Programación 2
Tipos recursivos de datos I

Fernando Orejas

Transparencias basadas en las de Ricard Gavaldà

1. Punteros y memoria dinámica
2. Conceptos básicos sobre los tipos recursivos de datos
3. Pilas
4. Colas
5. Listas
6. Árboles
7. Colas con prioridad

Punteros y memoria dinámica

Tipos recursivos de datos

Un tipo recursivo de datos es un tipo en el que, en su definición, hacemos referencia al propio tipo:

```
struct nodo{
   int dni;
   string name;
   nodo* siguiente;
}
```

Punteros

En C++, para cada tipo T, hay un tipo de apuntadores a T.

Una variable de este tipo puede contener:

- Una referencia a un objeto de tipo T (estático o dinámico)
- El valor nullptr
- Cualquier cosa rara

Cómo (no) usar punteros

```
int* p;
int x;
p = &x;
x = 5
int* q = p;
*p = 3;
x = *p + 1;
p = new int;
delete p;
delete q;
```

Cómo (no) usar punteros

```
nodo* p, q, r;
p = new nodo;
q = nullptr;
p->dni = 775577;
p->name = "abc";
(p->siguiente)->name = "cba";
r = p;
delete p;
p = q;
int x = r->dni;
nodo** p1;
```

Cómo (no) usar punteros

```
nodo* p1;
vector <nodo*> v,u (5);
....
for (int i = 0; i < 5; ++1) u[i] = v[i];
```

Conceptos básicos sobre tipos resursivos de datos

Tipos recursivos de datos

Un tipo recursivo de datos es un tipo en el que, en su definición, hacemos referencia al propio tipo:

- Pilas: Una pila o bién es una pila vacía o es el resultado de un push sobre otra pila y un valor

  intstack = Empty | Push of int * intstack

- Colas y listas: lo mismo
- Árboles : Un árbol es o bien un árbol vacío o es el resultado de enraizar un valor con

  int inttree = Empty | Cons of int * inttree * inttree

Pilas recursivas en C++

```
class stack;
    bool vacia;
    int primero;
    stack resto;
}
```

Daría lugar a un proceso infinito

Pilas recursivas en C++

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

Definición de una estructura de datos recursiva

Clase con:

- Struct privada que define nodos enlazados con punteros. En cada nodo
  - Información de un elemento de la estructura
  - Puntero a uno o más elementos

- Atributos que contienen información global de la estructura
- Punteros a elementos distinguidos de la estructura
- Siempre asumiremos como precondición e invariante que dos estructuras de datos diferentes no comparten ningún nodo.

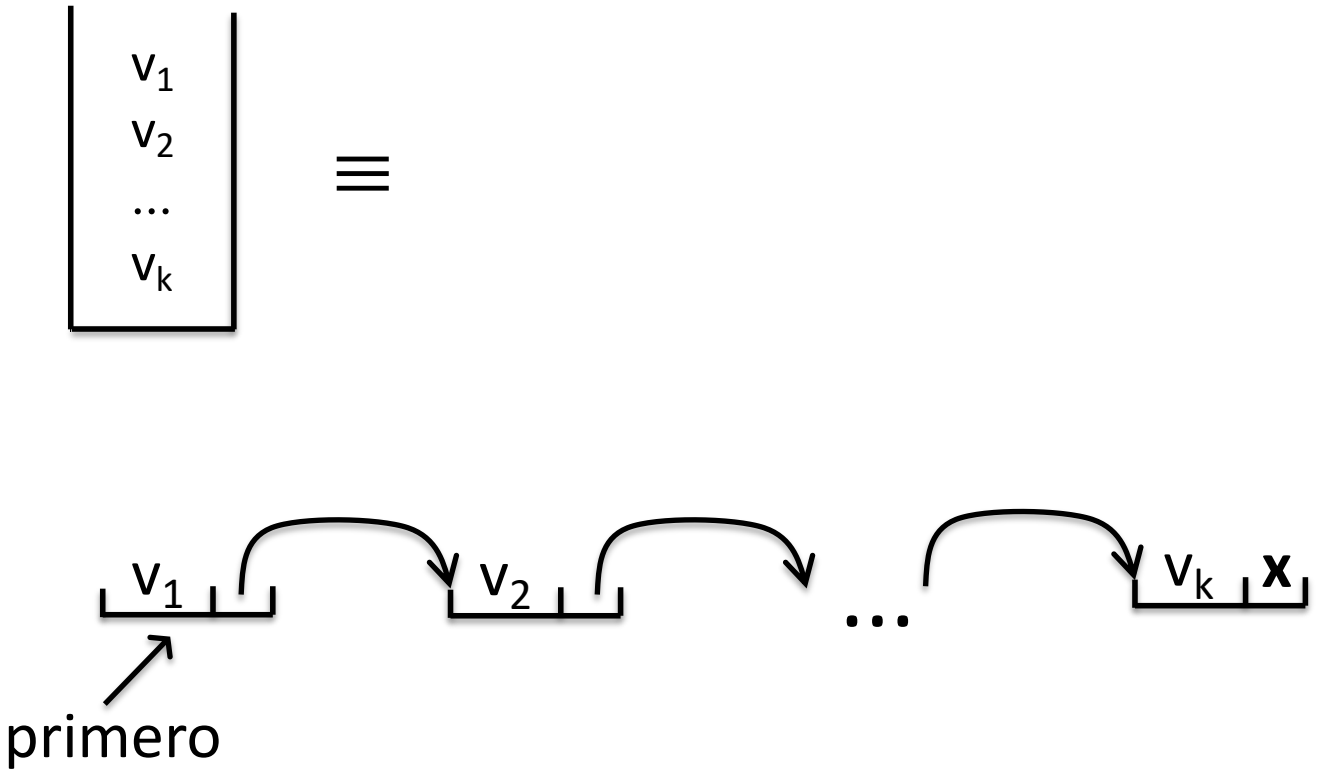
Ventajas de las estructuras de datos recursivas

- Correspondencia natural con la definición recursiva del tipo de datos
- No es necesario fijar a priori un tamaño máximo
- Se puede ir pidiendo memoria para los nodos, según se va necesitando
- Modificando los enlaces entre los nodos podemos:
  - Insertar o borrar elementos, sin tener que mover otros
  - Mover partes enteras de la estructura sin hacer copias

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
}
```

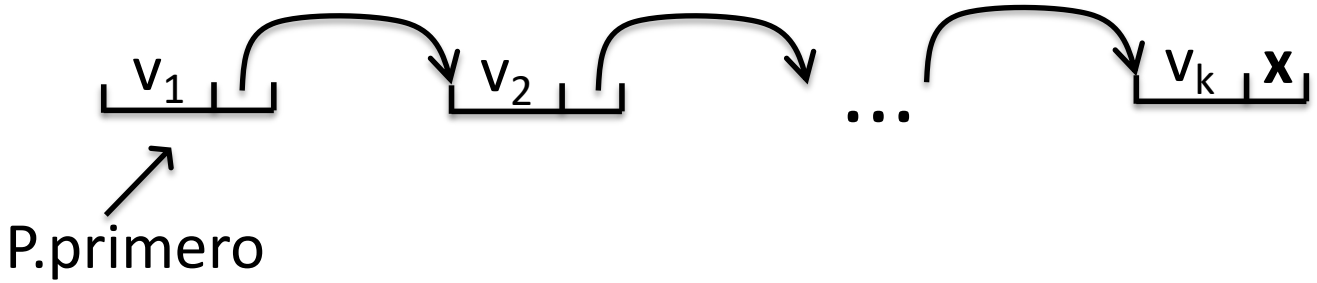

v_1
v_2
...
v_k
≡
v_1
v_2
...
v_k
x
primero

```
// Constructoras

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

    stack(const stack& P){


    }
```

```
// Constructoras
```

Si

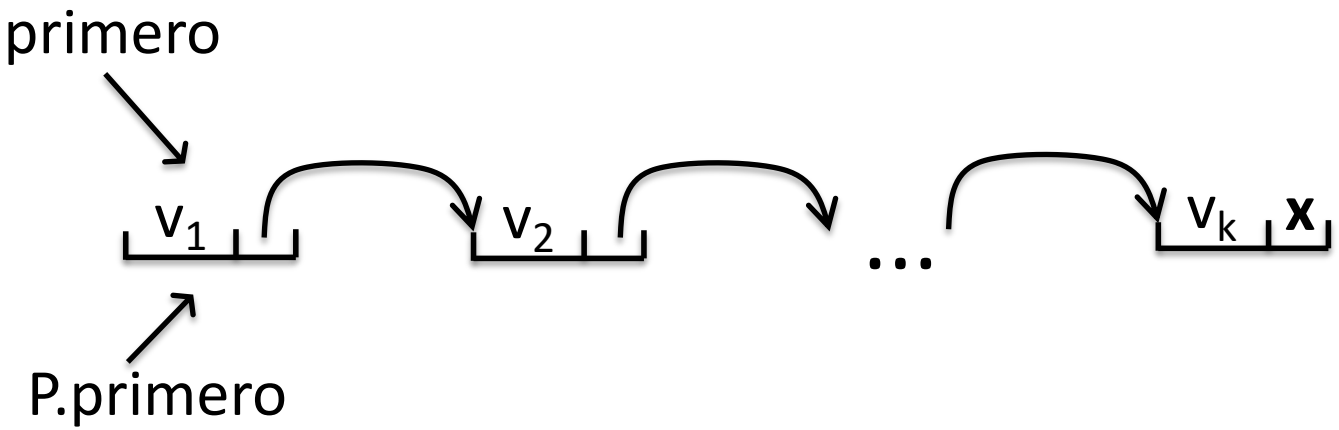
```
stack(const stack& P){
  altura = p.altura;
  primero = p.primero
}
```



// Constructoras

Si

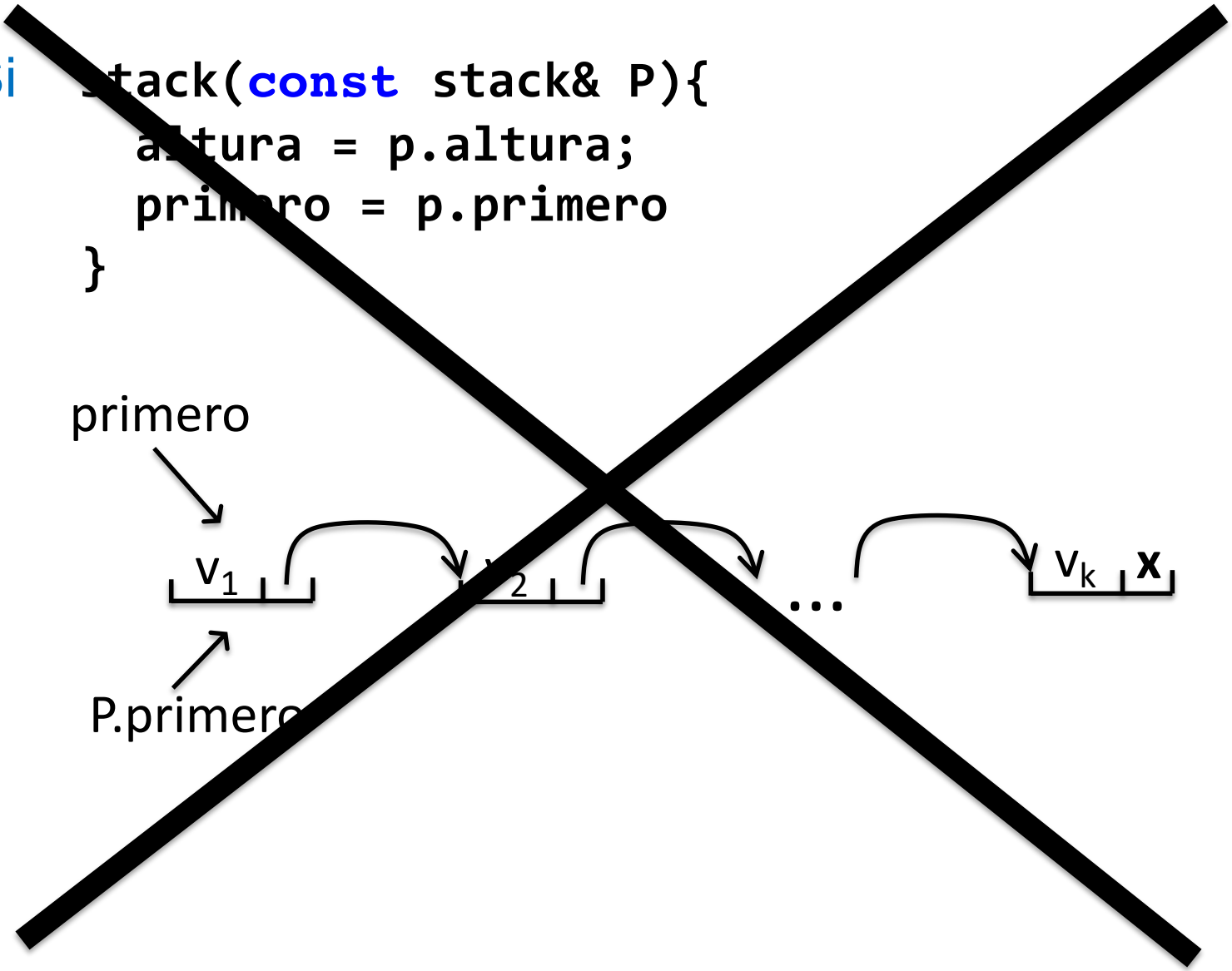
```
stack(const stack& P){
  altura = p.altura;
  primero = p.primero
}
```



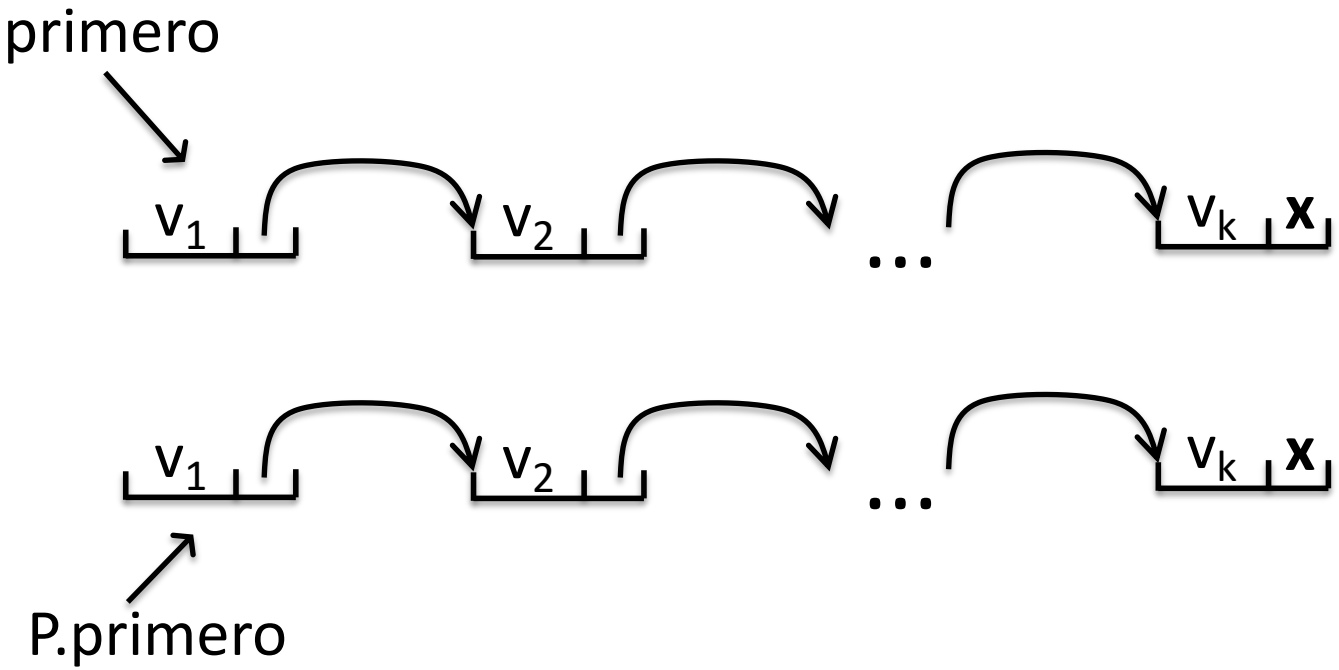
// Constructoras

Si

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



// Constructoras

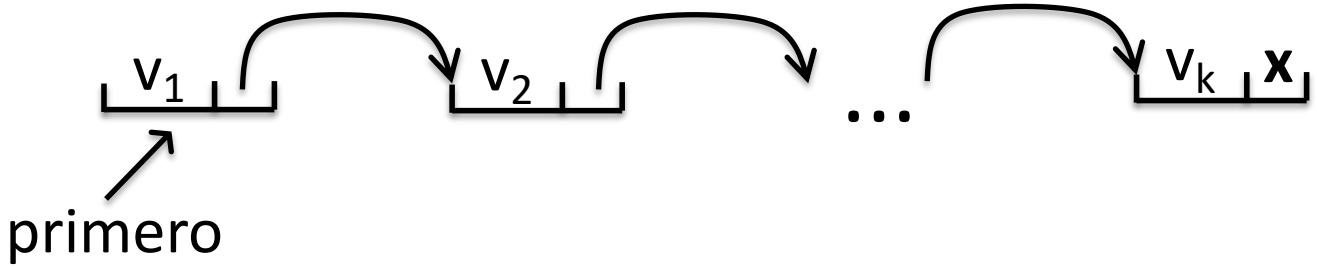


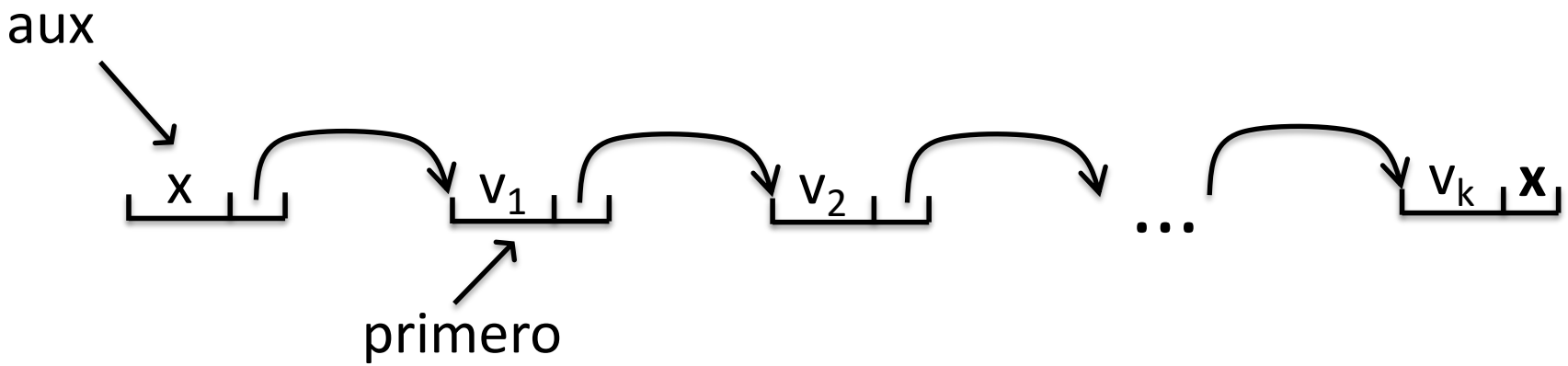
```
// Constructoras

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

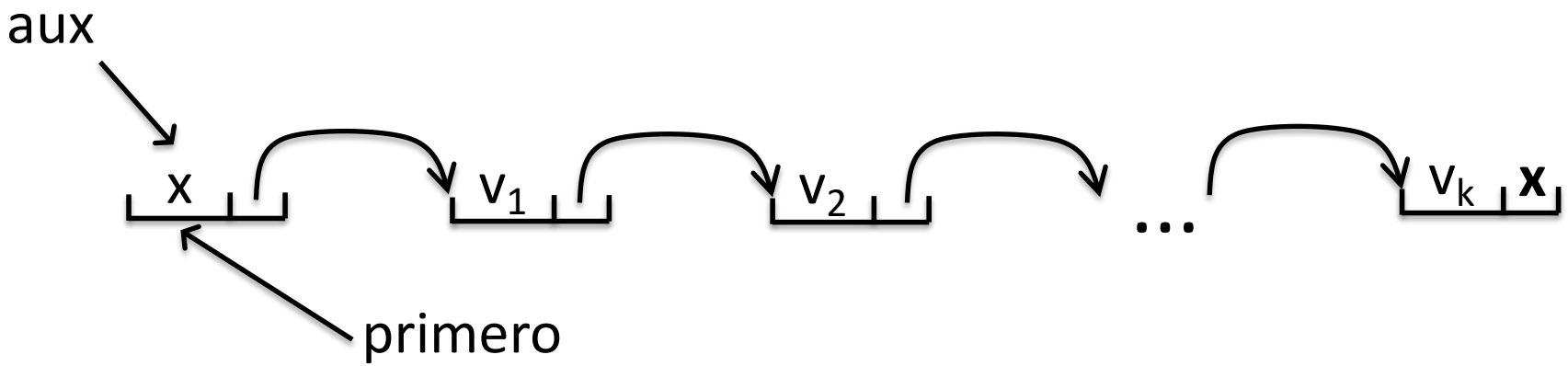
```
// Destructora

   ~stack(){
      borra_nodo_pila(primero);
                  //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;
   }
```

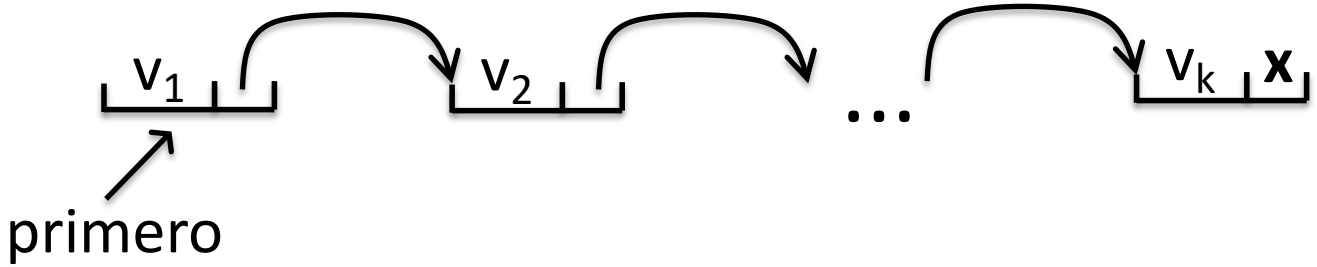
```
// Modificadoras

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

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

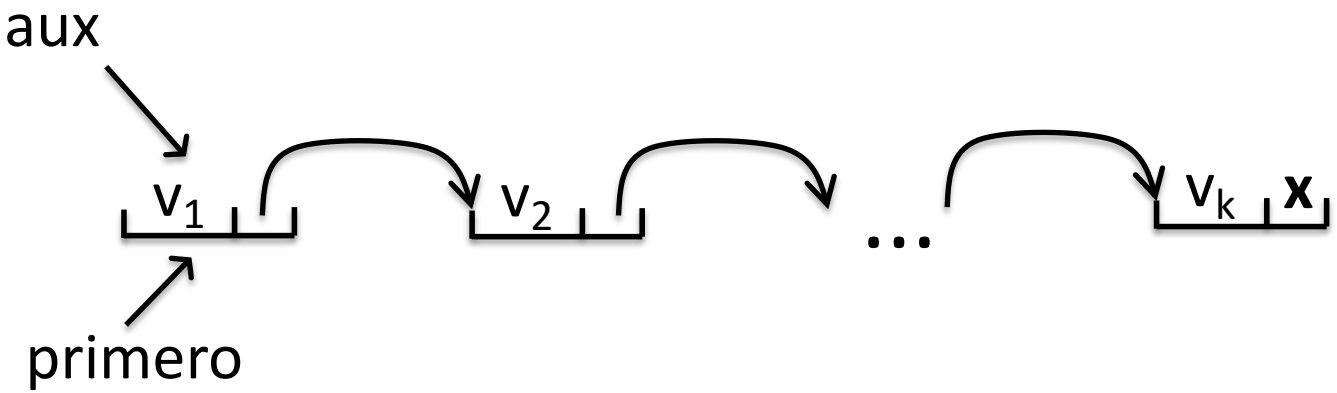

v1
v2
...
vk
x
primero

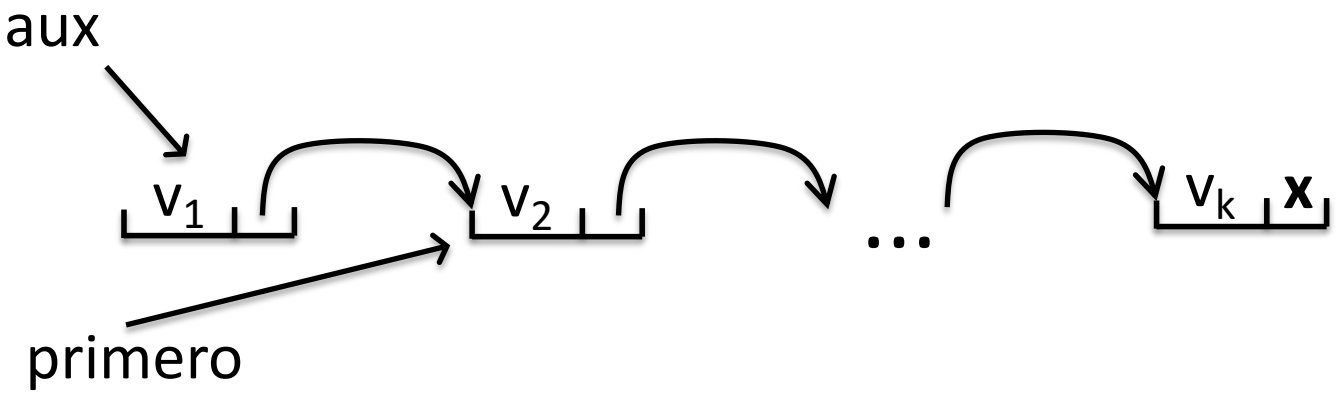

aux
x
v_1
v_2
...
v_k
x
primero

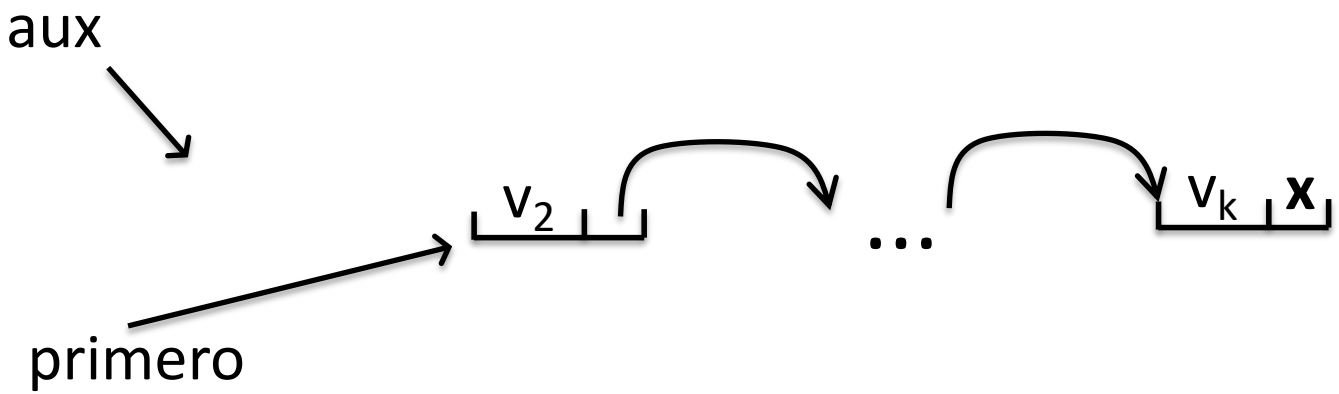

aux
x
v_1
v_2
...
v_k
x
primero

```
// Modificadoras

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

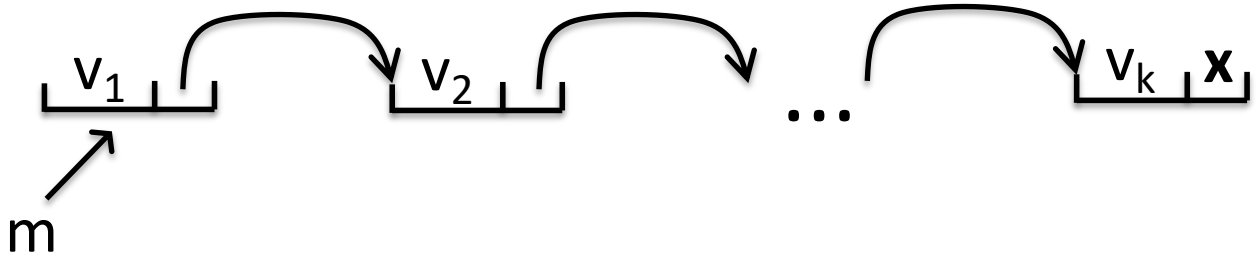

v1
v2
...
vk
x
primero

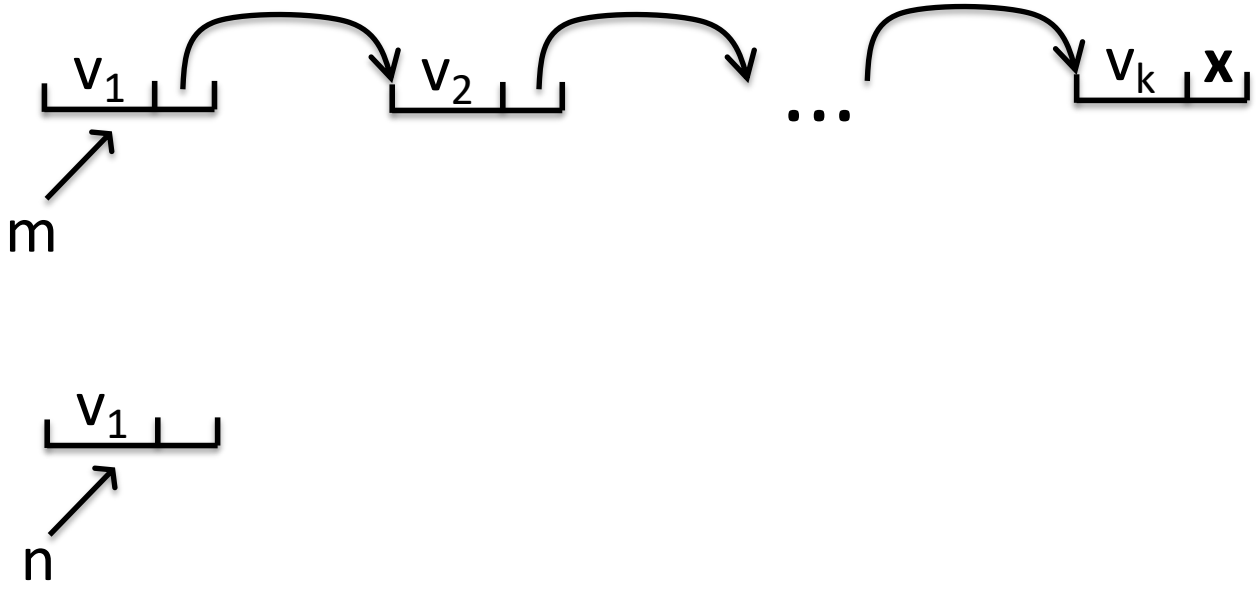

aux
v_1
v_2
...
v_k
x
primero

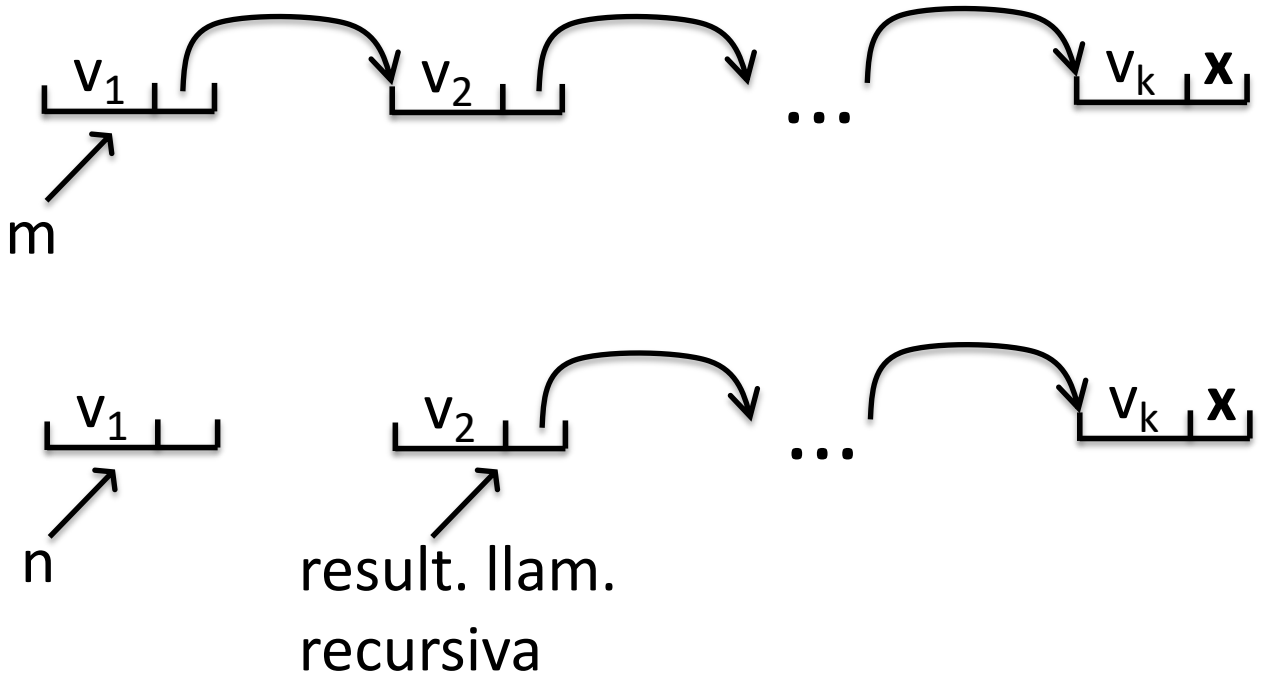

aux
v_1
v_2
...
v_k
x
primero

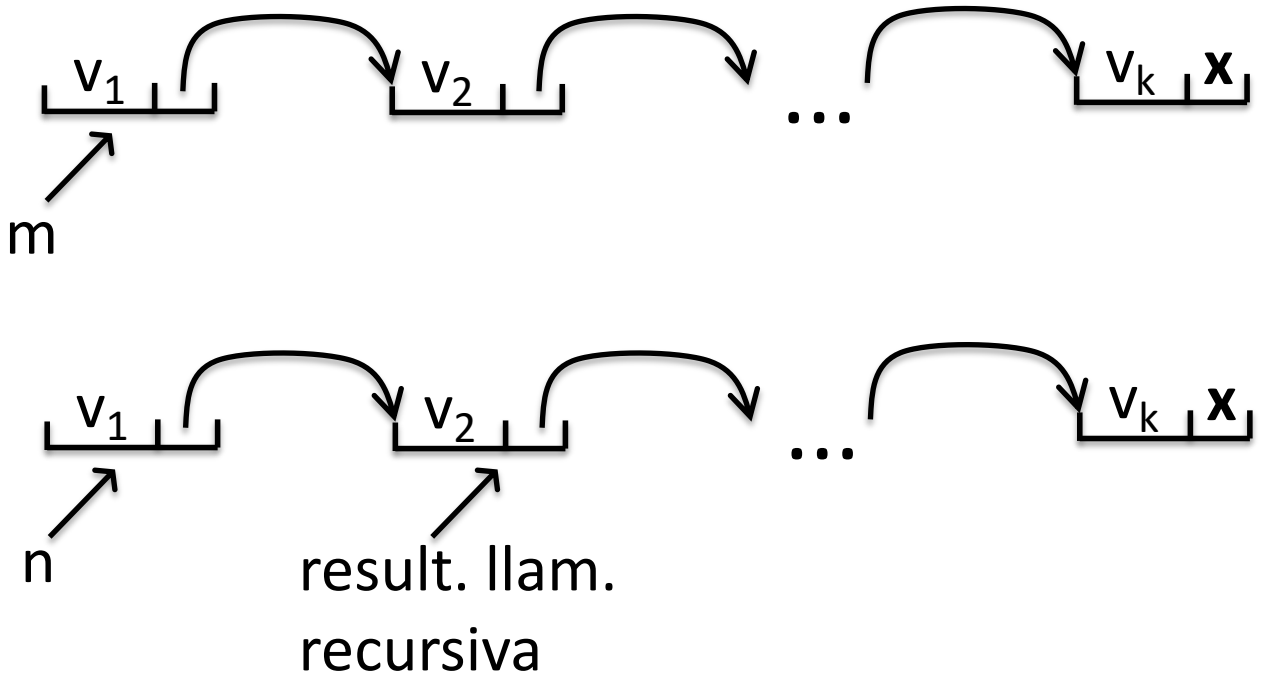

aux
primero
v_2
...
v_k
x

```
// 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;
   }
}
```


v_1
v_2
...
v_k
x
m


v_1
v_2
...
v_k
x
m
v_1
n


v_1
v_2
...
v_k
x
m
v_1
v_2
...
v_k
x
n
result. llam.
recursiva


v1
v2
...
vk
x
m
v1
v2
...
vk
x
n
result. llam.
recursiva

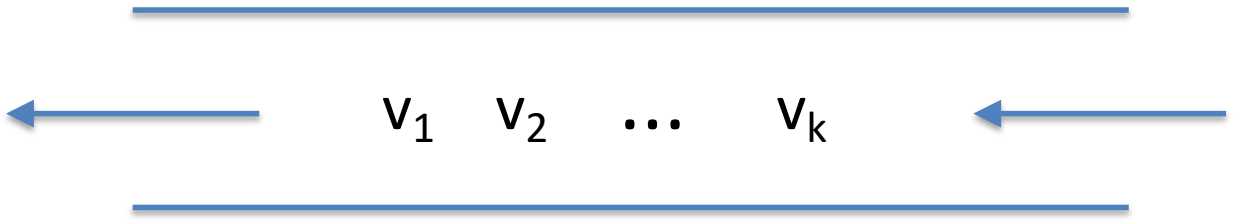
```
// Métodos privados
// Pre: true
/* Post: si m es nullptr no hace nada,
   si no libera el espacio ocupado por la cadena de
   nodos apuntada por m */

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

```
// La asignación

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

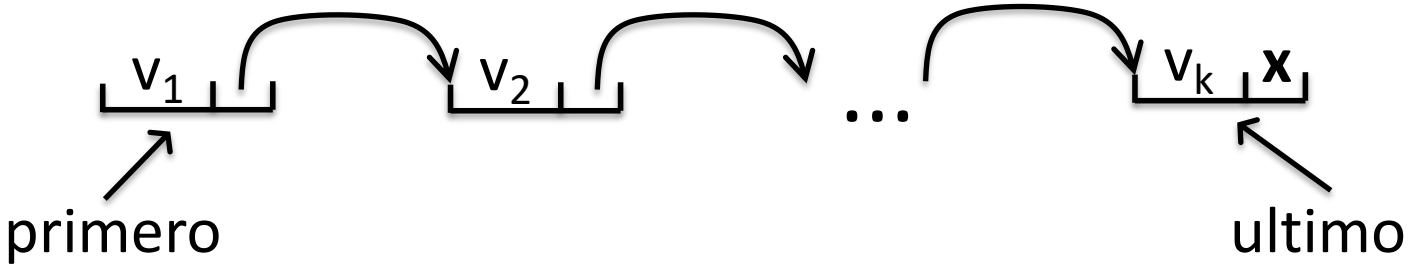
Colas

Implementación de colas

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


v_1 v_2 ... v_k

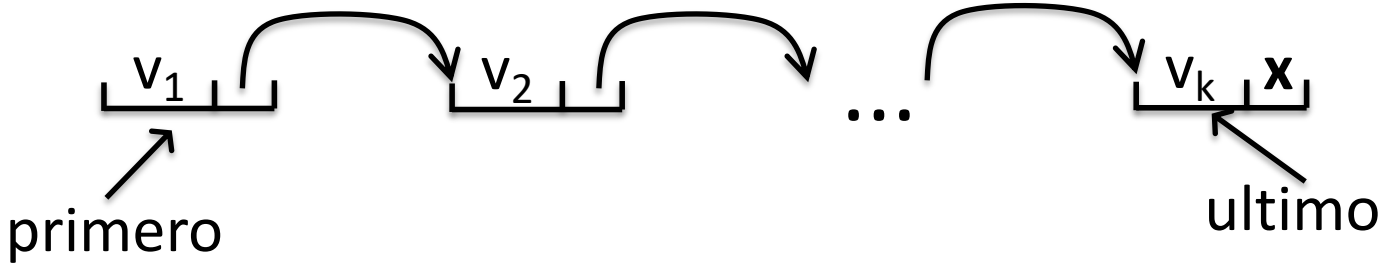
≡

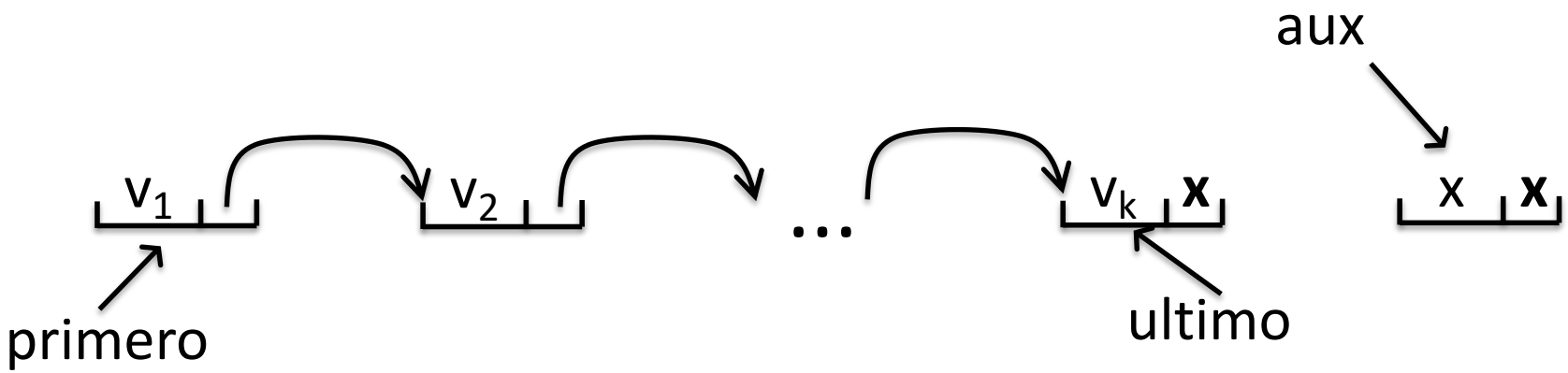

v_1
v_2
...
v_k
x
primero
ultimo

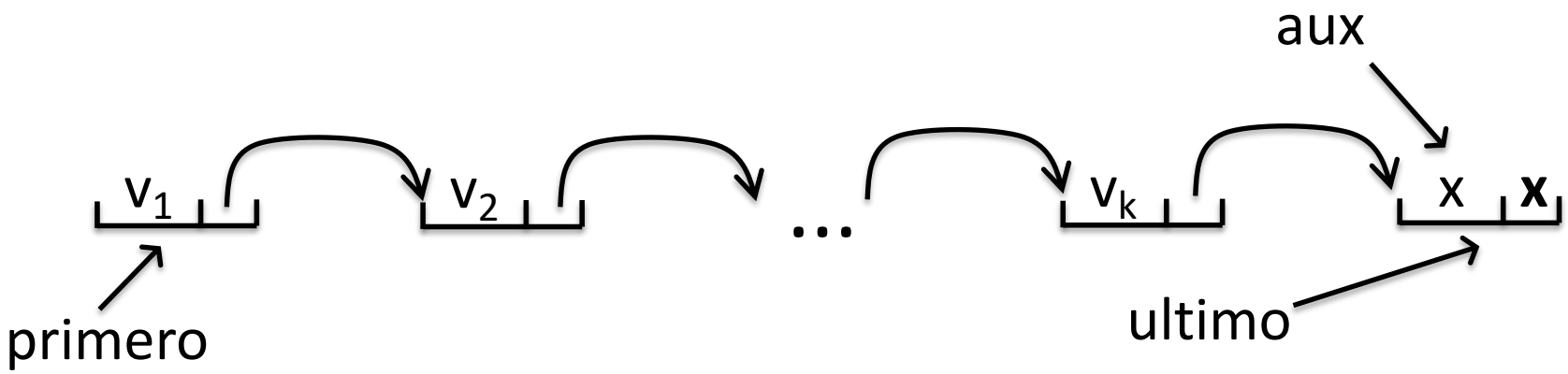
```
// Constructoras y destructoras

   queue(){
      longitud = 0;
      primero = nullptr;
      ultimo = nullptr;
   }

   queue(const queue& C){
      longitud = C.longitud;
      primero = copia_nodo_cola(C.primero, ultimo);
   }

   ~queue(){
      borra_nodo_cola(primero);
   }
```

```
// Consultoras

    T front() const {
    // Pre: la cola no está vacía
         return primero->info;
    }
    bool empty() const {
       return longitud == 0;
    }

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

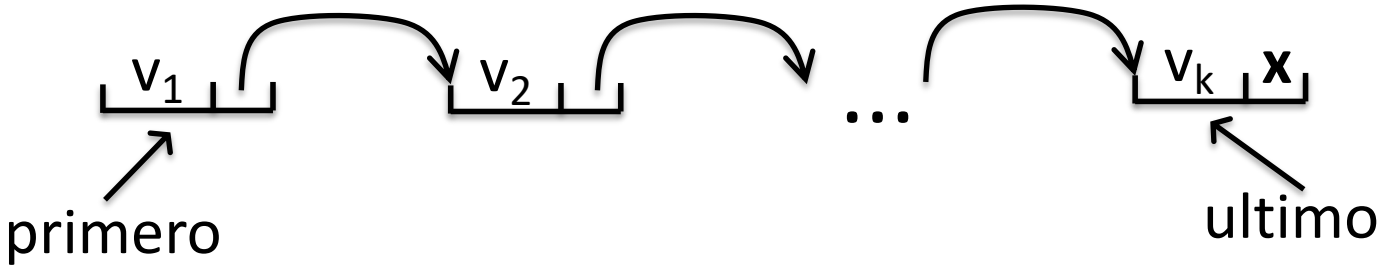
```
// Modificadoras

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

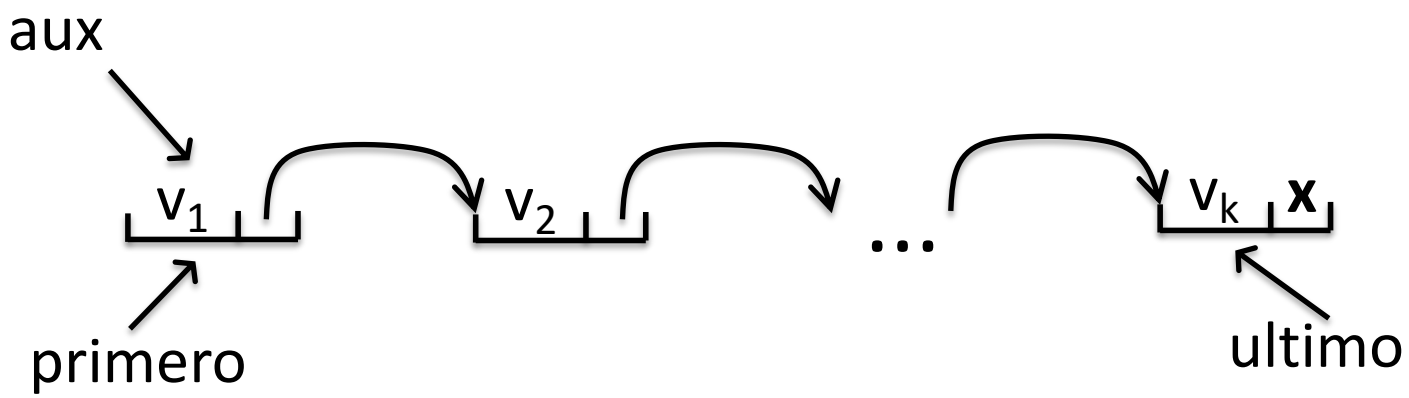

v_1
v_2
...
v_k
x
primero
ultimo

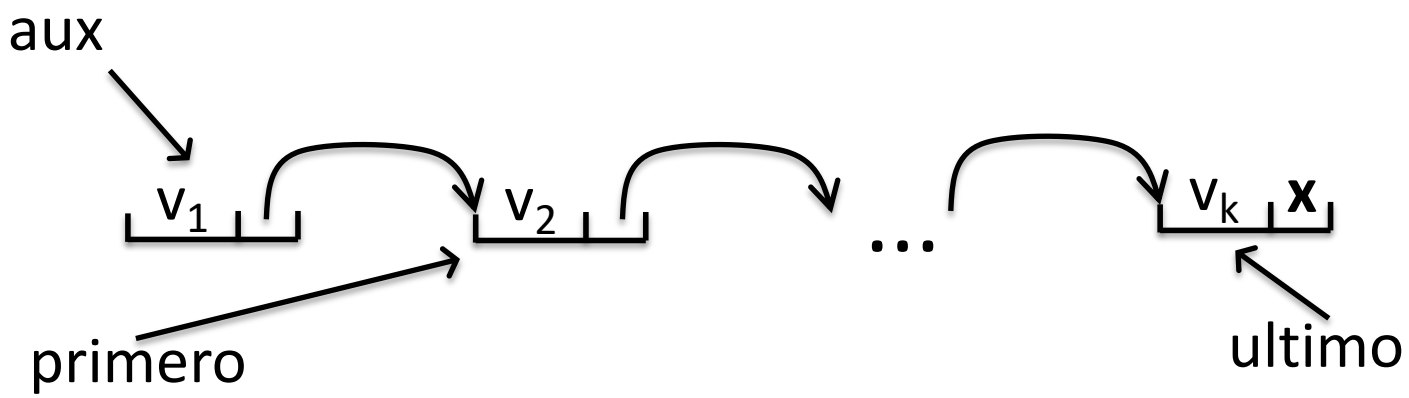

aux
v1
v2
...
vk
x
x
x
primero
ultimo

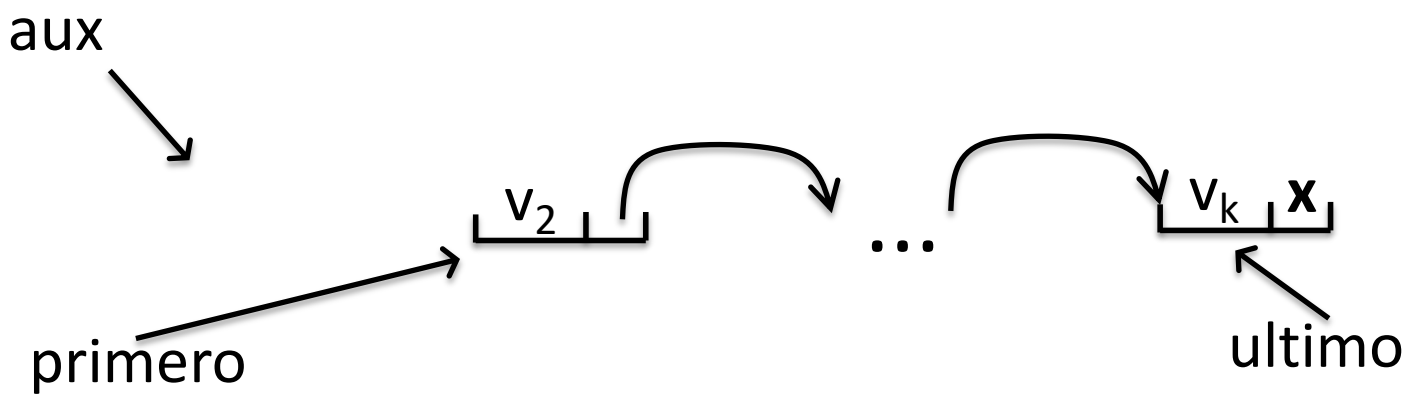

aux
v_1
v_2
...
v_k
x
x
primero
ultimo

```
// Modificadoras

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

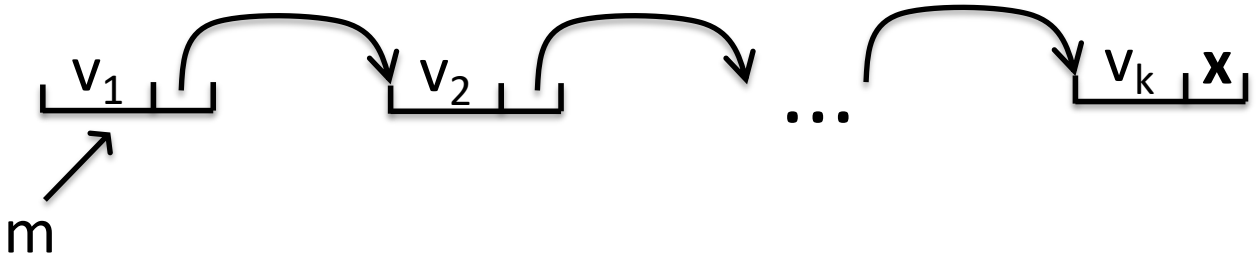

v_1
v_2
...
v_k
x
primero
ultimo

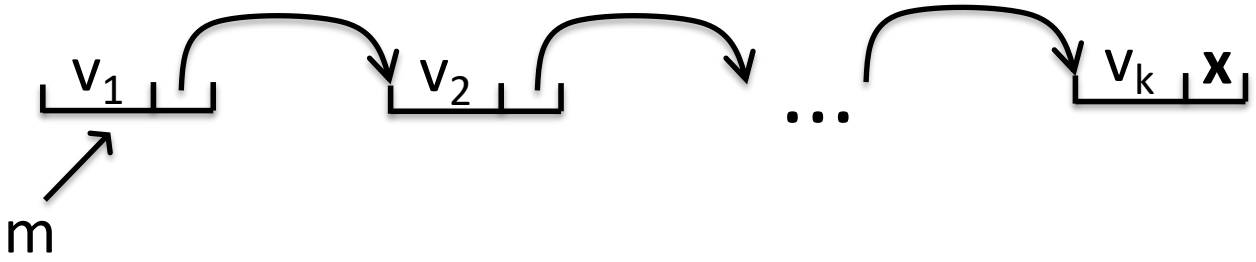

aux
v_1
v_2
...
v_k
x
primero
ultimo

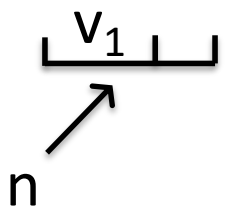

aux
v_1
v_2
...
v_k
x
primero
ultimo

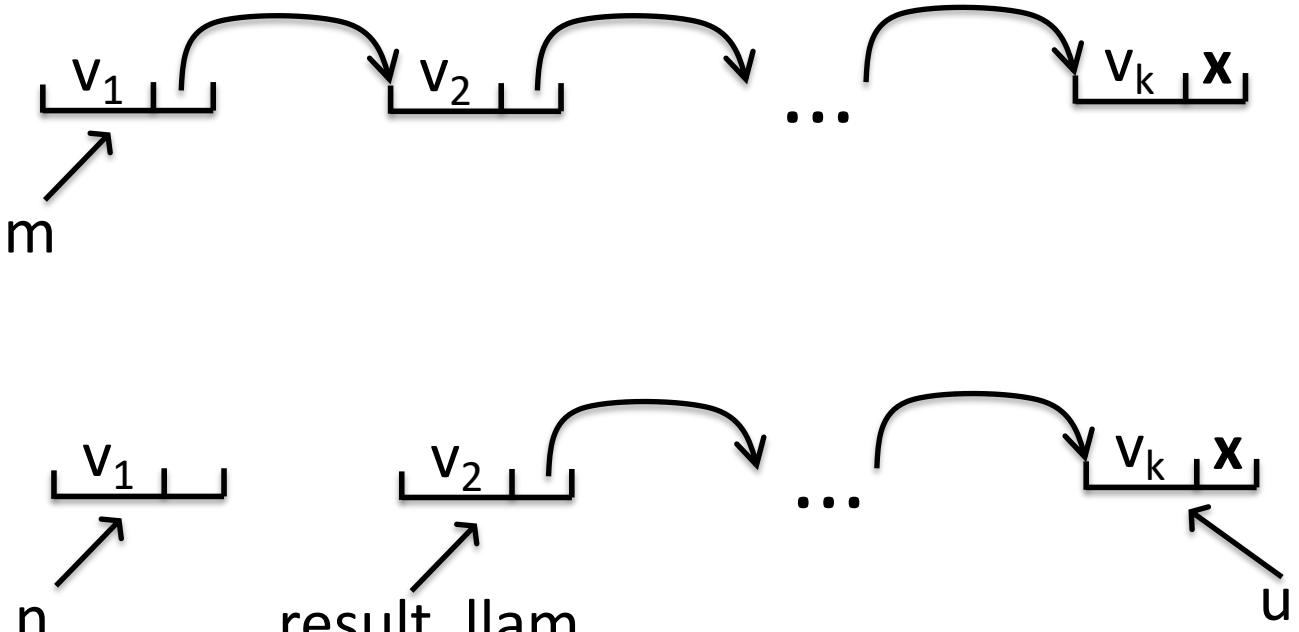

aux
primero
v_2
...
v_k
x
ultimo

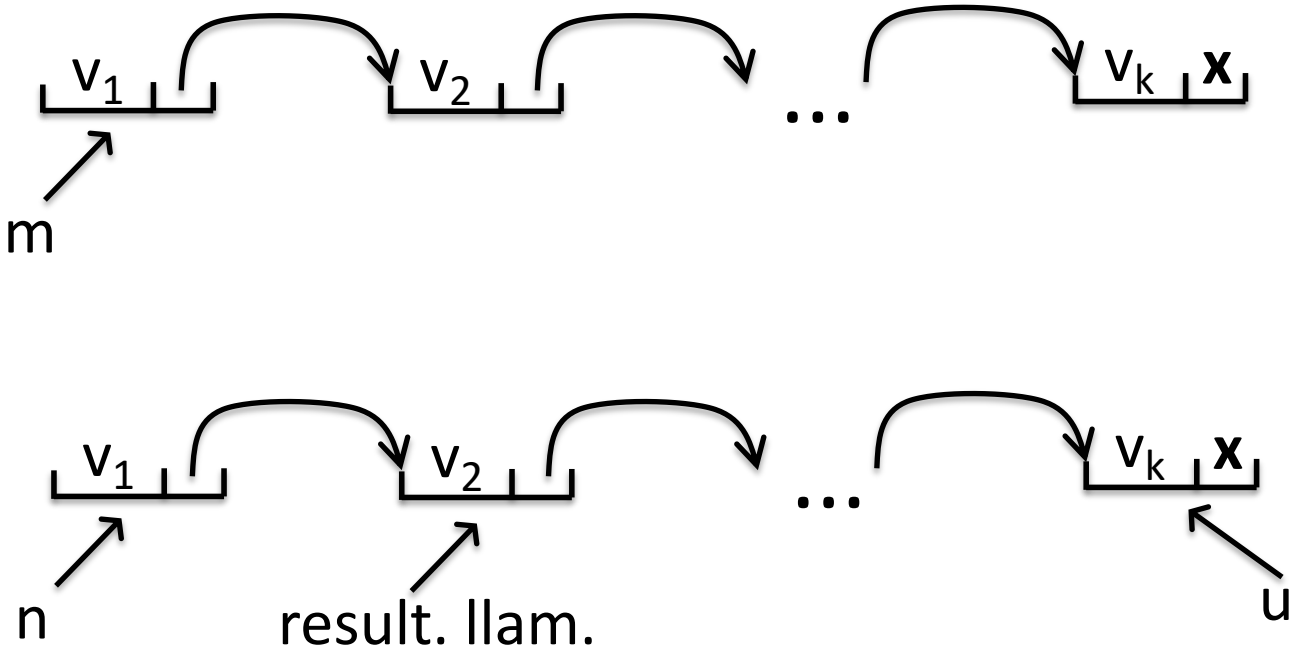
```
// Métodos privados
// Pre: true
/* Post: si m es nullptr el resultado y u son nullptr,
   si no, el resultado apunta a una cadena de nodos
   que es una copia de la cadena apuntada por m y u
   apunta al último nodo*/
static nodo_cola* copia_nodo_cola(nodo_cola* m,
                                  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;
   }
}
```


v_1
v_2
...
v_k
x
m


v_1
v_2
...
v_k
x
m


v_1
n


v_1
v_2
...
v_k
x
m
v_1
v_2
...
v_k
x
n
result. llam.
recursiva
u


v1
v2
...
vk
x
m
v1
v2
...
vk
x
n
result. llam.
recursiva
u

```
// 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_cola(nodo_cola* m){
   if (m != nullptr) {
      borra_nodo_cola(m->sig);
      delete m;
   }
}
```

```
// La asignación

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

```
// Ejemplo de incremento de eficiencia

// Pre: true
// Post: retorna true si la cola contiene x
bool busq(const T& x) const{
   nodo_cola* aux = primero;
   while (aux != nullptr) {
      if (aux->info == x) return true;
      aux = aux->sig);
   }
   return false;
}
```