UNIVERSIDAD DE GRANADA

Estructura de Datos

Temario de la asignatura

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Estructura de Datos

Ismael Sallami Moreno

https://elblogdeismael.github.io

Índice general

1 Introducción a la eficiencia de los algoritmos **7**

1.1 Por qué se mide y no se cronometra **7**

1.2 El modelo de coste **7**

1.3 Notaciones asintóticas **8**

1.4 Casos mejor, peor y medio **9**

1.5 Coste amortizado **10**

1.6 Recurrencias **10**

1.7 Eficiencia espacial **10**

1.8 Eficiencia teórica frente a empírica **11**

1.9 Ejercicios **11**

2 Abstracción de datos **13**

2.1 El problema que resuelve **13**

2.2 Especificar un TDA **13**

2.3 Encapsulamiento en C++ **14**

2.4 Constructores, destructor y copia **15**

2.5 Plantillas: un TDA para cualquier tipo **16**

2.6 Iteradores **17**

2.7 Relaciones entre TDAs **17**

2.8 Ejercicios **18**

3 Tipos de datos contenedores básicos **19**

3.1 Vectores **19**

3.2 Listas enlazadas **20**

3.3 Pilas **20**

3.4 Colas **21**

3.5 Colas con prioridad **22**
3.6 Conjuntos **22**
3.7 Diccionarios **23**
3.8 Cómo se elige **23**
3.9 Ejercicios **23**

4 Tipos de datos contenedores complejos **25**
4.1 Árboles **25**
4.2 Tablas hash **28**
4.3 Grafos **30**
4.4 Ejercicios **31**

5 Seminarios **33**
5.1 La biblioteca estándar **33**
5.2 Algoritmos **34**
5.3 Aplicación de los TDA a problemas reales **35**
5.4 Errores frecuentes con la STL **36**
5.5 Elegir contenedor: resumen **36**
5.6 Ejercicios **36**

6 Temario práctico **39**
6.1 Práctica 1. Eficiencia de algoritmos **39**
6.2 Práctica 2. Construcción de TDA básicos **40**
6.3 Práctica 3. Uso e implementación de TDA lineales **40**
6.4 Práctica 4. Uso e implementación de TDA no lineales **41**
6.5 Sobre la memoria de prácticas **41**

Capítulo 1

Introducción a la eficiencia de los algoritmos

Tema 1 del programa. Antes de elegir una estructura de datos hay que saber medir qué cuesta usarla, y esa medida es la que decide entre un vector y una tabla hash mucho más que cualquier preferencia de estilo.

1.1 Por qué se mide y no se cronometra

Cronometrar un programa responde a la pregunta equivocada. El tiempo que marca el reloj depende del procesador, del compilador, de las opciones de optimización, de lo que la caché tenga cargado y de qué más esté ejecutando la máquina. Dos ejecuciones del mismo binario sobre los mismos datos dan cifras distintas.

Lo que se quiere saber es otra cosa: **cómo crece el coste cuando crece la entrada**. Esa pregunta tiene respuesta independiente de la máquina, y es la que permite decidir antes de escribir el código.

Pregunta	Se responde con
¿Cuánto tarda este programa en mi portátil?	cronometrando
¿Qué pasa si los datos se multiplican por diez?	analizando su eficiencia
¿Cuál de estas dos implementaciones escala mejor?	analizando, y confirmando midiendo

Las dos cosas se hacen, y en la práctica se contrastan. El apartado final del tema y la primera práctica de la asignatura consisten justamente en eso: comparar la eficiencia teórica con la empírica y explicar dónde se separan.

1.2 El modelo de coste

Se cuenta el número de **operaciones elementales** que ejecuta el algoritmo en función del tamaño de la entrada, que se llama n. Una operación elemental es la que cuesta un tiempo acotado por una constante: una asignación entre tipos simples, una comparación, una operación aritmética, un acceso a una posición de un vector.

Bajo ese modelo, el coste de las construcciones del lenguaje se compone así:

Construcción	Coste
Secuencia de instrucciones	la suma de sus costes

Construcción	Coste
Condicional	el coste de la condición más el peor de las dos ramas
Bucle que se repite k veces	k veces el coste del cuerpo
Bucles anidados	el producto
Llamada a función	el coste de la función

Dos precauciones que se saltan a menudo:

- **Un acceso a un vector cuesta lo mismo esté donde esté**, pero recorrer una lista enlazada hasta la posición i cuesta i pasos. El corchete no siempre vale lo mismo, y de ahí sale la mitad de las decisiones de este temario.
- **Una llamada a una función de biblioteca no es una operación elemental.** Insertar en mitad de un `vector` de la STL mueve todo lo que hay detrás. Si se cuenta como un paso, el análisis sale mal.

1.3 Notaciones asintóticas

Interesa el comportamiento cuando n crece, así que se descartan las constantes multiplicativas y los términos de orden inferior. Las tres notaciones habituales:

Definición 1.1 (Cota superior). $f(n) \in O(g(n))$ si existen constantes $c > 0$ y n_0 tales que $f(n) \leq c\, g(n)$ para todo $n \geq n_0$.

Definición 1.2 (Cota inferior). $f(n) \in \Omega(g(n))$ si existen constantes $c > 0$ y n_0 tales que $f(n) \geq c\, g(n)$ para todo $n \geq n_0$.

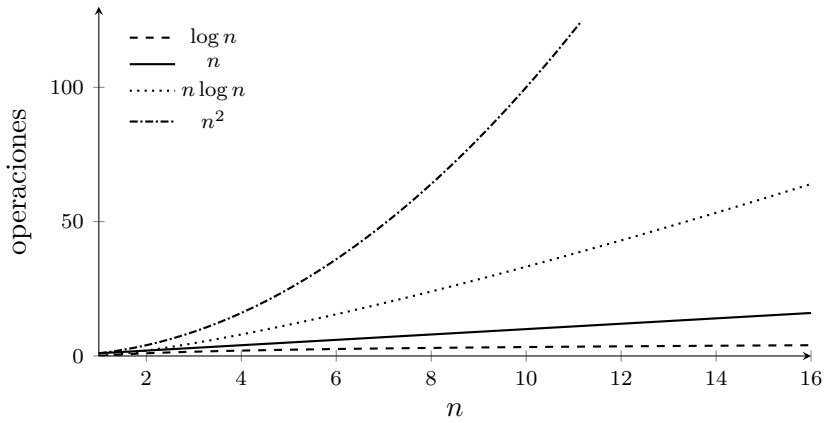
Definición 1.3 (Orden exacto). $f(n) \in \Theta(g(n))$ si $f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n))$.

En el uso corriente se escribe O donde se quiere decir Θ. No es grave mientras la cota sea ajustada, pero conviene saber que decir «este algoritmo es $O(n^2)$» no descarta que sea también $O(n^3)$: una cota superior sigue siendo cierta aunque sea mala.

1.3.1 Las órdenes que aparecen

Orden	Nombre	$n = 10^6$ da
$O(1)$	constante	1 operación
$O(\log n)$	logarítmico	unas 20
$O(n)$	lineal	10^6
$O(n \log n)$	casi lineal	$2 \cdot 10^7$
$O(n^2)$	cuadrático	10^{12}
$O(2^n)$	exponencial	inabordable

La tabla explica de un vistazo por qué este temario existe. Con un millón de elementos, un algoritmo cuadrático necesita un billón de operaciones y uno logarítmico veinte. La diferencia entre buscar en un vector desordenado y buscar en un árbol equilibrado es exactamente esa.



1.3.2 Reglas de manejo

Con $f \in O(n^a)$ y $g \in O(n^b)$:

Composición	Orden resultante
$f + g$	$O(n^{\text{máx}(a,b)})$
$f \cdot g$	$O(n^{a+b})$
$c \cdot f$, con c constante	$O(n^a)$

De la primera sale la regla que más se usa: **en una suma manda el término mayor**. Un algoritmo que ordena en $O(n \log n)$ y luego recorre el resultado en $O(n)$ es $O(n \log n)$; el recorrido no cambia nada.

1.4 Casos mejor, peor y medio

El coste rara vez depende solo del tamaño. Buscar un elemento en un vector desordenado recorriendo desde el principio cuesta:

Caso	Cuándo	Coste
Mejor	está el primero	$\Theta(1)$
Peor	está el último, o no está	$\Theta(n)$
Medio	uniformemente distribuido	$\Theta(n)$

El **peor caso** es el que se usa por defecto, porque es el único que garantiza algo. El caso medio exige hipótesis sobre la distribución de las entradas, y esas hipótesis a menudo son falsas: los datos reales llegan casi ordenados con más frecuencia de la que un modelo uniforme predice.

Nota 1.1 . Que el caso medio y el peor coincidan en orden, como en la búsqueda lineal, no es lo habitual. En quicksort el caso medio es $\Theta(n \log n)$ y el peor $\Theta(n^2)$, y la diferencia entre los dos es lo que hace que la elección del pivote importe tanto.

1.5 Coste amortizado

Hay operaciones que casi siempre son baratas y de vez en cuando muy caras. Añadir al final de un vector dinámico es el ejemplo canónico: normalmente cuesta $O(1)$, pero cuando se agota la capacidad hay que reservar un bloque mayor y copiar todo, que cuesta $O(n)$.

Analizar solo el peor caso diría que la operación es $O(n)$, y sería engañoso. El **análisis amortizado** reparte el coste de las operaciones caras entre las baratas que las rodean.

Ejemplo 1.1 . Si al llenarse la capacidad se dobla, una secuencia de n inserciones provoca redimensiones de tamaños $1, 2, 4, \ldots, n$. El total copiado es $1 + 2 + 4 + \cdots + n < 2n$, así que las n inserciones cuestan $O(n)$ entre todas y cada una sale a $O(1)$ amortizado.

El detalle importa: si en vez de doblar se aumentase en una cantidad fija, las redimensiones serían $\Theta(n)$ en número y el total $\Theta(n^2)$. **Doblar es lo que hace que la operación sea barata**, y es lo que hace la STL.

1.6 Recurrencias

Los algoritmos recursivos dan lugar a ecuaciones de recurrencia. La forma más frecuente en esta asignatura es la que produce dividir el problema:

$$T(n) = a\,T(n/b) + f(n)$$

con a subproblemas de tamaño n/b y $f(n)$ el coste de dividir y combinar.

Algoritmo	Recurrencia	Solución
Búsqueda binaria	$T(n) = T(n/2) + \Theta(1)$	$\Theta(\log n)$
Mergesort	$T(n) = 2T(n/2) + \Theta(n)$	$\Theta(n \log n)$
Recorrido de un árbol	$T(n) = 2T(n/2) + \Theta(1)$	$\Theta(n)$
Quicksort, peor caso	$T(n) = T(n-1) + \Theta(n)$	$\Theta(n^2)$

La primera fila es la que sostiene medio temario: **partir por la mitad de forma repetida da un coste logarítmico**, y de ahí que las estructuras de búsqueda eficientes sean todas, de una forma u otra, particiones sucesivas del conjunto.

1.7 Eficiencia espacial

El coste en memoria se analiza igual, y a veces decide. Tres cantidades distintas que conviene no confundir:

Cantidad	Qué mide
Espacio de los datos	lo que ocupan los elementos
Sobrecarga de la estructura	punteros, cabeceras, huecos reservados
Espacio auxiliar	lo que el algoritmo necesita además de la entrada

Una lista enlazada de enteros de 4 bytes gasta 8 o 16 bytes más por nodo solo en punteros. Con

un millón de elementos, eso es más memoria en punteros que en datos. Es la primera razón para preferir un vector cuando el acceso por posición basta.

Y el espacio auxiliar es lo que separa mergesort de quicksort: los dos son $\Theta(n \log n)$ en tiempo medio, pero mergesort necesita $\Theta(n)$ de memoria extra y quicksort ordena sobre el propio vector.

1.8 Eficiencia teórica frente a empírica

El análisis asintótico descarta constantes, y las constantes existen. Tres efectos que aparecen al medir de verdad y que el modelo no captura:

- **La localidad de referencia.** Recorrer un vector lee posiciones contiguas, que la caché trae por bloques. Recorrer una lista enlazada salta por la memoria y falla en caché casi siempre. Los dos recorridos son $\Theta(n)$ y uno puede ser un orden de magnitud más lento.
- **El punto de cruce.** Un algoritmo $\Theta(n \log n)$ con constante grande pierde frente a uno $\Theta(n^2)$ con constante pequeña mientras n sea chico. Por eso las implementaciones industriales de ordenación pasan a inserción directa por debajo de unas decenas de elementos.
- **El coste de reservar memoria.** Pedir memoria al sistema no es una operación elemental. Una estructura que reserva un nodo por elemento paga ese coste n veces.

El procedimiento para medir bien, que es el de la primera práctica:

1. Generar entradas de tamaños crecientes, con el mismo generador y semilla fija.
2. Repetir cada medida varias veces y quedarse con la mediana, no con la media: un pico del sistema operativo contamina la media y no la mediana.
3. Cronometrar solo la parte que se estudia, no la generación de los datos.
4. Representar tiempo frente a n, y comprobar si al dividir por la función teórica el cociente tiende a una constante.

El paso 4 es la comprobación de verdad. Si el algoritmo es $\Theta(n \log n)$, el cociente $t(n)/(n \log n)$ debe estabilizarse; si sigue creciendo, el análisis o la implementación no dicen lo mismo que el código.

Nota 1.2 . El compilador puede eliminar por completo un bucle cuyo resultado no se usa. Es la forma más común de medir cero segundos y creer que se ha escrito un algoritmo rápido. Se evita usando el resultado para algo, aunque sea acumularlo e imprimirlo al final.

1.9 Ejercicios

Ejercicio 1.9.1. Determinar el orden de eficiencia del siguiente fragmento en función de n:

```
for (i = 0; i < n; ++i)
for (j = i; j < n; ++j)
++cuenta;
```

Solución 1.9.1. El bucle interno se ejecuta $n - i$ veces, así que el total es $\sum_{i=0}^{n-1}(n-i) = n(n+1)/2$. Es $\Theta(n^2)$: que el segundo bucle no empiece en cero divide el trabajo entre dos, y una constante no cambia el orden.

Ejercicio 1.9.2. Una estructura resuelve la búsqueda en $\Theta(\log n)$ y la inserción en $\Theta(n)$. Otra resuelve las dos en $\Theta(\sqrt{n})$. ¿Cuál conviene para una carga con un 90 % de búsquedas y un 10 % de inserciones?

Solución 1.9.2. El coste medio por operación es $0{,}9 \log n + 0{,}1n$ frente a $\sqrt{n}$. El término $0{,}1n$ domina sobre $\sqrt{n}$ en cuanto n crece, así que la segunda gana pese a ser peor en búsqueda. La moraleja: se optimiza la mezcla real de operaciones, no la operación más frecuente.

Ejercicio 1.9.3. Un vector dinámico crece multiplicando su capacidad por 1,5 en vez de por 2. ¿Sigue siendo $O(1)$ amortizado el coste de insertar al final?

Solución 1.9.3. Sí. Los tamaños copiados forman una progresión geométrica de razón 1,5, cuya suma hasta n está acotada por $3n$. Cualquier factor de crecimiento estrictamente mayor que uno da coste amortizado constante; lo que lo rompe es crecer en una cantidad fija.

El desarrollo formal de estas nociones está en Rodríguez-Sánchez et al., 2020 y Garrido y Fdez-Valdivia, 2006, y el tratamiento del análisis amortizado en Carrano, 2017.

Capítulo 2

Abstracción de datos

Tema 2 del programa. Qué es un tipo de dato abstracto, cómo se especifica sin comprometerse con una implementación y qué ofrece C++ para construirlo.

2.1 El problema que resuelve

Un programa que manipula directamente la representación de sus datos queda atado a ella. Si una agenda guarda los contactos en un vector ordenado y cincuenta puntos del código recorren ese vector, cambiar a un árbol obliga a tocar los cincuenta.

La **abstracción de datos** separa dos cosas que se suelen confundir:

	Qué describe
Especificación	qué operaciones hay y qué hacen
Implementación	cómo se representan los datos y cómo se programan las operaciones

Quien usa el tipo trabaja contra la especificación. Quien lo implementa puede cambiar la representación entera mientras la especificación se mantenga, y ningún usuario se entera.

Definición 2.1 (Tipo de dato abstracto). Un TDA es un conjunto de valores junto con las operaciones definidas sobre ellos, descritas por su comportamiento observable y no por su representación interna.

Que la definición no mencione punteros ni vectores es el punto entero. Una pila es una pila porque el último que entra es el primero que sale, no porque por dentro haya un array.

2.2 Especificar un TDA

Una especificación completa dice, para cada operación:

Elemento	Qué recoge
Signatura	tipos de los argumentos y del resultado
Precondición	qué debe cumplirse antes de llamarla
Postcondición	qué garantiza al terminar
Coste	orden de eficiencia comprometido

La última fila se olvida a menudo y es parte del contrato. Un TDA conjunto que promete pertenencia

en $O(1)$ y otro que la promete en $O(\log n)$ tienen la misma signatura y **no son intercambiables**: el segundo puede sustituir al primero sin que nada deje de compilar, y arruinar el rendimiento del programa entero.

Ejemplo 2.1 . Especificación de la pila, sin decir cómo se guarda nada:

Operación	Precondición	Postcondición
`vacia()`	—	cierto si no hay elementos
`tope()`	la pila no está vacía	devuelve el último insertado
`poner(x)`	—	`x` pasa a ser el tope
`quitar()`	la pila no está vacía	elimina el tope

2.2.1 El invariante de la representación

Cuando se pasa a implementar, aparece una condición que la especificación no menciona porque no habla de representación: **el invariante**. Es lo que toda instancia válida cumple entre operación y operación.

En una pila sobre vector con un entero `n` que cuenta los elementos, el invariante es `0 <= n <= capacidad` y que las primeras `n` posiciones contienen los datos. Si una operación lo rompe a mitad, tiene que restaurarlo antes de terminar.

El invariante es la herramienta que hace demostrable la corrección de una estructura, y en la práctica es lo primero que se escribe al depurar: una función que lo comprueba y se llama al entrar y al salir de cada operación encuentra en minutos errores que de otro modo se manifiestan mucho después y lejos.

2.3 Encapsulamiento en C++

El mecanismo del lenguaje es la clase con sus niveles de acceso:

Nivel	Quién puede acceder
`public`	cualquiera: es la especificación
`private`	solo la propia clase: es la representación
`protected`	la clase y sus derivadas

```
class Pila {
public:
  bool vacia() const;
  const int& tope() const;
  void poner(const int& x);
  void quitar();

private:
  int* datos;        // representación: nadie de fuera la ve
  int n;             // elementos ocupados
  int capacidad;     // reservados
};
```

Cambiar `datos` por una lista enlazada no obliga a recompilar la lógica de quien usa la pila, solo a recompilar la clase. Ese es el beneficio concreto.

Nota 2.1 . Un `struct` de C++ es una clase cuyo acceso por defecto es `public`. La diferencia es esa y ninguna más. Usar `struct` para agregados sin invariante y `class` para tipos con invariante es una convención útil precisamente porque el compilador no la impone.

2.4 Constructores, destructor y copia

Una clase que gestiona memoria dinámica necesita algo más que las operaciones del TDA. Son las que el lenguaje genera solas si no se escriben, y las que genera solas suelen estar mal en cuanto hay un puntero de por medio.

Función	Cuándo se llama
Constructor por defecto	al declarar un objeto sin argumentos
Constructor con parámetros	al declararlo dándole valores iniciales
Constructor de copia	al inicializar a partir de otro objeto y al pasar por valor
Operador de asignación	al asignar un objeto ya construido
Destructor	al salir de ámbito o al hacer `delete`

2.4.1 El problema de la copia superficial

El constructor de copia que genera el compilador copia campo a campo. Con un puntero dentro, copia **la dirección**, no lo apuntado:

```
Pila a;              // reserva su vector
Pila b = a;          // b.datos apunta al mismo bloque que a.datos
```

Las consecuencias llegan las dos:

- Modificar `b` modifica `a`, porque comparten el bloque.
- Al destruirse las dos, el mismo bloque se libera dos veces, y eso es un error de memoria que puede no manifestarse hasta mucho después.

La solución es la **copia profunda**: reservar un bloque nuevo y copiar el contenido.

```
Pila::Pila(const Pila& otra)
    : n(otra.n), capacidad(otra.capacidad) {
  datos = new int[capacidad];
  for (int i = 0; i < n; ++i) datos[i] = otra.datos[i];
}
```

Y el operador de asignación, que además tiene dos casos que el constructor de copia no tiene: liberar lo que ya había, y protegerse de `a = a`.

```
Pila& Pila::operator=(const Pila& otra) {
  if (this != &otra) {           // sin esta guarda, a = a se destruye a sí mismo
    delete[] datos;
    n = otra.n;
    capacidad = otra.capacidad;
    datos = new int[capacidad];
```

```
    for (int i = 0; i < n; ++i) datos[i] = otra.datos[i];
  }
  return *this;
}
```

Nota 2.2 . La regla práctica: **si una clase necesita destructor, necesita también constructor de copia y operador de asignación**. Los tres aparecen y desaparecen juntos, porque los tres existen por la misma razón, que es poseer un recurso.

2.5 Plantillas: un TDA para cualquier tipo

Una pila de enteros y una de cadenas tienen el mismo código con un tipo distinto. Duplicarlo es la peor opción posible: dos copias que hay que corregir dos veces.

Las **plantillas** permiten escribirlo una vez con el tipo como parámetro:

```
template <typename T>
class Pila {
public:
  bool vacia() const;
  const T& tope() const;
  void poner(const T& x);
  void quitar();

private:
  T* datos;
  int n, capacidad;
};

Pila<int> pilaEnteros;
Pila<std::string> pilaCadenas;
```

El compilador genera una clase distinta por cada tipo usado. Dos consecuencias prácticas:

- **La plantilla se define entera en la cabecera.** El compilador necesita el cuerpo de los métodos en el punto donde se instancia, así que separar declaración e implementación en `.h` y `.cpp` como con una clase normal produce errores de enlazado.
- **Los errores aparecen al instanciar, no al escribir.** Una plantilla que llama a `operator<` compila sin quejarse hasta que alguien la instancia con un tipo que no lo define.

2.5.1 Requisitos sobre el tipo parámetro

Una plantilla impone condiciones implícitas sobre `T`. Las habituales:

Si la estructura hace	`T` necesita
copiar elementos	constructor de copia y asignación
declarar un hueco sin valor	constructor por defecto
ordenar	`operator<`
buscar por igualdad	`operator==`
imprimir	`operator<<`

Documentar esos requisitos es parte de la especificación del TDA. Es lo que distingue una plantilla utilizable de una que hay que descifrar leyendo mensajes de error del compilador.

2.6 Iteradores

Un TDA contenedor tiene que dejar recorrer sus elementos sin exponer cómo los guarda. La solución que adopta C++ es el **iterador**: un objeto que señala a un elemento y sabe avanzar al siguiente.

```
for (Lista<int>::iterator it = l.begin(); it != l.end(); ++it)
  std::cout << *it << " ";
```

El bucle es idéntico para un vector, una lista y un conjunto, y cada uno lo implementa de forma completamente distinta por dentro. Esa uniformidad es la que permite escribir algoritmos genéricos que funcionan sobre cualquier contenedor.

Las categorías de iterador según lo que soportan:

Categoría	Operaciones	Ejemplo
Entrada / salida	leer o escribir y avanzar, una pasada	flujos
Hacia delante	leer y avanzar, varias pasadas	lista simple
Bidireccional	además retroceder con `--`	lista doble, `set`, `map`
Acceso aleatorio	además `it + k` y comparaciones de orden	vector, `deque`

La categoría es lo que determina qué algoritmos se pueden aplicar. Ordenar exige acceso aleatorio, y por eso `std::sort` funciona sobre un `vector` y no sobre un `list`, que trae su propio `sort` con otro algoritmo.

Nota 2.3 . Modificar un contenedor mientras se recorre **invalida iteradores**. En un vector, insertar puede redimensionar y dejar todos los iteradores apuntando a memoria liberada; en una lista, borrar un nodo invalida solo el iterador a ese nodo. Las funciones de borrado devuelven un iterador válido al elemento siguiente justo para poder seguir el recorrido.

2.7 Relaciones entre TDAs

Al construir tipos complejos aparecen tres formas de apoyarse en otros, y distinguirlas evita diseños confusos:

Relación	Significado	Ejemplo
Composición	está formado por	una pila **contiene** un vector
Adaptación	se implementa usando otro y restringe su interfaz	una cola sobre una lista
Herencia	es un caso particular de	una cola con prioridad es una cola

En estructuras de datos la **composición domina**. Una pila que hereda de un vector expone el acceso por posición, y con él la posibilidad de insertar por el medio, que es justo lo que la pila

prohíbe. Contener el vector y publicar solo cuatro operaciones da un tipo correcto; heredarlo da uno que miente sobre lo que es.

2.8 Ejercicios

Ejercicio 2.8.1. Una clase `Cadena` guarda un `char*` reservado con `new[]` y define destructor, pero no constructor de copia. ¿Qué ocurre al pasar un objeto `Cadena` por valor a una función?

Solución 2.8.1. Se genera una copia superficial: el parámetro apunta al mismo bloque. Al terminar la función se destruye el parámetro y libera ese bloque, y el objeto original queda con un puntero a memoria liberada. El fallo aparece más tarde, al usarlo o al destruirlo por segunda vez, lejos de donde está la causa. Se corrige con copia profunda, o pasando por referencia constante.

Ejercicio 2.8.2. Especificar el TDA conjunto con las operaciones de inserción, borrado, pertenencia y cardinal, indicando el coste comprometido de cada una.

Solución 2.8.2. Las cuatro signaturas son `void insertar(T)`, `void borrar(T)`, `bool pertenece(T) const` y `int tam() const`. Insertar y borrar no tienen precondición y garantizan, respectivamente, que el elemento está y que no está; `pertenece` no modifica nada; `tam` devuelve el número de elementos distintos. Sobre el coste hay dos contratos razonables y distintos: $O(\log n)$ para las tres primeras si se exige recorrido ordenado, y $O(1)$ medio si no se exige. Elegir uno es parte de la especificación, no de la implementación.

Ejercicio 2.8.3. ¿Por qué una plantilla no puede separarse en `.h` y `.cpp` como una clase normal?

Solución 2.8.3. Porque el compilador genera el código de la plantilla en el punto donde se instancia, y para hacerlo necesita ver el cuerpo de los métodos. Compilando el `.cpp` por separado no hay ninguna instanciación a la vista, así que no se genera nada y el enlazador no encuentra los símbolos. Se resuelve incluyendo la implementación desde la propia cabecera.

El tratamiento de la abstracción y del encapsulamiento en C++ está desarrollado en Garrido y Fdez-Valdivia, 2006 y Carrano, 2017, y el de las plantillas en Garrido, 2017.

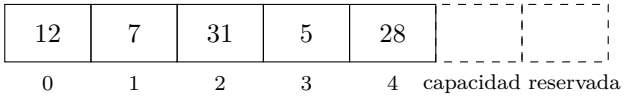
Capítulo 3

Tipos de datos contenedores básicos

Tema 3 del programa. Pilas, colas, colas con prioridad, conjuntos, diccionarios, vectores y listas, con sus implementaciones.

3.1 Vectores

El contenedor más simple: elementos consecutivos en memoria, accesibles por índice.



Operación	Coste
Acceso por índice	$\Theta(1)$
Insertar o borrar al final	$\Theta(1)$ amortizado
Insertar o borrar en posición i	$\Theta(n-i)$
Buscar sin orden	$\Theta(n)$
Buscar con orden, por bisección	$\Theta(\log n)$

La tercera fila es la que decide: insertar por el medio obliga a desplazar todo lo que hay detrás. Un vector es la estructura correcta cuando se accede mucho por posición y se modifica poco por el medio.

3.1.1 Redimensionado

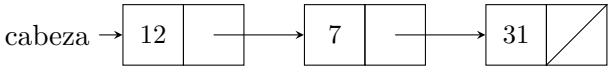
Cuando se agota la capacidad se reserva un bloque mayor, se copia y se libera el viejo. Doblar la capacidad da coste amortizado constante, como se vio en el tema 1.

```
void Vector::insertarFinal(const T& x) {
  if (n == capacidad) {
    capacidad = (capacidad == 0) ? 1 : capacidad * 2;
    T* nuevo = new T[capacidad];
    for (int i = 0; i < n; ++i) nuevo[i] = datos[i];
    delete[] datos;
    datos = nuevo;
  }
  datos[n++] = x;
}
```

Nota 3.1 . El redimensionado **invalida todos los punteros e iteradores** al contenido, porque el bloque cambia de dirección. Guardar un puntero a un elemento de un vector y luego insertar es un error que funciona hasta que el vector crece.

3.2 Listas enlazadas

Cada elemento vive en su propio nodo y guarda la dirección del siguiente. La memoria no es contigua, así que no hay acceso por índice, pero enlazar y desenlazar no mueve nada.



Operación	Lista simple	Vector
Acceso a la posición i	$\Theta(i)$	$\Theta(1)$
Insertar dado un iterador	$\Theta(1)$	$\Theta(n)$
Borrar dado un iterador	$\Theta(1)$	$\Theta(n)$
Insertar al principio	$\Theta(1)$	$\Theta(n)$
Memoria por elemento	dato + puntero	dato

La segunda fila lleva un matiz que se pierde a menudo: la inserción es $\Theta(1)$ **si ya se tiene el iterador**. Llegar hasta la posición cuesta $\Theta(i)$, así que insertar «en la posición 500» de una lista no es más barato que en un vector.

3.2.1 Variantes

Variante	Qué añade	Para qué
Doblemente enlazada	puntero al anterior	recorrer hacia atrás, borrar sin conocer el previo
Circular	el último apunta al primero	recorridos cíclicos, planificación por turnos
Con nodo centinela	un nodo ficticio al principio	elimina el caso especial de la lista vacía

El **centinela** merece una nota porque es una técnica general. Sin él, insertar al principio y en el medio son dos códigos distintos, porque uno actualiza la cabeza y otro un `siguiente`. Con un nodo ficticio que siempre existe, todos los casos son iguales, y desaparecen a la vez la mitad de los `if` y la mitad de los errores.

```
void Lista::borrar(Nodo* previo) {
  Nodo* victima = previo->siguiente;   // con centinela, previo nunca es nulo
  previo->siguiente = victima->siguiente;
  delete victima;
}
```

3.3 Pilas

Estructura LIFO: el último que entra es el primero que sale. Cuatro operaciones, todas $\Theta(1)$.

Operación	Qué hace
`poner(x)`	inserta en el tope
`quitar()`	elimina el tope
`tope()`	consulta el tope
`vacia()`	indica si no hay elementos

Se implementa sobre vector, con el tope al final para que insertar y borrar sean baratos, o sobre lista, con el tope en la cabeza por la misma razón.

Dónde aparecen las pilas, que es lo que justifica el tipo:

- **La pila de llamadas** de cualquier programa. Cada llamada apila su marco con parámetros, variables locales y dirección de retorno; al volver se desapila. Es lo que hace posible la recursividad.
- **Comprobar paréntesis equilibrados**: se apila cada apertura y se comprueba al cerrar.
- **Evaluar expresiones en notación postfija** y convertir de infija a postfija.
- **Deshacer** en cualquier editor.
- **Recorrer un grafo en profundidad**, y en general convertir un algoritmo recursivo en iterativo.

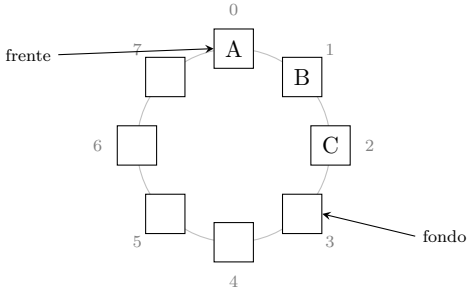
Ejemplo 3.1 . Evaluación de `3 4 + 2 *` en notación postfija: se apilan los operandos y, al leer un operador, se desapilan dos, se opera y se apila el resultado.

Lee	Acción	Pila
`3`	apila	3
`4`	apila	3 4
`+`	desapila 4 y 3, apila 7	7
`2`	apila	7 2
`*`	desapila 2 y 7, apila 14	14

3.4 Colas

Estructura FIFO: el primero que entra es el primero que sale. También cuatro operaciones en $\Theta(1)$, pero implementarla sobre un vector tiene una trampa.

Si el frente está en la posición 0, sacar obliga a desplazar todo: $\Theta(n)$. La solución es la **cola circular**: dos índices, frente y fondo, que avanzan módulo la capacidad.




```
void Cola::poner(const T& x) {
```

```
  datos[fondo] = x;
  fondo = (fondo + 1) % capacidad;
  ++n;
}
```

Con índices circulares hay una ambigüedad conocida: **frente igual a fondo describe tanto la cola vacía como la llena**. Se resuelve guardando el número de elementos, como arriba, o dejando siempre una posición sin usar. La primera opción es más clara y cuesta un entero.

Dónde aparecen: colas de impresión, planificación de procesos, gestión de peticiones en un servidor, y el recorrido en anchura de un grafo.

3.4.1 Bicolas

Una **bicola**, o cola doblemente terminada, permite insertar y extraer por los dos extremos, y absorbe pila y cola como casos particulares. La implementación habitual es un vector de bloques, que da acceso por índice en $\Theta(1)$ e inserción por los dos extremos en $\Theta(1)$ amortizado sin copiar todo al crecer.

3.5 Colas con prioridad

Sale el elemento de mayor prioridad, no el más antiguo. Con un vector ordenado la extracción es $\Theta(1)$ y la inserción $\Theta(n)$; con uno desordenado, al revés. La estructura que equilibra las dos es el **montículo**, que se trata en el tema siguiente porque es un árbol.

Implementación	Insertar	Extraer máximo
Vector desordenado	$\Theta(1)$	$\Theta(n)$
Vector ordenado	$\Theta(n)$	$\Theta(1)$
Lista ordenada	$\Theta(n)$	$\Theta(1)$
Montículo	$\Theta(\log n)$	$\Theta(\log n)$

Dónde aparecen: planificación por prioridades, simulación de eventos discretos, el algoritmo de Dijkstra y la construcción del código de Huffman.

3.6 Conjuntos

Colección sin duplicados y sin orden de inserción. Las operaciones son pertenencia, inserción, borrado y las de teoría de conjuntos: unión, intersección y diferencia.

Implementación	Pertenencia	Recorrido ordenado
Vector desordenado	$\Theta(n)$	no
Vector ordenado	$\Theta(\log n)$	sí
Árbol equilibrado	$\Theta(\log n)$	sí
Tabla hash	$\Theta(1)$ medio	no

La última columna es la que decide entre las dos últimas filas, y es la misma diferencia que hay en la STL entre `set` y `unordered_set`. Si nunca se recorre en orden, la tabla hash gana; si el orden importa, el árbol es la única opción de las dos.

3.6.1 Conjuntos sobre vector de bits

Cuando el universo de elementos posibles es pequeño y conocido, un conjunto se representa con un bit por elemento posible. La pertenencia es un acceso, y unión, intersección y diferencia son las operaciones lógicas OR, AND y AND-NOT aplicadas palabra a palabra: $n/64$ operaciones en vez de n.

Es la implementación más rápida que existe para ese caso, y la que se usa en análisis de programas y en motores de búsqueda.

3.7 Diccionarios

Asocian una **clave** con un **valor**. Es la abstracción de la búsqueda por contenido, y las estructuras de los dos temas siguientes existen sobre todo para implementarla bien.

Operación	Qué hace
`insertar(k, v)`	asocia la clave al valor
`buscar(k)`	devuelve el valor asociado, si existe
`borrar(k)`	elimina la asociación
`contiene(k)`	indica si la clave está

Las implementaciones son las mismas que las del conjunto —un conjunto es un diccionario cuyo valor no interesa— y con los mismos costes: árbol equilibrado para $\Theta(\log n)$ con recorrido ordenado, tabla hash para $\Theta(1)$ medio sin él.

Nota 3.2 . En la STL, `map[k]` **inserta la clave con un valor por defecto** si no existe. Es la causa más común de que un diccionario crezca al consultarlo, y en un método constante ni siquiera compila. Para consultar sin insertar están `find` y `count`.

3.8 Cómo se elige

El resumen del tema en una tabla. Se lee de arriba abajo hasta que una fila describa el problema:

Si lo que hace falta es	Estructura
acceso por posición, pocos cambios por el medio	vector
insertar y borrar por el medio con iterador	lista
último en entrar, primero en salir	pila
primero en entrar, primero en salir	cola
el más urgente primero	cola con prioridad
saber si algo está, sin orden	conjunto sobre hash
saber si algo está, y recorrer ordenado	conjunto sobre árbol
asociar clave con valor	diccionario, sobre hash o árbol

3.9 Ejercicios

Ejercicio 3.9.1. Implementar una cola usando dos pilas. ¿Cuál es el coste amortizado de extraer?

Solución 3.9.1. Se apila en la pila de entrada. Al extraer, si la de salida está vacía se vuelcan todos los elementos de la de entrada en ella, lo que invierte el orden, y se desapila de la de salida. Cada elemento se mueve como mucho dos veces en toda su vida, así que el coste amortizado por operación es $\Theta(1)$, aunque una extracción concreta pueda costar $\Theta(n)$.

Ejercicio 3.9.2. En una cola circular sobre un vector de capacidad C con índices `frente` y `fondo`, ¿cómo se distingue la cola vacía de la llena si no se guarda el número de elementos?

Solución 3.9.2. No se puede: las dos situaciones dan `frente == fondo`. Se resuelve dejando siempre una posición libre, con lo que la cola está llena cuando `(fondo + 1) % C == frente` y vacía cuando `frente == fondo`. El precio es una posición desperdiciada y una capacidad efectiva de $C - 1$.

Ejercicio 3.9.3. Un programa mantiene un millón de enteros y solo pregunta si un valor está o no. ¿Vector ordenado, árbol equilibrado o tabla hash?

Solución 3.9.3. Tabla hash, por la pertenencia en $\Theta(1)$ medio y porque el orden no se usa. El vector ordenado sería competitivo si el conjunto fuese estático —se ordena una vez y se busca por bisección con muy buena localidad de caché—, pero pierde en cuanto haya inserciones, que le cuestan $\Theta(n)$ cada una.

Las implementaciones detalladas de estos contenedores están en Rodríguez-Sánchez et al., 2020 y Garrido y Fdez-Valdivia, 2006, y su versión en la biblioteca estándar en Garrido, 2016.

Capítulo 4

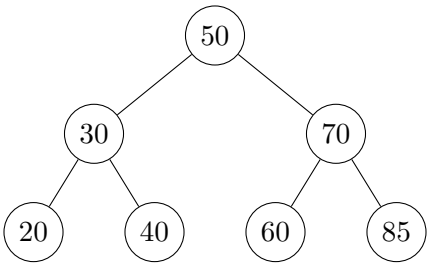
Tipos de datos contenedores complejos

Tema 4 del programa. Árboles, tablas hash y grafos, con sus implementaciones. Son las estructuras que hacen posible buscar en tiempo logarítmico o constante donde un contenedor lineal necesitaría recorrerlo todo.

4.1 Árboles

Estructura jerárquica: un nodo raíz y, colgando de él, subárboles. La terminología que se usa en todo el tema:

Término	Qué es
Raíz	el único nodo sin padre
Hoja	nodo sin hijos
Grado	número de hijos de un nodo
Profundidad de un nodo	aristas desde la raíz hasta él
Altura del árbol	profundidad máxima
Nivel k	conjunto de nodos a profundidad k



La altura es lo que determina el coste. Un árbol binario con n nodos tiene altura mínima $\lfloor \log_2 n \rfloor$ si está lleno, y altura $n-1$ si degenera en una lista. Toda la teoría de árboles equilibrados existe para evitar el segundo caso.

4.1.1 Recorridos

Recorrido	Orden	Para qué sirve
Preorden	raíz, izquierdo, derecho	copiar el árbol, serializarlo
Inorden	izquierdo, raíz, derecho	listar ordenado en un ABB

Recorrido	Orden	Para qué sirve
Postorden	izquierdo, derecho, raíz	liberar memoria, evaluar expresiones
Por niveles	de arriba abajo, de izquierda a derecha	búsqueda en anchura

```
void inorden(Nodo* n) {
  if (n != nullptr) {
    inorden(n->izq);
    visitar(n);
    inorden(n->der);
  }
}
```

Los tres primeros son recursivos y cuestan $\Theta(n)$. El de niveles no es recursivo: **necesita una cola**, y ahí se ve por qué el tema anterior viene antes.

Nota 4.1 . El postorden es el único que sirve para destruir un árbol. Liberar la raíz antes de recorrer los hijos deja los punteros a los subárboles en memoria ya liberada, y el árbol entero se pierde sin llegar a liberarse.

4.1.2 Árboles binarios de búsqueda

Un ABB impone un invariante: para todo nodo, las claves del subárbol izquierdo son menores y las del derecho mayores.

Definición 4.1 (Árbol binario de búsqueda). Árbol binario en el que, para todo nodo x, toda clave del subárbol izquierdo de x es menor que la de x, y toda clave del subárbol derecho es mayor.

De ese invariante salen tres consecuencias inmediatas:

- Buscar es descender comparando: en cada nodo se descarta medio árbol.
- El recorrido en inorden produce las claves ordenadas.
- El mínimo está bajando siempre a la izquierda, y el máximo a la derecha.

```
Nodo* buscar(Nodo* n, const T& k) {
  while (n != nullptr && n->clave != k)
    n = (k < n->clave) ? n->izq : n->der;
  return n;
}
```

El coste de todas las operaciones es $\Theta(h)$, con h la altura. Y ahí está el problema: **insertar claves ya ordenadas produce una lista**.

Ejemplo 4.1 . Insertando $1, 2, 3, 4, 5$ en ese orden, cada clave es mayor que todas las anteriores y acaba colgando a la derecha de la última. El resultado es un árbol de altura 4 con cinco nodos: la búsqueda pasa de $\Theta(\log n)$ a $\Theta(n)$ y la estructura no es mejor que una lista, solo más cara en memoria.

El borrado tiene tres casos y conviene tenerlos claros:

Caso	Qué se hace
Hoja	se elimina sin más
Un hijo	el hijo ocupa el lugar del padre
Dos hijos	se sustituye por el mayor del subárbol izquierdo, o el menor del derecho, y se borra ese

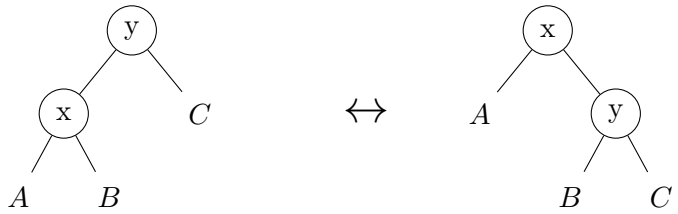
El tercero funciona porque ese sustituto es el único valor que conserva el invariante, y por construcción tiene como mucho un hijo, así que su borrado cae en uno de los dos casos anteriores.

4.1.3 Árboles equilibrados

Mantienen la altura en $\Theta(\log n)$ reorganizándose al insertar y al borrar.

Tipo	Criterio de equilibrio	Altura
AVL	las alturas de los subárboles difieren como mucho en 1	$\leq 1{,}44 \log_2 n$
Rojo-negro	ningún camino es más del doble de largo que otro	$\leq 2 \log_2 n$
B / B+	todas las hojas al mismo nivel, grado alto	$\log_m n$

La reorganización se hace con **rotaciones**, que cambian la forma sin romper el invariante de orden:



En los dos lados, el recorrido en inorden da A, x, B, y, C. Esa es la propiedad que hace correcta la rotación, y la que hay que comprobar al implementarla.

Un AVL está más equilibrado y busca algo más rápido; un rojo-negro rota menos y inserta y borra algo más rápido. Por eso las bibliotecas estándar —incluida la de C++— usan rojo-negro para `map` y `set`.

Los **árboles B** son distintos en propósito: cada nodo guarda muchas claves para que un nodo entero quepa en un bloque de disco. Con grado 100, un millón de claves caben en tres niveles, y eso son tres lecturas de disco en vez de veinte. Es la estructura de los índices de cualquier base de datos.

4.1.4 Montículos

Un montículo binario es un árbol **completo** —todos los niveles llenos salvo el último, que se rellena de izquierda a derecha— con el invariante de que todo padre es mayor o igual que sus hijos.

Que sea completo permite guardarlo en un vector sin punteros:

Nodo en la posición i	Está en
Padre	$(i-1)/2$

Nodo en la posición i	Está en
Hijo izquierdo	$2i+1$
Hijo derecho	$2i+2$

50	30	45	12	25	40
0	1	2	3	4	5

Las dos operaciones que lo mantienen:

- **Flotar**: tras insertar al final, el nuevo elemento sube mientras sea mayor que su padre. $\Theta(\log n)$.
- **Hundir**: tras extraer la raíz y poner el último en su lugar, baja mientras sea menor que alguno de sus hijos. $\Theta(\log n)$.

Con eso, la cola con prioridad del tema anterior queda resuelta en $\Theta(\log n)$ para las dos operaciones. Y de regalo sale **heapsort**: construir el montículo en $\Theta(n)$ y extraer el máximo n veces da una ordenación $\Theta(n \log n)$ sin memoria auxiliar.

4.2 Tablas hash

La idea es distinta de la de los árboles: en vez de comparar, **calcular** dónde está el elemento. Una función hash convierte la clave en un índice.

$$h : \text{claves} \longrightarrow \{0, 1, \ldots, m-1\}$$

Si no hubiera colisiones, la búsqueda sería un cálculo y un acceso: $\Theta(1)$. Pero el número de claves posibles supera al de posiciones, así que **las colisiones son inevitables** y toda la técnica consiste en gestionarlas.

4.2.1 La función hash

Una buena función hash cumple tres cosas:

Propiedad	Por qué
Reparte uniformemente	si no, se amontonan las colisiones
Es rápida	se ejecuta en cada operación
Usa toda la clave	si no, claves parecidas colisionan

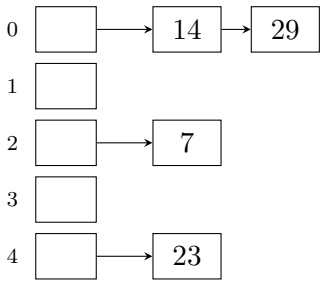
La tercera es la que se incumple con más frecuencia. Una función que solo mira los tres primeros caracteres de una cadena manda todos los apellidos que empiezan igual a la misma posición.

Dos métodos habituales:

- **División**: $h(k) = k \bmod m$, con m primo. Que sea primo importa: con $m = 2^p$ la función solo mira los p bits bajos de la clave.
- **Multiplicación**: $h(k) = \lfloor m\,(kA - \lfloor kA \rfloor) \rfloor$ con $0 < A < 1$. No exige nada de m.

4.2.2 Resolución de colisiones

Encadenamiento. Cada posición guarda una lista con todas las claves que caen ahí.



Direccionamiento abierto. Todo se guarda en la propia tabla; al colisionar se prueban otras posiciones según una secuencia:

Sondeo	Secuencia	Problema
Lineal	$h(k) + i$	agrupamiento primario: se forman bloques largos
Cuadrático	$h(k) + c_1 i + c_2 i^2$	agrupamiento secundario
Doble hash	$h_1(k) + i\, h_2(k)$	el mejor reparto, y el más caro

4.2.3 El factor de carga

$$\alpha = \frac{n}{m}$$

Es la cantidad que gobierna el rendimiento. Con encadenamiento, la longitud media de una lista es α, así que la búsqueda cuesta $\Theta(1 + \alpha)$. Con direccionamiento abierto el número medio de sondeos es aproximadamente $1/(1 - \alpha)$, que **se dispara al acercarse a 1**: con $\alpha = 0{,}9$ son diez sondeos, y con 0,99, cien.

Por eso las implementaciones reales **rehacen la tabla** cuando α pasa de un umbral, típicamente 0,75: se dobla m y se reinsertan todas las claves. Cuesta $\Theta(n)$ y ocurre pocas veces, así que el coste amortizado sigue siendo constante.

Nota 4.2 . El borrado con direccionamiento abierto no puede dejar el hueco vacío: cortaría la secuencia de sondeo y las claves que estaban detrás se volverían inalcanzables. Se marca la posición como borrada, y esas marcas hay que contarlas en el factor de carga.

4.2.4 Hash frente a árbol

	Tabla hash	Árbol equilibrado
Búsqueda, caso medio	$\Theta(1)$	$\Theta(\log n)$
Búsqueda, peor caso	$\Theta(n)$	$\Theta(\log n)$
Recorrido ordenado	no	sí
Mínimo, máximo, rango	no	sí
Requiere de la clave	función hash e igualdad	orden total
Memoria	huecos reservados	punteros por nodo

La fila del peor caso importa más de lo que parece: una tabla hash con una función mal elegida degenera en una lista, y hay ataques que consisten justamente en enviar claves que colisionan a

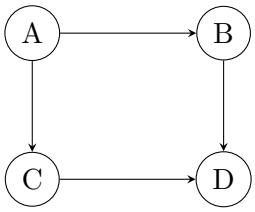
propósito.

4.3 Grafos

Un conjunto de vértices y de aristas que los conectan. Es la estructura más general del temario: un árbol es un grafo conexo sin ciclos, y una lista es un árbol degenerado.

Concepto	Qué es
Dirigido	las aristas tienen sentido
Ponderado	las aristas llevan un peso
Camino	secuencia de vértices unidos por aristas
Ciclo	camino que vuelve al origen
Conexo	hay camino entre todo par de vértices
Grado	número de aristas incidentes en un vértice

4.3.1 Representación



Representación	Espacio	¿Hay arista (u, v)?	Recorrer vecinos de u
Matriz de adyacencia	$\Theta(V^2)$	$\Theta(1)$	$\Theta(V)$
Listas de adyacencia	$\Theta(V + E)$	$\Theta(\text{grado})$	$\Theta(\text{grado})$

La elección depende de la densidad. Un grafo **disperso** —redes sociales, mapas de carreteras, dependencias entre módulos— tiene $E \ll V^2$ y desperdicia casi toda la matriz. Un grafo **denso** aprovecha la matriz y gana en la consulta directa. En la práctica la mayoría de los grafos reales son dispersos, así que la lista de adyacencia es la opción por defecto.

4.3.2 Recorridos

Recorrido	Estructura auxiliar	Qué encuentra
Profundidad (DFS)	pila, o recursión	componentes conexas, ciclos, orden topológico
Anchura (BFS)	cola	camino más corto en número de aristas

Los dos cuestan $\Theta(V+E)$ con listas de adyacencia y **necesitan marcar los vértices visitados**. Sin esa marca, un ciclo hace que el recorrido no termine nunca. Es el error más común al implementarlos.

```
void bfs(const Grafo& g, int origen) {
  std::vector<bool> visitado(g.numVertices(), false);
```

```
  std::queue<int> cola;
  visitado[origen] = true;
  cola.push(origen);
  while (!cola.empty()) {
    int u = cola.front(); cola.pop();
    for (int v : g.vecinos(u))
      if (!visitado[v]) { visitado[v] = true; cola.push(v); }
  }
}
```

Que BFS dé el camino más corto en número de aristas se ve en la propia cola: los vértices salen por niveles de distancia creciente al origen, así que el primero que alcanza un vértice lo hace por el camino más corto.

4.3.3 Problemas clásicos

Problema	Algoritmo	Coste
Camino mínimo con pesos no negativos	Dijkstra	$\Theta((V+E)\log V)$ con montículo
Camino mínimo con pesos negativos	Bellman-Ford	$\Theta(VE)$
Árbol de recubrimiento mínimo	Prim, Kruskal	$\Theta(E \log V)$
Orden topológico	DFS sobre un grafo acíclico dirigido	$\Theta(V+E)$
Componentes conexas	DFS o BFS repetidos	$\Theta(V+E)$

Los detalles de estos algoritmos son materia de Algorítmica. Lo que corresponde a esta asignatura es que **su coste depende de la estructura que los sostiene**: Dijkstra sobre una lista sin ordenar es $\Theta(V^2)$ y sobre un montículo es $\Theta((V+E)\log V)$. El algoritmo es el mismo; lo que cambia es el TDA.

4.4 Ejercicios

Ejercicio 4.4.1. Insertar $10, 20, 30, 40, 50$ en un ABB vacío, en ese orden. ¿Qué altura tiene? ¿Cuánto costaría buscar el 50?

Solución 4.4.1. Cada clave es mayor que todas las anteriores, así que cuelga siempre a la derecha: el árbol es una lista descendente de altura 4. Buscar el 50 exige recorrerla entera, $\Theta(n)$. Es el caso degenerado que motiva los árboles equilibrados; un AVL habría rotado y quedado con altura 2.

Ejercicio 4.4.2. Una tabla hash con encadenamiento tiene $m = 100$ y $n = 300$ claves bien repartidas. ¿Cuántas comparaciones cuesta una búsqueda infructuosa?

Solución 4.4.2. El factor de carga es $\alpha = 3$, así que la longitud media de cada lista es 3 y una búsqueda infructuosa la recorre entera: tres comparaciones de media. Sigue siendo constante respecto a n, pero la constante crece con α, y por eso se rehace la tabla en vez de dejarla llenarse.

Ejercicio 4.4.3. Un grafo tiene 10 000 vértices y 30 000 aristas. ¿Matriz o listas de adyacencia?

Solución 4.4.3. Listas. La matriz ocuparía 10^8 posiciones para almacenar 30 000 aristas, con una densidad del 0,03 %. Las listas ocupan $\Theta(V + E) = 40\,000$ entradas. La matriz solo compensaría si hiciesen falta muchísimas consultas de la forma «¿existe la arista (u, v)?» y el grafo fuese denso, que no es el caso.

El desarrollo de árboles equilibrados y tablas hash está en Garrido, 2018, y el de grafos y sus recorridos en Rodríguez-Sánchez et al., 2020 y Koffman y Wolfgang, 2006.

Capítulo 5

Seminarios

Los dos seminarios del programa: aplicación de los TDA sobre problemas reales y uso de la STL en problemas prácticos.

5.1 La biblioteca estándar

La STL es la implementación de todo el temario que trae el propio lenguaje. Está organizada en tres piezas que se combinan:

Pieza	Qué aporta
Contenedores	las estructuras de datos
Iteradores	la forma uniforme de recorrerlas
Algoritmos	operaciones que trabajan sobre rangos de iteradores

La separación es lo que la hace potente: hay C contenedores y A algoritmos, y funcionan las $C \times A$ combinaciones porque ninguno de los dos conoce al otro. Ambos hablan con iteradores.

5.1.1 Contenedores secuenciales

Contenedor	Estructura	Acceso $[i]$	Inserción por el medio
`vector`	vector dinámico	$\Theta(1)$	$\Theta(n)$
`deque`	bloques encadenados	$\Theta(1)$	$\Theta(n)$
`list`	lista doblemente enlazada	no hay	$\Theta(1)$ con iterador
`forward_list`	lista simple	no hay	$\Theta(1)$ con iterador
`array`	vector de tamaño fijo	$\Theta(1)$	no hay

`vector` es la opción por defecto, y no por comodidad: la localidad de caché compensa el desplazamiento de elementos en tamaños moderados. Una `list` gana cuando los elementos son grandes, se insertan y borran mucho por el medio y ya se dispone del iterador.

5.1.2 Contenedores asociativos

Contenedor	Implementación	Búsqueda	Ordenado
`set`, `map`	árbol rojo-negro	$\Theta(\log n)$	sí

Contenedor	Implementación	Búsqueda	Ordenado
`multiset`, `multimap`	ídem, con repetidos	$\Theta(\log n)$	sí
`unordered_set`, `unordered_map`	tabla hash	$\Theta(1)$ medio	no

La regla para elegir es la del tema anterior: si hace falta recorrer en orden, o preguntar por rangos, o el mínimo, van los ordenados; si solo hace falta saber si algo está, los de hash.

5.1.3 Adaptadores

`stack`, `queue` y `priority_queue` no son contenedores nuevos: **restringen la interfaz de otro**. Es la relación de adaptación del tema 2, aplicada.

```
std::stack<int> p;                          // por debajo, un deque
std::stack<int, std::vector<int>> pv;       // por debajo, un vector
std::priority_queue<int> cp;                // un montículo sobre vector
```

El adaptador solo expone las operaciones del TDA, así que sobre un `stack` no se puede recorrer ni acceder por índice. Esa restricción es el objetivo, no una limitación.

5.2 Algoritmos

Trabajan sobre un rango `[primero, ultimo)` de iteradores. El extremo derecho queda fuera, y esa convención tiene dos consecuencias útiles: el número de elementos es la resta de los iteradores, y el rango vacío se escribe con los dos iguales.

Grupo	Ejemplos
No modificadores	`find`, `count`, `all_of`, `equal`, `search`
Modificadores	`copy`, `transform`, `replace`, `remove`, `reverse`
Ordenación	`sort`, `stable_sort`, `partial_sort`, `nth_element`
Sobre rango ordenado	`binary_search`, `lower_bound`, `upper_bound`, `merge`
Numéricos	`accumulate`, `inner_product`, `partial_sum`
Montículo	`make_heap`, `push_heap`, `pop_heap`, `sort_heap`

```
std::sort(v.begin(), v.end());
auto it = std::lower_bound(v.begin(), v.end(), 42);
int suma = std::accumulate(v.begin(), v.end(), 0);
```

Nota 5.1 . `remove` **no borra nada**. Reordena el rango dejando delante los elementos que se conservan y devuelve un iterador al final de esa parte; el contenedor sigue teniendo el mismo tamaño. Para borrar de verdad hace falta `v.erase(std::remove(v.begin(), v.end(), x), v.end())`. Es la consecuencia directa de que un algoritmo solo vea iteradores y no pueda cambiar el tamaño del contenedor.

5.2.1 Criterios de comparación

Los algoritmos de ordenación aceptan un comparador, que puede ser una función, un objeto función o una lambda:

```
std::sort(v.begin(), v.end(),
          [](const Alumno& a, const Alumno& b) { return a.nota > b.nota; });
```

El comparador tiene que definir un **orden estricto débil**: si `cmp(a,b)` y `cmp(b,a)` son los dos ciertos para algún par, `sort` puede salirse del rango y corromper memoria. El error clásico es escribir `<=` en vez de `<`, que hace `cmp(a,a)` cierto y rompe el invariante.

5.3 Aplicación de los TDA a problemas reales

Los cuatro problemas de este seminario, con la estructura que los resuelve y por qué.

5.3.1 Contar palabras de un texto

Se necesita asociar cada palabra con su número de apariciones, y al final listar por frecuencia.

```
std::unordered_map<std::string, int> cuenta;
std::string palabra;
while (entrada >> palabra) ++cuenta[palabra];
```

La estructura es un diccionario sobre hash: solo se busca por clave exacta. Para el listado final por frecuencia se vuelca a un vector de pares y se ordena, porque ordenar una vez cuesta $\Theta(n \log n)$ y mantener el orden en cada inserción costaría $\Theta(\log n)$ por palabra sin aportar nada durante el conteo.

Aquí `cuenta[palabra]` sí se quiere: inserta con valor cero si la palabra es nueva, que es exactamente lo que hace falta.

5.3.2 Corrector de paréntesis anidados

Pila. Se apila cada símbolo de apertura y al leer uno de cierre se comprueba que el tope es su pareja.

Situación	Resultado
Cierre con pila vacía	error: sobra un cierre
Cierre que no casa con el tope	error: mal anidados
Fin de texto con pila no vacía	error: falta cerrar

Los tres casos hay que comprobarlos. Un corrector que solo cuenta aperturas y cierres da por bueno `([)]`.

5.3.3 Planificador de tareas con dependencias

Un grafo dirigido donde cada arista es una dependencia, y un **orden topológico** que da una secuencia válida de ejecución.

El algoritmo por grados de entrada usa dos TDA a la vez: un vector con el número de dependencias pendientes de cada tarea, y una cola con las que ya tienen cero. Si al terminar quedan tareas sin colocar, **hay un ciclo**, es decir, dependencias circulares. Ese diagnóstico sale gratis del algoritmo y es lo que hace útil el planificador.

5.3.4 Sistema de recomendación por vecindad

Un grafo de usuarios y productos, y una búsqueda en anchura acotada a dos niveles para encontrar lo que compraron quienes compraron lo mismo.

Con listas de adyacencia el recorrido cuesta $\Theta(V + E)$; con una matriz sobre un grafo disperso, $\Theta(V^2)$, y eso es la diferencia entre responder en milisegundos y no responder. Es el mismo algoritmo con distinta estructura debajo, que es la moraleja del seminario entero.

5.4 Errores frecuentes con la STL

Error	Qué ocurre
Guardar un iterador y luego insertar en el `vector`	el redimensionado lo invalida
Borrar dentro de un `for` sin usar el iterador devuelto	el iterador queda colgado
Usar `map[k]` para consultar	inserta la clave con valor por defecto
Pasar contenedores grandes por valor	copia profunda silenciosa en cada llamada
Comparador con `<=`	orden no estricto, comportamiento indefinido en `sort`
`std::sort` sobre una `list`	no compila: exige acceso aleatorio, hay que usar `list::sort`

El borrado correcto dentro de un recorrido:

```
for (auto it = l.begin(); it != l.end(); )
  if (*it % 2 == 0) it = l.erase(it);   // erase devuelve el siguiente válido
  else ++it;
```

5.5 Elegir contenedor: resumen

Situación	Contenedor
Por defecto, secuencia con acceso por posición	`vector`
Inserciones y borrados frecuentes por los dos extremos	`deque`
Muchas inserciones y borrados por el medio, con iterador	`list`
Saber si un elemento está, sin orden	`unordered_set`
Ídem, y recorrer u obtener rangos ordenados	`set`
Asociar clave con valor	`unordered_map` o `map`, mismo criterio
Último en entrar, primero en salir	`stack`
Primero en entrar, primero en salir	`queue`
El más urgente primero	`priority_queue`

5.6 Ejercicios

Ejercicio 5.6.1. ¿Qué imprime este código y por qué?

```
std::vector<int> v = {1, 2, 3, 4, 5};
std::remove(v.begin(), v.end(), 3);
std::cout « v.size();
```

Solución 5.6.1. Imprime 5. `remove` no cambia el tamaño del contenedor: solo reordena y devuelve un iterador al nuevo final lógico, que aquí se descarta. El vector queda con cinco posiciones, las cuatro primeras con $1, 2, 4, 5$ y la última con un valor sin especificar. Para borrar de verdad hay que llamar a `erase` con el iterador devuelto.

Ejercicio 5.6.2. Un programa mantiene una lista de eventos y necesita repetidamente el más próximo en el tiempo, insertando eventos nuevos sobre la marcha. ¿Qué contenedor?

Solución 5.6.2. Una `priority_queue`: extraer el mínimo y añadir cuestan $\Theta(\log n)$ las dos. Un vector ordenado daría extracción en $\Theta(1)$ pero inserción en $\Theta(n)$, y aquí las dos operaciones son frecuentes. Si además hiciese falta cancelar eventos concretos, la cola con prioridad no basta —no permite borrar por clave— y habría que ir a un `set` ordenado por instante.

Ejercicio 5.6.3. ¿Por qué `std::sort` no funciona sobre una `std::list`?

Solución 5.6.3. Porque exige iteradores de acceso aleatorio, que permiten saltar a una posición cualquiera en tiempo constante, y los de `list` son bidireccionales. El algoritmo interno de `sort` necesita esos saltos para particionar. La lista trae su propio `sort`, que usa mezcla reenlazando nodos en vez de mover elementos.

El uso de la biblioteca estándar está desarrollado en Garrido, 2017, Garrido, 2016 y Musser et al., 2009, y su referencia completa en Robson, 2013.

Capítulo 6

Temario práctico

Las cuatro prácticas del programa. Cada una construye un TDA, lo prueba y mide su eficiencia, que es lo que separa saber la teoría de saber usarla.

6.1 Práctica 1. Eficiencia de algoritmos

Comparar la eficiencia teórica con la empírica sobre varios ejemplos.

Qué se mide:

Algoritmo	Orden teórico	Qué se espera ver
Búsqueda lineal	$\Theta(n)$	recta
Búsqueda binaria	$\Theta(\log n)$	curva casi plana
Ordenación por inserción	$\Theta(n^2)$	parábola
Ordenación por mezcla	$\Theta(n \log n)$	casi recta, algo curvada

Cómo se mide. El procedimiento del tema 1, con dos precauciones que en la práctica deciden si los datos valen:

- **Cronometrar solo lo que se estudia.** Generar un vector de un millón de elementos aleatorios cuesta más que ordenarlo con un algoritmo bueno. Si la generación entra en el cronómetro, la gráfica mide el generador.
- **Usar el resultado.** Un bucle cuyo resultado se descarta puede desaparecer entero en la optimización. Acumular algo e imprimirlo al final lo impide.

Cómo se comprueba el ajuste. Se divide el tiempo medido por la función teórica y se mira si el cociente se estabiliza:

$$\frac{t(n)}{f(n)} \longrightarrow c$$

Si sigue creciendo, la función teórica se queda corta; si tiende a cero, sobra. Es mejor prueba que mirar la forma de la curva, porque una parábola y una $n \log n$ se parecen mucho en un rango estrecho.

Lo que se observa y hay que explicar:

Observación	Causa
Con n pequeño, la inserción gana a la mezcla	constante menor y sin memoria auxiliar
Recorrer una lista es más lento que un vector, siendo los dos $\Theta(n)$	fallos de caché por falta de localidad

Observación	Causa
La curva da saltos en potencias de dos	redimensionados del vector
Ordenar datos ya ordenados cambia mucho el tiempo	la inserción baja a $\Theta(n)$, el quicksort ingenuo sube a $\Theta(n^2)$

La última fila es la más instructiva: el **mismo tamaño de entrada** da tiempos radicalmente distintos según cómo estén los datos. Es la diferencia entre caso mejor, peor y medio, medida.

6.2 Práctica 2. Construcción de TDA básicos

Implementar desde cero, con plantillas y sin usar la STL, los contenedores del tema 3.

Qué se construye:

- Vector dinámico con redimensionado por duplicación.
- Lista simple y lista doblemente enlazada, con nodo centinela.
- Pila y cola, cada una sobre las dos representaciones anteriores.
- Cola circular sobre vector.

Lo que hay que respetar en cada clase:

Elemento	Por qué
Constructor, destructor, copia y asignación	la clase posee memoria: los cuatro van juntos
Comprobar la precondición	`tope()` sobre una pila vacía es error del que llama
Documentar el coste de cada operación	forma parte de la especificación
Función que comprueba el invariante	encuentra los errores donde se producen

Los errores que aparecen siempre, y merece la pena anticiparlos:

- **Fuga de memoria** por olvidar `delete[]` en el destructor, o por reasignar el puntero antes de liberar el bloque anterior.
- **Doble liberación** por copia superficial, cuando falta el constructor de copia.
- **Acceso fuera de rango** al insertar en una posición igual al tamaño, que es válida para insertar y no para acceder.
- **Perder la lista entera** al desenlazar sin guardar antes el puntero al siguiente.

Herramientas que encuentran los tres primeros sin depurar a mano: compilar con `-fsanitize=address` o pasar `valgrind`. Los dos delatan la línea exacta.

6.3 Práctica 3. Uso e implementación de TDA lineales

Aplicar los contenedores de la práctica anterior a problemas concretos, y comparar implementaciones.

Problemas propuestos:

Problema	TDA
Comprobar expresiones bien parentizadas	pila
Convertir de infija a postfija y evaluar	pila
Simular una cola de atención con estadísticas de espera	cola
Planificación por turnos entre procesos	lista circular
Mezclar dos listas ordenadas en una	lista

La comparación que se pide. Resolver el mismo problema con las dos representaciones y medir. La conclusión que suele salir, y que conviene entender en vez de memorizar: **la implementación sobre vector gana casi siempre en tiempo** aunque el análisis asintótico diga lo mismo, porque recorre memoria contigua. La lista gana cuando los elementos son grandes de copiar o cuando se inserta y borra mucho por el medio con el iterador ya en la mano.

6.4 Práctica 4. Uso e implementación de TDA no lineales

Los contenedores del tema 4.

Qué se construye:

- Árbol binario de búsqueda con inserción, borrado en los tres casos y los cuatro recorridos.
- Montículo binario sobre vector, y la cola con prioridad encima.
- Tabla hash con encadenamiento, con rehash automático al pasar el factor de carga.
- Grafo con listas de adyacencia, con recorridos en profundidad y en anchura.

Mediciones que se piden:

Medida	Qué demuestra
Altura del ABB con claves aleatorias frente a ordenadas	la degeneración a lista
Tiempo de búsqueda al crecer n en un ABB equilibrado	el $\Theta(\log n)$
Colisiones medias según el factor de carga	por qué se rehace la tabla
Comparación de la tabla hash con el ABB en búsqueda pura	$\Theta(1)$ frente a $\Theta(\log n)$

La primera medida es la que más enseña: con claves aleatorias la altura media de un ABB de n nodos ronda $1{,}39 \log_2 n$, y con claves ordenadas es exactamente $n - 1$. Los dos árboles tienen los mismos elementos y el mismo código; lo único que cambia es el orden de llegada.

Sobre los grafos, la comprobación que cierra la práctica: implementar el mismo recorrido con matriz y con listas de adyacencia sobre un grafo disperso y medir. La diferencia entre $\Theta(V^2)$ y $\Theta(V + E)$ deja de ser una fórmula en cuanto V pasa de unos miles.

6.5 Sobre la memoria de prácticas

Lo que se entrega:

1. Especificación de cada TDA implementado: signatura, precondiciones, postcondiciones y **coste comprometido**.

2. Decisiones de representación, con el invariante que mantiene cada clase.
3. Juego de pruebas, con los casos límite: contenedor vacío, un solo elemento, capacidad justo agotada, borrado del único elemento, clave repetida.
4. Mediciones, con las gráficas de tiempo frente a n.
5. Comparación entre lo teórico y lo medido, **explicando las diferencias**.

El punto 5 es el que se evalúa de verdad, y las causas posibles son casi siempre las mismas: constantes ocultas en el análisis, localidad de caché, coste de reservar memoria, punto de cruce entre dos órdenes distintos, o un algoritmo cuyo caso medio no es el que se está midiendo. Lo que se pide es identificar cuál explica lo observado.

Y el punto 3 es el que más errores encuentra. Un contenedor que funciona con diez elementos y falla con cero suele tener el mismo defecto: un caso especial que la implementación no contempla y que un nodo centinela habría eliminado.

Los guiones y los problemas propuestos siguen Rodríguez-Sánchez et al., 2020 y Garrido y Fdez-Valdivia, 2006, y la parte de STL, Garrido, 2016.

Bibliografía

Carrano, F. M. (2017). *Data Abstraction and Problem Solving with C++: Walls and Mirrors* (7.ª ed.). Pearson.

Garrido, A. (2016). *Fundamentos de Programación con la STL*. Editorial Universidad de Granada.

Garrido, A. (2017). *Programación genérica en C++: la biblioteca estándar*. Editorial Universidad de Granada.

Garrido, A. (2018). *Estructuras de Datos avanzadas con soluciones en C++*. Editorial Universidad de Granada.

Garrido, A., & Fdez-Valdivia, J. (2006). *Abstracción y Estructuras de Datos en C++*. Delta Publicaciones.

Koffman, E. B., & Wolfgang, P. A. T. (2006). *Objects, Abstraction, Data Structures and Design: Using C++*. Wiley.

Musser, D. R., Derge, G. J., & Saini, A. (2009). *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library* (3.ª ed.). Addison-Wesley.

Robson, R. (2013). *Using the STL* (2.ª ed.). Springer Verlag.

Rodríguez-Sánchez, R., Fdez-Valdivia, J., & García, J. A. (2020). *Estructuras de Datos en C++. Un enfoque práctico*.