

Estructuras de datos y algoritmos en C++

Mg. Ing. Patricia Calvo - Esp. Lic. Andrés Juárez

Índice general

I	C++ como herramienta de programación	7
1.	Comenzando con C++	9
1.1.	Proceso de compilación	9
1.2.	Integrated Development Environment	10
1.3.	Nociones básicas del lenguaje C++	11
1.4.	Módulos	35
2.	Memoria dinámica	37
2.1.	División de la memoria	37
2.2.	Punteros	39
2.3.	Punteros y vectores	42
2.4.	Pedidos de memoria y liberación	44
2.5.	Punteros a estructuras complejas	45
2.6.	Doble punteros	45
2.7.	Operaciones con punteros	46
2.8.	Persistencia de datos	48
3.	Programación Orientada a Objetos	51
3.1.	Paradigmas de programación	51
3.2.	Tipo Abstracto de Datos	54
3.3.	Diseño de un TDA	58
3.4.	Clases	60
3.5.	Objetos	63
3.6.	Algunas características de la POO	68
3.7.	Acceso a los atributos	71
3.8.	Constructores	72
3.9.	Sobrecarga de métodos	72
3.10.	Parámetros por defecto	73
3.11.	El puntero <i>this</i>	74
3.12.	Atributos y métodos estáticos	74
3.13.	Destruyores	77

3.14. Constructor de copia	79
3.15. Sobrecarga de operadores	82
3.16. Pre y pos condiciones	83
II Estructuras de datos lineales	87
4. Genericidad	89
4.1. TDA Vector	89
4.2. Punteros void	96
4.3. Tipos genéricos	99
4.4. Herencia	101
4.5. Polimorfismo	105
5. Listas	107
5.1. Propiedades y representación	107
5.2. Listas propiamente dichas	108
5.3. Pilas	111
5.4. Colas	111
5.5. Implementaciones	113
5.6. Listas con templates	126
III Estrategias y análisis	127
6. Recursividad	129
6.1. Principios de recursividad	129
6.2. Funcionamiento interno	131
6.3. Algoritmo divide y vencerás	138
6.4. Métodos de ordenamiento usando D y V	144
7. Complejidad	149
7.1. Complejidad temporal algorítmica	149
7.2. Conteo	154
7.3. Medidas asintóticas	160
7.4. Análisis de complejidad en algoritmos recursivos	166
7.5. Balance entre tiempo y espacio	173
IV Estructuras no lineales	177
8. Árboles	179

8.1. Árboles binarios	180
8.2. Árboles binarios de búsqueda	183
8.3. Recorridos en un ABB	194
8.4. Árboles balanceados	198
8.5. Árboles multivías	208
8.6. Array de bits	212
8.7. Trie	215
8.8. Cola con prioridad	217
8.9. Heap	218
9. Grafos	229
9.1. Definición matemática de grafo	230
9.2. TDA Grafo	232
9.3. Implementaciones	234
9.4. Recorridos	234
9.5. Caminos en un grafo	250
9.6. Árbol de expansión de coste mínimo	257
V Temas complementarios	263
10.Hashing	265
10.1. Introducción	265
10.2. Funciones de hashing	266
10.3. Colisiones	269
10.4. Borrado de un elemento	273
10.5. Funciones de hash perfectas	273
10.6. Funciones de hash para archivos extensibles	274
A. Código	275
A.1. Implementación y uso de Vector con templates	275
A.2. Implementación y uso pila estática	279
A.3. Implementación dinámica y uso de Pila	282
A.4. Implementación dinámica y uso de Cola	286
A.5. Implementación y uso lista simplemente enlazada	289

Parte I

C++ como herramienta de programación

Capítulo 1

Comenzando con C++

El lenguaje de programación C++ es un superconjunto del lenguaje C, es decir, incluye todas las características de C y agrega otras nuevas. El lenguaje C está diseñado para escribir programas bajo el paradigma de programación estructurada. Este paradigma enfoca los problemas manteniendo dos conceptos por separado: datos y funciones. Por un lado tenemos los datos y por otro las funciones que los manipulan. C++ evoluciona a partir de C consiguiendo que el lenguaje esté orientado a objetos. La Programación Orientada a Objetos (POO) ya no analiza por separado los datos y funciones, sino que los encapsula o engloba en una entidad que llamamos *clase*. Si bien C++ está desarrollado para construir soluciones con el enfoque POO, al incluir el lenguaje C, también nos permite hacer desarrollos solo de manera estructurada, o una mezcla entre los dos paradigmas que llamamos híbrido.

1.1. Proceso de compilación

El lenguaje C++ es un lenguaje compilado, es decir, se necesita de un compilador que entre otras cosas traduzca el código fuente que escribimos a lenguaje máquina y genere un archivo ejecutable. Si bien hay muchos compiladores de C++ recomendamos utilizar el GNU GCC de software libre que corre en sistemas operativos como Windows, Linux y Mac OS X.

Cuando escribimos un código y lo compilamos pasa por tres etapas:

- Preproceso. En esta etapa lo que el compilador realiza es eliminar los comentarios e incluir los archivos indicados con la directiva *#include*.
- Compilación propiamente dicha. En esta fase traduce el código a lenguaje máquina. El compilador nos indicará los errores que encuentre, como variables que no están declaradas, instrucciones mal escritas, falta

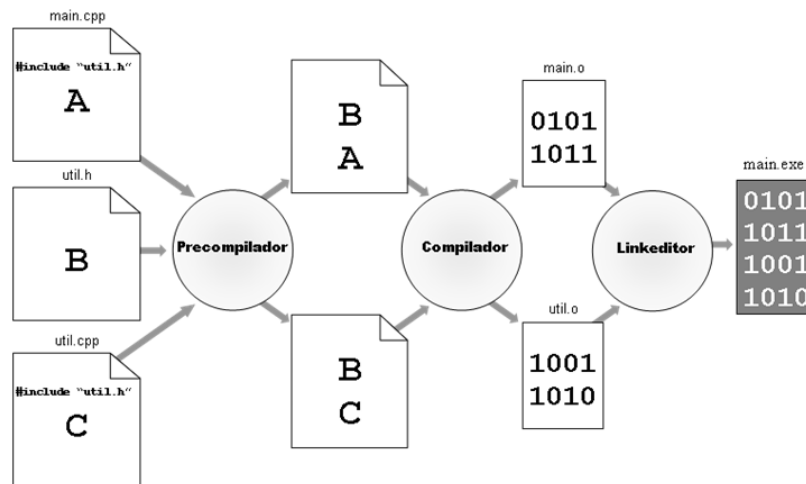


Gráfico 1.1.1: Proceso de compilación

de un punto y coma, etc. Si el código está escrito correctamente genera los archivos objeto que son archivos binarios.

- Enlazado o linkedicion. Finalmente, en esta última etapa el compilador junta (enlaza) todos los archivos traducidos a lenguaje máquina para generar el archivo ejecutable. Si estamos bajo una plataforma de Windows, el archivo tendrá extensión `.exe`. En Linux no agrega ninguna extensión.

En el gráfico 1.1.1 mostramos un ejemplo de cómo funciona el proceso de compilación para un proyecto con tres archivos: uno de cabecera `util.h` con su correspondiente archivo de definición `util.cpp`, junto con el archivo `main.cpp`.

1.2. Integrated Development Environment

En principio, para escribir nuestros programas solo necesitamos un compilador de C++. Por ejemplo, bajamos e instalamos el compilador gratuito GNU GCC que nos sirve tanto para Linux como para Windows mediante MinGW. Luego escribimos nuestro código fuente en cualquier editor de texto y lo guardamos en un archivo de extensión `cpp`. Finalmente, por línea de comando podemos compilarlo y luego ejecutarlo.

Sin embargo, esto no es recomendable salvo que nuestro código sea muy simple. ¿Por qué? Porque los simples editores de texto no nos marcarían las palabras reservadas con colores, tampoco nos brindarían una lista de los

métodos asociados a cierto objeto, ni nos indicarían palabras mal escritas, entre otras cosas. Se complicaría aún más si necesitáramos trabajar con varios archivos en un mismo proyecto. Tampoco tendríamos ninguna herramienta para seguir nuestro código, en el caso de que haya compilado bien pero tenga algún error a la hora de su ejecución o de tipo lógico, como una salida que no es la esperada. En estos casos estaríamos deseosos de poder seguir el código paso a paso, observando los valores de las distintas variables. Todos estos problemas los soluciona un Integrated Development Environment (IDE) o, en español, Entorno de Desarrollo Integrado. Los IDEs son aplicaciones que nos permiten el desarrollo de aplicaciones.

Un IDE, entre otras cosas viene con:

- Un editor de código que nos indica con distintos colores si la palabra es reservada o no.
- Nos despliega todos los métodos que se pueden llamar desde un objeto determinado.
- Un depurador o debugger para seguir el código paso a paso.

Hay varios IDEs que se pueden utilizar. Nosotros recomendamos utilizar Eclipse bajo plataforma Linux, ya que además podemos agregarle una herramienta para el control del manejo de la memoria dinámica que es Valgrind. Otros dos IDEs que recomendamos son Code::Blocks, ya que es muy sencilla su instalación y manejo, y Dev-C++. Para los usuarios de Mac, el IDE más utilizado es Xcode.

1.3. Nociones básicas del lenguaje C++

La mejor forma de aprender un lenguaje es comenzar a utilizarlo con ejemplos sencillos, los que irán incrementando su complejidad a medida que avancemos. Por tradición en todas las enseñanzas de lenguajes de programación el primer programa a realizar es un programa que imprime “Hola mundo” en nuestro monitor.

Ejemplo 1.1. Primer programa.

Abrimos el IDE que hemos instalado, generamos un nuevo proyecto y, dependiendo del IDE que usemos, nos genera algunas sentencias de manera automática. De lo contrario, escribimos:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     cout << "Hola mundo!" << endl;
8     return 0;
9 }
```

Al ejecutarlo observamos una ventana con un cartel que dice “Hola mundo!”.

Si analizamos este código, línea por línea, tenemos lo siguiente:

1. `#include <iostream>`

Esta línea indica que se está incluyendo una librería que se llama *iostream*. La palabra *stream* se utilizará como canal o corriente que se refiere a flujo de datos. En este caso, el flujo serán los caracteres de entrada / salida. Podemos pensarlo como un *canal* en donde van ingresando los caracteres a imprimirse. La *io* significa input / output, por lo tanto, esta librería tiene funciones para el manejo de la entrada y la salida.

La entrada estándar será el teclado y la salida estándar la consola o pantalla donde vemos los mensajes.

Luego, las líneas 2 y 4, están en blanco. Toda línea en blanco, el compilador la obviará, al igual que los comentarios. Se utilizan para una mayor claridad del código, dividiendo las distintas secciones de un programa.

3. `using namespace std;`

Con esta instrucción, estamos diciendo que utilice un espacio de nombres llamado *std*. Más adelante veremos mejor qué significa esto, pero se adelanta que, para utilizar la instrucción *cout* y *endl*, debemos indicarle que están definidas dentro del espacio de nombres estándar, es decir, en *std*.

5. `int main()`

Esta instrucción es la cabecera de una función. Toda función se indica de la siguiente forma:

tipo_de_devolución | nombre_de_la_función | parámetros

En este caso, el tipo de devolución es un *int*, es decir, un entero (luego se verá la utilidad de esto). El nombre de la función es *main*. Esta es una función especial, que dice que el programa se empezará a ejecutar a partir de esa función y no de otra. Por esta razón todo proyecto debe tener una función *main* para que comience a ejecutarse el código desde ahí.

6. {
9. }

Las líneas 6 y 9 son llaves que abren y cierran, respectivamente, el cuerpo de un bloque. En este caso, abren y cierran la función *main*.

Representa	Tipo	Tamaño (bytes)
Enteros	char	1
	int	entre 2 y 8
Booleano	bool	1
Flotantes	float	entre 4 y 8
	double	entre 8 y 16
Vacío	void	

Tabla 1.3.1: Tipos básicos

```
7. cout << "Hello world!" << endl;
```

Esta línea contiene la primera instrucción propiamente dicha del programa. Indica que imprima lo que se encuentra entre comillas. El *endl* se refiere a un fin de línea, lo que hace que el cursor avance al siguiente renglón o línea.

Lo que va luego de los dos símbolos de menores, se coloca en el *stream* (flujo) de datos. *cout* indica que el flujo que viene detrás debe enviarse a la salida estándar, es decir, la consola. Los distintos flujos se concatenan con los dos símbolos de menores. Aquí vemos que se concatenan los caracteres que forman la frase “Hola mundo!” y, luego, el carácter especial de fin de línea.

```
8. return 0;
```

Finalmente, dijimos que la función debe retornar un entero. Esto se realiza con la instrucción *return*, indicando que se devolverá el entero cero. Podríamos haber devuelto otro entero cualquiera, por ejemplo, uno o menos treinta. Pero, en general, se utiliza una devolución de cero para indicar que todo funcionó bien, sin ningún error. Los valores distintos de cero indican algún error en la ejecución.

Si el lector nunca programó en C o C++ y lo anterior le resulta un tanto extraño no se debe preocupar, ya que a medida que vayamos construyendo otros programas / ejemplos se irá comprendiendo el funcionamiento.

1.3.1. Tipos de variables básicas

En el cuadro 1.3.1 se muestran los tipos básicos:

Analizando un poco la tabla anterior nos encontramos que el tipo *char* está dentro de los tipos enteros. Si bien se utiliza para almacenar caracteres, también se puede utilizar para almacenar números no demasiado grandes, como una edad, etcétera. Cada carácter se representa según su código ASCII, en donde, por ejemplo, la A lleva la misma codificación binaria que el número 65, la B es 66, etcétera. Por lo tanto, el dato almacenado, se puede interpretar como un carácter o como un entero.

El tamaño de un `char` es de un byte, por lo tanto, se pueden almacenar $2^8 = 256$ símbolos diferentes, ya sean números o caracteres.

El tamaño del resto de los tipos depende de la plataforma, es decir, depende de la arquitectura de la máquina, del sistema operativo y del compilador que se utilice. En general, un `int` ocupará 4 bytes, por lo que puede almacenar 2^{32} números, más de 4 mil millones.

Si bien un dato booleano se podría almacenar en solo un bit, la menor medida posible de almacenamiento es un byte, por lo tanto, una variable de este tipo ocupará un byte. Nos sirve para almacenar dos tipos de valores: *true* o *false*. Tanto C, como C++, representan un valor falso como el entero 0 y, el verdadero, como cualquier valor distinto de cero (en el caso de los booleanos, como el uno). Por lo tanto, si imprimiéramos el valor de una variable booleana, tomándola como un entero (cosa que no tendría demasiado sentido a excepción de tener en cuenta el funcionamiento interno), veríamos que imprime un cero si la variable tiene valor falso y un uno si es verdadero.

Los datos de tipo flotantes almacenan números racionales, pero no todos, ya que entre dos números racionales hay infinitos racionales, por lo que se necesitaría una cantidad de memoria infinita para guardarlos. La diferencia entre dos números racionales se llama “ancho de paso”. Este ancho de paso no es fijo, es decir, no es constante entre todos los números. La idea es buscar una mayor precisión en números chicos, cercanos al cero, que en números grandes dado que interesa minimizar los errores relativos. Por ejemplo, un error de 60cm en una medición puede ser demasiado o insignificante. Si se afirma que algo mide 0,25mts en lugar de 0,85mts la equivocación es importante. En cambio, decir que algo mide 4563,25mts en lugar de 4563,85mts, la diferencia no es significativa. Por lo tanto, el ancho de paso es muy chico para los números cercanos a cero, y se va agrandando a medida que los números son más grandes en valor absoluto. Este ancho de paso puede llegar a representar varios miles, incluso billones de números enteros. Es por este motivo que está el tipo *double* para lograr una mayor precisión, ya sea en números chicos o en grandes. Al tener mayor capacidad, el ancho de paso es menor.

Finalmente, tenemos el tipo *void*, que indica ausencia de valor. Se utiliza en ciertos casos, por ejemplo, cuando una función no necesita devolver nada. Más adelante veremos más sobre este tipo de dato.

1.3.2. Identificadores

Los identificadores son palabras que definiremos para utilizarlas como nombres de variables, constantes o etiquetas. Pueden formarse por la combinación de letras, dígitos (números), guión bajo “_” o el signo \$. Pero no pueden comenzar con números. Por ejemplo *x1*, como nombre de variable es

válido pero no lo es *1x*. Estos nombres no pueden ser palabras reservadas del lenguaje. El lenguaje es *case sensitive*, es decir, sensible a las mayúsculas y minúsculas, por lo que el identificador *NUMERO* será diferente de *Numero* y de *numero*.

Estos identificadores pueden tener la cantidad de caracteres que uno desee pero, nunca se debe perder de vista el buen criterio. Los identificadores demasiado abreviados o demasiado largos pueden oscurecer el código.

1.3.3. Variables

Una variable la podemos pensar como un lugar donde guardar datos que pueden variar a lo largo del programa o pueden no estar definidos al principio del mismo. Este lugar estará en la memoria ocupando una o más celdas según sea su tipo. Las variables que vayamos a utilizar debemos declararlas. Si bien podemos declararlas con anterioridad, se recomienda siempre que se pueda declararlas en el mismo momento de su inicialización. ¿Cómo se declara una variable? Simplemente indicando su tipo y luego un nombre. Por ejemplo:

```
int numero;
```

Al finalizar cada sentencia se debe cerrar con un punto y coma.

Las variables viven en el ámbito donde se las declara, es decir, dentro del bloque donde se manifiestan. Fuera de estos bloques no pueden ser accedidas. Por ejemplo, si declaramos una variable dentro de un ciclo while no estará disponible una vez que el ciclo termine.

Las buenas prácticas de programación aconsejan escribir las variables con minúsculas. En caso de necesitar una variable con un nombre compuesto por más de una palabra es aconsejable separarlas con un guión bajo o con una mayúscula al comienzo de las diferentes palabras. Por ejemplo, si necesitáramos almacenar el importe neto de una factura, se recomienda utilizar alguna de estas dos opciones:

- `importeNeto`

- `importe_netto`

La segunda opción es más clara que la primera a la hora de leer el código, sin embargo la primera opción es la más utilizada. Este criterio de diferenciar las palabras se llama *camelCase*.

1.3.4. Constantes

Si bien la idea de constante es similar al de variable en el sentido en que también tendrá cierto tipo y ocupará cierto lugar en memoria, la diferencia

es que conocemos desde el principio su valor, el cual no mutará a lo largo del programa. Por ejemplo el valor de un impuesto o una cantidad máxima de clientes, etcétera.

Para definir una constante lo hacemos igual que una variable pero anteponiendo la palabra *const*. Las buenas prácticas aconsejan utilizar letras mayúsculas para diferenciarlas de las variables. En este caso, si tenemos un nombre compuesto no nos queda otra forma de separar las palabras mediante un guión bajo.

Algunos ejemplos:

- `const int IVA = 21;`
- `const float PI = 3.141593;`
- `const int MAX_CLIENTES = 5000;`

1.3.5. Comentarios

Los comentarios no son tenidos en cuenta por el compilador pero son importantes para la persona que mira el código. Debemos pensar que cualquier modificación que haya que hacer al código fuente de un programa implica en primer lugar su entendimiento, por lo que los comentarios nos ayudarán a comprender ciertos pasajes dudosos.

Hay dos tipos de comentarios:

- De una línea: con doble barra al principio del comentario.
`// es un comentario de una línea que no debe cerrarse`
- De más de una línea: con barra asterisco para abrir y asterisco barra para cerrar.
`/*
es un comentario
de varias líneas
que necesita cerrarse
*/`

1.3.6. Modificadores

Los modificadores son palabras reservadas que se anteponen a los tipos de variables, con el fin de adaptar el tipo según los datos que se almacenarán. Por ejemplo, si sabemos que utilizaremos números enteros, mayores o iguales a cero, podríamos utilizar el modificador *unsigned*. Por lo tanto, definimos


```
unsigned int variable;
```

Es decir, *variable* almacenará números de tipo entero, pero solo positivos. Los modificadores de los enteros son *signed* / *unsigned*. Por defecto, al no indicar nada, los enteros son de tipo *signed*, es decir, con signo, a excepción del tipo *char* que depende de la plataforma.

Además, al tipo *int* se le puede agregar el modificador *short* o *long*, según sea si necesitamos un entero corto o largo. Las longitudes varían según la plataforma. La regla general es la siguiente:

$$\text{size}(\text{short int}) \leq \text{size}(\text{int}) \leq \text{size}(\text{long int})$$

Es decir, el tamaño de un *short int* será menor o igual que el de un *int*, el que, a su vez, es menor o igual que el de un *long*. En mi plataforma los tamaños son 2, 4, 4, respectivamente. El tipo *double* también puede llevar el modificador *long*, aunque creemos que no tiene demasiado sentido, salvo que se esté trabajando con datos muy específicos.

Para declarar variables de los tipos vistos anteriormente, se debe, simplemente, indicar el tipo con un identificador (el nombre de la variable). A continuación se dan algunos ejemplos:

<code>int x;</code>	// Variable de tipo entera con signo
<code>char c;</code>	// Variable de tipo char, puede ser con signo o sin él
<code>unsigned int a;</code>	// Variable entera sin signo (solo positivos y cero)
<code>unsigned long int b;</code>	// Variable de tipo entero largo sin signo
<code>bool b;</code>	// Variable booleana

1.3.7. Tipos de datos derivados

Son los tipos de datos que crea el usuario. Un tipo de dato primordial es el de *clase*, pero lo veremos más adelante cuando se vean *Tipos de Datos Abstractos* y Programación Orientada a Objetos. Los otros tipos que veremos a continuación son los vectores, las estructuras y los enumerados.

Vectores

Los vectores son arreglos lineales de elementos del mismo tipo. Por ejemplo, si queremos definir un vector de 10 enteros indicamos:

```
int vector[10];
```

Esto nos genera un vector de 10 posiciones, en donde cada posición alberga un entero. La primera posición es la posición cero y, la última, es $N - 1$. En este caso, nueve. Entonces, para referirse a la posición *i*, simplemente, hay que indicar el nombre de la variable general, en este ejemplo es *vector* y, entre corchetes, la posición *i*.

```
vector[i]
```

Si necesitáramos una matriz, simplemente se indica con otro corchete la segunda dimensión. Por ejemplo:

```
int matriz [4] [3];
```

declara una matriz de enteros de cuatro filas por tres columnas.

Estructuras

Una estructura define un tipo compuesto (aunque podría no serlo). Por ejemplo, necesitamos guardar los datos de un empleado. Los mismos consisten en, número de legajo, nombre y sueldo. Entonces armamos una estructura de la siguiente forma:

```
struct Empleado
{
    int legajo;
    string nombre;
    float sueldo;
};
```

De esta forma creamos un nuevo tipo de variable: *Empleado*. Si la queremos utilizar, simplemente indicamos:

```
Empleado empleado;
```



Nota: a diferencia de C, en donde hay que indicar *struct* en la declaración de la variable o utilizar *typedef*, en C++ no es necesaria esta inclusión. Nótese que se utilizó el tipo *string* para el nombre, más adelante se profundizará sobre este tipo.

La variable empleado es una estructura (también se puede llamar registro), que tiene tres campos: *legajo*, *nombre* y *sueldo*. Para acceder a cada uno de ellos se debe utilizar el operador “.” Ejemplo:

```
empleado.legajo = 234;
```

Por supuesto, un campo de una estructura puede, a su vez, ser otra estructura. También se podrán crear vectores de estructuras y viceversa: estructuras con vectores, aunque esta última variante no sea recomendada.

Enumerados

Los tipos enumerados son enteros constantes especiales que se determinan en conjunto. Por ejemplo, si tenemos un *partido_de_futbol* que puede tener entre otras cosas un *estado*, y ese estado puede ser *NO_COMENZADO*, *INICIADO*, *FINALIZADO*, podemos crear un tipo *enum* con dichos estados. El código será:

```
enum Estado {
    NO_COMENZADO;
    INICIADO;
```

Tipo	Operación	Operador	Cantidad de operandos
Aritméticos	Suma	+	Binarios
	Resta	-	
	Producto	*	
	División	/	
	Resto en división entera	%	
Asignación	Asignación simple	=	Binario
	Post incremento	var++	Unitario
	Pre incremento	++var	Unitario
	Post decremento	var--	Unitario
	Pre decremento	--var	Unitario
Comparación	Igualdad	==	Binarios
	Distinto	!=	
	Mayor	>	
	Mayor o igual	>=	
	Menor	<	
	Menor o igual	<=	
Lógicos	Y (and)	&&	Binario
	O (or)		Binario
	No (not)	!	Unitario
Tamaño	Tamaño de una variable	sizeof	Unitario

Tabla 1.3.2: Operadores

```

    FINALIZADO;
};

```

Internamente, *NO_COMENZADO* tendrá el valor 0, *INICIADO* el valor 1, etc. Pero esto no nos debe interesar, el uso que le daremos no es en base a los valores que toma, sino utilizando dichas constantes. Un ejemplo de uso sería:

```

Estado estado;
estado = INICIADO;
...
if (estado == INICIADO)
    cout << "El partido se está jugando";

```

1.3.8. Operadores

Los operadores son símbolos que indican al compilador que debe realizar determinadas operaciones. En la tabla 1.3.2 se da un listado de los mismos.

Los operadores, en general, se utilizan combinados. Por ejemplo:

```
a = c + d;
```

Estamos utilizando el operador suma, el cual suma las variables *c* y *d* y, luego, el resultado, lo asigna (operador de asignación) a la variable *a*.

El operador / (división), realiza la división entera si los dos operandos son enteros. En cambio, si uno solo es de tipo flotante o *double* realiza la operación flotante. Algunos ejemplos:

Ejemplo 1.2. División

```
int x = 8, y = 5, z;  
z = x / y;
```

z queda con valor 1, ya que realiza la división entera

Ejemplo 1.3. Más sobre la división

```
int x = 8, y = 5;  
double z;  
z = x / y;
```

A diferencia de lo que uno podría esperar, *z* queda con valor 1.0, dado que realiza la división entera (tanto *x* como *y* son variables de tipo *int*). Luego, asigna el valor a *z*, por eso le agrega el ,0 para indicar que es un flotante. Sin embargo, la división quedó con un número entero.

Ejemplo 1.4. División con decimales

```
int x = 8;  
double y = 5, z;  
z = x / y;
```

Ahora *z* queda con valor 1,6, ya que al intervenir una variable de tipo *double* en la división, realiza la operación con flotantes.

Ejemplo 1.5. El operador % guarda el resto de una división entera.

```
int x = 8 % 5;
```

x guarda el valor 3 ya que la división entera entre 8 y 5 es 1, quedando como resto 3.

Ejemplo 1.6. Los operadores de incremento (o decremento), son una combinación resumida de operaciones:

```
int x = 3;  
x++;
```

x queda con el valor 4. *x++* es equivalente a hacer *x = x + 1*.

Ejemplo 1.7. Operador de post incremento

```
int x = 3, y;  
y = x++;
```

`x` queda con el valor 4, pero `y` con el valor 3, por ser de tipo “post” (primero asigna y luego incrementa).

Las sentencias anteriores equivalen a lo siguiente:

```
int x = 3, y;  
y = x;  
x = x + 1;
```

Ejemplo 1.8. Operador de pre incremento

```
int x = 3, y;  
y = ++x;
```

En este caso, tanto `x` como `y` quedan con el valor 4, por ser de tipo “pre” (primero incrementa y luego asigna).

Las sentencias anteriores equivalen a lo siguiente:

```
int x = 3, y;  
x = x + 1;  
y = x;
```

Lo mismo sucede con los operadores de decremento.

Con respecto a los operadores de comparación, se debe tener cuidado ya que la consulta sobre la igualdad de elementos es con un doble igual, dado que uno solo produciría una asignación, en lugar de una comparación.

Ejemplo 1.9. Operador *sizeof*. Indica el tamaño en bytes de una variable o de un tipo.

```
cout << sizeof (int) << endl;
```

imprime la cantidad de bytes que ocupa un entero.

Precedencia de los operadores

Cabe destacar que los operadores tienen diferentes niveles de precedencia, cuanto más alto sea el nivel de precedencia, antes se ejecutará ese operador. Por ejemplo, el operador `*` (multiplicación) tiene mayor nivel de precedencia que el `+` (suma). Por lo tanto:

```
int a = 4, b = 3, c = 2, d;  
d = a + b * c;
```

`d` guarda el valor 10, ya que en primer lugar resuelve la multiplicación entre las variables `b` y `c`, dado que la multiplicación se encuentra con mayor prioridad que la suma. Luego, a ese resultado, le practica la suma al valor que contiene `a`. Finalmente, lo asigna a la variable `d`. Es decir, respeta la operación matemática al dividir en términos y realizar la cuenta en forma manual. De todos modos, ante la duda, es aconsejable utilizar paréntesis para asegurarse de obtener los resultados que uno desea. Por ejemplo:

```
int a = 4, b = 3, c = 2, d;  
d = a + ( b * c );
```

```
// d guarda el valor 10
int a = 4, b = 3, c = 2, d;
d = ( a + b ) * c;
// d guarda el valor 14
```

Si bien hay más operadores, algunos los veremos más adelante, como los operadores *new* y *delete*, que se utilizan para el manejo de memoria dinámica. Otros se pasarán por alto, por ejemplo los operadores de bits o los de abreviaturas de operaciones, con la finalidad de no cargar al lector con demasiada información que por el momento no es relevante.

Ejemplo 1.10. Formas abreviadas de expresiones de asignación

Operación	Tipo
acumulador = acumulador + cuenta;	// Forma expandida
acumulador += cuenta;	// Forma abreviada

1.3.9. Conversiones de tipo

Cuando una variable se asigna a otra siendo de diferente tipo se debe hacer una conversión o casteo de tipo. Este casteo puede ser implícito o explícito. Por ejemplo:

```
char x = 5;
int y = x; // conversión implícita o automática
```

En el código anterior se asigna una variable de tipo *char* a otra de tipo *int*. Como el tipo *char* es un tipo de dato que está incluido en uno de tipo *int* se realiza una conversión automática o implícita. En cambio, si el código fuera al revés:

```
int x = 5;
char y = x; // da error
```

Aunque sepamos que el valor 5 se puede guardar en una variable de tipo *char*, da un error porque una variable de tipo *int* no está incluida en una de tipo *char*, por lo tanto debemos hacer un casteo explícito de la siguiente forma:

```
char y = (char) x; // correcto
```

En la línea anterior indicamos que tome la variable *x* que es de tipo *int* como si fuera de tipo *char*.

1.3.10. Funciones

Las funciones sirven para modularizar el código y hacerlo más legible y sencillo de corregir o modificar. Cada vez que veamos que una porción de código se repite en diferentes lugares, debemos pensar que esa porción debería conformar una función a la cual llamaremos cada vez que la necesitemos.

Las funciones como ya mencionamos constan de una cabecera o firma:

tipo_de_devolución | nombre_de_la_función | parámetros

Luego vienen las llaves de apertura y cierre. Dentro estará el código necesario para dicha función.

Por ejemplo, si queremos hacer una función que devuelva el mínimo entre dos números enteros podemos escribir:

```
int minimo (int x, int y) {
    if (x > y)
        return y;
    else
        return x;
}
```

La cabecera consta de:

tipo_de_retorno	nombre_de_la_funcion	parametro_1	parametro_2
int	minimo	int x	int y

Si bien el código anterior es correcto, como la sentencia *return* corta la ejecución de la función, se suele no escribir el *else*:

```
int minimo (int x, int y) {
    if (x > y)
        return y;
    return x;
}
```

El llamado a la función se realiza con el nombre y los parámetros necesarios para su ejecución. Por supuesto, debemos tomar en otra variable o imprimir el resultado que devuelve, de lo contrario dicho valor se pierde. Por ejemplo:

```
int x = minimo (5, 8);
```

Si una función solo debe realizar por ejemplo una impresión por pantalla, sin necesidad de ningún retorno, se coloca la palabra *void* como tipo de devolución, lo que significa devolución vacía, por lo que no hay necesidad de finalizar la función con una sentencia *return*. Por ejemplo:

```
int imprimir (int x) {
    cout << "El valor de la variable es: " << x << endl;
}
```

Parámetros

Una función puede recibir desde ninguno a la cantidad de parámetros que se necesiten. En el caso de no pasar ningún parámetro se dejan los paréntesis vacíos. En caso de recibir más de uno se listan separados por comas.

Los parámetros pueden ser por referencia o por valor. En el caso de tipos de datos simples, como un entero, un char, un float, etc, los parámetros se

Algoritmo 1.1 Parámetros por valor y por referencia

```
void intercambiar(int x, int y) {
    int aux = x;
    x = y;
    y = aux;
}

void intercambiar2(int & x, int & y) {
    int aux = x;
    x = y;
    y = aux;
}

int main() {
    int x = 5, y = 8;
    cout << "Antes del llamado a las funciones" << endl;
    cout << "x = " << x << " y = " << y << endl;
    intercambiar(x, y);
    cout << "Despues del llamado a la funcion intercambiar" << endl;
    cout << "x = " << x << " y = " << y << endl;
    intercambiar2(x, y);
    cout << "Despues del llamado a la funcion intercambiar2" << endl;
    cout << "x = " << x << " y = " << y << endl;
    return 0;
}
```

pasan por valor, salvo que se indique lo contrario. Esto quiere decir que la función copia los parámetros en una nueva variable. Por lo tanto, cualquier modificación que se haga a estos parámetros dentro de la función no modifica al parámetro actual, es decir, a la variable original. En cambio, si queremos que estas modificaciones afecten a la variable que se pasa por parámetro debemos pasarla por referencia anteponiendo el símbolo `&`. En el código que figura en 1.1 vemos dos funciones *intercambiar* e *intercambiar2* con el mismo código, la única diferencia es que en la primera los parámetros no se modifican ya que están pasados por valor, en cambio en la segunda están pasados por referencia.

Al ejecutar el código, lo que imprime es lo siguiente:

```
Antes del llamado a las funciones
x = 5 y = 8
Despues del llamado a la funcion intercambiar
x = 5 y = 8
Despues del llamado a la funcion intercambiar2
x = 8 y = 5
```


Vectores y matrices por parámetro

Los vectores y las matrices también son pasadas por valor, sin embargo, lo que se copia es solo la dirección donde se encuentran los datos por lo que a los efectos prácticos se puede pensar que están pasadas por referencia ya que este tipo de variables conservan las modificaciones cuando se alteran dentro de las funciones. La explicación se entenderá mejor cuando veamos punteros en el siguiente capítulo. Por ejemplo, el siguiente código carga en una función un vector de tamaño n con los cuadrados de los números desde 1 hasta n .

```
void cargar(int vec[], int n) {  
    for (int i = 0; i < n; i++)  
        vec[i] = (i+1)*(i+1);  
}
```

Si el código anterior no se comprende totalmente no hay que preocuparse ya que el ciclo *for* lo veremos más adelante. Lo importante a observar son dos cosas:

- El vector no lo estamos pasando por referencia, no tiene el símbolo `&` delante. Sin embargo, el vector queda cargado.
- No es necesario pasarle el tamaño del vector entre corchetes aunque podríamos hacerlo. Si fuera una matriz deberíamos indicarle la segunda dimensión porque internamente lo guarda en celdas contiguas, es decir al igual que un vector.

¿Qué sucede si la variable n del segundo parámetro no coincide con el tamaño real del vector?

- Si n es menor que el tamaño del vector estaremos desperdiciando memoria. Es una práctica habitual tener cierto porcentaje de desperdicio de memoria. En estos casos manejamos dos tamaños distintos:
 - Tamaño físico: es el espacio que realmente ocupa el vector en la memoria.
 - Tamaño lógico: es el que utilizamos para los cálculos, impresiones, etc.
- Si n es mayor que el tamaño real del vector nos sucederá que intentaremos escribir en una zona de la memoria que no hemos reservado, cosa que en general nos dará un error en tiempo de ejecución.

En el gráfico 1.3.1 vemos la representación de una matriz de 3 filas por 5 columnas. En la parte *a*) el gráfico muestra cómo lo interpretamos, y en la

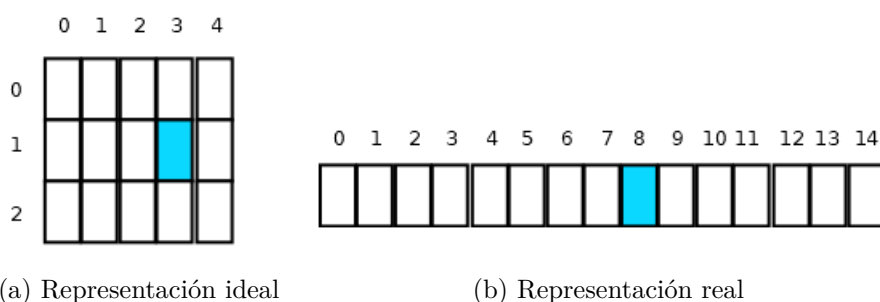


Gráfico 1.3.1: Representación de las matrices

parte *b*) cómo realmente la almacena. Tenemos que tener en cuenta que el almacenamiento es en la memoria que es una tira de *bytes*. Por lo tanto, si quisiéramos dirigirnos a la celda que está marcada en celeste deberíamos escribir el nombre de la variable, supongamos que se llama *matriz*, y en un corchete el número de fila y en un segundo el de la columna, de la siguiente manera:

```
matriz[1][3]
```

Sin embargo, el compilador debe ir a la celda número 8 que vemos en la figura *b*). Para hacer este cálculo necesita multiplicar el número de fila por la cantidad de columnas y sumar el número de columna:

$$\text{numeroFila} * \text{columnas} + \text{numeroColumna} = 1 * 5 + 3 = 8$$

Es por este motivo que el compilador necesita conocer la segunda dimensión de la matriz. Si fuera una matriz *n-dimensional* donde *n* puede ser 3, 4, etc, sucede lo mismo, solo la primera dimensión puede quedar sin definir.

1.3.11. Control de flujo

Para poder realizar programas útiles debemos poder controlar el flujo de las sentencias. Es decir, algunas sentencias se ejecutarán solo si se cumple determinada condición, aquí aparecen las estructuras condicionales. En otras situaciones, podríamos desear que se ejecutaran *n* veces algunas sentencias, siendo *n* algún número muy grande o un número variable. Por este motivo necesitamos de estructuras repetitivas. Veamos las distintas estructuras.

Sentencias secuenciales

Las sentencias secuenciales son las que hemos visto hasta el momento: sentencias de asignación, de impresión, etc. Al ejecutarse una detrás de la

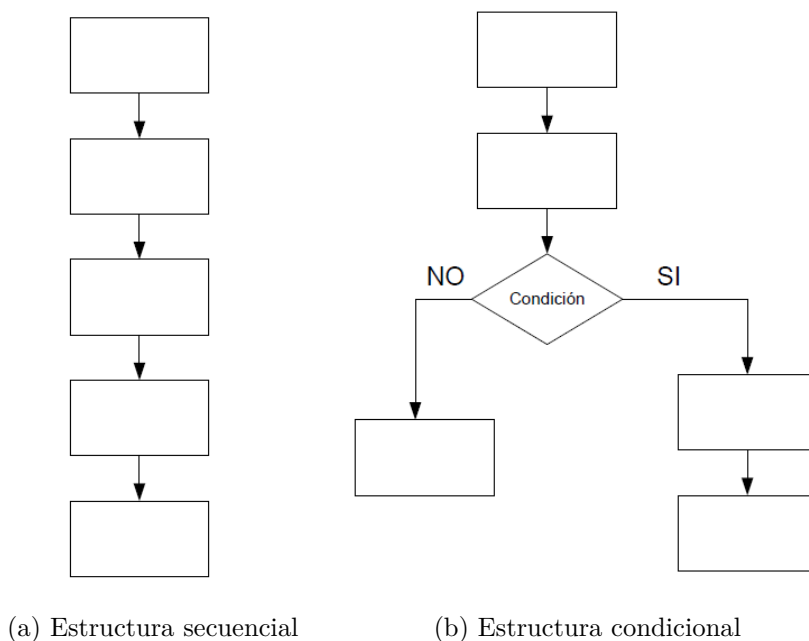


Gráfico 1.3.2: Diagramas de flujo

otra decimos que estamos en una estructura de tipo secuencial. Los bloques se delimitan con llaves:

```
{ // Se abre un bloque
// Sentencias del bloque
} // Se cierra el bloque
```

Sentencias condicionales o selectivas

Las estructuras condicionales hacen que el flujo de ejecución se ramifique. O, pensado de otra manera, si la ejecución de sentencias es siempre secuencial (una detrás de la otra), con una estructura condicional podremos decidir que una sentencia o más se ejecuten solo si se cumple determinada condición.

En el gráfico 1.3.2 vemos ejemplos de ambas estructuras: secuenciales y condicionales. Por supuesto que las estructuras condicionales están, a su vez, formadas por estructuras secuenciales.

if – else Funciona al igual que en C. La sintaxis es la siguiente:

```
if (condicion) sentencia1;
else sentencia2;
```

El *if* verifica que la condicion se cumpla, en ese caso ejecuta la sentencia1. El *else* es optativo ya que podríamos desear no ejecutar nada si la condición

no se cumple. Por otro lado, si necesitáramos ejecutar más de una sentencia deberíamos abrir bloques con llaves. De todas formas podría ser una buena práctica abrirlos siempre, aunque fuera solo una sentencia por una cuestión de claridad en el código. Veamos algunos ejemplos en los algoritmos que figuran en 1.2.

En el ejemplo 3 hay dos sentencias dentro del *if*, por este motivo está la necesidad de usar las llaves de apertura y cierre. El ejemplo 4 utiliza más de una condición para entrar al *if*, como están concatenadas con un *and*, ambas deben ser verdaderas para que ingrese en ese bloque. Es conveniente, cuando se anidan sentencias, dejar sangrías para ubicar rápidamente a qué bloque corresponde cada sentencia. En el ejemplo 5 vemos el uso de condiciones anidadas, es decir, una dentro de otra.

switch Cuando se necesitan anidar varios *if* y las condiciones son evaluaciones de expresiones que devuelven un entero, se puede y se recomienda utilizar *switch*. La sintaxis es:

```
switch (expresion) {
    case valor1: sentencia_1;
    case valor2: sentencia_2;
    ...
    default: sentencia_n; // El default es optativo
}
```

Por ejemplo, si queremos tomar un número del uno al cinco y escribirlo con letras podríamos hacer lo siguiente:

```
// Ejemplo sin switch.
if (x == 1) cout << "uno";
else if (x == 2) cout << "dos";
else if (x == 3) cout << "tres";
else if (x == 4) cout << "cuatro";
else if (x == 5) cout << "cinco";
else cout << "Opción no válida";

// Ejemplo utilizando switch.
switch (x) {
    case 1: cout << "uno";
    case 2: cout << "dos";
    case 3: cout << "tres";
    case 4: cout << "cuatro";
    case 5: cout << "cinco";
    default: cout << "Opción no válida";
}
```

Algoritmo 1.2 Ejemplos if - else

```
// Ejemplo1.
if (x > 5)
    y = x + 2;
else
    y = x * x;

// Ejemplo2.
if (x > 5)
    y = x + 2;

// Ejemplo3.
if (x > 5) {
    y = x + 2;
    y = x * x;
}
else {
    y = x + 5;
}

// Ejemplo4.
if ((x > 5) && (y < 3)) {
    y = x + 2;
    x = x * x;
}
else {
    y = x + 5;
}

// Ejemplo5.
if (x > 5) {
    if (y < 4) // x es mayor a 5 e y menor a 4
        y = x * x;
    else // x es mayor a 5 e y no es menor a 4
        y = x - 2;
}
else // x no es mayor a 5, el valor de y no interesa {
    y = x + 5;
}
```

Sin embargo, si por ejemplo x valiera 3, la porción del código anterior arrojaría lo siguiente:

```
tres
cuatro
cinco
Opción no válida
```

¿Por qué? Porque el `case` determina el punto de entrada pero no de salida.

Por lo que deberíamos modificar nuestro programa de la siguiente forma:

```
// Ejemplo utilizando switch (corregido).
switch (x) {
    case 1: cout << "uno";   break;
    case 2: cout << "dos";   break;
    case 3: cout << "tres";  break;
    case 4: cout << "cuatro"; break;
    case 5: cout << "cinco";  break;
    default: cout << "Opción no válida";
}
```

La sentencia *break* que se detallará más adelante, hace una ruptura (pega un salto) hacia fuera del bloque.

En ciertos casos está la necesidad de ejecutar cierta sentencia con varios valores de una variable y no con uno solo. Por ejemplo, si se desea imprimir “malo”, “regular”, “bueno”, “distinguido”, “sobresaliente”, según cierta nota, podríamos escribir:

```
switch (nota) {
    case 1:
    case 2:
    case 3: cout << "malo"; break;
    case 4:
    case 5: cout << "regular"; break;
    case 6:
    case 7: cout << "bueno"; break;
    case 8:
    case 9: cout << "distinguido"; break;
    case 10: cout << "sobresaliente"; break;
    default: cout << "nota inválida";
}
```

En el ejemplo anterior imprime “malo” en el caso de que la nota fuera 1, 2 o 3. Regular con 4 y 5, etc.

El `switch` se puede utilizar también con variables de tipo *char*, en ese caso, los valores hay que encerrarlos entre comillas simples:

```
switch (letra) {
    case 'A':
    case 'E':
    case 'I':
```

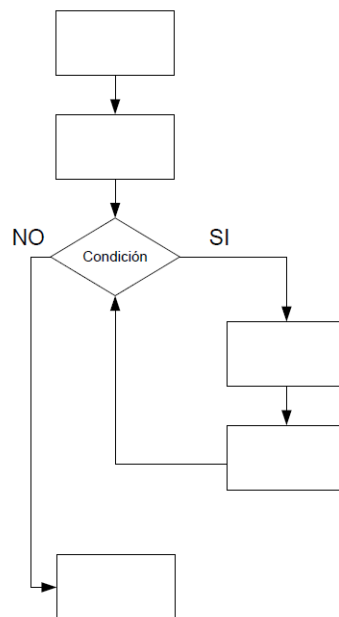


Gráfico 1.3.3: Diagrama de estructura repetitiva

```

case '0':
case 'U': cout << "vocal"; break;
default: cout << "consonante";
}

```

Sentencias iterativas o repetitivas

Muchas veces necesitaremos ejecutar una sentencia de manera repetida. Por ejemplo, imprimir un registro hasta que no haya más registros en el archivo o listar todos los números primos desde 2 hasta cierto valor N. Para resolver estos problemas debemos utilizar estructuras de tipo repetitivo o, también llamado, iterativo. En el gráfico 1.3.3 se ve el diagrama de flujo de una estructura repetitiva.

while Es idéntico al uso del while en C. La estructura es:

```

while (condicion)
    sentencia

```

Es decir, mientras la condición sea verdadera se ejecuta la o las sentencias que hubieran en el bloque. Por ejemplo:

```

while (i < 10) {
    i++;
}

```

```

        cout << i << endl;
    }

```

do – while Funciona de la misma manera que el while, solo que la condición se verifica al final. Ejemplo:

```

do {
    cout << x << endl;
    x--;
} while (x > 0);

```

La diferencia con el while es que el bloque se ejecuta por lo menos una vez. Por ejemplo, si x valiera -3 el bloque se ejecutaría igual porque la verificación de la condición es al final.

for Esta sentencia tiene múltiples usos, al igual que en C. Los códigos podrían llegar a ser poco legibles dependiendo de la forma en que lo utilicemos. La estructura es la siguiente:

```

for (sentencia_inicio; condicion; sentencias)
    otras_sentencias

```

La sentencia `_inicio` se ejecuta solo una vez al principio, luego verifica la condición, en caso de que la condición sea verdadera ejecuta las `otras _sentencias` y, luego, ejecuta las sentencias que figuran como último parámetro. Tanto las sentencias de inicio como las del tercer parámetro podrían ser varias, en este caso se deberían separar mediante una coma. Pero, también, podrían estar vacías al igual que la condición (una condición vacía genera un bucle infinito, a menos que haya alguna ruptura con *break* o *return*). Con ejemplos varios se entenderá mejor.

```

// Imprimir los números del 1 al 9
// Ejemplo1.
for (i = 1; i < 10; i++)
    cout << i << endl;

// Ejemplo2.
// Se ejecutan dos sentencias en el último parámetro
// El cuerpo del for es vacío
for (i = 1; i < 10; cout << i << endl, i++);

// Ejemplo3.
// El tercer parámetro está vacío
for (i = 1; i < 10; ) {
    cout << i << endl;
    i++;
}

```



```
// Ejemplo4.  
// El primer parámetro está vacío  
i = 1;  
for ( ; i < 10; i++ ) {  
    cout << i << endl;  
}
```

Por supuesto que, si bien los ejemplos 2, 3 y 4 funcionan correctamente, no son recomendables, ya que el ejemplo 1 es más claro.

Sentencias de ruptura

Son sentencias que cortan el hilo normal de ejecución, saliendo de los diferentes bloques.

break Sirve para salir de un bloque cualquiera. Por ejemplo:

```
for ( int i = 0 ; i < 10; i++ ) {  
    cout << i << endl;  
    if ( i == 4 ) break;  
}
```

El código anterior empezará listando los números 0, 1, 2, 3 y, luego de listar el número 4 saldrá del bloque y continuará con la sentencia que haya fuera de él.



Nota: generalmente se lo utiliza para salir de un bloque *switch*.
No se recomienda utilizar esta sentencia en otros casos.

continue La sentencia *continue* se utiliza dentro de un bucle *for*, *while* o *do-while* y sirve para transferir el control al final del cuerpo del bucle. Por ejemplo:

```
for ( int i = 0 ; i < 10; i++ ) {  
    if ( i == 4 ) continue;  
    cout << i << endl;  
}
```

El código anterior no imprime el valor 4, ya que transfiere el control al final del bucle y continúa con el siguiente valor. Aconsejamos no utilizarlo.

return El *return* realiza varias cosas:

- Finaliza la ejecución de una función.
- Devuelve el control a la función que la invocó.

- Como vimos en la sección de funciones, si la función tiene algún tipo de retorno, el *return* debe ir acompañado por alguna expresión (valor, nombre de una variable o alguna expresión más compleja) que sea de un tipo compatible al tipo de retorno.

1.3.12. Espacio de nombres

En los grandes proyectos, realizados por varios desarrolladores, es común que el nombre de una variable o función se repita. Por ejemplo, la sección de RRHH de una empresa tendrá la necesidad de manejar una tabla con los datos de sus empleados en donde tendrán sus direcciones, fechas de nacimientos, etc. El nombre natural para asignarle a la tabla será *empleados*. Pero en la oficina de Sueldos también tendrán que manejar otros datos de estos empleados, como horas trabajadas, sueldo básico, etc. Es lógico que también llamen *empleados* a esa tabla. Si estos módulos están al mismo nivel, nos dará un error indicando que esa variable ya existe.

El lenguaje C no tiene forma de solucionar esto más que cambiando los nombres de las variables. C++ nos proporciona los espacios de nombres (namespace) para resolver este problema. Dentro de un espacio de nombre las variables no pueden repetirse pero sí lo pueden hacer en distintos espacios de nombres. En el ejemplo anterior tendríamos dos espacios de nombres: uno para RRHH y otro para Sueldos. De esta manera, cuando uno se refiera a la variable *empleados* deberá indicar si es la variable que corresponde a RRHH o a Sueldos.

Esto nos sucede con el objeto *cout* que corresponde al espacio de nombres *std* (estándar). Por eso debíamos incluir la sentencia

```
using namespace std;
```

De lo contrario hay que indicar *std::* antes de cada *cout* y *endl*.

¿Cómo creamos un espacio de nombres? Simplemente escribiendo *namespace nombre_del_espacio* y entre llaves irán las declaraciones deseadas. Por ejemplo:

```
namespace Uno {
    int x;
    ...
}

namespace Dos {
    int x;
    ...
}

// Uso de los espacios de nombres
Uno::x = 5; // Se utiliza la variable x del espacio Uno
```

```
Dos::x = 8; // Se utiliza la variable x del espacio Dos
```

Cuando veamos objetos, cada clase representará un espacio de nombres de manera implícita.

1.4. Módulos

Cuando los proyectos empiezan a crecer y dejan de ser algo que se resuelve en no más de 10 líneas de código es conveniente y una necesidad modularizar. La modularización en primer lugar consiste en definir algunas funciones que resuelvan las distintas tareas, las cuales serán llamadas desde la función principal (*main*).

1.4.1. Declaración y definición

Debemos distinguir entre declaración y definición de una función.

- Declaración. Consta de la cabecera o firma de la función, que es la primera línea con el tipo de devolución, el nombre de la función y el tipo de sus parámetros. Si la definición se hace aparte, hay que cerrarla con un punto y coma. Por ejemplo

```
int sumar (int, int);
```

Nótese que no hace falta poner el nombre de los parámetros, aunque en general se recomienda hacerlo ya que esclarece el código.

- Definición. Consiste en el código completo de la función, incluyendo la cabecera.

Para que la función *main* pueda llamar a las demás funciones deben estar declaradas antes. Si bien pueden también definirse alcanza solo con la declaración.

1.4.2. División de un proyecto en módulos

A medida que los proyectos se van haciendo más complejos se empiezan a acumular muchas declaraciones antes del *main* en conjunto con las definiciones luego del *main*. El crecimiento no es solo en cantidad de funciones sino en niveles. Es decir, el proyecto deja de ser un *main* que llama a 4 o 5 funciones simples, porque estas a su vez llamarán a otras, y esto seguirá expandiéndose según la necesidad del proyecto. Por otra parte, las funciones empiezan a relacionarse según determinadas características, por ejemplo, en un proyecto que toma datos, hace cálculos y luego en base a estos cálculos

emite reportes, tendremos funciones encargadas de tomar los datos de entrada y darles el formato necesario. Otras se encargarán de hacer cálculos, finalmente tendremos el grupo de funciones que harán los reportes. En este esquema es conveniente dividir el proyecto en tres módulos además del principal.

En general, cada módulo constará de

- Un archivo de cabecera o archivo .h (header). En este archivo se pondrán todas las cabeceras de las funciones, y si existiera la definición de alguna constante o tipo de datos, pero no debería tener ninguna definición.
- Un archivo de implementación o definición de las funciones declaradas en el archivo .h. Este archivo será de extensión .cpp y deberá incluir el archivo de cabecera con la directiva `include` y el nombre del archivo entre comillas. Por ejemplo:
`#include "reportes.h"`

Los nombres de estos archivos podrían ser cualquiera pero se recomienda que el archivo de implementación se llame igual que el de cabecera, obviamente cambiando su extensión de .h a .cpp. Algo muy importante a tener en cuenta es que las inclusiones son solo de los archivos .h, los archivos .cpp no se deben incluir.

Ejercicios

Ejercicio 1.1.

Capítulo 2

Memoria dinámica

2.1. División de la memoria

A la memoria del ordenador la podemos considerar como una tira de celdas dividida en 4 segmentos lógicos que pueden variar en tamaño. Estos segmentos son los que vemos en la figura 2.1.1. Veamos cada uno:

- El Code Segment, o segmento de código, es donde se localiza el código resultante de compilar nuestra aplicación. Es decir, la algoritmia en código máquina.
- El Data Segment, o segmento de datos, almacena el contenido de las variables definidas como externas (variables globales) y las estáticas .
- El Stack Segment, o pila, almacena el contenido de las variables locales en la invocación de cada función, incluyendo las de la función main.
- El Extra Segment, o heap, es la zona de la memoria dinámica.

Cada celda corresponde a un byte, la unidad mínima de memoria. Recordemos que un byte es la conformación de 8 bits. En cada bit podemos almacenar dos intensidades distintas de corriente, en donde la intensidad más alta se interpreta como el valor 1, y la más baja como el valor 0. Por este motivo, en cada byte se pueden almacenar hasta $2^8 = 256$ valores diferentes. Cada celda se identifica con un número que llamamos *dirección*, este número es correlativo. En general se representa en formato hexadecimal por ser múltiplo de 8, quedando como en el gráfico 2.1.2.

Cada variable ocupa una o más de estas celdas, dependiendo del tipo de variable y la plataforma de la máquina. Por ejemplo, si una variable de tipo *int* ocupa 4 bytes, se reservan cuatro celdas contiguas, siendo la dirección

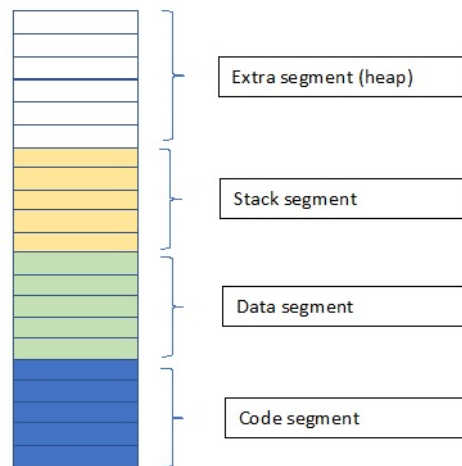


Gráfico 2.1.1: División lógica de la memoria



Gráfico 2.1.2: Direcciones de memoria

de la primera celda la dirección correspondiente a esa variable. Una variable de tipo *bool* ocupa un byte entero, aunque con un bit alcanza para guardar todos sus valores (0 o 1). Los otros 7 bits los rellena con ceros. Si queremos saber en qué dirección está determinada variable debemos usar el operador `&`, pero para imprimir su valor a veces, dependiendo de la plataforma, debemos indicar que lo tome como un entero sin signo. Por ejemplo:

```
int a = 8;
cout << (unsigned)&a << endl;
```

En este código, la variable entera *a* almacena el valor 8, y se encuentra en determinada dirección. Esta dirección se mantendrá fija durante todo el tiempo de vida de la variable, es decir, hasta que encuentre una llave de cierre de bloque, pero puede variar en las distintas ejecuciones del programa. En general, las direcciones en dónde están las variables no nos interesará conocerlas, salvo a modo ilustrativo.

De la gestión de los tres primeros bloques de memoria: el segmento de código, el de datos y el de pila, se encarga el compilador, quien reserva y libera los bloques de memoria de manera automática.

La gestión del bloque identificado como heap será responsabilidad del desarrollador. Es decir, los bloques de memoria que se soliciten al heap deben hacerse con una determinada instrucción en el programa, y cuando la variable no se necesite más debemos liberar la memoria ocupada con otra instrucción, en caso de no hacerlo esa porción de memoria queda inutilizada hasta que apaguemos la máquina.



Nota: debemos ser cuidadosos con el uso de la memoria, ya que si no la liberamos no se recupera ni siquiera cuando finaliza el programa.

2.2. Punteros

Un puntero es un tipo especial de variable que almacena direcciones de memoria. Es decir, en lugar de guardar datos como una edad, un sueldo, una nota, etc, almacena direcciones de memoria de la propia máquina. Cuando una variable de tipo puntero guarda una dirección de memoria decimos que *apunta* a esa dirección.

El lector puede preguntarse cuál es la necesidad de guardar esta información. Esta pregunta será respondida más adelante, primero veamos cómo se definen y cómo es el funcionamiento.

Punteros a datos simples

Un puntero tiene que conocer, salvo alguna excepción que veremos pronto, de qué tipo será la variable que esté en la dirección de memoria que almacenará su valor. Dicho de otra forma, qué tipo de dato está guardado o se guardará en la dirección adonde apunta. Por ejemplo, si un puntero guarda la dirección de memoria *AB0012*, debe saber qué tipo de dato se almacenará en dicha dirección, si será un *int*, un *float* o cualquier otro. Esto se debe a dos razones: en primer lugar debe saber qué cantidad de celdas ocupará el dato que está en dicha dirección. En segundo lugar debe saber cómo interpretar esos datos. Tenemos que recordar que cada celda está formada por bits pero esos bits pueden ser interpretados de distinta manera, por ejemplo, como enteros, como flotantes, si representan un carácter en código ASCII, etc. Por lo tanto, cuando definimos una variable de tipo entero debemos decir a qué tipo de variable apuntará.

Para llevar a cabo su función de almacenamiento debe ocupar el tamaño que ocupa una dirección, es decir una palabra de la arquitectura de la máquina. En el caso de procesadores de 16 bits, son 2 bytes, para los de 32, 4 bytes y, para los de 64, 8 bytes.

¿Cómo se declara una variable de tipo puntero? Al igual que cualquier otra variable, indicando su tipo y un identificador de la variable, con la salvedad que luego de indicar el tipo se debe colocar el signo asterisco “*” que indica que es un puntero. Ejemplos:

```
int * pi;
char * pc;
float * pf;
```

El símbolo asterisco puede ir inmediatamente después del tipo

```
int* pi;
```

inmediatamente antes del identificador de la variable

```
int *pi;
```

o como en los ejemplos indicados que no está contiguo a ninguna otra expresión. Hay que tener ciertos cuidados, por ejemplo:

```
int* pi, pi2, pi3;
```

no declara tres variables de tipo punteros a enteros, sino solo una, la primera. Tanto *pi2* como *pi3* son variables de tipo enteras.

Ahora bien, ¿cómo asignamos una dirección de memoria a una variable de tipo puntero? Las direcciones de memoria no podemos asignarlas de forma explícita, es decir, no podemos escribir algo de este estilo:



```
int * p = 11206656;
```

Suponiendo que 11206656 es una dirección de memoria válida y que se encuentra disponible, el código anterior es erróneo. Sí podemos asignar la

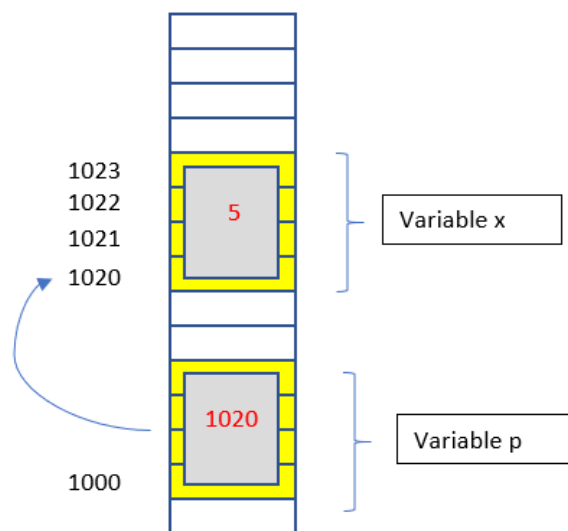


Gráfico 2.2.1: Ejemplo del contenido de una variable de tipo puntero

dirección de otra variable utilizando el operador “&”. Ejemplo:

```
int * p;  
int x = 5;  
p = &x;
```

Supongamos que la variable p está ubicada en la dirección 1000 y la variable x en la 1020. Entonces, la variable p contendrá como dato el valor 1020. Si un entero ocupa 4 bytes, la memoria quedará como vemos en el gráfico 2.2.1. La flecha indica la imagen gráfica del sentido de apuntar, de ahí el nombre de este tipo de variables.

Siguiendo con este ejemplo, si quisiéramos modificar el valor de x a, por ejemplo, 8 podríamos hacerlo indirectamente de la siguiente forma:

```
*p = 8;
```

La sentencia anterior equivale a

```
x = 8;
```

Hay que tener cuidado porque el operador $*$ tiene distinta funcionalidad según el caso, decimos que está *sobrecargado*. En un caso sirve para declarar una variable de tipo puntero. Es decir, su significado es decir que la variable que sigue es un puntero. En cambio, cuando la variable ya está declarada se llama operador de desreferenciación. En este caso el significado es el siguiente “el contenido de la dirección a la que apuntas” o “dirígete a la dirección a la que apuntas”.

Observen un momento el siguiente código y piensen qué hace cada instrucción y si son válidas, antes de ver los comentarios.

```
1 int main() {  
2     int x = 5, y = 8;  
3     int *p1;  
4     int *p2;  
5     p1 = x;  
6     p1 = &x;  
7     p2 = &y;  
8     cout << p1 << endl;  
9     p1 = p2;  
10    *p1 = *p2;  
11    return 0;  
12 }
```

En la línea 2 definimos dos variables de tipo enteras, x e y , asignándoles los valores de 5 y 8, respectivamente. Luego, en la línea 3 y 4 declaramos dos variables de tipo puntero a entero.

La línea 5 da error al momento de compilar ya que $p1$ espera una dirección, no el valor de x que contiene 5. También sería inválido hacer

```
*p1 = x;
```

dado que $p1$ aún no apunta a ningún lado.

Las líneas 6 y 7 son válidas: $p1$ pasa a tener la dirección de x o simplemente decimos que $p1$ apunta a x , y $p2$ apunta a y . La línea 8 podría dar error dependiendo de cómo tengamos configurado nuestro compilador, dado que intenta imprimir el valor que tiene $p1$, y este valor es una dirección. Puntualmente, la dirección de x . En caso de error podríamos castearlo a una variable de tipo entera sin signo (unsigned). Sin embargo, salvo por la finalidad de investigar cuáles son las direcciones de las variables no tendremos el deseo o la necesidad de imprimir una dirección. Sí nos interesará imprimir el contenido que hay en esa dirección. Es decir, si reemplazamos esa línea por $*p1$ imprime el valor 5.

Las líneas 9 y 10 son ambas correctas pero pueden llegar a confundirnos. En la línea 9 estamos diciendo que $p1$ apunta al mismo lugar que $p2$, es decir, a la variable y . De esta forma, $p1$ dejará de tener la dirección de x para pasar a tener la de y . En cambio, en la línea 10 lo que estamos igualando son los contenidos. Supongamos que la línea 9 no se ejecutó, por lo que $p1$ conserva la dirección de x y $p2$ la de y . De esta manera, cada puntero seguiría apuntando al mismo lugar pero se cambiaría el contenido de la variable x por un 8.

2.3. Punteros y vectores

Cuando declaramos un vector el compilador reserva la memoria solicitada y guarda solo la dirección del primer elemento. Por ejemplo, si declaramos un vector de 10 enteros

```
int vec[10];
```

el compilador reserva 10 lugares contiguos para almacenar los enteros. De estos 10 lugares solo guarda la dirección del primero en la variable que llamamos *vec*. Luego, es responsabilidad del programador no excederse de los 10 lugares pedidos. Supongamos que en dicho vector guardamos los números 100, 200, 300, etc. La siguiente línea

```
cout << vec[5] << endl;
```

imprime 600, la sexta posición del vector (hay que recordar que la primera posición es la 0). En cambio, la siguiente instrucción

```
cout << vec << endl;
```

debería imprimir (si el compilador no se queja) la dirección en donde está el número 100, ya que es el primer dato. Por lo tanto, como en la variable *vec* estamos guardando una dirección la podemos pensar como un puntero. Esto es cierto si lo pensamos como un puntero constante, ya que una variable de tipo puntero podemos cambiarle su contenido, lo cual la haría apuntar a otro lugar. En cambio, a una variable de tipo vector, no. Con esto en mente, la siguiente instrucción es válida:

```
int * p = vec;
```

O también como parámetro de una función:

```
void cargar (int * p, int n);
```

Entonces, si manejamos un vector a través de punteros, ¿cómo accedemos a sus posiciones? La primera respuesta a esto es mantener la lógica que vimos de punteros con el operador de desreferenciación:

```
*p // accede a la primera posición  
*(p+1) // accede a la segunda posición  
// etc
```

Sin embargo, tenemos otra forma más natural de acceder que es utilizando los corchetes con índices como con cualquier variable de tipo vector:

```
p[0] // accede a la primera posición  
p[1] // accede a la segunda posición  
// etc
```

Por lo visto, los vectores y los punteros no presentan diferencias, salvo que los primeros son constantes. Cuidado: los valores de los vectores pueden modificarse pero siempre en la misma porción de memoria.

El caso de las matrices lo veremos un poco más adelante, dado que es más complejo: hay que pensarlas como un vector de punteros, pero el primer vector también lo podemos pensar como un puntero. Todo esto implica un doble puntero que veremos en las otras secciones.

2.4. Pedidos de memoria y liberación

Lo que vimos hasta el momento sobre las variables de tipo puntero no tendría sentido si no tuviéramos la necesidad de utilizar memoria dinámica. Es decir, memoria que se solicita por el programador en el momento de ejecución. Esta zona de la memoria es la que llamamos *heap*.

¿Cómo solicitar memoria al heap?

Mediante la instrucción *new tipo*. Por ejemplo

```
new int;
```

crea una nueva variable de tipo `int` en la zona del heap. Sin embargo, esta variable anónima la perderíamos si no tuviéramos la dirección en donde fue creada. El operador *new* hace algo más que crear la variable solicitada: devuelve su dirección, la cual debemos guardarla en una variable adecuada. Una variable adecuada para guardar direcciones es un puntero, por lo tanto, la instrucción correcta es:

```
int * p = new int;
```

De esta forma *p* guarda la dirección de esa variable anónima y puede acceder mediante el operador *** para guardar / recuperar valores.

Cuando trabajemos con memoria dinámica debemos tener cuidado porque la memoria solicitada hay que liberarla. C++ no cuenta con un recolector de basura como el lenguaje Java que libera la memoria automáticamente, por lo que la memoria que no se libere quedará inhabilitada para su uso, aún cuando el programa hubiera finalizado.

La instrucción para liberar la memoria es con el operador *delete*, de la siguiente forma:

```
delete p;
```

Solicitud y liberación de un bloque de memoria

La potencia de los punteros se aprecia en las estructuras dinámicas. Si quisiéramos crear un vector de enteros en forma dinámica podemos solicitar todo un bloque al *heap*, de la siguiente forma:

```
int * p = new int [100];
```

En la instrucción anterior estamos creando un bloque contiguo de 100 enteros del cual guardamos la dirección del primer entero en la variable *p*. Ya vimos cómo podemos acceder a cada uno de esos enteros, trabajando con la variable *p* como si fuera un vector común y corriente. La única consideración es que al momento de liberar debemos indicarle al compilador que debe liberar todo el bloque colocando corchetes vacíos luego del operador *delete*, de esta forma:

```
delete [] p;
```

No debemos indicarle cuál es el tamaño del bloque a liberar ya que el mismo compilador se encarga de liberar todo el bloque solicitado.

2.5. Punteros a estructuras complejas

Supongamos que tenemos un struct Empleado:

```
struct Empleado {
    int legajo;
    string nombre;
    float sueldo;
};
```

Si queremos crear una variable de tipo Empleado en forma dinámica utilizaremos el operador *new* como en los demás casos:

```
Empleado * pe = new Empleado;
```

Para acceder a alguno de los campos deberíamos usar el operador de desreferenciación * sumado al punto (.). Por ejemplo:

```
(* pe).legajo = 145;
```

La línea anterior se interpreta:

* *pe* es la variable a la cual apunta *pe*, en este ejemplo es una variable de tipo *Empleado*. Luego

(.) accede al campo deseado.

Hay otra manera que se utiliza más: consiste en emplear el operador flecha

-> para acceder a los campos, de esta manera:

```
pe->legajo = 145;
```

Para liberar la memoria es igual que con cualquier variable simple:

```
delete pe;
```

2.6. Doble punteros

Los punteros dobles, triples, etc. los usaremos cuando necesitemos direcciones de direcciones. El caso más común se da con las matrices dinámicas. Una matriz dinámica la podemos pensar de la siguiente forma: cada fila es un vector dinámico. Por lo tanto, la matriz entera es un vector de vectores dinámicos. En el gráfico 2.6.1 vemos en la parte *a*) una matriz de 4 filas por 5 columnas. En la parte *b*) se representa la idea de la misma matriz pensada como un vector de vectores. En cada posición del vector tenemos un puntero simple, en cambio, en la primera posición de la estructura necesitamos un puntero de punteros, representado como *pp*.

El código para armar una matriz de esta forma es el que apreciamos en el algoritmo 2.1. En primer lugar se crea el vector de punteros que tendrá la

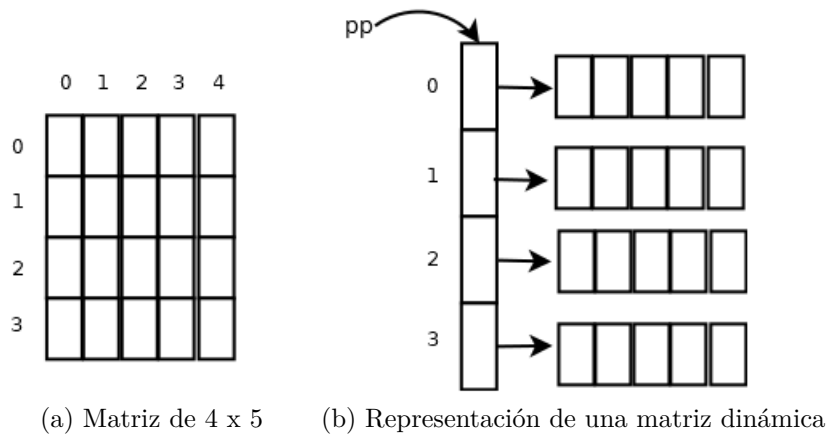


Gráfico 2.6.1: Matrices dinámicas

Algoritmo 2.1 Creación de una matriz dinámica

```
int ** matriz;
matriz = new int* [4]; // se crea el vector de vectores
for (int i = 0; i < 4; i++) // para cada una de las filas
    matriz[i] = new int[5]; // se crea el vector fila
```

longitud de las filas de la matriz. Esta dirección se guarda en el doble puntero *matriz*. Luego se accede a cada una de las posiciones del vector y se crea el vector de enteros que representará cada una de las filas.

Para la eliminación el proceso es inverso:

- Se eliminan cada una de las filas.
- Finalmente se elimina el vector que sostiene la estructura.

El código de liberación es el siguiente:

```
for (int i = 0; i < 4; i++) // para cada una de las filas
    delete [] matriz[i];    // se elimina la fila
delete [] matriz;           // se elimina el vector de vectores
```

La versatilidad de manejar las matrices de esta manera es que podemos crear matrices irregulares, en donde las filas tienen distinta longitud, como apreciamos en el gráfico 2.6.2.

2.7. Operaciones con punteros

Las operaciones que podemos hacer con los punteros están acotadas.

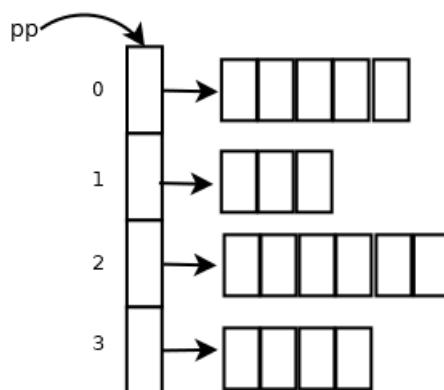


Gráfico 2.6.2: Matrices irregulares

- **Asignación.** Vimos que podemos asignarle direcciones de variables de dos formas:

- Si la variable no es dinámica con el operador `&`. Ejemplo

```
int x = 5;
int * p = &x;
```

Esta forma de asignación no es muy utilizada, salvo en la didáctica.

- Si la variable es dinámica se crea con el operador `new`. Ejemplo:

```
int * p = new int;
```

Esta forma es más utilizada, en especial cuando se crean bloques de memoria.

- Hay un solo valor válido que se puede asignar a un puntero que no sea la dirección de otra variable: el valor nulo. Se realiza poniendo un cero a la variable o un `NULL`:

```
int * p = 0;
int * p = NULL;
```

Las asignaciones anteriores son equivalentes pero se prefiere colocar un 0 porque la constante `NULL` no siempre está definida y puede dar error.

¿Para qué utilizar esta dirección nula? Se utiliza para saber si el puntero apunta a cierta variable o no.

- **Suma y resta.**

- La suma de un valor a un puntero hace que este puntero avance a las posiciones contiguas de memoria. Por ejemplo:

```
int * p = new int[100]; // p apunta al primer lugar del vector
p++; // p apunta al segundo lugar del vector
```

- Con la resta sucede al contrario que con la suma: el puntero retrocede. Hay que tener en cuenta que para liberar la memoria el puntero debe apuntar a la primera dirección.

Ejemplo. Utilización del valor nulo. En C++ al igual que en C el valor nulo representa un falso a la hora de evaluar una condición lógica y cualquier valor distinto de cero lo interpreta como cierto, por lo que podemos hacer lo siguiente:

```
if (p) // equivale a p != 0
    // p apunta a cierta dirección válida
    // realizar A
else   // p no apunta a ninguna dirección
    // realizar B
```

2.8. Persistencia de datos

En el capítulo anterior habíamos visto que las variables nacen cuando se las declara y mueren cuando finaliza el bloque en donde fueron declaradas. Por ejemplo:

```
int main() {
    int x = 5; // nace x
    if (x != 10) {
        int y; // nace y
        //...
    }          // muere la variable y
    //...
    return 0;
}              // muere la variable x
```

Sin embargo, con el manejo de la memoria dinámica a través de los punteros podemos tener persistencia de datos. Esto significa que las variables pueden trascender el bloque en donde fueron definidas. Por ejemplo, una función que pide memoria dinámica y devuelve un puntero:

```
int* f() {
    int* p; // nace p
    p = new int; // nace una variable anónima, su dirección la guarda p
    //...
    return p;
}          // la variable p muere pero la anónima sigue viviendo
```

En estos casos hay que tener mucho cuidado ya que es fácil olvidarse de liberar la memoria porque la función que la pide no se encarga de hacerlo. Debería encargarse de su liberación la función que llama a f o alguna otra.

Otra variante de hacer extender el ámbito de vida de una variable es a través de un puntero pasado por referencia. El siguiente código es equivalente al anterior:

```
void f(int* &p) {  
    p = new int;    // nace una variable anónima, su dirección la guarda p  
    //...  
}
```

Ejercicios

Capítulo 3

Programación Orientada a Objetos

3.1. Paradigmas de programación

Haremos un breve repaso por los principales paradigmas de la programación según su avance cronológico.

3.1.1. Programación no estructurada

En un principio, los programadores no conservaban ningún estilo, cada uno programaba a su gusto personal sin un criterio unificado. Algunos lenguajes, como el Basic, permitían “saltos” de una instrucción a otra, por medio de sentencias como *goto* y *exit*. Estas características hacían que los programas fueran muy difíciles de corregir y mantener. Si se detectaba algún error en la ejecución de un programa se debía revisar el código de forma completa para poder determinar dónde estaba el problema.

Por otro lado, con respecto al mantenimiento, actualizar un programa, por ejemplo, por nuevas normativas impositivas o agregarle nueva funcionalidad para obtener algún reporte que hasta el momento no se generaba; era muy complejo. Tomar un programa que, probablemente, había escrito otra persona y tratar de entender y seguir el hilo de la ejecución era una tarea prácticamente imposible de realizar.

3.1.2. Programación estructurada

A partir de la década del 70 se comenzó a implementar la programación estructurada, basada en el Teorema de Bohm y Jacopini, del año 1966, en

donde se determinaba que cualquier algoritmo podía ser resuelto mediante tres estructuras básicas:

- Secuenciales
- Condicionales
- Repetitivas

Estructuras que se vieron en materias anteriores. En la programación estructurada se prohíbe el uso del *goto* y otras sentencias de ruptura con la finalidad de lograr un código más legible y encontrar rápidamente los errores sin necesidad de estar revisando el código por completo.

La idea de la programación estructurada es resolver los problemas de forma *top-down*, es decir, de arriba hacia abajo, con el objetivo de ir particionando un problema complejo en varios más simples.

El programa principal estará formado por la llamada a las subrutinas (funciones) más importantes, las cuales, a su vez, llamarán a otras subrutinas y, así sucesivamente, hasta que cada subrutina solo se dedique a realizar una tarea sencilla. Por ejemplo, si necesitáramos tomar datos de un archivo A con los que se cargaría una tabla, luego mostrar los resultados, a continuación tomar datos desde el teclado y actualizar el archivo A; nuestro programa principal podría tener la siguiente forma:

```
// Programa en pseudocódigo
Principal
Comienzo
    abrir_archivo(A);
    cargar_tabla(A, tabla, n); // n: tamaño de tabla
    mostrar_resultado(tabla, n);
    tomar_datos(tabla, n);
    actualizar_archivo(A, tabla, n);
Fin.
```

Hay que tener en cuenta que primero se arma el esqueleto del programa y, luego, se va rellenando. Es decir, se van escribiendo las funciones que hagan falta.

3.1.3. Programación orientada a objetos

La necesidad de extender programas (agregar funcionalidad) y reutilizar código (no volver a escribir lo mismo dos o más veces), hizo que el paradigma de la programación estructurada evolucionara hacia un nuevo paradigma: la Programación Orientada a Objetos.

Este paradigma intenta modelar el mundo real en el cual nos encontramos con toda clase de objetos que se comunican entre sí y reaccionan ante determinados estímulos o mensajes. Los objetos están conformados por propiedades o atributos, incluso, estos atributos pueden ser, a su vez, otros objetos. Y tienen cierto comportamiento, es decir, ante un determinado estímulo reaccionan de una determinada manera. Además, en un momento preciso tienen un estado determinado que está definido por el valor de sus atributos. Por ejemplo, el objeto semáforo tiene luces (sus atributos), que reaccionan ante el paso del tiempo. Cada cierta cantidad de segundos, estas luces cambian de estado: de prendido a apagado o viceversa. Es decir, en un determinado momento su estado será:

- Luz roja prendida.
- Luz amarilla apagada.
- Luz verde apagada.

Cuando pasen x segundos, recibirá un mensaje *prender_luz_amarilla*. En cambio, el objeto *auto* no reaccionará ante el paso del tiempo, sino que reaccionará ante los estímulos de la persona que lo maneje, como encender, acelerar, frenar, etc. El objeto *auto* tendrá como atributos otros objetos, como *radio*, *control_remoto*, *puerta*, etc.

El objeto *celular_1* recibirá una señal de otro objeto que es el objeto *antena* y, a través de éste, se comunicará con otro objeto *celular_2* que, seguramente, tendrá otras características: otro tamaño, marca, etc. Sin embargo, aunque estos celulares sean muy distintos, ambos pertenecerán a una misma clase: la clase *Celular* y se comunicarán mediante una interfaz común a ambos.

¿Cómo pensar en POO?

¿Cómo se deben encarar los problemas con este paradigma? En la programación estructurada se pensaba la solución de un problema de la forma *top – down*, es decir, de arriba hacia abajo, llamando a funciones que fueran dividiendo nuestro problema en uno más pequeño, hasta encontrarnos con problemas muy sencillos.

En la POO la forma de resolver un problema es inversa. En lugar de determinar qué funciones necesitamos se debe determinar qué objetos se necesitarán. Los problemas se irán resolviendo de abajo hacia arriba, desde pequeños objetos que actuarán entre sí y conformarán otros más complejos.

Por ejemplo, si quisiéramos cargar un vector y, luego, ordenarlo y listarlo. En Programación Estructurada pensaríamos en las siguientes funciones:

```
// Estructurado
cargar(vector, n); // Funcion que carga el vector
ordenar(vector, n); // Funcion que ordena al vector
mostrar(vector, n); // Funcion que muestra al vector
```

Estas funciones tendrían, como parámetros, el vector que deseamos cargar, ordenar y listar; y un valor entero n , su tamaño. Hay que notar que, por ejemplo, la función ordenar no trabajaría de la misma forma en un vector de enteros que de strings, por lo que deberíamos escribir una nueva función para las distintas clases de vectores. Este era uno de los inconvenientes que mencionamos en la Programación Estructurada: no hay reutilización del código, ya que estos procedimientos serían prácticamente los mismos, solo que cambiarían sus tipos de datos. Más adelante veremos programación genérica y polimorfismo, dos formas de solucionar este problema.

La misma situación, pero encarada en POO, sería de la siguiente manera: en primer lugar se debe establecer una clase *Vector* que, tendrá como datos, los valores que se desean almacenar y el tamaño del vector. Además, tendrá métodos para cargarlos, ordenarlos y mostrarlos:

```
// POO
Vector v; // Se crea un objeto de tipo Vector
v.cargar(); // Se llama al metodo cargar
v.ordenar(); // Se llama al metodo ordenar
v.mostrar(); // Se llama al metodo mostrar
```

En el código anterior vemos que no hay necesidad de pasar el vector por parámetro ya que es el propio objeto que llama a los métodos. Tampoco necesitamos pasar su tamaño como en la programación estructurada porque el propio vector sabe cuál es su tamaño.

3.2. Tipo Abstracto de Datos

El lector se encontrará familiarizado con la expresión *tipo de dato* ya que habrá escrito una serie de programas o aplicaciones en las que utilizó diferentes tipos de datos, como un entero para guardar una edad o una cadena de caracteres para almacenar un nombre. Sin embargo, el término *abstracción*, probablemente, le resulte nuevo en el ámbito de la informática. Quizá esa palabra le represente una imagen de algún cuadro moderno o alguna letra de Spinetta. También, podría recordarle alguna frase peyorativa de su maestra de primaria “estás abstraído”, refiriéndose a que no le prestaba atención. Estas imágenes no tienen que ver con la idea que se le quiere dar al término *abstracción* en informática.

El diccionario de la Real Academia Española indica:

- Abstracción. Acción y efecto de abstraer o abstraerse.
- Abstraer. Separar, por medio de una operación intelectual, las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción.

Si bien esto no nos aclara demasiado ni da un indicio de cómo se puede utilizar en el desarrollo de alguna aplicación, nos da la idea que debemos separar las cualidades de un objeto para quedarnos con su esencia.

En el cuento *Funes el memorioso*, Jorge Luis Borges describe a una persona (Funes) que recuerda todo lo que ha vivido con la mayor exactitud. El narrador dice en un párrafo “Sospecho, sin embargo, que no era muy capaz de pensar. Pensar es olvidar diferencias, es generalizar, abstraer.”^b Esta es la clave del término *abstracción*, si no podemos abstraernos no podremos pensar, por lo tanto, tampoco actuar.

Imaginemos, por un momento, que venimos de un planeta donde no existen perros ni gatos. Al llegar a la tierra nos ponemos a charlar con un terrícola y vemos pasar, por al lado nuestro, un Gran Danés. Al interrogar sobre qué clase de animal es, el terrícola nos contesta que es un perro. Un rato más tarde vemos un Caniche. No podemos imaginarnos que, también, es un perro. Porque su tamaño, el color, el pelaje es completamente distinto al anterior. Sin embargo, el terrícola insiste en que eso que estamos viendo y que emite unos ladridos agudos, no es un juguete sino un perro. A los pocos minutos cruza un gato por nuestro camino. Le preguntamos al terrícola si, también, es un perro, porque no tiene tanta diferencia con el Caniche. Sin embargo, el terrícola, riéndose, nos dice que es un gato, que no tiene nada que ver con un perro. Diariamente nos cruzamos con perros que jamás hemos visto. Sin embargo logramos reconocerlos. ¿Por qué? Porque de chicos preguntamos miles de veces a los mayores “¿qué es eso?” y nos contestaban que era un perro. Luego de ver decenas de ellos, encontramos, inconscientemente, una regla que nos indica “si lo que ves es así, así y así, entonces, es un perro”. Es decir, nos quedamos con la esencia de las cosas, las simplificamos, nos abstraemos.

Por otro lado, es importante que tengamos la capacidad de utilizar las cosas sin saber cómo están implementadas, sabiendo solo cómo reaccionarán ante determinado estímulo. Para comprender esto se darán algunos ejemplos. Cuando giramos la llave de contacto de un automóvil sabemos que se pondrá en marcha y, luego de colocar el cambio adecuado y acelerar, se pondrá en movimiento. Sin embargo, no necesitamos saber qué mecanismos internos se están accionando en cada instante ni qué piezas del motor se ponen en actividad. Por más que lo sepamos, no estaremos pensando en cada detalle de lo que está sucediendo “por debajo” de la interfaz. Por interfaz nos referimos

al sistema que se comunica con el usuario, en este caso, llave de contacto, palanca de cambios, volante, acelerador, etc. El usuario sería el conductor. Estar pendientes de cada pieza que actúa en el automóvil al momento de estar conduciéndolo, sólo logrará distraernos y entorpecer nuestro objetivo principal que es el de conducir hacia un determinado lugar. Un médico, cuando camina, no está pendiente de las órdenes que envía el cerebro a determinados músculos, a través del sistema nervioso, para que se pongan en movimiento y logre avanzar algunos pasos, aunque lo haya estudiado en alguna materia.

Con respecto a los tipos de datos abstractos (TDA), en informática, nos encontraremos principalmente con tres enfoques distintos:

- Diseñador. Como diseñador haremos uno o varios diseños del tipo de dato a construir. En el ejemplo de los automóviles haríamos planos y gráficos, y algún documento indicando cómo será el automóvil a fabricar, qué cosas tendrá, etc.
- Implementador. Como implementadores construiremos nuevos tipos de datos abstractos para que otros los utilicen (podríamos ser nosotros mismos). Siguiendo el ejemplo del automóvil, en este papel, seríamos la fábrica de automóviles. Como fábrica desearemos hacer automóviles fuertes, seguros, sencillos de utilizar. Por otro lado, no desearemos que los usuarios “metan mano” en los dispositivos contruidos, por lo que se tomarán los recaudos necesarios para que esto no suceda como fajas de seguridad, etc. Hay que tener en cuenta que la garantía de una máquina se pierde si se detectara que una persona no autorizada estuvo modificando la implementación.
- Usuario. Como usuarios utilizaremos los tipos de datos abstractos implementados por otros desarrolladores (o por nosotros mismos), sin importarnos cómo están implementados, solo interesándonos en cómo debemos comunicarnos con dichos elementos (interfaz), confiando en que el comportamiento será el asegurado por el implementador. En este caso seríamos los conductores del automóvil.

Entonces, ¿qué es un tipo de dato abstracto?

Es un mecanismo de descripción de alto nivel que, al implementarse, genera una clase. Es decir, un TDA es un concepto matemático, mientras que una clase es la implementación del TDA. Aunque, se verá más adelante, una clase puede ser abstracta en parte o completamente. Si fuera una clase totalmente abstracta diremos que es un TDA. Entonces, el TDA corresponde a la etapa de diseño.

Un TDA se define indicando las siguientes cosas:

- Nombre (tipo)
- Invariantes
- Operaciones

El nombre nos indica al tipo que nos referimos. Los invariantes están relacionados con la validez de los elementos que componen el TDA, es decir, qué valores son válidos para conformar un TDA. Las operaciones son las que se necesitan que realice el TDA, se indican con el nombre de la operación, el dominio y el codominio, además, se deben definir las PRE y POST condiciones. Con algunos ejemplos ilustraremos estas definiciones.

Ejemplo 3.1. TDA Entero.

- Nombre: Entero
- Invariante: Entero $\in \mathbb{Z}$
- Operaciones
 - $+$: entero x entero \rightarrow entero
 - $-$: entero x entero \rightarrow entero
 - $*$: entero x entero \rightarrow entero
 - $/$: entero x entero \rightarrow double

El nombre tiene que ser descriptivo. En invariante estamos indicando que los objetos de tipo Entero pertenecerán al conjunto infinito de los enteros. Recordamos que en una máquina no podemos representar un número infinito, ya que necesitaríamos una memoria infinita.

En cuanto a Operaciones, tomemos, por ejemplo, la operación $+$, significa que, haciendo el producto cartesiano entre dos enteros (por ese motivo la x), nos devuelve otro entero. En cambio, la operación $/$ nos devuelve un double. Según Meyer, esta operación se representa con una flecha tachada, porque no todos los enteros pertenecen al dominio de la operación (o función). Si el segundo parámetro es un cero esta operación no se puede realizar, eso se indica en la PRE condición. La POST condición nos dice qué resultado se obtiene siempre que las PRE condiciones se cumplan.

Ejemplo 3.2. TDA Cadena.

- Nombre: Cadena.

- Invariante: $Cadena = "c_1c_2 \dots c_n"$. Los c_i son caracteres pertenecientes al conjunto Θ . $\Theta = \{A..Z\} \cup \{a..z\} \cup \{0..9\} \cup \{, -, /, (,), ", \text{¡}, \text{!}, =, \text{¿}, ?\}$
- Operaciones
 - $cadena : \rightarrow cadena$
 - $+ : cadena\ x\ cadena \rightarrow cadena$
 - $reemplazar : cadena\ x\ posición\ x\ carácter \rightarrow cadena$
 - $cadlen : cadena \rightarrow entero$
 - $valor : cadena\ x\ posición \rightarrow carácter$

Las operaciones las podemos clasificar en tres grupos:

- a. Constructoras. Es el caso de la primera operación, la cual crea o construye una cadena vacía.
- b. Modificadoras. Alteran el estado del TDA. Es el caso de las operaciones que están en segundo y tercer lugar. La operación $+$ concatena la cadena a otra, generando una nueva cadena. La operación $reemplazar$, modifica la cadena, en la posición indicada, colocando el carácter que se le pasa por parámetro.
- c. Analizadoras o de consulta. No alteran el estado del TDA, sirven para consultas. Por ejemplo la operación $cadlen$ nos indica la longitud de la cadena y la operación $valor$ nos devuelve el carácter que se encuentra en la posición solicitada. Ninguna de estas dos operaciones hace alguna modificación.

Más adelante se verá que en C++ necesitamos, a menudo, definir operaciones destructoras. Son las encargadas de liberar recursos utilizados, generalmente es cuando se debe liberar memoria dinámica solicitada. Estas operaciones son modificadoras, ya que destruyen el TDA (cambian su estado).

3.3. Diseño de un TDA

En la etapa de diseño no hablamos ni pensamos en su implementación. Ni siquiera se debe tener en cuenta el lenguaje en que se implementará. El TDA debe ser independiente de su implementación. Por ejemplo, sabemos que si estamos trabajando con enteros ejecutar la operación $x + y$ debe dar el mismo resultado que realizar $y + x$, cualquiera sea su implementación y la plataforma donde se ejecute. Por este motivo, un TDA puede tener distintas implementaciones. Veremos esto con un ejemplo.

Ejemplo 3.3. TDA Complejo.

- Nombre: Complejo.
- Invariante: Complejo = (real, imaginario), con real, imaginario $\in R$.
- Operaciones
 - Complejo : $R \times R \rightarrow \text{Complejo}$
 - sumar : Complejo \times Complejo \rightarrow Complejo
 - restar : Complejo \times Complejo \rightarrow Complejo
 - modulo : Complejo $\rightarrow R$

Para no complicar el TDA se definieron solo estas cuatro operaciones. Por supuesto, las operaciones a definir dependen del uso que se pretenda dar al nuevo tipo de datos.

Detalle de las operaciones

- Operación: Complejo
 - Descripción: construye un número complejo en base a dos parámetros, el primero conformará la parte real y el segundo la imaginaria.
 - Pre condición: los dos parámetros deben ser números reales.
 - Post condición: construye un número complejo. La parte real tendrá el valor del primer parámetro, la parte imaginaria el del segundo.
- Operación: sumar
 - Descripción: suma dos números complejos y devuelve el resultado.
 - Pre condición: los dos parámetros deben ser números complejos válidos.
 - Post condición: devuelve un número complejo tal que su parte real tendrá el valor de la suma de las partes reales de los dos parámetros y la parte imaginaria el valor de la suma de las partes imaginarias de los dos parámetros.
- Operación: restar
 - Descripción: resta dos números complejos y devuelve el resultado.

- Pre condición: los dos parámetros deben ser números complejos válidos.
 - Post condición: devuelve un número complejo. La parte real tendrá el valor de la diferencia entre la parte real del primer parámetro con la parte real del segundo. La parte imaginaria será la diferencia entre la parte imaginaria del primer parámetro con la parte imaginaria del segundo.
- Operación: modulo
- Descripción: retorna el módulo o valor absoluto de un número complejo.
 - Pre condición: el parámetro debe ser un número complejo válido.
 - Post condición: devuelve un valor real que se calcula como la raíz cuadrada de la suma entre los cuadrados de la parte real y la parte imaginaria del parámetro.

Diseño UML

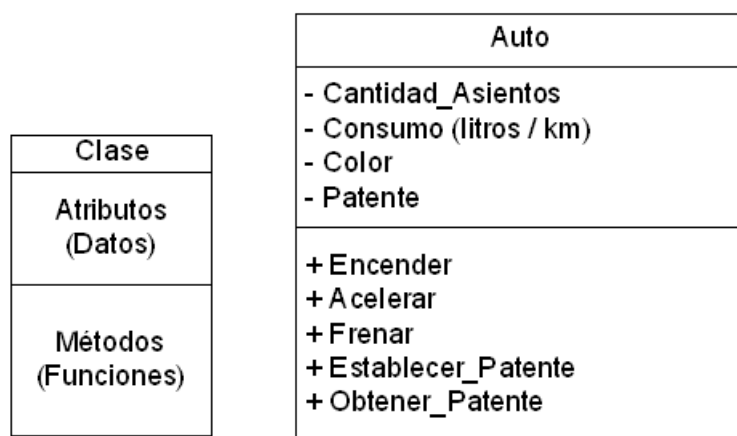
Otra forma habitual de diseñar un TDA es utilizando una simplificación de la notación UML (Lenguaje de Modelado Unificado). En UML Una clase se representa mediante un rectángulo que consiste de tres partes:

- El nombre de la clase
- Sus datos (atributos)
- Sus métodos (funciones u operaciones)

En el gráfico 3.3.1 vemos en la parte *a*) un diseño genérico, y en la parte *b*) un ejemplo. Los signos menos y más tienen que ver con el tipo de acceso: el menos significa privado o inaccesible del exterior y el más, público o accesible del exterior. Esto se explicará más adelante cuando veamos ocultamiento de la información.

3.4. Clases

Cuando implementamos un TDA para que pueda ser utilizado estamos creando una clase. En este momento hay que pensar en concreto: ¿qué lenguaje se utilizará? ¿cómo se hará? ¿qué tipos de datos se usarán dentro de la clase? Y otras preguntas más. En este momento estamos tomando el rol de implementador.



(a) Diseño genérico

(b) Ejemplo de diseño en UML

Gráfico 3.3.1: Diseño de clase en UML

Haremos dos implementaciones distintas para la clase *Complejo* y las analizaremos.

En primer lugar, por norma, dividiremos el código en dos archivos distintos: el código en 3.1 es la declaración de la clase con sus atributos y métodos (operaciones). Esto va en un archivo de cabecera `.h`. El código que se encuentra en 3.2, archivo `.cpp` es en donde se definen los métodos declarados en la clase. Si bien todo puede ser escrito y definido dentro de un mismo archivo es recomendable no hacerlo, salvo como excepción en los casos en los que trabajemos con plantillas (templates) que veremos más adelante.

También, es recomendable, no definir más de una clase en un mismo archivo. Analicemos el código del archivo de declaraciones (3.1). En este archivo de extensión `.h` que llamaremos como el nombre del tipo de dato pero en minúscula declararemos la clase, en este caso, con la sentencia

```
class Complejo (línea 4)
```

junto con las llaves de apertura y cierre (no olvidar el punto y coma luego del cierre de llaves) definen nuestra clase *Complejo*. Por norma, los nombres de las clases se declaran con su primera letra en mayúsculas. En el cuerpo de la clase irán los atributos (datos) y los métodos (funciones), pero solo la firma o cabecera de ellos. Siendo coherentes con lo que decíamos sobre la fábrica de automóviles, que no deseará que sus usuarios metan mano en el motor y que se comuniquen con el auto solo a través de su interfaz (volante, pedales, llave, etc), los atributos se colocan en una sección privada (`private`, en línea 6). Esto impide que el usuario pueda acceder directamente a estos atributos, por ejemplo si hubiera un objeto *c* de tipo *Complejo*, no podría

ejecutar la siguiente sentencia:

```
c.real = 3.6;
```

Luego de los atributos se declaran las cabeceras o firmas de los métodos que en general serán públicos, ya que a través de estos métodos los usuarios interactuarán con los atributos. Si bien los nombres de los parámetros no son necesarios en esta instancia, solo su tipo, se eligió colocarlos para definir las PRE y POST condiciones con los nombres de los parámetros. Por ejemplo, el constructor podría haberse declarado de la siguiente forma:

```
Complejo (double, double); (línea 16)
```

Nota. el orden de las declaraciones es indistinto. Se pueden declarar los atributos y los métodos en el orden en que se prefiera. Incluso pueden existir varias secciones públicas y varias privadas. En general se utilizan dos estilos:

- Primero la sección privada con sus atributos y luego la parte pública con sus métodos. Este estilo lleva el razonamiento de... (VER)
- Primero la sección pública con sus métodos (interfaz de comunicación) y luego la parte privada con sus atributos. Este estilo...

En la firma de los métodos debemos colocar el tipo que devuelve, el nombre del método y sus parámetros. Los constructores son métodos especiales que deben llevar el mismo nombre de la clase y no tienen tipo de retorno. Estos métodos se ejecutarán al momento de crear un objeto de la clase. Observando la firma del método *sumar*, por ejemplo, vemos que solo tiene un parámetro, cuando necesitamos dos números para poder sumarlos. Sin embargo, el primer parámetro está dado en forma implícita y es el propio objeto que llama al método el que se pasa como parámetro.

Otras sentencias que pueden llamar la atención son las dos primeras líneas:

```
#ifndef COMPLEJO_INCLUDED
#define COMPLEJO_INCLUDED
```

y la última:

```
#endif // COMPLEJO_INCLUDED
```

Estas líneas, que muchos IDEs colocan automáticamente al crear un archivo .h, sirven para no incluir o definir más de una vez la misma porción de código. Si no se recuerda esto se debe rever el capítulo donde se aborda el tema.

Nota. La parte pública de la clase es lo que conocemos como interfaz porque son los métodos que podrá utilizar el usuario, es decir, la comunicación con la clase. Cuando decimos *usuario* puede ser otro sistema, no necesariamente tiene que ser una persona. Por usuario se entiende “el que *usa* la clase”. La firma de los métodos con sus PRE y POST condiciones debería ser lo único que el usuario tiene que saber de la clase.

En el archivo `.cpp` (3.2) que, por convención se llama igual que el archivo `.h`, solo cambia su extensión, se definen todos los métodos. Se debe notar que hay que hacer uso del operador de ámbito, además de incluir el archivo `.h`, colocando el nombre de la clase y el operador `::` antes del nombre del método. De esta forma estamos indicando que el método, por ejemplo, *sumar*, es el que corresponde a la clase *Complejo* y no a otra clase que también podría tener un método que se llame de igual forma.

Continuando con el método *sumar* que comienza en la línea 10, se crea un número *Complejo* que llamamos *aux*, que es donde se guardará la suma y será el objeto que se devolverá. Se debe observar que, cuando se hace referencia a *real* o *imaginario* a secas, se está refiriendo al primer parámetro que es implícito, es decir el propio objeto desde donde se llama al método. Para referirnos a los atributos del segundo parámetro, que es el único que se pasa entre los paréntesis, debemos indicar el nombre del mismo, en este caso *c*, y con el operador “.” accedemos a sus atributos.

Cabe destacar que los atributos son privados para el exterior, es decir, para afuera de la clase, pero dentro de ella se puede acceder sin restricciones.

Nota. Se incluyó el archivo *cmath* para poder utilizar las funciones *sqr* (cuadrado) y *sqrt* (raíz cuadrada).

3.5. Objetos

Decimos que existe un objeto o que creamos un objeto cuando instanciamos una clase. Es decir, cuando tenemos un elemento concreto de ese molde que llamamos clase. Por ejemplo, una clase *Auto* tendría un número de patente, una marca y un color. Un objeto de tipo *Auto* puede ser ABC 123, Renault, rojo cuyo nombre es *auto_de_mi_tio*. La relación entre un objeto y su clase es similar a la que tiene una variable con su tipo. Cuando creamos un objeto es que estamos en el papel de usuario. Entonces la relación entre las etapas, los roles y los productos es la siguiente:

Algoritmo 3.1 Archivo complejo.h - Declaración 1

```
1  #ifndef COMPLEJO_INCLUDED
2  #define COMPLEJO_INCLUDED
3
4  class Complejo {
5
6  private:
7      // atributos
8      double real;
9      double imaginario;
10
11 public:
12     // constructor con parametros
13     // PRE: re, im son de tipo double
14     // POST: construye un Complejo, en donde
15     //   real = re e imaginario = im
16     Complejo (double re, double im);
17
18     // metodo sumar
19     // PRE: c es de tipo Complejo
20     // POST: devuelve un nuevo Complejo, en donde
21     //   nuevo.real = real + c.real
22     //   nuevo.imaginario = imaginario + c.imaginario
23     Complejo sumar(Complejo c);
24
25     // metodo restar
26     // PRE: c es de tipo Complejo
27     // POST: devuelve un nuevo Complejo, en donde
28     //   nuevo.real = real - c.real
29     //   nuevo.imaginario = imaginario - c.imaginario
30     Complejo restar(Complejo c);
31
32     // metodo modulo
33     // PRE:
34     // POST: devuelve un double calculado como
35     //   raiz(real * real + imaginario * imaginario)
36     double modulo();
37 };
38
39 #endif // COMPLEJO_INCLUDED
```

Algoritmo 3.2 Archivo complejo.cpp - Implementación 1

```
1  #include "complejo.h"
2  #include <cmath>
3
4  // constructor con parametros
5  Complejo::Complejo(double re, double im) {
6      real = re;
7      imaginario = im;
8  }
9
10 // metodo sumar
11 Complejo Complejo::sumar(Complejo c) {
12     Complejo aux(0.0, 0.0);
13     aux.real = real + c.real;
14     aux.imaginario = imaginario + c.imaginario;
15     return aux;
16 }
17
18 // metodo restar
19 Complejo Complejo::restar(Complejo c) {
20     Complejo aux(0.0, 0.0);
21     aux.real = real - c.real;
22     aux.imaginario = imaginario - c.imaginario;
23     return aux;
24 }
25
26 // metodo modulo
27 double Complejo::modulo() {
28     double aux = sqr(real) + sqr(imaginario);
29     return sqrt(aux);
30 }
```

Algoritmo 3.3 Uso de la clase *Complejo*

```

1  #include <iostream>
2  #include "complejo.h"
3
4  using namespace std;
5
6  int main() {
7      Complejo c1(5.0, 3.0);
8      Complejo c2(4.0, 2.0);
9      Complejo c3(0.0, 0.0);
10     c3 = c1.sumar(c2);
11     cout << "el modulo de c3 es " << c3.modulo() << endl;
12     return 0;
13 }

```

Etapas	Rol	Producto
Diseño	Diseñador	TDA
Implementación	Implementador	Clase
Utilización	Usuario	Objeto

¿Cómo se crea un objeto?

Similar a una variable: se indica el nombre de la clase (tipo), un nombre para el objeto y si el constructor lleva parámetros se deben pasar entre paréntesis. En las líneas 7, 8 y 9 del algoritmo 3.3 se crean 3 objetos de tipo *Complejo*. En la línea 10 se llama al método *sumar* a través del objeto *c1*. Este es el primer parámetro que mencionábamos. El segundo parámetro es *c2*. El método *sumar* devuelve un objeto de tipo *Complejo* que se lo asigna a *c3*.

Otro implementador podría haber pensado una estructura distinta para la clase *Complejo*. Por ejemplo, en lugar de utilizar dos atributos: uno para la parte real y otro para la imaginaria, se podría utilizar un vector. Con esta segunda implementación tenemos un vector de tipo *double*, este vector lo llamamos *z* y tiene dos posiciones (la primera para la parte real y la segunda para la parte imaginaria). Lo interesante es notar que (ver 3.4), del archivo *.h* sólo cambia la parte privada y, por supuesto, cambia la definición de los métodos en el archivo *.cpp* (ver 3.5). Sin embargo, para el usuario estos cambios son transparentes porque la interfaz no cambió en nada, por lo que el programa que utiliza el tipo de dato *Complejo* funciona a la perfección con ambas implementaciones. Esta es la ventaja de separar la interfaz de la implementación o, pensado de otra forma, la ventaja de *abstraerse*: no pensar cómo están hechas las cosas, sino cómo debemos tratar con ellas.

¿Por qué esto es importante? ¿Qué necesidad puede haber para cam-

Algoritmo 3.4 Archivo complejo.h - Declaración 2

```
1  #ifndef COMPLEJO_INCLUDED
2  #define COMPLEJO_INCLUDED
3
4  class Complejo {
5  private:
6      // atributos
7      double z[2];
8
9  public:
10     // constructor con parametros
11     // PRE: re, im son de tipo double
12     // POST: construye un Complejo, en donde
13     //      real = re e imaginario = im
14     Complejo (double re, double im);
15
16     // metodo sumar
17     // PRE: c es de tipo Complejo
18     // POST: devuelve un nuevo Complejo, en donde
19     //      nuevo.real = real + c.real
20     //      nuevo.imaginario = imaginario + c.imaginario
21     Complejo sumar(Complejo c);
22
23     ...
24 };
25
26 #endif // COMPLEJO_INCLUDED
```

biar una implementación? La implementación se puede querer cambiar por una cuestión de eficiencia, velocidad en la ejecución, una mejora en el mantenimiento de la clase, facilidad para agregar funcionalidad, etc. Estamos acostumbrados a que vayan saliendo nuevas versiones de las aplicaciones que utilizamos, con una interfaz más atractiva, amigable y nueva funcionalidad (algunas veces, esto no siempre sucede). Sin embargo, las nuevas versiones en general son capaces de trabajar con los archivos que hemos creado con versiones anteriores. De lo contrario tendríamos que generar un nuevo archivo para cada nueva versión que se lance. Ese es uno de los objetivos a perseguir cuando desarrollamos clases y trabajamos con objetos.

Algoritmo 3.5 Archivo complejo.cpp - Implementación 2

```
#include "complejo"
#include <cmath>

// constructor con parametros
Complejo::Complejo(double re, double im) {
    z[0] = re;
    z[1] = im;
}

// metodo sumar
Complejo Complejo::sumar(Complejo c) {
    Complejo aux(0.0, 0.0);
    aux.z[0] = z[0] + c.z[0];
    aux.z[1] = z[1] + c.z[1];
    return aux;
}
...
```

3.6. Algunas características de la POO

3.6.1. Encapsulamiento

AGREGAR

3.6.2. Ocultamiento de la información

Con la finalidad de proteger al objeto y a su usuario, los datos deberían ser, en general, inaccesibles desde el exterior para que no puedan ser modificados sin autorización. Por ejemplo: sería un caos si, en una biblioteca, cada persona tomara y devolviera por su cuenta los libros que necesitara ya que podría guardarlos en otro lugar, mezclarlos, dejar a otros usuarios sin libros, etc. Para que no suceda esto hay una persona, que es el bibliotecario, que funciona de interfaz entre el lector y los libros. El lector solicita al bibliotecario un libro. El bibliotecario toma del estante correspondiente el libro, toma la credencial de la persona que lo solicitó, lo registra y se lo presta.

Este es uno de los principios que se utilizan en la POO: colocar los datos de manera inaccesibles desde el exterior y poder trabajar con ellos mediante métodos estipulados a tal fin. Estos métodos forman parte de la interfaz del usuario.

3.6.3. Herencia

Tenemos en claro que un objeto *auto* no es lo mismo que un objeto *ómnibus* o *ferrocarril*. Pero, todos estos objetos tienen cosas en común. Lo más importante que comparten es que todos sirven como medio de transporte (ver gráfico 3.6.1). En este punto aparece un nuevo concepto en la POO que es la herencia. Las clases *Auto*, *Colectivo*, *Ferrocarril* heredan de una clase en común que será *Transporte*. Por lo tanto, tendrán comportamientos (métodos) y atributos que compartirán. Como ser la cantidad de personas que pueden transportar, el consumo de combustible, etc. Obviamente, luego, cada uno de los objetos mencionados podrá tener otros atributos y métodos que serán particulares de su clase.

Hay que notar que a las clases *Auto*, *Colectivo* y *Ferrocarril* pudimos ilustrarlas con una fotografía. Sin embargo, la clase *Transporte*, no. En estos casos, decimos que la clase *Transporte* es una clase abstracta. Una clase abstracta es aquella que no se puede instanciar, es decir, no se puede crear un objeto de esa clase. En este ejemplo no tendría sentido crear un objeto de tipo *Transporte* ya que no sería real. Decimos que las clases concretas o efectivas son las que se pueden instanciar, en este ejemplo, las que heredan de *Transporte*.



Nota. La herencia no siempre implica clases abstractas. Por ejemplo, una clase *Persona* podría ser una clase real (o concreta) y, la clase *Estudiante*, heredaría de persona. Es decir, sería una *Persona* con algunas características más.

Un diagrama UML podría ser el que se muestra en el gráfico 3.6.2. Vemos que la clase *Auto* tiene un solo dato, que es *espacio_en_el_baul*, sin embargo, esto no es cierto, ya que heredó de la clase *Transporte* *cantidad_asientos*, *color* y *patente*. Lo mismo sucede con sus métodos.

3.6.4. Polimorfismo

El polimorfismo significa que un objeto puede tener varias (poli) formas (morfo). Es la propiedad que tienen distintos objetos de comportarse de distinta manera ante un mismo mensaje. Veremos este tema en el próximo capítulo, pero se anticipa el concepto con un ejemplo: si a un objeto *Persona* le decimos que se alimente, no actuará de la misma forma que si se lo dijéramos al objeto *Planta*. Entonces, el mismo método: *alimentar* trabajaría de forma diferente dependiendo qué objeto lo invoque. Esto puede no ser novedoso si supiéramos cuál es ese objeto, la gracia del polimorfismo es no saber qué objeto lo está invocando.

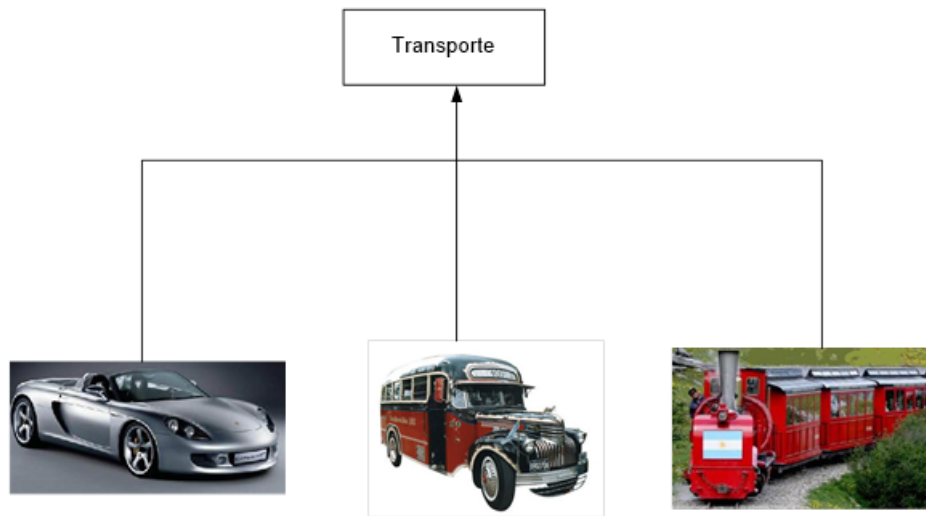


Gráfico 3.6.1: Herencia

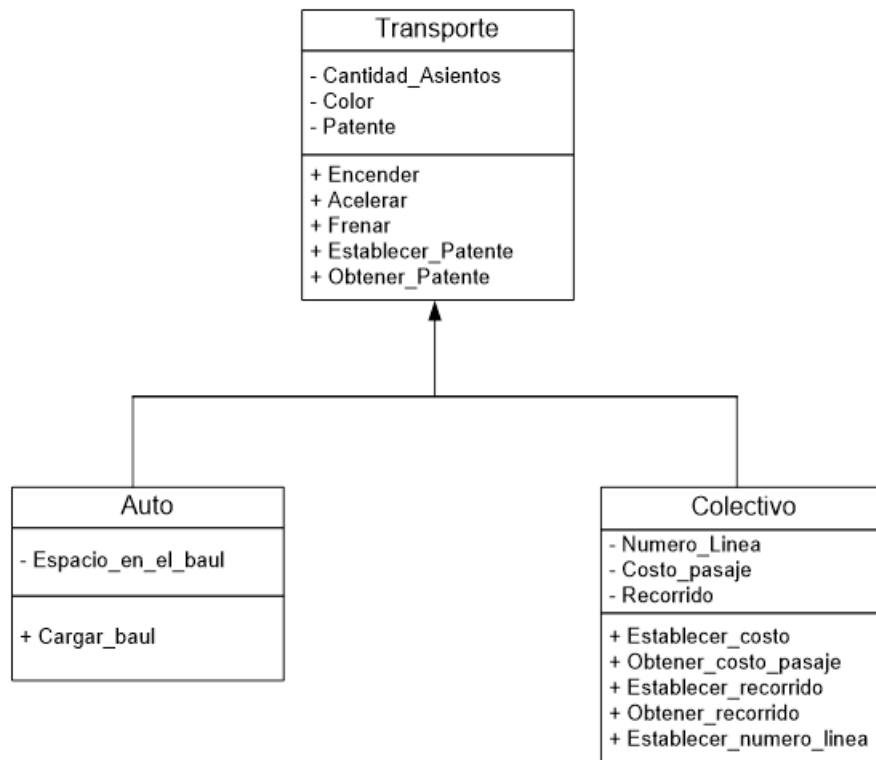


Gráfico 3.6.2: Herencia: diagrama UML

3.7. Acceso a los atributos

Unas secciones atrás recomendamos que los atributos deben ser privados con el fin de proteger los datos. En la clase *Complejo*, tanto el atributo *real* como *imaginario* son de tipo privado. Por este motivo debemos definir métodos para poder asignar valores a estos atributos. Hay que tener en cuenta que no nos sirve un solo método para asignar ambos valores, porque podríamos desear asignar solo la parte real y no modificar la parte imaginaria o viceversa. Entonces, debemos hacer un método para asignar su parte real y, otro, para su parte imaginaria.

En general, como regla, por cada atributo, tendremos un método específico para asignar o colocar su valor. Estos métodos, en general, los llamamos *asignarNombreAtributo*. Por ejemplo *asignarReal* o *asignarImaginario*.

El caso contrario es si quisiéramos consultar esos valores o imprimirlos. Tampoco podríamos acceder a dichos atributos, por lo que deberíamos tener los métodos para conseguir dichos valores, estos métodos se llaman por convención *obtenerNombreAtributo*. Por ejemplo *obtenerReal* u *obtenerImaginario*.



Nota. Podrían llamarse de cualquier manera, se destaca que es solo por convención. Otros nombres utilizados en lugar de *asignar* son *cambiar* o *modificar*.

Obviamente, los métodos de *asignar* y *obtener* tienen que ser públicos para poder ser accedidos fuera de la clase.

Los métodos *asignar* colocan el valor del atributo, el cual se deberá pasar por parámetro y no devuelven nada. En cambio los métodos *obtener* no deben recibir ningún parámetro pero sí deberán devolver un valor que es el del atributo. Por ejemplo con la clase *Complejo*, los métodos *asignar* son:

```
void Complejo::asignarReal(double r) {
    real = r;
}

void Complejo::asignarImaginario(double i) {
    imaginario = i;
}
```

Y los de obtener:

```
double Complejo::obtenerReal() {
    return real;
}

double Complejo::obtenerImaginario() {
```

```
        return imaginario;  
    }
```

3.8. Constructores

Habíamos dicho que los constructores son métodos especiales que no tienen ningún tipo de retorno, ni siquiera *void*, y que se llaman de forma automática cuando creamos un objeto. Estos métodos son tan importantes que si no escribiéramos uno, C++ nos ofrece un constructor que se llama constructor de oficio. Para el desarrollador será transparente ya que el código no está escrito pero se llamará al momento de crear un objeto. Por supuesto ese constructor de oficio no lleva parámetros y lo que hace es poner las variables numéricas en 0, las booleanas en falso, los punteros en nulo, etc.

Es decir, si no hubiéramos escrito un constructor para la clase *Complejo* deberíamos utilizar el constructor de oficio al crear los objetos de esta forma:

```
Complejo c;
```

Como veremos en la siguiente sección, podemos tener más de un constructor.

3.9. Sobrecarga de métodos

Dos o más métodos pueden tener el mismo nombre siempre y cuando difieran en

- la cantidad de parámetros
- los tipos de los parámetros
- ambas cosas

La sobrecarga vale también para los constructores. Imaginemos que quisiéramos construir un objeto de tipo *Complejo* sin asignarle en principio ningún valor en particular, que por defecto se inicialicen en 0 tanto la parte real como la imaginaria. Como el constructor lleva dos parámetros la siguiente línea

```
Complejo c;
```

da error en tiempo de compilación. El lector se preguntará si no actúa el constructor de oficio que provee C++. Lo que sucede es que al escribir explícitamente un constructor, el de oficio no interviene. Por lo que estamos obligados a escribir

```
Complejo c(0.0, 0.0);
```


Sin embargo, una solución es sobrecargar el constructor, es decir, escribir otro que no lleve parámetros, de la siguiente forma:

```
// Constructor sin parametros - definicion
Complejo::Complejo() {
    real = 0.0;
    imaginario = 0.0;
}
```

Entonces, si luego, como usuarios escribiéramos

```
Complejo c1;
Complejo c2(3.5, 2.4);
```

se ejecutaría el constructor sin parámetros para el objeto *c1*, asignando el valor de cero para la parte real y la imaginaria. En cambio, para la creación del objeto *c2* llamaría al constructor con parámetros.

3.10. Parámetros por defecto

El ejemplo anterior de sobrecarga de constructores también se puede solucionar con lo que se llama parámetros por defecto. ¿Qué es y cómo funciona? Lo vemos con un ejemplo:

```
// Constructor con parametros por defecto
Complejo::Complejo(double re = 0.0, double im = 0.0) {
    real = re;
    imaginario = im;
}
```

Un constructor como el anterior nos sirve para las siguientes sentencias:

```
Complejo c1;
Complejo c2(3.5, 2.4);
Complejo c3(3.5);
```

En el caso de tener dos parámetros, como en la creación del objeto *c2*, se ejecuta el cuerpo del constructor colocando el valor de 3.5 para la parte real y 2.4 para la imaginaria. En caso de no tener parámetros, como en la creación del objeto *c1*, utiliza los valores por defecto que, en este ejemplo, valen 0.0 para sus dos atributos. En cuanto a la creación del objeto *c3*, el valor 3.5 es asignado a su parte real porque es el primer atributo y, para su parte imaginaria, como falta el valor, toma el valor por defecto que es 0.0.

Los parámetros por defecto no pueden solucionar el caso inverso de la construcción del objeto *c3*: si quisiéramos que tome por defecto a la parte real y que asigne el valor pasado por parámetro al atributo *imaginario*, deberíamos indefectiblemente pasar los dos parámetros.

3.11. El puntero *this*

Todo objeto se crea con una referencia o puntero a sí mismo, este puntero se llama *this* y se genera de manera automática. ¿Cuándo es necesario utilizarlo? Se puede utilizar siempre pero nos veremos obligados a usarlo cuando el compilador no pueda resolver una ambigüedad en los nombres. Veamos algunos ejemplos:

```
// No se usa this
Complejo::Complejo(double re, double im) {
    real = re;
    imaginario = im;
}

// Se utiliza this pero no es necesario
Complejo::Complejo(double re, double im) {
    this->real = re;
    this->imaginario = im;
}

// Es necesario utilizar this
Complejo::Complejo(double real, double imaginario) {
    this->real = real;
    this->imaginario = imaginario;
}
```

En el último caso, es necesario utilizar el puntero *this* para diferenciar los parámetros *real* e *imaginario* de los atributos del propio objeto (*this*).

3.12. Atributos y métodos estáticos

Un atributo (o campo) estático es un atributo que no depende de un objeto en particular sino de la clase en sí. Por ejemplo, un objeto de la clase *Auto* tendrá un atributo *color*. Podríamos tener varios objetos de tipo *Auto* que tuvieran el mismo color. Sin embargo, el dato o atributo *patente* no se podría repetir. Por lo que deseáramos que ese atributo no pudiera manipularse ni cambiarse desde afuera y además tuviera valores correlativos, por lo que debería depender de otro atributo, como podría ser el atributo *patente_a_asignar*. Este atributo no debería pertenecer a un objeto *auto* en particular, sino a la clase *Auto* directamente para, de ese modo, controlar sus valores desde allí. Esto se logra indicando que el atributo *patente_a_asignar* es *static*. Cabe resaltar que por clase existe una sola copia de un campo que es *static*.

En el gráfico 3.12.1 representamos el ejemplo anterior. Vemos que cada auto tiene un *color* y una *patente*. Pero esa patente no puede ser cualquiera,

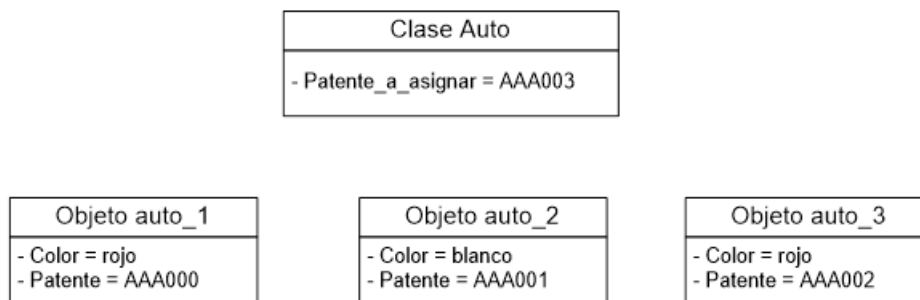


Gráfico 3.12.1: campos estáticos

conserva un orden, por lo tanto, el valor del atributo *patente* se controla por el atributo *patente_a_asignar* que es único para toda la clase.

Para que un atributo sea *static*, solo se debe agregar este modificador en el momento de declararlo. Por otro lado, los campos estáticos pueden ser accedidos desde cualquier objeto, por ejemplo:

```
auto_1.patente_a_asignar
```

Sin embargo, por una cuestión de claridad en el código, es preferible accederlos desde la misma clase:

```
Auto::patente_a_asignar
```

De todas formas, siendo coherentes con lo que venimos diciendo, el atributo *patente_a_asignar*, debería ser privado para no ser accedido desde fuera de la clase. Entonces, la forma más lógica de acceder a un atributo privado es mediante algún método escrito para tal fin. Sin embargo, como este atributo además de privado es estático, debería accederse mediante un método estático.



Nota. Los atributos estáticos deben ser inicializados ya que su valor debe estar disponible para la clase y no para un objeto en particular.

En el algoritmo 3.6 se muestra un ejemplo de cómo se definen e inicializan los atributos estáticos. Analizando ese código, se define una clase con nombre *ClaseA*, en un archivo *.h*, como siempre. Esta clase tiene un atributo *x*, que pertenece a cada objeto en particular. Pero, además, tiene un atributo *y* que pertenece directamente a la clase porque es estático.

Supongamos que deseamos que los objetos se vayan creando con un número de serie que empieza en 101, sigue en 102, etc. Debemos inicializar el atributo *y*, como se hace en el archivo *claseA.cpp*. Cabe destacar que esta inicialización está fuera de cualquier método.

Por otra parte, en el constructor nos encargamos de incrementar en uno

Algoritmo 3.6 Atributos y métodos estáticos

```
// Archivo claseA.h
class ClaseA {
private:
    int x;          // Atributo x (propio del objeto)
    static int y;   // Atributo y estático (propio de la clase)
public:
    ClaseA(int a); // Constructor
    // Metodos obtener (el de y debe ser estático)
    int obtener_x();
    static int obtener_y();
};

// Archivo claseA.cpp
#include "claseA"

int ClaseA::y = 100; // El atributo y se debe inicializar

ClaseA::ClaseA(int x) {
    this->x = x;
    y++;
}

int ClaseA::obtener_x() {
    return x;
}

int ClaseA::obtener_y() {
    return y;
}
```

a esta variable. Es decir, cada vez que se cree un objeto tipo *ClaseA* se incrementará este contador.

En la función main (ver 3.7), creamos dos objetos de tipo *ClaseA* pasando los parámetros para que inicialice el valor de x pero no para y , ya que este valor depende de la clase y no de cada objeto en particular.

El llamado para imprimir los valores de y se puede realizar tanto desde un objeto, como se hace en el primer caso, como desde su clase, como en el segundo. Es preferible hacerlo de esta última forma dado que en el primer caso pareciera que el atributo es un valor perteneciente al objeto en sí y no a su clase.

La salida de la ejecución del código 3.7 es:

```
Objeto a, atributo x: 3
ClaseA, atributo y: 101
```

Algoritmo 3.7 Uso de atributos estáticos

```
#include <iostream>
#include "claseA"

using namespace std;

int main() {
    ClaseA a(3);
    cout << "Objeto a, atributo x: " << a.obtener_x() << endl;
    // Se llama al metodo obtener_y desde el objeto
    cout << "ClaseA, atributo y: " << a.obtener_y() << endl;
    ClaseA b(5);
    cout << "Objeto b, atributo x: " << b.obtener_x() << endl;
    // Se llama al metodo obtener_y desde la clase
    cout << "ClaseA, atributo y: " << ClaseA::obtener_y() << endl;
    return 0;
}
```

```
Objeto b, atributo x: 5
ClaseA, atributo y: 102
```

3.13. Destruyores

Los destructores, al igual que los constructores, son métodos especiales que se ejecutan sin un llamado explícito. La función que tienen es liberar recursos, como cerrar archivos, liberar memoria dinámica utilizada, etc. Estos métodos son tan importantes que, de no definirlos, el lenguaje provee un destructor de oficio. En el ejemplo de la clase *Complejo* que estuvimos viendo no era necesario liberar ningún recurso, por este motivo no se programó ningún destructor.

Como sucede con los constructores, los destructores deben tener el mismo nombre que la clase, pero con un signo ~ adelante del nombre. Además, no llevan parámetros ni pueden ser sobrecargados. Una sobrecarga no sería permitida por el lenguaje ya que se estaría repitiendo la firma del método, aparte de que no tendría ningún sentido.

Es importante recalcar que los destructores no deben ser llamados en forma explícita, se llaman automáticamente cuando:

- En caso de un objeto no dinámico, se termina el ámbito en donde el objeto fue definido
- Si el objeto es dinámico, cuando se ejecuta la instrucción *delete*. Esto

sucede cuando el objeto fue creado con el operador new.

Por ejemplo:

```
class Clase {
    // atributos ...
    // metodos ...
    // constructores
    Clase();
    ...
    // destructor
    ~Clase();
};

void funcion() {
    Clase o1; // Se crea un objeto no dinamico de tipo Clase
    Clase *ptr = new Clase(); // Se crea un objeto dinamico tipo Clase
    ...
    delete ptr; // Se libera la memoria solicitada
                // En este momento se llama al destructor de Clase
    ...
    // Otras sentencias
    ...
    // Termina el bloque donde se define funcion
} // En este momento se llama al destructor del objeto o1
```

¿Cuándo y cómo programamos un destructor?

El caso típico es cuando se solicita memoria dinámica para uno o más atributos de la clase. Lo veremos con otro ejemplo. Supongamos que deseamos una clase para manejar vectores de enteros en forma dinámica, indicando, en su constructor el tamaño del vector. Entonces:

```
class Vector {
private:
    // atributos
    int tamano; // longitud del vector
    int *datos; // datos que tendra en el vector

public:
    // constructor
    Vector(int tam);

    // otros metodos
    ...
    // destructor
    ~Vector();
};
```

```
// Definicion de los metodos
// Constructor
Vector::Vector(int tam) {
    tamano = tam;
    // Se solicita memoria dinamica para guardar los datos
    datos = new int[tamano];
}

// Destructor
Vector::~Vector() {
    // Se libera la memoria solicitada
    if (tamano > 0)
        delete []datos;
}
```

3.14. Constructor de copia

El constructor de copia no es ni más ni menos que otro constructor más, el cual, si no es programado, el lenguaje también provee uno de oficio.

¿Cuándo se llama? Se llama cuando se necesita copiar un objeto, por ejemplo cuando se pasa un objeto por parámetro en alguna función o, como cuando se crea un objeto igualándolo a otro. Ejemplos:

```
1 void funcion(Clase o) {
2     // sentencias de la funcion
3     ...
4 }
5
6 Clase o1;
7 funcion(o1);
8 Clase o2 = o1;
```

En la línea 7 del código anterior se llama a la función definida al principio, la cual recibe un objeto por parámetro. Dentro de la función se debe trabajar con este objeto el cual está pasado por valor, es decir que se crea, internamente, una copia del mismo, llamando al constructor de copia.

En la última línea (línea 8), se crea un objeto *o2* igualándolo a otro objeto *o1* de su misma clase. También se crea una copia de este objeto, ya que se necesita construir el objeto *o2* en base al objeto *o1*.

¿Qué hace el constructor de copia de oficio? Simplemente copia todos los valores de los atributos de un objeto al otro.

¿Qué problema puede presentar esto? El problema surge cuando se utiliza memoria dinámica. Volvamos al ejemplo de la clase *Vector*. Esta clase

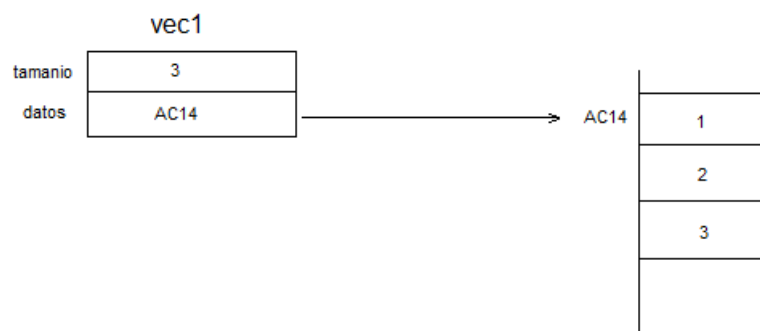


Gráfico 3.14.1: Atributos que manejan memoria dinámica

Vector tiene dos atributos: un entero que indica su tamaño y un puntero, a una zona de memoria donde estarán los datos en sí. Supongamos que creamos un objeto de tipo *Vector* para guardar los datos 1, 2 y 3. Como deseamos guardar tres enteros, creamos un objeto *Vector* con parámetro 3 de la siguiente manera:

```
Vector vec1(3);
```

Luego, con otro método colocaremos los datos 1, 2 y 3 en el objeto. Pero analicemos un poco la línea de código anterior, luego de ejecutarse dicha línea, se llamará al constructor que hará lo siguiente:

- el atributo *tamaño* quedará con el valor 3
- luego, pedirá memoria para albergar a tres enteros. La porción de memoria que asigne tendrá una dirección de comienzo, supongamos que esta dirección es AC14. Por lo tanto, este valor, AC14, será el dato que guarde el atributo *datos*.

Hay que tener muy en claro que, los valores 1, 2 y 3, que aún no se han guardado pero se supone que se harán a la brevedad con otro método, no están almacenados dentro del objeto *vec1*, sino que están fuera de él, en otra porción de memoria. En particular, ocuparán tres lugares contiguos para que cada uno albergue un entero a partir de la dirección AC14. Lo único que tenemos en el objeto *vec1* que se relaciona con estos datos es dicha dirección de memoria. La representación gráfica la vemos en 3.14.1.

Ahora bien, cuando otro objeto, supongamos *vec2* copie de *vec1*, copiará todos los valores de los atributos tal cual están. El constructor de copia toma lo que tiene el atributo *datos* como un entero más, no sabe si eso es una dirección u otra cosa. Por lo tanto, si ejecutamos la siguiente sentencia:

```
Vector vec2 = vec1;
```


Nuestro nuevo objeto, *vec2*, tendrá el valor de 3 en *tamano*, lo cual es correcto y, en *datos*, tendrá AC14, es decir, apuntará a la misma zona de memoria que *vec1*. Esto no está bien por varios motivos: uno de los problemas es que, si modificamos los valores de *vec2*, también se modificarán los de *vec1* y viceversa. En general esto no es lo que uno desea. Pero hay otro motivo por lo que lo anterior no es correcto, y este motivo hará que nuestra aplicación finalice de manera incorrecta: cuando el objeto *vec1* (o *vec2*) sea destruido, porque se terminó el ámbito de su definición, por ejemplo, se llamará al destructor y se liberará la memoria solicitada, lo que hará inaccesible los datos para el otro objeto, ya que la memoria se liberó. Es decir que un intento de acceso a la porción de memoria a través del otro objeto daría un error en tiempo de ejecución. También daría un error en tiempo de ejecución cuando se llame al destructor del objeto que siguió con vida, ya que se intentaría liberar memoria que ya ha sido liberada.

En conclusión:

siempre que manejemos atributos que utilicen memoria dinámica es necesario definir un constructor de copia.

El parámetro del constructor de copia debe ser una referencia a un objeto de la misma clase que estamos programando, dado que se va a copiar de él. No puede ser el objeto en sí, es decir, no se podría pasar por valor, porque se debería, a su vez, copiar el mismo parámetro, lo que crearía un llamado recursivo infinito imposible de resolver. Además, por convención, se suele indicar que esa referencia es constante para asegurar que el objeto no será modificado. Entonces, siguiendo con el ejemplo de la clase *Vector*:

```
// Constructor de copia
Vector::Vector(const Vector &vec) {
    // los tamanos deben ser iguales
    tamano = vec.tamano;
    // Se solicita nueva memoria para guardar los datos
    datos = new int[tamano];
    // Se copian los valores a esa nueva porcion de memoria
    for (int i = 0; i < tamano; i++)
        datos[i] = vec.datos[i];
}
```

De esta forma, al crear una copia del objeto *vec1*, se estaría creando otra copia de los datos en otra porción de memoria. Con lo cual, cada objeto tendría la dirección de distintas zonas de memoria, siendo independientes entre sí.



Nota. El llamado a los distintos constructores se realiza muchas veces sin que nos demos cuenta. Como ejercicio es aconsejable escribir un cartel de aviso dentro de cada constructor diciendo “Se llama al constructor de...” para verificar los llamados a los diferentes constructores.

3.15. Sobrecarga de operadores

Al igual que con los métodos, los operadores, también pueden sobrecargarse. Si se tiene que sumar dos números, lo más lógico es que se desee utilizar el operador `+` y no un método que se llame sumar, aunque estos números fueran números complejos o de cualquier otro tipo. De hecho, el operador `+` lo venimos utilizando con sobrecarga, por ejemplo:

```
int x = 5, y = 8, z;
string s1 = "Hola ", s2 = "Juan", s3;
z = x + y;
s3 = s1 + s2;
```

La variable `z` queda con el valor 13 y la variable `s3` con el string “Hola Juan”. Lo que significa que si los parámetros son enteros el operador `+` realiza la suma algebraica. En cambio, si los parámetros son de tipo string, el operador `+` los concatena.

Para sobrecargar el operador `+` en la clase *Complejo*, debemos agregar en el archivo `.h`, el siguiente método:

```
// operador +
Complejo operator+ (Complejo c);
```

Lleva las mismas PRE y POST condiciones que el método sumar. Luego, en el archivo `.cpp` definimos:

```
// operador+
Complejo Complejo::operator+(Complejo c) {
    Complejo aux;
    aux.real = real + c.real;
    aux.imaginario = imaginario + c.imaginario;
    return aux;
}
```

Como se observa, la sintaxis es exactamente la misma que la del método sumar.

¿Cómo es su uso? La primera forma de uso sería la misma que la de cualquier otro método, por lo tanto, podríamos escribir la siguiente sentencia:

```
c3 = c1.operator+(c2);
```

Sin embargo, la sobrecarga de operadores tiene sentido con la siguiente forma de uso:

```
c3 = c1 + c2;
```

Como se advierte, estamos realizando la suma de números complejos con la misma sentencia que si hiciéramos la suma de enteros o flotantes.

Sobrecarga del operador =

Los problemas indicados con el constructor de copia, se repiten cuando se utiliza el operador =. Por ejemplo:

```
vec2 = vec1;
```

En este caso no se utiliza el constructor de copia, porque los constructores son invocados cuando el objeto se crea, sin embargo, en la sentencia anterior, no se están creando objetos, se supone que ya están creados anteriormente. De esta forma se está utilizando una sobrecarga del operador =. Por lo tanto, deberíamos también sobrecargar el operador = para poder utilizarlo sin que se produzcan efectos no deseados.

Hay que tener cuidado distinguiendo estas sentencias:

```
Vector vec2 = vec1; // llama al constructor de copia
```

En la sentencia anterior no llama al operador = porque se está creando el objeto en ese momento, es equivalente a escribir:

```
Vector vec2(vec1);
```

El código del operador = es similar al del constructor de copia, con la diferencia de que hay que tener presente que el objeto ya está creado, por lo que debe verificar si tiene memoria solicitada anteriormente y liberarla. En el siguiente capítulo vamos a desarrollar por completo la clase *Vector* y veremos bien este tema.

3.16. Pre y pos condiciones

Es importante definir al principio de cada método la Pre y Post condición. ¿Qué significa esto? ¿Por qué son necesarias?

Las Pre y Post condiciones son comentarios que, además de aclarar el código del método, funcionan como un contrato entre el usuario y el implementador. En una precondición decimos en qué estado debe estar el objeto para llamar a determinado método y cuáles son los valores válidos de sus parámetros. En la poscondición decimos cómo va a quedar el estado de la clase luego de ejecutar el método y qué devuelve, si hubiera alguna devolución.

Estos comentarios son necesarios por las siguientes dos razones:

- a. La primera razón es porque ayuda a que el código sea más claro. A la hora de extenderlo o buscar errores de ejecución u otros motivos, estos comentarios nos ayudan a ver qué hace esa porción de código.
- b. La segunda razón es que forman parte del contrato implementador – usuario. El implementador o desarrollador de la clase dice lo siguiente: “si se cumplen las precondiciones, garantizo el resultado del método”. De esta manera, son un buen elemento para determinar responsabilidades. ¿Por qué falló el sistema? ¿Se cumplió la precondición? Si la respuesta es sí, el sistema falló por un desacierto del implementador. En cambio, si la respuesta es no, el responsable de la falla es el usuario de la clase.

Cuestionario

- a. ¿Qué es un TDA?
- b. ¿Qué paradigmas de programación se utilizaron antes de la POO? ¿Cuáles eran sus principales problemas?
- c. ¿Qué es una clase?
- d. ¿Qué es un objeto?
- e. ¿Cuáles son las principales características en la POO?
- f. ¿Cómo se debe encarar un problema en la POO?
- g. ¿Qué significa public y private?
- h. ¿Qué significa static? ¿Desde dónde conviene acceder a un atributo de tipo static? Dar un ejemplo de su uso.
- i. ¿Qué es un constructor?
- j. ¿Qué es un destructor? ¿Cuándo se debe programar uno?
- k. ¿Qué es la sobrecarga de métodos?
- l. ¿Qué es el puntero this? ¿Cuándo se debe utilizar?
- m. ¿Qué son las Pre y Post condiciones? ¿Para qué sirven?

Ejercicios

- a. Diseñar el TDA Fraccion. Una Fraccion tiene que poder inicializarse, simplificarse, sumarse, restarse, multiplicarse y dividirse con otra.
- b. Diseñar e implementar la clase Rectangulo, con atributos base y altura y los métodos para modificar y obtener sus valores, obtener el perímetro y el área.
- c. Diseñar e implementar la clase Alimento. Un alimento tiene un nombre y una cantidad de calorías asociada cada 100 gramos.
- d. Escribir en una clase Principal que utilice el objeto implementado en el punto anterior.
- e. Implementar la clase Fraccion diseñada antes. Luego, utilizando esta clase, escribir una calculadora de fracciones.

Parte II

Estructuras de datos lineales

Capítulo 4

Genericidad

La genericidad o condición de genérico la aplicaremos en todas las clases contenedoras, como vectores, listas, árboles, etc. Muchas veces veremos estas clases con tipos de datos enteros, como un *int* o *char* para concentrarnos en la estructura de datos y no en lo que guarda, pero la realidad es que en esas estructuras desearemos almacenar tipos diversos de datos. Veremos cómo hacerlo con un TDA contenedor como el TDA Vector. Comenzaremos con un tipo de dato simple para luego continuar con la evolución lógica hacia los tipos genéricos.

4.1. TDA Vector

Si bien es sencillo definir un vector en C++, lo que queremos es que este vector se estire o se acorte cuando el usuario lo decida. La realidad es que cuando definimos un vector, una vez que el bloque contiguo de memoria está determinado, no se puede modificar, así sea que dicho bloque se haya solicitado en forma estática o dinámica. Por ejemplo, si pedimos con la sentencia *new* un bloque para trabajar con 100 enteros, y más adelante se necesitan 10 más, no hay forma de indicar que el nuevo bloque de memoria sea contiguo al bloque de 100 elementos anteriormente solicitado. Por lo tanto, una alternativa es manejar el nuevo vector como dos partes no conexas, haciéndolo transparente para el usuario. Sin embargo, esta opción no es tan trivial, porque si se vuelve a indicar una nueva extensión del tamaño, deberían ser tres partes, etc.

Otra posibilidad para el problema planteado es solicitar un nuevo bloque de 110 enteros, copiar los 100 que ya se tenían y luego liberar ese bloque de memoria. Esto para el usuario es molesto ya que lo desvía de su objetivo: operar con un vector de enteros, por lo que la implementación del TDA

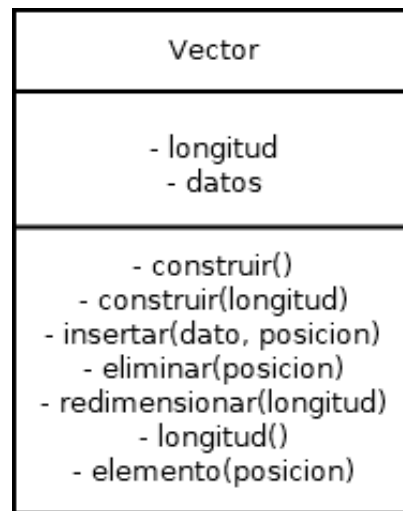


Gráfico 4.1.1: Diseño TDA Vector

Vector hará este trabajo.

En el gráfico 4.1.1 vemos su diseño UML.

Operaciones básicas

- Construir

Definimos dos constructores, uno sin parámetros y otro con un parámetro entero positivo que será la longitud del vector.

Constructor sin parámetro

// Pre: -

// Post: crea un Vector con longitud en 0 y datos en nulo

Constructor con parámetro

// Pre: longitud es un entero positivo

// Post: crea un Vector con la longitud indicada y cada uno de los datos en nulo

- insertar

// Inserta un dato en la posición indicada

// Pre: la posición tiene que ser válida: $0 < pos \leq longitud$

// Post: el dato queda en la posición indicada

- eliminar

// elimina el dato dejando un nulo en dicha posición

// Pre: la posición tiene que ser válida: $0 < pos \leq longitud$

// Post: deja un nulo en esa posición

- tamaño
// Devuelve la longitud del vector
- elemento
// Devuelve el dato que está en la posición
// Pre: la posición tiene que ser válida: $0 < pos \leq longitud$
// Post: devuelve el dato de la posición
- redimensionar
// Estira o achica el largo del vector
// Pre: el largo debe ser un entero positivo
// Post: - si el largo es mayor completa con nulos
// - si el largo es menor trunca los datos

Implementación

Listado de código 4.1: Archivo de declaracion vector.h

```
#ifndef VECTOR_H_INCLUDED
#define VECTOR_H_INCLUDED

typedef int Dato;
const int NULO = 0;

class Vector {
private:
    // Atributos
    unsigned longitud;
    Dato* datos;

public:
    // Metodos
    // Constructor sin parametros
    Vector();

    // Constructor con parametros
    Vector(unsigned largo);

    // Destructor
    ~Vector();

    // insertar (la primera posicion es la 1)
    void insertar(Dato d, unsigned pos);

    // eliminar (la primera posicion es la 1)
    void eliminar(unsigned pos);
};
```

```

// devuelve la longitud del vector
unsigned tamano();

// devuelve el dato que esta en pos
Dato elemento(unsigned pos);

// redimensiona al vector
void redimensionar(unsigned largo);

private:
    // Coloca valores nulos al vector en las posiciones indicadas
    // Pre: 0 < d <= h <= longitud del vector
    // Post: en datos coloca nulos desde d hasta h inclusive
    void anular(unsigned d, unsigned h);

    // Copia los datos del vector pasado por parametro desde d hasta h
    // Pre: 0 < d <= h <= longitud del vector
    // Post: en datos coloca los valores del vector
    void copiar(Dato* vec, unsigned d, unsigned h);
};

#endif // VECTOR_H_INCLUDED

```

Listado de código 4.2: Archivo de definicion vector.cpp

```

#include "vector.h"

// Constructor sin parametros
Vector::Vector() {
    longitud = 0;
    datos = 0;
}

// Constructor con parametros
Vector::Vector(unsigned largo) {
    longitud = largo;
    datos = new Dato[largo];
    anular(1, largo);
}

// Destructor
Vector::~Vector() {
    if (longitud > 0)
        delete [] datos;
}

```

```

// insertar (la primera posicion es la 1)
void Vector::insertar(Dato d, unsigned pos) {
    datos[pos-1] = d;
}

// eliminar (la primera posicion es la 1)
void Vector::eliminar(unsigned pos) {
    datos[pos-1] = NULO;
}

// devuelve la longitud del vector
unsigned Vector::tamano() {
    return longitud;
}

// devuelve el dato que esta en pos
Dato Vector::elemento(unsigned pos) {
    return datos[pos-1];
}

// redimensiona al vector
void Vector::redimensionar(unsigned largo) {
    if (largo != longitud) {
        Dato* borrar = datos;
        datos = new Dato[largo];
        copiar(borrar, 1, largo);
        delete [] borrar;
        if (largo > longitud)
            anular(longitud + 1, largo);
        longitud = largo;
    }
}

// Coloca valores nulos al vector en las posiciones indicadas
void Vector::anular(unsigned d, unsigned h) {
    for (unsigned i = d-1; i < h; i++)
        datos[i] = NULO;
}

// Copia los datos del vector pasado por parametro desde d hasta h
void Vector::copiar(Dato* vec, unsigned d, unsigned h) {
    for (unsigned i = d-1; i < h; i++)
        datos[i] = vec[i];
}

```

Un posible uso se muestra en el algoritmo 4.3.

Listado de código 4.3: Archivo principal donde usa TDA Vector

```

#include <iostream>
#include "vector.h"

using namespace std;

int main()
{
    Vector vec(3);
    unsigned tam = vec.tamano();
    for (unsigned i = 1; i <= tam; i++)
        vec.insertar(i*i, i);
    for (unsigned i = 1; i <= tam; i++)
        cout << vec.elemento(i) << endl;

    vec.redimensionar(8);
    tam = vec.tamano();
    cout << "Despues de la redimension " << endl;
    for (unsigned i = 1; i <= tam; i++)
        cout << vec.elemento(i) << endl;
    vec.redimensionar(2);
    tam = vec.tamano();
    cout << "Despues de la segunda redimension " << endl;
    for (unsigned i = 1; i <= tam; i++)
        cout << vec.elemento(i) << endl;

    return 0;
}

```

La salida luego de correr el programa anterior es la siguiente:

```

1 4 9
Despues de la redimension
1 4 9 0 0 0 0 0
Despues de la segunda redimension
1 4

```

Nota: por ahorro de papel pusimos los datos uno al lado del otro, pero los imprime uno debajo del otro.

Algunas aclaraciones:

- Utilizamos un *typedef Dato* para que pueda cambiarse el tipo de dato del vector por un *char*, un *float*, etc. Por este motivo siempre trabajamos con un tipo *Dato* en lugar de un *int*. Esta es la primera y tibia aproximación a la genericidad.
- También debemos definir una constante NULO ya que el nulo puede variar no solo según el tipo de dato, sino según el tipo de problema con

el que se trabaja. Por ejemplo, si tenemos un vector de 365 posiciones y los datos que almacenamos en cada celda representan la cantidad de agua que cayó en ese día en la región, un valor 0 puede representar que no llovió ese día. Podríamos querer tener algún valor que indique que ese dato aún no se cargó o no se dispone de ese registro, por lo cual podríamos indicarlo con un valor negativo, por ejemplo -1 .

Sin embargo si los datos representan la temperatura mínima registrada en ese día, un valor de 0 o -1 no indicarían nulos o datos inválidos, ya que son temperaturas admisibles. En ese caso podríamos usar un NULO con valor -274 ya que es una temperatura inválida en la escala Celsius.

- En el archivo *.h* no escribimos todas las pre y poscondiciones porque lo habíamos hecho en la sección anterior.
- Los métodos *anular* y *copiar* los pusimos como privados porque no estamos interesados en que el usuario los emplee. Pero son útiles para modularizar porque otros métodos los pueden utilizar, como el método de *redimensionar* y el constructor de copia que veremos enseguida.
- En el método *redimensionar*, si las longitudes son iguales no hace nada.
- Luego de la primera redimensión, como el vector se “estira” vemos que mantiene los datos y completa con nulos.
- Luego de la segunda redimensión, como el vector se achica observamos que trunca los datos.

Constructor de copia

Si bien la implementación anterior funciona correctamente para el ejemplo dado, no podríamos escribir sentencias como la siguiente:

```
Vector vec2 = vec1;
```

que es equivalente a

```
Vector vec2(vec1);
```

El motivo es el que explicamos en el capítulo anterior: el nuevo objeto *vec2* copia todos los datos del otro objeto *vec1*, y como uno de sus datos es un puntero, copia la misma dirección para guardar sus datos allí en lugar de copiar estos datos en otro lugar. Esto sucede también cuando un objeto de tipo Vector se pasa por parámetro a una función, es por esto que debemos escribir un constructor de copia.

El constructor de copia debe llevar un parámetro del mismo tipo que se está copiando, en este caso es Vector. Dicho parámetro debe estar pasado

por referencia, de lo contrario necesitará copiarlo y se produciría un llamado recursivo sin fin. Además se pasa como constante como garantía de que el objeto solo se utilizará para copiarlo pero no para modificarlo. El código queda como vemos a continuación:

```
// Constructor de copia
Vector::Vector(const Vector & vec) {
    longitud = vec.longitud;
    if (longitud > 0) {
        datos = new Dato[longitud];
        copiar(vec.datos, 1, longitud);
    }
    else
        datos = 0;
}
```

4.2. Punteros void

En los ejemplos que vimos antes utilizamos un *int* como *Dato*, por eso definimos

```
typedef int Dato;
```

y luego utilizamos *Dato* como tipo en donde lo necesitáramos, ya sea como parámetro o como un tipo que devuelve algún método. Si necesitáramos utilizar un vector de *char*, sólo deberíamos cambiar la palabra *int* por *char* y no tendríamos ningún problema. Sin embargo, se puede dar que necesitemos un vector de *char* y otro de *int*. En este punto se comienzan a complicar las cosas, ya que necesitaríamos dos tipos de vectores distintos. También, podríamos necesitar un vector de elementos más complejos, como los datos de un empleado (legajo, nombre, dirección, etc.).

No sería bueno en ningún método copiar o devolver un dato muy grande y complejo. Por otro lado, ¿si necesitamos que alguna primitiva nos devuelva un dato especial, por ejemplo, indicando que el dato es inválido o no se encuentra? Supongamos que el vector guarda los datos de varios empleados como habíamos comentado, y queremos que nos devuelva todos los datos del empleado con número de legajo 123. La idea es realizar una búsqueda en el vector y devolver un tipo *Dato* el cual será un struct con toda la información. El problema lo tendremos si no encuentra ese legajo.

La solución a este último problema será devolver una referencia o puntero al dato en lugar del propio dato, de esta forma, cuando no se encuentra un elemento se puede devolver un valor nulo. Pero el otro inconveniente, el de manejar datos genéricos, no lo podemos resolver de manera trivial.

C++ al igual que otros lenguajes nos permite utilizar un puntero genérico

(o puntero a nada), de tipo *void*. Void significa vacío, por lo que se puede interpretar que un puntero a *void* carece de tipo, no apunta o no está atado a ningún tipo en especial, por lo que puede apuntar a cualquier tipo. De esta forma, estos punteros serán genéricos, por lo que servirían para cualquier tipo de dato.

El problema con este tipo de implementación es que debemos castearlos cuando necesitemos utilizarlos.

La implementación prácticamente no cambia en nada, salvo por el tipo de dato que ahora será un puntero a void:

```
typedef void* Dato;
```

Pero, hay que tener cuidado en el uso porque debemos recordar dos cosas:

- Tanto los datos que se pasan en forma de parámetros como los devueltos son, ahora, punteros.
- Los punteros a los datos que nos devuelvan los métodos se deben castear para poder utilizarlos.

El siguiente código es un ejemplo de uso de la clase *Vector* con puntero *void*. Veremos que los podemos utilizar para distintos tipos de datos (en este caso enteros y strings).

Listado de código 4.4: Archivo de ejemplo de uso Vector con dato void*

```
#include <iostream>
#include <string>
#include "vector.h"

using namespace std;

int main()
{
    cout << "==== Vector de enteros ==== " << endl;

    int a = 7, b = 8, c = 9, d = 15;
    Vector vi(3);

    // No se pasan los datos sino sus direcciones
    vi.insertar(&a, 1);
    vi.insertar(&b, 2);
    vi.insertar(&c, 3);

    Vector vi2;
    vi2 = vi;
    vi2.redimensionar(4);
```

```

vi2.insertar(&d, 4);

for (unsigned i = 1; i <= vi.tamano(); i++)
    // El primer asterisco es para acceder al dato
    // (recordar que devuelve un puntero al dato)
    // El "(int*)" es el casteo porque el
    // puntero que devuelve es genérico
    cout << *(int*)vi.elemento(i) << endl;

cout << "==== Otro vector de enteros =====" << endl;
for (unsigned i = 1; i <= vi2.tamano(); i++)
    // El primer asterisco es para acceder al dato
    // (recordar que devuelve un puntero al dato)
    // El "(int*)" es el casteo porque el
    // puntero que devuelve es genérico
    cout << *(int*)vi2.elemento(i) << endl;

// Ahora, utilizamos un vector con otros
// tipos de datos, como strings
cout << "==== Vector de strings =====" << endl;

string s1 = "siete", s2 = "ocho", s3 = "nueve", s4 = "diez";
Vector vs(4);

vs.insertar(&s1, 1);
vs.insertar(&s2, 2);
vs.insertar(&s3, 3);
vs.insertar(&s4, 4);

for (unsigned i = 1; i <= vs.tamano(); i++)
    cout << *(string*)vs.elemento(i) << endl;

return 0;
}

```

La salida será:

```

===== Vector de enteros =====
7
8
9
===== Vector de strings =====
siete
ocho
nueve
diez

```

Nada nos impediría escribir y ejecutar la siguiente porción de código:

```
int a = 7, b = 8, c = 3;
string s1 = "hola";
Vecor vec(4);
vec.insertar(&a, 1);
vec.insertar(&s1, 2);
vec.insertar(&b, 3);
vec.insertar(&c, 4);
```

De esta forma estaríamos poniendo en el mismo vector punteros a enteros y un puntero a un *string*. Si bien esto sería aceptado y no arrojaría ningún error, debemos recordar que, cuando deseemos recuperar el elemento que está en la segunda posición debemos castearlo como (*string**) y no como (*int**) como al resto de los elementos. Obviamente, uno no puede estar recordando o llevando nota sobre qué tipos de elementos están en cada uno de los nodos, por lo que esta solución no es recomendable. Sin embargo es importante tener en cuenta lo siguiente: la porción de código escrita anteriormente podría deberse a un error en la programación. Es decir, el programador podría haber deseado insertar el puntero de *s1* en otro vector, uno de strings y no en el mismo vector donde están almacenados los enteros. Sin embargo el compilador no da ninguna advertencia sobre diferencias de tipos, ¡porque no la hay! Las posiciones del vector están preparadas para aceptar direcciones de cualquier clase de elementos, como le pasamos una dirección lo toma como algo correcto, sin importarle si esa dirección es a un string, a un entero o a cualquier otro objeto.

Para salvar este problema, es decir, para poder tener una verificación de tipos, veremos las plantillas (templates).

4.3. Tipos genéricos

Los vectores, al igual que otras estructuras, siempre operan de la misma manera sin importarle el tipo de dato que estén albergando. Por ejemplo, agregar un elemento en cierta posición o consultar qué elemento está en la posición 3 debe tener el mismo algoritmo ya sea que el elemento fuera un *char*, un *float* o una estructura. Sin embargo, cuando uno define los métodos debe indicar de qué tipo son los parámetros y de qué tipo van a ser algunas devoluciones.

No tendría sentido repetir varias veces el mismo código solo para cambiar una palabra: *char* por *float* o por *Empleado*, por ejemplo. En el apartado anterior vimos que esto se solucionaba utilizando un puntero genérico (*void**). Sin embargo, un problema se resolvía pero se introducían algunos nuevos, como la falta de control en los tipos. Pero este no es el único inconveniente, ya que uno podría decidir prescindir del control de tipos a cambio de un

mayor cuidado en la codificación. Otro problema es que un puntero a *void* no nos permite utilizar operadores, ya que el compilador no sabe a qué tipo de dato se lo estaría aplicando. Por ejemplo, el operador “+” actúa en forma muy distinta si los argumentos son números enteros (los suma) que si fueran strings (los concatena).

En el ejemplo de la sección anterior, en la función *main* uno tenía en claro qué tipo de dato estaba colocando en la lista (primero se colocaron direcciones de enteros y, en otra, strings) pero, en el momento de ingresar al vector pierden esa identidad, dado que se toman como direcciones a *void*. A partir de ahí uno pierde el uso de operadores, como el + o el − y, también pierde el uso de otros métodos.

Un enfoque, teniendo en la mira el objetivo de la programación genérica, son las plantillas (templates). Los templates, en lugar de reutilizar el código objeto, como se estudiará en el polimorfismo, reutiliza el código fuente. ¿De qué manera? Con parámetros de tipo no especificado. Se debe agregar, antes de definir la clase la siguiente sentencia

```
template < typename Dato > // se agrega
class Vector {
    ...
};
```

Luego, cuando vayamos a utilizar el vector debemos indicarle de qué tipo es. Por ejemplo:

```
Vector< char > vc; // Vector de char
Vector< string > vs; // Vector de strings
Vector< string > vs2; // Otro vector de strings
```

Cuando se compila, se resuelve el problema del tipo indefinido en un proceso que se llama especialización. ¿Qué es lo que hace? En el ejemplo que acabamos de dar, el compilador observa que necesita un vector para un tipo *char*, por lo que genera, automáticamente, el código por nosotros. Es decir, reemplaza *Dato* por *char* en todas sus ocurrencias, generando el código completo. Luego, cuando compile la segunda línea, hace lo mismo pero con strings, es decir, genera nuevamente todo el código pero reemplazando *string* en donde figure *Dato*. Por supuesto, en la tercera línea, en la declaración de *vs2*, no vuelve a generar código porque ya tiene el código generado para strings.

¿Cuáles son las ventajas de los templates?

- Generan código en forma automática por nosotros.
- Hay verificación de tipos. Ya no podría, en una lista de enteros, agregar un string como en el ejemplo de la sección anterior. Esto, lejos de ser

una molestia, es una ventaja, porque es una seguridad que el compilador verifique que los tipos sean correctos.

- Como la definición de tipos se resuelve en tiempo de compilación, los ejecutables son más rápidos que los generados utilizando herencia y polimorfismo.

Algunas características a tener en cuenta

- Un template no existe hasta que se instancia. Es decir, hasta que no se crea algún objeto indicando el tipo no genera ningún código.
- Una consecuencia del punto anterior es que se debe generar todo el código en el archivo punto *h*. Por lo tanto no se separarán las declaraciones por un lado (archivo punto *h*) y las definiciones por otro (archivo punto *cpp*). De todas formas, es recomendable seguir separando las declaraciones de las definiciones, aunque lo incluyamos todo en el mismo archivo. De lo contrario se generarían todos métodos inline, que tienen la ventaja de ser más rápidos en la ejecución pero generan un código mucho mayor, ya que se copian cada vez que se invocan.
- Otra característica del uso de templates es que se necesita indicar el parámetro cada vez que se indique la clase cuando se definen los métodos (esto se ve más claramente en el código que figura en el apéndice, implementado en A.1 y en A.2 un ejemplo de uso).
- Una desventaja es que tuvimos que sacar el método *anular* y la constante NULO dado que el valor nulo es distinto según el tipo de dato y el problema, esto se soluciona con el polimorfismo.
- Finalmente, cabe destacar que es indistinto utilizar la palabra *typename* como *class*. Pero se prefiere la primera ya que el parámetro no tiene por qué ser un objeto sino cualquier tipo.

4.4. Herencia

En el capítulo anterior hemos visto con un ejemplo, el de *Transporte*, cómo la herencia juega un papel importante en la POO. Pero podemos preguntarnos ¿por qué es necesario utilizarla? La respuesta es sencilla: para ahorrarnos de escribir mucho código, el objetivo de reutilizar lo que ya escribimos siempre tiene que estar presente.

La terminología que se utiliza en herencia es indistintamente la siguiente:

Clase de la cual hereda	Clase que hereda
Superclase	Subclase
Padre	Hijo
Antecesor	Sucesor
Clase superior	Especialización

También decimos que la clase hijo *extiende* a la clase padre porque en general tiene más atributos y/o métodos que su antecesora.

4.4.1. Ejemplos de herencia

Generemos un nuevo proyecto con dos clases, a una la llamaremos ClaseA y a la otra, ClaseB. ClaseA tendrá un solo atributo x de tipo entero. Al cual se accederá por medio de los métodos asignarX y obtenerX. El código será el siguiente:

```
// ClaseA
class ClaseA {
private:
    // Atributos
    int x;

public:
    // Metodos
    void asignarX(int x);
    int obtenerX();
};
```

ClaseB heredará de la ClaseA y tendrá otro atributo y de tipo entero además de x heredado de ClaseA. También se accederá al atributo y por medio de los métodos cambiarY y obtenerY. La herencia se indica en la cabecera de la clase por medio de un dos puntos (:) el tipo de herencia [public, protected, private] y el nombre de la clase de la cual hereda, en este caso ClaseA.

```
// ClaseB
class ClaseB: public ClaseA {
private:
    // Atributo
    int y;

public:
    // Metodos
    void asignarY(int y);
    int obtenerY();
};
```

La ClaseB tiene dos atributos:

- el atributo entero *y* propio de la *ClaseB*, representa la extensión de *ClaseA*.

Nota: en algunos lenguajes se utiliza la palabra *extend* para indicar la herencia, ya que las clases que heredan extienden a sus antecesoras.

- el atributo entero *x* heredado de la *ClaseA*

Por otro lado, tiene cuatro métodos:

- *asignarY* y *obtenerY* propios de la *ClaseB*
- *asignarX* y *obtenerX* heredados de la *ClaseA*

Por lo tanto, podemos utilizar ambas clases como lo venimos haciendo:

```
// Uso de las clases ClaseA y ClaseB
int main() {
    ClaseA a;
    ClaseB b;
    a.asignarX(8);
    cout << "a tiene un atributo x que vale " << a.obtenerX() << endl;
    b.asignarX(3);
    b.asignarY(7);
    cout << "b tiene un atributo x que vale " << b.obtenerX() << endl;
    cout << "b tiene un atributo y que vale " << b.obtenerY() << endl;
    return 0;
}
```

La salida del código anterior será:

```
a tiene un atributo x que vale 8
b tiene un atributo x que vale 3
b tiene un atributo y que vale 7
```

Formas de heredar

Las formas de heredar son tres: public (la que en general se utiliza), protected y private.

¿Cuáles son las diferencias?

- La herencia pública respeta la forma de acceder definida por su clase antecesora. Es decir, lo que sea público queda como público, lo que sea protegido sigue siendo protegido y lo que es privado queda como privado, por lo cual no se tiene acceso de las clases hijas.
- La herencia protegida es similar a la pública pero los métodos y atributos públicos los convierte en protegidos.
- La herencia privada convierte todo a privado (en general no se utiliza).

Lo vemos en los cuadros 4.4.1.

Atributos/métodos	Hereda
public	public
protected	protected
private	no tiene acceso

(a) Herencia pública

Atributos/métodos	Hereda
public	protected
protected	protected
private	no tiene acceso

(b) Herencia protegida

Atributos/métodos	Hereda
public	private
protected	private
private	no tiene acceso

(c) Herencia privada

Tabla 4.4.1: Tipos de herencia

4.4.2. La Construcción en la Herencia

Si a la *ClaseA* le agregáramos un constructor debemos, forzosamente, utilizarlo en la *ClaseB* con inicializadores. Por ejemplo, en el archivo *.h* agregamos un constructor en cada clase:

```
// Constructor de ClaseA
ClaseA(int x);
// Constructor de ClaseB
ClaseB(int x, int y);
```

Como *ClaseA* tiene un solo atributo, su constructor lleva un solo parámetro, en cambio, en *ClaseB* tenemos dos atributos: uno heredado de *ClaseA*, por lo que recibe dos parámetros. En el archivo *.cpp*, donde definimos los constructores escribimos:

```
// Constructor ClaseA
ClaseA::ClaseA(int x) {
    this->x = x;
}

// Constructor ClaseB
ClaseB::ClaseB(int x, int y) : ClaseA(x) {
    this->y = y;
}
```

El constructor de la superclase debe llamarse antes que cualquier instrucción, por este motivo deben usarse inicializadores, es decir, instrucciones que van antes del cuerpo del método. Esto es así porque se crea, en primer lugar, la porción del objeto de la clase ancestro y luego la de los herederos.

Con estos constructores, el ejemplo de uso visto antes quedaría:


```

int main() {
    ClaseA a(8);
    ClaseB b(3, 7);
    cout << "a tiene un atributo x que vale " << a.obtenerX() << endl;
    cout << "b tiene un atributo x que vale " << b.obtenerX() << endl;
    cout << "b tiene un atributo x que vale " << b.obtenerY() << endl;
    return 0;
}

```

4.4.3. Ocultación de Atributos

Supongamos que el atributo x de la *ClaseA* es protegido en lugar de privado, por lo que la *ClaseB* puede acceder en forma directa. Y supongamos también, aunque no sea recomendable hacerlo, que a *ClaseB* le agregamos otro atributo de tipo entero que también se llama x . En algún método *obtenerX* de la *ClaseB* al hacer

```
return x;
```

devolverá el valor del x perteneciente a la *ClaseB* siempre que trabajemos con un objeto de tipo *ClaseB*, obviamente. En estos casos decimos que el atributo x de *ClaseB* “oculta” al atributo x de *ClaseA*. De esta forma, si bien el atributo x de la *ClaseB* que hereda de *ClaseA* seguiría existiendo, se estaría ocultando mediante el nuevo atributo x .

¿Cómo acceder al atributo de la clase padre? Escribiendo

```
ClaseA::x;
```

En estos momentos empieza a cobrar sentido el concepto de *espacios de nombres* y el operador de ámbito, dado que podemos tener los mismos nombres pero en distintos espacios.

4.4.4. Redefinición

4.5. Polimorfismo

Capítulo 5

Listas

Una lista es una estructura de datos lineal flexible, ya que puede crecer a medida que se insertan nuevos elementos o acortarse cuando se borren. En principio, los elementos deben ser del mismo tipo, es decir, datos homogéneos pero utilizando polimorfismo, podremos trabajar con elementos distintos siempre y cuando hereden de algún antecesor común.

La estructura lista comprende, a su vez, otras tres estructuras:

- Listas propiamente dichas,
- Pilas.
- Colas.

En el caso de listas propiamente dichas, los elementos pueden insertarse en cualquier posición, ya sea al principio, al final o en alguna posición intermedia. Lo mismo sucede con el borrado. Hay dos casos especiales de listas que son las pilas y las colas.

Tanto en una pila como en una cola, la inserción de un elemento es siempre al final. Pero el borrado en una pila es al final mientras que en una cola es al principio.

Hay que tener en cuenta que toda pila y toda cola es una lista, pero no sucede al revés. Así como un cuadrado es un caso particular de un rectángulo (es un rectángulo que tiene sus lados de igual longitud), las pilas y colas son casos especiales de las listas, tienen ciertas restricciones que las listas no. Cuando hablemos de listas, a secas, nos referiremos a las propiamente dichas.

5.1. Propiedades y representación

Matemáticamente, una lista es una secuencia ordenada de n elementos, con $n \geq 0$. En el caso de que n sea 0, la lista se encontrará vacía. Una lista

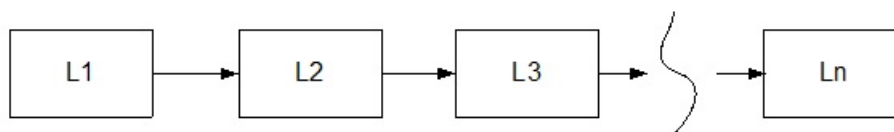


Gráfico 5.1.1: Representación de una lista

de n elementos la podemos representar de manera gráfica, como vemos en la figura 5.1.1, o de la siguiente forma:

$$L_1, L_2, L_3, \dots L_n$$



Nota. En la representación gráfica de la figura 5.1.1, las flechas no tienen por qué corresponder a punteros, aunque podrían serlo. Pero se deben tomar como la indicación de una secuencia ordenada.

5.2. Listas propiamente dichas

Como en todo tipo de dato abstracto debemos definir un conjunto de operaciones que trabajen con el tipo *Lista*. Estas operaciones no siempre serán adecuadas para cualquier necesidad. Por ejemplo, si queremos insertar un cierto elemento x debemos decidir si la lista permite o no tener elementos duplicados.

Las operaciones básicas que deberá manejar una lista, como cualquier tipo de estructura de datos, son las siguientes:

- Insertar un elemento en la lista (transformación).
- Borrar un elemento de la lista (transformación).
- Obtener un elemento de la lista (observación).

Las dos primeras operaciones transformarán la lista, ya sea agregando un elemento o quitándolo. La última operación solo será de inspección u observación, consultando por un determinado elemento pero sin modificar la lista. Obviamente, además se necesitará, como en todo TDA, una operación de creación.

A continuación, analicemos con mayor profundidad cada una de las operaciones mencionadas.

Insertar

La operación insertar puede tener alguna de estas formas:

- a. insertar (*x*)
- b. insertar (*x*, *posicion*)

Con respecto al primer caso tenemos distintas variantes:

- La lista podría mantenerse ordenada por algún campo clave, con lo cual, se compararía *x.clave* con los campos clave de los elementos de la lista, insertándose *x* en el lugar correspondiente.
- Por el contrario, podría ser una lista en la que no interesa guardar ningún orden en especial, por lo que la inserción podría realizarse en cualquier lugar, en particular podría ser siempre al principio o al final, solo por comodidad en la implementación.

En el segundo caso, *posicion* es el lugar en donde debe insertarse el elemento *x*. Por ejemplo, en una lista

$$L_1, L_2, L_3, \dots L_n$$

`insertar (x, 1);`
 produciría el siguiente resultado:

$$x, L_1, L_2, L_3, \dots L_n$$

en cambio
`insertar (x, n + 1);`
 agregaría a *x* al final de la lista y la siguiente sentencia
`insertar (x, 3);`

$$L_1, L_2, x, L_3, \dots L_n$$

dejaría a *x* en la tercera posición.

En esta operación se debe decidir qué hacer si *posicion* supera la cantidad de elementos de la lista en más de uno, es decir, si *posicion* > *n* + 1.

Las decisiones podrían ser varias, desde no realizar nada, es decir, no insertar ningún elemento; insertarlo al final o estipular una precondition del método en la que esta situación no pueda darse nunca. De esta última forma, se transfiere la responsabilidad al usuario del TDA, quien debería cuidar que nunca se dé esa circunstancia. Otro enfoque es manejarlo con excepciones. Por el momento no contemplaremos estos casos para no desviar la atención de nuestro objetivo.

Borrar

En el borrado de un elemento pasa una situación similar a la que se analizó en la inserción. Las operaciones podrían ser:

- a. borrar (x)
- b. borrar (*posicion*)

La primera, borra el elemento x de la lista. Si no lo encontrara podría, simplemente, no hacer nada. En la segunda opción se borra el elemento de la lista que se encuentra en *posicion*. Las decisiones en cuanto a si *posicion* es mayor que n (la cantidad de elementos de la lista) son las mismas que en el apartado anterior, es decir, no hacer nada, borrar el último elemento o lanzar una excepción.

Obtener

Este método debe recuperar un elemento determinado de la lista. Las opciones son las siguientes:

- a. obtenerElemento ($x.clave$)
- b. obtenerElemento (*posicion*)

En la opción 1, se tiene una parte del elemento, solo la clave, y se desea recuperarlo en su totalidad. En cambio, en la opción 2, se desea recuperar el elemento que está en *posicion*.

En ambas opciones, el resultado debe devolverse. Es decir, el método debería devolver el elemento buscado. Sin embargo, esto no sería consistente en el caso de no encontrarlo. ¿Qué devolvería el método si la clave no se encuentra? Por otro lado, devolver un objeto no siempre es lo ideal, ya que se debería hacer una copia del elemento que está en la lista, lo que sería costoso y difícil de implementar en muchos casos. Por estos motivos se prefiere devolver un puntero o referencia al objeto de la lista. De esta forma se ahorra tiempo, espacio y, si el elemento no se encontrara en la lista, se puede devolver un valor nulo.

Por supuesto pueden haber más operaciones que las mencionadas, pero estas son las operaciones básicas, a partir de las cuales se pueden obtener otras. Por ejemplo, se podría desear tener un borrado completo de la lista, pero esta operación se podría realizar llamando al borrado del primer elemento hasta que la lista quede vacía.

5.3. Pilas

Una pila es un caso especial de lista que tiene las siguientes dos restricciones:

- a. La inserción de un elemento es solo al final.
- b. El borrado de un elemento es también solo al final.

Decimos que una pila responde a una estructura de tipo LIFO (last in, first out). Último en entrar primero en salir.

¿Con qué finalidad se necesita una estructura con estas características?

Hay determinados algoritmos que se modelan correctamente con una pila, por ejemplo, cuando vamos haciendo cambios en un documento, la mayoría de los editores de texto van colocando esos cambios en una estructura de tipo pila. Cuando presionamos las teclas para deshacer esos cambios, comienzan por el último, es decir, los van desapilando.

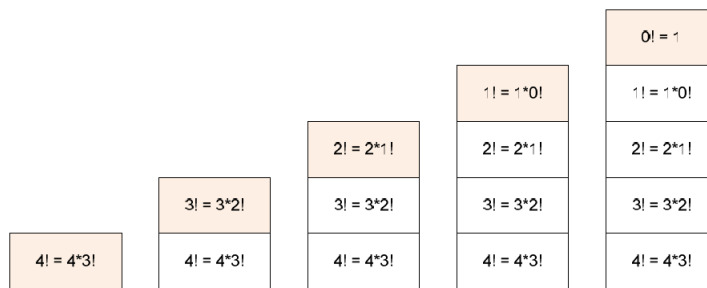
Otro uso que se le da a una pila es en los algoritmos que utilizan recursividad. Veamos cómo funciona un algoritmo recursivo que calcula el factorial de 4. Para su cálculo, el factorial de 4 lo expresamos como 4 por el factorial del número anterior: 3. De esta forma, 3 ingresa en la estructura pila, en donde se aplica el mismo procedimiento. Este algoritmo continúa hasta llegar a 0, que será el caso base de la recursividad. Cuando comienza a resolver las incógnitas, lo debe ir haciendo de atrás para adelante. En el gráfico 5.3.1 vemos en la parte *a*) cómo se va generando la pila ante cada llamado recursivo. Luego, en la parte *b*) observamos cómo se va desapilando una vez alcanzado el caso base.

Hay que notar que los elementos que están debajo de la cima están a la espera de los resultados de los elementos superiores. Es decir, el elemento que está en la posición 1 necesita que se resuelva el de la posición 2 y, así, sucesivamente.

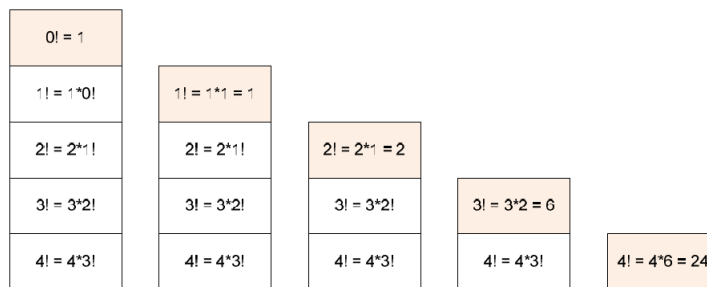
5.4. Colas

Una cola es un caso especial de lista que tiene las siguientes dos restricciones:

- a. La inserción de un elemento es solo al final.
- b. El borrado de un elemento es solo del principio.



(a) Llamados recursivos: apilar



(b) Resolución de los llamados recursivos: desapilar

Gráfico 5.3.1: Uso de pila en recursividad

Decimos que una cola responde a una estructura de tipo FIFO (first in, first out). Primero en entrar primero en salir.

Las colas modelan muchas situaciones de la vida diaria, y tienen el mismo sentido de la palabra que en el uso habitual. Por ejemplo, cuando vamos a un banco sacamos un número y esperamos nuestro turno. El hecho de sacar el número nos coloca al final de la cola, y van atendiendo a las personas que llegaron antes. Cada vez que atienden a una persona decimos que sale de la cola.

Un ejemplo de uso de cola en la informática es el de varias máquinas conectadas a una impresora, en donde los distintos usuarios envían a imprimir archivos desde sus terminales. La solicitud ingresa a una cola en donde se van procesando de una en una cada solicitud, de lo contrario, el resultado sería una mixtura ininteligible de las distintos archivos.

Por supuesto, en su forma más sencilla una cola respeta siempre el orden de llegada, aunque hay ciertas variantes en donde se pueden determinar distintas prioridades que alteren dicho orden. Por ejemplo, una solicitud de impresión tendrá prioridad más alta que el proceso de escaneo de toda la máquina en busca de virus. Un cliente premium de un Banco tendrá prioridad más alta que un cliente que no lo es, por lo que será atendido en primer lugar aunque haya llegado después. En estos casos tenemos una estructura que llamamos cola de prioridades que veremos en otro capítulo.

5.5. Implementaciones

¿Cómo se lleva a la práctica toda la teoría? Hay distintas maneras de implementar una lista (en el sentido amplio: lista, pila y cola). Básicamente, podremos dividir la implementación en dos grupos: estructuras estáticas y estructuras dinámicas. Vayamos viendo en cada tipo de lista las diferentes implementaciones. Comenzamos por la implementación de una Pila porque es el caso más simple.

5.5.1. Implementaciones de Pila

Estructuras estáticas

Una implementación sencilla es con un vector de elementos, pueden ser tipos simples, structs u otros objetos, donde cada posición del vector contiene un elemento. Este vector se define en forma estática, es decir, su tamaño debe ser definido en tiempo de compilación. Por ejemplo, una pila cuyos elementos sean caracteres pueden verla en la figura 5.5.1. En este ejemplo definimos

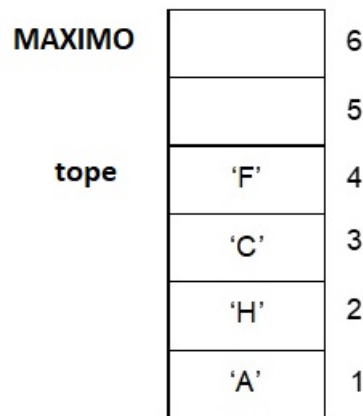


Gráfico 5.5.1: Implementación estática de una pila

un vector de 6 posiciones ($\text{MAXIMO} = 6$) y los elementos que se fueron insertando en la pila de forma secuencial son 'A', 'H', 'C', 'F'.



Nota. Estamos asumiendo que no hubo otras operaciones intermedias. Por ejemplo, podría haber ingresado el carácter 'A', luego 'M', luego desapilar un elemento, por lo que 'M' sale de la estructura, y luego continuar apilando 'H', 'C' y 'F' o alguna otra de las infinitas combinaciones que dejarían a la pila en el estado que vemos en la figura.

El atributo *tope* es el que tiene la posición del último elemento ingresado, por lo tanto este atributo valdrá 0 en el momento de la creación de la pila, luego valdrá 1 al colocar un elemento, etc.



Nota. El índice que figura al costado significa que 'A' es el primer elemento de la pila, 'H' el segundo, etc. No es el índice del vector, ya que en C++ los índices comienzan en 0.

Hay que decidir si el método *quitar* además de sacar el elemento de la pila lo devuelve.

En las implementaciones estáticas debemos controlar no excedernos del tamaño del vector. Una implementación la podemos ver en el apéndice A, en los algoritmos A.3, A.4 y A.5.

Algunas aclaraciones con respecto al ejemplo de implementación

- En primer lugar, el nombre de la clase `PilaEstatica`, se utiliza solo con fines didácticos y para diferenciar de otras implementaciones que se harán más adelante, pero no es recomendable utilizar nombres de clases que indiquen la forma de implementación. En este caso, lo correcto sería haber utilizado `Pila` a secas.

¿Por qué no debemos indicar el tipo de implementación en el nombre?
Por varias razones.

- Una de las razones, es que el paradigma de la programación orientada a objetos nos habla de ocultamiento de la implementación y abstracción de datos. Es decir, utilizar algo conociendo su interfaz pero sin preocuparnos cómo está hecho internamente. Por ejemplo, cuando ponemos un DVD en una reproductora y presionamos *play*, sabemos que podremos ver la película que colocamos, pero no nos interesa saber el mecanismo interno que hace la reproductora para reproducirla. Conocer el detalle interno de cada artefacto que utilizamos y estar pendientes del mismo nos confundiría y haría nuestra vida muy complicada.
 - Por otro lado, nos veríamos tentados a “meter mano” dentro de los artefactos lo cual no sería conveniente. Por algo las garantías se invalidan cuando el usuario intenta arreglar el dispositivo por su propia cuenta.
 - Otro motivo es que el día de mañana podríamos querer cambiar la implementación que actualmente estamos utilizando por otra, por ejemplo, por una pila dinámica. De esta forma tendríamos que cambiar en todo el código las indicaciones que digan `PilaEstatica` por `PilaDinamica`, lo cual no tiene sentido.
- Esta implementación es muy sencilla pero no muy útil. No tendría sentido utilizar una pila para almacenar 10 caracteres. Si bien la constante `MAXIMO` puede cambiarse fácilmente de 10 a 100 o 1000, esta implementación seguiría siendo estática, por lo que corremos el riesgo de quedarnos cortos con el tamaño o desperdiciar memoria.
 -
 - Nótese que, como la implementación es estática y los datos que se guardan también lo son, no hay necesidad de liberar memoria, por lo cual, el destructor no realiza nada (se podría haber dejado el destructor de oficio que provee el lenguaje).

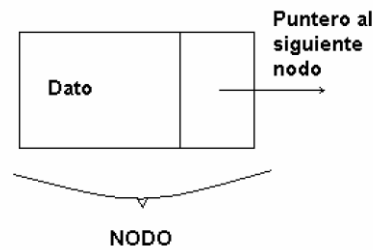


Gráfico 5.5.2: Nodo

- La diferencia entre los métodos *consultar* y *sacar* es que el primero devuelve el dato pero no lo quita de la Pila, por este motivo el atributo *tope* no se modifica. En cambio el método *sacar*, además de devolver el dato lo quita de la Pila. Este quitar es relativo, la realidad es que el dato sigue permaneciendo en la estructura vector pero ya no lo podremos consultar, dado que el atributo *tope* se decrementa. Además, si luego ejecutamos una operación *agregar*, sobrescribe el dato anterior con el nuevo.

Estructura dinámica

Una estructura dinámica será lo que en general utilizaremos para implementar una Pila. La idea es almacenar los datos en nodos, los cuales se encuentran enlazados entre sí, como si fueran eslabones de una cadena. Comenzaremos viendo la estructura de un Nodo.

Nodos

Un nodo es una estructura que tendrá los siguientes datos:

- El propio elemento que deseamos almacenar o un puntero a ese dato.
- Uno o más enlaces a otros nodos (punteros, con las direcciones de los nodos enlazados).

En el caso más simple, que es el que aplica para una Pila, el nodo tendrá el elemento y un puntero al siguiente nodo de la pila como vemos en el gráfico 5.5.2.

Pila Simplemente Enlazada

Asumiendo que una Pila es un caso particular de Lista, decimos que una Lista Simplemente Enlazada (LSE) consistirá en una sucesión de nodos que

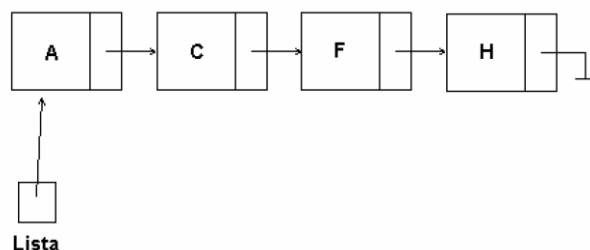


Gráfico 5.5.3: Lista simplemente enlazada

se irán enlazando cada uno con el siguiente. Por lo tanto, para implementarla solo necesitamos un puntero al primer nodo de esa lista, ya que el resto de los nodos se enlazarán mediante sus propios punteros. El último puntero, apuntará a nulo y se representa como un “cable a tierra” como vemos en el gráfico 5.5.3.

Ahora bien, ¿qué atributos debemos guardar en una PSE?

Nos alcanza solo con tener un puntero al primer nodo, supongamos que este se llame *primero*. Por lo tanto, al crear un objeto de tipo Pila, el atributo *primero* debe valer 0 (apuntar a nulo). Al agregar el primer dato, debemos crear un Nodo para almacenarlo allí, y luego hacer que primero apunte a dicho Nodo (ver figura 5.5.4, parte a). Cuando agreguemos un nuevo dato, debemos crear otro Nodo y agregarlo al final (5.5.4 parte b). Para agregarlo al final tendremos que ir al primer Nodo y cambiar su puntero al siguiente. Al agregar un dato más, esta operación se repite, pasando por el primer nodo y luego por el segundo hasta llegar al final de la pila.

Es decir que si la pila tiene n datos debemos recorrer n nodos para llegar al final e insertar allí el nuevo dato a ingresar. Por otro lado, para su consulta o eliminación debemos hacer lo mismo. Adelantándonos al estudio del orden de los algoritmos, vemos que esta implementación es ineficiente. Una solución que se nos ocurre es agregar un nuevo atributo que sea *ultimo* y apunte al final de la Pila, quedando como en la figura 5.5.5.

Sin embargo, esta implementación agrega complejidad innecesaria en el desarrollo. Una mejor decisión es pensar que la lista está orientada al revés, por lo que el único atributo, en lugar de apuntar al primer Nodo, siempre apunta al último. De esta forma, el agregado siempre será al principio (que ahora es el último), la consulta y el borrado, también.

Esta implementación se puede apreciar en el gráfico 5.5.6 y el código en los algoritmos A.6, A.7, A.8, A.9 y A.10.

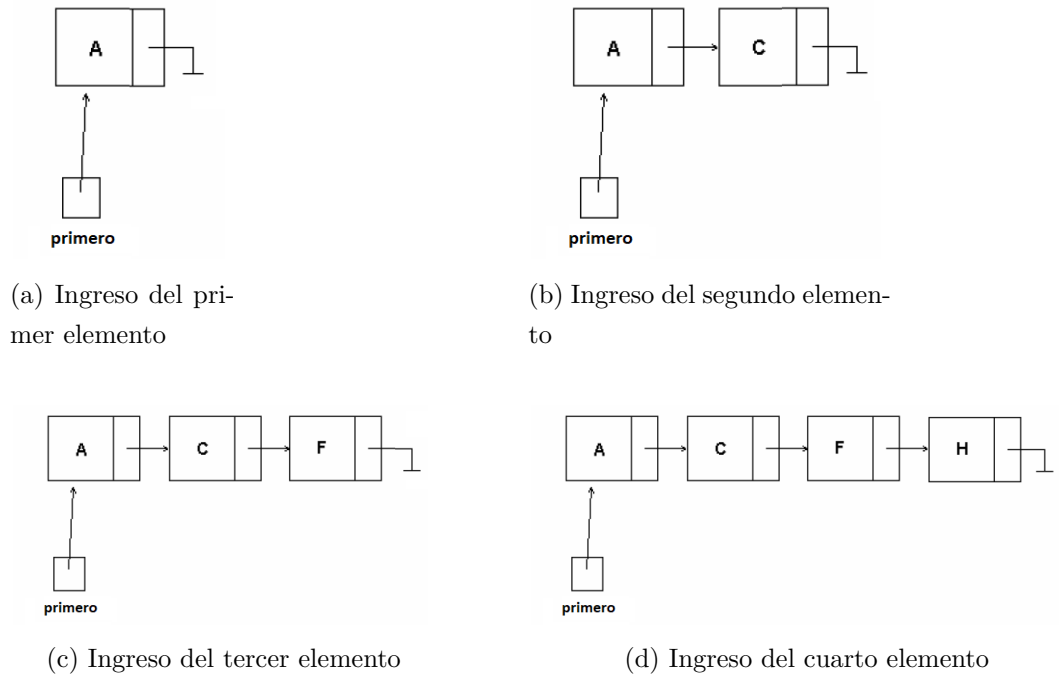


Gráfico 5.5.4: Construcción de una Pila SE

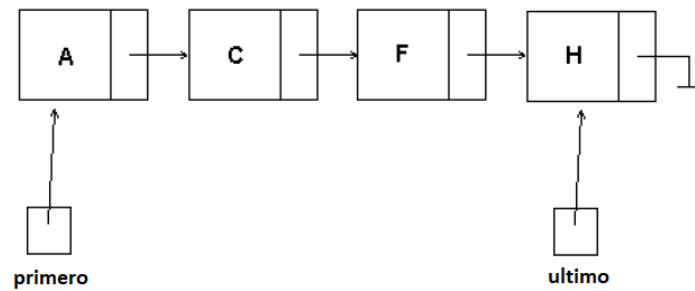


Gráfico 5.5.5: Pila con puntero al primero y al último

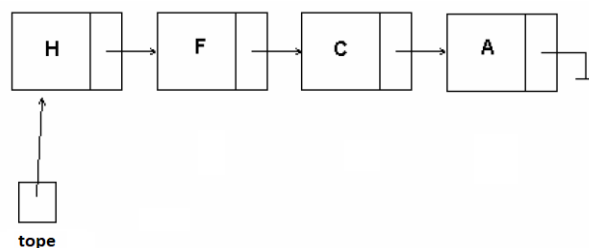


Gráfico 5.5.6: Pila SE con puntero al último



Nota. El crecimiento de una Pila SE solo estará restringido por la memoria disponible, por lo que no verificamos ningún tope como sucedía con la implementación estática.

Como se utiliza memoria dinámica cobra especial importancia el destructor, y el método de borrado de un Nodo, ya que no hay que olvidarse de liberar la memoria cada vez que se elimina un dato.

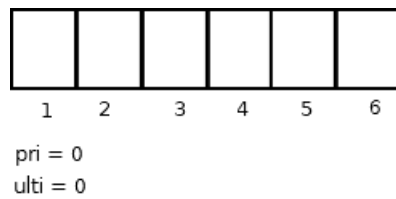
5.5.2. Implementaciones de Cola

Implementación estática

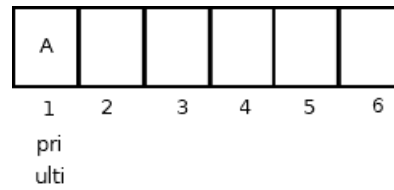
Al igual que en el caso de la Pila podemos implementar una Cola utilizando un vector de cierto tamaño MAXIMO. Como el ingreso de datos es por el principio y la salida por el final, es necesario en ciertas implementaciones y recomendable en otras tener un atributo que señale el principio y otro, el final.

El problema que se agrega con una implementación estática, que no sucedía con una Pila, es que al ingresar y sacar datos, los índices se irán deslizando en el vector, corriendo el peligro de quedarnos sin espacio en una Cola que esté casi vacía, como vemos en la figura 5.5.7. En el ejemplo tenemos una Cola estática con 6 posiciones como MAXIMO. Al principio, la Cola se creará vacía, por lo que el primero (*pri*) y el último (*ulti*) estarán en 0, como vemos en la parte a). Comentario: estamos asumiendo que la primera posición es la 1, a pesar de que en C++ las posiciones en los vectores comienzan con el índice 0.

Luego, en la parte b) ingresa un elemento, el valor “A”. Como la Cola estaba vacía, ingresa en la posición 1, y tanto *pri* como *ulti* tienen dicho valor. En la imagen c) ingresa un nuevo valor “B” en la posición 2. En este momento *ulti* queda con el valor 2, mientras que *pri* sigue con 1. En la parte d) se muestra la misma Cola luego de que ingresen los valores “C”, “D” y “E”,



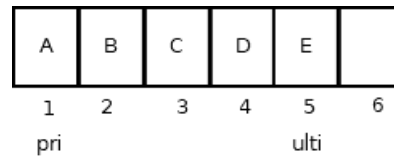
(a) Cola vacía



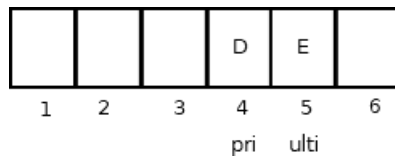
(b) Cola con un elemento



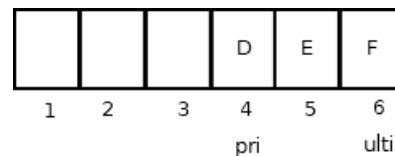
(c) Cola con dos elementos



(d) Cola con 5 elementos



(e) Se eliminan tres elementos



(f) Cola que parece llena

Gráfico 5.5.7: Implementación estática de una Cola

ulti queda con valor 5. En la imagen e) se muestra cómo queda la Cola luego de eliminar tres valores consecutivos, dado que se eliminan desde el principio, el índice *pri* queda con el valor 4. En la imagen f) ingresa un nuevo valor: “F”, por lo que *ulti* pasa a valer 6. En este momento no pueden seguir ingresando valores ya que el índice *ulti* tomó el valor máximo del vector. Aparentemente la Cola está llena cuando en realidad tiene solo tres elementos.

Una solución a esto sería ir corriendo los elementos al principio cuando quedan casillas vacías. Pero una implementación así no es eficiente, ya que este corrimiento constante sería costoso. Una opción mejor tomaría al vector como un vector circular, es decir, si en el ejemplo que vimos, ingresara un nuevo elemento, “G”, debería ir en la posición 1, por lo que *ulti* pasaría a valer 1. Si bien esto es más eficiente en cuanto al costo computacional, hay que tener cuidado con la implementación ya que aumenta la complejidad de la programación.

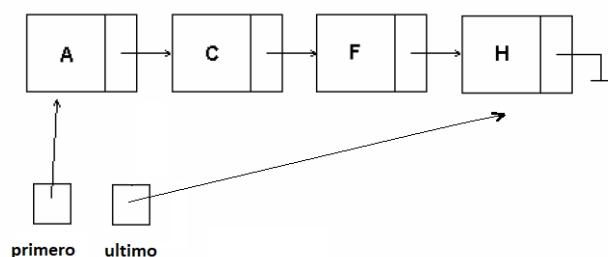


Gráfico 5.5.8: Implementación dinámica de Cola

Implementación dinámica

La idea explicada en la implementación dinámica de una Pila aplica también para una Cola. Se utilizan nodos creados en tiempo de ejecución que se van enlazando unos a otros. La única diferencia es que nos conviene tener un atributo más que es un puntero al último nodo. En el gráfico 5.5.8 vemos su representación y en el Anexo su implementación (A.12 y ??) y en A.13 su uso.

5.5.3. Implementaciones de Lista

Implementación estática

Con el mismo criterio que vimos en pilas y colas podemos utilizar un vector para representar una Lista. El problema que se agrega es que la inserción puede ser en cualquier posición. Lo mismo sucede con la eliminación. Por lo tanto, las inserciones y borrados generarán corrimientos de elementos en el vector con su correspondiente costo computacional, como vemos en el gráfico 5.5.9.

En el apéndice, ??, ?? tenemos un ejemplo reducido de implementación, junto con el uso ??.

¿Qué problemas tiene una implementación estática?

- a. El primer inconveniente que se presenta es el problema de estimar la dimensión antes de ejecutar el programa. Si estimamos muy poco corremos el riesgo de que se nos termine nuestro vector y no podamos seguir almacenando elementos. Por el contrario, si sobrestimamos, consumiremos memoria de más, lo cual puede llegar a ser crítico, ya que, además de poder quedarnos sin memoria disponible, la aplicación podría volverse notoriamente lenta.

Si bien podríamos definir un vector de forma dinámica, deberíamos

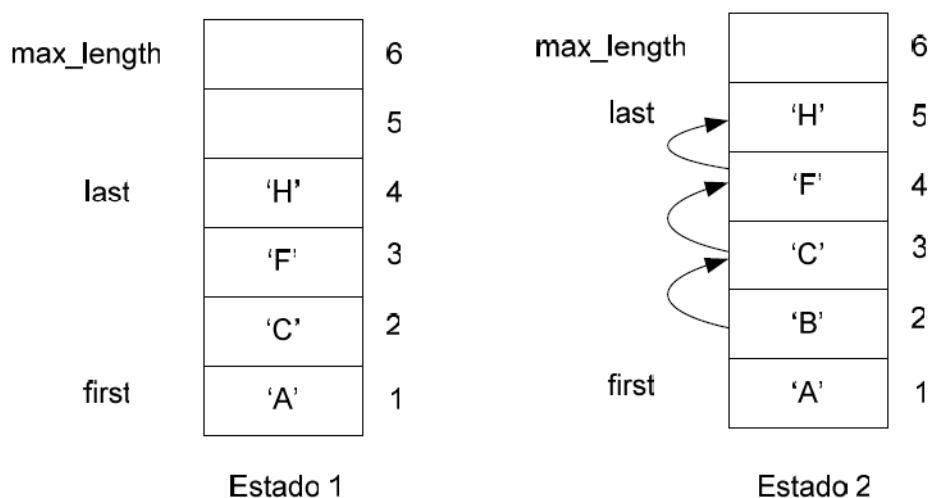


Gráfico 5.5.9: Corrimientos necesarios en el manejo de listas estáticas

pedirle al usuario que ingresara el tamaño a utilizar al principio de la aplicación. Transfiriéndole a él el mismo problema que se acaba de señalar.

- b. En segundo lugar, como ya mencionamos, si observamos el método de borrado, vemos que para borrar un elemento que se encuentra en la posición p debemos “correr” un lugar todos los elementos que están en las posiciones $p + 1, p + 2, \dots, tope$, ubicándolos en las posiciones $p, p + 1, \dots, tope - 1$, respectivamente.
- c. En el insertado, a excepción de la inserción al final (como se implementó), se debería realizar una operación similar al borrado pero a la inversa para crear el lugar en donde iría el nuevo elemento. En el gráfico 5.5.9 se muestran, con flechas, los movimientos que se deberían realizar para pasar del *Estado 1* al *Estado 2* de una lista ordenada, al insertar la letra ‘B’.

Por supuesto que en ejemplos de dimensión 6, como los de la figura, esto no es ningún problema, pero en los casos reales, en donde se tienen varios cientos o miles (y por qué no millones) de elementos, estas operaciones comienzan a ser costosas.

Hay alternativas para solucionar este último problema manteniendo estructuras estáticas. Una variante es que cada elemento además de tener almacenado el dato en sí, tendrá un índice a la posición del siguiente. El último, tendrá un índice que vale 0 o -1 . Esto simulará una lista dinámica en una

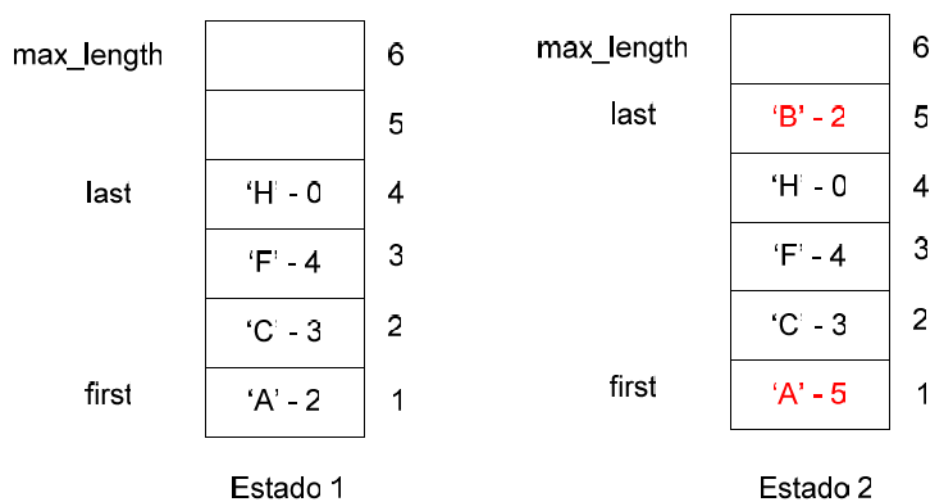


Gráfico 5.5.10: Lista estática con cursores

implementación que es estática. La misma situación del gráfico 5.5.9 quedaría como vemos en el gráfico 5.5.10. En el *Estado 1*, el primer elemento está en la posición 1 (podría estar en otra posición) y tiene el elemento 'A' e indica que, el siguiente elemento está en la posición 2 y, así, sucesivamente. La inserción de 'B' se realiza como en la implementación que realizamos, es decir, un agregado al final. Pero ahora no tendremos la necesidad de mover todos los elementos, solamente debemos modificar el índice que tiene 'A' ya que el elemento siguiente pasa a ser 'B' que está en la posición 5. 'B' “enlaza” con la posición 2, donde está 'C'. El resto no se modifica. En la figura, las celdas que sufren modificaciones, se marcaron en rojo.

A este tipo de implementaciones se les llama lista con cursores.

5.5.4. Estructuras dinámicas

Las estructuras dinámicas serán las que utilizaremos generalmente. Dentro de estas hay varias formas de implementación: listas simplemente enlazadas, doblemente enlazadas, circulares, etc. Iremos viendo cada una de ellas.

Listas simplemente enlazadas

Una lista simplemente enlazada consistirá en una sucesión de nodos que se irán enlazando cada uno con el siguiente al igual que vimos con pilas y colas. Por lo tanto, para implementarla solo necesitamos un puntero al primer nodo

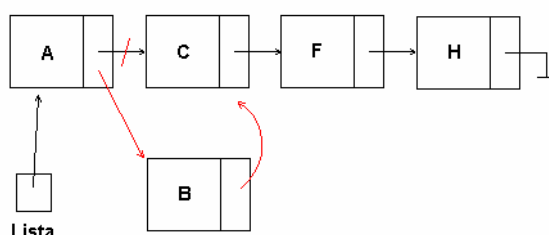


Gráfico 5.5.11: Inserción en una lista simplemente enlazada

de la lista, ya que el resto de los nodos se enlazarán mediante sus propios punteros. El último puntero, apuntará a nulo y se representa como un “cable a tierra” en el gráfico 5.5.3.

La inserción es una operación mucho menos costosa que en una implementación estática dado que no hay necesidad de correr los datos como vimos en el ejemplo de la sección anterior. En esta implementación, una vez ubicado el lugar donde se insertará el nodo, solo deben ajustarse dos punteros. Por ejemplo, para insertar el elemento ‘B’ conservando el orden solo tendremos que modificar el siguiente de ‘A’ y el siguiente de ‘B’, como vemos en el gráfico 5.5.11.

Se reasigna el puntero que tiene el nodo donde está ‘A’ apuntando al nuevo nodo que contiene ‘B’. El puntero del nodo de ‘B’ apuntará al mismo lugar donde apuntaba ‘A’, que es ‘C’.

Estos nodos se irán creando en tiempo de ejecución, en forma dinámica, a medida que se vayan necesitando. Si bien esta implementación necesita más memoria que la estática, ya que, además del dato, debemos almacenar un puntero, no hay desperdicio, debido a que solo se crearán los nodos estrictamente necesarios.

A modo de ejemplo, en el apéndice A se muestra una implementación muy básica de esta estructura.clase. El código se puede ver en los algoritmos A.14, A.15, para la clase ListaSE y en A.16 un uso de la clase ListaSE.

Listas doblemente enlazadas

Una estructura como la anterior si bien es simple presenta algunas deficiencias. Por ejemplo, si estamos parados en el nodo donde está ‘F’ y queremos ir al nodo que contiene ‘C’, uno anterior, debemos comenzar a recorrer la lista desde el principio, ya que no hay forma de retroceder porque no tenemos ningún puntero que nos lleve al nodo anterior. Las listas doblemente enlazadas o bidireccionales solucionan este punto agregando otro puntero en el nodo, que apunta al nodo anterior. Esta solución tiene un costo: mayor

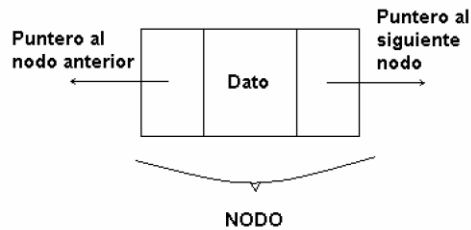


Gráfico 5.5.12: Nodo doblemente enlazado

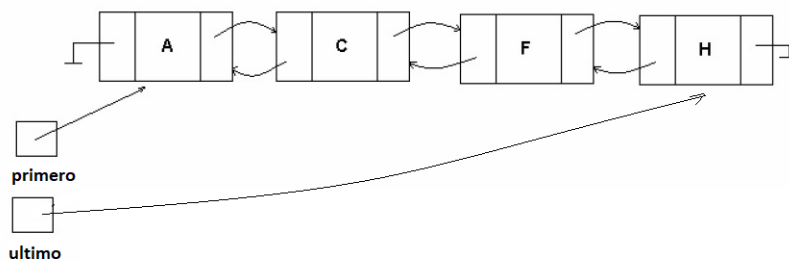


Gráfico 5.5.13: Lista doblemente enlazada

consumo de memoria y mayor complejidad a la hora de programar. Gráficamente, un nodo doblemente enlazado se puede representar como se muestra en el gráfico 5.5.12 y la lista queda como en 5.5.13. Cabe destacar que se agrega un puntero al último nodo aprovechando la bidireccionalidad de la lista.

Hay que tener en cuenta que, al insertar un nuevo nodo, se deben ajustar cuatro punteros: los dos del nuevo nodo, el anterior del siguiente y el posterior del anterior.

Una implementación de este tipo tendría sentido en aplicaciones donde se debe estar accediendo frecuentemente tanto a elementos anteriores de un nodo dado como posteriores, como pueden ser los algoritmos de compresión o suavizado de curvas, etc. También tendría sentido agregar un puntero más a la clase que sería el *actual*. Este puntero se usaría para que en cada momento señale el nodo donde se está poniendo el foco.

Otras implementaciones

Un tipo de implementación importante de mencionar son las listas circulares. Una lista circular es una lista en donde el último nodo apunta al primero, como se aprecia en el gráfico 5.5.14.

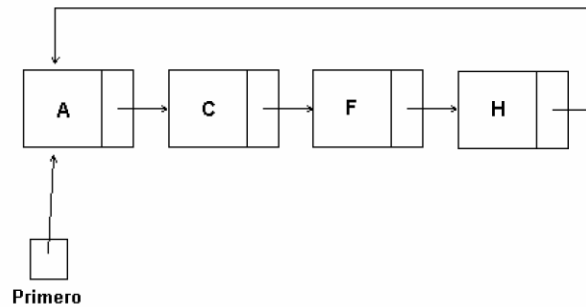


Gráfico 5.5.14: Lista circular

¿Cuál es la ventaja / utilidad de una implementación circular? Este tipo de implementaciones son útiles para el tratamiento de datos que se reciben en forma continua, como por ejemplo datos que se reciben constantemente de un sensor, el cual puede tomar temperaturas, flujos, etc. Como los procesos son continuos en el tiempo, una lista circular de n nodos nos permite tener siempre vigentes los últimos n datos.

5.6. Listas con templates

Todas las implementaciones que hemos visto, tanto de pilas, como de colas o listas, se pueden y recomiendan realizar en forma parametrizada, utilizando templates como ya vimos. También podemos aplicar polimorfismo dinámico para guardar distintas clases de objetos en una misma estructura. Hay que tener en cuenta que estos objetos deben heredar de un antecesor común.

Parte III

Estrategias y análisis

Capítulo 6

Recursividad

Las funciones recursivas son funciones en las que, dentro de la definición, tienen por lo menos un llamado a sí mismas, ya sea en forma directa o indirecta. La forma directa es una sentencia explícita en el código de la función llamándose a sí misma, como se ve en el algoritmo 6.1. En cambio, en la forma indirecta el llamado recursivo se realiza a través de otra función. Por ejemplo, tenemos una función A que llama a una función B y esta función vuelve a llamar a A (ver el algoritmo 6.2).

Decimos *por lo menos una vez* porque puede existir recursividad múltiple en la que en la definición hay varios llamados recursivos. Iremos viendo los distintos casos.

6.1. Principios de recursividad

El ejemplo más clásico de recursividad es el de la función factorial. De las matemáticas sabemos que $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ pero además lo podemos pensar como $5! = 5 \cdot 4!$. Por lo tanto, $n! = n * (n - 1)!$. Sin embargo, debemos dar algún caso base ya que si $n \in \mathbb{Z}$, tendríamos una recursión infinita pidiendo el valor de factoriales negativos. De modo que se necesita definir que $0! = 1$.

Algoritmo 6.1 Recursividad directa

```
valor_retorno A(parametros_formales) {  
    ...  
    A(parametros_actuales)  // llamado a sí misma  
    ...  
}
```

Algoritmo 6.2 Recursividad indirecta

```

valor_retornoA A(parametros_formalesA) {
    ...
    B(parametros_actualesB) // llama a B
    ...
}

valor_retornoB B(parametros_formalesB) {
    ...
    A(parametros_actualesA) // llama a A
    ...
}

```

Algoritmo 6.3 Cálculo del factorial de un número en forma iterativa

```

// PRE: n es un entero mayor o igual a cero
int factorial(int n) {
    int resultado = 1;
    for (int i = 2; i <= n; i++)
        resultado *= i;
    return resultado;
}

```

Las dos maneras de pensar el mismo problema serán:

$$n! = n * (n - 1) * (n - 2) \dots 2 * 1 \quad (6.1)$$

o:

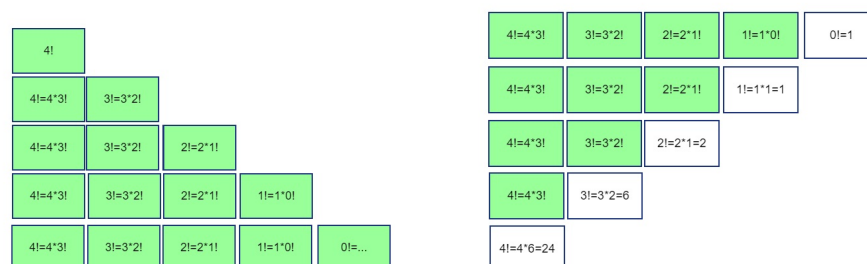
$$n! = n * (n - 1)! \text{ con } 0! = 1 \quad (6.2)$$

Teniendo en cuenta ambas definiciones haremos dos funciones distintas que buscarán el mismo objetivo: devolver el factorial de un entero mayor o igual a cero. Observando la ecuación 6.1 generamos el código que vemos desarrollado en 6.3. Esta función es iterativa o repetitiva, es decir, no utiliza recursividad. Lo que hace es ir haciendo los sucesivos productos mientras guarda el valor parcial en la variable que llamamos *resultado*, para finalmente devolver este valor. En cambio, guiándonos por la definición 6.2 escribimos el código recursivo que vemos en el algoritmo 6.4. Es interesante notar que esta función transcribe casi en forma literal la definición matemática.

Es importante tener en cuenta cómo es el funcionamiento en las llamadas recursivas. Si, por ejemplo, llamamos a la función con el factorial de 4, como 4 no es igual a 0 no entrará en el caso base e intentará hacer una devolución con la cláusula *return*. Sin embargo, antes de hacer dicha devolución,

Algoritmo 6.4 Cálculo del factorial de un número en forma recursiva

```
// PRE: n es un entero mayor o igual a cero
int factorial(int n) {
    if (n == 0)          // caso base
        return 1;
    return n * factorial(n-1); // llamado recursivo
}
```



(a) Llamados hasta alcanzar el caso base (b) Devolución de las llamadas en espera

Gráfico 6.1.1: Sucesivos llamados recursivos

necesitará resolver un nuevo resultado: el factorial de 3 que se pide en la llamada recursiva. Este proceso se vuelve a repetir, como vemos en la figura 6.1.1a, quedando todas las devoluciones en espera, hasta llegar al caso base, en donde, recién en ese momento empezará a hacer devoluciones sin ningún otro llamado recursivo. Entonces, cada uno de los llamados podrá hacer la devolución y finalizará su ejecución (ver imagen 6.1.1b).

6.2. Funcionamiento interno

Cada vez que una función es llamada el curso de ejecución de un programa se desvía hacia otra porción de código. Podemos imaginar que “pega un salto” hacia estas nuevas instrucciones, debiendo retornar al punto del salto cuando dicha función termina de ejecutarse. Además de la dirección de retorno necesita, entre otras cosas, guardar los valores de las variables que está utilizando. Todos estos datos son guardados en un registro antes de pasar a las instrucciones de la función. Este registro se guarda en el *stack* de la memoria y es consultado y liberado al regreso de la función llamada.

Entonces, si tenemos una porción de código como vemos en el algoritmo 6.5, la pila con los registros se va armando de la siguiente manera: cuando se

Algoritmo 6.5 Llamado de sucesivas funciones

```
1 tipo_retorno_g g(...) {
2     ...
3     return ... // debe regresar a B
4 }
5
6 tipo_retorno_f f(...) {
7     ...
8     g(...)      // instrucción B
9     ...
10    return ... // debe regresar a A
11 }
12
13 int main() {
14     ...
15     f(...) // instrucción A
16     ...
17 }
```

ejecuta el *main*, antes de la instrucción 15, la pila solo guarda un registro, como vemos en la imagen 6.2.1, parte *a*. Luego, en la línea 15, cuando llama a la función *f* se agrega un nuevo registro a la pila con los valores pertenecientes a esta función y la dirección donde debe regresar, que es de donde fue llamada. Esto lo vemos en la parte *b* de la misma figura. Dentro de la función *f* se llama a la función *g* en la línea 8, por lo tanto, se agrega otro registro a la pila con estos datos, como se aprecia en la parte *c* de la imagen.

Luego de ejecutar cada función debe regresar al punto de donde fue llamada y puede liberar de la pila el registro correspondiente con los datos almacenados. Es decir, cuando se encuentra con el *return* de la función *g* (línea 3) debe regresar a la línea 9 o la línea 8 si hubiera alguna asignación o instrucción para completar. En ese momento libera de la pila los datos pertenecientes a la función *g*. Lo mismo sucede con la pila de la función *f* en el *return* de la línea 10.

Es importante tener este procedimiento presente para comprender algunas consecuencias no deseadas en el uso de la recursividad y de qué modo evitarlas.

6.2.1. *Stack overflow*

Como vimos en el apartado anterior, por cada llamado a alguna función el programa debe guardar un registro con los datos locales y el punto de regreso. Cuando utilizamos recursividad, vimos que una misma función se

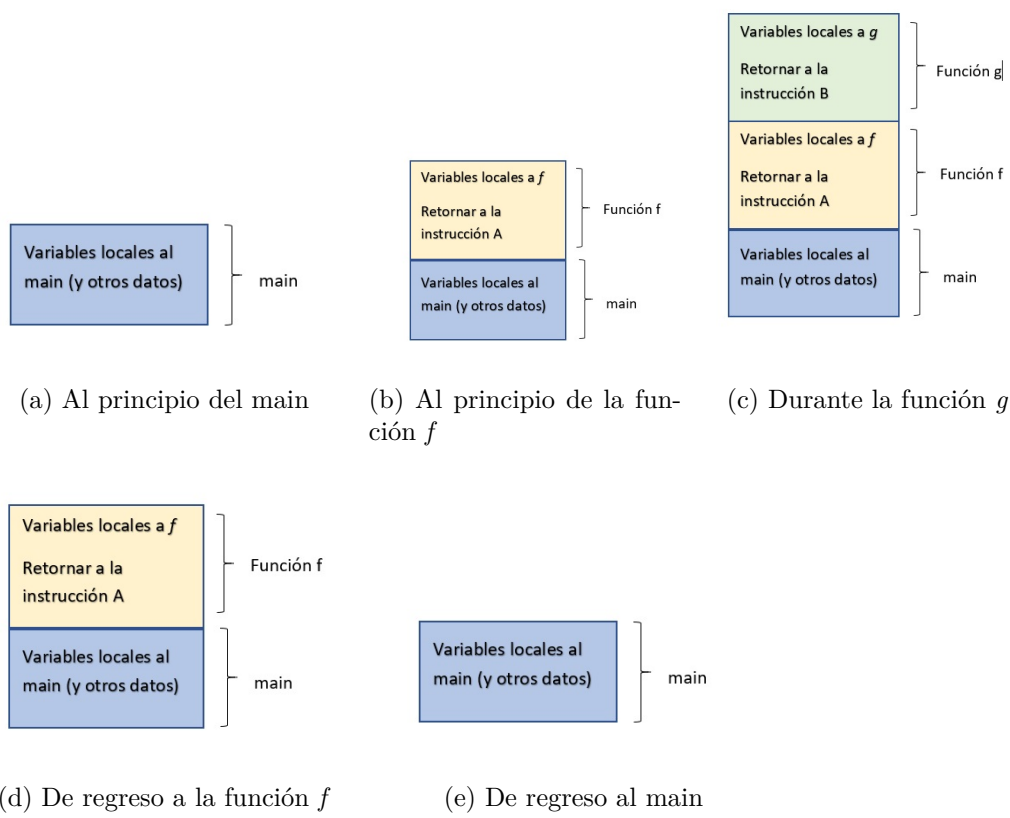


Gráfico 6.2.1: Variación de la pila de registros

puede llamar a sí misma muchas veces. Por ejemplo, si a la función factorial que vimos en el algoritmo 6.4 la llamamos con el valor 4, se llamará una vez por el número 4, otra por el 3, otra por el 2, otra por el 1 y, finalmente, otra por el 0. Es decir, deberá guardar los datos de 5 registros a pesar de que está llamando a la misma función pero, en cada instancia, será una función con valores distintos. Generalizando, si la llamamos con el parámetro n deberá guardar $n + 1$ registros con los datos de las variables locales y puntos de regreso. Dependiendo de la cantidad de datos que una función deba almacenar y la cantidad de llamados y la capacidad de la memoria, esta pila de registros puede llegar a agotarse. Es lo que llamamos *stack overflow* o pila desbordada. Es estos casos nuestra aplicación finalizará por haber consumido toda la memoria disponible. Por este motivo debemos ser cuidadosos a la hora de utilizar recursividad.

6.2.2. Recursividad de cola y posible optimización

Supongamos que en el algoritmo 6.5 se varía el código de la función f . Los cambios son: el llamado a la función g se elimina de la línea 8 y se agrega en el *return* de la línea 10. Es decir que en la línea 10 nos quedaría el llamado a la función g : *return g(...)*.

Esto ¿qué significa? Que apenas la función g retorne su valor, regresará a la función f que también retornará inmediatamente, ya que está devolviendo el valor que recibe de g sin hacer ninguna otra cosa. Por lo tanto, podríamos pensar que guardar el registro con los valores de f al momento de llamar a g es innecesario, ahorrándonos un registro. ¿Cuál es el único cambio que se debería hacer? En lugar de guardar la dirección de f como regreso, directamente hay que tomar la dirección del *main* para el regreso.

Esta mejora en el llamado a unas pocas funciones no sería significativa, pero en las funciones recursivas es importante ya que el ahorro de memoria y de tiempo al evitar estar guardando y liberando registros podría ser considerable. Hay algunos lenguajes que implementan esta optimización, la cual se llama *Tail Call Optimization* (TCO) u optimización de llamado de cola. Entonces, ¿cómo es una función recursiva de cola (*tail recursion*)? La última instrucción solo debe tener un llamado a sí misma. Aparentemente la función *factorial* que escribimos en el algoritmo 6.4 sería una función recursiva porque en la última línea se vuelve a llamar a la función. Sin embargo, no se podría obviar la vuelta a la función anterior porque necesita resolver una multiplicación con el valor devuelto. El n que multiplica al llamado recursivo arruina nuestros planes. En el algoritmo 6.6 convertimos la función factorial en una función recursiva de cola.

Algoritmo 6.6 Función factorial con recursividad de cola

```
// Calcula el factorial de n usando recursividad de cola
int f(int n, int res) {
    if (n == 0)
        return res;
    return f(n-1, n*res);
}

// PRE: n es un entero mayor o igual a cero
int factorial(int n) {
    return f(n, 1); // solo llama a la otra función
}
```



Nota. Si bien no hace falta definir la otra función *factorial*, se dejó por compatibilidad, ya que un usuario necesita saber el factorial de un número sin estar pendiente que le tiene que pasar además el parámetro 1.

6.2.3. Tipos de recursividad

Hemos visto que la recursividad puede ser *de cola* o, en el caso general, *no de cola*. Desarrollamos la función factorial de ambas maneras. También podemos clasificar la recursividad por el modo de su invocación en *directa* o *indirecta*, como mencionamos al principio de este capítulo. Los dos ejemplos que vimos hasta el momento son de forma *directa*. Veamos algún ejemplo de recursividad indirecta. Supongamos que de un vector queremos calcular la suma de sus valores pares y el producto de los impares. Por supuesto que esto lo podríamos hacer utilizando recursividad en una sola función, pero vamos a emplear tres funciones para ver cómo es la recursividad indirecta. Una función que llamamos *calcular* decide si lo que debe procesar es par o impar. Dependiendo de esto llama a la función *sumar* o *multiplicar*. Ambas, luego de hacer el cómputo correspondiente vuelven a llamar a *calcular* decrementando el valor del índice del vector. Este proceso se repite mientras el índice sea mayor o igual que cero. Cabe destacar que estas funciones no tienen ningún *return*, ya que la devolución la hacen a través de un vector de dos componentes: en la primera coordenada se guardará la suma y en la segunda el producto.

El código lo podemos ver en el algoritmo 6.7 con un *main* de ejemplo.

Un caso más raro es el de la recursividad anidada. Supongamos que nos

Algoritmo 6.7 Recursividad indirecta

```
// Firmas de las funciones
void calcular(int vec[], int res[], int n);
void sumar(int vec[], int res[], int n);
void multiplicar(int vec[], int res[], int n);

int main() {
    int vec[] = {5, 4, 8, 1, 9};
    int res[] = {0, 1};
    calcular(vec, res, 4);
    cout << "Resultado: " << res[0] << " " << res[1] << endl;
    return 0;
}

// acumula la suma con el valor actual y
// llama a calcular decrementando el índice
void sumar(int vec[], int res[], int n) {
    res[0] += vec[n];
    calcular(vec, res, n-1);
}

// acumula el producto con el valor actual y
// llama a calcular decrementando el índice
void multiplicar(int vec[], int res[], int n) {
    res[1] *= vec[n];
    calcular(vec, res, n-1);
}

// decide a qué función llamar:
// sumar o multiplicar
void calcular(int vec[], int res[], int n) {
    if (n >= 0) {
        if ( (vec[n] % 2) == 0 )
            sumar(vec, res, n);
        else
            multiplicar(vec, res, n);
    }
}
```

Algoritmo 6.8 Recursividad anidada

```

1 int anidada(int n) {
2     if (n == 1 || n >= 5)
3         return n;
4     return anidada(1 + anidada(2*n));
5 }

```

definen la siguiente sucesión:

$$a(n) = \begin{cases} 1 & \text{si } n = 1 \\ a(1 + a(2n)) & \text{si } 2 \leq n \leq 4 \\ n & \text{si } n \geq 5 \end{cases}$$

Escribamos los primeros términos de la sucesión:

$$a(1) = 1$$

$$a(2) = a(1 + a(4)) = a(1 + a(1 + a(8))) = a(1 + a(9)) = a(10) = 10$$

$$a(3) = a(1 + a(6)) = a(1 + 6) = a(7) = 7$$

$$a(4) = a(1 + a(8)) = a(1 + 8) = a(9) = 9$$

$$a(5) = 5$$

$$a(6) = 6$$

Tanto si n vale 1, como si vale 5 o más, la devolución es inmediata. Sin embargo, si vale 2, 3 o 4, hay un doble llamado recursivo, ya que se vuelve a calcular la función con un nuevo argumento, pero en ese argumento hay que hacer otro cálculo de dicha función. El código nos queda como vemos en el algoritmo 6.8. En la línea 4 tenemos el llamado recursivo y, como parámetro, a su vez otro llamado a la función.

Por último, podemos clasificar la recursividad por la cantidad de llamados que realiza a sí misma, en *simple* o *múltiple*. En los ejemplos de la función factorial el tipo de recursividad es simple. El caso múltiple típico es el de la función que devuelve la sucesión de Fibonacci. Esta sucesión está definida por:

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F(n-1) + F(n-2) & \text{si } n \geq 2 \end{cases}$$

Es decir que la sucesión para un cierto valor de n se obtiene mediante la

suma de las dos sucesiones anteriores, salvo que el n sea 0 o 1. De este modo:

$$\begin{aligned}F(0) &= 0 \\F(1) &= 1 \\F(2) &= F(1) + F(0) = 1 \\F(3) &= F(2) + F(1) = 2 \\F(4) &= F(3) + F(2) = 3\end{aligned}$$

El código de forma iterativa no es demasiado complejo pero tampoco es trivial, se necesita pensar un rato. Una forma de hacerlo es como se muestra en el algoritmo 6.8. En cambio, el algoritmo recursivo es inmediato viendo la definición matemática, se muestra en el algoritmo 6.10. La sencillez para escribir ese código y la facilidad para pensarlo, abruman. Pero no todo es de color rosa: si ejecutamos esta función con valores mayores a 30 o 40 vemos que el tiempo de ejecución es considerable. ¿Por qué sucede esto? Veamos qué pasa cuando llamamos a la función con un valor de 5, por ejemplo. Como 5 no es menor o igual que 1 la función devuelve un doble llamado a sí misma, pidiendo calcular la sucesión en 4 y en 3 para luego sumarlas. Pero con 4 sucede lo mismo. En la figura 6.2.2 vemos cómo el árbol que representa las llamadas a la función se va duplicando en cada nivel hasta, por supuesto, llegar a los casos base que son 0 y 1. Este crecimiento es exponencial, por este motivo debemos ser cuidadosos a la hora de trabajar con recursividad múltiple, a menos que sepamos que los valores de entrada son chicos. Cabe destacar mirando con detenimiento el árbol que un mismo cálculo lo hace varias veces, por ejemplo, el valor de la sucesión en 2 lo calcula tres veces, es decir, hay un desaprovechamiento de los cálculos ya realizados.

¿Cómo podríamos mejorar esta función sin hacerla iterativa? Con un poco de mayor complejidad podemos transformar esta función recursiva múltiple en otra recursiva simple y de cola. La idea es guardar los resultados parciales para no tener que volver a calcularlos. El resultado es un híbrido entre la forma iterativa y la recursiva. El código es el que se ve en el algoritmo 6.11. Claramente esta función es mucho más rápida y no consume tantos recursos como la anterior. Lamentablemente no podemos probarla con valores muy grandes porque excede la capacidad del entero.

6.3. Algoritmo divide y vencerás

En diversas disciplinas es frecuente indicar que se aplica una resolución divide y vencerás cuando se descompone un problema complejo en partes mas simples para tratarlas por separado, y así facilitar el desarrollo de la

Algoritmo 6.9 Fibonacci iterativo

```
int fibo(int n) {  
    int res = n;  
    if (n > 1) {  
        res = 1;  
        int ant = 0;  
        for (int i = 2; i <= n; i++) {  
            res = res + ant;  
            ant = res - ant;  
        }  
    }  
    return res;  
}
```

Algoritmo 6.10 Fibonacci recursivo (recursividad múltiple)

```
int fibo(int n) {  
    if (n <= 1)  
        return n;  
    return fibo(n-1) + fibo(n-2);  
}
```

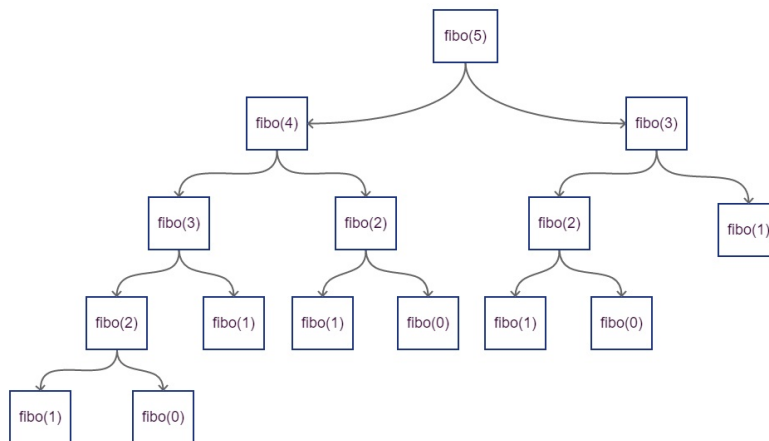


Gráfico 6.2.2: Recursividad múltiple: Fibonacci de 5

Algoritmo 6.11 Fibonacci con recursividad simple y de cola

```
// Función recursiva simple y de cola
int fibo(int n, int res, int res_ant) {
    if (n == 1)
        return res;
    return fibo(n-1, res + res_ant, res);
}

// solo llama a la función fibo con los
// dos primeros resultados: 0 y 1
int fibonacci(int n) {
    if (n == 0)
        return n;
    return fibo(n, 1, 0);
}
```

solución. La idea de dividir para conquistar se precisa de forma mucho mas rigurosa en el contexto de las llamadas estrategias de resolución de problemas de matemática o informática. Se trata de una forma de construir o diseñar algoritmos según etapas establecidas.

Definición. Si el tamaño del problema a tratar es $N > L$, siendo L un valor arbitrario, se debe dividir el problema en m subproblemas no solapados y que tengan la misma estructura que el problema original. Luego, a cada uno de esos m subproblemas se los dividirá nuevamente hasta que cada uno alcance un tamaño L o menor, los cuales se resolverán por cualquier otro método. Por último, la solución del problema original se obtiene por combinación de las m soluciones obtenidas de los subproblemas en que se lo había dividido.



Nota. Si bien este algoritmo se presenta en el capítulo de recursividad, no es obligatorio el uso de ella. Sin embargo, la lógica de dividir un problema en subproblemas nos lleva naturalmente a enfocarlo recursivamente.

Como puede inferirse, hay una cercanía conceptual entre la idea de la estrategia divide y vencerás (D y V) o dividir y conquistar y los algoritmos recursivos, dado que la división del problema en m subproblemas de la misma estructura sugiere llamados recursivos en donde se ha reducido el tamaño de la entrada de la forma indicada, y la solución cuando el tamaño del problema no supera L se corresponderá con el caso base. Pero no hay que pensar que la estrategia implica necesariamente aplicar recursividad. Estos algoritmos podrían ser implementados usando formas no recursivas, por ejemplo con pilas

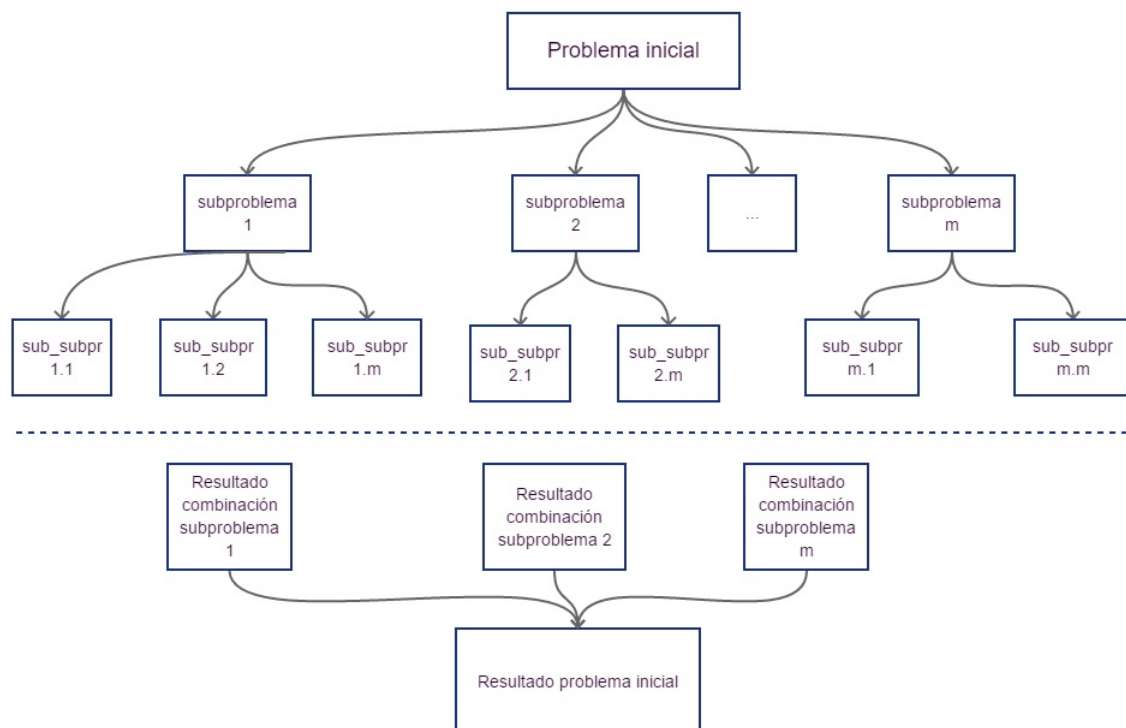


Gráfico 6.3.1: Algoritmo divide y vencerás

para almacenar los resultados parciales y otros parámetros. Recordemos que, como demuestra el teorema fundamental de las ciencias de la computación, todo algoritmo recursivo puede reescribirse iterativamente, y viceversa.

Al describir la idea de D y V se ha hecho mención al “tamaño del problema”. ¿A qué nos referimos con esto? En general, cuando resolvemos un problema computacional podemos indicar qué elemento o elementos tienen responsabilidad directa en el consumo de recursos del algoritmo, en particular del recurso espacio (memoria) y tiempo de ejecución. En general, es el número de datos sobre los cuales va a trabajar el algoritmo o también el valor de cierto dato. Tratar de hacer un ordenamiento en un vector es un ejemplo del primer caso. En cambio, calcular el factorial de un número lo es del segundo, ya que no es lo mismo calcular el factorial del número 3 que del número 20. Hilando un poco más fino, se puede definir el tamaño de la entrada de un problema como el número de símbolos necesarios de un cierto alfabeto finito para poder codificar todos los datos de un problema.

Entre los algoritmos más conocidos que utilizan D y V exitosamente están: la búsqueda binaria en un *array* ordenado, la potencia entera de enteros, los ordenamientos internos: *mergesort*, *quicksort*, *radixsort*, la resolución de las Torres de Hanoi, la multiplicación de enteros grandes de Karatsuba y Ofman, la multiplicación de matrices de Strassen, el cálculo de la transformada rápida de Fourier, la determinación del par de puntos más cercanos en un plano, el armado de un *fixture* para un campeonato. Algunos de ellos los desarrollaremos enseguida.

Existen también problemas que, si bien pueden ser resueltos aplicando DyV, no solo no presentan mejoras de eficiencia al utilizar la estrategia, sino que resultan más costosos en términos temporales que la resolución sin aplicar Dy V. Tal es el caso del problema de la subsecuencia de suma máxima en un array. Este problema propone determinar cuál es la suma máxima que puede obtenerse considerando elementos contiguos de un array. Este problema puede resolverse con un orden de $O(n \log(n))$ para el peor caso por D y V, y con un $O(n)$ con una exploración lineal.

Veamos algunos ejemplos de la aplicación de D y V, comenzando con algún caso sencillo.

Ejemplo 6.1. Se quiere calcular b^p donde $b \neq 0$ y $p \geq 0$, entero. Para eso vamos a hacer una función *potencia(base, exponente)* que nos devuelva dicho valor: $base^{exponente}$. En primer lugar haremos el código de manera iterativa sin utilizar D y V, lo podemos ver en el Algoritmo 6.12. El algoritmo hace un ciclo en donde va obteniendo los resultados parciales, comenzando desde 1 y obteniendo los productos sucesivos. ¿Cuántos ciclos realiza? Esto lo determina la variable *exponente*. Es decir que si queremos elevar un número a

Algoritmo 6.12 Función potencia iterativa

```
// calcula base elevado a exponente en forma iterativa
int potencia(int base, int exponente) {
    int resultado = 1;
    for (int i = 0; i < exponente; i++)
        resultado *= base;
    return resultado;
}
```

la 3, hará tres ciclos. Si es a la 20, hará veinte ciclos.

¿Cómo sería la versión recursiva sin la aplicación de D y V? Para esto tomamos en cuenta la propiedad matemática $b^p = b \cdot b^{p-1}$, es decir, el resultado de un número (base) elevado a otro entero (exponente), es igual que multiplicar dicho número por el resultado de elevar la base a un entero de una unidad menor que el original. Por supuesto que nos falta el caso base: esta propiedad se puede aplicar hasta que exponente vale 1, pero si es 0 el resultado es 1. De esta forma surge el código que apreciamos en 6.13. Comparando el rendimiento de esta función con la anterior vemos lo siguiente: en lugar de hacer p ciclos hace p llamados recursivos. Es decir que el rendimiento será menor ya que debe guardar los registros de cada una de las funciones en cada llamado (se debe tener en cuenta que no es recursividad de cola).

Apliquemos, finalmente, el algoritmo D y V. Para esto podemos por ejemplo pensar que si $2^{16} = 2^8 \cdot 2^8$, no tendríamos necesidad de calcular dos veces lo mismo. Entonces, calculemos una sola vez 2^8 , pero $2^8 = 2^4 \cdot 2^4$, entonces nuevamente calculemos 2^4 una vez sola. Este proceso se repite hasta llegar a un caso base que podemos fijar cuando el exponente es 1. El lector podría preguntarse qué sucede en el caso de que el exponente no sea un número par. En este caso solo debemos agregar una multiplicación por la base, por ejemplo $2^{15} = 2^7 \cdot 2^7 \cdot 2$. Guiándonos con esto generamos el código que figura en el algoritmo 6.14. Comparando esta función con las dos anteriores, si calculáramos la potencia de un número a la 16, con el algoritmo iterativo tendríamos 16 ciclos, con el recursivo 16 llamadas a la misma función, en cambio, con este algoritmo llamaríamos a la función con el valor 16, luego, en la línea 7, se volvería a llamar con el valor 8, esto se repetiría con el 4 y el 2 hasta que exponente valga 1. Es decir, haría cuatro llamados recursivos. Si pudiéramos usar un exponente muy grande sin que desborde la capacidad de almacenar un número entero, por ejemplo un exponente de valor 1000, los algoritmos que no utilizan D y V harían mil ciclos o mil llamados a la función exponente, en cambio este algoritmo haría solo diez.

Algoritmo 6.13 Función potencia recursiva

```
// calcula base elevado a exponente en forma recursiva
int potencia(int base, int exponente) {
    if (exponente == 0)
        return 1;
    return base * potencia(base, exponente - 1);
}
```

Algoritmo 6.14 Función potencia con algoritmo divide y vencerás

```
1 int potencia(int base, int exponente) {
2     if (exponente == 0)
3         return 1;
4     if (exponente == 1)
5         return base;
6     // caso general
7     int res = potencia(base, exponente / 2);
8     res *= res;
9     if ((exponente % 2) == 1)
10        res *= base;
11    return res;
12 }
```

6.4. Métodos de ordenamiento usando D y V

Si nos preguntan: “¿Qué es más fácil, ordenar un vector de mil elementos o uno de diez?” Obviamente responderemos que el de diez. La idea de los ordenamientos que utilizan D y V es dividir el vector a ordenar en dos vectores más cortos. Este procedimiento se repetirá de manera recursiva hasta quedarnos con un vector de tamaño uno que, trivialmente, ya estará ordenado. Luego, tendremos quizá la necesidad de combinar estos vectores para reconstruir el vector original pero ordenado.

Entonces, el pseudocódigo tendrá el siguiente alineamiento:

```
void ordenar(vec, min, max){
    if (max - min > 1) {
        particionar vec en vec_1 y vec_2
        y obtener punto_corte
        ordenar(vec_1, min, punto_corte)
        ordenar(vec_2, punto_corte+1, max)
        combinar(vec_1, vec_2)
    }
}
```

Hay dos algoritmos importantes que utilizan esta estrategia: Mergesort y

Quicksort.

6.4.1. Mergesort

En este algoritmo la partición es trivial: se divide al vector por la mitad sin hacer nada más. El verdadero esfuerzo estará en el último paso de combinar o “mezclar” los vectores, de ahí su nombre. Entonces, si el tamaño del vector es n , tendremos dos casos:

- Si $n = 2p$, el tamaño es par, tendremos que aplicar el método a dos vectores de tamaño p .
- Si $n = 2p + 1$, el tamaño es impar, tendremos que aplicar el método a un vector de tamaño $p + 1$ y a otro de tamaño p .

Como estos pasos son triviales, hay implementaciones del algoritmo que directamente comienzan por la parte del merge o combinación, tomando de a 2 elementos. Veamos cómo se realiza con un ejemplo.

- a.

35	49	51	38	18	12	20
----	----	----	----	----	----	----

 vector a ordenar
- b. Dividimos el vector en

35	49	51	38
----	----	----	----

 y

18	12	20
----	----	----
- c. Continúa el proceso con el primero de los vectores:

35	49
----	----

 y

51	38
----	----
- d. El primero queda dividido de manera trivial en

35

 y

49

- e. En este punto se empieza a realizar el merge ordenado, por lo que rearma

35	49
----	----
- f. Hace lo mismo con

51	38
----	----

 dividiéndolo en dos y rearmando en forma ordenada

38	51
----	----
- g. Ahora hace el merge entre ambos vectores:

35	49
----	----

 y

38	51
----	----

, por lo que debe comparar el primer elemento del primer vector con el primero del segundo, en este caso 35 es menor que 38, por lo que coloca 35 en la primera posición y luego compara el segundo elemento del primer vector con el primero del segundo: 49 con 38. Esto se repite hasta completar el merge entre ambos:

35	38	49	51
----	----	----	----

.
- h. Por la otra rama realiza la misma subdivisión, quedando:

18	12
----	----

 y

20

.

- i. Se subdivide

18	12
----	----

 en dos vectores de un elemento y comienza el merge entre ambos, quedando:

12	18
----	----

.
- j. Con el vector anterior se hace un merge con el vector de uno:

12	18
----	----

 y

20

, nos queda:

12	18	20
----	----	----

.
- k. Finalmente se hace el último merge entre

35	38	49	51
----	----	----	----

 y

12	18	20
----	----	----

. Como en este ejemplo, todos los elementos del segundo vector son más chicos que el primer elemento del primero, una vez que ingresan estos tres valores ya no hay necesidad de seguir comparando, por lo que el primer vector se copiará a continuación del segundo.

6.4.2. Quicksort

Este algoritmo, por el contrario del Mergesort, tiene un costo más alto en la partición pero lo gana luego en la combinación porque es trivial. De hecho, cuando se hace la última partición lo único que habría que hacer es unir todas las particiones que ya están ordenadas.

La idea es particionar el vector en dos partes, al igual que lo hace el Mergesort, pero esos dos vectores estarán divididos según un valor que llamaremos pivote. Entonces, en uno de los vectores quedarán todos valores menores que el pivote, y en el otro, todos valores mayores. Si seguimos con el mismo vector de ejemplo anterior:

35	49	51	38	18	12	20
----	----	----	----	----	----	----

y elegimos como pivote al primer elemento (35), nos quedará

18	12	20
----	----	----

 por un lado, y

49	51	38
----	----	----

 por el otro.

Luego, recursivamente, se vuelve a hacer lo mismo con estos dos vectores.

En el primero elegimos a 18 como pivote, quedándonos los vectores unitarios

12

 y

20

. Del otro lado, si elegimos al 49, nos quedará:

38

 y

51

.

Por último, lo que nos queda es unir todos esos vectores de la siguiente manera:

vector_izquierdo	pivote	vector_derecho
------------------	--------	----------------

En nuestro caso:

12	18	20
----	----	----

 y

38	49	51
----	----	----

, para finalmente unir todo en:

12	18	20	35	38	49	51
----	----	----	----	----	----	----

Este algoritmo presenta muchas variantes y cuestiones a decidir. Entre ellas, están:

- Elección del pivote. Es fundamental para la eficiencia del algoritmo una buena elección del pivote. Si en el ejemplo anterior hubiéramos elegido al valor 51 como el primer pivote, nos habría quedado un vector con

todos los elementos, salvo el 51, y otro vacío. De esta forma, no se aprovecha las ventajas del algoritmo. Hay que tener en cuenta que en el algoritmo Mergesort la división producía dos vectores de igual tamaño o de diferencia uno. En el quicksort esto no está asegurado pero es el objetivo que debemos buscar: dividir los vectores de manera equitativa. Por este motivo el pivote debe ser lo más aproximado al valor medio del vector.

- Algunas implementaciones contemplan un recorrido de todo el vector antes de elegir el pivote, para elegir el mejor valor posible. Incluso, dependiendo de la implementación, podría llegar a ser un valor que no esté en el vector. Esta implementación debería unir solo los vectores izquierdo y derecho sin agregar el pivote.
 - En muchos ejemplos eligen el primer valor como pivote solo por sencillez.
 - Si de antemano sabemos que el vector puede estar semiordenado, nos conviene elegir el valor que está en la mitad del vector como pivote.
- **Partición.** Se puede elegir que el vector izquierdo almacene los valores menores o iguales que el pivote (la decisión podría ser con el derecho). De esta manera, el pivote quedará dentro de uno de los vectores, por lo que en el momento de unir, solo tendremos que hacerlo con los vectores izquierdo y derecho.
 - **Vectores auxiliares.** Para entender el algoritmo se usan vectores auxiliares que son el resultado de las particiones. Sin embargo, con algunas variantes en la implementación esto no es necesario, ya que es el propio vector el que puede tener de un lado a los valores menores y del otro a los mayores. Esta implementación contempla intercambios en cada llamado recursivo. Y estos intercambios también generan distintas decisiones.

6.4.3. Orden y comparación del Merge y Quicksort

Si bien el orden de los algoritmos lo vamos a estudiar en el siguiente capítulo, podemos adelantar que el mergesort es de orden $O(n \cdot \log_2(n))$. Esto lo podemos deducir pensando que en el primer merge se debe recorrer el vector entero y hacer las correspondientes comparaciones. Esto nos demanda revisar los n elementos del vector. Lo mismo sucede en el segundo merge, etc. ¿Cuántas veces se hace esto? Diremos que se hace k veces. Por lo tanto, el

orden será $O(n \cdot k)$. Para averiguar cuánto vale k podemos pensarlo de este modo: si el vector se divide en dos recursivamente hasta llegar a 1, tendremos $\log_2(n)$ de estas capas. En ecuaciones:

$$\begin{aligned}((n/2)/2\dots)/2 &= 1 \\ \frac{n}{2^k} &= 1 \\ n &= 2^k \\ \log_2(n) &= k\end{aligned}$$

El algoritmo quicksort será, en el mejor de los casos, del mismo orden que el mergesort, porque también, en cada pasada debemos revisar todos los elementos. Por “mejor de los casos” nos referimos a buenas elecciones de pivotes que siempre dejan el vector dividido en dos partes iguales o con una diferencia mínima. En cambio, si en cada paso elegimos un pivote que es el menor elemento del vector (o el mayor), tendremos el peor de los casos, y la cantidad de capas (el valor de k) pasará a ser n . Es decir, el orden será $O(n \cdot n) = O(n^2)$, al igual que los métodos de selección y burbujeo.

A favor del quicksort podemos señalar que no necesita de vectores auxiliares como sí lo necesita el merge.

Capítulo 7

Complejidad

El concepto de eficiencia algorítmica hace referencia al consumo de recursos. Se dice que un algoritmo es más eficiente que otro si utiliza menos recursos. La eficiencia de un programa se determina sobre la base del consumo de tiempo de ejecución y espacio de memoria utilizados por el algoritmo codificado. Entonces:

- La complejidad temporal se refiere al el tiempo necesario para ejecutar un algoritmo.
- La complejidad espacial hace referencia a la cantidad de espacio de memoria (estática + dinámica) necesaria para ejecutar un algoritmo.

Habitualmente se busca una solución de compromiso considerando ambos tipos de complejidades.

7.1. Complejidad temporal algorítmica

Los estudios sobre el consumo de tiempo pueden ser:

- a posteriori o enfoque empírico: se mide el tiempo real de ejecución para determinados valores de entrada y en determinado procesador;
- a priori o enfoque teórico: se determina una función que indique el tiempo de ejecución para determinados valores de entrada.

En general se trabaja con este último enfoque ya que medir tiempos reales implicaría trabajar con una o varias máquinas en particular y en un lenguaje determinado, por lo que los estudios quedarían dependiendo de un reducido conjunto de características y no servirían para comparar distintos algoritmos o hacer predicciones.

Para el análisis de la complejidad temporal debemos distinguir algunos conceptos:

- Problema: necesidad inicial de la cual se busca alcanzar la solución mediante un algoritmo.
- Ejemplar: individuo de una especie o género.
- Algoritmo: serie de pasos ordenados y finitos cuyo objetivo es hallar la solución a un problema.

Para poder visualizar estas definiciones mencionaremos algunos ejemplos.

Ejemplo. El problema es obtener la multiplicación de dos números enteros que pueden tener varias cifras cada uno. Un ejemplar podría ser $(20, 13)$ es decir, el producto entre el número 20 y el 13. Este problema tiene infinitos ejemplares: la infinita combinación de dos números enteros. Aunque si lo llevamos al terreno computacional podríamos acotar a la combinación de dos números enteros de un cierto rango. Un algoritmo que solucione el problema debe servir para todos los ejemplares, de lo contrario decimos que el algoritmo *no es correcto*. Hay varios algoritmos que sirven para obtener el producto de dos números. El primero que nos viene a la mente es el que aprendimos en el colegio: colocar un número debajo de otro. Luego realizar el producto del dígito que está más a la derecha del multiplicador por cada uno de los dígitos del multiplicando, con cuidado de escribir solo un dígito como resultado, salvo en la última multiplicación, etcétera. Pero también hay otros algoritmos: el hindú con el producto en diagonales, el método japonés con líneas que se van marcando, el método egipcio y el ruso.

Ejemplo. El problema consiste en ordenar un vector de n elementos. Los algoritmos pueden ser los métodos de selección o burbujeo, etcétera. Los ejemplares son los distintos vectores para cada uno de los diferentes valores de n .

Al trabajar con el cálculo del consumo de tiempo o coste temporal se hace referencia al *tamaño de la entrada del problema*, o *tamaño del problema*. Este tamaño, que llamaremos n , depende de la naturaleza del problema y corresponde a aquel o aquellos elementos que produzcan, al crecer, un aumento de tiempo de ejecución. Por ejemplo, en el caso del cálculo del factorial el tamaño del problema será el valor sobre el cual pretendemos realizar el cálculo, ya que, cuanto mayor sea este número, mayor será el tiempo consumido por la ejecución del algoritmo. En cambio, en el caso de la búsqueda binaria, el tamaño del problema será el número de elementos que tenga el vector donde

se realizará la búsqueda. Si la finalidad del algoritmo es obtener la suma de dos matrices, el tamaño del problema quedará determinado por el producto entre la cantidad de filas y la cantidad de columnas de las matrices.

Además del tamaño del problema, el tiempo de ejecución de un programa depende de otros factores:

- la calidad del código objeto generado por el compilador;
- las características de las instrucciones en la máquina empleada en la ejecución del programa;

El tiempo que tarda en ejecutarse un algoritmo lo indicaremos por la letra T sin unidades específicas, podrían ser segundos, minutos, horas. Al analizar la complejidad temporal se hace abstracción de cuestiones tales como velocidad de procesamiento, número de microprocesadores, lenguaje utilizado, características del compilador, etcétera. Esto significa que se decide ignorar las diferencias de tiempo que provienen de características como las recién mencionadas, por lo cual el tiempo queda en función del tamaño del problema.

Dos implementaciones distintas de un mismo algoritmo guardan una relación que se indica mediante el principio de invarianza.

Definición. *Principio de invarianza.*

Dado un algoritmo y dos implementaciones del mismo, las cuales podrían ser en la misma máquina o en dos distintas, llamemos I_1 e I_2 a estas implementaciones, que tardarán un tiempo $T_1(n)$ y $T_2(n)$ respectivamente, entonces existe una constante real $c > 0$ y un $n_0 \in \mathbb{N}$ tales que $\forall n \geq n_0$ se verifica que:

$$T_1(n) \leq c \cdot T_2(n)$$

Este principio indica que dos ejecuciones distintas del mismo algoritmo solo difieren, para nuestro cálculo en cuanto a eficiencia, en un factor constante teniendo en cuenta valores de entrada suficientemente grandes.

Ejemplo. Se elige el algoritmo de ordenamiento mediante el método de selección de un vector de tamaño n . Se implementa en C++ (lenguaje compilado) y en Python (lenguaje interpretado) y en dos máquinas distintas. Es decir, tenemos cuatro implementaciones diferentes: en C++ en la máquina que llamaremos m_1 y en la máquina m_2 , esta última es más vieja y con menor capacidad de procesamiento. Lo mismo en el lenguaje Python. Tomamos los tiempos en segundos, mostramos los resultados en la tabla:

	n	Tm_1	Tm_2
C++	10 mil	0.467	0.87
	100 mil	40.34	75.15
Python	10 mil	16.24	31.2
	100 mil	1211	2325

Es decir que entre el lenguaje C++ y Python hay una proporción de más de 34 para $n = 10000$ y de 30 para $n = 100000$, en la primera máquina. Por lo tanto, podríamos suponer que el valor de c del que habla el principio de invarianza sería alrededor de 30. En la otra máquina, los tiempos casi se duplican para los diferentes valores de n y para los diferentes lenguajes. Vemos que esta proporción se mantiene. A simple vista podríamos decir que el valor de c entre ambas máquinas para un mismo lenguaje es 2, de 30 para una misma máquina y diferente lenguaje y de 60 entre ambas combinaciones.

El número de pasos requeridos por un algoritmo para resolver un problema específico es denominado “running time” o tiempo de ejecución del algoritmo y se analizará en función de la entrada del mismo (tamaño del problema), por lo que diremos que T es función de n , es decir $T = T(n)$. El valor de esta función es proporcional al tiempo de ejecución de un programa con una entrada de tamaño n .

Casos a considerar

Para un mismo algoritmo y un mismo valor de entrada n , podemos establecer tres situaciones o casos diferentes referidos al consumo temporal:

- Mejor caso. Corresponde a la secuencia de sentencias del algoritmo (traza) en la cual se ejecute una secuencia de instrucciones que corresponda al tiempo mínimo.
- Peor caso. Es el que corresponde a la secuencia de sentencias en la cual se ejecute una secuencia de instrucciones que corresponda al tiempo máximo.
- Caso promedio. Corresponde al promedio entre todas las trazas posibles ponderadas según su probabilidad. Este último caso es, en general el que presenta mayores dificultades para el análisis.

Ejemplo. Consideremos la búsqueda secuencial (ver algoritmo 7.1) de un dato en un vector de tamaño n . El mejor caso será cuando el dato a buscar se encuentra en la primera posición, ya que el algoritmo termina en la primera iteración. Entonces decimos que el tiempo en el mejor caso es 1, es decir

Algoritmo 7.1 Búsqueda secuencial

```

1 // PRE: se busca dato en el vector vec de longitud n
2 //     el vector no necesita estar ordenado
3 // POST: devuelve la posición en donde se encuentra el dato
4 //     de no encontrarse devuelve -1
5 int posicion(int []vec, int n, int dato) {
6     int pos = 0;
7     bool esta = false;
8     while ((pos < n) && (!esta)) {
9         if (vec[pos] == dato)
10             esta = true;
11         else
12             pos++;
13     }
14     if (esta)
15         return pos;
16     return -1;
17 }
```

$T(n) = 1$ (no estamos teniendo en cuenta las inicializaciones de las variables ni los tiempos de pasaje de datos, en la siguiente sección haremos un cálculo exacto). El peor caso es cuando el dato está en la última posición o no se encuentra, dado que ejecutará los n ciclos. Por lo tanto, $T(n) = n$. El caso promedio será la cantidad de iteraciones hasta llegar a la mitad del vector, suponiendo que el dato tiene la misma probabilidad de estar en cada una de las posiciones. Para hacer este cálculo de manera analítica hacemos lo siguiente:

$$\begin{aligned}
 T(n) &= 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + 3 \cdot \frac{1}{n} + \dots n \cdot \frac{1}{n} \\
 &= \sum_{i=1}^n i \cdot \frac{1}{n} \\
 &= n \cdot \frac{n+1}{2} \cdot \frac{1}{n} \\
 &= \frac{n+1}{2}
 \end{aligned}$$



En el desarrollo anterior el $1/n$ de cada término es la probabilidad que tiene el elemento de estar en esa posición. Estamos asumiendo una probabilidad uniforme, es decir, todos tienen la misma probabilidad.

El $n \cdot \frac{n+1}{2}$ surge de la sumatoria de los números 1, 2, etc.

En la práctica en general trabajamos con el peor caso, ya que nos da una

idea de cuál es el tiempo máximo que puede tardar un algoritmo, es decir, nos da una cota superior para prever situaciones indeseables.

7.2. Conteo

En el cálculo de la complejidad temporal, hay operaciones a las cuales se les asigna una duración de una unidad de tiempo. Estas operaciones las llamamos *operaciones elementales* (OE). Si bien un tipo de operación elemental no tiene por qué tener la misma duración que otra, por ejemplo, una suma demandará menos tiempo de procesamiento que una división, podemos acotar ese tiempo por una constante.

Tomaremos como operaciones elementales a:

- Asignación de variables
- Operaciones aritméticas básicas: suma, resta, producto, división.
- Comparaciones lógicas.
- Llamadas a funciones y retorno de ellas.
- El acceso a una estructura indexada, como un vector o matriz.

Veamos algunos ejemplos de conteo.

Ejemplo 7.1. En el algoritmo 7.2 tenemos una estructura exclusivamente secuencial, por lo tanto el peor, el mejor y el caso promedio coincidirán. El total de operaciones elementales que contabilizamos son cinco:

- En la primera línea tenemos una asignación
- En la segunda línea hay una suma, además de una asignación
- En la tercera línea sucede lo mismo a pesar de que parece una única operación. Sin embargo esa línea es equivalente a hacer $a = a + 1$.

Ejemplo 7.2. Veamos un algoritmo con una bifurcación, es decir, con una estructura condicional, como lo es el código 7.3.

- En las primeras dos líneas tenemos una operación elemental en cada una, dado que son dos asignaciones. Si bien son de distinto tipo porque la asignación de la segunda línea se realiza desde el teclado, también la consideramos una OE. Hasta el momento el algoritmo es secuencial y siempre contabilizará dos operaciones elementales.

Algoritmo 7.2 Ejemplo de conteo en una estructura secuencial

```

1  int a, b = 5;    // 1 O.E.
2  a = b + 1;      // 2 O.E.
3  a++;            // 2 O.E.
4
5  Total: T(n) = 5 O.E.
```

Algoritmo 7.3 Ejemplo de conteo en una estructura condicional

```

1  int a, b = 2;    // 1 O.E.
2  cin >> a;       // 1 O.E.
3  if (a > 5)       // 1 O.E.
4  {
5      a = a + 3;   // 2 O.E.
6      b = a + 1;   // 2 O.E.
7  }
8  else
9      b = 1;       // 1 O.E.
```

- En la línea número 3, hace una comparación lógica: contamos una OE más.
- A partir de la línea 3 la cantidad de operaciones lógicas dependerá del valor que el usuario ingresó. Si es mayor a 5, la condición dará verdadera e ingresará en el primer bloque, realizando cuatro OE más, y finalizando (líneas 5 y 6).
- En cambio si la condición es falsa ejecutará solo la asignación que está en la línea 9, es decir, una OE más.

Por lo tanto:

- En el mejor caso, el tiempo será $T(n) = 1 + 1 + 1 + 1 = 4$
- En el peor caso, el que más estudiaremos, el tiempo será $T(n) = 1 + 1 + 1 + 2 + 2 = 7$
- El caso promedio no podremos calcularlo simplemente viendo el código, ya que depende de la probabilidad de que el usuario ingrese un número mayor a 5. Lo que sí sabremos es que estará entre 4 y 7.

Ejemplo 7.3. En el caso de que haya bucles se debe sumar el número de OEs del cuerpo del bucle con las OEs del análisis de la condición, multiplicado

Algoritmo 7.4 Ejemplo de conteo en una estructura repetitiva

```

1 int n, i = 0, resultado = 1;      // 2 O.E.
2 cin >> n;                        // 1 O.E.
3 while (i < n)                     // 1 O.E.
4 {
5     resultado *= 2;               // 2 O.E.
6     i++;                         // 2 O.E.
7 }
8 cout << resultado;               // 1 O.E.

```

por el número de veces que se lleve a cabo la misma. En el algoritmo 7.4 tenemos que el tiempo es constante para un determinado valor de n . No hay que confundir menor caso con un n chico, siempre el peor y el mejor caso se estudian con el mismo valor de entrada. Por lo tanto:

- En la línea 1 tenemos dos asignaciones: 2 OE.
- En la línea 2, una asignación más: 1 OE.
- Dentro del bucle tenemos: en la línea 5, una multiplicación y una asignación, en la línea 6, una suma y una asignación. Es decir, 4 OE. Además está el chequeo de la condición, que es una más. Por lo tanto, estas 5 OE se ejecutarán n veces, y la comparación lógica una vez más, cuando $i = n$, que dará falso.
- En la línea 8 tenemos una impresión por pantalla: una OE más.

Teniendo en cuenta lo anterior, el tiempo total será $T(n) = 2+1+5*n+1+1 = 5n + 5$

Ejemplo 7.4. Ahora estamos preparados para hacer un conteo fiel de la búsqueda secuencial que codificamos en 7.1. Habíamos dicho que el mejor caso era cuando el dato estaba al principio y era $T(n) = 1$, pero no contamos cuántas OE hay exactamente. Vayamos viendo cada una de las líneas y cada bloque.

- En la cabecera de la función (línea 5) tenemos el pasaje de tres parámetros, por lo tanto tenemos 3 OE. Hay que tener en cuenta que los vectores no se copian enteros, sino que se utiliza la dirección del primer elemento, por lo tanto el pasaje por parámetro del vector cuenta como una sola OE al igual que las otras variables.
- En la línea 6 ponemos la variable *pos* en 0: 1 OE.

- En la línea 7, la booleana en falso: 1 OE.
- En la línea 8, debe comprobar dos condiciones. Los compiladores optimizan estas operaciones, en este caso hay un *and* entre ambas, por lo que si la primera condición da falso, no corrobora la segunda ya que no tendría sentido comprobar su valor. Sin embargo, esto solo puede suceder en el último caso cuando *pos* llegue al valor de *n*. Por lo tanto, serán 2 OE, salvo en el último caso que será 1.
- Dentro del cuerpo del *while*, hay un *if* (línea 9). En esa instrucción hay 2 OE: una por el acceso al vector y otra por la comparación.
- Si la condición de la línea 9 dio verdadero, tenemos una asignación en la línea 10: 1 OE.
- Si la condición de la línea 9 dio falso, tenemos 2 OE: el incremento y la asignación.
- A la salida del bucle *while*, línea 14, tenemos una consulta: 1 OE.
- Sin importar el valor de la línea 14, tendremos 1 *return*, ya sea el de la línea 15 o el de la 16: 1 OE.

Por lo tanto, tendremos que:

- En el mejor caso, el dato estará en la primera posición, la cantidad de OE será

$$T(n) = 3 + 1 + 1 + 2 + 2 + 1 + 2 + 1 + 1 = 14$$

Los primeros tres términos se ejecutan siempre, luego tenemos un 2 de las comparaciones del *while*, otro 2, por el *if*, y otro 2 porque vuelve a hacer las comparaciones en el *while* antes de salir. El penúltimo término es el del *if* de la línea 14, y el último del *return*.

- En el peor caso, el dato no se encontrará, por lo que el ciclo *while* se ejecutará *n* veces mediante la rama del *else*. Por lo tanto tenemos que

$$T(n) = 3 + 1 + 1 + (2 + 2 + 2) * n + 1 + 1 + 1 = 6n + 8$$

En la última consulta del *while* contabilizamos una sola OE ya que la segunda condición no la verifica.

- Dejamos de tarea para el lector el cálculo del caso promedio. Si el valor a buscar tiene la misma probabilidad de estar en cualquiera de las posiciones, el ciclo se ejecutará $n/2$ veces, pero en las primeras $n/2 - 1$ veces irá por la rama del *else*, en cambio, en la última dará un valor verdadero en la condición del *if*.

Antes de seguir con la siguiente sección dejamos algunos ejercicios para realizar.

Ejercicio 7.1. Contar las OE del siguiente algoritmo:

```
int i, a = 0;
for (i = 0; i < 10; i++)
    a = a + i;
```

Ejercicio 7.2. Contar las OE del siguiente algoritmo:

```
int i, n, a = 0;
cout << "Ingrese un entero positivo" << endl;
cin >> n;
for (i = 0; i < n; i++)
    a = a + i;
cout << "La sumatoria es: " << a << endl;
```

Ejercicio 7.3. Indicar las OE en el mejor caso y en el peor del siguiente algoritmo:

```
int i, k;
cout << "Ingrese un entero positivo" << endl;
cin >> i;
if (i > 5)
    k = i + 30;
else
    k = 2;
```

Ejercicio 7.4. Indicar las OE del siguiente código para el mejor caso, el peor y el caso promedio. Tener en cuenta que la función *rand()* es una OE en sí.

```
int i, k, a;
for (a = 0; a < 10; a++) {
    i = rand() % 10;
    if (i >= 5)
        k = i + 30;
    else
        k = 2;
    cout << i << " " << k << endl;
}
```

Luego, utilizando la librería *cstdlib*, escribir el código agregando un contador de OE que se incrementará cada vez que se haga una operación elemental. Dicho contador solo contabiliza los totales, no se tiene en cuenta como una OE aparte. Al finalizar el programa se debe imprimir el contador, verificar la salida con lo calculado de manera teórica. Por último, cambiar la semilla de la función random con *srand*, utilizando el reloj de la máquina (agregar la librería *ctime*), correrlo varias veces y sacar conclusiones para el caso promedio.

Ejercicio 7.5. Indicar las OE del algoritmo de ordenamiento de selección para el mejor caso (si ya está ordenado), el peor caso (está ordenado en forma descendente) y el caso promedio.

```
void inter(int &a, int &b) {
    int aux = a;
    a = b;
    b = aux;
}

void ordenar(int v[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (v[j] < v[i])
                inter(v[i], v[j]);
}
```

Luego agregar una variable global que cuente todas las OE, agregar la siguiente porción de código y ejecutarlo varias veces con distintos valores de *n*. Comparar con lo calculado anteriormente. Nota: la función *cargar* no la tenemos en cuenta para el conteo.

```
void cargar(int v[], int n) {
    srand(time(0));
    for (int i = 0; i < n; i++)
        v[i] = rand();
}
```

Ejercicio 7.6. Repetir el objetivo del ejercicio anterior pero con el método de ordenamiento burbujeo mejorado.

```
void burbujeo(int v[], int n) {
    bool ordenado = false;
    int i = 0, j;
    while ((i < n-1) and (! ordenado)) {
        ordenado = true;
        j = n;
        while (j > i) {
            if (v[j] < v[j-1]) {
```

```

        ordenado = false;
        inter(v[j], v[j-1]);
    }
    j--;
}
i++;
}
}

```

SECCIONES

- Nociones
- Conteo
- Análisis de distintos algoritmos
- Conteo en recurrencia?

7.3. Medidas asintóticas

Como podemos intuir, el cálculo de la expresión del número de operaciones elementales puede ser muy engorroso. Ahora bien, los casos que estudiaremos son para valores de entrada muy grandes y para poder comparar algoritmos obviaremos los términos de menor peso quedándonos con el término principal. Por ejemplo, si tenemos dos algoritmos en donde las OE son $T_1(n) = 3n^2 + 5n$ y $T_2(n) = 4n^2 + 7$, podremos despreciar el término $5n$ del primer algoritmo ya que cuando n es muy grande el término cuadrático tendrá una incidencia mucho mayor que el término lineal. Lo mismo sucede con la constante del segundo algoritmo. Además, al comparar ambos algoritmos diremos que son del mismo orden ya que los dos tendrán la misma forma a pesar de tener coeficientes principales distintos.

Por lo tanto, partimos de la idea de que, en general, no interesará calcular el número exacto de OE de un algoritmo, sino de acotarlo apropiadamente. La cota superior, por ejemplo, será una función no superada por aquella que expresa el número de OA, a partir de cierto valor de n . Por ejemplo, para el caso de los dos algoritmos anteriores, ambas funciones son acotadas, por ejemplo, por $f_1(n) = 5n^2$ o $f_2(n) = n^3$ a partir de cierto valor de n . Hay infinitas funciones que acoten superiormente a estas dos, obviamente trataremos de hallar la más chica para poder estimar la eficiencia del algoritmo y compararlo con otros.

De esta forma podremos decir que un algoritmo tiene orden lineal o cuadrático o logarítmico, y esto expresará el comportamiento dominante cuando

el tamaño de la entrada sea grande sin necesidad de calcular el coste exacto. Cabe destacar que cuando estudiemos el comportamiento asintótico, en general, estaremos analizando el peor caso de un algoritmo.

Podemos estar interesados en encontrar una función que, multiplicada por una constante, acote ese tiempo superiormente o acote ese tiempo inferiormente. También podemos buscar una función que multiplicada por dos constantes distintas en lugar de una permita acotar el tiempo tanto superior como inferiormente. Las medidas asintóticas definen las características de funciones que verifican lo que nos interesa. Veremos estas tres medidas.

7.3.1. Cota superior: notación O

El concepto de O grande o también llamada Big O o Big Omicron dice que un algoritmo es de orden $O(f(n))$ si existe alguna constante c tal que $T(n) \leq c \cdot f(n)$ para n grandes. Es decir, que los tiempos reales nunca podrán superar a la función propuesta multiplicada por cierta constante.

Definición. f pertenece a $O(g)$ si y solo si existen constantes positivas c y n_0 , tales que para todo $n \geq n_0$ se cumple que

$$0 \leq f(n) \leq c \cdot g(n)$$



Nota. Los valores de c y de n_0 no dependen de n , sino de f .

Ejemplo 7.5. Si el tiempo de un algoritmo es $T(n) = 5n^2 + 3n$, decimos que dicho algoritmo es $O(n^2)$ pero también $O(n^3)$ y $O(2^n)$, etcétera.

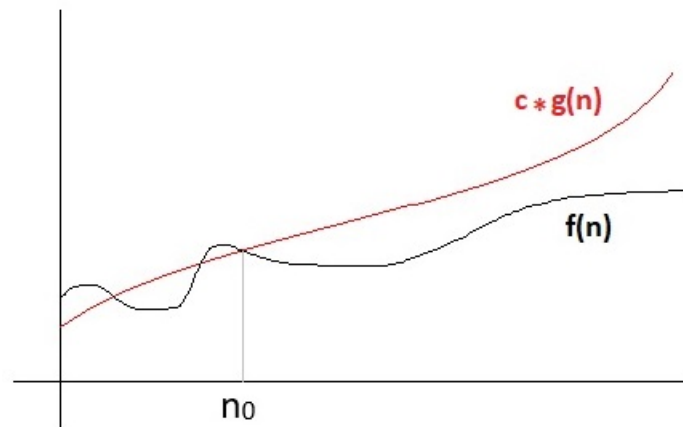
El lector podrá pensar que $5n^2 + 3n$ no es menor que n^2 , pero hay que recordar que no estamos indicando esa constante c , en este caso, si c fuera 8, la desigualdad se cumpliría para todo $n \in \mathbb{N}$. De todos los órdenes que indicamos siempre buscaremos quedarnos con el más chico, en este caso el de orden cuadrado. Pero hay que tener en cuenta que no siempre podremos calcularlo con exactitud, por eso usamos una cota.

Ejemplo 7.6. Si lo pensamos al revés: qué funciones incluye un $O(n^3)$, podemos escribir:

$$O(n^3) = \{4n^3 + 5n, 8n^2, 10n^2 + 6n - 7, n + 1, \dots\}$$

es decir, las infinitas funciones cúbicas o menores cuando n es un valor grande.

Podemos apreciar el concepto de O grande en el gráfico 7.3.1.

Gráfico 7.3.1: Cota superior: f es $O(g)$

Es importante tener en cuenta que $O(n^3)$ incluye a $O(n^2)$ el cual, a su vez, incluye $O(n \cdot \log(n))$, etcétera. Por lo tanto

$$O(n) \subset O(n \log(n)) \subset O(n^2) \subset O(n^3)$$

Cuando el $T(n)$ es una constante, es decir, cuando el algoritmo hace una cantidad finita de pasos independientemente del valor de entrada, decimos que es de orden $O(1)$, aunque esa cantidad de pasos fuera un número muy grande.

Propiedades de Big O

Cuando un algoritmo es largo y tiene varios bloques puede ser difícil calcular el orden. Pero utilizando las propiedades que listamos a continuación podremos dividir la tarea y utilizar cálculos ya realizados.

- a. Identidad: f es $O(f)$
- b. Inclusión: Si $O(f)$ es $O(g) \Rightarrow O(f) \subseteq O(g)$
- c. Doble inclusión: $O(f) = O(g) \iff f$ es $O(g)$ y g es $O(f)$
- d. Transitiva: Si f es $O(g)$ y g es $O(h) \Rightarrow f$ es $O(h)$
- e. Cota mínima: Si f es $O(g)$ y f es $O(h) \Rightarrow f$ es $O(\min \{g, h\})$
- f. Suma: Si f es $O(h)$ y g es $O(i) \Rightarrow f + g$ es $O(\max \{h, i\})$

g. Producto: Si f es $O(h)$ y g es $O(i) \Rightarrow f \cdot g$ es $O(h \cdot i)$

h. División: Si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k$, con k finito, entonces:

a) Si $k = 0$, f es $O(g)$ pero g no es $O(f)$

b) Si $k > 0$, entonces $O(f) = O(g)$

Los O se pueden ordenar de forma creciente, lo cual nos permite comparar la eficiencia temporal de los algoritmos mediante ellos. En este sentido, se verifica:

- Cualquier función exponencial de n domina sobre otra función polinomial de n .
- Cualquier función polinomial de n domina sobre una función logarítmica de n .
- Cualquier función de n domina sobre un término constante.
- Un polinomio de grado k domina sobre un polinomio de grado l si $k > l$.

Resumiendo:

Los comportamientos asintóticos para O de más frecuente aparición se pueden ordenar de menor a mayor según su crecimiento de la siguiente forma:

$$O(1) \subset O(\log N) \subset O(N) \subset O(N \cdot \log N) \subset O(N^2) \subset O(N^3) \dots O(2^N) \subset O(N!)$$

¿Cómo calcular O para el peor caso de forma sencilla y rápida?

Hay que tener en cuenta:

- a. Las O.E. son $O(1)$
- b. En las secuencias se suman las complejidades individuales aplicando la regla de la suma. Por ejemplo, una secuencia finita de bloques de $O(1)$ es $O(1)$.
- c. En los bucles en los que el número de iteraciones es fijo, el O es el del bloque ejecutado.

Ejemplo 7.7. Calcular el orden de:

```
int i, a = 2, b = 6, c = 1; //O(1)
for (i = 0; i < 1000; i++) {
    a = a * b + c - 5; //O(1)
}
```

En la primera línea se ejecutan algunas sentencias que son $O(1)$. Luego, en el ciclo se ejecuta 1000 veces una sentencia que es también $O(1)$, por lo tanto este código es

$$O(1) * O(1) = O(1)$$

por aplicación de la regla del producto.

Ejemplo 7.8. Bucles en los cuales el número de iteraciones corresponde al tamaño n del problema: el orden es el O del bloque multiplicado por n .

```
int i, a = 2, b = 6, c = 1, n; //O(1)
cout << "Ingrese el valor de n"; // O(1)
cin >> n; // O(1)
for (i = 0; i < n; i++) {
    a = a * b + c - 5; //O(1)
}
```

En el ciclo se ejecuta n veces algo de $O(1)$, por lo tanto este código, por aplicación de la regla del producto es n veces $O(1)$, o bien

$$O(N) * O(1) = O(N)$$

Ejemplo 7.9. En los bucles anidados se multiplican los O correspondientes a cada anidamiento.

```
int i, j, n, a = 3, b = 2; //O(1)
cout << "Ingrese el valor de n"; // O(1)
cin >> n; //O(1)
for (i = 0; i < n; i++) {
    for (j = 2; j > n; j++) {
        a = a+b;
        b++;
    }
}
```

El bloque interior que es $O(1)$ depende de un par de bucles anidados. Ambos iteran dependiendo de n , por lo tanto es

$$O(n) * O(n) = O(n^2)$$

por aplicación de la regla del producto.

Ejemplo 7.10. En las bifurcaciones se suma el O correspondiente al análisis de la condición más el de la peor rama.

```

int i, n, a = 1, b = 20; // 0(1)
cout << "Ingrese el valor de n"; // 0(1)
cin >> n; // 0(1)
if (n > b) // 0(1)
    for (i = 0; i < n; i++) {
        a = a * b + 5
    } //esta rama es 0(n)
else
    b = 1; //la otra rama es 0(1)

```

Entonces $T(n)$ es $O(n)$

7.3.2. Cota inferior: notación Ω

La notación Ω se refiere a la medida asintótica en los mejores casos. No es una medida crítica ya que tiene una visión optimista y no sirve para prevenir inconvenientes.

Definición:

$f(n)$ es $\Omega(g(n))$ si y solo si existen constantes positivas c y n_0 , tales que se verifica, para todo $x > n_0$, $0 \leq c * g(n) \leq f(n)$ para todo $n \geq k$. (los valores de c y n_0 no dependen de n , sino de f)

En símbolos:

$$\Omega(g(n)) = \{f : N \rightarrow R^+ : \exists c \in R^+, n_0 \in N, \forall n > n_0 : c * g(n) \leq f(n)\}$$

Es el conjunto de funciones que “acotan por abajo”.

Por ejemplo: Son $\Omega(N^2)$, N^2 , $15 * N^2$, $N^2 + 15 * N$, N^3 , $100 * N^5$, ...

Se debe observar que $f(n)$ es $\Omega(g(n))$ sii $g(n)$ es $O(f(n))$. En textos en inglés, suele decirse que la función f es ‘lowly bounded’ por g

La expresión $f(n) = (g(n))$ se acepta como abuso de notación, debería escribirse como $f(n)$ es o pertenece a $\Omega(g(n))$.

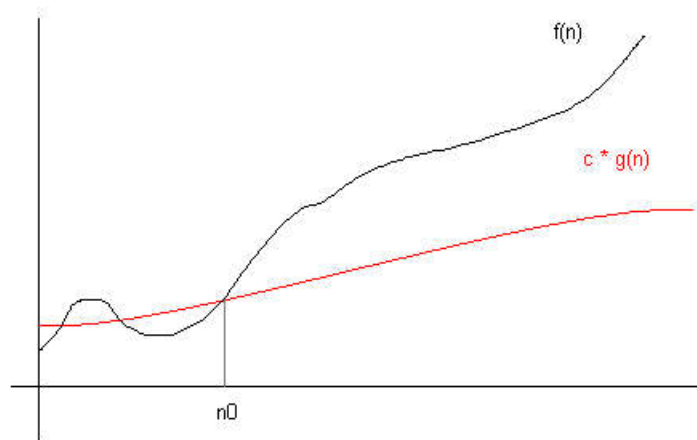
7.3.3. Orden exacto: notación Θ

Cuando se acota la complejidad de un algoritmo tanto superior como inferiormente por la misma función se usa la notación Theta. En estos casos se dice que $f(n)$ está acotado por “arriba” y por “abajo”, para n suficientemente grande. $\Theta(g)$ es el conjunto de funciones que crecen exactamente al mismo ritmo que g . f pertenece a $\Theta(g)$ si g es a la vez cota superior e inferior de f .

$f(n)$ es o pertenece a $\Theta(g(n))$ significa que existen constantes reales positivas c_1 , c_2 , y k , tales que se verifica $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ para todo $n \geq n_0$.

Ejemplo: Son $\Theta(n^2)$: n^2 , $3n^2 + 2n$, $54n^2 + 20n + 90$, ... Se dice que g y f poseen el mismo orden de crecimiento.

Si f es $\Theta(g)$, g es el orden exacto de f . Se verifica que $\Theta(f) = O(f) \cap \Omega(f)$.

Gráfico 7.3.2: Medida Ω : $f(n)$ es $\Omega(g(n))$

Si $f(n)$ es $\Theta(g(n))$ se dice que f es de orden (de magnitud) g , o que f es de (o tiene) coste (o complejidad) g .

7.4. Análisis de complejidad en algoritmos recursivos

En el caso de algoritmos recursivos, el cálculo de O da origen a expresiones de recurrencia que tienen diversas formas de resolución. Básicamente, estos modos de resolución son tres:

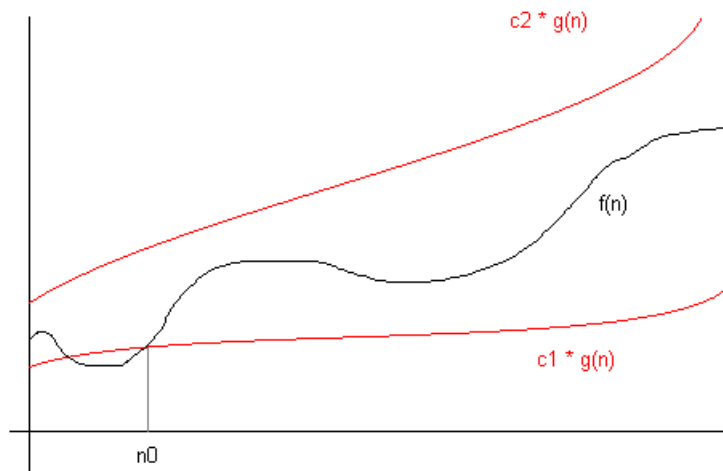
- Usando el método de expansión. En este curso mostraremos cómo puede aplicarse este método apelando también a la intuición en algunos pasos.
- Usando los métodos de resolución de ecuaciones de recurrencia que nos provee el Análisis Numérico (excede los objetivos de este curso).
- Usando algún teorema cuando se den las condiciones para su aplicación.

Consideremos algunos casos de algoritmos recursivos:

Ejemplo 7.11. Análisis de la Búsqueda Binaria recursiva.

Como ya sabemos, el algoritmo codificado indica:

```
// retorna el subíndice de la posición en donde está el dato
// o bien -1 si no está
int BB (int v[ ], int pri, int ulti, int dato) {
```

Gráfico 7.3.3: $f(n)$ es $\Theta(g(n))$

```

if ( pr > ul)
    return -1;
else {
    int medio = (pri + ulti)/2;
    if (v[medio] == dato)
        return medio;
    else if (dato > v[medio])
        return BB (v, medio + 1, ulti);
    else
        return BB (v, pri, medio - 1);
}
}

```

Analicemos paso a paso lo que indica este código. Para ello hemos realizado un diagrama en el gráfico 7.4.1 que permite ver más claramente la estructura del código.

Como vemos, las dos peores ramas tienen un coste de

$$T(N) = O(1) + O(1) + O(1) + O(1) + T(N/2)$$

$$T(N) = O(1) + T(N/2)$$

Pero $O(1)$ significa que el tiempo es acotado por una función constante, por ejemplo por $f(N) = 1$. Por lo tanto se puede y es conveniente para el cálculo expresarlo como:

$$T(N) = T(N/2) + 1$$

Esta ecuación de recurrencia expresa el coste temporal del algoritmo recursivo de búsqueda binaria en un array. Sabemos además que se verifica

$$O(0) = 1$$

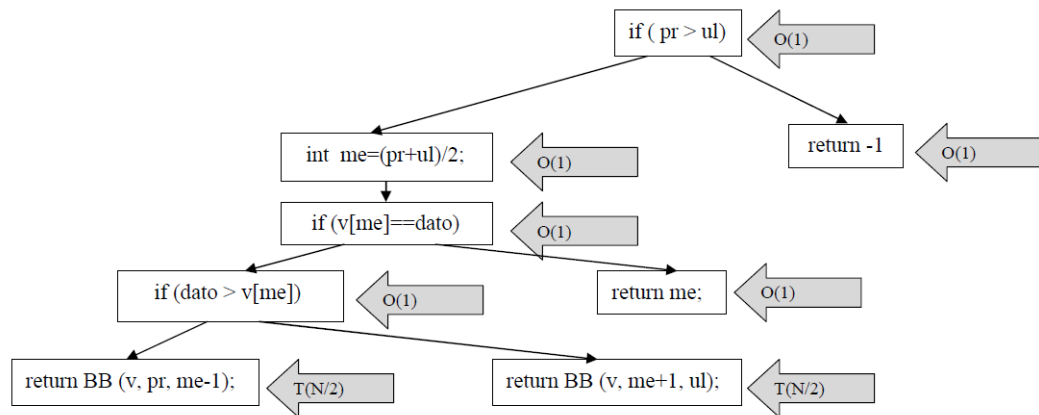


Gráfico 7.4.1: Diagrama de búsqueda binaria recursiva

que significa que si el tamaño del array es 0, el coste es 1, que es el coste de devolver el -1 , y que

$$O(1) = 1$$

si el array tiene tamaño 1, también el coste de determinar si el dato está o no es 1. Vamos a graficar el “árbol de llamadas recursivas” correspondiente a la ecuación. Un árbol de llamadas grafica lo que sucede al realizar invocaciones a funciones. Eventualmente puede usarse para mostrar lo que pasa cuando se trabaja con funciones que se llaman a si mismas, es decir recursivas. Considerando la situación de invocación inicial y llamados posteriores podemos graficar esta evolución del árbol de llamadas como vemos en 7.4.2.

Entonces: sabemos que cada nodo del árbol de llamadas debe ser ejecutado y tiene asociado un tiempo de ejecución, que en el gráfico 7.4.2 está indicado dentro del nodo mismo. El tiempo de ejecución del algoritmo será la sumatoria de los tiempos de los nodos. Tenemos nodos de coste 1, y hay una cantidad de ellos que se puede contabilizar como $\log_2 N$. Entonces

$$T(N) \in O(\log N)$$

Ejemplo 7.12. Análisis del algoritmo recursivo del cálculo del factorial de un número.

```

int factorial (int n) {
    if ((n == 0) || (n == 1))
        return 1;
    else
        return (n * factorial(n - 1));
}

```

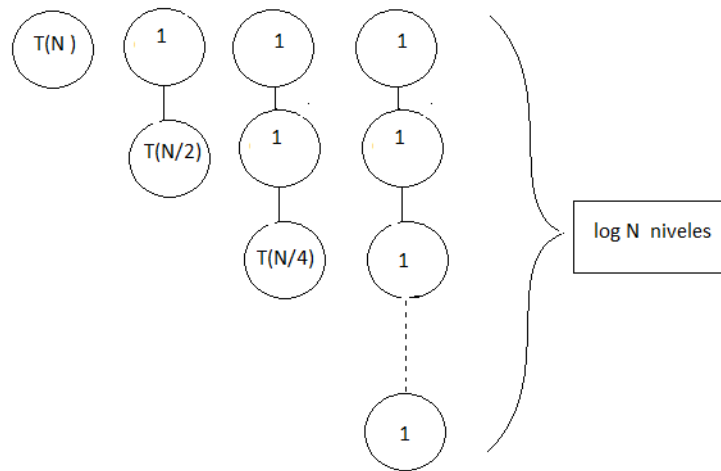



Gráfico 7.4.2: Árbol de llamadas recursivas en la búsqueda binaria

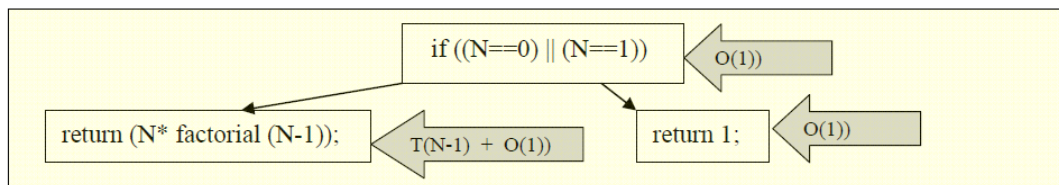


Gráfico 7.4.3: Diagrama de llamados recursivos en la función factorial

Haciendo un gráfico como en el caso anterior, nos queda el gráfico que vemos en 7.4.3.

Observar que el return de la izquierda realiza una operación más, de coste $O(1)$, luego de recibir el resultado de la invocación recursiva, que es la multiplicación por N . Entonces, el tiempo se puede expresar como:

$$T(N) = T(N - 1) + 1$$

Además

$$T(1) = T(0) = 1$$

Si realizamos, como en el ejemplo anterior, el árbol de llamadas, queda el gráfico que está en 7.4.4.

Tenemos N niveles de coste 1 cada uno. La sumatoria de los costes indica que $T(N)$ pertenece a $O(N)$.

Otra forma de resolverlo es planteando el método de expansión, donde

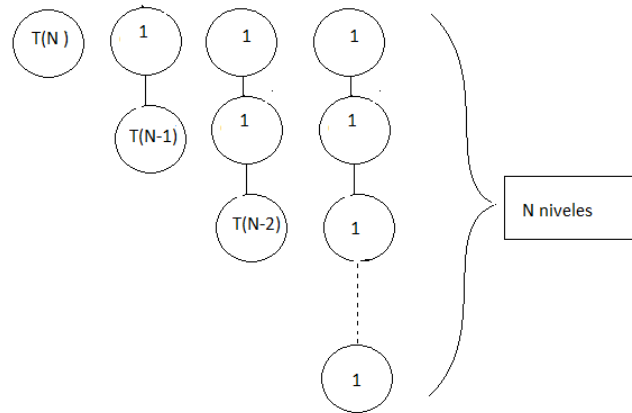


Gráfico 7.4.4: Árbol de llamadas recursivas de la función factorial

podemos hacer este desarrollo:

$$\begin{aligned} T(N) &= T(N-1) + 1 \\ T(N-1) &= T(N-2) + 1 \\ T(N-2) &= T(N-3) + 1 \end{aligned}$$

En el i -ésimo paso queda

$$T(N) = T(N-i) + i$$

y tomando $i = N$:

$$T(N) = T(0) + N$$

y como $T(0) = 1$ resulta que $T(N)$ pertenece $O(N)$.

Ejemplo 7.13. Merge sort.

7.4.1. Teorema maestro

El teorema maestro, también llamado teorema de reducción por división suele aplicarse a algoritmos que provienen de aplicar la estrategia “Divide y Vencerás”. No lo demostraremos en este curso pero podemos mostrar que lo que indica para cada caso es válido haciendo un análisis de la propiedad.

- Reducción por división:

La ecuación de la recurrencia es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la ecuación de recurrencia es:

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

siendo:

a: Número de llamadas recursivas.

b: Reducción del problema en cada llamada.

$c * n^k$: Todas aquellas operaciones que hacen falta además de las de recursividad.

Análisis de lo que dice el TM de reducción por división

Consideremos los árboles de llamadas correspondientes al enunciado del TM.

Consideremos los siguientes casos:

- $a < b^k$: entonces el tiempo de ejecución disminuye en cada nivel, predomina el tiempo de la raíz. El TM indica $O(n^k)$
- $a = b^k$: entonces el tiempo de ejecución es el mismo en cada nivel. El TM indica $O(n^k \log n)$
- $a > b^k$: el tiempo de ejecución aumenta en cada nivel, predomina el tiempo del nivel de las hojas. El TM indica $O(n^{\log_b a})$

Otra propiedad útil (a veces llamada T.M. de resolución por sustracción):

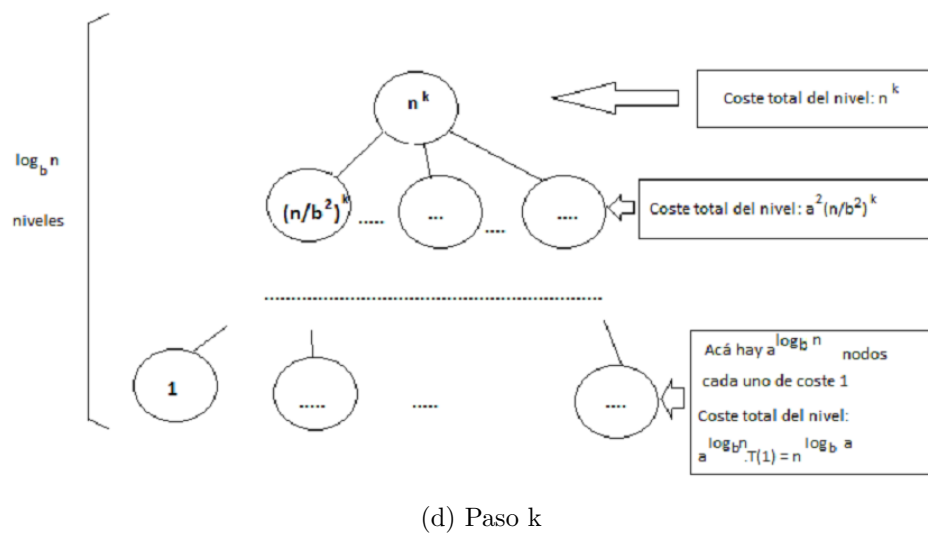
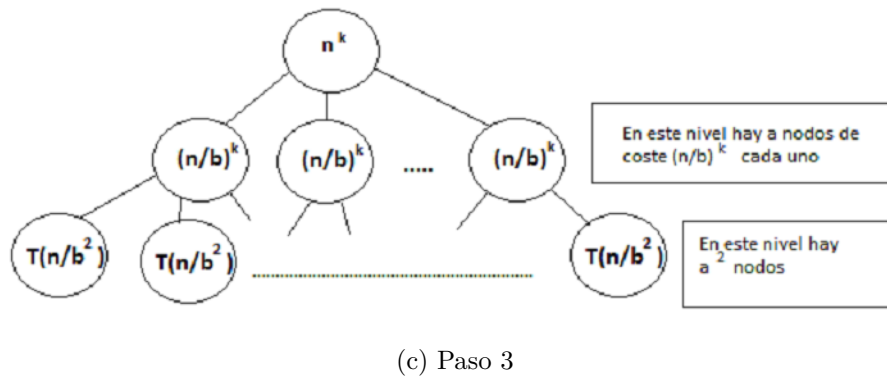
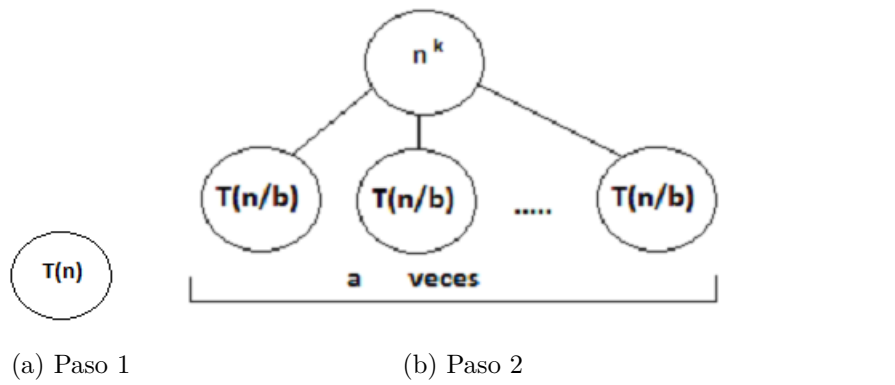


Gráfico 7.4.5: árboles de llamados en TM

Reducción por sustracción:

La ecuación de la recurrencia es la siguiente:

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

La resolución de la ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

7.5. Balance entre tiempo y espacio

Algunas cuestiones adicionales:

El análisis usando medidas asintóticas solo indica el modo de crecimiento con respecto al tamaño de la entrada. Hay que tener en cuenta, que los valores de constantes c y n_0 de la definición de la notación O a menudo ocultan detalles que son importantes en la práctica. En consecuencia, las pruebas de complejidad computacional deben ser consideradas el primer paso en un refinamiento del análisis del algoritmo que permita revelar más detalles acerca de sus características.

Ejercicios

- a. Indique el O de cada uno de estos códigos para el peor caso:

```
a)
int i = 3;
int a = 2, b;
b = i * a + 5;
```

```
b)
int i, a = 1, k = 100;
cout << "Ingrese el valor de k";
cin >> k;
for(i = 0; i < k; i++)
    a = a + i;
```

```
c)
int i, a = 1;
cout << "Ingrese el valor de n";
```

```

cin >> n;
for(i = 0; i < n; i++)
    a = a + i;

d)
int i, j, a = 1;
cout << "Ingrese el valor de n";
cin >> n;
for(i = 0; i < n; i++)
    for(j = 2; j < n; j++)
        a = a + i;

e)
int i, a = 1;
cout << "Ingrese el valor de n";
cin >> n;
for(i = 0; i < n; i++)
    a = n + i;

f)
int i = 1, n;
cout << "Ingrese el valor de n";
cin >> n;
while (i < n)
    i = i * 2;

g)
int i = 1, n;
cout << "Ingrese el valor de n";
cin >> n;
while (i < 100)
    i = i * 2 + n;

```

- b. Analizar y determinar el coste temporal, en términos de O para el peor caso de cada uno de estos algoritmos planteados de forma iterativa:
- a) Búsqueda secuencial en un array no ordenado de N elementos
 - b) Ordenamiento por selección de un array de tamaño N (y pensar también en el O del mejor caso ¿cuál es el mejor caso?)
 - c) Ordenamiento por burbujeo de un array de tamaño N (y pensar también en el O del mejor caso ¿cuál es el mejor caso?)
 - d) Ordenamiento por inserción de un array de tamaño N (y pensar también en el O del mejor caso ¿cuál es el mejor caso?)
 - e) Cálculo del factorial de un número X

- c. Intuitivamente, determinar el O del peor caso de cada una de estas operaciones sobre un array:
- a) Alta final de los elementos que se hayan incorporado previamente, conociendo la posición del último elemento.
 - b) Alta en posición 0, realizando un corrimiento de los elementos que hubiera previamente, una posición “hacia la derecha” comenzando desde el último.
- d. Considere diversas implementaciones del TDA Pila. Describa las mismas e indique:
- a) El coste de colocar un dato.
 - b) El coste de sacar un dato, teniendo en cuenta el peor caso.
- e. Idem el ejercicio anterior para un TDA Cola.
- f. Determine la expresión del coste temporal para el peor caso y calcule el O correspondiente para la siguiente función: `int suma(int v[], int N)`
`{ if (N==0) return v[0]; else return (v[N] + suma (v, N-1)) ; }`
- g. función que resuelve el problema de las “Torres de Hanoi”
- h. La siguiente función calcula la media de los números de un vector `v` que tiene un tamaño que es potencia de 2. `float Media1 (int v[], int pr, int ul)`
`{ if (pr ==ul) {return v[pr];} else {return (Media1(v, pr, (pr+ul)/2) + Media1(v, (pr+ul)/2+1, ul))/2.0}};`
- i. Para cada una de las siguientes ecuaciones básicas de recurrencia, indicar su O para el peor caso (utilizar expansión) y dar un ejemplo de algoritmo que responda a esa forma:
- a) $T(n)=T(n-1)+n$, con $T(0)=1$
 - b) $T(n)= 2 T(n-1)+n$, con $T(0)=1$
 - c) $T(n)=T(n-1)+1$, con $T(0)=1$
 - d) $T(n)= 3T(n-1)+1$, con $T(0)=1$
 - e) $T(n)=T(n/2)+1$, con $T(1)=1$
 - f) $T(n)= 2 T(n/2)+1$, con $T(1)=1$
 - g) $T(n)= 3 T(n/2)+1$, con $T(1)=1$
 - h) $T(n)=T(n/2)+n$, con $T(1)=1$

- i) $T(n) = 2 T(n/2) + n$, con $T(1)=1$
- j. Se tiene un vector de valores enteros no repetidos y ordenados de forma creciente. Desarrollar un algoritmo que establezca si algún elemento del vector tiene un valor que coincida con su índice. El algoritmo debe tener coste logarítmico.
- k. Determinar el O para el peor caso del máximo común divisor por el algoritmo de Euclides en su forma iterativa y en su forma recursiva.

Parte IV

Estructuras no lineales

Capítulo 8

Árboles

Los árboles son estructuras de datos que constan de un conjunto de nodos organizados jerárquicamente. Cada nodo tiene un solo padre uno excepto un nodo distinguido llamado raíz que no tendrá ninguno. Además, cada nodo puede tener cero o más hijos. Según las características del árbol, la cantidad de hijos puede estar limitada o no. En la imagen 8.0.1 vemos un ejemplo de la estructura de un árbol sin restricciones en cuanto a la cantidad de hijos.



Nota. Es fundamental diferenciar el TDA Arbol de la estructura de datos árbol. En el TDA Arbol las operaciones propias nos remiten a la manipulación de nodos, árboles, relación con el padre, etc. Por ejemplo, tendremos primitivas para determinar quién es el padre de un nodo, insertar un elemento como hijo de otro especificado, eliminar un subárbol, etc. Pero el TDA Arbol puede implementarse a través de una estructura que no sea arborescente. Como en todo TDA, lo que hace que sea Arbol y no otro tipo es aquello que ofrece a través de su interfaz, considerando estas primitivas junto con sus precondiciones y postcondiciones.

La cantidad de arcos que separan a un nodo de su raíz es la longitud del camino, el cual es único. Si un nodo no tiene hijos diremos que ese nodo es una hoja. Por ejemplo, en el gráfico 8.0.2 vemos la estructura del reino animal según una de las posibles clasificaciones. En donde

- *Reino animal* es el valor que contiene el nodo raíz.
- *Artrópodos*, *Moluscos*, *Mamíferos*, etc, son todos nodos hoja. Si hubiéramos agregado *Crustáceos* como hijo de *Artrópodos*, este último nodo dejaría de ser una hoja.

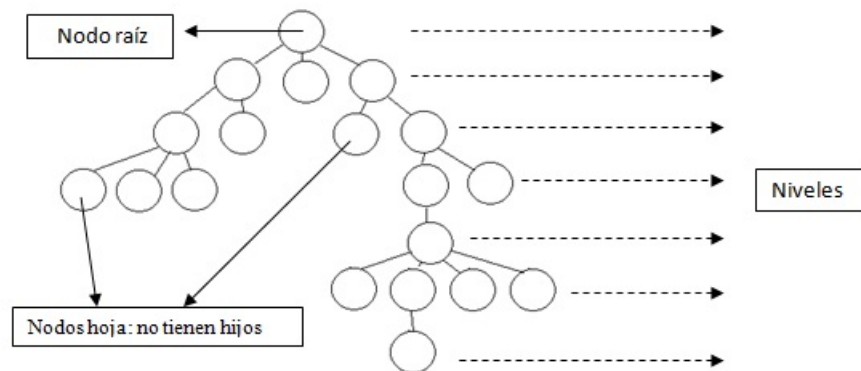


Gráfico 8.0.1: Estructura de un árbol

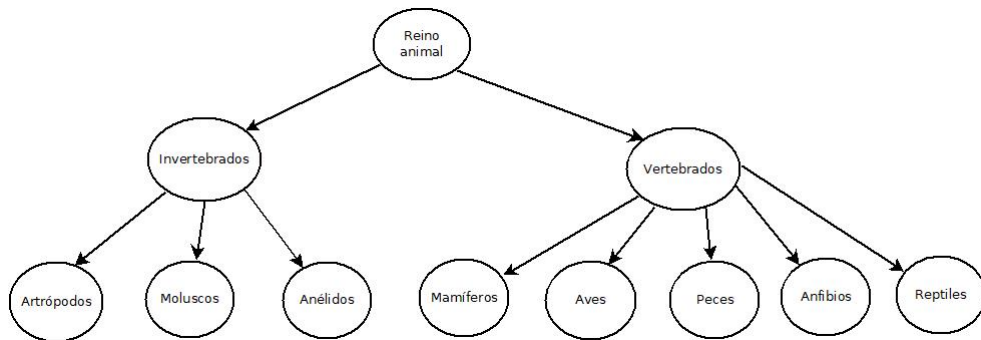


Gráfico 8.0.2: Ejemplo de árbol

- El padre de *Artrópodos*, *Moluscos* y *Anélidos* es *Invertebrados*.
- *Reino animal* es un ascendiente de *Artrópodos*, *Mamíferos*, *Aves*, etc.
- *Artrópodos*, *Mamíferos*, etc, son descendientes de *Reino animal*.
- La longitud del camino *Reino animal* - *Mamíferos* es 2.

8.1. Árboles binarios

En un árbol binario cada nodo puede tener, a lo sumo, dos hijos. Los llamaremos hijo izquierdo y derecho. La representación la podemos ver en el gráfico 8.1.1.

Propiedades de un árbol binario:

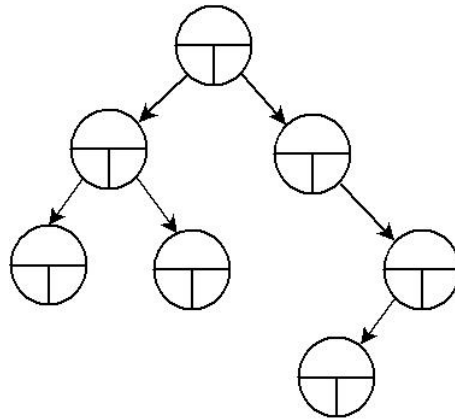


Gráfico 8.1.1: Árbol binario

- Si el árbol no es vacío, cada nodo tiene un solo padre, exceptuando al nodo raíz que no tiene.
- Cada nodo puede tener 0, 1 o 2 hijos.

Implementación

Para implementar un árbol binario podemos utilizar estructuras estáticas, como un *array* o estructuras dinámicas. Veamos con un ejemplo cómo sería alguna implementación estática. Una aclaración, en todos los ejemplos vamos a utilizar árboles con números para facilitar la explicación del tema, pero la realidad es que los datos a guardar serán claves de una información más compleja, con algún puntero a dicha información. Por ejemplo, un dato *Alumno*, podría tener nombre, dirección, teléfono, correo electrónico, carrera que cursa, promedio, etc. Además tendrá como clave su número de legajo. En los nodos, probablemente, guardaremos este número de legajo y una dirección de memoria donde está el resto de los datos, ya que guardarlos por completo en el árbol lo haría inmanejable si pensamos que la cantidad de alumnos de una facultad podría ser de unos cuantos miles.

Estructuras estáticas

Supongamos que tenemos el árbol del gráfico 8.1.2. Para guardarlo en una estructura estática colocaríamos el valor de la raíz en la primera posición de un vector, en este caso 25. Cada celda del array, es decir, cada nodo, debe guardar además de la información dos datos: la posición del hijo izquierdo y la del derecho. Si no tuviera hijos se pondrá un -1 . De esta forma, salvo

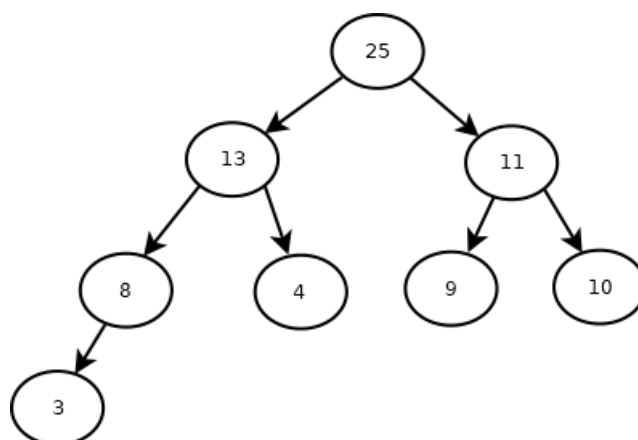


Gráfico 8.1.2: Ejemplo árbol binario

índice	dato	hijo izquierdo	hijo derecho
0	25	5	2
1	8	6	-1
2	11	4	7
3	4	-1	-1
4	9	-1	-1
5	13	1	3
6	3	-1	-1
7	10	-1	-1

Tabla 8.1.1: Representación de un árbol binario en una estructura estática.

el dato de la raíz, los demás pueden estar ubicados en cualquier posición del array. Lo vemos en la tabla

Otra implementación más eficiente que la anterior sería destinar la primera celda a la raíz, las dos siguientes a los hijos, las cuatro siguientes a los hijos de los hijos, etc. Por supuesto que para una representación así el árbol debería estar completo o semi ya que no deben quedar celdas vacías, a menos que se indiquen de alguna manera, por ejemplo, con un puntero nulo. En el gráfico 8.1.3, parte a) vemos la idea general y en la parte b) vemos cómo queda aplicado al ejemplo anterior.

Las estructuras estáticas en general no serán de gran utilidad ya que no nos permiten crecer más allá de un tope inicial, además son problemáticas a la hora de eliminar nodos debido al desplazamiento de celdas, si los hubiera. Sin embargo, la idea recién mostrada nos será útil a la hora de ordenar un vector mediante el método *heap sort* que veremos más adelante.

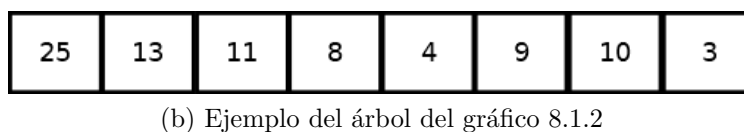
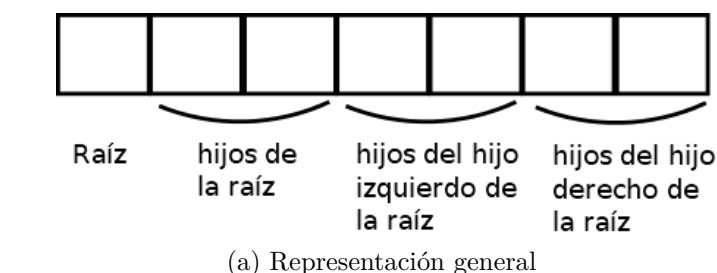


Gráfico 8.1.3: Otra representación estática de un árbol

- Una estructura con punteros es la más utilizada, cada nodo tiene dos punteros o enlaces a cada uno de sus hijos: el izquierdo y el derecho, siendo nulo uno o ambos en caso de no tenerlos. El gráfico es el mismo que observamos en el ejemplo 8.1.2.

8.2. Árboles binarios de búsqueda

La estructura de un árbol binario se aprovecha cuando los datos pueden ser ordenados, ya que las operaciones de búsqueda, inserción y borrado son más eficientes que en una estructura lineal. Cuando tenemos un árbol vacío, el primer dato a insertar obviamente será la raíz, ya que es el único hasta el momento. Cuando se inserte un nuevo dato, se comparará con la raíz, si es menor irá a formar parte del hijo izquierdo y si es mayor, del derecho. Esta comparación se irá repitiendo hasta insertarse en un nodo hoja.

Un árbol binario de búsqueda (ABB) es un árbol binario que satisface:

- Si el árbol no es vacío, la clave que se encuentra en el nodo raíz es más grande que cualquier clave del subárbol izquierdo.
- Si el árbol no es vacío, la clave que se encuentra en el nodo raíz es más chica que cualquier clave del subárbol derecho.
- Los subárboles de izquierda y de derecha son, a su vez, ABB.

Supongamos que tenemos un árbol vacío y se ingresan los siguientes datos en este orden: 18, 5, 9, 24, 12, 31, 22 y 27. En cada inserción el árbol irá tomando la forma que se muestra en el gráfico 8.2.1. En primer lugar, el número 18 será el único nodo y raíz del árbol. Luego, el número 5 se compara con el 18

y, como es menor, el nodo se insertará en el lugar del hijo izquierdo (figura b). Luego ingresa el 9, es menor que 18, por lo tanto baja por la rama del hijo izquierdo, donde encuentra el 5, como es mayor, se inserta como hijo derecho del nodo que contiene al 5 (figura c). Este proceso se repite con los demás valores.

La forma del árbol binario es muy importante para optimizar todos los procesos, principalmente el de búsqueda que consideraremos el más importante y el que necesitarán los otros procesos: ingreso y borrado. Si el árbol tiene una forma triangular será más eficiente que si no la tiene. Esta forma depende del orden del ingreso de datos. En el gráfico 8.2.2 vemos dos árboles con distintas configuraciones pero con los mismos datos. Del primero decimos que está mejor balanceado, en cambio el segundo se asemeja a una lista, perdiendo los beneficios que brinda una estructura arbórea.

Operaciones básicas

Las operaciones básicas que haremos con todo tipo de árboles serán: búsqueda, insertado, borrado y diferentes tipos de recorridos. Vamos a ver cada una de ellas para un ABB.

Búsqueda

La búsqueda es muy simple: debemos comenzar por el nodo raíz e ir descendiendo a izquierda o derecha, ya sea que el valor que estemos buscando sea menor o mayor que el dato del nodo que estemos comparando. Este proceso se repetirá hasta encontrar dicho valor o hasta toparnos con un nodo hoja, lo que indicará que el valor no está en el árbol. Se puede implementar de manera recursiva o iterativa. El pseudocódigo de la búsqueda iterativa lo podemos ver en el algoritmo 8.1 y el de la búsqueda recursiva en el algoritmo 8.2.

Lo que se menciona en el algoritmo recursivo 8.2 como “buscar...” implica un descenso en el árbol hasta localizar la clave o determinar que no está en la estructura. Es importante observar que de cada nivel del ABB se analiza un solo nodo. Esto implica que en el peor caso haremos tantas comparaciones como cantidad de niveles tenga el ABB. Si el árbol binario tiene n nodos y está completo en todos sus niveles, la altura es aproximadamente $\log_2(n)$. Por eso es más conveniente que la estructura tenga aspecto triangular y no similar a una lista ligada.

Veamos qué pasa si creamos un ABB e ingresamos los siguientes datos en el siguiente orden: 3, 8, 20, 11, 14, 19. El armado del árbol lo podemos ver en el gráfico 8.2.3 paso por paso. Observamos que cada nodo salvo el último

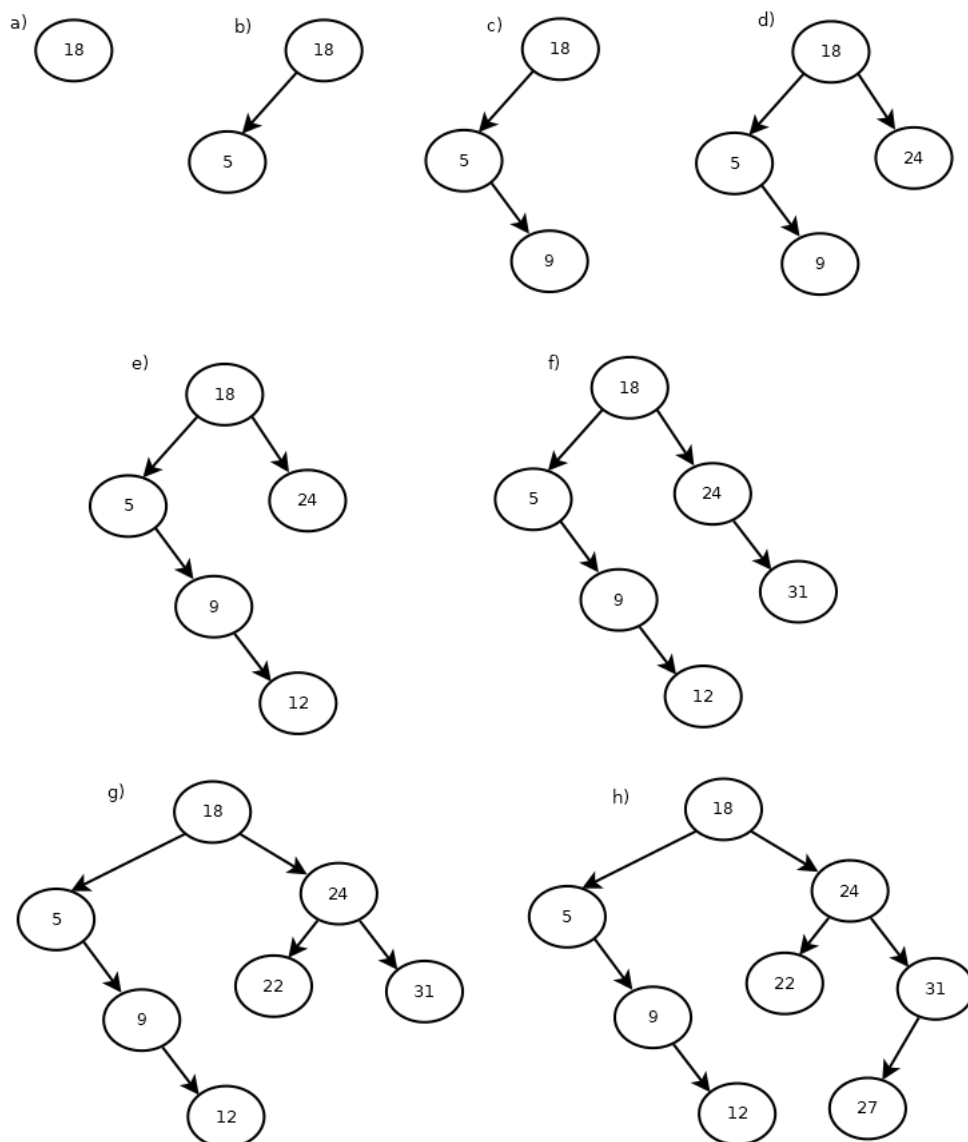


Gráfico 8.2.1: Ingreso de datos en un ABB

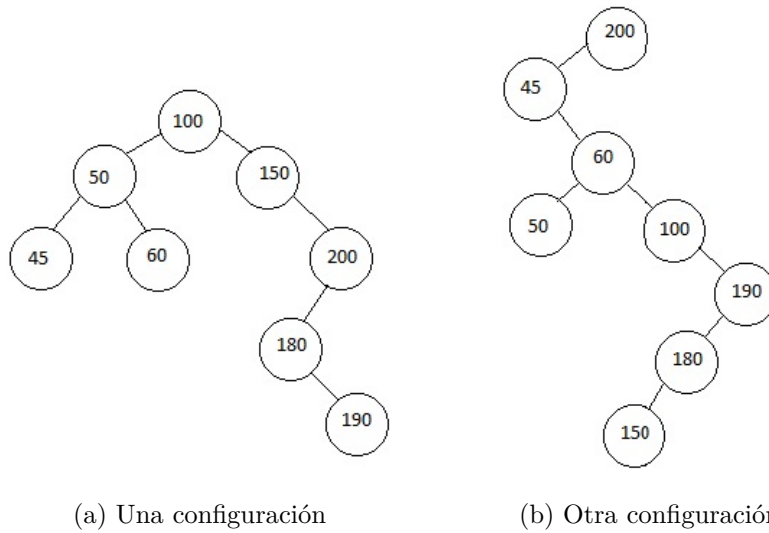


Gráfico 8.2.2: Dos árboles binarios distintos con los mismos datos

Algoritmo 8.1 Búsqueda iterativa en un ABB

```

// raiz es el puntero de entrada al ABB
// aux es puntero a nodo de ABB
// x es la clave que se busca
encontrado = falso;
si (raiz no es nulo) {
    aux = raiz;
    mientras ((aux no es nulo) y (no encontrado)) {
        si (x == clave de nodo apuntado por aux) {
            encontrado = verdadero;
        }
        si no si (x > clave de nodo apuntado por aux) {
            aux = puntero_derecho_de_aux ;
        }
        si no {
            aux = puntero_izquierdo_de_aux ;
        }
    }
}
retornar encontrado;

```

Algoritmo 8.2 Búsqueda recursiva en un ABB

```
si (p es nulo) {  
  // p es puntero a nodo, inicialmente apuntando a la raíz  
  retornar falso; // la clave x no está en el ABB - caso base 1  
}  
si no si (x == clave del nodo apuntado por p)  
{ // clave encontrada - caso base 2  
  retornar verdadero;  
}  
si no si (k > clave del nodo apuntado por p) {  
  buscar en subárbol derecho // llamado recursivo  
}  
si no {  
  buscar en subárbol izquierdo // llamado recursivo  
}
```

tiene un solo hijo, por lo tanto la configuración de este árbol es la de una lista, y el orden de la búsqueda será lineal en lugar de ser logarítmico.

Coste en la búsqueda

Como hemos visto, en el peor caso, la búsqueda será de orden $O(N)$. El orden es el mismo ya sea el algoritmo iterativo o recursivo. Es importante destacar que el tipo de recursividad es de cola, por lo que si el lenguaje permite su optimización, no tendremos un coste mayor en el recursivo.

Alta

Cuando vamos a dar de alta un valor en un ABB nos encontraremos con dos situaciones:

- El árbol está vacío. Si este es el caso, simplemente se crea un nodo que será la raíz del árbol.
- El árbol no está vacío. Se desciende comparando el dato a insertar con el valor que está en el nodo raíz, hacia el lado izquierdo o el derecho, según corresponda. Esta operación se repite hasta encontrarse con un puntero nulo, que es dónde el nodo deberá insertarse.

El pseudocódigo del algoritmo de manera iterativa se encuentra en 8.3, el de la forma recursiva se deja como ejercicio.

El coste, tanto el de la forma recursiva como el de la iterativa, es idéntico al de la búsqueda, es decir, $O(N)$ en el peor caso.

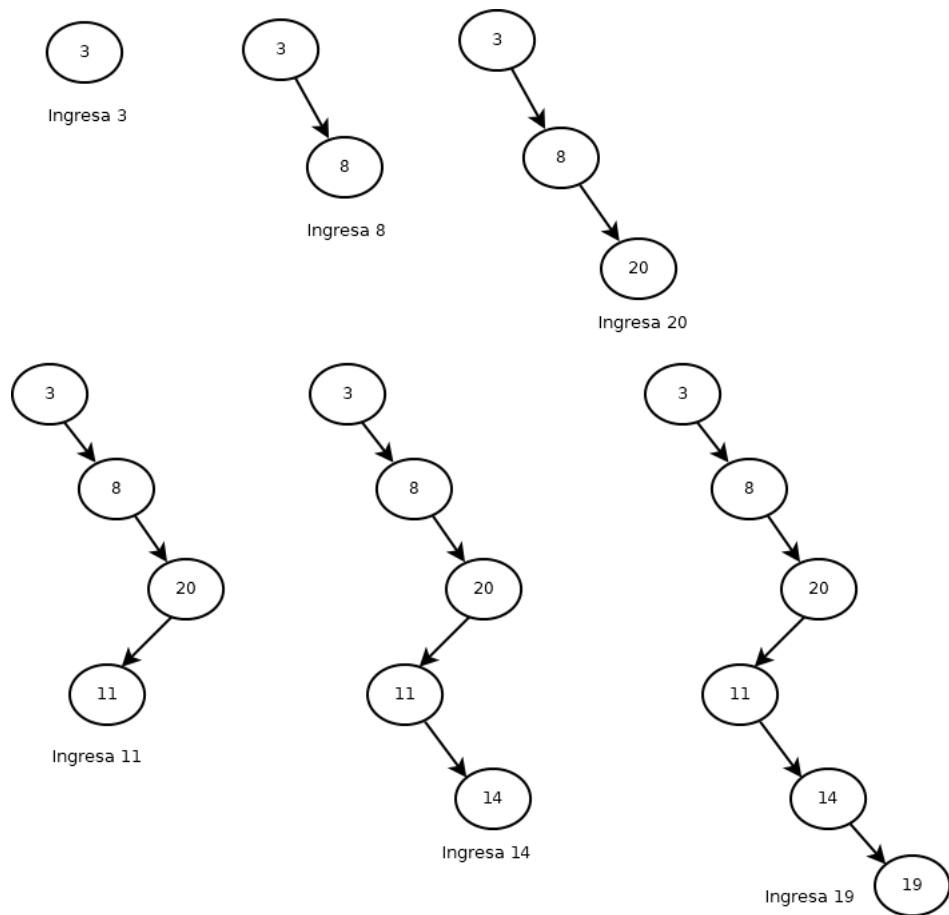


Gráfico 8.2.3: Árbol con forma de lista

Algoritmo 8.3 Inserción de un dato en un ABB de forma iterativa

```
nodo_nuevo = crear_nodo(dato)
aux = raiz
mientras aux != nulo {
    anterior = aux
    si dato < aux.obtener_dato()
        aux = aux.obtener_izquierdo()
    si no
        aux = aux.obtener_derecho()
}
si raiz == nulo
    raiz = nodo_nuevo
si no {
    si dato < anterior.obtener_dato()
        anterior.asignar_izquierdo(nodo_nuevo)
    si no
        anterior.asignar_derecho(nodo_nuevo)
}
```

Baja

La baja de un dato tiene un coste similar al de una búsqueda o inserción, pero es un tanto más compleja para pensarla e implementarla. Cuando se hace una baja puede suceder tres cosas:

- a. El nodo a dar de baja es una hoja. Dentro de este caso debemos contemplar uno más particular: el árbol tiene un solo nodo, el cual es hoja y raíz al mismo tiempo.
- b. El nodo a dar de baja tiene un solo hijo. Caso particular: ese nodo es la raíz del árbol.
- c. Caso general: el nodo a dar de baja tiene dos hijos.

El primer caso es muy simple: se libera la memoria que ocupa el nodo a borrar y se coloca el puntero que lo enlaza al árbol en nulo, como vemos en el gráfico 8.2.4. Si el árbol tuviera un solo nodo, el cual sería raíz y hoja al mismo tiempo, al eliminarlo quedará un árbol vacío. En este caso hay que tener la precaución de poner el puntero que guarda la raíz del árbol en nulo.

El segundo caso tampoco es muy complejo: si el nodo a borrar tiene un solo hijo, simplemente hay que enlazar el padre del nodo con el hijo del nodo, saltando el nodo que se busca eliminar. El efecto gráfico es que se levanta el subárbol del hijo al lugar del padre. Esto lo podemos ver en el gráfico 8.2.5. Si el nodo a dar de baja es el nodo raíz que tiene un solo hijo, nuevamente,

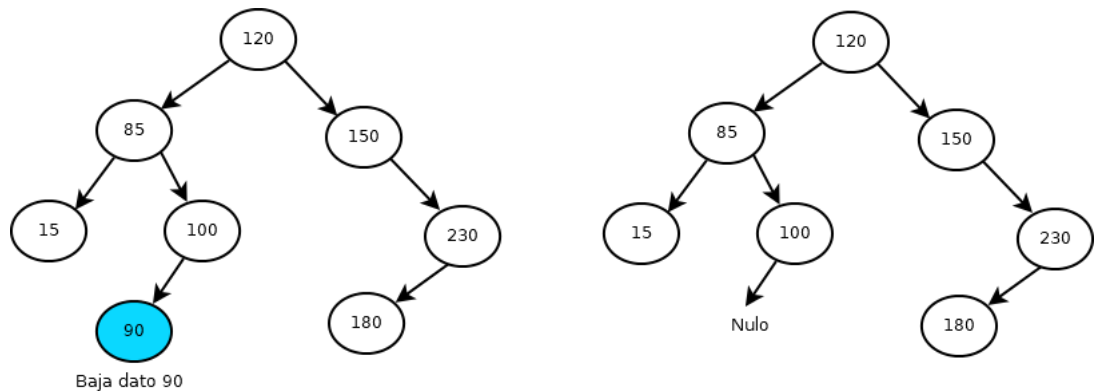


Gráfico 8.2.4: Baja de un nodo hoja

hay que tener cuidado de cambiar el puntero que guarda la dirección del nodo raíz, haciéndolo apuntar a su hijo.

El último caso es cuando debemos eliminar un nodo con dos hijos. Lo que haremos es buscar su predecesor (o su sucesor) inmediato, es decir, el número inmediato más chico que está en el árbol. También podemos hacerlo con el inmediato más grande. En el gráfico 8.2.6 se muestra un ejemplo.

¿Cómo se elige el predecesor inmediato? Se va al subárbol izquierdo del nodo que tiene el dato a eliminar y se baja por el lado derecho hasta encontrar un nulo como hijo derecho. Ese nodo es el más derecho del subárbol izquierdo. Por el contrario, si se hubiera buscado el sucesor inmediato se debería ir al hijo más izquierdo del subárbol derecho. En este caso es el nodo que tiene el dato 150, ya que no tiene hijo izquierdo.

Algoritmo de baja

```
//x: clave a eliminar
//pa: puntero al padre del nodo que contiene x
//p: puntero de entrada
si (p apunta a nodo con x) {
    si (nodo con x no tiene hijos)
        elimCaso1 (p);
    si no {
        si (nodo con x tiene un solo hijo)
            elimCaso2(p);
        si no
            elimCaso3 (p);
    }
}
si no {
    pa = encontrarPadre ();
```

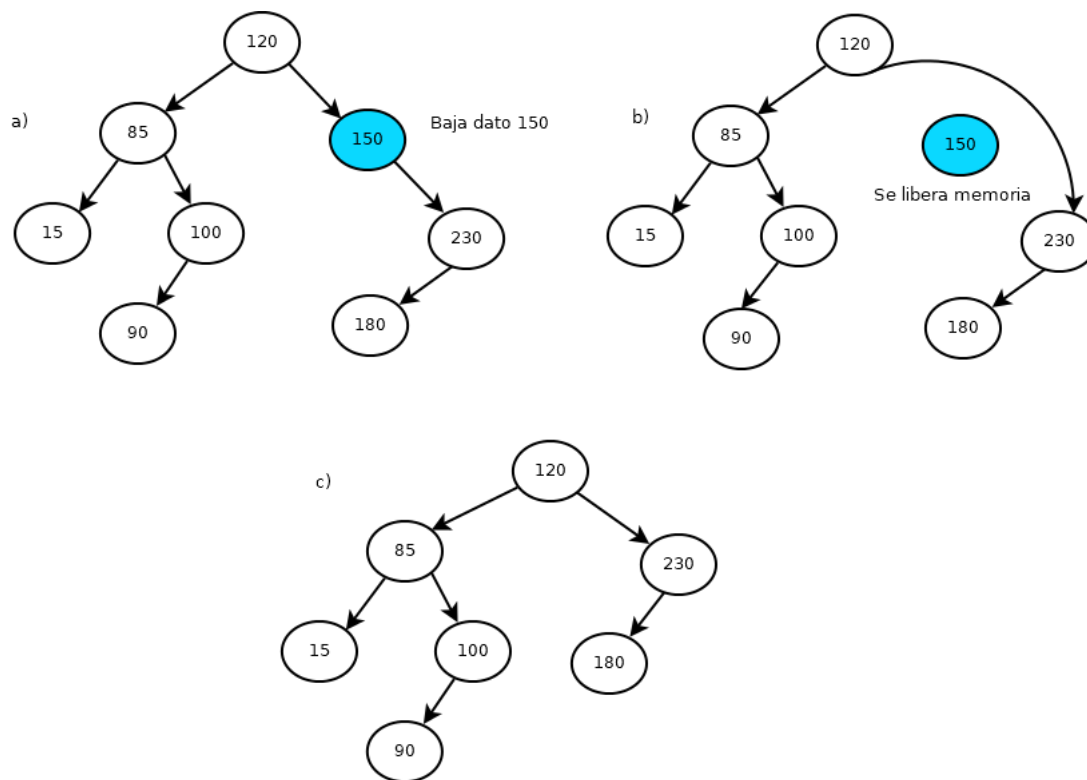


Gráfico 8.2.5: Baja de un nodo con un hijo

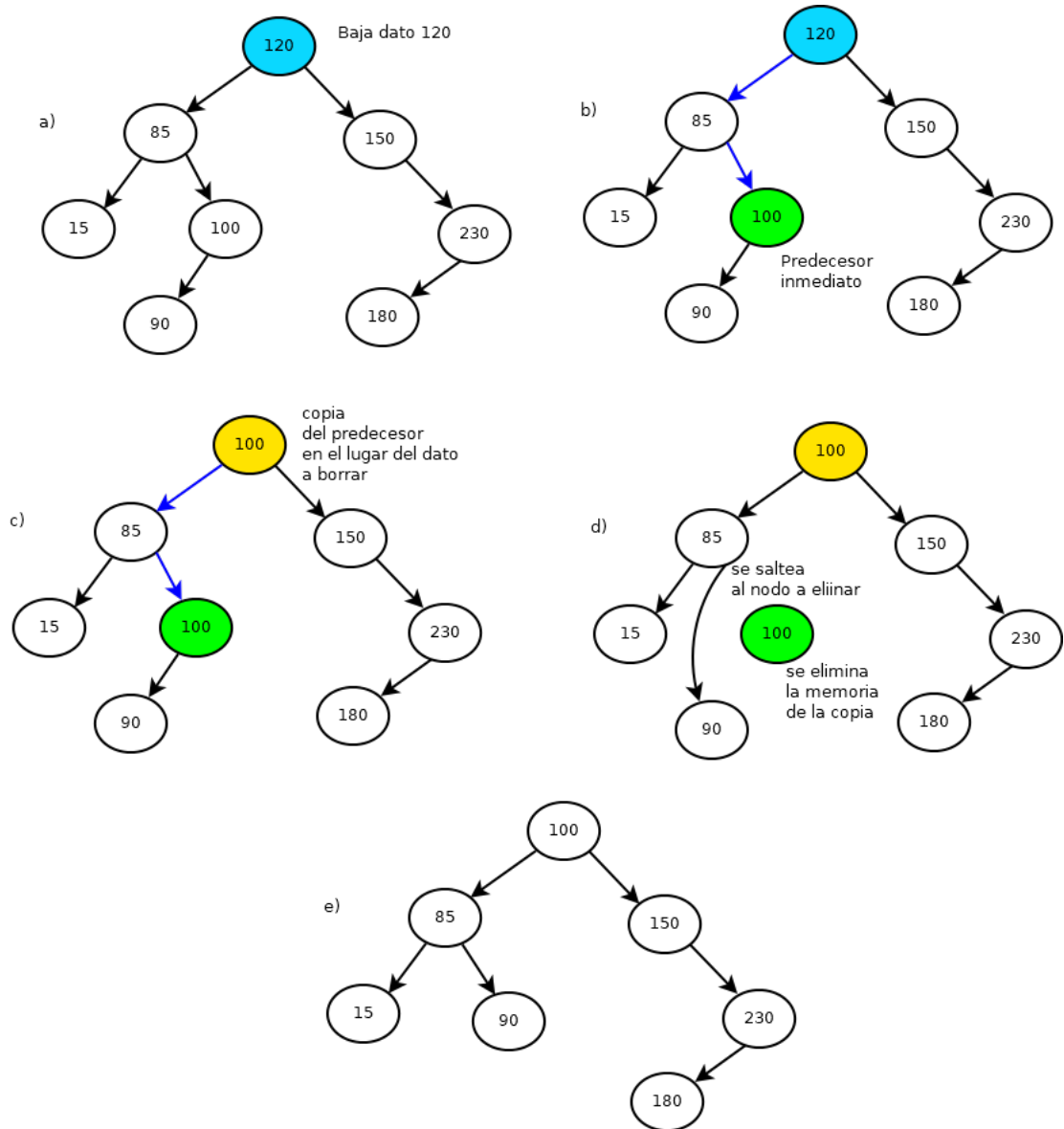


Gráfico 8.2.6: Baja de un nodo con dos hijos


```

// toma el puntero al padre del nodo que contiene x
si (pa no es nulo) { //si pa es nulo, x no está en el ABB
    si (nodo con x no tiene hijos)
        elimCaso1(pa);
    si no {
        si (nodo con x tiene un solo hijo)
            elimCaso2(pa);
        si no
            elimCaso3(pa);
    }
}

}

// Caso 1 de eliminacion
elimCaso1 (var pa: puntero a nodo) {
    //aux: puntero a nodo
    - asignar a aux la direccion del nodo que contiene x;
    - poner a nulo el puntero del nodo apuntado por pa que
      apuntaba a nodo que contiene x;
    - eliminar nodo que contiene x utilizando aux;
}

// Caso 2 de eliminacion
elimCaso2 (var pa: puntero a nodo) {
    - asignar a aux la dirección del nodo que contiene x;
    - asignar al puntero del nodo apuntado por pa que apuntaba
      a nodo que contiene x la dirección del
      hijo no nulo del nodo que contiene x;
    - eliminar nodo que contiene x utilizando aux;
}

// Caso 3 de eliminacion
elimCaso3 (var pa: puntero a nodo) {
    - aux2: puntero a nodo
    - asignar a aux la dirección del nodo que contiene x;
    - aux2 = pPadreMayorMenores(aux);
    // pPadreMayorMenores retorna el puntero al padre
    // del nodo que contiene el mayor de los menores a x
    - asignar al atributo clave del nodo que contiene x
      el mayor de los menores,
      apuntado por un hijo de aux2
    si (nodo apuntado por aux2 tiene un hijo)
        elimCaso1(aux2);
    si no
        elimCaso2(aux2);
    - eliminar nodo que contiene x utilizando aux;
}

```

Análisis del coste algoritmo de baja

- Si el nodo no tiene hijos, la búsqueda del padre del nodo a borrar se extenderá a lo sumo hasta el padre de alguna hoja, con un coste $O(n)$ si el ABB ha degenerado en lista.
- Si tiene un solo hijo, nos detendremos en el peor caso, un nivel antes del último.
- Si el nodo a borrar tiene dos hijos, el rastreo de la ubicación del mayor de los menores nos obliga a continuar el descenso (estamos en este caso situados en el padre del que hay que eliminar)), el cual, a lo sumo nos llevará desde el nodo a borrar hasta alguna de las hojas. Como se aprecia en el pseudocódigo, en cada nivel del árbol las sentencias a ejecutar tienen un coste temporal constante.

En consecuencia, en cualquiera de los casos, el coste temporal depende del número de niveles del ABB, y en el peor de los casos será $O(n)$.

8.3. Recorridos en un ABB

Los recorridos se clasifican en:

- profundidad
 - Pre orden
 - Post orden
 - In orden
- anchura

En los recorridos en profundidad se procesa cada nodo, su subárbol izquierdo y el derecho. El orden en que se realicen esos pasos diferencia distintas posibilidades del recorrido en profundidad: si el nodo se procesa antes que el subárbol se trata de preorden; si el nodo se procesa después que los subárboles, es post orden, y si se procesa primero el subárbol izquierdo, luego el nodo y luego el subárbol derecho, es inorden. Este último recorrido permite procesar las claves en orden creciente. Estos recorridos pueden implementarse recursivamente o iterativamente con ayuda de una pila.

El recorrido en anchura procesa nivel a nivel los nodos del ABB y generalmente se implementa con una cola.

Veamos los distintos recorridos tomando en cuenta el árbol del gráfico 8.3.2.

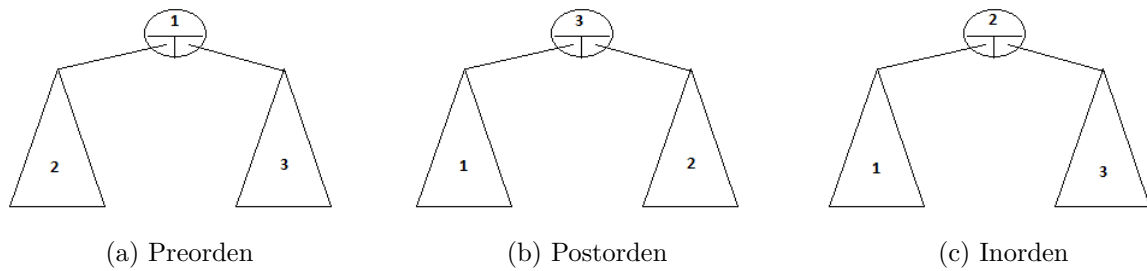


Gráfico 8.3.1: Recorridos en profundidad

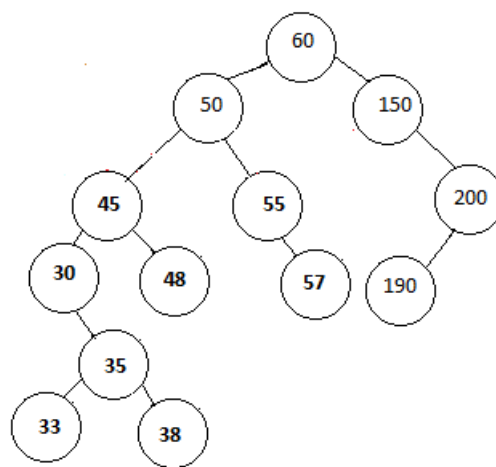


Gráfico 8.3.2: Árbol binario de búsqueda

- Recorrido Preorden: 60, 50, 45, 30, 35, 33, 38, 48, 55, 57, 150, 200, 190.
- Recorrido Postorden: 33, 38, 35, 30, 48, 45, 57, 55, 50, 190, 200, 150, 60.
- Recorrido Inorden: 30, 33, 35, 38, 45, 48, 50, 55, 57, 60, 150, 190, 200.
- Recorrido en anchura: 60, 50, 150, 45, 55, 200, 30, 48, 57, 190, 35, 33, 38.

Pseudocódigo

Si bien estos algoritmos pueden hacerse de forma iterativa, son netamente recursivos. Los distintos pseudocódigos son los siguientes.

```

// Recorrido Preorden
// p: puntero a nodo,
// inicialmente la raiz del árbol
preorden (p) {
    si (p no es nulo) {
        procesar clave nodo apuntado por p;
        preorden (puntero izquierdo de p);
        preorden (puntero derecho de p);
    }
}

// Recorrido Postorden
// p: puntero a nodo,
// inicialmente la raiz del árbol
postorden (p) {
    si (p no es nulo) {
        postorden (puntero izquierdo de p);
        postorden (puntero derecho de p);
        procesar clave nodo apuntado por p;
    }
}

// Recorrido Inorden
// p: puntero a nodo,
// inicialmente la raiz del árbol
inorden (p) {
    si (p no es nulo) {
        inorden (puntero izquierdo de p);
        procesar clave nodo apuntado por p;
        inorden (puntero derecho de p);
    }
}

```

Análisis del coste de los recorridos

Considerando las situaciones extremas de un ABB, a saber, que esté equilibrado o que haya degenerado en lista, se tiene:

- Si el árbol está equilibrado, al plasmar en una ecuación de recurrencia cualquiera de los algoritmos recursivos de recorrido en profundidad, es:

$$\begin{aligned}
 T(n) &= 2T(n/2) + 1 \\
 T(1) &= 1
 \end{aligned}$$

Lo que nos da $T(n) \in O(n)$

- Si el árbol ha degenerado en lista, en cada nivel recursivo una de las dos invocaciones corresponde a $O(0)$, ya que el subárbol izquierdo o el derecho estará nulo, entonces es:

$$\begin{aligned}T(n) &= T(n-1) \\ T(0) &= 1\end{aligned}$$

Lo que nos da $T(n) \in O(n)$

Entonces, vemos que independientemente de la configuración del ABB, el recorrido tiene un coste temporal lineal, lo cual es evidente, dado que el coste de visitar un nodo es constante, y debe visitarse cada uno de los n nodos del ABB.

Algoritmos iterativos de recorridos en profundidad en ABB

```
// Preorden
// pi: pila
// p: puntero a la raíz del árbol
// x: variable puntero a nodo
hacer {
    si (p no es nulo) {
        pi.apilar (p);
        mientras (pi no vacia) {
            x = pi.desapilar();
            procesar clave nodo apuntado por x
            si (puntero derecho de x no es nulo)
                pi. Apilar ( puntero derecho de p);
            si (puntero izquierdo de x no es nulo)
                pi.apilar ( puntero izquierdo de p)
        }
    }
} mientras ?
```

```
// Inorden
// pi: una pila
// p: puntero a la raíz del árbol
// x: variable puntero a nodo
hacer {
    x = p
    mientras (x no nulo) {
        pi.apilar (x);
        x = puntero izquierdo de x;
    }
    x = pi.desapilar();
```

```

    procesar clave nodo apuntado por x;
    x = puntero derecho de x;
} mientras (pila no vacía)

```

Análisis del coste en los algoritmos iterativos de recorrido

Se observa que para cada uno de los nodos del ABB, se apilará y desapilará posteriormente el puntero al mismo para procesarlo. El proceso termina cuando la pila se vacía. El coste temporal de apilar y desapilar considerando una implementación de la pila de las estudiadas (por ejemplo, en array o en lista ligada) es $O(1)$. Por lo cual, el coste temporal del recorrido iterativo es $O(n)$ al igual que en los recursivos.

Recorrido en anchura

Este recorrido visita los nodos nivel a nivel, de izquierda a derecha. Habitualmente se lo plantea iterativo y se implementa con una cola.

```

// Algoritmo iterativo de recorrido en anchura
// co: cola
// p: puntero a nodo, de entrada al ABB
// x: puntero a nodo
co.acolar(p)
mientras (co no vacía) {
    x = co.desacolar();
    procesar clave de nodo apuntado por p;
    si (puntero izquierdo de x no es nulo)
        co.acolar(puntero izquierdo de x);
    si (puntero derecho de x no es nulo)
        co.acolar(puntero derecho de x);
}

```

Análisis del coste recorrido en anchura

Se observa que para cada uno de los nodos del ABB, se acolará y desacolará el puntero que le apunta. El proceso termina cuando la cola queda vacía. Dado que el coste de acolar y desacolar puede ser $O(1)$ dependiendo de la implementación, lo que sucede por ejemplo para una lista ligada simple con punteros al principio y al final de la misma, el coste temporal del recorrido en anchura pertenece a $O(n)$.

8.4. Árboles balanceados

En el caso de los árboles binarios de búsqueda nos puede pasar que, si los datos que se insertan están ordenados, ya sea en forma ascendente o

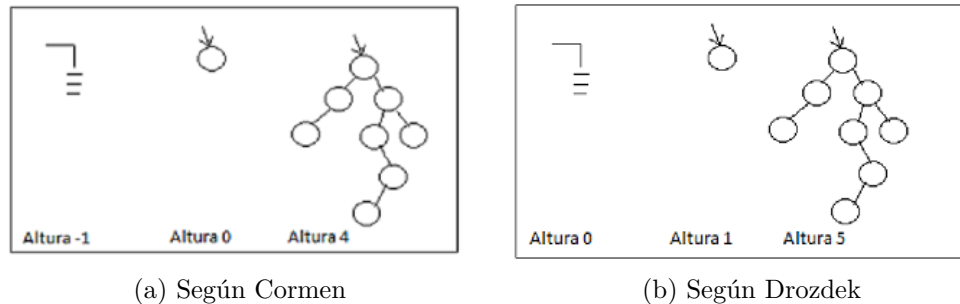


Gráfico 8.4.1: Altura de un árbol

descendente, el árbol se convertirá en una lista, ya que todos los nodos se insertarán siempre del lado izquierdo o del lado derecho. Si bien esto es un caso extremo, hay un alto riesgo de que el árbol quede desbalanceado, es decir, una subrama con notable diferencia de profundidad con respecto a la otra. Esto es problemático ya que se desperdiciará el potencial de búsqueda, convirtiendo un algoritmo de orden logarítmico en uno lineal. Por lo tanto es recomendable mantener el árbol balanceado, por supuesto, esto demanda un costo a la hora de insertar un nodo o de darlo de baja. Este costo, más que computacional es un costo de programación, dado que hay que analizar varias cuestiones. Pero antes debemos ver el tema de la altura de un árbol.

Altura de un ABB: Para este concepto se pueden encontrar definiciones ligeramente diferentes según el autor que se considere.

- Si el ABB es nulo su altura es -1. De otro modo, es la longitud del camino más largo que una la raíz con una hoja. (Cormen y otros). Lo cual nos da los valores que vemos en la parte *a*) del gráfico 8.4.1 para las alturas de los árboles allí graficados.
- Otra definición para altura de un árbol (A. Drozdek y otros) indica que si el ABB es nulo su altura es 0. De lo contrario, la altura de un ABB es el número de niveles del mismo. Entonces, para los mismos árboles del ejemplo visto tenemos las alturas que se muestran en la parte *b*).

Como vamos a utilizar el concepto de altura para analizar el balanceo de un ABB por su altura, y eso implica considerar la diferencia de alturas de subárboles hermanos, es indistinta la definición que se aplique, puesto que el valor resultante será el mismo.

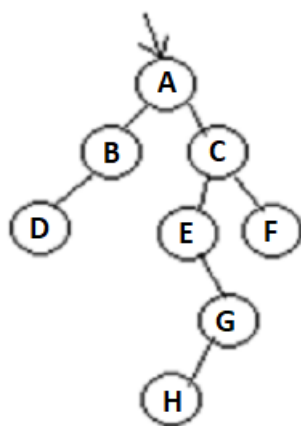


Gráfico 8.4.2: Diferencias de alturas

ABB balanceado por su altura

Un árbol binario está balanceado por su altura con diferencia permitida d sí y solo sí para todo nodo x del árbol se verifica:

$$|Altura(subárbol\ derecho\ de\ x) - Altura(subárbol\ izquierdo\ de\ x)| \leq d$$

Por ejemplo, para el árbol del gráfico 8.4.2, tenemos por cada nodo las siguientes diferencias:

- Nodo A: $|4 - 2| = 2$
- Nodo B: $|0 - 1| = 1$
- Nodo C: $|1 - 3| = 2$
- Nodo D: $|0 - 0| = 0$
- Nodo E: $|2 - 0| = 2$
- Nodo F: $|0 - 0| = 0$
- Nodo G: $|0 - 1| = 1$
- Nodo H: $|0 - 0| = 0$

Lo que significa que tomando un $d \geq 2$ el árbol está balanceado. Con $d = 1$ no.

ABB balanceado por el peso

También para este concepto encontramos distintas definiciones.

- Una de ellas enuncia que un ABB está α -balanceado sí y solo sí se verifica que para todo nodo x :
 - $(\text{Peso}(\text{subárbol izquierdo de } x) \leq \alpha \cdot \text{Peso}(\text{árbol con raíz en } x))$
y
 - $(\text{Peso}(\text{subárbol derecho de } x) \leq \alpha \cdot \text{Peso}(\text{árbol con raíz en } x))$
Siendo el peso de un árbol con raíz en r el número de nodos de dicho árbol y $\alpha \in [1/2, 1)$
- Otra definición que suele usarse para árbol balanceado por el peso enuncia: Un árbol binario está balanceado por su peso sí y solo sí para todo nodo la razón entre su propio peso y el peso de su hijo derecho (o izquierdo) está en el intervalo $[\alpha, 1 - \alpha]$ para un valor arbitrario $\alpha > 0$.
El peso de un nodo es el número de subárboles que posee.

En este curso no estudiaremos los árboles binarios balanceados por el peso, lo haremos por altura.

8.4.1. AVL

Los árboles AVL, llamados así por haber sido diseñados por Adelson-Velski y Landis en 1962, son ABB que se balancean por la altura (los otros son por peso). Se agrega un atributo al nodo que es el factor de balanceo (FB). Este factor tiene tres valores permitidos: 0, 1 o -1 , en cualquier otro caso se necesita rebalancear.

Como dijimos, el atributo agregado en los nodos es el factor de balanceo o FB el cual almacena la diferencia entre la altura del subárbol derecho y la altura del subárbol izquierdo de ese nodo. Si algún nodo del árbol binario tiene en algún momento un valor de FB mayor que 1 o menor que -1 , el árbol quedó desbalanceado y deben ejecutarse rutinas de rebalanceo.

El alta, o la baja siguen los siguientes pasos:

- a. Ejecutar algoritmo habitual de alta o baja en un ABB.
- b. Recalcular los valores de los FB desde el punto de la inserción o borrado y hacia la raíz; a lo sumo se analizarán y eventualmente modificarán tantos nodos como número de niveles tiene el árbol luego de la operación realizada.

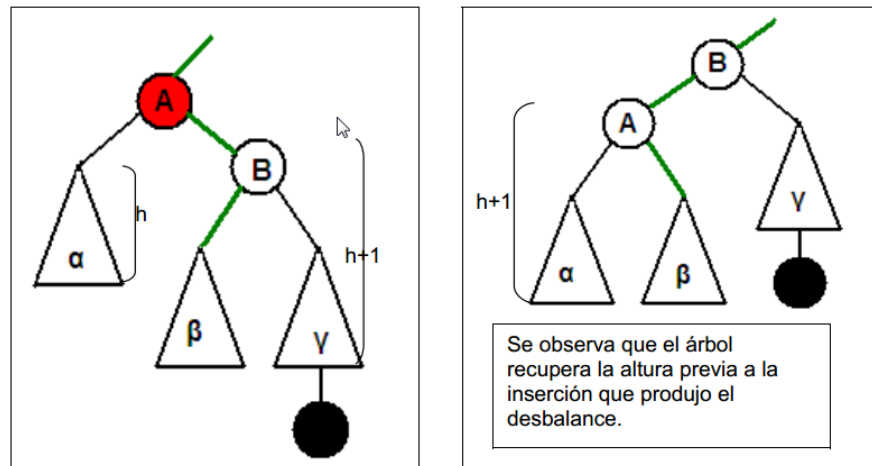


Gráfico 8.4.3: Rotación simple antihoraria

- c. Si hay desbalanceo, establecer cuál es el nodo pivote, es decir, el nodo más próximo al punto de inserción o de borrado de la clave que provocó el desbalanceo, ya que las rutinas de rebalanceo indican las reasignaciones a realizar en función de la variación de FB del pivote y alguno de sus hijos. Realizar entonces las rotaciones adecuadas. Las rotaciones son reasignaciones en función de la configuración del árbol desbalanceado.

Rotaciones

Rotaciones simples (involucran 3 punteros) Hay dos casos: antihoraria y horarias, imágenes especulares entre sí.

Rotaciones dobles (involucran 5 punteros). Hay también dos casos: antihoraria-horaria y horaria-antihoraria, cada uno de ellos se subdivide en forma trivial y forma no trivial. También entre si los triviales y los no triviales son imágenes especulares.

El primer caso, la rotación simple antihoraria la vemos en el gráfico 8.4.3. La podemos imaginar como “tirar de una cuerda hacia el lado izquierdo” para que el árbol quede balanceado. Por ejemplo tenemos un árbol balanceado como vemos en el gráfico 8.4.4, en donde en cada nodo se indica su FE. Insertamos un nodo con el valor 50, por lo tanto, en primer lugar el árbol queda como vemos en la parte a) del gráfico 8.4.5. Observamos que el nodo raíz quedó con un $FE = 2$, por lo tanto quedó desbalanceado. La corrección se realiza con una rotación simple antihoraria, quedando como en la parte

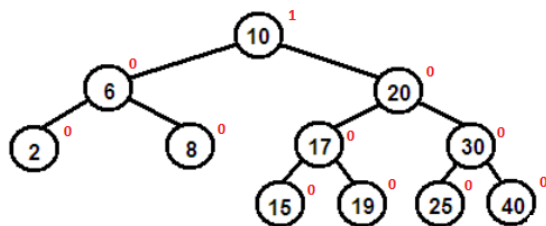
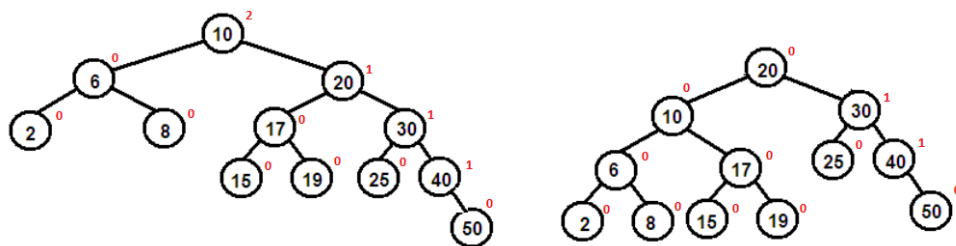


Gráfico 8.4.4: Árbol balanceado con sus FE en cada nodo



(a) AVL desbalanceado por inserción

(b) AVL rebalanceado

Gráfico 8.4.5: AVL con rebalanceo simple

b).

Rotación simple horaria Es similar a la antihoraria pero rota para el lado derecho (ver gráfico 8.4.6).

Rotaciones dobles

Caso 1 a: horaria - antihoraria trivial Este caso lo vemos en el gráfico 8.4.7.

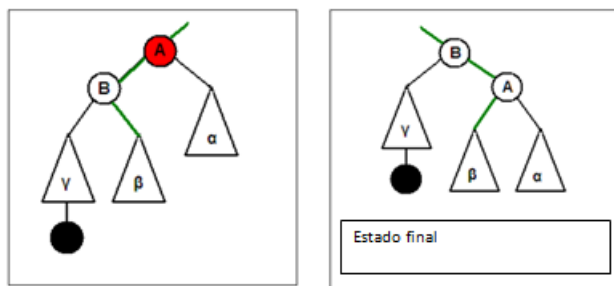


Gráfico 8.4.6: Rotación simple horaria

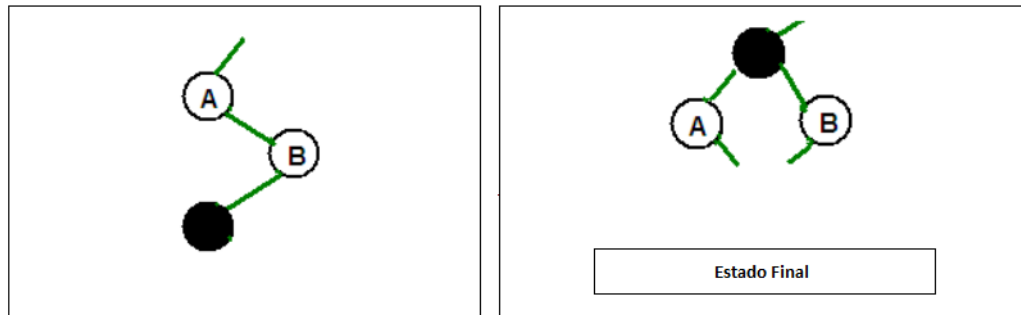


Gráfico 8.4.7: Rotación doble: horaria - antihoraria - trivial

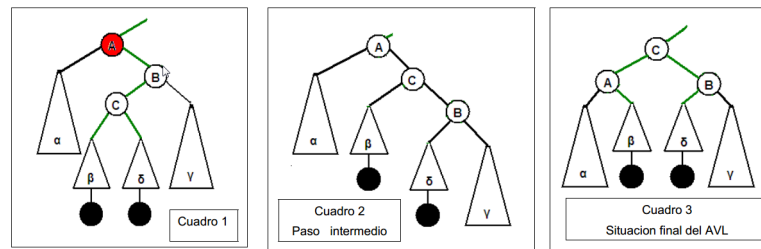


Gráfico 8.4.8: Rotación doble: horaria - antihoraria - no trivial

Caso 1 b: horaria - antihoraria - no trivial Este caso lo vemos en el gráfico 8.4.8.

Nota: en el gráfico, el nodo nuevo se inserta en una de las posiciones indicadas en negro (no en ambas, obviamente). Es decir que cualquiera de los nodos negros puede ser el nuevo.

Se deja como ejercicio plantear el otro caso de rotación (antihoraria-horaria) doble en sus formas trivial y no trivial y analizar qué sucede con el balance del pivote y del resto de los nodos involucrados. Estas rotaciones (la trivial y la no trivial) son imágenes especulares de las homólogas del caso 1.

Ejemplo 8.1. En el árbol que vemos en el gráfico 8.4.9 se da de alta el nodo con el valor 170, lo cual provoca las modificaciones en los FB indicadas en el gráfico 8.4.10 del lado izquierdo, y tras la rotación, la resolución del lado derecho. Observar que el AVL recupera la altura previa a la inserción que produjo el desbalanceo.

Borrado de un AVL

Cuando se lleva a cabo un borrado, el desequilibrio se produce cuando la eliminación de un nodo trae como consecuencia el no cumplimiento de la

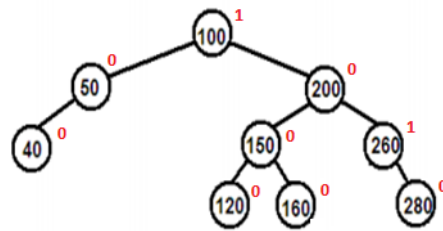


Gráfico 8.4.9: AVL ejemplo

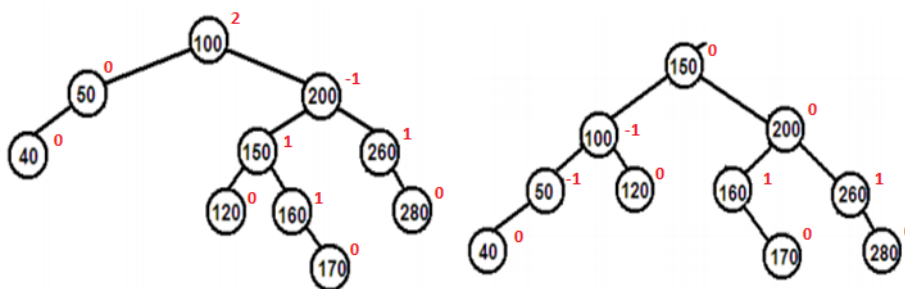


Gráfico 8.4.10: AVL con inserción y rebalanceo

propiedad del AVL.

En principio, el borrado en un AVL se lleva a cabo como en cualquier ABB; y luego se considera lo siguiente:

- Si la supresión es en el subárbol izquierdo del pivote, entonces debe analizarse la situación del subárbol derecho, ya que los algoritmos de rotación que deban realizarse dependen de la relación entre las alturas de los subárboles del subárbol derecho (las cuales no pueden ser 0).
- Análogo razonamiento se hace para un borrado en el subárbol derecho del pivote pero considerando el subárbol izquierdo.

Ejemplo 8.2. En el árbol del gráfico 8.4.11 se realiza el borrado del nodo con el valor 65, lo cual provoca las modificaciones en los FB indicadas en el gráfico 8.4.12 a la izquierda, y tras la rotación, la resolución del lado derecho.

Los estudios empíricos indican que, en promedio se realiza una rotación cada 2 altas y una rotación cada 5 bajas, y que ambas rotaciones (simples y dobles) son equiprobables.

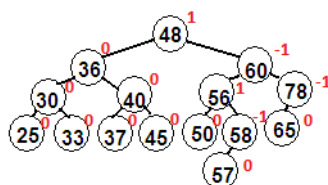


Gráfico 8.4.11: AVL en estado de equilibrio

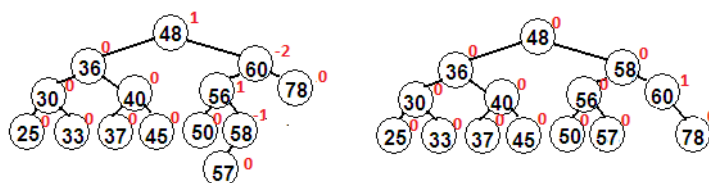


Gráfico 8.4.12: AVL rebalanceo en el borrado

Eficiencia de un AVL

La intención de las estructuras de árbol balanceadas por la altura, como los árboles AVL, es disminuir la complejidad temporal de las operaciones de alta, baja y búsqueda en un ABB.

- El algoritmo de búsqueda es idéntico al de un ABB común, por lo cual el coste temporal dependerá del número de niveles del árbol .
- El proceso de recalculer los FB luego de realizada un alta o una baja involucran a los nodos que están en el camino desde el punto en el que se hizo la alteración, y hasta la raíz, es decir tantos nodos como niveles tiene el árbol. Es frecuente que para mejorar los tiempos de estos cálculos, cada nodo tenga un enlace con su nodo padre, y almacenada la altura del subárbol del cual él mismo es raíz.
 - En el caso del alta, dado que los punteros que pueden ser reasignados para desbalancear son a lo sumo 5, el coste temporal del algoritmo depende de la altura del AVL. Se puede indicar que es $O(m)$, siendo m el número de niveles del AVL.
 - En el caso de la baja, puede llegar a ser necesario analizar subárboles englobados, pero el número de estos análisis no hace crecer el coste temporal del algoritmo más allá de $O(m)$, siendo m el número de niveles del AVL.

La complejidad de esas tres operaciones depende, pues, de la altura del árbol.







1	2	3	4	5	6
1	2	4	7	12	20
					

Gráfico 8.4.13: Cantidad mínima de nodos por nivel

Relación entre la altura de un AVL y la menor altura de un ABB

Consideremos árboles AVL lo más ralos que sea posible, es decir, que, para una altura determinada tengan la menor cantidad de nodos que sea posible. Esta es la peor situación para un árbol, dado que se “desaprovecha” la capacidad de cada nivel del mismo para contener nodos; se ha visto que de cada nivel de un ABB se analiza, en los procesos de alta, baja o búsqueda, un nodo por nivel, por tanto, cuántos más nodos haya por nivel, más serán “descartados” en el rastreo, y el proceso será tanto más eficiente. Veremos pues, cual es la peor situación que puede darse para un AVL para sacar algunas conclusiones.

En el gráfico 8.4.13 se muestra qué ocurre para cada nivel (los dibujos son orientativos, la forma del árbol AVL puede variar, pero el número de nodos, no). Se observa una relación entre la cantidad mínima de nodos para un árbol AVL de determinada altura y las cantidades correspondientes a los precedentes en altura. Llamando $F(h)$ a la menor cantidad de nodos para un árbol AVL de altura h , se puede indicar:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(h) = F(h - 1) + F(h - 2) + 1$$

Esta relación tiene similitud con la sucesión de Fibonacci, por lo cual a estos árboles se los llama árboles de Fibonacci. Un árbol de Fibonacci es un árbol AVL con la cantidad de nodos mínima. Estos árboles representan la mayor “deformación posible” de un AVL, en el sentido en el que “desperdician” la mayor cantidad de espacio disponible en los niveles. Se demuestra que, si $F(h) = n$, entonces $h \simeq 1,44 \cdot \log(n)$. Entonces, un árbol AVL tiene una altura que, a lo sumo (si es un árbol de Fibonacci) es de $1,44 \cdot \log(n)$.

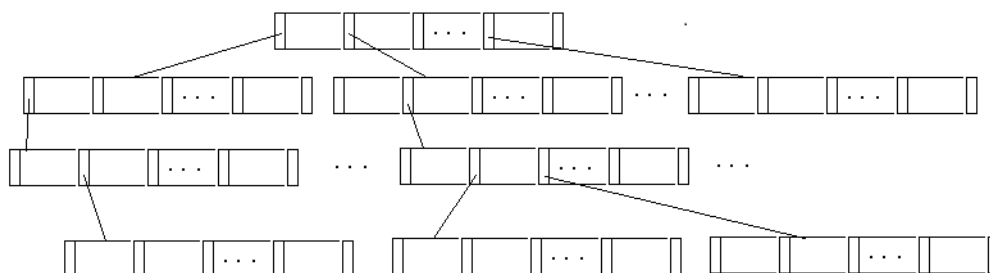


Gráfico 8.5.1: Árbol multivías

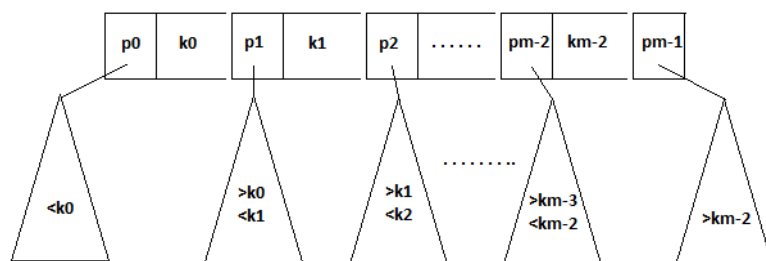


Gráfico 8.5.2: Nodo árbol multivías

Esto indica que el coste del alta, la baja y la búsqueda en un AVL pertenecen a $O(\log(n))$.

8.5. Árboles multivías

Los árboles multivías son árboles cuyos nodos pueden tener varias claves y enlaces. Para ser más precisos, si tienen m enlaces, tendrán $m - 1$ claves. Dentro de cada nodo, las claves se colocan de forma creciente sin dejar espacios entre dos claves. Los subárboles de la clave k_i contendrán claves que serán menores a la clave k_i . Por lo tanto, el último enlace tendrá claves mayores a la última clave de ese nodo.

Un árbol de m vías está formado por nodos que pueden contener hasta $m - 1$ claves ($k_0, k_1, k_2, \dots, k_{m-2}$). Las claves están ordenadas de forma creciente. No hay “posiciones vacías” entre dos claves, es decir, si hay una clave en la posición i y hay otra clave en la posición $i + 2$, entonces hay clave en la posición $i + 1$. Si hay clave en la posición 1, entonces hay clave en posición 0 (la primera según nuestra convención).

Los subárboles hijos verifican que las claves de los mismos Las claves del primer subárbol son menores que K_0 , las claves del segundo subárbol

son mayores que K_0 y menores que K_1 , etc. Las claves del último subárbol son mayores que la última clave, k_{m-2} . Los árboles de m vías pueden crecer desbalanceados, lo cual resta mucha eficiencia. Para garantizar el balanceo y mejorar su funcionamiento se desarrollaron a partir de ellos los árboles B.

8.5.1. Árboles B

Son estructuras diseñadas por R. Bayer y E. McCreight en 1972.

Definición

Un árbol B es un árbol de m vías que verifica

- Todos los nodos excepto la raíz están completos con claves al menos hasta la mitad.
- La raíz,
 - o bien es hoja,
 - o bien tiene al menos dos hijos.
- Si un nodo no hoja tiene h claves almacenadas, entonces tiene $h + 1$ hijos.
- Todas las hojas están en el mismo nivel.

Por lo anterior, es un árbol de m vías balanceado.

Alta en un árbol B

El esquema del algoritmo se muestra a continuación

```
alta(clave x)
pos <- Ubicar_posición_hoja_ adecuada
//si el árbol es nulo, pos será nula;
// de otro modo, se "desciende en el árbol
// hasta la hoja correcta"
clave_ubicada = falso
mientras (
  clave_ubicada == falso) {
  si (Pos es nula) {
    generar nodo
    almacenar clave x
    clave_ubicada <- verdadero
    asignar los valores a los enlaces de este nodo
  }
```

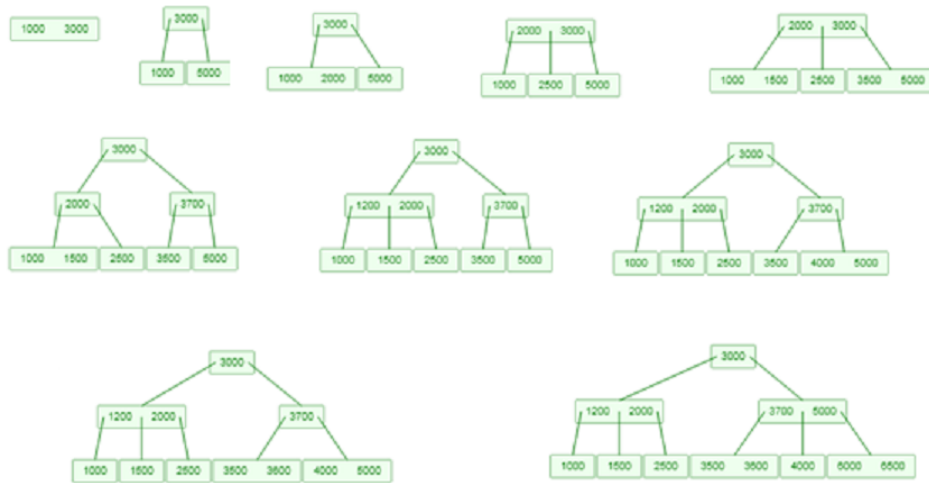


Gráfico 8.5.3: Evolución árbol B

```

si no {
  si (hay lugar en nodo señalado por pos) {
    ubicar clave x en nodo de forma ordenada
    asignar los valores a los enlaces de este nodo
    clave_ubicada = verdadero
  }
  si no {
    val = determinar valor central de secuencia asociada
    al nodo apuntado desde pos
    generar nodo nuevo
    claves menores que el valor central se acomodan
    en el nodo izquierdo
    claves mayores que el valor central se acomodan
    en el nodo derecho
    // Se continúa con el nodo padre de los dos involucrados;
    // se intenta ubicar el valor central
    pos = ubicar_posición_del_nodo_padre
  }
}

```

Ejemplo: si se da de alta la siguiente secuencia en un árbol B de 3 vías (también llamado árbol 2-3) inicialmente vacío, este evolucionará como se muestra en el gráfico 8.5.3. Secuencia de claves: 3000, 1000, 5000, 2000, 2500, 1500, 3500, 1200, 4000, 3600, 6000, 6500.

Baja en árbol B

El esquema del algoritmo se indica a continuación:

```
eliminar (clave x)
pos <- ubicar_nodo_clave_x
si (nodo señalado por pos es hoja) {
  eliminar clave sin dejar "huecos" entre las mismas
  finalizar <- falso
  mientras (Finalizar == falso) {
    si (quedaron menos claves que la cantidad
        mínima en nodo señalado por Pos) {
      si existe hermano izquierdo o derecho de
      ese nodo con más claves que la cantidad mínima {
        realizar una rotación adecuada para balancear
        finalizar <- verdadero
      }
      si no {
        //hay menos claves que el mínimo y no se puede "tomar"
        de un hermano para balancear
        fusionar nodos hermanos, bajando la clave
        del nodo padre que separaba los nodos involucrados
        pos <- ubicar_nodo_padre_de los fusionados
      }
    }
    si no { //el nodo tiene suficientes claves
      finalizar <- verdadero
    }
  } // fin mientras
si no { //nodo no es hoja
  reemplazar clave x por menor de los mayores o mayor de los menores
  // ese reemplazante está en un nodo hoja
  pos <- ubicar_nodo_hoja_del_reemplazante
  eliminar (reemplazante de x)
}
```

Ejemplo: Si en el árbol B del ejemplo anterior se dan de baja sucesivamente las claves que se indica, la evolución de la estructura será como se muestra en los gráficos 8.5.4. Secuencia de claves a eliminar: 5000, 3700, 3500, 6500, 6000, 1000.

Mínima cantidad de claves de un árbol B con h niveles

Tomando $q = \text{techo}(m/2)$, consideramos el peor árbol B de m vías, en el sentido de tener la menor cantidad de claves posibles por nivel, y es:

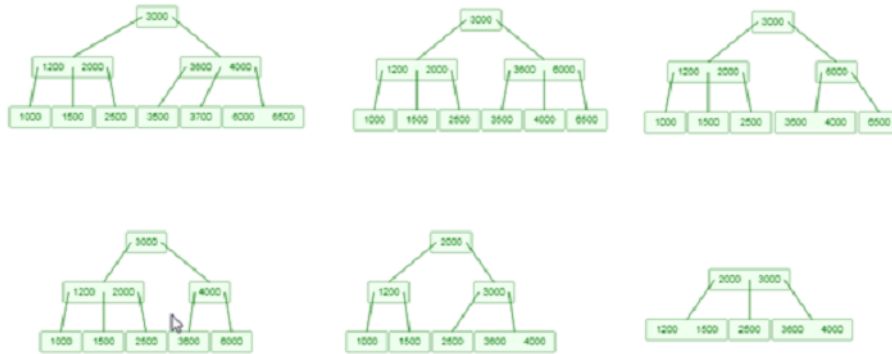


Gráfico 8.5.4: Baja en un árbol B

Nivel	Cantidad mínima de claves en el nivel
1 (raíz)	1
2	$2(q - 1)$
3	$2q(q - 1)$
4	$2q^2(q - 1)$
...	...
t	$2q^{t-2}(q - 1)$

Entonces la mínima cantidad de claves en los t niveles es $1 + (q - 1) \sum_{i=0}^{t-2} 2q^i$ que operando nos queda $-1 + 2q^{t-1}$ como cantidad mínima de claves en los t niveles. Si hay n claves almacenadas en total, entonces $n \geq -1 + 2q^{t-1}$, de donde se deduce que $h \leq \log_q((n + 1)/2) + 1$

8.6. Array de bits

En esta implementación se utilizan los bits de un array para indicar si un elemento dado por el subíndice está o no en el conjunto. Por ejemplo: consideremos un solo byte, es decir, una secuencia de 8 bits en el cual en b se implementó un conjunto A . Los elementos que pueden almacenarse en A serán por tanto los enteros de 0 a 7. Entonces, la situación del byte que vemos en el gráfico 8.6.1 nos indica que en el conjunto A están los elementos 0, 2, 3 y 5.

La convención es que un 1 en la posición i indica que i está en el conjunto y un 0 indica que no está.

Para implementar las operaciones de alta, baja y búsqueda de un elemento hay que tener en cuenta que un byte se manipula completo, es una unidad en

0	0	1	0	1	1	0	1
7	6	5	4	3	2	1	0

Gráfico 8.6.1: Array de bits

0	0	0	1	0	0	0	0
7	6	5	4	3	2	1	0

Gráfico 8.6.2: Máscara para manipular bits

si. Es imposible manipular cada bit por separado. Eso nos obliga a utilizar “máscaras” implementadas también a través de bytes para modificar el valor de un bit en particular, o bien para testear su contenido y establecer si está en 0 o en 1. Para, por ejemplo, dar de alta al elemento 4 en el conjunto tenemos que poner a 1 el valor de esa posición dejando inalterados los otros. Eso se puede lograr usando la máscara que se muestra en el gráfico 8.6.2. Con esa máscara, si se realiza la operación *or* entre ella y el byte b , modifica la situación del bit de posición 4, pasándolo a 1. Observar que la máscara tiene 0 en todas las posiciones excepto en aquella que queremos “encender” a través de un *or*.

Ahora consideremos que queremos eliminar el elemento 2 del conjunto. Se puede usar para ello la máscara que se ve en el gráfico 8.6.3 y hacer un *and* con ella y el byte b . La máscara tiene 1 en todas las posiciones, excepto en aquella cuyo valor queremos “apagar” a través del *and*.

Para determinar si un elemento, por ejemplo el 5, está en el conjunto A , se puede operar con una máscara completa de 0 salvo en la posición 5 como vemos en el gráfico 8.6.4 y luego hacer un *and* con el byte que representa al conjunto. Luego de esta operación, si el resultado es 0 significa que el 5 no está en el conjunto, cualquier otro valor distinto de 0 significa que sí está.

Ahora bien: si pensamos en una máscara que se prepara de diferente for-

1	1	1	1	1	0	1	1
7	6	5	4	3	2	1	0

Gráfico 8.6.3: Máscara para eliminar un dato

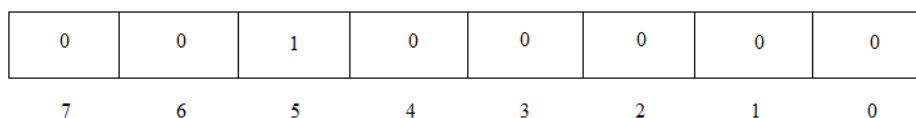


Gráfico 8.6.4: Máscara para consultar por un dato

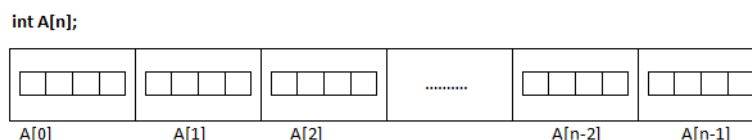


Gráfico 8.6.5: Array de enteros

ma según la posición del elemento a manipular, el algoritmo puede ser muy ineficiente. En lugar de ello se puede plantear siempre que, para manipular el elemento x , utilizamos un byte con 00000001 (siete 0 y un 1 al final). Desplazamos los bits del byte x posiciones a izquierda (con lo cual el 1 quedará ubicado en la posición x). Luego, si queremos dar de alta a x , la máscara ya está lista para operar con el *or*. Si, en cambio, queremos dar de baja un elemento, hay que invertir los unos y los ceros de la máscara, lo cual se puede efectuar negando la misma y operar con un *and*.

Vamos a generalizar la situación ampliando las posibilidades de almacenamiento: en lugar de trabajar con un byte, trabajaremos con un array de enteros. Cada posición del array tiene un tamaño s , correspondiente al tamaño (sizeof) de un entero, con lo cual la capacidad se amplía considerablemente.

Lo que antes planteamos para un byte aislado se llevará a cabo en alguno de los bytes que componen alguno de los enteros del array. Considerando n enteros en el array, para trabajar almacenando, borrando o determinando la pertenencia al conjunto del elemento k , lo importante es establecer en qué entero del array y en que bit de ese entero se ubica k . Puede hacerse lo siguiente:

```
i = k / s; // s es sizeof (int)
pos = k % s; // % corresponde al operador módulo
```

y, para preparar la máscara para operar con k (tener en cuenta que estamos usando los operadores de C/C++, en otros lenguajes el operador puede ser distinto)

```
unsigned int f = 1; // f = 00000...00001
f = f << pos; // desplazar el 1 k posiciones a izq
```

Implementación de la primitiva alta en el conjunto: `A[i] = A[i] | f; // Borrado de k del array de bits`

Implementación de la primitiva baja en el conjunto: `f = ~f ; // Modificar la máscara, invirtiendo los bits`
`A[i] = A[i] & f;`

Determinación de la pertenencia o no de k al array de bits: (implementación de la primitiva esta o pertenece en el conjunto):
`if (A[i] & f) { // k pertenece al conjunto}`
`else {k no pertenece al conjunto}`

Coste

Estas operaciones en array de bits son de tiempo constante. El alta, la baja y la búsqueda tienen un coste temporal que pertenece a $O(1)$. Sin embargo hay que tener en cuenta que solo permiten trabajar con claves numéricas enteras. Si las claves fueran de otra naturaleza, habría que considerar el costo de mapear las mismas sobre el conjunto de enteros correspondiente.

8.7. Trie

Estructura que permite el almacenamiento de cadenas de símbolos de un alfabeto finito compuesto por s símbolos. Los caracteres que forman cada clave no se encuentran almacenados de forma contigua. (de la Briandais, 1959).

El concepto de *trie* puede plasmarse de distintas formas con diversas implementaciones. Aquí mostraremos algunas de ellas.

Implementación de un trie con un árbol

En una implementación como árbol, un trie de orden m o bien es nulo, o bien es una secuencia ordenada de exactamente m tries de orden m . Cada nodo es un array de punteros. Si el conjunto de caracteres o símbolos es $\{a, b, c, \lambda\}$, en el trie que vemos en el gráfico 8.7.1 se han almacenado las claves: a, ab, bba, ca, bbc.

Esquema del alta de una clave x .

Toda clave es una sucesión de caracteres o símbolos

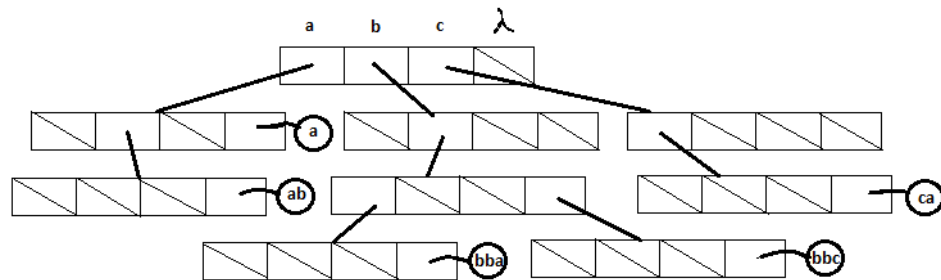


Gráfico 8.7.1: Trie

```
// Alta de un elemento
//aux: puntero a nodo
// p es puntero a la raíz del árbol
// pos: posición del carácter a considerar
aux = p;
pos=0; //consideramos el primer símbolo de la clave
mientras (secuencia de caracteres de la clave no haya terminado) {
    si ( aux[pos] no nulo)
        aux = aux[pos];
    si no {
        generar nodo nuevo desde aux[pos];
        poner todos los punteros del nodo nuevo en nulo;
        aux = aux[pos];
    }
    pos = pos + 1; // Avanzar al siguiente carácter de la clave;
}
almacenar en posición correspondiente al terminador
la dirección del valor asociado con la clave
```

La baja se lleva a cabo mediante un movimiento que parte del enlace en la hoja y retrocede poniendo a nulo los punteros correspondientes; si el array de punteros quedara nulo, se libera la memoria que ocupaba y se pone a nulo el puntero correspondiente del nodo padre (se deja como propuesta desarrollar el algoritmo).

Implementación de un trie con listas de listas

Como alternativa para reducir el costo espacial de la implementación anterior, se puede considerar una implementación con listas. Solo se almacenan los nodos de símbolos que correspondan a caracteres de claves almacenadas. En el gráfico 8.7.2 se ha implementado el mismo conjunto del ejemplo anterior.

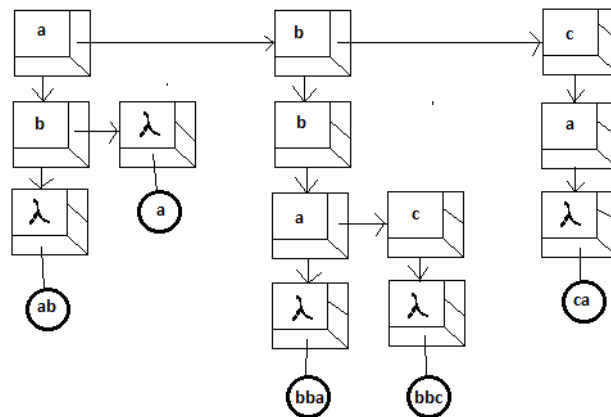


Gráfico 8.7.2: Trie implementado con listas

8.8. Cola con prioridad

Es un TDA contenedor de elementos del mismo tipo. Cada elemento es un par ordenado (clave, prioridad).

Las claves pueden repetirse y también las prioridades. Los elementos se almacenan de forma secuencial, como en una cola común, pero cada nuevo elemento al ser acolado se ubica delante de aquellos de prioridad menor, si es que hay elementos con estas características. Si no, se ubicará al final. La operación desacolar elimina el primer elemento de la cola, al igual que en las colas habituales.

Operaciones

- Creación de la Cola con Prioridad:
Precondiciones: ---
Postcondiciones: Cola creada en condiciones de ser usada
- Destrucción de la Cola con Prioridad:
Precondiciones: la cola debe existir
Postcondiciones: ----
- Alta de un elemento
Precondiciones: la cola debe existir
Postcondiciones: Cola modificada, ahora con un elemento más
- Extracción del mínimo elemento (el de mayor prioridad)
Precondiciones: la cola debe existir y no debe estar vacía
Postcondiciones: Cola modificada, ahora con un elemento menos.

Implementaciones posibles:

Una cola con prioridades puede implementarse de múltiples formas: con arrays, con listas ligadas y también con otras estructuras (se deja como ejercicio el planteo de los algoritmos correspondientes a las primitivas indicadas y el análisis de los costes). En particular nos interesa la utilización de los árboles *heap* o montículos para implementar el TDA cola con prioridad, por lo que desarrollaremos ese concepto. (No confundir "árbol heap" con la zona de memoria dinámica llamada heap).

8.9. Heap

Un árbol *heap* es un caso particular de un árbol binario, con las siguientes tres propiedades:

- El valor que se guarda en cada nodo es mayor que el de sus hijos.
- El árbol está balanceado
- y completo, si no lo está, los hijos están del lado izquierdo.

Nota: este es un árbol de máxima, si fuera de mínima cambia la primera propiedad en donde dice "mayor" por "menor".

- Completo significa que tiene 'aspecto triangular', es decir que contiene un nodo en la raíz, dos en el nivel siguiente, y así sucesivamente teniendo todos los niveles ocupados totalmente con una cantidad de nodos que debe ser 2^n , donde n es el número de nivel.
- Casi completo significa que tiene todos los niveles "saturados" menos el nivel de las hojas. Es decir, los niveles están ocupados con 1,2,4,8,16... nodos; en el nivel de las hojas puede no cumplirse esta condición, pero los nodos existentes deben estar ubicados 'a izquierda'.

En el gráfico 8.9.1 vemos dos árboles binarios: el de la izquierda se encuentra completo, el de la derecha es casi completo. Nótese que el único nivel que no está completo es el de sus hojas pero los nodos "faltantes" son los del lado derecho.

En cuanto a la primera propiedad se puede decir que es "parcialmente ordenado", lo que significa que para todo nodo se cumple que los valores de sus nodos hijos (si los tuviera) son ambos menores o iguales que el del nodo padre (árbol heap de máximo), o bien que para todo nodo se cumple que los valores de sus hijos son ambos mayores o iguales que los de su nodo padre (y

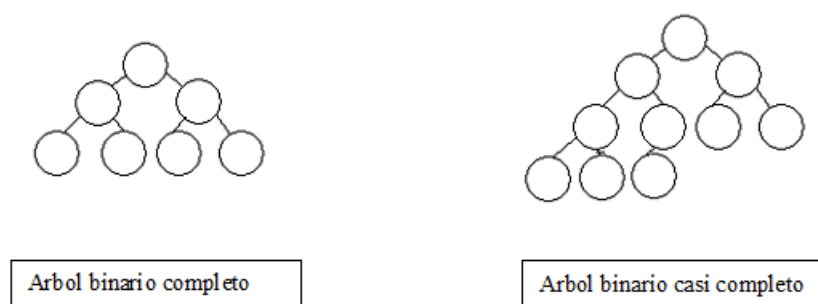


Gráfico 8.9.1: Árboles binarios, completo y casi completo

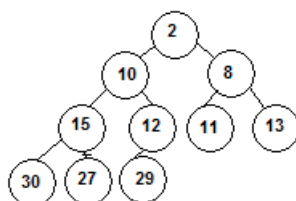


Gráfico 8.9.2: Heap de mínimo

se llama árbol heap de mínimo). Entre los hijos no se exige ningún orden en particular.

En el gráfico 8.9.2 vemos un árbol heap de mínimo.

Es muy frecuente que un árbol heap se implemente con un array, almacenando los valores por nivel, del siguiente modo:

2	10	8	15	12	11	13	30	27	29
---	----	---	----	----	----	----	----	----	----

Considerando que las posiciones del array son $0, 1, \dots, n-1$, se verifica que las posiciones de los nodos hijos con respecto a su nodo padre cumplen:

Padre en posición $k \Rightarrow$ Hijos en posiciones $2 * k + 1$ y $2 * k + 2$

Análogamente, si el hijo está en posición h , su padre estará en posición $(h - 1)/2$, considerando el cociente entero.

Un árbol heap o montículo tiene asociados algoritmos básicos para:

Extraer raíz:

Esta operación se lleva a cabo de la siguiente forma:

Se remueve la raíz. Se reemplaza el nodo removido por la última hoja.

Se restaura el heap o montículo, es decir se compara el valor de la nueva raíz con el de su hijo menor (si es un montículo ordenado de esta forma) y se realiza el eventual intercambio, y se prosigue comparando hacia abajo el



Gráfico 8.9.3: Borrado de raíz en un heap

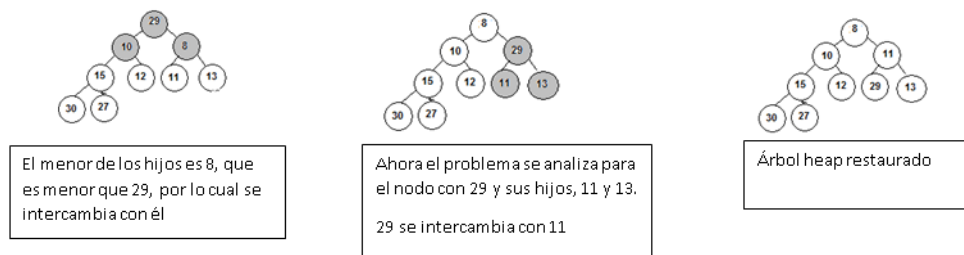


Gráfico 8.9.4: Restauración de un heap

valor trasladado desde la raíz hasta llegar al nivel de las hojas o hasta ubicar el dato en su posición definitiva.

Por ejemplo, en el heap mostrado en el gráfico 8.9.2 se extrae la raíz.

En el gráfico 8.9.4 vemos los movimientos que se realiza para su restauración.

Coste de esta operación: Consideremos una implementación en array. El reemplazo de la clave de la raíz por la última hoja lleva un tiempo constante. Luego, para restaurar el montículo y ubicar el valor que se ha colocado en la raíz en su posición definitiva se lleva a cabo, a lo sumo para cada par de niveles sucesivos, un par de comparaciones y un intercambio hasta llegar al nivel de las hojas. Como el árbol está completo o casi completo, su altura es aproximadamente $\log_2 N$, siendo N el número de nodos del heap. Las comparaciones y eventual intercambio tienen coste constante. Así que, en el peor caso, para restaurar el montículo, el número de ‘pasos’ (entendiendo por paso el par de comparaciones más el eventual intercambio) es aproximadamente $\log_2 N$.

Considerando una implementación en array entonces, se realiza lo siguiente.

Situación inicial: $N = 10$

2	10	8	15	12	11	13	30	27	29
---	----	---	----	----	----	----	----	----	----

Se reemplaza 2 por 29. $N = 9$

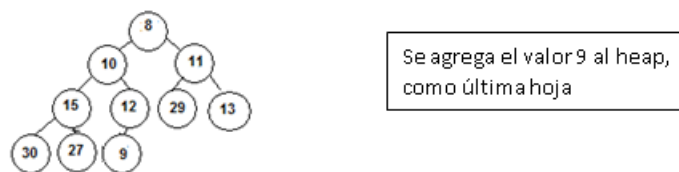


Gráfico 8.9.5: Heap: agregado de un nuevo elemento

29	10	8	15	12	11	13	30	27	29
----	----	---	----	----	----	----	----	----	----

Se compara el 29 con los hijos, se intercambia con el menor.

29	10	8	15	12	11	13	30	27
----	----	---	----	----	----	----	----	----

Intercambio del 29 con el 8

8	10	29	15	12	11	13	30	27
---	----	----	----	----	----	----	----	----

Se analiza la nueva situación del 29 con sus nuevos hijos:

8	10	29	15	12	11	13	30	27
---	----	----	----	----	----	----	----	----

Se intercambia el 29 con el 11, el menor de sus hijos

8	10	11	15	12	29	13	30	27
---	----	----	----	----	----	----	----	----

Como el 29 ya no tiene más hijos, el montículo quedó restaurado. Ahora $N = 9$.

8	10	11	15	12	29	13	30	27
---	----	----	----	----	----	----	----	----

Agregar un nuevo elemento

El nuevo elemento se ubica como ‘última hoja’. Luego se restaura el montículo analizando ternas ‘hacia arriba’ hasta ubicar el nuevo elemento en su posición definitiva:

Ahora el montículo debe ser restaurado: Analizamos la situación existente entre 9 y su nodo padre, (sólo dos nodos porque 9 ha quedado como hijo izquierdo de 12).

Queda:

Con implementación en un array, es:

8 10 11 15 12 29 13 30 27 9

Se almacena 9 como nueva hoja, es decir, se pone a continuación del último valor almacenado en el array. De esta forma, N , número de elementos del array, pasa de 9 a 10.

8 10 11 15 12 29 13 30 27 9

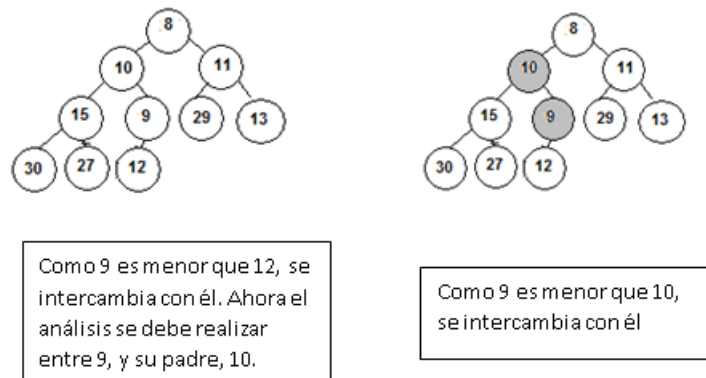
Se analiza la situación de 9 con respecto a su padre.

8 10 11 15 9 29 13 30 27 12

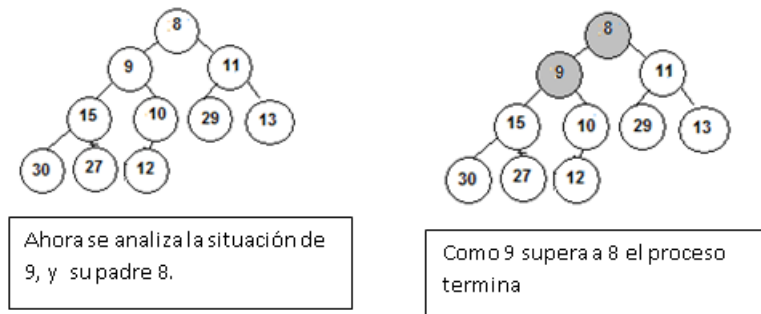
Como 9 es menor que 12, se intercambian los valores.

8 10 11 15 9 29 13 30 27 12

Ahora se analiza la situación de 9 con respecto a su padre 10



(a) Primera parte



(b) Segunda parte

Gráfico 8.9.6: Restauración de un heap en la inserción

8 9 11 15 10 29 13 30 27 12

Como 9 es menor que 10, se intercambian los valores.

8 9 11 15 10 29 13 30 27 12

Se analiza la situación de 9 con respecto a 8; como se verifica la condición del heap, no se realiza intercambio, y el proceso concluye.

8 9 11 15 10 29 13 30 27 12

El heap o montículo ha sido restaurado.

Coste de la operación:

Considerando una implementación en array, el agregado de una nueva hoja es de coste constante, ya que se agrega un elemento a continuación del último almacenado en el array. Como sucedió en el caso de la extracción de la raíz, al agregar un nuevo elemento, se realizan comparaciones (una en esta operación) y un eventual intercambio desde el nivel de las hojas hasta el nivel de la raíz. En el peor caso, se realizará esto tantas veces como niveles tenga el árbol heap, menos uno. Es decir que, considerando que la altura del árbol es aproximadamente $\log_2 N$, el coste es $O(\log N)$.

El alta de un elemento en el heap permite implementar la primitiva Acolar de la cola con prioridades.

La baja de la raíz en el árbol heap permite implementar la primitiva Desacolar de la cola con prioridades.

Aplicación al ordenamiento de arrays: el heapsort.

Este método de ordenamiento tiene 2 partes :

En la primera parte se convierte el array en un montículo, reubicando los elementos del mismo. En la segunda parte, en sucesivas etapas intercambia el elemento de la raíz con la última hoja, decrementa la posición considerada última del array y recompone el montículo desde la primera posición del array (posición de la raíz) hasta la última considerada. Observar que si se quiere ordenar el array de modo ascendente, el montículo debe verificar que el valor de cada nodo debe ser mayor que el de sus hijos, y lo contrario para un ordenamiento descendente.

Código del Heapsort:

```
void heapSort(int arr[], int n) {
// Construcción del heap o Build_heap
  for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
// extraemos de a uno los elementos del heap
  for (int i = n-1; i >= 0; i--) {
    // se intercambia el valor de la raíz
```

```

        //con el de la última hoja
        swap(arr[0], arr[i]);
        // restauramos le heap en una zona que se
        //redujo en una posición
        heapify(arr, i, 0);
    }
}

void heapify(int arr[], int n, int i) {
    int largest = i;
    // raíz del subárbol a analizar
    int l = 2*i + 1;
    // posición hijo izquierdo = 2*i + 1
    int r = 2*i + 2;
    // posición hijo derecho = 2*i + 2
    // si hijo izquierdo es mayor que la raíz
    if (l < n && arr[l] > arr[largest])
        largest = l;
    // si el hijo derecho es más grande que el mayor hasta aquí
    if (r < n && arr[r] > arr[largest])
        largest = r;
    // si el mayor no es raíz
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

```

Análisis del coste:

Como hemos dicho, el ordenamiento heapsort tiene dos etapas: en la primera se construye un heap sobre el array. En la segunda etapa, mientras la zona del heap sea mayor que 1, se intercambia el valor de la raíz con la última hoja (si se tiene un heap de máximo, esto manda dicho elemento al final de la zona del heap), se reduce la zona del heap en 1 (con lo cual no se toca la posición del elemento que antes estaba en la raíz) y se restaura el heap desde la primera posición hasta la última de la zona, es decir, se reubica en el lugar correcto la clave que se ha colocado como raíz.

La primera etapa, a primera vista puede llevarnos a la conclusión de que pertenece a $O(n * \log n)$, lo cual no está mal, pero se puede obtener un O más ajustado si consideramos que en la construcción del heap, cuando se llama a heapify, el número de pasos (comparaciones e intercambios) depende de hasta dónde cada elemento debe moverse hacia abajo en el heap. En el peor caso, el elemento podría bajar hasta el nivel de la hoja; el número de pasos depende del número de niveles del árbol. Ahora bien, si el árbol

tiene t niveles, en el nivel más bajo, hay 2^{t-1} nodos, pero no se llama a heapify para ninguno de ellos. En el siguiente nivel hay 2^{t-2} nodos, y cada uno podría moverse hacia abajo 1 nivel. En el tercer nivel (contando siempre desde abajo), hay 2^{t-3} nodos, y cada uno puede bajar 2 niveles. Haciendo la sumatoria correspondiente (se deja la última parte del cálculo como ejercicio) obtenemos que la construcción del heap pertenece a $O(n)$, siendo n el número de nodos del árbol.

En la segunda etapa, mientras el heap no esté vacío (es decir un número de veces que es $O(n)$) se realiza un intercambio entre la raíz y la última hoja, se reduce en 1 el tamaño del heap (todo esto de coste constante) y se restaura nuevamente el heap (lo cual puede tener un coste $O(\log n)$). Por tanto esta segunda etapa pertenece a $O(n \log n)$.

Por la regla de la suma, $O(n) + O(n \log n)$ es $O(n \log n)$, que es el coste del Heapsort.

Ejemplo:

Este es el array original

3 2 1 9 7 6

Las siguientes son las etapas de la conformación del heap

3 2 6 9 7 1

3 9 6 2 7 1

9 3 6 2 7 1

9 7 6 2 3 1

Ahora ya se tiene un montículo formado. Se lleva a cabo la segunda parte.

1 7 6 2 3 9

7 1 6 2 3 9

7 3 6 2 1 9

1 3 6 2 7 9

6 3 1 2 7 9

2 3 1 6 7 9

3 2 1 6 7 9

1 2 3 6 7 9

2 1 3 6 7 9

1 2 1 6 7 9

Ejercicios

- Escribir algoritmos para resolver los siguientes problemas, y luego codificarlos y evaluar su coste temporal para el peor caso.

- a) Dado un ABB, escribir un algoritmo para determinar el valor de la mayor y menor clave.
 - b) Dado un ABB y una clave, escribir un algoritmo para retornar el nivel en que se encuentra la misma, o -1 si no está en el ABB
 - c) Dado un ABB, escribir un algoritmo para establecer cuál es el nivel en el que hay más nodos, o el primero de ellos si el máximo se alcanza en más de uno.
 - d) Dado un ABB, escribir un algoritmo para determinar la altura del mismo
 - e) Dado un ABB, escribir un algoritmo para establecer si está balanceado por su altura con diferencia 1, o no
 - f) Dados dos ABB, escribir un algoritmo para determinar si son simétricos (en forma) o no.
- b. Para un ABB completo con m niveles ¿cuál es la cantidad de nodos del mismo? ¿Cuántos son hojas?
- c. Si un ABB es casi completo y tiene n nodos ¿cuántos nodos hoja tiene?
- d. Implementar las operaciones unión e intersección de conjuntos mediante implementaciones con ABB
- e. Considere que un árbol general de orden m (m es el máximo número de hijos por nodo) y altura h se representa mediante un árbol binario, en donde el enlace izquierdo sea al primer hijo y el enlace derecho sea al hermano siguiente. Escriba un algoritmo para determinar si un nodo es hijo de otro según la configuración inicial.
- f. En el peor caso ¿cuántas rotaciones hay que hacer para un alta en un AVL de altura h ?
- g. En el peor caso ¿cuántas rotaciones hay que hacer para un borrado en un AVL de altura h ?
- h. Indicar cómo se puede realizar en un array de enteros
- a) la unión,
 - b) intersección,
 - c) diferencia y
 - d) diferencia simétrica entre conjuntos.

- i. Desarrollar el algoritmo de búsqueda de una clave en un trie implementado con un árbol.
- j. Escribir el algoritmo de baja para un trie implementado con un árbol.
- k. Analizar el coste temporal del alta, baja y búsqueda de claves en un trie implementado con un árbol.
- l. Describir una estructura adecuada para implementar el trie con listas, definir los algoritmos para el alta, la baja y la búsqueda, y analizar el coste de dichas operaciones.
- m. Para las dos implementaciones vistas, diseñar algoritmos para obtener la unión y la intersección de conjuntos
- n. Para las estructuras vistas, realizar un análisis comparativo del coste temporal y espacial para las implementaciones de cada una de las operaciones básicas de un Conjunto. Discutir en qué situaciones elegiría una u otra implementación para el TDA Conjunto y Diccionario.
- ñ. Desarrollar un algoritmo Fusión, que a partir de dos montículos del mismo tipo genere uno nuevo del mismo tipo. ¿cuál es el menor coste de un algoritmo que realice esa tarea?
- o. Desarrollar un algoritmo Modificar_clave que permita modificar la prioridad de un elemento del montículo y restaurar el mismo. ¿cuál es el coste?

Capítulo 9

Grafos

El origen del concepto de grafo se remonta a la ciudad de Königsberg, en Prusia (actualmente, Kaliningrado, en Rusia). Por esa ciudad pasa el río Pregel y existen siete puentes, como muestra el gráfico 9.0.1. En épocas del matemático Euler, la gente del lugar se desafiaba a hacer un recorrido que permitiera atravesar cada puente una sola vez regresando al punto de partida.

■

Este problema, que tenía gran dificultad ya que nadie superaba el desafío, atrajo la atención de Euler, quien lo analizó empleando una técnica de graficado por la cual redujo a puntos la representación de las islas y las orillas, y a arcos los siete puentes. El gráfico resultante quedó como el que se aprecia en 9.0.2. A los puntos se los denominó ‘nodos’ o ‘vértices’. Los enlaces son ‘aristas’ o ‘arcos’.

El problema original se volvió entonces análogo al de intentar dibujar el grafo de la figura partiendo de un vértice, sin levantar el lápiz, sin pasar dos veces por el mismo arco y terminando en el mismo vértice desde donde se

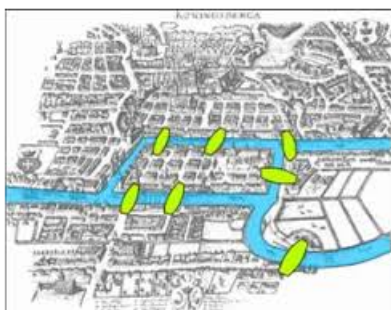


Gráfico 9.0.1: Kaliningrado

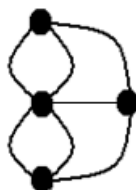


Gráfico 9.0.2: Grafo de Königsberg

había partido. Euler consideró que en un gráfico de este tipo había nodos “intermedios” y de “inicio o finalización”. Los nodos intermedios tienen un número par de aristas incidentes en ellos (ya que “si se llega” a uno de ellos se debe “volver a salir”, puesto que son intermedios). Si hay un nodo de inicio y otro de finalización, es decir, si el dibujo se comienza en uno y se termina en otro, ambos deben tener un número impar de aristas incidentes en ellos, pero si el nodo de inicio coincide con el de finalización, el número de aristas incidentes en él también debe ser par.

Analizando la situación de los puentes, concluye que el problema de los puentes de Königsberg no tenía solución. Así comenzó la teoría de grafos. (puede resultar interesante mirar este texto

<http://eulerarchive.maa.org/docs/originals/E053.pdf>)

Los grafos constituyen una muy útil herramienta matemática para modelizar situaciones referidas a cuestiones tan diferentes como lo son mapas de interrelación de datos, carreteras, cañerías, circuitos eléctricos, diagrama de dependencia, etc.

9.1. Definición matemática de grafo

Digrafo

Un grafo dirigido o digrafo o grafo orientado consiste en una dupla formada por un conjunto V de vértices o nodos del grafo, y un conjunto de pares ordenados A (aristas orientadas) pertenecientes a $V \times V$. La relación establecida entre los vértices en un digrafo es antisimétrica.

En símbolos el grafo dirigido G es $G = (V; A)$ donde A es un subconjunto de $V \times V$ (aristas orientadas o dirigidas)

Por ejemplo, el grafo que vemos en 9.1.1 es:

$$V = A, B, C, D, E$$

$$A = (A, D), (A, E), (E, A), (D, E), (C, B)$$

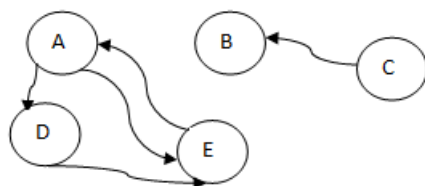


Gráfico 9.1.1: Digrafo

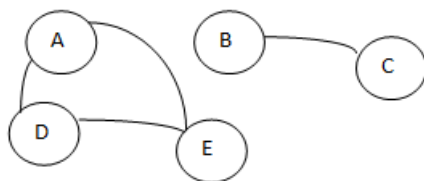


Gráfico 9.1.2: Grafo no dirigido

Grafo no dirigido

Un grafo no dirigido (o no orientado) es una dupla formada por un conjunto V de vértices o nodos del grafo, y un conjunto de pares *no* ordenados A (aristas no orientadas) pertenecientes a $V \times V$. La relación establecida entre los vértices es simétrica.

En símbolos el grafo dirigido G es $G = (V; A)$ donde A es un conjunto de pares no ordenados de $V \times V$ (Aristas *no* orientadas o *no* dirigidas). Esto significa que si hay un camino o modo de llegar desde F hasta G , también lo habrá de G a F .

Por ejemplo, el grafo que vemos en 9.1.2 se define como:

$V = A, B, C, D, E$

$A = (A, D), (A, E), (D, E), (C, B)$ (pares no ordenados)

En ambos casos, si (v, w) pertenece a A se dice que w es adyacente a v .

Algunas definiciones más

- Camino: serie alternada de vértices y aristas que inicia y finaliza con vértices y donde cada arista conecta el vértice que le precede con el que le sucede.
- Longitud de camino: cantidad de aristas del camino.
- Camino abierto: camino donde el vértice inicial y final difieren.
- Camino cerrado: camino donde el vértice inicial y final coinciden.

- Recorrido: camino que no repite aristas.
- Ciclo: camino simple cerrado o bien, camino que contiene al menos tres vértices distintos tales que el primer vértice es adyacente al último.
- Grafo no dirigido conexo: un grafo no orientado es conexo si para todo vértice del grafo hay un camino que lo conecte con otro vértice cualquiera del grafo.
- Árbol libre: es un grafo no dirigido conexo sin ciclos.
- Grafo subyacente de un grafo: es el grafo no dirigido que se obtiene reemplazando cada arista (orientada) del mismo, por una arista no orientada.
- Grafo dirigido fuertemente conexo: un grafo dirigido es fuertemente conexo si y solo si entre cualquier par de vértices hay un camino que los une.
- Grafo dirigido débilmente conexo: es aquel grafo dirigido que no es fuertemente conexo y cuyo grafo subyacente es conexo.

9.2. TDA Grafo

Es un TDA contenedor de un conjunto de datos llamados nodos y de un conjunto de aristas, donde cada una de las cuales se determina mediante un par de nodos.

Operaciones

- Crear grafo.
Esta primitiva genera un grafo vacío.
Precondición: -----
Poscondición: grafo generado vacío
- Destruir grafo:
esta primitiva destruye el grafo.
Precondición: que el grafo exista .
Poscondición: recursos liberados
- Insertar nodo:
esta primitiva inserta en el grafo un nodo nuevo recibido como argumento.

Precondición: que el grafo exista y que el nodo no esté previamente.

Poscondición: el grafo queda modificado por el agregado del nuevo nodo

- Insertar arista:

esta primitiva inserta en el grafo una arista nueva recibida como argumento

Precondición: que el grafo exista, que la arista no esté previamente y que existan en el grafo los nodos origen y destino de la arista.

Poscondición: el grafo queda modificado por el agregado de la nueva arista

- Eliminar nodo:

esta primitiva elimina del grafo un nodo recibido como argumento

Precondición: que el grafo exista y que el nodo a eliminar esté en él y no tenga aristas incidentes en él.

Poscondición: el grafo queda modificado por la eliminación del nodo

- Eliminar arista:

esta primitiva elimina del grafo una arista recibida como argumento

Precondición: que el grafo exista y la arista estén en él.

Poscondición: el grafo queda modificado por la eliminación de la arista

- Existe arista:

esta primitiva recibe una arista y retorna un valor logico indicando si la arista existe en el grafo

Precondición: que el grafo exista

Poscondición: arista es un valor válido

- Existe nodo:

esta primitiva recibe una arista y retorna un valor logico indicando si el nodo existe en el grafo.

Precondición: que el grafo exista

Poscondición: nodo es un valor válido

Pueden considerarse también las operaciones correspondientes a los recorridos como básicas en el TDA o bien plantearlas utilizando iteradores para navegar dentro del contenedor.

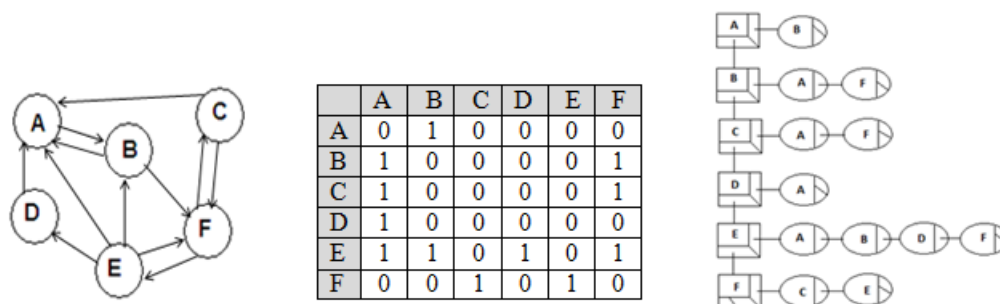


Gráfico 9.3.1: Representación de un grafo

9.3. Implementaciones

Estructuras de datos utilizadas para implementar grafos

Se puede utilizar una matriz de adyacencia, la cual, para N nodos es de $N \times N$.

- Si M es la matriz de adyacencia del grafo G entonces verifica que $M[i][j]$ es true o 1 si la arista (i, j) pertenece al grafo, o false o bien 0 si no pertenece.
- Otra posibilidad es usar listas de adyacencia. En este caso, la representación es mediante una lista de listas. De este modo, se tiene una lista con nodos; cada uno de ellos conteniendo la información sobre un vértice y un puntero a una lista ligada a los vértices adyacentes al vértice indicado.

Por ejemplo, el grafo que vemos en la parte izquierda de la figura 9.3.1, tiene la matriz y la lista de adyacencia que allí se muestra.

9.4. Recorridos

Recorridos de un grafo dirigido

Los dos recorridos básicos en los grafos dirigidos son en profundidad y en anchura. Cada uno de ellos procesa o visita cada vértice del grafo una y solo una vez. Además, si se verifica que el grafo sea acíclico, se pueden llevar a cabo los recorridos topológicos.

Recorrido en profundidad

Depth first search o “busqueda en profundidad” consiste en para cada vértice del grafo que no ha sido visitado previamente visitarlo y luego pasar al primer adyacente no visitado, luego al primer adyacente del adyacente que no haya sido visitado, y así hasta llegar a un nodo que no tenga adyacentes no visitados. Entonces se retrocede para pasar al siguiente adyacente del anterior vértice y realizar el mismo proceso hasta agotar los adyacentes no visitados. El retroceso se hace cuando los vértices se agotan. Para implementar el algoritmo suele usarse una pila o bien un algoritmo recursivo. Puede, además, numerarse los vértices a medida que se visitan, como se lleva a cabo en los algoritmos que se describen a continuación.

```

DFSNumerandoRec {
    para cada vértice v {
        marcar v como no visitado;
        asignar 0 a numero de v;
        indice = 0;
    }
    para cada vértice v {
        si (v no visitado)
            num (v, indice);
    }
}

num (vertice u, entero& nd) {
    nd = nd + 1;
    asignar nd a número de u;
    marcar u como visitado
    para cada vértice w adyacente a u {
        si (w no visitado)
            num( w, nd);
    }
}

```

Análisis del coste temporal El coste depende básicamente de las estructuras que se usen para almacenar el grafo. Considerando un grafo de v vértices y a aristas, si se trabaja con una implementación de matriz de adyacencia, el ciclo que marca cada vértice como no visitado es $O(v)$. Luego se ejecuta el ciclo que revisa si cada vértice ha sido o no visitado para invocar a *num*. El ciclo externo tiene $O(v)$ iteraciones y el coste del bloque depende del coste de *num*.

num comienza con unas sentencias $O(1)$ y luego hay un ciclo que analiza cada adyacente a u para invocar a *num* para cada adyacente que no haya sido visitado. Como para determinar los adyacentes a u hay que recorrer la fila correspondiente de la matriz, el ciclo tiene $O(v)$ iteraciones, siendo n el número total de veces que se ejecuta *num*. Entonces, el coste de la búsqueda en profundidad para el caso de una implementación con matriz de adyacencia pertenece a $O(v^2)$.

Si se ha implementado el grafo con listas de adyacencia, marcar cada vértice como no visitado es $O(v)$. *num* se invoca para cada vértice en el peor caso tantas veces como adyacentes tenga el vértice, por tanto, el ciclo que analiza cada vértice w adyacente a u realiza en total $2a$ iteraciones. Entonces tenemos que el coste se puede expresar como $O(v)+O(a)$, lo que generalmente se indica como $O(v + a)$.

Recorrido en Profundidad iterativo con numeración de vértices

```
DFSNumerandoIt {
    inicializar pila p a vacío;
    indice = 0;
    para cada vértice v {
        marcar v como no visitado;
        asignar 0 a numero de v;
        p.apilar (v);
    }
    mientras p no vacía {
        u = p.cima( );
        p.desapilar( );
        si (u no visitado) {
            marcar u como visitado
            índice = índice + 1;
            asignar índice a numero de u;
            para cada w adyacente a u
                si (w no visitado)
                    p. apilar(w);
        }
    }
}
```

Análisis del coste temporal Si se trabaja con una implementación de matriz de adyacencia y considerando $O(1)$ el coste de apilar y desapilar, el primer ciclo es $O(v)$. Luego se tiene un ciclo que se ejecuta mientras no se haya vaciado la pila. En ese ciclo, si el vértice desapilado no se ha visitado (lo cual sucederá v veces en total), se lo numera y se analizan sus adyacentes (a lo sumo $v - 1$) para apilarlos si no han sido visitados aún. De lo cual se deduce un coste temporal $O(v^2)$ para el recorrido con la implementación

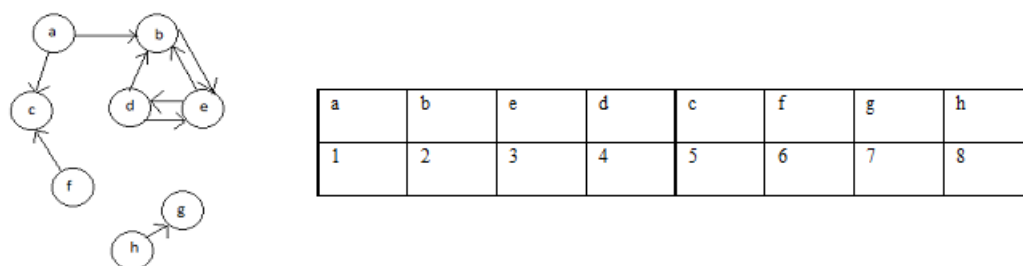


Gráfico 9.4.1: Recorrido en profundidad

indicada.

Con una implementación con listas de adyacencia, el primer ciclo tiene coste $O(v)$. Luego se tiene, para cada vértice desapilado y no visitado, el análisis de la situación de sus adyacentes, para establecer si se visitaron o no. En total esto se realiza $O(a)$ de veces. Entonces, el coste temporal total del recorrido en profundidad iterativo con implementación en listas de adyacencia es $O(v + a)$.

En el gráfico 9.4.1 vemos un grafo dirigido y su recorrido en profundidad.

Bosque asociado al recorrido en profundidad

Al ejecutar un recorrido en profundidad en un grafo $G = (V, E)$ se puede obtener un bosque de ejecución. La raíz de cada árbol de este bosque es cada uno de los vértices por los cuales se ha comenzado un recorrido parcial. Las aristas que aparezcan en cada árbol del bosque enlazan vértices visitados con el que se visitó a continuación. A continuación vemos un algoritmo variante del recorrido en profundidad que permite obtener el bosque asociado a un grafo dirigido.

```

Bosque-DFS {
  para cada vértice v
    marcar v como no visitado
  inicializar el bosque B como vacío
  para cada vértice v {
    si (v no visitado) {
      inicializar el árbol T como vacío;
      arbolDFS(v, T);
      agregar T al bosque B;
    }
  }
  retornar B
}

```

```

ArbolDFS (vértice u; árbol & T) {

```

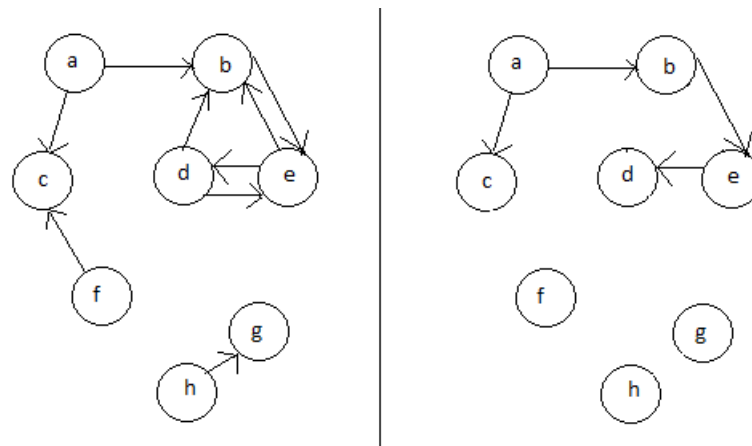


Gráfico 9.4.2: Grafo con bosque asociado

```

marcar u como visitado;
agregar vértice al árbol T;
para cada w adyacente a u {
    si (w no visitado) {
        ArbolDFS(w,T)
        agregar arista (u, w) a T;
    }
}

```

Análisis del coste temporal Si se asume un coste $O(1)$ para las operaciones de inicializar Árbol, inicializar Bosque, agregar vértice al árbol y agregar arista al Árbol, la generación del bosque asociado al recorrido en profundidad tendrá un coste análogo al del recorrido en profundidad según se haya implementado el grafo con matriz de adyacencia o con listas de adyacencia.

En el gráfico 9.4.2 vemos un grafo del lado izquierdo con el bosque asociado al recorrido en profundidad en la parte derecha.

Test de aciclicidad

El test de aciclicidad es una aplicación del recorrido en profundidad. A menudo es necesario determinar si un grafo tiene o no ciclos (de no tenerlos se lo denomina acíclico). Para averiguarlo es posible usar el recorrido en profundidad y detectar si hay “aristas de retroceso” que conecten un vértice con otro que aparece como adyacente pero que haya sido visitado antes en el recorrido parcial que se está realizando.

Dado $G = (V, E)$ grafo dirigido y $TDFS$ árbol del recorrido en profundi-

dad de G , decimos que G contiene un ciclo si y sólo si G contiene al menos una arista de retroceso.

```

TestAciclicidad {
    inicializar b en falso; //valor lógico a retornar
    para cada vértice v {
        marcar v como no visitado;
        v.enCaminoActual = falso ;
    }
    para cada vértice v {
        si ( -b ) y (v no visitado)
            analizarRecParcial(v, b);
    }
    retornar b;
}

AnalizarRecParcial (vértice u, bool& b) {
    marcar u como visitado;
    u.enCaminoActual = verdadero;
    para cada w adyacente a u {
        si ( -b ) {
            si (w visitado) y (w.enCaminoActual)
                b = verdadero;
            si (w no visitado)
                AnalizarRecParcial(w,b);
        }
    }
    u.enCaminoActual = falso;
}

```

Análisis del coste temporal Los costes son los mismos que se han discutido para el recorrido en profundidad, según la implementación que se haya hecho para el grafo.

Obtención de los puntos de articulación de un grafo

Otra aplicación del recorrido en profundidad es la obtención de los puntos de articulación de un grafo. Un punto de articulación o vértice de corte de un grafo no dirigido y conexo es un vértice tal que al ser eliminado el grafo deja de ser conexo. Si un grafo no tiene puntos de articulación significa que para todo par de vértices existe más de un camino que los enlaza. Entonces, si un vértice del grafo “fallara” (es decir, si hubiera que considerarlo como retirado del grafo por algún motivo) el grafo permanecería aún conexo. A los grafos sin puntos de articulación se los llama biconexos.

Definiciones

Punto de articulación: dado $G = (V, A)$, grafo no dirigido conexo, $v \in V$ es un punto de articulación sí y solo sí el subgrafo $G' = (V - v, A')$, no es conexo, siendo $A' = A - (s, t) / (s, t) \in A, (s = v) \vee (u = v)$

Grafo biconexo: grafo conexo y sin puntos de articulación (aunque “falle” un vértice, se mantiene la conectividad).

Ahora bien, si consideramos un grafo no dirigido y conexo, el bosque generado en un DFS y determinadas las aristas de retroceso

- Si r es raíz es el árbol DFS y tiene más de un hijo en el árbol, entonces r es un punto de articulación.
- Para todo vértice u que no sea raíz en árbol DFS, u es punto de articulación si al eliminarlo del árbol resulta imposible “volver” desde alguno de sus descendientes (en el árbol) hasta alguno de los antecesores de u .

Para analizar el caso de los vértices que no son raíz, establecemos para cada uno cuál es el valor más bajo (orden de un nodo en recorrido) entre el número de orden del nodo, el que al que se puede acceder desde él “bajando” arcos en el árbol generado con el recorrido, y el de aquel al que se acceda “subiendo” a través de alguna arista de retroceso.

$\text{bajo}(u) = \text{mínimo número asignado en el recorrido en profundidad considerando estos valores:}$

- numdfs del nodo u
- numdfs de cada nodo w al que se accede desde u por una arista de retroceso (es decir (u, v) es una arista que no está en el árbol que se genera durante el recorrido).
- $\text{bajo}(x)$ para todo x hijo de u en el árbol que se genera en el recorrido.

Es decir, $\text{bajo}(u) = \text{Minimo}(\text{numdfs}(u), \text{numdfs}(w) / \exists w \in V \text{ tal que } (u, w) \in A \text{ y } (u, w) \notin \text{árbol DFS (salimos de } u \text{ subiendo por una arista de retroceso back)}, \text{bajo}(x) / \forall x \in \text{hijos}(u, \text{TDFS})$

Entonces el vértice u , si no es raíz, es punto de articulación si tiene al menos un hijo x tal que $\text{bajo}(x) > \text{numdfs}(u)$.

A continuación el pseudocódigo:

```
Puntos de Articulación {
// g es grafo no dirigido conexo
// se retorna lis, lista de vértices que son puntos de articulación
para cada vértice v {
```



```

    marcar v como no visitado;
    v.ndfs = 0; //inicializa en 0 en la numeración a v
    v.bajo = 0; //inicializa bajo de v a 0
}
indice = 0;
inicializar lis como vacía;
PuntoArticRec(v, indice, lis, -1);
//el ultimo argumento es un valor que representa
//un vértice a analizar en función de v;
//inicialmente se pasa un valor no válido, a modo de "nulo"
retornar lis;
}

PuntoArticRec (vértice v,
    entero& i,
    lista lis;
    vértice p)
i = i + 1;
v.ndfs = i;
v.masbajo = v.ndfs;
marcar v como visitado;
esPuntoArtic = falso;
numeroHijos = 0;
// corresponde al numero de hijos procesados
para cada w adyacente a v {
    si (w visitado) y (w != p)
        v.masBajo = min (v.masBajo,w.ndfs);
    si (w no visitado) { P
        untoArticRec( w, i, l, v );
        v.ma := min (v.masbajo, w.masbajo);
    }
    si ( -p )
        numerohijos++;
    si no {
        si (w.masbajo >= v.ndfs) {
            esPuntoArtic = verdadero;}
    }
    si ( -p ) y (numerohijos > 1)
        lis.agregar(v);
    si (p != null) y (esPuntoArtic)
        lis.agregar.(v);
}
}

```

Análisis del coste temporal: Como el algoritmo utiliza una búsqueda en profundidad a la cual incorpora sentencias y bloques $O(1)$, el coste temporal del algoritmo es análogo al de la búsqueda en profundidad según el tipo de implementación que se haya hecho del grafo.

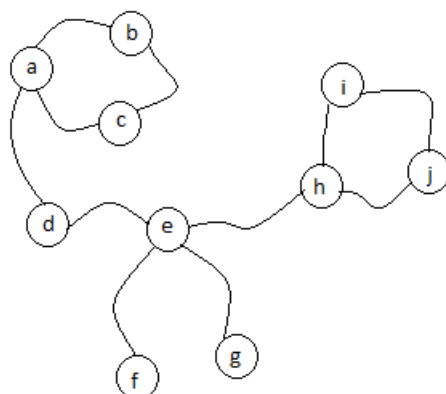


Gráfico 9.4.3: Grafo para obtener puntos de articulación

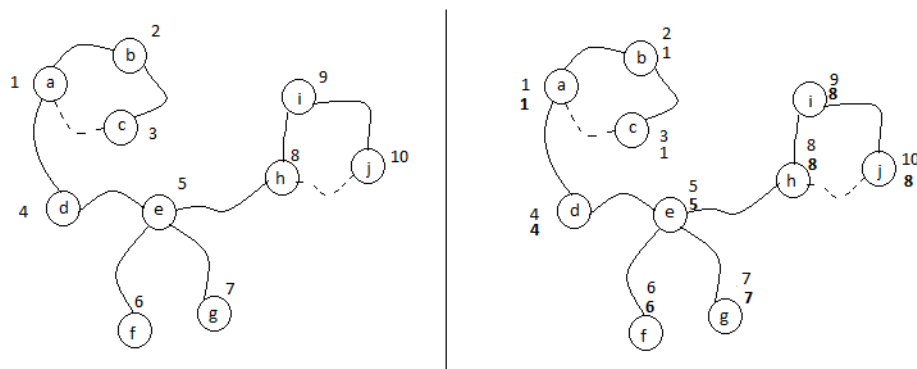


Gráfico 9.4.4: Búsqueda de puntos de articulación

Ejemplo 9.1. Obtención de los puntos de articulación del grafo que vemos en 9.4.3.

Se realiza el recorrido, obteniéndose la numeración según el gráfico 9.4.4, parte de la izquierda. Luego se calcula $\text{bajo}(v)$ para cada vértice v , comenzando por el de valor más alto, obteniéndose la información del gráfico de la derecha (los valores de bajo están en negrita).

Vértices padres e hijos en árboles generados durante el DFS (ver 9.4.5).

Puntos de articulación:

- a, por ser raíz y tener dos hijos en el árbol
- d, porque $\text{bajo}(e) = 5$ y $\text{numero}(d) = 4$
- e porque $\text{bajo}(h) = 8$ y $\text{numero}(e) = 5$, porque $\text{bajo}(g) = 7$ y $\text{numero}(e) = 5$, porque $\text{bajo}(f) = 6$ y $\text{numero}(e) = 5$, (es suficiente

Vértice	Hijos en el árbol del recorrido
a	b, d
b	c
c	---
d	e
e	f, g, h
f	---
g	---
h	i
i	j
j	---

Gráfico 9.4.5: Puntos articulación

que suceda con uno de los hijos)

- h, porque bajo de i es 8 y numero de h es 8

Obtención de las componentes fuertemente conexas en un grafo dirigido: (aplicación del recorrido en profundidad).

Una componente fuertemente conexa (CFC) de un grafo es un conjunto maximal de vértices tal que el subgrafo formado es fuertemente conexo. Las componentes conexas para un grafo $G(V, A)$ pueden obtenerse de la siguiente forma:

- a. Realizar un recorrido DFS(G) etiquetando los vértices con la numeración en el orden inverso del recorrido (invnum).
- b. Obtener el grafo transversal GT invirtiendo el sentido de las aristas de G .
- c. Realizar un recorrido DFS(GT) en orden decreciente de los valores invnum de los vértices. Cada árbol obtenido es una componente fuertemente conexa de G .

Se deja como ejercicio el análisis del coste temporal.

Ejemplo 9.2. Obtención de las componentes fuertemente conexas del grafo de la figura 9.4.6.

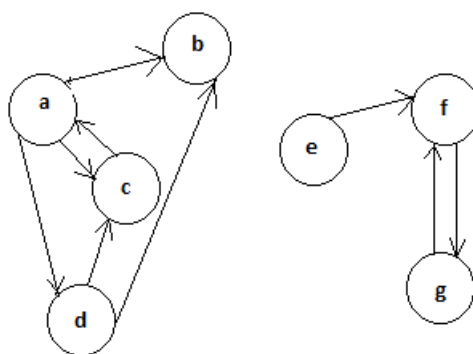


Gráfico 9.4.6: Grafo para búsqueda de CFC

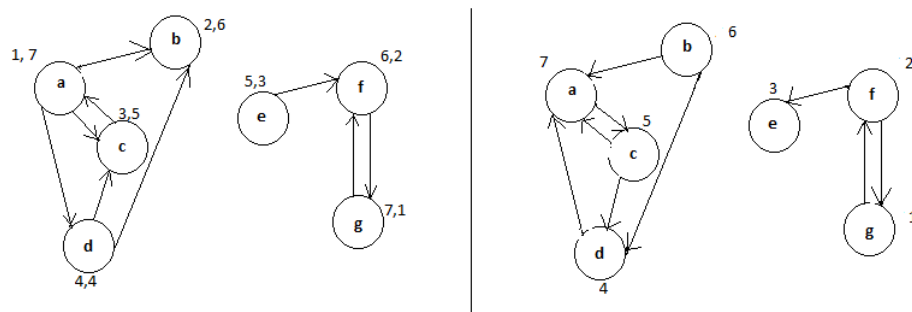


Gráfico 9.4.7: Componentes fuertemente conexos

A la izquierda de la figura 9.4.7, el grafo recorrido. Para cada nodo se indican dos valores: el número de recorrido en orden directo, y en el orden inverso. A la derecha, el grafo invertido con la numeración en orden inverso.

Al recorrer el grafo invertido considerando los vértices en orden inverso, se obtienen las componentes conexas del grafo, que en este caso son:

- Componente 1: a, c, d
- Componente 2: b
- Componente 3: e
- Componente 4: f, g

Recorrido en anchura

Breadth first search o 'búsqueda primero en anchura'. Este recorrido marca inicialmente todos los nodos como no visitados, y luego, para cada nodo no visitado, lo visita y marca, procediendo luego a visitar cada uno

de sus adyacentes no visitados antes, luego de lo cual se continúa con los adyacentes del primer adyacente, los del segundo adyacente, etc., hasta agotar el conjunto de vértices del conjunto. Se presenta debajo el algoritmo que recorre en anchura numerando los vértices en el orden en que se visitan.

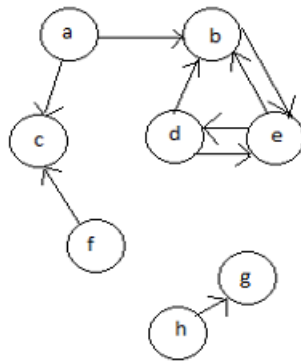
```

BFS {
    para todo vértice v
        asignar a v numero 0
    índice = 1;
    incializar cola c;
    mientras exista un vértice v tal que numero de v sea 0 {
        asignar a v numero índice;
        incrementar índice;
        c.acolar (v);
        mientras cola no vacía {
            v = c.desacolar();
            para cada vértice u adyacente a v {
                si (numero de u es 0 {
                    asignar a número de u el valor de índice
                    incrementar índice
                    c.acolar(u);
                }
            }
        }
    }
}

```

Análisis del coste temporal: El coste depende de cómo se haya implementado el grafo. Si se lo implementó con una matriz de adyacencia, tenemos lo siguiente: el coste de marcar cada nodo como no visitado es $O(v)$. Luego hay un ciclo de $O(v)$ iteraciones, en cada una de las cuales se analizan los adyacentes del vértice que se desacola (como la implementación es con matriz de adyacencia, esto tiene un coste $O(v)$). Por lo cual el coste del recorrido en anchura con implementación de matriz de adyacencia es $O(v^2)$. Si la implementación es con listas de adyacencia, hay que considerar el coste de marcar cada nodo como no visitado, que es $O(v)$, y luego observar que en total tantas veces como aristas haya se va a analizar si el vértice adyacente al que se ha desacolado se ha visitado, para acolarlo si aún no se lo visitó. Entonces, el coste es $O(v) + O(a)$, indicado como $O(v + a)$.

Ejemplo 9.3. recorrido en anchura: se muestra el recorrido en anchura del grafo que está en 9.4.8.



a	b	c	e	d	f	g	h
1	2	3	4	5	6	7	8

Gráfico 9.4.8: Recorrido en anchura

Recorridos topológicos

Estos recorridos solo se aplican a grafos dirigidos acíclicos, y permiten linealizar un grafo, es decir, recorrer los vértices del mismo respetando las precedencias. Pueden plantearse recorridos topológicos como variantes del recorrido en profundidad y del recorrido en anchura.

Recorrido topológico en profundidad

```

// devuelve una lista en la cual quedan insertados los nodos
// según fueron visitados en el recorrido
TopolDFS {
    inicializar lista de salida; //lista vacía
    para cada vértice v
        marcarlo como no visitado
    para cada vértice v {
        si (v no visitado)
            Rec (v, lista)
    }
    retornar lista
}

Rec (vertice v, lista lis) {
    marcar v como visitado
    para cada vértice w adyacente a v {
        si (w no visitado)
            Rec(w, lista)
    }
    insertar v al frente en lis
}

```

Análisis de coste temporal: Análogo al del recorrido en profundidad habitual, según la implementación usada.

Recorrido topológico en anchura

Se comienza calculando para cada vértice el grado de entrada del mismo (número de aristas incidentes en el vértice). Los vértices de grado de entrada 0 se acolan (tiene que haber vértices con grado de entrada 0; de no ser así, el grafo tendría ciclos). Luego se procesa la cola del siguiente modo: mientras no esté vacía se desacola y se visita un vértice (y se dirigirá a la salida, si queremos el listado respetando precedencias) y para cada vértice adyacente al mismo se decrementa en 1 el grado de entrada del mismo (como si se retirara el vértice desacolado del grafo). Cada vértice que llegue a 0 por este decremento, se acola.

```

TopolBFS {
    //genera lista de salida, con todos los vértices del grafo
    inicializar T; //tabla de grados de entrada,
    // de N posiciones, siendo n los vértices.
    inicializar cola c; // cola para vértices del grafo;
    // se inicializa vacía.
    inicializar lis; // lista de salida, se inicializa vacía.
    para cada vértice v {
        T[v] = grado de entrada de v;
        //se registra el grado de entrada de cada vértice
        //Siempre hay al menos un vértice de grado 0;
        //de otro modo, el grafo no seria acíclico
        si (grado de entrada de v == 0)
            c. acolar (v);
    }
    mientras (cola c no vacía) {
        x = c.desacolar();
        lis.insertarAlFinal(x);
        para cada vértice w adyacente a v {
            T[w] = T[w] - 1;
            si (T[w] == 0)
                c.acolar (w);
        }
    }
    retornar lis;
}

```

Se deja como ejercicio el análisis del coste temporal para diversas implementaciones.

Ejemplo 9.4. Consideremos el grafo dirigido y acíclico que vemos en 9.4.9.

Según el algoritmo anterior:

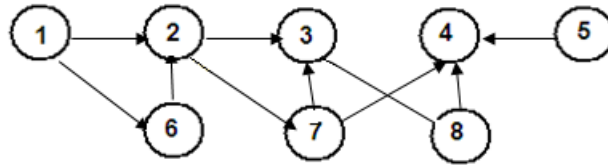


Gráfico 9.4.9: Grafo dirigido y acíclico

```

{
  rec(1)
  adyacentes(1): 2,6 {
    rec(2)
    adyacentes(2): 3,7 {
      rec(3)
      adyacentes(3):8 {
        rec(8)
        adyacentes(8):4 {
          rec(4)
          adyacentes(4): -
          insertar 4 al frente en la lista de salida
        }
        insertar 8 al frente en la lista de salida
      }
      insertar 3 al frente en la lista de salida
    }
  }
}

{
  rec(7)
  adyacentes(7): 3, 4 (ambos ya visitados)
  insertar 7 al frente en la lista de salida
}

insertar 2 al frente en la lista de salida
}

{
  rec(6)
  adyacentes(6): 2 (ya visitado)
  insertar 6 al frente en la lista de salida
}

insertar 1 al frente en la lista de salida
}

{
  rec(5)
  adyacentes(5): 4 (ya visitado)
  insertar 5 al frente en la lista de salida
}
}

```

De este modo, la lista generada que respeta las precedencias entre los nodos con un recorrido en profundidad es: 5, 1, 6, 2, 7, 3, 8, 4.

Ejemplo de recorrido topológico en anchura para el grafo anterior,

La tabla de grados de entrada es:

vértice	1	2	3	4	5	6	7	8
grado entrada	0	2	2	3	0	1	1	1

Se acolan el 1 y el 5

Cola : 1 – 5

Se desacola y procesa el 1

Se decrementan los adyacentes a 1, que son 2 y 6.

vértice	1	2	3	4	5	6	7	8
grado entrada	0	1	2	3	0	0	1	1

6 ahora vale 0 y se acola.

Cola: 5 – 6

Se desacola y procesa 5.

Se decrementan los adyacentes a 5, que es 4.

vértice	1	2	3	4	5	6	7	8
grado entrada	0	1	2	2	0	0	1	1

Cola: 6

Se desacola y procesa 6.

Se decrementa el adyacente a 6, que es 2.

vértice	1	2	3	4	5	6	7	8
grado entrada	0	0	2	2	0	0	1	1

2 vale ahora 0 y se acola.

Cola: 2

Se desacola y procesa 2.

Se decrementan los adyacentes a 2, que son 3 y 7.

vértice	1	2	3	4	5	6	7	8
grado entrada	0	0	1	2	0	0	0	1

7 llega a 0, se acola.

Cola: 7

Se desacola y procesa 7

Se decrementan los adyacentes a 7, que son 3 y 4

vértice	1	2	3	4	5	6	7	8
grado entrada	0	0	0	1	0	0	0	1

3 llega a 0, se acola

Cola: 3

Desacolar 3

Decrementar el adyacente a 3, que es 8

vértice	1	2	3	4	5	6	7	8
grado entrada	0	0	0	1	0	0	0	0

8 vale 0, se acola

Cola: 8

Desacolar 8

Decrementar el adyacente a 8, que es 4

vértice	1	2	3	4	5	6	7	8
grado entrada	0	0	0	0	0	0	0	0

4 se acola porque llego a 0.

Cola: 4

Desacolar y procesar 4, que no tiene adyacentes.

La linealización del grafo da por resultado esta lista (se han indicado las precedencias con arcos) 1-5- 6- 2- 7- 3- 8- 4

9.5. Caminos en un grafo

Hay un gran número de problemas sobre caminos en grafos dirigidos y en grafos no dirigidos. Algunos plantean determinar si dado un par de vértices, o todos los pares de vértices, hay o no camino entre ellos. Otros trabajan sobre grafos con aristas o con vértices ponderados. Una ponderación es un valor asociado a una arista o a un vértice o a ambos.

En esta sección trataremos algunos de los problemas más comunes sobre caminos en grafos con aristas ponderadas.

Problema de los caminos más cortos con un solo origen

Dado un grafo dirigido $G = (v, a)$, en el cual cada arco tiene asociado un costo no negativo, y donde un vértice se considera como origen, el problema de los “caminos más cortos con un solo origen” consiste en determinar el costo del camino más corto desde el vértice considerado origen a todos los otros vértices de v . Este es uno de los problemas más comunes que se plantean para los grafos dirigidos con aristas ponderadas (es decir con peso en las aristas). La ‘longitud’ o ‘costo’ de un camino es la sumatoria de los pesos de las aristas que lo conforman.

El modelo de grafo dirigido con aristas ponderadas no negativas puede, por ejemplo, corresponder a un mapa de vuelos en el cual cada vértice represente una ciudad, y cada arista (v, w) una ruta aérea de la ciudad v a la ciudad w .

Algoritmo de Dijkstra para el cálculo de los caminos mínimos

Este algoritmo desarrollado por Dijkstra en 1959 se caracteriza por usar una estrategia greedy, voraz (o ávida). Para aplicar este algoritmo el peso de

las aristas debe ser no negativo.

En esta estrategia se trata de optimizar (alcanzar el máximo o el mínimo) de una función objetivo, la cual depende de ciertas variables, cada una de las cuales tiene un determinado dominio y está sujeta a restricciones. En cada etapa se toma una decisión que no tiene vuelta atrás, y que involucra elegir el elemento más promisorio (es decir, el que parece ofrecer más posibilidades de mejorar la función objetivo) de un conjunto y se analizan ciertas restricciones asociadas a cada variable de la función objetivo para ver si se verifican mejoras en ella. La solución puede expresarse como una sucesión de decisiones, y en cada etapa de la estrategia se elegirá la mejor opción de las disponibles (óptimo local), analizándose luego si esta elección verifica las restricciones, constituyendo una solución factible.

La estrategia greedy, muy interesante en sí, no sirve para cualquier problema: se debe tratar de un problema de optimización; pero aún en este caso, *no asegura* que se llegue al óptimo global (podría llevarnos a un óptimo local) ni tampoco, en algunos casos, llegar a una solución factible: el problema en cuestión debe verificar ciertas condiciones para que quede garantizado que la estrategia greedy funcione (estas condiciones se verifican para el problema de los caminos mínimos con origen dado).

Dado un grafo dirigido $G = (V, A)$, con aristas ponderadas, y considerando un vértice como origen, determinar los costos del camino mínimos desde el vértice considerado origen a todos los otros vértices de V .

Resolución de Dijkstra utilizando una matriz de pesos y una tabla de vértices visitados: Se tiene un grafo dirigido $G = (V, A)$ con aristas no negativas. Consideraremos los vértices etiquetados $1..n$

El vértice de partida será 1.

S será el conjunto de vértices visitados a lo largo del procesamiento.

```
CamMinDijk {
  // D es un vector de n-1 elementos
  // para calcular los valores de los respectivos caminos minimos
  inicializar S = { 1 }
  para cada vertice i desde 2 hasta N {
    D[i] = coste inicial de alcanzar i desde el vertice 1
    //corresponde al peso de la arista (1, i),
    //si existe, o bien un peso maximo
  }
  mientras (conjunto de vertices no visitados V-S no este vacio) {
    elegir vertice w perteneciente a V-S tal que D[w] sea minimo
    S = S U {w}
  {
    D[v] = min ( D[v], D[w] + M[w,v])
  }
}
```

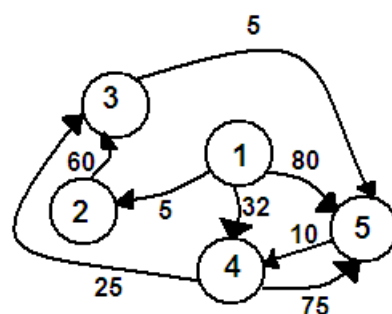


Gráfico 9.5.1: Grafo a aplicar Dijkstra

}

Consideraciones sobre el coste temporal: Si se utiliza una matriz de adyacencia y una tabla para almacenar los vértices y sus marcas de visitados, resulta: $O(v)$ la carga inicial de D y la marca de no visitado para los $n - 1$ vértices candidatos. El ciclo que se ejecuta mientras haya candidatos, es obviamente $O(v)$. La función de elección de un vértice de coste mínimo no visitado implica recorrer tanto la tabla de visitados como el vector D , es $O(v)$.

El análisis de cada adyacente al nodo elegido, por estar utilizándose una matriz de pesos, implica recorrer la fila correspondiente al vértice elegido, es decir $O(v)$. Aplicando las conocidas reglas de la suma y del producto, se llega a que el algoritmo es $O(v^2)$.

Ejemplo 9.5. Consideraremos el grafo que se muestra en 9.5.1 y los valores asociados a los arcos los presentaremos en una tabla; pero hay que recordar que la implementación del grafo podría tener estos datos en una matriz o en listas de adyacencia. Usaremos la primera versión del algoritmo (sin cola de prioridad para los vértices no visitados).

Si usamos una matriz de pesos M , es:

-	1	2	3	4	5
1	0	5	∞	32	80
2	∞	0	60	∞	∞
3	∞	∞	0	∞	5
4	∞	∞	25	0	75
5	∞	∞	∞	10	0

Se ha colocado el símbolo ∞ en aquellas celdas que corresponden a celdas para las cuales no hay aristas asociadas. Al programar esto en lugar de ∞ se

puede tomar un valor muy grande para que no sea elegido, por ejemplo 9999.

Consideraremos además el vector de vértices visitados, en el cual ya hemos marcado el vértice 1 por ser de salida.

1	2	3	4	5
+	-	-	-	-

Además, tenemos el vector D en el cual vamos a trabajar para obtener los tiempos mínimos. Inicialmente el vector D es:

2	3	4	5
5	∞	32	80

Etapla 1: Se selecciona, de los vértices no visitados, aquel cuyo tiempo es el menor, evaluando los tiempos de D . El vértice con menor tiempo es 2. Se marca entonces 2 como visitado en el vector de marcas. Ahora se analiza si se mejoran los tiempos pasando por 2 como vértice intermedio. Solo el vértice 3 es alcanzable desde 2. Comparamos el valor de $D[3]$, que es ∞ , con $D[2] + M[2][3]$, que es $5 + 60 = 65$. $D[3] > D[2] + M[2][3]$ por lo tanto, reemplazamos en $D[3]$, ∞ por 65. Los vertices 4 y 5 no son alcanzables desde 2, sus valores quedan iguales por ahora. Entonces, finalizada esta etapa, se tiene: Vector de vértices visitados:

1	2	3	4	5
+	+	-	-	-

Vector D:

2	3	4	5
5	65	32	80

Etapla 2: Seleccionamos el vértice no visitado de menor tiempo. Es el 4, con un tiempo de 32. 3 y 5 son alcanzables desde 4. $D[3] = 65$ y $D[4] + M[4][3] = 32 + 25 = 57$, mejora el vértice 3 $D[5] = 80$ y $D[4] + M[4][5] = 32 + 75 = 107$, no mejora el vértice 5. Finalizada esta etapa, se tiene: Vector de vértices visitados:

1	2	3	4	5
+	+	-	+	-

Vector D:

2	3	4	5
5	57	32	80

Etapla 3 : De los vértices no visitados, el de menor tiempo es 3. Desde 3 el único vértice alcanzable es 5. $D[5] = 80$ y $D[3] + M[3][5] = 57 + 5 = 62$, proporcionando una mejora para 5. Vector de vértices visitados:

1	2	3	4	5
+	+	+	+	-
Vector D:				
2	3	4	5	
5	57	32	62	

Etapla 4: Queda solo el vértice 5 por analizar; hechos los cálculos se observa que solo el vértice 4 es alcanzable desde ahí, sin apreciarse mejora alguna.

Entonces, los valores de los caminos mínimos son:

2	3	4	5
5	57	32	62

Variante del algoritmo con listas de adyacencia y cola de prioridades Si se usa una lista de adyacencia para la implementación del grafo y una cola con prioridades para almacenar los nodos aún no elegidos con el coste del camino desde el origen hasta él (esta sería la prioridad), puede mejorarse el coste temporal haciendo una variante más eficiente del algoritmo.

Inicialmente se construye el heap de vértices con sus costes iniciales y se inicializa el vector D . Mientras haya candidatos, se elige al mejor (el primero de la cola con prioridades) y se restaura el heap. Para cada adyacente al elegido (lo cual se realiza usando la correspondiente lista de adyacentes) se analiza si hay una mejora, de haberla se registrar y en ese caso, si el vértice mejorado está en el heap, se modifica su prioridad en el heap y se restaura el mismo. Entonces, quedaría

```

CamMinDijks {
    //Inicializar heap h con todos los vértices del grafo
    //excepto el origen;
    //la prioridad es el peso de la arista (1, i)
    //si existe, o bien un peso máximo.
    para cada vértice i desde 2 hasta N {
        D[i] = coste inicial de alcanzar i desde el vértice 1
        //corresponde al peso de la arista (1, i ),
        //si existe, o bien un peso máximo
    }
    mientras (heap h no vacío) {
        x = h.remover_raiz();
        para cada vértice w adyacente a x {
            D[w] = mín ( D[w], D[x] + M[x,w] )
            si (w fue mejorado) y (w pertenece al heap)
                h.actualizarValor(w);
        }
    }
}

```

}

Análisis del coste temporal: El armado inicial del heap se puede hacer cargando los costos de los vértices ($2..n$) en un array y construyendo un heap sobre él, lo cual puede lograrse en $O(v)$. Mientras el heap no se haya vaciado, se elimina la raíz y se restaura el heap. En total esto se hará v veces, a un costo total $O(v * \log v)$. Para cada adyacente al elegido puede tener que modificarse, en el peor caso, las prioridades de todos los vértices que estén en el heap, el cual deberá ser restaurado. Esto tiene un coste de $O(a * \log v)$.

Ahora bien, hay que tener en cuenta que la operación de determinar si el vértice cuyo costo se modifica está en el heap y en qué posición del heap se encuentra, tiene un coste asociado. Si este coste fuera $O(1)$, entonces, el costo total es: $O(v) + O(v * \log v) + O(a * \log v)$, es decir $O((a + v) \log v)$.

Problema de los caminos más cortos entre todos los pares de vértices

Se trata de determinar los caminos mínimos que unen cada vértice del grafo con todos los otros. Si bien lo más común es que se aplique a grafos que no tengan aristas negativas, la restricción necesaria es que no existan ciclos con costos negativos. Este problema no verifica las condiciones que aseguran que un algoritmo “greedy” funcione, por lo cual no puede aplicarse esa estrategia. Sin embargo, verifica las condiciones que permiten aplicar un algoritmo de los llamados de “programación dinámica”.

La programación dinámica suele aplicarse a problemas de optimización. También considera que a la solución se llega a través de una secuencia de decisiones, las cuales deben verificarla condición de que “en una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima” (lo cual, dicho sea de paso, se verificaba también en la estrategia “greedy”). En estos problemas se realiza una división de problemas en otros problemas menores, pero, a diferencia de lo que ocurría en “Divide y Vencerás”, estos problemas resultado de la división no son independientes entre sí, sino que tienen subproblemas en común, hay un ‘solapamiento’ de problemas. La técnica consiste en resolver y almacenar las soluciones de las zonas solapadas para no volver a realizar los mismos cálculos. En general en esta estrategia se llega a la solución realizando comparaciones y actualizaciones en datos que han sido tabulados (las soluciones de los subproblemas).

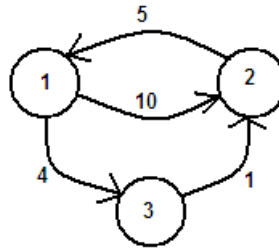


Gráfico 9.5.2: Grafo para aplicar Floyd

Algoritmo de Floyd

Este algoritmo desarrollado en 1962 se basa en una técnica de programación dinámica que almacena en cada iteración el mejor camino entre el que pasa por el nodo intermedio k y el que va directamente del nodo i al nodo j . Para determinar cuál es el mejor camino considera mínimo $(D[v], D[u] + M(u, v))$. Inicialmente se carga una matriz de $N \times N$ con los pesos correspondientes a las aristas, asignándose 0 a $M[i][i]$ para todo i . El método lleva a cabo N iteraciones del proceso en cada una de las cuales se evalúa la eventual mejora de los tiempos por la consideración del vértice i (que variara entre 1 y N a lo largo del proceso) como intermedio.

```

Algoritmo de Floyd
//expresado de manera informal;
//N es el número de vértices
{
    //inicializar A con ceros en la diagonal principal
    //y los pesos indicados por el grafo para cada uno
    //de los restantes elementos.
    //Si una arista no existe, se coloca peso infinito.
    para k desde 1 hasta N
        Para i desde 1 hasta N
            Para j desde 1 hasta N
                A[i][j] = mínimo (A[i][j] , A[i][k] + A[k][j])
}
  
```

El coste temporal de este algoritmo es, obviamente $O(N^3)$.

Ejemplo 9.6. Consideremos el grafo que está en 9.5.2 para aplicar el algoritmo de Floyd. En el gráfico 9.5.3 vemos la aplicación del algoritmo.

Cerradura transitiva

Conjunto de pares de vértices de un grafo orientado cuyas componentes son vértices ligados por algún camino (no importa la longitud del mismo).

Matriz A			
	1	2	3
1	0	10	4
2	5	0	∞
3	∞	1	0

Pasando por el vértice 1, A queda así			
	1	2	3
1	0	10	4
2	5	0	9
3	∞	1	0

Pasando por el vértice 2, A queda así			
	1	2	3
1	0	10	4
2	5	0	9
3	6	1	0

Pasando por el vértice 3, A queda así			
	1	2	3
1	0	5	4
2	5	0	9
3	6	1	0

Gráfico 9.5.3: Algoritmo de Floyd

Algoritmo de Warshall (1962).

(<http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=CC6AAB27A661D231154DC7C8D2C>)

```
//Algoritmo de Warshall:
{
    //inicializar la matriz de adyacencia del grafo
    //colocando 1 en la diagonal principal
    para k desde 1 hasta n {
        para i desde 1 hasta n {
            para j desde 1 hasta n {
                A[i][j] = Maximo ( A[i][j] , A[i][k] * A[k][j]);}
            }
        }
    }
```

Algunos problemas sobre Grafos no dirigidos

Como ya se ha definido, un árbol libre es un grafo no dirigido conexo sin ciclos. Se verifica que en todo árbol libre con N vértices ($N > 1$), el árbol contiene $N - 1$ aristas. Si se agrega una arista a un árbol libre, aparece un ciclo.

9.6. Árbol de expansión de coste mínimo

Arbol libre: Grafo no dirigido conexo sin ciclos. Se verifica que en todo árbol libre con N vértices ($N > 1$), el árbol contiene $N - 1$ aristas.

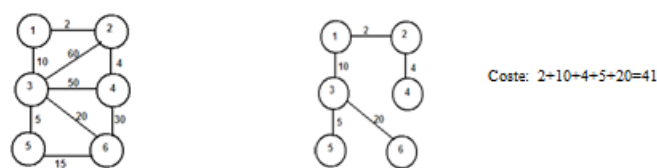


Gráfico 9.6.1: Árbol de expansión de coste mínimo

Árbol abarcador: Dado $G = (V, A)$ grafo no dirigido conexo, un árbol abarcador para G , es un árbol libre que conecta todos los vértices de V .

Arbol Abarcador de Coste Mínimo: Dado un grafo $G = (V, A)$ no dirigido con aristas ponderadas (es decir, que cada arista (v, w) de A tiene un costo asociado, $C(v, w)$), se llama árbol abarcador de coste mínimo al árbol abarcador para G que verifica que la sumatoria de los pesos de las aristas es mínima. (El árbol abarcador de costo mínimo, puede no ser único).

Por ejemplo: para el grafo que está en la izquierda de la figura 9.6.1, un árbol abarcador de coste mínimo es el de la derecha.

Algoritmos que permiten obtener el arbol de expansion de coste minimo

Para un grafo dado existen varios algoritmos que permiten obtener el árbol de expansión de coste mínimo. Se detallarán dos de ellos: el algoritmo de Prim y el de Kruskal. En ambos casos se aplica la estrategia ‘greedy’ o voraz para alcanzar el óptimo.

Algoritmo de Prim Desarrollado por Prim en 1957 sobre un algoritmo previo de Jarnik de 1930. (<https://archive.org/details/bstj36-6-1389>) Usa una estrategia greedy. Las aristas deben ser no negativas.

Conjunto de candidatos: conjunto de vértices del grafo aún no incorporados al árbol (cualquiera de ellos se elige como inicial)

Función de selección: elección del vértice que aún no está en el árbol para el cual existe alguna arista de peso mínimo que lo enlaza con algún vértice del árbol.

Función a optimizar: sumatoria de pesos de aristas del árbol

Criterio de finalización: todos los vértices están en el árbol.

Esquema de Prim: Llamando G al grafo y A al conjunto de aristas del árbol abarcador de coste mínimo, es

{

```

A se inicializa en vacío U = {1}
//se almacena vértice inicial
mientras U distinto de V {
    //determinar la arista de costo mínimo, (u,v)
    //tal que u pert. U y v pert. (V-U)
    //agregar a A la arista (u, v)
    //agregar a U el vértice v
}
}

```

Algoritmo de Kruskal

Desarrollado por Kruskal en 1956. Usa una estrategia greedy. Las aristas deben ser no negativas.

Conjunto de candidatos: conjunto de aristas del grafo.

Función de selección: elección de la arista de coste mínimo.

Restricción: la arista no debe formar ciclo.

Función a optimizar: sumatoria de pesos de aristas del árbol

Criterio de finalización: el árbol tiene $n-1$ aristas.

Esquema del algoritmo:

```

//G.(V, A) grafo no dirigido con aristas no negativas
//Consideraremos los vértices etiquetados como 1..n.
//Se comenzará incorporando al vértice 1 (arbitrariamente)
//E: conjunto de aristas del árbol abarcador;
//U: conjunto de vértices del árbol
{
    ordenar aristas de forma creciente
    inicializar E en vacío
    inicializar U en vacío
    mientras #E < n-1 {
        elegir arista (u,v) de peso mínimo que no forme ciclo
        E = E U {(u,v)}
        incorporar al conjunto U el vértice u , o el V, según corresponda
    }
}

```

Ejemplo: para el grafo considerado en 9.6.2.

¿Cómo chequear si cada nueva arista considerada (u,v) forma ciclo?

u y v en la misma componente \Rightarrow incorporar (u,v) genera ciclo. Se comienza con una componente para cada vértice. Las componentes se unen al añadir una arista. Función que retorna componente de un vértice v : Find(v)
 Función que une la componente de u y la componente de v : Union(u , v). (Se demuestra que ejecutar k uniones con n elementos tiene un coste temporal $k \log(n)$, ya que en k uniones hay $2*k$ elementos involucrados)

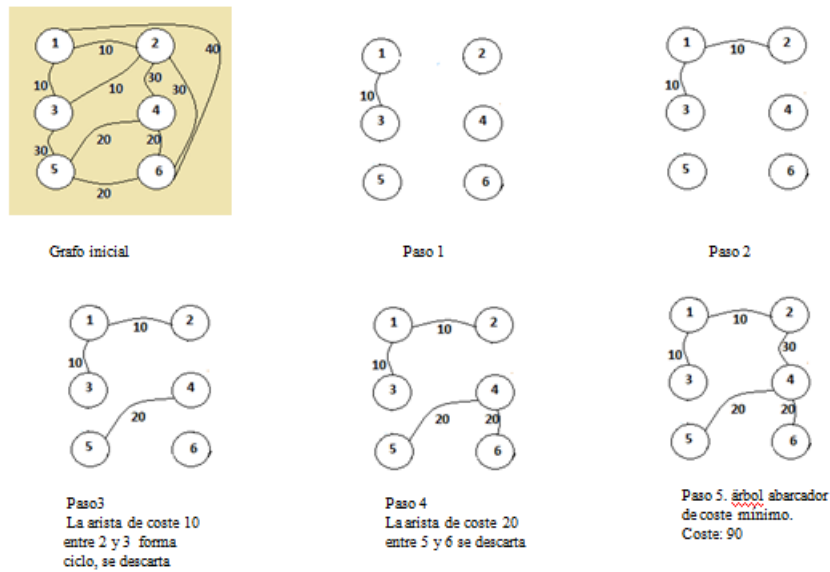


Gráfico 9.6.2: Algoritmo de Kruskal

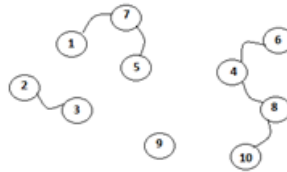


Gráfico 9.6.3: Union y find

```

Kruskal
//versión con Union y Find para un grafo de n vértices
Inicializar A conjunto de aristas del árbol abarcador en vacío
Ordenar S conjunto de aristas del grafo de forma creciente
Para cada vértice v {MakeSet (v);}
Mientras (#A < n-1) {
    Elegir arista (u, v) de peso mínimo y retirarla de S
    si (Find (u) != Find (v)) {
        Union (Set(u) , Set(v));
        A = A U {(u,v)};
    }
}

```

Posibles implementaciones para Union y Find: Las diferentes formas de implementar las funciones Union y Find influyen en el costo del algoritmo de Kruskal. Ejemplo, para el siguiente grafo,

Puede plantearse una implementación de este tipo,

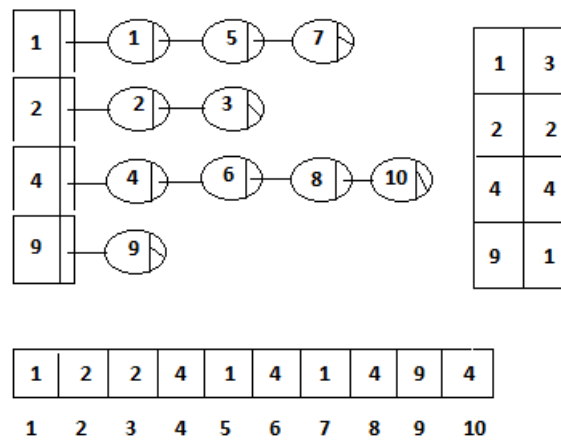


Gráfico 9.6.4: Implementación union y find

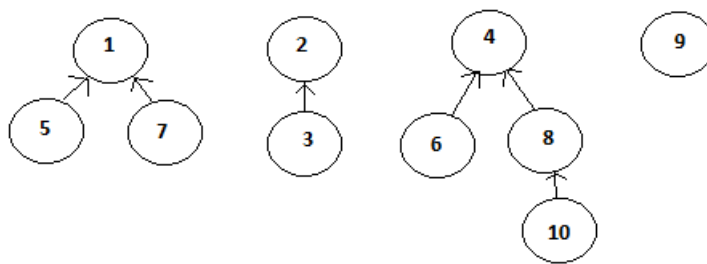


Gráfico 9.6.5: Union y find, otra implementación

En esta implementación, para las componentes hay un array de listas. El menor valor es representante de la lista. Para unir se elige la lista menor que se incorpora a la otra. Para determinar los tamaños, hay una tabla con los mismos (a la derecha) El array inferior permite implementar Find con $O(1)$. Otra posibilidad es

En esta implementación, para las componentes hay estructuras arborescentes “invertidas”. En cada una, cada nodo, excepto uno, apunta a su padre. El que está en el lugar de la raíz es el representante de la componente.

Ejercicios

- Determinar las componentes conexas del siguiente grafo.
- Determinar los puntos de articulación del siguiente grafo
- Desarrollar un algoritmo que, para un par de vértices (x, y) , determine

la mínima cantidad de aristas del camino que enlace x con y (si existe).

- d. Diseñar un algoritmo que permita determinar, para un grafo (dirigido o no) y con ciclos, la cantidad de vértices del camino más largo contenido en el grafo.
- e. Dado un grafo dirigido y acíclico con pesos no negativos en las aristas, se quiere determinar la distancia máxima desde un vértice origen a cada uno de los otros. ¿Se puede diseñar un algoritmo similar al de Dijkstra eligiendo al candidato de coste mayor en cada etapa?
- f. Modificar el algoritmo de Floyd para calcular el número de caminos con distancia mínima que hay entre cada par de nodos. ¿Cómo queda el coste del algoritmo?
- g. Mostrar mediante un contraejemplo que el algoritmo de Dijkstra no sirve si el grafo tiene alguna arista negativa.
- h. Analizar el coste del algoritmo de Kruskal con las estructuras sugeridas para implementar Union y Find.
- i. ¿Cuál de los dos algoritmos (Prim y Kruskal) conviene más si la densidad de aristas es alta?
- j. Analizar la validez, y en ese caso el coste del siguiente algoritmo para determinar las componentes conexas de un grafo no dirigido: Para cada vértice v del grafo {MakeSet (v);} Para cada arista de (x,y) perteneciente a A hacer { Si (Find(x) == Find (y)) { Union (x, y); } }
¿Cuántas veces se ejecuta Union? ¿Y Find?
- k. Se tienen los registros de todos los tramos posibles de viajes que realiza una agencia de turismo. Cada tramo especifica la ciudades de origen y de destino, y para todo tramo (x, y) existe el tramo (y,x) . Se necesita resolver lo siguiente: dados los pares (origen x , destino y), ¿cuál es el mínimo número de transbordos que debe realizarse? Diseñar un algoritmo apropiado y evaluar su coste.

Parte V

Temas complementarios

Capítulo 10

Hashing

10.1. Introducción

Supongamos que tenemos que administrar una cochera donde entran 500 autos. En una primera reflexión pensaríamos en ir guardando en un vector o un archivo los registros de cada uno (patente, marca del auto, horario de ingreso, etc), en el orden que ingresen. Sin embargo, una búsqueda en este vector o archivo debería ser secuencial, por lo que tendría $O(N)$. En este caso 500.

Para hacer más eficiente la búsqueda podríamos ordenar este vector, por ejemplo por patente, y hacer una búsqueda binaria. De esta forma bajaríamos el orden a $\log_2(N)$. En este caso 9.

Pero si el tiempo de búsqueda es muy crítico y quisiéramos llevarlo a $O(1)$, es decir, obtener el registro en el primer intento, lo que se nos ocurriría es hacer coincidir la posición del registro con el propio valor de la clave. Por ejemplo, la patente AAA000 iría en la posición 1, la patente AAA001 en la 2, etc. Nota: estamos asumiendo las patentes viejas para no complicar el ejercicio ya que en la realidad tendríamos dos formatos distintos de patentes.

El problema de generar un vector de estas características es que necesitaríamos un vector de $26 \times 26 \times 26 \times 10 \times 10 \times 10 = 17576000$ posiciones, lo cual no tendría sentido para guardar solo 500 datos.

En estos casos intervienen las funciones de hashing. Lo que hacen es llevar esta clave a una nueva, más manejable. Por ejemplo, una función de hashing para la cochera podría ser quitar la parte de las letras de las patentes y quedarse solo con la parte numérica. Entonces tendríamos un vector de 1000 posiciones (ya mucho más coherente) en donde la patente MNK025 iría en la posición 25 y la patente IXY162 iría en la 162. El problema es que si viene un auto con la patente KHH162 debería ocupar el mismo registro que

el anterior. Cuando sucede esto se dice que hay una colisión.

Las técnicas de hashing tratan básicamente sobre las diferentes funciones de generación de claves y cómo resolver el problema de las colisiones.

10.2. Funciones de hashing

Una función de hash o dispersión toma la clave del dato y devuelve un valor que será el índice de entrada de la tabla. Entonces, si k es una clave y p es una posición o entrada de la tabla decimos que:

$$\begin{aligned} h : K &\rightarrow P \\ p &= h(k) \end{aligned}$$

Las buenas funciones de dispersión son las que producen resultados uniformes evitando en primer lugar colisiones y en segundo lugar agrupamientos en ciertas zonas de la tabla.

Un primer problema que debemos definir es el tamaño de la tabla. Si conociéramos la cantidad de claves a guardar o tenemos un número aproximado, digamos n , el tamaño de la tabla m debería ser tal que la proporción entre n y m sea aproximadamente 0.8. Esta proporción se llama *factor de carga* λ . Entonces:

$$\lambda = n/m$$

y pedimos:

$$\lambda \leq 0,8$$

Un valor más chico de λ produciría una tabla con mucho desperdicio, es decir, con demasiados lugares vacíos. Un valor más grande generaría muchas colisiones por lo que perderíamos eficiencia en cuanto a los tiempos de búsqueda, inserción y borrado.

Un segundo problema que debemos abordar es que como los índices de una tabla son números naturales más el 0: $I = \{0, 1, 2, 3, \dots\}$ si nuestras claves no son de tipo numéricas enteras debemos convertirlas a un formato de ese tipo. Por ejemplo, si son letras, podríamos pasarlas a un valor entero con alguna convención, como tomar sus valores ASCII y sumarlos.

- “ab” = $97 + 98 = 195$
- Sin embargo, es mejor tomar estos valores en cierta base. Como los códigos ASCII se representan con 128 posiciones se aconseja aplicar esta base para dichos códigos, de esta manera
 $ab = 97 \cdot 128 + 98 = 12416 + 12514.$

Para los métodos que veremos a continuación asumiremos que las claves son números naturales no muy pequeños porque de serlo no tendría sentido utilizar funciones hashing.

Division

El método de la división o módulo es la función de dispersión más simple. Toma el resto de la división entera entre el valor k de la clave y cierto t que será el tamaño de la tabla. Entonces:

$$\begin{aligned}p &= h(k) \\p &= k \% t\end{aligned}$$

El operador resto nos genera valores que van en el rango $0..t - 1$. Con la finalidad de lograr la mayor dispersión posible se recomienda que el valor de t , que es el tamaño de la tabla, sea un número primo.

Ejemplo 10.1. La cantidad de claves a almacenar es aproximadamente 5000. Teniendo en cuenta un factor de carga de 0.8 deberíamos tener un tamaño de la tabla aproximado a

$$\begin{aligned}t &= 5000/0,8 \\t &= 6250\end{aligned}$$

Por lo tanto buscamos un número primo cercano a 6250, preferentemente superior. El número primo superior más cercano es 6257. Entonces, el tamaño de la tabla será 6257, los índices irán desde 0 hasta 6256. La función de hash a utilizar será:

$$p = k \% 6257$$

Entonces, si necesitamos ingresar a la tabla la clave 113521 deberá ir a la posición:

$$\begin{aligned}p &= 113521 \% 6257 \\p &= 895\end{aligned}$$

Y la clave 28413 debe ir a la posición:

$$\begin{aligned}p &= 28413 \% 6257 \\p &= 3385\end{aligned}$$

Como vemos, no conserva un orden en que las claves más chicas van en las posiciones inferiores de la tabla y las más grandes en las superiores. Si luego debemos ingresar el dato con la clave 44694 debemos colocarlo en:

$$p = 44694 \% 6257$$

$$p = 895$$

que es la misma posición que la clave 113521 ya ingresada. En estos casos se produce una colisión. Elegir un t que sea un número primo no garantiza que las colisiones no se produzcan pero las minimiza. Luego de ver las distintas funciones de dispersión veremos cómo tratar las colisiones.

La función división o módulo se puede utilizar sola o, como veremos, las demás funciones terminan aplicando esta función para que los valores no excedan el tamaño de la tabla.

Folding

En esta función de dispersión las claves se dividen en varias partes las cuales luego se suman. Por ejemplo, el número de CUIT 23-31562313-7 podemos dividirlo en 4 partes tomando de a 3 dígitos: 233 - 156 - 231 - 37, luego estos valores los sumamos: $233 + 156 + 231 + 37 = 657$. Finalmente, si el tamaño de la tabla es menor al número obtenido se aplica la función módulo.

En el caso de claves de tipo string se puede hacer algún corte obteniendo strings más pequeños para realizar un *xor* entre ellos tomando sus valores ASCII.

Ejemplo 10.2. Se tiene la cadena $s = "abcd"$, la cortamos en dos cadenas: $s1 = "ab"$ y $s2 = "cd"$, tomamos los valores ASCII de cada una, el valor de a es 97, el de b , 98, etc. Por lo tanto, tomamos los bits correspondientes y hacemos un *xor*. Debemos recordar que el *xor* es el *o* exclusivo: pone el bit en 1 si alguno de los bits está en 1 pero no ambos. De esta manera:

posición/cadena	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ab	0	1	1	0	0	0	0	1	0	1	1	0	0	0	1	0
cd	0	1	1	0	0	0	1	1	0	1	1	0	0	1	0	0
xor	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0

El resultado anterior nos da: $512 + 4 + 2 = 518$. Por último se puede tomar módulo con el tamaño de la tabla.

Mid - square

Esta función toma la clave, la eleva al cuadrado, y luego se queda con los dígitos centrales, dado que son los que presentan una variación más uniforme.

Ejemplo 10.3. La clave es 1536, entonces se hace $1536^2 = 2359296$, por último nos quedamos con los dígitos centrales: 592. Siempre se puede aplicar si fuera necesario la función módulo.

Extraction

En la función extracción el valor se forma extrayendo alguna parte de la clave. Si utilizamos el mismo ejemplo del número de CUIT 23-31562313-7 nos podemos quedar con los 4 primeros dígitos: 2331, con los últimos 4: 3137, una combinación tomando los dos primeros con los dos últimos: 2337, etc.

En esta función hay que ser cuidadosos con los dígitos a tomar porque en los ejemplos de números de CUIT los valores siempre comienzan con 20, 23, 24 y 27, si son personas físicas. De esta manera corremos el riesgo de que se produzcan muchas colisiones.

Radix transformation

Esta función toma la clave y la cambia de base. Por ejemplo, si la clave es 425, lo asume como que está expresado en base 16, por lo tanto, la nueva clave será

$$k' = 4 \cdot 16^2 + 2 \cdot 16 + 5 = 1061$$

Finalmente, se puede aplicar la función módulo como en los demás casos.

10.3. Colisiones

Las colisiones, como hemos mencionado, se producen cuando más de una clave nos devuelve el mismo valor luego de aplicarle la o las funciones de dispersión. La cantidad de colisiones depende de la función de hash aplicada y también del tamaño de la tabla. Según el tratamiento de las colisiones este problema no solo afectará el ingreso de un nuevo dato, sino que también puede afectar la eliminación y la búsqueda.

A continuación veremos los métodos más utilizados para el tratamiento de las colisiones.

Open addressing

En el direccionamiento abierto toda la información se guarda en la misma tabla. Si se produce una colisión se busca una nueva posición tomando alguna nueva función que prueba en primer lugar con el valor 1, luego con el 2, etc. De esta forma, si $h(k)$ está ocupada prueba con $h(k) + p(1)$, y si también lo

está intenta con $h(k) + p(2)$, etc, donde p es una nueva función. Al finalizar siempre se aplica la función módulo. Dentro del direccionamiento abierto tenemos distintas variantes dependiendo de esta nueva función p .

Linear probing Linear probing o sondeo lineal es la decisión más simple del direccionamiento abierto. Se define la función p como $p(i) = i$ por lo tanto, lo que hace es ir verificando las posiciones siguientes a la que la función de hash indica. Por ejemplo, si $h(k) = 132$ y esta posición no está libre, se intenta en la posición 133, luego en la 134, etc. Este procedimiento se repite hasta alcanzar el final de la tabla, en cuyo caso se comienza desde el principio hasta encontrar la primera posición libre.

Quadratic probing El problema con un sondeo lineal es que agrupa claves en ciertas zonas de la tabla. Una mejora es tomar la función p como cuadrática. En este caso $p(i) = i^2$. Por lo tanto, en el mismo ejemplo anterior, si $h(k) = 132$ se encuentra ocupada, intenta

$$h(k) + p(1) = 132 + 1^2 = 133$$

es decir, en el siguiente lugar, al igual que el lineal. Pero si esta posición está también ocupada, intenta con

$$h(k) + p(2) = 132 + 2^2 = 136$$

en lugar de 134. Estos “saltos” hacen que los datos queden de forma más dispersa y no se agrupen en ciertas zonas.

Double hashing El doble hashing es la mejor solución para el direccionamiento abierto. La secuencia hasta encontrar una posición vacía es la siguiente:

$$p = h(k) + i \cdot h_2(k)$$

con $i = 0, 1, 2, \dots$

El caso en el que i vale 0 es simplemente $h(k)$, la primera prueba. Si se produce una colisión se intenta sumándole una nueva función de dispersión, en el caso de seguir colisionando se sumará el doble de dicha función, etc. Siempre se debe finalizar tomándole módulo con el tamaño de la tabla.

Ejemplo 10.4. Supongamos que la función $h(k)$ toma el módulo de la clave con cierto valor t (tamaño de la tabla), digamos que $t = 1019$. Entonces, podemos elegir que la segunda función de hash también sea módulo pero

con un valor distinto, por ejemplo, 1018. De esta forma, si tenemos la clave 15924, $h(15924) = 639$. En caso de tener ocupada esa posición hacemos:

$$\begin{aligned} p &= h(15924) + h_2(15924) \\ &= 639 + 654 \\ &= 1293 \end{aligned}$$

A este valor volvemos a aplicarle módulo con 1019 y queda 274. Si tuviéramos una nueva colisión, los cálculos serían:

$$\begin{aligned} p &= h(15924) + 2 \cdot h_2(15924) \\ &= 639 + 2 \cdot 654 \\ &= 639 + 1308 \\ &= 1947 \end{aligned}$$

Valor que al aplicarle el módulo con 1019 nos queda 928. Como vemos, este tipo de redireccionamiento dispersa muchísimo más las claves que los métodos anteriores. Esto es ventajoso porque una buena dispersión evita colisiones, lo cual redundará en tiempos de búsqueda, inserción y borrado.

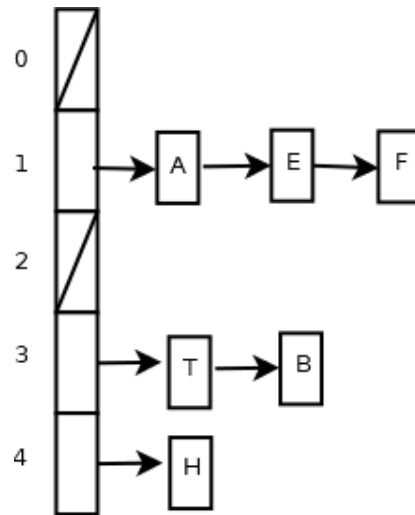
Chaining

En español es encadenamiento. Este método contempla que las claves no tienen por qué estar en la misma tabla, por lo que cada posición de la tabla en realidad será un puntero a una lista enlazada con las claves. Por ejemplo, supongamos que las claves A, E y F devuelven el valor 1 luego de aplicarles la función de hash, las claves B y T, el valor 3 y H el valor 4. Entonces, si las claves ingresan en el siguiente orden: A, H, E, T, F y B, la tabla queda como apreciamos en el gráfico 10.3.1.

Bucket addressing

En español sería “direccionamiento de cubos”. La idea es guardar los datos en la misma tabla pero en cada posición, en lugar de tener solo un lugar podríamos tener varias posiciones. Esto se puede hacer perfectamente con una matriz. Para el ejemplo visto en *chaining* la resolución con buckets de 4 lugares sería como se ve en el gráfico 10.3.2.

El problema con este tipo de soluciones es que una gran cantidad de *buckets* genera mucho desperdicio de memoria, y una cantidad muy acotada puede, a su vez, agotar un *bucket* fácilmente. Por lo cual habría que determinar qué decisión tomar en estos casos. Las variantes son utilizar *open addressing* (ir

Gráfico 10.3.1: Resolución de colisiones con *chaining*

	0	1	2	3
0				
1	A	E	F	
2				
3	T	B		
4	H			

Gráfico 10.3.2: Resolución de colisiones con *bucket*

a la siguiente posición), también se puede usar memoria adicional mediante un puntero a un *bucket* dinámico para estos casos.

10.4. Borrado de un elemento

El borrado de un elemento no es tan sencillo como se podría pensar en un primer momento. En primer lugar hay que ubicar al elemento, es decir, se realiza una búsqueda. Esta búsqueda dependerá de la forma en que se resolvieron las colisiones. Supongamos que se desea borrar a la clave k , entonces, si en $h(k)$ hay otro elemento deberíamos aplicar el método de resolución de colisiones. Supongamos que se resolvió con *open addressing* lineal, entonces debemos revisar $h(k) + 1$, $h(k) + 2$, etc, hasta encontrar la clave o determinar que dicha clave no está en la tabla. Pero imaginemos que la clave está en $h(k) + 1$, no podemos simplemente borrar la clave ya que si tuviéramos otra clave en $h(k) + 2$ producto de una colisión en $h(k)$ jamás la encontraríamos, dado que al detectar un lugar libre dejaríamos de buscar.

Por lo tanto, si utilizamos un método de *open addressing* para la resolución de colisiones debemos dejar algún tipo de marca en la celda para indicar que dicho lugar estuvo anteriormente ocupado y, que en caso, de buscar otra clave, la búsqueda no debe detenerse. Para esta marca alcanza con un campo adicional de tipo *boolean* que indique falso si nunca fue ocupado o verdadero si lo fue.

10.5. Funciones de hash perfectas

Una función de hash que no produce colisiones es una función de hash perfecta. Por supuesto, si las claves no son conocidas no podemos garantizar que esto suceda porque desconocemos si alguna futura clave pueda llegar a producir una colisión. Sin embargo, en los problemas en que las claves son conocidas de antemano podemos conseguir plantear una función de dispersión sin colisiones.

Ejemplos de situaciones en las que conocemos de antemano las claves pueden ser palabras reservadas de un compilador, archivos en un disco no regrabable, diccionarios, etc.

Las funciones de hash perfectas logran un tiempo de búsqueda óptimo: $O(1)$. Si una función, además de no producir colisiones, tiene una tabla de tamaño n en donde se almacenarán n datos, decimos que es una función de hash perfecta mínima. Es decir, no hay desperdicio de tiempo en la búsqueda ni desperdicio de memoria.

Aún conociendo de antemano las claves, lograr una función de hash perfecta no es sencillo. Está el método de Cichelli y el algoritmo FHCD que determinan formas de hacerlo pero no los veremos en este curso.

10.6. Funciones de hash para archivos extensibles

Los métodos vistos hasta el momento manejan tablas de tamaño estático. La realidad es que muchas tablas de hash son índices de entrada para archivos y estos archivos pueden crecer al agregarse nuevos elementos. Veremos dos métodos de hashing extensible.

Extendible hashing

Linear hashing

Apéndice A

Código

A.1. Implementación y uso de Vector con templates

Listado de código A.1: Archivo vector.h con templates

```
#ifndef VECTOR_H_INCLUDED
#define VECTOR_H_INCLUDED

template <typename Dato>
class Vector {
private:
    // Atributos
    unsigned longitud;
    Dato* datos;

public:
    // Metodos
    // Constructor sin parametros
    Vector();

    // Constructor con parametros
    Vector(unsigned largo);

    // Constructor de copia
    Vector(const Vector & vec);

    // Destructor
    ~Vector();

    // insertar (la primera posicion es la 1)
    void insertar(Dato d, unsigned pos);
```

```
// eliminar (la primera posicion es la 1)
void eliminar(unsigned pos);

// devuelve la longitud del vector
unsigned tamano();

// devuelve el dato que esta en pos
Dato elemento(unsigned pos);

// redimensiona al vector
void redimensionar(unsigned largo);

private:

    // Copia los datos del vector pasado por parametro desde d hasta h
    // Pre: 0 < d <= h <= longitud del vector
    // Post: en datos coloca los valores del vector
    void copiar(Dato* vec, unsigned d, unsigned h);
};

// Constructor sin parametros
template <typename Dato>
Vector<Dato>::Vector() {
    longitud = 0;
    datos = 0;
}

// Constructor con parametros
template <typename Dato>
Vector<Dato>::Vector(unsigned largo) {
    longitud = largo;
    datos = new Dato[largo];
    //anular(1, largo);
}

// Constructor de copia
template <typename Dato>
Vector<Dato>::Vector(const Vector & vec) {
    longitud = vec.longitud;
    if (longitud > 0) {
        datos = new Dato[longitud];
        copiar(vec.datos, 1, longitud);
    }
    else
        datos = 0;
}
```

A.1. IMPLEMENTACIÓN Y USO DE VECTOR CON TEMPLATES 277

```
// Destructor
template <typename Dato>
Vector<Dato>::~~Vector() {
    if (longitud > 0)
        delete [] datos;
}

// insertar (la primera posicion es la 1)
template <typename Dato>
void Vector<Dato>::insertar(Dato d, unsigned pos) {
    datos[pos-1] = d;
}

// eliminar (la primera posicion es la 1)
template <typename Dato>
void Vector<Dato>::eliminar(unsigned pos) {
    datos[pos-1] = 0;
}

// devuelve la longitud del vector
template <typename Dato>
unsigned Vector<Dato>::tamano() {
    return longitud;
}

// devuelve el dato que esta en pos
template <typename Dato>
Dato Vector<Dato>::elemento(unsigned pos) {
    return datos[pos-1];
}

// redimensiona al vector
template <typename Dato>
void Vector<Dato>::redimensionar(unsigned largo) {
    if (largo != longitud) {
        Dato* borrar = datos;
        datos = new Dato[largo];
        copiar(borrar, 1, largo);
        delete [] borrar;
        longitud = largo;
    }
}

// Copia los datos del vector pasado por parametro desde d hasta h
template <typename Dato>
void Vector<Dato>::copiar(Dato* vec, unsigned d, unsigned h) {
    for (unsigned i = d-1; i < h; i++)
```

```

        datos[i] = vec[i];
    }

#endif // VECTOR_H_INCLUDED

```

Listado de código A.2: Uso clase Vector con templates

```

// Uso de clase Vector con templates
#include <iostream>
#include <string>
#include "vector.h"

using namespace std;

int main()
{
    cout << "==== Vector de enteros ====" << endl;

    Vector<int> vi1(3);

    vi1.insertar(7, 1);
    vi1.insertar(8, 2);
    vi1.insertar(9, 3);

    for (unsigned i = 1; i <= vi1.tamano(); i++)
        cout << vi1.elemento(i) << endl;

    cout << "==== Otro vector de enteros ====" << endl;

    Vector<int> vi2(2);

    vi2.insertar(100, 1);
    vi2.insertar(150, 2);

    for (unsigned i = 1; i <= vi2.tamano(); i++)
        cout << vi2.elemento(i) << endl;

    // Ahora, utilizamos un vector con otros
    // tipos de datos, como strings
    cout << "==== Vector de strings ====" << endl;

    string s1 = "siete", s2 = "ocho", s3 = "nueve", s4 = "diez";
    Vector<string> vs(4);

    vs.insertar(s1, 1);
    vs.insertar(s2, 2);
    vs.insertar(s3, 3);

```

```

        vs.insertar(s4, 4);

        for (unsigned i = 1; i <= vs.tamano(); i++)
            cout << vs.elemento(i) << endl;

        return 0;
}

```

A.2. Implementación y uso pila estática

Listado de código A.3: Archivo pila.h

```

#ifndef PILA_ESTATICA_H_INCLUDED
#define PILA_ESTATICA_H_INCLUDED

// Tipo de dato que contiene la pila
typedef char Dato;

// Tamano maximo de la pila
const int MAXIMO = 10;

/** Clase PilaEstatica
    Implementada con un vector de elementos
    y un tamano fijo (MAXIMO)
*/

class PilaEstatica {

private:
    // Vector donde se iran agregando los elementos
    Dato pila[MAXIMO];
    // Tamano logico de la pila
    unsigned tope;

public:
    // Constructor
    // PRE: ninguna
    // POST: crea una pila vacía con tope = 0
    PilaEstatica( );

    // Destructor
    // PRE: la pila fue creada
    // POST: nada (no tiene código ya que es estática)
    ~PilaEstatica( );

    // Agrega un dato
    // PRE: d es un Dato valido y hay lugar en la pila

```

```

// POST: Si la pila no esta llena
// - se inserta al final el dato d
// - tope se incrementa en 1
// Si la lista esta llena no hace nada
void agregar(Dato d);

// Indica si la pila está vacía o no
// PRE: lista creada
// POST: Devuelve TRUE si la pila esta vacia
//       Si no devuelve FALSE
bool pilaVacía( );

// Indica si la pila esta llena o no
// PRE: pila creada
// POST: Devuelve TRUE si la pila esta llena
//       Si no devuelve FALSE
bool pilaLlena( );

// Devuelve el dato que esta en tope
// PRE: - no vacia
// POST: devuelve el dato que esta en tope
Dato obtener( );

// Sacar un dato
// PRE: - pila creada y no vacia
// POST: - se quita el dato que esta en tope y se devuelve
//       - tope se decrementa en 1
Dato sacar( );

// Devuelve el tamaño logico de la Pila
// PRE: pila creada
// POST: devuelve el valor de tope
unsigned obtenerTope();
};
// =====

// Invariantes
// x = sacar(agregar(x))

#endif // LISTA_ESTATICA_H_INCLUDED

```

Listado de código A.4: Archivo de implementacion pilaEstatica

```

// Implementacion clase ListaEstatica
#include "pila_estatica.h"

// Constructor

```



```

PilaEstatica::PilaEstatica() {
    tope = 0;
}

// Destructor
PilaEstatica::~~PilaEstatica() { }

// PilaLlena
bool PilaEstatica::pilaLlena() {
    return (tope == MAXIMO);
}

// listaVacía
bool PilaEstatica::pilaVacía() {
    return (tope == 0);
}

// agregar un dato
void PilaEstatica::agregar(Dato d) {
    if (!(this->pilaLlena()))
        pila[tope++] = d;
}

// obtener un dato
Dato PilaEstatica::obtener( ) {
    return pila[tope - 1];
}

// sacar el dato que esta encima
Dato PilaEstatica::sacar( ) {
    tope--;
    return pila[tope];
}

// obtener el tope o maximo logico
unsigned PilaEstatica::obtenerTope() {
    return tope;
}

```

Listado de código A.5: Archivo de uso clase pilaEstatica

```

// Ejemplo de uso de la clase ListaEstatica
#include <iostream>
#include "pila_estatica.h"

using namespace std;

int main()

```

```

{
    PilaEstatica pila;
    pila.agregar('A');
    pila.agregar('H');
    pila.agregar('E');

    // Muestra los elementos
    while (pila.obtenerTope() > 0) {
        cout << pila.sacar() << endl;
    }

    return 0;
}

```

A.3. Implementación dinámica y uso de Pila

Listado de código A.6: Archivo nodo.h

```

#ifndef NODO_H_INCLUDED
#define NODO_H_INCLUDED

#include <iostream>

// Tipo de dato char
typedef char Dato;

class Nodo
{
private:
    Dato dato; // Dato a almacenar
    Nodo* psig; // Puntero a otro nodo

public:
    // Constructor con parametro
    // PRE: Ninguna
    // POST: Crea un nodo con el dato d
    // y el puntero a NULL
    Nodo(Dato d);

    // Destructor
    // PRE: Nodo creado
    // POST: No hace nada
    ~Nodo();

    // Setea el dato (lo cambia)
    // PRE: el nodo tiene que estar creado
    // POST: El nodo queda con el dato d
    void asignarDato(Dato d);

```

```

    // Obtener el dato
    // PRE: nodo creado
    // POST: devuelve el dato que contiene el nodo
    Dato obtenerDato();

    // Setear el puntero al siguiente nodo
    // PRE: nodo creado
    // POST: el puntero al siguiente nodo apuntará a ps
    void asignarSiguiente(Nodo* ps);

    // Obtener el puntero al nodo siguiente
    // PRE: nodo creado
    // POST: Devuelve el puntero al siguiente nodo
    // Si es el último devuelve NULL
    Nodo* obtenerSiguiente();
};

#endif // NODO_H_INCLUDED

```

Listado de código A.7: Archivo nodo.cpp

```

#include "nodo.h"

// Constructor
Nodo::Nodo(Dato d) {
    std::cout << "Se construye el nodo ";
    std::cout << std::hex << (unsigned)this << std::endl;
    dato = d;
    psig = 0;
}

// Destructor
Nodo::~~Nodo() {
    std::cout << "Se destruye el nodo ";
    std::cout << std::hex << (unsigned)this << std::endl;
}

// Cambia el dato
void Nodo::asignarDato(Dato d) {
    dato = d;
}

// Obtiene el dato
Dato Nodo::obtenerDato() {
    return dato;
}

```

```
// cambia el puntero
void Nodo::asignarSiguiente(Nodo* ps) {
    psig = ps;
}

// obtiene el puntero
Nodo* Nodo::obtenerSiguiente() {
    return psig;
}
```

Listado de código A.8: Archivo pila.h

```
#ifndef PILA_H_INCLUDED
#define PILA_H_INCLUDED

#include "nodo.h"

class Pila
{
private:
    // Primer elemento de la pila
    Nodo* ultimo;

public:
    // Constructor
    // PRE: ninguna
    // POST: construye una pila vacía
    // - primero apunta a nulo
    Pila();

    // Destructor
    // PRE: pila creada
    // POST: Libera todos los recursos de la pila
    ~Pila();

    // ¿La pila es vacía?
    // PRE: pila creada
    // POST: devuelve verdadero si la pila es vacía
    // falso de lo contrario
    bool pilaVacía();

    // Agregar un elemento a la pila
    // PRE: pila creada
    // POST: agrega un dato (dentro de un nodo) al principio
    void agregar(Dato d);

    // Obtener el dato que está en la cima
```

```

    // PRE: - pila creada y no vacía
    // POST: devuelve el dato que está en la cima
    Dato obtenerDato();

    // Borrado del nodo que está en la cima
    // PRE: - pila creada y no vacía
    // POST: libera el nodo que está en la cima
    Dato sacar();
};

#endif // PILA_H_INCLUDED

```

Listado de código A.9: Archivo pila.cpp

```

#include "pila.h"

// Constructor
Pila::Pila() {
    ultimo = 0;
}

// Destructor
Pila::~~Pila() {
    while ( ! (pilaVacía()) )
        sacar();
}

// Pila vacía?
bool Pila::pilaVacía() {
    return (ultimo == 0);
}

// Agregar dato
void Pila::agregar(Dato d) {
    Nodo* pnodo = new Nodo(d);
    pnodo->asignarSiguiente(ultimo);
    ultimo = pnodo;
}

// Obtener el dato
Dato Pila::obtenerDato() {
    return ultimo->obtenerDato();
}

// Sacar dato
Dato Pila::sacar() {
    Nodo* paux = ultimo;
    ultimo = paux->obtenerSiguiente();
}

```

```

        Dato d = paux->obtenerDato();
        delete paux;
        return d;
    }

```

Listado de código A.10: Archivo uso estructura Pila

```

// Uso estructura Pila
#include "pila.h"

using namespace std;

int main()
{
    Pila p;
    p.agregar('A');
    p.agregar('H');
    p.agregar('B');
    while ( ! p.pilaVacía() ) {
        cout << p.sacar() << endl;
    }

    return 0;
}

```

Comentarios

- Se utilizó un char como dato para concentrarnos en la implementación y no estar pendiente de otras cosas.
- El puntero ultimo siempre apunta a la cima de la pila. Como el ingreso es al final, por una comodidad de implementación se toma que el final es el primer elemento de la Lista.
- Se utilizaron carteles en los constructores y destructores para indicar cuándo se construye un nodo y cuándo se destruye, además de sus direcciones. Obviamente, estos carteles deben ser borrados pero es una buena práctica utilizarlos en las pruebas, con el fin de controlar la memoria pedida y liberada.

A.4. Implementación dinámica y uso de Cola

Listado de código A.11: Archivo cola.h

```

#ifndef COLA_H_INCLUDED

```

```
#define COLA_H_INCLUDED

#include "nodo.h"

class Cola
{
private:
    // Primer elemento de la cola
    Nodo* primero;
    // Ultimo elemento de la cola
    Nodo* ultimo;

public:
    // Constructor
    // PRE: ninguna
    // POST: construye una cola vacía
    // - primero y ultimo apuntan a nulo
    Cola();

    // Destructor
    // PRE: cola creada
    // POST: Libera todos los recursos de la cola
    ~Cola();

    // ¿La cola es vacía?
    // PRE: cola creada
    // POST: devuelve verdadero si la cola es vacía
    // falso de lo contrario
    bool colaVacía();

    // Agregar un elemento a la cola
    // PRE: cola creada
    // POST: agrega un dato (dentro de un nodo) al final
    void insertar(Dato d);

    // Obtener el dato que está al principio
    // PRE: - cola creada y no vacía
    // POST: devuelve el dato que está al principio
    Dato obtenerDato();

    // Borrado del nodo que está al principio
    // PRE: - cola creada y no vacía
    // POST: libera el nodo que está al principio
    void sacarDato();
};

#endif // COLA_H_INCLUDED
```

Listado de código A.12: Archivo cola.cpp

```
#include "cola.h"

// Constructor
Cola::Cola() {
    primero = ultimo = 0;
}

// Destructor
Cola::~Cola() {
    while (!(colaVacia()))
        sacarDato();
}

// Cola vacia?
bool Cola::colaVacia() {
    return (primero == 0);
}

// Agrega un dato
void Cola::insertar(Dato d) {
    Nodo* pnode = new Nodo(d);
    if (this->colaVacia()) {
        primero = pnode;
    }
    else
        ultimo->asignarSiguiente(pnode);
    ultimo = pnode;
}

// Obtener el dato
Dato Cola::obtenerDato() {
    return primero->obtenerDato();
}

// Sacar un dato
void Cola::sacarDato() {
    if (primero == ultimo)
        ultimo = 0;
    Nodo* paux = primero;
    primero = paux->obtenerSiguiente();
    delete paux;
}
```

Listado de código A.13: Archivo main.cpp

```
// Uso estructura Cola
#include "cola.h"
```



```
using namespace std;

int main()
{
    Cola c;
    c.insertar(1);
    c.insertar(2);
    c.insertar(3);
    while (!c.colaVacía()) {
        cout << c.obtenerDato() << endl;
        c.sacarDato();
    }
    return 0;
}
```

Comentarios

- Al igual que en la Pila dinámica los archivos nodo.h y nodo.cpp son los mismos.
- También se utilizó int como Dato con el fin de centrarnos en la implementación de la Cola y no en otras cuestiones.
- Utilizamos un puntero al último nodo para lograr mayor eficiencia a la hora de insertar un nodo, pero no es necesario.

A.5. Implementación y uso lista simplemente enlazada

Listado de código A.14: Archivo listaSE.h

```
#ifndef LISTASE_H_INCLUDED
#define LISTASE_H_INCLUDED

#include "nodo.h"

class ListaSE
{
private:
    // Primer elemento de la lista
    Nodo* primero;
    // Tamaño de la lista
    unsigned tam;
```

```
public:
    // Constructor
    // PRE: ninguna
    // POST: construye una lista vacia
    // - primero apunta a nulo
    // - tam = 0
    ListaSE();

    // Destructor
    // PRE: lista creada
    // POST: Libera todos los recursos de la lista
    ~ListaSE();

    // La lista es vacía?
    // PRE: lista creada
    // POST: devuelve verdadero si la lista es vacia
    // falso de lo contrario
    bool listaVacia();

    // Agregar un elemento a la lista
    // PRE: lista creada
    // POST: agrega un dato en la posicion pos
    // incrementa tam en 1
    void insertar(Dato d, unsigned pos);

    // Obtener el dato que está en la posición pos
    // PRE: - lista creada y no vacia
    // - 0 < pos <= tam
    // POST: devuelve el dato que esta en la posicion pos
    // se toma 1 como el primero
    Dato obtenerDato(unsigned pos);

    // Borrado del nodo que está en la posición pos
    // PRE: - lista creada y no vacia
    // - 0 < pos <= tam
    // POST: libera el nodo que esta en la posición pos
    // se toma 1 como el primero
    void eliminarDato(unsigned pos);

    // Obtener el tamaño de la lista
    // PRE: Lista creada
    // POST: Devuelve tam (cantidad de nodos de la lista)
    unsigned obtenerTam();

private:
    // Obtiene un puntero al nodo de la posicion pos
    // PRE: 0 < pos <= tam
    // POST: devuelve un puntero al nodo solicitado
```

A.5. IMPLEMENTACIÓN Y USO LISTA SIMPLEMENTE ENLAZADA 291

```
        Nodo* obtenerNodo(unsigned pos);
};

#endif // LISTASE_H_INCLUDED
```

Listado de código A.15: Archivo listaSE.cpp

```
#include "listaSE.h"

// Constructor
ListaSE::ListaSE() {
    primero = 0;
    tam = 0;
}

// Destructor
ListaSE::~ListaSE() {
    while (!(listaVacía()))
        eliminarDato(1);
}

// Lista vacía_
bool ListaSE::listaVacía() {
    return (primero == 0);
}

// Agrega un dato al final
void ListaSE::insertar(Dato d, unsigned pos) {
    Nodo* nuevo = new Nodo(d);
    if (pos == 1) {
        nuevo->asignarSiguiente(primero);
        primero = nuevo;
    }
    else {
        Nodo* anterior = obtenerNodo(pos - 1);
        nuevo->asignarSiguiente(anterior->obtenerSiguiente());
        anterior->asignarSiguiente(nuevo);
    }
    tam++;
}

// obtiene el dato que esta en pos
Dato ListaSE::obtenerDato(unsigned pos) {
    Nodo* paux = obtenerNodo(pos);
    return paux->obtenerDato();
}

// Elimina el dato que esta en pos
```

```

void ListaSE::eliminarDato(unsigned pos) {
    Nodo* borrar = primero;
    if (pos == 1) {
        primero = borrar->obtenerSiguiente();
    }
    else {
        Nodo* anterior = obtenerNodo(pos - 1);
        borrar = anterior->obtenerSiguiente();
        anterior->asignarSiguiente(borrar->obtenerSiguiente());
    }
    delete borrar;
    tam--;
}

// devuelve el tamaño
unsigned ListaSE::obtenerTam() {
    return tam;
}

// Devuelve un puntero al nodo que esta en pos
Nodo* ListaSE::obtenerNodo(unsigned pos) {
    Nodo* aux = primero;
    unsigned i = 1;
    while (i < pos) {
        aux = aux->obtenerSiguiente();
        i++;
    }
    return aux;
}

```

Listado de código A.16: Archivo main ejemplo de uso de la listaSE

```

// Ejemplo de uso de la lista
#include "listaSE.h"

using namespace std;

int main()
{
    ListaSE lista;
    lista.insertar('A', 1);
    lista.insertar('C', 2);
    lista.insertar('F', 1);
    lista.insertar('H', 2);
    for (unsigned i = 1; i <= lista.obtenerTam(); i++ ) {
        cout << lista.obtenerDato(i) << endl;
    }
    // Se borra el segundo nodo

```

```

    lista.eliminarDato(2);
    for (unsigned i = 1; i <= lista.obtenerTam(); i++ ) {
        cout << lista.obtenerDato(i) << endl;
    }
    return 0;
}

```

Comentarios

- Los archivos nodo.h y nodo.cpp no se incluyeron porque son los mismos que los utilizados en la implementación de Pila.
- Se utilizó un char como dato para concentrarnos en la implementación y no estar pendiente de otras cosas.
- El puntero primero siempre apunta a la cima de la pila. Si bien el ingreso es al final, por una comodidad de implementación se toma que el final es el primer elemento.
- Como se puede observar la implementación de una pila es mucho más sencilla que la de una lista.

Comentarios generales

Las implementaciones mostradas son solo orientaciones sencillas como primeras aproximaciones. Hay muchas otras formas de implementar estas estructuras, seguramente más eficientes. Por ejemplo, en lugar de devolver un dato se podría devolver un puntero, los parámetros podrían pasarse por referencia y ser constantes, de este modo aseguraríamos que no se modificarán los datos pasados, etc. También se podrían utilizar funciones virtuales para utilizar herencia. Sin embargo, se eligió la manera más sencilla de implementar las estructuras para comenzar a asimilarlas sin desviar nuestra atención demasiado.

La recomendación es que de a poco, las implementaciones que desarrollen, las vayan ampliando y mejorando hasta lograr implementaciones robustas y genéricas, sin perder de lado la eficiencia y la claridad en el código. De todas maneras, se recalca que lo visto es teórico, ya que C++ tiene definidas varias clases en la librería STL (Standard Template Library), entre ellas está la clase list que ya provee todos los métodos para que solo tengamos que usarlos.