ESTRUCTURA DE DATOS

STL: Contenedores

LINEALES

T.D.A.	Clase	Cómo incluirla
Vector Dinámico	`vector`	`#inlcude <vector>`
Lista	`list`	`#inlcude <list>`
Cola / Cola con prioridad	`queue / priority_queue`	`#inlcude <queue>`
Pila	`stack`	`#inlcude <stack>`
Doble Cola	`deque`	`#inlcude <deque>`
Vector Estático	`array`	`#inlcude <array>`
Listas enlace simple	`forward_list`	`#inlcude <forward_list>`

Functor: Funciones Objeto

Los functores son objetos que pueden ser tratados como funciones o punteros a funciones.

Listas

Funciones

I. **push_back(i)**: añade el elemento i al final de la lista
II. **iterator**: un objeto de la clase iterator recorre la lista de principio a final.
III. **reverse_iterator**: un objeto de la clase reverse_iterator recorre la lista de final a principio.
IV. **rbegin()**: método que devuelve un objeto reverse_iterator que apunta al último elemento válido de la lista
V. **rend()**: método que devuelve un objeto de reverse_iterator que apunta antes del elemento inicial de la lista.
VI. **size()**: devuelve el número de nodos de la lista
VII. **empty()**: nos dice si la lista está vacía
VIII. **front()**: nos devuelve el primer nodo de la lista
IX. **back()**: nos devuelve el último nodo de la lista
X. **pop_front()**: borra el primer elemento de la lista
XI. **pop_back()**: borra el último elemento de la lista. Ahora vamos a ver una función nueva: erase.
XII. **Erase.** Tiene dos versiones:
    A. **iterator erase (iterator posicion)**: borra un elemento con su posición

B. **iterator erase (iterator first, iterator end)**: elimina todos los elementos desde first hasta end, end no incluido

XIII. **assign**: según los parámetros, asigna a la lista unos valores u otros. Las posibilidades de llamada de assign son

A. **assign(iterator inicio,iterator fin)**: el inicio y el final de una serie de elementos dados por dos iteradores. Asigna a la lista los valores de ese rango.

B. **assign(int n,const T &v)**: un entero y un objeto de un tipo concreto. Asigna a la lista el segundo parámetro tantas veces como diga el primer parámetro.

XIV. **insert**: según los parámetros funciona de una manera u otra:

A. **insert(iterator it,int n,const T v)** tres parámetros: un iterador, un entero y un objeto de un tipo. Inserta en la posicion apuntada por el iterador el objeto tantas veces como diga el entero. El iterador sigue apuntando al valor que apuntaba antes de la inserción

B. **insert(iterator it,const T v)** dos parámetros: una posicion y un objeto. Inserta en la posición dada por el iterador el objeto y el iterador pasa a apuntar a ese objeto

C. **insert(iterator it_dest,iterator it_source_inicio,iterator it_source_fin)** tres parámetros: inserta en la primera posición, de la lista destino, todos los elementos desde la segunda hasta la tercera posicion del mismo u otro contenedor.

XV. **swap(l)**: intercambia dos listas. Necesita un único parámetro que es la lista con la que se intercambia.

XVI. **clear()**: deja la lista que llama a este método vacía

XVII. **Splice.** Mueve los elementos de una lista a otra. Según sus parámetros puede ser:

A. **splice(iterator it,lista l)** dos parámetros: un iterador y una lista. Coge todos los elementos de la lista l y los mueve a la lista con la que se llama a splice. Se inserta a partir de la posicion dada por it.La lista l queda vacía.

B. **splice(iterator itdes, lista l, iterator itsource)** Tres parámetros. Mueve el elemento apuntado por itsource de l a la lista con la que se llama al método y lo pone donde apunta itdes.

C. **splice(iterator itdes, lista l, iterator itsource_ini,iterator itsource_fin)** Cuatro parámetros: mueve todos los elementos apuntados desde itsource_ini hasta itsource_fin de l a la lista con la que se llama al método y se inserta a partir de la posición itdes.

XVIII. **Remove**. **void remove (const value &val)**: Elimina de una lista todos los elementos con valor val.

XIX. **Remove_if. void remove_if (function)**: Elimina los elementos que cumplan una determinada condición (functor).

XX. **Unique.** Elimina valores duplicados que se encuentran consecutivamente, dejando una única ocurrencia. Tenemos distintas formas de usar la función:

A. **unique()**: establece la igualdad con lo que esté predefinido en el operator==

B. **unique(function)**: establece la igualdad según la hayamos definido en function.

XXI. **Sort.** Con respecto a sort, que ordena los elementos, podemos pasarle una función que teniendo dos parámetros del tipo base de la lista devuelve un booleano indicando si se cumple o no una propiedad entre los dos parámetros de entrada.

A. **void sort();**

B. **void sort (function);**

C. **void sort (begin, end);**

D. **void sort (begin, end, function);**

XXII. **Merge.** Mezcla dos listas ordenadas. Si no le pasamos ningún argumento, usa el operator < predefinido para saber quién es menor, pero podemos pasarle una función para establecer la relación de orden. Vacía la lista que se pasa como argumento.

XXIII. **Reverse.** Invierte una lista.

Pair: par de valores

Es una estructura muy utilizada sobre todo cuando queremos representar datos que van en pares de valores diferentes.

Pilas.

Funciones típicas

I. **size:** número de elementos de la pila
II. **empty**: true si la pila está vacía, falso en caso contrario.
III. **top**: devuelve el elemento que está el la posición tope.
IV. **pop**: elimina el elemento del tope.
V. **push**: inserta un nuevo elemento por el tope

Colas

Las colas como ya vimos siguen la **política FIFO** (first input firts output) y la STL la implementa como la clase queue en la biblioteca con el mismo nombre.

Funciones típicas

I. **size**: número de elementos de la cola
II. **empty**: true si la pila está vacía, falso en caso contrario.
III. **front**: accede al elemento en el frente
IV. **push**: inserta un elemento por el final
V. **back**: accede al elemento por el final. (Esta no es una operación estándar de las colas).
VI. **pop**: elimina el elemento en el frente

Colas con prioridad

Permiten mantener una colección de elementos ordenados por su prioridad o preferencia. La stl implementa las colas con prioridad en la clase priority_queue en la biblioteca queue.

Funciones típicas

I. **size**: número de elementos de la cola
II. **empty**: true si la pila está vacía, falso en caso contrario.
III. **top**: devuelve el elemento mas prioritario
IV. **pop**: elimina el elemento mas prioritario
V. **push**: inserta un elemento en la posición dictada por su prioridad. Vamos a ver un ejemplo de uso:

Doble cola

La doble cola (deque) contiene secuencias de elementos que cambian de tamaño de forma dinámica. De esta forma un objeto de tipo doble cola se puede expandir y contraer por los dos extremos (por el principio y final). Son similares a los vectores, pero con una mejor eficiencia en los procesos de inserción y borrado de los elementos. A diferencia de los vectores, la doble cola no garantiza almacenar sus elementos en localizaciones de memoria contiguas. Aunque se pueden acceder de forma directa a cada elemento. Si las operaciones de inserción y borrado se hacen por los extremos las doble colas se comportan mejor que los vectores. En cambio

si estas operaciones se realizan en cualquier otro punto son menos eficientes que como se realizan en un vector. La doble cola se implementa en la STL en la clase deque en la biblioteca con el mismo nombre.

Array

Los arrays son contenedores de tamaño fijo: mantienen un número específico de elementos en una estructura lineal. Es mas eficiente que el vector en cuanto a almacenamiento, ya que no se expanden ni se contraen de forma dinámica. Esta estructura es la que conocemos como vector estático.

Forward List

Son contenedores de secuencias que permiten insertar y borrar en cualquier punto de la secuencia en tiempo constante. Se implementan como listas con celdas enlazadas simples. Así la diferencia fundamental entre forward_list y list es que mantienen, forward_list, un enlace al siguiente elemento, mientras que la list mantiene dos enlaces por elemento, uno apunta al siguiente y otro al anterior. Así que sobre una forward_list se puede iterar hacia adelante solamente. Al igual que la list la forward_list se muestra poco eficiente para acceder a un elemento por su posicion. Debes de iterar desde el principio para acceder por ejemplo al sexto elemento. Esta clase no contiene la funcion size para saber cuántos elementos tiene, por razones de optimización de espacio. Así que para saber cuántos elementos tiene una forward_list podemos usar el algoritmo distance con begin y end de la forward_list.

Contenedores asociativos

T.D.A.	Clase	Cómo incluirla
Conjunto / Bolsa	`set / multiset`	`#inlcude <set>`
Diccionario	`map / multimap`	`#inlcude <map>`
Tabla Hash	`unordered_map`	`#inlcude <unordered_map>`

DEFINICIÓN

Contenedor Asociativo: Es una colección de pares, en la que cada par se conforma de una **clave** y **valor**. Puede que no aparezca valor asociado a la clave y puede que el valor clave aparezca más de una vez para asoiciarle diferentes valores. Las operaciones más frecuentes de estos contenedores son:

- Añadir un par a la colección
- Eliminar un par de la colección
- Modificar un par existente
- Buscar un valor asociado con un determinada clave

Estás operaciones son las operaciones típicas de un diccionario. En la STL de C++ los contenedores asociativos ordenados que vamos a estudiar son:

- **Set/Multiset**: conjunto de claves (no llevan valor asociado), con la posibilidad de repetir clave (multiset) o no.
- **Map/Multimap**: conjunto de pares, (clave, valor asociado). En el map solamente se permite una entrada por clave (no se repiten), en el multimap se permite varias entradas para una misma clave (se pueden repetir).

Debajo de estos T.D.A. se ha usado para su implementación Árboles. Tanto set/multiset y map/multimap se mantienen los datos ordenados por el valor de la clave. Otros contenedores asociativos que no están ordenados por clave son las tablas hash que en la STL se representan como:

- **unordered_set/unordered_multiset**
- **unordered_map/unordered_multimap**

Set/multiset

La clase set representa un conjunto de elementos que se disponen de manera ordenada y en el que no se repiten elementos. Los datos que insertamos en el set se llaman claves. La clase multiset es lo mismo pero permite incluir elementos repetidos en el conjunto.

Funciones

I. **Count.** Devuelve el número de elementos que son iguales a un valor de entrada. Si lo ejecutamos con un set como mucho obtendremos un 1, pero si lo ejecutamos con un objeto multiset, podemos obtener cualquier valor mayor o igual que cero.
II. **Swap.** Intercambia dos conjuntos.
III. **Find.** Busca en el conjunto un elemento y devuelve un iterador al elemento. Si no lo encuentra, devuelve un iterador a end().
IV. **Equal_range. pair<iterator, iterator> equal_range (const value_type & val) const**: El primer iterador apunta al primer elemento del conjunto que coincida con val y el segundo, al primer elemento del conjunto distinto de val.
V. **lower_bound** devuelve un iterador al primer elemento que coincide con el elemento de entrada, si no existe devuelve un iterador al valor que va antes más proximo. En el caso que todos sean menores devuelve end().
VI. **upper_bound** devuelve un iterador al primer elemento mayor al valor de entrada, si no existe devuelve un iterador al mayor más próximo.
VII. **Value_comp** Devuelve un objeto comparador de set. El objeto se puede usar para comparar dos elementos del conjunto de manera que al comparar, el objeto comparador devuelve true si el primer objeto es menor que el segundo

Map/multimap

Un map está formado por **parejas de valores**: al primero se lo conoce como clave, y al segundo como el valor asociado a dicha clave. No permite valores de clave repetidos y se ordena según su clave. Podemos acceder, de forma directa, al valor asociado a la clave a través de la clave, pero no al revés. Para poder acceder a la clave a través del valor asociado a ésta hay que realizar una búsqueda secuencial por valor asociado. Los multimap son lo mismo pero permiten valores de clave repetidos. Usan los **mismos métodos que set y multiset**. En un map podemos usar el **operator[]** para acceder a los elementos asociados a la clave o modificarlos.

Contenedores asociativos no ordenados

Introducción

Este tipo de contenedores sirven para representar las **tablas hash**. La clasificación de estos contenedores se hará: 1) dependiendo si admiten valores repetidos o no; 2) y si las claves tienen valores asociados

Los elementos se encuentran en un orden particular, y la recuperación de los mismos se hacen por su valor de una forma muy rápida. En un conjunto no ordenado el valor de un elemento es a su vez su llave. Este valor una vez insertado en el conjunto no ordenado no puede modificarse. Solamente podemos insertar, consultar y eliminar. Internamente los elementos no están ordenados, pero se organizan en cubetas dependiendo del valor hash asociado. Por lo tanto este tipo de contenedor es el mas eficiente para acceder a elementos individuales pero no es eficiente cuando se quiere consultar un rango de valores. Los datos se almacenan en cubetas. Todos aquellos datos que tengan la misma función hash se almacenan en la misma cubeta. Por lo tanto ocurre colisión cuando la función hash para dos claves diferentes devuelve la misma cubeta.

Factor de Carga: Razón entre el número de elementos del contenedor y el número de cubetas (valor obtenido con la función bucket_count).

Funciones relevantes

Funciones de capacidad:

I. **empty**: comprueba si el contenedor está vacío
II. **size**: devuelve el tamaño del contenedor
III. **max_size**: devuelve el máximo tamaño del contenedor

Iteradores

IV. **begin**: devuelve un iterador al principio del contenedor
V. **end**: devuelve un iterador al final
VI. **cbegin**: devuelve un iterador constante al principio del contenedor
VII. **cend**: devuelve un iterador constante al final

Consulta

VIII. **find**: obtiene un iterador el al elemento
IX. **count**: nos da el número de elementos con un valor determinado (0 o 1).
X. **equal_range**: consigue un rango de elemento con una llave específica.

Modificadores

XI. **insert**:inserta elementos
XII. **erase**: elimina elementos
XIII. **clear**: limpia el contenido
XIV. **swap**: intercambia el contenido Cubetas
XV. **bucket_count**: devuelve el número de cubetas
XVI. **max_bucket_count**: devuelve el número máximo de cubetas
XVII. **bucket_size**: devuelve el tamaño de la cubeta
XVIII. **bucket**: localiza la cubeta de un elemento

Aspectos de la función hash

XIX. **load_factor**: devuelve el factor de carga.
XX. **max_load_factor**: máximo factor de carga.
XXI. **rehash**: modifica el número de cubetas
XXII. **reserve**: solicita un cambio de capacidad

Observadores

XXIII. **hash_function**: obtiene la funcion hash

XXIV. **key_eq**: toma dos elementos y devuelve un booleano indicadno si lo elementos son equivalentes porque tienen la misma funcion hash

Unordered_set/Unordered_multiset

Son contenedores que almacenan claves no repetidas (unordered_set) o si permiten claves repetidas tenemos el contenedor unordered_multiset. Para poder usar estos contenedores debemos hacer el include de la biblioteca unordered_set.

Unordered_map/Unordered_multimap

Almacenan elementos formados por la combinación valor clave y valor asociado. Estos contenedores son aconsejables cuando se necesita búsquedas rápidas por clave. Al igual que en los unordered_set los pares clave valor asociado, se almacenan internamente en la cubeta determinada por el valor hash asociado a la clave. Las funciones que hemos visto anteriormente para Unordered_set/Unordered_multiset son aplicables también para Unordered_map/Unordered_multimap.