

---

# Estructuras de Datos y Algoritmos

---



**APUNTES DE CLASE**  
(versión adaptada para la asignatura del Grado en  
Desarrollo de Videojuegos)

Ricardo Peña Mari  
Marco Antonio Gómez Martín  
Gonzalo Méndez Pozo  
Manuel Freire Morán  
Antonio Sánchez Ruiz-Granados  
Isabel Pita Andreu  
Ramón González del Campo  
Miguel Valero Espada  
Clara Segura Díaz  
Miguel Gómez-Zamalloa

Facultad de Informática  
Universidad Complutense de Madrid

20 de noviembre de 2023

Documento maquetado con TEXIS v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

# Estructuras de Datos y Algoritmos

## **Apuntes de clase**

Versión adaptada para la asignatura del Grado en Desarrollo de Videojuegos

**Facultad de Informática  
Universidad Complutense de Madrid**

**20 de noviembre de 2023**

Copyright © Ricardo Peña Mari, Marco Antonio Gómez Martín, Gonzalo Méndez Pozo, Manuel Freire Morán, Antonio Sánchez Ruiz-Granados, Isabel Pita Andreu, Ramón González del Campo, Miguel Valero Espada, Clara Segura Díaz, Miguel Gómez-Zamalloa

# Índice

---

|  |           |
|--|-----------|
| <b>1. Análisis de la eficiencia</b>  | <b>1</b>  |
| 1. Introducción . . . . .  | 1         |
| 2. Medidas asintóticas de la eficiencia . . . . .                          | 3         |
| 3. Jerarquía de órdenes de complejidad . . . . .                           | 6         |
| 4. Propiedades de los órdenes de complejidad . . . . .                     | 8         |
| 5. Reglas prácticas para el cálculo de la eficiencia . . . . .             | 9         |
| 5.1. Ejemplos de cálculo de complejidad de algoritmos iterativos . . . . . | 10        |
| Notas bibliográficas . . . . .   | 14        |
| Ejercicios . . . . .   | 14        |
| <b>2. Divide y Vencerás</b>  | <b>19</b> |
| 1. Introducción a los algoritmos recursivos . . . . .                      | 20        |
| 1.1. Recursión simple . . . . .  | 21        |
| 1.2. Recursión final . . . . .   | 23        |
| 1.3. Recursión múltiple . . . . .  | 25        |
| 1.4. Resumen de los distintos tipos de recursión . . . . .                 | 26        |
| 1.5. Implementación recursiva de la búsqueda binaria . . . . .             | 26        |
| 1.6. Algoritmos avanzados de ordenación . . . . .                          | 30        |
| 2. Análisis de la complejidad de algoritmos recursivos . . . . .           | 37        |
| 2.1. Ecuaciones de recurrencias . . . . .                                  | 37        |
| 2.2. Despliegue de recurrencias . . . . .                                  | 38        |
| 2.3. Resolución general de recurrencias . . . . .                          | 40        |
| 3. Introducción al esquema <i>divide y vencerás</i> . . . . .              | 42        |
| 4. Ejemplos de aplicación del esquema con éxito . . . . .                  | 44        |
| 5. Problema de selección . . . . .   | 45        |
| 6. Organización de un campeonato . . . . .                                 | 48        |
| 6.1. Implementación . . . . .  | 49        |
| 7. El problema del par más cercano . . . . .                               | 51        |
| 7.1. Corrección . . . . .  | 52        |
| 7.2. Implementación . . . . .  | 53        |
| 8. La determinación del umbral . . . . .                                   | 57        |
| Notas bibliográficas . . . . .   | 59        |
| Ejercicios . . . . .   | 59        |

|   |            |
|---|------------|
| <b>3. Vuelta Atrás</b>  | <b>63</b>  |
| 1. Motivación . . . . .   | 63         |
| 2. Introducción . . . . .   | 64         |
| 2.1. Esquema básico de la <i>vuelta atrás</i> . . . . .           | 66         |
| 2.2. Resolución del problema de las palabras . . . . .            | 67         |
| 3. Vuelta atrás con marcaje . . . . .                             | 68         |
| 4. Ejemplo: problema de las <i>n</i> -reinas . . . . .            | 69         |
| 5. Ejemplo de búsqueda de una sola solución: Dominó . . . . .     | 71         |
| 6. Ejemplo que no necesita desmarcar: El laberinto . . . . .      | 73         |
| 7. Optimización . . . . .   | 75         |
| 7.1. Ejemplo: Problema del viajante . . . . .                     | 75         |
| 7.2. Ejemplo: Problema de la mochila . . . . .                    | 77         |
| 8. Para terminar... . . . . .                                     | 78         |
| Notas bibliográficas . . . . .                                    | 79         |
| Ejercicios . . . . .  | 79         |
| <b>4. Implementación y uso de TADs</b>                            | <b>83</b>  |
| 1. Motivación . . . . .   | 83         |
| 2. Introducción . . . . .   | 84         |
| 2.1. Abstracción . . . . .  | 84         |
| 2.2. TADs de usuario . . . . .                                    | 85         |
| 2.3. Diseño con TADs . . . . .                                    | 88         |
| 3. Implementación de TADs . . . . .                               | 91         |
| 3.1. Tipos de operaciones y el modificador <i>const</i> . . . . . | 91         |
| 3.2. Uso de sub-TADs y extensión de TADs . . . . .                | 92         |
| 3.3. TADs genéricos . . . . .                                     | 93         |
| 3.4. Implementación, parcialidad y errores . . . . .              | 94         |
| 3.5. Implementaciones dinámicas y estáticas . . . . .             | 104        |
| 4. Probando y documentando TADs . . . . .                         | 108        |
| 4.1. Documentando TADs . . . . .                                  | 110        |
| 5. Para terminar... . . . . .                                     | 113        |
| Notas bibliográficas . . . . .                                    | 114        |
| Ejercicios . . . . .  | 114        |
| <b>5. Diseño e implementación de TADs lineales</b>                | <b>119</b> |
| 1. Motivación . . . . .   | 119        |
| 2. Estructuras de datos lineales . . . . .                        | 120        |
| 2.1. Array de elementos . . . . .                                 | 120        |
| 2.2. Nodos enlazados . . . . .                                    | 123        |
| 2.3. En el mundo real... . . . . .                                | 125        |
| 3. El TAD vector . . . . .  | 126        |
| 3.1. Implementación . . . . .                                     | 127        |
| 4. Pilas . . . . .  | 130        |
| 4.1. Implementación de pilas con vector dinámico . . . . .        | 130        |
| 4.2. Implementación de pilas con una lista enlazada . . . . .     | 132        |
| 5. Colas . . . . .  | 134        |
| 5.1. Implementación de colas con un vector . . . . .              | 135        |

|           |  |            |
|-----------|--|------------|
| 5.2.      | Implementación de colas con una lista enlazada . . . . .                 | 135        |
| 5.3.      | Implementación de colas con una lista enlazada y nodo fantasma . . . . . | 137        |
| 6.        | Colas dobles . . . . .   | 137        |
| 7.        | Listas . . . . .   | 141        |
| 7.1.      | Iteradores . . . . .   | 142        |
| 7.2.      | Implementación de un iterador básico . . . . .                           | 143        |
| 7.3.      | Usando iteradores para insertar elementos . . . . .                      | 145        |
| 7.4.      | Usando iteradores para eliminar elementos . . . . .                      | 146        |
| 7.5.      | Peligros de los iteradores . . . . .                                     | 146        |
| 7.6.      | En el mundo real... . . . . .  | 147        |
| 8.        | Para terminar... . . . . .   | 148        |
|           | Notas bibliográficas . . . . .   | 149        |
|           | Ejercicios . . . . .   | 149        |
| <b>6.</b> | <b>Diseño e implementación de TADs arborescentes</b>                     | <b>155</b> |
| 1.        | Motivación . . . . .   | 155        |
| 2.        | Introducción . . . . .   | 156        |
| 2.1.      | Modelo matemático . . . . .  | 156        |
| 2.2.      | Terminología . . . . .   | 157        |
| 2.3.      | Tipos de árboles . . . . .   | 158        |
| 3.        | Árboles binarios: operaciones . . . . .                                  | 158        |
| 4.        | Implementación de árboles binarios . . . . .                             | 159        |
| 5.        | Recorridos . . . . .   | 164        |
| 6.        | Implementación eficiente de los árboles binarios . . . . .               | 166        |
| 7.        | Implementación con <i>punteros inteligentes</i> . . . . .                | 169        |
| 8.        | Implementación estática ad-hoc de árboles binarios . . . . .             | 171        |
| 9.        | Árboles generales . . . . .  | 172        |
| 10.       | Para terminar... . . . . .   | 172        |
|           | Notas bibliográficas . . . . .   | 173        |
|           | Ejercicios . . . . .   | 173        |
| <b>7.</b> | <b>Diccionarios</b>  | <b>181</b> |
| 1.        | Motivación . . . . .   | 181        |
| 2.        | Introducción . . . . .   | 181        |
| 2.1.      | Especificación . . . . .   | 182        |
| 2.2.      | Implementación con acceso basado en búsqueda . . . . .                   | 183        |
| 3.        | Árboles de búsqueda . . . . .  | 184        |
| 3.1.      | Implementación . . . . .   | 185        |
| 3.2.      | Coste de las operaciones . . . . .                                       | 191        |
| 3.3.      | Recorrido de los elementos mediante un iterador . . . . .                | 192        |
| 3.4.      | Búsqueda en el diccionario con iteradores . . . . .                      | 194        |
| 4.        | Tablas dispersas . . . . .   | 195        |
| 4.1.      | Función de localización . . . . .  | 195        |
| 4.2.      | Colisiones . . . . .   | 197        |
| 5.        | Tablas dispersas abiertas . . . . .                                      | 197        |
| 5.1.      | Invariante de la representación . . . . .                                | 199        |
| 5.2.      | Implementación de las operaciones auxiliares . . . . .                   | 201        |

|           |  |            |
|-----------|--|------------|
| 5.3.      | Implementación de las operaciones públicas . . . . . | 202        |
| 5.4.      | Tablas dinámicas . . . . .                           | 204        |
| 5.5.      | Recorrido usando iteradores . . . . .                | 205        |
| 6.        | Funciones de localización . . . . .                  | 207        |
| 6.1.      | Para enteros . . . . .                               | 208        |
| 6.2.      | Para cadenas . . . . .                               | 209        |
| 6.3.      | Para clases definidas por el programador . . . . .   | 209        |
| 7.        | En el mundo real... . . . . .                        | 210        |
| 8.        | Para terminar... . . . . .                           | 211        |
|           | Notas bibliográficas . . . . .                       | 211        |
|           | Ejercicios . . . . .                                 | 211        |
| <b>8.</b> | <b>Aplicaciones de tipos abstractos de datos</b>     | <b>215</b> |
| 1.        | Problemas resueltos: . . . . .                       | 215        |
| 1.1.      | Confederación hidrográfica. . . . .                  | 215        |
| 1.2.      | Motor de búsqueda. . . . .                           | 221        |
| 1.3.      | Agencia de viajes. . . . .                           | 224        |
| 1.4.      | E-reader. . . . .                                    | 227        |
| 2.        | Problemas propuestos: . . . . .                      | 237        |
|           | <b>Bibliografía</b>                                  | <b>243</b> |

# Índice de figuras

---

|     |   |     |
|-----|---|-----|
| 1.  | Crecimiento de distintas funciones de complejidad . . . . .   | 7   |
| 2.  | Jerarquía de órdenes de complejidad . . . . .   | 8   |
| 1.  | Ejecución de <i>factorial(3)</i> . . . . .  | 21  |
| 2.  | Llamadas recursivas de <i>factorial(3)</i> . . . . .  | 21  |
| 3.  | Vuelta de las llamadas recursivas de <i>factorial(3)</i> . . . . .  | 22  |
| 4.  | Ejecución de <i>fib(4)</i> . . . . .  | 26  |
| 5.  | Cálculo del punto medio . . . . .   | 27  |
| 6.  | Búsqueda en la mitad derecha . . . . .  | 27  |
| 7.  | Búsqueda en la mitad izquierda . . . . .  | 27  |
| 8.  | Planteamiento del algoritmo <i>quicksort</i> . . . . .  | 31  |
| 9.  | Diseño de <i>particion</i> . . . . .  | 32  |
| 10. | Planteamiento del algoritmo <i>mergesort</i> . . . . .  | 34  |
| 11. | Diseño de <i>mezcla</i> . . . . .   | 35  |
| 12. | Solución gráfica del problema del torneo . . . . .  | 50  |
| 13. | Razonamiento de corrección del problema del par más cercano . . . . .                                     | 53  |
| 1.  | Función de abstracción y vistas del usuario y el implementador . . . . .                                  | 96  |
| 2.  | Conjunto antes y después de insertar el elemento 4 . . . . .  | 99  |
| 3.  | Conjuntos equivalentes (según la función de equivalencia del TAD) . . . . .                               | 99  |
| 1.  | Posiciones y valores de cada nodo de un árbol. . . . .  | 157 |
| 2.  | Distintas formas de recorrer un árbol. . . . .  | 164 |
| 1.  | Dos árboles de búsqueda distintos pero equivalentes. . . . .  | 186 |
| 2.  | Ejemplo de tabla abierta. La función de localización usada en el ejemplo es claramente mejorable. . . . . | 198 |



---

## Capítulo 1

# Análisis de la eficiencia de los algoritmos<sup>1</sup>

---

*No puede haber orden cuando hay mucha prisa*

Seneca

**RESUMEN:** En este tema se muestra la importancia de contar con algoritmos eficientes, se define qué se entiende por coste de un algoritmo y se enseña a comparar las funciones de coste de distintos algoritmos con el fin de decidir cuál de ellos es preferible. Se define una jerarquía de órdenes de complejidad que ilustra la separación entre algoritmos eficientes y los que no lo son.

## 1. Introducción

- \* Aproximadamente cada año y medio se duplica el número de instrucciones por segundo que son capaces de ejecutar los computadores. Ello puede inducir a pensar que basta con esperar algunos años para que problemas que hoy necesitan muchas horas de cálculo puedan resolverse en pocos segundos.
- \* Sin embargo hay algoritmos tan ineficientes que ningún avance en la velocidad de las máquinas podrá conseguir para ellos tiempos aceptables. El factor predominante que delimita lo que es soluble en un tiempo razonable de lo que no lo es, es precisamente el **algoritmo** elegido para resolver el problema.
- \* En este capítulo enseñaremos a medir la **eficiencia** de los algoritmos y a comparar la eficiencia de distintos algoritmos para un mismo problema. Después de la **corrección**, conseguir eficiencia debe ser el principal objetivo del programador. Mediremos principalmente la eficiencia en **tiempo de ejecución**, pero los mismos conceptos son aplicables a la medición de la eficiencia en **espacio**, es decir a medir la memoria que necesita el algoritmo.
- \* La siguiente función ordena los  $n$  primeros elementos del vector  $v$  que recibe como parámetro utilizando el método de selección:

---

<sup>1</sup>Ricardo Peña es el autor principal de este tema.

---

```

1 void ordenaSeleccion(int v[], int n) {
2
3     for (int i = 0; i < n-1; i++) {
4         // pmin calcula la posición del mínimo de v[i..n-1]
5         int pmin = i;
6         for (int j = i+1; j < n; j++)
7             if (v[j] < v[pmin])
8                 pmin = j;
9
10        // Tenemos la posición del mínimo. Lo ponemos en v[i]
11        // Código equivalente a swap(v[i], v[pmin])
12        int temp = v[i];
13        v[i] = v[pmin];
14        v[pmin] = temp;
15    }
16 }
```

---

- \* Una manera de medir la eficiencia en tiempo de esta función es **contar** cuántas instrucciones de cada tipo se ejecutan, multiplicar este número por el tiempo que emplea la instrucción en ejecutarse, y realizar la suma para los diferentes tipos. Sean

$$\begin{aligned}
 t_a &= \text{tiempo de una asignación entre enteros} \\
 t_c &= \text{tiempo de una comparación entre enteros} \\
 t_i &= \text{tiempo de incrementar un entero} \\
 t_v &= \text{tiempo de acceso a un elemento de un vector}
 \end{aligned} \tag{1.1}$$

- La línea (3) da lugar a una asignación, a  $n - 1$  incrementos y a  $n$  comparaciones, es decir, a un tiempo  $t_a + (n - 1)t_i + nt_c$ .
- La línea (5) da lugar a un tiempo  $(n - 1)t_a$ .
- El bucle interior `for` se ejecuta  $n - 1$  veces, cada una con un valor diferente de  $i$ . Para cada valor de  $i$  y siguiendo el cálculo hecho para la (3), la línea (6) da lugar a un tiempo  $t_a + (n - i - 1)t_i + (n - i)t_c$ .
- La línea (7) da, para cada valor de  $i$ , un tiempo mínimo de  $(n - i - 1)(2t_v + t_c)$ , suponiendo que la instrucción `pmin = j` nunca se ejecuta. A ello hay que sumar  $(n - i - 1)t_a$  en el caso más desfavorable en que dicha rama se ejecute todas las veces. El caso promedio tendrá un tiempo de ejecución entre estos dos.
- Finalmente, la línea (9) dará lugar a un tiempo  $(n - 1)(4t_v + 3t_a)$ .
- Por tanto, el tiempo del bucle interior `for`, en el caso más desfavorable, se calcula mediante el siguiente sumatorio:

$$\sum_{i=0}^{n-2} (t_a + t_c + (n - i - 1)(t_i + 2t_v + t_a + 2t_c)) = P(n - 1) + \frac{1}{2}Qn(n - 1)$$

$$\text{siendo } P = t_a + t_c \text{ y } Q = t_i + 2t_v + t_a + 2t_c.$$

Para no cansar al lector con tediosos cálculos, concluiremos que la suma de todos estos tiempos da lugar a dos polinomios de la forma:

$$\begin{aligned}
 T_{\min} &= An^2 - Bn + C \\
 T_{\max} &= A'n^2 - B'n + C'
 \end{aligned}$$

donde  $A, A', B, B', C$  y  $C'$  son expresiones racionales positivas que dependen linealmente de los tiempos elementales descritos en 1.1.

- \* En este sencillo ejemplo se observan claramente los tres factores de los que en general depende el tiempo de ejecución de un algoritmo:
  1. El **tamaño** de los datos de entrada, simbolizado aquí por la longitud  $n$  del vector.
  2. El **contenido** de los datos de entrada, que en el ejemplo hace que el tiempo para diferentes vectores del mismo tamaño esté comprendido entre los valores  $T_{min}$  y  $T_{max}$ .
  3. El código generado por el **compilador** y el **computador** concreto utilizados, que afectan a los tiempos elementales 1.1.
- \* Como el objetivo es poder comparar algoritmos independientemente del valor de los datos de entrada, el segundo factor podemos eliminarlo de dos maneras:
  - O bien midiendo solo el **caso peor**, es decir la ejecución que tarde más tiempo de todos los ejemplares de tamaño  $n$ .
  - O bien midiendo todos los casos de tamaño  $n$  y calculando el tiempo del **caso promedio**.

En esta asignatura nos concentraremos en el caso peor por dos razones:

1. El caso peor establece una cota superior fiable para **todos** los casos del mismo tamaño.
2. El caso peor es más fácil de calcular.

El caso promedio es más difícil de calcular pero a veces es más informativo. Además exige conocer la probabilidad con la que se va a presentar cada caso. Muy raramente puede ser útil conocer el **caso mejor** de un algoritmo para un tamaño  $n$  dado. Ese coste es una *cota inferior* al coste de cualquier otro ejemplar de ese tamaño.

- \* El tercer factor impediría comparar algoritmos escritos en diferentes lenguajes, traducidos por diferentes compiladores, o ejecutados en diferentes máquinas. El criterio que seguiremos es **ignorar** estos factores.
- \* Por tanto solo mediremos la eficiencia de un algoritmo en función del **tamaño** de los datos de entrada. Este criterio está en la base de lo que llamaremos **medida asintótica** de la eficiencia.

## 2. Medidas asintóticas de la eficiencia

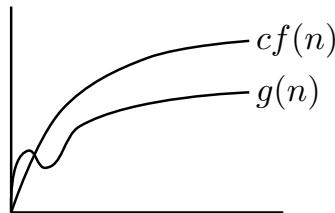
- \* El **criterio asintótico** para medir la eficiencia de los algoritmos tiene como objetivo comparar algoritmos **independientemente** de los lenguajes en que están escritos, de las máquinas en que se ejecutan y del valor concreto de los datos que reciben como entrada. Tan solo considera importante el **tamaño** de dichos datos. Para cada problema habrá que definir qué se entiende por tamaño del mismo.
- \* Se basa en tres principios:

1. El coste o eficiencia es una función que solo depende del tamaño de la entrada, e.g.  $f(n) = n^2$ .
  2. Las constantes multiplicativas o aditivas no se tienen en cuenta, e.g.  $f(n) = n^2$  y  $g(n) = 3n^2 + 27$  se consideran costes equivalentes.
  3. La comparación entre funciones de coste se hará para valores de  $n$  **suficientemente grandes**, es decir los costes para tamaños pequeños se consideran irrelevantes.
- \* Sea  $\mathbb{N}$  el conjunto de los números naturales y  $\mathbb{R}^+$  el conjunto de los reales estrictamente positivos.

**Definición 1.1** *Sea  $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$ . El conjunto de las funciones **del orden de**  $f(n)$ , denotado  $\mathcal{O}(f(n))$ , se define como:*

$$\mathcal{O}(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. g(n) \leq cf(n)\}$$

Asimismo, diremos que una función  $g$  es **del orden de**  $f(n)$  cuando  $g \in \mathcal{O}(f(n))$ . También diremos que  $g$  **está** en  $\mathcal{O}(f(n))$ .



- \* Generalizando, admitiremos también que una función negativa o indefinida para un número finito de valores de  $n$  pertenece al conjunto  $\mathcal{O}(f(n))$  si eligiendo  $n_0$  suficientemente grande, satisface la definición.
- \* Esta garantiza que, si el tiempo de ejecución  $g(n)$  de una implementación concreta de un algoritmo es del orden de  $f(n)$ , entonces el tiempo  $g'(n)$  de cualquier otra implementación del mismo que difiera de la anterior en el lenguaje, el compilador, o/y la máquina empleada, también será del orden de  $f(n)$ . Por tanto, el coste  $\mathcal{O}(f(n))$  expresa la eficiencia del algoritmo *per se*, no el de una implementación concreta del mismo.
- \* Las clases  $\mathcal{O}(f(n))$  para diferentes funciones  $f(n)$  se denominan **clases de complejidad**, u **órdenes de complejidad**. Algunos órdenes tienen nombre propio. Así, al orden de complejidad  $\mathcal{O}(n)$  se le llama **lineal**, al orden  $\mathcal{O}(n^2)$ , **cuadrático**, el orden  $\mathcal{O}(1)$  describe la clase de las funciones **constantes**, etc. Eligiremos como representante del orden  $\mathcal{O}(f(n))$  la función  $f(n)$  más sencilla posible dentro del mismo.
- \* Nótese que la definición 1.1 se puede aplicar tanto a un análisis en el caso peor, como a un análisis en el caso promedio. Por ejemplo, hay algoritmos cuyo coste en tiempo está en  $\mathcal{O}(n^2)$  en el caso peor y en  $\mathcal{O}(n \log n)$  en el caso promedio.
- \* Nótese también que las unidades en que se mide el coste en tiempo (horas, segundos, milisegundos, etc.), o en memoria (octetos, palabras, celdas de longitud fija, etc.) **no**

**son relevantes** en la complejidad asintótica: dos unidades distintas se diferencian en una constante multiplicativa (e.g.  $120 n^2$  segundos son  $2 n^2$  minutos, ambos en  $\mathcal{O}(n^2)$ ).

- \* Aplicando directamente la definición de  $\mathcal{O}(f(n))$ , demostremos que  $(n+1)^2 \in O(n^2)$ . Un modo de hacerlo es por inducción sobre  $n$ . Elegimos  $n_0 = 1$  y  $c = 4$ , es decir demostraremos  $\forall n \geq 1 . (n+1)^2 \leq 4n^2$ :

**Caso base:**  $n = 1, (1+1)^2 \leq 4 \cdot 1^2$

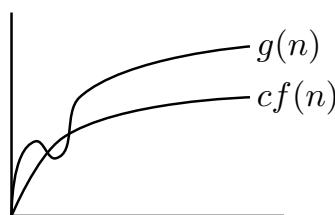
**Paso inductivo:** h.i.  $(n+1)^2 \leq 4n^2$ . Demostrémoslo para  $n + 1$ :

$$\begin{aligned} (n+1+1)^2 &\leq 4(n+1)^2 \\ (n+1)^2 + 1 + 2(n+1) &\leq 4n^2 + 4 + 8n \\ (n+1)^2 &\leq 4n^2 + \underbrace{6n+1}_{\geq 0} \end{aligned}$$

- \* También podemos probar que  $3^n \notin O(2^n)$ . Si perteneciera, existiría  $c \in \mathbb{R}^+$ ,  $n_0 \in \mathbb{N}$  tales que  $3^n \leq c \cdot 2^n$  para todo  $n \geq n_0$ . Esto implicaría que  $(\frac{3}{2})^n \leq c$  para todo  $n \geq n_0$ . Pero esto es falso porque dado un  $c$  cualquiera, bastaría tomar  $n > \log_{1,5} c$  para que  $(\frac{3}{2})^n > c$ , es decir  $(\frac{3}{2})^n$  no se puede acotar superiormente.
- \* La notación  $\mathcal{O}(f(n))$  nos da una cota superior al tiempo de ejecución  $t(n)$  de un algoritmo. Normalmente estaremos interesados en la **menor** función  $f(n)$  tal que  $t(n) \in \mathcal{O}(f(n))$ . Una forma de realizar un análisis más completo es encontrar además la **mayor** función  $g(n)$  que sea una cota inferior de  $t(n)$ . Para ello introducimos la siguiente medida.

**Definición 1.2** Sea  $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$ . El conjunto  $\Omega(f(n))$ , leído **omega de  $f(n)$** , se define como:

$$\Omega(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} . \forall n \geq n_0 . g(n) \geq cf(n)\}$$



- \* Es frecuente confundir la medida  $\mathcal{O}(f(n))$  como aplicable al caso peor y la medida  $\Omega(f(n))$  como aplicable al caso mejor. Esta idea es **errónea**. Aplicaremos **ambas medidas al caso peor** (también podríamos aplicar ambas al caso promedio, o al caso mejor). Si el tiempo  $t(n)$  de un algoritmo en el caso peor está en  $\mathcal{O}(f(n))$  y en  $\Omega(g(n))$ , lo que estamos diciendo es que  $t(n)$  no puede valer más que  $c_1 f(n)$ , ni menos que  $c_2 g(n)$ , para dos constantes apropiadas  $c_1$  y  $c_2$  y valores de  $n$  suficientemente grandes.
- \* Es fácil demostrar (ver ejercicios) el llamado **principio de dualidad**:  $g(n) \in \mathcal{O}(f(n))$  si y solo si  $f(n) \in \Omega(g(n))$ .

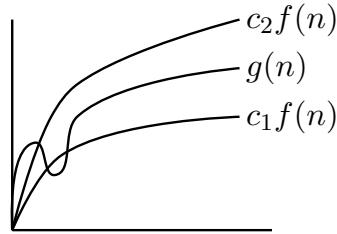
- \* Sucede con frecuencia que una misma función  $f(n)$  es a la vez cota superior e inferior del tiempo  $t(n)$  (peor, promedio, etc.) de un algoritmo. Para tratar estos casos, introducimos la siguiente medida.

**Definición 1.3** El conjunto de funciones  $\Theta(f(n))$ , leído del orden exacto de  $f(n)$ , se define como:

$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

También se puede definir como:

$$\Theta(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0 . c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$



- \* Siempre que sea posible, daremos el orden exacto del coste de un algoritmo, por ser más informativo que dar solo una cota superior.

### 3. Jerarquía de órdenes de complejidad

- \* Es importante visualizar las implicaciones prácticas de que el coste de un algoritmo pertenezca a una u otra clase de complejidad. La Figura 1 muestra el crecimiento de algunas de estas funciones, suponiendo que expresan un tiempo en milisegundos ( $ms$  = milisegundos,  $s$  = segundos,  $m$  = minutos,  $h$  = horas, etc.).
- \* Se aprecia inmediatamente la **extraordinaria eficiencia** de los algoritmos de coste en  $\mathcal{O}(\log n)$ : pasar de un tamaño de  $n = 10$  a  $n = 1\,000\,000$  solo hace que el tiempo crezca de 1 milisegundo a 6. La búsqueda binaria en un vector ordenado, y la búsqueda en ciertas estructuras de datos de este curso, tienen este coste en el caso peor.
- \* En sentido contrario, los algoritmos de coste  $\mathcal{O}(2^n)$  son **prácticamente inútiles**: mientras que un problema de tamaño  $n = 10$  se resuelve en aproximadamente un segundo, la edad del universo conocido ( $1,4 \times 10^8$  siglos) sería totalmente insuficiente para resolver uno de tamaño  $n = 100$ . Algunos algoritmos de *vuelta atrás* que veremos en este curso tienen ese coste en el caso peor.
- \* Esta tabla confirma la afirmación hecha al comienzo de este capítulo de que para ciertos algoritmos es inútil esperar a que los computadores sean más rápidos. Es más productivo invertir esfuerzo en diseñar **mejores algoritmos** para ese problema.

| $n$    | $\log_{10} n$ | $n$     | $n \log_{10} n$ | $n^2$    | $n^3$         | $2^n$                        |
|--------|---------------|---------|-----------------|----------|---------------|------------------------------|
| 10     | 1 ms          | 10 ms   | 10 ms           | 0.1 s    | 1 s           | 1.02 s                       |
| $10^2$ | 2 ms          | 0.1 s   | 0.2 s           | 10 s     | 16.67 m       | $4.02 \times 10^{20}$ sig    |
| $10^3$ | 3 ms          | 1 s     | 3 s             | 16.67 m  | 11.57 d       | $3.4 \times 10^{291}$ sig    |
| $10^4$ | 4 ms          | 10 s    | 40 s            | 1.16 d   | 31.71 a       | $6.3 \times 10^{3000}$ sig   |
| $10^5$ | 5 ms          | 1.67 m  | 8.33 m          | 115.74 d | 317.1 sig     | $3.16 \times 10^{30093}$ sig |
| $10^6$ | 6 ms          | 16.67 m | 1.67 h          | 31.71 a  | 317 097.9 sig | $3.1 \times 10^{301020}$ sig |

Figura 1: Crecimiento de distintas funciones de complejidad

- \* Para mejorar la intuición anterior, hagamos el siguiente experimento: supongamos seis algoritmos con los costes anteriores, tales que tardan todos ellos 1 hora en resolver un problema de tamaño  $n = 100$ . ¿Qué ocurre si duplicamos la velocidad del computador? O lo que es lo mismo, ¿qué ocurre si duplicamos el tiempo disponible?

| $t(n)$               | $t = 1h.$ | $t = 2h.$     |
|----------------------|-----------|---------------|
| $k_1 \cdot \log n$   | $n = 100$ | $n = 10\,000$ |
| $k_2 \cdot n$        | $n = 100$ | $n = 200$     |
| $k_3 \cdot n \log n$ | $n = 100$ | $n = 178$     |
| $k_4 \cdot n^2$      | $n = 100$ | $n = 141$     |
| $k_5 \cdot n^3$      | $n = 100$ | $n = 126$     |
| $k_6 \cdot 2^n$      | $n = 100$ | $n = 101$     |

- \* Observamos que mientras el de coste logarítmico es capaz de resolver problemas 100 veces más grandes, el de coste exponencial resuelve un tamaño prácticamente igual al anterior. Obsérvese que los de coste  $\mathcal{O}(n)$  y  $\mathcal{O}(n \log n)$  se comportan de acuerdo a la intuición de un usuario no informático: al duplicar la velocidad del computador (o el tiempo disponible), se duplica aproximadamente el tamaño del problema resuelto. En los de coste  $\mathcal{O}(n^k)$ , al duplicar la velocidad, el tamaño se multiplica por un factor  $\sqrt[k]{2}$ .
- \* En la Figura 2 se muestra la **jerarquía de órdenes de complejidad**. Las inclusiones estrictas expresan que se trata de clases distintas. Los algoritmos cuyos costes están en la parte izquierda resuelven problemas que se denominan **tratables**. Estos costes se denominan en su conjunto **polinomiales**. Hay problemas que solo admiten algoritmos de complejidad exponencial o superior. Se llaman **intratables**.
- \* También hay muchos problemas interesantes cuyos mejores algoritmos conocidos son exponenciales en el caso peor, pero no se sabe si existirán para ellos algoritmos polinomiales. Se llaman **NP-completos** y se verán en el próximo curso. El más conocido de todos es el problema **SAT** que consiste en determinar si una fórmula de la lógica proposicional es satisfactible.

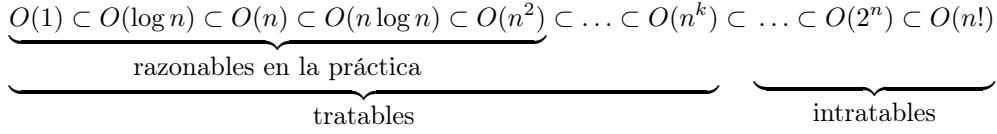


Figura 2: Jerarquía de órdenes de complejidad

## 4. Propiedades de los órdenes de complejidad

- \*  $O(a \cdot f(n)) = O(f(n))$  con  $a \in \mathbb{R}^+$ .

( $\subseteq$ )  $g \in O(a \cdot f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que  $\forall n \geq n_0 . g(n) \leq c \cdot a \cdot f(n)$ . Tomando  $c' = c \cdot a$  se cumple que  $\forall n \geq n_0 . g(n) \leq c' \cdot f(n)$ , luego  $g \in O(f(n))$ .

( $\supseteq$ )  $g \in O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que  $\forall n \geq n_0 . g(n) \leq c \cdot f(n)$ .

Entonces tomando  $c' = \frac{c}{a}$  se cumple que  $\forall n \geq n_0 . g(n) \leq c' \cdot a \cdot f(n)$ , luego  $g \in O(a \cdot f(n))$ .

- \* La base del logaritmo no importa:  $O(\log_a n) = O(\log_b n)$ , con  $a, b > 1$ . La demostración es inmediata sabiendo que:

$$\log_b n = \frac{\log_a n}{\log_a b}$$

- \* Si  $f \in O(g)$  y  $g \in O(h)$ , entonces  $f \in O(h)$ .

$$\begin{aligned} f \in O(g) &\Rightarrow \exists c_1 \in \mathbb{R}^+, n_1 \in \mathbb{N} \text{ tal que } \forall n \geq n_1 . f(n) \leq c_1 \cdot g(n) \\ g \in O(h) &\Rightarrow \exists c_2 \in \mathbb{R}^+, n_2 \in \mathbb{N} \text{ tal que } \forall n \geq n_2 . g(n) \leq c_2 \cdot h(n) \end{aligned}$$

Tomando  $n_0 = \max(n_1, n_2)$  y  $c = c_1 \cdot c_2$ , se cumple

$$\forall n \geq n_0 . f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

Y por tanto  $f \in O(h)$ .

- \* Regla de la suma:  $O(f + g) = O(\max(f, g))$ .

( $\subseteq$ )  $h \in O(f + g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} . \forall n \geq n_0 . h(n) \leq c \cdot (f(n) + g(n))$ . Pero  $f \leq \max(f, g)$  y  $g \leq \max(f, g)$ , luego:

$$h(n) \leq c \cdot (\max(f(n), g(n)) + \max(f(n), g(n))) = 2 \cdot c \cdot \max(f(n), g(n))$$

Tomando  $c' = 2 \cdot c$  se cumple que  $\forall n \geq n_0 . h(n) \leq c' \cdot \max(f(n), g(n))$  y por tanto  $h \in O(\max(f, g))$ .

( $\supseteq$ )  $h \in O(\max(f, g)) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} . \forall n \geq n_0 . h(n) \leq c \cdot \max(f(n), g(n))$ . Pero  $\max(f, g) \leq f + g$ , luego  $h \in O(f + g)$  trivialmente.

- \* Regla del producto: Si  $g_1 \in O(f_1)$  y  $g_2 \in O(f_2)$ , entonces  $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$ . La demostración es similar.

\* Teorema del límite

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) = O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ y } g \notin O(f) \Leftrightarrow O(f) \subset O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f \notin O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) \supset O(g)$

\* Por el principio de dualidad, también tenemos:

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow g \in \Omega(f) \text{ y } f \in \Omega(g) \Leftrightarrow \Omega(f) = \Omega(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g \in \Omega(f) \text{ y } f \notin \Omega(g) \Leftrightarrow \Omega(f) \supset \Omega(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \notin \Omega(f) \text{ y } f \in \Omega(g) \Leftrightarrow \Omega(f) \subset \Omega(g)$

\* Aplicando la definición de  $\Theta(f)$ , también tenemos:

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow g \in \Theta(f) \text{ y } f \in \Theta(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ pero } f \notin \Theta(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \in O(f) \text{ pero } g \notin \Theta(f)$

## 5. Reglas prácticas para el cálculo de la eficiencia

1. Las instrucciones de asignación, de entrada-salida, los accesos a elementos de un vector y las expresiones aritméticas y lógicas (siempre que no involucren variables cuyos tamaños dependan del tamaño del problema) tendrán un coste constante,  $\Theta(1)$ . No se cuentan los *return*.
2. Para calcular el coste de una composición secuencial de instrucciones,  $S_1; S_2$  se suman los costes de  $S_1$  y  $S_2$ . Si el coste de  $S_1$  está en  $\Theta(f_1(n))$  y el de  $S_2$  está en  $\Theta(f_2(n))$ , entonces el coste de  $S_1; S_2$  está en:  $\Theta(f_1(n) + f_2(n)) = \Theta(\max(f_1(n), f_2(n)))$ .
3. Para calcular el coste de una instrucción *condicional*:

**if** ( $B$ ) { $S_1$ } **else** { $S_2$ }

Si el coste de  $S_1$  está en  $\mathcal{O}(f_1(n))$ , el de  $S_2$  está en  $\mathcal{O}(f_2(n))$  y el de  $B$  en  $\mathcal{O}(f_B(n))$ , podemos señalar dos casos para el *condicional*:

- Caso peor:  $\mathcal{O}(\max(f_B(n), f_1(n), f_2(n)))$ .
- Caso promedio:  $\mathcal{O}(\max(f_B(n), f(n)))$  donde  $f(n)$  es el promedio de  $f_1(n)$  y  $f_2(n)$ .

Se pueden encontrar expresiones análogas a éstas para la clase Omega.

4. Bucles con la forma:

**while** ( $B$ ) {  $S$  }

Para calcular el coste de un bucle hay que calcular primero el coste de cada vuelta del bucle y después sumar los costes de todas las vueltas que se hagan en el bucle. El número de iteraciones dependerá de lo que tarde en hacerse falso  $B$ , teniendo en cuenta los datos concretos sobre los que se ejecute el programa y lo grandes que sean.

### 5.1. Ejemplos de cálculo de complejidad de algoritmos iterativos

De manera práctica, se analizará el código de los algoritmos presentados de arriba hacia abajo y de dentro hacia fuera.

- Búsqueda secuencial

---

```

1 bool buscaSec( int v[], int n, int x ) {
2     int j;
3     bool encontrado;
4
5     j = 0;
6     encontrado = false;
7     while ( (j < n) && ! encontrado ) {
8         encontrado = ( v[j] == x );
9         j++;
10    }
11    return encontrado;
12 }
```

---

- Caso peor: el elemento buscado  $x$  condición no está en el vector
  - Cuerpo del bucle while: 4 (3 en la línea 8 y 1 en la línea 9)
  - Bucle while:  $\underbrace{n}_{j=0..n-1} * (\underbrace{4}_{cuerpo} + \underbrace{3}_{condición}) + \underbrace{3}_{condición} = 7n + 3$
  - Total:  $7n + 3 + 2_{(inicializaciones)} = 7n + 5$
- Caso mejor: el elemento buscado  $x$  está en la primera posición del vector; solo se entra una vez en el bucle while
  - Cuerpo del bucle while: 4
  - Bucle while:  $(4 + 3) + 3 = 10$
  - Total:  $10 + 2 = 12$

- Búsqueda binaria

---

```

1 int buscaBin( int v[], int n, int x ) {
2     int izq, der, centro;
3
4     izq = -1;
5     der = n;
6     while ( der != izq+1 ) {
7         centro = (izq+der) / 2;
8         if ( v[centro] <= x )
9             izq = centro;
10        else
11            der = centro;
12    }
13    return izq;
14 }
```

---

- En esta ocasión no hay caso mejor ni peor, el bucle se ejecuta el mismo número de veces independientemente de la posición de  $x$  en el vector o incluso si  $x$  no aparece en el vector. Aunque el elemento buscado esté en el centro, el bucle sigue ejecutándose hasta acotar un subvector de longitud 1.

- Cuerpo del bucle *while*:  $\underbrace{3}_{\text{línea 7}} + \underbrace{3}_{\text{if}} = 6$
- Bucle *while*:  $\underbrace{\log(n)}_{\text{vueltas while}} * (\underbrace{6}_{\text{cuerpo}} + \underbrace{2}_{\text{condición}}) + \underbrace{2}_{\text{condición}} = 8\log(n) + 2$
- Total:  $(8\log(n) + 2) + 2_{(\text{inicializaciones})} = 8\log(n) + 4$

En caso de que el elemento buscado estuviese repetido, ¿qué posición se devolvería?

■ Ordenación por inserción

---

```

1 void ordenaIns ( int v[], int n ) {
2     int i, j, x;
3
4     for ( i = 1; i < n; i++ ) {
5         x = v[i];
6         j = i-1;
7         while ((j >= 0) && (v[j] > x)) {
8             v[j+1] = v[j];
9             j = j-1;
10        }
11        v[j+1] = x;
12    }
13 }
```

---

- Caso peor: el vector está ordenado a la inversa; la segunda parte de la condición del *while* se cumple siempre
  - Cuerpo del bucle *while*: 6 (4 de la línea 8 y 2 de la línea 9)
  - Bucle *while*:  $\underbrace{i}_{j=0..i-1} * (\underbrace{6}_{\text{cuerpo}} + \underbrace{4}_{\text{condición}}) + \underbrace{4}_{\text{condición}} = 10i + 4$
  - Cuerpo del bucle *for*:  $\underbrace{(10i + 4)}_{\text{while}} + \underbrace{2}_{l5} + \underbrace{2}_{l6} + \underbrace{3}_{l11} + \underbrace{1}_{i++} = 10i + 12$
  - Bucle *for*:

$$\begin{aligned}
 & \sum_{i=1}^{n-1} (\underbrace{(10i + 12)}_{\text{cuerpo}} + \underbrace{1}_{\text{condición}}) + \underbrace{1}_{\text{condición}} + \underbrace{1}_{\text{ini}} = \sum_{i=1}^{n-1} (10i + 13) + 2 = \\
 & = 10 \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 13 + 2 = 10 \frac{n(n-1)}{2} + 13(n-1) + 2 = \\
 & = 5n^2 - 5n + 13n - 13 + 2 = 5n^2 + 8n - 11
 \end{aligned}$$

- Caso mejor: el vector está ordenado; la segunda parte de la condición del *while* no se cumple nunca
  - Bucle *while*:  $4_{(\text{condición})}$
  - Cuerpo del bucle *for*:  $4 + 2 + 2 + 3 + 1 = 12$
  - Bucle *for*:

$$\sum_{i=1}^{n-1} (12 + 1) + 1 + 1 = \sum_{i=1}^{n-1} 13 + 2 = 13(n-1) + 2 = 13n - 11$$

- Ordenación por selección

---

```

1 void ordenaSel ( int v[], int n ) {
2     int i, j, menor, aux;
3
4     for ( i = 0; i < n; i++ ) {
5         menor = i;
6         for ( j = i+1; j < n; j++ )
7             if ( v[j] < v[menor] )
8                 menor = j;
9         if ( i != menor ) {
10            aux = v[i];
11            v[i] = v[menor];
12            v[menor] = aux;
13        }
14    }
15 }
```

---

- Caso peor: vector desordenado

- Cuerpo del *for* interno:  $\underbrace{3}_{if} + \underbrace{1}_{l8} + \underbrace{1}_{j++} = 5$
- Bucle *for* interno:  $\underbrace{(n - i - 1)}_{j=i+1..n-1} (\underbrace{5}_{cuerpo} + \underbrace{1}_{condición}) + \underbrace{1}_{condición} + \underbrace{2}_{ini} = 6n - 6i - 3$
- Cuerpo del *for* externo:  $\underbrace{1}_{l5} + \underbrace{(6n - 6i - 3)}_{for} + \underbrace{1}_{if} + \underbrace{2}_{l10} + \underbrace{3}_{l11} + \underbrace{2}_{l12} + \underbrace{1}_{i++}$   
 $= 6n - 6i + 7$
- Bucle *for* externo:

$$\begin{aligned}
& \sum_{i=0}^{n-1} ((6n - 6i + 7) + \underbrace{1}_{condición}) + \underbrace{1}_{condición} + \underbrace{1}_{ini} = 6 \sum_{i=0}^{n-1} n - 6 \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 8 + 2 = \\
& = 6n^2 - 6 \frac{n(n-1)}{2} + 8n + 2 = 6n^2 - \frac{6}{2}n^2 + \frac{6}{2}n + 8n + 2 = 3n^2 + 11n + 2
\end{aligned}$$

- Caso mejor: vector ordenado; la condición del *if* más interno no se cumple nunca, y como consecuencia la del más externo tampoco:

- Cuerpo del *for* interno: 4
- Bucle *for* interno:  $(n - i - 1)(4 + 1) + 1 + 2 = 5n - 5i - 2$
- Cuerpo del *for* externo:  $1 + (5n - 5i - 2) + 1 + 1 = 5n - 5i + 1$
- Bucle *for* externo:

$$\begin{aligned}
& \sum_{i=0}^{n-1} ((5n - 5i + 1) + 1) + 1 + 1 = 5 \sum_{i=0}^{n-1} n - 5 \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 2 + 2 = \\
& = 5n^2 - 5 \frac{n(n-1)}{2} + 2n + 2 = 5n^2 - \frac{5}{2}n^2 + \frac{5}{2}n + 2n + 2 = \frac{5}{2}n^2 + \frac{9}{2}n + 2
\end{aligned}$$

¿Qué sucede si el vector está ordenado al revés?

- Ordenación por el método de la burbuja modificado

---

```

1 void ordenaBur ( int v[], int n ) {
2     int i, j, aux;
3     bool modificado;
4
5     i = 0;
6     modificado = true;
7     while ( (i < num-1) && modificado ) {
8         modificado = false;
9         for ( j = n-1; j > i; j-- )
10            if ( v[j] < v[j-1] ) {
11                aux = v[j];
12                v[j] = v[j-1];
13                v[j-1] = aux;
14                modificado = true;
15            }
16            i++;
17        }
18    }

```

---

- Caso peor: El vector está ordenado al revés; se entra siempre en el *if*:

- Cuerpo del bucle *for*:  $\underbrace{4}_{if} + \underbrace{2}_{l11} + \underbrace{4}_{l12} + \underbrace{3}_{l13} + \underbrace{1}_{l14} + \underbrace{1}_{j--} = 15$
- Bucle *for*:  $\underbrace{(n - i - 1)}_{j=i+1..n-1} (\underbrace{15}_{cuerpo} + \underbrace{1}_{condición}) + \underbrace{1}_{condición} + \underbrace{2}_{ini} = 16n - 16i - 13$
- Cuerpo del bucle *while*:  $\underbrace{1}_{l8} + \underbrace{16n - 16i - 13}_{for} + \underbrace{1}_{i++} = 16n - 16i - 11$
- Bucle *while*:

$$\sum_{i=0}^{n-2} ((\underbrace{16n - 16i - 11}_{cuerpo}) + \underbrace{3}_{condición}) + \underbrace{3}_{condición} = 16 \sum_{i=0}^{n-2} n - 16 \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 8 + 3 =$$

$$= 16n(n - 1) - 16 \frac{(n - 2)(n - 1)}{2} - 8(n - 1) + 3 =$$

$$= 16n^2 - 16n - 8n^2 + 24n - 16 - 8n + 8 + 3 = 8n^2 - 5$$

$$\circ \text{Total: } \underbrace{8n^2 - 5}_{while} + \underbrace{2}_{ini} = 8n^2 - 3$$

- Caso mejor: El vector está ordenado; nunca se entra en el *if* y solo se da una vuelta al *while*:

- Cuerpo del bucle *for*:  $4 + 1 = 5$
- Bucle *for*:  $(n - 1)(5 + 1) + 1 + 2 = 6n - 3$
- Cuerpo del bucle *while*:  $1 + (6n - 3) + 1 = 6n - 1$
- Bucle *while*:

$$((6n - 1) + 3) + 3 = 6n + 5$$

- Total:  $6n + 5 + 2 = 6n + 7$

## Notas bibliográficas

Se recomienda ampliar el contenido de estas notas estudiando el Capítulo 1 de (Peña, 2005) en el cual se han basado. También la Sección 1.4 de (Rodríguez Artalejo et al., 2011). Finalmente, el Capítulo 3 de (Martí Oliet et al., 2012) contiene también material adicional y numerosos ejercicios resueltos.

## Ejercicios

- Si tenemos dos algoritmos con costes  $t_1(n) = 3n^3$  y  $t_2(n) = 600n^2$ , ¿cuál es mejor en términos asintóticos? ¿A partir de qué umbral el segundo es mejor que el primero?
- Si el coste de un algoritmo está en  $\mathcal{O}(n^2)$  y tarda 1 segundo para un tamaño  $n = 100$ , ¿de qué tamaño será el problema que puede resolver en 10 segundos?
- (Martí Oliet et al. (2012)) Indicar cuántas multiplicaciones realizan los siguientes algoritmos para calcular potencias en el caso peor. Indicar también sus ordenes asintóticos.

---

```

1 int potencial(int x, int y) {
2     int p = 1;
3     while (y > 0) {
4         p = p*x;
5         y--;
6     }
7     return p;
8 }
```

---

```

1 int potencia2(int x, int y) {
2     int w = x; int p = 1;
3     while (y > 0) {
4         if (y%2 == 1) p = p*w;
5         y = y/2;
6         w = w*w;
7     }
8     return p;
9 }
```

---

- Sea la siguiente función que implementa el algoritmo de inserción de un elemento en una lista (`tLista`) ordenada de enteros con posibles repeticiones, representada como una estructura con dos campos, `elems`, un array de enteros y `cont`, un entero. Indicar el número de instrucciones que ejecuta esta función y su orden asintótico en los casos mejor y peor. Nota: Para simplificar, se considera una instrucción a una línea de programa distinta de llave o `else`.

---

```

1 bool insertar(tLista& lista, int x) {
2     if (lista.cont == MAX) {
3         return false;
4     } else{
5         int pos = lista.cont;
6         while (pos > 0 && x < lista.elems[pos-1]) {
7             lista.elems[pos] = lista.elems[pos-1];
8             pos--;
9 }
```

---

---

```

9      }
10     lista.elems[pos] = x;
11     lista.cont++;
12     return true;
13   }
14 }
```

---

5. Sea la siguiente función que implementa el algoritmo de inserción de un elemento en una lista ordenada de enteros sin repeticiones. Indicar el número de instrucciones que ejecuta (ver Ej. 4) en los casos mejor y peor. Indicar también su orden asintótico en los casos mejor y peor en los siguientes dos supuestos: a) buscar es una búsqueda lineal; b) buscar es una búsqueda binaria.

---

```

1 bool insertar(tLista& lista, int x) {
2   if (lista.cont == MAX)
3     return false;
4   else {
5     int pos = 0;
6     if (!buscar(lista, x, pos)) { // pos indica la posición en la
7       for (int i = lista.cont; i > pos; i--) // que debe colocarse x
8         lista.elems[i] = lista.elems[i-1];
9       lista.elems[pos] = x;
10      lista.cont++;
11      return true;
12    }
13    else return false;
14  }
15 }
```

---

6. Dada la siguiente función que implementa el algoritmo de ordenación por inserción directa. Indicar el número de instrucciones que ejecuta (ver Ej. 4) y su orden asintótico en los casos mejor y peor.

---

```

1 void ordenar(tLista& lista) {
2   int nuevo, pos;
3   for (int i = 1; i < lista.cont; i++) {
4     nuevo = lista.elems[i];
5     pos = 0;
6     while ((pos < i) && !(lista.elems[pos] > nuevo)) {
7       pos++;
8     }
9     // pos: índice del primer mayor; i si no lo hay
10    for (int j = i; j > pos; j--) {
11      lista.elems[j] = lista.elems[j - 1];
12    }
13    lista.elems[pos] = nuevo;
14  }
15 }
```

---

7. Dada la siguiente función que implementa el algoritmo de ordenación de la burbuja. Indicar el número de instrucciones que ejecuta (ver Ej. 4) y su orden asintótico en los casos mejor y peor.

---

```

1 void ordenar(tLista& lista) {
```

---

```

2   bool inter = true;
3   int i = 0; int tmp;
4   while ((i < lista.cont - 1) && inter) {
5       inter = false;
6       for (int j = lista.cont - 1; j > i; j--) {
7           if (lista.elems[j] < lista.elems[j - 1]) {
8               tmp = lista.elems[j];
9               lista.elems[j] = lista.elems[j - 1];
10              lista.elems[j - 1] = tmp;
11              inter = true;
12          }
13          if (inter)
14              i++;
15      }
16  }

```

---

8. (Martí Oliet et al. (2012)) Demostrar o refutar cada una de las siguientes afirmaciones:

- a)  $n^2 + n + \log n \in O(n^2)$ .
- b)  $n^2 + n + \log n \in \Omega(n)$ .
- c)  $n^2 + n + \log n \in \Theta(\log n)$ .
- d)  $21n^3 + 7n^2 + n \log n - 17n + 8 \in O(n^4)$ .
- e)  $2^n + 3^n + n^{59} \in O(n^{59})$ .
- f)  $2^n + 3^n + n^{59} \in \Omega(2^n)$ .
- g) Si  $f(n) = n^2$ , entonces  $f(n)^3 \in \Theta(f(n)^3)$ .
- h)  $\Theta(2^n) = \Theta(2^{n+2}) = \Theta(4^n)$ .
- i)  $\log_2 n \in O(\log_3 n)$ .

9. (Martí Oliet et al. (2012)) Para cada una de las siguientes funciones  $f(n)$ , obtener el menor número natural  $k$  tal que  $f(n) \in O(n^k)$ .

- a)  $f(n) = 2n^3 + n^2 \log n$ .
- b)  $f(n) = 3n^5 + (\log n)^4$ .
- c)  $f(n) = (n^4 + n^2 + 1)/(n^3 + 1)$ .
- d)  $f(n) = (n^4 + n^2 + 1)/(n^4 + 1)$ .
- e)  $f(n) = (n^4 + 5 \log n)/(n^4 + 1)$ .
- f)  $f(n) = (n^3 + 5 \log n)/(n^4 + 1)$ .

10. (Martí Oliet et al. (2012)) Para cada una de las siguientes funciones  $f(n)$ , encontrar una función  $g(n)$  del menor orden posible tal que  $f(n) \in O(g(n))$ .

- a)  $f(n) = (n \log n + n^2)(n^3 + 2)$ .
- b)  $f(n) = (2^n + n^2)(n^3 + 3^n)$ .
- c)  $f(n) = n \log(n^2 + 1) + n^2 \log n$ .
- d)  $f(n) = n^{2^n} + n^{n^2}$ .
- e)  $f(n) = (n! + 2^n)(n^3 + \log(n^2 + 1))$ .

11. (Martí Oliet et al. (2012)) Comparar con respecto a  $O$  y  $\Omega$  los siguientes pares de funciones:

- a)  $n^2 + 3n + 7, n^2 + 10.$
- b)  $n^2 \log n, n^3.$
- c)  $n^4 + \log(3n^8 + 7), (n^2 + 17n + 3)^2.$
- d)  $(n^3 + n^2 + n + 1)^4, (n^4 + n^3 + n^2 + n + 1)^3.$
- e)  $\log(n^2 + 1), \log n.$
- f)  $2^{n+3}, 2^{n+7}.$
- g)  $2^{2^n}, 2^{n^2}.$



---

## Capítulo 2

# El esquema “Divide y vencerás”<sup>1</sup>

---

*Divide et impera*

Julio César (100 a.C.-44 a.C)

**RESUMEN:** En este tema se presenta el esquema algorítmico *Divide y vencerás*, que es un caso particular del diseño recursivo, y se ilustra con ejemplos significativos en los que la estrategia reporta beneficios claros. Se espera del alumno que incorpore este método a sus estrategias de resolución de problemas.

- ★ En este capítulo iniciamos la presentación de un conjunto de *esquemas algorítmicos* que pueden emplearse como estrategias de resolución de problemas. Un esquema puede verse como un algoritmo *genérico* que puede resolver distintos problemas. Si se concretan los tipos de datos y las operaciones del esquema genérico con los tipos y operaciones específicos de un problema concreto, tendremos un algoritmo para resolver dicho problema.
- ★ Además de *divide y vencerás*, este curso veremos el esquema de *vuelta atrás*. En cursos posteriores aparecerán otros esquemas con nombres propios tales como el *método voraz*, el de *programación dinámica* y el de *ramificación y poda*. Cada uno de ellos resuelve una familia de problemas de características parecidas.
- ★ Los esquemas o métodos algorítmicos deben verse como un conjunto de algoritmos *prefabricados* que el diseñador puede ensayar ante un problema nuevo. No hay garantía de éxito pero, si se alcanza la solución, el esfuerzo invertido habrá sido menor que si el diseño se hubiese abordado desde cero.

Antes de ver el esquema *divide y vencerás* propiamente dicho recordaremos los fundamentos de los algoritmos recursivos, veremos una serie de ejemplos paradigmáticos de algoritmos recursivos y estudiaremos cómo se calcula su complejidad asintótica. A continuación se presentará el esquema algorítmico *divide y vencerás*. Veremos cómo algunos de los algoritmos recursivos vistos encajan perfectamente en este esquema y presentaremos más ejemplos de soluciones a problemas clásicos mediante este esquema.

---

<sup>1</sup>Ricardo Peña es el autor principal de este tema. Modificado por Clara Segura en el curso 2013-14.

## 1. Introducción a los algoritmos recursivos

- ★ La recursión aparece de forma natural en programación, tanto en la definición de tipos de datos (como veremos en temas posteriores) como en el diseño de algoritmos.
- ★ Hay lenguajes (funcionales y lógicos) en los que no hay iteración (no hay bucles), sólo hay recursión. En C++ tenemos la opción de elegir entre iteración y recursión.
- ★ Optamos por una solución recursiva cuando sabemos cómo resolver de manera directa un problema para un cierto conjunto de datos y para el resto de los datos somos capaces de resolverlo utilizando la solución al mismo problema con unos datos “más simples”.
- ★ Cualquier solución recursiva se basa en un análisis (clasificación) de los datos,  $\vec{x}$ , para distinguir los casos de solución directa y los casos de solución recursiva:
  - caso(s) directo(s):  $\vec{x}$  es tal que el resultado  $\vec{y}$  puede calcularse directamente de forma sencilla.
  - caso(s) recursivo(s): sabemos cómo calcular a partir de  $\vec{x}$  otros datos más pequeños  $\vec{x}'$ , y sabemos además cómo calcular el resultado  $\vec{y}$  para  $\vec{x}$  suponiendo conocido el resultado  $\vec{y}'$  para  $\vec{x}'$ .
- ★ Para implementar soluciones recursivas en un lenguaje de programación tenemos que utilizar una acción que se invoque a sí misma con datos cada vez “más simples”: funciones o procedimientos.
- ★ Intuitivamente, el subprograma se invoca a sí mismo con datos cada vez más simples hasta que son tan simples que la solución es inmediata. Posteriormente, la solución se puede ir elaborando hasta obtener la solución para los datos iniciales.
- ★ Tres elementos básicos en la recursión:
  - Autoinvocación: el proceso se llama a sí mismo.
  - Caso directo: el proceso de autoinvocación eventualmente alcanza una solución directa (sin invocarse a sí mismo) al problema.
  - Combinación: los resultados de la llamada precedente son utilizados para obtener una solución, posiblemente combinados con otros datos.
- ★ Para entender la recursividad, a veces resulta útil considerar cómo se ejecutan las acciones recursivas en una computadora, como si se crearan múltiples copias del mismo código, operando sobre datos diferentes (en realidad sólo se copian las variables locales y los parámetros por valor). El ejemplo clásico del factorial:

---

```
int factorial(int n) {
    int r;
    if (n == 0) r = 1;
    else // n > 0
        r = n * factorial(n-1);
    return r;
}
```

---

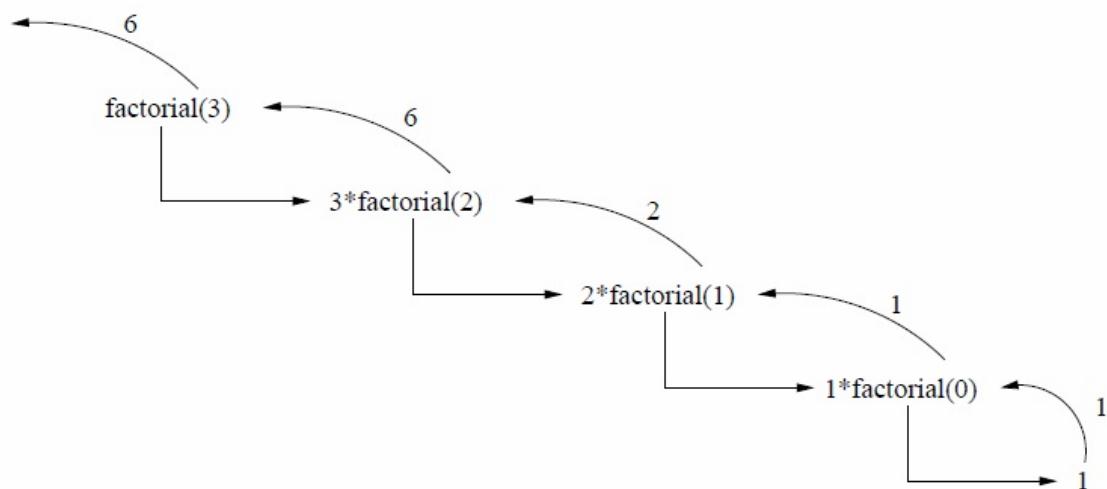


Figura 1: Ejecución de *factorial(3)*

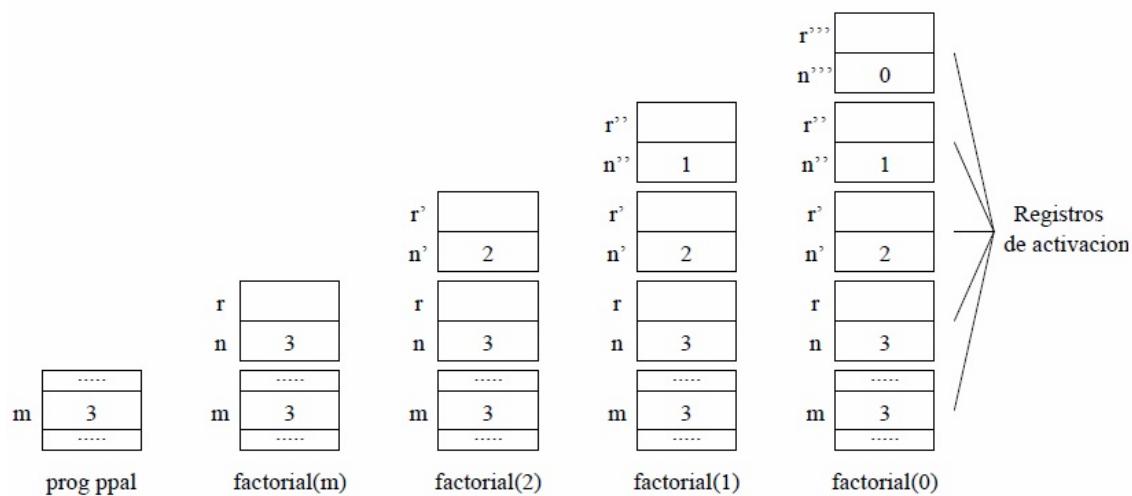


Figura 2: Llamadas recursivas de *factorial(3)*

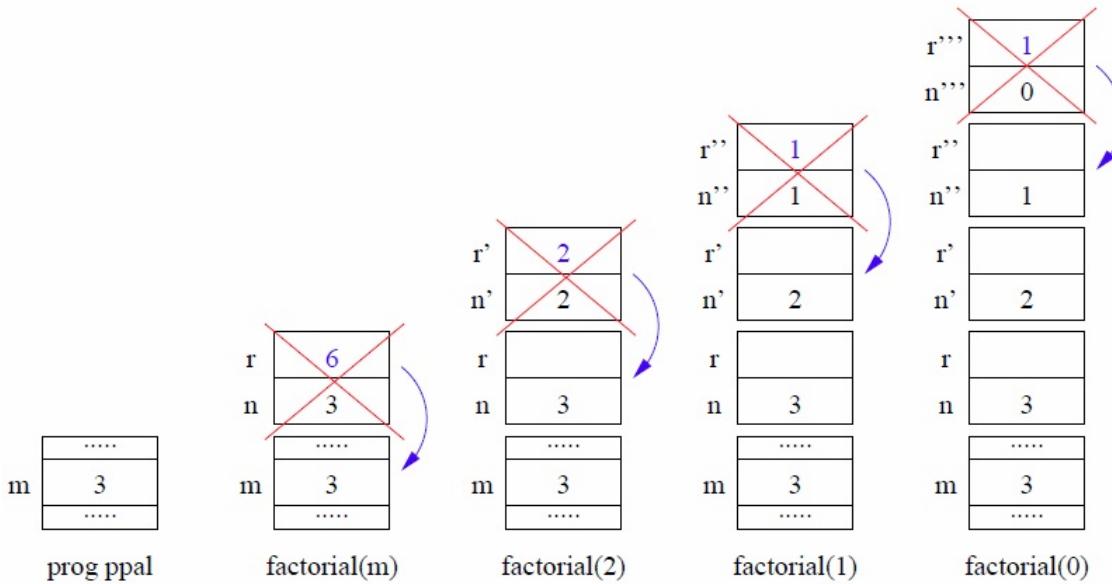
¿Cómo se ejecuta la llamada *factorial(3)*?

¿Y qué ocurre en memoria?

- ★ ¿La recursión es importante?
    - Un método muy potente de diseño y razonamiento formal.
    - Tiene una relación natural con la inducción y, por ello, facilita conceptualmente la resolución de problemas y el diseño de algoritmos.

## 1.1. Recursión simple

- ★ Una acción recursiva tiene recursión simple (o lineal) si cada caso recursivo realiza exactamente una llamada recursiva. Puede describirse mediante el esquema general:

Figura 3: Vuelta de las llamadas recursivas de *factorial(3)*


---

```

void nombreProc ( $\tau_1 x_1 , \dots , \tau_n x_n , \delta_1 \& y_1 , \dots , \delta_m \& y_m$ ) {
    // declaración de constantes

     $\tau_1 x'_1 ; \dots ; \tau_n x'_n$ ; //  $\vec{x}'$ , parámetros de la llamada recursiva
     $\delta_1 y'_1 ; \dots ; \delta_m y'_m$ ; //  $\vec{y}'$ , resultados de la llamada recursiva

    if ( $d(\vec{x})$ ) // caso directo
         $\vec{y} = g(\vec{x})$ ; // solución al caso directo
    else if ( $r(\vec{x})$ ) // caso no directo
         $\vec{x}' = s(\vec{x})$ ; // función sucesor: descomposición de datos
        nombreProc( $\vec{x}'$ ,  $\vec{y}'$ ); // llamada recursiva
         $\vec{y} = c(\vec{x}, \vec{y}')$ ; // función de combinación: composición de solución
    }
}

```

---

donde:

- $\vec{x}$  representa a los parámetros de entrada  $x_1, \dots, x_n$ ,  $\vec{x}'$  a los parámetros de la llamada recursiva  $x'_1, \dots, x'_n$ ,  $\vec{y}'$  a los resultados de la llamada recursiva  $y'_1, \dots, y'_m$ , e  $\vec{y}$  a los parámetros de salida  $y_1, \dots, y_m$
- $d(\vec{x})$  es la condición que determina el caso directo
- $r(\vec{x})$  es la condición que determina el caso recursivo
- $g$  calcula el resultado en el caso directo
- $s$ , la *función sucesor*, calcula los argumentos para la siguiente llamada recursiva
- $c$ , la *función de combinación*, obtiene la combinación de los resultados de la llamada recursiva  $\vec{y}'$  junto con los datos de entrada  $\vec{x}$ , proporcionando así el resultado  $\vec{y}$ .

- ★ Veamos cómo la función factorial se ajusta a este esquema de declaración:

```
int factorial(int n) {
    int r;
    if (n == 0) r = 1;
    else // n > 0
        r = n * factorial(n-1);
    return r;
}
```

$$\begin{aligned}d(n) &= (n == 0) \\g(n) &= 1 \\r(n) &= (n > 0) \\s(n) &= n - 1 \\c(n, \text{fact}(s(n))) &= n * \text{fact}(s(n))\end{aligned}$$

- \* El esquema de recursión simple puede ampliarse considerando varios casos directos y también varias descomposiciones para el caso recursivo.  $d(\vec{x})$  y  $r(\vec{x})$  pueden desdoblarse en una alternativa con varios casos. Lo importante es que las alternativas sean **exhaustivas** y **excluyentes**, y que en cada caso sólo se ejecute una llamada recursiva.
- \* Ejemplo: multiplicación de dos naturales por el método del *campesino egipcio*.

```
int prod(int a, int b) {
    // Precondición: {(a ≥ 0) ∧ (b ≥ 0)}
    int r;

    if (b == 0) {
        r = 0;
    } else if (b == 1) {
        r = a;
    } else if (b % 2 == 0) {           // b > 1
        r = prod(2*a, b/2);
    } else {                         // b > 1 && (b % 2 == 1)
        r = prod(2*a, b/2) + a;
    }
    return r;
}
```

$$\begin{array}{ll}d_1(a, b) = (b == 0) & d_2(a, b) = (b == 1) \\g_1(a, b) = 0 & g_2(a, b) = 1 \\r_1(a, b) = ((b > 1) \&\& \text{par}(b)) & r_2(a, b) = ((b > 1) \&\& \text{impar}(b)) \\s_1(a, b) = (2 * a, b/2) & s_2(a, b) = (2 * a, b/2) \\c_1(a, b, \text{prod}(s_1(a, b))) = \text{prod}(s_1(a, b)) & c_2(a, b, \text{prod}(s_2(a, b))) = \text{prod}(s_2(a, b)) + a\end{array}$$

## 1.2. Recursión final

- \* La recursión final o de cola (*tail recursion*) es un caso particular de recursión simple donde la función de combinación se limita a transmitir el resultado de la llamada

recursiva. Se llama final porque lo último que se hace en cada pasada es la llamada recursiva.

- ★ El resultado será siempre el obtenido en uno de los casos base.
- ★ Los algoritmos recursivos finales tienen la interesante propiedad de que pueden ser traducidos de manera directa a soluciones iterativas, más eficientes.

---

```
void nombreProcItr( $\tau_1 x_1 , \dots , \tau_n x_n , \delta_1 \& y_1 , \dots , \delta_m \& y_m$ ) {
    // declaración de constantes
     $\tau_1 x'_1 ; \dots ; \tau_n x'_n ; // \vec{x}'$ 

     $\vec{x}' = \vec{x};$ 
    while ( $r(\vec{x}')$ )
         $\vec{x}' = s(\vec{x}');$ 
     $\vec{y} = g(\vec{x}');$ 
}
}
```

---

- ★ Como ejemplo de función recursiva final veamos el algoritmo de cálculo del máximo común divisor por el algoritmo de Euclides.

---

```
int mcd(int a, int b) {
    // Precondición:  $\{(a > 0) \wedge (b > 0)\}$ 
    int m;

    if (a == b) m = a;
    else if (a > b) m = mcd(a-b, b);
    else // a < b
        m = mcd(a, b-a);
    return m;
}
```

---

que se ajusta al esquema de recursión simple:

$$\begin{aligned} d(a, b) &= (a == b) & g(a, b) &= a \\ r_1(a, b) &= (a > b) & r_2(a, b) &= (a < b) \\ s_1(a, b) &= (a - b, a) & s_2(a, b) &= (a, b - a) \end{aligned}$$

y donde las funciones de combinación se limitan a devolver el resultado de la llamada recursiva

$$\begin{aligned} c_1(a, b, mcd(s_1(a, b))) &= mcd(s_1(a, b)) \\ c_2(a, b, mcd(s_2(a, b))) &= mcd(s_2(a, b)) \end{aligned}$$

Si traducimos esta versión recursiva a una iterativa del algoritmo obtenemos:

---

```
int itmcd(int a, int b) {
    int auxa = a; int auxb = b;
    while (auxa != auxb)
        if (auxa > auxb)
            auxa = auxa - auxb;
        else
```

---

---

```

    auxb = auxb-auxa;
return auxa;
}

```

---

### 1.3. Recursión múltiple

- \* Este tipo de recursión se caracteriza por que, al menos en un caso recursivo, se realizan varias llamadas recursivas. El esquema correspondiente es el siguiente:

---

```

void nombreProc( $\tau_1 x_1, \dots, \tau_n x_n, \delta_1 \& y_1, \dots, \delta_m \& y_m$ ) {
// declaración de constantes
 $\tau_1 x_{11}; \dots; \tau_n x_{1n}; \dots; \tau_1 x_{k1}; \dots; \tau_n x_{kn};$ 
 $\delta_1 y_{11}; \dots; \delta_m y_{1m}; \dots; \delta_1 y_{k1}; \dots; \delta_m y_{km};$ 

if ( $d(\vec{x})$ )
     $\vec{y} = g(\vec{x});$ 
else if ( $r(\vec{x})$ ) {
     $\vec{x}_1 = s_1(\vec{x});$ 
    nombreProc( $\vec{x}_1, \vec{y}_1;$ 
    ...
     $\vec{x}_k = s_k(\vec{x});$ 
    nombreProc( $\vec{x}_k, \vec{y}_k;$ 
     $\vec{y} = c(\vec{x}, \vec{y}_1, \dots, \vec{y}_k);$ 
}
}

```

---

donde

- $k > 1$ , indica el número de llamadas recursivas
  - $\vec{x}$  representa los parámetros de entrada  $x_1, \dots, x_n, \vec{x}_i$  a los parámetros de la  $i$ -ésima llamada recursiva  $x_{i1}, \dots, x_{in}, \vec{y}_i$  a los resultados de la  $i$ -ésima llamada recursiva  $y_{i1}, \dots, y_{im}$ , para  $i = 1, \dots, k$ , e  $\vec{y}$  a los parámetros de salida  $y_1, \dots, y_m$
  - $d(\vec{x})$  es la condición que determina el caso directo
  - $r(\vec{x})$  es la condición que determina el caso recursivo
  - $g$  calcula el resultado en el caso directo
  - $s_i$ , las *funciones sucesor*, calculan la descomposición de los datos de entrada para realizar la  $i$ -ésima llamada recursiva, para  $i = 1, \dots, k$
  - $c$ , la *función de combinación*, obtiene la combinación de los resultados  $\vec{y}_i$  de las llamadas recursivas, para  $i = 1, \dots, k$ , junto con los datos de entrada  $\vec{x}$ , proporcionando así el resultado  $\vec{y}$ .
- \* Otro ejemplo clásico, los números de Fibonacci:

---

```

int fib(int n) {
// Precondición: { $n \geq 0$ }
int r;

if (n == 0) r = 0;
else if (n == 1) r = 1;

```

---

```

else // n > 1
    r = fib(n-1) + fib(n-2);

return r;
}

```

que se ajusta al esquema de recursión múltiple ( $k = 2$ )

$$\begin{aligned}
 d_1(n) &= (n == 0) & d_2(n) &= (n == 1) \\
 g(0) &== 0 & g(1) &== 1 \\
 r(n) &= (n > 1) \\
 s_1(n) &= n - 1 & s_2(n) &= n - 2 \\
 c(n, fib(s_1(n)), fib(s_2(n))) &= fib(s_1(n)) + fib(s_2(n))
 \end{aligned}$$

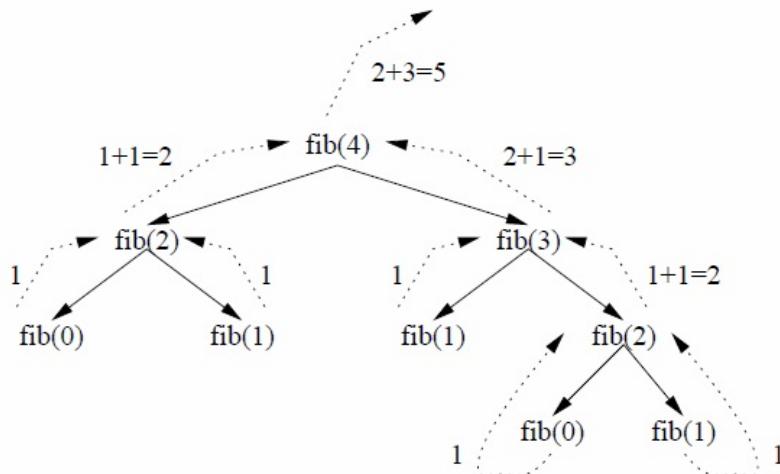


Figura 4: Ejecución de  $fib(4)$

- ★ En la recursión múltiple, el número de llamadas puede crecer muy rápidamente, como se puede ver en el cómputo de  $fib(4)$  que se muestra en la Figura 4. Nótese que algunos valores se computan más de una vez (p.e.  $fib(2)$  se evalúa 2 veces).

#### 1.4. Resumen de los distintos tipos de recursión

- ★ Para terminar con esta introducción a los algoritmos recursivos, recapitulamos los distintos tipos de funciones recursivas que hemos presentado:
  - Simple. Una llamada recursiva en cada caso recursivo:
    - No final. Requiere combinación de resultados
    - Final. No requiere combinación de resultados
  - Múltiple. Más de una llamada recursiva en algún caso recursivo.

#### 1.5. Implementación recursiva de la búsqueda binaria

- ★ Partimos de un vector ordenado, donde puede haber elementos repetidos, y un valor  $x$  que pretendemos encontrar en el vector. Buscamos la aparición más a la derecha

del valor  $x$ , o, si no se encuentra en el vector, buscamos la posición anterior a donde se debería encontrar por si queremos insertarlo. Es decir, estamos buscando el punto del vector donde las componentes pasan de ser  $\leq x$  a ser  $> x$ .

- \* La idea es que en cada pasada por el bucle se reduce a la mitad el tamaño del subvector donde puede estar el elemento buscado.

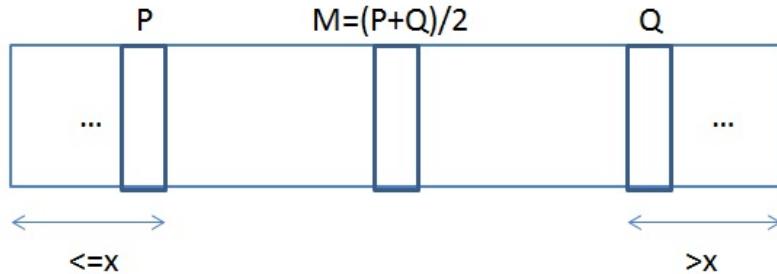


Figura 5: Cálculo del punto medio

Si  $v[m] \leq x$  entonces debemos buscar a la derecha de m

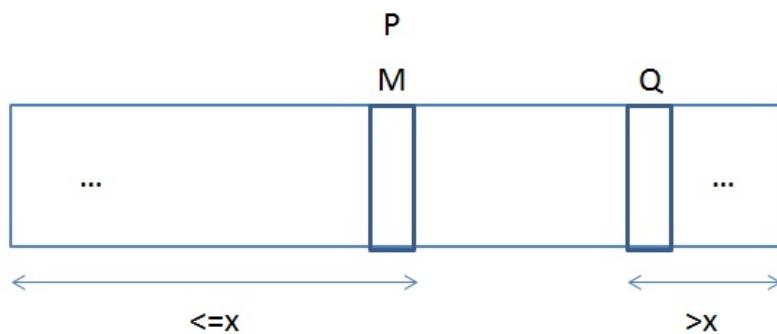


Figura 6: Búsqueda en la mitad derecha

y si  $v[m] > x$  entonces debemos buscar a la izquierda de m

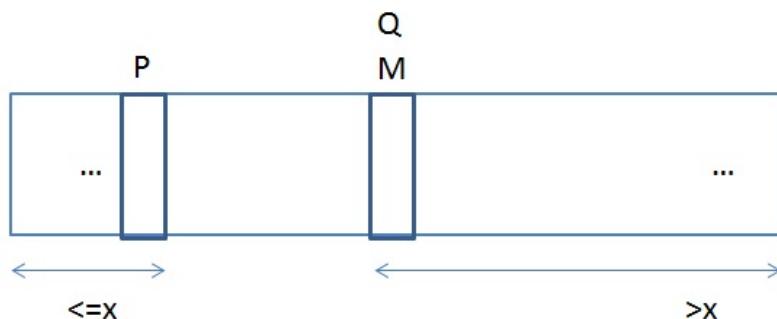


Figura 7: Búsqueda en la mitad izquierda

- \* Como el tamaño de los datos se reduce a la mitad en cada pasada tenemos claramente una complejidad logarítmica en el caso peor (en realidad en todos los casos, pues aunque encontremos  $x$  en el vector hemos de seguir buscando ya que puede que no sea la aparición más a la derecha).
- \* Hay que ser cuidadoso con los índices, sobre todo si:
  - $x$  no está en el vector, o si, en particular,
  - $x$  es mayor o menor que todos los elementos del vector; además, es necesario pensar con cuidado cuál es el caso base.

```

using TElem = int;

// Precondición: {0 ≤ n ∧ ord(v, n)}
int buscaBin(TElem v[], int n, TElem x) {
    int pos;

    // cuerpo de la función

    return pos;
}
// Postcondición:
// {(\exists u : 0 ≤ u < n : v[u] ≤ x ∧ pos = máx k : (0 ≤ k ≤ n - 1) ∧ v[k] ≤ x : k)
// ∨ (\forall z : 0 ≤ z < n : x < v[z] ∧ pos = -1)}
```

Comentarios:

- Utilizamos el tipo TElem para resaltar la idea de que la búsqueda binaria es aplicable sobre cualquier tipo que tenga definido un orden, es decir, los operadores  $=$  y  $\leq$ .
- Si  $x$  no está en  $v$  devolvemos la posición anterior al lugar donde debería estar. En particular, si  $x$  es menor que todos los elementos de  $v$ , el lugar a insertarlo será la posición 0 y, por lo tanto, devolvemos -1.
- \* El planteamiento recursivo parece claro: para buscar  $x$  en un vector de  $n$  elementos tenemos que comparar  $x$  con el elemento central y
  - si  $x$  es mayor o igual que el elemento central, seguimos buscando recursivamente en la mitad derecha,
  - si  $x$  es menor que el elemento central, seguimos buscando recursivamente en la mitad izquierda.
- \* Utilizaremos una función -o un procedimiento- auxiliar que nos permita implementar el planteamiento recursivo.

En el caso de la búsqueda binaria, se trata de una función que en lugar de recibir el número de elementos del vector, reciba dos índices,  $a$  y  $b$ , que señalen dónde empieza y dónde acaba el fragmento de vector a considerar (ambos incluidos en este caso).

```
int buscaBin(TElem v[], TElem x, int a, int b)
```

De esta forma, la función que realmente nos interesa se obtiene como

$$\text{buscaBin}(v, n, x) = \text{buscaBin}(v, x, 0, n - 1)$$

- 
- \* La función recursiva es, por tanto, la función auxiliar:

---

```
int buscaBin(TElem v[], TElem x, int a, int b) {
    // cuerpo de la función
}
```

---

y para buscar un elemento en el vector debemos llamar a `buscaBin(v, x, 0, n-1)`, siendo  $n$  la longitud del vector  $v$ .

- \* Aunque el planteamiento recursivo está claro: dados  $a$  y  $b$ , obtenemos el punto medio  $m$  y
  - si  $v[m] \leq x$  seguimos buscando en  $m+1..b$
  - si  $v[m] > x$  seguimos buscando en  $a..m-1$ ,

Es necesario ser cuidadoso con los índices. La idea consiste en garantizar que en todo momento se cumple que:

- todos los elementos a la izquierda de  $a$  -sin incluir  $v[a]$ - son menores o iguales que  $x$ , y
- todos los elementos a la derecha de  $b$  -sin incluir  $v[b]$ - son estrictamente mayores que  $x$ .

Una primera idea puede ser considerar como caso base  $a = b$ . Si lo hiciésemos así, la solución en el caso base quedaría:

```
if (a == b)
    if (v[a] == x) p = a;
    else if (v[a] < x) p = a;    // x no está en v
    else p = a-1; // x no está en v y v[a] > x
```

Sin embargo, también es necesario considerar el caso base  $a = b+1$  pues puede ocurrir que en ninguna llamada recursiva se cumpla  $a = b$ . Por ejemplo, en un situación como esta

$$x = 8 \quad a = 0 \quad b = 1 \quad v[0] = 10 \quad v[1] = 15$$

el punto medio  $m = (a+b)/2$  es 0, para el cual se cumple  $v[m] > x$  y por lo tanto la siguiente llamada recursiva se hace con

$$a = 0 \quad b = -1$$

que es un caso base donde debemos devolver -1 y donde para alcanzarlo no hemos pasado por  $a = b$ .

Como veremos a continuación, el caso  $a = b$  se puede incluir dentro del caso recursivo si consideramos como caso base el que cumple  $a = b+1$ , que además tiene una solución más sencilla y que siempre se alcanza.

- \* La solución sería la siguiente:

---

```
int buscaBin(TElem v[], TElem x, int a, int b) {
    int p, m;

    if (a == b+1)
```

---

```

    p = a - 1;
else { // a <= b
    m = (a+b)/2;
    if (v[m] <= x)
        p = buscaBin(v, x, m+1, b);
    else
        p = buscaBin(v, x, a, m-1);
}
return p;
}

int buscaBin(TElem v[], int n, TElem x) {
    return buscaBin(v, x, 0, n-1);
}

```

---

Nótese que es necesario escribir primero la función auxiliar para que sea visible desde la otra función.

## 1.6. Algoritmos avanzados de ordenación

- \* La ordenación rápida (quicksort) y la ordenación por mezcla (mergesort) son dos algoritmos de ordenación de complejidad cuasilineal,  $O(n\log n)$ .
- \* La idea recursiva es similar en los dos algoritmos: para ordenar un vector se procesan por separado la mitad izquierda y la mitad derecha.
  - En la ordenación rápida, se colocan los elementos pequeños a la izquierda y los grandes a la derecha, y luego se sigue con cada mitad por separado.
  - En la ordenación por mezcla, se ordena la mitad de la izquierda y la mitad de la derecha por separado y luego se mezclan los resultados.

### 1.6.1. Ordenación rápida (*quicksort*)

- \* Como en el caso de la búsqueda binaria, la necesidad de encontrar un planteamiento recursivo nos lleva a definir un procedimiento auxiliar con los parámetros  $a$  y  $b$ , para así poder indicar el subvector del que nos ocupamos en cada llamada recursiva.

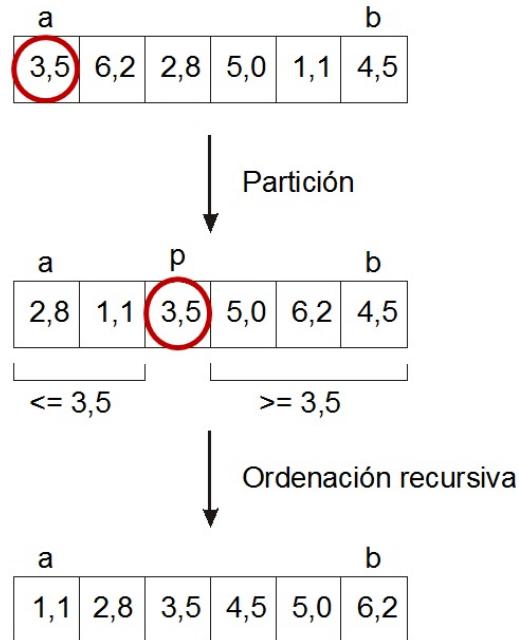
```

void quickSort(TElem v[], int n) {
    quickSort(v, 0, n-1);
}

void quickSort(TElem v[], int a, int b) {
    // Cuerpo de la función recursiva
}

```

- \* El planteamiento recursivo consiste en:
  - Elegir un pivote: un elemento cualquiera del subvector  $v[a..b]$ . Normalmente se elige  $v[a]$  como pivote.
  - Particionar el subvector  $v[a..b]$ , colocando a la izquierda los elementos menores que el pivote y a la derecha los mayores. Los elementos iguales al pivote pueden quedar indistintamente a la izquierda o a la derecha. Al final del proceso de

Figura 8: Planteamiento del algoritmo *quicksort*

partición, el pivote debe quedar en el centro, separando los menores de los mayores.

- Ordenar recursivamente los dos fragmentos que han quedado a la izquierda y a la derecha del pivote.

\* Análisis de casos

- Caso directo:  $a = b + 1$  El subvector está vacío y, por lo tanto, ordenado.
- Caso recursivo:  $a \leq b$  Se trata de un segmento no vacío y aplicamos el planteamiento recursivo:
  - considerar  $x = v[a]$  como elemento pivote
  - reordenar parcialmente el subvector  $v[a..b]$  para conseguir que  $x$  quede en la posición  $p$  que ocupará cuando  $v[a..b]$  esté ordenado.
  - ordenar recursivamente  $v[a..(p - 1)]$  y  $v[(p + 1)..b]$ .

\* El algoritmo quedaría como sigue:

```
void quickSort(TElem v[], int a, int b) {
    int p;

    if (a <= b) {
        particion(v, a, b, p);
        quickSort(v, a, p-1);
        quickSort(v, p+1, b);
    }
}

void quickSort(TElem v[], int n) {
    quickSort(v, 0, n-1);
}
```

- \* Faltaría implementar el algoritmo de partición. Dado un vector definido entre dos índices  $a$  y  $b$ , el problema consiste en separar las componentes del vector, dejando en la parte izquierda aquellos valores que sean menores o iguales que el valor de la componente  $v[a]$  y en la parte derecha aquellos valores que sean mayores o iguales que el valor de dicha componente. El valor de  $v[a]$  debe quedar separando ambas partes.

La idea es obtener un bucle que mantenga invariante la situación de la Figura 9, de forma que  $i$  y  $j$  se vayan acercando hasta cruzarse, y finalmente intercambiemos  $v[a]$  con  $v[j]$ .

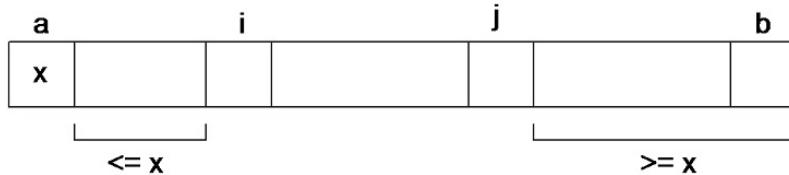


Figura 9: Diseño de *particion*

El bucle termina cuando se cruzan los índices  $i$  y  $j$ , es decir, cuando se cumple  $i = j + 1$ , y, por lo tanto, la condición de repetición es

$$i \leq j$$

A la salida del bucle, el vector estará particionado salvo por el pivote  $v[a]$ . Para terminar el proceso basta con intercambiar los elementos de las posiciones  $a$  y  $j$ , quedando la partición en la posición  $j$ .

---

```
p = j;
aux = v[a];
v[a] = v[p];
v[p] = aux;
```

---

El objetivo del bucle es conseguir que  $i$  y  $j$  se vayan acercando, y además se mantenga el invariante en cada iteración. Para ello, se hace un análisis de casos comparando las componentes  $v[i]$  y  $v[j]$  con  $v[a]$

- $v[i] \leq v[a] \rightarrow$  incrementamos  $i$
- $v[j] \geq v[a] \rightarrow$  decrementamos  $j$
- $v[i] > v[a] \wedge v[j] < v[a] \rightarrow$  intercambiamos  $v[i]$  con  $v[j]$ , incrementamos  $i$  y decrementamos  $j$

De esta forma el avance del bucle queda de la siguiente forma:

---

```
if (v[i] <= v[a] ) ++i;
else if (v[j] >= v[a] ) --j;
else { // (v[i] > v[a]) && (v[j] < v[a])
    aux = v[i];
    v[i] = v[j];
    v[j] = aux;
    ++i;
    --j;
}
```

---

Nótese que las dos primeras condiciones no son excluyentes entre sí pero sí con la tercera y, por lo tanto, la distinción de casos se puede optimizar teniendo en cuenta esta circunstancia.

- \* Con todo esto el algoritmo queda:

---

```
void particion(TElem v[], int a, int b, int & p) {

    int i, j;
    TElem aux;

    i = a+1;
    j = b;

    while (i <= j) {
        if ((v[i] > v[a]) && (v[j] < v[a])) {
            aux = v[i]; v[i] = v[j]; v[j] = aux;
            ++i; --j;
        } else {
            if (v[i] <= v[a]) ++i;
            if (v[j] >= v[a]) --j;
        }
    }

    p = j;
    aux = v[a]; v[a] = v[p]; v[p] = aux;
}
```

---

### 1.6.2. Ordenación por mezcla (*mergesort*)

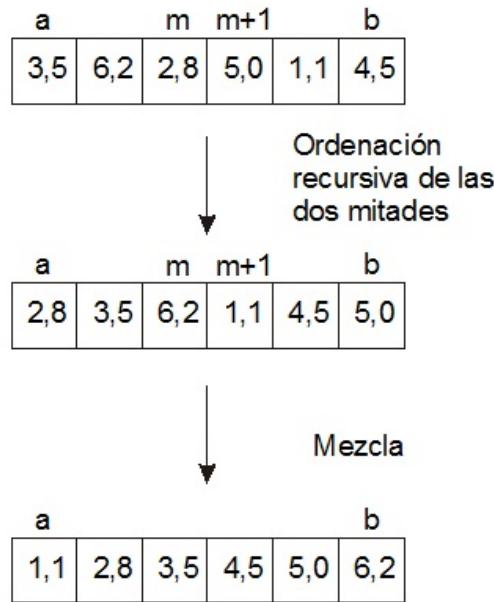


Figura 10: Planteamiento del algoritmo *mergesort*

- ★ En este caso, el planteamiento recursivo sería el siguiente. Para ordenar el subvector  $v[a..b]$ 
  - Obtenemos el punto medio  $m$  entre  $a$  y  $b$ , y ordenamos recursivamente los subvectores  $v[a..m]$  y  $v[(m + 1)..b]$ .
  - Mezclamos ordenadamente los subvectores  $v[a..m]$  y  $v[(m + 1)..b]$  ya ordenados.
- ★ Análisis de casos
  - Caso directo:  $a \geq b$   
El subvector está vacío o tiene longitud 1 y, por lo tanto, está ordenado.
  - Caso recursivo:  $a < b$   
Tenemos un subvector de longitud mayor o igual que 2, y aplicamos el planteamiento recursivo:
    - Dividir  $v[a..b]$  en dos mitades. Al ser la longitud  $\geq 2$  es posible hacer la división de forma que cada una de las mitades tendrá una longitud estrechamente menor que el segmento original (por eso hemos considerado como caso directo el subvector de longitud 1).
    - Tomando  $m = (a + b)/2$  ordenamos recursivamente  $v[a..m]$  y  $v[(m + 1)..b]$ .
    - Usamos un procedimiento auxiliar para mezclar las dos mitades, quedando ordenado todo  $v[a..b]$ .
- ★ El algoritmo quedaría como sigue:

---

```

void mergeSort(TElem v[], int a, int b) {
    int m;

    if (a < b) {
        m = (a+b)/2;
        mergeSort(v, a, m);
        mergeSort(v, m+1, b);
        mezcla(v, a, m, b);
    }
}

void mergeSort(TElem v[], int n) {
    mergeSort(v, 0, n-1);
}

```

---

- \* Faltaría implementar el algoritmo de mezcla. Dado un vector definido entre dos índices  $a$  y  $b$  y una posición intermedia entre ambos valores  $a \leq m \leq b$  tal que se cumple que los subvectores  $v[a..m]$  y  $v[m+1..b]$  están ordenados, se pide ordenar los valores de forma que al finalizar el proceso el vector  $v[a..b]$  este ordenado.

Para conseguir una solución eficiente,  $O(n)$ , utilizaremos un vector auxiliar donde iremos realizando la mezcla, para luego copiar el resultado al vector original.

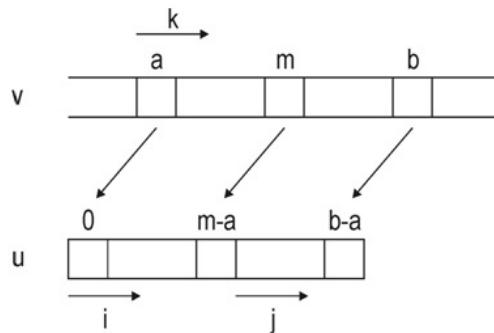


Figura 11: Diseño de *mezcla*

La idea del algoritmo es colocarse al principio de cada subvector e ir tomando, de uno u otro, el menor elemento, y así ir avanzando. Uno de los subvectores se acabará primero y habrá entonces que copiar el resto del otro subvector. En el array auxiliar tendremos los índices desplazados pues mientras el subvector a mezclar es  $v[a..b]$ , en el array auxiliar tendremos los elementos almacenados en  $v[0..b-a]$ , y habrá que ser cuidadoso con los índices que recorren ambos arrays.

Con todo esto, el algoritmo de mezcla queda:

---

```

void mezcla(TElem v[], int a, int m, int b) {

    TElem* u = new TElem[b-a+1];
    int i, j, k;

    for (k = a; k <= b; k++)
        u[k-a] = v[k];

```

---

```
i = 0;
j = m-a+1;
k = a;

while ((i <= m-a) && (j <= b-a)) {
    if (u[i] <= u[j]){
        v[k] = u[i];
        ++i;
    } else {
        v[k] = u[j];
        ++j;
    }
    ++k;
}

while (i <= m-a) {
    v[k] = u[i];
    ++i;
    ++k;
}

while (j <= b-a) {
    v[k] = u[j];
    ++j;
    ++k;
}

delete[] u;
}
```

---

## 2. Análisis de la complejidad de algoritmos recursivos

### 2.1. Ecuaciones de recurrencias

- ★ La recursión no introduce nuevas instrucciones en el lenguaje. Sin embargo, cuando intentamos analizar la complejidad de una función o un procedimiento recursivo nos encontramos con que debemos conocer la complejidad de las llamadas recursivas.

La *definición natural* de la función de complejidad de un algoritmo recursivo también es recursiva, y viene dada por una o más *ecuaciones de recurrencia*.

#### 2.1.1. Ejemplos

- ★ Cálculo del factorial.

Tamaño de los datos: n

Caso directo, n = 0 : T(n) = 2

Caso recursivo:

- 1 de evaluar la condición +
- 1 de evaluar la descomposición  $n - 1$  +
- 1 del producto  $n * \text{fact}(n - 1)$  +
- 1 de la asignación de  $n * \text{fact}(n - 1)$  +
- $T(n - 1)$  de la llamada recursiva.

$$\text{Ecuaciones de recurrencia: } T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 4 + T(n - 1) & \text{si } n > 0 \end{cases}$$

- ★ Multiplicación por el método del campesino egipcio.

Tamaño de los datos: n = b

Caso directo, n = 0, 1: T(n) = 3

En ambos casos recursivos:

- 4 de evaluar todas las condiciones en el caso peor +
- 1 de la asignación +
- 2 de evaluar la descomposición  $2 * a$  y  $b/2$  +
- 1 de la suma  $\text{prod}(2 * a, b/2) + a$  en una de las ramas +
- $T(n/2)$  de la llamada recursiva.

$$\text{Ecuaciones de recurrencia: } T(n) = \begin{cases} 3 & \text{si } n = 0, 1 \\ 8 + T(n/2) & \text{si } n > 1 \end{cases}$$

- ★ Para calcular el orden de complejidad no nos interesa el valor exacto de las constantes, ni nos preocupa que sean distintas (en los casos directos, o cuando se suma algo constante en los casos recursivos): ¡estudio asintótico!

#### 2.1.2. Ejemplos

- ★ Números de Fibonacci.

Tamaño de los datos: n

$$\text{Ecuaciones de recurrencia: } T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ T(n - 1) + T(n - 2) + c, & \text{si } n > 1 \end{cases}$$

- \* Ordenación rápida (quicksort)

Tamaño de los datos:  $n = num$

En el caso directo tenemos complejidad constante  $c_0$ .

En el caso recursivo:

- El coste de la partición:  $c * n +$
- El coste de las dos llamadas recursivas. El problema es que la disminución en el tamaño de los datos depende de los datos y de la elección del pivote.
  - El caso peor se da cuando el pivote no separa nada (es el máximo o el mínimo del subvector):  $c_0 + T(n - 1)$
  - El caso mejor se da cuando el pivote divide por la mitad:  $2 * T(n/2)$

Ecuaciones de recurrencia en el caso peor:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ T(n - 1) + c * n + c_0 & \text{si } n \geq 1 \end{cases}$$

Ecuaciones de recurrencia en el caso mejor:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ 2 * T(n/2) + c * n & \text{si } n \geq 1 \end{cases}$$

Se puede demostrar que en promedio se comporta como en el caso mejor.

Cambiando la política de elección del pivote se puede evitar que el caso peor sea un vector ordenado.

## 2.2. Despliegue de recurrencias

- \* Hasta ahora, lo único que hemos logrado es expresar la función de complejidad mediante ecuaciones recursivas. Pero es necesario encontrar una *fórmula explícita* que nos permita obtener el orden de complejidad buscado.
- \* El objetivo de este método es conseguir una fórmula explícita de la función de complejidad, a partir de las ecuaciones de recurrencias. El proceso se compone de tres pasos:

1. **Despliegue.** Sustituimos las apariciones de  $T$  en la recurrencia tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de llamadas recursivas  $k$ .
2. **Postulado.** A partir de la fórmula paramétrica resultado del paso anterior obtenemos una fórmula explícita. Para ello, se obtiene el valor de  $k$  que nos permite alcanzar un caso directo y, en la fórmula paramétrica, se sustituye  $k$  por ese valor y la referencia recursiva  $T$  por la complejidad del caso directo.
3. **Demostración.** La fórmula explícita así obtenida sólo es correcta si la recurrencia para el caso recursivo también es válida para el caso directo. Podemos comprobarlo demostrando por inducción que la fórmula obtenida cumple las ecuaciones de recurrencia.

### 2.2.1. Ejemplos

\* Factorial

- Ecuaciones

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 4 + T(n - 1) & \text{si } n > 0 \end{cases}$$

- Despliegue

$$\begin{aligned} T(n) &= 4 + T(n - 1) \\ &= 4 + 4 + T(n - 2) \\ &= 4 + 4 + 4 + T(n - 3) \\ &\dots \\ &= 4 * k + T(n - k) \end{aligned}$$

- Postulado

El caso directo se tiene para  $n = 0$

$$n - k = 0 \Leftrightarrow k = n$$

$$T(n) = 4n + T(n - n) = 4n + T(0) = 4n + 2$$

Por lo tanto  $T(n) \in O(n)$

\* Multiplicación por el método del campesino egipcio

- Ecuaciones

$$T(n) = \begin{cases} 3 & \text{si } n = 0, 1 \\ 8 + T(n/2) & \text{si } n > 1 \end{cases}$$

- Despliegue

$$\begin{aligned} T(n) &= 8 + T(n/2) \\ &= 8 + (8 + T(n/2/2)) \\ &= 8 + 8 + 8 + T(n/2/2/2) \\ &\dots \\ &= 8 * k + T(n/2^k) \end{aligned}$$

- Postulado

Las llamadas recursivas terminan cuando se alcanza 1

$$n/2^k = 1 \Leftrightarrow k = \log n$$

$$T(n) = 8 \log n + T(n/2^{\log n}) = 8 \log n + T(1) = 8 \log n + 3$$

Por lo tanto  $T(n) \in O(\log n)$

Si  $k$  representa el número de llamadas recursivas ¿qué ocurre cuando  $k = \log n$  no tiene solución entera?

La complejidad  $T(n)$  del algoritmo es una función monótona no decreciente, y, por lo tanto, nos basta con estudiar su comportamiento sólo en algunos puntos: los valores de  $n$  que son una potencia de 2. Esta simplificación no causa problemas en el cálculo asintótico.

\* Números de Fibonacci

■ Ecuaciones

$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ T(n-1) + T(n-2) + c, & \text{si } n > 1 \end{cases}$$

Podemos simplificar la resolución de la recurrencia, considerando que lo que nos interesa es una cota superior:

$$T(n) \leq 2 * T(n-1) + c_1 \quad \text{si } n > 1$$

■ Despliegue

$$\begin{aligned} T(n) &\leq c_1 + 2 * T(n-1) \\ &\leq c_1 + 2 * (c_1 + 2 * T(n-2)) \\ &\leq c_1 + 2 * (c_1 + 2 * (c_1 + 2 * T(n-3))) \\ &\leq c_1 + 2 * c_1 + 2^2 * c_1 + 2^3 * T(n-3) \\ &\dots \\ &\leq c_1 * \sum_{i=0}^{k-1} 2^i + 2^k * T(n-k) \end{aligned}$$

■ Postulado

Las llamadas recursivas terminan cuando se alcanzan 0 y 1. Como buscamos una cota superior, consideramos 0.

$$n - k = 0 \Leftrightarrow k = n$$

$$\begin{aligned} T(n) &\leq c_1 * \sum_{i=0}^{n-1} 2^i + 2^n T(n-n) \\ &= c_1 * \sum_{i=0}^{n-1} 2^i + 2^n T(0) \\ &= c_1 * (2^n - 1) + c_0 * 2^n \\ &= (c_0 + c_1) * 2^n - c_1 \end{aligned}$$

donde hemos utilizado la fórmula para la suma de progresiones geométricas:

$$\sum_{i=0}^{n-1} r^i = \frac{r^n - 1}{r - 1} \quad r \neq 1$$

Por lo tanto  $T(n) \in O(2^n)$

Las funciones recursivas múltiples donde el tamaño del problema disminuye por sustracción tienen costes prohibitivos, como en este caso donde el coste es exponencial.

## 2.3. Resolución general de recurrencias

- ★ Utilizando la técnica de despliegue de recurrencias y algunos resultados sobre convergencia de series, se pueden obtener unos resultados teóricos para la obtención de fórmulas explícitas, aplicables a un gran número de ecuaciones de recurrencias.

### 2.3.1. Disminución del tamaño del problema por sustracción

- ★ Cuando: (1) la descomposición recursiva se obtiene restando una cierta cantidad constante, (2) el caso directo tiene coste constante, y (3) la preparación de las llamadas

y de combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} c_0 & \text{si } 0 \leq n < n_0 \\ a * T(n - b) + c * n^k & \text{si } n \geq n_0 \end{cases}$$

donde:

- $c_0$  es el coste en el caso directo,
- $a \geq 1$  es el número de llamadas recursivas,
- $b \geq 1$  es la disminución del tamaño de los datos, y
- $c * n^k$  es el coste de preparación de las llamadas y de combinación de los resultados.

Se puede demostrar:

$$T(n) \in \begin{cases} O(n^{k+1}) & \text{si } a = 1 \\ O(a^{n/b}) & \text{si } a > 1 \end{cases}$$

Vemos que, cuando el tamaño del problema disminuye por sustracción,

- En recursión simple ( $a = 1$ ) el coste es polinómico y viene dado por el producto del coste de cada llamada ( $c * n^k$ ) y el coste lineal de la recursión ( $n$ ).
- En recursión múltiple ( $a > 1$ ), por muy grande que sea  $b$ , el coste siempre es exponencial.

### 2.3.2. Disminución del tamaño del problema por división

- \* Cuando: (1) la descomposición recursiva se obtiene dividiendo por una cierta cantidad constante, (2) el caso directo tiene coste constante, y (3) la preparación de las llamadas y de combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} c_1 & \text{si } 0 \leq n < n_0 \\ a * T(n/b) + c * n^k & \text{si } n \geq n_0 \end{cases}$$

donde:

- $c_1$  es el coste en el caso directo,
- $a \geq 1$  es el número de llamadas recursivas,
- $b \geq 2$  es el factor de disminución del tamaño de los datos, y
- $c * n^k$  es el coste de preparación de las llamadas y de combinación de los resultados.

Se puede demostrar:

$$T(n) = \begin{cases} O(n^k) & \text{si } a < b^k \\ O(n^k * \log n) & \text{si } a = b^k \\ O(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Si  $a \leq b^k$  la complejidad depende de  $n^k$  que es el término que proviene de  $c * n^k$  en la ecuación de recurrencias, y, por lo tanto, la complejidad de un algoritmo de este tipo se puede mejorar disminuyendo la complejidad de la preparación de las llamadas y la combinación de los resultados.

Si  $a > b^k$  las mejoras en la eficiencia se pueden conseguir

- disminuyendo el número de llamadas recursivas  $a$  o aumentando el factor de disminución del tamaño de los datos  $b$ , o bien
- optimizando la preparación de las llamadas y combinación de los resultados, pues, si esto hace disminuir  $k$  suficientemente, podemos pasar a uno de los otros casos:  $a = b^k$  o incluso  $a < b^k$ .

### 2.3.3. Ejemplos

- \* Búsqueda binaria.

Tamaño de los datos:  $n = \text{num}$

Recurrencias:

$$T(n) = \begin{cases} c_1 & \text{si } n = 0 \\ T(n/2) + c & \text{si } n > 0 \end{cases}$$

Se ajusta al esquema teórico de disminución del tamaño del problema por división, con los parámetros:

$$a = 1, b = 2, k = 0$$

Estamos en el caso  $a = b^k$  y la complejidad resulta ser:

$$O(n^k \log n) = O(n^0 \log n) = O(\log n)$$

- \* Ordenación por mezcla (mergesort).

Tamaño de los datos:  $n = \text{num}$

Recurrencias:

$$T(n) = \begin{cases} c_1 & \text{si } n \leq 1 \\ 2 * T(n/2) + c * n & \text{si } n \geq 2 \end{cases}$$

donde  $c * n$  es el coste del procedimiento mezcla.

Se ajusta al esquema teórico de disminución del tamaño del problema por división, con los parámetros:

$$a = 2, b = 2, k = 1$$

Estamos en el caso  $a = b^k$  y la complejidad resulta ser:

$$O(n^k \log n) = O(n \log n)$$

Este es también el coste del algoritmo *pareado* que vimos en la sección anterior, ya que la recurrencia es la misma. En la siguiente sección vamos a ver cómo generalizar aun más esta función para hacerla más eficiente.

## 3. Introducción al esquema *divide y vencerás*

- \* El esquema *divide y vencerás* (DV) consiste en **descomponer** el problema dado en uno o varios subproblemas del mismo tipo, el tamaño de cuyos datos es **una fracción** del tamaño original. Una vez resueltos los subproblemas por medio de la aplicación recursiva del algoritmo, se **combinan** sus resultados para construir la solución del problema original. Existirán uno o más **casos base** en los que el problema no se subdivide más y se resuelve, o bien directamente si es sencillo, o bien utilizando un algoritmo distinto.

- \* Aparentemente estas son las características generales de todo diseño recursivo y de hecho el esquema DV es un caso particular del mismo. Para distinguirlo de otros diseños recursivos que no responden a DV se han de cumplir las siguientes condiciones:
  - Los subproblemas han de tener un tamaño que sea una *fracción* del tamaño original (un medio, un tercio, etc ...). No basta simplemente con que sean más pequeños.
  - Los subproblemas se generan *exclusivamente* a partir del problema original. En algunos diseños recursivos, los parámetros de una llamada pueden depender de los resultados de otra previa. En el esquema DV, no.
  - La solución del problema original se obtiene *combinando los resultados* de los subproblemas entre sí, y posiblemente con parte de los datos originales. Otras posibles combinaciones no encajan en el esquema.
  - El (los) caso(s) base no son necesariamente los casos triviales. Como veremos más adelante podría utilizarse como caso base (incluso debería utilizarse en ocasiones) un algoritmo distinto al algoritmo recursivo DV.

- \* Puesto en forma de código, el esquema DV tiene el siguiente aspecto:

---

```
template <class Problema, class Solución>
Solución divide-y-vencerás (Problema x) {
    Problema x_1, ..., x_k;
    Solución y_1, ..., y_k;

    if (base(x))
        return método-directo(x);
    else {
        (x_1, ..., x_k) = descomponer(x);
        for (i=1; i<=k; i++)
            y_i = divide-y-vencerás(x_i);
        return combinar(x, y_1, ..., y_k);
    }
}
```

---

- \* Los tipos Problema, Solución, y los métodos base, método-directo, descomponer y combinar, son específicos de cada problema resuelto por el esquema.
- \* Para saber si la aplicación del esquema DV a un problema dado resultará en una solución eficiente o no, se deberá calcular su complejidad asintótica como hemos explicado (en particular usando la plantilla para resolver recurrencias en las que el tamaño del problema disminuye *mediante división*):

$$T(n) = \begin{cases} c_1 & \text{si } 0 \leq n < n_0 \\ a * T(n/b) + c * n^k & \text{si } n \geq n_0 \end{cases}$$

- \* Recordemos que la solución de la misma era:

$$T(n) = \begin{cases} O(n^k) & \text{si } a < b^k \\ O(n^k * \log n) & \text{si } a = b^k \\ O(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

- \* Para obtener una solución eficiente, hay que conseguir a la vez:

- que el tamaño de cada subproblema sea lo más pequeño posible, es decir, **maximizar**  $b$ .
  - que el número de subproblemas generados sea lo más pequeño posible, es decir, **minimizar**  $a$ .
  - que el coste de la parte no recursiva sea lo más pequeño posible, es decir **minimizar**  $k$ .
- \* La recurrencia puede utilizarse para **anticipar** el coste que resultará de la solución DV, sin tener por qué completar todos los detalles. Si el coste sale igual o peor que el de un algoritmo ya existente, entonces no merecerá la pena aplicar DV.

#### 4. Ejemplos de aplicación del esquema con éxito

- \* Algunos de los algoritmos recursivos vistos hasta ahora encajan perfectamente en el esquema DV.
- \* La **búsqueda binaria** en un vector ordenado es un primer ejemplo. En este caso, la operación **descomponer** selecciona una de las dos mitades del vector y la operación **combinar** es vacía. Obteníamos los siguientes parámetros de coste:

$b = 2$  Tamaño mitad del subvector a investigar en cada llamada recursiva.

$a = 1$  Un subproblema a lo sumo.

$k = 0$  Coste constante de la parte no recursiva.

dando un coste total  $O(\log n)$ .

- \* La **ordenación mediante mezcla** o *mergesort* también responde al esquema: la operación **descomponer** divide el vector en dos mitades y la operación **combinar** mezcla las dos mitades ordenadas en un vector final. Los parámetros del coste son:

$b = 2$  Tamaño mitad de cada subvector.

$a = 2$  Siempre se generan dos subproblemas.

$k = 1$  Coste lineal de la parte no recursiva (la mezcla).

dando un coste total  $O(n \log n)$ .

- \* La **ordenación rápida** o *quicksort*, considerando solo el caso mejor, también responde al esquema. La operación **descomponer** elige el pivote, partitiona el vector con respecto a él y lo divide en dos mitades. La operación **combinar** en este caso es vacía. Los parámetros del coste son:

$b = 2$  Tamaño mitad de cada subvector.

$a = 2$  Siempre se generan dos subproblemas.

$k = 1$  Coste lineal de la parte no recursiva (la partición).

dando un coste total  $O(n \log n)$ .

- \* La comprobación en un vector  $v$  estrictamente ordenado de si **existe un índice  $i$  tal que  $v[i] = i$**  sigue un esquema similar al de la búsqueda binaria:

$b = 2$  Tamaño mitad del subvector a investigar en cada llamada recursiva.

$a = 1$  Un subproblema a lo sumo.

$k = 0$  Coste constante de la parte no recursiva.

dando un coste total  $O(\log n)$ .

- \* Un problema históricamente famoso es el de la solución DV a la transformada discreta de Fourier (DFT), dando lugar al algoritmo conocido como **transformada rápida de Fourier**, o FFT (J.W. Cooley y J.W. Tukey, 1965). La transformada discreta convierte un conjunto de muestras de amplitud de una señal, en el conjunto de frecuencias que resultan del análisis de Fourier de la misma. Esta transformación y su inversa (que se realiza utilizando el mismo algoritmo DFT) tienen gran interés práctico pues permiten filtrar frecuencias indeseadas (p.e. ruido) y mejorar la calidad de las señales de audio o de vídeo.

La transformada en esencia multiplica una matriz  $n \times n$  de números complejos por un vector de longitud  $n$  de coeficientes reales, y produce otro vector de la misma longitud. El algoritmo clásico realiza esta tarea del modo obvio y tiene un coste  $O(n^2)$ . La FFT descompone de un cierto modo el vector original en dos vectores de tamaño  $n/2$ , realiza la FFT de cada uno, y luego combina los resultados de tamaño  $n/2$  para producir un vector de tamaño  $n$ . Las dos partes no recursivas tienen coste lineal, dando lugar a un algoritmo FFT de coste  $O(n \log n)$ . El algoritmo se utilizó por primera vez para analizar un temblor de tierra que tuvo lugar en Alaska en 1964. El algoritmo clásico empleó 26 minutos en analizar la muestra, mientras que la FFT de Cooley y Tukey lo hizo en 6 segundos.

## 5. Problema de selección

- \* Dado un vector  $v$  de  $n$  elementos que se pueden ordenar y un entero  $1 \leq k \leq n$ , el *problema de selección* consiste en encontrar el  $k$ -ésimo menor elemento.
- \* El problema de encontrar la mediana de un vector es un caso particular de este problema en el que se busca el elemento  $\lceil n/2 \rceil$ -ésimo del vector en el caso de estar ordenado (en los vectores de C++ corresponde a la posición  $(n - 1) \div 2$ ).
- \* Una primera idea para resolver el problema consiste en ordenar el vector y tomar el elemento  $v[k]$ , lo cual tiene la complejidad del algoritmo de ordenación utilizado. Nos preguntamos si podemos hacerlo más eficientemente.
- \* Otra posibilidad es utilizar el algoritmo *partición* con algún elemento del vector:
  - Si la posición  $p$  donde se coloca el pivote es igual a  $k$ , entonces  $v[p]$  es el elemento que estamos buscando.
  - Si  $k < p$  entonces podemos pasar a buscar el  $k$ -ésimo elemento en las posiciones anteriores a  $p$ , ya que en ellas se encuentran los elementos menores o iguales que  $v[p]$  y  $v[p]$  es el  $p$ -ésimo elemento del vector.
  - Si  $k > p$  entonces podemos pasar a buscar el  $k$ -ésimo elemento en las posiciones posteriores a  $p$ , ya que en ellas se encuentran los elementos mayores o iguales que  $v[p]$  y  $v[p]$  es el  $p$ -ésimo elemento del vector.
- \* Al igual que hemos hecho en anteriores ocasiones generalizamos el problema añadiendo dos parámetros adicionales  $a$  y  $b$  tales que  $0 \leq a \leq b \leq \text{long}(v) - 1$ , que nos

indican la parte del vector que nos interesa en cada momento. La llamada inicial que deseamos es `seleccion(v, 0, long(v)-1, k)`.

- \* En esta versión del algoritmo la posición  $k$  es una posición absoluta dentro del vector. Se puede escribir una versión alternativa en la que  $k$  hace referencia a la posición relativa dentro del subvector que se está tratando.

---

```
TElem seleccion1(TElem v[], int a, int b, int k) {
//Pre: 0<=a<=b<=long(v)-1 && a<=k<=b
    int p;
    if (a == b) return v[a];
    else {
        particion(v,a,b,p);
        if (k == p) return v[p];
        else if (k < p) return seleccion1(v,a,p-1,k);
        else return seleccion1(v,p+1,b,k);
    }
}
```

---

- \* El caso peor de este algoritmo se da cuando el pivote queda siempre en un extremo del subvector correspondiente y la llamada recursiva se hace con todos los elementos menos el pivote: por ejemplo si el vector está ordenado y le pido el último elemento. En dicho caso el coste está en  $O(n^2)$  siendo  $n = b - a + 1$  el tamaño del vector. La situación es similar a la del algoritmo de ordenación rápida.
- \* Nos preguntamos qué podemos hacer para asegurarnos de que el tamaño del problema se divida aproximadamente por la mitad. Si en lugar de usar el primer elemento del vector como pivote usásemos la mediana del vector, entonces solamente tendríamos una llamada recursiva de la mitad de tamaño, lo que nos da un coste en  $O(n)$  siendo  $n$  el tamaño del vector.
- \* Pero resulta que el problema de la mediana es un caso particular del problema que estamos intentando resolver y del mismo tamaño. Por ello, nos vamos a conformar con una aproximación suficientemente buena de la mediana, conocida como *mediana de las medianas*, con la esperanza de que sea suficiente para tener un coste mejor.
- \* Para calcularla se divide el vector en trozos consecutivos de 5 elementos, y se calcula directamente la mediana para cada uno de ellos. Después, se calcula recursivamente la mediana de esas  $n \div 5$  medianas mediante el algoritmo de selección. Con este pivote se puede demostrar que el caso peor anterior ya no puede darse.
- \* Para implementar este algoritmo vamos a definir una versión mejorada de partición:
  - Recibe como argumento el pivote con respecto al cual queremos hacer la partición, con lo que es más general. Así podemos calcular primero la mediana de las medianas y dársela después a partición.
  - En lugar de devolver solamente una posición  $p$  vamos a devolver dos posiciones  $p$  y  $q$  que delimitan la parte de los elementos que son estrictamente menores que el pivote (desde  $a$  hasta  $p - 1$ ), las que son iguales al pivote (desde  $p$  hasta  $q$ ) y las que son estrictamente mayores (desde  $q + 1$  hasta  $b$ ). De esta forma, si el pivote se repite muchas veces el tamaño se reduce más.

- Este algoritmo también se puede usar en el de la ordenación rápida. Así, por ejemplo, si todos los elementos del vector son iguales, con el anterior algoritmo de partición se tiene coste  $O(n \log n)$  mientras con el nuevo, descartando la parte de los que son iguales al pivote, tenemos coste en  $O(n)$ .
- Para implementarlo usamos tres índices  $p, k, q$ : los dos primeros se mueven hacia la derecha y el tercero hacia la izquierda. El invariante nos indica que los elementos en  $v[a..p - 1]$  son estrictamente menores que el pivote, los de  $v[p..k - 1]$  son iguales al pivote y los de  $v[q + 1..b]$  son estrictamente mayores. La parte  $v[k..q]$  está sin explorar hasta que  $k = q + 1$  en cuyo caso el bucle termina y tenemos lo que queremos.
- En cada paso se compara  $v[k]$  con el pivote: si es igual, está bien colocado y se incrementa  $k$ ; si es menor se intercambia con  $v[p]$  para ponerlo junto a los menores y se incrementan los indices  $p$  y  $k$ , ya que  $v[i]$  es igual al pivote; si es mayor se intercambia con  $v[q]$  para ponerlo junto a los mayores y se decrementa la  $q$ .

---

```

void particion2(TElem v[], int a, int b, TElem pivote, int& p, int& q) {
    //PRE: 0<=a<=b<=long(v)-1
    int k;
    TElem aux;
    p = a; k = a; q = b;

    //INV: a<=p<=k<=q+1<=b+1<=long(v)
    //      los elementos desde a hasta p-1 son < pivote
    //      los elementos desde p hasta k-1 son = pivote
    //      los elementos desde q+1 hasta b son > pivote
    while (k <= q) {
        if (v[k] == pivote) k = k+1;
        else if (v[k] < pivote) {
            aux = v[p]; v[p] = v[k]; v[k] = aux;
            p = p+1; k = k+1;
        } else {
            aux = v[q]; v[q] = v[k]; v[k] = aux;
            q = q-1;
        }
    }
    //POST: los elementos desde a hasta p-1 son < pivote
    //      los elementos desde p hasta q son = pivote
    //      los elementos desde q+1 hasta b son > pivote
}

```

---

- \* Por tanto, los pasos del nuevo algoritmo, `seleccion2`, son:

1. calcular la mediana de cada grupo de 5 elementos. En total  $n \text{ div } 5$  medianas, y cada una se puede calcular en tiempo constante: ordenar los 5 elementos y quedarnos con el tercero. Para no usar espacio adicional, dichas medianas se trasladan al principio del vector.
2. calcular la mediana de las medianas,  $mm$ , con una llamada recursiva a `seleccion2` con  $n \text{ div } 5$  elementos.
3. llamar a `particion2(v, a, b, mm, p, q)`, utilizando como pivote  $mm$ .

4. hacer una distinción de casos similar a la de `seleccion1`:

---

```
if ((k >= p) && (k <= q))
    return mm;
else if (k < p)
    return seleccion2(v, a, p-1, k);
else
    return seleccion2(v, q+1, b, k);
```

---

- ★ Es necesario elegir adecuadamente los casos base, ya que si hay 12 elementos o menos, es decir  $b - a + 1 \leq 12$ , es más costoso seguir el proceso recursivo que ordenar directamente y tomar el elemento  $k$ .
- ★ Con estas ideas el algoritmo queda:

```
TElem seleccion2(TElem v[], int a, int b, int k){
//0<=a<=b<=long(v)-1 && a<=k<=b
    int l, p, q, s, pm, t;
    TElem aux,mm;

    t = b-a+1;
    if (t <= 12){ // Umbral base = 12
        ordenarInsercion(v,a,b);
        return v[k];
    } else {
        s = t/5;
        for (l = 1; l <= s; l++) {
            ordenarInsercion(v,a+5*(l-1),a+5*l-1);
            pm = a+5*(l-1)+(5/2);
            aux = v[a+l-1];
            v[a+l-1] = v[pm];
            v[pm] = aux;
        }
        mm = seleccion2(v,a,a+s-1,a+(s-1)/2);
        particion2(v,a,b,mm,p,q);
        if ((k >= p) && (k <= q)) return mm;
        else if (k<p) return seleccion2(v,a,p-1,k);
        else return seleccion2(v,q+1,b,k);
    }
}
//POST: v[k] es mayor o igual que v[0..k - 1] y
// menor o igual que v[k + 1..long(v) - 1]
```

---

donde *ordenarInsercion* es una versión del algoritmo de ordenación por inserción en la que indicamos el trozo del vector que deseamos ordenar.

- ★ La llamada inicial es `seleccion2(v, 0, long(v)-1, k)`.
- ★ Se puede demostrar, por inducción constructiva, que el tiempo requerido por *seleccion2* en el caso peor es lineal en  $n = b - a + 1$  Brassard y Bratley (1997).

## 6. Organización de un campeonato

- ★ Se tienen  $n$  participantes para un torneo de ajedrez y hay que organizar un calendario para que todos jueguen contra todos de forma que:

1. Cada participante juegue exactamente una partida con cada uno de los  $n - 1$  restantes.
2. Cada participante juegue a lo sumo una partida diaria.
3. El torneo se complete en el menor número posible de días.

En este tema veremos una solución para el caso, más sencillo, en que  $n$  es potencia de 2.

- \* Es fácil ver que el número de parejas distintas posibles es  $\frac{1}{2}n(n - 1)$ . Como  $n$  es par, cada día pueden jugar una partida los  $n$  participantes formando con ellos  $\frac{n}{2}$  parejas. Por tanto se necesita un mínimo de  $n - 1$  días para que jueguen todas las parejas.
- \* Una posible forma de representar la solución al problema es en forma de matriz de  $n$  por  $n$ , donde se busca llenar, en cada celda  $a_{ij}$ , el día en que se enfrentarán entre sí los contrincantes  $i$  y  $j$ , con  $j < i$ . Es decir, tratamos de llenar, con fechas de encuentros, el área bajo la diagonal de esta matriz; sin que en ninguna fila o columna haya días repetidos.
- \* Se ha de planificar las parejas de cada día, de tal modo que al final todos jueguen contra todos sin repetir ninguna partida, ni descansar innecesariamente.
- \* Podemos ensayar una solución DV según las siguientes ideas (ver Figura 12):
  - Si  $n$  es suficientemente grande, dividimos a los participantes en dos grupos disjuntos  $A$  y  $B$ , cada uno con la mitad de ellos.
  - Se resuelven recursivamente dos torneos más pequeños: el del conjunto  $A$  jugando sólo entre ellos, y el del conjunto  $B$  también jugando sólo entre ellos. En estos sub-torneos las condiciones son idénticas a las del torneo inicial por ser  $n$  una potencia de 2; con la salvedad de que se pueden jugar ambos en paralelo.
  - Después se planifican partidas en las que un participante pertenece a  $A$  y el otro a  $B$ . En estas partidas, que no se pueden solapar con los sub-torneos, hay que llenar todas las celdas de la matriz correspondiente.
- \* Esta última parte se puede resolver fácilmente fila por fila, rotando, en cada nueva fila, el orden de las fechas disponibles. Como hay que llenar  $\frac{n}{2} \cdot \frac{n}{2}$  celdas, el coste de esta fase está en en  $\Theta(n^2)$ .
- \* Los casos base,  $n = 2$  o  $n = 1$ , se resuelven trivialmente en tiempo constante.
- \* Esta solución nos da pues los parámetros de coste  $a = 2$ ,  $b = 2$  y  $k = 2$ , que conducen a un coste esperado de  $\Theta(n^2)$ . No puede ser menor puesto que la propia planificación consiste en llenar  $\Theta(n^2)$  celdas. Pasamos entonces a precisar los detalles.

### 6.1. Implementación

- \* Usaremos una matriz cuadrada declarada como `int a[MAX][MAX]` (donde MAX es una constante entera) para almacenar la solución, inicializada con ceros. La primera fecha disponible será el dia 1.

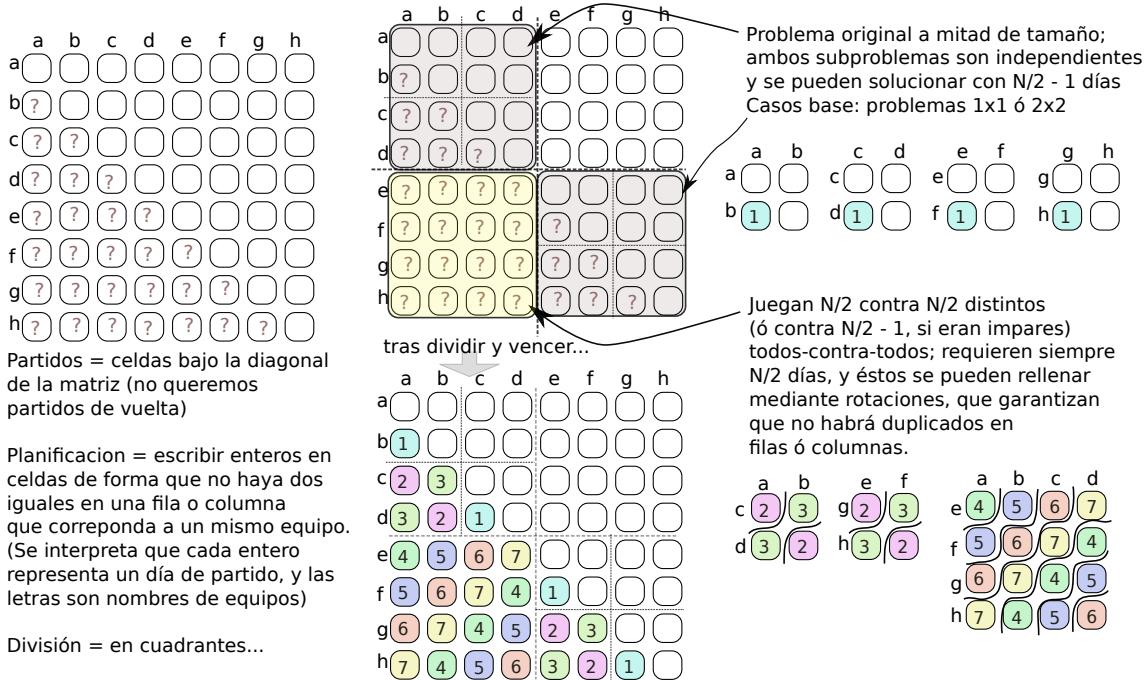


Figura 12: Solución gráfica del problema del torneo

- ★ De forma similar a ejemplos anteriores la función recursiva, llamada *rellena*, recibe dos parámetros adicionales,  $c$  y  $f$  que delimitan el trozo de la matriz que estamos rellenando, y que por tanto se inicializan respectivamente con 0 y  $num - 1$ , siendo  $num$  el número de participantes. Se asume que  $dim = f - c + 1$  es potencia de 2.
- ★ En el caso base en que solo haya un equipo ( $dim = 1$ ) no se hace nada. Si hay dos equipos ( $dim = 2$ ), juegan el dia 1.
- ★ El cuadrante inferior izquierdo representa los partidos entre los equipos de los dos grupos. El primer dia disponible es  $mitad = dim/2$  y hacen falta  $mitad$  días para que todos jueguen contra todos. Las rotaciones se consiguen con la fórmula  $mitad + (i + j) \% mitad$ .

```

void rellena(int a[MAX][MAX], int c, int f) {
    //PRE:  $\exists k : f - c + 1 = 2^k \wedge 0 \leq c \leq f < MAX \wedge \forall i, j : c \leq i, j \leq f : a[i][j] = 0$ 
    int dim = f - c + 1;
    int mitad = dim/2;
    // si dim = 1 nada, la diagonal principal no se rellena
    if (dim == 2) a[c+1][c] = 1;
    else if (dim > 2) {
        rellena(a, c, c+mitad-1);           //cuadrante superior izquierdo
        rellena(a, c+mitad, f);             //cuadrante inferior derecho
        for (int i = c+mitad; i <= f; i++) //cuadrante inferior izquierdo
            for (int j = c; j <= c+mitad-1; j++)
                a[i][j] = mitad + (i+j) % mitad;
    }

    //POST:  $\forall i, j : c \leq j < i \leq f : 1 \leq a[i][j] \leq f - c \wedge$ 
    //       $\forall i : c < i \leq f : (\forall j, k : c \leq j < k < i : a[i][j] \neq a[i][k]) \wedge$ 
    //       $\forall j : c \leq j < f : (\forall i, k : j < i < k \leq f : a[i][j] \neq a[k][j]) \wedge$ 
    //       $\forall i : c \leq i \leq f : (\forall j : c \leq j < i : (\forall k : i < k \leq f : a[i][j] \neq a[k][i]))$ 
}

```

---

}

---

## 7. El problema del par más cercano

- \* Dada una nube de  $n$  puntos en el plano,  $n \geq 2$ , se trata de encontrar el par de puntos cuya distancia euclídea es menor (si hubiera más de un par con esa distancia mínima, basta con devolver uno de ellos). El problema tiene interés práctico. Por ejemplo, en un sistema de control del tráfico aéreo, el par más cercano nos informa del mayor riesgo de colisión entre dos aviones.
- \* Dados dos puntos,  $p_1 = (x_1, y_1)$  y  $p_2 = (x_2, y_2)$ , su distancia euclídea viene dada por  $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ . El algoritmo de “fuerza bruta” calcularía la distancia entre todo posible par de puntos, y devolvería el mínimo de todas ellas. Como hay  $\frac{1}{2}n(n - 1)$  pares posibles, el coste resultante sería **cuadrático**.
- \* El enfoque DV trataría de encontrar el par más cercano a partir de los pares más cercanos de conjuntos de puntos que sean una fracción del original. Una posible estrategia es:

**Dividir** Crear dos nubes de puntos de tamaño mitad. Podríamos ordenar los puntos por la coordenada  $x$  y tomar la primera mitad como nube izquierda  $I$ , y la segunda como nube derecha  $D$ . Determinamos una linea vertical imaginaria  $l$  tal que todos los puntos de  $I$  están sobre  $l$ , o a su izquierda, y todos los de  $D$  están sobre  $l$ , o a su derecha.

**Conquistar** Resolver recursivamente los problemas  $I$  y  $D$ . Sean  $\delta_I$  y  $\delta_D$  las respectivas distancias mínimas encontradas y sea  $\delta = \min(\delta_I, \delta_D)$ .

**Combinar** El par más cercano de la nube original, o bien es el par con distancia  $\delta$ , o bien es un par compuesto por un punto de la nube  $I$  y otro punto de la nube  $D$ . En ese caso, ambos puntos se hallan a lo sumo a una distancia  $\delta$  de  $l$ . La operación *combinar* debe investigar los puntos de dicha banda vertical.

- \* Antes de seguir con los detalles, debemos investigar el coste esperado de esta estrategia. Como el algoritmo fuerza-bruta tiene coste  $\Theta(n^2)$ , trataremos de conseguir un coste  $\Theta(n \log n)$  en el caso peor. Sabemos por la experiencia de algoritmos como *mergesort* que ello exige unos parámetros  $a = 2$ ,  $b = 2$ ,  $k = 1$ , en decir tan solo podemos consumir un coste lineal en las operaciones de dividir y combinar.
- \* La ordenación de los puntos por la coordenada de  $x$  se puede realizar una sola vez al principio (es decir, fuera del algoritmo recursivo DV) con un coste  $\Theta(n \log n)$  en el caso peor, lo que es admisible para mantener nuestro coste total. Una vez ordenada la nube, la división en dos puede conseguirse con coste constante o lineal, dependiendo de si se utilizan vectores o listas como estructuras de datos de la implementación.
- \* Una vez resueltos los dos subproblemas, se pueden filtrar los puntos de  $I$  y  $D$  para conservar sólo los que estén en la banda vertical de anchura  $2\delta$  y centrada en  $l$ . El filtrado puede hacerse con coste lineal tanto en tiempo como en espacio adicional. Llamemos  $B_I$  y  $B_D$  a los puntos de dicha banda respectivamente a la izquierda y a la derecha de  $l$ .

- \* Para investigar si en la banda hay dos puntos a distancia menor que  $\delta$ , aparentemente debemos calcular la distancia de cada punto de  $B_I$  a cada punto de  $B_D$ . Es fácil construir nubes de puntos en las que todos ellos caigan en la banda tras el filtrado, de forma que en el caso peor podríamos tener  $|B_I| = |B_D| = \frac{n}{2}$ . En ese caso, el cálculo de la distancia mínima entre los puntos de la banda sería cuadrático, y el coste total del algoritmo DV también.
- \* Demostraremos que basta ordenar por la coordenada  $y$  el conjunto de puntos  $B_I \cup B_D$  y después recorrer la lista ordenada comparando cada punto **con los 7 que le siguen**. Si de este modo no se encuentra una distancia menor que  $\delta$ , concluimos que todos los puntos de la banda distan más entre sí. Este recorrido es claramente de coste lineal.
- \* Suponiendo que esta estrategia fuera correcta, todavía quedaría por resolver la ordenación por la coordenada  $y$ . Si ordenáramos  $B_I \cup B_D$  en cada llamada recursiva, gastaríamos un coste  $\Theta(n \log n)$  en cada una, lo que conduciría un coste total en  $\Theta(n \log^2 n)$ .
- \* Usando la técnica de los **resultados acumuladores** podemos exigir que cada llamada recursiva devuelva un resultado extra: la lista de sus puntos ordenada por la coordenada  $y$ . Este resultado puede propagarse hacia arriba del árbol de llamadas con un coste lineal, porque basta aplicar el algoritmo de mezcla de dos listas ordenadas. La secuencia de acciones de la operación *combinar* es entonces la siguiente:
  1. Realizar la mezcla ordenada de las dos listas de puntos devueltas por las llamadas recursivas. Esta lista se devolverá al llamante.
  2. Filtrar la lista resultante, conservando los puntos a una distancia de la línea divisoria  $l$  menor o igual que  $\delta$ . Llamemos  $B$  a la lista filtrada.
  3. Recorrer  $B$  calculando la distancia de cada punto a los 7 que le siguen, comprobando si aparece una distancia menor que  $\delta$ .
  4. Devolver los dos puntos a distancia mínima, considerando los tres cálculos rea- lizados: parte izquierda, parte derecha y lista  $B$ .

## 7.1. Corrección

- \* Consideremos un rectángulo cualquiera de anchura  $2\delta$  y altura  $\delta$  centrado en la línea divisoria  $l$  (ver Figura 13). Afirmamos que, contenidos en él, puede haber a lo sumo 8 puntos de la nube original. En la mitad izquierda puede haber a lo sumo 4, y en caso de haber 4, situados necesariamente en sus esquinas, y en la mitad derecha otros 4, también en sus esquinas. Ello es así porque, por hipótesis de inducción, los puntos de la nube izquierda están separados entre sí por una distancia de al menos  $\delta$ , e igualmente los puntos de la nube derecha entre sí. En la línea divisoria podrían coexistir hasta dos puntos de la banda izquierda con dos puntos de la banda derecha.
- \* Si colocamos dicho rectángulo con su base sobre el punto  $p_1$  de menor coordenada  $y$  de  $B$ , estamos seguros de que a los sumo los 7 siguientes puntos de  $B$  estarán en dicho rectángulo. A partir del octavo, él y todos los demás distarán más que  $\delta$  de  $p_1$ . Desplazando ahora el rectángulo de punto a punto, podemos repetir el mismo razonamiento. No es necesario investigar los puntos con menor coordenada  $y$  que el punto en curso, porque esa comprobación ya se hizo cuando se procesaron dichos puntos.

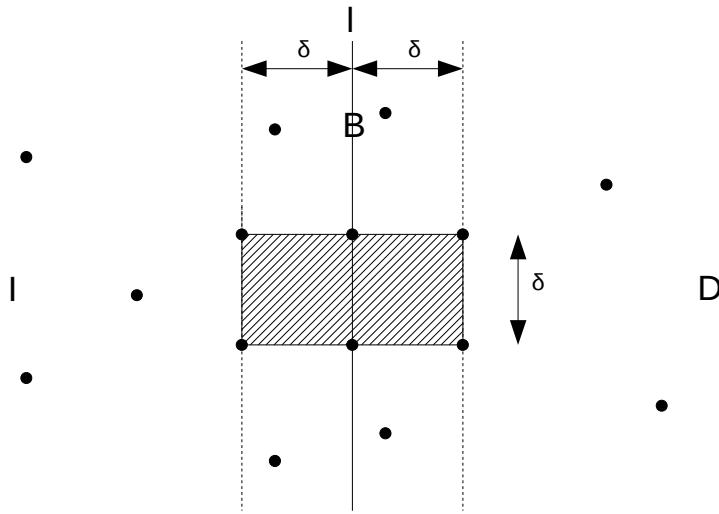


Figura 13: Razonamiento de corrección del problema del par más cercano

- \* Elegimos como caso base de la inducción  $n < 4$ . De este modo, al subdividir una nube con  $n \geq 4$  puntos, nunca generaremos problemas con un solo punto.

## 7.2. Implementación

- \* Definimos un punto como una pareja de números reales.

---

```
struct Punto
{
    double x;
    double y;
};
```

---

La entrada al algoritmo será un vector  $p$  de puntos y los límites  $c$  y  $f$  de la nube que estamos mirando. El vector de puntos no se modificará en ningún momento y se supone que está ordenado respecto a la coordenada  $x$ .

- \* La solución

---

```
void parMasCercano(Punto p[], int c, int f, int indY[], int& ini,
                     double& d, int& p1, int& p2)
```

---

constará de:

- Los límites  $c$  y  $f$ . Las llamadas correctas cumplen  $0 \leq c \wedge f \leq \text{long}(v) - 1 \wedge f \geq c + 1$ .
- La distancia  $d$  entre los puntos más cercanos.
- Los puntos  $p_1$  y  $p_2$  más cercanos.
- Un vector de posiciones  $\text{indY}$  y un índice inicial  $\text{ini}$  que representa cómo se ordenan los elementos de  $p$  con respecto a la coordenada  $y$ . El índice inicial  $\text{ini}$  nos indica que el punto  $p[\text{ini}]$  es el que tiene la menor coordenada  $y$ . El vector  $\text{indY}$  contiene en cada posición la posición del siguiente punto en la ordenación,

siendo  $-1$  el valor utilizado para indicar que no hay siguiente. Por ejemplo, si el vector de puntos es:

$$\{-0.5, 0.5\}, \{0, 3\}, \{0, 0\}, \{0, 0.25\}, \{1, 1\}, \{1.25, 1.25\}, \{2, 2\}$$

la variable *ini* valdrá  $2$  y el vector *indY* será:

$$\{4, -1, 3, 0, 5, 6, 1\}$$

Es decir:

- el elemento con menor *y* es el punto *p[2]*;
- como *indY[2] = 3* el siguiente es *p[3]*;
- como *indY[3] = 0*, el siguiente es *p[0]*;
- como *indY[0] = 4*, el siguiente es *p[4]*;
- como *indY[4] = 5*, el siguiente es *p[5]*;
- como *indY[5] = 6*, el siguiente es *p[6]*;
- como *indY[6] = 1*, el siguiente es *p[1]*;
- como *indY[1] = -1*, ya no hay más puntos

- 
- \* Usaremos las siguientes funciones auxiliares:

---

```
double absolute(double x) {
    if (x >= 0) return x;
    else return -x;
}

double distancia(Punto p1, Punto p2) {
    return (sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y)));
}

double minimo(double x, double y) {
    double z;
    if (x <= y) z = x; else z = y;
    return z;
}
```

---

- \* Los casos base, cuando hay 2 o 3 puntos los resuelve la función

---

```
void solucionDirecta(Punto p[], int c, int f, int indY[], int& ini,
                      double& d, int& p1, int& p2) {
    double d1, d2, d3;
    if (f == c+1) {
        d = distancia(p[c], p[f]);
        if ((p[c].y) <= (p[f].y)) {
            ini = c; indY[c] = f; indY[f] = -1; p1 = c; p2 = f;
        } else {
            ini = f; indY[f] = c; indY[c] = -1; p1 = f; p2 = c;
        }
    } else if (f == c+2) {
        // Menor distancia y puntos que la producen
        d1 = distancia(p[c], p[c+1]);
        d2 = distancia(p[c], p[c+2]);
        d3 = distancia(p[c+1], p[c+2]);
        d = minimo(minimo(d1, d2), d3);
    }
}
```

---

---

```

if (d == d1) {p1 = c; p2 = c+1;}
else if (d == d2) {p1 = c; p2 = c+2;}
else {p1 = c+1; p2 = c+2}

//Ordenar
if (p[c].y <= p[c+1].y){
    if (p[c+1].y <= p[c+2].y){
        ini = c; indY[c] = c+1; indY[c+1] = c+2; indY[c+2] = -1;
    } else if (p[c].y <= p[c+2].y){
        ini = c; indY[c] = c+2; indY[c+2] = c+1; indY[c+1] = -1;
    } else{
        ini = c+2; indY[c+2] = c; indY[c] = c+1; indY[c+1] = -1;
    }
} else{
    if (p[c+1].y > p[c+2].y){
        ini = c+2; indY[c+2] = c+1; indY[c+1] = c; indY[c] = -1;
    } else if (p[c].y > p[c+2].y){
        ini = c+1; indY[c+1] = c+2; indY[c+2] = c; indY[c] = -1;
    } else{
        ini = c+1; indY[c+1] = c; indY[c] = c+2; indY[c+2] = -1;
    }
}
}
}

```

---

- \* El método *mezclaOrdenada* recibe en *indY* dos listas de enlaces que comienzan en *ini1* y *ini2* que representan respectivamente la ordenación con respecto a *y* de los puntos de la nube izquierda y derecha, y las mezcla para obtener una única lista de enlaces con todos los puntos de la nube.

---

```

void mezclaOrdenada(Punto p[], int ini1, int ini2, int indY[], int& ini){
    int i = ini1;
    int j = ini2;
    int k;
    if (p[i].y <= p[j].y){
        ini = ini1; k = ini1; i = indY[i];
    } else{
        k = ini2; ini = ini2; j = indY[j];
    }
    while ((i != -1) && (j != -1)){
        if (p[i].y <= p[j].y){
            indY[k] = i; k = i; i = indY[i];
        } else{
            indY[k] = j; k = j; j = indY[j];
        }
    }
    if (i == -1) indY[k] = j;
    else indY[k] = i;
}

```

---

En el ejemplo de antes, despues de las dos llamadas recursivas tendríamos que *ini1* = 2, *ini2* = 4 y el vector *indY* es:

$$\{1, -1, 3, 0, 5, 6, -1\}$$

representando dos listas de puntos  $p[2], p[3], p[0], p[1]$  y por otro lado  $p[4], p[5], p[6]$ . Después de ejecutar *mezclaOrdenada* obtenemos el resultado de arriba. Este método se puede utilizar igualmente en una versión del algoritmo *mergesort* que devuelva un vector de enlaces.

- \* Así el algoritmo queda:

---

```

void parMasCercano(Punto p[], int c, int f, int indY[], int& ini,
                     double& d, int& p1, int& p2) {
    int m; int i, j, ini1, ini2, p11, p12, p21, p22;
    double d1, d2;

    if (f-c+1 < 4)
        solucionDirecta(p,c,f,indY,ini,d,p1,p2);
    else{
        m = (c+f)/2;
        parMasCercano(p,c,m,indY,ini1,d1,p11,p12);
        parMasCercano(p,m+1,f,indY,ini2,d2,p21,p22);

        if (d1 <= d2){
            d = d1; p1 = p11; p2 = p12;
        } else{
            d = d2; p1 = p21; p2 = p22;
        }

        //Mezcla ordenada por la y
        mezclaOrdenada(p,ini1,ini2,indY,ini);

        //Filtrar la lista
        i = ini;
        while (absolute(p[m].x-p[i].x)>d) i = indY[i];

        int iniA = i;
        int indF[f-c+1];
        for (int l = 0; l <= f-c+1; l++) indF[l] = -1;
        int k = iniA - c;
        while (i != -1){
            if (absolute(p[m].x-p[i].x) <= d) {indF[k] = i-c; k = i-c;}
            i = indY[i];
        }

        //Calcular las distancias
        i = iniA-c;
        while (i != -1){
            int count = 0; j = indF[i];
            while (j != -1 && count < 7){
                double daux = distancia(p[i+c],p[j+c]);
                if (daux < d) {d = daux; p1 = i+c; p2 = j+c;}
                j = indF[j];
                count = count+1;
            }
            i = indF[i];
        }
    }
}

```

---

- \* La llamada inicial es:

```
parMasCercano(p, 0, long(v)-1, indY, ini, d, p1, p2).
```

## 8. La determinación del umbral

- \* Dado un algoritmo DV, casi siempre existe una versión asintóticamente menos eficiente pero de constantes multiplicativas más pequeñas. Le llamaremos el *algoritmo sencillo*. Eso hace que para valores pequeños de  $n$ , sea más eficiente el algoritmo sencillo que el algoritmo DV.
- \* Se puede conseguir un algoritmo óptimo combinando ambos algoritmos de modo inteligente. El aspecto que tendría el algoritmo compuesto es:

---

```
Solucion divideYvenceras (Problema x, int n) {
    if (n <= n_0)
        return algoritmoSencillo(x)
    else /* n > n_0 */ {
        descomponer x
        llamadas recursivas a divideYvenceras
        y = combinar resultados
        return y;
    }
}
```

---

- \* Es decir, se trata de convertir en casos base del algoritmo recursivo los problemas que son *suficientemente* pequeños. Nos planteamos cómo determinar el **umbral**  $n_0$  a partir del cual compensa utilizar el algoritmo sencillo con respecto a continuar subdividiendo el problema.
- \* La determinación del umbral es un tema fundamentalmente **experimental**, depende del computador y lenguaje utilizados, e incluso puede no existir un óptimo único sino varios en función del tamaño del problema.
- \* A pesar de eso, se puede hacer un estudio teórico del problema para encontrar un umbral aproximado. Para fijar ideas, centrémonos en el problema de encontrar el par más cercano y escribamos su recurrencia con constantes multiplicativas (suponemos  $n$  potencia de 2):

$$T_1(n) = \begin{cases} c_0 & \text{si } 0 \leq n \leq 3 \\ 2T_1(n/2) + c_1n & \text{si } n \geq 4 \end{cases}$$

Si desplegamos esta recurrencia y la resolvemos exactamente, la expresión de coste resulta ser:

$$T_1(n) = c_1n \log n + \left(\frac{1}{2}c_0 - c_1\right)n$$

- \* Por otra parte, el algoritmo sencillo tendrá un coste  $T_2(n) = c_2n^2$ . Las constantes  $c_0$ ,  $c_1$  y  $c_2$  dependen del lenguaje y de la máquina subyacentes, y han de ser determinadas experimentalmente para cada instalación.
- \* Aparentemente, para encontrar el umbral hay que resolver la ecuación  $T_1(n) = T_2(n)$ , es decir encontrar un  $n_0$  que satisfaga:

$$c_1n \log n + \left(\frac{1}{2}c_0 - c_1\right)n = c_2n^2$$

Sin embargo, este planteamiento es **incorrecto** porque el coste del algoritmo DV está calculado subdividiendo  $n$  hasta los casos base. Es decir, estamos comparando el algoritmo DV puro con el algoritmo sencillo puro y lo que queremos saber es cuándo subdividir es más costoso que no subdividir.

- \* La ecuación que necesitamos es la siguiente:

$$2T_2(n/2) + c_1n = c_2n^2 = T_2(n)$$

que expresa que en una llamada recursiva al algoritmo DV decidimos subdividir **por última vez** porque es tan costoso subdividir como no hacerlo. Nótese que el coste de las dos llamadas internas está calculado con el algoritmo sencillo, lo que confirma que esta subdivisión es la última que se hace.

- \* Resolviendo esta ecuación obtenemos:

$$2c_2 \left(\frac{n}{2}\right)^2 + c_1n = c_2n^2 \Rightarrow n_0 = \frac{2c_1}{c_2}$$

Para  $n > n_0$ , la expresión de la izquierda crece más despacio que la de la derecha y merece la pena subdividir. Para valores menores que  $n_0$ , la expresión de la derecha es menos costosa.

- \* Como sabemos,  $c_1$  mide el número de operaciones elementales que hay que hacer con cada punto de la nube de puntos en la parte no recursiva del algoritmo DV. Es decir la suma por punto de dividir la lista en dos, mezclar las dos mitades ordenadas, filtrar los puntos de la banda y recorrer la misma, comparando cada punto con otros siete. Por su parte,  $c_2$  mide el coste elemental de cada una de las  $n^2$  operaciones del algoritmo sencillo. Este coste consiste en esencia en la mitad de calcular la distancia entre dos puntos y comparar con el mínimo en curso.

Supongamos que, una vez medidas experimentalmente, obtenemos  $c_1 = 32c_2$ . Ello nos daría un umbral  $n_0 = 64$ .

- \* Es interesante escribir y resolver la recurrencia del algoritmo híbrido así conseguido y comparar el coste con el del algoritmo DV original:

$$T_3(n) = \begin{cases} c_2n^2 & \text{si } n \leq 64 \\ 2T_3(n/2) + c_1n & \text{si } n > 64 \end{cases}$$

Si desplegamos  $i$  veces, obtenemos:

$$T_3(n) = 2^i T_3\left(\frac{n}{2^i}\right) + i c_1 n$$

que alcanza el caso base cuando  $\frac{n}{2^i} = 2^6 \Rightarrow i = \log n - 6$ . Entonces sustituimos  $i$ :

$$\begin{aligned} T_3(n) &= \frac{n}{2^6} T_3(2^6) + c_1(\log n - 6)n \\ &= c_1 n \log n + c_2 \frac{n}{2^6} 2^{12} - 6c_1 n \\ &= c_1 n \log n - 4c_1 n \end{aligned}$$

- \* Comparando el coste  $T_3(n)$  del algoritmo híbrido con el coste  $T_1(n)$  del algoritmo DV puro, se aprecia una diferencia importante en la constante multiplicativa del término de segundo orden.

## Notas bibliográficas

En (Martí Oliet et al., 2013, Cap. 11) se repasan los fundamentos de la técnica DV y hay numerosos ejercicios resueltos. El capítulo (Brassard y Bratley, 1997, Cap. 7) también está dedicado a la técnica DV y uno de los ejemplos es el algoritmo de Karatsuba y Ofman. El ejemplo del par más cercano está tomado de (Cormen et al., 2001, Cap. 33).

## Ejercicios

1. Dos amigos matan el tiempo de espera en la cola del cine jugando a un juego muy sencillo: uno de ellos piensa un número natural positivo y el otro debe adivinarlo preguntando solamente si es menor o igual que otros números. Diseñar un algoritmo eficiente para adivinar el número.
2. Desarrollar un algoritmo DV para multiplicar  $n$  números complejos usando tan solo  $3(n - 1)$  multiplicaciones.
3. Dados un vector  $v[0..n - 1]$  y un valor  $k$ ,  $1 \leq k \leq n$ , diseñar un algoritmo DV de coste constante en espacio que trasponga las  $k$  primeras posiciones del vector con las  $n - k$  siguientes. Es decir, los elementos  $v[0] \dots v[k - 1]$  han de copiarse a las posiciones  $n - k \dots n - 1$ , y los elementos  $v[k] \dots v[n - 1]$  han de copiarse a las posiciones  $0 \dots n - k - 1$ .
4. Dado un vector  $T[1..n]$  de  $n$  elementos (que tal vez no se puedan ordenar), se dice que un elemento  $x$  es *mayoritario* en  $T$  cuando el número de veces que  $x$  aparece en  $T$  es estrictamente mayor que  $N/2$ .
  - a) Escribir un algoritmo DV que en tiempo  $O(n \log n)$  decida si un vector  $T[0..n - 1]$  contiene un elemento mayoritario y devuelva tal elemento cuando exista.
  - b) Suponiendo que los elementos del vector  $T[0..n - 1]$  se puedan ordenar, y que podemos calcular la mediana de un vector en tiempo lineal, escribir un algoritmo DV que en tiempo lineal decida si  $T[0..n - 1]$  contiene un elemento mayoritario y devuelva tal elemento cuando exista. La mediana se define como el valor que ocuparía la posición  $\frac{n-1}{2}$  si el vector estuviese ordenado.
  - c) Idear un algoritmo que no sea DV, tenga coste lineal, y no suponga que los elementos se pueden ordenar.
5. a) (ACR295) Dado un valor  $x$  fijo, escribir un algoritmo para calcular  $x^n$  con un coste  $O(\log n)$  en términos del número de multiplicaciones.
  - b) Sea  $F$  la matriz  $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ . Calcular el producto del vector  $(i \ j)$  y la matriz  $F$ . ¿Qué ocurre cuando  $i$  y  $j$  son números consecutivos de la sucesión de Fibonacci?
  - c) (ACR306) Utilizando las ideas de los dos apartados anteriores, desarrollar un algoritmo para calcular el  $n$ -ésimo número de Fibonacci  $f(n)$  con un coste  $O(\log n)$  en términos del número de operaciones aritméticas elementales.
6. En un habitación oscura se tienen dos cajones, en uno de los cuales hay  $n$  tornillos de varios tamaños, y en el otro las correspondientes  $n$  tuercas. Es necesario emparejar cada tornillo con su tuerca correspondiente, pero debido a la oscuridad no se pueden comparar tornillos con tornillos ni tuercas con tuercas, y la única comparación

posible es la de intentar enroscar una tuerca en un tornillo para comprobar si es demasiado grande, demasiado pequeña, o se ajusta perfectamente al tornillo. Desarrollar un algoritmo para emparejar los tornillos con las tuercas que use  $O(n \log n)$  comparaciones en promedio.

7. Dado un vector  $C[0..n - 1]$  de números enteros distintos, y un número entero  $S$ , se pide:
  - a) Diseñar un algoritmo de complejidad  $\Theta(n \log n)$  que determine si existen o no dos elementos de  $C$  tales que su suma sea exactamente  $S$ .
  - b) Suponiendo ahora ordenado el vector  $C$ , diseñar un algoritmo que resuelva el mismo problema en tiempo  $\Theta(n)$ .
8. Mr. Scrooge ha cobrado una antigua deuda, recibiendo una bolsa con  $n$  monedas de oro. Su olfato de usurero le asegura que una de ellas es falsa, pero lo único que la distingue de las demás es su peso, aunque no sabe si este es mayor o menor que el de las otras. Para descubrir cuál es la falsa, Mr. Scrooge solo dispone de una balanza con dos platillos para comparar el peso de dos conjuntos de monedas. En cada pesada lo único que puede observar es si la balanza queda equilibrada, si pesan más los objetos del platillo de la derecha o si pesan más los de la izquierda. Suponiendo  $n \geq 3$ , diseñar un algoritmo DV para encontrar la moneda falsa y decidir si pesa más o menos que las auténticas.
9. Se dice que un punto del plano  $A = (a_1, a_2)$  domina a otro  $B = (b_1, b_2)$  si  $a_1 > b_1$  y  $a_2 > b_2$ . Dado un conjunto  $S$  de puntos en el plano, el rango de un punto  $A \in S$  es el número de puntos que domina. Escribir un algoritmo de coste  $O(n \log^2 n)$  que dado un conjunto de  $n$  puntos calcule el rango de cada uno; demostrar que su coste efectivamente es el requerido.
10. **La línea del cielo de Manhattan.** Dado un conjunto de  $n$  rectángulos (los edificios), cuyas bases descansan todas sobre el eje de abcisas, hay que determinar mediante DV la cobertura superior de la colección (la línea del cielo).

La línea del cielo puede verse como una lista ordenada de segmentos horizontales, cada uno comenzando en la abcisa donde el segmento precedente terminó. De esta forma, puede representarse mediante una secuencia alternante de abcisas y ordenadas, donde cada ordenada indica la altura de su correspondiente segmento:

$$(x_1, y_1, x_2, y_2, \dots, x_{m-1}, y_{m-1}, x_m)$$

con  $x_1 < x_2 < \dots < x_m$ . Por convenio, la línea del cielo está a altura 0 hasta alcanzar  $x_1$ , y vuelve a 0 después de  $x_m$ . Se usa la misma representación para cada rectángulo, es decir  $(x_1, y_1, x_2)$  corresponde al rectángulo con vértices  $(x_1, 0)$ ,  $(x_1, y_1)$ ,  $(x_2, y_1)$  y  $(x_2, 0)$ .

11. Sea un vector  $V[0..n - 1]$  que contiene valores enteros positivos que se ajustan al perfil de una curva cóncava; es decir, para una cierta posición  $k$ , tal que  $0 \leq k < n$ , se cumple que  $\forall j \in \{0..k - 1\}. V[j] > V[j + 1]$  y  $\forall j \in \{k + 1..n - 1\}. V[j - 1] < V[j]$  (por ejemplo el vector  $V = [9, 8, 7, 3, 2, 4, 6]$ ). Se pide diseñar un algoritmo que encuentre la posición del mínimo en el vector (la posición 4 en el ejemplo), teniendo en cuenta que el algoritmo propuesto debe de ser más eficiente que el conocido de búsqueda secuencial del mínimo en un vector cualquiera.

12. La *envolvente convexa* de una nube de puntos en el plano es el menor polígono convexo que incluye a todos los puntos. Si los puntos se representaran mediante clavos en un tablero y extendiéramos una goma elástica alrededor de todos ellos, la forma que adoptaría la goma al soltarla sería la envolvente convexa. Dada una lista  $l$  de  $n$  puntos, se pide un algoritmo de coste  $\Theta(n \log n)$  que calcule su envolvente convexa.
13. Las matrices de Hadamard  $H_0, H_1, H_2, \dots$ , se definen del siguiente modo:

- $H_0 = (1)$  es una matriz  $1 \times 1$ .
- Para  $k > 0$ ,  $H_k$  es la matriz  $2^k \times 2^k$

$$H_k = \left( \begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array} \right)$$

Si  $v$  es un vector columna de longitud  $n = 2^k$ , escribir un algoritmo que calcule el producto  $H_k \cdot v$  en tiempo  $O(n \log n)$  suponiendo que las operaciones aritméticas básicas tienen un coste constante. Justificar el coste.

14. Diseñar un algoritmo de coste en  $O(n)$  que dado un conjunto  $S$  con  $n$  números, y un entero positivo  $k \leq n$ , determine los  $k$  números de  $S$  más cercanos a la mediana de  $S$ .
15. La  $p$ -mediana generalizada de una colección de  $n$  valores se define como *la media* de los  $p$  valores ( $p+1$  si  $p$  y  $n$  tuvieran distinta paridad) que ocuparían las posiciones centrales si se ordenara la colección. Diseñar un algoritmo que resuelva el problema en un tiempo a lo sumo  $O(n \log(n))$ , asumiendo  $p$  constante distinto de  $n$ .
16. Se tiene un vector  $V$  de números enteros distintos, con pesos asociados  $p_1, \dots, p_n$ . Los pesos son valores no negativos y verifican que  $\sum_{i=1}^n p_i = 1$ . Se define la *mediana ponderada* del vector  $V$  como el valor  $V[m]$ ,  $1 \leq m \leq n$ , tal que

$$\left( \sum_{V[i] < V[m]} p_i \right) < \frac{1}{2} \quad \text{y} \quad \left( \sum_{V[i] \leq V[m]} p_i \right) \geq \frac{1}{2}.$$

Por ejemplo, para  $n = 5$ ,  $V = [4, 2, 9, 3, 7]$  y  $P = [0.15, 0.2, 0.3, 0.1, 0.25]$ , la media ponderada es  $V[5] = 7$  porque

$$\sum_{V[i] < 7} p_i = p_1 + p_2 + p_4 = 0.15 + 0.2 + 0.1 = 0.45 < \frac{1}{2}, \text{ y}$$

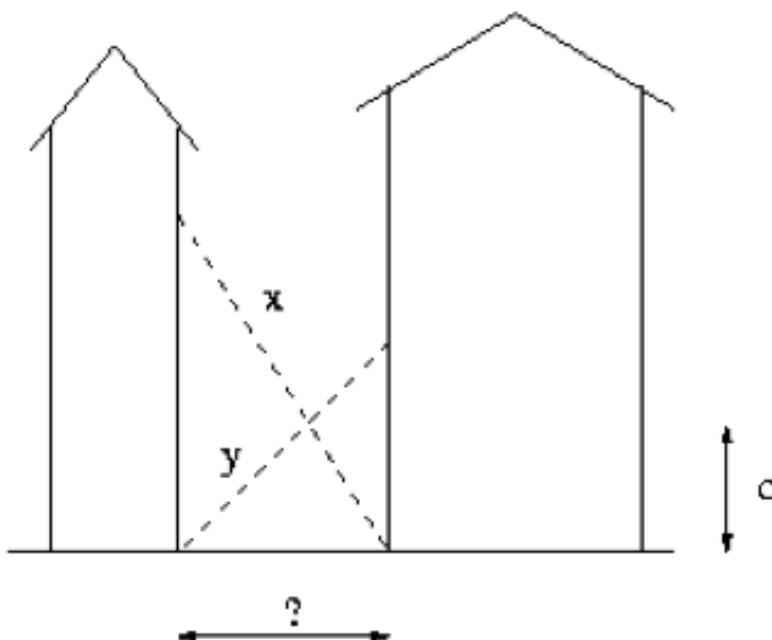
$$\sum_{V[i] \leq 7} p_i = p_1 + p_2 + p_4 + p_5 = 0.15 + 0.2 + 0.1 + 0.25 = 0.7 \geq \frac{1}{2}.$$

Diseñar un algoritmo de tipo *divide y vencerás* que encuentre la mediana ponderada en un tiempo lineal en el caso peor. (Obsérvese que  $V$  puede no estar ordenado.)

17. Se tienen  $n$  bolas de igual tamaño, todas ellas de igual peso salvo dos más pesadas, que a su vezpesan lo mismo. Como único medio para dar con dichas bolas se dispone de una balanza romana clásica. Diseñar un algoritmo que permita determinar cuáles son dichas bolas, con el mínimo posible de pesadas (que es logarítmico).

**Indicación:** Considerar también el caso en el que solo hay una bola diferente.

- 18. Escaleras cruzadas** Una calle estrecha está flanqueada por dos edificios muy altos. Se colocan dos escaleras en dicha calle como muestra el dibujo: una de ellas, de longitud  $x$  metros, colocada en la base del edificio que está en el lado derecho de la calle y apoyada sobre la fachada del edificio situado en el lado izquierdo de la calle; la otra, de longitud  $y$  metros, colocada en la base del edificio que está en el lado izquierdo de la calle y apoyada sobre la fachada del edificio situado en el lado derecho de la calle. El punto donde se cruzan ambas escaleras está a altura exactamente  $c$  metros del suelo. Se pide diseñar un algoritmo que calcule la anchura de la calle con tres decimales de precisión.



(Enunciado original en <http://uva.onlinejudge.org/external/105/10566.pdf>)

- 19. Resolución de una ecuación** Dados números reales  $p, q, r, s, t, u$  tales que  $0 \leq p, r \leq 20$  y  $-20 \leq q, s, t \leq 0$ , se desea resolver la siguiente ecuación en el intervalo  $[0, 1]$  con cuatro decimales de precisión:

$$p * e^{-x} + q * \operatorname{sen}(x) + r * \cos(x) + s * \tan(x) + t * x^2 + u = 0$$

(Enunciado original en <http://uva.onlinejudge.org/external/103/10341.pdf>)

---

## Capítulo 3

# Vuelta Atrás<sup>1</sup>

---

*Más vale una retirada a tiempo que una batalla perdida.*

Napoleón Bonaparte.

**RESUMEN:** En este tema se introduce el esquema algorítmico de *Vuelta atrás*, o *Backtracking*, que es una técnica de carácter general utilizada para resolver una gran clase de problemas, en especial de problemas que exigen un recorrido exhaustivo del universo de soluciones.

### 1. Motivación

Dado un mapa  $M$  y un número  $n > 0$  se pide encontrar las formas de colorear los países de  $M$  utilizando un máximo de  $n$  colores, de tal manera que ningún par de países fronterizos tenga el mismo color.

Una forma de resolver el problema es generar todas las posibles maneras de colorear el mapa y después desechar aquellas en que dos países fronterizos tienen el mismo color. Esta forma de resolver el problema sólo resulta aceptable si el conjunto de países,  $m$ , y el conjunto de colores,  $n$ , son pequeños, ya que el número de posibles formas de colorear el mapa viene dado por las variaciones con repetición de  $n$  elementos tomados de  $m$  en  $m$ :  $VR_n^m = n^m$ . En la siguiente tabla se muestra el número de posibilidades que habría que generar para algunos valores de  $n$  y  $m$ .

| $n$ : colores | $m$ : países | nº de posibilidades             |
|---------------|--------------|---------------------------------|
| 3             | 17           | 129.140.163                     |
| 5             | 17           | 762.939.453.125                 |
| 3             | 50           | 717.897.987.691.852.588.770.249 |

Se observa, que si tomamos las 17 comunidades autónomas de España, el número de posibilidades a generar con 5 colores sería superior a los 700 mil millones. Si consideramos las 50 provincias, con sólo 3 colores, el número de posibilidades supera el millón de billones.

---

<sup>1</sup> Miguel Valero Espada es el autor principal de este tema.  
Modificado por Isabel Pita en el curso 2012/13.

En este capítulo se explica como abordar aquellos problemas cuya única forma conocida de resolver es la generación de todas sus posibles soluciones, para obtener de entre ellas la solución o soluciones reales.

## 2. Introducción

Existen problemas, como el que se presenta en el apartado anterior, para los que no parece existir una forma o regla fija que nos lleve a obtener una solución de una manera eficiente y precisa. La manera de resolverlos consiste en realizar una búsqueda exhaustiva entre todas las soluciones potenciales hasta encontrar una solución válida, o el conjunto de todas las soluciones válidas. A veces se requiere encontrar *la mejor* (en un sentido preciso) de todas las soluciones válidas.

La búsqueda exhaustiva en un espacio finito dado se conoce como *fuerza bruta* y consiste en ir probando sistemáticamente todas las soluciones potenciales hasta encontrar una solución satisfactoria, o bien agotar el universo de posibilidades. En general, la *fuerza bruta* es impracticable para espacios de soluciones potenciales grandes, lo cual ocurre muy a menudo, ya que el número de soluciones potenciales tiende a crecer de forma exponencial con respecto al tamaño de la entrada en la mayoría de los problemas que trataremos. Obsérvense los valores que se proporcionan en el ejemplo de la sección anterior.

El esquema algorítmico de *vuelta atrás* es una mejora a la estrategia de *fuerza bruta*, ya que la búsqueda se realiza de manera estructurada, descartando grandes bloques de soluciones para reducir el espacio de búsqueda. La diferencia principal entre ambos esquemas es que en el primero las soluciones se forman de manera progresiva, generando soluciones parciales, comprobando en cada paso si la solución que se está construyendo puede conducir a una solución satisfactoria. Mientras que en la *fuerza bruta* la estrategia consiste en probar una solución potencial completa tras otra sin ningún criterio.

En el esquema de *vuelta atrás*, si una solución parcial no puede llevar a una solución completa satisfactoria, la búsqueda se aborta y se vuelve a una solución parcial viable, deshaciendo decisiones previas. Este esquema debe su nombre a este salto hacia atrás.

Tomemos como ejemplo el siguiente problema: dadas  $n$  letras diferentes, diseñar un algoritmo que calcule las palabras con  $m$  letras ( $m \leq n$ ) diferentes escogidas entre las dadas. El orden de las letras es importante: no será la misma solución *abc* que *bac*.

El número de soluciones potenciales o *espacio de búsqueda* es de  $n^m$ , que representan las variaciones con repetición de  $n$  letras tomadas de  $m$  en  $m$ . Realizar una búsqueda exhaustiva en este espacio es impracticable.

Antes de ponernos a probar sin criterio alguno entre todas las posibles combinaciones de letras, dediquemos un segundo a evaluar la estructura de la solución. Es evidente que no podemos poner dos letras iguales en la misma palabra, así que vamos a replantear el problema. Trataremos de colocar las letras de una en una, de forma que no se repitan. De esta manera, toda solución del problema se puede representar como una tupla  $(x_1, \dots, x_m)$  en la que  $x_i$  representa la letra que se coloca en el lugar  $i$ -ésimo de la palabra.

La solución del problema se construye de manera incremental, colocando una letra detrás de otra. En cada paso se comprueba que la última letra no esté repetida con las anteriores. Si la última letra colocada no está repetida, la solución parcial se dice *prometedora* y la búsqueda de la solución continúa a partir de ella. Si no es prometedora, se abortan todas las búsquedas que partan de esa tupla parcial.

De manera general, en los algoritmos de *vuelta atrás*, se consideran problemas cuyas soluciones se puedan construir por etapas. Una solución se expresa como  $n$ -tupla  $(x_1, \dots, x_n)$

donde cada  $x_i \in S_i$  representa la decisión tomada en la  $i$ -ésima etapa de entre un conjunto finito de alternativas.

Una solución tendrá que minimizar, maximizar, o simplemente satisfacer cierta *función criterio*. Se establecen dos categorías de restricciones para los posibles valores de una tupla:

- **Restricciones explícitas**, que indican los conjuntos  $S_i$ . Es decir, el conjunto finito de alternativas entre las cuales pueden tomar valor cada una de las componentes de la tupla solución.
- **Restricciones implícitas**, que son las relaciones que se han de establecer entre las componentes de la tupla solución para satisfacer la función criterio.

Volviendo al ejemplo de las palabras con letras diferentes, hemos visto que la solución será una  $m$ -tupla y que cada valor de la tupla es una letra en la palabra. Las restricciones serán:

- **Restricciones explícitas para el problema de las palabras:**
  - $S_i = \{1, \dots, n\}$ ,  $1 \leq i \leq m$ . Es decir, cada letra tiene que pertenecer al alfabeto.
- **Restricciones implícitas para el problema de las palabras:**
  - No puede haber dos letras iguales en la misma palabra.
  - Las soluciones son, por tanto, variaciones sin repetición de  $n$  elementos tomados de  $m$  en  $m$ , es decir  $\frac{n!}{(n-m)!}$ . Si consideramos palabras de 5 letras sobre un alfabeto de 27 letras distintas, el número de posibilidades se reduce de  $27^5 = 14.348.907$  a  $\frac{27!}{22!} = 9.687.600$ .

El espacio de soluciones potenciales a explorar estará formado por el conjunto de tuplas que satisfacen las restricciones explícitas. Este espacio, se puede estructurar como un *árbol de exploración*, donde en cada nivel se toma la decisión sobre la etapa correspondiente.

Se denomina *nodo de estado* a cualquier nodo del árbol de exploración que satisface **las restricciones explícitas**, y corresponde a una tupla parcial o una tupla completa. Los *nodos solución* serán los correspondientes a las tuplas completas que además satisfagan **las restricciones implícitas**.

Un elemento adicional imprescindible es la *función de poda* o *test de factibilidad*, que permite determinar cuándo una solución parcial puede conducir a una solución satisfactoria. De tal manera que si un *nodo* no satisface la función de poda es inútil continuar la búsqueda por esa rama del árbol. La *función de poda* permite pues reducir la búsqueda en el árbol de exploración.

Una vez definido el árbol de exploración, el algoritmo realizará un recorrido del árbol en cierto orden, hasta encontrar la primera solución. El mismo algoritmo, con ligeras modificaciones, se podrá utilizar para encontrar todas las soluciones, o una solución óptima.

El árbol de exploración no se construye de manera explícita, es decir no se almacena en memoria, sino que se va construyendo de manera implícita conforme avanza la búsqueda por medio de llamadas recursivas. Durante el proceso, para cada nodo se generarán los nodos sucesores (estados alcanzables tomando una determinada decisión correspondiente a la siguiente etapa). En el proceso de generación habrá distintos tipos de nodos:

**Nodos vivos** Aquellos para los cuales aún no se han generado todos sus hijos. Todavía pueden expandirse.

**Nodo en expansión** Aquel para el cual se están generando sus hijos.

**Nodos muertos** Aquellos que no hay que seguir explorando porque, o bien no han superado el test de factibilidad, o bien se han explorado insatisfactoriamente todos sus hijos.

El recorrido del árbol de exploración se realiza en profundidad. Cuando se llega a un nodo muerto, hay que deshacer la última decisión tomada y optar por otra alternativa (*vuelta atrás*). La forma más sencilla de expresar este retroceso es mediante un algoritmo recursivo, ya que la vuelta atrás se consigue automáticamente haciendo terminar la llamada recursiva y volviendo a aquella que la invocó.

El coste de los algoritmos de *vuelta atrás* en el caso peor es del orden del tamaño del árbol de exploración, ya que en el peor de los casos nos veremos obligados a recorrer exhaustivamente todas las posibilidades. El espacio de soluciones potenciales suele ser, como mínimo, exponencial en el tamaño de la entrada. La efectividad de la *vuelta atrás* va a depender decisivamente de las funciones de poda que se utilicen, ya que si son adecuadas permitirán reducir considerablemente el número de nodos explorados.

Se podrían realizar búsquedas más inteligentes haciendo que en cada momento se explore el nodo más prometedor, utilizando para ello algún tipo de heurística que permita ordenar los nodos en un tipo de datos denominado *cola de prioridad*. Esta estrategia da lugar al esquema conocido como de **ramificación y poda**. Las colas de prioridad y el esquema de ramificación y poda se estudiará el próximo curso.

## 2.1. Esquema básico de la *vuelta atrás*

El esquema de *Vuelta atrás* en pseudocódigo es el siguiente:

---

```

vueltaAtras (Tupla & sol,  int k) {
    prepararRecorridoNivel(k);
    while (!ultimoHijoNivel(k)) {
        sol[k] = siguienteHijoNivel(k);
        if (esValida(sol, k)){
            if (esSolucion(sol, k))
                tratarSolucion(sol);
        } else
            vueltaAtras(sol, k + 1);
    }
}

```

---

El tipo de la solución *sol* es una tupla de cierto tipo específico para cada problema. En ella se va acumulando la solución. La variable *k* es la que determina en qué nivel del árbol de exploración estamos.

El método *prepararRecorridoNivel* genera los candidatos para ampliar la solución en la siguiente etapa y depende del problema en concreto. En el cuerpo de la función, iteramos a lo largo de todas las posibles soluciones candidatas, dadas por la función: *siguienteHijoNivel* hasta la última candidata, dada por la función: *ultimoHijoNivel*. Para cada solución candidata, ampliamos la solución con el nuevo valor y comprobamos si satisface las restricciones implícitas/explícitas con la función booleana *esValida*. Esta función implementa la *función de factibilidad* que presentábamos más arriba.

En el caso de que la solución parcial sea válida tenemos dos posibilidades: o bien hemos alcanzado el final de la búsqueda, por lo que ya podemos mostrar la solución final, o bien continuamos nuestra búsqueda mediante la llamada recursiva.

Este esquema encontrará todas las soluciones del problema. Si quisieramos que sólo encontrara una solución bastaría con añadir una variable booleana `éxito` que haga finalizar los bucles cuando se encuentra la primera solución.

## 2.2. Resolución del problema de las palabras

Veamos cómo se aplica el esquema al problema de las palabras que hemos descrito más arriba. El esquema presentado tendrá pequeñas variaciones en cada problema, hay que tomarlo como una referencia y no como un patrón estricto. En el caso de las palabras la función recursiva en C++ podría ser como sigue.

---

```
void variaciones(int solucion[], int k, int n, int m) {
    for(int letra = 0; letra < n; letra++) {
        solucion[k] = letra;
        if(esValida(solucion, k)) {
            if(esSolucion(k, m))
                tratarSolucion(solucion,m);
            else
                variaciones(solucion, k + 1, n, m);
        }
    }
}
```

---

La función `prepararRecorridoNivel` no existe explícitamente ya que en este caso la iteración es muy simple y se aplica sobre valores numéricos del 0 al  $n - 1$ .

La función `esValida` es la encargada de comprobar que la nueva letra que hemos incorporado a la solución no está repetida con las anteriores. La podríamos escribir como sigue:

---

```
bool esValida(int solucion[], int k) {
    int i == 0;
    while(i < k && solucion[i] != solucion[k]) i++;
    return i == k;
}
```

---

La función `esSolucion` simplemente comprueba que hemos colocado todas las letras.

---

```
bool esSolucion(int k, int m) {
    return k == (m - 1);
}
```

---

Cuando encontramos una solución, podemos simplemente escribirla por la salida estándar:

---

```
void tratarSolucion(int solucion[], int m) {
    cout << "Solucion: ";
    for(int i = 0; i < m; i++)
        cout << solucion[i] << " ";
    cout << endl;
}
```

---

Por último, la llamada inicial será de la siguiente manera:

---

```
void variaciones(int n, int m) {
    int solucion[m];
    variaciones(solucion, 0, n, m);
```

---

```

    }

int main()
{
    variaciones(27, 5);
    return 0;
}

```

---

La función `esValida` recorre la lista de letras y comprueba si la última letra insertada en la solución coincide con alguna de las anteriores. Esta operación tiene un coste lineal en función de la entrada. La función se ejecuta muchas veces por lo que el coste de la función repercute negativamente en la ejecución del programa. Podríamos ahorrarnos este coste utilizando lo que se conoce como la técnica de *marcaje*.

### 3. Vuelta atrás con marcaje

La técnica de *marcaje* consiste en guardar cierta información que ayuda a decidir si una solución parcial es válida o no. La información del *marcaje* se pasa en cada llamada recursiva. Por lo tanto, reduce el coste computacional a cambio de utilizar más memoria. El esquema de *vuelta atrás* con marcaje es el siguiente:

```

vueltaAtrasConMarcaje (Tupla & sol, int k, Marca & marcas) {
    prepararRecorridoNivel(k);
    while (!ultimoHijoNivel(k)){
        sol[k] = siguienteHijoNivel(k);
        if (esValida(sol, k, marcas)){
            if (esSolucion(sol, k))
                tratarSolucion(sol);
            else{
                marcar(marcas, sol, k);
                vueltaAtrasConMarcaje(sol, k + 1, marcas);
                desmarcar(marcas, sol, k);
            }
        }
    }
}

```

---

El tipo *Marca* depende de cada problema concreto.

Normalmente, *desmarcaremos* después de la llamada recursiva para devolver las marcas a su estado anterior a la llamada. En algunos casos no es necesario *desmarcar*, como ocurre en el ejemplo 6.

En el ejemplo de las palabras, podemos utilizar marcaje para evitar el bucle de comprobación cada vez que aumentamos la solución. Para ello utilizamos un vector de booleanos de tamaño el número de letras del alfabeto considerado. Cada posición del vector indica si la letra correspondiente ha sido ya utilizada. De esta forma, las operaciones de *marcar* una letra como ya utilizada y consultar si una letra ya está utilizada tienen ambos coste constante. La solución con marcaje quedaría como sigue.

```

void variaciones(int solucion[], int k, int n, int m, bool marcas[]){
    for(int letra = 0; letra < n; letra++){
        if (!marcas[letra]){
            solucion[k] = letra;

```

---

```
        if(k == m - 1) {
            tratarSolucion(solucion,m);
        }
        else{
            marcas[letra] = true; //marcar
            variaciones(solucion, k + 1, n, m, marcas);
            marcas[letra] = false; //desmarcar
        }
    }
}
```

Observar que al finalizar la llamada recursiva se procede a desmarcar la letra correspondiente a esa llamada recursiva. Los parámetros de entrada `solucion` y `marcas`, modifican su valor de una llamada recursiva a otra. No se utiliza en este caso el paso de parámetros por referencia de forma explícita, debido al tratamiento dado por C++ a los vectores, como punteros a la primera posición de memoria. Esto hace que las modificaciones realizadas en un vector queden reflejadas en las sucesivas llamadas recursivas.

#### 4. Ejemplo: problema de las $n$ -reinas

El problema de las 8 reinas consiste en colocar 8 reinas en un tablero de ajedrez sin que se amenacen. Dos reinas se amenazan si comparten la misma fila, columna o diagonal.

El número de soluciones potenciales o espacio de búsqueda teórico es de  $\binom{64}{8} = 4.426.165.368$ , que representan todas las combinaciones en las que podemos poner 8 reinas en un tablero de 64 casillas. Realizar una búsqueda exhaustiva en este espacio es impracticable, siendo todavía más problemático si ampliamos el tamaño de la entrada, por ejemplo tratando de colocar 11 reinas en un tablero de  $11 \times 11$  (743.595.781.824 soluciones potenciales).

Si evaluamos la estructura de la solución vemos que no podemos poner dos reinas en la misma fila. Trataremos de colocar una reina **en cada fila del tablero**, de forma que no se amenacen. De esta manera, toda solución del problema se puede representar como una 8-tupla  $(x_1, \dots, x_8)$  en la que  $x_i$  representa la columna en la que se coloca la reina que está en la fila  $i$ -ésima del tablero.

La tupla  $(4, 7, 3, 8, 2, 5, 1, 6)$  representa el siguiente tablero.

|   |   |   |   |   |
|---|---|---|---|---|
|   |   | X |   |   |
|   |   |   |   | X |
|   | X |   |   |   |
|   |   |   |   | X |
| X |   |   | X |   |
|   |   |   |   | X |

Las restricciones serán:

- Restricciones explícitas para el problema de las reinas:

- $S_i = \{1, \dots, 8\}$ ,  $1 \leq i \leq 8$ . Es decir, cada columna tiene que estar dentro del tablero.

- Esta representación hace que el espacio de soluciones potenciales se reduzca a  $8^8$  posibilidades (16.777.216 valores).

■ **Restricciones implícitas para el problema de las reinas:**

- No puede haber dos reinas en la misma columna, ni en la misma diagonal.
- Al no poder haber dos reinas en la misma columna, se deduce que todas las soluciones son permutaciones de la 8-tupla (1, 2, 3, 4, 5, 6, 7, 8). Por lo tanto el espacio de soluciones potenciales se reduce a  $8!$  (40.320 valores diferentes).

Una función recursiva en C++ que resuelve el problema para  $n$  reinas es.

---

```
void nReinas(int solucion[], int k, int n) {
    for(int i = 0; i < n; i++) {
        solucion[k] = i;
        if (esValida(solucion, k)) {
            if(k == n - 1) {
                tratarSolucion(k, n);
            }
            else{
                nReinas(solucion, k + 1, n);
            }
        }
    }
}
```

---

La función `esValida` es la encargada de comprobar que la nueva reina que hemos incorporado a la solución no amenaza a las anteriores. La podríamos escribir como sigue:

---

```
bool esValida(int solucion[], int k) {
    bool correcto = true;
    int i = 0;
    while (i < k && correcto){
        if ((solucion[i] == solucion[k])
            || abs(solucion[k] - solucion[i]) == k - i)
            correcto = false;
        else
            i++;
    }
    return correcto;
}
```

---

Comprobamos que la nueva reina no está en la misma columna que las anteriores, `solucion[i] == solucion[k]` y que no comparten diagonal, `abs(solucion[k] - solucion[i]) == k - i`. Obviamente nunca puede estar en la misma fila por la manera en que construimos la solución.

Cuando encontramos una solución, podemos simplemente escribirla por la salida est醤dar:

---

```
void tratarSolucion(int solucion[], int n) {
    cout << "Solucion: ";
    for(int i = 0; i < n; i++)
        cout << solucion[i] << " ";
    cout << endl;
}
```

---

---

Por último, la llamada inicial será de la siguiente manera:

---

```
void nReinas(int n) {
    int solucion[n];
    nReinas(solucion, 0, n);
}

int main()
{
    nReinas(8);
    return 0;
}
```

---

Podríamos reducir el espacio de búsqueda teniendo en cuenta que las soluciones son simétricas. Si hay una solución colocando la primera reina en la casilla 2 también la habrá colocando la reina inicial en la casilla  $n - 2$ . Así pues, podríamos lanzar el método recursivo para el primer nivel sólo para las casillas menores de  $n/2$ , reduciendo el espacio de búsqueda a la mitad.

La función `esValida` recorre la lista de reinas y comprueba si la última reina insertada en la solución amenaza a las anteriores. Esta operación tiene un coste lineal en función de la entrada. Se puede reducir este coste con la técnica de marcaje.

Podemos utilizar como *marca* una estructura de datos *tablero*. Cada vez que insertamos una reina en la solución *se marcan* las casillas amenazadas por la nueva reina. Para comprobar si una nueva reina está amenazada basta con comprobar si esta marcada la casilla correspondiente del tablero. El problema de esta solución es que marcar en el tablero las casilla que amenaza la nueva reina supone un coste lineal, lo cual sigue resultando poco eficiente.

La solución está en no utilizar un tablero completo para realizar el marcaje, sino dos vectores: uno con las columnas amenazadas y otro con las diagonales amenazadas. El vector de columnas tendrá tamaño  $n$ , sin embargo, existen muchas más diagonales. Para poder resolver el problema se deben de numerar las diagonales y considerar cada posición del vector como una de ellas. La modificación y acceso a ambos vectores tendrá coste constante. El problema esta resuelto en detalle en el capítulo 14 del libro (Martí Oliet et al., 2004).

## 5. Ejemplo de búsqueda de una sola solución: Dominó

Se trata de encontrar una cadena circular de fichas de dominó. Teniendo en cuenta:

- Cada cadena tiene que utilizar las 28 fichas diferentes que contiene el juego de dominó.
- No se puede repetir ninguna ficha.
- Las cadenas tienen que ser correctas, es decir, cada ficha tiene que ser compatible con la siguiente y la cadena tiene que cerrar (el valor de un extremo de la última ficha tiene que coincidir con el otro extremo de la primera). Por ejemplo:  $6|3 \rightarrow 3|4 \rightarrow 4|1 \rightarrow 1|0 \rightarrow \dots \rightarrow 5|6$  es una cadena correcta.

La solución va a ser una tupla de 29 valores  $(x_0, \dots, x_{28})$  cada  $x_i$  es un número del 0 al 6. Es decir, en la solución no guardaremos las fichas, sino los valores de uno de los extremos. En el ejemplo de más arriba la solución tendría la siguiente forma:  $(6, 3, 4, 1, 0, \dots, 5)$ . No necesitamos guardar explícitamente los dos extremos de las fichas, ya que cada ficha tiene

que coincidir con la siguiente. Se declara una posición más en la tupla para poder realizar la comprobación de que la cadena es cerrada.

Para evitar fichas repetidas utilizaremos una matriz ( $7 \times 7$ ) donde marcaremos las fichas usadas. Hay que tener en cuenta que si marcamos la casilla  $(i, j)$ , habrá que marcar la simétrica  $(j, i)$ , ya que se trata de la misma ficha.

El problema pide que se encuentre una sola solución, no todas las que existan, así que vamos a tener que abortar la búsqueda en el momento que aparezca la primera. Para ello utilizaremos una variable de control `exito`.

La función principal será:

---

```
void domino(int sol[], int k, int n, bool marcas[NUM_VAL][NUM_VAL], bool &exito) {
    int i = 0;
    int m = (n * n + n) / 2;
    while (i < n && !exito) {
        if (!marcas[sol[k-1]][i]) {
            sol[k] = i;
            if (k == m) {
                if (sol[0] == sol[k]) {
                    tratarSolucion(sol, m);
                    exito = true;
                }
            }
            else {
                marcas[sol[k-1]][i] = true;
                marcas[i][sol[k-1]] = true;
                domino(sol, k + 1, n, marcas, exito);
                marcas[sol[k-1]][i] = false;
                marcas[i][sol[k-1]] = false;
            }
        }
        i++;
    }
}
```

---

donde  $n$  es el número de valores posibles de las fichas, en nuestro caso  $n = 7$  ya que las fichas toman valores del 0 al 6, la matriz de marcas se declara de dimensión  $n \times n$  y el vector solución es de tamaño  $(n \times n + n)/2 + 1$ .

Por tradición en el juego del dominó, siempre se empieza por el doble 6, así que pondremos los dos primeros valores como 6. Podríamos haber utilizado cualquier par de valores.

La llamada principal del programa será de la siguiente manera:

---

```
int main()
{
    const int NUM_VAL = 7;
    const int TAM_SOL = (NUM_VAL*NUM_VAL+NUM_VAL)/2+1;
    int sol[TAM_SOL];
    bool marcas[NUM_VAL][NUM_VAL];
    for(int i = 0; i < NUM_VAL; i++)
        for(int j = 0; j < NUM_VAL; j++)
            marcas[i][j] = false;

    sol[0] = NUM_VAL-1;
    sol[1] = NUM_VAL-1;
    marcas[NUM_VAL-1][NUM_VAL-1] = true;
    bool exito = false;
```

---

---

```

domino(sol, 2, NUM_VAL, marcas, exito);
return 0;
}

```

---

## 6. Ejemplo que no necesita desmarcar: El laberinto

Podemos representar un laberinto como una matriz booleana  $L$  de  $n \times n$  de tal manera que se puede pasar por las casillas con *true*. Las casillas con *false* representan los muros infranqueables. Solo nos podemos desplazar a las cuatro casillas adyacentes: arriba, abajo, izquierda y derecha. Se pide escribir un algoritmo que encuentre la salida, asumiendo que la entrada al laberinto está en la casilla  $(0, 0)$  y la salida en la  $(n - 1, n - 1)$ .

Las posibles soluciones son todas las listas de posiciones del laberinto, de longitud  $n^2$  como máximo, dado que en el peor caso visitaremos todas y cada una de las casillas. Representamos la solución como un vector  $\text{solucion}[n^2]$  de casillas, de tal manera que cada una de las casillas es transitable y cada casilla es adyacente a su siguiente.

Cada nodo del árbol de búsqueda tendrá 4 hijos correspondientes a cada una de las posibles continuaciones (arriba, abajo, izquierda y derecha).

Para controlar que no pasamos dos veces por la misma casilla mantendremos un marcador en forma de matriz  $\text{marcas}[n][n]$ , marcando aquellas casillas por las que hemos pasado. Veamos como queda el algoritmo principal:

---

```

void laberinto(bool lab[N][N], casilla solucion[], int k, int n,
               bool marcas[N][N], bool &exito) {
    int dir=0;
    while ((dir < 4)&&!exito) {
        solucion[k] = sigDireccion(dir, solucion[k-1]);
        if(esValida(lab, solucion[k], n, marcas)){
            if(esSolucion(solucion[k],n)){
                tratarSolucion(solucion, k);
                exito=true;
            }
            else{
                // marcar
                marcas[solucion[k].fila][solucion[k].columna] = true;
                laberinto(lab, solucion, k + 1, n, marcas,exito);
                //desmarcar
                //marcas[solucion[k].fila ][solucion[k].columna] = false;
            }
        }
        dir++;
    }
}

```

---

En este caso, el algoritmo no *desmarca* las posiciones utilizadas ya que al volver de la llamada recursiva o hemos encontrado la solución o un callejón sin salida, por lo que no nos interesa volver a considerarla. Nótese que si se pidiesen todas las soluciones posibles, sería necesario desmarcar ya que un camino que haya sido utilizado, si ha conducido a una solución puede ser parte de otra solución.

Para saber si una casilla es válida basta con preguntar que esté dentro de los límites del tablero, que no sea un *muro* y que no esté marcada.

---

```
bool esValida(bool lab[N][N], casilla c, int n, const bool marcas[N][N]) {
    return c.fila >= 0 && c.columna >= 0 && c.fila < n && c.columna < n
        && lab[c.fila][c.columna] && !marcas[c.fila][c.columna];
}
```

---

Para enumerar las cuatro direcciones posibles de movimiento echamos mano de la siguiente función auxiliar:

---

```
casilla sigDireccion(int dir, casilla pos) {
    switch (dir) {
        case 0:
            ++ pos.columna;
            break;
        case 1:
            ++ pos.fila;
            break;
        case 2:
            -- pos.columna;
            break;
        case 3:
            -- pos.fila;
            break;
        default:
            break;
    }
    return pos;
}
```

---

El problema asume que la salida está en la posición  $(n-1, n-1)$ , así que la comprobación de la solución será sencilla:

---

```
bool esSolucion(casilla pos, int n) {
    return pos.fila == n - 1 && pos.columna == n - 1;
}
```

---

Por último, la llamada inicial será:

---

```
int main()
{
    const int N = ...;
    bool Laberinto[N][N];
    InicializarLab(Laberinto, N);
    bool marcas[N][N];
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            marcas[i][j] = false;

    casilla sol[N*N];
    sol[0].fila = 0;
    sol[0].columna = 0;
    marcas[0][0]=true;
    bool exito=false;
    laberinto(Laberinto, sol, 1, N, marcas, exito);
    return 0;
```

---

---

 }

Vemos que fijamos el primer valor de la solución como  $(0, 0)$ , ya que esa es la posición de la salida. La función `IniciarLab` inicializa el laberinto poniendo las paredes.

Este algoritmo no tiene porque encontrar el camino óptimo, encuentra simplemente una solución; Sin embargo, *vuelta atrás* se puede utilizar para problemas de optimización como se explica más adelante.

## 7. Optimización

En muchos casos necesitamos obtener la mejor solución entre todas las soluciones posibles.

Para tratar este tipo de problemas de optimización tenemos que modificar el esquema anterior de forma que se almacene la mejor solución hasta el momento. Así, a la hora de tratar una nueva solución se comparará con la que tenemos almacenada. En general, guardaremos la mejor solución junto con su valor.

### 7.1. Ejemplo: Problema del viajante

El problema del viajante, en inglés *Travelling Salesman Problem* es uno de los problemas de optimización más estudiados a lo largo de la historia de la computación. A priori parece tener una solución sencilla pero en la práctica encontrar soluciones óptimas es muy complejo computacionalmente.

El problema se puede enunciar de la siguiente manera. Sean  $N$  ciudades de un territorio. El objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades intermedias y minimice la distancia recorrida por el viajante.

Básicamente hay que encontrar una permutación del conjunto de ciudades  $P = \{c_0, \dots, c_N\}$  tal que la suma de las distancias entre una ciudad y la siguiente sea mínimo, es decir  $\sum_{i:0..N-1} d[c_i, c_{(i+1)\%N}]$  sea mínimo. La distancia  $d$  entre dos ciudades viene dada en una matriz. Para indicar la ausencia de conexión entre dos ciudades se puede usar un valor especial en la matriz distancias. En el código utilizaremos una función booleana `hayArista` que nos indica si hay conexión entre dos ciudades dadas.

El tamaño del árbol de soluciones es  $(N - 1)!$ , ya que corresponde a todas las posibles permutaciones, teniendo en cuenta que el principio y el final es el mismo.

El problema tiene considerables aplicaciones prácticas, aparte de las más evidentes en áreas de logística de transporte. Por ejemplo, en robótica, permite resolver problemas de fabricación para minimizar el número de desplazamientos al realizar una serie de perforaciones en una plancha o en un circuito impreso. También puede ser utilizado en control y operativa optimizada de semáforos, etc.

Veamos cómo es la solución para este problema.

---

```
void viajante(int distancias[N][N], int solucion[], int &coste, int k,
              int n, int solucionMejor[], int &costeMejor, bool usadas[]) {
    //para evitar soluciones repetidas fijamos como ciudad de comienzo la 0,
    //por lo que no se va a volver a considerar
    for(int i = 1; i < n; i++) {
        solucion[k] = i;
        if (esValida(distancias,solucion, k,usadas)) {
            coste += distancias[solucion[k-1]][solucion[k]];
            usadas[i]=true;
```

---

---

```

        if (k==n-1) {
            if hayArista(distancias,solucion[k],solucion[0])
            {
                if(coste +distancias[k][0]< costeMejor){
                    costeMejor = coste+distancias[k][0];
                    copiarSolucion(solucion, solucionMejor);
                }
            }
            else viajante(distancias, solucion, coste, k+1, n,
                            solucionMejor, costeMejor);
            usadas[i]=false;
            coste -= distancias[solucion[k-1]][solucion[k]];
        }
    }
}

```

---

Para mejorar el coste de la función `esValida` se utiliza la técnica de marcaje. En este caso, se declara un vector `usadas` de  $n$  componentes, donde el valor de cada componente indica si la ciudad correspondiente ha sido visitada.

---

```

bool esValida(int distancias[N][N], int solucion[], int k, bool usadas[])
{
    return (hayArista(distancias,solucion[k-1],solucion[k])
             && !usadas[solucion[k]]);

}

```

---

Llevamos dos parámetros en los que guardamos la mejor solución hasta el momento y el coste de la misma (*solucionMejor* y *costeMejor*). Cuando encontramos una solución comprobaremos si es mejor que la que tenemos almacenada, en caso positivo la consideraremos como la nueva mejor solución. El algoritmo de *vuelta atrás* garantiza que al final de la búsqueda la solución encontrada tiene el coste óptimo. El parámetro *coste* acumula el coste de la solución parcial, que se va calculando de forma incremental en cada llamada recursiva, evitando realizar el cálculo en cada llamada. Al finalizar la llamada recursiva debe actualizar su valor, igual que se hace en la técnica de marcaje.

- Podríamos mejorar la búsqueda del camino óptimo utilizando una estimación optimista para realizar podas tempranas. La idea es prever cuál es el mínimo coste de lo que falta por recorrer. Si ese coste, sumado al que llevamos acumulado, supera la mejor solución encontrada hasta el momento, entonces podemos abandonar la búsqueda porque estamos seguros de que ningún camino va a mejorar la solución. Utilizamos una estimación optimista, cuanto menos optimista mejor, para saber si es posible mejorar el resultado.
- Para calcular una estimación optimista es suficiente con encontrar la mínima distancia entre cualquier par de ciudades, y considerar que todos los desplazamientos van a tener esa distancia. Se pueden realizar cálculos más ajustados del coste del camino que queda por recorrer, pero hay que tener en cuenta que el cálculo debe ser sencillo para no aumentar el coste del algoritmo.

Así tendremos que añadir antes de realizar la llamada recursiva, el cálculo del coste estimado. Se calcula considerando que el resto de desplazamientos tienen un coste mínimo. El *costeMinimo* se puede calcular muy fácilmente recorriendo la matriz de distancias;

consideramos que se ha procesado al principio de la ejecución y que lo tenemos almacenado en un parámetro. Solo realizaremos la recursión si la solución se puede mejorar.

---

```
int costeEstimado = coste + (n - k + 1) * costeMinimo;
if(costeEstimado < costeMejor)
    viajante(distancias, solucion, coste, k+1, n, solucionMejor, costeMejor,
              usadas, costeMinimo);
```

---

En la llamada inicial se debe fijar una ciudad de comienzo para evitar soluciones repetidas:

---

```
costeMinimo=calcularMinimo(distancias);
solucion[0]=0;usadas[0]=true;
for (int i=1;i<n;i++) {usadas[i]=false;};
coste=0;
costeMejor = ...
//una cota superior, como por ejemplo la suma de todas las aristas del grafo
viajante(distancias,solucion,coste,1,n,solucionMejor,costeMejor,usadas,costeMinimo);
```

---

## 7.2. Ejemplo: Problema de la mochila

Otro problema clásico de optimización es el problema de la mochila. La idea es que tenemos  $n$  objetos con valor  $(v_0, \dots, v_{n-1})$  y peso  $(p_0, \dots, p_{n-1})$ , y tenemos que determinar qué objetos transportar en la mochila sin superar su capacidad  $m$  (en peso) para maximizar el valor del contenido de la mochila.

Para resolver este problema en cada nivel del árbol de búsqueda vamos a decidir si cogemos o no el  $i$ -ésimo elemento. Así pues la solución será una tupla  $(b_0, \dots, b_{n-1})$  de booleanos.

Se consideran las siguientes restricciones:

- Deberemos maximizar el valor de lo que llevamos  $\sum_{i:0..n-1} b_i v_i$ .
- El peso no debe exceder el máximo permitido  $\sum_{i:0..n-1} b_i p_i \leq m$ .

Una posible solución es:

---

```
void mochila(float P[], float V[], bool solucion[], int k, int n, int m,
              float &peso, float &beneficio, int solucionMejor[], int &valorMejor) {
    // hijo izquierdo [cogemos el objeto]
    solucion[k] = true;
    peso = peso + P[k];
    beneficio = beneficio + V[k];
    if(peso <= m) {
        if(k == n-1) {
            if(valorMejor < beneficio) {
                valorMejor = beneficio;
                copiarSolucion(solucion, solucionMejor);
            }
        }
        else{
            mochila(P,V,solucion, k+1,n, m, peso, beneficio,
                    solucionMejor, valorMejor);
        }
    }
}
```

---

```

peso = peso - P[k];           //desmarcamos peso y beneficio
beneficio = beneficio - V[k];
// hijo derecho [no cogemos el objeto]
solucion[k] = false;
if(k == n-1){
    if(valorMejor < beneficio)){
        valorMejor = beneficio;
        copiarSolucion(solucion, solucionMejor);
    }
}
else{
    mochila(P,V,solucion, k+1,n, m,peso, beneficio,
            solucionMejor, valorMejor);
}
}

```

---

En el hijo de la derecha no tendremos que comprobar si excedemos el peso total, ya que al descartar el objeto no aumentamos el peso acumulado.

Este problema da pie a optimización, ya que podemos calcular una cota superior (una evaluación optimista) del beneficio que podemos obtener con lo que nos resta para llenar la mochila. Para calcularla, organizamos inicialmente los objetos en los vectores  $P$  y  $V$  de manera que estén ordenados por “densidad de valor” decreciente. Llamamos densidad de valor al cociente  $v_i/p_i$ . De esta forma, cogeremos primero los objetos que tienen más valor por unidad de peso. Si en un cierto nodo, la tupla parcial es  $(b_0, \dots, b_k)$  y hemos alcanzado un beneficio  $beneficio$  y un peso  $peso$ , estimamos el beneficio optimista como la suma de  $beneficio$  más el beneficio conseguido cogiendo los objetos que quepan en el orden indicado desde el  $k + 1$  al  $n - 1$ . Si se llega a un objeto  $j$  que ya no cabe, se fracciona y se suma el valor de la fracción que quepa. Esta forma de proceder se llama solución *voraz* al problema de la mochila con posible fraccionamiento de objetos y produce siempre una cota superior a cualquier solución donde no se permita fraccionamiento.

La poda se produce si el beneficio optimista es **menor** que el beneficio de la mejor solución alcanzada hasta el momento.

## 8. Para terminar...

Terminamos el tema con la solución al problema del coloreado de mapas planteado en la primera sección. Como en los casos anteriores, lo primero que nos debemos preguntar es sobre la forma de la solución y del árbol de búsqueda.

En este caso:

- Si el mapa  $M$  tiene  $m$  países, numerados 0 a  $m - 1$ , entonces la solución va a ser una tupla  $(x_0, \dots, x_{m-1})$  donde  $x_i$  es el color asignado al  $i$ -ésimo país.
- Cada elemento  $x_i$  de la tupla pertenecerá al conjunto  $\{0, \dots, n - 1\}$  de colores válidos.

Cada vez que vayamos a pintar un país de un color tendremos que comprobar que ninguno de los adyacentes está pintado con el mismo color. En este caso es más sencillo hacer la comprobación cada vez que coloreamos un vértice en lugar de utilizar *marcaje*.

Así pues la función principal será:

---

```

void colorear(int solucion[], int k, int n, int m) {
    for(int c = 0; c < n; c++) {
        solucion[k] = c;
        if(esValida(solucion, k)) {
            if(esSolucion(k,m)) {
                tratarSolucion(solucion,m);
            }
            else{
                colorear(solucion, k + 1, n,m);
            }
        }
    }
}

```

---

La función `esValida` será la encargada de comprobar si la solución parcial no vulnera la restricciones de que dos países limítrofes compartan color; para implementarla asumiremos que tenemos acceso a cierto objeto `M` donde se guarda el mapa, y que tiene un método que dice si dos países son fronterizos.

---

```

bool esValida(int solucion[], int k) {
    int i = 0; bool valida = true;
    while (i < k && valida) {
        if (M.hayFrontera(i, k) && solucion[k] == solucion[i])
            valida = false;
        i++;
    }
    return valida;
}

```

---

A la hora de hacer la llamada inicial podemos asignar al primer país del mapa un color arbitrario.

## Notas bibliográficas

Gran parte del contenido de este capítulo está basado en el capítulo correspondiente de (Martí Oliet et al., 2013).

## Ejercicios

1. Vamos a realizar una modificación en el problema de las n-reinas. Asumimos que cada casilla del tablero de ajedrez tiene un número asignado que representa el valor de esa casilla.

Se pide hacer un algoritmo que resuelva el problema de las n-reinas, de tal manera que la suma de los números de las posiciones donde se colocan las reinas sea máxima.

2. Dado un número entero  $M$  y un vector de  $n$  números naturales  $V[0..n-1]$ , determinar si existe una forma de insertar entre los  $n$  números del vector (tal como están colocados en el vector) operaciones de suma y resta de forma que se obtenga el número  $M$  como resultado final.

El programa debe determinar si es posible o no realizar la expresión, y devolver la expresión oportuna en el caso afirmativo.

3. A partir de un tablero de ajedrez de  $n \times n$  posiciones y de un caballo situado en una posición arbitraria  $(i, j)$  se pide diseñar un algoritmo que determine un secuencia de movimientos que visite todas las casillas del tablero una sola vez. El último movimiento debe devolver el caballo a su posición inicial.
4. Optimizar el problema de la mochila descrito más arriba. Para ello podemos utilizar la siguiente idea para realizar una estimación optimista:
  - Consideramos que tenemos los objetos ordenados, los que tienen mayor valor por unidad de peso son los primeros.
  - Asumimos que los objetos son fraccionables; que podemos coger una determinada parte de cada uno.

Con estas dos consideraciones podemos realizar una algoritmo de estimación optimista muy sencillo, que siempre escoja una solución óptima, que nunca podrá ser superada por objetos nos fraccionables. Este algoritmo se puede utilizar como cota superior.

5. Encontrar  $n$  puntos del eje real a partir de las  $n(n-1)/2$  distancias (no necesariamente diferentes) entre cada par de puntos, sabiendo que el menor de dichos puntos es el origen y que el multiconjunto de las distancias viene dado en orden creciente.

Ejemplo: si el multiconjunto de distancias es  $\{1,2,2,2,3,3,3,4,5,5,5,6,7,8,10\}$ , una solución válida es  $\{0,3,5,6,8,10\}$ .

6. Deseamos decorar una pared de  $L$  metros de ancho. Hemos tenido la innovadora idea de hacerlo colgando una hilera de cuadros pegados lado con lado. Nos disponemos a comprar los cuadros en la feria de arte moderno, donde tenemos la posibilidad de elegir entre  $n$  cuadros. Cada cuadro tiene un prestigio  $p_i$ , y unas dimensiones de  $a_i$  metros de alto por  $b_i$  metros de ancho,  $1 \leq i \leq n$ . Dado lo peculiar de los cuadros, podemos elegir colgar cada cuadro tanto en horizontal como en vertical sin que por ello se vea afectado su prestigio. Lo que no podemos hacer es trocear un cuadro. Diseñar un algoritmo que determine qué cuadros comprar de forma que la longitud de la hilera de cuadros sume exactamente  $L$  metros y se maximice el prestigio acumulado en la pared.
7. [Examen Junio, 2012] Implementar una función que encuentre la forma *más rápida* de viajar desde una casilla de salida hasta una casilla de llegada de una rejilla. Cada casilla de la rejilla está etiquetada con una letra, de forma que en el camino desde la salida hacia la llegada se debe ir formando (de forma cíclica) una palabra dada. Desde una celda se puede ir a cualquiera de las cuatro celdas adyacentes.

Como ejemplo, a continuación aparece la forma más corta de salir de una rejilla de  $5 \times 8$  en la que el punto de salida está situado en la posición  $(0, 4)$  y hay que llegar a la posición  $(7, 0)$  y la palabra que hay que ir formando por el camino es EDA.

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 0 | M | D | A | A | E | E | D | A |
| 1 | A | E | E | D | D | A | N | D |
| 2 | D | B | D | X | E | D | A | E |
| 3 | E | A | E | D | A | R | T | D |
| 4 | E | D | M | P | L | E | D | A |
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

El tablero tendrá un tamaño  $N \times M$  con  $N > 0$  y  $M > 0$ .

Las función recibirá el punto origen, el punto destino y la palabra a formar y deberá determinar la forma más rápida de viajar de uno a otro (si es que esto es posible), dando las direcciones que hay que ir cogiendo (en el caso del ejemplo será E, N, E, E, E, etc.). Si necesitas parámetros adicionales, añádelos indicando sus valores iniciales.

- [Examen Septiembre, 2012] Los ferrys son barcos que sirven para trasladar coches de una orilla a otra de un río. Los coches esperan en fila al ferry y cuando éste llega un operario les va dejando entrar.

En nuestro caso el ferry tiene espacio para dos filas de coches (la fila de babor y la fila de estribor) cada una de  $N$  metros. El operario conoce de antemano la longitud de cada uno de los coches que están esperando a entrar en él y debe, según van llegando, mandarles a la fila de babor o a la fila de estribor, de forma que el ferry quede lo más cargado posible. Ten en cuenta que los coches deben entrar en el ferry **en el orden en que aparecen** en la fila de espera, por lo que en el momento en que un coche ya no entra en el ferry porque no hay hueco suficiente para él, no puede entrar ningún otro coche que esté detrás aunque sea más pequeño y sí hubiera hueco para él.

Implementa una función que reciba la capacidad del ferry en metros y la colección con las longitudes de los coches que están esperando (puedes utilizar el TAD que mejor te venga) y devuelva la colocación óptima de los coches, entendiendo ésta como la secuencia de filas en las que se van metiendo los coches. Supón la existencia de los símbolos BABOR y ESTRIBOR.

*Ejemplo:* Si el ferry tiene 5 metros de longitud y en la fila tenemos coches con longitudes (del primero al último) 2.5, 3, 1, 1, 1.5, 0.7 y 0.8, una solución óptima es [BABOR, ESTRIBOR, ESTRIBOR, ESTRIBOR, BABOR, BABOR].

- [Examen Febrero, 2014] Deseamos organizar un festival de rock al aire libre para lo cual vamos a contratar exactamente a  $N$  artistas de entre  $M$  disponibles ( $N < M$ ). No todos los artistas aceptan tocar juntos en el festival. Los “vetos” entre artistas son conocidos de antemano. Para cada artista  $i \in \{1..M\}$  conocemos el beneficio o pérdida generado por dicho artista  $B[i]$ , es decir si  $B[i] > 0$ , dicho artista genera beneficio mientras que si  $B[i] < 0$  genera pérdida. Diseñar un algoritmo de vuelta atrás que resuelva el problema de planificar el festival que maximice la suma de los beneficios/pérdidas de los artistas participantes.
- [Examen Junio, 2014] Alonso Rodríguez tiene que hacer la compra de la semana. Ha hecho una lista de  $n$  productos que quiere comprar. En su barrio hay  $m$  supermercados en cada uno de los cuales se dispone de todos esos productos. Pero como es un

comprador compulsivo no quiere comprar más de tres productos en cada uno de los supermercados ya que así pasa más tiempo comprando (se puede suponer que  $n \leq 3m$ ). Dispone de una lista de precios de los productos en cada uno de los supermercados. Se pide diseñar un algoritmo de vuelta atrás que permita a Alonso decidir cómo hacer la compra de forma que compre todo lo que necesita y que el coste total sea mínimo.

---

Capítulo 4

# Implementación y uso de TADs<sup>1</sup>

---

*La perfección no se obtiene  
cuando se ha añadido todo lo añadible,  
sino cuando se ha quitado todo lo superfluo.*

Antoine de Saint-Exupéry (Escritor francés,  
1900-1944)

**RESUMEN:** En este tema se introducen los **tipos abstractos de datos** (TADs), que permiten definir y abstraer tipos de forma similar a como las funciones definen y abstraen código. Presentaremos la forma de definir un TAD, incluyendo tanto su especificación externa como su representación interna, y veremos, como caso de estudio, el TAD “iterador”.

## 1. Motivación

Te plantean el siguiente problema: Dado un número  $x$ , se cogen sus dígitos y se suman sus cuadrados, para dar  $x_1$ . Se realiza la misma operación, para dar  $x_2$ , y así mucho rato hasta que ocurra una de las dos cosas siguientes:

- se llega a 1, y se dice entonces que el número es “feliz”
- nunca se llega a 1 (porque se entra en un ciclo que no incluye el 1), y se dice entonces que el número es “infeliz”

Ejemplos:

- el 7 es feliz:  $7 \rightarrow 49 \rightarrow 97 \rightarrow 130 \rightarrow 10 \rightarrow 1$
- el 38 es infeliz:  $38 \rightarrow 73 \rightarrow 58 \rightarrow 89 \rightarrow 145 \rightarrow 42 \rightarrow 20 \rightarrow 4 \rightarrow 16 \rightarrow 37 \rightarrow 58$

Sin estructuras ni TADs, necesitarás bastante imaginación para encontrar una solución (existe, eso sí). Al final de este capítulo verás una solución mucho más legible, usando un TAD Conjunto.

---

<sup>1</sup>Manuel Freire es el autor principal de este tema.

## 2. Introducción

### 2.1. Abstracción

- ★ El exceso de detalles hace más difícil tratar con los problemas. Para concentrarnos en lo realmente importante, abstraemos el problema, eliminando todos los detalles innecesarios.
- ★ Por ejemplo, para leer un fichero de configuración desde un programa, el programador no tiene que preocuparse del formato del sistema de archivos, ni de su soporte físico concreto (si se trata de una unidad en red, una memoria USB o un disco que gira miles de veces por segundo) - en lugar de eso, las librerías del sistema y el sistema operativo le permiten usar la abstracción “fichero” – una entidad de la que se pueden leer y a la que se pueden escribir bytes, y que se identifica mediante una ruta y un nombre. Un fichero es un ejemplo de tipo abstracto de datos (TAD).
- ★ Se puede hablar de dos grandes tipos de abstracción:
  - funcional: abstrae una *operación*. Es la que hemos visto hasta ahora. La operación se divide en “especificación” (qué es lo que hago) e “implementación” (cómo lo llevo a cabo). De esta forma, un programador puede llamar a `ordena(v, n)` con la plena confianza de que obtendrá un vector ordenado en  $O(n \log n)$ , y sin tener que preocuparse por los detalles de implementación del algoritmo concreto.
  - de datos: abstrae un *tipo de dato*, o mejor dicho, el uso que se puede hacer de este tipo (por ejemplo, un fichero) de la forma en que está implementado internamente (por ejemplo, bloques de un disco magnetizado estructurados como un sistema de archivos NTFS). Si la abstracción funcional es una forma de *añadir operaciones* a un lenguaje, la abstracción de datos se puede ver como una forma de *añadir tipos* al lenguaje.

### TADs predefinidos

- ★ Algunos tipos de TADs vienen predefinidos con los lenguajes. Por ejemplo, enteros, caracteres, números en coma flotante, booleanos o arrays tienen todas representaciones internas “opacas” que no es necesario conocer para poderlos manipular. Por ejemplo:
  - los enteros (`char`, `int`, `long` y derivados) usan, internamente, representación binaria en complemento a 2. Las operaciones `+`, `-`, `*`, `/`, `%` (entre otras) vienen predefinidas, y en general no es necesario preocuparse por ellas
  - los números en coma flotante (`float` y `double`) usan una representación binaria compuesta por mantisa y exponente, y disponen de las operaciones que cabría esperar; en general, los programadores evitan recurrir a la representación interna.
  - los vectores (en el sentido de *arrays*) tienen su propia representación interna y semántica externa. Por ejemplo, `v[i]` se refiere a la  $i$ -ésima posición, y es posible tanto escribir como leer de esta posición, sin necesidad de pensar en cómo están estructurados los datos físicamente en la memoria.

- ★ Aunque, en general, los TADs se comportan de formas completamente esperadas (siguiendo el *principio de la mínima sorpresa*), a veces es importante tener en cuenta los detalles. Por ejemplo, este código demuestra una de las razones por las cuales los `float` de C++ no se pueden tratar igual que los `int`: sus dominios se solapan, pero a partir de cierto valor, son cada vez más distintos.

---

```
int ejemplo() {
    int i=0;      // 31 bits en complemento a 2 + bit de signo
    float f=0;    // 24 bits en complemento a 2 + 7 de exponente + signo
    while (i == f) { i++; f++; } // int(224+1) ≠ float(224+1) (!)
    return i;
}
```

---

los dominios de `float` e `int` no son equivalentes

- ★ Este otro fragmento calcula, de forma aproximada, la inversa de la raíz cuadrada de un número  $n$  - abusando para ello de la codificación de los enteros y de los flotantes en C/C++. Es famosa por su uso en el videojuego *Quake*, y aproxima muy bien (y de forma más rápida que `pow`) el resultado de `pow(n, .5)`. Los casos en los que está justificado recurrir a este tipo de optimizaciones son *muy* escasos; pero siempre que se realiza una abstracción que no permite acceso a la implementación subyacente, se sacrifican algunas optimizaciones (como ésta, y a menudo más sencillas) en el proceso.

---

```
float Q_rsqrt(float n) {
    const float threehalfs = 1.5f;
    float x2 = n * 0.5f;
    float y = n;
    int i = *(long*)&y;           // convierte de float a long (!)
    i = 0x5f3759df - (i >> 1);
    y = *(float*)&i;            // convierte de long a float (!)
    y = y * (threehalfs - (x2 * y * y));
    return y;
}
```

---

optimización que rompe la encapsulación de los tipos `float` e `int`

- ★ Un ejemplo clásico de TAD predefinido de C++ es la cadena de caracteres de su librería estándar: `std::string`. Dada una cadena `s`, es posible saber su longitud (`s.length()` o `s.size()`), concatenarle otras cadenas (`s += t`), sacar copias (`t = s`), o mostrarla por pantalla (`std::cout << s`), entre otras muchas operaciones – y todo ello sin necesidad de saber cómo reserva la memoria necesaria ni cómo codifica sus caracteres.

## 2.2. TADs de usuario

- ★ Los `structs` de C, o los `records` de Pascal, sólo permiten definir nuevos tipos de datos, pero no permiten ocultar los detalles al usuario (el programador que hace uso de ellos). Esta ocultación sí es posible en C++ u otros lenguajes que soportan orientación a objetos.
- ★ Un tipo de datos Fecha muy básico, sin ocultación o abstracción alguna, sería el siguiente:

---

```
struct Fecha {
    int dia;
    int mes;
    int anyo; // identificadores en C++: sólo ASCII estándar
    Fecha(int d, int m, int a): dia(d), mes(m), anyo(a) {}
};
```

---

estructura Fecha. Es fácil generar fechas inválidas.

- \* Ahora, es posible escribir Fecha f = Fecha(14, 7, 1789) o Fecha f(14, 7, 1789), y acceder a los campos de esta fecha mediante f.dia o f.mes: hemos definido un nuevo tipo, que no existía antes en el lenguaje. Por el lado malo, es posible crear fechas inconsistentes; por ejemplo, nada impide escribir f.mes = 13 o, más sutil, f.mes = 2; f.dia = 30 (febrero nunca puede tener 30 días). Es decir, tal y como está escrita, es fácil salirse fuera del *dominio* de las fechas válidas.
- \* Además, esta Fecha es poco útil. No incluye operaciones para sumar o restar días a una fecha, ni para calcular en qué día de la semana cae una fecha dada, por citar dos operaciones frecuentes. Tal y como está, cada programador que la use tendrá que (re)implementar el mismo conjunto de operaciones básicas para poder usarlas, con el consiguiente coste en tiempo y esfuerzo; y es altamente probable que dos implementaciones distintas sean incompatibles entre sí (en el sentido de distintas precondiciones/postcondiciones). Una buena especificación de TAD debería incluir las *operaciones* básicas del tipo, facilitando así su uso estándar.
- \* Todo TAD especifica una serie de **operaciones** del tipo; estas operaciones son las que el diseñador del tipo prevee que se van a querer usar con él. Pueden hacer uso de otros tipos; por ejemplo, un tipo Tablero podría usar un tipo Ficha y un tipo Posicion en su operación mueve.

### Ejemplo expandido: Fechas

Este ejemplo presenta un tipo Fecha<sup>2</sup> algo más completo, donde se especifica (de forma informal) el dominio de una implementación concreta y una serie de operaciones básicas.

Nota: en este ejemplo se usa notación de C++ “orientado a objetos”. El lector que no sea familiar con esta notación debe entender que, en todas las operaciones de una clase (excepto los constructores), se está pasando un parámetro adicional implícito (*this*): el objeto del tipo que se está definiendo (en este caso una Fecha) sobre el que operar. De esta forma, dada una Fecha f, la instrucción f.dia() se convierte internamente a dia(*this=f*).

---

<sup>2</sup> Para ver un ejemplo real (y muy, muy flexible) de TAD fecha para su uso en C++, [http://www.boost.org/doc/libs/1\\_48\\_0/doc/html/date\\_time.html](http://www.boost.org/doc/libs/1_48_0/doc/html/date_time.html). También hay una discusion de Fecha más instructiva en la sección 10.2 de Stroustrup (1998).

---

```

class Fecha {
    public: // parte pública del TAD
        // Constructor
        // dia, mes y año forman una fecha en el dominio esperado
        Fecha(int dia, int mes, int anyo);

        // Constructor alternativo
        // dias es un número de días a sumar o restar a la fecha base
        Fecha(const Fecha &base, int dias);

        // distancia , en días , de la fecha actual con la dada (Obs.)
        int distancia(const Fecha &otra) const;

        // devuelve el dia de esta fecha (Obs.)
        int dia() const;

        // devuelve el día de la semana de esta fecha (Observadora)
        int diaSemana() const;

        // devuelve el mes de esta fecha (Observadora)
        int mes() const;

        // devuelve el año de esta fecha (Observadora)
        int anyo() const;

    private: // parte privada , accesible solo para la implementación
        int _dia;      // entre 1 y 28,29,30,31, en función de mes y año
        int _mes;     // entre 1 y 12, ambos inclusive
        int _anyo;
}

```

---

clase Fecha, con operaciones para acceder y modificar fechas de forma controlada

- \* La **parte privada** de la clase determina la estructura interna de los objetos del tipo, y es la que limita el dominio. Con la estructura interna de Fecha, podríamos ampliar el dominio a cualquier año representable con un entero de 32 bits. Puede contener operaciones auxiliares, que por su posibilidad de romper y/o complicar innecesariamente el tipo, no se quieren exponer en la parte pública.
  - `int diasDesde1AD()` facilitaría mucho la implementación de `distancia()`, pero se podría ver como una complicación innecesaria del tipo (ya que no tendría mucho sentido fuera de esta operación). Por tanto, se podría declarar como privada.
- \* La **parte pública** corresponde a las operaciones que se pueden usar externamente, y es completamente independiente (desde el punto de vista del usuario) de la parte privada. Las operaciones públicas del tipo constituyen un *contrato* entre el autor del TAD y estos usuarios. En tanto en cuanto el autor del TAD se limite a modificar la parte privada (y las implementaciones de la parte pública), y los usuarios se limiten a usar las operaciones de la parte pública, el contrato seguirá vigente, y todo el código escrito contra el contrato seguirá funcionando. También es frecuente referirse a la parte pública de un tipo o módulo como su **API** (*Application Programming Interface*).

- ★ Cada lenguaje de programación ofrece unas ciertas facilidades para la realización de este tipo de contratos entre implementador y usuario. Para que el usuario no acceda a la parte privada, el lenguaje puede ofrecer
  - **privacidad:** los detalles de implementación quedan completamente ocultos e inaccesibles para el usuario. Este es el caso, por ejemplo, de las `Interface` (interfaces) de Java. Es posible simularlo<sup>3</sup> con C++, pero no forma, estrictamente hablando, parte del lenguaje.
  - **protección:** intentar acceder a partes privadas de TADs produce errores de compilación. Tanto C++ como Java tienen este tipo de controles, que se regulan mediante las notaciones `public:` o `private:` en los propios TADs.
  - **convención:** si no hay ningún otro mecanismo, solo queda alcanzar un “acuerdo entre caballeros” entre usuario e implementador para no tocar aquello marcado como privado. Una marca típica es comenzar identificadores privados con un ‘`_`’, escribiendo, por ejemplo, `_día`'. Esto ya no es necesario en C++ (aunque sigue siendo útil como recordatorio), pero sí lo era en C.
- ★ Con la versión actual de la parte privada, será muy fácil implementar `dia()` o `mes()`, pero harán falta más cálculos en `distancia()`, `diaSemana()` y `diaAnyo()`. Podríamos haber elegido una implementación que almacenase solo `_diaAnyo` y `_anyo`. Esto facilitaría calcular `distancia()` entre fechas del mismo año y haría muy sencillo implementar `diaAnyo()`, pero complicaría `dia()` o `mes()`. En cualquier caso, el usuario del TAD no sería capaz de ver la diferencia, ya que la forma de llamar a las operaciones externas (la parte pública) no habría cambiado. En general, todo TAD contiene decisiones de implementación que deberían tener en cuenta los casos de uso a los que va a ser sometido, y sus frecuencias relativas.

### 2.3. Diseño con TADs

- ★ El uso de TADs simplifica el desarrollo, ya que establece una diferenciación clara entre las partes importantes de un tipo (sus operaciones públicas) y aquellas que no lo son (la implementación interna). Por tanto, *disminuye la carga cognitiva* del desarrollador.
- ★ Al igual que la abstracción funcional, la abstracción de tipos es vital para la *reutilización*: un único TAD se puede usar múltiples veces (para eso se escribe). Además, es más fácil reutilizar un código que usa TADs: será más compacto y legible que un código equivalente que no los use, y en general los TADs están (y deben estar) mucho mejor documentados que otros tipos de código.
- ★ Además, facilita el *diseño descendente*: empezando por el máximo nivel de abstracción, ir refinando conceptos y tipos hasta alcanzar el nivel más bajo, el de implementación. Y también el *diseño ascendente*: empezar por las abstracciones más sencillas, e ir construyendo abstracciones más ambiciosas a partir de éstas, hasta llegar a la abstracción que resume todo el programa. En ambos casos, el programador puede concentrarse sobre lo importante (cómo se usan las cosas) y olvidar lo accesorio (cómo

<sup>3</sup>mediante la técnica de “puntero a implementación”, abreviada generalmente a “`pimpl`”: un campo privado `void *pimpl;` contiene y oculta toda la implementación. Sólo el desarrollador de la librería tiene que preocuparse de trabajar con el contenido de ese puntero para cumplir la especificación.

están implementadas internamente – o incluso, cómo se van a implementar, cuando llegue el momento de hacerlo).

- Ejemplo de diseño descendente: Quiero programar un juego de ajedrez. Empiezo definiendo un Tablero con sus operaciones básicas: generar jugadas, realizar una jugada, ver si ha acabado la partida, guardar/cargar el tablero, etcétera. De camino, empiezo a definir lo que es una Jugada (algo que describe cómo se mueve una pieza sobre un tablero), una Posicion (un lugar en un tablero) y una Pieza (hay varios tipos; cada una tiene Jugadas distintas). Esta metodología permite decidir qué operaciones son importantes, pero no permite hacer pruebas hasta que todo está bastante avanzado.
- Ejemplo de diseño ascendente: Quiero programar un juego de ajedrez. Empiezo definiendo una Posicion y una Jugada (que tiene posiciones inicial y final). Añado varios tipos de Pieza, cada una de las cuales genera jugadas distintas. Finalmente, junto todo en un Tablero. Esta metodología permite ir haciendo pruebas a medida que se van implementando los tipos; pero dificulta la correcta elección de las operaciones a implementar, ya que se tarda un tiempo en adquirir una visión general.
- \* En la práctica, se suele usar una combinación de ambos métodos de diseño; el método *top-down* (descendente) para las especificaciones iniciales de cada tipo, y el *bottom-up* (ascendente) para la implementación real y pruebas, convergiendo en la estrategia *meet-in-the-middle*.

## TADs y módulos

- \* Todos los lenguajes incluyen el concepto de *módulo* - un conjunto de datos y operaciones fuertemente relacionadas que, externamente, se pueden ver como una unidad, y a las cuales se accede mediante una interfaz bien definida. El concepto de módulo incluye de forma natural a los TADs, y por tanto, *en general, los TADs se deben implementar como módulos*.
- \* En C++, esto implica la *declaración* de una clase en un fichero .h (por ejemplo, fecha.h), y su *implementación* en un fichero .cpp asociado (por ejemplo, fecha.cpp):

```
// includes imprescindibles para que compile el .h
#include <libreria_sistema>
#include "modulo_propio.h"

// protección contra inclusión múltiple
#ifndef _FECHA_H_
#define _FECHA_H_

// ... declaración de la clase Fecha aquí ...

// fin de la protección contra inclusión múltiple
#endif
```

---

declaración del módulo fecha, en fecha.h

---

```

#include "fecha.h"

// includes adicionales para que compile el .cpp
#include <libreria_sistema>
#include "modulo_propio.h"

// ... definicion de Fecha aqui ...

```

---

su implementación, en fecha.cpp

- ★ En general, la documentación de un módulo C++ se escribe en su .h. No es recomendable duplicar la documentación del .h en el .cpp, aunque sigue siendo importante mantener comentarios con versiones reducidas de las cabeceras de las funciones, como separación visual y para añadir comentarios “de implementación” dirigidos a quienes quieran mantener o consultar esta implementación.
- ★ En un módulo C++ más general, el .h podría contener más de una declaración de clase, o mezclar declaraciones de tipos con prototipos de funciones.
- ★ El diseño modular y la definición de TADs están fuertemente relacionados on el diseño orientado a objetos (OO). En la metodología OO, el programa entero se estructura en función de sus tipos de datos (llamados *objetos*), que pasan así al primer plano. Por el contrario, en un diseño imperativo tradicional, el énfasis está en las *acciones*.
- ★ C++ soporta ambos tipos de paradigmas, tanto OO como imperativo, y es perfectamente posible (y frecuente) mezclar ambos. Un módulo OO estará caracterizado por clases (objetos) que contienen acciones, mientras que un módulo imperativo contendrá sólo funciones que manipulan tipos.

### TADs y estructuras de datos

- ★ Un TAD está formado por una colección de valores y un conjunto de operaciones sobre dichos valores.
- ★ Una **estructura de datos** es una estrategia de almacenamiento en memoria de la información que se desea guardar. Muchos TADs se implementan utilizando estructuras de datos. Por ejemplo, los TADS pilas y colas pueden implementarse usando la estructura de datos de las listas enlazadas.
- ★ Algunos TADs son de uso tan extendido y frecuente, que es de esperar que en cualquier lenguaje de programación de importancia exista una implementación. Por ejemplo, las librerías estándares de C++, Java o Delphi soportan, todas ellos, árboles de búsqueda o tablas hash.
- ★ Hay estructuras que solo se usan en dominios reducidos. Por ejemplo, ninguna de estas librerías estándares soporta el TAD *quadtree*, muy popular en sistemas de información geográfica, y que permite la búsqueda del punto más cercano a otro dado en tiempo logarítmico.
- ★ La importancia de estudiar estructuras de datos radica no solamente en los detalles de su implementación (aunque sean buenos ejemplos de programación), sino en sus *características generales* incluyendo los costes de las operaciones que proporcionan,

que es imprescindible conocer si se quiere poder tomar decisiones acerca de cuándo preferir una estructura a otra.

### 3. Implementación de TADs

- \* En este apartado se analizan con mayor detalle cómo se debe programar los TADs en C++. C++ es un lenguaje tremadamente flexible y poderoso; esto quiere decir que hay más de una forma de hacer las cosas, y las desventajas de ciertas implementaciones pueden no ser aparentes. El contenido de este apartado sigue las “mejores prácticas” reconocidas en la industria; úsalo como referencia cuando hagas tus propias implementaciones.

#### 3.1. Tipos de operaciones y el modificador *const*

- \* Las operaciones se pueden clasificar como *generadoras*, *modificadoras* u *observadoras*.

**Generador** Crea una nueva instancia del tipo (puede o no partir de otras instancias anteriores). Usando operaciones generadoras, es posible construir todos los valores posibles del tipo.

**Modificador** Igual que las generadoras, pero no pertenece al “conjunto mínimo” que permite construir los valores del tipo.

**Observador** Permite acceder a aspectos del tipo principal, codificados en otros tipos - pero *no modifica* el tipo al que observa (y, por tanto, su declaración lleva el calificativo *const*). Por ejemplo, conversión a cadena, obtención del día-del-mes de una Fecha como entero, etcétera.

- \* La **clasificación que usaremos** en la asignatura es equivalente, pero está mejor adaptada a lenguajes orientados a objetos:

**Constructor** Crea una nueva instancia del tipo. En C++, un constructor se llama siempre como el tipo que construye (con los argumentos que se deseé utilizar). Se llaman automáticamente cuando se declara una nueva instancia del tipo. Si no hay nada que inicializar, se pueden omitir.

**Mutador** Modifica la instancia actual del tipo. En C++, *no pueden* llevar el modificador *const* al final de su definición.

**Observador** No modifican la instancia actual del tipo. En C++, *deben* llevar el modificador *const* (aunque el compilador no genera errores si se omite).

**Destructor** Destruye una instancia del tipo, liberando cualquier recurso que se haya reservado en el momento de crearla (por ejemplo, cerrando ficheros, liberando memoria, o cerrando conexiones de red). Los destructores se invocan automáticamente cuando las instancias a las que se refieren salen de ámbito. Si no hay que liberar nada, se pueden omitir.

- \* *Se debe usar el modificador **const** siempre que sea posible* (es decir, para todas las operaciones observadoras), ya que hacer esta distinción tiene múltiples ventajas. Por ejemplo, una instancia de un tipo *inmutable* (sin mutadores) se puede compartir sin que haya riesgo alguno de que se modifique el original.

- \* Es frecuente, durante el diseño de un TAD, poder elegir entre suministrar una misma operación como mutadora o como observadora. Por ejemplo, en un TAD Fecha, podríamos elegir entre suministrar una operación suma (int dias) que modifique la fecha actual sumándola delta días (mutadora), o que devuelva una *nueva* fecha delta días en el futuro (observadora, y por tanto *const*).

### 3.2. Uso de sub-TADs y extensión de TADs

- \* Es legal, y frecuente, que un TAD use a otro. Un ejemplo de buen uso de TAD interno sería, dentro de un Rectangulo, el uso de Punto:

---

```

#include "Punto.h" // usa Punto
/*
 * Un Rectangulo en 2D alineado con los ejes.
 */
class Rectangulo {
private:
    ...
public:
    // constructor
    //     origen es la esquina inf. izquierda
    Rectangulo(Punto origen, float alto, float ancho);

    // constructor
    //     interpreta ambos puntos como esquinas opuestas
    Rectangulo(Punto uno, Punto otro);

    // devuelve el punto con coordenadas x e y mínimas (Obs.)
    const Punto &origen() const;
    float alto() const;
    float ancho() const;

    // igualdad (Observadora)
    //     todos los rectangulos vacíos son iguales
    bool igual(const Rectangulo &r) const;

    // calcula area (Observadora)
    float area() const;

    // verifica si esta vacío, es decir, si tiene área 0 (Obs.)
    bool esVacio() const;

    // devuelve true si el punto está dentro (Obs.)
    //     el borde se considera dentro, excepto si el r. está vacío
    bool dentro(const Punto &p) const;

    // indica si este rectangulo contiene a r (Obs.)
    //     true si todos los puntos de r están dentro
    bool dentro(const Rectangulo &r) const;

    // calcula intersección (Observadora)
    //     devuelve el mínimo r con todos los puntos dentro de ambos
    Rectangulo intersección(const Rectangulo &r) const;
};

```

---

- \* Es frecuente también *enriquecer* tipos: partiendo de un tipo, añadir operaciones y/o extender su dominio. En lenguajes que soportan orientación a objetos, el uso de herencia permite hacer esto de una forma compacta y explícita. En general, no se pedirá manejo de herencia en este curso – pero se debe saber que existe y está disponible en C++.

### 3.3. TADs genéricos

- \* Un TAD genérico es aquel en el que uno o más de los tipos que se usan se dejan sin identificar, permitiendo usar las mismas operaciones y estructuras con distintos tipos concretos. Por ejemplo, en un TAD Conjunto, no debería haber grandes diferencias entre un conjunto de enteros, Puntos, o palabras. Hay varias formas de conseguir esta genericidad, entre ellas:
  - plantillas: usado en C++; permite declarar tipos como “de plantilla” (*templates*), que se resuelven en tiempo de compilación para producir todas las variantes concretas que se usan realmente. Mantienen un tipado fuerte y transparente al programador.
  - herencia: disponible en cualquier lenguaje con soporte OO. Requiere que todos los tipos concretos usados desciendan de un tipo base que implemente las operaciones básicas que se le van a pedir. Tienen mayor coste que los templates.
  - lenguajes dinámicos: JavaScript o Python son lenguajes que permiten a los tipos adquirir o cambiar sus operaciones en tiempo de ejecución. En estos casos, basta con que los objetos de los tipos introducidos soporten las operaciones requeridas en tiempo de ejecución (pero se pierde la comprobación de tipos en compilación).

- \* En C++, se pueden definir TADs genéricos usando la sintaxis

**template <class  $T_1$ , ... class  $T_n$ > contexto**

y refiriéndose a los  $T_i$  igual que se haría con cualquier otro tipo a partir de este momento. Generalmente se escogen mayúsculas que hacen referencia a su uso; por ejemplo, para un tipo cualquiera se usaría T, para un elemento E; etcétera.

Ejemplo de TAD genérico Pareja:

---

```
template <class A, class B>
class Pareja {
    // una pareja inmutable generica
    A _a; B _b;
public:
    // Constructor sencillo
    Pareja(A a, B b) { _a=a; _b=b; } // cuerpos en el .h (!)
    // Observadoras
    A primero() const { return _a; }
    B segundo() const { return _b; }
};
```

---

clase Pareja en pareja.h, con las implementaciones dentro de la clase

---

```
#include <iostream>
#include <string>
#include "pareja.h"
using namespace std;
int main() {
    Pareja<int, string> p(4, "hola");
    cout << p.primer() << " " << p.segundo() << "\n";
    return 0;
}
```

---

main.cpp que incluye a pareja.h

NOTA: en C++ es legal definir los cuerpos de funciones en el .h en lugar de en el .cpp (tal y como se hace en el ejemplo). En el caso de TADs genéricos, esto es **obligatorio** (en caso contrario se producirán errores de enlazado), aunque para implementaciones más grandes es preferible usar esta versión alternativa, que deja el tipo más despejado, a costa de repetir la declaración de los tipos de la plantilla para cada contexto en el que se usa:

---

```
template <class A, class B>
class Pareja {
    // una pareja inmutable generica
    A _a; B _b;
public:
    // Constructor sencillo
    Pareja(A a, B b);
    // Observadoras
    A primero() const;
    B segundo() const;
};

template <class A, class B>
Pareja<A,B>::Pareja(A a, B b) { _a = a; _b = b; }

template <class A, class B>
A Pareja<A,B>::primer() const { return _a; }

template <class A, class B>
B Pareja<A,B>::segundo() const { return _b; }
```

---

pareja.h alternativo, dejando más despejado el tipo

### 3.4. Implementación, parcialidad y errores

- \* Es vital hacer bien la distinción entre un **TAD** y una **implementación** concreta del mismo. Un TAD es, por definición, una abstracción. Al elegir una implementación concreta, siempre se introducen limitaciones - que podrían ser distintas en otra implementación distinta. Por ejemplo, las implementaciones típicas de los enteros tienen solo 32 bits (pero hay otras con 64, y otras que tienen longitud limitada solo por la memoria disponible, pero que son mucho más lentas). Es interesante ver que *todo lo que sea cierto para el TAD lo será para todas sus implementaciones*. No obstante, los límites de una implementación no tienen porqué afectar a otra del mismo TAD.
- \* Es posible descomponer la implementación de un TAD en dos pasos:
  1. Implementar la parte privada, eligiendo para ello los tipos de implementación concretos que se va a usar para representar el TAD (los *tipos representantes*);

la interpretación que se va a hacer de sus valores (la *función de abstracción*, que incluye también decir cuándo dos valores del tipo representante representan lo mismo: la *función de equivalencia*); y las condiciones que se deben cumplir para que los valores se consideren válidos (los *invariantes de representación*).

2. Implementar las operaciones, de forma que nunca se rompan los invariantes de representación (posiblemente con restricciones adicionales debidas al tipo representante elegido).
- ★ Ejemplos de decisiones para implementar partes privadas, usando la nueva terminología:
- Un TAD Rectangulo en 2D alineado con los ejes se puede representar con
    - un Punto origen y un par de int para ancho y alto (= *tipos representantes*). Todos los rectángulos vacíos se podrían considerar equivalentes, independientemente de origen (ya que no contienen ningún punto); esta interpretación nos proporciona la *función de abstracción* y la de *equivalencia*; y el ancho y el alto tendrían que ser, para rectángulos no-vacíos, estrictamente mayores que 0 (*invariante de representación*).
    - dos Puntos en esquinas opuestas, con ambos puntos idénticos en el caso de un rectángulo vacío. Podríamos exigir que las coordenadas *x* e *y* del segundo punto fuesen siempre mayores que las del primero, lo cual nos proporcionaría un invariante adicional (y simplificaría muchas operaciones).
  - El TAD Complejo se puede representar con
    - una pareja de float, uno para la parte entera y otro para la imaginaria. Las funciones de abstracción y de equivalencia son aquí triviales, y el invariante de representación se cumple siempre.
    - dos double, usando una representación polar ángulo-magnitud. La función de abstracción es en este caso la interpretación ángulo-magnitud (y la regla para pasar de esta representación a (*real*, *imaginario*), y el invariante de representación podría usarse para restringir el rango de ángulos posibles al intervalo  $[0, 2\pi]$ ).
  - Un TAD Fecha se puede representar con tres int que almacenen los días, meses y años; o con un int que represente segundos desde el 1 de enero de 1970. Las reglas que limitan sus valores válidos para una fecha determinada constituirían los invariantes de representación de esta implementación.
- ★ Ejemplos de restricciones impuestas por el tipo representante al dominio de valores del TAD:
- Muchos sistemas desarrollados durante las décadas de los 70 y 80 representaban el año mediante dos caracteres “XX”, interpretándolo como “19XX”. Esto ocasionó numerosos problemas cuando se llegó al año 2000, que estos sistemas interpretaban como si fuese 1900.
  - De forma similar, un valor entero representado mediante un int de 32 bits sólo puede tomar  $2^{32}$  valores distintos - lo cual restringe el dominio de implementación de un TAD que use ints de 32 bits como tipos representantes. Si representamos una Fecha con un int para los segundos desde 1970, será imposible producir fechas más allá del 19 de enero de 2038<sup>4</sup>.

---

<sup>4</sup>Esta representación es muy popular, y aunque muchas implementaciones ya se han actualizado a

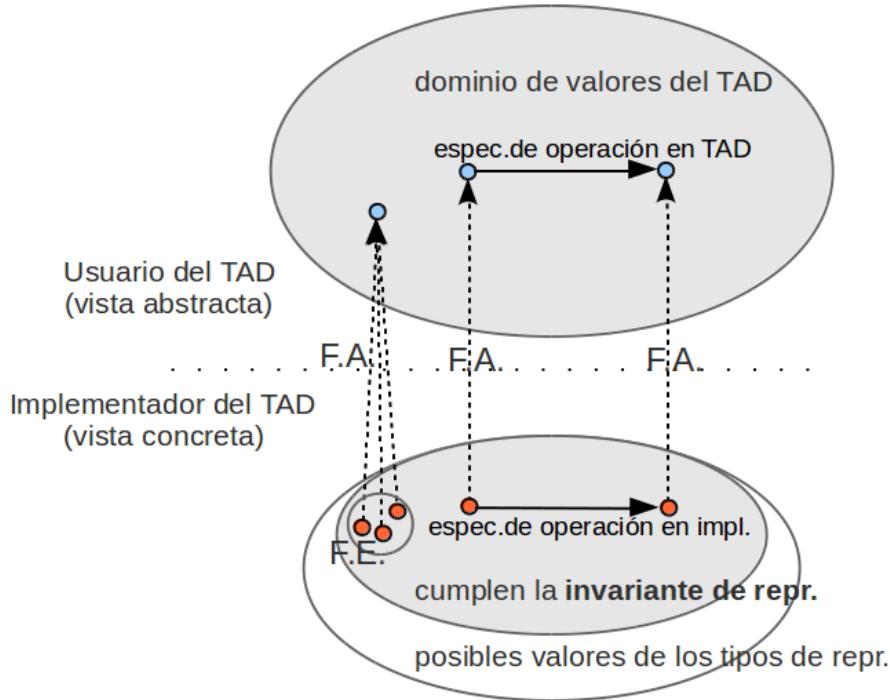


Figura 1: La función de abstracción (F.A.), la vista de usuario (abstracta), y la vista de implementador (concreta). El invariante de representación delimita un subconjunto de todos los valores posibles del tipo representante como *válidos*, y la función de abstracción los pone en correspondencia con términos del TAD. Es frecuente que varios valores del tipo representante puedan describir al mismo término del TAD – la función de equivalencia (F.E.) permite encontrar estos casos.

- \* El **invariante de representación** (ver Figura 1) consiste en el *conjunto de condiciones que se tienen que cumplir para que una representación se considere como válida*. Ejemplos:

- Rectangulo (usando Punto `_origen`, float `_ancho`, float `_alto`)
 
$$0 \leq \text{_ancho} \wedge 0 \leq \text{_alto}$$
- Rectangulo (usando Punto `_origen`, Punto `_extremo`)
 
$$\text{_origen.x} \leq \text{_extremo.x} \wedge \text{_origen.y} \leq \text{_extremo.y}$$
- Complejo (usando float `_real`, float `_imaginario`):
 
$$\text{True}$$
 (siempre se cumple)
- Complejo (usando float `_magnitud`, float `_angulo`):
 
$$0 \leq \text{_angulo} \leq 2\pi$$
- Hora (usando int `_horas`, int `_minutos`, int `_segundos`):
 
$$0 \leq \text{_horas} \wedge 0 \leq \text{_minutos} < 60 \wedge 0 \leq \text{_segundos} < 60$$
- Hora (usando sólo int `_segundos`):
 
$$0 \leq \text{_segundos}$$

enteros de 64 bits, el problema es comparable al del año 2000. Para más referencias, ver [http://en.wikipedia.org/wiki/Year\\_2038\\_problem](http://en.wikipedia.org/wiki/Year_2038_problem)

- \* La **relación de equivalencia** indica cuándo dos valores del tipo implementador representan el *mismo valor abstracto*. Usando el tipo Rectangulo, todos los rectángulos vacíos son idénticos, tengan los orígenes que tengan. En la Figura 3 se puede ver un ejemplo de conjuntos equivalentes. La relación de equivalencia está ilustrada en la Figura 1. Esta relación (o función) es abstracta (es decir, existe independientemente de que se implemente o no). No obstante, para hacer pruebas, es muy útil implementarla. En C++, la forma de hacerlo es sobrecargando el operador ‘==’:

---

```
class Rectangulo {
    ...
    // permite ver equivalencia de rectangulos mediante r1 == r2
    bool operator==(const Rectangulo &r) const {
        return (esVacio() && r.esVacio()) ||
            (_alto == r._alto && _ancho == r._ancho
            && _origen == r._origen);
    }
    ...
};
```

---

sobrecarga del operador == en C++

- \* Un TAD puede incluir aspectos *parciales*, es decir, no totalmente definidos. Es frecuente que las limitaciones impuestas por recursos finitos (memoria, ficheros) o debidas a los tipos de representante seleccionados (límites de `int` o `float`, vectores de tamaño fijo, etcétera) se traduzcan en precondiciones adicionales. Ciertas operaciones son por definición erróneas (intentar acceder más allá del final de un vector, o intentar dividir por cero). Cualquier operación que conlleve estas posibilidades se debe considerar *parcial*, y debe estar debidamente comentada explicando las precondiciones que garantizan un funcionamiento predecible.
- \* Ejemplo: un conjunto. En un TAD Conjunto, es posible añadir elementos, consultar si existe un elemento dado, y eliminar un elemento. En este caso, sería un error intentar eliminar un elemento que no exista.

---

```
template <class E>
class Conjunto {
    ... // aquí iría el tipo representante
public:
    // Constructor
    Conjunto();
    // inserta un elemento (mutadora)
    void inserta(const E &e);
    // elimina un elemento (mutadora)
    // parcial: si no contiene(e), error
    void elimina(const E &e);
    // si contiene el elemento, devuelve 'true' (observadora)
    bool contiene(const E &e) const;
};
```

---

un TAD Conjunto. La implementación de `elimina()` es parcial.

1. Tipo representante: Usaremos un vector de tamaño fijo, con el índice del último elemento indicado mediante un entero `_tam`:

---

```
static const int MAX = 100;
int _tam = 0;
E _elementos[MAX];
```

---

tipo representante para Conjunto

Debido a la limitación de tamaño, `inserta()` también es parcial:

---

```
// inserta un elemento (mutadora)
// parcial: si se intenta insertar más de MAX elementos, error
void inserta(const E &e);
```

---

debido al uso de un vector estático, la inserción es parcial

2. Invariante de representación:  $0 \leq \_tam \leq \text{MAX}$  (siempre se deben usar constantes<sup>5</sup> para los valores límite).

La elección del invariante tiene impacto en la forma de implementar las operaciones y por tanto también en el coste de las mismas. Podríamos considerar tres invariantes de representación diferentes:

- a) No hay ninguna restricción adicional a lo que se ha dicho hasta el momento.
- b) Exigimos que los elementos del vector entre  $0$  y  $\_tam - 1$  no estén repetidos.
- c) Exigimos que los elementos del vector entre  $0$  y  $\_tam - 1$  no estén repetidos y además estén ordenados (supuesto que exista una relación de orden entre ellos).

Supongamos  $\_tam = 3$ . Así, mientras que el vector

|   |   |   |     |
|---|---|---|-----|
| 2 | 1 | 2 | ... |
|---|---|---|-----|

es válido en la representación (1), no lo es ni en la (2) ni en la (3) porque el 2 está repetido.

Análogamente, el vector

|   |   |   |     |
|---|---|---|-----|
| 2 | 1 | 3 | ... |
|---|---|---|-----|

es válido en las representaciones (1) y (2) pero no en la (3), porque los elementos no están ordenados.

3. Función de abstracción y relación de equivalencia: Para pasar de un valor de `_elementos` y `_tam` a un conjunto abstracto, entenderemos que los elementos con índice  $0$  a  $\_tam$  (exclusive) forman parte del conjunto salvo posibles repeticiones.

Nótese que puede haber múltiples valores del tipo representante (en este caso, múltiples vectores de `_elementos`) que se refieran al mismo conjunto abstracto:

- Por una parte, a partir de `_tam` (inclusive), el contenido de `_elementos` es completamente indiferente desde el punto de vista del TAD, como se muestra en la Figura 3.
- Por otra parte, las repeticiones y el orden son irrelevantes. Los ejemplos anteriores representan los conjuntos  $\{1, 2\}$  y  $\{1, 2, 3\}$  en aquellas representaciones en las que eran válidos. Por ejemplo, vectores equivalentes en la

---

<sup>5</sup>El uso de `static const` declara `MAX` como constantes (`const`) para esta implementación del TAD, definidas una única vez para cualquier número de conjuntos (`static`), en lugar de una vez para cada conjunto individual (ausencia de `static`). Es preferible usar constantes estáticas de clase que `#defines`.

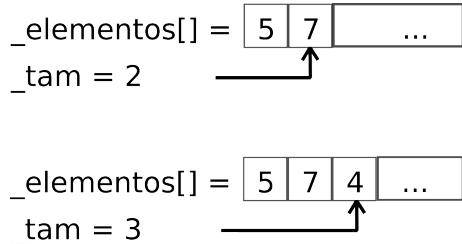


Figura 2: Conjunto antes y después de insertar el elemento 4

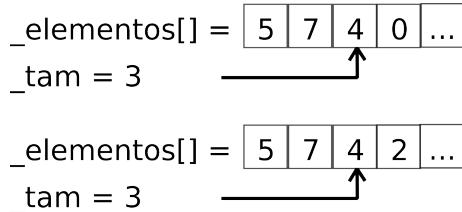


Figura 3: Conjuntos equivalentes (según la función de equivalencia del TAD)

representación (1) son:  $\{1, 1, 2, 1, 2, \dots\}$  con  $_tam = 5$  y  $\{1, 2, \dots\}$  con  $_tam = 2$ . Vectores equivalentes en la representación (2) son:  $\{1, 2, \dots\}$  con  $_tam = 2$  y  $\{2, 1, \dots\}$  con  $_tam = 2$ . En la representación (3) los vectores equivalentes solamente difieren en las posiciones de  $_tam$  en adelante.

4. Implementaciones de cada operación, respetando los invariantes. Vamos a ver aquí la representación con invariante (1):

```

Conjunto() { _tam = 0; }

constructor

void inserta(const E &e) {
    if (_tam == MAX) throw "Conjunto Lleno";
    _elementos[_tam] = e;
    _tam++;
}

inserta()

```

Donde, en caso de error, se usa la sentencia “`throw`” para *lanzar* una excepción y salir de la función (tiene efectos similares, pero más fuertes, que un `return`); en el punto siguiente se muestran más estrategias posibles, y se recomienda que, en lugar de lanzar excepciones de tipo `const char *`, se usen otros tipos de dato que permitan al programa interpretar fácilmente de qué excepción se trata. Por ejemplo, se podrían definir los tipos `ConjuntoLleno` y `ElementoInvalido`, que se usarían luego en las sentencias `throw` en substitución de las actuales cadenas de caracteres.

Para implementar la operación de eliminación, hemos de tener en cuenta que puesto que los elementos pueden estar repetidos, hay que eliminar todas las apariciones de dicho elemento:

---

```

void elimina(const E &e) {
    bool esta = false;
    int i=0;
    while (i<_tam)
        { if (_elementos[i]==e){
            _elementos[i] = _elementos[_tam-1];
            esta = true;
            _tam --;
            // paso el ultimo elemento a su lugar
        else i++;
        };
    if (!esta) throw "Elemento Invalido";
}

elimina()

```

---

```

bool contiene(const E &e) const {
    bool esta = false;
    int i=0;
    while (i<_tam && !esta)
        {esta = (_elementos[i]==e);
        i++;
        };
    return esta;
}

```

---

```

contiene()

```

Con esta forma de representar los conjuntos, las operaciones tienen los siguientes costes:

- El constructor y la operación `inserta` están en  $\Theta(1)$ .
- Las operaciones `elimina` y `contiene` están en  $\Theta(\_tam)$  en el caso peor.

Obsérvese que puesto que puede haber elementos repetidos, `_tam` puede ser mucho más grande que el cardinal del conjunto al que representa. En las otras representaciones, que exigen que no haya repetición de elementos, `_tam` representa exactamente el cardinal del conjunto.

- \* Hasta ahora, hemos ignorado el manejo de los errores que se pueden producir en un programa (ya sea una función independiente o una función que forma parte de un TAD). Hay varias estrategias posibles para tratar errores:
  - Devolver (vía `return`) un “valor de error”.
    - Requiere que haya un valor de error disponible (típicamente `NULL` o `-1`) que no puede ser confundido con una respuesta de no-error; cuando no lo hay, lo común es devolver un booleano indicando el estado de error y pasando el antiguo valor devuelto a un argumento por referencia.
    - Requiere que el código que llama a la función verifique si ha habido o no error mediante algún tipo de condicional, y lleva a código difícil de leer.

No obstante, este patrón se usa en muchos sitios; por ejemplo, la API C++ de Windows, la de Linux, o la librería estándar de C. En todas estas librerías se

permite recabar más información sobre el error con llamadas adicionales, lo cual implica guardar información sobre el último error que se produjo por si alguien la pide luego<sup>6</sup>.

- **Lanzar una excepción**

- Requiere soporte del lenguaje de programación (C o Pascal no las soportan; Java, C++ o Delphi sí). En general, casi todos los lenguajes con orientación a objetos soportan excepciones.
- Requiere que el código que llama a la función decida si quiere manejar la excepción (lanzada con `throw`), usando una sentencia `catch` apropiada.

El código resultante es más limpio (requiere menos condicionales), y la excepción puede contener toda la información disponible sobre cómo se produjo - no hace falta que las librerías que lanzan excepciones mantengan siempre el último error. Esta es la estrategia que se sigue en las librerías orientadas a objetos de, por ejemplo, la API .NET de Windows o la librería estándar de Java.

- Mostrar un mensaje por pantalla y devolver cualquier valor. Esta estrategia sólo es admisible cuando se está depurando, la librería es muy pequeña, y somos su único usuario; e incluso entonces, es poco elegante. Su única ventaja es que resulta fácil de implementar.
- Exigir una función adicional, pasada como parámetro (en muchos lenguajes, C++ inclusive, es posible pasar funciones como parámetros), a la que llamar en caso de error. Esto es común en librerías con un fuerte componente asíncrono; por ejemplo, cuando se hacen llamadas “AJAX” desde JavaScript.

- ★ Ejemplo: Lanzando y capturando excepciones en C++

---

```
#include <iostream>
int divide(int a, int b) {
    if (b==0) {
        throw "imposible dividir por cero"; // tipo 'const char *'
    }
    return a/b;
}

using namespace std;
int main() {
    try {
        cout << divide(12,3) << "\n";
        cout << divide(1, 0) << "\n";
    } catch (const char * &s) { // ref. al tipo lanzado
        cout << "ERROR: " << s << "\n";
    }
    return 0;
}
```

---

- ★ Las excepciones lanzadas pueden ser de cualquier tipo. No obstante, **se deben usar clases específicas**, tanto por legibilidad como por la posibilidad de usarlas para su-

<sup>6</sup>Se suele usar para ello una variable global (en el caso de Unix/Linux, `errno`). El uso de esta variable permite adoptar una convención adicional: en algunos casos, en lugar de devolver un “valor de error”, se espera que se consulte el valor de `errno` para saber si hubo o no error. Esto requiere mucha disciplina por parte del programador para consultar el valor de `errno` tras cada llamada, antes de que la siguiente lo sobreescriba con un nuevo valor – algo que puede ser imposible en presencia de programación concurrente.

ministrar más información dentro de las propias excepciones. En el siguiente ejemplo, usamos una clase base `Excepcion`, con subclases `DivCero` y `Desbordamiento`:

```
#include <iostream>
#include <string>
#include <climits>

class Excepcion {
    const std::string _causa;
public:
    Excepcion(std::string causa) : _causa(causa) {};
    const std::string &causa() const { return _causa; };
};

class DivCero : public Excepcion {
public: DivCero(std::string causa) : Excepcion(causa) {};
};

class Desbordamiento : public Excepcion {
public: Desbordamiento(std::string causa) : Excepcion(causa) {};
};

int suma(int a, int b) {
    if ((a>0 && b>0 && a>INT_MAX-b) || (a<0 && b<0 && a<INT_MIN-b)) {
        throw Desbordamiento("excede limites en suma");
    }
    return a+b;
}

int divide(int a, int b) {
    if (b==0) {
        throw DivCero("en division");
    }
    return a/b;
}

using namespace std;
int main() {
    try {
        cout << divide(12,3) << "\n";
        cout << suma(INT_MAX, -10) << "\n";
        cout << divide(1, 0) << "\n";           // lanza excepcion
        cout << suma(INT_MAX, 10) << "\n";   // esto nunca se evalua
    } catch (DivCero &dc) {                  // trata DivCero (si se produce)
        cout << "Division por cero: " << dc.causa() << "\n";
    } catch (Desbordamiento &db) {          // trata Desbordamiento (si hay)
        cout << "Desbordamiento: " << db.causa() << "\n";
    } catch (...) {                         // trata cualquier otra excepcion
        cout << "Excepcion distinta de las anteriores!\n";
    }
    return 0;
}
```

- \* NOTA: en C++, es posible usar el modificador `const` en 3 contextos completamente distintos:

**para indicar una constante** – usado en la declaración de la constante, y preferible al uso de `#define`.

**para indicar un método observador** – en el interior del cual no se modifica el valor de la instancia actual de una clase o estructura.

**como modificador de tipo** – en los argumentos de entrada o salida de un método, para indicar que el valor pasado no se puede cambiar dentro de la función. Tiene más sentido para argumentos pasados por referencia (&) o puntero (\*), porque los argumentos pasados por valor siempre son copias, y por tanto modificarlos nunca tiene efectos externos a la función.

Ejemplo:

```
// devuelve el vértice del polígono más cercano al punto pasado
const Punto &Poligono::contiene(const Punto &p) const;
```

El primer y segundo const indican que el punto devuelto y el punto pasado son referencias no-modificables (uso como modificador de tipo). El último const indica que esta operación es observadora, y no modifica el polígono sobre el que se llama.

- \* Las operaciones de un TAD pueden devolver valores o referencias, tanto como valor de la función como a través de argumentos. A continuación proponemos una serie de **convenciones** a seguir en cuanto a los valores devueltos, asumiendo que se están usando excepciones para el manejo de las condiciones de error:

- Si la función no tiene un “valor devuelto” claro, como es el caso de las operaciones de mutación, no se debe devolver nada (declarando la función como `void`).
- En el caso de observadores, si *sólo se devuelve un valor*, el tipo de retorno de la función debe ser el del valor devuelto.
  - Si el valor es un `bool`, un `int`, un puntero, o similarmente pequeño (y por tanto una referencia no ahorraría nada), el tipo del resultado es el del valor devuelto, lo que significa que la instrucción `return` hará una copia del valor devuelto.
  - Si el valor es más grande, pero se trata de un nuevo objeto construido en el método, el tipo del resultado es el del valor devuelto y análogamente se hará una copia del valor devuelto por la instrucción `return`.
  - Si el valor es grande, pero comparte con un objeto ya existente, es más eficiente devolver una referencia, ya que en tal caso no se hará copia. Para evitar que modificaciones sobre ese valor afecten al objeto con el que comparte (lo que habitualmente recibe el nombre de efecto lateral), dicha referencia debe ser constante.

```
// copia: bool no es grande,
bool contiene(const E &e) const;
// ref. cte: ocupa más que un int y es parte de un rectángulo
const Punto &origen() const;
// copia: devuelve un nuevo rectángulo
Rectangulo Rectangulo::interseccion(const Rectangulo &r1) const;
```

- Si hay que devolver *más de un valor a la vez*, se puede devolver a través de argumentos “por referencia”. El tipo de retorno de la función queda libre, y de nuevo se debería devolver `void`. Alternativamente, es posible agrupar todos los valores a devolver en una estructura o clase, y devolverla como único valor.

```
// devuelve coordenadas del i-esimo punto en x e y
void Poligono::punto(int i, float &x, float &y) const;
```

---

```
// version mejorada: devuelve referencia a Punto
const Punto &Poligono::punto(int i) const;
```

### 3.5. Implementaciones dinámicas y estáticas

- ★ Algunos TADs pueden llegar a requerir mucho espacio. Por ejemplo, el tamaño de un Conjunto puede variar entre uno o dos elementos y cantidades astronómicas de los mismos, en función de las necesidades de la aplicación que lo use. Cuando un TAD tiene estos requisitos, no tiene sentido asignarle un espacio fijo “en tiempo de compilación”, y es mejor solicitar memoria al sistema operativo (SO) a medida que va haciendo falta. **La memoria obtenida en tiempo de ejecución mediante solicitudes al SO se denomina dinámica. La que no se ha obtenido así se denomina estática**<sup>7</sup>.
- ★ La memoria dinámica se gestiona mediante punteros. Un puntero no es más que una dirección de memoria; pero en conexión con la memoria dinámica, se usa también como identificador de una reserva de memoria (que hay que devolver). En C++, la memoria se reserva y se libera mediante los operadores **new** y **delete**:

---

```
Conjunto<int> *c = new Conjunto<int>();
// conjunto de enteros en memoria dinámica
c->inserta(4); // inserto un 4
(*c).inserta(2); // "(*algo)." equivale a "algo->"
c->elimina(4); // quito el primer 4
delete c; // libero el conjunto
```

---

Y sus variantes **new[]** y **delete[]**, que se usan para reservar y liberar vectores, respectivamente:

---

```
int cuantos = 1<<30; // = 230 (y<<x equivale a y·2x)
int *dinamicos = new int[cuantos]; // reserva 1GB de memoria dinámica
...
delete[] dinamicos; // libera memoria anterior
```

---

Recordatorio sobre punteros:

---

```
int x = 10, y[] = {3, 8, 7};
Pareja<int, int> *par = new Pareja<int, int>(4, 2);
int *px = &x; // &x: dirección de memoria de la variable x
int z = *px; // *px: acceso a los datos apuntados por px == 10
int x = par->primero(); // = (*par).primero() = 4
int *py = y; // un vector es también un puntero a su comienzo
int w = *(py+1); // = 8 = py[1] = y[1]
```

---

- ★ Puedes saber cuánta memoria “directa” (es decir, sin contar la memoria apuntada mediante punteros) ocupa una estructura o clase C++ usando el operador **sizeof**. Para el conjunto estático con 100 elementos declarado antes, **sizeof(Cierto<int>)** será **sizeof(\_tam) + sizeof(\_elementos) = 404 bytes** (es decir, el tamaño ocupado por sus tipos representantes: tanto **\_tam** como cada uno de los 100

<sup>7</sup>El sistema operativo asigna, a cada proceso, un espacio fijo de pila de programa (o *stack*) de unos 8MB, que es de donde viene la memoria estática; la memoria dinámica (también denominada *heap* o “del montón”) se obtiene del resto de la memoria libre del sistema, y suele ser de varios GB. Puedes probar a quedarte sin memoria estática escribiendo una recursión infinita: el mensaje resultante, “stack overflow” o “desbordamiento de pila”, indica que tu pila ha desbordado su tamaño máximo prefijado.

`_elementos` ocupan 4 bytes, por ser de tipo `int`). De igual forma, `sizeof` (`Conjunto<char>`) habría sido 104 (ya que los `_elementos` de tipo `char` ocuparían sólo 1 byte cada uno). En una versión dinámica de este conjunto, `_elementos` se habría reservado mediante un `new` (y por tanto sería de tipo puntero); en este caso, el tamaño de la pila sería siempre `sizeof(_tam) + sizeof(E *) = 12` (en sistemas operativos de 64 bits; u 8 si tu sistema operativo es de 32), independientemente del tamaño del vector y del número de elementos que contenga: el operador `sizeof` ignora completamente la existencia de memoria dinámica.

- ★ Errores comunes con punteros y memoria dinámica:
  - **Usar un puntero sin haberle asignado memoria** (mediante un `new / new[]` previo). El puntero apuntará a una dirección al azar; el resultado queda indefinido, pero frecuentemente resultará en que la dirección estará fuera del espacio de direcciones del programa, ocasionando una salida inmediata.
  - **No liberar un puntero tras haber acabado de usarlo** (mediante un `delete / delete[]`). La memoria seguirá reservada, aunque no se pueda acceder a ella. Esto se denomina “fuga de memoria”, o “escape de memoria” (*memory leak*), y al cabo de un tiempo (en función de la frecuencia y cantidad de memoria perdida), el sistema operativo podría quedarse sin memoria. En este momento empezarán a fallar las reservas de los programas; si falla una reserva por falta de memoria, se lanzará una excepción (de tipo `bad_alloc`) que hará que acabe tu programa.
  - **Liberar varias veces el mismo puntero.** La librería estándar de C++ asume que sólo liberas lo que no está liberado, e intentar liberar cosas ya liberadas o que nunca reservaste en primer lugar conduce a comportamiento no definido (típicamente a errores de acceso).
  - **Acceder a memoria ya liberada.** No hay ninguna garantía de que esa memoria siga estando disponible para el programa (si no lo está, se producirá error de acceso), o de que mantenga su contenido original.
  - **Intentar escribir y leer el valor de un puntero** (por ejemplo, a un archivo) **fuera de una ejecución concreta**. Un puntero sólo sirve para apuntar a una dirección de memoria. Cuando esa dirección deja de contener lo que contenía, el valor del puntero ya no sirve para nada. Por ejemplo, si en un momento un programa tiene, en la dirección `0xA151E0FF`, el comienzo de un array dinámico, no hay ninguna garantía de que esa misma dirección vaya a contener nada interesante (o incluso accesible) en la siguiente ejecución<sup>8</sup>.
- ★ Para evitar estos errores, se recomiendan las siguientes prácticas:
  - Usa memoria dinámica sólo cuando sea necesaria. No es “más elegante” usar memoria dinámica cuando no es necesaria; pero sí es más difícil y resulta en más errores.
  - Inicializa tus punteros nada más declararlos. Si no puedes inicializarlos inmediatamente, asígnales el valor `NULL (= 0)` para que esté garantizado que todos los accesos produzcan errores inmediatos (y por tanto más fáciles de diagnosticar).

<sup>8</sup>Es más, hay sistemas operativos que intentan, a propósito, hacer que sea muy difícil adivinar dónde acabarán las cosas (vía *address space randomization*); esto tiene ventajas frente a programas malintencionados.

- Nada más liberar un puntero, asígnale el valor NULL (= 0); **delete()** nunca libera punteros a NULL; de esta forma puedes evitar algunas liberaciones múltiples:

---

```
// asigno un valor en la misma declaración
Conjunto *c = new Conjunto();
delete c; // comprueba NULL antes de liberar
c = NULL; // para evitar accesos/liberaciones futuros
```

---

### 3.5.1. Constructores y destructores

- ★ En cualquier *implementación* de TAD dinámico (los TADs en sí no son ni dinámicos ni estáticos; *solo sus implementaciones* pueden serlo), habrá dos fragmentos de código especialmente importantes: la inicialización de una nueva instancia del TAD (donde, como parte de las inicializaciones, se hacen los *new*s), y la liberación de recursos una vez se acaba de trabajar con una instancia dada (donde se hacen los *delete*s que faltan). En C++, estos fragmentos reciben el nombre de **constructor** y **destructor**, respectivamente, y tienen una sintaxis algo distinta de las demás funciones:
  - No devuelven nada (solo pueden comunicar errores mediante excepciones).
  - Sus nombres se forman a partir del nombre del TAD. Dado un TAD X, su constructor se llamará X, y su destructor ~X.
  - Si no se especifican, el compilador genera versiones “por defecto” de ambos, públicas, con 0 argumentos y totalmente vacías. Es equivalente escribir `class X {};` o escribir `class X { public: X() {} ~X() {} };`.

Veamos un ejemplo con una implementación con memoria dinámica de Mapa:

---

```
class Mapa {
    unsigned short *_celdas; // 2 bytes por celda
    int _ancho;
    int _alto;
public:
    // Generador - Constructor
    Mapa(int ancho, int alto) : _ancho(ancho), _alto(alto) {
        if (ancho<1||alto<1) throw "dimensiones mal";
        _celdas = new unsigned short[_alto*_ancho];
        for (int i=0; i<alto*ancho; i++) _celdas[i] = 0;
        std::cout << "construido: mapa de "
            << _ancho << "x" << _alto << "\n";
    }
    // Generador - Mutador
    void celda(int i, int j, unsigned short v) {
        _celdas[i*_ancho + j] = v;
    }
    // Observador
    unsigned short celda(int i, int j) {
        return _celdas[i*_ancho + j];
    }
    // Destructor
    ~Mapa() {
        delete[] _celdas;
```

---

```

        std::cout << "destruyendo: mapa de "
        << _ancho << "x" << _alto << "\n";
    }
};

int main() {
    Mapa *m0 = new Mapa(128, 128); // llama a constructor (m0)
    Mapa m1 = Mapa(2560, 1400); // llama a constructor (m1)
    { // comienza un nuevo bloque
        Mapa m2(320, 200); // llama a constructor (m2)
    } // al cerrar bloque llama a los destructores (m2)
    delete m0; // destruye m0 explícitamente
    return 0;
}
// al cerrar bloque llama a destructores (m1; m0 es puntero)

```

---

Es importante la distinción entre la forma en que se liberan las variables estáticas (`m1` y `m2`: automáticamente al cerrarse los bloques en que están declarados) y la forma en que se liberan las variables dinámicas (`m0`: explícitamente, ya que fue reservado con un `new`). Si no se hubiese liberado explícitamente `m0`, se habría producido una fuga de memoria. El ejemplo produce la siguiente salida:

```

construido: mapa de 128x128
construido: mapa de 2560x1400
construido: mapa de 320x200
destruyendo: mapa de 320x200
destruyendo: mapa de 128x128
destruyendo: mapa de 2560x1400

```

- \* Finalmente, además de constructores y destructores, hay otras dos operaciones que se crean por defecto en cualquier clase nueva: el **operador de copia**, con la cabecera siguiente (por defecto, pública):

```
UnTAD& operator=(const UnTAD& otro);
```

Y el **constructor de copia**, que es similar en función:

```
UnTAD (const UnTAD& otro);
```

Ambos se usan como sigue:

```

Conjunto<int> c1().inserta(42);
Conjunto<int> c2;
c2 = c1; // operador asignacion
Conjunto<int> c3 = c1; // constructor de copia, equivale a c3(c1)
Mapa m1(100, 100);
Mapa m2;
m2 = m1; // operador asignacion
Mapa m3 = m1; // constructor de copia, equivale a m3(m1)

```

En sus versiones por defecto (generadas por el compilador), ambos copian campo a campo del objeto `otro` al objeto actual (la razón de que exista un constructor de copia es que, si no existiera, primero habría que crear un objeto que no se usaría, y luego machacarlo con los valores del otro; es decir: ahorra una copia). Esto sólo funciona como se espera si la implementación no usa punteros. En el caso de una implementación dinámica, copiar un campo de tipo puntero resulta en que ambos punteros (el original y la copia) apuntan a lo mismo; y modificar la copia resultará

en que también se modifica el original, y viceversa. Si esto no es lo que se desea en una implementación dada, se debe escribir una versión explícita de este operador. En el ejemplo anterior, modificar `m1` y `m2` es equivalente (ambos comparten los mismos datos). En cambio, `c1` y `c2` son independientes (se trataba de implementaciones estáticas).

\* Ventajas de las implementaciones estáticas:

- Más sencillas que las dinámicas, ya que no requieren código de reserva y liberación de memoria.
- El operador de copia por defecto funciona tal y como se espera (asumiendo que no se usen punteros de ningún tipo), y las copias son siempre independientes.
- Más rápidos que los dinámicos; la gestión de memoria puede requerir un tiempo nada trivial, en función de cuántas reservas y liberaciones hagan los algoritmos empleados. Esto se puede solucionar parcialmente haciendo pocas reservas grandes en lugar de muchas pequeñas.

\* Ventajas de las implementaciones dinámicas:

- Permiten tamaños mucho más grandes que las estáticas: hasta el máximo de memoria disponible en el sistema, típicamente varios GB; en comparación con los ~8MB que pueden ocupar (en total) las reservas estáticas.
- No necesitan malgastar memoria - pueden reservar sólo la que realmente requieren.
- Pueden compartir memoria, mediante el uso de punteros (pero esto requiere mucho cuidado para evitar efectos indeseados).

## 4. Probando y documentando TADs

- \* Las verificaciones formales son una buena forma de convencernos de la corrección de los programas. No obstante, es posible cometer errores en las demostraciones, o demostrar algo distinto de lo que realmente se ha implementado.
- \* Las pruebas se pueden dividir en varios tipos:
  - Pruebas **formales**: verifican la corrección teórica de los algoritmos, sin necesidad de acceso a una implementación concreta. Son las que hemos visto en los temas 3 (iterativos) y 4 (recursivos).
  - Pruebas de **caja negra**: verifican una implementación sólo desde el punto de vista de su interfaz, sin necesidad de acceder a su código. Se usan para ver si “desde fuera de la caja” todo funciona como debe.
  - Pruebas de **caja blanca**: verifican aspectos concretos de una implementación, teniendo en cuenta el código de la misma.
- \* La verificación formal, si se realiza correctamente, *es la única que puede demostrar la corrección de un algoritmo o la consistencia de una especificación*. Las pruebas de implementaciones sólo pueden encontrar errores, pero no encontrar un error no es garantía de que no exista. Desgraciadamente, requiere un trabajo sustancial, y (a no ser que se use asistencia de herramientas informáticas) es susceptible a errores en las

demostraciones. En un escenario real, sólo se deben usar en casos muy puntuales, o en campos de aplicación donde la seguridad e integridad de los datos es realmente crítica (como aviación o control de centrales nucleares).

- \* Las pruebas de *caja negra* permiten verificar la especificación de una implementación “desde fuera”, contrastando una serie de comportamientos esperados (según la especificación) con los comportamientos obtenidos. No requieren acceso a los fuentes.

Unas pruebas de caja negra deberían incluir, para cada operación, comprobaciones de que produce la salida esperada para cada uno de los ejemplos de entrada con los que se va a enfrentar. En general, es imposible probar todos los casos, por lo que se elige un subconjunto representativo; cuantos más casos se puedan cubrir, más fiables serán las pruebas. Un ejemplo de pruebas de caja negra para Conjunto podrían ser las siguientes:

---

```
#include <iostream>
#include <cassert>
using namespace std;
int main() {
    Conjunto<int> vacio;
    Conjunto<int> c;
    // tras insertar un elemento, el elemento existe
    c.inserta(21);
    assert(c.contiene(21));
    // y se puede borrar
    c.elimina(21);
    // y despues ya no existe
    assert( ! c.contiene(21));
    // y si intento borrarlo se produce un error
    bool error = false;
    try {
        c.elimina(12);
    } catch (ElementoInvalido e) {
        error = true;
    }
    assert(error);
}
```

---

algunas pruebas de caja negra para Conjunto

Donde el uso de `assert` (importado mediante `#include <cassert>`) permite introducir comparaciones “todo o nada”: si en algún caso el contenido es `false`, se sale del programa inmediatamente con un mensaje indicando la línea del error. El código también asume que se ha implementado un operador “`==`” para verificar igualdad entre conjuntos (que deberá reflejar correctamente la *relación de equivalencia* entre conjuntos - ver Figura 3). Para estar razonablemente seguros de que la implementación funciona como se espera, habría que ejecutar el código anterior con muchos conjuntos y elementos distintos.

- \* Las pruebas de caja blanca van un paso más allá que las de caja negra, y partiendo del código de una implementación, ejercitan ciertas trazas de ejecución para verificar que funcionan como se espera.
  - Una *traza de ejecución (execution path)* es una secuencia concreta de instrucciones que se pueden ejecutar con una entrada concreta; por ejemplo, en este

fragmento hay 3 trazas posibles:

```
if (a() && b()) c(); else d();
```

1. a(), d() (asumiendo que a() devuelva `false`)
2. a(), b(), d() (asumiendo que a() devuelva `true` y b() devuelva `false`)
3. a(), b(), c() (asumiendo que a() y luego b() devuelvan `true`)

- Las trazas relacionadas se agrupan en “tests unitarios” (*unit tests*), cada uno de los cuales ejercita una función o conjunto de funciones pequeño.
- Al porcentaje de código que está cubierto por las trazas de un conjunto de pruebas de caja blanca se le denomina “cobertura”. Idealmente, el 100 % del código debería estar cubierto - pero en un comienzo, lo más importante es concentrarse en los fragmentos más críticos: aquellos que se usan más o que pueden producir los problemas más serios. Existen herramientas que ayudan a automatizar el seguimiento de la cobertura de un programa.
- Es posible que exista código que no puede ser cubierto por ninguna traza (por ejemplo, `if (false) cout << "impossible!\n";`). A este código se le denomina “código muerto” - y sólo puede ser detectado por inspección visual, herramientas de cobertura, o (en algunos casos), compiladores suficientemente avanzados.

#### 4.1. Documentando TADs

- \* La documentación de un TAD es crítica, ya que es el único sitio en el que se describe la abstracción que representa el tipo, cómo se espera que se use, y las especificaciones a las que se adhiere. Uno de los principales objetivos de un TAD es ser *reutilizable*. Típicamente, el equipo o persona encargados de reutilizarlo no será el mismo que lo implementó (e incluso si lo es, es muy fácil olvidarse de cómo funciona algo que se escribió hace meses, semanas o años). Leer código fuente es difícil, y a veces no estará disponible - una buena forma de entender un programa complejo es examinar la documentación de sus TADs para hacerse una idea de las abstracciones que usa.
- \* Todo TAD debe incluir, en su documentación:
  - Una **descripción de alto nivel** de la abstracción usada. Por ejemplo, en un TAD Sudoku, la descripción podría ser *Un tablero de Sudoku 9x9, que contiene las soluciones reales, las casillas iniciales, y los números marcados por el usuario en casillas inicialmente vacías..* En algunos casos, el TAD es tan bien conocido que no hace falta extenderse en esta descripción; por ejemplo, un Punto2D o una Pareja son casi auto-explicativos.
  - Una **descripción de cada uno de sus campos públicos**, ya sean operaciones, constantes, o cualquier otro tipo. La descripción debe especificar cómo se espera que se use, y en general puede incluir notación tanto formal como informal. En casos particularmente complejos, tiene sentido incluir pequeños ejemplos de uso.
- \* En C++, la documentación de un TAD se escribe *exclusivamente* en su .h. Los comentarios en el .cpp se refieren a la implementación concreta de las operaciones, aunque es útil y recomendable incluir cabeceras de función mínimas (resumidas) en los .cpp.

- ★ Los TADs auxiliares de un TAD principal, si son visibles para un usuario (es decir, accesibles mediante operaciones públicas) deben tener el mismo nivel de documentación que el TAD principal. Así, si un TAD Polígono usa un TAD Punto devolviendo o aceptando puntos, el TAD Punto también debe estar bien documentado. No es necesario (pero sí recomendable) tener el mismo cuidado con los TADs auxiliares privados.
- ★ El estilo de la documentación de un proyecto, junto con el estilo del código del proyecto, se especifican al comienzo del mismo (idealmente mediante una “guía de estilo”). A partir de este momento, todos los desarrollos deberán atenerse a esta guía. Para nuevos proyectos, puedes usar un estilo similar al siguiente:

---

```

/*
 * @file    conjunto.h
 * @author manuel.freire@fdi.ucm.es
 * @date    2014-02-12
 * @brief   Conjunto sencillo
 *
 * Un conjunto estatico muy sencillo, que usa un vector
 * fijo para los elementos
 */
#ifndef CONJUNTO_H_
#define CONJUNTO_H_

/// excepcion de conjunto lleno
class ConjuntoLleno {};

/// excepcion de elemento inexistente
class ElementoInvalido {};

template <class E>
class Conjunto {
    static const int MAX = 100; /*< max. de elementos */
    int _tam;                /*< numero de elementos actuales */
    E _elementos[MAX];       /*< vector estatico de elementos */

public:

    /**
     * Inicializa un conjunto vacio (constructor).
     */
    Conjunto() { _tam = 0; }

    /**
     * Inserta un elemento en el conjunto
     * (mutador parcial; si se llena, lanza ConjuntoLleno).
     * @param e elemento a insertar
     */
    void inserta(const E &e) {
        if (_tam == MAX) throw ConjuntoLleno();
        _elementos[_tam] = e;
        _tam++;
    }

    /**
     * Borra un elemento del conjunto

```

```

* (mutador parcial; si no existe, lanza ElementoInvalido)
*/
void elimina(const E &e) {
    bool esta = false;
    int i=0;
    while (i<_tam)
        { if (_elementos[i]==e){
            _elementos[i] = _elementos[_tam-1];
            esta = true;
            _tam --;
            // paso el ultimo elemento a su lugar
        } else i++;
        };
    if (!esta) throw ElementoInvalido();
}

/**
 * Devuelve true si el elemento esta contenido en el conjunto
 * (observador)
 */
bool contiene(const E &e) const {
    bool esta = false;
    int i=0;
    while (i<_tam && !esta)
        {esta = (_elementos[i]==e);
        i++;
        };
    return esta;
}

};

#endif

#include <iostream>
#include <cassert>
using namespace std;
int main() {
    Conjunto<int> vacio;
    Conjunto<int> c;
    // tras insertar un elemento, el elemento existe
    c.inserta(21);
    assert(c.contiene(21));
    // y se puede borrar
    c.elimina(21);
    // y despues ya no existe
    assert( ! c.contiene(21));
    // y si intento borrarlo se produce un error
    bool error = false;
    try {
        c.elimina(12);
    } catch (ElementoInvalido e) {
        error = true;
    }
}

```

```
    assert(error);
}
```

---

listado completo de la implementación de una clase Conjunto

En este ejemplo, se usan anotaciones tipo *doxygen*<sup>9</sup>. El sistema doxygen permite generar documentación pdf y html muy detallada a partir de los fuentes y las anotaciones que contiene su documentación.

## 5. Para terminar...

Terminamos el tema con la implementación de la función descrita en la primera sección de motivación. Como se ve, al hacer uso de un TAD Conjunto el código queda realmente legible: nos podemos centrar en la idea central (detectar el primer duplicado) sin preocuparnos para nada de los detalles. Además, si mejoramos la implementación del Conjunto (y la de este tema es muy ineficiente), no habrá que modificar en nada nuestra solución – gracias al uso de un TAD, nos hemos aislado de los detalles de implementación.

```
#include "conjunto.h"
#include <iostream>

using namespace std;

int siguiente(int n) {
    int suma = 0;
    while (n>0) {
        int digito = n%10;
        suma += digito*digito;
        n /= 10; // avanza de digito
    }
    return suma;
}

void psicoanaliza(int n) {
    Conjunto<int> c;
    while ( ! c.contiene(n) ) {
        cout << n << " "; // para ver los numeros por los que pasa
        c.inserta(n);
        n = siguiente(n);
    }
    if (n==1) {
        cout << "feliz (llega a 1)\n";
    } else {
        cout << n << " infeliz (repite del " << n << " en adelante)\n";
    }
}

int main() {
    psicoanaliza(7);
    psicoanaliza(38);
}
```

---

una solución que hace uso de un Conjunto de enteros

---

<sup>9</sup>Puedes leer más sobre *doxygen* en <http://www.doxygen.org>

## Notas bibliográficas

Parte de este capítulo se basa en el capítulo correspondiente de (Rodríguez Artalejo et al., 2011). Algunos ejemplos están inspirados en (Stroustrup, 1998).

## Ejercicios

### Introducción

1. Proporciona 3 ejemplos de TADs que podrían resultar útiles para implementar un juego de cartas (elige un único juego concreto para tus ejemplos). Para cada uno de esos TADs, enumera sus operaciones básicas.
2. Un diccionario de palabras permite, dada una palabra, buscar su significado. ¿Qué operaciones de tipo necesitaría un TAD Diccionario que permita tanto crear como consultar el diccionario?
3. Modifica el TAD diccionario del ejercicio anterior para que pueda almacenar más de un significado por palabra; define para ello un TAD básico Lista (que almacene sólo definiciones de tipo string), que usarás dentro del Diccionario.
4. Implementa el .h y el .cpp para un TAD Complejo que permita representar números complejos en C++. Usa float para ambas partes, real e imaginaria, y proporciona operaciones para suma, resta, multiplicación (no es necesario usar sobrecarga de operadores<sup>10</sup>), y observadores que devuelvan las partes real e imaginaria.
5. Supón que existe un TAD MultiConjunto para enteros, de forma que, dado un multiconjunto con capacidad para  $n$  elementos  $mc = \text{MultiConjunto}(n)$ ,  $mc.\text{pon}(e)$  añade un nuevo entero (descartando el más grande, si es que ya hay  $n$ ), y  $mc.\text{min}(e)$  devuelve el entero más bajo. Implementa

---

```
func menores (k: Natural; v[n] enteros) devuelve mc : MultiConjunto
{ P0 : 0 <= k < n y n >= 1 }
{ Q0 : mc contiene los k elementos menores de v[0..n-1] }
```

---

### Implementación de TADs

6. Implementa las siguientes operaciones de los conjuntos utilizando el invariante de representación (1) presentado en el texto:
  - `esVacio`, que determina si un conjunto es vacío
  - `unitario`, que crea un conjunto unitario conteniendo un elemento dado
  - `cardinal`, que devuelve el cardinal del conjunto
  - `union`, `interseccion` y `diferencia` para llevar a cabo respectivamente la unión, intersección y diferencia entre dos conjuntos.
  - el operador de igualdad de dos conjuntos, mediante la sobrecarga del operador `==`

---

<sup>10</sup>En C++, es posible sobrecargar casi todos los operadores (+, -, \*, /, ...) para tipos del usuario; por ejemplo, TAD `std::complex` de la librería estándar redefine todos estos operadores.

Proporciona el coste de todas las operaciones.

7. Implementa todas las operaciones vistas hasta el momento de los conjuntos usando los invariantes de representación (2) y (3) vistos en el texto.
8. Implementa el TAD de los polinomios con coeficientes naturales, representando un polinomio de grado  $g \sum_{i=0}^g c_i * x^i$  mediante un vector  $P[0..N]$  con  $N \geq g$  de forma que  $P[i] = c_i$  para cada  $i$ ,  $0 \leq i \leq g$ .
9. Implementa las operaciones del TAD Fecha, usando la versión que toma un `int` para indicar los segundos transcurridos desde el 1 de enero de 1970.
10. Implementa una versión genérica (usando *templates* C++) del Complejo descrito en el ejercicio 4, de forma que `Complejo<float>` sea un complejo que usa precisión sencilla, y `Complejo<double>` uno con precisión doble.
11. Implementa una versión genérica que empiece por

---

```
template <class E, int i>
class MultiConjunto {
    E _elems[i]; // vector genérico de i elementos de tipo E
    ...
};
```

---

del MultiConjunto descrito en el ejercicio 5, donde  $E$  será el tipo de elemento del que está compuesto el MultiConjunto.

12. Implementa el .h y el .cpp correspondientes a un TAD Rectangulo (alineado con los ejes), que use por debajo un TAD Punto muy básico (con un `struct` vale), y con las mismas operaciones del ejemplo visto en la teoría. Usa `float` para coordenadas y dimensiones. ¿Cuánto ocupa (en bytes) un Rectangulo si lo implementas con un punto origen, un ancho, y un alto? ¿y si lo implementas con puntos origen y extremo, entendidos como opuestos?
13. Implementa el .h y el .cpp correspondientes a un TAD TresEnRaya, que deberá contener la posición de las piezas en un tablero del juego de 3 en raya, y el turno actual. Especifica e implementa las operaciones imprescindibles para poder jugar con este tablero, y describe qué otras operaciones podrían facilitar una implementación completa del juego.
14. Implementa el .h y el .cpp correspondientes a un TAD BarajaPoker, que deberá contener las cartas correspondientes a una baraja para jugar al poker. ¿Qué partes podrían ser comunes con una baraja española?

## TADs dinámicos

15. Escribe un programa que determine cuánta memoria estática tiene disponible un PC. Para ello, usa una función recursiva infinita, que reserve memoria de 100 Kb en 100 Kb (y muestre por pantalla cuánta ha reservado) antes de volver a llamarse. El último valor mostrado, sumando un poquito más dedicado a los marcos de las funciones que se han ido llamando, corresponde al número que buscas. Ejecuta el mismo programa en otro sistema operativo y/o PC.

16. Escribe un programa que determine cuánta memoria dinámica tienes disponible, por el método de hacer infinitos `news` de arrays de 100 Kb hasta que uno de ellos falle (en este momento, finalizará el programa). Ve mostrando el progreso a medida que avancen las reservas – y ejecuta este programa en una máquina que no te moleste reiniciar, porque durante las últimas reservas se ralentizará significativamente la velocidad del sistema. Ejecuta el mismo programa en otro sistema operativo y compara los resultados.
17. Implementa un tipo “vector dinámico” de enteros llamado `Vector`. En el constructor, habrá que especificar un tamaño (también llamado dimensión) inicial. Habrá una operación de acceso `get(i)` para acceder al elemento  $i$ -ésimo, otra llamada `dimension()` para consultar la dimensión actual del vector y un mutador `set(i, valor)` para cambiar el valor del elemento  $i$ -ésimo. Lanza excepciones para evitar accesos fuera del vector. En cualquier momento, usando la operación `dimensiona(d)`, debe ser posible cambiar el tamaño del vector a uno  $d$ , que podrá ser tanto mayor como menor que el actual. En ambos casos, se deberán mantener todos los elementos que quepan. Ten cuidado con el destructor y el operador de copia por defecto.
18. Modifica el `Vector` del ejercicio anterior para que, además de enteros, acepte cualquier otro tipo de elemento; es decir, convierte a tu `Vector` en parametrizable<sup>11</sup>.
19. Modifica el código de ejemplo del Conjunto parametrizable estático para escribir un conjunto dinámico basado en el `Vector` parametrizable del ejercicio anterior. Inicialmente, usarás un `Vector` de `INICIAL` elementos (una constante, que puede ser por ejemplo 100; ya no necesitarás la constante `MAX`). Cuando se inserten muchos elementos, en lugar de lanzar excepciones de tamaño agotado, llama a  
`dimensiona(_elementos.dimension() + INICIAL).`

## Probando TADs

20. Escribe una batería de pruebas de caja negra para el TAD Pareja.
21. Escribe una batería de pruebas de caja blanca para el TAD Conjunto (estático) - haz énfasis en los aspectos que no puedes comprobar con las pruebas de caja negra: todos los casos límites de intentar más elementos de los que caben, o intentar eliminar elementos cuando no hay ninguno.
22. Escribe una batería de pruebas de caja blanca para el TAD Conjunto (dinámico, según el ejercicio 19). ¿Cómo tendrás que adaptar las pruebas del ejercicio anterior para este cambio en la implementación del TAD? ¿Cómo se te ocurre que puedes probar el caso de “pila llena”?
23. Escribe una batería de pruebas de caja negra usando `assert` para los TADs diccionario de los ejercicios 2 y 3.

<sup>11</sup>La librería estándar de C++ ya proporciona un vector dinámico parametrizable, llamado `vector`. Siempre que no sepas de antemano el tamaño de un array, y que ese tamaño va a ser constante, deberías usar `vector` (definido mediante `#include <vector>`). En general (excepto cuando el ejercicio pide lo contrario), siempre es mejor usar implementaciones de la librería estándar en lugar de las propias.

### Documentando TADs

24. Aplica el estilo de documentación propuesto en la sección de Documentando TADs a los .h de ejercicios anteriores.



---

## Capítulo 5

# Diseño e implementación de TADs lineales<sup>1</sup>

---

*Empieza por el principio –dijo el Rey con gravedad – y sigue hasta llegar al final; allí te paras.*

Lewis Carroll definiendo, sin pretenderlo, el recorrido de una estructura lineal en Alicia en el país de las maravillas.

**RESUMEN:** En este tema se presentan los TADs lineales, dando al menos una implementación de cada uno de ellos. Se presenta también el concepto de iterador que permite recorrer una colección de objetos y se extiende el TAD lista para que soporte recorrido y modificación mediante iteradores.

## 1. Motivación

El agente 0069 ha inventado un nuevo método de codificación de mensajes secretos. El mensaje original  $X$  se codifica en dos etapas:

1.  $X$  se transforma en  $X'$  reemplazando cada sucesión de caracteres consecutivos que no sean vocales por su imagen especular.
2.  $X'$  se transforma en la sucesión de caracteres  $X''$  obtenida al ir tomando sucesivamente: el primer carácter de  $X'$ , luego el último, luego el segundo, luego el penúltimo, etc.

Ejemplo: para  $X = \text{"Bond, James Bond"}$ , resultan:

$X' = \text{"BoJ ,dnameB sodn"}$

y

$X'' = \text{"BnodJo s, dBneam"}$

¿Serías capaz de implementar los algoritmos de codificación y decodificación?

---

<sup>1</sup>Marco Antonio Gómez es el autor principal de este tema.

Apostamos que sí; inténtalo. A buen seguro te dedicarás a utilizar vectores de caracteres y enteros a modo de “índice” a los mismos. En este tema aprenderás a hacerlo de una forma mucho más fácil gracias a los TADs lineales. Al final del tema vuelve a implementarlo y compara las dos soluciones.

## 2. Estructuras de datos lineales

Antes de plantearnos los distintos tipos abstractos de datos lineales nos planteamos cómo podemos guardar en memoria una colección de datos lineal. Hay dos aproximaciones básicas:

- Todos los elementos de forma consecutiva en la memoria: array de elementos.
- Elementos dispersos en memoria guardando enlaces entre ellos: listas enlazadas.

Cada una de las alternativas tiene sus ventajas y desventajas. Las implementaciones de los TADs lineales que veremos en el tema podrán hacer uso de una u otra estructura de datos; la elección de una u otra podrá influir en la complejidad de sus operaciones.

Veamos cada una de ellas. Es importante hacer notar que estamos hablando aquí de *estructuras de datos* o estrategias para almacenar información en memoria. Por eso *no* planteamos de forma exhaustiva qué operaciones vamos a tener, ni el invariante de la representación ni relación de equivalencia. Introducimos aquí simplemente los métodos típicos que las implementaciones de los TADs que hacen uso de estas estructuras de datos suelen incorporar para el manejo de la propia estructura.

### 2.1. Array de elementos

La idea fundamental es guardar todos los elementos en un array utilizando el tipo primitivo del lenguaje. Dado que un vector *no* puede cambiar de tamaño una vez creado, se impone desde el momento de la creación un *límite* en el número de elementos que se podrán almacenar; de ellos solo los  $n$  primeros tendrán información útil (el resto debe verse como espacio reservado para almacenar otros elementos en el futuro).

Para superar la limitación del tamaño fijo es habitual hacer uso de *arrays dinámicos*: se crea un *array* en la memoria dinámica capaz de albergar un número fijo de elementos; cuando el array se llena se construye un nuevo *array* más grande, se copian los elementos y se elimina el antiguo.

#### 2.1.1. Definición de tipos

Las implementaciones que quieran hacer uso de esta estructura de datos utilizan normalmente tres atributos:

- Puntero al array almacenado en memoria dinámica.
- Tamaño de ese array (o lo que es lo mismo, número de elementos que podría almacenar como máximo).
- Número de elementos ocupados actualmente. Los índices ocupados casi siempre se *condensan* al principio del array.

---

```

template <class T>
class VectorDinamico {
public:

    ...

protected:

    /** Puntero al array que contiene los datos. */
    T* array;

    /** Tamaño del vector array. */
    int capacidad;

    /** Número de elementos reales guardados. */
    int nelems;
};


```

---

### 2.1.2. Creación

La creación consiste en crear un vector con un tamaño inicial. En el código siguiente ese tamaño (definido en una constante) es 10.

---

```

template <class T>
class VectorDinamico {
public:

    /** Tamaño inicial del vector dinámico. */
    static const int TAM_INICIAL = 10;

    /** Constructor */
    VectorDinamico() :
        array(new T[TAM_INICIAL]),
        capacidad(TAM_INICIAL),
        nelems(0) {
    }

    ...

};


```

---

### 2.1.3. Operaciones sobre la estructura de datos

Las operaciones relevantes en esta estructura de datos son:

- Método para ampliar el vector: cuando el TAD quiera añadir un nuevo elemento en el vector pero esté ya lleno debe crear un vector nuevo. Para que el coste amortizado de las inserciones sea constante el tamaño del vector *se dobla*<sup>2</sup>.

---

<sup>2</sup>El coste amortizado se utiliza cuando una función presenta costes muy distintos en distintas llamadas y se quiere obtener un coste más preciso que el caso peor de la función. En ese caso se calcula el caso peor de una secuencia de llamadas a la función. Decir que una función requiere un tiempo amortizado constante significa que para cualquier secuencia de  $n$  llamadas, el tiempo total de la secuencia está acotado superiormente por  $cn$ , para una cierta constante  $c > 0$ . Se permite por tanto un tiempo excesivo para una

- Al eliminar un elemento intermedio del vector hay que *desplazar* los elementos que quedan a la derecha del eliminado.

---

```

template <class T>
class VectorDinamico {

    ...

protected:

    /**
     * Duplica el tamaño del vector.
     */
    void amplia() {
        T* viejo = array;
        capacidad *= 2;
        array = new T[capacidad];

        for (int i = 0; i < nelems; ++i)
            array[i] = viejo[i];

        delete[] viejo;
    }

    /**
     * Elimina un elemento del vector; compacta los elementos al principio del vector.
     * @param pos En el intervalo 0..numElems-1.
     */
    void quitaElem(int pos) {
        assert((0 <= pos) && (pos < nelems));

        --nelems;
        for (int i = pos; i < nelems; ++i)
            array[i] = array[i+1];
    }

};


```

---

#### 2.1.4. Destrucción

La destrucción requiere simplemente eliminar el vector creado en el constructor o en el método `amplia()`.

---

```

template <class T>
class VectorDinamico {
public:

    ...

    ~VectorDinamico() {
        delete[] array;

```

llamada sólo si se han registrado tiempos muy breves anteriormente. En el caso de la inserción, el vector se dobla tras  $n$  inserciones de coste  $\mathcal{O}(1)$ , por lo que el coste de la secuencia sería  $n * \mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(n)$ . Puede entonces considerarse que el coste amortizado de cada inserción está en  $\mathcal{O}(1)$ .

```

}
...
};
```

---

## 2.2. Nodos enlazados

En este caso cada elemento es almacenado en un espacio de memoria independiente (un *nodo*) y la colección completa se mantiene utilizando punteros. Hay dos alternativas:

- Nodos enlazados con enlace simple: cada nodo mantiene un puntero al siguiente elemento.
- Nodos enlazados con enlace doble: cada nodo mantiene dos punteros: un puntero al nodo siguiente y al nodo anterior.

Aquí aparece la implementación de esta segunda opción, por ser más versátil. No obstante en ciertas implementaciones de TADs esa versatilidad no aporta ventajas adicionales (por ejemplo en las pilas), por lo que será más eficiente (en cuanto a consumo de memoria) el uso de enlace simple.

### 2.2.1. Definición de tipos

Dependiendo del TAD que utilice esta estructura de datos, sus atributos serán distintos. Todas las implementaciones tendrán, eso sí, la definición de la clase *Nodo* que es la que almacena por un lado el elemento y por otro lado los punteros al nodo siguiente y al nodo anterior. Esa clase en C++ la implementaremos como una clase interna.

```

template <class T>
class ListaEnlazada {
public:
    ...
protected:
    /**
     * Clase nodo que almacena internamente el elemento (de tipo T),
     * y dos punteros, uno al nodo anterior y otro al nodo siguiente.
     * Ambos punteros podrían ser nullptr si el nodo es el primero
     * y/o último de la lista enlazada.
     */
    class Nodo {
public:
    Nodo() : sig(nullptr), ant(nullptr) {}
    Nodo(const T& elem) : elem(elem), sig(nullptr), ant(nullptr) {}
    Nodo(Nodo* ant, const T& elem, Nodo* sig) :
        elem(elem), sig(sig), ant(ant) {}

    T elem;
    Nodo* sig;
    Nodo* ant;
    };
};
```

---

### 2.2.2. Creación

Dado que una lista vacía no tiene ningún nodo, el proceso de creación solo implica inicializar los atributos que apuntan al primero/último de la lista a nullptr, para indicar la ausencia de elementos.

### 2.2.3. Operaciones sobre la estructura de datos

Hay dos operaciones: creación de un nuevo nodo y su inserción en la lista enlazada y eliminación.

- La inserción que implementaremos recibe dos punteros, uno al nodo anterior y otro al nodo siguiente al nodo nuevo a añadir; crea un nuevo nodo y lo devuelve. Notar que algunos de los punteros pueden ser nullptr (cuando se añade un nuevo nodo al principio o al final de la lista enlazada).
- La operación de borrado recibe únicamente el nodo a eliminar. La implementación utiliza el propio nodo para averiguar cuáles son los nodos anterior y siguiente para modificar sus punteros. Notese que si estuvieramos implementando una lista enlazada (y no doblemente enlazada) la operación necesitaría recibir un puntero al nodo anterior.

---

```
template <class T>
class ListaEnlazada {

    ...

protected:

    /**
     * Inserta un elemento entre el nodo1 y el nodo2.
     * Devuelve el puntero al nodo creado.
     * Caso general: los dos nodos existen.
     *     nodo1->sig == nodo2
     *     nodo2->ant == nodo1
     * Casos especiales: alguno de los nodos no existe
     *     nodo1 == nullptr y/o nodo2 == nullptr
    */
    static Nodo* insertaElem(const T& e, Nodo* nodo1, Nodo* nodo2) {
        Nodo* nuevo = new Nodo(nodo1, e, nodo2);
        if (nodo1 != nullptr)
            nodo1->sig = nuevo;
        if (nodo2 != nullptr)
            nodo2->ant = nuevo;
        return nuevo;
    }

    /**
     * Elimina el nodo n. Si el nodo tiene nodos antes
     * o después, actualiza sus punteros anterior y siguiente.
     * Caso general: hay nodos anterior y siguiente.
     * Casos especiales: algunos de los nodos (anterior o siguiente
     * a n) no existen.
    */
}
```

---

---

```

static void borraElem(Nodo* n) {
    assert(n != nullptr);
    Nodo* ant = n->ant;
    Nodo* sig = n->sig;
    if (ant != nullptr)
        ant->sig = sig;
    if (sig != nullptr)
        sig->ant = ant;
    delete n;
}

};


```

---

#### 2.2.4. Destrucción

La destrucción requiere ir recorriendo uno a uno todos los nodos de la lista enlazada y eliminándolos.

---

```

template <class T>
class ListaEnlazada {

    ...

protected:
 $\ast\ast$ 
Elimina todos los nodos de la lista enlazada cuyo primer nodo se pasa como parámetro.
Se admite que el nodo sea nullptr (no habrá nada que liberar). En caso de pasarse un nodo válido, su puntero al nodo anterior debe ser nullptr (si no, no sería el primero de la lista!).
 $\ast\ast$ 
static void libera(Nodo* prim) {
    assert((prim == nullptr) || (prim->ant == nullptr));

    while (prim != nullptr) {
        Nodo* aux = prim;
        prim = prim->sig;
        delete aux;
    }
}
};


```

---

Con esto terminamos el análisis de las dos estructuras de datos que utilizaremos para guardar en memoria los elementos almacenados en los TADs lineales que veremos a lo largo del tema. Aunque en las explicaciones anteriores hemos hecho uso de las clases VectorDinamico y ListaEnlazada, en la práctica tendremos las clases que implementan los distintos TADs y que tendrán los atributos o clases internas y los métodos que hemos descrito aquí.

### 2.3. En el mundo real...

Las estructuras de datos que hemos visto en este apartado se utilizarán en la implementación de los TADs que veremos a continuación. Lo mismo ocurre en las librerías de

colecciones de lenguajes como C++ o Java. Nosotros no nos preocuparemos de la reutilización aquí, por lo que el código de gestión de estas estructuras estará repetido en todas las implementaciones de los TADs que veamos. En una implementación seria esta aproximación sería inadmisible. Por poner un ejemplo de diseño correcto, la librería de C++ tiene implementadas estas dos estructuras de datos a las que llama *contenedores* (son la clase `std::vector` y `std::list`). Las implementaciones de los distintos TADs son después parametrizadas con el tipo de contenedor que se quiere utilizar. Dependiendo de la elección, la complejidad de cada operación variará.

Desde el punto de vista de la eficiencia de las estructuras de datos el vector dinámico tiene un comportamiento que no queríamos en un desarrollo serio. En concreto, la inocente:

```
array = new T[capacidad];
```

que aparece en el constructor lo que provoca es la llamada al constructor de la clase T base, de forma que cuando se construye un vector dinámico *vacío* se *crean* un puñado de elementos. Utilizando técnicas de C++ se puede retrasar la invocación a esos constructores hasta el momento de la inserción.

Peor aún es el momento del borrado de un elemento: cuando se elimina un elemento se desplaza el resto una posición hacia la izquierda pero *ese desplazamiento se realiza mediante copia*, por lo que el último elemento del vector queda *duplicado*, y no se destruirá hasta que no se elimine por completo el vector en el

```
delete[] array;
```

La última pega de los vectores dinámicos es el consumo de memoria. Los vectores crecen indefinidamente, nunca decrecen. Si un vector almacena puntualmente muchos elementos pero después se suprimen todos ellos el consumo de memoria no disminuye<sup>3</sup>. En la librería de C++ existe otro tipo de contenedor (`std::deque`) que no sufre este problema.

Por último, en nuestra implementación (y en las implementaciones de los TADs que veremos en las secciones siguientes) hemos obviado métodos recomendables (e incluso necesarios) en las clases de C++ como el constructor por copia, operaciones de igualdad, etc. En las implementaciones proporcionadas como material adicional a estos apuntes aparecerán implementados todos esos métodos, pero no los describiremos aquí por entender que son detalles de C++ no relevantes para la parte de teoría.

### 3. El TAD vector

El TAD `vector` es el contenedor de datos lineal más conocido y utilizado. Abstiene a los arrays dotándolos de gestión propia de memoria (redimensión cuando es necesario). Permite por tanto acceso directo por índice a sus elementos tanto para lectura como para modificación. Además los extiende con operaciones extra como la inserción (abriendo hueco) y eliminación (cerrándolo). A parte de dos constructoras (una que construye un vector vacío y otra que lo crea con el número de elementos indicado), la constructora por copia, operador de asignación y destructora, incluye las siguientes operaciones: `operator[]`, `push_back`, `pop_back`, `insert`, `erase`, `empty`, `size` y `resize`.

---

<sup>3</sup>Esta desventaja, no obstante, es un problema de nuestra implementación; para solucionarlo, con hacer que cuando el vector pierde un número suficiente de elementos, su tamaño se reduce automáticamente en una función inversa a `amplia()`. Si se hace un análisis del coste amortizado similar al utilizado para la inserción se llegaría a la misma conclusión: las operaciones siguen siendo  $\mathcal{O}(1)$ .

### 3.1. Implementación

Utilizaremos como estructura de datos un puntero al array dinámico redimensionable (según lo visto en el apartado 2.1), que almacena los elementos del vector, y dos enteros: el número de elementos almacenados y la capacidad actual del array (como vimos en la sección 2.1).

El *invariante de la representación*, o lo que es lo mismo las condiciones que deben cumplir los objetos de la clase para considerarse válidos, para un vector  $v$  cuyos elementos son de tipo  $T$  es:

$$\begin{aligned} R_{vector_T}(v) \\ \iff_{def} \\ 0 \leq v.\text{nelems} \leq v.\text{capacidad} \wedge \\ \forall i : 0 \leq i < v.\text{nelems} : R_T(v.\text{array}[i]) \end{aligned}$$

También es lícito plantearnos cuándo dos objetos que utilizan la misma implementación representan a vectores idénticos. Es lo que se llama relación de equivalencia que permite averiguar si dos objetos válidos (es decir, que cumplen el invariante de la representación) son iguales. En este caso concreto dos vectores son iguales si el número de elementos almacenados coincide y sus valores respectivos, uno a uno, también (independientemente de la capacidad interna que tengan):

$$\begin{aligned} v1 \equiv_{vector_T} v2 \\ \iff_{def} \\ v1.\text{nelems} = v2.\text{nelems} \wedge \\ \forall i : 0 \leq i < v1.\text{nelems} : v1.\text{array}[i] \equiv_T v2.\text{array}[i] \end{aligned}$$

La implementación quedaría como sigue:

---

```
template <class T>
class vector {
protected:
    static const int TAM_INI = 10; // tamaño por defecto del array dinámico

    // número de elementos reales del vector
    int nelems;

    // tamaño del array
    int capacidad;

    // puntero al array que contiene los datos (redimensionable)
    T* array;

public:

    // constructor: vector vacío
    vector() : nelems(0), capacidad(TAM_INI), array(new T[capacidad]) {}

    // constructor: vector con n elementos ocupados y capacidad n
    vector(int n) : nelems(n), capacidad(n), array(new T[capacidad]) {}
```

---

```

// destructor
~vector() {
    libera();
}

// constructor por copia
vector(vector<T> const& other) {
    copia(other);
}

// operador de asignación
vector<T>& operator= (vector<T> const& other) {
    if (this != &other) {
        libera();
        copia(other);
    }
    return *this;
}

// añade un elemento al final. O(1) amortizado
void push_back(T const& elem) {
    if (nelems == capacidad)
        amplia(capacidad*2);
    array[nelems] = elem;
    ++nelems;
}

// quita el último elemento. O(1)
void pop_back() {
    if (nelems > 0) --nelems;
}

// devuelve referencia constante a elemento en pos i. O(1)
const T& operator[](int i) const {
    if (i < 0 || i >= nelems)
        throw std::out_of_range("indice no valido");
    return array[i];
}

// devuelve referencia a elemento en posición i. O(1)
T& operator[](int i) {
    if (i < 0 || i >= nelems)
        throw std::out_of_range("indice no valido");
    return array[i];
}

// inserta e en pos i desplazando para abrir hueco. O(nelems)
void insert(const T& e, int i) {
    if (i < 0 || i >= nelems)
        throw std::out_of_range("indice no valido");
    if (nelems == capacidad) amplia(capacidad*2);
    desplazaDchaDesde(i);
    array[i] = e;
    ++nelems;
}

```

```

// borra elemento de pos i (desplaza para cerrar hueco). O(nelems)
void erase(int i) {
    if (i < 0 || i >= nelems)
        throw std::out_of_range("indice no valido");
    desplazaIzdaDesde(i);
    --nelems;
}

// consulta si el vector es vacío. O(1)
bool empty() const {
    return nelems == 0;
}

// consulta el tamaño del vector. O(1)
int size() const {
    return nelems;
}

// cambia tamaño del vector dando mas capacidad si necesario. O(nelems)
void resize(int n) {
    if (n < 0) throw std::out_of_range("parametro no valido");
    if (n <= nelems) nelems = n;
    // No se redimensiona hacia abajo, aunque se podría hacer
    else {
        amplia(n);
        nelems = n;
    }
}

protected:

void libera() {
    delete[] array;
}

// this está sin inicializar
void copia(vector const& other) {
    capacidad = other.nelems + TAM_INI;
    nelems = other.nelems;
    array = new T[capacidad];
    for (int i = 0; i < nelems; ++i)
        array[i] = other.array[i];
}

void amplia(int nuevaCapacidad) {
    T* viejo = array;
    capacidad = nuevaCapacidad;
    array = new T[capacidad];
    for (int i = 0; i < nelems; ++i)
        array[i] = std::move(viejo[i]);
    delete[] viejo;
}

void desplazaDchaDesde(int i){
    for (int j = nelems; j > i; j--)
        array[j] = array[j-1];
}

```

```

    }

void desplazaIzdaDesde(int i) {
    for (; i < nelems-1; i++)
        array[i] = array[i+1];
}

;

```

---

## 4. Pilas

Una pila representa una colección de valores donde es posible acceder al último elemento añadido, implementando la idea intuitiva de *pila* de objetos.

El TAD *pila* tiene dos operaciones generadoras: la que crea una pila vacía y la que apila un nuevo elemento en una pila dada. Tiene además una operación modificadora que permite desapilar el último objeto (y que es parcial, pues si la pila está ya vacía falla) y al menos dos operaciones observadoras: la que permite acceder al último elemento añadido (también parcial) y la que permite averiguar si una pila tiene elementos.

Las pilas tienen muchas utilidades, como por ejemplo “dar la vuelta” a una secuencia de datos (ver ejercicio 1).

### 4.1. Implementación de pilas con vector dinámico

En esta implementación se utiliza como estructura de datos un vector dinámico que almacena en sus primeras posiciones los elementos almacenados en la pila y que duplica su tamaño cuando se llena (según lo visto en el apartado 2.1).

Por lo tanto, el tipo representante tendrá tres atributos: el puntero al vector, el número de elementos almacenados y el tamaño máximo del vector que ya utilizamos en la sección 2.1 llamados `array`, `nelems` y `capacidad`. Notar que `array[nelems-1]` es el elemento de la cima de la pila.

El *invariante de la representación*, o lo que es lo mismo las condiciones que deben cumplir los objetos de la clase para considerarse válidos, para una pila `p` cuyos elementos son de tipo `T` es:

$$\begin{aligned}
 & R_{Stack_T}(p) \\
 \iff_{def} & 0 \leq p.nelems \leq p.capacidad \wedge \\
 & \forall i : 0 \leq i < nelems : R_T(p.array[i])
 \end{aligned}$$

También es lícito plantearnos cuándo dos objetos que utilizan la misma implementación representan a pilas idénticas. Es lo que se llama relación de equivalencia que permite averiguar si dos objetos válidos (es decir, que cumplen el invariante de la representación) son iguales. En este caso concreto dos pilas son iguales si el número de elementos almacenados coincide y sus valores respectivos, uno a uno, también:

$$\begin{aligned}
 p1 &\equiv_{Stack_T} p2 \\
 \iff_{def} & \\
 p1.\text{nelems} &= p2.\text{nelems} \wedge \\
 \forall i : 0 \leq i < p1.\text{nelems} : p1.\text{array}[i] &\equiv_T p2.\text{array}[i]
 \end{aligned}$$

La implementación es sencilla basándonos en los métodos vistos anteriormente<sup>4</sup>:

```

template <class T>
class stack {
protected:
    static const int TAM_INICIAL = 10; // tamaño inicial del array dinámico

    // número de elementos en la pila
    int nelems;

    // tamaño del array
    int capacidad;

    // puntero al array que contiene los datos (redimensionable)
    T* array;

public:

    // constructor: pila vacía
    stack() : nelems(0), capacidad(TAM_INICIAL), array(new T[capacidad]) {}

    // destructor
    ~stack() {
        libera();
    }

    // apilar un elemento
    void push(T const& elem) {
        array[nelems] = elem;
        ++nelems;
        if (nelems == capacidad)
            amplia();
    }

    // consultar la cima
    T const& top() const {
        if (empty())
            throw std::domain_error("la pila vacía no tiene cima");
        return array[nelems - 1];
    }

    // desapilar el elemento en la cima
    void pop() {
        if (empty())
            throw std::domain_error("desapilando de la pila vacía");
    }
}

```

<sup>4</sup>Durante todo el tema utilizaremos métodos que ya han aparecido anteriormente; en una implementación *desde cero* es posible que el código de algunos de esos métodos apareciera directamente integrado en las operaciones en vez de utilizar un método auxiliar.

```

    --nelems;
}

// consultar si la pila está vacía
bool empty() const {
    return nelems == 0;
}

// consultar el tamaño de la pila
int size() const {
    return nelems;
}

protected:

    ...
};
```

---

## 4.2. Implementación de pilas con una lista enlazada

La implementación con listas enlazadas consiste en almacenar como primer elemento de la lista el que aparece en la *cima* de la pila. La base de la pila se guarda en el último elemento de la lista. De esta forma:

- La clase `stack` necesita un único atributo: un puntero al nodo que contiene la cima (*cima*). Si la pila está vacía, el puntero valdrá `nullptr`.
- Dado que lo único que hacemos con la lista es insertar y borrar el primer elemento las listas enlazadas simples son suficiente.

El invariante debe garantizar que la secuencia de nodos termina en `nullptr` (eso garantiza que no hay ciclos) y que todos los nodos deben estar correctamente ubicados y almacenar un elemento del tipo base válido:

$$\begin{aligned}
 R_{Stack_T}(p) \\
 \iff_{def} \\
 \text{null} \in \text{cadena}(p.\text{cima}) \wedge \\
 \forall n \in \text{cadena}(p.\text{cima}) : n \neq \text{null} \rightarrow (\text{ubicado}(n) \wedge R_T(n.\text{elem}))
 \end{aligned}$$

Donde `ubicado` es un predicado que viene a asegurar que se ha pedido memoria para el puntero (y ésta no ha sido liberada) y `cadena(ptr)` representa el conjunto de todos los nodos que forman la lista enlazada que comienza en `ptr`, incluido el posible “nodo nulo” representado por el valor `null`:

$$\begin{array}{ll}
 \text{cadena}(ptr) = \{\text{null}\} & \text{si } ptr = \text{null} \\
 \text{cadena}(ptr) = \{ptr\} \cup \text{cadena}(ptr.\text{sig}) & \text{si } ptr \neq \text{null}
 \end{array}$$

Tras el invariante, definimos la relación de equivalencia en la implementación: dos objetos pila serán iguales si su lista enlazada contiene el mismo número de elementos y sus valores uno a uno coinciden (están en el mismo orden):

$$\begin{aligned}
 p1 &\equiv_{Stack_T} p2 \\
 \iff_{def} & iguales_T(p1.cima, p2.cima)
 \end{aligned}$$

donde

$$\begin{array}{lll}
 \text{iguales(ptr1, ptr2) = true} & \text{si } \text{ptr1} = \text{null} \wedge \text{ptr2} = \text{null} \\
 \text{iguales(ptr1, ptr2) = false} & \text{si } (\text{ptr1} = \text{null} \wedge \text{ptr2} \neq \text{null}) \vee \\
 & (\text{ptr1} \neq \text{null} \wedge \text{ptr2} = \text{null}) \\
 \text{iguales(ptr1, ptr2) = ptr1.elem} \equiv_T \text{ptr2.elem} \wedge & \text{si } (\text{ptr1} \neq \text{null} \wedge \text{ptr2} \neq \text{null}) \\
 \text{iguales(ptr1.sig, ptr2.sig)} &
 \end{array}$$

La implementación aparece a continuación; el nombre de la clase lo hemos cambiado a `LinkedListStack` (pila implementada con listas enlazadas). En aras de la simplicidad omitimos los comentarios de los métodos (al implementar el mismo TAD y ser una implementación sin limitaciones, coinciden con los de la implementación con array dinámico):

---

```

template <class T>
class LinkedListStack {
public:

    LinkedListStack() : cima(nullptr), nelems(0) {
    }

    ~LinkedListStack() {
        libera();
        cima = nullptr;
    }

    void push(const T& elem) {
        cima = new Nodo(elem, cima);
        nelems++;
    }

    void pop() {
        if (empty())
            throw std::domain_error("desapilando de la pila vacia");
        Nodo* aBorrar = cima;
        cima = cima->sig;
        delete aBorrar;
        --nelems;
    }

    const T& top() const {
        if (empty())
            throw std::domain_error("la pila vacia no tiene cima");
        return cima->elem;
    }

    bool empty() const {
        return cima == nullptr;
    }
}

```

---

```

// consultar el tamaño de la pila
int size() const {
    return nelems;
}

protected:
    /**
     Clase nodo que almacena internamente el elemento (de tipo T),
     y un puntero al nodo siguiente, que podría ser nullptr si
     el nodo es el último de la lista enlazada.
    */
class Nodo {
public:
    Nodo() : sig(nullptr) {}
    Nodo(const T& elem) : elem(elem), sig(nullptr) {}
    Nodo(const T& elem, Nodo* sig) :
        elem(elem), sig(sig) {}

    T elem;
    Nodo* sig;
};

Nodo* cima;
int nelems;
};

```

La complejidad de las operaciones de ambas implementaciones es similar<sup>5</sup>:

| Operación | Vectores         | Listas enlazadas |
|-----------|------------------|------------------|
| stack     | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| push      | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| pop       | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| top       | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| empty     | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| size      | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |

## 5. Colas

Las colas son TADs lineales que permiten introducir elementos por un extremo (el *final* de la cola) y las consultas y eliminaciones por el otro (el *inicio* de la cola). Es decir, son estructuras en las que el primer elemento que entra es el primero que saldrá; de ahí que también se las conozca como estructuras FIFO (del inglés, *first in, first out*).

Las operaciones de las colas son:

- **Constructora:** genera una cola vacía.
- **push:** añade un nuevo elemento a la cola.

---

<sup>5</sup>En todas las tablas de complejidades de operaciones de TADs que pondremos *asumiremos* que el tipo base tiene operaciones de construcción, destrucción y copia constantes  $\mathcal{O}(1)$ .

- `pop`: modificadora parcial que elimina el primer elemento de la cola. Falla si la cola está vacía.
- `front`: observadora parcial que devuelve el primer elemento de la cola (el más antiguo).
- `empty`: también observadora, permite averiguar si la cola tiene elementos.
- `size`: también observadora, devuelve el número de elementos de la cola.

### 5.1. Implementación de colas con un vector

Una posible implementación con un vector dinámico consistiría en hacer crecer el vector al ir llegando elementos, de forma que el primer elemento de la cola esté siempre en la posición 0 del vector.

La principal pega que tiene esa implementación es el coste de la operación `pop`: al eliminar el elemento debemos desplazar todos los elementos válidos una posición a la izquierda, elevando el coste de la operación a  $\mathcal{O}(n)$ .

El invariante de la representación y la relación de equivalencia de esta implementación es análoga a aquella vista para las pilas.

Existe una implementación sobre vectores más eficiente en la que la complejidad de la operación es  $\mathcal{O}(1)$ ; ver el ejercicio 12.

### 5.2. Implementación de colas con una lista enlazada

Otra posible implementación de las colas es utilizando un esquema similar a aquél visto en las pilas: una lista enlazada en la que el primer nodo contiene el elemento que hay en la cabecera de la cola. Igual que ocurría en las pilas esa lista puede ser simple para ahorrarnos el espacio en memoria ocupado por los punteros a los nodos anteriores que no necesitamos.

Si hiciéramos la implementación nos daríamos cuenta de que la operación `push` tiene complejidad  $\mathcal{O}(n)$  pues debemos recorrer todos los elementos almacenados en la lista para saber dónde colocar el nuevo nodo.

La forma de solucionarlo es hacer que la implementación almacene *dos punteros*: un puntero al primer nodo y otro puntero al último nodo.

El invariante de la representación es similar al visto en la implementación de las pilas, y la relación de equivalencia también.

La implementación es en cierto sentido algo incómoda pues tenemos que tener cuidado con los casos especiales en los que trabajamos con una cola vacía<sup>6</sup>.

---

```
template <class T>
class queue {
protected:

    // punteros al primer y último elemento
    Nodo* prim;
    Nodo* ult;

    // número de elementos en la cola
    int nelems;
```

---

<sup>6</sup>En el código no se aprecian todos los casos especiales gracias a las operaciones auxiliares implementadas en la sección 2.2 y que no mostramos de nuevo aquí.

```

public:

    // constructor: cola vacía
    queue() : prim(nullptr), ult(nullptr), nelems(0) {}

    // destructor
    ~queue() {
        libera();
    }

    // añadir un elemento al final de la cola
    void push(T const& elem) {
        Nodo* nuevo = new Nodo(elem);

        if (ult != nullptr)
            ult->sig = nuevo;
        ult = nuevo;
        if (prim == nullptr) // la cola estaba vacía
            prim = nuevo;
        ++nelems;
    }

    // consultar el primero de la cola
    T const& front() const {
        if (empty())
            throw std::domain_error("la cola vacia no tiene primero");
        return prim->elem;
    }

    // eliminar el primero de la cola
    void pop() {
        if (empty())
            throw std::domain_error("eliminando de una cola vacia");
        Nodo* a_borrar = prim;
        prim = prim->sig;
        if (prim == nullptr) // la cola se ha quedado vacía
            ult = nullptr;
        delete a_borrar;
        --nelems;
    }

    // consultar si la cola está vacía
    bool empty() const {
        return nelems == 0;
    }

    // consultar el tamaño de la cola
    int size() const {
        return nelems;
    }
};

```

La complejidad de las operaciones es distinta dependiendo de la estructura utilizada:

| Operación | Vectores         | Vectores circulares | Listas enlazadas |
|-----------|------------------|---------------------|------------------|
| queue     | $\mathcal{O}(1)$ | $\mathcal{O}(1)$    | $\mathcal{O}(1)$ |
| push      | $\mathcal{O}(1)$ | $\mathcal{O}(1)$    | $\mathcal{O}(1)$ |
| front     | $\mathcal{O}(1)$ | $\mathcal{O}(1)$    | $\mathcal{O}(1)$ |
| pop       | $\mathcal{O}(n)$ | $\mathcal{O}(1)$    | $\mathcal{O}(1)$ |
| empty     | $\mathcal{O}(1)$ | $\mathcal{O}(1)$    | $\mathcal{O}(1)$ |
| size      | $\mathcal{O}(1)$ | $\mathcal{O}(1)$    | $\mathcal{O}(1)$ |

### 5.3. Implementación de colas con una lista enlazada y nodo fantasma

Para evitar los casos especiales que teníamos antes en los que había que tener en cuenta que la cola podía estar o no vacía se puede utilizar un *nodo fantasma*: un nodo extra al principio de la lista enlazada que hace de *cabecera* especial que siempre tenemos, que no guarda ningún elemento, pero nos permite no tener que distinguir el caso en el que `_prim es nullptr` (cola vacía). La complejidad de las operaciones no varía, pero su programación es más sencilla; puedes comprobarlo realizando la implementación. La misma técnica la utilizaremos a continuación para las colas dobles.

## 6. Colas dobles

Las colas dobles son una generalización de las colas en donde se pueden añadir, quitar y consultar elementos en los dos extremos. En concreto las operaciones serán:

- **Constructora:** Genera una cola doble vacía.
- **push\_back:** Añade un nuevo elemento al final.
- **push\_front:** Añade un nuevo elemento al principio.
- **pop\_front:** Modificadora parcial que elimina el primer elemento de la cola. Lanza excepción si la cola está vacía.
- **pop\_back:** Modificadora parcial que elimina el último elemento de la cola. Lanza excepción si la cola está vacía.
- **front:** Observadora parcial que devuelve el primer elemento de la cola. Lanza excepción si la cola está vacía.
- **back:** Observadora parcial que devuelve el último elemento de la cola. Lanza excepción si la cola está vacía.
- **empty:** Observadora que permite averiguar si la cola tiene elementos.
- **size:** Observadora que devuelve el número de elementos de la cola.

Dada su semejanza con las colas de la sección anterior existen opciones similares para la implementación, aunque en el caso de la implementación utilizando nodos, es preferible el uso de listas doblemente enlazadas frente a las simples.

No obstante las implementaciones requieren el doble de cuidado que las del TAD hermano de las colas, pues necesitamos cubrir más casos especiales. Es por eso que la opción de implementación con una lista enlazada circular y nodo fantasma aparece como la mejor candidata.

En concreto la cola vacía estará representada por un nodo fantasma que no contiene ningún elemento y cuyos punteros anterior y siguiente apuntan a él mismo. Ese nodo fantasma, que se crea al principio y no se borrará hasta el final, es apuntado por el único atributo de la clase. La implementación hará que *el siguiente* al nodo fantasma sea el primero de la cola (la cabecera), mientras que el anterior será el último.

La relación de equivalencia no es muy diferente de la vista para las pilas y colas; el único cambio es la condición que marca el final de la recursión en el predicado *iguales* que utilizábamos entonces. Ahora no debemos terminar cuando se llega al `null`, sino cuando volvemos al nodo de partida. Para eso extendemos los parámetros de *iguales* para que además de tener los nodos de partida tenga también como parámetros los nodos donde la recursión debe terminar.

$$\begin{aligned}
 c1 \equiv_{Deque_T} c2 \\
 \iff_{def} \\
 & iguales_T( \\
 & \quad c1.fantasma.sig, c1.fantasma.fin, \\
 & \quad c2.fantasma.sig, c2.fantasma.fin \\
 & )
 \end{aligned}$$

donde

$$\begin{array}{ll}
 iguales(i1, e1, i2, e2) = \text{true} & \mathbf{si} \quad i1 = e1 \wedge i2 = e2 \\
 iguales(i1, e1, i2, e2) = \text{false} & \mathbf{si} \quad (i1 = e1 \wedge i2 \neq e2) \vee \\
 & \quad (i1 \neq e1 \wedge i2 = e2) \\
 iguales(i1, e1, i2, e2) = i1.elem \equiv_T i2.elem \wedge & \mathbf{si} \quad (i1 \neq e1 \wedge i2 \neq e2) \\
 & iguales(i1.sig, e1, i2.sig, e2)
 \end{array}$$

Por su parte el invariante de la representación tiene que incorporar también la circularidad de la lista. Para la definición del invariante utilizamos un conjunto similar al *cadena* utilizado en las pilas pero que indica qué elementos son alcanzables desde un nodo dado si seguimos su puntero *sig* por un lado y *ant* por otro. Es claro que:

- El conjunto de nodos alcanzables desde el nodo cabecera por un lado y por otro debe ser el mismo.
- Dado que la lista es circular, el nodo cabecera debe aparecer en el conjunto de nodos alcanzables a partir de él.
- Todos esos nodos deben estar ubicados y tener los enlaces al nodo anterior y al nodo siguiente correctos (lo que implica que si vamos al nodo anterior de *n* y luego pasamos a su siguiente deberíamos volver a *n* y al contrario).
- Por último, todos los nodos (excepto el nodo cabecera) deben contener elementos válidos del tipo base.

Con esta idea, el invariante de la representación queda (por comodidad en las definiciones siguientes se entenderá *ini* como *c.fantasma*):

$$\begin{aligned}
 R_{Deque_T}(c) \\
 \iff_{def} & alcanzables(ini) = alcanzablesHaciaAtras(ini) \wedge \\
 & ini \in alcanzables(ini) \wedge \\
 & \forall p \in alcanzables(ini) : \\
 & \quad ( \\
 & \quad \quad ubicado(p) \wedge buenEnlace(p) \wedge \\
 & \quad \quad (p \neq ini \rightarrow R_T(p.elem)) \\
 & \quad )
 \end{aligned}$$

Donde, como hemos dicho antes, *alcanzables* es el conjunto de todos los nodos que pueden alcanzarse desde un nodo dado utilizando el puntero *sig* y *alcanzablesHaciaAtras* utilizando *ant*. Por último, *buenEnlace* indica si los punteros son correctos desde el punto de vista de una lista doblemente enlazada:

$$\begin{aligned}
 alcanzables(p) &= \emptyset & \text{si } p = \text{null} \\
 alcanzables(p) &= \{p.sig\} \cup alcanzables(p.sig) & \text{si } p \neq \text{null} \\
 \\ 
 alcanzablesHaciaAtras(p) &= \emptyset & \text{si } p = \text{null} \\
 alcanzablesHaciaAtras(p) &= \{p.ant\} \cup alcanzablesHaciaAtras(p.ant) & \text{si } p \neq \text{null} \\
 \\ 
 buenEnlace(p) &= p.sig.ant = p \wedge p.ant.sig = p
 \end{aligned}$$

Observa que la implementación crea en el momento de su construcción el nodo “fantasma” y que en la destrucción se rompe la circularidad de la lista para poder utilizar *libera* sin riesgo a entrar en bucles infinitos. Por otro lado verás que la implementación de las operaciones no tienen que preocuparse de casos especiales (más allá de las precondiciones de *esVacia*), haciendo la implementación casi trivial:

---

```

template <class T>
class deque {
    protected:

        // puntero al nodo fantasma
        Nodo* fantasma;

        // n mero de elementos
        int nelems;

    public:

        // constructor: cola doble vac a
        deque() : fantasma(new Nodo()), nelems(0) {
            fantasma->sig = fantasma->ant = fantasma; // circular
        }

        // destructor
    }
}

```

```

~deque() {
    libera();
}

// añadir un elemento por el principio
void push_front(T const& e) {
    inserta_elem(e, fantasma, fantasma->sig);
}

// añadir un elemento por el final
void push_back(T const& e) {
    inserta_elem(e, fantasma->ant, fantasma);
}

// consultar el primer elemento de la dcola
T const& front() const {
    if (empty())
        throw std::domain_error("la dcola vacia no tiene primero");
    return fantasma->sig->elem;
}

// consultar el último elemento de la dcola
T const& back() const {
    if (empty())
        throw std::domain_error("la dcola vacia no tiene ultimo");
    return fantasma->ant->elem;
}

// eliminar el primer elemento
void pop_front() {
    if (empty())
        throw std::domain_error("eliminando el primero de una dcola vacia");
    borra_elem(fantasma->sig);
}

// eliminar el último elemento
void pop_back() {
    if (empty())
        throw std::domain_error("eliminando el ultimo de una dcola vacia");
    borra_elem(fantasma->ant);
}

// consultar si la dcola está vacía
bool empty() const {
    return nelems == 0;
}

// consultar el tamaño de la cola doble
int size() const {
    return nelems;
}

protected:

// insertar un nuevo nodo entre anterior y siguiente
Nodo* inserta_elem(T const& e, Nodo* anterior, Nodo* siguiente) {

```

```

Nodo* nuevo = new Nodo(e, anterior, siguiente);
anterior->sig = nuevo;
siguiente->ant = nuevo;
++nelems;
return nuevo;
}

// eliminar el nodo n
void borra_elem(Nodo* n) {
    assert(n != nullptr);
    n->ant->sig = n->sig;
    n->sig->ant = n->ant;
    delete n;
    --nelems;
}

};


```

La complejidad de las operaciones en esta implementación es:

| Operación  | Listas enlazadas |
|------------|------------------|
| deque      | $\mathcal{O}(1)$ |
| push_back  | $\mathcal{O}(1)$ |
| front      | $\mathcal{O}(1)$ |
| pop_front  | $\mathcal{O}(1)$ |
| push_front | $\mathcal{O}(1)$ |
| back       | $\mathcal{O}(1)$ |
| pop_back   | $\mathcal{O}(1)$ |
| empty      | $\mathcal{O}(1)$ |
| size       | $\mathcal{O}(1)$ |

## 7. Listas

Las listas son los TADs lineales más generales posibles<sup>7</sup>. Como las colas dobles, permiten la consulta, inserción y eliminación por los dos extremos, pero también permiten acceder a cualquier punto intermedio tanto para consultar como también para eliminar e insertar en él. Partiremos por tanto de la interfaz e implementación de las colas dobles mediante herencia:

---

```

template <class T>
class list : public deque<T> {
    ...
};


```

---

Obsérvese que dado que las listas son los TADs lineales más generales es posible desarrollar implementaciones del resto de TADs lineales basándose directamente en las listas. Ver el ejercicio 16.

---

<sup>7</sup>No confundir el TAD *lista* con la estructura de datos *lista enlazada*; las listas enlazadas deben verse como un método de organizar en memoria una colección de elementos; el TAD *lista* es un tipo abstracto de datos con una serie de operaciones que puede implementarse utilizando listas enlazadas pero también otro tipo de estructuras de datos, como los vectores dinámicos.

Para poder acceder a puntos intermedios de la lista una primera idea consistiría en proporcionar la operación `at` (análoga a la operación del mismo nombre en el TAD `vector`) que nos devuelva una referencia al elemento en la posición  $i$ -ésima de la lista.

---

```
/**  
 * Devuelve el elemento  $i$ -ésimo de la lista, teniendo en cuenta que  
 * el primer elemento (first()) es el elemento 0 y el último es  
 * size()-1, es decir  $idx$  está en  $[0..size()-1]$ . Operación parcial que  
 * puede fallar si se da un índice incorrecto. El índice es entero sin  
 * signo, para evitar que se puedan pedir elementos negativos.  
 */  
T& at(unsigned int idx) const {  
    if (idx >= nElems)  
        throw std::out_of_range("Indice no valido");  
    Nodo* aux = fantasma->sig;  
    for (int i = 0; i < idx; ++i)  
        aux = aux->sig;  
    return aux->elem;  
}
```

---

Sin embargo un uso descuidado de esta operación podría llevar a situaciones no deseadas en cuanto a ineficiencia. Por ejemplo, la complejidad de un bucle tan inocente como el siguiente:

```
list<int> l;  
  
...  
  
for (int i = 0; i < l.size(); ++i)  
    std::cout << l.at(i) << ' ';
```

---

que simplemente escribe uno a uno todos los elementos *no* tiene coste lineal sino cuadrático! Por esta razón algunas implementaciones reales de las listas (como es el caso de la clase `list` de la STL de C++) de hecho no incluyen esta operación.

## 7.1. Iteradores

La solución adoptada de manera estandarizada para recorrer estructuras de datos y también para acceder a (e insertar y eliminar en) puntos intermedios son los *iteradores*. Entenderemos un iterador como un objeto de una clase que:

- Representa un punto intermedio en el recorrido de una colección de datos (una lista en este caso).
- Tiene un método que devuelve el elemento por el que va el recorrido (y tendrá el tipo base utilizado en la colección). La operación será *parcial*, pues fallará si el recorrido ya ha terminado. En C++ este método se implementa mediante el operador `*` por su analogía con lo que sería acceder al contenido de una variable de tipo puntero.
- Tiene un método que hace que el iterador pase al siguiente elemento del recorrido. En C++ lo estándar es sobrecargar el operador `++`.
- Tiene implementada la operación de comparación, de forma que se puede saber si dos iteradores son iguales. Dos iteradores son iguales si: representan el mismo punto en el recorrido de una lista concreta o los dos representan el final del recorrido.

Extenderemos el TAD `list` para que proporcione dos operaciones adicionales:

- `begin()`: devuelve un iterador inicializado al primer elemento del recorrido<sup>8</sup>
- `end()`: devuelve un iterador apuntando *fueras* del recorrido, es decir un iterador cuya operación `*` *falla*.

Haciendo que la operación `end()` devuelva un iterador *no válido* implica que los elementos válidos de la lista son el *intervalo abierto* [ `begin()`, `end()` ), y la forma de recorrer una lista sería por tanto (en este caso simplemente para imprimir cada elemento):

---

```
list<int> l;
...
for (list<int>::iterator it = l.begin(); it != l.end(); ++it) {
    cout << *it << endl;
}
```

---

Desde el estándar *C++11* este tipo de recorridos también puede escribirse mediante un bucle de tipo *range-based for* de esta forma:

---

```
list<int> l;
...
for (int e : l) {
    cout << e << endl;
}
```

---

Este tipo de bucle puede interpretarse como “para cada elemento *e* en el recorrido usando el iterador de la estructura *l*”. Aunque esta forma es más cómoda es también menos expresiva que la forma anterior usando explícitamente objetos iteradores.

## 7.2. Implementación de un iterador básico

La implementación se basa en la existencia de la clase interna y protegida `Iterador` la cual es instanciada correspondientemente para los iteradores constantes y no-constantes en los tipos públicos `const_iterator` e `iterator` respectivamente. Como podemos observar tiene como atributos un puntero al nodo actual en el recorrido (`act`) y otro al nodo fantasma `fan`, para poder saber cuándo estamos fuera del recorrido.

---

```
template <class T>
class list : public deque<T> {
protected:
    using Nodo = typename deque<T>::Nodo;

    template <class Apuntado>
    class Iterador {
        // puntero al nodo actual del recorrido
        Nodo* act;
        // puntero al nodo fantasma (para saber cuándo estamos fuera)
```

---

<sup>8</sup>También incluiremos una versión constante llamada `cbegin`, que se utilizará en casos en los que queramos (o necesitemos) indicar que el iterador no permite hacer modificaciones sobre la lista durante el recorrido.

```

Nodo* fan;

public:

Iterador() : act(nullptr), fan(nullptr) {}

// para acceder al elemento apuntado
Apuntado& operator*() const {
    if (act == fan) throw std::out_of_range("fuera de la lista");
    return act->elem;
}

Iterador& operator++() { // ++ prefijo (recomendado)
    if (act == fan) throw std::out_of_range("fuera de la lista");
    act = act->sig;
    return *this;
}

bool operator==(Iterador const& that) const {
    return act == that.act && fan == that.fan;
}

bool operator!=(Iterador const& that) const {
    return !(*this == that);
}

private:
// para que list pueda construir objetos del tipo iterador
friend class list;

// constructora privada
Iterador(Nodo* ac, Nodo* fa) : act(ac), fan(fa) {}
}; // Fin de la clase interna Iterador

public: // de la clase list
/*
Iteradores que permiten recorrer la lista pero no cambiar sus elementos.
*/
using const_iterator = Iterador<T const>;

// devuelven un iterador constante al principio de la lista
const_iterator cbegin() const {
    return const_iterator(this->fantasma->sig, this->fantasma);
}

// devuelven un iterador constante al final del recorrido y fuera de este
const_iterator cend() const {
    return const_iterator(this->fantasma, this->fantasma);
}

/*
Iteradores que permiten recorrer la lista y cambiar sus elementos.
*/
using iterator = Iterador<T>;

// devuelve un iterador al principio de la lista

```

---

```

iterator begin() {
    return iterator(this->fantasma->sig, this->fantasma);
}

// devuelve un iterador al final del recorrido y fuera de este
iterator end() {
    return iterator(this->fantasma, this->fantasma);
}
};

```

---

Obsérvese que mediante un iterador no-constante podemos hacer recorridos que vayan alterando la lista. Por ejemplo, la siguiente función incrementa cada elemento de una lista de enteros.

---

```

list<int> l;
...
for (list<int>::iterator it = l.begin(); it != l.end(); ++it) {
    (*it)++;
}

```

---

Usando un bucle de tipo *range-based-for* quedaría de esta forma:

---

```
for (int& e : l) e++;
```

---

En este caso es necesario el & para indicar que cada elemento del recorrido se debe capturar en la variable *e* por referencia.

### 7.3. Usando iteradores para insertar elementos

El TAD `list` puede extenderse para permitir insertar elementos en medio de la lista. Para eso se puede utilizar el mecanismo de iteradores: la operación recibirá un iterador situado en el punto de la lista donde se desea insertar un elemento y el elemento a insertar. En concreto, el elemento lo añadiremos *a la izquierda* del punto marcado. Eso significa que si insertamos un elemento a partir de un iterador colocado al principio del recorrido, el nuevo elemento añadido pasará a ser el primero de la lista y el iterador apunta al segundo. Si el iterador está al final del recorrido (en `end()`), el elemento insertado será el nuevo último elemento de la lista, y el iterador sigue apuntando al `end()`, es decir por el hecho de insertar, la posición del iterador no cambia.

Por ejemplo, la siguiente función duplica todos los elementos de la lista, de forma que si el contenido inicial era por ejemplo [1, 3, 4] al final será [1, 1, 3, 3, 4, 4]:

---

```

void repiteElementos(list<int>& lista) {

    list<int>::iterator it = lista.begin();
    while (it != lista.end()) {
        lista.insert(it, *it);
        ++it;
    }
}

```

---

La implementación de la operación `insert` de la lista, por tanto, recibe el iterador que marca el lugar de la inserción y el elemento a insertar.

---

```
// Inserta un elemento delante del apuntado por el iterador it
// (it puede estar apuntado detrás del último)
```

---

---

```
// devuelve un iterador al nuevo elemento
iterator insert(iterator const& it, T const& elem) {
    assert(fantasma == it.fan); // chequea que it es de esta lista
    Nodo* nuevo = this->inserta_elem(elem, it.act->ant, it.act);
    return iterator(nuevo, this->fantasma);
}
```

---

Nótese que, a parte de insertar el nuevo elemento, la operación devuelve un iterador apuntando al nuevo elemento, lo que puede resultar muy útil en ciertos contextos.

## 7.4. Usando iteradores para eliminar elementos

También se puede extender el TAD `list` para permitir borrar elementos internos a la lista. La operación recibe un iterador situado en el punto de la lista que se desea borrar. En esta ocasión, dado que ese elemento dejará de existir ese iterador recibido *deja de ser válido*. Para poder seguir recorriendo la lista la operación devuelve un nuevo iterador que deberá utilizarse desde ese momento para continuar el recorrido.

Por ejemplo, la siguiente función elimina todos los elementos pares de una lista de enteros:

---

```
void quitaPares(list<int>& l) {
    for (list<int>::iterator it = l.begin(); it != l.end(); ) {
        if ((*it) % 2 == 0)
            it = l.erase(it);
        else
            ++it;
    }
}
```

---

La implementación de la operación `erase` de la lista, por tanto, elimina el elemento y devuelve un iterador apuntando al *siguiente* elemento (o devuelve el iterador que marca el fin del recorrido si no quedan más).

---

```
// elimina el elemento apuntado por el iterador (debe haber uno)
// devuelve un iterador al siguiente elemento al borrado
iterator erase(iterator const& it) {
    assert(this->fantasma == it.fan); // comprueba que it es de esta lista
    if (it.act == this->fantasma)
        throw std::out_of_range("fuera de la lista");
    iterator next(it.act->sig, this->fantasma);
    this->borra_elem(it.act);
    return next;
}
```

---

## 7.5. Peligros de los iteradores

El uso de iteradores conlleva un riesgo debido a la existencia de *efectos laterales* en las operaciones. Al fin y al cabo un iterador abre la puerta a acceder a los elementos de la lista *desde fuera* de la propia lista. Eso significa que los cambios que ocurran en ella afectan al resultado de las operaciones del propio iterador. Por ejemplo el código siguiente fallará:

---

```
list<int> lista;
lista.push_front(3);
list<int>::iterator it = lista.begin();
```

---

---

```
lista.pop_front();    // Quitamos el primer elemento
cout << **it << endl; // Accedemos a él... CRASH
```

---

Cuando el iterador permite cambiar el valor y, sobre todo, cuando se pueden borrar elementos utilizando iteradores las posibilidades de provocar funcionamientos incorrectos crecen.

No obstante, las ventajas de los iteradores al permitir recorridos eficientes (y generales, ver ejercicio 28) superan las desventajas. Pero el programador deberá estar atento a los iteradores y ser consciente de que las operaciones de modificación del TAD que está siendo recorrido pueden invalidar sus iteradores.

## 7.6. En el mundo real...

Como hemos dicho antes, a pesar de los posibles problemas de los iteradores, son muy utilizados (en distintas modalidades) en los lenguajes mayoritarios, como C++, Java o C#. La ventaja de los iteradores es que permiten abstraer el TAD que se recorre y se pueden tener algoritmos genéricos que funcionan bien independientemente de la colección utilizada. Por ejemplo un algoritmo que sume todos los elementos dentro de un intervalo de una colección será algo así:

---

```
template <class iterador>
int sumaTodos(iterador it, iterador fin) {
    int ret = 0;

    while (it != fin) {
        ret += *it;
        ++it;
    }

    return ret;
}
```

---

donde el tipo de los parámetros “iterador” es un parámetro de la plantilla, para poder utilizarlo con cualquier iterador (que tenga los operadores \* y++) independientemente de la colección que se recorre.

Dada la liberalidad de C y C++ con los tipos y la identificación deliberada que hacen entre punteros, enteros y arrays (lo que se conoce como *dualidad puntero-array*), la sintaxis de iteración utilizada anteriormente permite recorrer incluso un vector sin la necesidad de declarar una clase iterador:

```
int v[100];
...
cout << sumaTodos(&v[0], &v[100]) << endl;

// Versión alternativa, utilizando la dualidad
// puntero-array y aritmética de punteros
// cout << sumaTodos(v, v + 100) << endl;
```

Por último, la librería de C++ además de distinguir entre los iteradores constantes y no constantes, permite abstraer la dirección del recorrido e incluso moverse en ambas direcciones.

## 8. Para terminar...

Terminamos el tema con la implementación de la función de codificación descrita en la primera sección de motivación. Como se ve, al hacer uso de los TADs ya implementados el código queda muy claro; nos podemos centrar en la implementación del algoritmo de codificación sin preocuparnos del manejo de vectores, índices, listas de nodos o estructuras de datos que se llenan y hay que redimensionar.

---

```

list<char> codifica(list<char>& mensaje) {

    // Primera fase; el resultado lo metemos
    // en una doble cola para facilitarnos
    // la segunda fase
    deque<char> resFase1;
    stack<char> aInvertir;
    list<char>::const_iterator it = mensaje.cbegin();

    while (it != mensaje.cend()) {
        char c = *it;
        ++it;

        // Si no es una vocal, metemos el carácter
        // en la pila para invertirlo posteriormente
        if (!esVocal(c))
            aInvertir.push(c);
        else {
            // Una vocal: damos la vuelta
            // a todas las consonantes que nos
            // hayamos ido encontrando
            while (!aInvertir.empty()) {
                resFase1.push_back(aInvertir.top());
                aInvertir.pop();
            }
            // Y ahora la vocal
            resFase1.push_back(c);
        }
    }

    // Volcamos las posibles consonantes que queden
    // por invertir
    while (!aInvertir.empty()) {
        resFase1.push_back(aInvertir.top());
        aInvertir.pop();
    }

    // Segunda fase de la codificación: seleccionar
    // el primero/último de forma alternativa.
    list<char> ret; // Mensaje devuelto
    while (!resFase1.empty()) {
        ret.push_back(resFase1.front());
        resFase1.pop_front();
        if (!resFase1.empty()) {
            ret.push_back(resFase1.back());
            resFase1.pop_back();
        }
    }
}

```

---

```
    return ret;
}
```

## Notas bibliográficas

Gran parte de este capítulo se basa en el capítulo correspondiente de (Rodríguez Ar-talejo et al., 2011) y de (Peña, 2005). Es interesante ver las librerías de lenguajes como C++, Java o C#. Todas ellas tienen al menos una implementación de cada uno de los TADs lineales vistos en el tema e incluso más de una con distintas complejidades.

## Ejercicios

Algunos de los ejercicios puedes probarlos en el portal y juez en línea “Acepta el reto” (<https://www.aceptaelreto.com>). Son los que aparecen marcados con el icono  seguido del número de problema dado en el portal.

1. (ACR140) Implementa, con ayuda de una pila, un procedimiento *no* recursivo que reciba como parámetro un número entero  $n \geq 0$  y escriba por pantalla sus dígitos en orden lexicográfico y su suma. Por ejemplo, ante  $n = 64323$  escribirá:

$$6 + 4 + 3 + 2 + 3 = 18$$

2. Implementa una función que reciba una pila y escriba todos sus elementos desde la base hasta la cima separados por espacios. Haz dos versiones, una versión recursiva y otra iterativa.

Haz una tercera versión, implementando la funcionalidad como parte de la clase Stack.

3. Completa las dos implementaciones de las pilas con una operación nueva, `size` que devuelva el número de elementos almacenados en ella. En ambos casos la complejidad debe ser  $\mathcal{O}(1)$ .

4. (ACR141) Implementa una función que reciba una secuencia de caracteres en una lista que contiene, entre otros símbolos, paréntesis, llaves y corchetes abiertos y ce-rreados y decida si está equilibrada. Entendemos por secuencia equilibrada respecto a los tres tipos de símbolos si cada uno de ellos tiene tantos abiertos como cerrados y si cada vez que aparece uno cerrado, el último que apareció fue su correspondiente abierto.

5. Dada una cadena que contiene únicamente los caracteres <, > y ., implementa una función que cuente cuántos *diamantes* podemos encontrar como mucho (<>), tras quitar toda la arena (.). Ten en cuenta que puede haber diamantes dentro de otros diamantes. Por ejemplo, en la cadena <..<..>.><..>< hay tres diamantes.

6. (ACR252) Una frase se llama palíndroma si la sucesión de caracteres obtenida al recorrerla de izquierda a derecha (ignorando los espacios) es la misma que de derecha a izquierda. Esto sucede, por ejemplo, con la socorrida frase “dábale arroz a la zorra el abad” (ignorando la tilde de la primera palabra; podemos asumir que la

entrada no tendrá tildes y todo serán o bien letras o bien espacios). Construye una función *iterativa* ejecutable en tiempo lineal que decida si una frase dada como lista de caracteres es o no palíndroma. Puedes utilizar TADs auxiliares.

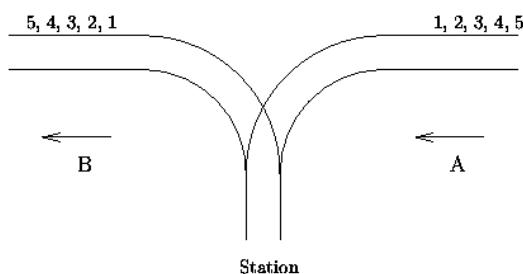
7. (ACR143) Implementa una función que reciba una pila como parámetro de E/S y un número  $n$  e invierta los  $n$  valores almacenados en la pila, comenzando a contar desde la cima.
8. (ACR198) Una expresión aritmética construida con los operadores binarios  $+$ ,  $-$ ,  $*$  y  $/$  y operandos (representados cada uno por un único carácter) se dice que está en forma *postfija* si es o bien un sólo operando o dos expresiones en forma postfija, una tras otra, seguidas inmediatamente de un operador. Lo que sigue es un ejemplo de una expresión escrita en notación infija habitual, junto con su forma postfija:

Forma infija:  $(A / (B - C)) * (D + E)$

Forma postfija: ABC-/DE+\*

Diseña un algoritmo iterativo que calcule el valor de una expresión dada en forma postfija (mediante una secuencia de caracteres) por el siguiente método: se inicializa una pila vacía de números y se van recorriendo de izquierda a derecha los caracteres de la expresión. Cada vez que se pasa por un operando, se apila su valor. Cada vez que se pasa por un operador, se desapilan los dos números más altos de la pila, se componen con el operador, y se apila el resultado. Al acabar el proceso, la pila contiene un solo número, que es el valor de la expresión.

9. (ACR198) Repite el ejercicio anterior pero utilizando en lugar de una pila, una cola. Ante un operador se cogen los dos parámetros de la cabecera de la cola y se mete el resultado en la parte trasera de la misma.
10. Dicen las malas lenguas que el alcalde de Dehesapila recibió una buena tajada haciendo un chanchullo en la construcción de la estación de trenes. El resultado fue una estación con forma de “punto ciego” en la que únicamente hay una vía de entrada y una de salida. Para empeorar las cosas según la forma de la figura:



Todos los trenes llegan desde  $A$  y van hacia  $B$ , pero reordenando los vagones en un orden distinto. ¿Puedes ayudar a establecer si el tren que entra con los vagones numerados  $1, 2, \dots, n$  puede reordenarse para que salga en un orden diferente enganchando y desenganchando vagones?

Por ejemplo, si entran los vagones  $1, 2, 3, 4$  y  $5$ , se podrá sacarlos en el mismo orden (haciéndolos entrar y salir de uno en uno), en orden inverso (haciéndolos entrar a

todos y luego sacándolos), pero no se podrá conseguir que salgan en el orden 5, 4, 1, 2, 3.

Escribe una función que lea de la entrada estándar un entero que indica el número de vagones que entran ( $n$ ) y luego vaya leyendo en qué orden se quiere que vayan saliendo de la estación ( $n$  enteros). La función escribirá POSIBLE si el jefe de estación puede ordenar los vagones e IMPOSIBLE si no<sup>9</sup>.

11. Extiende la función del ejercicio anterior para que en vez de indicar si es posible o no reordenar los vagones como se desea, escriba los movimientos que debe hacer el jefe de estación (meter un vagón en la estación / sacar un vagón de la estación).
12. Realiza una implementación de las colas utilizando vectores (dinámicos) de forma que la complejidad de todas las operaciones sea  $\mathcal{O}(1)$ . Para eso, en vez de utilizar un único atributo que indica cuántos elementos hay en el vector, utiliza dos atributos que permitan especificar el intervalo de elementos del vector que se están utilizando. Esta implementación se conoce con el nombre de *implementación mediante vectores circulares*.
13. Realiza una implementación con lista enlazada y nodo fantasma de las colas.
14. Completa la implementación de las colas con lista enlazada de nodos con una nueva operación numElems cuya complejidad sea  $\mathcal{O}(1)$ .
15. Extiende la implementación del TAD lista con una nueva operación concatena que reciba otra lista y añada sus elementos a la lista original. Por ejemplo, si  $xs = [3, 4, 5]$  y concatenamos  $ys = [5, 6, 7]$  con  $xs.concatena(ys)$ , al final tendremos que  $xs$  será  $[3, 4, 5, 5, 6, 7]$ . ¿Qué complejidad tiene la operación? ¿Podríamos conseguir complejidad  $\mathcal{O}(1)$  de alguna forma?
16. Implementa los TADs de las pilas, colas y colas dobles utilizando la implementación del TAD lista, de forma que las nuevas implementaciones tengan como único atributo una lista.
17. Dados dos números con un número de dígitos indeterminado expresados en una lista de enteros, donde el primer elemento de cada lista es el dígito más significativo (dígito izquierdo) de cada número, implementa una función que devuelva una lista con la suma de ambos números.
18. (ACR142) El profesor de EDA ha decidido sacar a un alumno a hacer un examen sorpresa. Para seleccionar al “afortunado” ha numerado a cada uno de los  $n$  alumnos con un número del 1 al  $n$  y los ha colocado a todos en círculo. Empezando por el número 1, va “salvando” a uno de cada dos (es decir, “salva” al 2, luego al 4, luego al 6, etc.), teniendo en cuenta que al ser circular, cuando llega al final sigue por los que quedan sin salvar. ¿Qué número tendrá el alumno “afortunado”?

Implementa una función:

```
int selecciona(int n);
```

<sup>9</sup>Este ejercicio es una traducción casi directa de [http://uva.onlinejudge.org/index.php?option=onlinejudge&page=show\\_problem&problem=455](http://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=455); la imagen del enunciado es una copia directa.

que devuelva el número de alumno seleccionado si tenemos  $n$  alumnos.

Generaliza la función anterior para el caso en el que el profesor salve a uno de cada  $m$  en lugar de a uno de cada 2.

19. (ACR146) Dado un número natural  $N \geq 2$ , se llaman números afortunados a los que resultan de ejecutar el siguiente proceso: se comienza generando una cola que contiene los números desde 1 hasta  $N$  en este orden; se elimina de la cola un número de cada 2 (es decir, los números 1, 3, 5, etc.<sup>10</sup>); de la nueva cola, se elimina ahora un número de cada 3; etc. El proceso termina cuando se va a eliminar un número de cada  $m$  y el tamaño de la cola es menor que  $m$ . Los números que queden en la cola en este momento son los afortunados.

Diseña un procedimiento que reciba  $N$  como parámetro y produzca una lista formada por los números afortunados resultantes.

(Indicación: para eliminar de una cola de  $n$  números un número de cada  $m$  puede reiterarse  $n$  veces el siguiente proceso: extraer el primer número de la cola, y añadirlo al final de la misma, salvo si le tocaba ser eliminado.)

20. Dada una lista de números, se repite el siguiente proceso hasta que queda únicamente uno: se elimina el primer elemento de la lista, y se pone el segundo al final. Implementa una función que, dada la lista, escriba el último número superviviente.

21. (ACR197) Implementa la función de *decodificación* de un mensaje según el algoritmo descrito en la primera sección del tema.

22. (ACR144) ¿Cuál será el texto final resultado de la siguiente pulsación de teclas, y dónde quedará el cursor colocado? d, D, <Inicio>, <Supr>, <FlechaDcha>, A, <Inicio>, E, <Fin>

Haz una función que reciba una lista con las pulsaciones de teclas y devuelva una lista con el texto resultante.

23. (ACR258) Implementa una función que reciba un vector de enteros de tamaño  $N$  y un número  $K$ , y escriba el valor mínimo de cada subvector de tamaño  $K$  en  $\mathcal{O}(N)$ .

Por ejemplo, para el vector [1, 3, 2, 5, 8, 5] y  $k = 3$ , debe escribir [1, 2, 2, 5].

## Recorridos

24. Implementa una función que reciba una lista e imprima todos sus elementos. La complejidad de la operación debe ser  $\mathcal{O}(n)$  incluso para la implementación del TAD lista utilizando listas enlazadas.
25. Implementa utilizando iteradores una función que reciba una secuencia de caracteres y devuelva el número de “a” que tiene.
26. Dada una secuencia de enteros, contar cuántas posiciones hay en ella tales que el entero que aparece en esa posición es igual a la suma de todos los precedentes.
27. Implementa una función que reciba dos listas de enteros ordenados y devuelva una nueva lista ordenada con la unión de los enteros de las dos listas.

---

<sup>10</sup>Observa que este tipo de eliminación es distinto al del ejercicio 18.

28. Extiende el TAD lista implementando dos operaciones nuevas `reverse_begin` y `reverse_end` que devuelva sendos `ReverseIterator` que permitan recorrer la lista desde el final al principio.
29. Utilizando el iterador del ejercicio anterior junto con el iterador de las listas, implementa una función que reciba un `list` y mire si la lista es palíndroma o no. Recuerda: *no* está permitido usar una pila (ese es el ejercicio 6).
30. Implementa una función que reciba una lista de enteros y duplique (multiplique por dos) todos sus elementos. Haz uso de los iteradores.
31. Implementa:
  - Una función que busque un elemento en un `list<T>` y devuelva un iterador mutable al primer elemento encontrado (o a `end()` si no hay ninguno).
  - Mejora la función anterior para que, en vez de recibir la lista, reciba dos iteradores que marcan el intervalo (abierto) sobre el que hay que buscar el elemento, de forma que la función anterior pueda implementarse como:

```
template <class T>
list<T>::iterator busca(const T& elem, list<T>& l) {
    return busca(elem, l.begin(), l.end());
```
  - Implementa una función parecida a la anterior pero que en vez de buscar un elemento *lo borre*. En concreto, borrará todos los elementos que aparezcan en el intervalo (no solamente el primero).
  - Utilizando las funciones anteriores, implementa una función que reciba una lista de caracteres y elimine todas las “a” de la primera palabra.



---

## Capítulo 6

# Diseño e implementación de TADs arborescentes<sup>1</sup>

---

*Los ordenadores son buenos siguiendo instrucciones, no leyendo tu mente*

Donald Knuth

**RESUMEN:** En este tema se presentan los TADs basados en árboles, prestando especial atención a los árboles binarios. Además, se introducen distintos tipos de recorridos usando tanto listas como iteradores.

## 1. Motivación

Queremos hacer un pequeño juego en el que la máquina pide al usuario que piense un animal y ésta trata de adivinarlo haciendo distintas preguntas<sup>2</sup>:

```
¿Tiene cuatro patas? (Si/No)
no
¿Vive en el agua? (Si/No)
no
Mmmmmm, dejame que piense.....
Creo que el animal en que estabas pensando era...: ¡Serpiente!
¿Acerté? (Si/No)
si
¡Estupendo! Gracias por jugar conmigo.
```

Cuando la máquina falla, le pide al usuario que diga una pregunta que discrimine entre el animal por el que apostó y el animal real que pensó. De esta forma, la aplicación aprende de sus errores e incorpora en su base de conocimiento el nuevo animal. Si por ejemplo el usuario pensó en el orangután, en la siguiente partida en vez de decir *serpiente*, seguiría

---

<sup>1</sup>Marco Antonio Gómez y Antonio Sánchez son los autores principales de este tema.

<sup>2</sup>Un clon de un antiguo juego de los ordenadores de 8 bits de la década de 1980, llamado *Animal, vegetal, mineral*, <http://www.amstrad.es/programas/amsdos/educativos/animalvegetalmineral.php>.

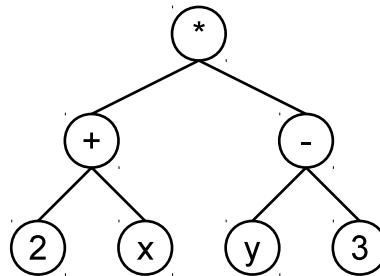
preguntando algo como ¿Es un reptil? para en base a la respuesta, contestar *serpiente* u *orangután*.

¿Cómo implementarías la aplicación?

## 2. Introducción

En el capítulo anterior estudiamos distintos TADs lineales para representar datos organizados de manera secuencial. En este capítulo usaremos árboles para representar de manera intuitiva datos organizados en jerarquías. Este tipo de estructuras jerárquicas surge de manera natural dentro y fuera de la Informática:

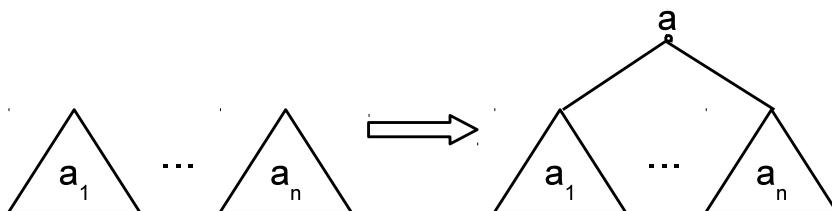
- Árboles genealógicos.
- Organización de un libro en capítulos, secciones, etc.
- Estructura de directorios y archivos de un sistema operativo.
- Árboles de análisis de expresiones aritméticas.



### 2.1. Modelo matemático

Desde un punto de vista matemático, los árboles son estructuras jerárquicas formadas por *nodos*, que se construyen de manera inductiva:

- Un solo nodo es un árbol  $a$ . El nodo es la *raíz* del árbol.
- Dados  $n$  árboles  $a_1, \dots, a_n$ , podemos construir un nuevo árbol  $a$  añadiendo un nuevo nodo como raíz y conectándolo con las raíces de los árboles  $a_i$ . Se dice que los  $a_i$  son *subárboles* de  $a$ .



Para identificar los distintos nodos de un árbol, vamos a usar una función que asigna a cada posición una cadena de números naturales con el siguiente criterio:

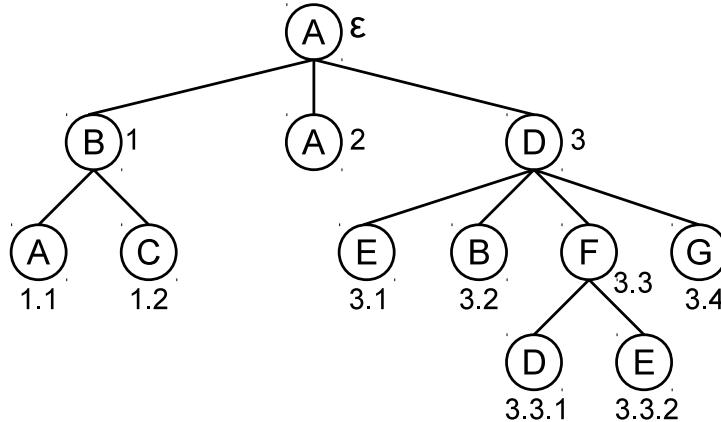


Figura 1: Posiciones y valores de cada nodo de un árbol.

- La raíz del árbol tiene como posición la *cadena vacía*  $\epsilon$ .
- Si un cierto nodo tiene como posición la cadena  $\alpha \in \mathbb{N}^*$ , el hijo  $i$ -ésimo de ese nodo tendrá como posición la cadena  $\alpha.i$ .

Por ejemplo, la figura 1 muestra un árbol y las cadenas que identifican las posiciones de sus nodos.

Un árbol puede describirse como una aplicación  $a : N \rightarrow V$  donde  $N \subseteq \mathbb{N}^*$  es el conjunto de posiciones de los nodos, y  $V$  es el conjunto de valores posibles asociados a los nodos. Podemos describir el árbol de la figura 1 de la siguiente manera:

$$\begin{aligned} N &= \{\epsilon, 1, 2, 3, 1.1, 1.2, 3.1, 3.2, 3.3, 3.4, 3.3.1, 3.3.2\} \\ V &= \{A, B, C, D, E, F, G\} \end{aligned}$$

$$\begin{array}{lll} a(\epsilon) = A & & \\ a(1) = B & a(2) = A & a(3) = D \\ a(1.1) = A & a(1.2) = C & a(3.1) = E \quad \text{etc.} \end{array}$$

## 2.2. Terminología

Antes de seguir adelante, debemos establecer un vocabulario común que nos permita describir árboles. Dado un árbol  $a : N \rightarrow V$

- Cada *nodo* es una tupla  $(\alpha, a(\alpha))$  que contiene la posición y el valor asociado al nodo. Distinguimos 3 tipos de nodos:
  - La *raíz* es el nodo de posición  $\epsilon$ .
  - Las *hojas* son los nodos de posición  $\alpha$  tales que no existe  $i$  tal que  $\alpha.i \in N$
  - Los *nodos internos* son los nodos que no son hojas.
- Un nodo  $\alpha.i$  tiene como *padre* a  $\alpha$ , y se dice que es *hijo* de  $\alpha$ .
- Dos nodos de posiciones  $\alpha.i$  y  $\alpha.j$  ( $i \neq j$ ) se llaman *hermanos*.

- Un *camino* es una sucesión de nodos  $\alpha_1, \alpha_2, \dots, \alpha_n$  en la que cada nodo es padre del siguiente. El camino anterior tiene *longitud n*.
- Una *rama* es cualquier camino que empieza en la raíz y acaba en una hoja.
- El *nivel* o *profundidad* de un nodo es la longitud del camino que va desde la raíz hasta al nodo. En particular, el nivel de la raíz es 1.
- La *talla* o *altura* de un árbol es el máximo de los niveles de todos los nodos del árbol.
- El *grado* o *aridad* de un nodo interno es su número de hijos. La *aridad de un árbol* es el máximo de las aridades de todos sus nodos internos.
- Decimos que  $\alpha$  es *antepasado* de  $\beta$  (resp.  $\beta$  es *descendiente* de  $\alpha$ ) si existe un camino desde  $\alpha$  hasta  $\beta$ .
- Cada nodo de un árbol  $a$  determina un *subárbol*  $a_0$  con raíz en ese nodo.
- Dado un árbol  $a$ , los subárboles de  $a$  (si existen) se llaman *árboles hijos* de  $a$ .

### 2.3. Tipos de árboles

Distinguimos distintos tipos de árboles en función de sus características:

- Ordenados o no ordenados. Un árbol es ordenado si el orden de los hijos de cada nodo es relevante.
- Generales o  $n$ -ários. Un árbol es  $n$ -ário si el máximo número de hijos de cualquier nodo es  $n$ . Un árbol es general si no existe una limitación fijada al número máximo de hijos de cada nodo.

## 3. Árboles binarios: operaciones

Un árbol binario consiste en una estructura recursiva cuyos nodos tienen como mucho dos hijos, un hijo izquierdo y un hijo derecho. El TAD de los árboles binarios (lo llamaremos *bintree*) tiene las siguientes operaciones:

- Constructora sin argumentos: operación generadora que construye un árbol vacío (un árbol sin ningún nodo).
- Constructora con argumentos: segunda operación generadora que construye un árbol binario a partir de otros dos (el que será el hijo izquierdo y el hijo derecho) y de la información que se almacenará en la raíz.
- *root*: operación observadora que devuelve el elemento almacenado en la raíz del árbol. Es parcial pues falla si el árbol es vacío.
- *left*, *right*: dos operaciones observadoras (ambas parciales) que permiten obtener el hijo izquierdo y el hijo derecho de un árbol dado. Las operaciones no están definidas para árboles vacíos.
- *empty*: otra operación observadora para saber si un árbol tiene algún nodo o no.

Con sólo estas operaciones la utilidad del TAD está muy limitada. En apartados siguientes lo extenderemos con otras operaciones observadoras.

## 4. Implementación de árboles binarios

Igual que ocurre con los TADs lineales, podemos implementar el TAD *bintree* utilizando distintas estructuras en memoria. Sin embargo cuando la forma de los árboles no está restringida la única implementación factible es la que utiliza nodos enlazados.

La intuición detrás de la implementación es sencilla y sale directamente de las representaciones de los árboles que hemos utilizado en la sección anterior: cada nodo del árbol será representado como un nodo en memoria que contendrá tres atributos: el elemento almacenado y punteros al hijo izquierdo y al hijo derecho.

No obstante, *no* debe confundirse un elemento del TAD *bintree* con la estructura en memoria utilizada para almacenarlo. Si bien existe una transformación directa entre uno y otro son conceptos distintos. Un árbol es un elemento del TAD construido utilizando las operaciones generadoras anteriores (y que, cuando lo programemos, será un objeto de la clase *bintree*); los nodos en los que nos basamos forman una *estructura jerárquica de nodos* que tiene sentido únicamente *en la implementación*. Veremos en la implementación que hay métodos (privados o protegidos) que trabajan directamente con esta *estructura jerárquica* a la que el usuario del TAD no tendrá acceso directo (en otro caso se rompería la barrera de abstracción impuesta por las operaciones del TAD).

A continuación aparece la definición de la estructura *TreeNode* que, como no podría ser de otra manera, es una estructura interna a *bintree*. La clase *bintree* necesita únicamente almacenar un *puntero* a la raíz de la *estructura jerárquica de nodos* que representan el árbol<sup>3</sup>.

---

```
template <class T>
class bintree {
protected:
    /*
        Nodo que almacena internamente el elemento (de tipo T),
        y punteros al hijo izquierdo y derecho, que pueden ser
        nullptr si el hijo es vacío (no existe).
    */
    struct TreeNode;

    using Link = TreeNode*;

    struct TreeNode {
        TreeNode(Link const& l, T const& e, Link const& r) : elem(e), left(l), right(r)
            T elem;
            Link left, right;
    };

    // puntero a la raíz del árbol
    Link raiz;

    ...
};

}
```

---

El *invariante de la representación* de esta implementación debe asegurarse de que:

- Todos los nodos contienen información válida y están ubicados correctamente en

---

<sup>3</sup>Estas estructuras jerárquicas de nodos son útiles también para resolver otros problemas distintos a la implementación del TAD de los árboles binarios; ver por ejemplo el ejercicio 13.

memoria.

- El subárbol izquierdo y el subárbol derecho no comparten nodos.
- No hay ciclos entre los nodos, o lo que es lo mismo, los nodos alcanzables desde la raíz no incluyen a la propia raíz.

Con estas ideas el invariante queda:

$$\begin{aligned} R_{bintree_T}(p) \\ \iff_{def} \\ buenaJerarquia(p.raiz) \end{aligned}$$

donde

$$\begin{aligned} buenaJerarquia(ptr) = true & \quad \text{si } ptr = \text{null} \\ buenaJerarquia(ptr) = \text{ubicado}(ptr) \wedge R_T(ptr.elem) \wedge & \\ \text{nodos}(ptr.left) \cap \text{nodos}(ptr.right) = \emptyset \wedge & \\ ptr \notin \text{nodos}(ptr.left) \wedge & \\ ptr \notin \text{nodos}(ptr.right) \wedge & \\ buenaJerarquia(ptr.left) \wedge & \\ buenaJerarquia(ptr.right) & \quad \text{si } ptr \neq \text{null} \\ \\ \text{nodos}(ptr) = \emptyset & \quad \text{si } ptr = \text{null} \\ \text{nodos}(ptr) = \{ptr\} \cup \text{nodos}(ptr.left) \cup \text{nodos}(ptr.right) & \quad \text{si } ptr \neq \text{null} \end{aligned}$$

La definición de la relación de equivalencia que se utiliza para saber si dos árboles son o no iguales también sigue una definición recursiva:

$$\begin{aligned} p1 \equiv_{bintree_T} p2 \\ \iff_{def} \\ igualesT(p1.raiz, p2.raiz) \\ \\ iguales(ptr1, ptr2) = true & \quad \text{si } ptr1 = \text{null} \wedge ptr2 = \text{null} \\ iguales(ptr1, ptr2) = false & \quad \text{si } (ptr1 = \text{null} \wedge ptr2 \neq \text{null}) \vee \\ & \quad (ptr1 \neq \text{null} \wedge ptr2 = \text{null}) \\ iguales(ptr1, ptr2) = ptr1.elem \equiv_T ptr2.elem \wedge & \\ iguales(ptr1.left, ptr2.left) \wedge & \\ iguales(ptr1.right, ptr2.right) & \quad \text{si } (ptr1 \neq \text{null} \wedge ptr2 \neq \text{null}) \end{aligned}$$

Antes de seguir con la implementación de las operaciones debemos crear algunos métodos que trabajan directamente con la *estructura jerárquica*, igual que en las implementaciones de los TADs lineales del tema anterior empezamos por las operaciones que trabajaban con las listas enlazadas y doblemente enlazadas. Dada la naturaleza recursiva de la estructura de nodos, todas esas operaciones serán recursivas; recibirán, al menos, un puntero a

un nodo que debe verse como la raíz de la estructura de nodos, o al menos la raíz del “subárbol”<sup>4</sup> sobre el que debe operar.

Aunque pueda parecer empezar la casa por el tejado, comencemos con la operación que *libera* toda la memoria ocupada por la estructura jerárquica. El método recibe como parámetro el puntero al primer nodo y va eliminando de forma recursiva:

---

```
static void libera(Link ra) {
    if (ra != NULL) {
        libera(ra->left);
        libera(ra->right);
        delete ra;
    }
}
```

---

Algunas de las operaciones pueden trabajar no con una sino con *dos* estructuras jerárquicas. Por ejemplo el siguiente método compara dos estructuras, dados los punteros a sus raíces:

---

```
static bool comparaAux(Link r1, Link r2) {
    if (r1 == r2)
        return true;
    else if ((r1 == nullptr) || (r2 == nullptr))
        // En el if anterior nos aseguramos de
        // que r1 != r2. Si uno es nullptr, el
        // otro entonces no lo será, luego
        // son distintos.
        return false;
    else {
        return (r1->elem == r2->elem) &&
            comparaAux(r1->left, r2->left) &&
            comparaAux(r1->right, r2->right);
    }
}
```

---

Como veremos más adelante, este método estático nos resultará muy útil para implementar el operador de igualdad del TAD *bintree*.

Por último, algunas de las operaciones pueden devolver otra información. Por ejemplo, el siguiente método hace una *copia* de una estructura jerárquica y devuelve el puntero a la raíz de la copia. Igual que todas las anteriores la implementación es recursiva. Como es lógico, la estructura recién creada deberá ser eliminada (utilizando el *libera* implementado anteriormente) posteriormente:

---

```
static Link copiaAux(Link ra) {
    if (ra == nullptr) return nullptr;
    return new TreeNode(copiaAux(ra->left),
                        ra->elem,
                        copiaAux(ra->right));
}
```

---

Tras esto, estamos en disposición de implementar las operaciones públicas del TAD. No obstante, antes de abordarlas expliquemos una decisión de diseño relevante. Hay una

---

<sup>4</sup>La palabra “subárbol” la ponemos entre comillas para hacer notar que no estamos aquí haciendo referencia al concepto de subárbol como elemento del TAD sino más bien a la “subestructura” jerárquica de nodos.

operación generadora (que implementaremos como constructor) que recibe dos árboles ya construidos:

---

```
bintree(const bintree &iz, const T &elem, const bintree &dr);
```

---

Una implementación ingenua crearía un nuevo nodo y “cosería” los punteros haciendo que el hijo izquierdo del nuevo nodo fuera la raíz de *iz* y el derecho la raíz de *dr*:

---

```
// IMPLEMENTACIÓN NO VALIDA (POR EL MOMENTO...)
bintree::bintree(const bintree &iz, const T &elem, const bintree &dr) {
    raiz = new TreeNode(iz.raiz, elem, dr.raiz);
}
```

---

Esta implementación, no obstante, no es válida (por el momento) porque tendríamos una poco deseable compartición de memoria: la estructura jerárquica de los nodos de *iz* y de *dr* formarían también parte de la estructura jerárquica del nodo recién construido. Eso tiene como consecuencia inmediata que al destruir *iz* se destruiría automáticamente los nodos del árbol más grande.

Para solucionar el problema podemos plantear dos alternativas (similares a las que se tienen cuando implementamos la operación de concatenación de las listas, ver ejercicio 15 del tema anterior)<sup>5</sup>:

- Hacer una copia de las estructuras jerárquicas de nodos de *iz* y *dr*.
- Utilizar esas estructuras jerárquicas para el nuevo árbol y *vaciar* *iz* y *dr* de forma que la llamada al constructor los *vacíe*<sup>6</sup>.

En nuestra implementación nos decantaremos por la primera opción<sup>7</sup>:

---

```
bintree(const bintree &iz, const T &elem, const bintree &dr) :
    raiz(new TreeNode(copiaAux(iz.raiz), elem, copiaAux(dr.raiz))) {
```

---

Una copia similar hay que realizar con las operaciones *left* e *right* lo que automáticamente hace que el coste de estas operaciones sea  $\mathcal{O}(n)$ :

---

```
/**
 * Devuelve un árbol copia del árbol izquierdo.
 * Es una operación parcial (falla con el árbol vacío).
 */
bintree left() const {
    if (empty())
        throw std::domain_error("El árbol vacío no tiene hijo izquierdo.");

    return bintree<T>(copiaAux(raiz->left));
}

/**
 * Devuelve un árbol copia del árbol derecho.
 * Es una operación parcial (falla con el árbol vacío).
 */
bintree right() const {
```

---

<sup>5</sup>En realidad existe una tercera alternativa que veremos en la sección 6 más tarde en este mismo tema.

<sup>6</sup>Para eso en la cabecera deberían pasarse como parámetros de entrada/salida, es decir como referencias *no constantes*.

<sup>7</sup>Ver, no obstante, el ejercicio 3.

---

```

if (empty())
    throw std::domain_error("El arbol vacio no tiene hijo derecho.");

return bintree<T>(copiaAux(raiz->right));
}

```

---

La otra operación observadora, utilizada para acceder al elemento que hay en la raíz, no tiene necesidad de copias:

---

```

/**
Devuelve el elemento almacenado en la raiz
*/
const T& root() const {
    if (empty())
        throw std::domain_error("El arbol vacio no tiene raiz.");
    return raiz->elem;
}

```

---

Otra operación observadora sencilla de implementar es empty:

---

```

/**
Operación observadora que devuelve si el árbol
es vacío (no contiene elementos) o no.
*/
bool empty() const {
    return raiz == nullptr;
}

```

---

La última operación observadora que implementaremos será el operador de igualdad. En este caso delegamos en el método estático que compara las estructuras jerárquicas de nodos:

---

```

/**
Operación observadora que indica si dos bintree
son equivalentes.
*/
bool operator==(const bintree &a) const {
    return comparaAux(raiz, a.raiz);
}

```

---

Con esta terminamos la implementación de las operaciones del TAD. Existen otras operaciones observadoras que suelen ser habituales en la implementación de los árboles binarios que permiten conocer algunas de las propiedades definidas en la sección 2. La implementación de la mayoría de ellas requiere la creación de métodos recursivos auxiliares que trabajen con la estructura jerárquica de los nodos. Ver por ejemplo el ejercicio 1.

| Operación                   | Complejidad      |
|-----------------------------|------------------|
| Constructora sin argumentos | $\mathcal{O}(1)$ |
| Constructora con argumentos | $\mathcal{O}(n)$ |
| left                        | $\mathcal{O}(n)$ |
| right                       | $\mathcal{O}(n)$ |
| empty                       | $\mathcal{O}(1)$ |
| operator==                  | $\mathcal{O}(n)$ |

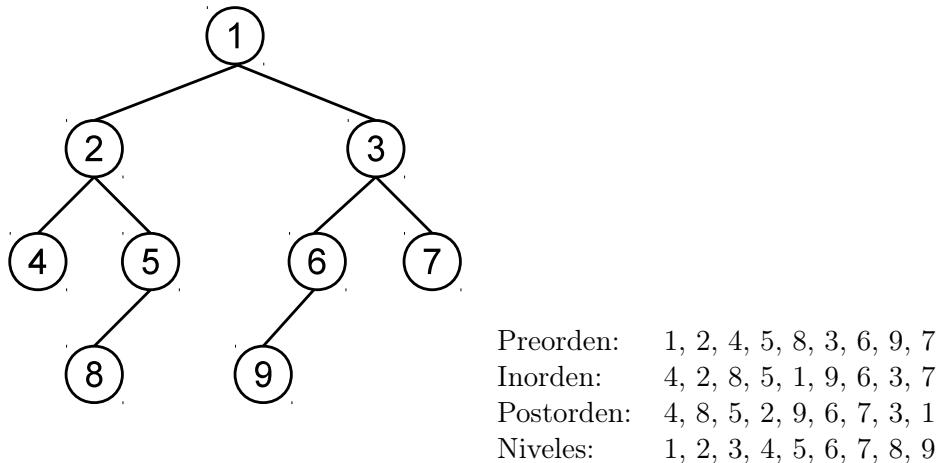


Figura 2: Distintas formas de recorrer un árbol.

## 5. Recorridos

Además de las operaciones observadoras indicadas anteriormente podemos *enriquecer* el TAD `bintree` con una serie de operaciones que permitan recorrer todos los elementos del árbol. Si bien en el caso de las estructuras lineales el recorrido no ofrecía demasiadas posibilidades (solo había dos órdenes posibles, de principio al final o al contrario), en los árboles binarios hay distintas formas de recorrer el árbol.

Los cuatro recorridos que veremos procederán de la misma forma: serán operaciones observadoras que devolverán vectores (`vector<T>`) con los elementos almacenados en el árbol donde cada elemento aparecerá una única vez. El orden concreto en el que aparecerán dependerá del recorrido concreto (ver figura 2).

Los tres primeros recorridos que consideraremos tienen definiciones recursivas:

- Recorrido en *preorden*: se visita en primer lugar la *raíz* del árbol y, a continuación, se recorren en preorden el hijo izquierdo y el hijo derecho.
- Recorrido en *inorden*: la raíz se visita tras el recorrido en inorder del hijo izquierdo y antes del recorrido en inorder del hijo derecho.
- Recorrido en *postorden*: primero los recorridos en postorden del hijo izquierdo y derecho y al final la raíz.

Podemos hacer una implementación recursiva directa de la definición anterior si asumimos la existencia de una función que concatene vectores:

---

```

vector<T> bintree<T>::preorder() const {
    return preorder(raiz);
}

static vector<T> bintree<T>::preorder(Link p) {
    if (p == nullptr)
        return vector<T>(); // Vector vacío

    vector<T> ret;
    ret.push_back(p->elem);
    concatena(ret, preorder(p->left));
}
  
```

---

---

```

concatena(ret, preorder(p->right));

return ret;
}

```

---

Sin embargo esa operación de concatenación no está disponible, y su implementación tendría coste lineal. Podemos implementarla por ejemplo basándonos en el método `insert` de la clase `vector` de esta forma:

---

```

template <class T>
void concatena(vector<T>& v1, const vector<T>& v2) {
    v1.insert(v1.end(), v2.begin(), v2.end());
}

```

---

Esto da lugar a una implementación ineficiente. En concreto sería  $\mathcal{O}(n\log n)$  de manera análoga a los algoritmos *quicksort* y *mergesort*. Observa que se hacen dos llamadas recursivas con la mitad de los datos y una llamada extra con coste lineal. Sin embargo, existe una implementación mejor que, siendo también recursiva, hace uso de un vector como parámetro de entrada/salida que va *acumulando* el resultado hasta que termina el recorrido. El método `preorder` auxiliar anterior se convierte en un método con dos parámetros: un puntero a la raíz de la estructura jerárquica de nodos que visitar, y un vector (no necesariamente vacío) al que se *añadirán* por la derecha los elementos del recorrido del resto del árbol.

---

```

std::vector<T> preorder() const {
    std::vector<T> pre;
    preorder(raiz, pre);
    return pre;
}

static void preorder(Link a, std::vector<T> & pre) {
    if (a != nullptr) {
        pre.push_back(a->elem);
        preorder(a->left, pre);
        preorder(a->right, pre);
    }
}

```

---

La complejidad del recorrido es  $\mathcal{O}(n)$ , a lo que se puede llegar tras el análisis de rencurrencias que utilizábamos en los algoritmos recursivos de los temas pasados. La misma complejidad tienen las implementaciones de los recorridos inorden y postorden que utilizan la misma idea:

---

```

std::vector<T> inorder() const {
    std::vector<T> in;
    inorder(raiz, in);
    return in;
}

static void inorder(Link a, std::vector<T> & in) {
    if (a != nullptr) {
        inorder(a->left, in);
        in.push_back(a->elem);
        inorder(a->right, in);
    }
}

```

---

---

```

    std::vector<T> postorder() const {
        std::vector<T> post;
        postorder(raiz, post);
        return post;
    }

    static void postorder(Link a, std::vector<T> & post) {
        if (a != nullptr) {
            postorder(a->left, post);
            postorder(a->right, post);
            post.push_back(a->elem);
        }
    }
}

```

---

El último tipo de recorrido que consideraremos es el recorrido *por niveles* (ver figura 2), que consiste en visitar primero la raíz, luego todos los nodos del nivel inmediatamente inferior de izquierda a derecha, a continuación todos los nodos del nivel tres, etc.

La implementación no puede ser recursiva sino iterativa. Hace uso de una cola que contiene todos los subárboles que aún quedan por visitar:

---

```

    std::vector<T> levelorder() const {
        std::vector<T> levels;
        if (!empty()) {
            std::queue<Link> pendientes;
            pendientes.push(raiz);
            while (!pendientes.empty()) {
                Link sig = pendientes.front();
                pendientes.pop();
                levels.push_back(sig->elem);
                if (sig->left != nullptr)
                    pendientes.push(sig->left);
                if (sig->right != nullptr)
                    pendientes.push(sig->right);
            }
        }
        return levels;
    }
}

```

---

La complejidad de este recorrido también es  $\mathcal{O}(n)$  aunque para llegar a esa conclusión nos limitaremos, por una vez, a utilizar la intuición: cada una de las operaciones del bucle tienen coste constante,  $\mathcal{O}(1)$ . Ese bucle se repite una vez por cada nodo del árbol; para eso basta darse cuenta que cada subárbol (o mejor, cada subestructura) aparece una única vez en la cabecera de la cola.

## 6. Implementación eficiente de los árboles binarios

El coste de las operaciones observadoras `left` y `right` de los árboles binarios es lineal, ya que devuelven una *copia* nueva de los árboles. Eso hace que una función aparentemente inocente como la siguiente, que cuenta el número de nodos:

---

```

template <typename E>
unsigned int numNodos(const bintree<E> &arbol) {
    if (arbol.empty())

```

---

---

```

    return 0;
else
    return 1 +
        numNodos(arbol.left()) +
        numNodos(arbol.right());
}

```

---

no tenga un coste lineal.

La razón fundamental de esto es que no hemos permitido la *compartición* de la estructura jerárquica de nodos. Una forma (ingenua, como veremos en breve) de solucionar el problema de eficiencia de las operaciones observadoras sería que esa estructura fuera compartida por ambos árboles. La operación generadora devuelve un nuevo árbol cuya raíz *apunta al mismo nodo* que es apuntado por el puntero `left` de la raíz del árbol más grande.

La compartición de memoria es posible gracias a que los árboles son objetos *inmutables*. Efectivamente, una vez que hemos construido un árbol, éste *no* puede ser modificado ya que todas las operaciones disponibles (`left`, `root`, etc.) son *observadoras* y nunca modifican el árbol. Gracias a eso sabemos que la devolución del hijo izquierdo compartiendo nodos *no* es peligrosa en el sentido de que modificaciones en un árbol afecten al contenido de otro, pues esas modificaciones son imposibles por definición del TAD. Si hubieramos tenido disponible una operación como `cambiaRaiz` la compartición no habría sido posible pues el siguiente código:

---

```

bintree<int> arbol;

// ...
// aquí construimos el árbol con varios nodos
// ...

bintree<int> otro;
otro = arbol.left();
otro.cambiaRaiz(1 + otro.left());

```

---

al cambiar el contenido de la raíz del árbol `otro` está también cambiando un elemento de `arbol`.

Desgraciadamente, sin embargo, esta solución de compartición de memoria no funciona en lenguajes como C++. Pensemos en el siguiente código aparentemente inocente:

---

```

bintree<int> arbol;

// ...
// aquí construimos el árbol con varios nodos
// ...

bintree<int> otro;
otro = arbol.left();

```

---

Cuando el código anterior termina y las dos variables salen de ámbito, el compilador llamará a sus destructores. La destrucción de la variable `arbol` eliminará todos sus nodos; cuando el destructor de `otro` vaya a eliminarlos, se encontrará con que ya no existían, generando un error de ejecución.

Por lo tanto, el principal problema es la destrucción de la estructura de memoria compartida. En lenguajes como Java o C# con recolección automática de basura la solución

anterior es perfectamente válida. En el caso de C++ hay que buscar otra aproximación. En concreto, se pueden traer ideas del mundo de la recolección de basura a nuestra implementación de los árboles. La solución que vamos a implementar a continuación es utilizar lo que se llama *conteo de referencias*: cada nodo de la estructura jerárquica de nodos tendrá un contador (entero) indicando *cuántos punteros lo referencian*. Así, si tengo un único árbol, todos sus nodos tendrán un 1 en ese contador. La operación `left` construirá un nuevo árbol cuya raíz apuntará al nodo del hijo izquierdo, por lo que su contador *se incrementará*. Cuando se invoque al destructor del árbol, se decrementará el contador y si llega a cero se eliminará él y recursivamente todos los hijos.

La implementación de la clase `nodo` con el contador quedaría así (aparecen subrayados los cambios):

---

```
class TreeNode {
public:
    TreeNode() : left(nullptr), right(nullptr), numRefs(0) { }
    TreeNode(Link iz, const T &elem, Link dr) :
        elem(elem), left(iz), right(dr), numRefs(0) {
        if (left != nullptr) left->addRef();
        if (right != nullptr) right->addRef();
    }

    void addRef() { assert(numRefs >= 0); numRefs++; }
    void remRef() { assert(numRefs >0); numRefs--; }

    T elem;
    Link left;
    Link right;

    int numRefs;
};
```

---

Las operaciones como `left()` ya no necesitan hacer la copia profunda de la estructura jerárquica de nodos; el constructor especial que recibe el puntero al nodo raíz simplemente incrementa el contador de referencias:

---

```
class bintree {
public:
    ...

    bintree left() const {
        if (empty())
            throw std::domain_error("El arbol vacio no tiene hijo izquierdo.");

        return bintree(copiaAux(raiz->left));
    }

protected:
    ...

    bintree(Link raiz) : raiz(raiz) {
        if (raiz != nullptr)
            raiz->addRef();
    }
}
```

---

Tampoco se necesitaría la copia en la construcción de un árbol nuevo a partir de los dos hijos, pues el árbol grande compartiría la estructura. El constructor crearía el nuevo `TreeNode` (cuyo constructor incrementaría los contadores de los nodos izquierdo y derecho) y pondría el contador del nodo recién creado a uno:

---

```
bintree(const bintree &iz, const T &elem, const bintree &dr) :
    raiz(new TreeNode(copiaAux(iz.raiz), elem, copiaAux(dr.raiz))) {
    raiz->addRef();
}
```

---

Por último, la liberación de la estructura jerárquica de nodos sólo se realiza si nadie más referencia el nodo. Por lo tanto el método de liberación recursivo cambia:

---

```
static void libera(Link ra) {
    if (ra != nullptr) {
        ra->remRef();
        if (ra->numRefs == 0) {
            libera(ra->left);
            libera(ra->right);
            delete ra;
        }
    }
}
```

---

Gracias a estas modificaciones la complejidad de todas las operaciones pasaría a ser constante, y el consumo de memoria se reduce pues no desperdiciamos nodos con información repetida.

| Operación  | Complejidad      |
|------------|------------------|
| ArbolVacio | $\mathcal{O}(1)$ |
| Cons       | $\mathcal{O}(1)$ |
| left       | $\mathcal{O}(1)$ |
| right      | $\mathcal{O}(1)$ |
| empty      | $\mathcal{O}(1)$ |

Ahora que disponemos de operaciones eficientes, podríamos implementar todos los recorridos que vimos en el apartado anterior (preorden, inorden, etc.) como funciones externas al TAD `bintree`. Ten en cuenta que ahora podemos navegar por su estructura usando los métodos públicos `left` y `right` sin necesidad de hacer copias.

Finalmente, te proponemos que realices el ejercicio 5 para terminar de ver la diferencia entre las dos implementaciones del TAD.

## 7. Implementación con *punteros inteligentes*

La técnica del conteo de referencias utilizada en la sección previa es muy común y anterior a la existencia de recolectores de basura (en realidad los primeros recolectores de basura se implementaron utilizando esta técnica, por lo que podríamos incluso decir que lo que hemos implementado aquí es un pequeño recolector de basura para los nodos de los árboles).

En un desarrollo más grande el manejo explícito de los contadores invocando al `addRef` y `remRef` es incómodo y propenso a errores, pues es fácil olvidar llamarlas. Para nuestra pequeña implementación hemos preferido utilizar ese manejo explícito para que se vea más

claramente la idea, pero un olvido en un `remRef` de algún método habría tenido como consecuencia fugas de memoria pues el contador nunca recuperará el valor 0 para indicar que nadie lo está utilizando y que por tanto puede borrarse.

Para evitar este tipo de problemas se han implementado estrategias que gestionan de forma automática esos contadores. En particular, son muy utilizados lo que se conoce como *punteros inteligentes* (en inglés *smart pointers*) que son variables que se comportan igual que un puntero pero que, además, cuando cambian de valor incrementan o decrementan el contador del nodo al que comienzan o dejan de apuntar.

Así quedaría la implementación de la parte básica de la clase `bintree` haciendo uso de punteros inteligentes. En particular, usaremos un puntero inteligente de tipo `shared_ptr` (puntero compartido) en la definición del tipo `Link` y haremos uso de la función `make_shared` cuando haya que construirlo en las constructoras de `bintree`.

---

```

template <class T>
class bintree {
protected:
    struct TreeNode;

    // Uso de puntero inteligente en la definición del tipo Link
    using Link = std::shared_ptr<TreeNode>;
    struct TreeNode {
        TreeNode(Link const& l, T const& e, Link const& r)
            : elem(e), left(l), right(r) {};
        T elem;
        Link left, right;
    };

    // puntero a la raíz del árbol
    Link raiz;

    // constructora privada
    bintree(Link const& r) : raiz(r) {}

public:
    // árbol vacío
    bintree() : raiz(nullptr) {}

    // árbol hoja
    bintree(T const& e) : // Uso de make_shared para la creación de nodos
        raiz(std::make_shared<TreeNode>(nullptr, e, nullptr)) {}

    // árbol con dos hijos
    bintree(bintree<T> const& l, T const& e, bintree<T> const& r) :
        // Uso de make_shared para la creación de nodos
        raiz(std::make_shared<TreeNode>(l.raiz, e, r.raiz)) {}

    // consultar si el árbol está vacío
    bool empty() const {
        return raiz == nullptr;
    }

    // consultar la raíz
    T const& root() const {
        if (empty())
            throw std::domain_error("El arbol vacio no tiene raiz.");
    }

```

---

```

        else return raiz->elem;
    }

    // consultar el hijo izquierdo
bintree<T> left() const {
    if (empty())
        throw std::domain_error("El arbol vacio no tiene hijo izquierdo.");
    else return bintree<T>(raiz->left);
}

// consultar el hijo derecho
bintree<T> right() const {
    if (empty())
        throw std::domain_error("El arbol vacio no tiene hijo derecho.");
    else return bintree(raiz->right);
}

...
};
```

---

## 8. Implementación estática ad-hoc de árboles binarios

Las implementaciones anteriores de los árboles binarios utilizando una estructura jerárquica de nodos con memoria compartida dificulta (o mejor dicho, hace imposible) construir árboles “de arriba a abajo”, es decir añadiendo nodos adicionales como hijos de otros nodos ya existentes.

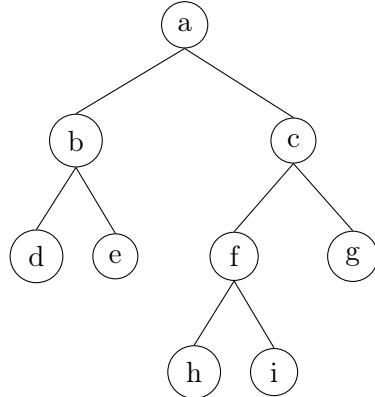
Para poder implementarlo, se puede prescindir de la memoria compartida y dotar al árbol binario de algún tipo de iterador complejo de forma que se extienda el TAD con operaciones de inserción y borrado de nodos en el lugar marcado por el iterador.

Otra alternativa que puede utilizarse es la de almacenar el árbol en un vector en donde cada posición del vector contiene la información de un nodo: elemento almacenado en el nodo y los índices en donde se encuentran el hijo izquierdo y el hijo derecho.

```

class InfoNode {
public:
    ...
protected:
    T elem;
    int indexIz; // -1 si no hay hijo izquierdo
    int indexDr;
};
```

Así por ejemplo, si tenemos el siguiente árbol binario de caracteres:



Lo podemos almacenar en un vector de 9 posiciones<sup>8</sup>. Si el orden de inserción en el árbol anterior fue a, b, c, f, h, g, d, e, i, los nodos estarían colocados en ese mismo orden en el vector, y su contenido sería:

| a | b | c | f | h | g | d | e | i  |     |
|---|---|---|---|---|---|---|---|----|-----|
| 1 | 2 | 6 | 7 | 3 | 5 | 4 | 8 | -1 | ... |

## 9. Árboles generales

Los árboles generales no imponen una limitación *a priori* sobre el número de hijos que puede tener cada nodo. Por tanto, para implementarlos debemos buscar mecanismos generales que nos permitan almacenar un número de hijos variable en cada nodo.

Dos posibles soluciones:

- Cada nodo contiene una lista de punteros a sus nodos hijos. Si el número de hijos se conoce en el momento de la creación del nodo y no se permite añadir hijos a un nodo ya creado, en lugar de una lista podemos utilizar un array.
- Cada nodo contiene un puntero al primer hijo y otro puntero al hermano derecho. De esa forma, podemos acceder al hijo  $i$ -ésimo de un nodo accediendo al primer hijo y luego recorriendo  $i - 1$  punteros al hermano derecho.

## 10. Para terminar...

Terminamos el tema con una aproximación inicial a la solución a la motivación dada al principio del tema.

La implementación puede hacerse con un árbol binario de cadenas. En los nodos hoja se almacena el nombre del animal al que representa, mientras que en cada nodo interno del árbol se almacena la pregunta que discrimina entre los animales almacenados en el hijo izquierdo y en el hijo derecho. En concreto, los animales almacenados en el hijo izquierdo son todo aquellos con los que se responde afirmativamente a la pregunta, mientras que los del hijo derecho son los que no la cumplen.

Así, el árbol con el que se puede conseguir la partida mostrada al principio del tema podría ser:

<sup>8</sup>En realidad lo almacenaríamos en un vector más grande, llevando la cuenta de cuántas posiciones hay ocupadas, igual que se hizo con las implementaciones estáticas de algunos TADs lineales el tema anterior.

---

```
bintree<string> bd =
    bintree<string>(
        bintree<string>("Tigre"),
        "?Tiene cuatro patas?",
        bintree<string>(
            bintree<string>("Ballena"),
            "?Vive en el agua?",
            bintree<string>("Serpiente")
        )
    );
}
```

---

El juego que dirige la partida recibe el árbol y en un bucle va descendiendo por él hasta que llega a una hoja, momento en el que hace la apuesta final:

```
void juegaPartida(const bintree<string> &bd) {
    bintree<string> current = bd;

    while (!esHoja(current)) {
        cout << current.root() << " (Si/No)\n";
        string line;
        cin >> line;
        if (line == "Si")
            current = current.left();
        else if (line == "No")
            current = current.right();
    }

    cout << "Mmmmmm, dejame que piense.....\n";
    cout << "Creo que el animal en que estabas pensando era...: ¡"
        << current.root() << "!\n";

    cout << "Espero haber acertado.\n";
}
```

---

Implementar el aprendizaje consiste en sustituir la hoja en la que se termina por un nodo interno con la pregunta que discrimina y dos hijos hoja con los dos animales.

El TAD `bintree<T>` no permite añadir nodos al árbol (pues éstos son inmutables), por lo que para poder implementar ese aprendizaje habría que extender el TAD o utilizar una implementación distinta, como la que hace uso de vectores estáticos de nodos vista en el apartado 8.

## Notas bibliográficas

Gran parte de este capítulo se basa en el capítulo correspondiente de (Rodríguez Ar-talejo et al., 2011) y de (Peña, 2005); algunos ejercicios son copias casi directas de los encontrados allí.

## Ejercicios

Algunos de los ejercicios puedes probarlos en el portal y juez en línea “Acepta el reto” (<https://www.aceptaelreto.com>). Son los que aparecen marcados con el icono  seguido del número de problema dado en el portal.

1. Extiende la implementación de los árboles binarios que no comparten memoria con las siguientes operaciones:
  - `numNodos`: devuelve el número de nodos del árbol.
  - `esHoja`: devuelve cierto si el árbol es una hoja (los hijos izquierdo y derecho son vacíos).
  - `numHojas`: devuelve el número de hojas del árbol.
  - `talla`: devuelve la talla del árbol.
  - `frontera`: devuelve una lista con todas las hojas del árbol de izquierda a derecha.
  - `(🔗 ACR228)espejo`: devuelve un árbol nuevo que es la imagen especular del original.
  - `minElem`: devuelve el elemento más pequeño de todos los almacenados en el árbol. Es un error ejecutar esta operación sobre un árbol vacío.
2. Implementa las mismas operaciones del ejercicio anterior pero como funciones *externas* al TAD. Estas nuevas funciones tendrá sentido utilizarlas si el TAD está implementado con compartición de memoria. ¿Por qué?
3. Implementa una nueva operación generadora de los árboles binarios que no comparten memoria similar a la ofrecida por el constructor con tres parámetros que construya un árbol a partir de los subárboles izquierdo y derecho pero que, para evitar la sobrecarga de las copias, deje vacíos los árboles que recibe como parámetro.

```
// Pre: iz y dr son dos árboles binarios válidos
bintree<T> construyeYVacia(bintree<T> &iz,
                           const T &elem,
                           bintree<T> &dr);
// Post: devuelve el árbol Cons(iz, elem, dr), y
// altera iz y dr, de forma que empty(iz) y empty(dr).
```

4. Crea una función recursiva:

```
template <typename E>
void printArbol(const bintree<E> &arbol);
```

que escriba por pantalla el árbol que recibe como parámetro, según las siguientes reglas:

- Si el árbol es vacío, escribirá `<vacío>` y después un retorno de carro.
- Si el árbol es un “árbol hoja”, escribirá el contenido de la raíz y un retorno de carro.
- Si el árbol tiene algún hijo, escribirá el contenido del nodo raíz, y recursivamente en las siguientes líneas el hijo izquierdo y después el hijo derecho. Los hijos izquierdo y derecho aparecerán *tabulados*, dejando tres espacios.

Como ejemplo, el dibujo del árbol

```
Cons( Cons (vacío, 3, vacío),
      5,
      Cons (vacío, 6, Cons(vacio, 7, vacio)))
```

sería el siguiente (se han sustituido los espacios por puntos para poder ver más claramente cuántos hay):

```
5
...3
...6
.....<vacío>
.....7
```

5. Dibuja cómo queda la memoria de la máquina tras la ejecución del siguiente código con las dos implementaciones del TAD de los árboles binarios: la que no comparte memoria y la que sí lo hace:

```
bintree<int> vacio;
bintree<int> iz(vacio, 3, vacio);
bintree<int> dr(vacio, 4, vacio);
bintree<int> a(iz, 5, dr);
bintree<int> o = a.left();
```

Dibuja el proceso de destrucción de los árboles si éste se realizara en el siguiente orden:

- a, iz, dr, o.
- o, iz, a, dr.

6. (ACR200) Los famosos números de Fibonacci (cuya serie es 0, 1, 1, 2, 3, 5, 8, 13, ...) tienen un equivalente en el mundo de los árboles. Dado un  $n$ , entendemos por un árbol de Fibonacci de ese  $n$  aquel cuya raíz contiene el número de Fibonacci  $\text{fib}(n)$  y cuyo hijo izquierdo representa el árbol de Fibonacci de  $n - 2$  y el derecho el de  $n - 1$ . Evidentemente, cuando  $n = 0$  el árbol es un árbol hoja con un 0 en la raíz y cuando  $n = 1$  su raíz será 1.

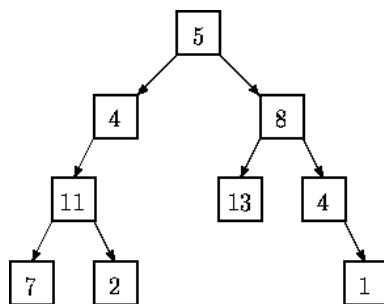
Implementa una función que, dado un  $n$ , devuelva el árbol de Fibonacci. ¿Cuántos nodos tiene el árbol? ¿Podrías encontrar una versión mejorada de la función anterior que maximice la compartición de nodos entre los distintos subárboles? Pinta el árbol de Fibonacci de  $n = 4$  con y sin compartición de estructura.

7. (ACR201) Decimos que un árbol binario es *homogéneo* cuando todos sus subárboles, excepto las hojas, tienen dos hijos no vacíos. Implementa una función que devuelva si un árbol dado es o no homogéneo.
8. (ACR201) Escribe una operación `degenerado()` que devuelva si un árbol es *degenerado*. Se entiende por árbol degenerado aquel en el que cada nodo tiene a lo sumo un hijo izquierdo o un hijo derecho.
9. Extiende la implementación de los árboles binarios con la siguiente operación:

```
/*
Devuelve true si el árbol binario cumple las propiedades
de los árboles ordenados: la raíz es mayor que todos los elementos
del hijo izquierdo y menor que los del hijo derecho y tanto el
hijo izquierdo como el derecho son ordenados.
*/
template <class T>
bool bintree::esOrdenado() const;
```

Implementa la misma operación como función externa al TAD.

10. Dado un árbol binario de enteros, escribe una función que determine si existe un camino desde la raíz hasta una hoja cuyos nodos sumen un valor dado. Por ejemplo, en el siguiente árbol, las sumas de los distintos caminos son 27, 22, 26 y 18, por lo que si se utiliza la función con esos números devolverá `true`, devolviendo `false` en cualquier otro caso<sup>9</sup>.



11. Diremos que un árbol binario es de *altura mínima* si no existe otro árbol binario con el mismo número de nodos con una altura menor. Dado un número de nodos  $n$ , ¿cuál es la altura mínima?
12. (ACR275) Se entiende por árbol balanceado aquel en el que la talla del hijo izquierdo y la del hijo derecho no difieren en más de una unidad y ambos subárboles están a su vez balanceados. Extiende la implementación de los árboles binarios para que incorpore una nueva operación que diga si el árbol binario está balanceado. ¿Qué complejidad tiene? ¿Podrías idear una forma de conseguir la operación en coste  $\mathcal{O}(1)$ ?
13. ¿Notas algo extraño en la siguiente lista de teléfonos?

- Emergencias: 911
- Alicia: 97 625 999
- Bob: 91 12 54 26

Efectivamente, la lista es *inconsistente*: el número de emergencias hace imposible poder llamar a Bob: cuando se marca el 911 la central telefónica dirige la llamada directamente hacia ellas aunque luego sigamos tecleando el resto del número de Bob.

Implementa una función que reciba una lista de números de teléfono e indique si la lista es o no consistente según la explicación anterior<sup>10</sup>.

14. Un árbol de codificación es un árbol binario que almacena en cada una de sus hojas un carácter diferente. La información almacenada en los nodos internos se considera irrelevante. Si un cierto carácter  $c$  se encuentra almacenado en la hoja de posición  $\alpha$  definida como secuencias de 1's y 2's donde un 1 implica descender por el hijo izquierdo y un 2 por el hijo derecho, se considera que  $\alpha$  es el código asignado a  $c$  por

<sup>9</sup>Este ejercicio es una traducción casi directa de [http://uva.onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&category=3&page=show\\_problem&problem=48](http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=3&page=show_problem&problem=48); la imagen del enunciado es una copia directa.

<sup>10</sup>El ejercicio es traducción casi directa de [http://uva.onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=2347](http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2347).

ese árbol de codificación. Más en general, el código de cualquier cadena de caracteres dada se puede construir concatenando los códigos de los caracteres que la forman, respetando su orden de aparición.

- Dibuja el árbol de codificación correspondiente al código siguiente:

| Carácter | Código |
|----------|--------|
| A        | 1.1    |
| T        | 1.2    |
| G        | 2.1.1  |
| R        | 2.1.2  |
| E        | 2.2    |

- Construye el resultado de codificar la cadena de caracteres “RETA” utilizando el código representado por el árbol de codificación anterior.
  - Descifra 1.2.1.1.2.1.2.1.2.1.1 usando el código que estamos utilizando en estos ejemplos, construyendo la cadena de caracteres correspondiente.
  - Desarrolla un módulo que implemente la clase ArbCod que representa a los árboles de codificación descritos, equipada con las siguientes operaciones:
    - Nuevo: genera un árbol de codificación vacío.
    - Inserta: dado un carácter y un código representado como lista de enteros en [1, 2], añade el carácter al árbol en el lugar indicado por la secuencia. La operación no está definida si el carácter ya formaba parte del árbol.
    - codifica: dado un mensaje, devuelve su codificación. La operación no está definida si la cadena contiene algún carácter que no se encuentra codificado en el árbol.
    - decodifica: dado un código, devuelve la cadena que oculta. La operación no está definida si no es posible decodificar toda la entrada.
15. (ACR204)Dado un árbol binario de enteros se dice que éste está *pareado* si la diferencia entre el número de números pares del hijo izquierdo y del hijo derecho no excede la unidad y, además, tanto el hijo izquierdo como el derecho es *pareado*. Implementa una función que diga si un árbol binario de enteros está o no pareado.
16. Extiende los árboles binarios añadiendo una función que cree un recorrido en inorden con paréntesis anidados que identifiquen cada subárbol (incluidos los vacíos). Por ejemplo:
- () representa el árbol vacío.
  - (( )) A ( ) representa un árbol hoja con la A en la raíz.
  - ((( )) B ( )) A ( (( )) C ( )) representa un árbol con la A en la raíz y en cada hijo una hoja con la B y la C respectivamente.
17. Implementa una función que a partir del recorrido en forma de lista anterior reconstruya el árbol de partida.
18. (ACR203)Extiende la implementación de los árboles binarios implementando el siguiente constructor (y todas las funciones auxiliares que necesites):

```
template <class T>
bintree::bintree(const bintree &a1, const bintree &a2);
```

El método construye un nuevo árbol mezclando (o “sumando”) los dos árboles que se reciben como parámetro. Se entiende por mezcla de dos árboles a “solapar” los dos árboles uno encima del otro y guardar el resultado de aplicar el operador suma a los contenidos de los nodos que aparezcan en ambos árboles.

¿Podrías implementar la misma idea con una función externa al TAD?

19. Extiende la implementación de los árboles binarios con una nueva operación, `maxNivel` que obtenga el máximo número de nodos de un nivel del árbol, es decir, el número de nodos del “nivel más ancho”. Analiza su complejidad.
20. Dado un árbol binario de enteros, implementa una función que determine cuál es la rama (camino desde la raíz hasta la hoja) cuya suma sea *mínima*, entendiendo ésta como la suma de todos los nodos por los que pasa.
21. Dado el *dibujo* de un árbol binario que representa una expresión aritmética con los operadores binarios de suma, resta, multiplicación y división y con operandos entre 0 y 9, implementa una función que devuelva el valor de la expresión.

## Recorridos

22. Las implementaciones de los recorridos vistos en la sección 5 se hicieron como métodos de la clase (es decir, extendiendo el TAD de los árboles). Se hizo así porque en ese momento el TAD no compartía las estructuras de nodos por lo que la implementación de otra forma habría sido muy ineficiente. Implementa los tres tipos de recorrido utilizando funciones externas, contando con que todas las operaciones de los árboles tienen complejidad  $\mathcal{O}(1)$ .
23. (DACR215) Diseñar una función que reconstruya un árbol binario a partir de dos listas que contienen respectivamente su recorrido en preorden y en inorden. Suponer, si es necesario, que todos los elementos son distintos.
24. (DACR218) Repite el ejercicio anterior sustituyendo la lista del preorden por el recorrido en postorden.
25. Implementa una función que reciba el recorrido de cada nivel de un árbol ordenado y resconstruya el árbol.
26. Dado el recorrido en preorden de un árbol ordenado sin elementos repetidos, escribe su recorrido en postorden.
27. En el ejercicio 9 del tema anterior nos pedían que evaluáramos una expresión determinada utilizando una cola como estructura de datos auxiliar. Implementa una función que, dada una expresión en notación postfija, la traduzca al formato utilizado por el algoritmo del ejercicio 9.
28. Extiende la implementación de los árboles binarios con las siguientes operaciones:
  - `esCompleto`: observadora, dado un árbol devuelve cierto si el árbol es completo.
  - `esSemicompleto`: igual que la anterior, pero para averiguar si un árbol es semicompleto.

29. Dadas dos listas de enteros  $A$  y  $B$ , implementa una función que diga cuántos árboles binarios pueden construirse que tengan a  $A$  como recorrido en preorden y a  $B$  como recorrido en inorden.

## Montículos

30. Cuando se trabaja con árboles semicompletos, las operaciones generadoras que se plantean son distintas: basta con poder crear un árbol vacío y añadir un nuevo elemento (al final del último nivel, o “abriendo” un nivel nuevo). Con estas operaciones es posible implementar los árboles utilizando vectores en lugar de estructuras jerárquicas de nodos, de forma que los elementos aparecen en el mismo orden en el que aparecerían en un recorrido por niveles.

Dado un índice  $i$  del vector que representa un nodo del árbol, ¿en qué índice aparecerá si hijo izquierdo? ¿y el derecho? ¿y el padre?

31. Un árbol binario se dice que es un *montículo* (en inglés *heap*) si:

- Es un árbol semicompleto.
- La raíz del árbol contiene al elemento *más pequeño*.
- El hijo izquierdo y el hijo derecho son, a su vez, montículos.

Extiende la implementación de los árboles binarios para añadir una nueva operación `esMonticulo` que devuelva si el árbol es o no montículo.

32. Los montículos anteriores, al ser árboles binarios semicompletos, se almacenan más cómodamente en un vector.

Dado un montículo almacenado en un vector en la que se ha añadido un nuevo elemento en el último nivel (que rompe la propiedad de *ser montículo*), implementa una función `flotar` que, con complejidad logarítmica, modifique el vector para que vuelva a ser un montículo.

33. Imaginemos que tenemos un montículo almacenado en un vector. Si cambiamos el valor de la raíz a un valor distinto, el árbol resultante puede dejar de ser montículo. Implementa una función `hundir` que, en ese escenario, modifique el árbol para conseguir recuperar la propiedad de ser montículo en tiempo logarítmico.

34. Utilizando la idea de los montículos (y las operaciones `flotar` y `hundir` de los ejercicios anteriores) implementa el TAD con las operaciones: `min`, `insertar` y `borraMin` que guardan una colección de objetos (no repetidos) y permiten: acceder al elemento más pequeño en tiempo constante, e insertar un nuevo elemento y borrar el más pequeño en tiempo logarítmico.

35. Dado un vector de elementos no repetidos, conviértelo en un montículo utilizando la operación `flotar` sin utilizar espacio adicional.

36. Utilizando el resultado de los ejercicios anteriores, implementa el método de inserción `heapsort` que consiste en ordenar un vector de mayor a menor convirtiéndolo primero en montículo y extrayendo el elemento más pequeño  $n$  veces colocándolo al final.



---

# Capítulo 7

# Diccionarios<sup>1</sup>

---

*If debugging is the process of removing bugs,  
then programming must be the process of putting  
them in.*

Edsger W. Dijkstra

**RESUMEN:** En este tema se introducen los diccionarios y se presentan dos implementaciones distintas. La primera utiliza árboles de búsqueda, mientras que la segunda se basa en tablas dispersas abiertas. También estudiaremos como recorrer los datos almacenados en la tabla usando iteradores, y algunas propiedades deseables en las funciones de localización.

## 1. Motivación

Podemos ver un texto como una secuencia (lista) de palabras, donde cada una de ellas es un `string`. El problema de las *concordancias* consiste en contar el número de veces que aparece cada palabra en ese texto.

Implementar una función que reciba un texto como lista de las palabras (en forma de cadena) y escriba una línea por cada palabra distinta del texto donde se indique la palabra y el número de veces que aparece. La lista debe aparecer ordenada alfabéticamente.

## 2. Introducción

Los diccionarios (en inglés *maps*) son contenedores que almacenan colecciones de pares (*clave, valor*), y que permiten acceder a los valores a partir de sus claves asociadas. Podemos verlas como una *extensión* de los arrays incorporados en los lenguajes de programación pero en los que los índices en vez de estar acotados por un rango de enteros determinado (en lenguajes como C/C++, Java o C# entre 0 y  $n-1$ , siendo  $n$  el tamaño del array) permiten indicar un tipo de índice distinto a los enteros y cuyo rango de valores tampoco está acotado.

De ahí es de donde surge la idea de *diccionario*, en donde hay una *entrada* para cada una de las palabras contenidas en él (de tipo cadena, y no un simple entero) y cuyo valor es, por ejemplo, una lista con las distintas definiciones.

---

<sup>1</sup>Marco Antonio Gómez y Antonio Sánchez Ruiz-Granados son los autores principales de este tema.

Es por esto que los diccionarios están parametrizados por *dos* tipos distintos: el utilizado para la clave y el utilizado para el valor.

Aunque la sintaxis concreta puede variar, conceptualmente, pues, un diccionario es como un array donde se puede utilizar como índices otra cosa distinta a enteros:

---

```
map<string, list<string>> v;

list<string> definiciones;
definiciones.push_back("Libro en el que se recogen...");
definiciones.push_back("Catálogo numeroso de noticias...");

v["diccionario"] = definiciones;

cout << v["diccionario"].front() << endl;
```

---

En este tema veremos dos implementaciones distintas, una basada en árboles binarios (los conocidos como *árboles de búsqueda*) y otra basada en tablas dispersas (*hash tables*). Ambas tienen complejidades mejores que  $\mathcal{O}(n)$ , pero ambas exigen ciertas condiciones a los tipos que pueden utilizarse como clave.

Como última aclaración antes de empezar, decir que a pesar de que en el tema veremos dos implementaciones distintas del TAD map, cada una de ellas será independiente. Es decir no nos planteamos aquí la posibilidad de que exista una clase abstracta a modo de interfaz map de la que hereden ambas o algún otro diseño de clases que venga a reflejar que ambas implementaciones representan el mismo TAD de origen.

## 2.1. Especificación

Desde un punto de vista matemático, los diccionarios son aplicaciones  $t : Key \rightarrow Value$  que asocian a cada clave  $c \in Key$  un determinado valor  $v \in Value$ . *Key* y *Value* serán parámetros del tipo (que proporcionaremos en la plantilla cuando lo implementemos en C++).

Las operaciones públicas del TAD map son:

- *Constructora*  $\rightarrow map$  [Generadora]

Construye un diccionario vacío, es decir, sin elementos.

- *insert* :  $map, Key, Value \rightarrow map$  [Generadora]

Añade un nuevo par (clave, valor) al diccionario. Si la clave ya existía en el diccionario inicial, se sobrescribe su valor asociado. Es decir, los diccionarios *no* permiten almacenar más de un valor por cada clave. Si se desea algo así, o bien se utiliza otro TAD distinto o bien se utiliza una lista de valores como tipo para el valor del diccionario igual que se hizo en el ejemplo del apartado 2.

- *erase* :  $map, Key \rightarrow map$  [Modificadora]

Elimina un par a partir de la clave proporcionada. Si el diccionario no contiene ningún par con dicha clave, no se modifica.

- *contains* :  $map, Key \rightarrow bool$  [Observadora]

Permite averiguar si la clave está o no en el diccionario (es decir, si tiene un valor asociado).

- $at : map, Key \rightarrow Value$  [Observadora parcial]

Devuelve el valor asociado a la clave proporcionada, siempre que la clave exista en el diccionario.

- $empty : map \rightarrow bool$  [Observadora]

Indica si el diccionario está vacío o no.

- $size : map \rightarrow int$  [Observadora]

Indica si el diccionario está vacío o no.

Las operaciones generadoras presentan una peculiaridad que no ha aparecido hasta ahora: un `insert` puede *anular* el resultado de un `insert` anterior. Eso implica que hay *más de una forma* de construir el mismo diccionario.

## 2.2. Implementación con acceso basado en búsqueda

Usando las estructuras de datos que ya conocemos, podemos implementar las tablas como colecciones de parejas (*clave, valor*), en las que el acceso por clave se implementa mediante una búsqueda en la estructura correspondiente. A continuación planteamos dos posibles implementaciones usando listas y árboles de búsqueda.

Una manera sencilla de implementar las tablas es mediante una lista de pares (*clave, valor*). Dependiendo de si la lista está ordenada o no, tendríamos los siguientes costes asociados a las operaciones:

| Operación    | Lista desordenada | Lista ordenada basada en vectores |
|--------------|-------------------|-----------------------------------|
| Constructora | $\mathcal{O}(1)$  | $\mathcal{O}(1)$                  |
| insert       | $\mathcal{O}(n)$  | $\mathcal{O}(n)$                  |
| erase        | $\mathcal{O}(n)$  | $\mathcal{O}(n)$                  |
| contains     | $\mathcal{O}(n)$  | $\mathcal{O}(\log n)$             |
| at           | $\mathcal{O}(n)$  | $\mathcal{O}(\log n)$             |
| empty        | $\mathcal{O}(1)$  | $\mathcal{O}(1)$                  |

La operación más habitual cuando se utilizan tablas es `at` (o alguna de sus versiones), que consulta el valor asociado a una clave, por lo que usar una implementación basada en una lista desordenada no parece la elección más acertada. Aún así, incluimos esta implementación en la tabla por su simplicidad y como base con la que poder comparar.

Un dato que puede llamar la atención es el coste lineal de la operación `inserta` cuando usamos listas desordenadas. Este coste se produce porque antes de insertar el nuevo par (*clave, valor*) es necesario comprobar si la clave ya estaba en la tabla para, en ese caso, modificar su valor asociado. La operación de inserción tiene coste  $\mathcal{O}(n)$  porque la búsqueda tiene coste  $\mathcal{O}(n)$ .

Podemos mejorar el coste de las operaciones usando una lista ordenada basada en vectores. Con esta nueva implementación las operaciones `at` y `contains` pasan a ser logarítmicas, ya que se pueden resolver usando búsqueda binaria. Sin embargo, las operaciones `insert` y `erase` siguen siendo lineales, ya que para insertar o borrar un elemento en un vector debemos desplazar todos los que hay a la derecha.

¿Mejorarían los costes si usamos una lista enlazada ordenada? Pues en realidad no, porque el algoritmo de búsqueda binaria no se puede aplicar a listas enlazadas, ya que necesita acceder al elemento central de un intervalo en tiempo constante.

Se necesita, pues, otra estrategia distinta que permita hacer reducir la complejidad de las operaciones.

### 3. Árboles de búsqueda

La implementación de los diccionarios mediante lo que se conoce como árboles de búsqueda intenta conseguir las ventajas de la búsqueda binaria (y sus complejidades logarítmicas) eliminando las desventajas de las inserciones lineales. Para eso *evita* almacenar todos los elementos seguidos en memoria.

Para ser capaces de entenderla tenemos que dar primero un pequeño paso atrás y hablar de los árboles binarios ordenados. En el ejercicio 9 del tema anterior nos pedían implementar la siguiente función en los árboles binarios:

```
/**  
 * Devuelve true si el árbol binario cumple las propiedades  
 * de los árboles ordenados: la raiz es mayor que todos los elementos  
 * del hijo izquierdo y menor que los del hijo derecho y tanto el  
 * hijo izquierdo como el derecho son ordenados.  
 */  
template <class T>  
bool bintree::esOrdenado() const;
```

Para que ésta operación tenga sentido, es evidente que el tipo T debe poderse ordenar (en C++ eso se traduce a que tienen implementada la comparación mediante el operador `<`). Entenderemos que un árbol está ordenado si su recorrido en inorden está ordenado en orden estrictamente mayor<sup>2</sup>. De lo anterior se deduce inmediatamente que la raíz del árbol es mayor que todos los elementos del hijo izquierdo y menor que todos los elementos del hijo derecho; además tanto el hijo izquierdo como el hijo derecho deben estar, a su vez, ordenados.

Con un árbol ordenado, saber si un elemento está en el árbol no requiere un recorrido por todos los nodos del mismo, sino un recorrido que recuerda a la búsqueda binaria. En efecto, durante el proceso de búsqueda se mira si la raíz contiene el elemento y en caso contrario, se busca únicamente en el hijo izquierdo o derecho dependiendo del resultado de la comparación del elemento buscado y el contenido en la raíz.

La ventaja de los árboles frente a los vectores ordenados, además, es que la inserción de un nuevo elemento *no* tiene coste lineal, pues no necesitaremos desplazar todos los elementos para hacer hueco en el vector; basta con encontrar en qué lugar del árbol debe ir el elemento para mantener el árbol ordenado y crear el nuevo nodo. Por su parte el borrado, aunque más difícil de ver de forma intuitiva, también presenta un escenario similar (no hay que desplazar todos los elementos a la izquierda para “borrar” el elemento).

Parece fácil ver que si los elementos están distribuidos uniformemente, es decir si cada nodo tiene aproximadamente el mismo número de nodos en el subárbol izquierdo y el subárbol derecho, la talla del árbol es del orden de  $\log(n)$ . Eso, unido al hecho de que las operaciones anteriores lo que hacen es *descender* por el árbol en busca del elemento, hace que bajo la premisa de un árbol balanceado, éstas tengan coste logarítmico (igual que en búsquedas binarias sobre vectores ordenados, pues al fin y al cabo cada vez que descendemos por uno de los lados del árbol dejamos atrás la mitad de los elementos).

Los árboles de búsqueda consisten en utilizar esta misma idea para la implementación de los diccionarios: en vez de almacenar un único elemento en el nodo, utilizaremos dos. Por un lado el valor que utilizaremos para ordenar (que hace las veces de *clave*) y por otro la información adicional asociada a ella (el *valor*). En cada nodo guardaremos, pues, una *pareja*: la clave (por la que se ordena) y su valor asociado. Eso implica automáticamente que

---

<sup>2</sup>Eso implica que no permitimos la aparición de elementos repetidos; existen otras variaciones de estos árboles que sí lo hacen.

los árboles de búsqueda están parametrizados, al menos, *por dos tipos distintos* en vez de sólo por uno: el tipo de la clave (que debe poderse ordenar) y el tipo del valor. Añadiremos un tercer tipo (opcional) a la plantilla para poder proporcionar un comparador (objeto cuya único método implemente la operación de orden entre claves).

### 3.1. Implementación

La implementación del diccionario utilizando árboles de búsqueda la llamaremos directamente map para coincidir con el nombre del tipo análogo de la STL de C++<sup>3</sup>. Se basa en tener una estructura jerárquica de nodos, igual que los árboles binarios del tema anterior. La diferencia fundamental es que en esta ocasión esos nodos guardan dos elementos y que no es necesario hacer uso de punteros inteligentes:

---

```
template <class Clave, class Valor, class Comparador = std::less<Clave>>
class map {
public:
    // parejas <clave, valor> mediante std::pair
    using clave_valor = std::pair<const Clave, Valor>;
protected:
    using map_t = map<Clave, Valor, Comparador>; // Alias para acortar

    /*
        Clase nodo que almacena internamente la pareja <clave, valor>
        y punteros al hijo izquierdo y al hijo derecho.
    */
    struct TreeNode;
    using Link = TreeNode *;
    struct TreeNode {
        clave_valor cv;
        Link iz, dr;
        TreeNode(clave_valor const& e, Link i = nullptr, Link d = nullptr)
            : cv(e), iz(i), dr(d) {}
    };

    // puntero a la raíz de la estructura jerárquica de nodos
    Link raiz;

    // número de parejas <clave, valor>
    int nelems;

    // objeto función que compara elementos
    Comparador menor;
};
```

---

Las operaciones generales que vimos para los árboles binarios siguen siendo válidas (la liberación, copia, etc.), pero extiéndolas cuando corresponda para que tengan en cuenta la existencia de dos valores en vez de sólo uno.

El invariante de la representación del árbol de búsqueda exige lo mismo que en el caso de los árboles binarios (añadiendo que se cumpla el invariante de la representación tanto de la clave como del valor), y que se mantenga el orden de las claves<sup>4</sup>:

<sup>3</sup>TreeMap es otro nombre comúnmente utilizado para estos diccionarios, por ejemplo, en Java.

<sup>4</sup>La definición podría haberse hecho también en base al recorrido en inorden.

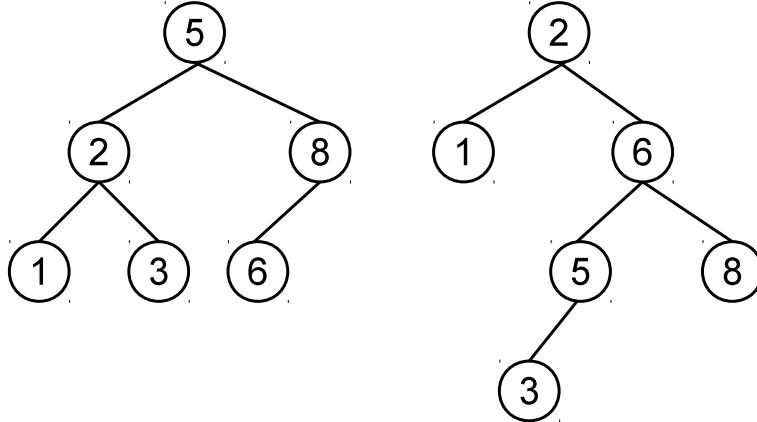


Figura 1: Dos árboles de búsqueda distintos pero equivalentes.

$$\begin{aligned}
 & R_{map_{(C,V)}}(\mathbf{p}) \\
 \iff_{def} & R_{bintree_{Pareja(C,V)}}(\mathbf{p})^5 \wedge \\
 & ordenado(\mathbf{p.raiz})
 \end{aligned}$$

donde

$$\begin{aligned}
 ordenado(\mathbf{ptr}) &= true & \mathbf{si} \quad \mathbf{ptr} = \text{null} \\
 ordenado(\mathbf{ptr}) &= \forall c \in claves(\mathbf{ptr.iz}) : c < \mathbf{ptr.cv.first} \wedge \\
 & \quad \forall c \in claves(\mathbf{ptr.dr}) : \mathbf{ptr.cv.first} < c \wedge \\
 & \quad ordenado(\mathbf{ptr.iz}) \wedge ordenado(\mathbf{ptr.dr}) & \mathbf{si} \quad \mathbf{ptr} \neq \text{null}
 \end{aligned}$$

$$\begin{aligned}
 claves(\mathbf{ptr}) &= \emptyset & \mathbf{si} \quad \mathbf{ptr} = \text{null} \\
 claves(\mathbf{ptr}) &= \{\mathbf{ptr.cv.first}\} \cup claves(\mathbf{ptr.iz}) \cup claves(\mathbf{ptr.dr}) & \mathbf{si} \quad \mathbf{ptr} \neq \text{null}
 \end{aligned}$$

La relación de equivalencia, sin embargo, no sigue la misma idea que en los árboles binarios. Para entender por qué basta ver los dos árboles de búsqueda de la figura 1, que guardan la misma información pero tienen una estructura arbórea totalmente distinta.

En realidad dos árboles de búsqueda son equivalentes si almacenan los mismos elementos. Los árboles de búsqueda tienen la misma interfaz que los conjuntos, y su relación de equivalencia también es la misma:

$$\begin{aligned}
 \mathbf{a1} &\equiv_{map_T} \mathbf{a2} \\
 \iff_{def} & elementos(\mathbf{a1}) \equiv_{Conjunto_{Pareja(C,V)}} elementos(\mathbf{a2})
 \end{aligned}$$

donde *elementos* devuelve un conjunto con todos los pares (clave, valor) contenidos en el árbol.

---

<sup>5</sup>Extendido para considerar claves y valores; el *Pareja*(C, V) viene a indicarlo.

### 3.1.1. Implementación de la constructora y de las observadoras `empty` y `size`

El árbol de búsqueda se implementa, igual que los árboles binarios, guardando un puntero a la raíz. Por lo tanto el constructor sin parámetros para generar un árbol de búsqueda vacío lo inicializa a `nullptr`. Las operaciones `empty` y `size` se implementan por tanto de manera trivial:

---

```
// constructor (diccionario vacío)
map(Comparador c = Comparador()) : raiz(nullptr), nelems(0), menor(c) {}

bool empty() const {
    return raiz == nullptr;
}

int size() const {
    return nelems;
}
```

---

### 3.1.2. Implementación de `contains` y `at`

Las operaciones observadoras que permiten averiguar si una clave aparece en el árbol o acceder al valor asociado a una clave se basan en un método auxiliar que trabaja con la estructura de nodos y busca el nodo que contiene una clave dada, y que se implementa de manera trivial con recursión. La operación `contains` tomará el nombre de `count` para tener una interfaz equivalente a la operación análoga del TAD `map` de la STL de C++.

En lugar de devolver un booleano devuelve un entero que será 1 en caso de encontrarse la clave y 0 en caso contrario (de ahí el nombre de `count`).

---

```
// Método protegido/privado
/**
Busca una clave en la estructura jerárquica de
nodos cuya raíz se pasa como parámetro, y devuelve
el nodo en el que se encuentra (o nullptr si no está).
*/
Link busca(Clave const& c, Link a) const {
    if (a == nullptr) {
        return nullptr;
    }
    else if (menor(c, a->cv.first)) {
        return busca(c, a->iz);
    }
    else if (menor(a->cv.first, c)) {
        return busca(c, a->dr);
    }
    else { // c == a->cv.first
        return a;
    }
}
```

---

Con ella las operaciones `count` y `at` son casi inmediatas:

---

```
int count(Clave const& c) const {
    return (busca(c, raiz) != nullptr) ? 1 : 0;
```

---

---

```
Valor const& at(Clave const& c) const {
    Link p = busca(c, raiz);
    if (p == nullptr)
        throw std::out_of_range("La clave no se puede consultar");
    return p->cv.second;
}
```

---

También incluiremos (al igual que se hace en la clase map de la STL de C++) una versión más avanzada de la operación at, que se implementa mediante el operador [] y que además de permitir modificar el valor asociado en caso de encontrar la clave buscada, inserta un nuevo par con la clave buscada (y un valor construido por defecto) en el caso de no encontrarla.

---

```
Valor & operator[](Clave const& c) {
    return corchete(c, raiz);
}

Valor & corchete(Clave const& c, Link & a) {
    if (a == nullptr) {
        // se inserta la nueva clave, con un valor por defecto
        a = new TreeNode(clave_valor(c, Valor()));
        ++nelems;
    }
    return a->cv.second;
}
else if (menor(c, a->cv.first)) {
    return corchete(c, a->iz);
}
else if (menor(a->cv.first, c)) {
    return corchete(c, a->dr);
}
else { // la clave ya está, se devuelve el valor asociado
    return a->cv.second;
}
}
```

---

### 3.1.3. Implementación de la inserción

La inserción debe garantizar que:

- Si la clave ya aparecía en el árbol, el valor antiguo se sustituye por el nuevo.
- Tras la inserción, el árbol de búsqueda sigue cumpliendo el invariante de la representación, es decir, sigue estando ordenado por claves.

La implementación debe “encontrar el hueco” en el que se debe crear el nuevo nodo, y si por el camino descubre que la clave ya existía, sustituir su valor.

Lo importante de la implementación es darse cuenta de que la raíz de la estructura jerárquica *puede cambiar*. En concreto, si el árbol de búsqueda estaba vacío, en el momento de insertar el elemento se crea un nuevo nodo que pasa a ser la raíz (de ahí el paso por referencia del puntero al nodo). Teniendo esto en cuenta la implementación (con un método auxiliar recursivo) sale casi sola:

---

```
// solamente se inserta si la clave no existe ya en el diccionario
bool insert(Clave_Valor const& cv) {
```

---

```

    return inserta(cv, raiz);
}

// Método protegido
bool inserta(clave_valor const& cv, Link & a) {
    if (a == nullptr) {
        // se inserta el nuevo par <clave, valor>
        a = new TreeNode(cv);
        ++nelems;
        return true;
    }
    else if (menor(cv.first, a->cv.first)) {
        return inserta(cv, a->iz);
    }
    else if (menor(a->cv.first, cv.first)) {
        return inserta(cv, a->dr);
    }
    else { // la clave ya está
        return false;
    }
}

```

---

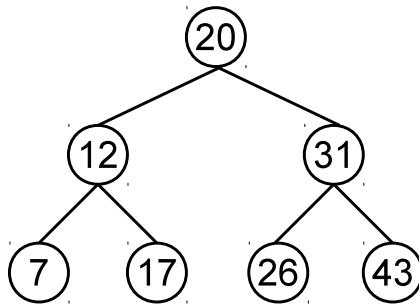
Si suponemos que el árbol está equilibrado, la complejidad del método anterior es logarítmica pues en cada llamada recursiva se divide el tamaño de los datos entre dos.

### 3.1.4. Implementación del borrado

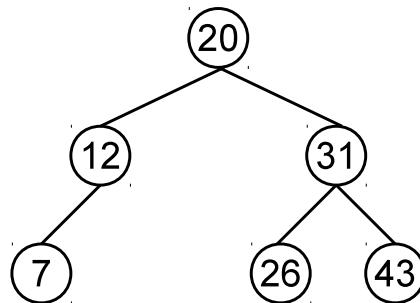
La operación de borrado es más complicada que la de la inserción, porque puede exigir reestructurar los nodos para mantener la estructura ordenada por claves. Si nos piden eliminar la clave  $c$ , se busca el nodo que la contiene y:

- Si la búsqueda fracasa (la clave no está), se termina sin modificar el árbol.
- Si la búsqueda tiene éxito, se localiza un nodo  $\alpha$ , que es el que hay que borrar. Para hacerlo:
  - Si  $\alpha$  es hoja, se puede eliminar directamente (actualizando el puntero del padre).
  - Si  $\alpha$  tiene un sólo hijo, se elimina el nodo  $\alpha$  y se coloca en su lugar el subárbol hijo cuya raíz quedará en el lugar del nodo  $\alpha$ .
  - Si  $\alpha$  tiene dos hijos la estrategia que utilizaremos será “subir” el elemento más pequeño del hijo derecho (que no tendrá hijo izquierdo, pues de otra forma no sería el más pequeño) a la raíz. Para eso:
    - Se busca el nodo  $\alpha'$  más pequeño del hijo derecho.
    - Si ese nodo tiene hijo derecho, éste pasa a ocupar su lugar.
    - El nodo  $\alpha'$  pasa a ocupar el lugar de  $\alpha$ , de forma que su hijo izquierdo y derecho cambian a los hijos izquierdo y derecho de la raíz antigua.

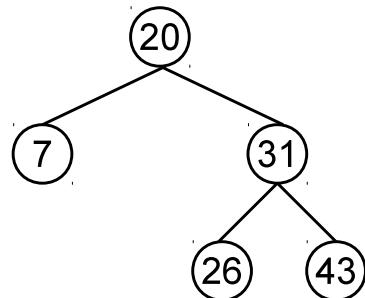
A modo de ejemplo, a continuación mostramos la evolución de un árbol de búsqueda  $a$  cuando vamos borrando sus nodos:



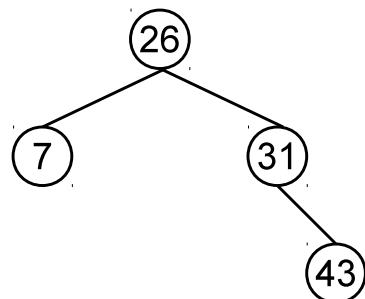
■ a.erase(17)



■ a.erase(12)



■ a.erase(20)



Igual que ocurría con la inserción, la implementación utiliza métodos auxiliares; como la raíz puede cambiar éstos deben pasar el parámetro del puntero al nodo por referencia:

---

```

bool erase(Clave const& c) {
    return borra(c, raiz);
}
  
```

---

```

bool borra(Clave const& c, Link & a) {
    if (a == nullptr) {
        return false;
    } else if (menor(c, a->cv.first)) {
        return borra(c, a->iz);
    }
    else if (menor(a->cv.first, c)) {
        return borra(c, a->dr);
    }
    else { // c == a->cv.first
        if (a->iz == nullptr || a->dr == nullptr) {
            Link aux = a;
            a = (a->iz == nullptr) ? a->dr : a->iz;
            --nelems;
            delete aux;
        }
        else { // tiene dos hijos
            subirMenorHD(a);
            --nelems;
        }
        return true;
    }
}

static void subirMenorHD(Link & a) {
    Link b = a->dr, padre = nullptr;
    while (b->iz != nullptr) {
        padre = b;
        b = b->iz;
    }
    if (padre != nullptr) {
        padre->iz = b->dr;
        b->dr = a->dr;
    }
    b->iz = a->iz;
    delete a;
    a = b;
}

```

---

### 3.2. Coste de las operaciones

Los costes de las operaciones de los diccionarios implementados mediante árboles de búsqueda, si suponemos árboles equilibrados, son los siguientes:

| Operación    | Árboles de búsqueda   |
|--------------|-----------------------|
| Constructora | $\mathcal{O}(1)$      |
| insert       | $\mathcal{O}(\log n)$ |
| count        | $\mathcal{O}(\log n)$ |
| at           | $\mathcal{O}(\log n)$ |
| erase        | $\mathcal{O}(\log n)$ |
| empty        | $\mathcal{O}(1)$      |

Es importante resaltar que para que un tipo pueda utilizarse como clave en un dic-

cionario implementado como árbol de búsqueda, éste debe poseer una relación de orden, ya que los árboles de búsqueda almacenan los elementos ordenados. A cambio, conseguimos que las operaciones sean logarítmicas (si los árboles se mantienen equilibrados). Esta operación de orden puede proporcionarse como tercer parámetro al instanciar el tipo (en la declaración de la variable), o en caso de no proporcionarse se asumirá la existencia del operador de orden `<` del tipo Clave.

Por otro lado, insistimos en que para que esas complejidades sean ciertas, el árbol debe estar equilibrado. Y eso sólo se garantiza si las inserciones y borrados realizados sobre el diccionario son aleatorias. Si el usuario de la clase hace algo como lo siguiente:

---

```
map<int, int> tablaMultiplicarDel17;
for (int i = 1; i <= 100; ++i)
    tablaMultiplicarDel17.insert({i, 17*i});
```

---

estará construyendo, seguramente sin darse cuenta, un árbol degenerado e incurrirá en costes lineales en las operaciones de búsqueda sobre él. Las implementaciones reales de librerías de colecciones de datos se basan en estructuras de datos más avanzadas que utilizan las ideas de los árboles de búsqueda vistos aquí, pero que incluyen lógica de *auto-balanceo* en las operaciones de inserción y borrado; si se determina que el árbol corre el riesgo de desequilibrarse se reestructura.

### 3.3. Recorrido de los elementos mediante un iterador

En los árboles de búsqueda podríamos también añadir operaciones para el recorrido de sus elementos (preorden, por niveles, etc.). Sin embargo, dado que el único recorrido que parece tener sentido es el recorrido en inorden (pues las claves salen en orden creciente), vamos a extender la clase añadiendo la posibilidad de recorrerlo mediante un iterador.

El iterador permitirá acceder tanto a la clave como al valor del elemento visitado (devolviendo un par con ambos datos). Igual que en el caso de las listas, tendremos dos versiones del iterador, por un lado el `const_iterator` que no permitirá modificar los datos, y por otro lado el `iterator` que permitirá modificar el *valor* asociado (no se permite cambiar la clave, pues pondría en peligro el invariante de la representación: el árbol podría dejar de estar ordenado).

El recorrido, sin embargo, no es tan fácil como en el caso de las listas, porque averiguar el siguiente elemento a un nodo dado no es directo. En concreto, hay dos casos:

- Cuando el nodo visitado actualmente tiene hijo derecho, el siguiente nodo a visitar será el elemento más pequeño del hijo derecho. Éste se obtiene bajando siempre por la rama de la izquierda hasta llegar a un nodo que no tiene hijo izquierdo.
- Si el nodo visitado no tiene hijo derecho, el siguiente elemento a visitar es el *ascendiente más cercano* que aún no ha sido visitado. Dado que la estructura jerárquica mantiene punteros hacia los hijos pero no hacia los padres, necesitamos una estructura de datos auxiliar. En particular, el iterador mantendrá una *pila* con todos los ascendientes que aún quedan por recorrer. En la búsqueda del hijo más pequeño descrito anteriormente vamos descendiendo por una rama, vamos apilando todos esos descendientes para poder visitarlos después.

El recorrido termina cuando el nodo actual no tiene hijo derecho y la pila queda vacía. En ese caso se cambia el puntero interno a `nullptr` para indicar que estamos “fuera” del recorrido.

---

```

// iteradores que recorren los pares de menor a mayor clave
template <class Apuntado>
class Iterador {
public:
    Apuntado & operator*() const {
        if (act == nullptr)
            throw std::out_of_range("No hay elemento a consultar");
        return act->cv;
    }

    Iterador & operator++() { // ++ prefijo
        next();
        return *this;
    }

    bool operator==(Iterador const& that) const {
        return act == that.act;
    }
    bool operator!=(Iterador const& that) const {
        return !(this->operator==(that));
    }
}

protected:
    friend class map;
    Link act;
    std::stack<Link> ancestros; // antecesores no visitados

// construye el iterador al primero
Iterador(Link raiz) { act = first(raiz); }

// construye el iterador al último
Iterador() : act(nullptr) {}

Link first(Link ptr) {
    if (ptr == nullptr) {
        return nullptr;
    } else { // buscamos el nodo más a la izquierda
        while (ptr->iz != nullptr) {
            ancestros.push(ptr);
            ptr = ptr->iz;
        }
        return ptr;
    }
}

void next() {
    if (act == nullptr) {
        throw std::out_of_range("El iterador no puede avanzar");
    } else if (act->dr != nullptr) { // primero del hijo derecho
        act = first(act->dr);
    } else if (ancestros.empty()) { // hemos llegado al final
        act = nullptr;
    } else { // podemos retroceder
        act = ancestros.top();
        ancestros.pop();
    }
}

```

```

        }
    }
}; // fin de la clase Iterador

public: // de la clase map

    // iterador que no permite modificar el elemento apuntado
    using const_iterator = Iterador<clave_valor const>;

    const_iterator cbegin() const {
        return const_iterator(raiz);
    }

    const_iterator cend() const {
        return const_iterator();
    }

    // iterador que sí permite modificar el elemento apuntado (su valor)
    using iterator = Iterador<clave_valor>;

    iterator begin() {
        return iterator(raiz);
    }

    iterator end() {
        return iterator();
    }

```

---

Las operaciones funcionan de forma similar a lo visto en las listas: la operación `begin` (o `cbegin`) devuelve un iterador al principio del recorrido, mientras que `end` (`cend()`) devuelve un iterador que queda *fuerza* del recorrido.

### 3.4. Búsqueda en el diccionario con iteradores

Un problema de la implementación anterior de los diccionarios es que si un usuario quiere protegerse de los accesos indebidos al llamar al método `at` debe asegurarse primero de que la clave existe utilizando `count`. Eso fuerza a hacer dos búsquedas sobre el árbol. Si, además, ese mismo usuario quiere posteriormente modificar el valor asociado a esa tabla, incurrirá en un nuevo recorrido al llamar a `insert`.

Una solución que reduce los tres recorridos anteriores a uno consiste en añadir un método adicional de *búsqueda* de un elemento, al que llamaremos `find`, que en lugar de devolver el `bool` o el `Valor`, devuelva el *iterador* al punto donde está la clave buscada (o al final del recorrido si no está). Implementaremos la búsqueda propiamente dicha mediante la siguiente constructora en la clase interna `Iterador`, la cual será invocada desde el nuevo método `find`.

---

```

class Iterador {
    ...
    Iterador(map_t const* m, Clave const& c) {
        act = m->raiz;
        bool encontrado = false;
        while (act != nullptr && !encontrado) {
            if (m->menor(c, act->cv.first)) {
                ancestros.push(act);

```

---

---

```

        act = act->iz;
    } else if (m->menor(act->cv.first, c)) {
        act = act->dr;
    } else
        encontrado = true;
    }
if (!encontrado) { // vaciamos la pila
    // act == nullptr
    ancestros = std::stack<Link>();
}
}
...
}; //fin de la clase Iterador

// En la clase map
const_iterator find(Clave const& c) const {
    return const_iterator(this, c);
}

iterator find(Clave const& c) {
    return iterator(this, c);
}

```

---

## 4. Tablas dispersas

Las tablas dispersas se basan en almacenar en un vector los *valores* y usar las *claves* como índices. De esa forma, dada una clave podemos acceder a la posición del vector que contiene su valor asociado en tiempo constante. Las tablas dispersas permiten implementar todas las operaciones en tiempo  $O(1)$  en promedio, aunque en el caso peor *insert*, *count*, *at* y *erase* serán  $O(n)$ .

¿Cómo asociamos cada posible clave a una posición del vector? Obviamente esta idea no puede aplicarse si el conjunto de claves posible es demasiado grande. Por ejemplo, si usamos como claves cadenas de caracteres con un máximo de 8 caracteres elegidos de un conjunto de 52 caracteres posibles, habría un total de

$$L = \sum i : 1 \leq i \leq 8 : 52^i$$

claves distintas.

Es absolutamente impensable reservar un vector de tamaño  $L$  para implementar la tabla del ejemplo anterior, sobre todo si tenemos en cuenta que el conjunto de cadenas que llegará a utilizarse en la práctica será mucho menor. Necesitamos algún mecanismo que permita establecer una correspondencia entre un conjunto de claves potencialmente muy grande y un vector de valores mucho más pequeño.

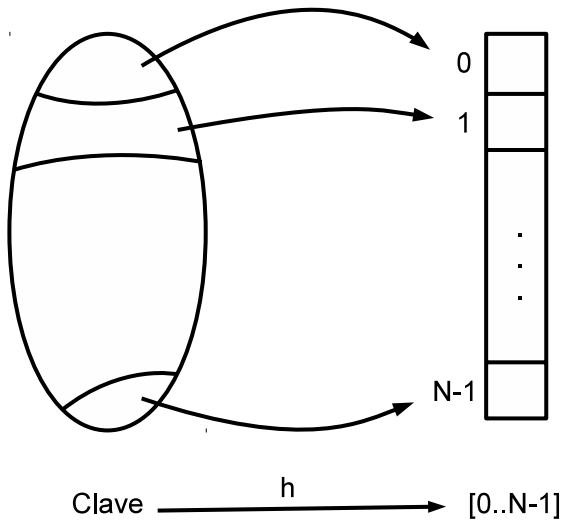
### 4.1. Función de localización

Lo que sí tiene sentido es reservar un vector de  $N$  posiciones para almacenar los valores (siendo  $N$  mucho más pequeño que  $L$ ) y usar una *función de localización* (*hashing function*) que asocia a cada clave un índice del vector

$$h : Clave \rightarrow [0..N - 1]$$

de manera que dada una clave  $c$ ,  $h(c)$  represente la posición del vector que debería contener su valor asociado.

Puesto que el número de claves posible,  $L$ , es mucho mayor que el número de posiciones del vector,  $N$ , la función de localización  $h$  no puede ser inyectiva. En otras palabras, existen varias claves distintas que se asocian al mismo índice dentro del vector. Gráficamente la situación es la siguiente:



Para que la búsqueda funcione de manera óptima, las funciones de localización deben tener las siguientes propiedades:

- *Eficiencia*: el coste de calcular  $h(c)$  debe ser bajo.
- *Uniformidad*: el reparto de claves entre posiciones del vector debe ser lo más uniforme posible. Idealmente, para una clave  $c$  elegida al azar la probabilidad de que  $h(c) = i$  debe valer  $1/N$  para cada  $i \in [0..N - 1]$ .

Supongamos que tenemos un vector de 16 posiciones ( $N = 16$ ) y que usamos cadenas de caracteres como claves. Una posible función de localización es la siguiente:

$$h(c) = \text{ord}(\text{ult}(c)) \bmod 16$$

donde  $\text{ult}(c)$  devuelve el último carácter de la cadena, y  $\text{ord}$  devuelve el código ASCII de un carácter. Usando esta función de localización tenemos:

$$\begin{aligned} h("Fred") &= \text{ord}('d') \bmod 16 = 100 \bmod 16 = 4 \\ h("Joe") &= \text{ord}('e') \bmod 16 = 101 \bmod 16 = 5 \\ h("John") &= \text{ord}('n') \bmod 16 = 110 \bmod 16 = 14 \end{aligned}$$

Aunque esta función de localización no es demasiado buena, la seguiremos usando en los siguientes ejemplos debido a su sencillez. Más adelante, en el apartado 6, discutiremos algunas ideas para definir mejores funciones de localización.

## 4.2. Colisiones

Como ya hemos explicado, en general la función de localización no puede ser inyectiva. Cuando se encuentran dos claves  $c$  y  $c'$  tales que

$$c \neq c' \wedge h(c) = h(c')$$

se dice que se ha producido una *colisión*. Se dice también que  $c$  y  $c'$  son *claves sinónimas* con respecto a  $h$ .

Es fácil encontrar claves sinónimas con respecto a la función de localización que acabamos de definir:

$$h("Fred") = h("David") = h("Violet") = h(Roland") = 4$$

En realidad, la probabilidad de que no se produzcan colisiones es mucho más baja de lo que podríamos pensar. Mediante un cálculo de probabilidades puede demostrarse la llamada “paradoja del cumpleaños”, que dice que en un grupo de 23 o más personas, la probabilidad de que al menos dos de ellas cumplan años el mismo día del año es mayor que  $1/2$ .

Debemos pensar, por tanto, qué hacer cuando se produzca una colisión. Fundamentalmente, existen dos estrategias que dan lugar a dos tipos de tablas dispersas:

- *Tablas abiertas*: cada posición del vector almacena una lista de parejas (clave, valor) con todos los pares que colisionan en dicha posición.
- *Tablas cerradas*: si al insertar un par (clave, valor) se produce una colisión, se busca otra posición del vector vacía donde almacenarlo. Para ello se van comprobando los índices del vector en algún orden determinado hasta alcanzar alguna posición vacía (técnicas de *relocalización*).

En este tema nos centraremos en el estudio de las tablas dispersas abiertas, pero animamos a los estudiantes que lo deseen a profundizar a través de la lectura de los capítulos correspondientes de los libros (Rodríguez Artalejo et al., 2011) y (Peña, 2005). En realidad, cada tipo de tabla tiene sus ventajas y sus inconvenientes: las tablas abiertas son más sencillas de entender y suelen tener mejor rendimiento, pero también ocupan más memoria que las cerradas.

## 5. Tablas dispersas abiertas

En las tablas abiertas, cada posición del vector contiene una *lista de colisión* que almacena los pares con claves sinónimas. Nosotros vamos a implementar estas listas de colisión como listas enlazadas en las que cada nodo almacena una clave y un valor.

Llamaremos a la clase `unordered_map` para coincidir con la clase análoga de la STL de C++. En Java por ejemplo se denomina `HashMap` mientras que en C# es `Dictionary`.

La operación *insert* calculará el índice del vector asociado a la nueva clave, y añadirá un nuevo nodo a la lista correspondiente si la clave no existía, o modificará su valor asociado si ya existía. De manera simétrica, la operación de borrado calculará el índice asociado a la clave que recibe como parámetro y a continuación buscará en la lista correspondiente algún nodo que contenga dicha clave para eliminarlo.

Veamos un ejemplo. Supongamos que tenemos una tabla abierta implementada con un vector de 16 posiciones y la función de localización de siempre. Tras realizar las siguientes operaciones:

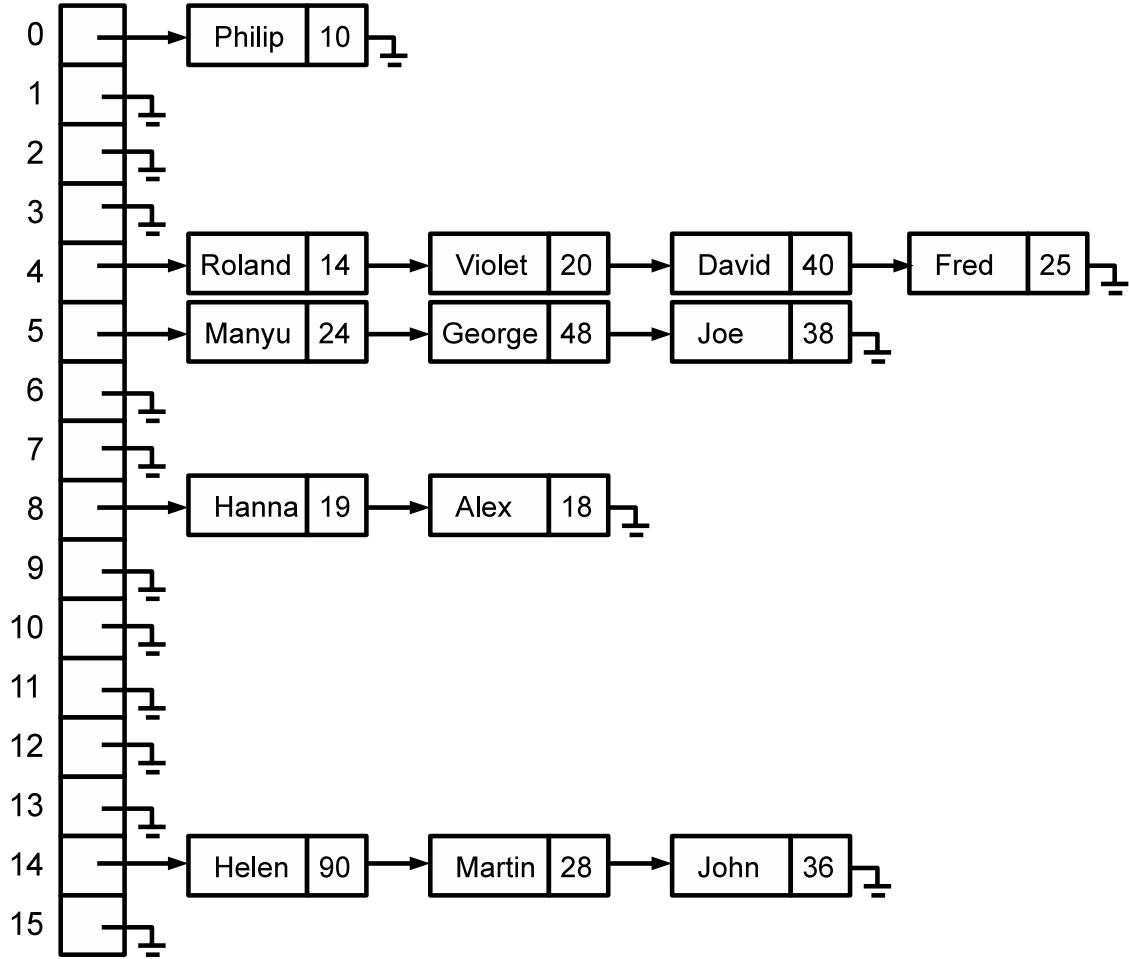


Figura 2: Ejemplo de tabla abierta. La función de localización usada en el ejemplo es claramente mejorable.

```
unordered_map<std::string, int> edades; // (*)  
edades.insert({"Fred", 25});           edades.insert({"Alex", 18});  
edades.insert({{"Philip", 10}});        edades.insert({"Joe", 38});  
edades.insert({"John", 36});           edades.insert({"Hanna", 19});  
edades.insert({"David", 40});           edades.insert({"Martin", 28});  
edades.insert({"Violet", 20});          edades.insert({"George", 48});  
edades.insert({"Helen", 90});           edades.insert({"Manyu", 24});  
edades.insert({"Roland", 14});
```

el resultado en memoria sería similar al que se muestra en la Figura 2.

Una característica interesante es la *tasa de ocupación* de la tabla, que se define como la relación entre el número de pares almacenados y el número de posiciones del vector. En el ejemplo anterior, la tabla contiene  $n = 13$  pares en un vector con  $N = 16$  posiciones, por lo que la tasa de ocupación en el estado actual es

$$\alpha = n/N = 13/16 = 0,8125$$

Es evidente que las tablas abiertas pueden llegar a almacenar más de  $N$  pares (clave, valor), pero debemos tener en cuenta que si la tasa de ocupación crece excesivamente, la velocidad de las consultas se puede degradar hasta llegar a ser lineal.

Por ejemplo, supongamos que en una tabla con un vector de  $N = 16$  posiciones introducimos  $n = 16000$  elementos uniformemente distribuidos en el vector. Para consultar el valor asociado a una clave, tendremos que realizar una búsqueda secuencial en una lista de 1000 elementos, una operación con coste  $O(n/16) = O(n)$ .

### 5.1. Invariante de la representación

Vamos a implementar las tablas abiertas mediante una plantilla parametrizada con los dos tipos de datos involucrados: el tipo de la *clave* y el tipo del *valor*. Tendremos otros dos parámetros opcionales para proporcionar la función hash y el operador == (en caso de no proporcionarlos se usarán las correspondientes operaciones por defecto para el tipo de las claves). De esa forma podremos crear distintos tipos de tablas de la manera habitual:

```
unordered_map<std::string, int> concordancias;
unordered_map<List<char>, Persona> dniPersonas;
unordered_map<List<char>, List<Libro>> librosPrestados;
```

La única limitación es que necesitamos definir una función de localización adecuada para el tipo de datos usado como clave. Más adelante estudiaremos cómo definir estas funciones de localización, por ahora supondremos que existe una función *hash* que realiza la conversión entre claves e índices del vector.

Como ya hemos explicado, para implementar una tabla abierta necesitamos representar un vector de listas de colisión, que vamos a implementar como listas enlazadas donde cada nodo almacena una clave y su valor asociado. La estructura *ListNode*, como no podría ser de otra manera, será interna a la clase *unordered\_map* y almacenará la clave, el valor, y el puntero al siguiente nodo de la lista. La clase *unordered\_map*, a su vez, contiene un vector de punteros a nodos (las listas de colisión), y el número de elementos almacenados en la tabla.

---

```
template <class Clave, class Valor,
          class Hash = std::hash<Clave>,
          class Pred = std::equal_to<Clave>>
class unordered_map {
public:
    // parejas <clave, valor> mediante std::pair
    using clave_valor = std::pair<const Clave, Valor>;

protected:
    using umap_t = unordered_map<Clave, Valor, Hash, Pred>;
    // Alias como shortcut

/*
 * Clase nodo que almacena internamente la pareja <clave, valor>
 * y un puntero al siguiente.
 */
struct ListNode;
using Link = ListNode *;
struct ListNode {
    clave_valor cv;
    Link sig;
    ListNode(clave_valor const& e, Link s = nullptr) : cv(e), sig(s) {}
};

// vector de listas (el tamaño se ajustará a la carga)
std::vector<Link> array;
```

---

```

static const int TAM_INICIAL = 17; // tamaño inicial de la tabla
static const int MAX_CARGA = 75; // máxima ocupación permitida 75%

// número de parejas <clave, valor>
int nelems;

// objeto función para hacer el hash de las claves
Hash hash;

// objeto función para comparar claves
Pred pred;
};

```

---

Pasamos a continuación a definir el *invariante de la representación*, que debe asegurar que la tabla está bien formada. Decimos que una tabla está bien formada si:

- La variable *nelems* contiene el número de elementos almacenados.
- Las listas de colisión son listas enlazadas de nodos bien formadas.
- La lista de colisión asociada al índice *i* del vector sólo contiene pares (*clave, valor*) tales que  $h(\text{clave}) = i$ .
- Ninguna lista de colisión contiene dos pares con la misma clave.

Con estas ideas, podemos formalizar el invariante de la representación de la siguiente manera:

$$R_{\text{Tabla}_{C,V}}(t) \iff \text{ubicado}(t.\text{array}) \wedge t.\text{nelems} = \text{totalElems}(t.\text{array}, t.\text{size}()) \wedge \forall i : 0 \leq i < t.\text{size}() : \text{buenaLista}(t.\text{array}, i)$$

$$\begin{aligned} \text{buenaLista}(v, i) &= R_{\text{Lista}_{Par(C,V)}}(v[i]) \wedge \text{buenaLoc}(v[i], i) \wedge \text{claveUnica}(v[i]) \\ \text{buenaLoc}(l, i) &= \forall j : 0 \leq j < \text{numElems}(l) : h(\text{clave}(l, j)) = i \\ \text{claveUnica}(l) &= \forall j, k : 0 \leq j < k < \text{numElems}(l) : \text{clave}(l, j) \neq \text{clave}(l, k) \end{aligned}$$

El predicado *ubicado* indica que se ha reservado memoria para el array, y *buenaLista* comprueba que cada una de las listas de colisión está bien formada. Consideramos que una lista está bien formada si es una lista enlazada bien formada, sólo contiene elementos cuya clave se asocia a ese índice del vector, y no contiene elementos con claves repetidas.

Para completar la formalización, definimos las funciones *numElems* que devuelve el número de elementos de una lista, *clave* que devuelve la clave del elemento *i*-ésimo de la lista, y *totalElems* que devuelve el número de elementos almacenados en la tabla.

$$\begin{aligned} \text{numElems}(p) &= 0 & \text{si } p = \text{nullptr} \\ \text{numElems}(p) &= 1 + \text{numElems}(p.\text{sig}) & \text{si } p \neq \text{nullptr} \\ \\ \text{clave}(p, i) &= p.\text{cv}.first & \text{si } i = 0 \\ \text{clave}(p, i) &= \text{clave}(p.\text{sig}, i - 1) & \text{si } i > 0 \end{aligned}$$

$$\text{totalElems}(\mathbf{v}, \mathbf{N}) = \sum i : 0 \leq i < N : \text{numElems}(\mathbf{v}[i])$$

Respecto a la relación de equivalencia, al igual que ocurría con el TAD `map`, decimos que dos objetos de tipo `unordered_map` son equivalentes si almacenan el mismo conjunto de pares (clave, valor):

$$\begin{aligned} \mathbf{t1} &\equiv_{\text{unordered\_map}_T} \mathbf{t2} \\ \iff_{\text{def}} & \\ \text{elementos}(\mathbf{t1}) &\equiv_{\text{Conjunto}_{\text{Par}(C,V)}} \text{elementos}(\mathbf{t2}) \end{aligned}$$

donde `elementos` devuelve un conjunto con todos los pares (clave, valor) contenidos en la tabla.

## 5.2. Implementación de las operaciones auxiliares

Al igual que en temas anteriores, comenzamos definiendo algunas operaciones auxiliares que serán de utilidad para implementar las operaciones públicas del TAD. En concreto, vamos a definir métodos privados para liberar la memoria reservada por la tabla, y para buscar nodos en una lista enlazada.

Comenzamos con la operación que libera toda la memoria dinámica reservada para almacenar el vector de listas de colisión.

---

```
void libera() {
    for (int i = 0; i < array.size(); ++i) {
        // liberamos los nodos de la lista array[i]
        Link act = array[i];
        while (act != nullptr) {
            Link a_borrar = act;
            act = act->sig;
            delete a_borrar;
        }
        array[i] = nullptr;
    }
}
```

---

A continuación se muestra un método auxiliar que permite buscar un nodo con una cierta clave en una lista enlazada. A parte de devolver el puntero apuntando al nodo buscado devuelve un puntero al nodo anterior (necesario para la eliminación pues estamos trabajando con listas enlazadas simples). En ambos casos, si buscamos un nodo con una clave que no existe en la lista enlazada, se devolverá `nullptr` como nodo encontrado.

---

```
/*
 * Busca un nodo a partir de "pos" (el primero de la lista
 * correspondiente) que contenga c. Si lo encuentra, "pos" quedará
 * apuntando a dicho nodo y "ant" al nodo anterior. Si no lo
 * encuentra "pos" quedará apuntando a nullptr.
 */
bool localizar(Clave const& c, Link & ant, Link & pos) const {
    ant = nullptr;
    while (pos != nullptr) {
        if (pred(c, pos->cv.first))
            return true;
        else {
```

---

```

        ant = pos; pos = pos->sig;
    }
}
return false;
}

```

---

### 5.3. Implementación de las operaciones públicas

El constructor inicializa el vector con punteros nullptr en todas sus posiciones (representando listas de colisión vacías). El destructor utiliza la operación auxiliar que vimos en el apartado anterior para *liberar* toda la memoria reservada.

```

unordered_map(int n = TAM_INICIAL, Hash h = Hash(), Pred p = Pred()) :
    array(n, nullptr), nelems(0), hash(h), pred(p) {}

~unordered_map() {
    libera();
}

```

---

La siguiente operación que vamos a explicar es la que inserta un nuevo par (clave, valor) en la tabla. Esta operación debe calcular el índice del vector asociado a la clave usando la función de localización, y buscar si la lista enlazada correspondiente ya contiene algún nodo con esa clave. Si ya existía un nodo con esa clave actualiza su valor, y si no, crea un nuevo nodo y lo inserta en la lista. En nuestra implementación insertamos el nuevo nodo por delante, como el primer nodo de la lista.

```

/*
 * Inserta un nuevo par (clave, valor) en la tabla. Si ya existía un
 * elemento con esa clave, actualiza su valor.
 */
bool insert(Clave_valor const& cv) {
    int i = hash(cv.first) % array.size();
    Link ant, pos = array[i];
    if (localizar(cv.first, ant, pos)) { // la clave ya existe
        return false;
    } else {
        array[i] = new ListNode(cv, array[i]);
        ++nelems;
        return true;
    }
}


```

---

La operación de borrado es similar a la anterior. Comenzamos usando la función de localización para calcular el índice del vector asociado a la clave. A continuación buscamos un nodo con esa clave en la lista, y si lo encontramos lo eliminamos. Como puede verse en el código, debemos tener especial cuidado con los punteros cuando el nodo a eliminar es el primero de la lista.

```

/*
 * Elimina el elemento de la tabla con la clave dada. Si no existía ningún
 * elemento con dicha clave, la tabla no se modifica.
 */
bool erase(Clave const& c) {
    int i = hash(c) % array.size();
    Link ant, pos = array[i];


```

---

---

```

if (localizar(c, ant, pos)) {
    if (ant == nullptr)
        array[i] = pos->sig;
    else
        ant->sig = pos->sig;
    delete pos;
    --nelems;
    return true;
} else
    return false;
}

```

---

La operación *count* es muy sencilla de implementar: usamos la función de localización para calcular el índice del vector que podría contener la clave y a continuación realizamos la búsqueda dentro de la lista enlazada de nodos.

---

```

/*
 * Operación observadora que busca la clave c y devuelve
 * un 1 si la encuentra y un 0 si no lo hace.
 */
int count(Clave const& c) const {
    int i = hash(c) % array.size();
    Link ant, pos = array[i];
    return localizar(c, ant, pos) ? 1 : 0;
}


```

---

La operación quizás más importante de las tablas es *at*, que devuelve el valor asociado a una clave. Como es una operación parcial, lanza una excepción si la clave no existe. De nuevo resolvemos la búsqueda en dos fases: primero calculamos el índice del vector que debería contener la clave y a continuación buscamos el nodo en la lista de colisión correspondiente.

---

```

/*
 * Devuelve el valor asociado a la clave dada. Si la tabla no contiene
 * esa clave lanza una excepción.
 */
Valor const& at(Clave const& c) const {
    int i = hash(c) % array.size();
    Link ant, pos = array[i];
    if (localizar(c, ant, pos))
        return pos->cv.second;
    else
        throw std::out_of_range("La clave no se puede consultar");
}


```

---

Como también hicimos en la clase *map*, incluiremos (al igual que se hace en la clase *unordered\_map* de la STL de C++) una versión más avanzada de la operación *at*, que se implementa mediante el operador *[]* y que además de permitir modificar el valor asociado en caso de encontrar la clave buscada, inserta un nuevo par con la clave buscada (y un valor construido por defecto) en el caso de no encontrarla.

---

```

Valor & operator[](Clave const& c) {
    int i = hash(c) % array.size();
    Link ant, pos = array[i];
    if (localizar(c, ant, pos)) {
        return pos->cv.second;
    }
    else

```

---

---

```

    } else {
        array[i] = new ListNode(clave_valor(c,Valor()), array[i]);
        ++nelems;
        return array[i]->cv.second;
    }
}

```

---

Por último, terminamos con las operaciones `empty` y `size` cuyas implementaciones son triviales.

---

```

bool empty() const {
    return nelems == 0;
}

int size() const {
    return nelems;
}

```

---

#### 5.4. Tablas dinámicas

Ya sabemos que si insertamos demasiados elementos en una tabla, el rendimiento de la búsqueda se empieza a degradar. Cuantos más elementos metemos en la tabla más colisiones se producen y, por tanto, las listas de colisiones empiezan a crecer. Si el número de elementos es muy superior al número de posiciones del vector, la mayor parte del tiempo de búsqueda se invierte en buscar la clave dentro de la listas de colisión, lo que puede degradar el coste hasta hacerlo lineal.

Una forma habitual de resolver este problema es permitir que el vector de listas de colisión pueda ampliar su tamaño automáticamente cuando el número de elementos contenidos en la tabla es demasiado grande. Al ampliar el tamaño del vector disminuye la tasa de ocupación (número de elementos / tamaño del vector), lo que favorece mantener el coste de la búsqueda constante.

Necesitaremos modificar la operación `insert` (y también el caso en el que el operador `[]` inserta) para que compruebe si la tabla ya contiene demasiados elementos y por tanto debe expandirse. Para hacerlo, calculamos la tasa de ocupación y, si es demasiado alta, ampliamos el tamaño del vector antes de insertar el nuevo elemento.

---

```

bool insert(clave_valor const& cv) {
    int i = hash(cv.first) % array.size();
    Link ant, pos = array[i];
    if (localizar(cv.first, ant, pos)) { // La clave ya existe
        return false;
    } else {
        if (muy_llena()) {
            amplia();
            i = hash(cv.first) % array.size();
        }
        array[i] = new ListNode(cv, array[i]);
        ++nelems;
        return true;
    }
}

// Operación privada
bool muy_llena() const {

```

---

---

```
    return 100.0 * nelems / array.size() > MAX_CARGA;
}
```

---

Finalmente, falta por implementar la operación auxiliar *amplia* que amplia la capacidad del vector al siguiente número primo<sup>6</sup>. Para implementar correctamente esta operación es importante tener en cuenta que los índices asociados a las claves pueden cambiar, ya que la función de localización depende del tamaño del vector. Eso quiere decir que debemos recalcular el índice asociado a cada nodo y colocarlo en la nueva posición del nuevo vector.

Una forma sencilla de implementar esta operación sería crear una nueva tabla más grande y volver a insertar todos los pares (clave, valor) contenidos en la tabla original. No vamos a utilizar esa estrategia porque implicaría volver a crear todos los nodos en memoria. En su lugar, vamos a “mover” los nodos desde el vector original a la nueva posición que les corresponde en el nuevo vector.

---

```
void amplia() {
    std::vector<Link> nuevo(siguiente_primo(array.size()*2), nullptr);
    for (int j = 0; j < array.size(); ++j) {
        Link act = array[j];
        while (act != nullptr) {
            Link a_mover = act;
            act = act->sig;
            int i = hash(a_mover->cv.first) % nuevo.size();
            a_mover->sig = nuevo[i];
            nuevo[i] = a_mover;
        }
    }
    swap(array, nuevo);
}
```

---

## 5.5. Recorrido usando iteradores

A veces resulta útil poder recuperar todos los pares (clave, valor) almacenados en la tabla. En este apartado vamos a extender el TAD con nuevas operaciones que permitan recorrer los elementos almacenados usando un iterador.

Como siempre, las clases *const\_iterator* e *iterator* serán clases internas con la operación `++` que permite ir hasta el siguiente elemento del recorrido. Como los elementos no se almacenan en la tabla siguiendo ningún orden (de hecho, puede que el tipo de datos usado como clave ni siquiera sea ordenado), podemos recorrerlos de cualquier forma. En este caso hemos elegido recorrer la lista de colisiones de la posición 0 del vector, luego la lista de colisiones de la posición 1, etc. Durante el recorrido debemos tener en cuenta que algunas de estas listas pueden estar vacías, y por tanto puede que tengamos que saltarnos varias posiciones del vector de listas.

Para realizar ese recorrido el iterador necesita tener acceso al vector, y guardar el índice actual dentro del vector, y el puntero al nodo actual dentro de la lista de colisiones actual. A continuación mostramos el código para el caso del iterador no constante (el iterador constante es análogo).

---

```
// iteradores que recorren los pares de la tabla (no ordenados)
class Iterador {
public:
    clave_valor & operator*() const {
```

---

<sup>6</sup>El uso de números primos para los tamaños del vector decrementa la probabilidad de colisiones

```

    if (act == nullptr)
        throw std::out_of_range("No hay elemento a consultar");
    return act->cv;
}

Iterador & operator++() { // ++ prefijo
    next();
    return *this;
}

bool operator==(Iterador const& that) const {
    return act == that.act;
}

bool operator!=(Iterador const& that) const {
    return !(this->operator==(that));
}

protected:
    friend class unordered_map;
    umap_t * tabla; // la tabla que se está recorriendo
    Link act;       // nodo actual
    int ind;        // índice de la lista actual

    // iterador al primer elemento o al último
Iterador(umap_t* t, bool first = true) : tabla(t) {
    if (first) {
        ind = 0;
        while (ind < tabla->array.size() &&
                tabla->array[ind] == nullptr) {
            ++ind;
        }
        act = (ind < tabla->array.size() ? tabla->array[ind]
                                         : nullptr);
    } else {
        act = nullptr;
        ind = tabla->array.size();
    }
}

void next() {
    if (act == nullptr)
        throw std::out_of_range("El iterador no puede avanzar");
    act = act->sig;
    while (act == nullptr && ++ind < tabla->array.size()) {
        act = tabla->array[ind];
    }
}
};

public:
    // iterador que sí permite modificar el elemento apuntado (su valor)
    using iterator = Iterador;

    iterator begin() {
        return iterator(this);

```

---

```

        }

    iterator end() {
        return iterator(this, false);
    }
};
```

---

Por ejemplo, un recorrido para imprimir todos los elementos contenidos en una tabla se podría escribir así:

---

```

unordered_map<string, int> edades;

... // Insertar elementos en la tabla

for (auto it = edades.begin(); it != edades.end(); ++it) {
    cout << "(" << it->first << ", " << it->second << ") \n";
}
```

---

Usando un bucle *range-based-for* podría escribirse de esta otra forma (solo válido de esta forma a partir de la versión C++17):

```

for (auto [nombre, edad] : t)
    cout << "(" << nombre << ", " << edad << ") \n";
```

Por último, como hicimos en el TAD map, incluiremos la operación `find` para buscar una clave devolviendo el correspondiente iterador. De nuevo, implementaremos la búsqueda propiamente dicha en una constructora en la clase interna Iterador, la cual será invocada desde el nuevo método `find`.

---

```

class Iterador {

    ...
    // iterador a una clave
    Iterador(unmap_t* t, Clave const& c) : tabla(t) {
        ind = tabla->hash(c) % tabla->array.size();
        Link ant;
        act = tabla->array[ind];
        if (!tabla->localizar(c, ant, act)) { // iterador al final
            act = nullptr; ind = tabla->array.size();
        }
    }
    ...
}; //fin de la clase Iterador

// En la clase unordered_map
iterator find(Clave const& c) {
    return iterator(this, c);
}
```

---

## 6. Funciones de localización

Una buena función de localización debe ser uniforme y sencilla de calcular. En este apartado vamos a estudiar algunas funciones de localización habituales.

## 6.1. Para enteros

### 6.1.1. Aritmética modular y uso de números primos

El índice asociado a un número es el resto de la división entera entre otro número  $N$  prefijado, preferiblemente primo. Por ejemplo, para  $N = 23$ :

$$\begin{aligned} 1679 \bmod 23 &= 0 \\ 4567 \bmod 23 &= 13 \\ 8471 \bmod 23 &= 7 \\ 0435 \bmod 23 &= 21 \\ 5033 \bmod 23 &= 19 \end{aligned}$$

También es frecuente multiplicar resultados intermedios por un número primo grande antes de combinarlos con los restantes.

### 6.1.2. Mitad del cuadrado

Consiste en elevar al cuadrado la clave y coger las cifras centrales.

$$\begin{aligned} 709^2 &= 502681 \rightarrow 26 \\ 456^2 &= 207936 \rightarrow 79 \\ 105^2 &= 011025 \rightarrow 10 \\ 879^2 &= 772641 \rightarrow 26 \\ 619^2 &= 383161 \rightarrow 31 \end{aligned}$$

### 6.1.3. Truncamiento

Consiste en ignorar parte del número y utilizar los dígitos restantes como índice. Por ejemplo, para números de 7 cifras podríamos coger los dígitos segundo, cuarto y sexto para formar el índice.

$$\begin{aligned} 5700931 &\rightarrow 703 \\ 3498610 &\rightarrow 481 \\ 0056241 &\rightarrow 064 \\ 9134720 &\rightarrow 142 \\ 5174829 &\rightarrow 142 \end{aligned}$$

### 6.1.4. Plegamiento

Consiste en dividir el número en diferentes partes y realizar operaciones aritméticas con ellas, normalmente sumas o multiplicaciones. Por ejemplo, podemos dividir un número en bloques de dos cifras y después sumarlas.

$$\begin{aligned} 570093 &\rightarrow 57 + 00 + 93 = 150 \\ 349861 &\rightarrow 34 + 98 + 61 = 193 \\ 005624 &\rightarrow 00 + 56 + 24 = 80 \\ 913472 &\rightarrow 91 + 34 + 72 = 197 \\ 517492 &\rightarrow 51 + 74 + 92 = 217 \end{aligned}$$

### 6.1.5. Operaciones de bits

También es muy habitual usar los operadores de manipulación de bits para “mezclar” fragmentos de hash. Dos operadores son particularmente frecuentes:  $\wedge$  (xor binario) y  $\ll$  (desplazamiento de bits). Por ejemplo:

$$\begin{aligned} 570093 &\rightarrow (570 \ll 3) \wedge 093 = 4493 \\ 349861 &\rightarrow (349 \ll 3) \wedge 861 = 2485 \\ 005624 &\rightarrow (005 \ll 3) \wedge 624 = 600 \\ 913472 &\rightarrow (913 \ll 3) \wedge 472 = 7504 \\ 517492 &\rightarrow (517 \ll 3) \wedge 492 = 4548 \end{aligned}$$

Desarrollando el último ejemplo,  $1000000101 \ll 3 = 1000000101000$ , que  $\wedge 111101100$  es  $1000111000100$ .

## 6.2. Para cadenas

En este tema ya hemos visto una función de localización para cadenas:

$$h(c) = \text{ord}(\text{ult}(c)) \bmod N$$

El problema de esta función radica en que los códigos ASCII de los caracteres alfanuméricos están comprendidos entre los números 48 y 122, por lo que esta función no se comporta muy bien con valores de  $N$  grandes (tablas grandes).

Una mejora evidente consiste en tener en cuenta todos los caracteres de la cadena, en lugar de sólo el último, por ejemplo sumando sus códigos ASCII:

$$h(c) = (\text{ord}(c[0]) + \text{ord}(c[1]) + \dots + \text{ord}(c[k])) \bmod N$$

Aunque esta función se comporta mejor que la anterior, la suma de los códigos ASCII de los caracteres sigue sin ser un valor muy elevado. Si trabajamos con tablas grandes, esta función tenderá a agrupar todos los datos en la parte inicial de la tabla.

La última función que vamos a plantear tiene en cuenta tanto los caracteres de la cadena como su posición. La idea es interpretar los caracteres de la cadena como dígitos de una cierta base  $B$ :

$$h(c) = (\text{ord}(c[0]) + \text{ord}(c[1]) * B + \text{ord}(c[2]) * B^2 + \dots + \text{ord}(c[k]) * B^k) \bmod N$$

Esta función de localización se comporta bastante mejor que las anteriores con valores de  $N$  grandes. Además, por motivos de eficiencia, todas las operaciones aritméticas se realizan módulo  $2^w$  siendo  $w$  la longitud de la palabra del ordenador. Dicho de otra forma, el ordenador ignora los desbordamientos que producen las operaciones anteriores cuando el número resultante es mayor que el que puede representar de manera natural.

Una buena elección es  $B = 131$  porque para ese valor  $B^i$  tiene un ciclo máximo  $\bmod 2^k$  para  $8 \leq k \leq 64$ .

## 6.3. Para clases definidas por el programador

En los ejemplos vistos hasta ahora, siempre hemos empleado tipos básicos como claves: cadenas, enteros, etc. Sin embargo, la auténtica potencia de las tablas reside en poder

utilizar cualquier clase definida por el programador, siempre que se proporcione una función de localización capaz de transformar concreciones de ese tipo en índices del vector.

En particular, para que una clase pueda usarse como tipo para las claves de una tabla hash es necesario definir dos cosas:

1. El operador de igualdad de la clase: Esto puede hacerse bien sobrecargando el `operator==` (como método de la clase o como función externa), o bien implementando una clase (*clase función*) cuyo `operator()` defina la igualdad y proporcionando esta clase en el cuarto parámetro de la instanciación del objeto tabla.
2. Implementar la función *hash* para la clase, la cual calcula el valor hash para objetos de la clase. Esto debe hacerse implementando una clase función que sobreescriba el `operator()`. De esta forma, delegamos el cálculo de la función de localización sobre el propio tipo de datos, siendo responsabilidad del programador de dicho tipo implementar una buena función de localización.

El siguiente ejemplo muestra como usar una clase `Persona` (definida por el programador) como clave en una tabla hash.

---

```
struct Persona{
    string nombre;
    string dni;
    bool operator==(const Persona& other) const{
        return dni == other.dni;
    }
};

struct PersonaHasher{
    std::size_t operator()(const Persona& p) const {
        return hash<string>()(p.nombre + p.dni);
    }
};

//Instanciación de un objeto
unordered_map<Persona, int, PersonaHasher> edades;
```

---

## 7. En el mundo real...

La mayor parte de los lenguajes de programación modernos incorporan algún tipo de contenedor asociativo implementado mediante tablas de dispersión. De hecho, en algunos lenguajes de *script* los “arrays” permiten utilizar cualquier tipo de dato como índice porque internamente están implementados como tablas de dispersión. En general, un array puede verse como un caso particular de tabla donde las claves son números enteros consecutivos y la función de localización es la identidad.

La librería estándar de C++ incorpora el tipo `unordered_map` análogo al definido aquí (aunque más completo y complejo).

Finalmente, en lenguajes como Java, todas las clases heredan de *Object* que define un método `hashCode()` que devuelve un valor numérico basado en la dirección de memoria del objeto. Los programadores pueden sobrescribir este comportamiento por defecto en sus clases cuando sea necesario, en particular al usarlos como claves en los contenedores `HashMap` y `HashTable`.

## 8. Para terminar...

Terminamos el tema con la solución a la motivación dada al principio del tema. La implementación utiliza un diccionario como variable local que va almacenando, para cada palabra encontrada en el texto, el número de veces que ha aparecido.

---

```
void refsCruzadas(const list<string> &texto) {

    list<string>::const_iterator it = texto.cbegin();
    map<string, int> refs;

    while (it != texto.cend()) {
        map<string, int>::iterator p = refs.find(*it);
        if (p == refs.end())
            refs.insert({*it, 1});
        else
            (*p).second++;
        ++it;
    }

    // Y ahora escribimos
    for (auto [palabra,reps] : refs) // Válido desde C++17
        cout << palabra << " " << reps << endl;
}
```

---

En el código hemos utilizado un map. La implementación con unordered\_map sería igual excepto que la lista de palabras no saldría ordenada.

## Notas bibliográficas

Gran parte del contenido de este capítulo está basado en el capítulo correspondiente de (Rodríguez Artalejo et al., 2011) y de (Peña, 2005). Animamos al lector a consultar ambos libros para profundizar en el estudio de las tablas asociativas, especialmente en el caso de las tablas cerradas que en este capítulo hemos omitido.

## Ejercicios

1. Extiende la implementación de los árboles de búsqueda con la siguiente operación:

```
iterator erase(const iterator &it);
```

que recibe un iterador y elimina la pareja (clave,valor) del diccionario, devolviendo un iterador al siguiente elemento en el recorrido.

2. Implementa una operación en los árboles de búsqueda que *balancee* el árbol. Se permite el uso de estructuras de datos auxiliares.
3. Los árboles de búsqueda y las tablas dispersas son dos tipos de contenedores asociativos que permiten almacenar pares (clave, valor) indexados por clave. Discute sus similitudes y diferencias: ¿qué requisitos impone cada uno sobre el tipo usado como clave?, ¿cuándo es más conveniente usar árboles de búsqueda y cuándo tablas dispersas?
4. Añade las siguientes operaciones a los árboles de búsqueda y analiza su complejidad:

- `consultaK`: recibe un entero  $k$  y devuelve la  $k$ -ésima clave del árbol de búsqueda, considerando que en un árbol con  $n$  elementos,  $k = 0$  corresponde a la menor clave y  $k = n - 1$  a la mayor.
  - `recorreRango`: dadas dos claves,  $a$  y  $b$ , devuelve una lista con los valores asociados a las claves que están en el intervalo  $[a..b]$ .
5. ¿Qué cambios realizarías en la implementación de los árboles de búsqueda para permitir almacenar distintos valores para la misma clave? Es decir:
- Al insertar un par (clave, valor), si la clave ya se encontrase en el árbol, en lugar de sustituir el valor antiguo por el nuevo, se asociaría el valor adicional con la clave.
  - La operación de consulta en vez de devolver un único valor, devuelve una lista con todos ellos en el mismo orden en el que fueron insertados.
  - La operación de borrado elimina todos los valores asociados con la clave dada.
- Para la implementación no debes utilizar otros TADs.
6. En la sección de motivación veíamos que un texto puede venir como una lista de palabras. Otra alternativa es que venga como una lista de *líneas*, donde cada línea es a su vez una lista de palabras (es decir, el tipo sería `list<list<string>>`). El problema de las *referencias cruzadas* consiste en crear el listado en orden alfabético de todas las palabras que aparecen en el texto, indicando, para cada una de las palabras, el número de líneas en las que aparece (si una palabra aparece varias veces en una línea, el número de línea aparecerá repetido). Implementa en C++ un método que reciba un texto y escriba la lista de palabras ordenada alfabéticamente; cada línea contendrá una palabra seguida de todas las líneas en las que ésta aparece. Analiza su complejidad si se utilizan árboles de búsqueda o tablas abiertas.
  7. Implementa un TAD *Conjunto* basado en tablas dispersas con las operaciones habituales: *ConjuntoVacio*, *inserta*, *borra*, *esta*, *union*, *interseccion* y *diferencia*.
  8. Propón una función de localización adecuada para la clase *Conjunto*, de manera que sea posible usar conjuntos como claves de una tabla.
  9. Se define el *índice radial* de una tabla abierta como la longitud del vector por el número de elementos de la lista de colisión más larga. Extiende el TAD Tabla con un método que devuelva su índice radial.
  10. Se llaman *vectores dispersos* a los vectores implementados por medio de tablas dispersas. Esta técnica es recomendable cuando el conjunto total de índices posibles es muy grande, y la gran mayoría de los índices tiene asociado un *valor por defecto* (por ejemplo, cero). Usando esta idea, podemos representar un vector disperso de números reales como una tabla `Tabla<int,float>` que sólo almacena las posiciones del vector que no contienen un 0. Implementa funciones que resuelvan la *suma* y el *producto escalar* de dos vectores dispersos de números reales.
  11. (ACR270) La evaluación continua se le ha ido de las manos al profesor. Les pide a los alumnos que no lo dejen todo para el final sino que vayan estudiando día a día, pero él no predica con el ejemplo. Ahora tiene todos los ejercicios que los alumnos han ido entregando durante todo el año en una pila de folios y le toca revisarlos. Los

ejercicios o están bien (y entonces puntúan positivamente) o están mal (y entonces restan).

Al final del día quiere tener imprimida una lista con los nombres de todos los alumnos ordenados alfabéticamente y su puntuación en la evaluación continua (resultado de sumar todos los ejercicios que tienen bien menos los que tienen mal). Si un alumno tiene un 0 como balance no debería aparecer en la lista.

Aunque sea abusar un poco del alumnado... ¿puedes ayudar al profesor? Lo que se pide es que implementes una función que reciba dos listas de cadenas con los nombres de los alumnos. La primera lista tiene un elemento por cada ejercicio correcto entregado y contiene el nombre del alumno que lo entregó (por lo tanto un mismo alumno puede aparecer varias veces, si entregó varios ejercicios bien). De forma similar, la segunda lista contiene los ejercicios incorrectos. La función debe imprimir por pantalla el resultado final de la evaluación de los alumnos cuyo balance es distinto de 0 por orden alfabético.

12. Plantea una implementación de un TAD *Consultorio* que simule el comportamiento de un consultorio médico simplificado. Dicha implementación hará uso de los TADs *Medico* y *Paciente*, que se suponen ya conocidos. Las operaciones del TAD *Consultorio* son las siguientes:

- *ConsultorioVacio*: crea un nuevo consultorio vacío.
- *nuevoMedico*: da de alta un nuevo médico en el consultorio.
- *pideConsulta*: un paciente se pone a la espera de ser atendido por un médico que ha sido dado de alta previamente en el consultorio.
- *siguientePaciente*: consulta el paciente al que le toca el turno para ser atendido por un médico dado. El médico debe haber sido dado de alta en el consultorio y tener pacientes en espera para que la operación funcione.
- *atiendeConsulta*: elimina el siguiente paciente de un médico. El médico debe estar dado de alta y tener pacientes en la lista de espera.
- *tienePacientes*: indica si un médico tiene o no pacientes esperando a ser atendidos.

Explica razonadamente los tipos abstractos de datos que vas a utilizar.

Para cada operación indica si es generadora, observadora o modificadora, y si es total o parcial. ¿Se necesita exigir algo a los TADs *Medico* y *Paciente*?

Razona la complejidad de las operaciones del TAD en base a la implementación elegida. Para ello considera que hay  $M$  médicos dados de alta en el consultorio, y que el médico con la lista de espera más larga tiene  $P$  pacientes esperando a ser atendidos.



---

## Capítulo 8

# Aplicaciones de tipos abstractos de datos<sup>1</sup>

---

**RESUMEN:** En este tema se estudia la resolución de problemas mediante el uso de distintos tipos de datos. Para ello se proponen varios ejercicios resueltos así como problemas a realizar por el alumno.

## 1. Problemas resueltos:

### 1.1. Confederación hidrográfica.

(Obtenido del examen final de Junio de 2010. Titulación: ITIS. Asignatura: EDI. Grupos A y B)

Una confederación hidrográfica gestiona el agua de ríos y pantanos de una cuenca hidrográfica. Para ello debe conocer los ríos y pantanos de su cuenca, el agua embalsada en la cuenca y en cada pantano. También se permite trasvasar agua entre pantanos del mismo río o de distintos ríos dentro de su cuenca.

Se pide diseñar un TAD que permita a la confederación hidrográfica gestionar la cuenca. En particular, el comportamiento de las operaciones que debe ofrecer debe ser el siguiente:

- `crea`, crea una confederación hidrográfica vacía.
- `an_rio(r)` añade el río `r` a la confederación hidrográfica. Si el río ya existe la operación se ignora. En una confederación no puede haber dos ríos con el mismo nombre.
- `an_pantano(r, p, n1, n2)` crea un pantano de capacidad `n1` en el río `r` de la confederación. Además lo carga con `n2 Hm3` de agua (si `n2 > n1` lo llena). Si ya existe algún pantano de nombre `p` en el río `r` o no existe el río la operación se ignora.
- `embalsar(r, p, n)` carga `n Hm3` de agua en el pantano `p` del río `r` de la confederación. Si no cabe todo el agua el pantano se llena. Si el río o el pantano no están dados de alta en la confederación la operación se ignora.
- `embalsado_pantano(r, p)` devuelve la cantidad de agua embalsada en el pantano `p` del río `r`. Si el pantano no existe o no existe el río devolverá el valor `-1`.

---

<sup>1</sup>Isabel Pita Andreu es la autora principal de este tema.

- `reserva_cuenca(r)` devuelve la cantidad de agua embalsada en la cuenca del río `r`.
- `transvase(r1, p1, r2, p2, n)` transvaza `n Hm3` de agua del pantano `p1` del río `r1` al pantano `p2` del río `r2`, en la confederación. Si `n` es mayor que la cantidad de agua embalsada en `p1` o no cabe en `p2` la operación se ignora.

Todas las operaciones son totales. Se puede suponer definidos los tipos `Rio` y `Pantano`.

### 1.1.1. Solución

#### Representación:

Se propone utilizar una tabla con clave la identificación de los ríos y valor todos los pantanos existentes en el río con su capacidad y litros embalsados. Los pantanos del río se almacenan en otra tabla, con clave la identificación del pantano y valor su capacidad y litros embalsados. Se utilizan tablas porque las operaciones que se van a realizar tanto sobre ríos como sobre pantanos son consultas e inserciones.

La implementación de la clase es la siguiente. El tipo `Rio` representa la identificación de los ríos (para este ejercicio es suficiente con que esta identificación sea el nombre del río) y el tipo `Pantano` representa la identificación de los pantanos (es suficiente con el nombre del pantano).

---

```
using Rio = string;
using Pantano = string;

class ConfHidrografica {
public:
    ConfHidrografica();
    ~ConfHidrografica();
    void an_rio(const Rio& r);
    void an_pantano(const Rio& r, const Pantano& p, int n1, int n2);
    void embalsar(const Rio& r, const Pantano& p, int n);
    int embalsado_pantano(const Rio& r, const Pantano& p) const;
    int reserva_cuenca(const Rio& r) const;
    void transvase(const Rio& r1, const Pantano& p1, const Rio& r2,
                   const Pantano& p2, int n);

private:
    struct InfoPantano {
        int capacidad;
        int litros_embalsados;
    };
    using InfoRio = unordered_map<Pantano, InfoPantano>;
    unordered_map<Rio, InfoRio> rios;
};
```

---

#### Implementación de las operaciones.

##### Operación `ConfHidrografica`, constructor.

El constructor implícito es suficiente para inicializar la tabla de ríos y pantanos, y tiene coste constante, ya que las tablas que se han estudiado, se crean con un tamaño inicial independiente del número de elementos de la tabla.

##### Operación `an_rio`.

Al añadir un río, si éste está en la cuenca no se debe modificar su valor (comportamiento por defecto de la operación *insert*). La tabla correspondiente al valor del río se crea vacía.

---

```
void ConfHidrografica::an_rio(const Rio& r) {
    // Si el río r estaba no se inserta
    rios.insert({r, InfoRio()});
}
```

---

Coste de la operación implementada:

- El coste de crear una tabla vacía es constante.
- El coste de insertar un elemento en la tabla es constante (operación *insert*).

Por lo tanto, el coste de la operación es constante.

#### Operación *an\_pantano*.

La operación que añade un pantano a un río, si el río está en la confederación, y si el pantano no está en el río, lo añade con la capacidad y litros embalsados que se indican. Si los litros embalsados son mayores que la capacidad se embalsa la capacidad total del pantano. Si el río no está en la confederación, o si el pantano ya está en el río la operación no tiene ningún efecto.

---

```
void ConfHidrografica::an_pantano(const Rio& r, const Pantano& p,
                                    int n1, int n2) {
    if (rios.count(r) && !rios[r].count(p)) {
        // crea la informacion del pantano
        InfoPantano i;
        i.capacidad = n1;
        if (n2 < n1) i.litros_embalsados = n2;
        else i.litros_embalsados = n1;
        // añade informacion al río
        rios[r][p] = i;
    }
}
```

---

Coste de la operación implementada:

- El coste de consultar una tabla es constante (operaciones *count* y *[]*).
- El coste de insertar un elemento en una tabla es constante (operación *[]*).
- El resto de operaciones tiene coste constante.

Por lo tanto, el coste de la operación es constante.

Aunque el coste de buscar una clave en una tabla sea constante no se puede considerar despreciable. Es por ello que se debe evitar en la medida de lo posible repetir búsquedas.

El siguiente código hace uso de una variable de tipo referencia (alias) para evitar parte de dichas repeticiones. También se usa *insert* para así insertar el pantano solo si existe (evitando la consulta previa con la llamada al *count*):

---

```
void ConfHidrografica::an_pantano(const Rio& r, const Pantano& p,
                                    int n1, int n2) {
    if (rios.count(r)) {
        InfoRio & infoR = rios[r];
```

---

```

    // crea la informacion del pantano
    InfoPantano infoP;
    infoP.capacidad = n1;
    if (n2 < n1) infoP.litros_embalsados = n2;
    else infoP.litros_embalsados = n1;
    // añade informacion al rio
    infoR.insert({p, infoP});
}
}

```

---

Puesto que el operador [] inserta una clave cuando no está en la tabla y modifica el valor asociado en caso contrario, si lo utilizamos en este caso nos vemos obligados a consultar previamente si el río está en la tabla, lo cual implica que todavía estamos repitiendo búsquedas.

El uso de iteradores con el método *find* permite optimizar el código en cuanto a la no repetición de búsquedas:

```

void ConfHidrografica::an_pantano(const Rio& r, const Pantano& p,
                                    int n1, int n2){
    unordered_map<Rio, InfoRio>::iterator itR = rios.find(r);
    if (itR != rios.end()) {
        // crea la informacion del pantano
        InfoPantano infoP;
        infoP.capacidad = n1;
        if (n2 < n1) infoP.litros_embalsados = n2;
        else infoP.litros_embalsados = n1;
        // añade informacion al rio
        itR->second.insert({p, infoP});
    }
}

```

---

### Operación *embalsar*.

La operación de embalsar añade  $n \text{ Hm}^3$  al agua embalsada en un pantano. Si se supera la capacidad, el pantano se considera lleno.

```

void ConfHidrografica::embalsar(const Rio& r, const Pantano& p, int n) {
    if (rios.count(r) && rios[r].count(p)) {
        // Añade al pantano
        rios[r][p].litros_embalsados += n;
        if (rios[r][p].litros_embalsados > rios[r][p].capacidad)
            rios[r][p].litros_embalsados = rios[r][p].capacidad;
    }
}

```

---

El coste de la operación implementada es constante. La justificación es análoga a la de la operación *an\_pantano*. Sin embargo podemos optimizar mucho el código (en este caso aún más por las numerosas búsquedas que se están realizando, 11 de las cuales son redundantes y se pueden evitar) gracias al método *find*.

```

void ConfHidrografica::embalsar(const Rio& r, const Pantano& p, int n) {
    unordered_map<Rio, InfoRio>::iterator itR = rios.find(r);
    if (itR != rios.end()) {
        InfoRio::iterator itP = itR->second.find(p);
        if (itP != itR->second.end()) {
            // Añade al pantano

```

---

---

```

        itP->second.litros_embalsados += n;
        if (itP->second.litros_embalsados > itP->second.capacidad)
            itP->second.litros_embalsados = itP->second.capacidad;
    }
}

```

---

### Operación *embalsado\_pantano*.

La operación consulta los  $Hm^3$  embalsados en un pantano.

---

```

int ConfHidrografica::embalsado_pantano(const Rio& r,
                                           const Pantano& p) const {
    int n = -1;
    if (rios.count(r) && rios[r].count(p))
        n = rios[r][p].litros_embalsados;
    return n;
}

```

---

El coste de la operación viene dado por el coste de consultar la información de un pantano en un río y por lo tanto es constante. Optimizando los accesos a las tablas usando el método *find* quedaría como sigue:

---

```

int ConfHidrografica::embalsado_pantano(const Rio& r,
                                           const Pantano& p) const {
    int n = -1;
    unordered_map<Rio, InfoRio>::const_iterator itR = rios.find(r);
    if (itR != rios.cend()) {
        InfoRio::const_iterator itP = itR->second.find(p);
        if (itP != itR->second.cend())
            n = itP->second.litros_embalsados;
    }
    return n;
}

```

---

Una versión más concisa y también optimizada usando el método *at* y capturando su posible excepción sería la siguiente.

---

```

int ConfHidrografica::embalsado_pantano(const Rio& r,
                                           const Pantano& p) const {
    try {
        return rios.at(r).at(p).litros_embalsados;
    } catch (std::out_of_range& e) {
        return -1;
    }
}

```

---

### Operación *reserva\_cuenca*.

La operación *reserva\_cuenca* obtiene la suma de los  $Hm^3$  embalsados en todos los pantanos de la cuenca. Para implementarla se utiliza un iterador sobre la tabla que recorre todos los pantanos del río.

---

```

int ConfHidrografica::reserva_cuenca(const Rio& r) const {
    int n = 0;
    unordered_map<Rio, InfoRio>::const_iterator itR = rios.find(r);
    if (itR != rios.cend()) {
        InfoRio::const_iterator itP = itR->second.cbegin();
        while (itP != itR->second.cend()) {

```

---

```

        n += itP->second.litros_embalsados;
        ++it;
    } //while
} //if
return n;
}

```

---

Coste de la operación implementada:

- El coste del método *find* es constante (en promedio).
- El coste de crear un iterador es constante.
- El coste del bucle es lineal respecto al número de elementos de la tabla que almacena los pantanos de un río, ya que las operaciones que se realizan en cada vuelta del bucle tienen coste constante y el número de veces que se ejecuta el bucle coincide con el número de elementos de la tabla.

Por lo tanto el coste de la operación es lineal respecto al número de pantanos en un río. Es importante hacer notar que se podría obtener coste constante en esta operación simplemente guardando y manteniendo un número entero en el tipo *InfoRio* con la suma de los  $Hm^3$  embalsados en todos los pantanos de la cuenca del río. Este dato se tendría que actualizar correspondientemente en las operaciones *embalsar* y *transvase*. No se ha incluido en la implementación para hacer notar al lector este aspecto explícitamente.

### Operación *transvase*.

La operación realiza el transvase de un pantano a otro de  $n Hm^3$  de agua. Si no hay suficiente agua embalsada en el pantano de origen, o si el agua a trasvasar no cabe en el pantano de destino la operación se ignora.

```

void ConfHidrografica::transvase(const Rio& r1, const Pantano& p1,
                                   const Rio& r2, const Pantano& p2,
                                   int n) {
    if (rios.count(r1) && rios.count(r2) &&
        rios[r1].count(p1) && rios[r2].count(p2) &&
        rios[r1][p1].litros_embalsados >= n &&
        rios[r2][p2].litros_embalsados + n <= rios[r2][p2].capacidad) {
            rios[r1][p1].litros_embalsados -= n;
            rios[r2][p2].litros_embalsados += n;
        }
}

```

---

El coste de la operación es constante ya que las operaciones que se realizan son las de consultar e insertar en una tabla (coste constante). La siguiente versión optimiza los accesos a las tablas.

```

void ConfHidrografica::transvase(const Rio& r1, const Pantano& p1,
                                   const Rio& r2, const Pantano& p2, int n) {
    unordered_map<Rio, InfoRio>::iterator itR1 = rios.find(r1);
    unordered_map<Rio, InfoRio>::iterator itR2 = rios.find(r2);
    if (itR1 != rios.end() && itR2 != rios.end()) {
        InfoRio::iterator itP1 = itR1->second.find(p1);
        InfoRio::iterator itP2 = itR2->second.find(p2);
        if (itP1 != itR1->second.end() && itP1 != itR1->second.end()) {
            if (itP1->second.litros_embalsados >= n &&

```

---

```

    itP2->second.litros_embalsados + n <= itP2->second.capacidad) {
    itP1->second.litros_embalsados -= n;
    itP2->second.litros_embalsados += n;
}
}

```

---

## 1.2. Motor de búsqueda.

Se desea crear un motor de búsqueda para documentos de texto, de forma que, tras indexar muchos documentos, sea posible solicitar todos aquellos que contengan todas las palabras de una búsqueda (por ejemplo, “elecciones rector ucm”). Implementa las siguientes operaciones en un TAD *Buscador*:

- *crea*: crea un buscador vacío, sin documentos.
- *indexa(d)*: indexa el documento *d*, que contendría solamente texto, de forma que resulte buscable si la consulta contiene palabras de ese texto; y devuelve un identificador para ese documento, que se usará en los resultados.
- *busca(t)*: devuelve una lista con todos los identificadores de documentos que contienen los términos que aparecen en *t*, una frase.
- *consulta(i)*: muestra el documento al que se le ha asignado el identificador *i*.

Se pide:

- a) Obtener una representación eficiente del tipo utilizando estructuras de datos conocidas.
- b) Implementar todas las operaciones indicando el coste de cada una de ellas. La operación *busca()*, en particular, debe ser lo más eficiente posible, incluso si el número de documentos es muy elevado.

### 1.2.1. Solución.

#### Representación:

Una primera aproximación podría, para cada documento, almacenar en un *unordered\_set* las palabras que aparecen en él. De esta forma podríamos ver si contiene todas las palabras de una consulta con *t* palabras en  $\mathcal{O}(t)$  operaciones. No obstante, el coste de *busca()* se incrementaría linealmente con el número de documentos *d*: como para cada uno habría que ver si contiene o no todas las *t* palabras, el coste total estaría en  $\mathcal{O}(d \cdot t)$ .

Una forma mucho más eficiente de implementar *busca()* es mantener, para cada palabra (también llamada *término*), el conjunto de identificadores de documentos en los que aparece. A esto se le llama un *índice invertido*, ya que en lugar de ir de documentos a palabras, va de palabras a documentos. El coste de *busca()* en este caso sería de  $\mathcal{O}(1)$  para cada una de las *t* palabras, calculando cada vez la intersección del conjunto de documentos devuelto. Como la intersección de dos conjuntos de *d<sub>a</sub>* y *d<sub>b</sub>* documentos se puede calcular en  $\mathcal{O}(\min(d_a, d_b))$ , el coste total de *busca()*, asumiendo que ordenemos los *t* términos de más raros a más frecuentes, está limitado por  $\mathcal{O}(d_{\min} \cdot t)$ , donde *d<sub>min</sub>* es el número de documentos que contienen el término más infrecuente (y que usaremos como conjunto de partida, sobre el que ir calculando las sucesivas intersecciones). La mejora de usar un índice invertido y ordenar los resultados por frecuencia creciente se hace más significativa cuantos más

documentos haya, y cuanto más específicas sean las consultas. Así, si buscamos “elecciones rector ucm”, hay  $10^6$  documentos, y el número de documentos con cada término es de 40, 100 y 2000, respectivamente, podremos calcular la intersección en  $40 + 40 + 40 = 120$  (o menos, según resultados intermedios) consultas en tabla si empezamos por el término más infrecuente (“elecciones”); podríamos necesitar hasta  $2000 + 100 + 40 = 2140$  consultas si lo hacemos todo en orden contrario; y necesitaríamos más de  $10^6$  consultas si no usásemos un índice invertido.

La definición de las clases queda como:

---

```
class MotorBusqueda {
private:
    unordered_map<int, string> docs;
    // Podría ser tb vector<string>, mientras no haya eliminaciones

    unordered_map<string, Conjunto> indice;
    int numDocs; // usado para asignar ids crecientes a los documentos

public:
    int indexa(const string& texto);
    list<int> busca(const string& consulta) const;
    string consulta(int docId) const;
};
```

---

Donde la implementación de *Conjunto*, que representa un conjunto de ids de documentos, es la siguiente:

---

```
class Conjunto {
private:
    unordered_set<int> elems;

public:
    Conjunto(): {}
    Conjunto(const Conjunto& otro): elems(otro.docs) {}

    Conjunto interseccion(const Conjunto& otro) const {
        Conjunto resultado;
        const Conjunto& min = size < otro.size ? *this : otro;
        const Conjunto& max = size >= otro.size ? *this : otro;
        unordered_set<int>::const_iterator it = min.elems.cbegin();
        for ( ; it != min.elems.cend(); ++it) {
            if (max.elems.count(*it)) {
                resultado.elems.insert(*it);
            }
        }
        return resultado;
    }

    void inserta(int elem) {
        elems.insert(elem);
    }

    list<int> comoLista() const {
        list<int> lista;
        unordered_set<int>::const_iterator it = elems.cbegin();
        for ( ; it != elems.cend(); ++it) lista.push_back(*it);
        return lista;
```

---

```

    }

unsigned int size() const {
    return elems.size();
}
};

```

---

Y las de Buscador quedan como sigue:

```

int indexa(const string& texto) {
    int docId = numDocs++;
    docs[docId] = texto; // Si docs fuese vector se haría push_back
    istringstream iss(texto);
    string termino;
    while (iss >> termino) {
        indice[termino].inserta(docId);
    }
    return docId;
}

list<int> busca(const string& consulta) const {
    istringstream iss(consulta);
    string termino;
    bool terminoHuerfano = false;
    list<const Conjunto*> resultados; // * para evitar copias en push_back
    while (iss >> termino && !terminoHuerfano) {
        unordered_map<string, Conjunto*>::const_iterator itT = indice.find(termino);
        if (itT == indice.cend()) terminoHuerfano = true;
        else {
            const Conjunto* c = &(itT->second);
            if (resultados.empty() || c->size() < resultados.front()->size())
                resultados.push_front(c);
            else
                resultados.push_back(c);
        } //else
    } //while
    // el conjunto mas pequeño queda colocado al principio de la lista
    if (terminoHuerfano) return list<int>();
    else {
        Conjunto c = *(resultados.front());
        resultados.pop_front();
        while (!resultados.empty() && c.size() != 0) {
            c = c.interseccion(*(resultados.front()));
            resultados.pop_front();
        }
        return c.comoLista();
    }
}

string consulta(int docId) const {
    return docs.at(docId);
}

```

---

Las complejidades resultantes son  $\mathcal{O}(|\text{texto}|)$  para `indexa()` (hace falta una inserción de  $\mathcal{O}(1)$  para cada palabra del texto a indexar), y  $\mathcal{O}(1)$  para `consulta()`, además del

anteriormente mencionado  $\mathcal{O}(d_{min} \cdot t)$  para *busca()* con  $t$  términos.

### 1.3. Agencia de viajes.

(Obtenido del examen extraordinario Febrero 2010)

Se desea definir un tipo abstracto de datos *Agencia* que representa a una agencia hotelera. Dicho TAD debe ofrecer las siguientes operaciones:

- *crea*: crea una agencia vacía.
- *aloja(c, h)*: modifica el estado de la agencia alojando a un cliente  $c$  en un hotel  $h$ . Si  $c$  ya tenía antes otro alojamiento, éste queda cancelado. Si  $h$  no estaba dado de alta en el sistema, se le dará de alta.
- *desaloja(c)*: modifica el estado de una agencia desalojando a un cliente  $c$  del hotel que éste ocupase. Si  $c$  no tenía alojamiento, el estado de la agencia no se altera.
- *alojamiento(c)*: permite consultar el hotel donde se aloja un cliente  $c$ , siempre que éste tuviera alojamiento. En caso de no tener alojamiento produce un error.
- *listado\_hoteles()*: obtiene una lista ordenada de todos los hoteles que están dados de alta en la agencia.
- *huespedes(h)*: permite obtener el conjunto de clientes que se alojan en un hotel dado. Dicho conjunto será vacío si no hay clientes en el hotel.

Se pide:

- a) Obtener una representación eficiente del tipo utilizando estructuras de datos conocidas.
- b) Implementar todas las operaciones indicando el coste de cada una de ellas. La operación *huespedes* debe producir una lista de clientes en lugar de un conjunto.

#### 1.3.1. Solución.

##### Representación:

Se propone representar la agencia mediante una tabla con clave la identificación de los *clientes* y valor la identificación de los hoteles. Esta tabla permite obtener un coste constante para la operación *alojamiento*.

Sin embargo, la operación *huespedes* exige el recorrido de toda la tabla para obtener los clientes de un hotel dado. Para mejorar el coste de esta operación añadimos a la representación un árbol binario de búsqueda con clave la identificación de los hoteles. El valor asociado será una lista con todos los clientes del hotel. Con esta nueva estructura el coste de la operación *huespedes* es lineal respecto al número de clientes del hotel. Si el listado de los huespedes se quisiese ordenado habría que utilizar un árbol binario de búsqueda para almacenar los clientes en lugar de una lista.

Se selecciona un árbol binario de búsqueda para almacenar la información referente a los hoteles, para obtener la lista ordenada de los hoteles dados de alta en la agencia en tiempo lineal respecto al número de hoteles. Si se utilizase una tabla como ocurre con la información de los clientes, se podría obtener la lista de los hoteles en tiempo lineal, pero después habría que ordenarla con lo que la complejidad de la operación sería del orden de  $\mathcal{O}(n \log n)$

El tipo *Cliente* representa la información de los clientes (para este ejercicio es suficiente con que esta información sea el nombre del cliente) y el tipo *Hotel* representa la información de los hoteles (es suficiente con el nombre del hotel). Así, la definición de la clase queda como se muestra a continuación:

---

```
using Hotel = string;
using Cliente = string;
using InfoHotel = unordered_set<Cliente>

class Agencia{
public:
    Agencia() {};
    void aloja(const Cliente& c, const Hotel& h);
    void desaloja(const Cliente& c);
    const Hotel& alojamiento(const Cliente& c) const;
    list<Hotel> listado_hoteles() const ;
    list<Cliente> huespedes(const Hotel& h) const;
private:
    unordered_map<Cliente, Hotel> clientes;
    map<Hotel, InfoHotel> hoteles;
};
```

---

### Implementación de las operaciones:

El constructor implícito es suficiente para inicializar las tablas de clientes y hoteles.

#### Operación *aloja*

---

```
void Agencia::aloja(const Cliente& c, const Hotel& h) {
    unordered_map<Cliente, Hotel>::iterator it = clientes.find(c);
    if (it != clientes.end()){
        hoteles[it->second].erase(c);
    }
    clientes[c] = h; // Inserta o actualiza si ya estaba
    hoteles[h].insert(c);
}
```

---

El coste de la operación es el siguiente:

- Los costes de las operaciones *find*, *insert* y *erase* sobre el *unordered\_map* son constantes en promedio.
- El coste de la operación [] sobre el *map* es logarítmico en su tamaño (suponiendo un árbol equilibrado).

El coste de la operación es por tanto logarítmico en el número de hoteles dados de alta.

Podemos eso sí optimizar el código en cuanto a evitar búsquedas redundantes gracias al par iterador-booleano que se obtiene como salida de la operación *insert* (en los maps de la STL) informando de si existía o no la clave en la tabla (y por tanto de si se ha insertado), y en caso de existir, el iterador apuntando al par ya existente:

---

```
void Agencia::aloja(const Cliente& c, const Hotel& h) {
    pair<unordered_map<Cliente, Hotel>::iterator, bool> ret =
        clientes.insert({c, h});
    if (!ret.second) {
        // Quitamos c del hotel en el que estaba
```

---

```

        hoteles[ret.first->second].erase(c);
        ret.first->second = h; // Se actualiza el hotel del cliente c
    }
    hoteles[h].insert(c); // Ponemos c como cliente del hotel h
}

```

---

### Operación *desaloja*

```

void Agencia::desaloja(const Cliente& c) {
    unordered_map<Cliente, Hotel>::iterator it = clientes.find(c);
    if (it != clientes.end()) {
        hoteles[it->second].erase(c);
        clientes.erase(c);
    }
}

```

---

El coste de la operación se calcula igual que el coste de la operación *aloja* y sería también logarítmico en el número de hoteles dados de alta.

### Operación *alojamiento*

```

const Hotel& Agencia::alojamiento(const Cliente& c) const {
    try {
        return clientes.at(c);
    } catch (std::out_of_range& e) {
        throw ENoExisteCliente();
    }
}

```

---

El coste de la operación es el coste de consultar un cliente en la tabla (operación *at*). Por lo tanto el coste es constante.

### Operación *listado\_hoteles*

Se recorre el árbol de búsqueda utilizando el iterador, ya que el recorrido definido para éste es en inorden (lo que produce un recorrido ordenado por nombre de hotel).

```

list<Hotel> Agencia::listado_hoteles() const {
    list<Hotel> l;
    for (auto it = hoteles.cbegin(); it != hoteles.cend(); ++it)
        l.push_back(it->first);
    return l;
}

```

---

Sea  $H$  el número de hoteles dados de alta. El coste de la operación  $++$  del iterador del map es logarítmico en  $H$ , y el bucle se ejecuta  $H$  veces. Esto, en principio, nos daría un coste  $\mathcal{O}(H * \log(H))$ . Sin embargo, si analizamos el comportamiento del operador  $++$  a lo largo de todo el recorrido del árbol se puede demostrar que nunca se pasa por un nodo más de dos veces, lo que indica que el coste es en realidad  $\mathcal{O}(H)$ .

### Operación *huespedes*

```

list<Cliente> Agencia::huespedes(const Hotel& h) const {
    list<Cliente> l;
    map<Hotel, InfoHotel>::const_iterator itH = hoteles.find(h);
    if (itH != hoteles.end()) {

```

---

---

```

    for (auto itC = itH->second.cbegin(); itH->second.cend(); ++itC)
        l.push_back(itC->first);
}
return l;
}

```

---

El coste de la operación es el máximo entre el coste de consultar un hotel en el árbol binario de búsqueda de los hoteles y el coste de generar la lista de clientes que se devuelve. Por lo tanto, es el máximo entre el logaritmo del número de hoteles (suponemos el árbol equilibrado) y el máximo número de clientes en un hotel.

#### 1.4. E-reader.

(Obtenido del examen final Junio 2011. Titulación: II. Asignatura: EDI. Grupos A y C)

Se desea diseñar una aplicación para gestionar los libros guardados en un e-reader. Suponemos que contamos con un TAD *libro* que representa la clave única para identificar un libro.

El comportamiento de las operaciones es el siguiente:

1. *crear*: crea un e-reader sin ningún libro.
2. *poner\_libro(x,n)*: Añade un libro *x* al e-reader. *n* representa el número de páginas del libro, puede ser cualquier número positivo. Si el libro ya existe la acción no tiene efecto.
3. *abrir(x)*: El usuario abre un libro *x* para leerlo. Si el libro *x* no está en el e-reader se produce un error. Si el libro ya había sido abierto anteriormente se considerará este libro como el último libro abierto.
4. *avanzar\_pag()*: Pasa una página del último libro que se ha abierto. La página posterior a la última es la primera. Si no existe ningún libro abierto se produce un error.
5. *abierto()*: Devuelve el último libro que se ha abierto. Si no se encuentra ningún libro abierto se produce un error.
6. *pag\_libro(x)*: devuelve la página, del libro *x*, en la que se quedó leyendo el usuario. Se considera que todos los libros empiezan en la página 1, y ese será el resultado en caso de no haberse abierto nunca el libro. Si el libro no está dado de alta se produce un error.
7. *elim\_libro(x)*: Elimina el libro *x* del e-reader. Si el libro no existe la acción no tiene efecto. Si el libro es el último abierto se elimina y queda como último abierto el que se abrió con anterioridad.
8. *esta\_libro(x)*: Consulta si el libro *x* está en el e-reader.
9. *recientes(n)*: Obtiene una lista con los *n* últimos libros que fueron abiertos, ordenada según el orden en que se abrieron los libros, del más reciente al más antiguo. Si el número de libros que fueron abiertos es menor que el solicitado, la lista contendrá todos ellos. Si un libro se ha abierto varias veces solo aparecerá en la posición más reciente.

10. *num\_libros()*: Consulta el número de libros que existen en el e-reader.

Se pide:

- Obtener una representación eficiente del tipo utilizando estructuras de datos conocidas. No se permite utilizar vectores ni memoria dinámica (listas enlazadas). Implementar todas las operaciones indicando el coste de cada una de ellas. El tipo de retorno de la operación *recientes* debe ser un tipo lineal, seleccionar uno adecuado y justificarlo.
- Modificar la representación anterior utilizando memoria dinámica (listas enlazadas) de forma que el coste de la operación *abrir* sea constante y el coste de *recientes* sea lineal respecto al parámetro de entrada (número de libros que se quieren obtener). El coste de las demás operaciones no debe ser mayor que con la representación del ejercicio anterior, ni debe aumentar de forma significativa el gasto en memoria. Implementar todas las operaciones indicando su coste.

#### 1.4.1. Solución.

##### Primera representación:

Se propone representar el e-reader mediante una tabla con clave la identificación de los libros y valor la información referente al total de páginas del libro, la página en que está abierto y una variable booleana que indique si está abierto.

Para poder implementar la operación *recientes* se añade a la representación una lista con los libros que se han abierto en el orden en que se abren. Si un libro ya abierto se vuelve a abrir se coloca en primer lugar de esta lista.

Para conseguir coste constante en la operación *num\_libros* se añade una variable entera, *cantidad*, con el número de libros del e-reader.

La definición de la clase es la siguiente:

---

```
using Libro = string;

class EReader {
    public:
        EReader();
        void poner_libro(const Libro& x, int n);
        void abrir(const Libro& x);
        void avanzar_pag();
        const Libro& abierto() const;
        int pag_libro(const Libro& x) const;
        void elim_libro(const Libro& x);
        bool esta_libro(const Libro& x) const;
        list<Libro> recientes(int n) const;
        int num_libros() const;
    private:
        struct InfoLibro {
            int totalPags;
            int pagActual;
            bool abierto;
        };
        unordered_map<Libro, InfoLibro> libros;
        list<Libro> secAbiertos;
        int cantidad;
};
```

---

### Implementación de las operaciones:

#### Operación *e-reader*, constructora

Solo es necesario inicializar la variable entera.

---

```
EReader::EReader() : cantidad(0) { }
```

---

El coste de la operación es constante.

#### Operación *poner\_libro*.

---

```
void EReader::poner_libro(const Libro& x, int n) {
    // si el libro ya esta o el numero de paginas
    // es negativo no se hace nada
    if (n > 0) {
        // Se crea la informacion del libro
        InfoLibro i = {n, 1, false};
        libros.insert({x, i}); // Se inserta en la tabla
        // Se incrementa el numero de libros del e-reader
        cantidad++;
    }
}
```

---

El coste de la operación es el siguiente:

- El coste de realizar asignaciones es constante.
- El coste de insertar en una tabla es constante (operación *insert*).

El coste de la operación es la suma de los costes de las instrucciones, por lo tanto es constante.

#### Operación *abrir*.

---

```
void EReader::abrir(const Libro& x) {
    if (!libros.count(x)) throw ENoExiste();
    else { // El libro esta en la tabla, consulta su informacion
        InfoLibro& infoL = libros[x];
        if (infoL.abierto) { // Si esta abierto lo borra de su posicion
            list<Libro>::iterator it = secAbiertos.begin();
            while ((*it) != x) ++it;
            secAbiertos.erase(it);
        }
        else { // Si no esta abierto cambia su estado a abierto
            infoL.abierto = true;
        }
        // inserta el libro al comienzo de la secuencia
        secAbiertos.push_front(x);
    }
}
```

---

El coste de la operación es el siguiente:

- El coste de consultar un libro en la tabla es constante (operaciones *count* y *[]*). Se podría evitar usando *find* y el iterador en lugar de *count* y *[]*.

- El coste del bucle que recorre la lista de libros abiertos es, en el caso peor, lineal respecto a los libros abiertos del e-reader, que pueden ser todos los libros del e-reader.
- El coste de eliminar un elemento de la lista a través del iterador es constante (operación *erase*).
- El coste de añadir un elemento al principio de una lista es constante (operación *push\_front*).

Por lo tanto, el coste de la operación es lineal respecto al número de libros del e-reader.

### Operación *avanzar\_pag*.

---

```
void EReader::avanzar_pag() {
    // Si no hay ningún libro abierto produce error
    if (secAbiertos.empty()) throw ENoabiertos();
    else { // Si hay libros abiertos. Obtiene el primero
        Libro l = secAbiertos.front();
        // incrementa la pagina en la informacion de la tabla
        InfoLibro& infoL = libros[l];
        infoL.pagActual++;
        // Si la pagina es mayor que la ultima vuelve a la primera
        if (infoL.pagActual > infoL.totalPags)
            infoL.pagActual = 1;
    }
}
```

---

El coste de la operación es constante ya que:

- El coste de consultar si una lista es vacía y el primero de una lista es constante (operaciones *empty* y *front*).
- El coste de consultar y modificar el valor en una tabla es constante (operación *[]*).

### Operación *abierto*.

---

```
const Libro& EReader::abierto() const{
    if (secAbiertos.empty()) throw ENoabiertos();
    else return secAbiertos.front();
}
```

---

El coste de la operación es constante, ya que el coste de consultar si una lista es vacía y el primero de una lista son constantes.

### Operación *pag\_libro*.

---

```
int EReader::pag_libro(const Libro& x) const{
    if (!libros.count(x)) throw ENoExiste();
    else {
        return libros[x].pagActual;
    }
}
```

---

El coste de la operación es constante, ya que el coste de consultar en una tabla (operaciones *count* y *at*) es constante. De nuevo se podría optimizar usando *find*.

### Operación *elim\_libro*.

---

```
void EReader::elim_libro(const Libro& x) {
    auto itL = libros.find(x);
    if (itL == libros.end()) throw ENoExiste();
    else {
        InfoLibro infoL = it->second;
        libros.erase(x);
        // En STL se puede hacer más eficiente con libros.erase(itL)
        cantidad--;
        // Si el libro esta abierto lo borra de la lista
        if (infoL.abierto) {
            list<Libro>::iterator it = secAbiertos.begin();
            while ((*it) != x)
                ++it;
            secAbiertos.erase(it);
        }
    }
}
```

---

El coste de la operación es el siguiente:

- El coste de consultar y borrar en una tabla es constante (operaciones *find* y *erase*).
- El coste de recorrer la lista para eliminar el libro si está abierto es, en el caso peor, lineal respecto al número de libros del e-reader.
- El coste de borrar el elemento indicado por el iterador es constante (operación *erase*).

El coste de la operación, por lo tanto, es lineal respecto al número de libros del e-reader.

#### Operación *esta\_libro*.

---

```
bool EReader::esta_libro(const Libro& x) const{
    return libros.count(x);
}
```

---

El coste de consultar si un libro está en el e-reader es constante, ya que el coste de consultar en una tabla es constante.

#### Operación *recientes*.

---

```
list<Libro> recientes(int n) const{
    list<Libro> l;
    int cont = 0;
    list<Libro>::const_iterator it = secAbiertos.cbegin();
    while (it != secAbiertos.cend() && cont < n) {
        l.push_back(*it);
        ++it;
        cont++;
    }
    return l;
}
```

---

El coste de la operación es lineal respecto al valor del parámetro de entrada, ya que este es el número de veces que como máximo se ejecuta el bucle y el coste de todas las operaciones que se realizan en el bucle es constante: consultar el final de un iterador *end*,

añadir un elemento al final de una lista *push\_back*, consultar el elemento apuntado por un iterador, y avanzar un iterador.

#### Operación *num\_libros*.

---

```
int EReader::num_libros() const {
    return cantidad;
}
```

---

El coste de la operación es constante, ya que el coste de devolver un valor de tipo entero es constante.

#### Segunda representación (alternativa a la primera):

Se propone a continuación otra implementación del e-reader en que se mejora el coste de algunas operaciones a costa de empeorar el coste de otras. Dependiendo del uso que se vaya a hacer del e-reader será más apropiada una implementación o la otra.

Se define una tabla con clave la información del libro y valor el número de páginas, la página por la que se va leyendo, y un contador que indica el orden de apertura de los diversos libros. Nótese que se ha cambiado la variable booleana *abierto* de la representación anterior por este contador. De esta forma no solo tenemos información de si el libro ha sido abierto sino también del orden en que fueron abiertos.

La secuencia de libros abiertos se sustituye por una variable que almacena el último libro abierto. Se añade un contador de los libros que se van abriendo, que sirve para actualizar el orden en que se abre un libro dentro de la información de los libros en la tabla. Por último se mantiene la variable que almacena la cantidad de libros del e-reader.

---

```
class EReader {
    public:
        // mismas operaciones que en el caso anterior
        ...
    private:
        struct InfoLibro {
            int totalPags;
            int pagActual;
            int numAbierto; // 0 indica que no se ha abierto todavía
        };
        unordered_map<Libro, InfoLibro> libros;
        Libro ultimoAbierto;
        int acumulador; // Para contar orden de apertura
        int cantidad; // numero total de libros del e-reader
};
```

---

**Implementación de las operaciones:** Se muestran sólo las operaciones que se modifican respecto a la representación anterior.

#### Operación *e-reader, constructora*

En este caso es necesario inicializar también el acumulador. Se elige el valor uno como inicialización y se incrementará su valor después de utilizarlo.

---

```
EReader::EReader(): cantidad(0), acumulador(1) { }
```

---

El coste es constante.

#### Operación *poner\_libro*

La única modificación es la inicialización de la variable numAbierto a cero, en lugar de la variable booleana existente en la otra representación.

---

```
void EReader::poner_libro(const Libro& x, int n) {
    // si el libro ya esta o el numero de paginas es negativo
    // no se hace nada
    if (n > 0) {
        // Se crea la informacion del libro
        InfoLibro i = {n, 1, 0};
        libros.insert({x, i}); // Se inserta en la tabla
        // Se incrementa el numero de libros del e-reader
        cantidad++;
    }
}
```

---

La operación consulta y modifica elementos en una tabla, además de hacer algunas asignaciones, sumas y comparaciones. Por lo tanto el coste de la operación es constante.

### Operación *abrir*

En este caso se modifica el valor de la variable ultimoAbierto con el libro que se está abriendo en esta operación. Se modifica también el orden en que se abrió el libro usando el valor del acumulador. No es necesario diferenciar el caso en que el libro ya había sido abierto anteriormente. Se incrementa el valor del acumulador para utilizarlo cuando se abra otro libro.

---

```
void EReader::abrir(const Libro& x) {
    if (!libros.count(x)) throw ENoExiste();
    else { // El libro esta en la tabla, consulta su informacion
        InfoLibro& infoL = libros[x];
        // Pone el libro como ultimo abierto y actualiza su contador
        ultimoAbierto = x;
        infoL.numAbierto = acumulador;
        acumulador++;
    }
}
```

---

La operación consulta y modifica información en una tabla, y modifica el valor de algunas variables. Su coste es constante.

### Operación *avanzar\_pag*

Se comprueba que hay algún libro abierto utilizando el valor del acumulador. En este caso no podemos comprobar que la secuencia de libros abiertos es vacía como se hacía en la representación anterior.

El último libro abierto se obtiene directamente de la variable ultimoAbierto, en lugar de buscar el primero de la secuencia.

---

```
void EReader::avanzar_pag() {
    // Si no hay ningun libro abierto produce error
    if (acumulador == 1) throw ENoabiertos();
    else {
        // Si hay libros abiertos. Obtiene el primero
        Libro l = ultimoAbierto;
        // incrementa la pagina en la informacion de la tabla
        InfoLibro& infoL = libros[l];
        infoL.pagActual++;
```

---

```
// Si la pagina es mayor que la ultima vuelve a la primera
    if (infoL.pagActual > infoL.totalPags)
        infoL.pagActual = 1;
}
}
```

---

La operación consulta y modifica información en una tabla. Su coste es constante.

### Operación *abierto*

Se obtiene el valor directamente de la variable `ultimoAbierto`.

```
const Libro& EReader::abierto() const{
    if (acumulador == 1) throw ENoabiertos();
    else return ultimoAbierto;
}
```

---

El coste de la operación es constante.

### Operación *elim\_libro*

Si el libro no era el último abierto simplemente lo borraremos de la tabla, en otro caso hay que dar valor a la variable `ultimoAbierto`. Para ello se recorre la tabla buscando el libro que tenga un valor mayor en su variable `numAbierto` que indica en qué posición fue abierto. Si el único libro que se había abierto era el libro que se está eliminando, el acumulador se pone a uno para indicar que no queda ningún libro abierto.

```
void EReader::elim_libro(const Libro& x) {
    auto itL = libros.find(x);
    if (itL == libros.end()) throw ENoExiste();
    else {
        libros.erase(x);
        cantidad--;
        // Si el libro es el ultimo abierto debe buscarse el anterior
        if (ultimoAbierto == x) {
            unordered_map<Libro, InfoLibro>::const_iterator it = libros.cbegin();
            int aux = 0;
            Libro libro_aux;
            while (it != libros.cend()) {
                if (it->second.numAbierto > aux) {
                    aux = it->second.numAbierto;
                    libro_aux = it->first;
                } // No se puede buscar el valor del acumulador
                // directamente porque se puede haber eliminado el libro
                ++it;
            }
            if (aux != 0) {
                ultimoAbierto = libro_aux;
                acumulador = aux + 1;
            }
            else acumulador = 1;
        }
    }
}
```

---

La operación realiza varias operaciones sobre la tabla de libros, todas ellas de coste constante. Se declara un iterador que recorre toda la tabla de libros. En cada vuelta del

bucle las operaciones que se realizan son de acceso a los valores del iterador, por lo tanto el coste del cuerpo del bucle es constante. El coste de la operación es, por lo tanto, lineal respecto al número de libros del e-reader.

### Operación *recientes*

Para obtener los  $n$  libros que se han abierto más recientemente hay que buscarlos en la tabla. Para ello se crea un árbol binario de búsqueda en el que se van añadiendo todos los libros de la tabla que han sido abiertos en algún momento. La clave es el orden en que fueron abiertos y el valor la información del libro.

De esta forma, al recorrer el árbol en inorden añadiendo los elementos al principio de la lista resultante en lugar de al final, obtenemos los libros ordenados por el momento en que fueron abiertos de mayor a menor. A este algoritmo de ordenación que consiste en insertar los elementos a ordenar en un árbol y después recorrerlo en inorden se le llama *treesort*.

---

```
list<Libro> recientes(int n) const{
    list<Libro> l;
    map<int,Libro> aux; // clave el orden de apertura
    Hashmap<Libro>::const_iterator it = libros.cbegin();
    while (it != libros.cend()) {
        if (it->second.numAbierto != 0) // libro abierto
            aux.insert({it->second.numAbierto, it->first});
        ++it;
    }
    int cont = 0;
    map<int,Libro>::const_iterator ita = aux.cbegin();
    while (ita != aux.cend() && cont < n) {
        l.push_front(ita->second);
        ++ita;
        cont++;
    }
    return l;
}
```

---

La operación declara un iterador sobre la tabla de libros y la recorre entera. Cada elemento de la tabla se inserta en un árbol binario de búsqueda. Suponiendo el árbol equilibrado, el coste de la inserción es logarítmico respecto al número de libros abiertos del e-reader. Por último se recorre el árbol de búsqueda insertando cada elemento en una lista. Este recorrido tiene coste lineal respecto al número de libros abiertos, que es el número de nodos del árbol, ya que el coste de insertar por el principio de una lista es constante. El coste de la operación, al estar los dos recorridos en secuencia, es el máximo de los dos (el del primer bucle), esto es:  $\mathcal{O}(n_1 \log n_2)$  siendo  $n_1$  el número de libros del e-reader y  $n_2$  el número de libros abiertos del e-reader. En el caso peor en que todos los libros hayan sido abiertos el coste es  $\mathcal{O}(n_1 \log n_1)$ .

### Tercera representación (apartado b. del enunciado):

Esta representación mejora los costes de la primera. Para mejorar la eficiencia de la operación *abrir* hay que evitar recorrer la lista de libros abiertos. Para ello guardamos junto a la información de cada libro un nuevo campo con un iterador a su posición en la lista de libros abiertos. Así, al abrir un libro, en lugar de recorrer la lista para cambiar su posición, lo borramos en tiempo constante mediante el operador *erase* de la clase *list*. La operación de eliminar un libro tendría también coste constante (también se borraría el libro de la lista de abiertos mediante el método *erase* usando el iterador).

La declaración de la clase sería la siguiente:

---

```
class EReader {
    public:
        EReader();
        void poner_libro(const Libro& x, int n);
        void abrir(const Libro& x);
        void avanzar_pag();
        const Libro& abierto() const;
        int pag_libro(const Libro& x) const;
        void elim_libro(const Libro& x);
        bool esta_libro(const Libro& x) const;
        list<Libro> recientes(int n) const;
        int num_libros() const;
    private:
        struct InfoLibro {
            int totalPags;
            int pagActual;
            list<Libro>::iterator posSecAbiertos;
        };
        unordered_map<Libro, InfoLibro> libros;
        list<Libro> secAbiertos;
        int cantidad;
};
```

---

### Operación *poner\_libro*.

---

```
void EReader::poner_libro(const Libro& x, int n) {
    // si el libro ya esta o el numero de paginas es
    // negativo no se hace nada
    if (n > 0) {
        // Se crea la informacion del libro
        InfoLibro infoL = {n, 1, secAbiertos.end()};
        libros.insert({x, infoL}); // Se inserta en la tabla
        // Se incrementa el numero de libros del e-reader
        cantidad++;
    }
}
```

---

El iterador a la lista de abiertos se inicializa a `secAbiertos.end()` para indicar que el libro no se encuentra en la lista. El coste de la operación es constante, no cambia respecto a la anterior representación.

### Operación *abrir*.

---

```
void EReader::abrir(const Libro& x) {
    auto itL = libros.find(x);
    if (itL == libros.end()) throw ENoExiste();
    else { // El libro esta en la tabla. Consulta su informacion
        InfoLibro& infoL = itL->second;
        if (infoL.posSecAbiertos != secAbiertos.end()) {
            // Si esta abierto lo elimina de su posicion
            secAbiertos.erase(infoL.posSecAbiertos);
        }
        // Añade el libro al principio de la lista y guarda el iterador
        infoL.posSecAbiertos = secAbiertos.insert(secAbiertos.begin(), x);
    }
}
```

---

---

}

---

El recorrido de la lista de libros abiertos se ha sustituido por la llamada al método `erase` cuyo coste es constante. El coste de la operación es por tanto constante.

### Operación `elim_libro`.

---

```
void EReader::elim_libro(const Libro& x) {
    auto itL = libros.find(x);
    if (itL == libros.end()) throw ENoExiste();
    else {
        InfoLibro infoL = itL->second;
        // Si el libro esta abierto lo elimina de la lista
        if (infoL.posSecAbiertos != secAbiertos.end()) {
            secAbiertos.erase(infoL.posSecAbiertos);
        }
        libros.erase(x); // Mejor libros.erase(itL); (solo versión STL)
        cantidad--;
    }
}
```

---

De nuevo la búsqueda para encontrar el libro en la lista ya no es necesaria gracias al iterador. El coste es por tanto constante.

La implementación del resto de operaciones es la misma que en la primera representación.

### Comparación de los costes obtenidos con ambas representaciones

|                    | Primera repr.    | Segunda repr.           | Tercera repr.    |
|--------------------|------------------|-------------------------|------------------|
| Constructora       | $\mathcal{O}(1)$ | $\mathcal{O}(1)$        | $\mathcal{O}(1)$ |
| <i>poner_libro</i> | $\mathcal{O}(1)$ | $\mathcal{O}(1)$        | $\mathcal{O}(1)$ |
| <i>abrir</i>       | $\mathcal{O}(n)$ | $\mathcal{O}(1)$        | $\mathcal{O}(1)$ |
| <i>avanzar_pag</i> | $\mathcal{O}(1)$ | $\mathcal{O}(1)$        | $\mathcal{O}(1)$ |
| <i>abierto</i>     | $\mathcal{O}(1)$ | $\mathcal{O}(1)$        | $\mathcal{O}(1)$ |
| <i>pag_libro</i>   | $\mathcal{O}(1)$ | $\mathcal{O}(1)$        | $\mathcal{O}(1)$ |
| <i>elim_libro</i>  | $\mathcal{O}(n)$ | $\mathcal{O}(n)$        | $\mathcal{O}(1)$ |
| <i>esta_libro</i>  | $\mathcal{O}(1)$ | $\mathcal{O}(1)$        | $\mathcal{O}(1)$ |
| <i>recientes</i>   | $\mathcal{O}(n)$ | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n)$ |
| <i>num_libros</i>  | $\mathcal{O}(1)$ | $\mathcal{O}(1)$        | $\mathcal{O}(1)$ |

Donde  $n$  representa el número de libros del e-reader.

## 2. Problemas propuestos:

1. Librería. (Obtenido del examen final Septiembre 2011. Titulación: II. Asignatura: EDI. Grupos A y C)

Se desea diseñar una aplicación para gestionar un sistema de venta de libros por Internet. El TAD será paramétrico respecto a la información asociada a un libro.

El comportamiento de las operaciones es el siguiente:

- *crear*: crea un sistema sin ningún libro.

- *an-libro(x,n)*: Añade  $n$  ejemplares de un libro  $x$  al sistema. Si  $n$  toma el valor cero significa que el libro está en el sistema, aunque no se tienen ejemplares disponibles. Se pueden añadir mas ejemplares de un libro que ya esté en el sistema.
- *comprar(x)*: Un usuario compra un libro  $x$ . Si no existen ejemplares disponibles del libro  $x$  se produce un error.
- *esta-libro(x)*: Indica si un libro  $x$  se ha añadido al sistema. El resultado será cierto aunque no haya ejemplares disponibles del libro.
- *elim-libro(x)*: Elimina el libro  $x$  del sistema. Si el libro no existe la operación no tiene efecto.
- *num-ejemplares(x)*: Devuelve el número de ejemplares de un libro  $x$  que hay disponibles en el sistema. Si el libro no está dado de alta se produce un error.
- *top10()*: Obtiene una lista con los 10 libros que más se han vendido. Si hay mas de 10 libros distintos con un máximo número de ventas la aplicación obtiene 10 de ellos, sin que se especifique cuales. Si no se han vendido 10 libros distintos se listarán todos ellos.
- *num-libros()*: Obtiene el número de libros distintos que existen en el sistema.

Se pide:

- a) Obtener una representación eficiente del tipo. No se permite utilizar directamente vectores ni memoria dinámica (listas enlazadas con nodos y punteros). Implementar todas las operaciones indicando el coste de cada una de ellas. El tipo de retorno de la operación *top10* debe ser un tipo lineal, seleccionar uno adecuado y justificarlo. Las operaciones de consulta sobre los TAD devuelven una copia de la estructura.
  - b) Generalizar la operación *top10* anterior con una operación *topN(n)* que obtenga una lista con los  $n$  libros que más se han vendido, ordenada según el número de ejemplares vendidos, de los más vendidos a los menos vendidos. Si se ha vendido el mismo número de ejemplares de varios libros, se mostrará primero el que se haya vendido más recientemente. Si el número de libros vendidos es menor que el solicitado, la lista contendrá todos ellos.
2. Restaurante. (Obtenido del examen final Septiembre 2006. Titulación: II. Asignatura: EDI. Grupos A, B y C)

El conocido restaurante Salmonelis necesita una base de datos para gestionar mejor sus afamados platos. Cada plato dispondrá de un código único de tipo plato y tendrá asociado un código de tipo y un precio. Ejemplos de códigos de tipo son: entrante, carne, pescado, sopa, etc.

Las operaciones que debe ofrecer el TAD son las siguientes:

- *crear*, crea una base de datos vacía
- *anadir(p,t,n)*, añade un plato  $p$  con su tipo  $t$  y precio  $n$ . Produce error si el plato ya está en la base de datos.
- *modif-tipo(p,t)*, modifica el tipo de un plato  $p$ . Produce error si el plato no está en la base de datos.

- $modif-precio(p,n)$ , modifica el precio de un plato  $p$ . Produce error si el plato no está en la base de datos.
- $tipo(p)$ , devuelve el tipo de un plato  $p$ . Produce error si el plato no está en la base de datos.
- $precio(p)$ , devuelve el precio de un plato  $p$ . Produce error si el plato no está en la base de datos.
- $platos-tipo(t)$ , devuelve una lista de platos, de un tipo  $t$  con sus precios, ordenada por precios crecientes. Si no existe ningún plato del tipo pedido devuelve la lista vacía.

Se pide obtener una representación eficiente del tipo utilizando tipos conocidos. Implementar todas las operaciones indicando el coste de cada una de ellas.

3. Clínica. (Obtenido del examen final Septiembre 2004. Titulación: II. Asignatura: EDI.)

Tras evaluar su funcionamiento durante el último año, la dirección de la Clínica *Casi Me Muero* ha decidido renovar el sistema informático de su consultorio médico para realizar (al menos) las siguientes operaciones:

- $crear$ , genera un consultorio vacío, sin ninguna información,
- $alta-médico(m)$ , da de alta a un nuevo médico  $m$  que antes no figuraba en el consultorio,
- $pedir-vez(p,m)$  hace que un paciente  $p$  pida la vez para ser atendido por un médico  $m$ ,
- $atender-consulta(m)$ , atiende al paciente que le toque en la consulta de un médico  $m$ ,
- $cancelar-cita(m)$  permite que el último paciente en la consulta de un médico  $m$ , debido al cansancio de la espera, cancele la cita,
- $pedir-vez-enchufe(p,m)$  hace que un paciente  $p$ , haciendo uso de algún *enchufe* que aquí no nos interesa, pida la vez para colocarse el primero en la consulta de un médico,
- $baja-médico(m)$  da de baja a un médico  $m$ , borrando todas sus citas,
- $num-citas(p)$ , devuelve el número de citas que un mismo paciente tiene en todo el consultorio.

Se pide obtener una representación eficiente del tipo utilizando tipos conocidos. Implementar todas las operaciones indicando el coste de cada una de ellas.

4. Campeonato de atletismo. (Obtenido del examen final de Junio de 2009. Titulación: ITIS. Asignatura: EDI. Grupos A y B)

El TAD *Campeonato* se utiliza para manejar la información relativa a unas pruebas de atletismo.

El comportamiento de las operaciones que debe ofrecer el TAD es el siguiente:

- $crea$ . Constructora que crea un TAD vacío.
- $an\_prueba(p)$ . Añade una prueba  $p$  al campeonato. Si la prueba ya está en el TAD éste no se modifica.

- *an\_atleta(a,p)*. Añade un atleta a una prueba del campeonato. Si no está la prueba en el campeonato la operación produce error. El orden de los atletas en una prueba es importante.
- *obtener-sig-atleta(p)*. Obtiene el siguiente atleta a participar en una prueba. El orden viene dado por el orden en que se incorporaron al sistema. Produce un error si la prueba no está dada de alta o no hay ningún atleta en la prueba.

Se pide obtener una representación para el TAD e implementar todas las operaciones. Para cada operación calcular el coste de la implementación realizada.

### 5. MultaMatic (EDA - Junio 2013)

Desarrolla MultaMatic, el nuevo sistema de gestión de multas de tráfico por exceso de velocidad. La red de carreteras contiene tramos vigilados en los que se coloca una cámara al principio del tramo y otra al final. Cada vez que un coche pasa frente a una cámara, se toma una foto de su matrícula y se apunta el momento en que pasó; si el tiempo transcurrido entre la foto del comienzo y la del final es demasiado breve, se le pone una multa. Para simplificar, asumiremos que los tramos no comparten cámaras ni se solapan entre sí. Las operaciones públicas del TAD son:

- *insertaTramo*: añade un nuevo tramo al sistema. Recibe un identificador de tramo, los identificadores de sus cámaras inicial y final, y el número mínimo de segundos que deben transcurrir entre las fotos de comienzo y final para *no* recibir multa. Si el tramo ya existía debe generar un error.
- *fotoEntrada*: se invoca cada vez que un coche entra en un tramo vigilado. Recibe el identificador de la cámara, la matrícula del coche, y el instante actual (en segundos desde el 1 de enero de 1970).
- *fotoSalida*: se invoca cada vez que un coche sale de un tramo vigilado. Recibe el identificador de la cámara, la matrícula del coche, y el instante actual. Si el coche ha ido demasiado rápido en el tramo, se le multará.
- *multasPorMatricula*: devuelve el número de multas asociadas a una matrícula.
- *multasPorTramos*: devuelve una lista con las matrículas de los coches multados en un determinado tramo. Si un coche ha sido multado varias veces, su matrícula aparecerá varias veces en la lista. Si el tramo no existe debe generar un error.

Se pide: prototipos de las operaciones públicas, representación eficiente del TAD (basada en tipos vistos durante el curso), coste de cada operación e implementación en C++ de todas las operaciones.

### 6. BarcoMatic (EDA - Septiembre 2013)

Te han contratado para implementar un sistema de gestión de barcos de pesca. Cada barco tiene una bodega de carga donde los pescadores que van en el barco van depositando las capturas que realizan, anotando siempre la especie del pez y su peso. Cuando el barco llega a puerto, cada pescador se lleva a casa lo que ha pescado de cada especie.

Las operaciones públicas del TAD *BarcoMatic* son:

- *nuevo*: Crea una nueva instancia de la estructura *BarcoMatic*, recibiendo como argumento un el peso máximo (en kilos) admitido en la bodega.

- *altaPescador*: Da de alta a un pescador, identificado por su nombre. No devuelve nada.
- *nuevaCaptura*: Registra que un pescador (que debe estar registrado) ha pescado un ejemplar de tantos kilos de una especie concreta. Las especies y los pescadores se indican mediante sus nombres, y el peso en kilos se especifica mediante un número. Si el peso de la captura, añadido a la bodega, haría que la bodega excediese su capacidad, esta operación debe fallar.
- *capturasPescador*: Recibe el nombre de un pescador, y devuelve una lista de parejas especie-kilos. Si, para una especie dada, el pescador no ha pescado nada, no la debes incluir en la lista devuelta. Puedes asumir la existencia de un TAD *Pareja* $\langle A, B \rangle$  similar al visto en clase.
- *kilosEspecie*: Recibe el nombre de una especie, y devuelve el número total de kilos de esa especie pescados, sumando las capturas de todos los pescadores.
- *kilosPescador*: Recibe el nombre de un pescador, y devuelve el número total de kilos que ha pescado, sumando todas las especies.
- *bodegaRestante*: Devuelve el número de kilos restantes en la bodega.

Se pide: prototipos de las operaciones públicas, representación eficiente del TAD (basada en tipos vistos durante el curso), coste de cada operación (*especificando qué representa la N en cada caso concreto*) e implementación en C++ de todas las operaciones.

## 7. Complejidad Instantánea (adaptado del repositorio de la UVA, problema 586)

Es posible calcular la complejidad de algunos programas de forma muy fácil, siempre y cuando no haya condicionales. El lenguaje “VeryBasic”, que no tiene condicionales, es ideal para esto. La gramática de este lenguaje es la siguiente:

```

<Program> ::= "BEGIN" <Statementlist> "END"
<Statementlist> ::= <Statement> | <Statement> <Statementlist>
<Statement> ::= <LOOP-Statement> | <OP-Statement>
<LOOP-Statement> ::= <LOOP-Header> <Statementlist> "END"
<LOOP-Header> ::= "LOOP" <number> | "LOOP n"
<OP-Statement> ::= "OP" <number>

```

Así, el siguiente programa tiene complejidad  $n^2 + 1997$ , porque OP tiene el coste que indica el número que le sigue, y estar dentro de un bucle con  $n$  repeticiones multiplica el coste de lo que tiene dentro por  $n$ :

```

BEGIN
    OP 1997
    LOOP n
        LOOP n
            OP 1
        END
    END
END

```

Escribe un programa que, dado un programa “VeryBasic”, devuelva la complejidad de este programa. Probablemente debas implementar una clase *Polinomio* que te ayude a modularizar esta tarea (puedes probarlo en el juez de la UVA<sup>2</sup>).

---

<sup>2</sup>[http://uva.onlinejudge.org/index.php?option=onlinejudge&page=show\\_problem&problem=527](http://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=527)



# Bibliografía

---

*Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.*

Miguel de Cervantes Saavedra

- BRASSARD, G. y BRATLEY, P. *Fundamentos de Algoritmia*. Prentice Hall, 1997.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. y STEIN, C. *Introduction to Algorithms*. MIT Press, 2nd edición, 2001.
- MARTÍ OLIET, N., SEGURA DÍAZ, C. M. y VERDEJO LÓPEZ, J. A. *Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios*. Ibergarceta Publicaciones, 2012.
- MARTÍ OLIET, N., ORTEGA MALLÉN, Y. y VERDEJO LÓPEZ, J. A. *Estructuras y datos y métodos algorítmicos: 213 Ejercicios resueltos*. Ibergarceta Publicaciones, 2013.
- PEÑA, R. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.
- RODRÍGUEZ ARTALEJO, M., GONZÁLEZ CALERO, P. A. y GÓMEZ MARTÍN, M. A. *Estructuras de datos: un enfoque moderno*. Editorial Complutense, 2011.
- STROUSTRUP, B. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1998. ISBN 0201889544.