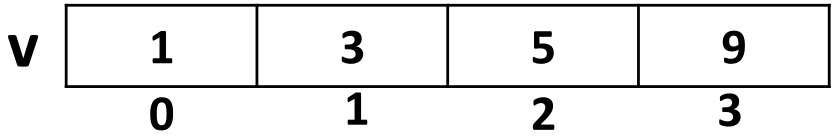
Programación 2
Tipos recursivos de datos I

Fernando Orejas

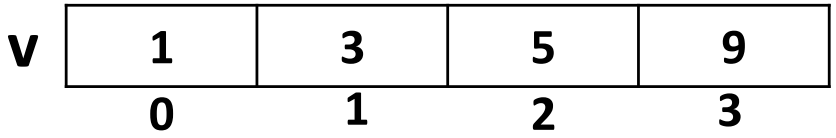
```
// Pre:  -1 <= i < v.size()
// Post: retorna el número de elementos frontera que
//       hay en v[0..i]

int r_front(const vector<int> &v, int i);

// Pre:  true
// Post: retorna el número de elementos frontera que
//       hay en v

int fronteras(const vector<int> &v){
   return r_front(v,v.size()-1);
}
```

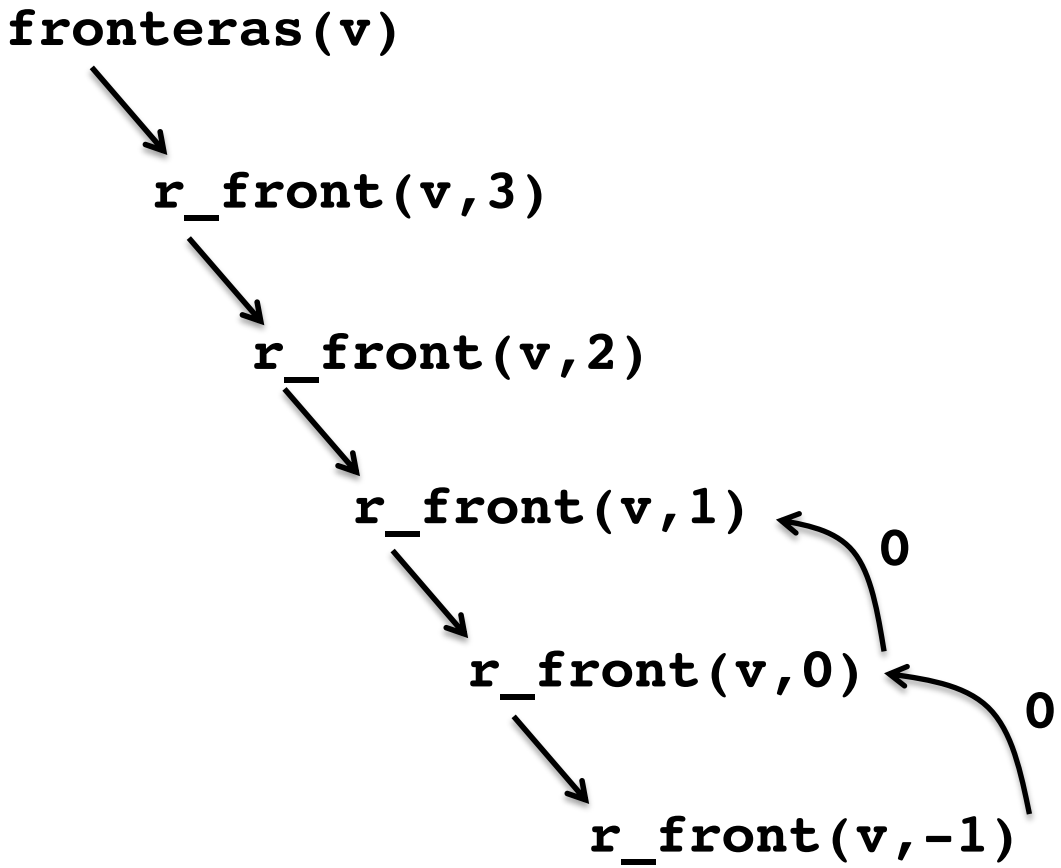
Elementos frontera

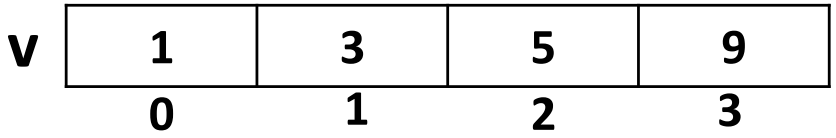
```
// Pre:  -1 <= i < v.size()
// Post: retorna el número de elementos frontera que
//       hay en v[0..i]

int r_front(const vector<int> &v, int i){
   if (i == -1) return 0;
   else {
      int n = r_front(v,i-1);
      if (v[i] == suma(v,i+1,v.size()-1)-suma(v,0,i-1)) ++n;
      return n;
   }
}
```

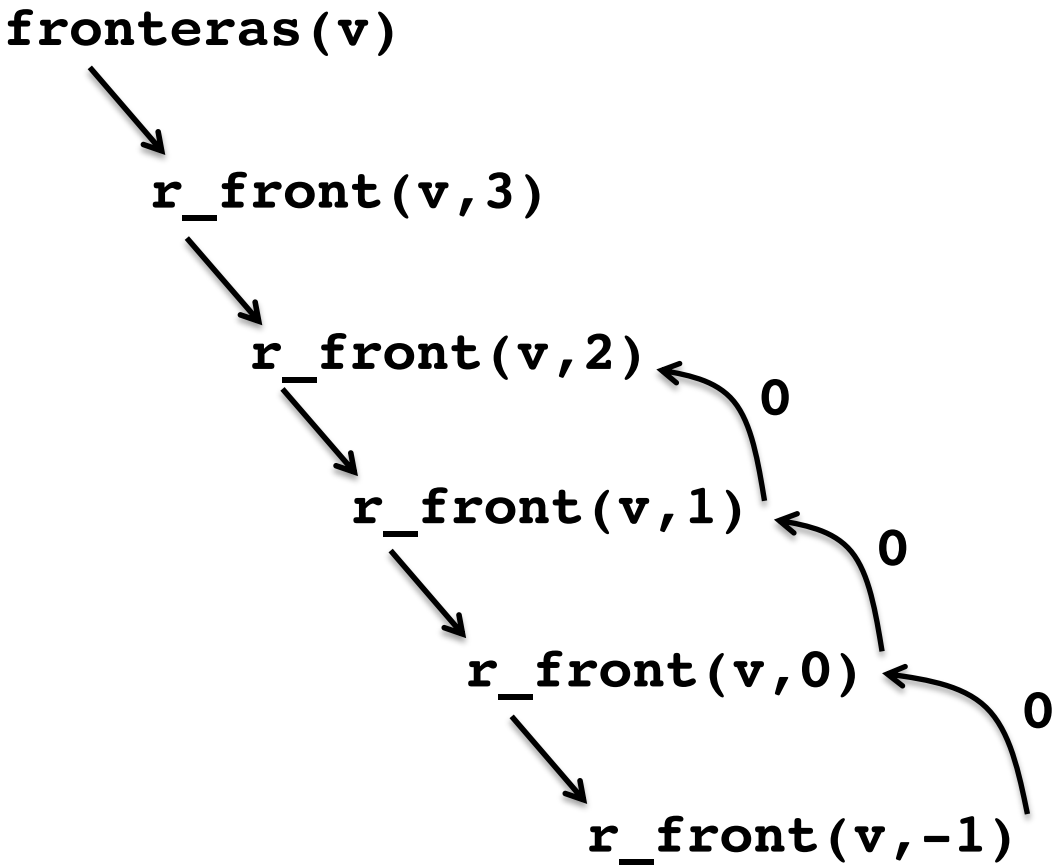


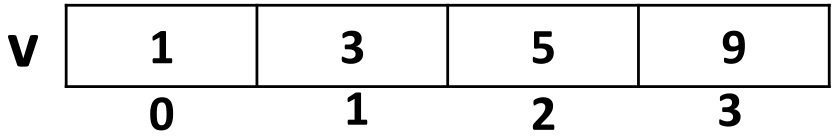

```
fronteras(v)
    r_front(v,3)
        r_front(v,2)
            r_front(v,1)
                r_front(v,0)
                    r_front(v,-1)
```

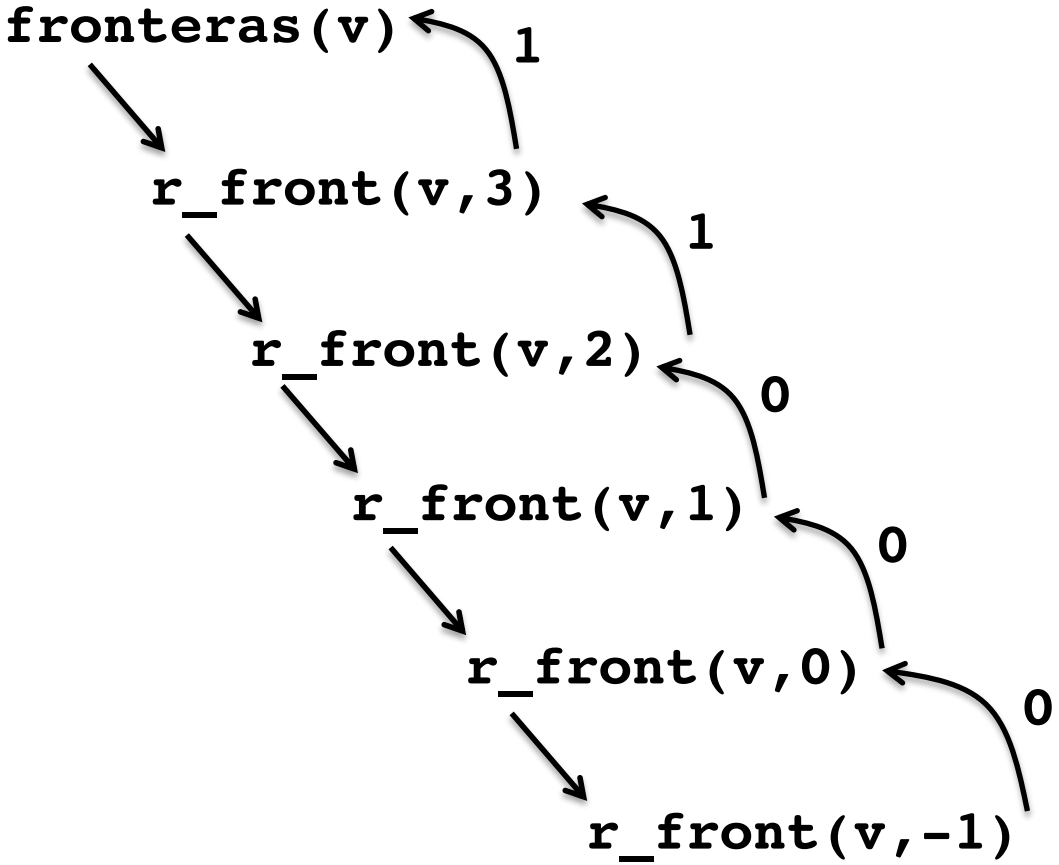

v
1
3
5
9
0
1
2
3


fronteras(v)
r_front(v,3)
r_front(v,2)
r_front(v,1)
r_front(v,0)
r_front(v,-1)
0
0


v
1
3
5
9
0
1
2
3

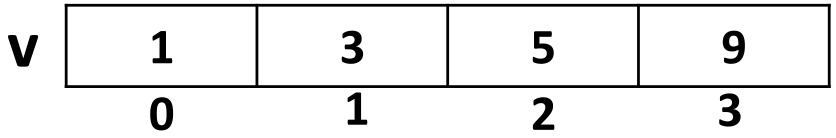

fronteras(v)
r_front(v,3)
r_front(v,2)
r_front(v,1)
r_front(v,0)
r_front(v,-1)
0
0
0


v
1
3
5
9
0
1
2
3


fronteras(v)
1
r_front(v,3)
1
r_front(v,2)
0
r_front(v,1)
0
r_front(v,0)
0
r_front(v,-1)

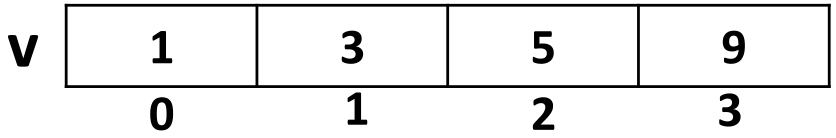
```
// Inmersión de eficiencia
// Pre:  -1 <= i < v.size(), sumaant = suma(v,0,i-1),
//    sumapost = suma(v,i+1,v.size()-1)
// Post: retorna el número de elementos frontera que
//       hay en v[0..i]

int i_front(const vector<int> &v, int i, int sumaant,
            int sumapost);

// Llamada inicial

int fronteras(const vector<int> &v){
   return i_front(v, v.size()-1, suma(v,0,v.size()-2), 0);
   }
}
```

```
// Inmersión de eficiencia
// Pre:  -1 <= i < v.size(), sumaant = suma(v,0,i-1),
//    sumapost = suma(v,i+1,v.size()-1)
// Post: retorna el número de elementos frontera que
//       hay en v[0..i]

int i_front(const vector<int> &v, int i, int sumaant,
            int sumapost){
   if (i == -1) return 0;
   else {
      if (i == 0) sant = 0;
      else sant = sumaant-v[i-1];
      spost = sumapost+v[i];
      int n = i_front(v,i-1,sant, spost);
      if (v[i] == sumapost-sumaant) ++n;
      return n;
   }
}
```




```
fronteras(v)
```

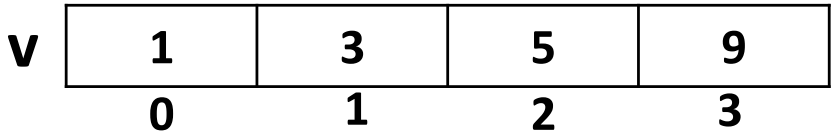
```
    1_front(v,3,9,0)
```




```
fronteras(v)
```

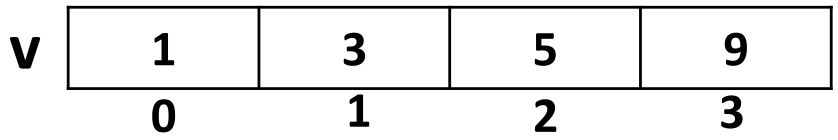
```
    1_front(v,3,9,0)
```

```
      1_front(v,2,4,9)
```



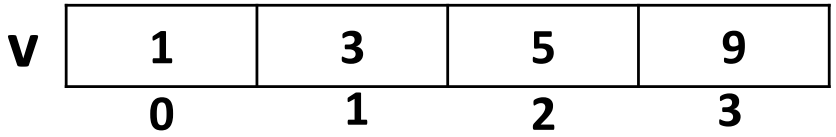

```
fronteras(v)
    1_front(v,3,9,0)
        1_front(v,2,4,9)
            1_front(v,1,1,14)
                1_front(v,0,0,17)
                    1_front(v,-1,0,18)
```

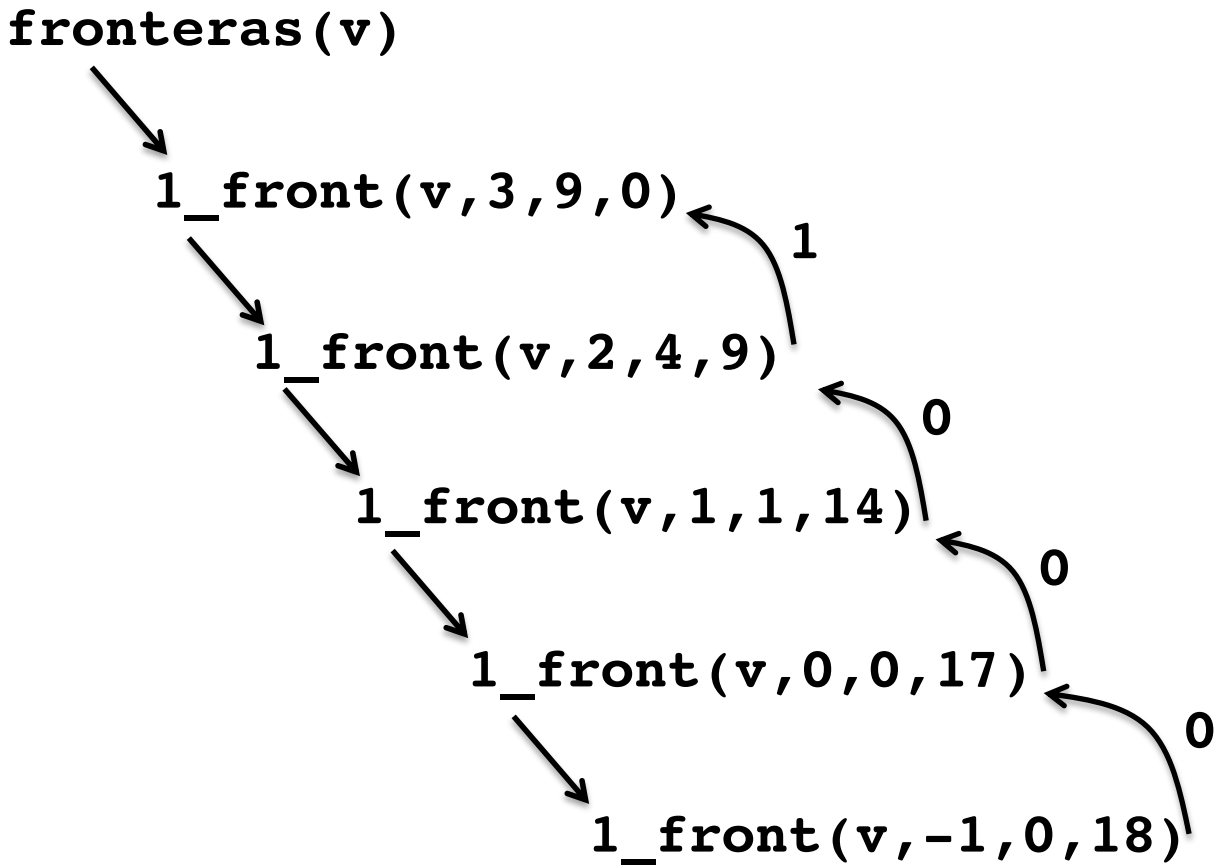
```
// Inmersión de eficiencia
// Pre:  -1 <= i < v.size(), sumaant = suma(v,0,i-1),
//    sumapost = suma(v,i+1,v.size()-1)
// Post: retorna el número de elementos frontera que
//       hay en v[0..i]

int i_front(const vector<int> &v, int i, int sumaant,
            int sumapost){
   if (i == -1) return 0;
   else {
      if (i == 0) sant = 0;
      else sant = sumaant-v[i-1];
      spost = sumapost+v[i];
      int n = i_front(v,i-1,sant, spost);
      if (v[i] == sumapost-sumaant) ++n;
      return n;
   }
}
```

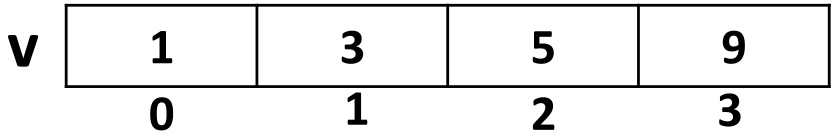


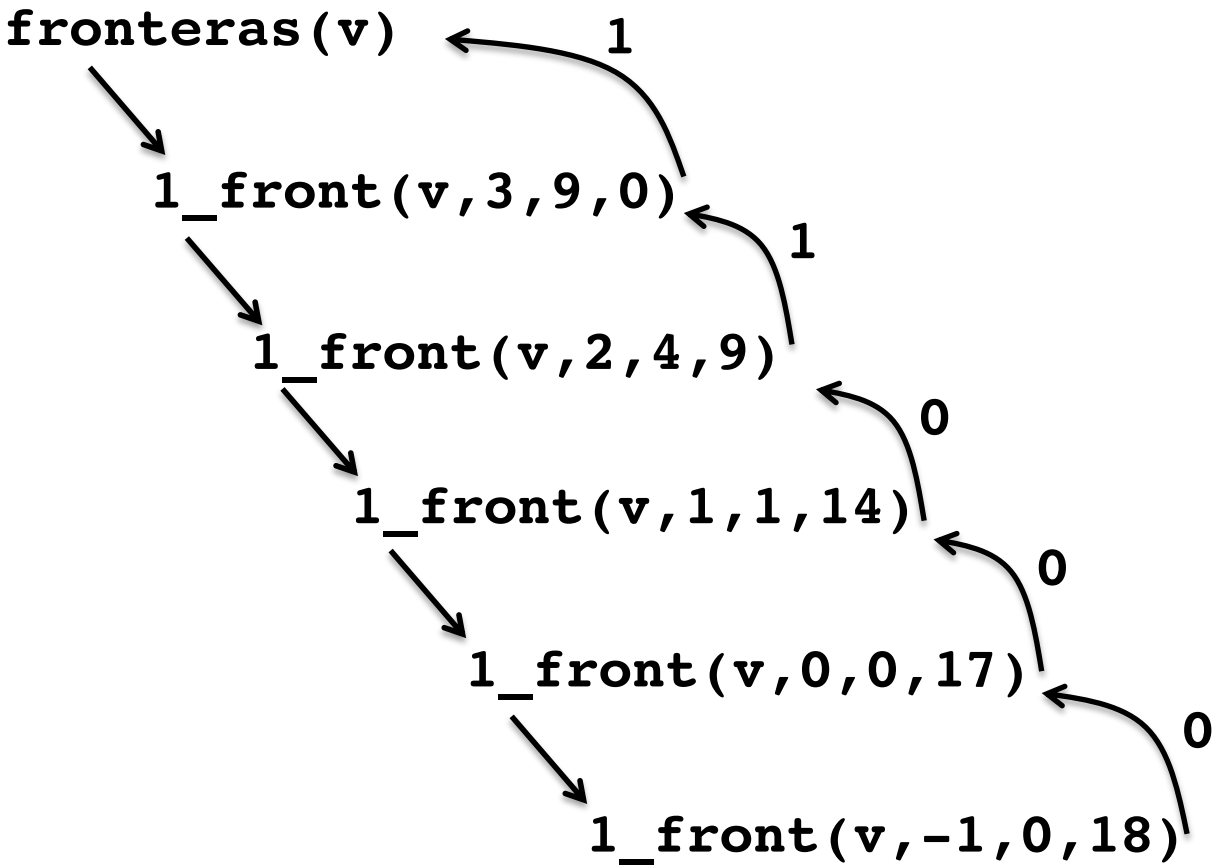

```
fronteras(v)
    1_front(v,3,9,0)
        1_front(v,2,4,9)
            1_front(v,1,1,14)
                1_front(v,0,0,17)
                    1_front(v,-1,0,18)
```

0


v
1
3
5
9
0
1
2
3


fronteras(v)
1_front(v,3,9,0)
1_front(v,2,4,9)
1_front(v,1,1,14)
1_front(v,0,0,17)
1_front(v,-1,0,18)
1
0
0
0


v
1
3
5
9
0
1
2
3


fronteras(v)
1
1_front(v,3,9,0)
1
1_front(v,2,4,9)
0
1_front(v,1,1,14)
0
1_front(v,0,0,17)
0
1_front(v,-1,0,18)

Árbol de medias

Dado un árbol de doubles, se quiere retornar otro, con la misma forma, en que cada nodo contenga la media de los nodos del subárbol original enraizado en ese nodo

```
// Pre:  true
// Post: retorna el árbol de medias de A

BinTree<double> a_medias(BinTree<double> &a};
```

```
// Pre:  true
// Post: retorna el árbol de medias de A


BinTree<double> a_medias(BinTree<double> &a}{
    if (not a.empty()) {
        double x = a.value();
        BinTree<double> b1 = a_medias(a.left());
        BinTree<double> b2 = a_medias(a.right());
        int n1 = b1.size(); int n2 = b2.size();
        double s1 = suma(a.left());
        double s2 = suma(a.right());
        return BinTree<double>((x+s1+s2)/(1+n1+n2),b1,b2);
    }
}
```

```
// Inmersión de eficiencia
// Pre:  true
// Post: b es el árbol de medias de A, s contiene
//   la suma de los nodos de A y n contiene el número de
//   nodos de A

void i_medias(BinTree<double> &a, BinTree <double> &b,
double &s, int &n);

// Llamada inicial


BinTree<double> a_medias(Arbol<double> &a){
   double s; int n;
   BinTree <double> b;
   i_medias(a, b, s, n);
   return b;
}
```

```
// Pre:  a = A, b está vacío
// Post: b es el árbol de medias de A, s contiene
//   la suma de los nodos de A y n contiene el número de
//   nodos de A


void i_medias(BinTree <double> &a, BinTree <double> &b,
              double &s, int &n){
   if (a.empty()) {s = 0; n = 0;}
   else {
      double s1,s2; int n1, n2;
       BinTree <double> b1, b2;
      double x = a.value();
      i_medias(a.left(),b1,s1,n1);
      i_medias(a.right,b2,s2,n2);
      s = x+s1+s2;
      n = n1+n2+1;
      b = BinTree <double> (s/n,b1,b2);
   }
}
```

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

Punteros

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

*p denota el valor que contiene el valor del objeto apuntado por p.

Punteros

Podemos inicializar un puntero p:

- Asignándole un puntero q:

```
p = q;
```

- Asignandole la dirección de un objeto existente:

```
p = &x;
```

- Creando una posición nueva y asignándosela a p:

```
p = new int;
```

- Asignándole nullptr:

```
p = nullptr;
```

Cómo (no) usar punteros

```
int* p;
int x;
p = &x; //*p y x pasan a ser alias
x = 5
int* q = p; //*q, *p y x son alias
*p = 3;
x = *q + 1;
p = new int;
delete p; //OK
delete q; //ERROR
```

Cómo (no) usar punteros

```
nodo* p, q, r;
p = new nodo;
q = nullptr;
p->dni = 775577; // p->dni es el campo dni de *p
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

  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++

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

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.

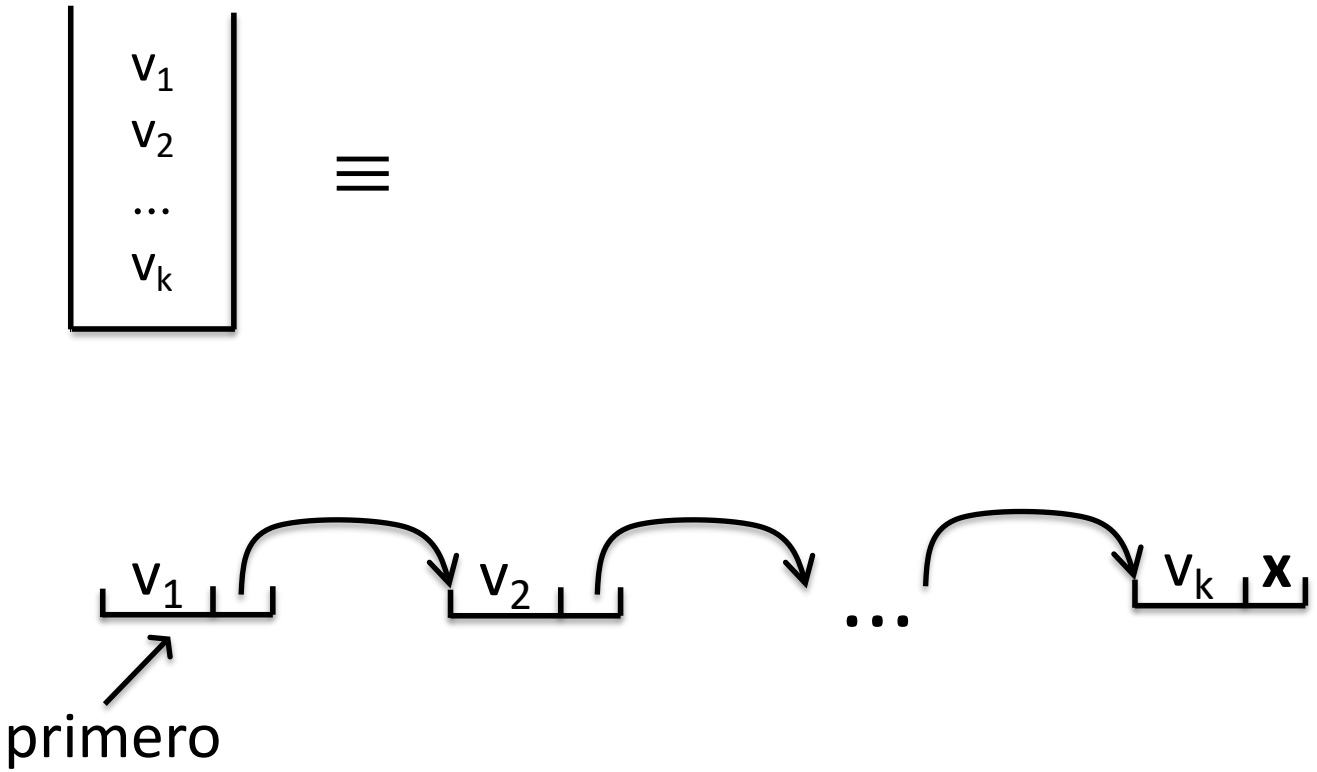
Ventajas de las estructuras de datos recursivas

- 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

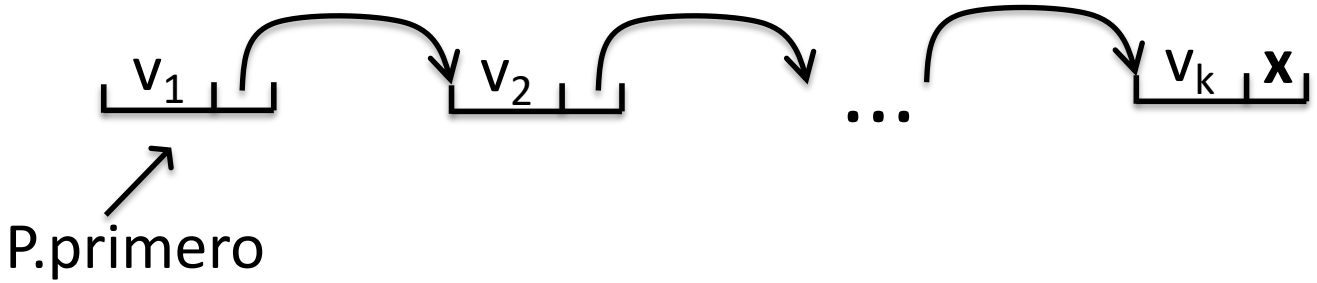
Implementación de pilas

```
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){


    }
```

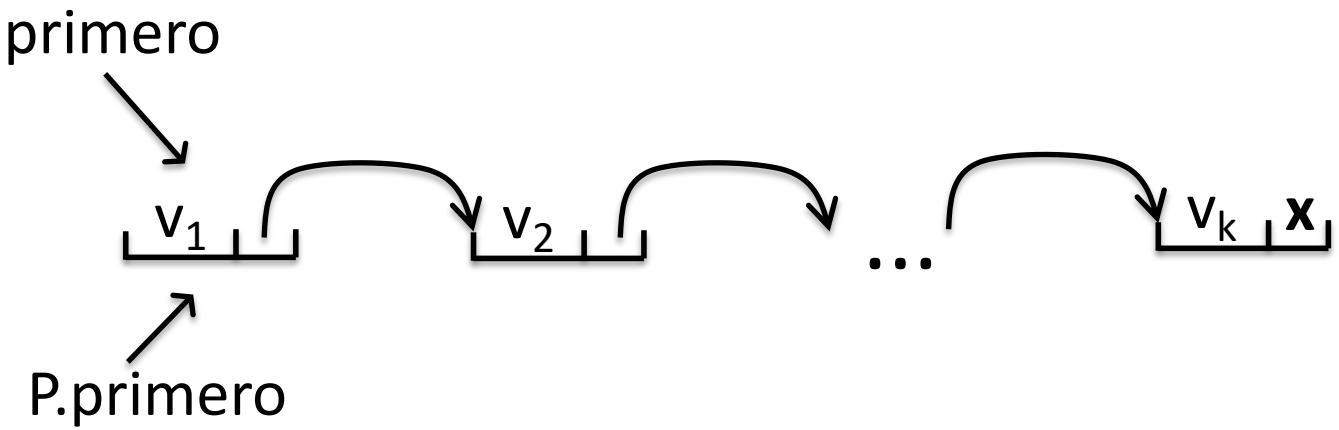
```
// Constructoras

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



// Constructoras

Si

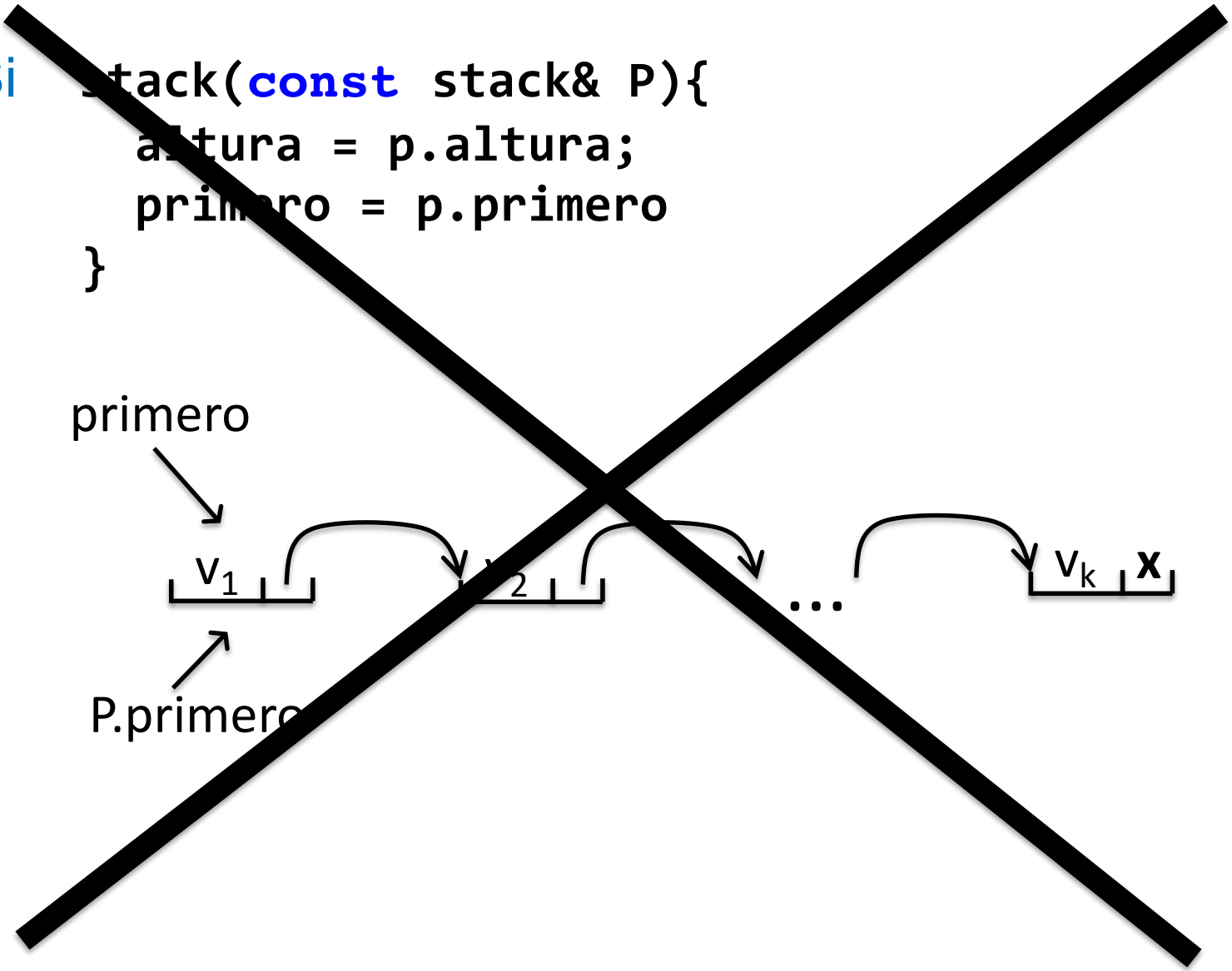
```
stack(const stack& P){
  altura = p.altura;
  primero = p.primero
}
```



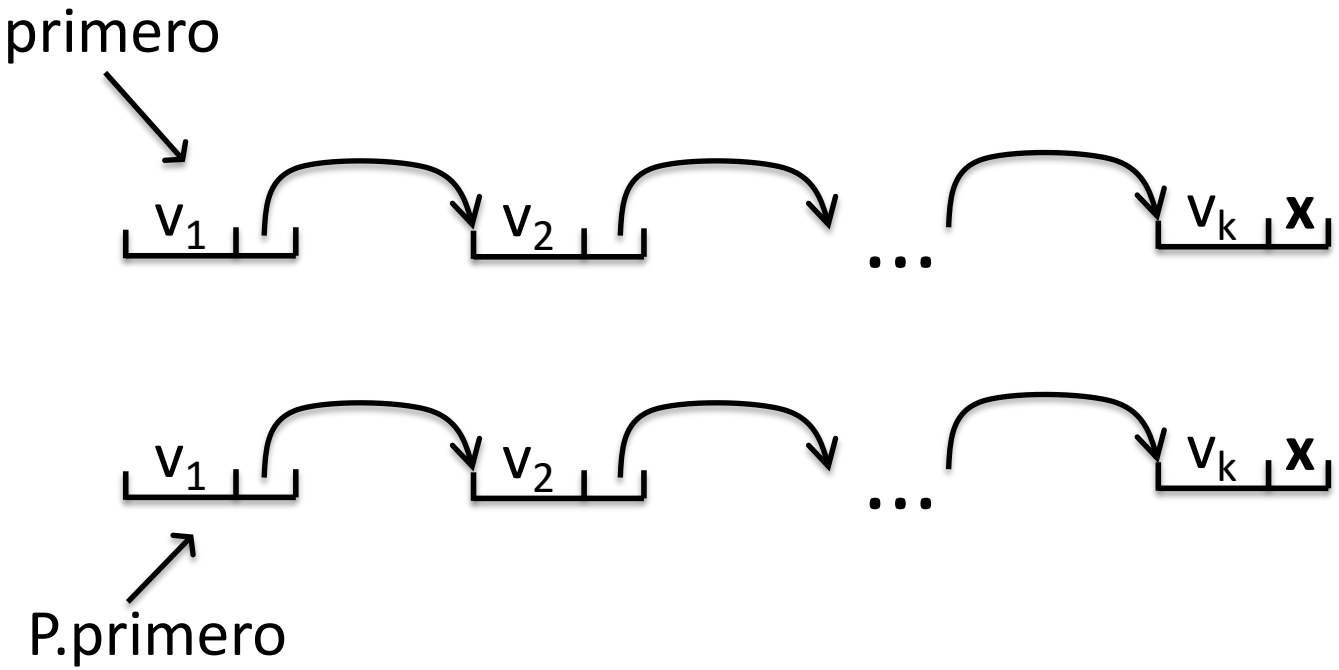
// Constructoras

Si

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



// Constructoras

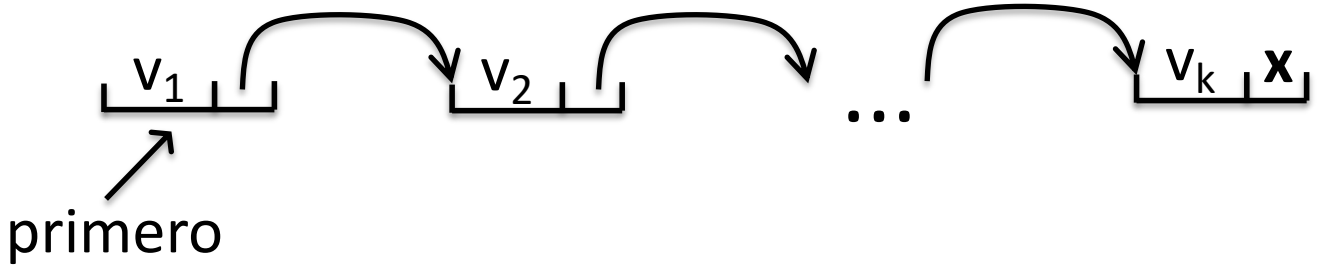


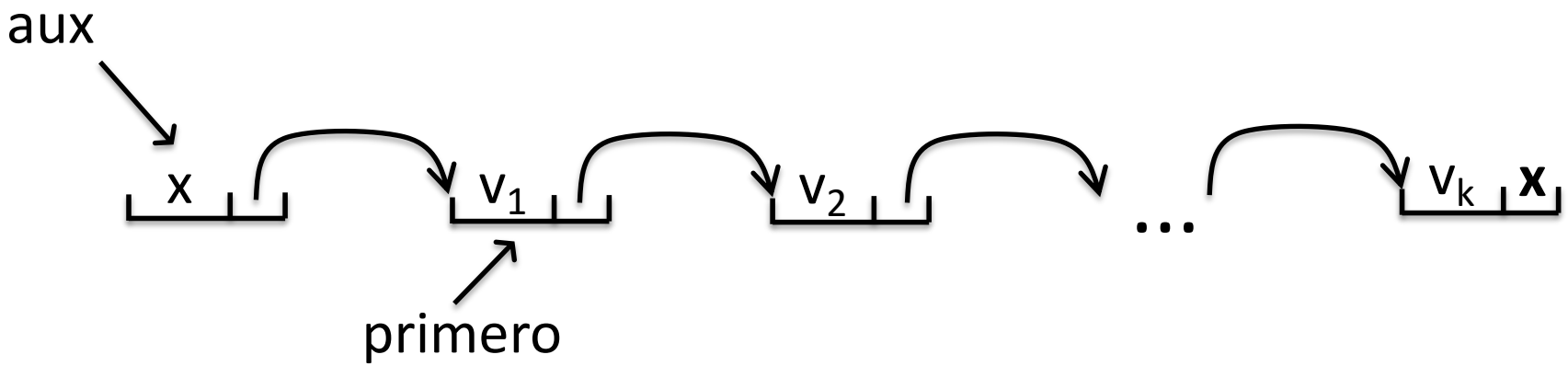
```
// Constructoras

   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
   }
```

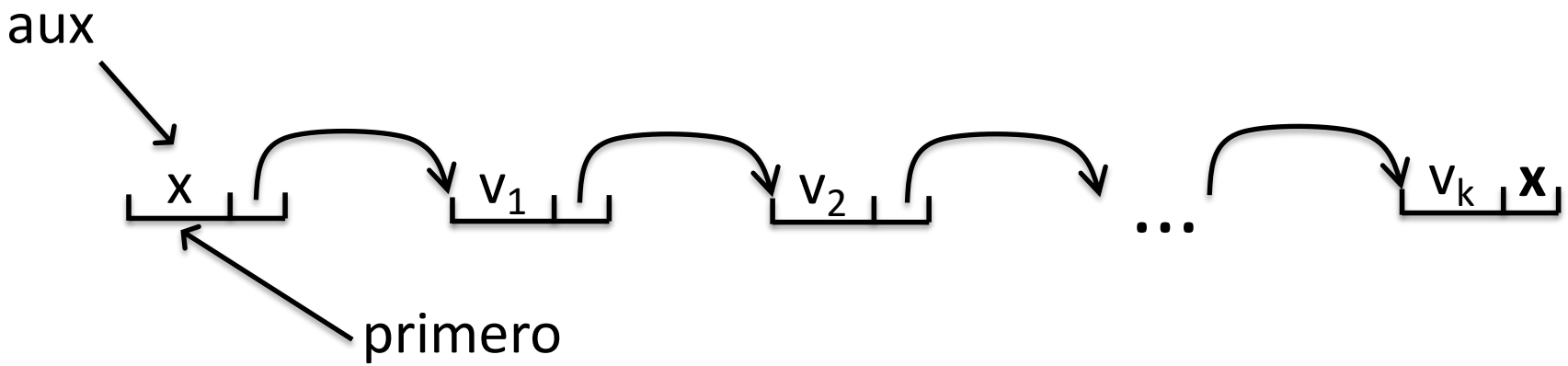
```
// Destructora

   ~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;
   }
```

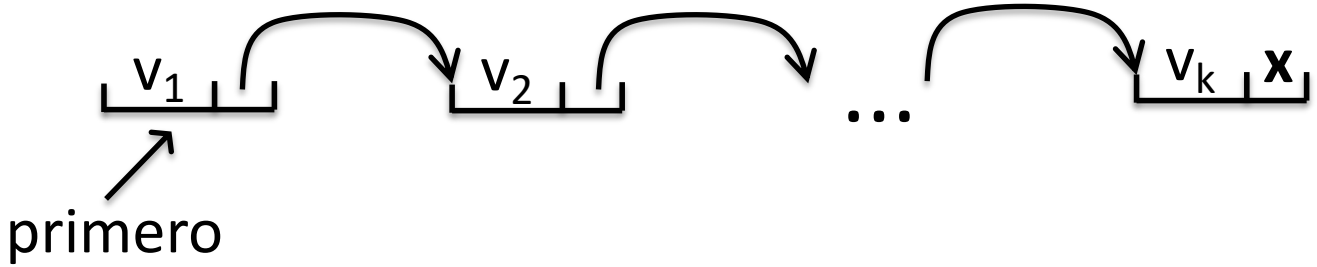
```
// Modificadoras

   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;
   }
```

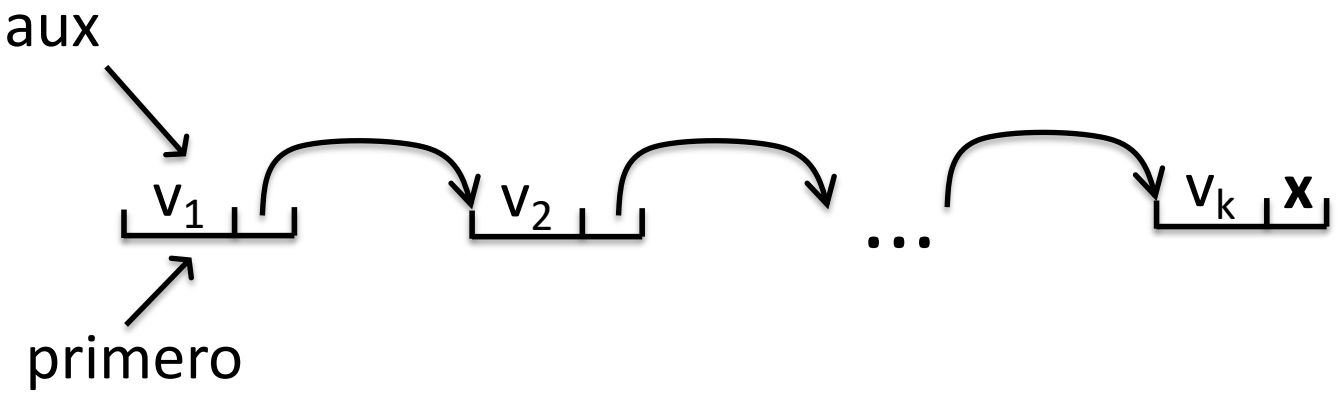

v1
v2
...
vk
x
primero

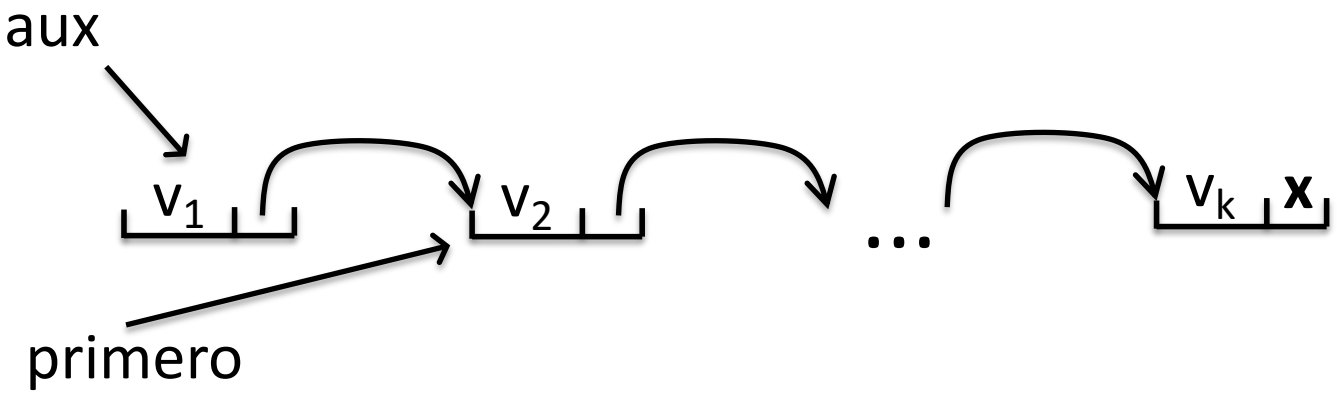

aux
x
v_1
v_2
...
v_k
x
primero

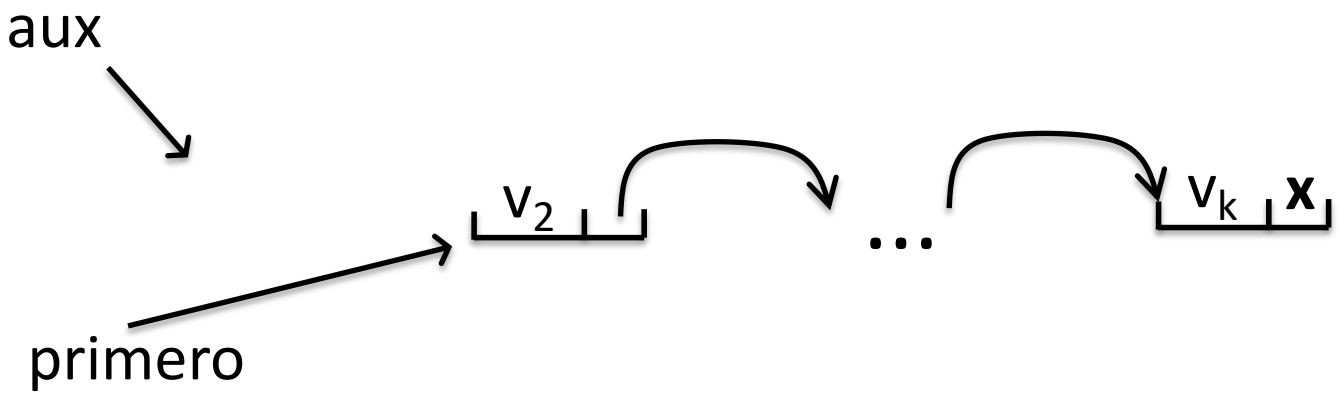

aux
x
v_1
v_2
...
v_k
x
primero

```
// 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