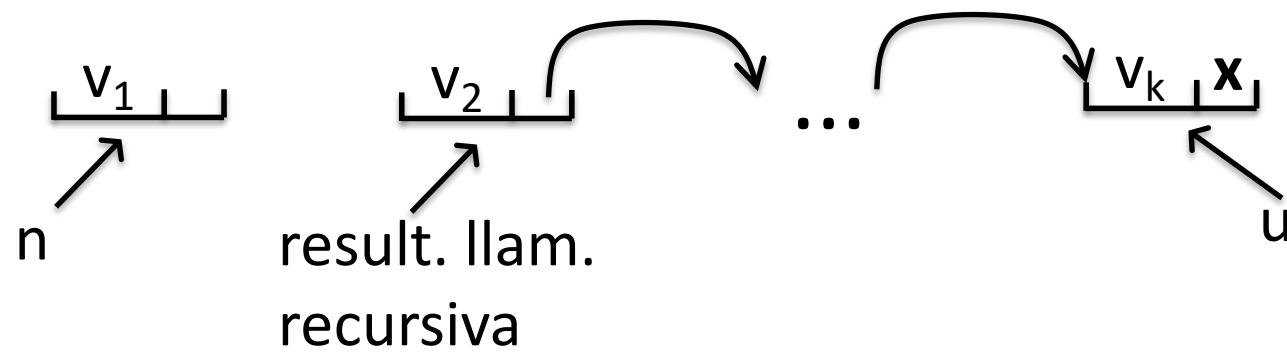
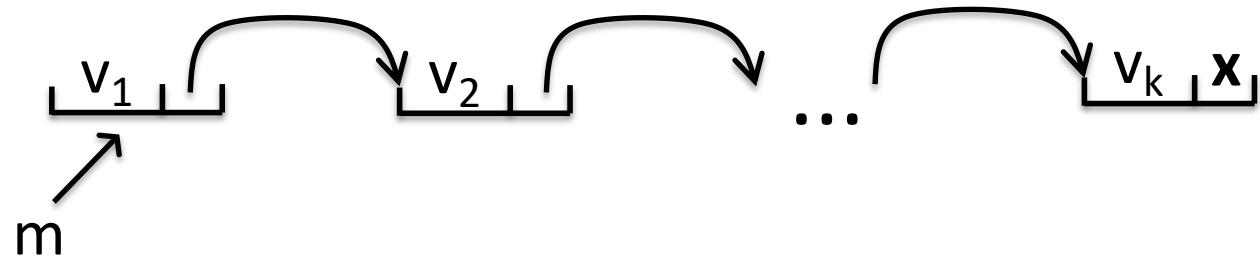


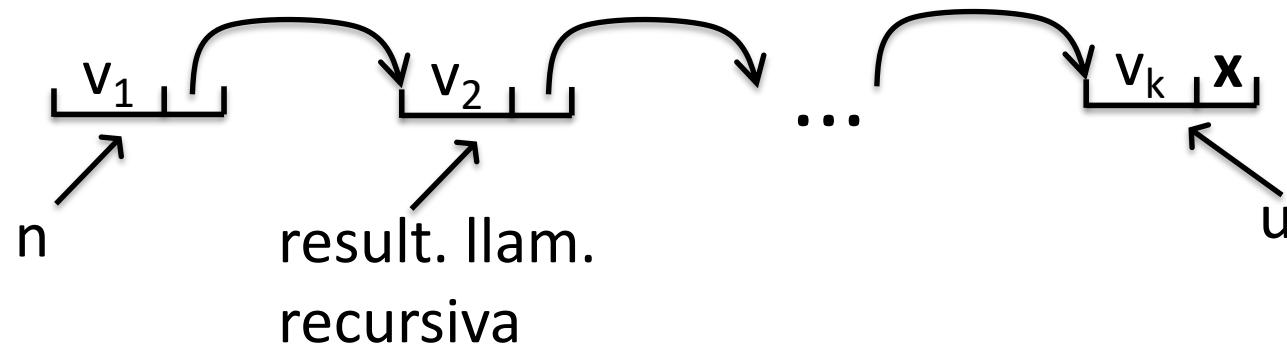
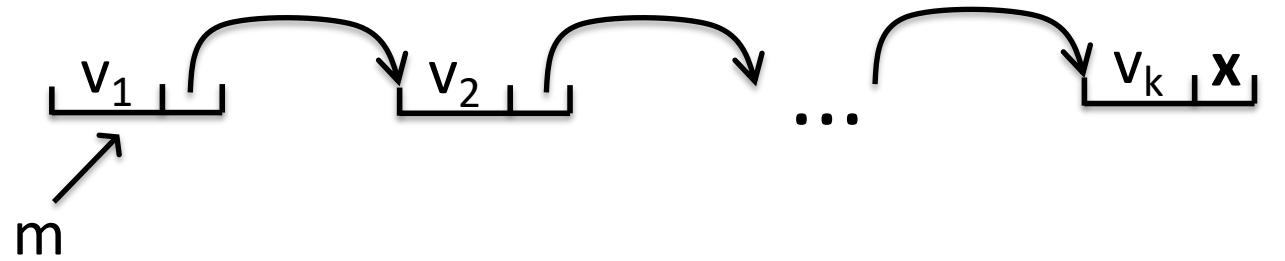


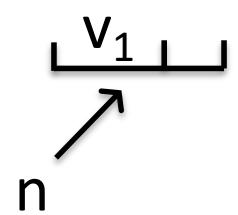
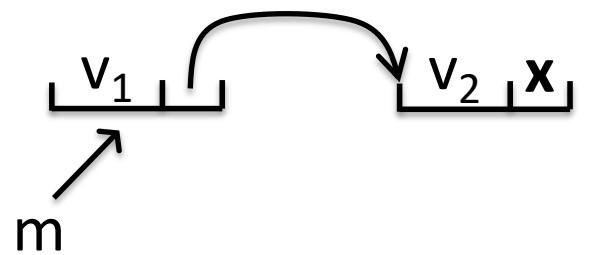
```
// Métodos privados
// Pre: true
/* Post: si m es nullptr el resultado y u son nullptr,
   si no, el resultado apunta a una cadena de nodos
   que es una copia de la cadena apuntada por m y u
   apunta al último nodo*/
static nodo_cola* copia_nodo_cola(nodo_cola* m,
                                    nodo_cola* &u){
    if (m == nullptr) {u = nullptr; return nullptr; }
    else { nodo_cola* n = new nodo_cola;
    n->info = m->info;
    n->sig = copia_nodo_cola(m->sig,u);
    if (n->sig == nullptr) u = n;
    return n;
}
```

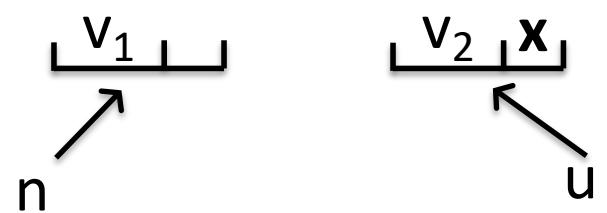
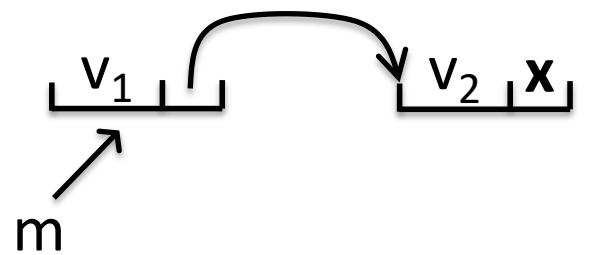


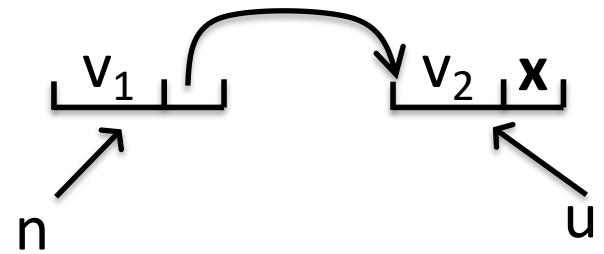
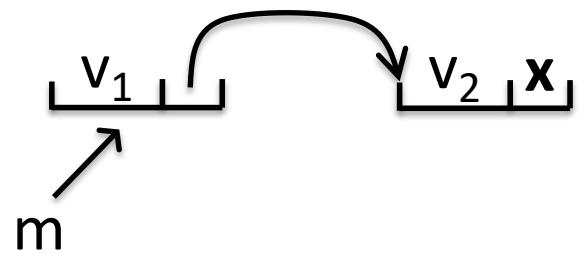












```
// Métodos privados
// Pre: true
/* Post: si m es nullptr no hace nada,
   si no libera el espacio ocupado por la cadena de
   nodos apuntada por m */

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

// La asignación

```
queue& operator=(const queue& Q){  
    if (this != &Q) {  
        longitud = Q.longitud;  
        borra_nodo_cola(primer);  
        primero = copia_nodo_cola(Q.primer, ultimo);  
    }  
    return *this;  
}
```

Listas

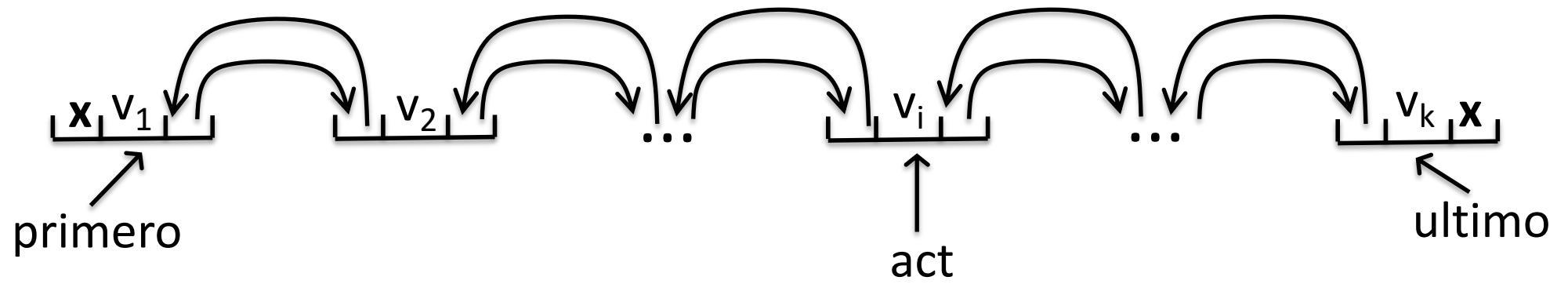
Listas

- En este curso no implementaremos los iteradores de manera general.
- Implementaremos *listas con punto de interés*, que tienen funcionalidades similares, pero algunas restricciones.

Listas con punto de interés

Podemos:

- Desplazar adelante y atrás el punto de interés
- Añadir y eliminar en el punto de interés
- Consultar y modificar el elemento en el punto de interés



Implementación de Lista

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

```
// Constructores y destructores

Lista(){
    longitud = 0;
    primero = nullptr;
    ultimo = nullptr;
    act = nullptr;
}

Lista(const Lista& L){
    longitud = L.longitud;
    primero = copia_nodo_lista(L.primer, L.act,
                                ultimo, act);
}

~Lista(){
    borra_nodo_lista(primer);
}
```

```
// 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 == nullptr;  
}
```

```
int al_principio() const {  
    return act == primero;  
}
```

// Modificadoras

```
void l_vacia(){
    borra_nodo_lista(primer);
    longitud = 0;
    primero = nullptr;
    ultimo = nullptr;
    act = nullptr;
}
```

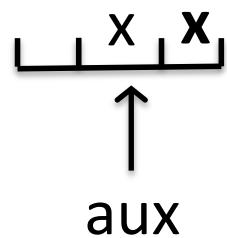
```
// Inserción
// Pre: true
/* Post: Se ha añadido un nodo con el valor x antes
del punto de interés que sigue siendo el mismo que
antes de la operación*/
void añadir(const T& x){
    nodo_lista * aux = new nodo_lista;
    aux->info = x; aux->sig = act;
    if (longitud == 0) {
        aux->ant = nullptr;
        primero = aux; ultimo = aux;
    } else if (act == nullptr) {
        aux->ant = ultimo;
        ultimo->sig = aux;
        ultimo = aux;
    }
    ...
}
```

(Continuación)

```
else if (act == primero) {  
    aux->ant = nullptr;  
    act->ant = aux;  
    primero = aux;  
} else {  
    aux->ant = act->ant;  
    (act->ant)->sig = aux;  
    act->ant = aux;  
}  
++longitud;  
}
```

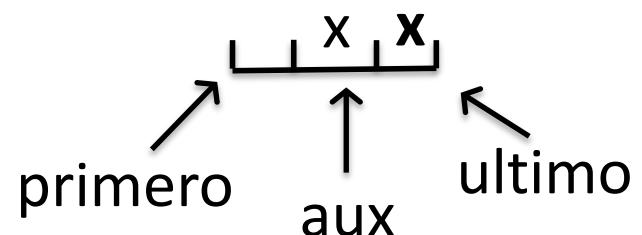
// Caso longitud == 0 – inserción en lista vacía

primero = ultimo = act = nullptr

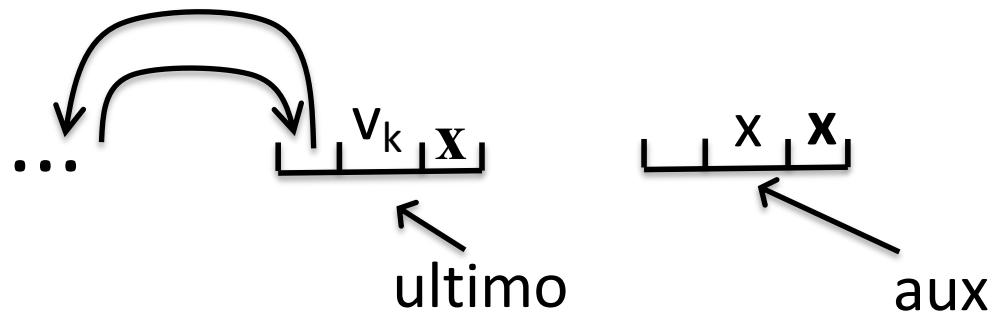


// Caso longitud == 0 – inserción en lista vacía

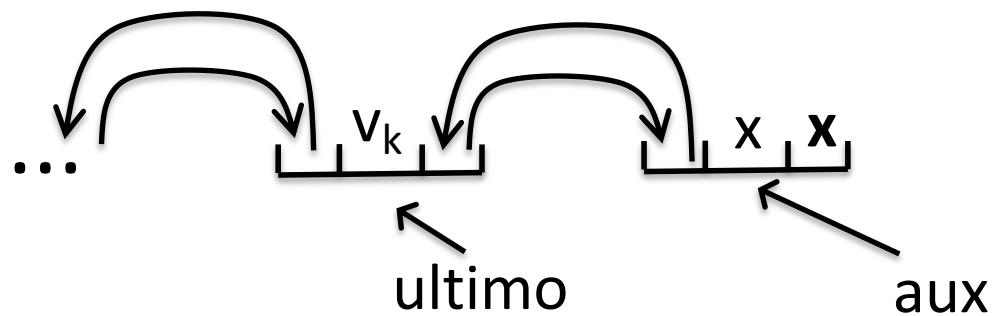
act = nullptr



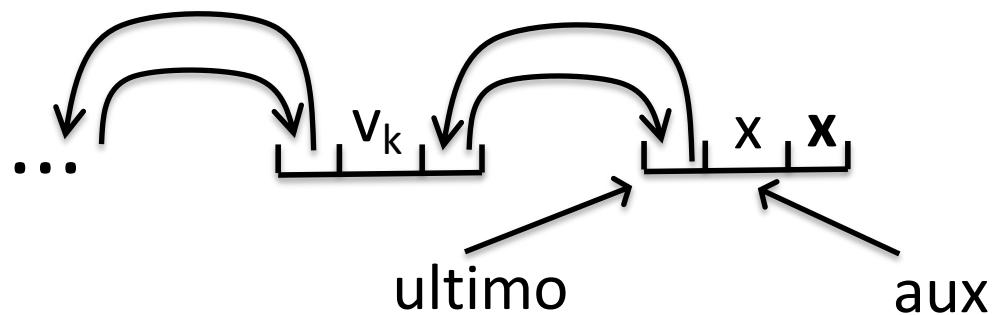
// Caso act == nullptr – inserción al final



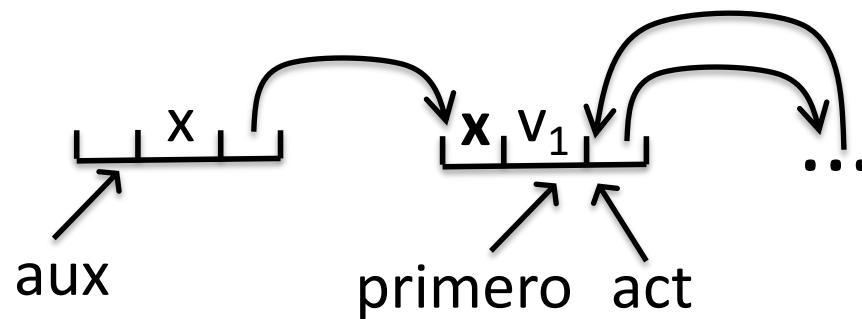
// Caso act == nullptr – inserción al final



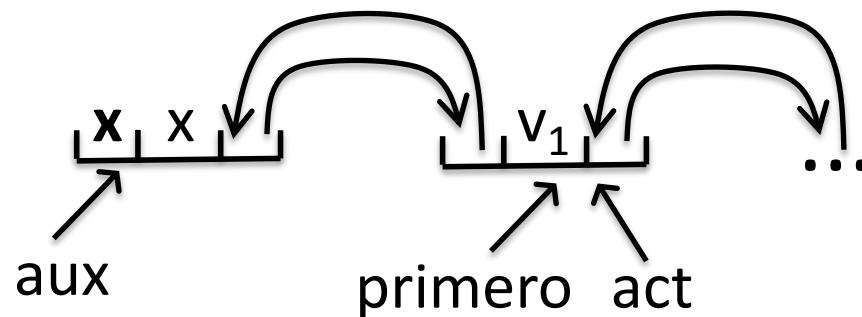
// Caso act == nullptr – inserción al final



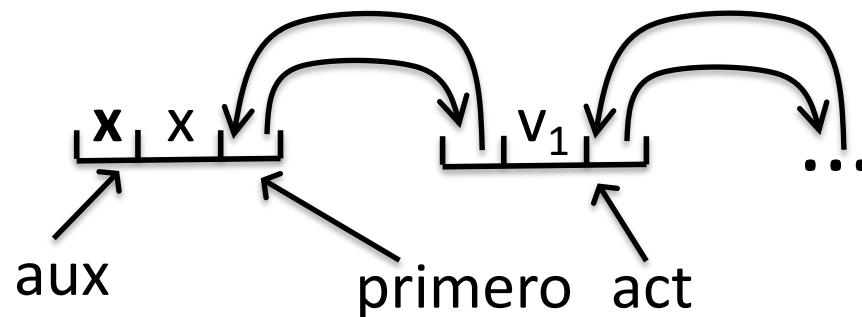
// Caso $\text{act} == \text{primero}$ – inserción al principio



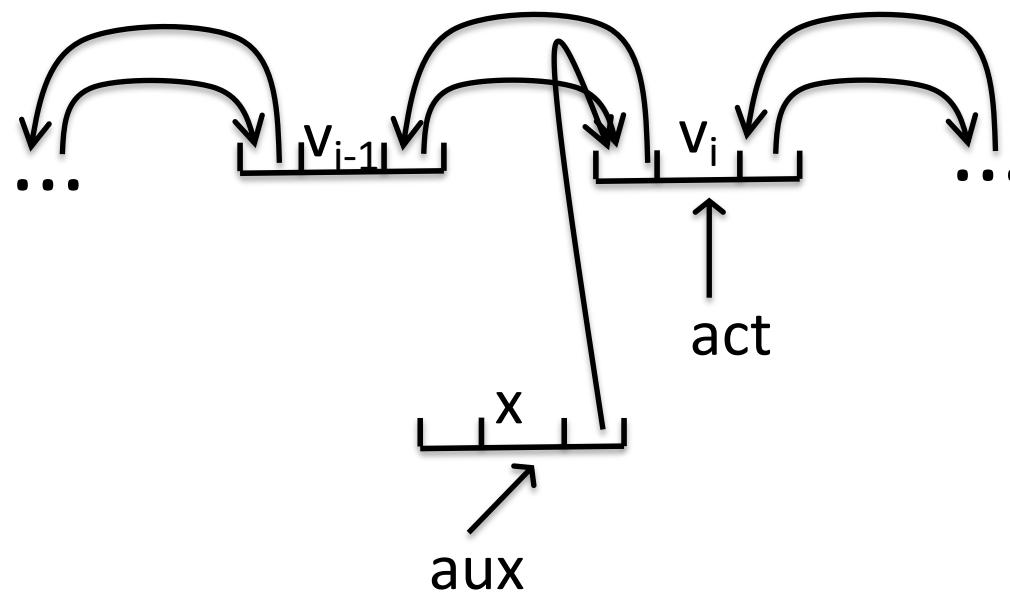
// Caso $\text{act} == \text{primero}$ – inserción al principio



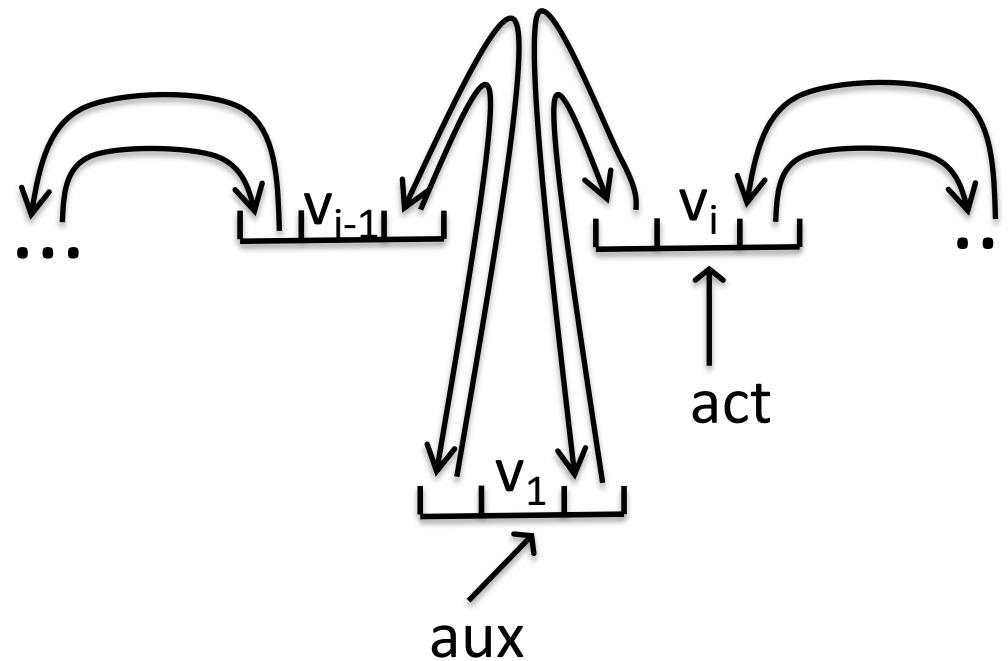
// Caso `act == primero` – inserción al principio



// Caso inserción en medio



// Caso inserción en medio



```
// 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;

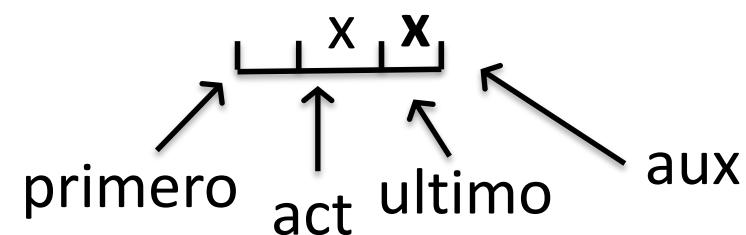
    if (longitud == 1) {
        primero = nullptr; ultimo = nullptr;
    } else if (act == primero) {
        primero = act->sig;
        primero->ant = nullptr;
    }

    ...
}
```

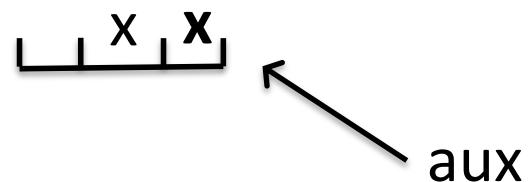
(Continuación)

```
else if (act == ultimo) {  
    ultimo = act->ant; ultimo->sig = nullptr;  
} else {  
    (act->ant)->sig = act->sig;  
    (act->sig)->ant = act->ant;  
}  
act = act->sig;  
delete aux;  
--longitud;  
}
```

// Caso longitud == 1 – lista con un nodo

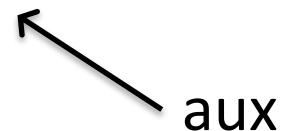


// Caso longitud == 1 – lista con un nodo



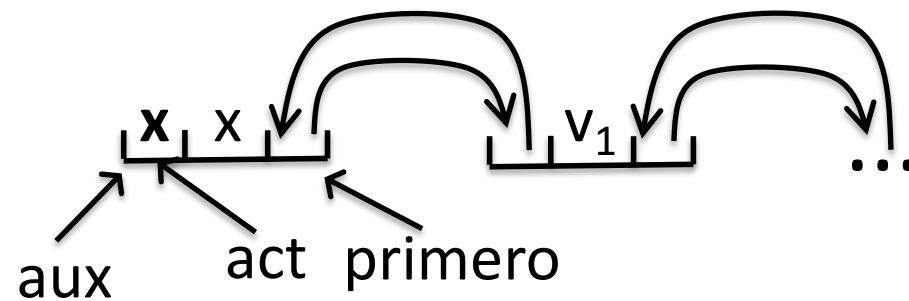
primero = ultimo = act = nullptr

// Caso longitud == 1 – lista con un nodo

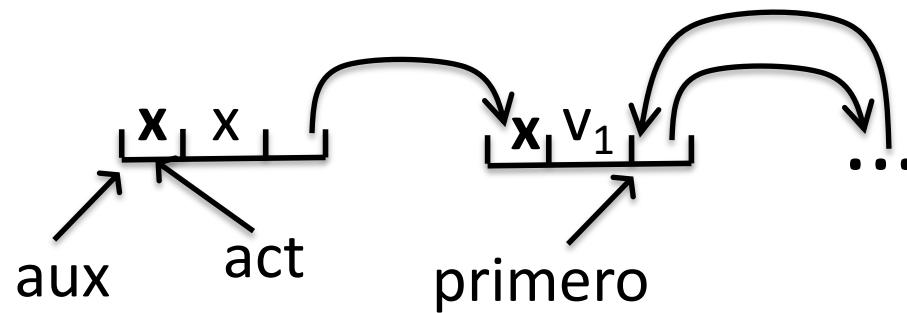


primero = ultimo = act = nullptr

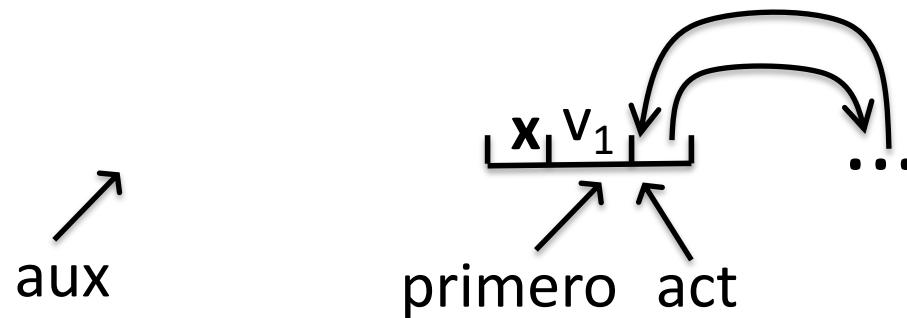
// Caso $\text{act} == \text{primero}$ – eliminación al principio



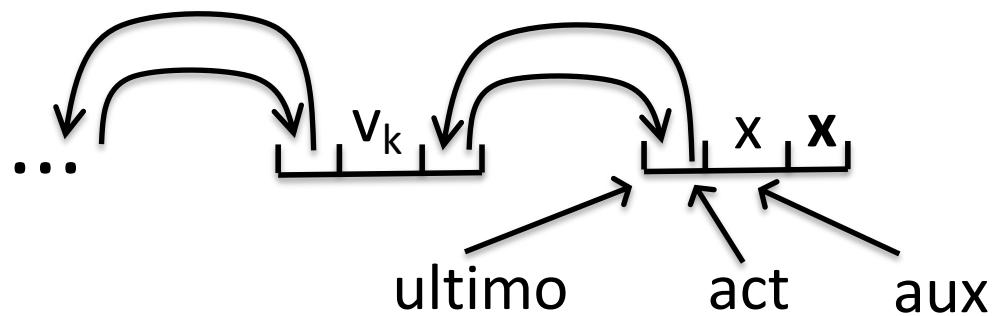
// Caso `act == primero` – eliminación al principio



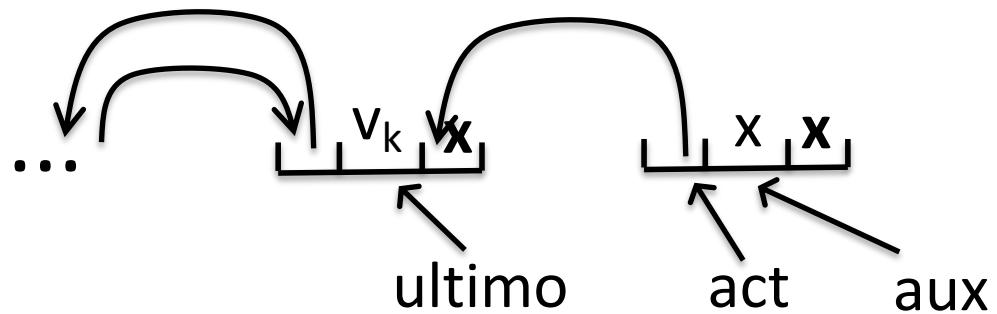
// Caso `act == primero` – eliminación al principio



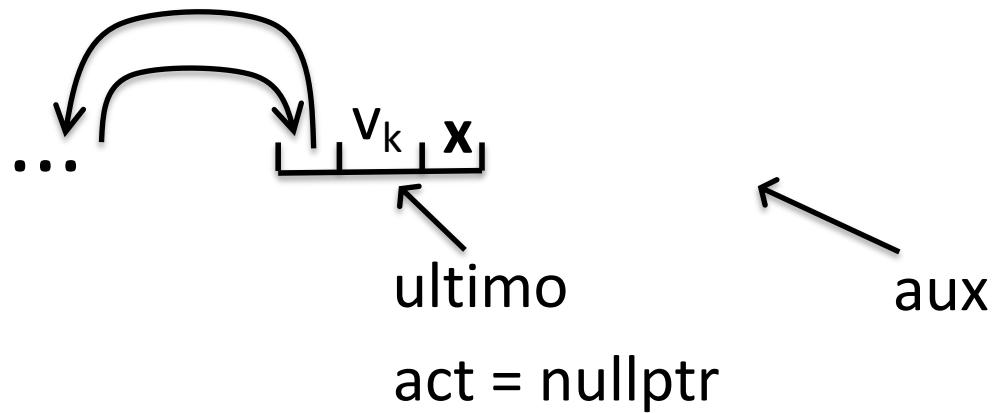
// Caso $act == \text{ultimo}$ – eliminación al final



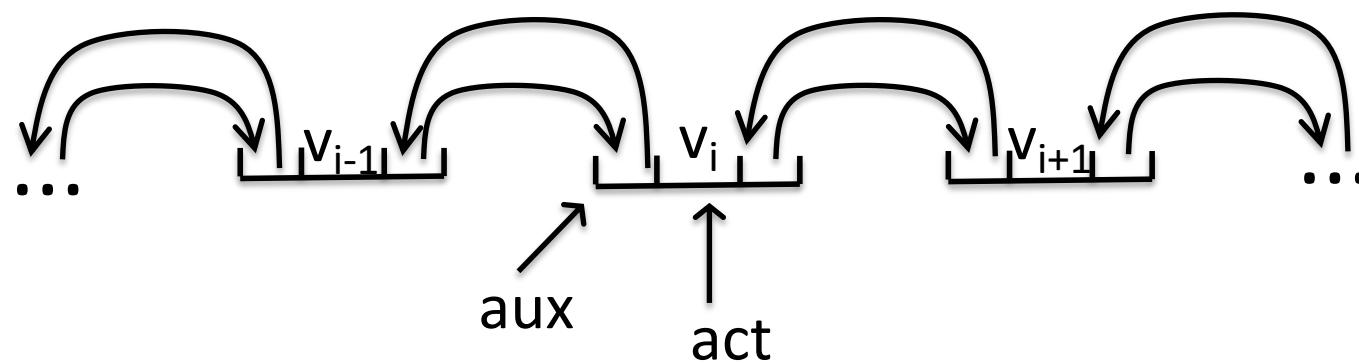
// Caso act == ultimo – eliminación al final



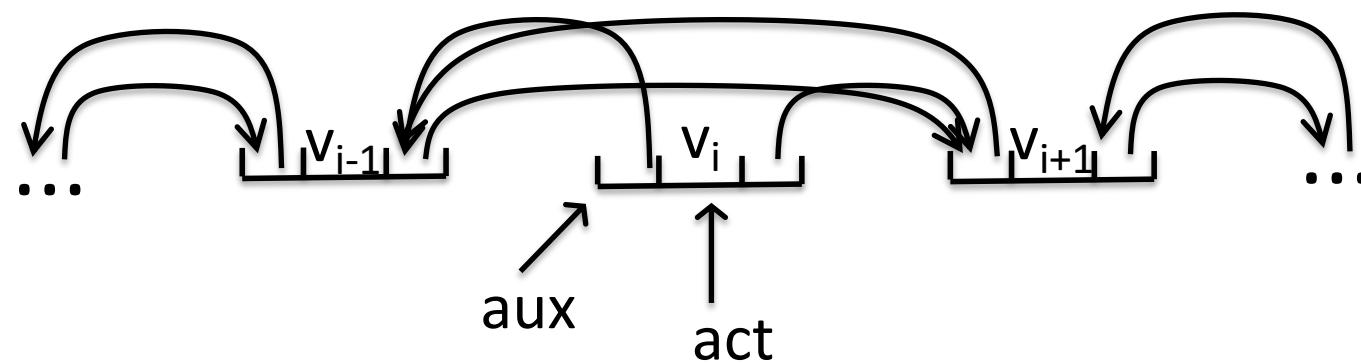
// Caso act == ultimo – eliminación al final



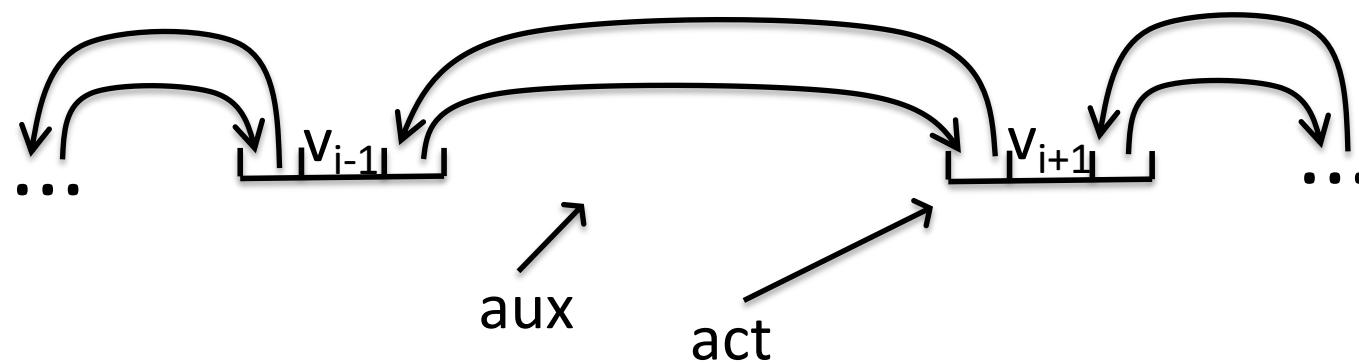
// Caso eliminación en medio



// Caso eliminación en medio



// Caso eliminación en medio



```
// modificación y movimiento del punto de interés

/* Pre: La lista no está vacía y el punto de interés no
   es nullptr */
/* Post: Se ha reemplazado el valor del punto de interés por
   x */

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

/* Pre: true */
/* Post: Se ha movido el punto de interés al principio de la
   lista */

void inicio(){
    act = primero;
}
```

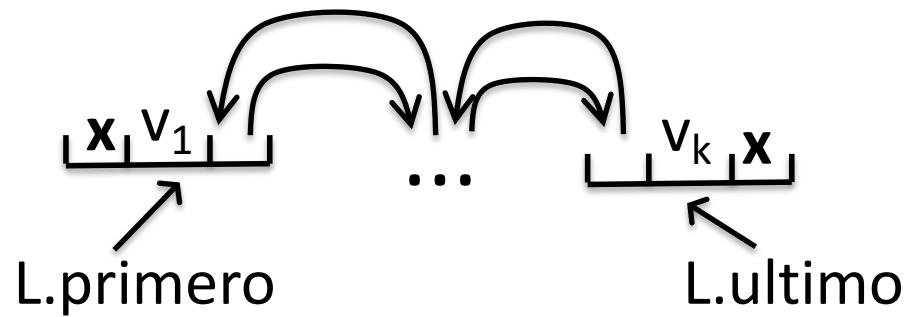
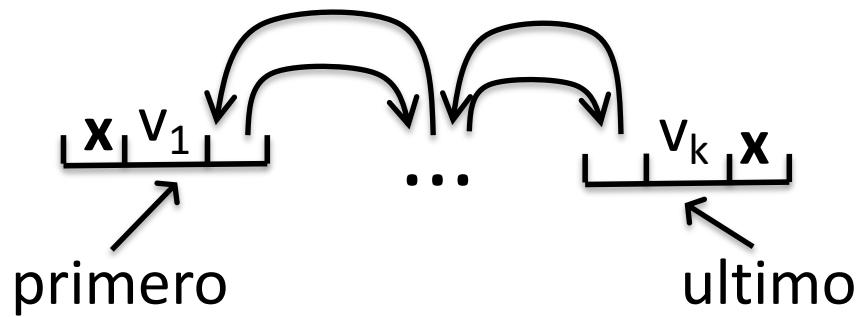
```
/* Pre: true */
/* Post: Se ha movido el punto de interés al final de la
lista */
void fin(){
    act = nullptr;
}

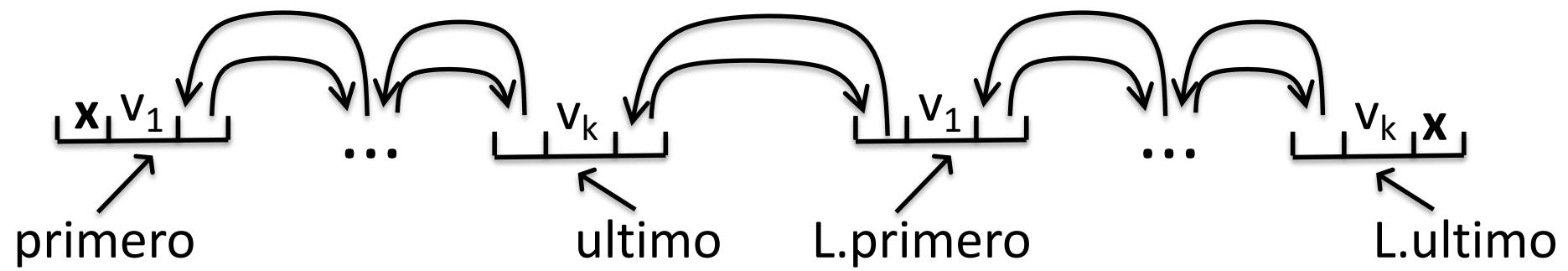
/* Pre: La lista no está vacía y el punto de interés no
es nullptr */
/* Post: Se ha movido el punto de interés una posición hacia
el final */
void avanza(){
    act = act->sig;
}
```

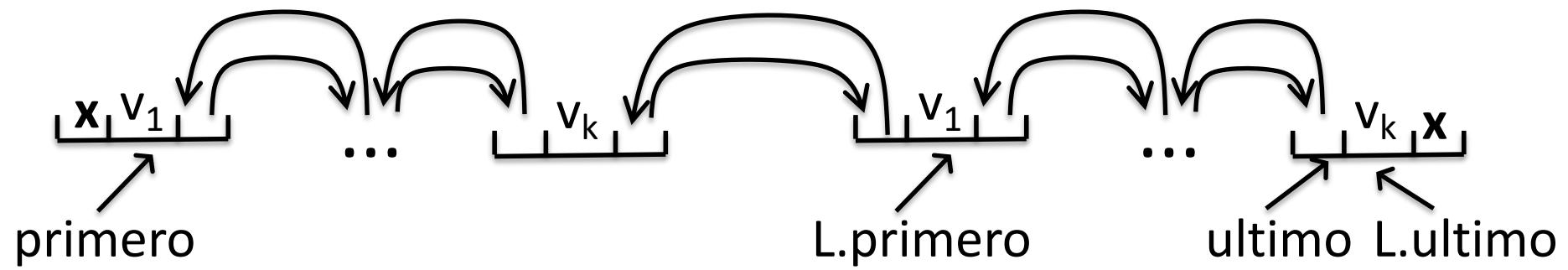
```
/* Pre: La lista no está vacía y el punto de interés no
   es el primer elemento */
/* Post: Se ha movido el punto de interés una posición hacia
   el principio */

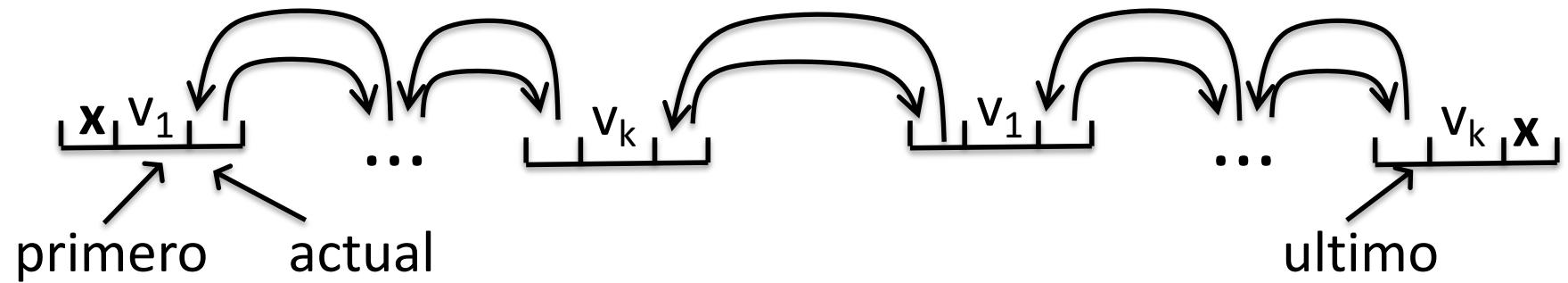
void retrocede(){
    if (act == nullptr) act = ultimo;
    else act = act->ant;
}
```

```
// 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;
}
```





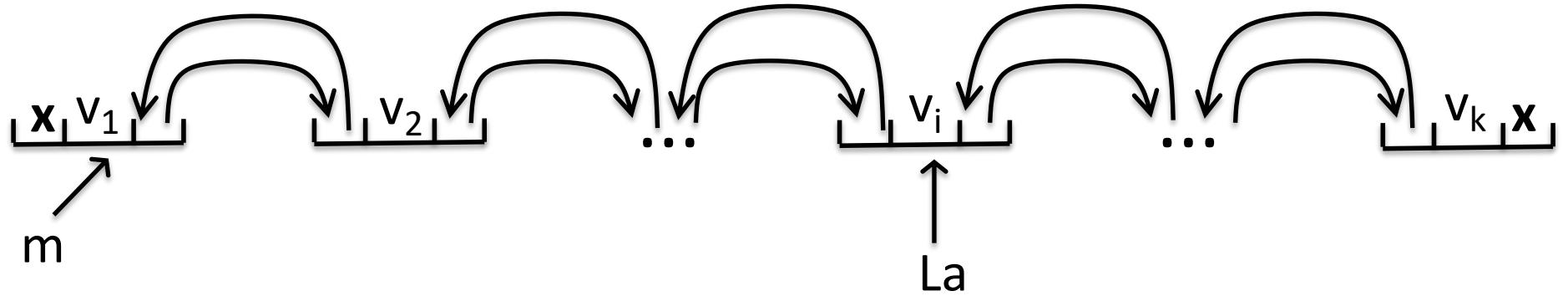


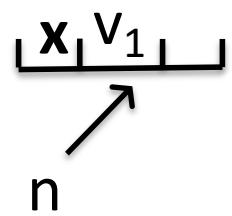
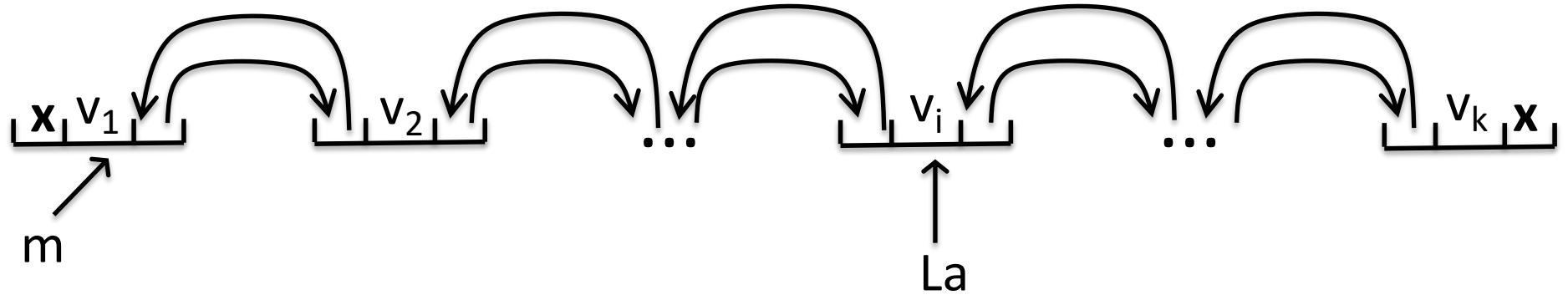


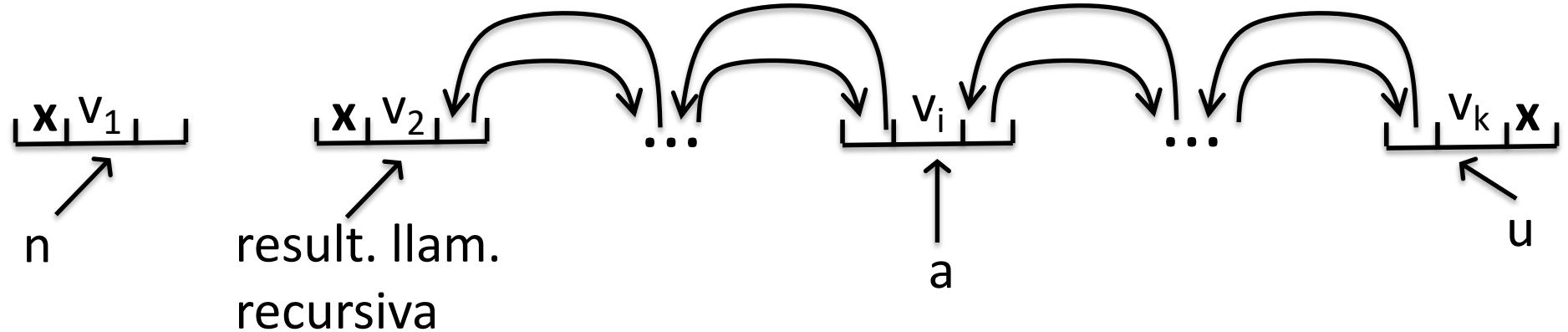
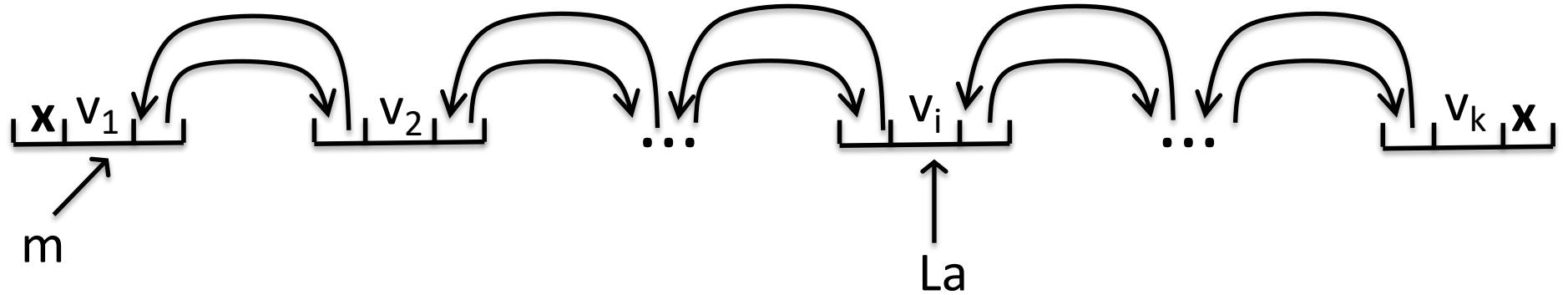
$L.\text{primero} = L.\text{ultimo} = L.\text{act} = \text{nullptr}$

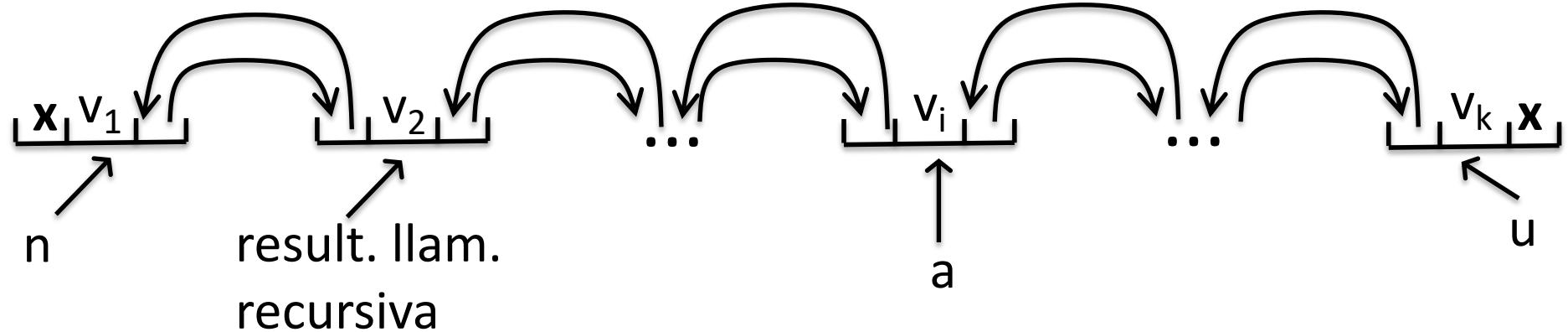
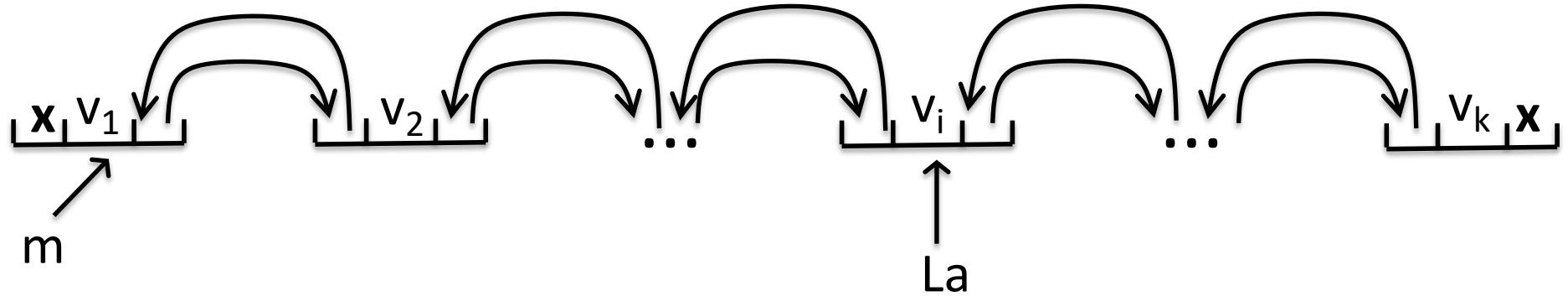
```
// Métodos privados
// Copiar secuencia de nodos
static nodo_Lista* copia_nodo_Lista (
    nodo_Lista* m, nodo_Lista* La,
    nodo_Lista* &u, nodo_Lista* &a);
// Pre: true
/* Post: si m es nullptr el resultado, u y a son nullptr,
si no, el resultado apunta a una cadena de nodos
que es una copia de 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_list* n = new nodo_list;
    n->info = m->info;
    n->ant = nullptr;
    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;
}
}
```









```
// 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;  
}
```