list<T>

Descripción

`El TDA lista (list)` representa una lista doble enlazada. Es decir, permite almacenar una secuencia de elementos de tipo T permitiendo el recorrido hacia delante como hacia atrás. La insercion se puede realizar en el principio, al final o en posiciones intermedias de la lista en un orden constante (realizando un análisis amortizado). La inserción de elementos en la lista no invalida los iteradores. Tampoco el borrado invalida los iteradores, salvo los que apuntaban al elemento borrado. Sin embargo, el orden de los iteradores puede cambiar después de una operación de borrado o inserción.

Existe el TDA lista simplemente enlazada (`slist`) que sólo permite iterar hacia delante. En los casos en que no sea necesario utilizar el iterador hacia atrás `slist` es más eficiente que `list`

Definición

Definido en el fichero cabecera <list> o en <list.h> para compatibilidad con versiones anteriores.

Ejemplo

```
list<int> L;
L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
list<int>::iterator it;
for (it = L.begin(); it!= l.end(); ++it)
       cout << *it  << " ";
```

Parámetros de la plantilla

Parámetro	Descripción	Defecto
`T`	El tipo del objeto que se almacena en la lista.	

Requisitos del tipo

T debe tener el operador de asignación.

Miembros

Tipos		Descripción
`size_type`		Entero sin signo
`iterator`		Iterator sobre `lista`.
`const_iterator`		El iterator constante de una `lista`.
`reverse_iterator`		Iterator que recorre en orden inverso una `lista`.
`const_reverse_iterator`		Iterator constante que recorre en orden inverso una `lista`.
Miembros	**Eficiencia**	**Descripción**
`iterator begin()`	O(1)	Devuelve un `iterator` que señala al principio de la `lista`.
`iterator end()`	O(1)	Devuelve un `iterator` que señala al final de la `lista`.
`const_iterator begin() const`	O(1)	Devuelve un `const_iterator` que señala al principio de la `lista`.
`const_iterator end() const`	O(1)	Devuelve un `const_iterator` que señala al final de la `lista`.
`reverse_iterator rbegin()`	O(1)	Devuelve un `reverse_iterator` que señala al principio de la lista en orden inverso.
`reverse_iterator rend()`	O(1)	Devuelve un `reverse_iterator` que señala al final de la lista en orden inverso.
`const_reverse_iterator rbegin() const`	O(1)	Devuelve un `const_reverse_iterator` que señala al principio de la lista en orden inverso.
`const_reverse_iterator rend() const`	O(1)	Devuelve un `const_reverse_iterator` que señala al final de la lista en orden inverso.

`size_type size() const`	X	Devuelve el tamaño de la `lista`. Nota: no debes asumir que esta función es tiempo constante. Se permite que pueda ser *O (N*), donde está el número *N de* elementos en la `lista`. Si deseas probar si una `lista` es vacía, debes escribir L.empty() == true.
`size_type max_size() const`	O(1)	Devuelve el tamaño más grande posible de la `lista`.
`bool empty() const`	O(1)	`verdad` si el tamaño de la `lista` es `0`.
`list()`	O(1)	Crea una lista vacía.
`list(size_type n)`	O(n)	Crea una lista con n elementos, cada uno de ellos copia del elemento por defecto T().
`list(size_type n, const T& t)`	O(n)	Crea una lista con `n copias de t.`
`list(const list&)`	O(n)	El constructor de copia.
`~list()`	O(n)	El destructor.
`list& operator=(const list&)`	O(n)	El operador de asignación
`T & front()`	O(1)	Devuelve una referencia al primer elemento.
`const T & front() const`	O(1)	Devuelve una referencia constante al primer elemento.
`T & back()`	O(1)	Devuelve una referencia al último de la lista
`const T & back() const`	O(1)	Devuelve una referencia constante al último de la lista
`void push_front(const T&)`	O(1)	Inserta un nuevo elemento al principio de la lista

`void push_back(const T&)`	O(1)	Inserta un nuevo elemento en el final.
`void pop_front()`	O(1)	Quita el primer elemento.
`void pop_back()`	O(1)	Quita el elemento último elemento
`void swap(list&)`	O(1)	Intercambiar el contenido de dos listas.
`iterator insert(iterator pos, const T& x)`	O(1)	Inserta un elemento delante de la posición pos, Precondiciones: Pos debe ser una posición válida dentro de la lista Devuelve la posición del elemento insertado.
`Void insert(iterator pos, size_type n, const T& x)`	O(n)	Inserta n copias de `x` antes de la posicion `pos`.
`iterator erase(iterator pos)`	O(1)	Borra el elemento en la `posición pos`.
`iterator erase(iterator first, iterator last)`		Borra los elementos comprendidos en el rango `[first,last)`
`void clear()`	O(n)	Borra todos los elementos.
`void resize(n, t = T())`		Inserta o borra elementos con el valor t al final de la lista hasta que el tamaño se convierte en `N`.
`void splice(iterator pos, list& L)`	O(1)	`Pos` debe ser un iterador válido en `*this`, `L` debe ser una lista distinta de `*this`. (Es decir, se requiere que `&L != this`.) Todos los elementos de `x` se insertan anter de `pos` y son borrados de `L`. Todos los iteradores permanecen válidos, incluyendo los iteradores que apuntan a los elementos de L.
`void splice(iterator pos, list & L, iterator i)`	O(1)	Pos debe ser un iterador válido en *this, i debe ser un iterador en L. Splice mueve el elemento apuntado por i desde la lista L a *this, insertándolo antes de pos. Todos los iteradores permanecen válidos, incluyendo los iteradores que apuntan a los elementos de L. Si pos == i o pos==++i, esta función es una operación nula. Esta función es tiempo constante.
`void splice(iterator pos,list& L,`	O(1)	`pos` debe ser un iterator válido en `*this`, y `[f,l)` deben representar un rango válido en L. Splice mueve los elementos dentro del rango

`iterator f, iterator l)`		`[f,l)` desde L `a *this`, insertándolos antes de `pos`. Todos los iterators siguen siendo válidos, incluyendo los iteradores que apuntan a los elementos L
`void remove(const T& value)`	O(n)	Elimina todos los elementos con valor igual a `value`. El orden relativo de los elementos que no se quitan no se modifica y los iteradores que apuntan a los elementos que no se quitan sigue siendo válido.
`void unique()`	O(n)	Elimina todos los elementos, menos el primero en cada grupo consecutivo de elementos iguales. El orden relativo de los elementos que no se quitan no se modifica y los iteradores a los elementos que no se han borrado siguen siendo válidos.
`void merge(list& L)`	O(n+m)	Mezcla dos listas, `*this` y L. Ambas deben estar ordenadas según el `operator<`, y deben ser distintas (`&L!= this`.). Elimina todos los elementos de L y los inserta de forma ordenada en `*this`. El proceso de mezcla es estable (siun elemento de `*this` es equivalente a un elemento de L, entonces el elemento de `*this` precede al de L en el orden) . Todos los iteradores de `*this` y L permancen válidos.
`void sort()`	O(n log n)	Ordena los elementos de `*this` según `operator<`. La ordenación es estable, es decir, la posición relativa de elementos se preserva. Todos los iteradores siguen siendo válidos y continúan señalando a los mismos elementos.
`bool operator== (const list&, const list&)`		Operador de igualdad. Es una función global, no una función miembro.
`bool operator< (const list&, const list&)`		Comparación lexicográfica. Esto es una función global, no una función miembro.