



Programación 2

Estructuras Lineales

Fernando Orejas

Objetivos

1. Estudiar algunas estructuras de datos
2. Ver cómo podemos construir programas que usan clases predefinidas

Estructuras lineales

Estructuras lineales

- Son estructuras de datos que contienen secuencias de valores
- Los accesos típicos que podemos tener son
 - Al primer elemento
 - Al último elemento
 - Al siguiente elemento
 - Al anterior elemento
- Las modificaciones típicas son inserciones o supresiones que pueden estar limitadas a los extremos

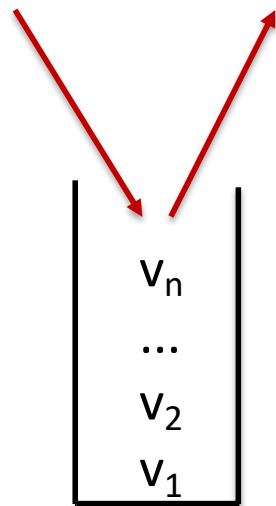
Estructuras lineales

- Ejemplos típicos son pilas, colas, deque, listas, listas con prioridades, etc.
- En la STL de C++: stack, queue, deque, list, priority list, etc.
- Son *templates* (tienen un tipo como parámetro)
- Son contenedores (*containers*)

Pilas

Pilas

- Son estructuras lineales a las que solo se puede *acceder* por un extremo.
- El último insertado es el primero suprimido: *Last In – First Out*
- También se les llama *pushdown stores*



Estructuras lineales

- Operaciones básicas:
 - push
 - pop
 - top
 - empty

Especificación de la clase stack

```
template <class T> class stack {  
    public:  
        // Constructoras  
  
        // Pre: cert  
        // Post: crea una pila vacía  
        stack ();  
  
        // Pre: cert  
        // Post: crea una pila que es una copia de S  
        stack (const stack& S);  
  
        // Destructora  
        ~stack();
```

```
// Modificadoras
```

```
/* Pre: La pila es  $[a_1, \dots, a_n]$ ,  $n \geq 0$  */
```

```
/* Post: Se añade el elemento x como primero de la pila,  
es decir, la pila será  $[x, a_1, \dots, a_n]$  */
```

```
void push(const T& x);
```

```
/* Pre: La pila es  $[a_1, \dots, a_n]$  y no está vacía ( $n > 0$ ) */
```

```
/* Post: Se ha eliminado el primer elemento de la pila  
original, es decir, la pila será  $[a_2, \dots, a_n]$  */
```

```
void pop ();
```

```
// Consultoras
```

```
/* Pre: la pila es  $[a_1, \dots, a_n]$  y no está vacía ( $n > 0$ ) */
```

```
/* Post: Retorna  $a_1$  */
```

```
T top() const;
```

```
/* Pre: cert */
```

```
/* Post: Retorna true si y solo si la pila está vacía */
```

```
bool empty() const;
```

```
/* Pre: cert */
```

```
/* Post: Retorna el número de elementos de la pila*/
```

```
int size() const;
```

```
private:
```

```
...
```

```
};
```

Suma de los elementos de una pila

// Pre: cert

// Post: Si la pila está vacía retorna 0

// si la pila es $[a_1, \dots, a_n]$, retorna $a_1 + \dots + a_n$

int suma (stack <int>& p);

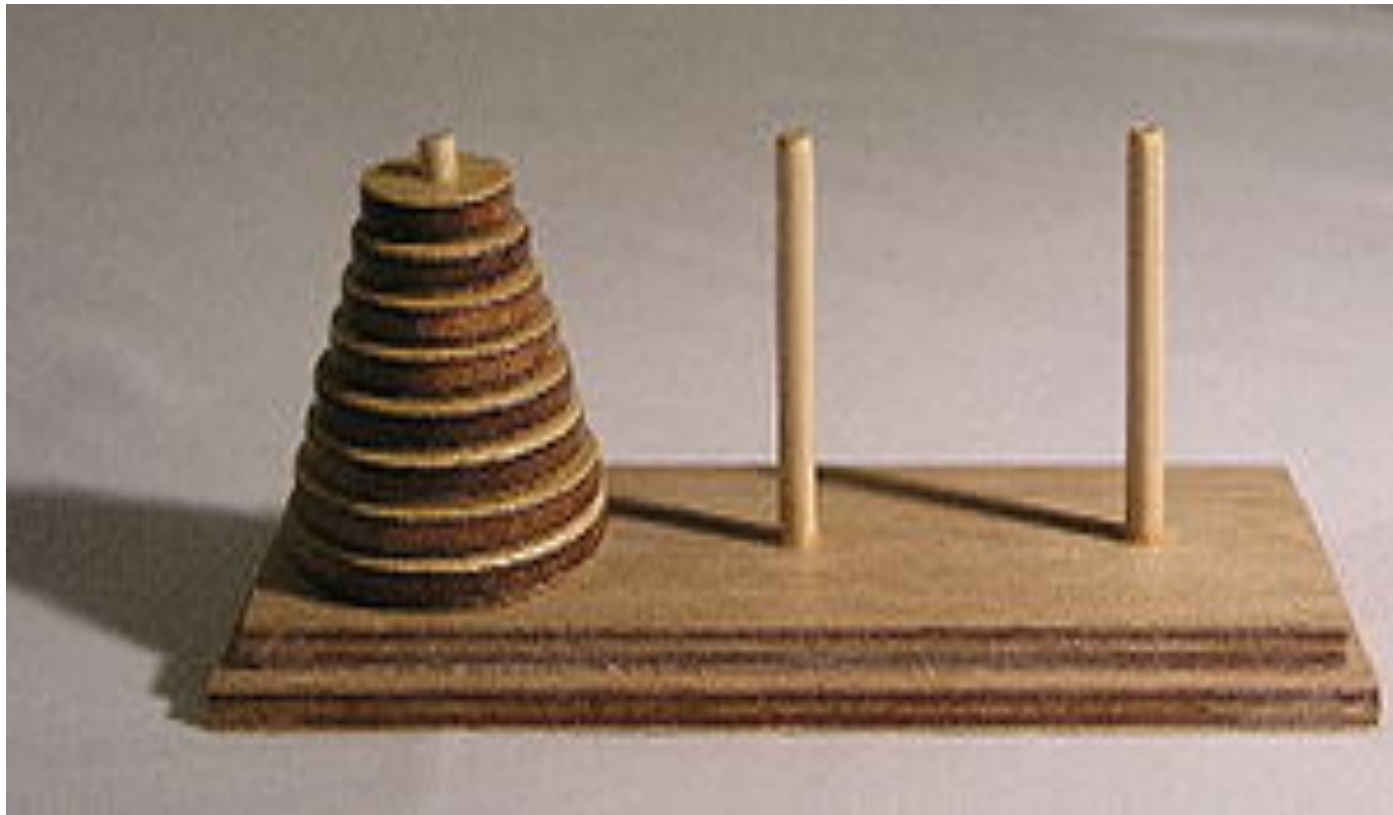
Suma de los elementos de una pila

```
// Pre: cert
// Post: Si la pila está vacía retorna 0
//       si la pila es  $[a_1, \dots, a_n]$ , retorna  $a_1 + \dots + a_n$ 
int suma (stack <int>& p) {
    int s = 0;
    while (not p.empty()) {
        s = s + p.top();
        p.pop()
    }
    return s;
}
```

Utilización de las pilas

Las pilas tienen muchos usos en programación, pero uno de los más importantes es la transformación recursiva-iterativa.

Torres de Hanoi





Especificación

```
// Pre:  N es el número de discos ( $N \geq 0$ ).  
// Post: Se escribe una solución de cómo mover n discos  
//       del poste org al poste dst usando el poste aux como  
//       auxiliar
```

```
typedef char poste;  
void hanoi(int N, poste org, poste aux, poste dst);
```

> Hanoi

5

Mueve un disco desde L a R
Mueve un disco desde L a M
Mueve un disco desde R a M
Mueve un disco desde L a R
Mueve un disco desde M a L
Mueve un disco desde M a R
Mueve un disco desde L a R
Mueve un disco desde L a M
Mueve un disco desde R a M
Mueve un disco desde R a L
Mueve un disco desde M a L
Mueve un disco desde R a M
Mueve un disco desde L a R
Mueve un disco desde L a M
Mueve un disco desde R a M



Mueve un disco desde L a R
Mueve un disco desde M a L
Mueve un disco desde M a R
Mueve un disco desde L a R
Mueve un disco desde M a L
Mueve un disco desde R a M
Mueve un disco desde R a L
Mueve un disco desde M a L
Mueve un disco desde M a R
Mueve un disco desde L a R
Mueve un disco desde L a M
Mueve un disco desde R a M
Mueve un disco desde L a R
Mueve un disco desde M a L
Mueve un disco desde M a R
Mueve un disco desde L a R

Caso general:

- Movemos $n-1$ discos de la izquierda al centro
- Movemos el mayor disco a la derecha
- Movemos $n-1$ discos del centro a la derecha



Algoritmo recursivo

```
void Hanoi(int N, poste org, poste aux, poste dst) {  
    if (N == 1) {  
        cout << "Mueve un disco desde " << org  
            << " a " << dst << endl;  
    }  
    else  
        Hanoi(N-1, org, dst, aux);  
    cout << "Mueve un disco desde " << org  
        << " a " << dst << endl;  
    Hanoi(N-1, aux, org, dst);  
}
```

Algoritmo iterativo

```
class HanoiTask {  
private:  
    int n_;  
    poste org_, aux_, dst_;  
public:  
    HanoiTask(int n, poste org, poste aux, poste dst) {  
        n_ = n; org_ = org; aux_ = aux; dst_ = dst;  
    }  
}
```

Algoritmo iterativo

```
class HanoiTask {  
    ...  
    // consultoras  
    int ndiscos() const { return n_; }  
    poste origen() const { return org_; }  
    poste auxiliar() const { return aux_; }  
    poste destino() const { return dst_; }  
    // Pre: ndiscos() == 1  
    // Post: escribe en el cout el movimiento necesario  
    // para mover un disco)  
    void escribir_mov() {  
        cout << "Mueve disco de " << org_ << " a " << dst_  
            << endl;  
    }  
};
```

Algoritmo iterativo

```
void hanoi(int N, poste org, poste aux, poste dst) {  
    HanoiTask initial_task(N, org, aux, dst);  
    stack <HanoiTask > S;  
    S.push(initial_task);  
    while (not S.empty()) {  
        HanoiTask curr = S.top(); S.pop();  
        if (curr.ndiscos() == 1) {  
            curr.escribir_mov();  
        } else {  
            // curr.ndiscos() > 1  
            ...  
        }  
    }  
}
```

Algoritmo iterativo

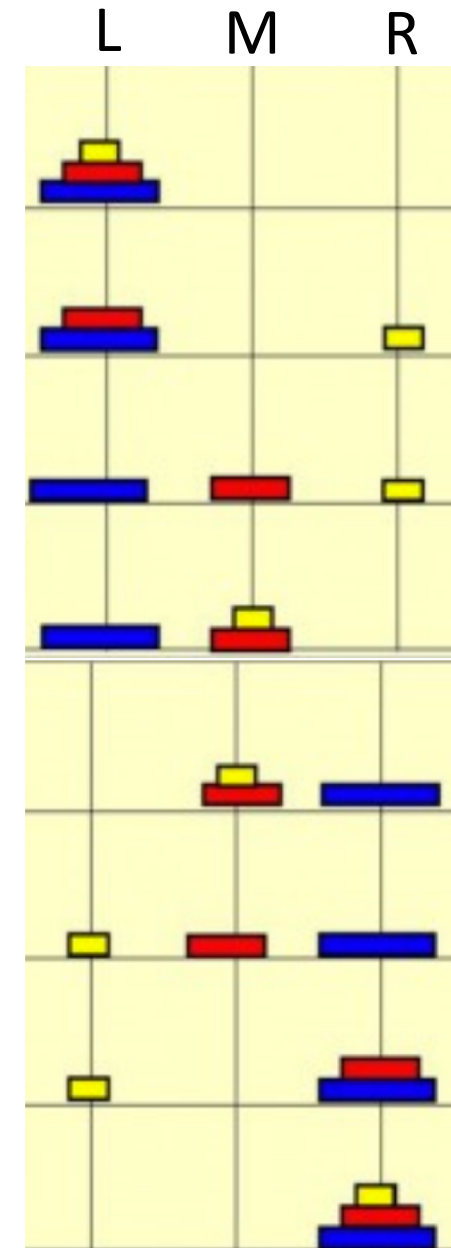
```
    } else {  
        // curr.ndiscos() > 1  
        ...  
        HanoiTask task1(curr.ndiscos()-1,  
            curr.origen(), curr.destino(), curr.auxiliar());  
        HanoiTask task2(1, curr.origen(), '*',  
            curr.destino());  
        HanoiTask task3(curr.ndiscos()-1, curr.auxiliar(),  
            curr.origen(), curr.destino());  
        S.push(task3);  
        S.push(task2);  
        S.push(task1);  
    }  
}
```


Torres de Hanoi (N=3) iterativo usando una pila

			(1,L,M,R)
			(1,L*,M)
			(1,R,L,M)
	(2,L,R,M)	(2,L,R,M)	(1,L*,R)
	(1,L*,R)	(1,L*,R)	(2,M,L,R)
(3,L,M,R)	(2,M,L,R)	(2,M,L,R)	

(1, L, *, M)			
(1, R, L, M)	(1, R, L, M)		
(1, L, *, R)	(1, L, *, R)	(1, L, *, R)	
(2, M, L, R)	(2, M, L, R)	(2, M, L, R)	(2, M, L, R)

(1, M, R, L)			
(1, M, *, R)	(1, M, *, R)		
(1, L, M, R)	(1, L, M, R)	(1, L, M, R)	

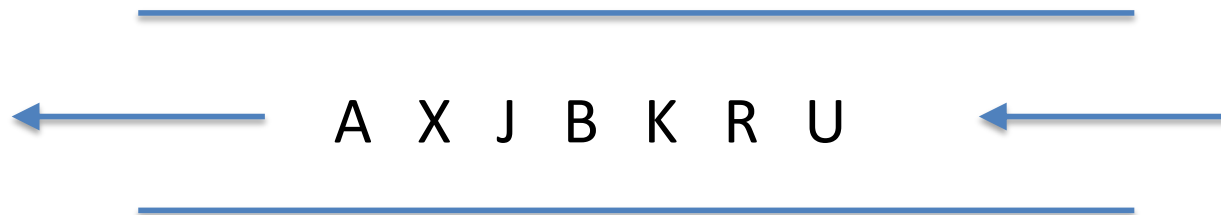


Colas

La clase queue

Tres operaciones básicas:

- Añadir un nuevo elemento (entrar, push)
- Eliminar el primer elemento que ha entrado (salir, pop)
- Ver quién es el primer elemento (primero, front)



Especificación de la clase queue

```
template <class T> class queue {  
    public:  
        // Constructoras  
  
        // Pre: cert  
        // Post: crea una cola vacía  
        queue ();  
  
        // Pre: cert  
        // Post: crea una cola que es una copia de q  
        queue (const queue& q);  
  
        // Destructora  
        ~queue();
```

```
// Modificadoras
```

```
/* Pre: La cola es  $[a_1, \dots, a_n]$ ,  $n \geq 0$  */
```

```
/* Post: Se añade el elemento x como último de la cola,  
        es decir, la cola será  $[a_1, \dots, a_n, x]$  */
```

```
void push(const T& x);
```

```
/* Pre: La cola es  $[a_1, \dots, a_n]$  y no está vacía ( $n > 0$ ) */
```

```
/* Post: Se ha eliminado el primer elemento de la cola  
        original, es decir, la cola será  $[a_2, \dots, a_n]$  */
```

```
void pop ();
```

```
// Consultoras
```

```
/* Pre: la cola es  $[a_1, \dots, a_n]$  y no está vacía ( $n > 0$ ) */
```

```
/* Post: Retorna  $a_1$  */
```

```
T front() const;
```

```
/* Pre: cert */
```

```
/* Post: Retorna true si y solo si la cola está vacía */
```

```
bool empty() const;
```

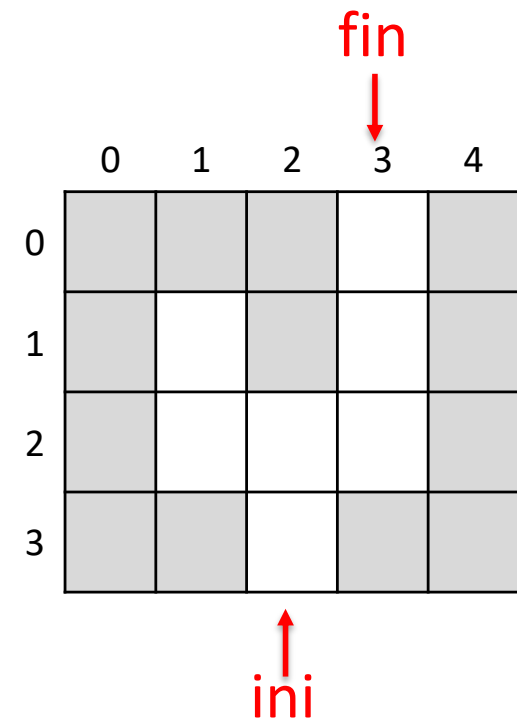
```
private:
```

```
...
```

```
};
```

Laberinto

- Laberintos rectangulares: $m \times n$ posiciones
- Cada posición (i,j) puede estar libre o ser una pared
- Una posición (i,j) es válida si $0 \leq i < m$ y $0 \leq j < n$
- Un camino de ini a fin es una secuencia de posiciones válidas, libres y adyacentes siendo ini la primera posición y fin la última.
- Todas las posiciones, salvo las del borde, tienen cuatro posiciones adyacentes: norte, sur, este y oeste.



La clase laberinto

- Operaciones:
 - `marcar(p)`: deja una marca en la posición `p`.
 - `libre(p)`: retorna `true` si `p` es válida y está libre y `false` en caso contrario.
 - `marcada(p)`: retorna `true` si `p` está marcada y `false` en caso contrario.
 - ...
- La clase `pos` representa un par `(fila, columna)` y tiene una constructora `pos(i,j)`, las consultoras `fila` y `col`, `es_igual` para comparar, etc.


```
// Pre: L.libre(ini), L.libre(fi) y todas las
// posiciones válidas de L están sin marcar
bool busca_cami(const Laberinto& L, pos ini, pos fin) {
    queue<pos> Q;
    Q.push(ini); L.marcar(ini);
    bool encontrado = false;
    while (not Q.empty() and not encontrado) {
        pos p = Q.front(); Q.pop();
        if (p.es_igual(fin)) encontrado = true;
        else {
            ...
        }
    }
    // encontrado==true si y solo si hay un camino
    // entre 'ini' y 'fin'
    return encontrado;
}
```

```
// p = (i,j) != fi
pos norte(p.fila()-1, p.col());
if (L.libre(norte) and not L.marcada(norte)) {
    Q.push(norte); L.marcas(norte);
}
pos este(p.fila(), p.col()+1);
if (L.libre(este) and not L.marcada(este)) {
    Q.push(este); L.marcas(este);
}
pos sur(p.fila()+1, p.col());
if (L.libre(sur) and not L.marcada(sur)) {
    Q.push(sur); L.marcas(sur);
}
pos oeste(p.fila(), p.col()-1);
if (L.libre(oeste) and not L.marcada(oeste)) {
    Q.push(oeste); L.marcas(oeste);
}
```

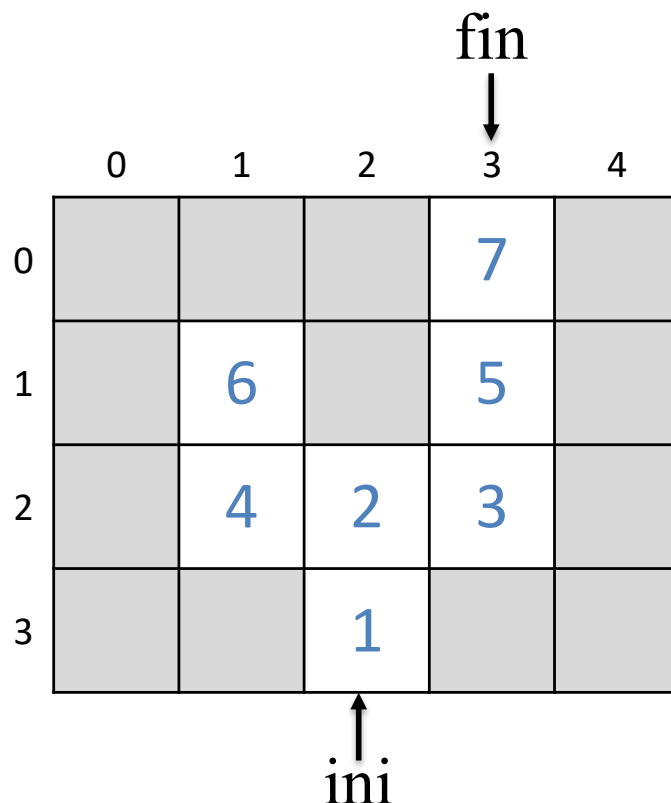
Buscar la salida de un laberinto usando una cola

// Pre: L.libre(ini), L.libre(fin) y todas las

// posiciones válidas de L están sin marcar

// Post: Retorna true si y solo si la cola está vacía

bool busca_camino(const Laberinto& L, pos ini, pos fin)



(3,2)	
(2,2)	
(2,3)	(2,1)
(2,1)	(1,3)
(1,3)	(1,1)
(1,1)	(0,3)
(0,3)	

Listas

Listas

Las listas son estructuras lineales que permiten:

- Recorridos secuenciales de sus elementos.
- Insertar elementos en cualquier punto.
- Eliminar cualquier elemento
- Concatenar una lista a otra.

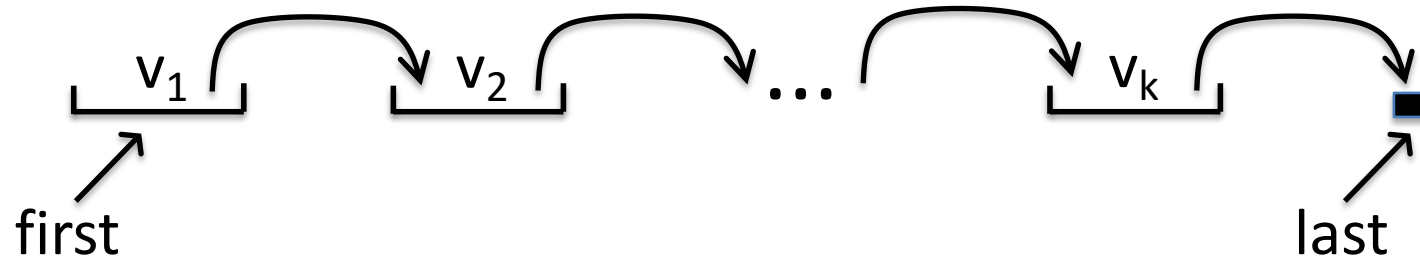
Contenedores e iteradores

- Un *contenedor* es una estructura de datos (un template) para almacenar objetos.
- Las listas son casos particulares de contenedores.
- Un *iterador*, es un objeto que designa un elemento de un contenedor para desplazarnos por él.

Iteradores: declaración (instanciación)

- Método **begin()**
- Método **end()**
- `list<Estudiant>::iterator it = l.begin();`
- `list<Estudiant>::iterator it2 = l.end();`
- "::": La definición del iterador pertenece al contenedor
- Si una llista **l** está vacía, entonces `l.begin() = l.end()`

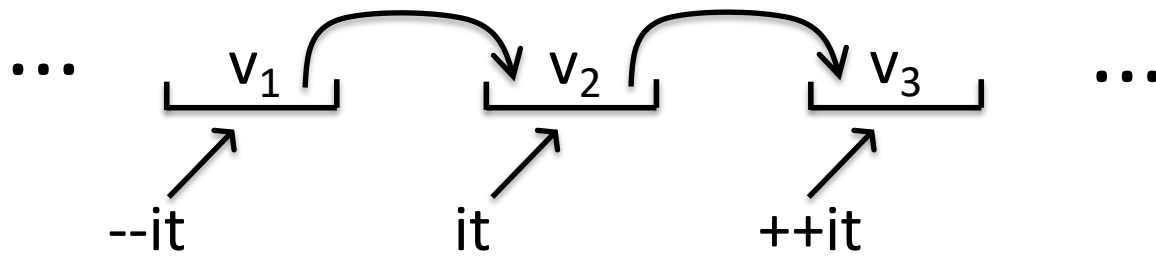
Iteradores



Operaciones con iteradores

- `it1 = it2;`
- `it1 == it2, it1 != it2`
- `*it` (si `it != l.end()`)
- `++it, --it` (salvo si estamos en `l.begin` o en `l.end()`)
- **NO:** `it + 3, it1 + it2, it1 < it2, ...`

Iteradores



Iteradores constantes

Los iteradores constantes prohíben modificar el objeto referenciado por el iterador. Por ejemplo:

```
list<Estudiant>::const_iterator it1 = l.begin();
```

```
list<Estudiant>::iterator it2 = l.end();
```

Estaría prohibido:

```
*it1 = ...;
```

pero no:

```
it1 = it2;
```

```
*it2 = ...;
```

Esquema frecuente

```
list<t> l;
```

```
...
```

```
list<t>::iterator it = l.begin();
```

```
while (it != l.end() and not  
    cond(*it)) {
```

```
    ... acceder a *it ...
```

```
    ++it; }
```

Imprimir una lista de estudiantes

```
void imprimir_llista(const list<Estudiant>& L) {  
    for(list<Estudiant >::const_iterator it = L.begin();  
        it != L.end(); ++it)  
        (*it).escriure();  
}
```

La clase list

```
template <class T> class list {  
public:  
  
// Subclases de la clase lista  
  
class iterator { ... };  
  
class const_iterator { ... };  
  
// Constructoras  
  
// Crea la lista vacía  
  
list();  
  
// Crea una lista que es una copia de original  
  
list(const list & original);
```

// Destructora:

~list();

// Modificadoras

// Pre: true

// Post: El PI pasa a ser la lista vacía

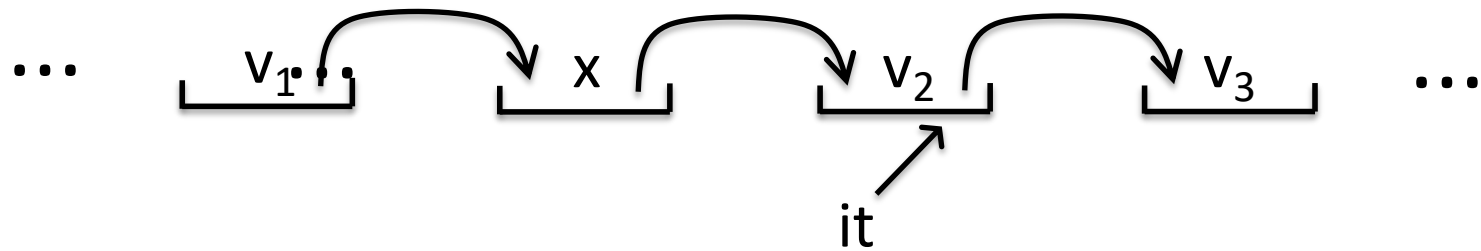
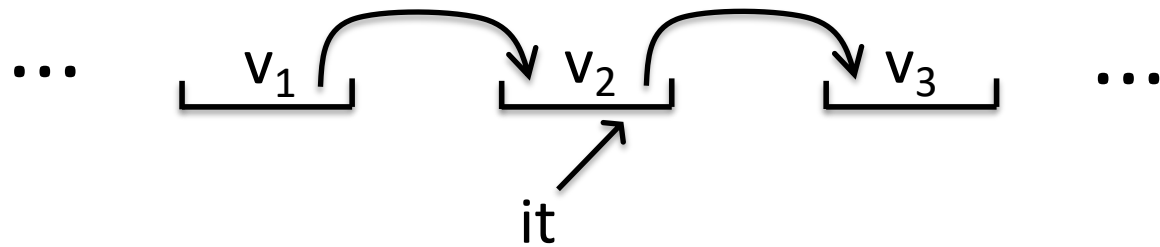
void clear();

// Pre: true

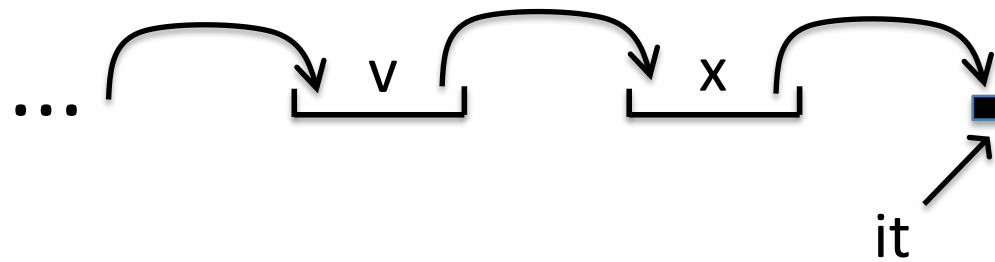
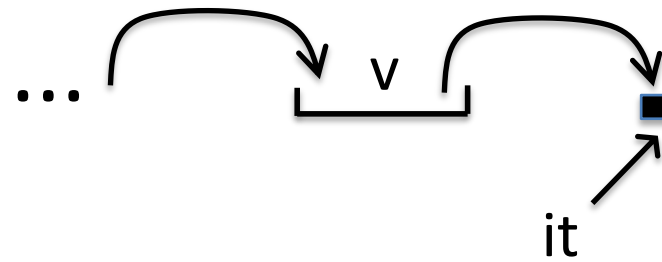
/* Post: Se inserta en el PI un nodo con el valor x delante de la posición apuntada por it */

void insert(iterator it, const T& x);

insert(it,x)

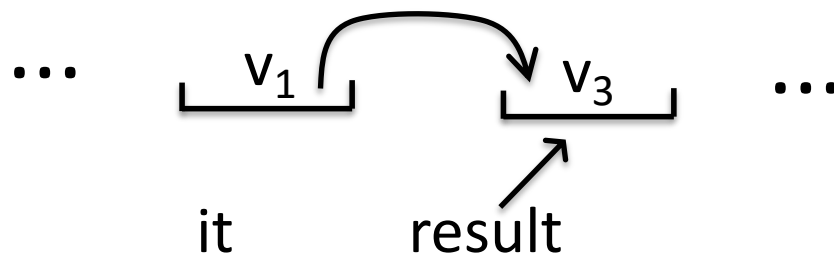
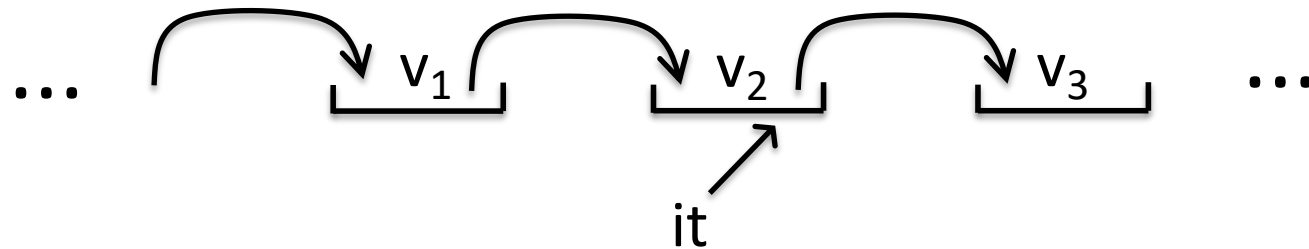


insert(it,x)



```
// Pre: it es distinto de end
/* Post: Se elimina del PI el elemento apuntado por it,
    it queda indefinido y se retorna la posición siguiente a
    la que apuntaba it */
```

```
iterator erase(iterator it);
```



```
// Es habitual
```

```
it = erase(iterator it);
```

```
// Pre: true  
/* Post: Se inserta en el PI toda la lista l delante del  
        elemento apuntado por it*/
```

```
void splice(iterator it, list& l);
```

```
// Consultoras
```

```
// Pre: true  
/* Post: Retorna si el PI es la lista vacia*/
```

```
bool empty() const;
```

```
// Pre: true  
/* Post: Retorna el numero de elementos del PI*/
```

```
int size() const;
```

Suma de los elementos de una lista de enteros

```
/* Pre: true */
```

```
/* Post: El resultado es la suma de los elementos de l */
```

```
int suma(const list<int>& l) {  
    int s = 0;  
    for (list<int>::const_iterator it = l.begin();  
         it != l.end(); ++it){  
        s = s + *it;  
    }  
    return s;  
}
```

Búsqueda en una lista de enteros

```
/* Pre: true */  
/* Post: El resultado indica si x está o no en l */  
bool pertenece(const list<int>& l, int x) {  
    list<int>::const_iterator it = l.begin();  
    while ((it != l.end()) and (*it != x))  
        ++it;  
    return it != l.end();  
}
```

Suma k a todos los elementos de una lista

```
/* Pre: l=[x1,...,xn] */
/* Post: l=[x1+k,x2+k,...,xn+k] */
void suma_k(list<int>& l, int k) {
    list<int>::iterator it = l.begin();
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
```

Inserción en una lista ordenada

```
/* Pre: L=[x1,...,xn], está ordenada */  
/* Post: L contiene a x, x1,...,xn, y está ordenada */  
void inserc_ordenada(list<int>& L, int x) {  
    list<int>::iterator it = L.begin();  
    while (it != L.end() and (x > *it) ++it;  
    L.insert(it,x);  
}
```

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

/* Pre: L1=[x1,...,xn], L2=[y1,...,ym] y las dos listas están ordenadas */

/* Post: L1 contiene x1,...,xn,y1,...,ym y está ordenada, L2 no se modifica*/

```
void inserc_ordenada(list<int>& L1, const list<int>& L2) {  
    list<int>::iterator it1 = L1.begin();  
    list<int>::iterator it2 = L2.begin();  
    while (it1 != L1.end() and it2 != L2.end() ) {  
        if (*it1 < *it2) ++it1;  
        else {L1.insert(it1,*it2); ++it2;  
    }  
    while (it2 != L2.end() ) {  
        L1.insert(it1,*it2);  
        ++it2;  
    }  
}
```


Vectores vs Listas

- Recorrido secuencial: tiempo lineal en los dos casos
- Acceso directo al i -ésimo elemento: constante en vectores, tiempo i en listas.
- Insertar un elemento es
 - constante en listas
 - Al final de un vector es constante (si no hay que reservar memoria adicional)
 - En medio de un vector es costoso
- Borrar un elemento es constante en listas y costoso en vectores
- Splice: constante en listas y costoso en vectores