

Contenedores Lineales

- Las estructuras de datos lineales se caracterizan porque consisten en una secuencia de elementos del mismo tipo, a_0, a_1, \dots, a_n , dispuestos a lo largo de una dimensión
- Ejemplo: Vector dinámico o celdas enlazadas
- Los contenedores lineales se comportan de forma idéntica independientemente del tipo de dato que almacenen (candidatos a ser implementados como clases plantillas)
 - Contenedor: estructura que almacenan datos de un mismo tipo base.
 - Lineal: contiene una secuencia de elementos dispuestos en una dimensión.

Pilas

Son un tipo de ED lineales que se caracterizan por su comportamiento LIFO; donde todas las inserciones y borrados se realizan en un extremo de la pila llamado **tope**. Si queremos que acceder a otra posición distinta al tope de la pila debemos usar otros tipos de datos.

Especificación

Contiene una secuencia de elementos haciendo en la que las inserciones, consultas y borrados se realizan sólo por uno de los extremos.

- Sobre **qui** {
 - Consultas
 - Inserciones
 - Borrados

- No se puede acceder a la pila por otro lado que no sea el tope

Operaciones

- **Tope**: consulta el elemento del tope
- **Empty**: devuelve true si la pila está vacía.
- **Pop**: elimina el elemento del tope
- **Push**: inserta un elemento en el tope.

Celdas enlazadas

Esquema de la interfaz

```
#ifndef __PILA_H__  
#define __PILA_H__  
  
class Pila{  
private:  
    ... //La implementación que se elija  
  
public:  
    Pila();  
    Pila(const Pila & p);  
    ~Pila() = default;  
    Pila & operator=(const Pila & p);  
  
    bool vacia() const;  
    void poner(const Tbase & c);  
    void quitar();  
    Tbase tope() const; → Tbase & tope();  
};  
const Tbase & tope() const;  
  
#endif /* Pila.hpp */
```

Podríamos sobrecargar quitar() y poner() en los operadores -- y +=

Implementación

- Basada en vectores estáticos
- Basada en vectores dinámicos
- Basada en celdas enlazadas.

Colas

Una cola es una estructura por la cual se inserta y borran los elementos por extremos opuestos.
Tiene un comportamiento FIFO.

1. Especificación:

- Contiene una secuencia de datos $\{a_0, a_1, a_n\}$ especialmente diseñada para hacer las inserciones por un extremo y los borrados y consultas por otro.
- El extremo en el que están el primer elemento $\{a_0, a_1, \dots\}$ se llama **frente**, y es por el que se hacen las consultas y borrados.
- El extremo en el que están los últimos valores (a_n) se llama **última** y es por el que se realizan las inserciones.
- Las colas responden a la política FIFO (First In, First Out).

2. Operación:

- Frente: consulta o accede al elemento en el frente
- Vacía: devuelve true si la cola está vacía
- Quitar (pop): elimina el elemento que está en el frente
- Poner (push): inserta un nuevo elemento por el final (la posición última)

3. Implementación:

- Dinámica: Basada en celdas enlazadas y dos punteros (para que todas las operaciones sean $O(1)$)
- Estática: Basada en vectores (**circulares**)

Colas con prioridad.

Es una estructura de datos igual diseñada para realizar accesos y borrados en uno de sus extremos. Las inserciones se realizan en cualquier posición, de acuerdo a su valor de prioridad.

1. Especificación:

- Contienen una secuencia de valores especialmente diseñadas para realizar accesos y borrados por el frente.
- A diferencia de las colas normales, las inserciones se realizan en cualquier punto de acuerdo a un criterio de prioridad.
- Cada dato que se guarda en la cola con prioridad debe componerse del dato y su prioridad.

2. Operación:

- Frente: devuelve el elemento en el frente
- Prioridad: devuelve la prioridad del elemento en el frente
- Quitar: elimina el elemento que está en el frente. Este es el más prioritario.
- Vacía: indica si la cola está vacía
- Poner: inserta un nuevo elemento de acuerdo a su prioridad

3. Implementación:

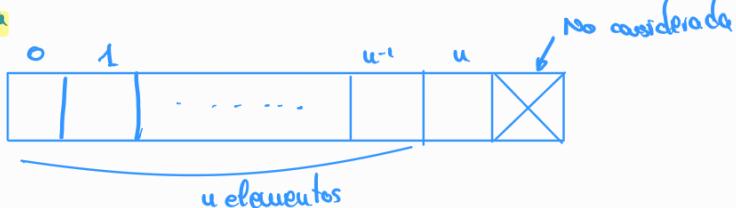
- Al igual que con las colas simples podríamos plantearnos diferentes estructuras de datos: vector dinámico, vectores circulares, celdas enlazadas.

Clistas

Es una estructura de datos lineal que contiene una secuencia de elementos diseñada para insertar, borrar y acceder a cualquier posición.

- Set
- Get
- Borrar
- Insertar
- Nuevos elementos

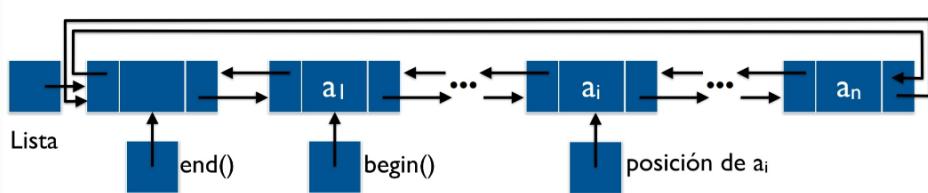
Caso dato importante, consideremos u+1 posiciones y u elementos, donde incluir la siguiente posición a la última.



Hay varias formas de implementación, vectores, celdas y la que se debe usar, celdas delemente entrelazadas con cabecera y circulares.

Para su implementación usual, necesitamos la clase Posición, donde tenemos un elemento elegido y nos ayudará a movernos por nuestra estructura. El inicio se conseguirá con begin() → posición del primer elemento y el final con end() que devuelve la posición después del último elemento.

Implementación con celdas delemente entrelazadas circulares:



- Lista = puntero o cabecera
- Posición (iterador) = puntero dentro de celda
- Insertiones y borrar independientes de la posición. (reasignación de punteros)

Iteradores

Para trabajar con ellos, vamos a realizar 4 pasos:

1. Inicializar en la primera posición del iterador.

2. Usar *it para acceder a su contenido.

3. Avanzar usando ++it.

4. Para saber si hemos acabado comparamos con it.end().

Tipos

- Input (lectura/palabra)
- Output (escritura/palabra)
- Forward (uso + escritura/palabra)
- Bidirectional (Forward+palabra)
- Random access (bidireccional)

Los iteradores pueden ser constantes y el lugar de la imposibilidad de modificar el vector que contiene.

STANDARD TEMPLATE LIBRARY (STL)

Tareas Comunes

- Substrar datos
- Organizar datos
- Recuperar datos
- Insertar datos
- Eliminar datos
- Ordenar y buscar datos
- Operaciones aritméticas simples

Idea principal

Usar tipos de datos genéricos para crear códigos más versátiles independientes del tipo de dato usado.

Ideas secundarias

- Código legible y de formato razonable
- Acceso uniforme a los datos
- Manejo sencillo y estructurado de los datos.
- Versátiles estandar de algoritmos muy usados.

Ventajas

- Ofrece nuevos **contenedores** (crece y decrece de tamaño automáticamente)
- Proporciona **algoritmos e iteradores** para el manejo de los contenedores.
- Es extensible (permite crear uno)

Componentes

1. **Contenedores** (clases template genéricas)
2. **Algoritmos** (funciones template genéricas)
3. **Iteradores** (generalización del concepto de puntero)

Secuenciales (estructuras lineales, primera clase)

Asociativos (no lineales y buscarrápido, clave/valor, primera clase).

Quasi (contenedores con funcionalidad reducida)

TypeDefs:
No se suele usar

- value-type
- iterator
- const_iterator
- reverse_iterator
- size-type

Contenedores

Secuenciales

Almacenar datos en una secuencia:

- **vector**: array unidimensional dinámico (si hay muchos elementos, si insertamos pedimos factor -1) bit.
- **list**: lista doblemente enlazada (nunca tiene que iteradores) inserciones
- **deque**: colado doble enlazado doble salto
- **vector + list** (solamente es bueno de concatenación)
- **Acceso por índices** → **Inserciones y borrar elementos**

TDA	Clase	#include
Vector Dinámico	vector	#include<vector>
Lista	list	#include<list>
Cola/Cola con prioridad	queue / priority_queue	#include<queue>
Pila	stack	#include<stack>
Doble Cola	deque	#include<deque>
Vector Estático	array	#include<array>
Listas enlazadas simples	forward_list	#include<forward_list>

Asociativas ordenadas todos
 Usar una clave para acceder a los elementos:
 → Árbol binario + función `less<T>` → Claves compatibles con el tipo de datos
 multiset ← **Set**: almacena elementos con claves que se pueden ordenar. (sin rep)
 multimap ← **Map**: almacena pares (clave, valor) → Llaves ordenadas por clave. (sin rep)
 → insertarlos con par
 • Son compatibles con iteradores bidireccionales
 • Todos tienen: constructor primitivo, desplaza con rango y su rango, destruir, empty, clear, swap, size, reserve para el acceso a elem. operadores buscar. Iteradores y funciones esenciales
 Para tratar entradas como una colección
 . mantienen compatibilidad de tipos

Reserve vs Reserve

Ambas modifican el tamaño del vector pero:

Reserve (u)

- Reserva memoria sin inicializar
- No da valor por defecto

Reserve (u)

- Cambia el tamaño del vector para contener elementos
- Da valor por defecto

Deque

• Espacio adicional para una deque se reservan usando bloques de memoria. (array dinámico o deque)

- Métodos reservar + push_front, pop_front.

Adaptadores de contenidos

Tenemos 3 tipos:
 - stack
 - Queue
 - Priority queue
 que se implementan con los demás contenidos; priority, stack, list.

no son estructuras nuevas pero sí impone restricciones sobre una estructura de datos ya existente.

Stack (pila)

Usan mecanismo LIFO y todo se hace en el top de la estructura. Como implementaciones podemos usar vector, list o deque (por defecto). Incluir estándar.

Queue

Usa mecanismos FIFO, se inserta por un lado y se consulta por el otro. Incluir **agregar** y **quitar**. Puede implementarse **stack** o **deque** (por defecto)

Priority-queue

Es una cola donde se inserta en cualquier posición manteniendo un orden de prioridad, se suele usar un función para ordenar. La salida de los datos se realiza por un extremo de la cola. Es importante no confundir la prioridad de un elemento con su valor porque por lo general no suelen ser medibles los valores elementales. Tener **queues** y se pueden implementar por **vectores** o **deques**.

Introducción a los algoritmos

- Toma una **secuencia** a la que aplica operaciones. Si hay fallo devuelve **obj.error()**.
↳ **iteradores** ↳ **fuentes**

Modelo básico

Consiste en **usar iteradores** para definir secuencias desde `begin()` hasta `end()` (el cual no se procesa). Como operaciones básicas que soporta `++ -- * =`
↳ y su valor es **borra**

Algunos algoritmos estándar muy usados

- `r=find(b,e,v)` r apunta a la primera ocurrencia de v en [b,e]
- `r=find_if(b,e,p)` r apunta al primer elemento x en [b,e] para el que el predicado p es cierto ($p(x) \text{ true}$)
- `x=count(b,e,v)` x es el número de ocurrencias de v en [b,e]
- `x=count_if(b,e,p)` x es el número de elementos en [b,e] para los cuales el predicado p es cierto ($p(x) \text{ true}$)
- `sort(b,e)` ordena [b,e] usando <
- `sort(b,e,p)` ordena [b,e] usando el predicado p
- `copy(b,e,b2)` copia [b,e] en $[b2, b2+(e-b)]$ debe haber suficiente espacio después de $b2$
- `unique_copy(b,e,b2)` copia [b,e] en $[b2, b2+(e-b)]$ pero no copia duplicados adyacentes
- `merge(b,e,b2,e2,r)` mezcla dos secuencias ordenadas $[b2,e2]$ and $[r,r+(e-b)+(e2-b2)]$
- `r=equal_range(b,e,v)` r es la subsecuencia de [b,e] con el valor v en todas las posiciones (hace una búsqueda binaria para encontrar v)
- `? equal(b,e,b2,e2)` ¿Son iguales todos los elementos de [b,e] y $[b2,b2+(e-b)]$?

Visión general de los algoritmos

Operaciones sobre secuencias

- Sin modificar: `for_each`, `find`, `count`, `search`, `mismatch`, `equal`
- Modificando: `transform`, `copy`, `swap`, `replace`, `fill`, `generate`, `remove`, `unique`, `reverse`, `rotate`, `random_shuffle`

Operaciones sobre secuencias ordenadas

- `sort`, `lower_bound`, `upper_bound`, `equal_range`, `binary_search`, `merge`, `includes`, `set_union`, `intersection`, `set_difference`, `set_symmetric_difference`

Algunas funciones útiles son `find()` (que busca independientemente de donde), `find_if()` (busca según una condición) o `copy()` (copia otras estructuras en otras)

Predicados y funciones

Un **predicado** es una **función** o una **objeto función** (función) que toma un argumento y devuelven **bool**
↳ `struct Odd`
`bool operator()(int i) const {return i%2 != 0;}`

Functores

Es un objeto de una clase templatizada que tiene una sola función miembro, ésta puede estar predeterminada o definida por el usuario.

Cualquier algoritmo útil puede ser parametrizado por functores

Functores predeterminados	Tipo
<code>divides< T ></code>	aritmético
<code>equal_to< T ></code>	relacional
<code>greater< T ></code>	relacional
<code>greater_equal< T ></code>	relacional
<code>less< T ></code>	relacional
<code>less_equal< T ></code>	relacional
<code>logical_and< T ></code>	lógico
<code>logical_not< T ></code>	lógico
<code>logical_or< T ></code>	lógico
<code>minus< T ></code>	aritmético
<code>modulus< T ></code>	aritmético
<code>negate< T ></code>	aritmético
<code>not_equal_to< T ></code>	relacional
<code>plus< T ></code>	aritmético
<code>multiplies< T ></code>	aritmético

Algunos predeterminados de <functional>

Functores Lambda

Son funciones que se pueden pasar a otras funciones y se usan para definirlas:

auto func = [] () {cout << "Hello world";}

Y para llamarlos: func();

Functores vs Lambda

• Fuctor como argumento si: { - hacer algo complejado
- hacer lo mismo en varios sitios

• Lambda como argumento si: - hacer algo rápido y sencillo.

• Lambdas son más rápidas que los argumentos de functores predeterminados.

→ Configuraciones prediseñadas
se refiere a puntero a función